# EDB Postgres Distributed -PGD

***Release 6.4.0***

**EnterpriseDB Corporation**

**Jun 03, 2026**

# Contents


**1 foundation** **3**
- 1.1 The foundation: PostgreSQL transformed . . . 3
- 1.2 Continuous business continuity (Active-Active) . . . 3
- 1.3 Simplified consistency via commit scopes . . . 3
- 1.4 Granular data locality & sovereignty . . . 4
- 1.5 Automated conflict management . . . 4
- 1.6 Global read scaling with subscriber only nodes . . . 4
- 1.7 Managed data lifecycle: AutoPartition and Tiered tables . . . 4
- 1.8 Seamless lifecycle management . . . 5

**2 EDB Postgres Distributed 6 release notes** **7**
- 2.1 EDB Postgres Distributed 6.4.0 release notes . . . 7
  - 2.1.1 Highlights . . . 7
  - 2.1.2 Features . . . 8
  - 2.1.3 Enhancements . . . 8
  - 2.1.4 Bug Fixes . . . 9
- 2.2 EDB Postgres Distributed 6.3.1 release notes . . . 12
  - 2.2.1 Bug Fixes . . . 12
- 2.3 EDB Postgres Distributed 6.3.0 release notes . . . 12
  - 2.3.1 Highlights . . . 12
  - 2.3.2 Features . . . 12
  - 2.3.3 Enhancements . . . 13
  - 2.3.4 Changes . . . 14
  - 2.3.5 Bug Fixes . . . 14
- 2.4 EDB Postgres Distributed 6.2.0 release notes . . . 16
  - 2.4.1 Highlights . . . 16
  - 2.4.2 Features . . . 16
  - 2.4.3 Enhancements . . . 17
  - 2.4.4 Bug Fixes . . . 18
- 2.5 EDB Postgres Distributed 6.1.2 release notes . . . 19
  - 2.5.1 Bug Fixes . . . 19
- 2.6 EDB Postgres Distributed 6.1.1 release notes . . . 20
  - 2.6.1 Bug Fixes . . . 20
- 2.7 EDB Postgres Distributed 6.1.0 release notes . . . 20
  - 2.7.1 Highlights . . . 20
  - 2.7.2 Features . . . 21

2.7.3 Enhancements . . . 21
2.7.4 Changes . . . 22
2.7.5 Bug Fixes . . . 22
2.8 EDB Postgres Distributed 6.0.2 release notes . . . 24
2.8.1 Bug Fixes . . . 24
2.9 EDB Postgres Distributed 6.0.1 release notes . . . 24
2.9.1 Highlights . . . 25
2.9.2 Features . . . 25
2.9.3 Enhancements . . . 26
2.9.4 Changes . . . 28
2.9.5 Bug Fixes . . . 29

**3 Overview** **31**
3.1 Known issues and limitations . . . 31
3.1.1 Known issues . . . 31
3.1.2 Limitations . . . 33
3.2 PGD compatibility . . . 36
3.2.1 PGD compatibility with PostgreSQL versions . . . 36
3.2.2 PGD compatibility with operating systems and architectures . . . 36
3.3 Postgres Distributed terminology . . . 37
3.3.1 Asynchronous replication . . . 37
3.3.2 Commit scopes . . . 37
3.3.3 CAMO or commit-at-most-once . . . 37
3.3.4 Conflicts . . . 38
3.3.5 Connection Manager . . . 38
3.3.6 Consensus . . . 38
3.3.7 Cluster . . . 38
3.3.8 DDL (data definition language) . . . 38
3.3.9 DML (data manipulation language) . . . 38
3.3.10 Durability . . . 38
3.3.11 Eager . . . 38
3.3.12 Eventual consistency . . . 38
3.3.13 Failover . . . 39
3.3.14 Group commit . . . 39
3.3.15 Immediate consistency . . . 39
3.3.16 Lag Control . . . 39
3.3.17 Logical replication . . . 39
3.3.18 Logical standby node . . . 39
3.3.19 Node . . . 39
3.3.20 Node groups . . . 39
3.3.21 PGD cluster . . . 40
3.3.22 PGD node . . . 40
3.3.23 Physical replication . . . 40
3.3.24 Postgres cluster . . . 40
3.3.25 Quorum . . . 40
3.3.26 Quorum Commit . . . 40
3.3.27 Replicated available fault tolerance (Raft) . . . 40
3.3.28 Read scalability . . . 40
3.3.29 Subscription . . . 41
3.3.30 Switchover . . . 41
3.3.31 Synchronous replication . . . 41
3.3.32 Subscriber-only nodes . . . 41
3.3.33 Two-phase commit (2PC) . . . 41
3.3.34 Vertical scaling or scale up . . . 41

3.3.35 Witness nodes . . . 41
3.3.36 Write leader . . . 42
3.3.37 Writer . . . 42
3.4 Quickstart guide . . . 42
3.4.1 Creating your first cluster . . . 42
3.4.2 Working with SQL and the PGD cluster . . . 44
3.4.3 Loading data into your PGD Cluster . . . 46
3.4.4 Using PGD CLI . . . 48

**4 Planning your PGD deployment** **57**
4.1 Deployment patterns . . . 57
4.1.1 Choosing a pattern . . . 57
4.1.2 Architectural elements . . . 58
4.2 Choosing a Postgres distribution . . . 77
4.3 Choosing your deployment method . . . 78
4.4 Comparing replication solutions . . . 79
4.4.1 Deployment . . . 79
4.4.2 Performance . . . 80
4.4.3 Durability . . . 80
4.4.4 Consistency . . . 80
4.4.5 Data distribution . . . 81
4.4.6 Maintenance . . . 81
4.4.7 Connection Management . . . 82
4.4.8 DDL support . . . 82
4.4.9 Postgres compatibility . . . 83
4.4.10 Management . . . 83
4.4.11 Licensing . . . 83
4.5 Choosing between Physical Streaming Replication (PSR) and EDB Postgres Distributed (PGD) . . . 83
4.5.1 PSR with EFM or Patroni . . . 84
4.5.2 EDB Postgres Distributed (PGD) . . . 84
4.6 Other considerations . . . 85
4.6.1 Sizing . . . 85
4.6.2 Time and PGD . . . 85

**5 Managing lifecycle operations** **87**
5.1 Deploying and configuring EDB Postgres Distributed . . . 87
5.1.1 Manually . . . 88
5.1.2 Deploying and configuring PGD using TPA . . . 100
5.1.3 Deploying and configuring PGD on Kubernetes . . . 101
5.1.4 Deploying and configuring PGD on Hybrid Manager . . . 101
5.2 Upgrading . . . 101
5.2.1 Upgrading PGD . . . 101
5.2.2 Upgrading Postgres or Postgres and PGD major versions . . . 102
5.2.3 Other upgrades . . . 102
5.3 Testing and tuning PGD clusters . . . 129
5.3.1 pgd_bench . . . 129
5.3.2 Notes on pgd_bench usage . . . 130
5.4 Backup and recovery . . . 131
5.4.1 Logical backup and restore . . . 131
5.4.2 Physical backup and restore . . . 133
5.4.3 Eventual consistency . . . 135
5.4.4 Monitoring . . . 136
5.5 Migrating to Postgres Distributed (PGD) . . . 137
5.5.1 Migrating from Postgres Physical Streaming Replication (PSR) to PGD . . . 137

**6 Configuration and Management** **153**
6.1 Node types and capabilities . . . 153
6.1.1 An overview of PGD Node types . . . 153
6.1.2 Witness nodes . . . 154
6.1.3 Logical standby nodes . . . 154
6.1.4 Subscriber-only nodes and groups . . . 156
6.2 Node management . . . 160
6.2.1 Creating PGD nodes . . . 161
6.2.2 Groups and subgroups . . . 162
6.2.3 Creating and joining PGD groups . . . 163
6.2.4 Viewing PGD topology . . . 164
6.2.5 Removing nodes and groups . . . 165
6.2.6 Node restart and down node recovery . . . 166
6.2.7 Automatic synchronization . . . 167
6.2.8 Node UUIDs . . . 168
6.2.9 Replication slots created by PGD . . . 170
6.3 Connection Manager . . . 171
6.3.1 Connection Manager overview . . . 171
6.3.2 Connection Manager Authentication . . . 173
6.3.3 Connection pooling . . . 174
6.3.4 Load Balancing with Connection Manager . . . 176
6.3.5 Configuring Connection Manager . . . 178
6.3.6 Monitoring the Connection Manager . . . 179
6.4 Postgres configuration . . . 180
6.4.1 Postgres settings . . . 180
6.4.2 Max prepared transactions . . . 181
6.4.3 Considerations for global configuration . . . 181
6.5 Monitoring . . . 182
6.5.1 Monitoring through SQL . . . 182
6.6 AutoPartition in PGD . . . 194
6.6.1 Auto creation of partitions . . . 195
6.6.2 AutoPartition examples . . . 195
6.6.3 RANGE-partitioned tables . . . 195
6.6.4 Stopping automatic creation of partitions . . . 196
6.6.5 Waiting for partition creation . . . 196
6.6.6 Finding a partition . . . 197
6.6.7 Enabling or disabling autopartitioning . . . 197
6.6.8 Dropping or detaching a partition . . . 197
6.7 Commit Scopes . . . 197
6.7.1 Introducing . . . 197
6.7.2 Commit scope kinds . . . 198
6.7.3 Working with commit scopes . . . 198
6.8 Conflict Management . . . 235
6.8.1 Conflicts . . . 235
6.8.2 Column-level conflict detection . . . 249
6.8.3 Conflict-free replicated data types . . . 254

**7 Advanced functionality** **267**
7.1 DDL replication . . . 267
7.1.1 DDL overview . . . 267
7.1.2 DDL replication options . . . 269
7.1.3 DDL locking details . . . 269
7.1.4 Managing DDL with PGD replication . . . 271
7.1.5 DDL command handling matrix . . . 273

7.1.6 DDL and role manipulation statements . . . 282
7.1.7 Workarounds for DDL restrictions . . . 283
7.1.8 PGD functions that behave like DDL . . . 283
7.2 Decoding worker . . . 284
7.2.1 Enabling . . . 285
7.2.2 Disabling . . . 285
7.3 CDC Failover support . . . 286
7.3.1 Background . . . 286
7.3.2 CDC Failover support . . . 286
7.3.3 Enabling CDC Failover support . . . 287
7.3.4 CDC Failover support with Postgres 17+ . . . 287
7.3.5 Obtaining Initial Consistent Snapshot . . . 287
7.3.6 Failing Over to Newly Joined Nodes . . . 288
7.3.7 Tracking Per-Origin Progress . . . 288
7.3.8 Limitations . . . 289
7.4 Parallel Apply . . . 289
7.4.1 What is Parallel Apply? . . . 289
7.5 Replication sets . . . 291
7.5.1 Using replication sets . . . 291
7.5.2 Behavior of partitioned tables . . . 291
7.5.3 Behavior with foreign keys . . . 292
7.5.4 Replication set membership . . . 293
7.5.5 Listing replication sets . . . 293
7.5.6 DDL replication filtering . . . 294
7.5.7 Selective replication example . . . 294
7.6 Security and roles . . . 298
7.6.1 Roles . . . 298
7.6.2 Role management . . . 299
7.6.3 PGD predefined roles . . . 300
7.6.4 Roles and replication . . . 307
7.6.5 Access control . . . 308
7.7 Sequences . . . 309
7.7.1 Automatic sequence conversion . . . 311
7.7.2 UUIDs, KSUUIDs, and other approaches . . . 315
7.7.3 KSUUID v2 functions . . . 315
7.7.4 See also . . . 317
7.8 Replicated large objects . . . 317
7.8.1 Using replicated large objects . . . 317
7.8.2 How large objects are stored . . . 317
7.8.3 Limitations . . . 318
7.9 Stream triggers . . . 318
7.9.1 Trigger execution during apply . . . 318
7.9.2 Terminology of row-types . . . 319
7.9.3 Conflict triggers . . . 320
7.9.4 Transform triggers . . . 320
7.9.5 Row contents . . . 321
7.9.6 Execution order . . . 321
7.9.7 Stream triggers examples . . . 321
7.10 Transaction streaming . . . 323
7.10.1 PGD enhancements . . . 323
7.10.2 Configuration . . . 324
7.10.3 Monitoring . . . 326
7.11 Explicit two-phase commit (2PC) . . . 326
7.11.1 Use . . . 326

**8 Guidance for DBAs** **329**
8.1 Encrypting columns in PGD . . . 329
8.1.1 Comparing encryption methods . . . 329
8.1.2 Planning for encryption in a distributed cluster . . . 330
8.1.3 Encrypting columns using pgcrypto . . . 331
8.1.4 Encrypting columns using DBMS_CRYPTO . . . 332
8.1.5 Managing encryption keys securely . . . 334
8.2 Using tiered storage . . . 334
8.2.1 Understanding table types . . . 335
8.2.2 Tiered storage strategies . . . 335
8.2.3 Metadata management . . . 336
8.3 Recovering a failed node . . . 346
8.3.1 Assessing the cluster state . . . 347
8.3.2 Removing the failed nodes from the cluster . . . 347
8.3.3 Recovering the failed nodes . . . 348
8.3.4 Validating the recovered node . . . 349
8.3.5 Verifying cluster health . . . 349
8.3.6 Troubleshooting . . . 350
8.3.7 Post-recovery checklist . . . 350
8.4 Developing applications for geo-replication . . . 350
8.4.1 When geo-replication affects your application . . . 350
8.4.2 Make your application geo-replication ready . . . 351
8.5 Setting up access controls . . . 359
8.5.1 Securing inter-node connections . . . 359
8.5.2 Securing client application connections . . . 363
8.6 Handling replication apply errors . . . 366
8.6.1 Choosing an error policy . . . 366
8.6.2 Configuring the error policy . . . 366
8.6.3 Responding to a disabled subscription . . . 367
8.6.4 Monitoring for accumulated failures . . . 369
8.7 Configuring connection pooling . . . 369
8.7.1 Prerequisites . . . 369
8.7.2 Enabling connection pooling . . . 369
8.7.3 Monitoring the pool . . . 370
8.7.4 Migrating from pgBouncer . . . 370

**9 Guidance for Develoepers** **371**
9.1 Designing applications for PGD . . . 371
9.1.1 Preparing your application for PGD . . . 371
9.1.2 Assessing an existing application for PGD . . . 373
9.2 Managing geo-replication . . . 374
9.2.1 When to use geo-replication . . . 374
9.2.2 Deployment workflow . . . 374
9.3 Using connection pooling in your application . . . 393
9.3.1 Understanding the active pool mode . . . 394
9.3.2 Adapting your application for transaction mode . . . 394
9.4 Application use . . . 395
9.4.1 DML and DDL replication and nonreplication . . . 396
9.4.2 Nodes with differences . . . 396
9.4.3 Timing considerations and synchronous replication . . . 398
9.4.4 Using extensions with PGD . . . 398
9.4.5 Use of table access methods (TAMs) in PGD . . . 400
9.4.6 Feature compatibility . . . 400

**10 Reference** **403**
10.1 PGD Reference . . . . . 403
10.1.1 Tables, views and functions reference . . . . . 403
10.1.2 EDB Postgres Distributed Command Line Interface (PGD CLI) . . . . . 585

EDB Postgres Distributed (PGD) is an integrated software stack that transforms standard PostgreSQL into a distributed, always-on database platform. Designed for the modern enterprise where downtime is no longer an option, PGD provides a robust framework for global data distribution, high availability, and simplified consistency management.

CHAPTER 1

# foundation

## 1.1 The foundation: PostgreSQL transformed

PGD is implemented as a sophisticated extension compatible with standard PostgreSQL, EDB Postgres Extended Server (PGE), or EDB Postgres Advanced Server (EPAS). At its core, PGD is powered by the latest generation of BDR (Bi-Directional Replication). While several active-active systems on the market are still built on the legacy BDR 1.0 architecture, PGD incorporates over ten years of continuous enhancements and enterprise hardening since that initial release. This battle-hardened version provides the specific performance and reliability required for Tier 1 applications, leveraging a decade of production-proven refinements to handle modern distributed workloads.

## 1.2 Continuous business continuity (Active-Active)

Unlike traditional physical replication, which requires complex and often minutes-long failover processes, PGD is built on logical replication.

- **Zero-wait failover:** Every node in a PGD cluster is inherently writable. If a node or region fails, applications can immediately route traffic to any other active node without waiting for a primary election or recovery process.
- **Always-on access:** Traffic remains fluid across the cluster, ensuring that "recovery time" is measured in seconds, not minutes.

## 1.3 Simplified consistency via commit scopes

PGD is the only solution that provides true distributed consistency for PostgreSQL. It acknowledges that different applications have different needs regarding the CAP Theorem (Consistency, Availability, and Partition Tolerance).

- **Predefined guarantees:** Instead of manually tuning dozens of `postgresql.conf` parameters, PGD uses commit scopes. These are predefined, intuitively named configurations that allow you to choose the exact trade-off between performance and durability your application requires.
- **Reduced complexity:** You can toggle between asynchronous, synchronous, or quorum commit logic using single global settings. For more complex requirements, you can modify these scopes down to the individual transaction level using simple commands to meet specific business needs on the fly.

## 1.4 Granular data locality & sovereignty

To meet the demands of a global user base and strict regulatory environments like GDPR, PGD supports sophisticated data routing:

- **Selective filtering:** Control exactly which data is replicated to which nodes.
- **Regional sovereignty:** Pin sensitive user data to specific geographic regions while maintaining a unified global cluster view.
- **Reduced latency:** Keep data physically close to application instances. By maintaining local data availability at the edge, you eliminate "speed of light" delays associated with cross region round trips.

## 1.5 Automated conflict management

While we recommend application designs that minimize row-level contention, we recognize that real-world production environments are inherently messy. For those inevitable moments when concurrent writes overlap, PGD provides a robust, automated conflict resolution engine.

Acting as a critical safety net, PGD detects and resolves discrepancies based on established business rules (such as last update wins or custom triggers). This ensures data remains identical across the global fabric without requiring manual DBA intervention or "data cleanup" shifts.

## 1.6 Global read scaling with subscriber only nodes

For workloads requiring massive read capacity, PGD supports subscriber only nodes. These groups allow you to scale horizontally far beyond the limits of a standard cluster.

- **Massive scale:** PGD is proven in production environments with clusters containing up to 200 subscriber nodes.
- **Edge access:** These nodes can be deployed anywhere in the world to provide low-latency local access for global users.
- **Flexible topology:** You can choose how data is distributed. All nodes in a group can connect directly to the primaries, or a single node per group can act as a "hub" that redistributes data to others. This redistribution model includes automatic high availability to ensure the edge remains online.
- **Performance tuning:** Architects have the choice between minimizing replication latency to the edge and managing the connection load on the primary nodes.

## 1.7 Managed data lifecycle: AutoPartition and Tiered tables

New data is being created exponentially and old data often needs to remain accessible to meet regulator requirements. PGD builds upon its AutoPartition capability, which automates the creation of time-interval partitions, by introducing Tiered tables. This feature provides a seamless way to manage hot and cold data across different storage tiers while integrating directly with modern data lakehouse architectures.

- **Automated management:** PGD automatically handles the creation and maintenance of time-interval apparitions. You define the policy, such as daily or monthly intervals, and the system ensures the necessary partitions exist before the data arrives.
- **Unified query interface:** You can query both hot (row based) and cold (columnar) data from a single table. PGD handles the complexity of retrieving data from different storage formats behind the scenes.
- **Automated data offloading:** PGD automatically migrates older data partitions to a highly compressed columnar format in object storage. This keeps your active SSD focused on high performance, low latency transactions.

- **Flexible storage and data lake targets:** Tiered tables support cost-effective targets including Amazon S3, MinIO, or local file systems. Furthermore, PGD now supports replication to Apache Iceberg through the Postgres Analytics Accelerator (PGAA).
- **Streamlined analytics:** With Apache Iceberg support, you can stream data directly from your PGD cluster to a data lake and query that Iceberg data directly from within PGD. This allows you to join real-time transactional data with massive historical datasets without complex ETL pipelines.
- **Significant cost efficiency:** By moving historical data to object storage or an Iceberg lakehouse, you can reduce storage costs by up to 18x compared to high-performance SSDs

## 1.8 Seamless lifecycle management

PGD eliminates the "maintenance window." Because the architecture is distributed and nodes are independent, you can perform rolling upgrades and OS patches one node at a time. The application remains online and connected to the rest of the cluster throughout the entire lifecycle of the database.

| Feature | Standard PostgreSQL | EDB Postgres Distributed (PGD) |
|---|---|---|
| Write availability | Single primary only | Multi-master (all nodes) |
| Failover speed | Minutes (detection + promotion) | Near-instant (traffic re-routing) |
| Consistency | Rigid (primary/standby) | Configurable via commit scopes |
| Data locality | All-or-nothing replication | Granular/selective filtering |
| Upgrades | Significant downtime/re-build standbys | Zero-downtime rolling upgrades |

CHAPTER 2

# EDB Postgres Distributed 6 release notes

The EDB Postgres Distributed documentation describes the latest version of EDB Postgres Distributed 6, including minor releases and patches. The release notes provide information on what was new in each release. For new functionality introduced in a minor or patch release, the content also indicates the release that introduced the feature.

| Release Date | EDB Postgres Distributed |
|---|---|
| 26 May 2026 | *EDB Postgres Distributed 6.4.0 release notes* |
| 14 Apr 2026 | *EDB Postgres Distributed 6.3.1 release notes* |
| 26 Mar 2026 | *EDB Postgres Distributed 6.3.0 release notes* |
| 05 Dec 2025 | *EDB Postgres Distributed 6.2.0 release notes* |
| 06 Nov 2025 | *EDB Postgres Distributed 6.1.2 release notes* |
| 09 Oct 2025 | *EDB Postgres Distributed 6.1.1 release notes* |
| 19 Aug 2025 | *EDB Postgres Distributed 6.1.0 release notes* |
| 25 Jun 2025 | *EDB Postgres Distributed 6.0.2 release notes* |
| 09 Jun 2025 | *EDB Postgres Distributed 6.0.1 release notes* |

## 2.1 EDB Postgres Distributed 6.4.0 release notes

Released: 26 May 2026

EDB Postgres Distributed (PGD) 6.4.0 includes new features, enhancements, and bug fixes.

### 2.1.1 Highlights

- ** *QUORUM COMMIT* :** A new commit scope kind that enforces pre-commit coordination across the cluster. A majority of nodes must agree before any node commits a transaction. If agreement isn't reached, the transaction rolls back everywhere, guaranteeing a consistent global view with no conflicting updates and no divergence across nodes.
- ** *Connection pooling* :** Connection Manager now supports transaction-mode connection pooling in addition to session-mode pooling. In transaction pooling mode, a backend connection is held only for the duration of a transaction, reducing pressure on `max_connections` and improving scalability under high concurrency.

- ** *Replicated large objects* :** PGD 6.4.0 adds support for large objects in distributed deployments. PGD replicates large objects across all nodes and generates globally unique OIDs using galloc sequences to eliminate conflicts. The standard `lo_*` API, including `lo_import` and `lo_export` , works transparently through Connection Manager, so applications using large objects can migrate to PGD without changes to their architecture.

### 2.1.2 Features

| Description | Addresses |
|---|---|
| Added configurable apply error policy for handling replication failures.PGD now provides a configurable policy for handling replication apply failures. When a transaction fails to apply, PGD can capture the failed changes and expose them for review and resolution via new SQL functions. Configure the policy per node group using *bdr.alter_node_group_option*. For more information, see <a href="/pgd/current/apply-error-policy">Handling replication apply errors</a>.</p> | |
| Added support for logically replicated large objects.PGD now provides its own implementation of large objects as a drop-in replacement for the built-in Postgres large objects. PGD large objects are logically replicated across all nodes in the cluster. The feature is available via *bdr.lo_* functions directly, or transparently via *bdr.override_large_objects*. For more information, see <a href="/pgd/current/large-objects">Replicated large objects</a> and the <a href="/pgd/current/reference/tables-views-functions/large-objects">large object functions reference</a>.</p> | |
| Added the *pgd raft sync-snapshot* command to synchronize Raft state.New <a href="/pgd/current/reference/cli/command_ref/raft/sync-snapshot">`pgd raft sync-snapshot`</a> command that synchronizes the Raft state across all nodes by disabling Raft, exporting state from the leader node, importing it on all nodes, and re-enabling Raft. Use *--group* to specify a target group; defaults to global group.</p> | |
| Added <a href="/pgd/current/reference/tables-views-functions/functions#bdrtransfer_a to transfer write and Raft leadership away from the local node.Transfers write and Raft leadership away from the local node across all groups, useful for draining a node before maintenance.</p> | _all_leaderships`</ |

### 2.1.3 Enhancements

| Description | Addresses |
|---|---|
| Parallel apply is now compatible with Quorum Commit.Unlike Group Commit, Quorum Commit is compatible with parallel apply. Quorum Commit improves on legacy Group Commit, which was only supported with *num_writers = 1*.</p> | |
| Enhanced the *pgd raft show* command to display more information.<a href="/pgd/current/reference/cli/command_ref/raft/show">`pgd raft show`</a> now displays three sections by default to provide deeper visibility into the Raft consensus layer: State (consensus state), Followers (follower status as seen from the leader), and Journal (global journal details).</p> | |
| Connection Manager now respects *IntervalStyle* specified in connection startup parameters.Previously, *IntervalStyle* was only supported via the *options* connection parameter. Connection Manager now correctly applies *IntervalStyle* when specified directly as a startup parameter.</p> | |

continues on next page

Table 3 – continued from previous page

| Description | Addresses |
|---|---|
| Improved Connection Manager idle connection handling on topology and configuration changes.Connection Manager now closes outdated idle connections proactively on topology or configuration changes, and reports server connection errors directly to the client.</p> | |
| Added per-subgroup connection string support via *node_group_dsn*.A new *node_group_dsn* column in *bdr.node_config* lets nodes within the same subgroup use a dedicated connection string, useful in Kubernetes and cloud environments to reduce egress costs. Set it via <a href="/pgd/current/reference/tables-views-functions/nodes-management-interfaces#bdralter_node_option">`bc | > |
| Enhanced <a href="/pgd/current/reference/tables-views-functions/functions#bdrrouting to support automatic candidate selection.`bdr.routing_leadership_transfer` now accepts *NULL* for *leader_name*, triggering automatic selection of the best candidate based on write leader progress. Automatic selection is the recommended approach when using Quorum Commit as the commit scope kind. Requires PGD 6.4 with Raft protocol version 6004 or above.</p> | uting_leadership_ |
| Connection Manager now reloads Postgres configuration parameters dynamically on *SIGHUP*.Previously, configuration parameters were only loaded on Connection Manager process start. Connection Manager now reloads them on *SIGHUP* without requiring a restart.</p> | |
| DDL on objects in the *bdr* schema is no longer permitted.Database users can no longer create or modify database objects in the *bdr* schema, or set the namespace of an object to the *bdr* schema. This restriction prevents catalog corruption and extension upgrade issues that such operations could cause. For more information, see <a href="/pgd/current/reference/tables-views-functions/pgd-settings#bdrallow_bdr_schema_changes">`bdr.allow_ | 53224 :/p> |
| Added sanity checks for joining and parting nodes in CAMO-configured groups.Attempting to join a data node to a group where existing nodes are configured as CAMO partners, or to part a data node when the group's *default_commit_scope* or the local node's *bdr.commit_scope* is set to a CAMO commit scope, now raises an error. To perform these operations, drop the CAMO commit scope first.</p> | 55212 |
| The configuration parameter *bdr.enable_auto_sync_reconcile* is now visible in *pg_settings*.The *bdr.enable_auto_sync_reconcile* configuration parameter is now exposed in the *pg_settings* view, making it visible to monitoring tools.</p> | |
| Improved log output when the transaction change tracker buffer is exceeded.The emitted log message now makes it clear that the issue is transient and that processing resumes following an automatic restart of the writer.</p> | |

### 2.1.4 Bug Fixes

| Description | Addresses |
|---|---|
| Fixed zero OID assigned to tablespaces during logical join sync.`bdr_tablespace_upsert()` wasn't setting the OID when inserting a new tablespace, causing a duplicate key violation when multiple user tablespaces were synced during logical join.</p> | 59715 |
| Fixed *pgd assess* command failing on Postgres instances without BDR extension.The *pgd assess* command was failing with *ERROR: schema "bdr" does not exist*. The command now gracefully handles instances without BDR installed by checking for the BDR schema first, and performs the full assessment with compatibility suggestions for both PGD Essential and Expanded editions.</p> | |

continues on next page

Table 4 – continued from previous page

| Description | Addresses |
|---|---|
| Fixed a memory leak in the walsender for the temporary replication slot used by joining nodes.Fixed a memory leak in *bdr_copy_to_repl_dest_receiver_receive*, where per-row allocations in the walsender for the temporary replication slot used by joining nodes were not freed correctly.</p> | 59663 |
| Fixed unbounded WAL accumulation caused by stale *node_peer_progress_mapping* entries.Stale *node_peer_progress_mapping* entries left behind by cancelled forwarding subscriptions were pinning the group replication slot and preventing WAL cleanup. The maintenance path now removes entries that have been superseded by direct subscription progress.</p> | 58999, 59320 |
| Fixed deadlocks and stalls in the part and sync node state machines.The part and sync state machines could deadlock or stall when end-of-recovery markers were delivered through *FORWARD* or *CATCHUP* subscriptions. Multiple fixes restrict confirmation waits to *NORMAL* subscriptions, ensure atomic origin advancement during *PART_CATCHUP*, and prevent a double-delete error on *node_peer_progress_mapping* rows.</p> | |
| Fixed a writer segfault when a replication origin was already in use.Enabling a *NORMAL* subscription while a *sync_node* request was in progress for the same origin caused a second writer to crash with a *SIGSEGV*. PGD now blocks enabling a *NORMAL* subscription in this case, and the origin-acquisition path returns a recoverable error instead of crashing.</p> | |
| Fixed *pgd manager* crash during BDR 5 to PGD 6 rolling upgrade.When upgrading the last node in a cluster from BDR 5 to PGD 6, the *pgd manager* process crashed with a *SIGSEGV* during subscription reconfiguration when replication slots had been renamed. The *pgd manager* now handles missing or renamed slots gracefully.</p> | |
| Fixed system OID assignment in *pgd node setup* causing upgrade failures.`pgd node setup` was creating BDR objects with system-range OIDs, causing *pgd node upgrade* to fail with *schema "bdr" does not exist*. The setup command now assigns user-range OIDs correctly, and the upgrade command detects the issue early on existing affected clusters.</p> | |
| Fixed upgrade failures from PGD 6.2 to 6.3 or 6.4 when *pgaudit* follows *bdr* in *shared_preload_libraries*.When *pgaudit* followed *bdr* in *shared_preload_libraries*, upgrades from PGD 6.2 failed with *ERROR: pgaudit not loaded before call to pgaudit_ddl_command_end()*. Hook deinitialization was resetting hooks to *NULL*, crashing pgaudit in the manager process. The fix adds bypass macros so hooks defer to the next handler instead of deinitializing.</p> | 59715 |
| Fixed Connection Manager process not stopping after postmaster is killed. | 59631 |
| Fixed memory leak in Postgres worker caused by *bdr_connection.client_addr* updates.Handling updates to the *bdr_connection.client_addr* configuration parameter, used by Connection Manager in session pooling mode, was leaking memory in the Postgres worker.</p> | |
| Fixed Connection Manager using outdated routing configuration when routing was enabled or disabled on a node group. | |
| Fixed segfaults in *bdr.stat* shared-memory entry handling.Concurrent statistics readers observed partially-initialized or freed Dynamic Shared Area (DSA) pointers when a stat entry was being created or dropped, causing *bdr_stat_write_file* to dereference NULL or freed memory. The initialization path now marks entries invalid until allocation succeeds, the drop path holds an exclusive lock, and all dshash scan paths guard against invalid DSA pointers.</p> | |

continues on next page

Table 4 – continued from previous page

| Description | Addresses |
|---|---|
| Fixed an undetectable deadlock in *bdr.alter_sequence_set_kind* under parallel writers.When multiple writers concurrently changed sequence kinds, two writers could attempt to upgrade *ShareRowExclusiveLock* to *AccessExclusiveLock* on the same sequence relation simultaneously, deadlocking outside the deadlock detector's view. The lock is now acquired at *AccessExclusiveLock* upfront to eliminate the lock-upgrade window.</p> | |
| Fixed an assertion crash in *bdr_load_commit_decisions* at end of WAL.Reaching the end of WAL while loading commit decisions tripped an assertion (*buf.len > 0*) on debug builds. The loop now terminates normally when no more data is available.</p> | |
| Fixed a crash in the failover slot standby reply hook before BDR catalogs are available.The standby reply hook could fire before BDR catalogs were fully initialized, causing a crash. The hook now checks catalog availability and exits early when catalogs are not yet ready.</p> | |
| Replaced a writer-start assertion with a graceful exit when the receiver role is reset mid-launch.If the receiver role was reset between writer launch and writer initialization, the writer tripped an assertion. The writer now exits cleanly so it can be restarted by the supervisor without producing a core file.</p> | |
| Fixed *pgd raft set-leader* to wait correctly for asynchronous transfers and validate group scope.The *pgd raft set-leader* command didn't properly wait for leadership change in asynchronous mode, allowed invalid cross-group transfers, and produced an unclear error message when the target group didn't exist. All three issues are now corrected.</p> | |
| Made *routing_leadership_transfer* retry when no Raft leader is currently available.`routing_leadership_transfer` returned immediately when consensus reported no leader, instead of retrying like other transient consensus errors. *BDR_CONSENSUS_NO_LEADER* is now part of the retryable error set, so leadership transfers complete after a brief leader election rather than failing.</p> | |
| Downgraded analytics lease renewal failures from *ERROR* to *LOG*.Transient lease renewal failures in the analytics (Seafowl) replicator caused the worker to crash and restart. Because these failures are self-recovering on the next renewal cycle, they are now logged at *LOG* level and the worker continues running.</p> | |
| Fixed AutoPartition wait reliability and task manager backoff delays.The AutoPartition task manager's exponential backoff wasn't reset after processing work items, causing unnecessary iteration delays. *autopartition_wait_for_partitions_on_all_nodes* was also hardened with better signalling, a *statement_timeout = 0* override on remote connections, and fixes for *PGresult*/*PGconn* leaks.</p> | |
| Fixed Connection Manager log level filtering to match Postgres behavior.Connection Manager was incorrectly interpreting *log_min_messages*, causing inconsistent log filtering compared to Postgres.</p> | |
| Changed the default display value of *bdr.ddl_locking* to *auto*.Previously the value was displayed as *true*, which has the same meaning but exists only as a backwards-compatibility alias. This change requires no configuration changes.</p> | |
| Fixed pglogical manager process spinning at 100% CPU when a PGD worker becomes unresponsive to *SIGTERM*.When the pglogical manager process died with an error and tried to terminate other PGD workers, it hung if those workers had died uncleanly or were stuck and unable to receive signals. The process now exits correctly in all such cases.</p> | 60272 |

## 2.2 EDB Postgres Distributed 6.3.1 release notes

Released: 14 April 2026

EDB Postgres Distributed 6.3.1 includes only bug fixes focused on improving stability and reliability. No new features or enhancements are included in this release. For a list of known issues, see *'Known issues and limitations'* .

### 2.2.1 Bug Fixes

| Description | Addresses |
|---|---|
| Fixed a segfault in the consensus worker during rolling upgrade to 6.3.0 when *bdr.enable_auto_sync_reconcile* is on.When *bdr.enable_auto_sync_reconcile = on*, an older-version node sending a *reconcile_2pc_requests* Raft upsert to a 6.3.0 node could cause a crash loop on restart. This setting is on by default in 6.x versions and off by default in 5.x versions.</p> | 59715 |
| Fixed *bdr_init_physical* failure when PGAA extension is present.When scaling up a PGD cluster from a single data group to multiple data groups on deployments using the Postgres Analytics Accelerator (PGAA) extension, *bdr_init_physical* failed with *no analytics lease found for database* during the *bdr.remove_unwanted_data()* step. This issue only affected clusters initially provisioned with a single data group that were later scaled out.</p> | |
| Fixed user login attributes not replicated to peer nodes.After a *CREATE USER* statement, the user's login attribute was not propagated to peer nodes, preventing login on any node other than the originating one. This fix ensures role login attributes are correctly included in replication.</p> | |

## 2.3 EDB Postgres Distributed 6.3.0 release notes

Released: 26 March 2026

EDB Postgres Distributed (PGD) 6.3.0 includes new features, enhancements, and bug fixes focused on improving stability and reliability.

### 2.3.1 Highlights

- **Connection Manager on subscriber-only nodes:** Connection Manager now starts by default on subscriber-only nodes, providing access to advanced connection pooling and session management features. It also re-routes all read-only traffic to the local node to reduce latency.
- **Consensus management via PGD CLI:** The PGD CLI now includes commands to enable or disable Raft consensus on specific nodes or groups. The commands *pgd raft enable* and *pgd raft disable* provide granular control over node behavior.
- **Platform support update:** PGD is now supported on SUSE Linux Enterprise Server (SLES) 15 SP7 beginning with 6.3.0.

### 2.3.2 Features

| Description | Addresses |
|---|---|
| Connection Manager starts by default on subscriber-only nodes and includes new health check endpoints. Connection Manager now initializes automatically on subscriber-only nodes. This release also adds the */connection/is-ready-ro* and */connection/is-ready-rw* HTTP endpoints to provide granular readiness checks for read-only and read-write traffic. | |
| Added the *pgd raft set-leader* command. Introduced the *pgd raft set-leader* command, allowing users to manually change the Raft leader for a specific group. | |
| Added *pgd raft enable* and *pgd raft disable* commands. Added commands to enable or disable consensus (Raft) across different nodes, groups, or cluster levels. | |

### 2.3.3 Enhancements

| Description | Addresses |
|---|---|
| Improved the group deletion process to automatically drop associated replication sets. Enhanced the node group removal logic to automatically drop replication sets of the same name. | 57531 |
| Improved the consistency of Connection Manager against *pg_hba.conf*. Enhanced how the Connection Manager matches database names and usernames against *pg_hba.conf* fields. Key improvements include support for *sameuser* matching, refined *samerole* logic, and expanded *+rolename* matching to include the role itself and indirect members. Additionally, quotes are now stripped from usernames before matching to ensure consistent authentication behavior. | |
| Added support for user name mapping (*pg_ident.conf*) to the Connection Manager. This enhancement allows the Connection Manager to use *pg_ident.conf* for user name mapping. Currently, this functionality applies to the *cert* authentication method, aligning Connection Manager behavior with PostgreSQL's native authentication identity mapping. | |
| Improved the *bdr.reconcile_2pc_requests* table and added the view *bdr.reconcile_2pc_requests_summary*. Enhanced *bdr.reconcile_2pc_requests* to include more details such as request time, origin LSN, and commit timestamp. Additionally, added a user-friendly view, *bdr.reconcile_2pc_requests_summary* , to simplify monitoring of Two-Phase Commit (2PC) reconciliation. | |
| Improved conflict handling for origin changes in group commit. Enhanced group commit to perform eager conflict detection and resolution when the origin of a transaction changes. Conflict handling now relies on timestamp-based detection rather than replication-progress speculation. To ensure optimal performance and avoid increased commit latency, you must synchronize system clocks across all nodes. | |
| Added the *bdr.stat_reconcile_commit_scope* view for improved monitoring. Introduced a new system view, *bdr.stat_reconcile_commit_scope* , which provides real-time statistics for commit scope reconciliation. | |
| Improved write leader election logic via two new *bdr.node_group_config* options. Introduced new options for *bdr.node_group_config* , which allow the cluster to select the most advanced candidate replica during leader transitions: <ul><li>`leader_progress_tracking_enable`: Periodically reports replication progress to the Raft leader.</li><li>`leader_progress_tracking_period`: Configures the reporting interval.</li></ul> | |

continues on next page

Table 7 – continued from previous page

| Description | Addresses |
|---|---|
| Added support for *ALTER SEQUENCE ... RENAME* and *ALTER SEQUENCE ... SET SCHEMA* for *galloc* sequences. Renaming *galloc* sequences or moving them between schemas is now fully supported. This is enabled by a global UUID identifier (*sequuid*) in the *bdr.sequence_kind* catalog, which allows PGD to track these sequences consistently and concurrently. This functionality works even for sequences upgraded from versions prior to PGD 6.3, as PGD automatically generates the required UUIDs for existing sequences to ensure compatibility. | |
| Added the *bdr.prepared_xacts_summary* diagnostic view. Introduced *bdr.prepared_xacts_summary* , a local diagnostic view for monitoring prepared transactions implicitly created by group commit scopes. | |

### 2.3.4 Changes

| Description | Addresses |
|---|---|
| Standalone macOS PGD CLI is now ARM64 only. The standalone PGD CLI for macOS is now available exclusively for the Apple Silicon (ARM64) architecture. This change applies only to the standalone macOS binary; the availability and support for standalone PGD CLI binaries on GNU/Linux remain unchanged. | |

### 2.3.5 Bug Fixes

| Description | Addresses |
|---|---|
| Fixed a replication crash loop caused by interrupted partition detaching. Resolved an issue where logical replication would enter an infinite crash loop if a *DETACH PARTITION CONCURRENTLY* command was interrupted. Previously, the walsender would fail with the error *relation has no parent because it's being detached* when decoding WAL for a partition in a *detach-pending* state. | 53628, 59291 |
| Fixed a *walsender* crash loop during partition detachment. Fixed an issue where the *walsender* process could enter a crash loop when encountering partitions with *inhdetachpending = true*, which reported *ERROR: relation %u has no parent failures*. | 53628, 59291 |
| Fixed *ALTER ROLE* replication failures. Fixed an issue where replication would break because attributes like *NOSUPERUSER* or *NOREPLICATION* were automatically added to the replicated command. Since these options can only be altered by a superuser, the apply process failed on downstream nodes. This change ensures the replicated command only includes options that were explicitly specified and for which the user has sufficient privileges. | 54906 |
| Fixed encoding charset issue with *pgd setup* command. The command now attempts to fetch the encoding value directly from the remote node, defaulting to UTF8 if unavailable. | |
| Fixed the in-line help for the *pgd group set-option* command. Corrected the help text for *pgd group set-option* to accurately reflect group-level options. | |
| Fixed the matrix view output for the *pgd replication show* command. Inactive/disconnected slots now display *!* instead of * when lag is below 10 MB, restoring consistency. | 56295 |
| Fixed a memory growth in the consensus shared memory queue. Prevented shared memory queues used by consensus from growing while a node is unreachable, ensuring that memory remains stable during node outages. | |

continues on next page

Table 9 – continued from previous page

| Description | Addresses |
|---|---|
| Fixed a memory leak in the consensus process when subscriber-only nodes are down. Resolved a malloc-based memory leak related to the improper freeing of connection strings. | 57094 |
| Fixed an issue where the *use_https* node group configuration option was ignored. Resolved a bug where Connection Manager incorrectly ignored the *use_https* setting. When upgrading to PGD 6.3.0 or 5.9.2, you must explicitly enable HTTPS by setting the *use_https* group option to *true* (in addition to having *ssl_cert_file* and *ssl_key_file* configured in *postgresql.conf* or via the command line) before the upgrade to maintain previous behavior. | |
| Fixed a Connection Manager memory leak occurring during configuration reloads. Resolved a memory leak in the Connection Manager caused by incorrect handling of Postgres memory contexts during configuration reloads. | |
| Fixed a segmentation fault occurring during *COPY* commands when *pgaudit.role* is set to a non-default value. Resolved an issue where Postgres would crash with a segmentation fault when executing a *COPY* command if the *pgaudit.role* configuration was set to a non-default value. | 58060 |
| Fixed an issue where setting *retention_period* and *analytics_offload_period* simultaneously on AutoPartition tables caused conflicts. Improved the coordination between data retention and analytics offloading. Partitions are now offloaded using Postgres Analytics Accelerator (PGAA) before the data retention policy is applied. | |
| Fixed an issue with the analytics replicator, which could cause redundant replication processes. In cluster topologies with subgroups and analytics enabled in the parent group, a bug caused each subgroup's write leader to start its own analytics replicator instance, resulting in duplicate rows. This fix ensures that only one analytics replicator per PGD group runs across the cluster. | |
| Fixed an infinite loop in the Connection Manager that occurred when a server connection was unexpectedly closed. Resolved a race condition in the Connection Manager where an unexpected server disconnection during the retrieval process from a connection pool would trigger an infinite loop. This issue was specifically limited to clusters operating in session pooling mode. | |
| Improved lock handling during change application to prevent data corruption from concurrent backends. Table and index locks are now held until transaction commit when applying changes. This ensures correct serialization and prevents data consistency issues during concurrent operations, specifically addressing risks introduced by parallel apply. | |
| Parallel Apply can now be enabled when PGD is deployed with community Postgres. Resolved an issue where enabling Parallel Apply on community PostgreSQL caused writer hangs and stuck subscriptions due to frequent lock timeouts. This fix allows Parallel Apply to be used reliably without requiring it to be disabled as a workaround. | 54565, 57005, 57327 |
| Improved Connection Manager resilience during file descriptor exhaustion. Connection Manager now automatically recovers from file descriptor exhaustion. If the system limit is reached, the process will close listening sockets and automatically restart after 30 seconds to restore connectivity. | |
| Fixed false error reporting during node group creation. Resolved a race condition where *bdr.create_node_group* could return an error even if the group was successfully created. The function now correctly waits for node creation to fully complete before returning. | |
| Fixed proxy configuration loss after *bdr_init_physical*. Fixed an issue where proxy configurations were not preserved after running *bdr_init_physical* due to Raft snapshot restores failing to persist proxy settings. | 58319 |

continues on next page

Table 9 – continued from previous page

| Description | Addresses |
|---|---|
| Fixed stale *galloc* sequence allocation entries after *DROP* and *CREATE*. Fixed a bug where *sequence_alloc* entries persisted after a *galloc* sequence was dropped and recreated. The fix ensures proper cleanup propagation across the cluster, including witness nodes. | |
| Fixed *bdr.alter_sequence_set_kind* ignoring start values for *galloc* sequences. Fixed an issue where changing a sequence kind to *galloc* would silently discard the provided *start_value*. Additionally, resolved a bug where a *startval* of *0* was lost during WAL replication. | |
| Prevented duplicate data replication during analytics write leader transitions. Fixed an issue where the analytics replicator on a previous write leader could continue running simultaneously with the replicator on the new leader. This synchronization improvement ensures the old process stops completely before the new one initializes, preventing duplicate data inserts. This fix includes the addition of the *bdr.group_lease* catalog table and the *bdr.group_lease_override* function. | |

## 2.4 EDB Postgres Distributed 6.2.0 release notes

Released: 5 December 2025

EDB Postgres Distributed 6.2.0 includes new features, enhancements, and bug fixes focused on improving stability and reliability.

### 2.4.1 Highlights

- **First release with Postgres 18 support**: PGD 6.2.0 introduces support for the newly released PostgreSQL 18, EDB Postgres Extended (PGE) 18, and EDB Postgres Advanced Server (EPAS) 18.
- **Connection Manager enhancements**: PGD 6.2.0 introduces significant enhancements to Connection Manager, including LDAP authentication support and improved routing stability that prevents existing client connections from dropping during leader changes.
- **Easier adoption of Transparent Data Encryption (TDE)**: The `pgd node setup` and `pgd node upgrade` commands now include TDE options. This allows simplified adoption of TDE such as single-command conversion of non-TDE nodes to TDE nodes, significantly streamlining adopting TDE when upgrading PGD clusters running older PGE or EPAS versions (prior to v15).

### 2.4.2 Features

| Description | Addresses |
|---|---|
| Added TDE options to *pgd node upgrade* and *pgd node setup* commands.<hr/>The *pgd node upgrade* command now supports in-place conversion of a non-TDE PGD node to a TDE-enabled node with the *--key-unwrap-command* and *--copy-by-block* options. The *pgd node setup* command now includes comprehensive options to configure TDE during the initial provisioning of a new cluster node. These new options are: *--data-encryption*, *--copy-key-from*, *--key-wrap-command*, *--key-unwrap-command*, *--no-key-wrap*, and *--data-encryption-keylen* (accepts the AES key length). | |

continues on next page

Table 10 – continued from previous page

| Description | Addresses |
|---|---|
| Added *bdr.stat_receiver_transactions* view.<hr/>The view provides real-time information on transactions in the receiver pipeline. This view reports on transactions currently being received, sent to writers for processing, and indicates whether a transaction is being streamed to a file or writer, including the writer's application progress. | |
| Added support for LDAP authentication to Connection Manager.<hr/>Connection Manager now supports the same LDAP authentication features as PostgreSQL, with the exception that OpenLDAP configuration files and environment variables are not supported. | |
| Added detailed monitoring columns to *bdr.writers* view.<hr/>The *bdr.writers* view now includes several new columns for enhanced monitoring. It now reports the transaction apply source, the total size of the transaction, the progress of the application, the last change applied, and the upstream LSN corresponding to that last applied change. | |

### 2.4.3 Enhancements

| Description | Addresses |
|---|---|
| Inferred default *superuser* value for EPAS cluster *setup* command.<hr/>When provisioning an EPAS cluster, the *--superuser* option for the *pgd node setup* command now automatically defaults to *enterprisedb* unless explicitly specified. | |
| Added *Details* column to *pgd cluster verify --verbose --arch* output.<hr/>The *pgd cluster verify* command now includes a *Details* column in the architecture verification section when using the *--verbose* option. This column provides specific, detailed information about any issues encountered during the architecture check. | |
| Enhanced *pgd node part* command for reuse of parting node name.<hr/>The behavior of the *pgd node part* command is modified so it now only waits until the node is renamed (allowing its original name to be reused) and continues the complex parting operation in the background. This enhancement also deprecates the *–-no-wait* option when all nodes in the PGD cluster are running version 6.2 or above. | |
| Allow reuse of parting node names with *bdr.part_node*.<hr/>The PGD node management interface now allows you to reuse the names of parting nodes even while the parting operation is still in progress. This is accomplished through by setting the new *concurrent=true* option when calling the *bdr.part_node* function. | |
| Support for encrypted server side certificates.<hr/>Connection Manager now supports the PostgreSQL *ssl_passhrase_command* option to decrypt the server private key. | 53695 |
| Improved PGD database user authentication for *pgd node setup*.<hr/>The command has been enhanced to read user authentication credentials from a password file. This file location is resolved by checking the *--dsn* global option, the *PGPASSFILE* environment variable, or the *~/.pgpass* file, in that order. | |
| The *bdr.prefer_analytics_engine* setting now works for Iceberg REST catalogs by using the *analytics_write_catalog* node group option.<hr/>Previously, this setting only applied to PGFS storage locations. | |
| Added *--group filter* to *pgd nodes list* and *pgd raft show* commands.<hr/>The *--group* filter provides the capability to filter the output by a specified data group when administering a multi-group PGD cluster. | |

### 2.4.4 Bug Fixes

| Description | Addresses |
|---|---|
| Fixed an issue where the *pgd node upgrade* command failed with an *invalid_slots* error when upgrading to PostgreSQL 18 or later.<hr/>The BDR extension has been updated to improve compatibility with the *pg_upgrade* utility, which ensures the upgrade process runs cleanly. | |
| Fixed contradictory output for switchover with Connection Manager DSN.<hr/>Resolved an issue where the *pgd group set-leader* command displayed confusing messages of both success and failure. This issue occurred because after the switchover completed successfully, the Connection Manager immediately reset the connection due to the write-leader change. | |
| Fixed unexpected subscription and slot creation involving subscriber-only and witness nodes.<hr/>This fix prevents the incorrect creation of both subscriptions and replication slots between subscriber-only nodes and witness nodes. The fix also addresses a problem seen during upgrades from PGD 5.9.0 to 6.2.0 where an unexpected slot was left behind on the witness node. | |
| Improved Connection Manager routing change logic.<hr/>Resolved an issue where the Connection Manager dropped existing client connections whenever a write-leader or read-only node routing change occurred. With this fix, client connections are no longer affected by a routing refresh, as long as the PGD node they are currently connected to remains operational. | |
| Fixed an issue whereby a connection is silently dropped in the Connection Manager.<hr/>The fix improves the handling of authentication failures in the Connection Manager. | |
| Fixed an issue whereby Connection Manager crashed when full hostname is used in the *pg_hba.conf address* field.<hr/>The fix also corrects the handling of hostname suffixes. Since hostname matching is not supported by the Connection Manager, HBA rules using hostnames or hostname suffixes are now correctly ignored by the Connection Manager, preventing instability. | |
| Fixed writer error *could not open temporary file* after restart.<hr/>This issue occurred when streaming a large transaction, and the writer restarted only to find the temporary file name had been incorrectly freed. The bug was introduced with parallel apply and is resolved now. | 53341 |
| Fixed an issue to ensure that the values of *node_uuid* and *node_kind* of a node are preserved after joining a cluster in a different PGD version.<hr/>This bug caused the *node_uuid* in system tables (*bdr.local_node* and *bdr.node*) to diverge after a full cluster upgrade. The fix ensures that the joining node's UUID and kind are correctly preserved throughout the process. | |
| Allow rename of sequence or schema containing galloc sequences.<hr/>Resolved an issue where concurrent operations, such as renaming a sequence or schema containing a galloc sequence, could cause problems with Raft synchronization. The fix introduces a global UUID identifier for each sequence in the *bdr.sequence_kind* catalog, which remains constant despite any sequence or schema renames. | |
| Added version and distribution checks for physical join in *pgd node setup*.<hr/>The *pgd node setup* command now performs stricter compatibility checks before allowing a physical join:<ul><li>If the BDR version of the local and remote nodes are not an exact match, the command attempts a logical join instead of a physical join.</li><li>If the Postgres major version of the local and remote nodes are not the same, the command attempts a logical join instead of a physical join.</li><li>If the PostgreSQL distribution (flavor) of the local and remote nodes are not the same, the command exits with an error.</li></ul> | |

continues on next page

Table 12 – continued from previous page

| Description | Addresses |
|---|---|
| Fixed backward compatibility issue for *pgd group* commands.<hr/>Resolved a backward compatibility issue where PGD 6.x CLI commands, such as *pgd group show*, reported the error *No query found for version* when invoked against an older PGD 5.x cluster. The fix ensures these commands function correctly across major PGD versions. | |
| Fixed invalid system identifier check for *pgd node setup*.<hr/>Resolved an issue where the *pgd node setup* command incorrectly reported a mismatch error when validating a valid base backup node of the given remote node. | |
| Handle returned *CONTEXT* value for *bdr.run_on_x* functions.<hr/>The PGD CLI now adjusts the database connection's verbosity to ignore the *CONTEXT* value, which previously added noise to the output, especially for errors from *bdr.run_on_x* functions. | |
| Fixed the *bdr.msgb_connect* hang by introducing a connection timeout.<hr/>Resolved an issue where the *bdr.msgb_connect* function could hang indefinitely due to a missing network timeout setting. The fix adds a timeout to the function. | 53442 |
| Fixed *pgd node setup* to create a PGD node from an existing data directory.<hr/>Resolved an issue which prevented the *pgd node setup* command from successfully creating and configuring a PGD node using an existing data directory. | |
| Connection Manager now starts and responds on its configured ports on all nodes, even those that are not currently part of a node group with routing enabled.<hr/>In these cases, the configuration inherited from the top-level node group is automatically applied. | |
| Fixed a unique constraint violation error that could occur during an *UPDATE* operation on tables configured with both primary and unique keys.<hr/>Previously, an *UPDATE* conflict with an *update_missing* resolution (which attempts to insert a new record) did not check for conflicts against existing primary or unique keys, leading to an error. This issue is now fixed by treating the scenario as an *insert_exists* conflict and applying the appropriate resolution logic. | 50539, 51417, 54323 |
| The *ALTER TABLE SET pgd.replicate_to_analytics* property can now be applied to individual partitions.<hr/>This allows for granular control over which partitions replicate data to analytical targets. | |
| Fixed an issue that caused consensus breakage when a duplicate value was assigned to either the *analytics_storage_location* or *analytics_write_catalog* settings. | |

## 2.5 EDB Postgres Distributed 6.1.2 release notes

Released: 6 November 2025

EDB Postgres Distributed 6.1.2 includes only bug fixes focused on improving stability and reliability. No new features or enhancements are included in this release. Please see the details below.

### 2.5.1 Bug Fixes

| Description | Addresses |
|---|---|
| Fixed ALTER TABLE failure during streamed transaction with table rewrite<hr/>The *ALTER TABLE* command failed to create a temporary table when using table rewrite support and streaming to a file or a writer. The fix implements a stream message callback method, preventing a potential data loss scenario during these operations. | |

## 2.6 EDB Postgres Distributed 6.1.1 release notes

Released: 09 October 2025

EDB Postgres Distributed 6.1.1 includes only bug fixes focused on improving stability and reliability. No new features or enhancements are included in this release. Please see the details below.

### 2.6.1 Bug Fixes

| Description | Addresses |
|---|---|
| Fix physical join when running 'pgd node setup'<hr/>The *pgd node setup* command in the PGD CLI has been fixed to correctly set up a physical copy as a physical replica. Previously, a bug prevented the command from properly configuring physical replication which could result in some changes being missed on the new node. For any nodes added using *pgd node setup* with earlier versions of the PGD CLI, we recommend performing a consistency check. | |
| Fix handling of DDL commands that trigger table rewrites<hr/>Executing specific DDL operations (such as *ALTER TABLE ... ADD COLUMN ... DEFAULT <volatile expression>* or *ALTER TABLE ... ALTER COLUMN ... SET DATA TYPE* when not binary-compatible) could cause replication failures. With this fix in PGD 6.1.1, replication now correctly handles DDL commands that trigger table rewrites. | |
| Connection Manager respects currentSchema JDBC parameter<hr/>Previously, a JDBC client connected through the PGD Connection Manager would have the *currentSchema* parameter ignored. As a result, a connection's *search_path* would incorrectly default to *public*. This issue has now been resolved. | |
| prefer_analytics_engine works with Iceberg REST Catalog tables<hr/>The *prefer_analytics_engine* flag has been updated to correctly function with tables that are replicated to Iceberg REST Catalog. Setting *bdr.prefer_analytics_engine = true* will now cause a query referencing a table with *pgaa.replicate_to_analytics = true* to be executed by the analytics engine when possible. | |

## 2.7 EDB Postgres Distributed 6.1.0 release notes

Released: 19 August 2025

This is a minor release of PGD 6, which introduces direct upgrade support from PGD 4.4.0+ and PGD 5.9.0+. This makes it the first version of PGD 6 that allows for a seamless, in-place upgrade from older versions. In addition to this key feature, PGD 6.1.0 also includes a variety of new features, enhancements, and important bug fixes.

### 2.7.1 Highlights

- **Seamless upgrades from PGD 4 and PGD 5 to PGD 6**: PGD 4.4.0+ and PGD 5.9.0+ users can now perform a direct upgrade to PGD 6, beginning with PGD 6.1.0, ensuring a smooth transition to the latest features.
- **Replication to Apache Iceberg**: PGD now supports replication to Apache Iceberg when using the Analytics Accelerator. This new capability allows you to stream data directly from your PGD cluster to an Iceberg data lake and also query the Iceberg data directly from within PGD. This significantly expands your options for large-scale data analytics and integration.

- **PGD 6 now supports Materialized Views**: The CREATE, ALTER, REFRESH, and DROP commands are replicated across the cluster, ensuring that materialized views are consistent on all nodes. When a materialized view is created or concurrently refreshed, the data is populated on the upstream node and then replicated to others, providing full support for this functionality.
- **PGD 6 is now on IBM LinuxONE**: PGD's native capabilities are expanded with new support for IBM LinuxONE, offering optimized uptime, throughput, and energy efficiency.

### 2.7.2 Features

| Description | Addresses |
|---|---|
| Added CLI commands to get and update Postgres settings (GUC).<hr/>Add the *pgd node get-config* and *pgd node set-config* commands to allow users to retrieve and modify Postgres settings (GUC) via the CLI. | |
| New *drop_after_retention_period* argument added to the *bdr.autopartition* function<hr/>`bdr.autopartition()` now takes an additional boolean argument *drop_after_retention_period* which can be set to false if the user does not want the partition to be dropped, but only detached. | 82332 |
| Simplified Node Configuration with the PGD CLI<hr/>The pgd command-line tool now includes new commands to make managing individual node configurations easier. You can now use *pgd node <node-name> get-config* and *pgd node <node-name> set-config* to retrieve and update specific Postgres settings (GUCs) for any node in your cluster directly from the CLI. | |
| PGD CLI View Commit Scopes<hr/>The PGD CLI now includes a new command, *pgd commit-scopes list*, that allows you to view the commit scope configuration for your cluster. This command provides details on all commit scopes, including their rules and the groups they apply to, which improves the visibility and management of your data consistency settings. | |
| PGD CLI - Part a Node from a Cluster<hr/>The PGD CLI now includes a new command, *pgd node part*, which provides a streamlined way to remove a node from an active cluster. This command automates the process of calling the *node_part* SQL function and reports on its progress, simplifying node management directly from the command line. | |
| Leader DDL Lock<hr/>New lock type leader DDL lock is used by default for locking DDL statements. This lock locks on write-leaders only, not requiring majority nodes to participate in the locking operation. Old behavior can be restored by adjusting *bdr.ddl_locking* configuration parameter. | |
| Replication of Roles to witness nodes<hr/>Roles are now replicated to witness nodes when they join the nodegroup, and also as and when they are created or altered on data nodes. This is useful mainly when dealing with GUC permissions. | |
| Added command to list commit scopes for PGD cluster.<hr/>Add a command *pgd commit-scopes list* to list commit scopes information for PGD cluster. | |
| Added command to part a PGD node from cluster.<hr/>Add a command *pgd node part* to part a PGD node from cluster. | |

### 2.7.3 Enhancements

| Description | Addresses |
|---|---|
| Update the *pgd group get-option* and *pgd group show* commands to show default values.<hr/>The *pgd group get-option* and *pgd group show* commands will now display the default values for all options. The *pgd group show* command will display the source of the option values as well. | |
| Apply the PGD Essential constraints for CLI commands.<hr/>The CLI will check for and enforce the PGD Essential constraints for the commands as applicable. See <a href="https://www.enterprisedb.com/docs/pgd/latest/essential-how-to/pgd-cli/#pgd-essential-constraints">PGD Essential CLI Constraints</a> for more details. | |
| Improve Raft snapshot export/import to reset Raft completely<hr/>We've made improvements to how our Raft consensus mechanism handles snapshots. This ensures that in the rare event of a data consistency issue, restoring a snapshot will now fully reset the system's state, allowing operations to resume smoothly from a consistent starting point. | |
| EPAS Interval Partition Support for PGD<hr/>A new enhancement allows partitions created by EDB Postgres Advanced Server (EPAS) AutoPartition to work correctly within a multi-node PGD environment. This fix ensures partitions are created consistently on each node and are visible across all nodes. | |
| Improved max_connections Error Reporting<hr/>The Connection Manager now provides more helpful and specific error messages when a max_connections limit is reached. Users will now see a clear message like "sorry, too many clients already," making it much easier to diagnose connection issues. | |
| Support for commands that generate Table Rewrites in an explicit transaction block<hr/>PGD now supports executing ALTER TABLE commands that generate a table rewrite within a transaction block. Multiple commands are supported in a single transaction block. This simplifies complex DDL operations by applying all changes as a single atomic unit and provides performance optimizations when multiple commands target the same table. | |
| Postgres Compatibility with PGD - Phase 5<hr/>This is a placeholder for future updates regarding Postgres compatibility. | |

### 2.7.4 Changes

| Description | Addresses |
|---|---|
| Removal of listen_address Group Option<hr/>The *listen_address* option has been removed from *pgd group set-option* to simplify configuration. The Connection Manager will now use the value from the *listen_addresses* GUC directly, resolving issues with setting hostnames or multiple addresses that were previously unsupported. | |

### 2.7.5 Bug Fixes

| Description | Addresses |
|---|---|
| Fix the *pgd cluster verify* command warning for *witness-only* group.<hr/>The *pgd cluster verify* command displayed warning for *Data nodes per group* check for *witness-only* group. The check is not applicable for *witness-only* groups and hence the status *Ok* will be displayed. | |

continues on next page

Table 18 – continued from previous page

| Description | Addresses |
|---|---|
| Fix the data dir check for the setup command.<hr/>The *pgd node setup* command checks for the presence of *PG_VERSION* file in the given data dir option. However the code incorrectly uses the *bindir* option instead of *pgdata*. | |
| Fix for Mixed_version Clusters<hr/>We've resolved a bug that could cause replication to fail in a mixed-version PGD cluster (e.g., combining Postgres 12 and Postgres 16), an architecture used temporarily during postgres rolling upgrade. Previously, DDL statements on a newer node could lead to replication slot disconnection on older nodes, with an "unrecognized configuration parameter" error. This fix ensures that replication now functions correctly during a rolling upgrade or in a temporary mixed-version state. | 46373 |
| Connection Manager Behavior on Routing Changes<hr/>The Connection Manager now correctly handles connections following a change in cluster routing configuration (e.g., switching between global and no routing). This resolves unexpected behavior and ensures client connections are routed as intended after a configuration update. | |
| Fix for node_pkey Errors When Rebuilding Parted Nodes<hr/>A fix has been added to address a bug that could cause a consensus worker to crash with a duplicate key value violates unique constraint "node_pkey" error when a parted node was being rebuilt. This fix ensures that you can now successfully rebuild a parted node using *bdr_init_physical* without encountering this issue. | 44467, 45084 |
| PGD CLI cluster verify Command Fix for Witness-Only Groups<hr/>The pgd cluster verify command no longer displays a warning for witness-only groups that do not contain data nodes. This resolves an issue where the command incorrectly flagged these groups, ensuring that the verification output is now accurate for all cluster types. | |
| Raft Fix for Parting Lagging Nodes<hr/>Raft no longer fails to recover after a heavily-lagging node that was previously a leader is parted from the cluster. This fixes a bug where the Raft leader could crash with a "BDR node XXX not found" error, which previously required a manual, forceful reset of the Raft state. Now, Raft correctly handles this scenario and recovers automatically, ensuring cluster stability. | 50099 |
| Fix for PGD CLI Multi-Group Setup<hr/>A bug has been fixed in the PGD CLI setup command that prevented the creation of multi-group clusters. Previously, the command would fail with an unhelpful error when attempting to add a node to a second group. The fix ensures that the command now correctly handles the creation of nodes in new groups, allowing for successful multi-group cluster setup. | |
| Fix for CLI setup Command GUC Update<hr/>The PGD CLI setup command no longer fails when attempting to update Postgres settings on remote nodes. This fix resolves a "malformed array literal" error that occurred during the cluster setup, improving the reliability of the initialization process. | |
| Fix for "Clock Has Moved Backwards" Error with Snowflake Sequences<hr/>A race condition has been fixed that could cause a "clock has moved backwards" error when using Snowflake sequences, even when the system clock was functioning correctly. This fix ensures the sequence correctly handles concurrent access, preventing this error and improving the reliability of Snowflake sequence generation. | 49376, 43659 |
| Fix a crash with pg_failover_slots<hr/>Ensure that CDC hooks are no-op when recovery is in progress or the backend is not connected to a database. This can definitely happen if *pg_failover_slots* is active. | |
| Fix a crash on a stat view for connection manager stat<hr/>The view *bdr.stat_connection_manager_hba_file_rules* had some parsing issues on the *pg_hba.conf* file specifically in the *Address/Mask* column for the "all" value. | |

continues on next page

Table 18 – continued from previous page

| Description | Addresses |
|---|---|
| Node kind check<hr/>PGD CLI now has a verify check for nodes not having a node kind properly set. This check emits a warning on nodes that are coming from a previous major version upgrade that missed setting the node kind. | |
| Skip the node GUCs update with *pgd node setup* command.<hr/>The *pgd node setup* verifies the Postgres settings while setting up a PGD node. The command will now skip the node GUCs update as the *bdr_superuser* role, used by CLI is not allowed to run ALTER SYSTEM queries. The command will continue to verify the Postgres settings and will display a warning message if any of the settings require modification. | |

## 2.8 EDB Postgres Distributed 6.0.2 release notes

Released: 25 June 2025

This is the first patch release of EDB Postgres Distributed 6.0, which includes bug fixes and enhancements to the new features introduced in PGD 6.0.

### 2.8.1 Bug Fixes

| Description | Addresses |
|---|---|
| Fixed an issue with the slot names generated by *node setup* command.<hr/>The command generated the slot names as UUID values with underscore (_) which is not compatible with PGD slot names. | |
| Fixed segmentation fault in bdr.stat_connection_manager_get_stats().<hr/>The memory returned by the function was allocated in an incorrect memory context. | |
| Ensure that GRANT check for BDR objects works for EPAS object types<hr/>The mechanism for determining whether user can be granted permissions on BDR extension objects was failing for EPAS-specific objects. This was a regression from PGD release 5.7.0 and is now fixed. | 49649 |
| Renamed the check *Clock Skew* to *Clock Drift* for *pgd cluster show* command.<hr/>The check is renamed to be consistent with the terminology used at other places in PGD. | |
| Added a warning message if CLI fails to fetch value of a GUC for *pgd cluster verify* command.<hr/>The command could crash if the CLI could not fetch a value for a GUC for some reason. | |

## 2.9 EDB Postgres Distributed 6.0.1 release notes

Released: 9 June 2025

PGD 6 delivers simpler, more resilient high availability for Postgres. Traditional streaming replication often requires downtime for upgrades and routine maintenance—and depends on complex tooling. PGD solves these challenges with a built-in, logical replication-based architecture that enables online upgrades and maintenance without disrupting applications, helping teams keep services running smoothly even during operational changes. It also provides seamless failover and eliminates the need for external proxies, load balancers, or consensus systems.

### 2.9.1 Highlights

- **New built-in Connection Manager**: Automatically routes client connections to the correct node, simplifies application architecture, supports dynamic topology changes, and includes a built-in session pooler and dedicated read/write and read-only ports, all without external software or complex configuration. This new component replaces PGD Proxy, which is no longer available starting with PGD 6.
- **Predefined Commit Scopes**: Simplify consistency choices with built-in transaction durability profiles—no complicated setup needed. Choose the right balance of performance and protection, with scopes defined in system catalogs and ready to use out of the box.
- **New CLI command for Cluster Setup**: The pgd node setup command now enables initial cluster creation and node addition directly from the command line. This gives users more flexibility in how they deploy PGD and allows deployment tools to standardize on a consistent method.

### 2.9.2 Features

| Description | Addresses |
|---|---|
| Built-in connection manager<hr/>New built-in connection manager which handles routing of connections automatically and allows enforcing of read-only connections to non-leader. | |
| CLI cluster setup<hr/>The PGD CLI now allows initial cluster setup as well as adding nodes from command-line using *pgd node setup* command. | |
| Set sequence kind on group create/join<hr/>Transform the sequences in distributed based on the *bdr.default_sequence_kind* GUC when creating/joining a bdr group instead of when creating the node as done in older versions. | |
| Set startvalue for distributed sequences automatically<hr/>Set the startvalue for galloc sequences to the following valid number after the last used by the local sequence. With this change, when creating distributed sequences and specifically galloc, there is no need to adjust the startvalue based on what might be already used. | |
| Enabling of automatic sync and reconciliation<hr/>Automatic synchronization and reconciliation of node states is now enabled by default. This means that nodes will automatically synchronize their state with the leader node and reconcile any differences without requiring manual intervention. Read more in the <a href="/pgd/latest/reference/node_management/automatic_sync">documentation</a>. | |
| Add node_uuid column to bdr.node and bdr.local_node<hr/>The node_uuid uniquely identifies instance of a node of a given name. Random node_uuid is generated when node is created and remains constant for the lifetime of the node. The node_id column is now derived from node_uuid instead of node name.For the time being a node needs to be fully parted before before node of the same name can be rejoined, this may be relaxed in future releases to permit rejoin as soon as part_node process for the old instance has commenced and before it completed.For the time being upgrades from older PGD versions and mixed-version operation in clusters with older PGD nodes are not supported. This limitation will be addressed in future releases. | |

continues on next page

Table 20 – continued from previous page

| Description | Addresses |
|---|---|
| Change replication origin and slot naming scheme<hr/>Replication origin and slot names now use node uuid and thus correspond to particular incarnation of a node of a given name. Similarly node group uuid is used instead of group name. Hash of database name is used in lieu of database name.Please note that origin and node names should be treated as opaque identifiers from user's perspective, one shouldn't rely on the structure of these names nor expect these to be particularly meaningful to a human operator.The new naming scheme is as follows:<h4>Slots Naming Convention</h4><ul><li>normal slot to a node => `bdr_node_<targetuuid>_<dbhash>`</li><li>join slot for node => `bdr_node_<targetuuid>_<dbhash>_tmp`</li><li>group slot for a topgroup => `bdr_group_<topgroupuuid>_<dbhash>`</li><li>slot for any forwarding + lead to lead => `bdr_node_<targetuuid>_<originidhex>_<dbhash>`</li><li>analytics slot => `bdr_analytics_<groupuuid>_<dbhash>`</li><li>decoding slot => `bdr_decoder_<topgroupuuid>_<dbhash>`</li></ul><h4>Origins Naming Convention:</h4><ul><li>normal origin to a node => `bdr_<originuuid>_<dbhash>`</li><li>fwd origin to a source node => `bdr_<originuuid>_<sourceoidhex>_<dbhash>`</li></ul> | |
| Limit on the number of node groups allowed in the system for PGD Essential.<hr/>Ensure that no more than three node groups (one top group and two subgroups) can exist at any given time. If the limit is exceeded, an error is raised. | |
| Enforced PGD Essential limits - data node count<hr/>Don't allow PGD Essential clusters to join more than 4 data nodes. | |
| Added *bdr.wait_node_confirm_lsn()* function which waits until a given reaches a given LSN<hr/>`bdr.wait_node_confirm_lsn(`) will look at the confirmed_flush_lsn of the given node when available, otherwise it will query *pg_replication_origin_progress()* of that node, and wait for the specified LSN to be reached by said node. | |
| Subscriber-only nodes can now be added to data node groups<hr/>In previous versions, subscriber-only nodes could only be added to node groups of type "subscriber-only". In PGD 6, a subscriber-only node can be also be added to a data node group by specifying node_kind='subscriber_only' when using create_node. The join_node_group can then be done using a data node group. | |
| Add *bdr.local_analytics_slot_name()* SQL function.<hr/>Returns name of analytics slot. This merely produces the correct name irrespective of whether analytics feature is in use. | |
| Add node_uuid column to *bdr.node_summary* view.<hr/>Added to complement the addition of the node_uuid column to bdr.node and bdr.local_node | |

### 2.9.3 Enhancements

| Description | Addresses |
|---|---|
| Multiple conflicting rows resolution<hr/>Both *pk_exists* and *multiple_unique_conflicts* conflict types can now resolve more than one conflicting row by removing any old rows that are part of the conflict. The *multiple_unique_conflicts* now defaults to *update_if_newer* resolver, so it does not throw error by default anymore. | |

continues on next page

Table 21 – continued from previous page

| Description | Addresses |
|---|---|
| Improved *bdr.stat_activity* view<hr/>The *backend_type* now shows consistent worker type for PGD workers without the extra process identification. The *wait_event_type* and *wait_event* include more wait events now, instead of showing "extension" for some events. Also, connection management related columns are added to show real client address/port and whether the session is read-only. | |
| The PARTED node is removed automatically from all nodes in the cluster.<hr/>From PGD 6.0.0, bdr.part_node functionality is enhanced to remove the parted node's metadata automatically from all nodes in the cluster.<ul><li>For local node, it will remove all the node metadata, including information about remote nodes.</li><li>For remote node, it removes only metadata for that specific node. Hence with this release</li><li>A node will remain in PART_CLEANUP state till group slots of all nodes are caught up to all the transactions originating from the PARTED node</li><li>A node will not remain in PARTED state as the node is removed as soon as it moves to PARTED state.</li></ul> | |
| The *--summary* and *--options* flags for *pgd node show* CLI command.<hr/>Add the *--summary* and *--options* flags to *pgd node show* command to filter the output of the *pgd node show* command. This also maintains symmetry with other *show* commands. | |
| More GUCs verfied in *pgd cluster verify* CLI command.<hr/>Add the *bdr.lock_table_locking* and *bdr.truncate_locking* GUCs to list of GUCs verfied in *pgd cluster verify* command. | |
| Table rewriting *ALTER TABLE... ALTER COLUMN* calls are now supported.<hr/>Changing a column's type command which causes the whole table to be rewritten and the change isn't binary coercible is now supported:<pre><code class="language-sql">CREATE TABLE foo (c1 int,c2 int, c3 int, c4 box, UNIQUE(c1, c2) INCLUDE(c3,c4)); ALTER TABLE foo ALTER c1 TYPE bigint; – results into table rewrite </pre>*This also includes support for `ALTER TYPE* when using the *USING* clause:<pre><code class="language-sql">CREATE TABLE foo (id serial primary key,data text); ALTER TABLE foo ALTER data TYPE BYTEA USING data::bytea; `</pre>Table rewrites can hold an AccessExclusiveLock for extended periods on larger tables. | |
| Restrictions on non-immutable *ALTER TABLE... ADD COLUMN* calls have been removed.<hr/>The restrictions on non-immutable *ALTER TABLE... ADD COLUMN* calls have been removed. | |
| Synchronize roles and tablespaces during logical join<hr/>Roles and tablespaces are now synchronized before the schema is restored from the join source node. If there are already existing roles or tablespaces (or EPAS profiles, they will be updated to have the same settings, passwords etc. as the ones from the join source node. System roles (i.e. the ones created by initdb) are not synchronized. | |
| Introduce *bdr.node_group_config_summary* view<hr/>The new *bdr.node_group_config_summary* view contains detailed information about group options, including effective value, source of the effective value, default value, whether the value can be inherited, etc. This is in similar spirit to *pg_settings* | |
| Leader DML lock<hr/>New lock type leader DML lock is used by default for locking DDL statements that need to block DML. This lock locks on write-leaders only, no requiring all nodes to participate in the locking operation. Old behavior can be restored by adjusting *bdr.ddl_locking* configuration parameter. | |
| Disabling bdr.xact_replication in run_on_* functions<hr/>Functions *run_on_nodes*, *run_on_all_nodes* and *run_on_group* now sets *bdr.xact_replication* to *off* by default. | |

continues on next page

Table 21 – continued from previous page

| Description | Addresses |
|---|---|
| Replica Identity full by default<hr/>The *auto* value for *bdr.default_replica_identity* changed to REPLICA IDENTITY FULL. This setting prevents some edge cases in conflict detection between inserts, updates and deletes across node crashes and recovery.When the PGD group is created and the database of the initial PGD node is not empty (i.e. has some tables with data) the REPLICA IDENTITY of all tables will be set according to *bdr.default_replica_identity*. | |
| Tablespace replication as a DDL operation is supported.<hr/>Tablespace operations *CREATE/ALTER/DROP TABLESPACE* are now replicated as a DDL operation. Where users are running a configuration with multiple nodes on the same machine, you will need to enable the developer option <a href="https://www.postgresql.org/docs/current/runtime-config-developer.html#GUC-ALLOW-IN-PLACE-TABLESPACES">`allow_in_place_tablespace`</a>. | |
| Improve the CLI debug messages.<hr/>Improve the formating of the log messages to be more readable and symmetrical with Postgres log messages. | |
| New column for *pgd cluster verify --settings* CLI command output.<hr/>Add the *recommended_value* column to the result of the *pgd cluster verify --settings* command. The column will not be displayed in tabular output but will be displayed in JSON output. | |
| Display sorted output for CLI.<hr/>The output for the commands with tabular output are now sorted by the resource name. Commands that display more than one resource will sort output by each resource column in order. | |
| Subscriber-only nodes replication.<hr/>Subscriber-only nodes now receive data only after it has been replicated to majority of data nodes. This does not require any special configuration. Subsequently bdr.standby_slot_names and bdr.standby_slots_min_confirmed options are removed as similar physical standby functionality is provided in pg_failover_slots extension and in PG17+. | |
| automatic node sync and reconciliation is enabled by default.<hr/>The GUC <a href="/pgd/latest/reference/tables-views-functions/pgd-settings#bdrenable_auto_sync_reconcile">`bdr.enabl was off by default, but is made on by default in 6.0. This GUC setting ensures that when a node is down for some time, all other nodes get caught up equally with respect to this node automatically. It also ensures that if there are any prepared transactions that are orphaned by the node going down, they are resolved, either aborted or committed as per the rules of the commit scope that created them. | |
| Remove the deprecated legacy CLI commands.<hr/>Remove the old (PGD 5 and below) CLI commands, which were deprecated but supported for backward compatibility. | |
| Commit scope logic is now only run on data nodes.<hr/>Previously, non-data nodes would attempt to handle, but not process commit scope logic, which could lead to confusing, albeit harmless log messages. | |
| Explicitly log the start and stop of dump and restore operations.<hr/>This provides greater visibility into the node cloning process and assists with debugging possible issues. | |

### 2.9.4 Changes

| Description | Addresses |
|---|---|
| Routing is now enabled by default on subgroups<hr/>Routing (and by extension raft) is now enabled by default on data-groups (subgroups with data nodes). | |
| Function *bdr.join_node_group* may no longer be executed in a transaction.<hr/>As it is not possible to roll back a group join, it can not form part of an idempotent transaction. | |
| Deprecated *pause_in_standby* parameter removed from function *bdr.join_node_group()*.<hr/>`pause_in_standby` has been deprecated since PGD 5.0.0. Logical standby nodes should be specified as such when executing *bdr.create_node()* | |
| BDR global sequences can no longer created as or set to `UNLOGGED`<hr/>Unlogged BDR sequences may display unexpected behaviour following a server crash. Existing unlogged BDR sequences may be converted to logged ones. | |

### 2.9.5 Bug Fixes

| Description | Addresses |
|---|---|
| Fix the CLI *pgd cluster show* command issues on a degraded cluster.<hr/>The *pgd cluster show* command failed with an error for clock drift if only one node was up and running in a N node cluster. The command now returns valid output for the other components, *health* and *summary*, while reporting an appropriate error for *clock-drift*. | |
| Fix the CLI *pgd node show* command issue if a non-existent node is specified.<hr/>The *pgd node show* command crashed if a non-existent node is specified to the command. The command is fixed to fail gracefully with appropriate error message. | |
| Fix the broken replication slot issue after rolling Postgres upgrade using *pgd node upgrade* command.<hr/>Merge writer origin positions to the parent origin during node upgrade. In PGD 5 and older writer origin names map to parent origin id which may change during inplace upgrade. | 46412,48747 |
| Fixed the timestamp parsing issue for *pgd replication show* CLI command.<hr/>The *pgd replication show* command previously crashed when formatting EPAS timestamps. | |
| Fixed issue where parting node may belong to a non-existing group<hr/>When parting a given node, that same node may have subscriptions whose origin was already parted and the group dropped. Previously this would break PGD, and has since been fixed. | |
| num_writers should be positive or -1<hr/>The num_writers option, used in bdr.alter_node_group_option() and bdr.alter_node_group_config() should be positive or -1. | |
| Fix replication breakage with updates to non-unique indexes<hr/>Fixes the case where an update to a table with non-unique indexes results in the ERROR *concurrent INSERT when looking for delete rows*, which breaks replication. | 43523,43802,45244,47815 |
| Fix Raft leader election timeout/failure after upgrade<hr/>Ensure that any custom value set in the deprecated GUC *bdr.raft_election_timeout* is applied to the replacement *bdr.raft_global_election_timeout* | |
| Ensure that disables subscriptions on subscriber-only nodes are not re-enabled<hr/>During subscription reconfiguration, if there is no change required to a subscription, do not enable it since it could have been disabled explicitly by the user. Skip reconfiguring subscriptions if there are no leadership changes. | 46519 |
| Subscriber-only nodes will not take a lock when running DDL<hr/>Subscriber-only nodes will no longer attempt to take a lock on the cluster when running DDL. The DDL will be executed locally and not replicated to other nodes. | 47233 |

continues on next page

Table 23 – continued from previous page

| Description | Addresses |
|---|---|
| Fixed hang in database system shutdown.<hr/>Fixed non-transactional WAL message acknowledgment by downstream that could cause a WAL sender to never exit during fast database system shutdown. | 49022 |
| Fixed deadlock issue in bdr_init_physical.<hr/>Fixed deadlock between bdr_init_physical cleaning unwanted node data and concurrent monitoring queries. | 46952 |
| Fixed new cluster node consistency issue.<hr/>Fixed an issue when new node joining the cluster finishes CATCHUP phase before getting its replication progress against all data nodes. This may cause new node being out of sync with the cluster. | |
| Ensure correct sequence type is displayed in CREATE SEQUENCE warnings<hr/>In some cases, warning messages referred to *timeshard* when the sequence was actually *snowflakeid*. | |

CHAPTER 3

# Overview

EDB Postgres Distributed (PGD) provides multi-master replication and data distribution with advanced conflict management, data-loss protection, and throughput up to 5X faster than native logical replication . It also enables distributed Postgres clusters with high availability up to five 9s.

## 3.1 Known issues and limitations

### 3.1.1 Known issues

These are currently known issues in EDB Postgres Distributed {Version 6.4}. These known issues are tracked in PGD's ticketing system and are expected to be resolved in a future release.

- Replication of materialized views only works with newly-created materialized views. Materialized views that were created before upgrading to {Version 6.4}.0 do not get replicated after upgrading. To prevent this issue, you can do one of the following:
- Before upgrading to PGD {Version 6.4}.0., manually create the materialized view on each node. - OR
- After upgrading to PGD {Version 6.4}.0., drop and create the materialized view so that it gets replicated.
- If the resolver for the `update_origin_change` conflict is set to `skip` , `synchronous_commit=remote_apply` is used, and concurrent updates of the same row are repeatedly applied on two different nodes, then one of the update statements might hang due to a deadlock with the PGD writer. As mentioned in *bdr_read_all_conflicts* , `skip` isn't the default resolver for the `update_origin_change` conflict, and this combination isn't intended to be used in production. It discards one of the two conflicting updates based on the order of arrival on that node, which is likely to cause a divergent cluster. In the rare situation that you do choose to use the `skip` conflict resolver, note the issue with the use of the `remote_apply` mode.
- The Decoding Worker feature doesn't work with CAMO/Eager/Group Commit. Installations using CAMO/Eager/Group Commit must keep `enable_wal_decoder` disabled.
- Lag Control doesn't adjust commit delay in any way on a fully isolated node, that's in case all other nodes are unreachable or not operational. As soon as at least one node connects, replication Lag Control picks up its work and adjusts the PGD commit delay again.

- For time-based Lag Control, PGD currently uses the lag time, measured by commit timestamps, rather than the estimated catch up time that's based on historic apply rates.
- Changing the CAMO partners in a CAMO pair isn't currently possible. It's possible only to add or remove a pair. Adding or removing a pair doesn't require a restart of Postgres or even a reload of the configuration.
- Group Commit can't be combined with *Commit At Most Once* .
- Transactions using Eager Replication can't yet execute DDL. The TRUNCATE command is allowed.
- Parallel Apply isn't currently supported in combination with Group Commit. Make sure to disable it when using Group Commit by either (a) Setting `num_writers` to 1 for the node group using bdr.alter_node_group_option or (b) using the GUC bdr.writers_per_subscription . See

Configuration of generic replication .

- There currently is no protection against altering or removing a commit scope. Running transactions in a commit scope that's concurrently being altered or removed can lead to the transaction blocking or replication stalling completely due to an error on the downstream node attempting to apply the transaction. Make sure that any transactions using a specific commit scope have finished before altering or removing it.
- The PGD CLI can return stale data on the state of the cluster if it's still connecting to nodes that were previously parted from the cluster. Edit the

pgd-cli-config.yml file, or change your --dsn

settings to ensure only active nodes in the cluster are listed for connection.

To modify a commit scope safely, use bdr.alter_commit_scope .

- DDL run in serializable transactions can face the error: `ERROR: could not serialize access due to read/write dependencies among transactions` . A workaround is to run the DDL outside serializable transactions.
- The EDB Postgres Advanced Server 17 data type

**BFILE is not**
currently supported. This is due to `BFILE` being a file reference that is stored in the database, and the file itself is stored outside the database and not replicated.

- When running PGD on community PostgreSQL with Parallel Apply enabled, writer processes may terminate unexpectedly due to lock timeouts. This occurs because PGD sets a 30 second `lock_timeout` in the writer processes, which can be insufficient when one writer waits on a transaction lock held by another writer that is processing a large transaction. When the timeout is reached, the waiting writer exits, causing other writers and the receiver to terminate as well, which resets the replication process. As a workaround, disable parallel apply by setting `bdr.enable_parallel_apply = off` or migrate to an EDB Postgres variant. This issue is resolved in PGD 6.3.
- In PGD 6.3.0, user login attributes aren't replicated to peer nodes after a `CREATE USER` statement. Users created on one node can log in on the originating node but are blocked on all other nodes because the login attribute is missing from the replicated role definition. This issue is resolved in PGD 6.3.1.

To identify affected roles, run the following command in psql:

```
\du
```

In the **Attributes** column, any role that should have login access but displays `Cannot login` is affected.

Alternatively, query the system catalog directly:

```
SELECT rolname AS role_name, rolcanlogin
FROM pg_roles
WHERE rolcanlogin = false;
```

To resolve the issue without upgrading, manually grant login access on each affected node:

```
ALTER USER <username> WITH LOGIN;
```

### 3.1.2 Limitations

Take these EDB Postgres Distributed (PGD) design limitations into account when planning your deployment.

#### Nodes

- PGD can run hundreds of nodes, assuming adequate hardware and network. However, for mesh-based deployments, we generally don't recommend running more than 48 nodes in one cluster. If you need extra read scalability beyond the 48-node limit, you can add subscriber-only nodes without adding connections to the mesh network.
- The minimum recommended number of nodes in a group is three to provide fault tolerance for PGD's consensus mechanism. With just two nodes, consensus would fail if one of the nodes were unresponsive. Consensus is required for some PGD operations, such as distributed sequence generation. For more information about the consensus mechanism used by EDB Postgres Distributed, see *Architectural details* .

#### Multiple databases on single instances

Support for using PGD for multiple databases on the same Postgres instance is **deprecated** beginning with PGD 5 and will no longer be supported with PGD 6. As we extend the capabilities of the product, the added complexity introduced operationally and functionally is no longer viable in a multi-database design.

It's best practice and we recommend that you configure only one database per PGD instance.

The tooling such as the CLI and Connection Manager currently codify that recommendation.

While it's still possible to host up to 10 databases in a single instance, doing so incurs many immediate risks and current limitations:

- If PGD configuration changes are needed, you must execute administrative commands for each database. Doing so increases the risk for potential inconsistencies and errors.
- You must monitor each database separately, adding overhead.
- Connection Manager works at the Postgres instance level, not at the database level, meaning the leader node is the same for all databases.
- Each additional database increases the resource requirements on the server. Each one needs its own set of worker processes maintaining replication, for example, logical workers, WAL senders, and WAL receivers. Each one also needs its own set of connections to other instances in the replication cluster. These needs might severely impact performance of all databases.
- Synchronous replication methods, for example, CAMO and Group Commit, won't work as expected. Since the Postgres WAL is shared between the databases, a synchronous commit confirmation can come from any database, not necessarily in the right order of commits.
- CLI integration assumes one database.

### Durability options (Quorum Commit/Group Commit/CAMO)

There are various limits on how the PGD durability options work. These limitations are a product of the interactions between Quorum Commit, Group Commit, and CAMO, and how they interact with PGD features such as the *Decoding worker* and *Using with transaction streaming* .

Also, there are limitations on interoperability with legacy synchronous replication, interoperability with explicit two-phase commit, and unsupported combinations within commit scope rules.

The following limitations apply to the use of commit scopes and the various durability options they enable.

### General durability limitations

- ```
  :ref:`Legacy synchronous replication using PGD <Legacy synchronous replication
  ↪using PGD>`  uses a mechanism for transaction confirmation different from the
  ↪one used by CAMO, Eager, Group Commit, and Quorum Commit. The two aren't
  ↪compatible, so don't use them together. Whenever you use Quorum Commit, Group
  ↪Commit, CAMO, or Eager, make sure none of the PGD nodes are configured in
  ↪`synchronous_standby_names` .
  ```
- Postgres two-phase commit (2PC) transactions (that is,

**PREPARE TRANSACTION**
) can't be used with CAMO, Group Commit, Quorum Commit, or Eager because those features use two-phase commit underneath.

### Group Commit

*Group Commit (legacy)* enables configurable synchronous commits over nodes in a group. If you use this feature, take the following limitations into account:

- Not all DDL can run when you use Group Commit. If you use unsupported DDL, a warning is logged, and the transactions commit scope is set to local. The only supported DDL operations are:
- Nonconcurrent `CREATE INDEX`
- Nonconcurrent `DROP INDEX`
- Nonconcurrent `REINDEX` of an individual table or index
- `CLUSTER` (of a single relation or index only)
- `ANALYZE`
- `TRUNCATE`
- Explicit two-phase commit isn't supported by Group Commit as it already uses two-phase commit.
- Combining different commit decision options in the same transaction or combining different conflict resolution options in the same transaction isn't supported.
- Currently, Raft commit decisions are extremely slow, producing very low TPS. We recommended using them only with the `eager` conflict resolution setting to get the Eager All-Node Replication behavior of PGD 4 and older.

### Eager

*Eager conflict resolution* is available through Group Commit and is always active in Quorum Commit. It avoids conflicts by eagerly aborting transactions that might clash. When used through Group Commit, it's subject to the same limitations as Group Commit.
  ```

Eager doesn't allow the `NOTIFY` SQL command or the `pg_notify()` function. It also doesn't allow `LISTEN` or `UNLISTEN`.

### Quorum Commit

*QUORUM COMMIT* is the commit scope kind for distributed transaction consistency. If you use this feature, take the following limitations into account:

- All nodes in the cluster must be running PGD 6.4.0 or later. A cluster with any node on an earlier version can't use Quorum Commit, including clusters undergoing a rolling upgrade.
- `MAJORITY` with an explicit group list, `ANY n`, and `NOT` aren't supported. Supported groups are `ALL`, `MAJORITY ORIGIN GROUP`, `MAJORITY CLUSTER`, and `MAJORITY FOR EACH GROUP`.
- Transaction streaming is supported only in `file` or `off` mode. Other streaming modes aren't compatible with Quorum Commit.
- Explicit two-phase commit (2PC) isn't supported by Quorum Commit, as Quorum Commit already uses 2PC internally.
- `DEGRADE TO` isn't supported.

### CAMO

*Commit At Most Once* (CAMO) is a feature that aims to prevent applications committing more than once. If you use this feature, take these limitations into account when planning:

- CAMO is designed to query the results of a recently failed COMMIT on the origin node. In case of disconnection, the application must request the transaction status from the CAMO partner. Ensure that you have as little delay as possible after the failure before requesting the status. Applications must not rely on CAMO decisions being stored for longer than 15 minutes.
- If the application forgets the global identifier assigned, for example, as a result of a restart, there's no easy way to recover it. Therefore, we recommend that applications wait for outstanding transactions to end before shutting down.
- For the client to apply proper checks, a transaction protected by CAMO can't be a single statement with implicit transaction control. You also can't use CAMO with a transaction-controlling procedure or in a `DO` block that tries to start or end transactions.
- CAMO resolves commit status but doesn't resolve pending notifications on commit. CAMO doesn't allow the `NOTIFY` SQL command or the `pg_notify()` function. They also don't allow `LISTEN` or `UNLISTEN`.
- When replaying changes, CAMO transactions might detect conflicts just the same as other transactions. If timestamp-conflict detection is used, the CAMO transaction uses the timestamp of the prepare-on-the-origin node, which is before the transaction becomes visible on the origin node itself.
- CAMO isn't currently compatible with transaction streaming. Be sure to disable transaction streaming when planning to use CAMO. You can configure this option globally or in the PGD node group. See *Transaction streaming configuration*.
- CAMO isn't currently compatible with decoding worker. Be sure to not enable decoding worker when planning to use CAMO. You can configure this option in the PGD node group. See *Decoding worker disabling*.
- Not all DDL can run when you use CAMO. If you use unsupported DDL, a warning is logged and the transactions commit scope is set to local only. The only supported DDL operations are:
- Nonconcurrent `CREATE INDEX`
- Nonconcurrent `DROP INDEX`
- Nonconcurrent `REINDEX` of an individual table or index

- `CLUSTER` (of a single relation or index only)
- `ANALYZE`
- `TRUNCATE`
- Explicit two-phase commit isn't supported by CAMO as it already uses two-phase commit.
- You can combine only CAMO transactions with the `DEGRADE TO` clause for switching to asynchronous operation in case of lowered availability.

#### Mixed PGD versions

PGD was developed to *enable rolling upgrades of PGD* by allowing mixed versions of PGD to operate during the upgrade process. We expect users to run mixed versions only during upgrades and, once an upgrade starts, that they complete that upgrade. We don't support running mixed versions of PGD except during an upgrade.

#### Other limitations

This noncomprehensive list includes other limitations that are expected and are by design. We don't expect to resolve them in the future. Consider these limitations when planning your deployment:

- A `galloc` sequence might skip some chunks if you create the sequence in a rolled back transaction and then create it again with the same name. Skipping chunks can also occur if you create and drop the sequence when DDL replication isn't active and then you create it again when DDL replication is active. The impact of the problem is mild because the sequence guarantees aren't violated. The sequence skips only some initial chunks. Also, as a workaround, you can specify the starting value for the sequence as an argument to the `bdr.alter_sequence_set_kind()` function.

## 3.2 PGD compatibility

### 3.2.1 PGD compatibility with PostgreSQL versions

The following table shows the major versions of PostgreSQL that EDB Postgres Distributed (PGD) is compatible with.

| PGD 6 | Postgres Version |
|---|---|
| *UUID-related changes in PGD 6* | 18.1.0+ |
| *UUID-related changes in PGD 6* | 17.5.0+ |
| *UUID-related changes in PGD 6* | 16.9.0+ |
| *UUID-related changes in PGD 6* | 15.13.0+ |
| *UUID-related changes in PGD 6* | 14.18.0+ |

EDB recommends that you use the latest minor version of any Postgres major version with a supported PGD.

### 3.2.2 PGD compatibility with operating systems and architectures

The following tables show the versions of EDB Postgres Distributed and their compatibility with various operating systems and architectures.

#### Linux

| Operating System | x86_64 (amd64) | ppc64le | arm64/ aarch64 |
|---|---|---|---|
| RHEL 8 | Yes | Yes | |
| RHEL 9 | Yes | Yes | Yes |
| Oracle Linux 8 | Yes | | |
| Oracle Linux 9 | Yes | | |
| Rocky Linux/AlmaLinux | Yes | | |
| SUSE Linux Enterprise Server 15SP6 | Yes | Yes | |
| SUSE Linux Enterprise Server 15SP7 | Yes | Yes | |
| Ubuntu 22.04 | Yes | | |
| Ubuntu 24.04 | Yes | | |
| Debian 12 | Yes | | Yes |
| IBM Linux One | Yes | | |

**Note**

See *Compatibility changes* for previous versions of PGD.

## 3.3 Postgres Distributed terminology

This terminology list includes terms associated with EDB Postgres Distributed (PGD) that you might be unfamiliar with.

### 3.3.1 Asynchronous replication

A type of replication that copies data to other PGD cluster members after the transaction completes on the origin node. Asynchronous replication can provide higher performance and lower latency than *Legacy synchronous replication using PGD* . However, asynchronous replication can see a lag in how long changes take to appear in the various cluster members. While the cluster will be *Eventual consistency* , there's potential for nodes to be apparently out of sync with each other.

### 3.3.2 Commit scopes

Rules for managing how transactions are committed between the nodes and groups of a PGD cluster. Used to configure *Legacy synchronous replication using PGD* , *QUORUM COMMIT* , *SYNCHRONOUS COMMIT and GROUP COMMIT* , *CAMO or commit-at-most-once* , *Eager conflict resolution* , *LAG CONTROL* , and other PGD features.

### 3.3.3 CAMO or commit-at-most-once

High-value transactions in some applications require that the application successfully commits exactly once, and in the event of failover and retrying, only once. To ensure this happens in PGD, CAMO can be enabled, allowing the application to actively participate in the transaction.

### 3.3.4 Conflicts

As data is replicated across the nodes of a PGD cluster, there might be occasions when changes from one source clash with changes from another source. This is a conflict and can be handled with conflict resolution. Conflict resolution is a set of rules that decide which source is correct or preferred. Conflicts can also be avoided with conflict-free data types.

### 3.3.5 Connection Manager

To ensure that clients can connect to the right nodes in the distributed cluster, PGD provides a connection management system that allows clients to connect to the appropriate nodes based on their needs.

This system is designed to ensure that clients can access the data they need while maintaining the performance and availability of the cluster. Unlike Proxy systems, this connection management system is built into the database instance itself, allowing for more efficient and reliable connections.

### 3.3.6 Consensus

How *Using Raft for the commit decision* makes group-wide decisions. Given a number of nodes in a group, Raft looks for a consensus of the majority (number of nodes divided by 2 plus 1) voting for a decision. For example, when a write leader is being selected, a Raft consensus is sought over which node in the group will be the write leader. Consensus can be reached only if there's a quorum of voting members.

### 3.3.7 Cluster

Generically, a cluster is a group of multiple systems arranged to appear to end users as one system. See also *pgd cluster* and *Postgres cluster* .

### 3.3.8 DDL (data definition language)

The subset of SQL commands that deal with defining and managing the structure of a database. DDL statements can create, modify, and delete objects (that is, schemas, tables, and indexes) in the database. Common DDL commands are CREATE, ALTER, and DROP.

### 3.3.9 DML (data manipulation language)

The subset of SQL commands that deal with manipulating the data held in a database. DML statements can create, modify, and delete rows in tables in the database. Common DML commands are INSERT, UPDATE, and DELETE.

### 3.3.10 Durability

Durability guarantees that once a transaction is committed, it remains committed even in the event of a system failure. PGD manages this through *Commit scopes* .

### 3.3.11 Eager

A synchronous commit mode that avoids conflicts by detecting incoming potentially conflicting transactions and "eagerly" aborts one of them to maintain consistency.

### 3.3.12 Eventual consistency

A distributed computing consistency model stating changes to the same item in different cluster members will eventually converge to the same value. Asynchronous logical replication with conflict resolution and conflict-free replicated data types exhibit eventual consistency in PGD.

### 3.3.13 Failover

The automated process that recognizes a failure in a highly available database cluster and takes action to maintain consistency and availability. The goal is to minimize downtime and data loss.

### 3.3.14 Group commit

A synchronous commit mode that requires more than one PGD node to successfully receive and confirm a transaction at commit time.

### 3.3.15 Immediate consistency

A distributed computing model where all replicas are updated synchronously and simultaneously. This model ensures that all reads after a write completes will see the same value on all nodes. The downside of this approach is its negative impact on performance.

### 3.3.16 Lag Control

Lag Control is a throttling mechanism used to ensure that secondary nodes do not fall too far behind the primary. If a node's replication delay exceeds a set threshold, PGD automatically throttles the ingestion on the origin node to allow the rest of the cluster to catch up.

### 3.3.17 Logical replication

A more efficient method of replicating changes in the database. While physical streaming replication duplicates the originating database's disk blocks, logical replication instead takes the changes made, independent of the underlying physical storage format, and publishes them to all systems that subscribed to see the changes. Each subscriber then applies the changes locally. Logical replication can't support most DDL commands.

### 3.3.18 Logical standby node

Logical standby nodes are used to provide a read-only replica of the data in the cluster. They are similar to subscriber-only nodes, but they are designed to be more flexible and can be used in a wider range of scenarios.

### 3.3.19 Node

A general term for an element of a distributed system. A node can play host to any service. In PGD, *Creating PGD nodes* run a Postgres database, the BDR extension and the Connection Manager.

Typically, for high availability, each node runs on separate physical hardware, but that's not always the case.

### 3.3.20 Node groups

PGD nodes in PGD clusters can be organized into groups to reflect the logical operation of the cluster. For example, the data nodes in a particular physical location can be part of a dedicated node group for the location.

Each group uses the *Monitoring Raft consensus* algorithm to elect a leader; in data-specific groups, this leader acts as the write leader, which handles all incoming write operations to maintain consistency.

Groups manage data replication, with a "top-level" group representing the entire cluster and serving as the parent to all other sub-groups.

### 3.3.21 PGD cluster

A group of multiple redundant database systems and proxies arranged to avoid single points of failure while appearing to end users as one system. PGD clusters can be run on Docker instances, cloud instances or "bare" Linux hosts, or a combination of those platforms. A PGD cluster can also include backup nodes. The data nodes in a cluster are grouped together in a top-level group and into various local *Node groups* .

### 3.3.22 PGD node

In a PGD cluster are nodes that run databases and participate in the PGD cluster. A typical PGD node runs a Postgres database, the BDR extension, and the Connection Manager. PGD modes are also referred to as *data nodes*, which suggests they store data. However, some PGD nodes, specifically *Witness nodes* , don't do that.

### 3.3.23 Physical replication

By making an exact copy of database disk blocks as they're modified to one or more standby cluster members, physical replication provides an easily implemented method to replicate servers. But there are restrictions on how it can be used. For example, only one master node can run write transactions. Also, the method requires that all cluster members are on the same major version of the database software with the same operating system and CPU architecture.

### 3.3.24 Postgres cluster

Traditionally, in PostgreSQL, a number of databases running on a single server is referred to as a cluster (of databases). This kind of Postgres cluster isn't highly available. To get high availability and redundancy, you need a *pgd cluster* .

### 3.3.25 Quorum

A quorum is the minimum number of voting nodes needed to participate in a distributed vote. It ensures that the decision made has validity. For example, when a *Using Raft for the commit decision Monitoring Raft consensus* is needed by a PGD cluster, a minimum number of voting nodes participating in the vote are needed. With a 5-node cluster, the quorum is 3 nodes in the cluster voting. A consensus is 5/2+1 nodes, 3 nodes voting the same way. If there are only 2 voting nodes, then a consensus is never established. Quorums are required in PGD for *DDL locking details* and Raft decisions.

### 3.3.26 Quorum Commit

A commit scope kind that guarantees distributed transaction consistency by coordinating the commit decision across all participating nodes before any node commits locally. All nodes commit together or roll back together. Eager conflict resolution is always active.

### 3.3.27 Replicated available fault tolerance (Raft)

A consensus algorithm that uses votes from a quorum of machines in a distributed cluster to establish a consensus. PGD uses Raft within groups (top-level or local) to establish the node that's the write leader.

### 3.3.28 Read scalability

The ability of a system to handle increasing read workloads. For example, PGD can introduce one or more read replica nodes to a cluster and have the application direct writes to the primary node and reads to the replica nodes. As the read workload grows, you can increase the number of read replica nodes to maintain performance.

### 3.3.29 Subscription

PGD nodes will publish changes being made to data to nodes that are interested. Other PGD nodes will ask to subscribe to those changes. This behavior creates a subscription and is the mechanism by which each node is updated. PGD nodes bidirectionally subscribe to other PGD nodes' changes.

### 3.3.30 Switchover

A planned change in connection between the application or proxies and the active database node in a cluster, typically done for maintenance.

### 3.3.31 Synchronous replication

When changes are updated at all participating nodes at the same time, typically leveraging a two-phase commit. While this approach replicates changes and resolves conflicts before committing, a performance cost in latency occurs due to the coordination required across nodes.

### 3.3.32 Subscriber-only nodes

A PGD cluster is based around bidirectional replication. But in some use cases, such as needing a read-only server, bidirectional replication isn't needed. A subscriber-only node is used in this case. It subscribes only to changes in the database to keep itself up to date and provide correct results to any run directly on the node. This feature can be used to enable horizontal read scalability in a PGD cluster. They can be configured in two ways:

- Within a data group: These nodes act as read-only replicas alongside standard data nodes. While they provide local access to data, they do not benefit from specific replication optimizations.
- Within a subscriber-only group: This configuration lacks a write leader. All nodes are strictly read-only, allowing PGD to apply optional optimizations that streamline the replication process.

### 3.3.33 Two-phase commit (2PC)

A multi-step process for achieving consistency across multiple database nodes. The first phase sees a transaction prepared on an originating node and sent to all participating nodes. Each participating node validates that it can apply the transaction and signals its readiness to the originating node. This is the prepare phase. In the second phase, if all the participating nodes signal they're ready, the originating node proceeds to commit the transaction and signals the participating nodes to commit, too. This is the commit phase. If, in the prepare phase, any node signals it isn't ready, the entire transaction is aborted. This process ensures all nodes get the same changes.

### 3.3.34 Vertical scaling or scale up

A traditional computing approach of increasing a resource (CPU, memory, storage, network) to support a given workload until the physical limits of that architecture are reached, for example, Oracle Exadata.

### 3.3.35 Witness nodes

Witness nodes primarily serve to help the cluster establish a consensus. An odd number of data nodes is needed to establish a consensus. Where resources are limited, a witness node can be used to participate in cluster decisions but not replicate the data. Not holding the data means it can't operate as a standby server or provide majorities in synchronous commits.

### 3.3.36 Write leader

In an Always-on architecture, a node is selected as the correct connection endpoint for applications. This node is called the write leader. Once selected, the PGD Connection Manager routes queries and updates to it. With only one node receiving writes, unintended multi-node writes can be avoided. The write leader is selected by consensus of a quorum of data nodes. If the write leader becomes unavailable, the data nodes select another node to become write leader. Nodes that aren't the write leader are referred to as *shadow nodes*.

### 3.3.37 Writer

When a *Responding to a disabled subscription* delivers data changes to a PGD node, the database server tasks a worker process, called a writer, with getting those changes applied.

## 3.4 Quickstart guide

This guide will guide your through installing a basic PGD cluster, connect to the it, load data, and perform basic SQL operations.

- *Creating your first cluster* .
- *Working with SQL and the PGD cluster* .
- *Loading data into your PGD Cluster* .
- *Using PGD CLI* .

### 3.4.1 Creating your first cluster

This guide helps you create a local cluster using Docker Compose. This is a great way to get familiar with the EDB Postgres Distributed (PGD) features and functionality.

#### Prerequisites

- Docker and Docker Compose installed on your local machine.

#### Install the PGD Docker Quickstart kit

To create your first PGD cluster, you can use the Docker Compose file provided by EDB. This will set up a local cluster with three nodes, which is perfect for testing and development purposes.

1. Make sure you have Docker and Docker Compose installed on your local machine. You can follow the Docker installation guide if you haven't done so already.
2. Open a terminal and on the machine where you have docker installed, create a new directory for your PGD cluster, for example:

```
mkdir pgd-cluster
cd pgd-cluster
```

3. Run the following command to download the PGD Docker Compose file:

```
curl -L https://enterprisedb.com/docs/pgd/latest/overview/quickstart/assets/pgd_
↪quickstart.sh | bash
```

This will download the PGD Docker Quickstart kit, which includes the Docker Compose file and other necessary files to get started with PGD Essential.

4. Once the download is complete, you will need to prepare the environment for the PGD cluster. This is done by running the following command:

```
./qs.sh prepare
```

This command will create the necessary directories and files for the PGD cluster.

5. Now you have to build the Docker images for the PGD cluster. You can do this by running the following command:

```
export EDB_SUBSCRIPTION_TOKEN=...
./qs.sh build
```

This command will build the Docker image needed for the PGD Quickstart cluster.

6. After the images are built, you can start the PGD cluster using Docker Compose. Run the following command:

```
./qs.sh start
```

This command will start the Docker containers and create a local cluster with the default configuration, running in the background.

### Accessing the PGD Cluster

1. Once the containers are up and running, you can access the PGD cluster using the following command:

```
docker compose exec host-1 psql pgddb
```

This command will connect you directly to the first node of the cluster using the `psql` command-line interface.

This is how you would connect to the database for maintenance and management tasks.

For application and user access you will usually connect using the connection manager which, by default, is running on TCP port 6432 of all the hosts in the cluster.

1. You can connect to the write leader node in the cluster using the following command:

```
docker compose exec host-1 psql -h host-1 -p 6432 pgddb
```

You can replace `-h host-1` with the name of any host in the cluster, as they all run the connection manager.

If you have the psql client installed on your local machine, you can also connect to the cluster using the following command:

```
export PGPASSWORD=secret
psql -h localhost -p 6432 -U postgres pgddb
```

This connects to the connection manager running on the host-3 container on port 6432. This is then routed to the write leader node in the cluster.

```
pgddb=# select node_name from bdr.local_node_summary;
node_name
-----------
node-1
(1 row)
```

1. To use the PGD CLI from outside the containers, you can run the following command:

```
docker compose exec host-1 pgd nodes list
__OUTPUT__
 Node Name | Group Name | Node Kind | Join State | Node Status
- ----------+------------+-----------+------------+-------------
 node-1    | group-1    | data      | ACTIVE     | Up
 node-2    | group-1    | data      | ACTIVE     | Up
 node-3    | group-1    | data      | ACTIVE     | Up
```

This pgd command will lists the nodes in the cluster and their status.

You can also get a shell on the host-1 container and run the pgd command directly:

```
docker compose exec host-1 bash
pgd nodes list
__OUTPUT__
 Node Name | Group Name | Node Kind | Join State | Node Status
- ----------+------------+-----------+------------+-------------
 node-1    | group-1    | data      | ACTIVE     | Up
 node-2    | group-1    | data      | ACTIVE     | Up
 node-3    | group-1    | data      | ACTIVE     | Up
```

This will give you access to the PGD CLI and allow you to run any PGD commands directly on the host-1 container.

#### Next Steps

Now that you have created your first PGD cluster, you can explore the following topics:

- *Working with SQL and the PGD cluster* to understand how to connect and interact with the cluster using SQL commands.
- *Loading data into your PGD Cluster* into the cluster using the `COPY` command or `pg_dump` and `pg_restore` .
- *Using PGD CLI* to monitor and manage the cluster.

### 3.4.2 Working with SQL and the PGD cluster

The first step in working with your PGD cluster is to connect to it using SQL. You can do this using the `psql` command-line interface or any other SQL client that supports PostgreSQL.

The `psql` command is already installed and configured, if you are using the Docker Compose setup of the Quickstart kit. That means all you have to do is get a shell on one of the hosts so you can start working with it. You can sign into the `host-1` container to run commands within the cluster.

```
docker compose exec host-1 bash
```

This will give you a shell inside the `host-1` container where you can run PGD and SQL commands against the PGD cluster.

#### Connecting within the PGD Cluster

With PGD Essential, unless you are performing maintenance tasks, you will usually connect to the cluster using the connection manager, which is running on TCP port 6432 of all the hosts in the cluster.

You can connect to the write leader node in the cluster using the following command:

```
psql -h <host> -p 6432 -U <username> <database>
```

As we have a new cluster running with no users (apart from the `postgres` superuser) and one replicated database (`pgddb` ), you can connect to the cluster using the following command:

```
psql -h host-1 -p 6432 -U postgres pgddb
```

This connects to the connection manager running on the `host-1` container on port 6432, which is then routed to the write leader node in the cluster. You can replace `host-1` with the name of any host in the cluster, as they all run the connection manager.

If we run the following command, we can see which node we are connected to in the cluster:

```
select node_name from bdr.local_node_summary;
__OUTPUT__
 node_name
- ----------
 node-1
```

Which doesn't surprise us, as we connected to the `host-1` container, which is running the `node-1` node in the cluster.

If we exit `psql` , and reconnect with:

```
psql -h host-2 -p 6432 -U postgres pgddb
```

We can see that we are now connected to the `node-1` node in the cluster:

```
select node_name from bdr.local_node_summary;
 node_name
- ----------
 node-1
```

That's the connection manager routing us to the write leader node in the cluster, which is `node-1` . To confirm this, we can run:

```
\! pgd group group-1 show --summary
__OUTPUT__
 Group Property     | Value
- ------------------+---------
 Group Name         | group-1
 Parent Group Name  | pgd
 Group Type         | data
 Write Leader       | node-1
 Commit Scope       |
```

(You can use the `\!` command in `psql` to run shell commands directly from within the `psql` session.)

### Working with SQL

Now that you are connected to the cluster, you can start working with SQL commands. You can create tables, insert data, and run queries just like you would in a regular PostgreSQL database.

For example, you can create a table and insert some data:

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
```

(continues on next page)

(continued from previous page)

```
);
INSERT INTO users (name, email) VALUES
(Alice, alice@example.com),
(Bob, bob@example.com);
```

You can then query the data:

```
SELECT * FROM users;
__OUTPUT__
 id |  name   |        email
- ---+--------+---------------------
  2 | Alice   |      alice@example.com
  3 | Bob     |      bob@example.com
(2 rows)
```

You can also run more complex queries, join tables, and use all the features of PostgreSQL. It's not withing the scope of this guide to cover all SQL commands, but you can refer to the PostgreSQL documentation for more information on SQL syntax and commands.

### Differences with PGD

What is important is that those SQL commands are replicated across the cluster. PGD has taken care of the replication for you. For example, that `serial` key has automatically been converted to a globally unique key across the cluster, so you can insert data on any node in the cluster and it will be replicated to all other nodes. For PGD Essential, this is less important as you are required to connect to the write leader, but with PGD Expanded, you can connect to any node in the cluster and run SQL commands, and this automatic change enables you to do that without worrying about conflicts or duplicates. With PGD Essential you are future proofed and can easily move to PGD Expanded later, with no changes to your SQL commands or application code.

### Next Steps

Now that you have connected to your PGD cluster and run some SQL commands, you can explore the following topics:

- *Loading data into your PGD Cluster* to learn how to import data from external sources.
- *Using PGD CLI* to manage your PGD cluster from the command line.

## 3.4.3 Loading data into your PGD Cluster

PGD is, at its core, a Postgres database, so you can use the same tools and methods to load data into your PGD cluster as you would with any PostgreSQL database. To get you started, this guide will walk you through the process of loading data into your PGD cluster.

### Online CSV Importing

First, we are going to show how you can import data from an online CSV file into your PGD cluster. In this case, it's some historical baseball data from Baseball Databank . We are going to use the `\COPY` command in psql to import directly from a URL. One thing `\COPY` doesn't do is create the table for you, so we will need to create the table first.

Connect to your PGD cluster using `psql` , either using `docker compose exec host-1 psql` or if you have `psql` installed locally, using that to connect to port 6432 on your host machine.

```
CREATE TABLE batters (
                      id SERIAL,
                      playerid VARCHAR(9),
```

(continues on next page)

(continued from previous page)

```
                    yearid INTEGER,
                    stint INTEGER,
                    teamid VARCHAR(3),
                    lgid VARCHAR(2),
                    g INTEGER,
                    ab INTEGER,
                    r INTEGER,
                    h INTEGER,
                    "2b" INTEGER,
                    "3b" INTEGER,
                    hr INTEGER,
                    rbi INTEGER,
                    sb INTEGER,
                    cs INTEGER,
                    bb INTEGER,
                    so INTEGER,
                    ibb INTEGER,
                    hbp INTEGER,
                    sh INTEGER,
                    sf INTEGER,
                    gidp INTEGER,
                    PRIMARY KEY (id)
);
```

Now we can import the CSV data into the `batters` table using the `\COPY` command:

```
\COPY batters(playerid,yearid,stint,teamid,lgid,g,ab,r,h,"2b","3b",hr,rbi,sb,cs,bb,so,
↪ibb,hbp,sh,sf,gidp) FROM PROGRAM curl "https://raw.githubusercontent.com/cbwinslow/
↪baseballdatabank/master/core/Batting.csv" DELIMITER , CSV HEADER
```

This command uses `curl` to fetch the CSV file from the URL and pipes it directly into the `\COPY` command, which imports the data into the `batters` table. The batters(…) entry defines which fields in the row the CSV data should go to. The `DELIMITER ',' CSV HEADER` options specify that the file is a CSV, using commas, with a header row, that gets skipped.

Copy and the command and paste it into your `psql` session. If everything is set up correctly, you should see the data being imported without any errors. You should see output indicating the number of rows copied, like this:

```
COPY 110495
```

To verify that the data has been loaded correctly, you can run a simple query:

```
SELECT COUNT(*) FROM batters;
```

You should see a result like this:

```
 count
- ------
 110495
(1 row)
```

This confirms that 110,495 rows have been successfully imported into the `batters` table.

Let's quickly user it to work out who 1998's home run leader was

```
SELECT playerid, yearid, teamid, hr
FROM batters
WHERE yearid = 1998
ORDER BY hr DESC
LIMIT 1;
```

You should see output like this:

```
 playerid  | yearid | teamid | hr
- ----------+--------+--------+----
 mcgwima01 |   1998 | SLN    | 70
(1 row)
```

And if we want to put that into the context of the top 5 highest ranked home run hitters in 1998, we can do:

```
SELECT playerid, yearid, teamid,
       rank() OVER (PARTITION BY yearid ORDER BY hr desc) hr_rank,
       hr
FROM batters
WHERE yearid = 1998
ORDER BY hr_rank LIMIT 5;
```

You should see output like this:

```
 playerid  | yearid | teamid | hr_rank | hr
- ----------+--------+--------+---------+----
 mcgwima01 |   1998 | SLN    |       1 | 70
 sosasa01  |   1998 | CHN    |       2 | 66
 griffke02 |   1998 | SEA    |       3 | 56
 vaughgr01 |   1998 | SDN    |       4 | 50
 belleal01 |   1998 | CHA    |       5 | 49
(5 rows)
```

With PGD, you can enjoy the full power of PostgreSQL, including advanced SQL features like window functions, to analyze your data, but with the added benefit of it being fully replicated and highly available across multiple nodes even when a node goes offline.

### Next Steps

Now that you have loaded some data into your PGD cluster, you can explore the following topics:

- *Using PGD CLI* to manage your PGD cluster from the command line.

## 3.4.4 Using PGD CLI

PGD CLI is a command-line interface for managing and monitoring your EDB Postgres Distributed (PGD) clusters. It provides a set of commands to perform various operations on the cluster, such as creating nodes, joining nodes, and managing replication.

It's already installed and configured if you are using the *Creating your first cluster* .

Log into the first host in your PGD cluster:

```
docker compose exec host-1 bash
```

and check the version of PGD CLI:

```
pgd --version
__OUTPUT__
pgd-cli version {Version 6.4}.0
```

!!!note

You can also run any of the following commands from outside the containers, using the `docker compose exec` command to run them in the context of the first host in your PGD cluster:

```
docker compose exec host-1 pgd <command>
```

And you can run the `pgd` command from any host in the cluster, as they all have the PGD CLI installed and configured.

Start by viewing the cluster's overall status with the `pgd cluster show` command:

```
pgd cluster show
__OUTPUT__

#  Summary

 Group Name | Parent Group | Group Type | Node Name | Node Kind
- ----------+--------------+------------+-----------+-----------
 group-1    | pgd          | data       | node-1    | data
 group-1    | pgd          | data       | node-2    | data
 group-1    | pgd          | data       | node-3    | data
 pgd        |              | global     |           |

#  Health

 Check             | Status | Details
- ------------------+--------+-------------------------------------------------
 Connections       | Ok     | All BDR nodes are accessible
 Raft              | Ok     | Raft Consensus is working correctly
 Replication Slots | Ok     | All PGD replication slots are working correctly
 Clock Skew        | Ok     | Clock drift is within permissible limit
 Versions          | Ok     | All nodes are running the same PGD version

#  Clock Drift

 Reference Node | Node Name | Clock Drift
- ---------------+-----------+-------------
 node-3         | node-2    | *
 node-3         | node-1    | *
```

This command provides a summary of the cluster, its nodes, and their health status. It also shows the clock drift between nodes, which is important for replication consistency.

You can also view the status of individual nodes using the `pgd node show` command:

```
pgd node node-1 show
__OUTPUT__

#  Summary

 Node Property   | Value
```

(continues on next page)

(continued from previous page)

```
- ----------------+------------
 Node Name        | node-1
 Group Name       | group-1
 Node Kind        | data
 Join State       | ACTIVE
 Node Status      | Up
 Node ID          | 4153941939
 Snowflake SeqID | 1
 Database         | pgddb

#  Options

 Option Name     | Option Value
- ----------------+-----------------------------------------------------
 route_dsn       | port=5432 dbname=pgddb host=host-1 user=postgres
 route_fence     | false
 route_priority | -1
 route_reads     | true
 route_writes    | true
```

The structure of the pgd CLI commands is hierarchical, with commands grouped by functionality. You can view the available commands and their descriptions by running:

```
pgd --help
__OUTPUT__
Manages PGD clusters

Usage: pgd [OPTIONS] <COMMAND>

Commands:
  cluster       Cluster-level commands
  group         Group related commands
  groups        Groups listing commands
  node          Node related commands
  nodes         Nodes listing commands
  events        Event log commands
  replication   Replication related commands
  raft          Raft related commands
  commit-scope  Commit scope management commands
  assess        PGD compatibility assessment of Postgres server
  completion    Generate the autocompletion script for pgd for the specified shell

Options:
  -V, --version  Print version

Global Options:
  -f, --config-file <CONFIG_FILE>  Sets the configuration file path
      --dsn <DSN>                  Sets the PostgreSQL connection string e.g.
↪"host=localhost port=6000 user=postgres dbname=postgres" [env: PGD_CLI_DSN=]
  -o, --output <OUTPUT_FORMAT>     Sets the output format for tables [env: PGD_CLI_
↪OUTPUT=] [default: psql] [possible values: json, psql, modern, markdown, simple]
      --debug                      Print debug messages, useful while troubleshooting␣
```

(continues on next page)

(continued from previous page)

```
↪[env: PGD_CLI_DEBUG=]
  -h, --help                          Print help
```

Commands such as `group` , `node` take a group or a node name as their next argument, followed by a specific command. Commands such as `cluster` , `groups` , and `nodes` do not require a group or node name, as they operate at the cluster level or list all groups or nodes.

You can also get help for a specific command by running:

```
pgd <COMMAND> --help
```

### Viewing cluster status

To view the overall status of your PGD cluster, we have already used the `pgd cluster show` command. This shows all the cluster information. To see just the health status of the cluster, you can use the `--health` option:

```
pgd cluster show --health
__OUTPUT__
 Check              | Status | Details
- ------------------+--------+-----------------------------------------------------
 Connections        | Ok     | All BDR nodes are accessible
 Raft               | Ok     | Raft Consensus is working correctly
 Replication Slots  | Ok     | All PGD replication slots are working correctly
 Clock Skew         | Ok     | Clock drift is within permissible limit
 Versions           | Ok     | All nodes are running the same PGD version
```

Or if you want to see the summary status only, you can use the `--summary` option:

```
pgd cluster show --summary
__OUTPUT__
 Group Name | Parent Group | Group Type | Node Name | Node Kind
- -----------+--------------+------------+-----------+-----------
 group-1    | pgd          | data       | node-1    | data
 group-1    | pgd          | data       | node-2    | data
 group-1    | pgd          | data       | node-3    | data
 pgd        |              | global     |           |
```

### Viewing groups and group status

To view the status of all groups in the cluster, you can use the `pgd groups list` command:

```
pgd groups list
__OUTPUT__
 Group Name | Parent Group Name | Group Type | Nodes
- -----------+-------------------+------------+-------
 group-1    | pgd               | data       | 3
 pgd        |                   | global     | 0
```

Now we can see the top level group `pgd` and the data group `group-1` with 3 nodes in it. All nodes are a member of the top-level group which coordinates all activity across the cluster. The data group `group-1` is a group of three data nodes which are replicating data between themselves, routing incoming queries within the group to the write leader node in the group.

We can dig deeper into the group details using the `pgd group show` command:

```
pgd group group-1 show
__OUTPUT__

#  Summary

 Group Property    | Value
- ------------------+---------
 Group Name        | group-1
 Parent Group Name | pgd
 Group Type        | data
 Write Leader      | node-1
 Commit Scope      |

#  Nodes

 Node Name | Node Kind | Join State | Node Status
- ----------+-----------+------------+-------------
 node-1    | data      | ACTIVE     | Up
 node-2    | data      | ACTIVE     | Up
 node-3    | data      | ACTIVE     | Up

#  Options

 Option Name                       | Option Value
- ----------------------------------+----------------------
 analytics_storage_location        |  (inherited)
 apply_delay                       | 00:00:00 (inherited)
 check_constraints                 | true (inherited)
 default_commit_scope              |  (inherited)
 enable_raft                       | true
 enable_routing                    | true
 enable_wal_decoder                | false (inherited)
 http_port                         |  (inherited)
 location                          |
 num_writers                       | -1 (inherited)
 read_only_consensus_timeout       |  (inherited)
 read_only_max_client_connections  |  (inherited)
 read_only_max_server_connections  |  (inherited)
 read_only_port                    |  (inherited)
 read_write_consensus_timeout      |  (inherited)
 read_write_max_client_connections |  (inherited)
 read_write_max_server_connections |  (inherited)
 read_write_port                   |  (inherited)
 route_reader_max_lag              | -1
 route_writer_max_lag              | -1
 route_writer_wait_flush           | false
 streaming_mode                    | default (inherited)
 use_https                         | true
```

This command provides a summary of the group, its nodes, and their status. It also shows the group options, such as whether routing is enabled, the HTTP port for monitoring, and other configuration settings.

Like the cluster command, you can also use the `--summary` options to view just the summary of the group:

```
pgd group group-1 show --summary
__OUTPUT__
 Group Property    | Value
- ------------------+---------
 Group Name        | group-1
 Parent Group Name | pgd
 Group Type        | data
 Write Leader      | node-1
 Commit Scope      |
```

Now we can see the group is a child of the top-level group `pgd` , it is a data group, and the write leader node in the group is `node-1` . There are no commit scopes set for this group, which means it is using the default commit scope.

The `--nodes` option can be used to view the nodes in the group:

```
pgd group group-1 show --nodes
__OUTPUT__
 Node Name | Node Kind | Join State | Node Status
- ----------+-----------+------------+-------------
 node-1    | data      | ACTIVE     | Up
 node-2    | data      | ACTIVE     | Up
 node-3    | data      | ACTIVE     | Up
```

And, similarly, you can use the `--options` option to view the group options:

```
pgd group group-1 show --options
__OUTPUT__
 Option Name                        | Option Value
- -----------------------------------+----------------------
 analytics_storage_location         |  (inherited)
 apply_delay                        | 00:00:00 (inherited)
 check_constraints                  | true (inherited)
 default_commit_scope               |  (inherited)
 enable_raft                        | true
 enable_routing                     | true
 enable_wal_decoder                 | false (inherited)
 http_port                          |  (inherited)
 location                           |
 num_writers                        | -1 (inherited)
 read_only_consensus_timeout        |  (inherited)
 read_only_max_client_connections   |  (inherited)
 read_only_max_server_connections   |  (inherited)
 read_only_port                     |  (inherited)
 read_write_consensus_timeout       |  (inherited)
 read_write_max_client_connections  |  (inherited)
 read_write_max_server_connections  |  (inherited)
 read_write_port                    |  (inherited)
 route_reader_max_lag               | -1
 route_writer_max_lag               | -1
 route_writer_wait_flush            | false
 streaming_mode                     | default (inherited)
 use_https                          | true
```

As you can see, many of the options are inherited from the parent group, which is the top-level group `pgd` . The

`enable_raft` and `enable_routing` options are set to `true` , which means that the group is using Raft consensus for replication and routing queries (that are made through the connection manager port) to the write leader node.

Let's take a look at the parent group `pgd` using the `pgd group pgd show` command:

```
pgd group pgd show
__OUTPUT__

#  Summary

 Group Property    | Value
- ------------------+--------
 Group Name        | pgd
 Parent Group Name |
 Group Type        | global
 Write Leader      |
 Commit Scope      |
```

This shows that the top-level group `pgd` is a global group, which means it is not a data group and does not have any data nodes of its own. In this case, it is just userd to coordinate the activity of the data groups in the cluster. It does not have a write leader, as it does not have any data nodes.

The next part of the output shows the nodes in the group, which is empty:

```
#  Nodes

 Node Name | Node Kind | Join State | Node Status
- ----------+-----------+------------+-------------
```

The options for the `pgd` group are shown next:

```
#  Options

 Option Name                       | Option Value
- ----------------------------------+--------------
 analytics_storage_location        |
 apply_delay                       | 00:00:00
 check_constraints                 | true
 default_commit_scope              |
 enable_raft                       | true
 enable_routing                    | false
 enable_wal_decoder                | false
 http_port                         |
 location                          |
 num_writers                       | -1
 read_only_consensus_timeout       |
 read_only_max_client_connections  |
 read_only_max_server_connections  |
 read_only_port                    |
 read_write_consensus_timeout      |
 read_write_max_client_connections |
 read_write_max_server_connections |
 read_write_port                   |
 route_reader_max_lag              | -1
 route_writer_max_lag              | -1
```

(continues on next page)

(continued from previous page)

```
 route_writer_wait_flush           | false
 streaming_mode                    | default
 use_https                         | true
```

These are the options for the top-level group `pgd` . This is where `group-1` inherits its options from. Here though, the `enable_routing` option is set to `false` , which means that the top-level group does not route queries to any data nodes, because it does not have any data nodes of its own. The `enable_raft` option is set to `true` , which means that the top-level group uses Raft consensus to coordinate management of the cluster.

Where options are not set, the default values are used, such as the `apply_delay` option which is set to `00:00:00` , meaning there is no delay in applying changes to the cluster.

### Viewing nodes and node status

To view the status of all nodes in the cluster, you can use the `pgd nodes list` command:

```
pgd nodes list
__OUTPUT__
 Node Name | Group Name | Node Kind | Join State | Node Status
- ----------+------------+-----------+------------+-------------
 node-1    | group-1    | data      | ACTIVE     | Up
 node-2    | group-1    | data      | ACTIVE     | Up
 node-3    | group-1    | data      | ACTIVE     | Up
```

You can also view the status of a specific node using the `pgd node show` command:

```
pgd node node-1 show
__OUTPUT__

#  Summary

 Node Property   | Value
- ----------------+------------
 Node Name       | node-1
 Group Name      | group-1
 Node Kind       | data
 Join State      | ACTIVE
 Node Status     | Up
 Node ID         | 4153941939
 Snowflake SeqID | 1
 Database        | pgddb

#  Options

 Option Name    | Option Value
- ---------------+-----------------------------------------------------
 route_dsn      | port=5432 dbname=pgddb host=host-1 user=postgres
 route_fence    | false
 route_priority | -1
 route_reads    | true
 route_writes   | true
```

Here we can see more about the node itself. We can see the node's name and group it belongs to, that it is a data node, that it is actively joined to the group and that it is up and running. The node ID is a unique identifier for the node, and

the Snowflake SeqID is used for ordering events in the cluster. Finally, we can see that its database is `pgddb` , which is the default database created in the Quickstart Docker Compose kit.

The options for the node are shown next, and these are specific to this particular node:

- `route_dsn` is the connection string for the node, which is used by the connection manager to route queries to this node.
- `route_fence` is set to `false` , which means that the node does not have a fence set up to prevent routing queries to it.
- `route_priority` is set to `-1` , which means that the node does not have a specific priority for routing queries.
- `route_reads` and `route_writes` are both set to `true` , which means that the node can handle both read and write queries.

These are used by the connection manager when routing queries to the node. They are also how you can control which nodes are active, without taking them down. Setting `route_fence` to `true` will prevent the connection manager from routing queries to this node, while still allowing it to be part of the cluster and replicate data.

### Setting node options

You can set options for a node using the `pgd node set` command. For example, to set the `route_fence` option to `true` for the `node-1` , you can run:

```
pgd node node-1 set-option route_fence true
```

If we now try and connect to the `node-1` 's connection manager:

```
psql -h host-1 -p 6432
```

We get a connection. But it is not routed to the `node-1` node, as it is fenced off from routing queries. Instead, it is routed to the current write leader in the group, which is `node-2` :

```
select node_name from bdr.local_node_summary;
 node_name
 -----------
 node-2
(1 row)
```

If we exit and undo the fencing by running:

```
pgd node node-1 set-option route_fence false
```

We can now connect to the `node-1` node's connection manager again:

```
psql -h host-1 -p 6432
```

And we can see that we are now connected to the `node-1` node:

```
select node_name from bdr.local_node_summary;
 node_name
 -----------
 node-1
(1 row)
```

CHAPTER 4

# Planning your PGD deployment

Planning your EDB Postgres Distributed (PGD) deployment involves understanding the requirements of your application and the capabilities of PGD. This section provides an overview of the key considerations for planning your PGD deployment.

- Choosing your deployment pattern: Understand the deployment patterns PGD supports and choose the one that best fits your requirements.
- *Choosing a Postgres distribution* Choose the Postgres distribution to deploy with PGD.
- *Choosing your deployment method* Pick the deployment method that suits your needs.
- *Comparing replication solutions* A comparison of EDB Postgres Distributed with other replication solutions.
- *Choosing between Physical Streaming Replication (PSR) and EDB Postgres Distributed (PGD)* Evaluate the trade-offs between standard Postgres Streaming Replication and PGD to determine the right high-availability solution for your workload.
- Other considerations: Consider other factors that may affect your deployment.

## 4.1 Deployment patterns

PGD offers several deployment patterns, from a single high-availability (HA) group to globally distributed multi-region clusters. The pattern you choose determines where your data lives, which locations accept writes, and how the cluster recovers from failures. The key dimensions are the number of active write locations, the level of disaster recovery you need, and whether regulatory requirements constrain where data can be stored. Each choice carries implications for write latency, conflict handling, and operational complexity.

### 4.1.1 Choosing a pattern

Most PGD deployments start from one of four goals, each mapping to one or more patterns.

### Maintaining continuous availability

PGD keeps the database available through node failures, data center (DC) failures, and rolling maintenance operations including major Postgres version upgrades. Within a single location, the *Single data group, global read scaling* pattern provides HA across two or more nodes. For datacenter-level protection without the cost of a full second active site, the *Primary active group, DR group* pattern adds a reduced-capacity DR site that takes over if the primary DC fails.

### Writing across multiple regions

Applications serving users in multiple geographies benefit from placing write capacity close to those users. Both the *Two data groups, active-active* and *Three data groups, active-active-active* patterns run active-active: all locations accept writes and replicate to each other. The two data groups pattern suits a primary and secondary location. A witness-only third location achieves Raft majority without a full third data site, and three full data groups take Raft majority further by surviving an entire region failure without interrupting the remaining regions.

Active-active replication across regions makes write conflicts possible. PGD resolves conflicts automatically using configurable conflict resolution policies, but schemas and application writes need to be designed with conflict-awareness in mind.

### Enforcing data locality

Regulations such as GDPR and CCPA require that personal and sensitive data stays within specific jurisdictions. The *Multiple locations, data residency* pattern implements selective replication by data classification. Local-only data, including personally identifiable information (PII), financial records, and healthcare data, replicates only within its region. Global reference data, such as product catalogs and configuration, replicates everywhere. Each region runs independently with its own HA group.

### Scaling out reads

Read-heavy workloads can overwhelm a write cluster if not isolated. The *Single data group, global read scaling* pattern offloads analytics, reporting, and API read traffic to subscriber-only nodes. Subscriber-only nodes receive all changes from the write group but don't participate in writes or consensus, so adding them doesn't affect write performance or commit scope quorum. They can be placed in additional regions to reduce read latency for local users.

| Pattern | Locations | Active write locations | Disaster recovery |
|---|---|---|---|
| *Single data group, global read scaling* | 1 | 1 | Node failure only |
| *Two data groups, active-active* | 2 | 2 | DC failure |
| *Three data groups, active-active-active* | 3+ | 3+ | Region failure |
| *Single data group, global read scaling* | 1+ | 1 | Node failure |
| *Primary active group, DR group* | 2 | 1 | DC failure (manual) |
| *Multiple locations, data residency* | 3+ | 3+ | Region failure |

## 4.1.2 Architectural elements

Each node in a PGD cluster is a Postgres instance running the PGD extension. Nodes are organized into groups, and each group elects a write leader to coordinate writes. Connection Manager runs on each node and routes client

connections to the current write leader, enabling automatic failover without changes to the application connection string.

### Node types

A PGD cluster uses three node types, each serving a different role:

- **Data nodes:** Store and manage data, handle reads and writes, and participate in replication and consensus.
- **Subscriber-only nodes:** Receive all changes from the write group but don't accept writes or participate in consensus. They're suited to read-heavy workloads like analytics and reporting.
- **Witness nodes:** Participate in Raft consensus but don't store user data. A witness in a third location resolves split-brain scenarios when two data locations lose contact with each other.

### Node roles

Data nodes take on transient roles that transfer automatically between nodes as conditions change:

- **Write leader:** The current target for all write operations when applications connect through Connection Manager. If the write leader fails, another node is elected in seconds.
- **Raft leader:** Manages consensus decisions across the group, including write leader election and schema change coordination.

### Commit scopes

Commit scopes control the durability guarantees PGD provides for each transaction. The default is asynchronous replication with eventual consistency. Stronger options, from majority protect through Quorum Commit, add synchronous coordination at the cost of added write latency. Quorum Commit provides the strictest consistency guarantees, coordinating the commit decision across all participating nodes before any node commits locally. Each pattern page notes the commit scopes recommended for that topology. For the full reference, see *Commit scopes* .

### Single data group

The single data group is the foundational PGD deployment pattern, providing high availability within a single location. Three data nodes form a Raft consensus group, with Connection Manager handling write leader routing automatically.

Nodes can be spread across multiple availability zones for added resiliency, though doing so increases write latency when using majority or all protect commit scopes.

### Diagram legend

Symbols represent node types, group containers, and connection types.

### When to use this pattern

The single data group is the standard high availability (HA) deployment within a single data center. It tolerates single node failure and provides automatic failover in seconds, making it well suited to applications that need HA within a single region and can't afford extended downtime during major version upgrades or blocking maintenance tasks.

The advantages of this pattern include the following:

- Simplest topology to deploy, operate, and troubleshoot.
- Raft majority always available, surviving one node failure.
- Lowest write latency when deployed in a single data center (DC).
- Survives single AZ failure when deployed across multiple availability zones.

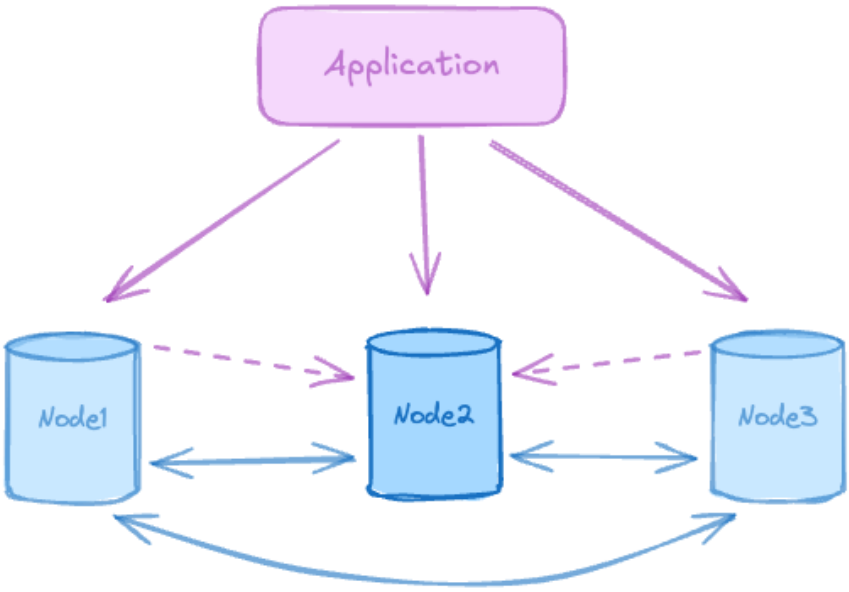


Fig. 1: Single data group topology

- Automatic failover in seconds with no operator intervention.
- Low cost at three nodes minimum.

There are some limitations to keep in mind:

- No disaster recovery. A DC-level failure takes down the entire cluster.
- When deployed in a single DC, all nodes share the same failure domain.
- Not suitable for regulatory requirements mandating geographic separation.
- When used with two data nodes and a witness, durability options are limited.

### Recommended commit scopes for this pattern

- **Majority protect**: synchronous replication to the majority of the nodes. The most commonly recommended scope for production use.
- **Quorum Commit**: coordinates the commit decision across all participating nodes before any node commits locally. Recommended for workloads requiring strict distributed consistency, such as payments and core banking. Witness nodes don't contribute to quorum, so all three nodes must be data nodes. Use `MAJORITY ORIGIN GROUP QUORUM COMMIT ABORT ON (timeout = 6s)`.
- **Adaptive protect**: synchronous replication to the majority of nodes when available, degrading automatically to asynchronous replication after a configurable timeout. Recommended when using two data nodes and one witness.
- **All protect**: synchronous replication to all nodes in the group. Provides the highest durability guarantee, but increases write latency and reduces availability tolerance when any node fails.
- **Local protect**: asynchronous replication with durability only for the local node. Not recommended for production use.

Commit scopes make different trade-offs across performance, durability, consistency, and scalability.

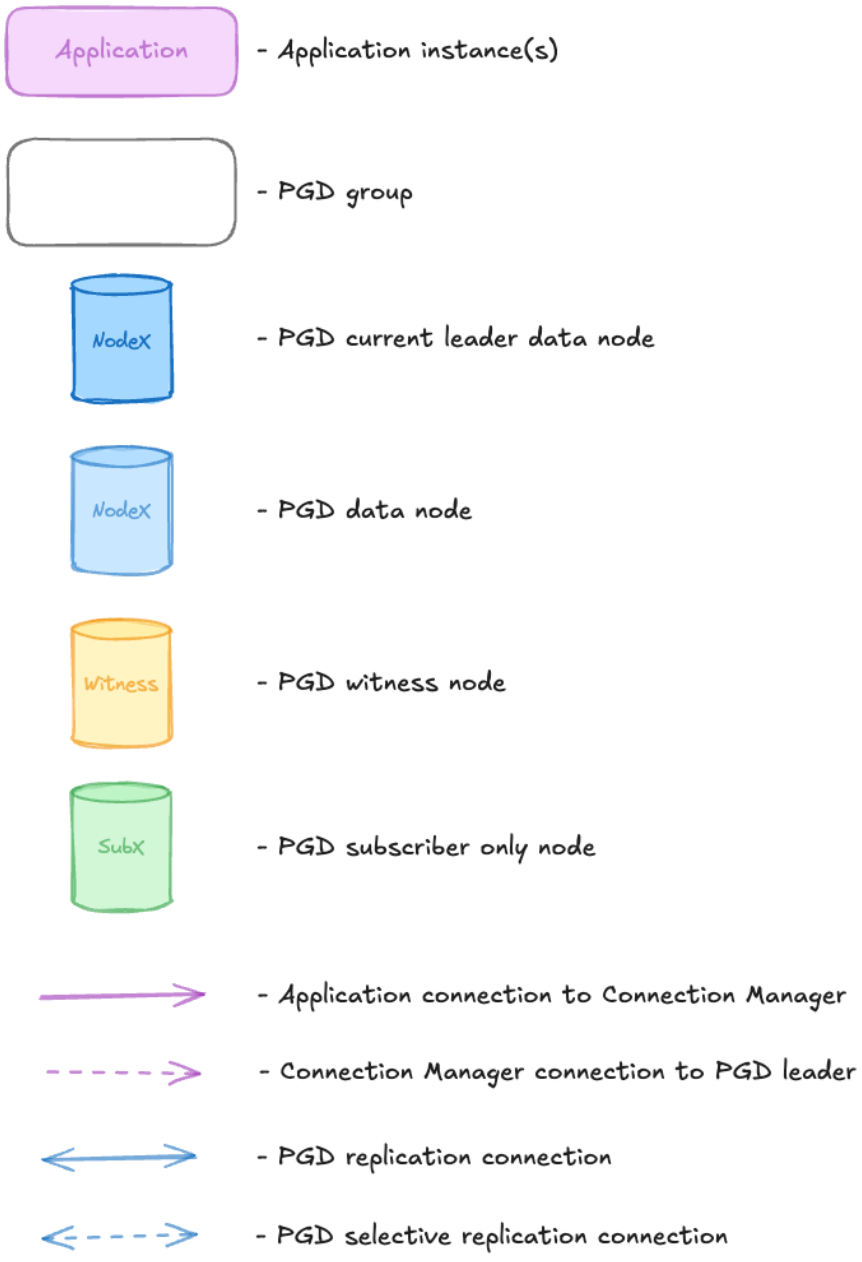


Fig. 2: Diagram legend showing node types, group containers, and connection types

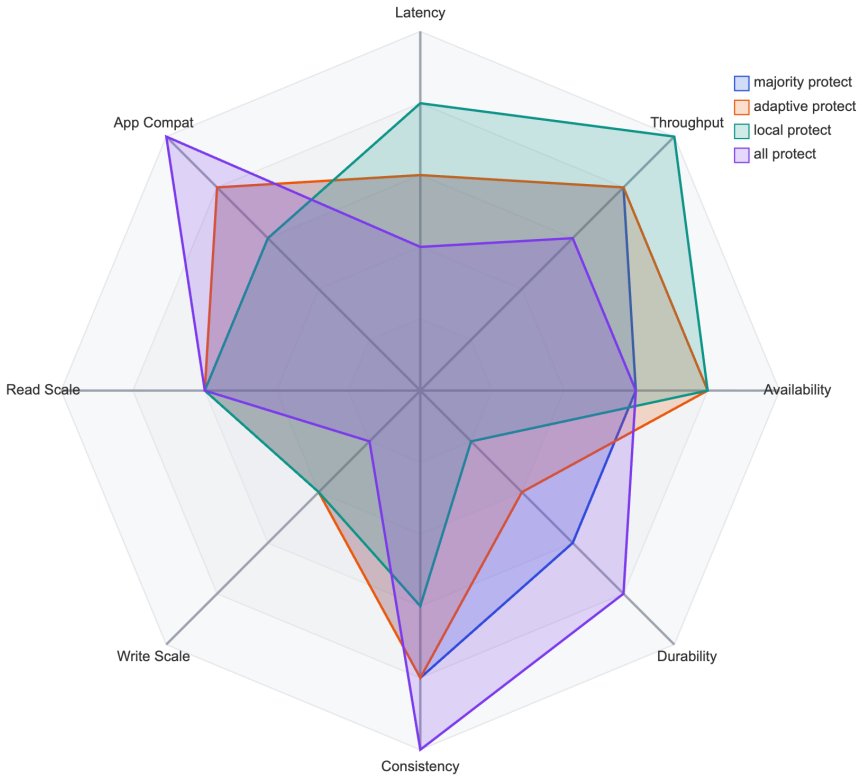


Fig. 3: Single data group behavior diagram

### Two data groups, active-active

The two data groups pattern spans two geographically separated data centers, each running its own independent Raft consensus group for local high availability (HA). With three or more data nodes per group, each group can use majority-based commit scopes within its own site.

Both data centers serve local reads and writes simultaneously, with replication between groups keeping both locations current.

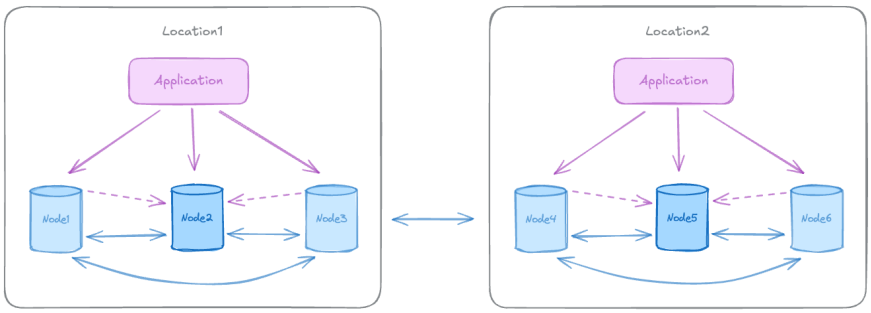


Fig. 4: Two data groups topology

### Diagram legend

Symbols represent node types, group containers, and connection types.

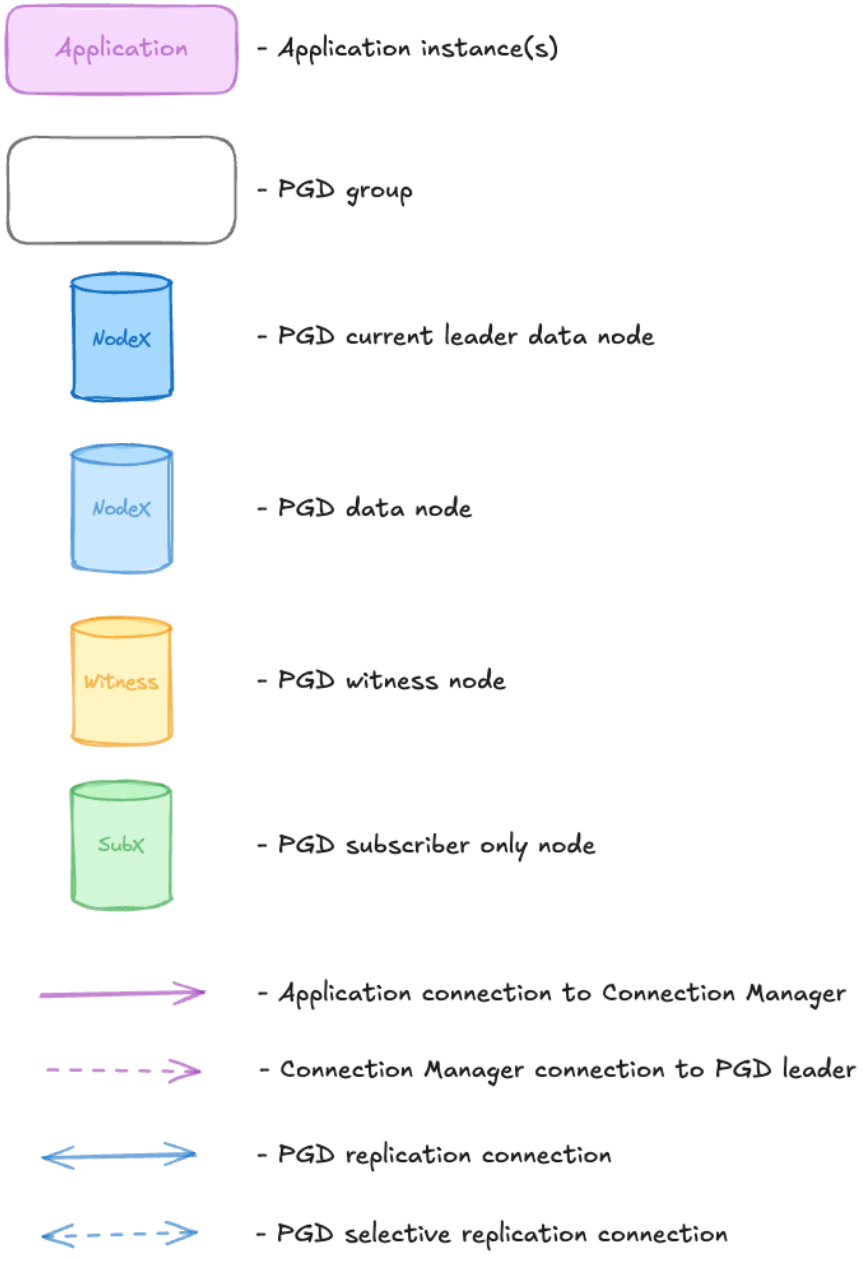


Fig. 5: Diagram legend showing node types, group containers, and connection types

### When to use this pattern

This pattern suits financial services, healthcare, and any application requiring geo-replication with zero data loss across two active locations. Both data centers serve writes simultaneously with conflict resolution, providing true active-active operation across locations.

The advantages of this pattern include the following:

- Provides true disaster recovery, surviving complete data center (DC) failure.
- Runs active-active, with both DCs serving reads and writes for local latency.
- Geographic redundancy for compliance requirements.
- With local routing, each DC operates independently for most operations.

There are some limitations to keep in mind:

- Conflicts are inevitable with optimistic cross-region consistency handling.
- Write latency and zero data loss on DC failure are in tension. Stronger commit scopes increase latency.
- Without Raft majority (on DC loss), global allocation sequences (galloc) eventually run out of allocated ranges and block.
- Global routing isn't available. The loss of one DC breaks Raft majority for the whole cluster.

The Raft majority and galloc limitations can be addressed by adding a witness location, as described below.

### Adding a witness location

Adding a third location, possibly in cloud, containing a single witness node resolves the Raft majority and galloc limitations. The witness participates in Raft consensus but doesn't store user data. With an odd-number quorum (seven voters: six data nodes and one witness), the cluster maintains Raft majority even when one DC is lost, restoring global routing and preventing galloc sequences from exhausting their allocated ranges.

A witness location is also required when using just two data nodes per group without a per-group witness, since that configuration can't achieve local HA on its own.

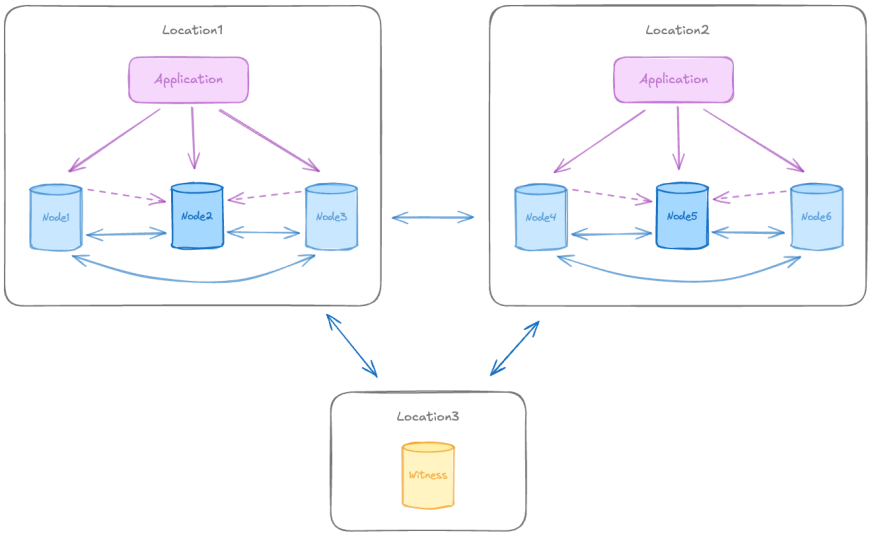


Fig. 6: Two data groups with witness topology

### Recommended commit scopes for this pattern

- **Majority protect**: synchronous replication to the majority of the nodes within the group. The most commonly recommended scope for production use.
- **Quorum Commit**: coordinates the commit decision across all participating nodes before any node commits locally, ensuring the transaction is durable across both regions before it commits. Recommended for workloads requiring strict distributed consistency, such as payments and core banking. Use `MAJORITY FOR EACH GROUP`

`IN CLUSTER QUORUM COMMIT ABORT ON (timeout = 6s)` , which guarantees that any new write leader has the latest committed data acknowledged by the prior leader.

- **CAMO**: when using two data nodes and one witness per group, CAMO adds exactly-once transaction guarantees over synchronous replication.
- **Local protect**: asynchronous replication with durability only for the local node. Not recommended for production use.

Commit scopes make different trade-offs across performance, durability, consistency, and scalability.

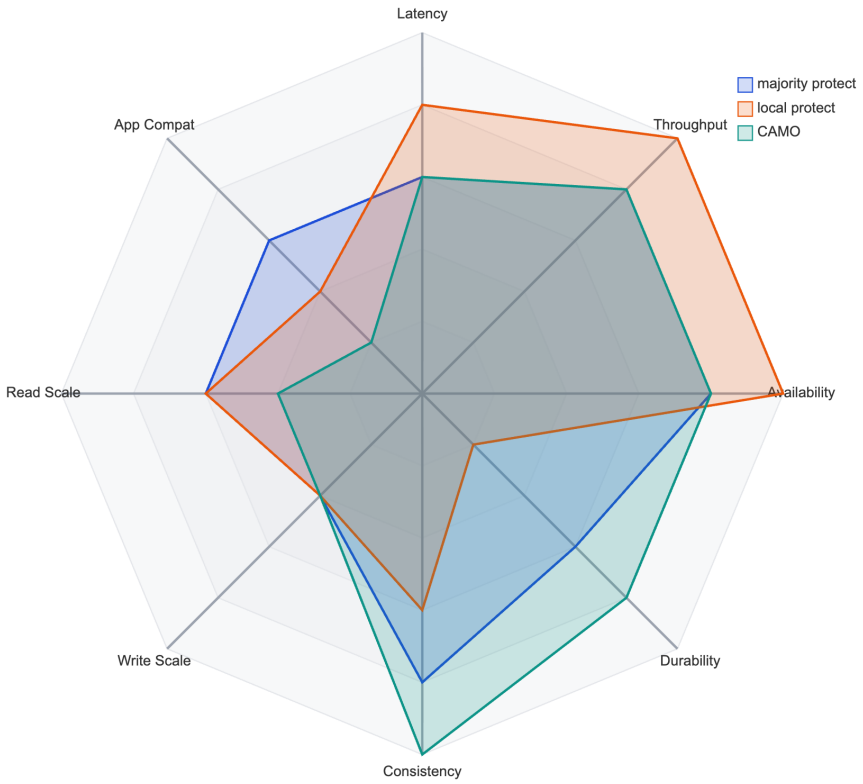


Fig. 7: Two data groups behavior diagram

### Three data groups, active-active-active

The three data groups pattern spans three or more geographic regions, each running its own independent Raft consensus group for local high availability (HA). With three or more data nodes per group, each group can use majority-based commit scopes within its own region, with Connection Manager handling write leader routing locally.

Applications connect to their local region for low-latency reads and writes. Inter-region replication is typically asynchronous to avoid cross-region latency penalties.

### Diagram legend

Symbols represent node types, group containers, and connection types.

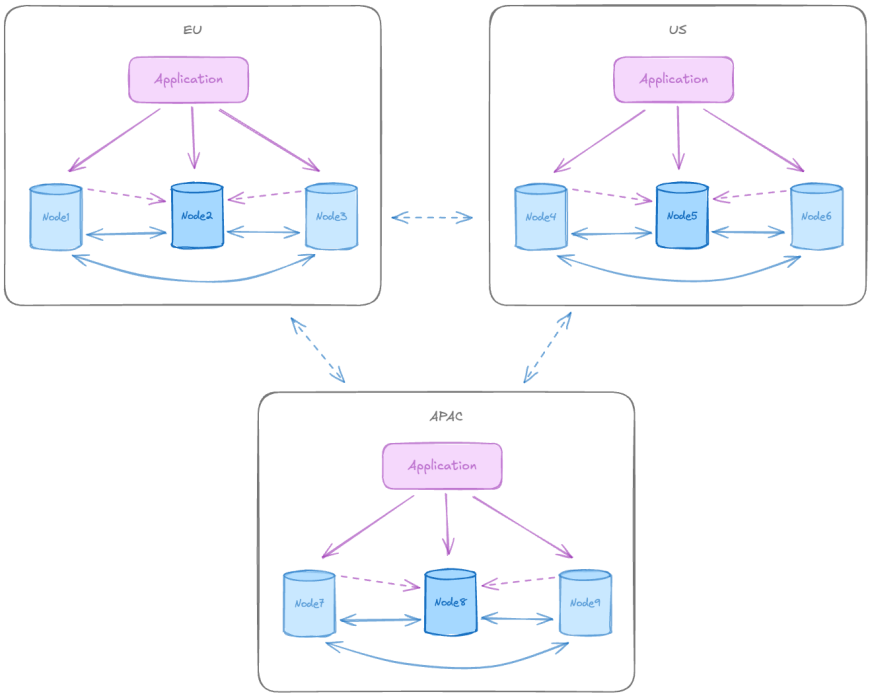


Fig. 8: Three data groups topology

### When to use this pattern

This pattern suits global SaaS applications, CDN-like data distribution, and multinational enterprises requiring geo-replication with data locality and eventual consistency across three or more regions.

The advantages of this pattern include the following:

- Enables true global presence with low-latency reads and writes in every region.
- Survives entire region failure while other regions continue operating.
- Natural Raft majority with three or more regions.
- Best end-user experience for globally distributed applications.

There are some limitations to keep in mind:

- Conflicts are inevitable with optimistic cross-region consistency handling.
- Higher operational complexity with more nodes, more locations, and more monitoring.
- Inter-region latency affects replication lag and conflict detection window.

### Adding global routing

Three or more data groups can also be used with global routing, where the whole cluster has a single write leader rather than one per group. A single global write leader eliminates write conflicts by routing all writes through one location, at the cost of increased write latency for regions not hosting the write leader. When the write leader changes location, latency characteristics shift for all regions.

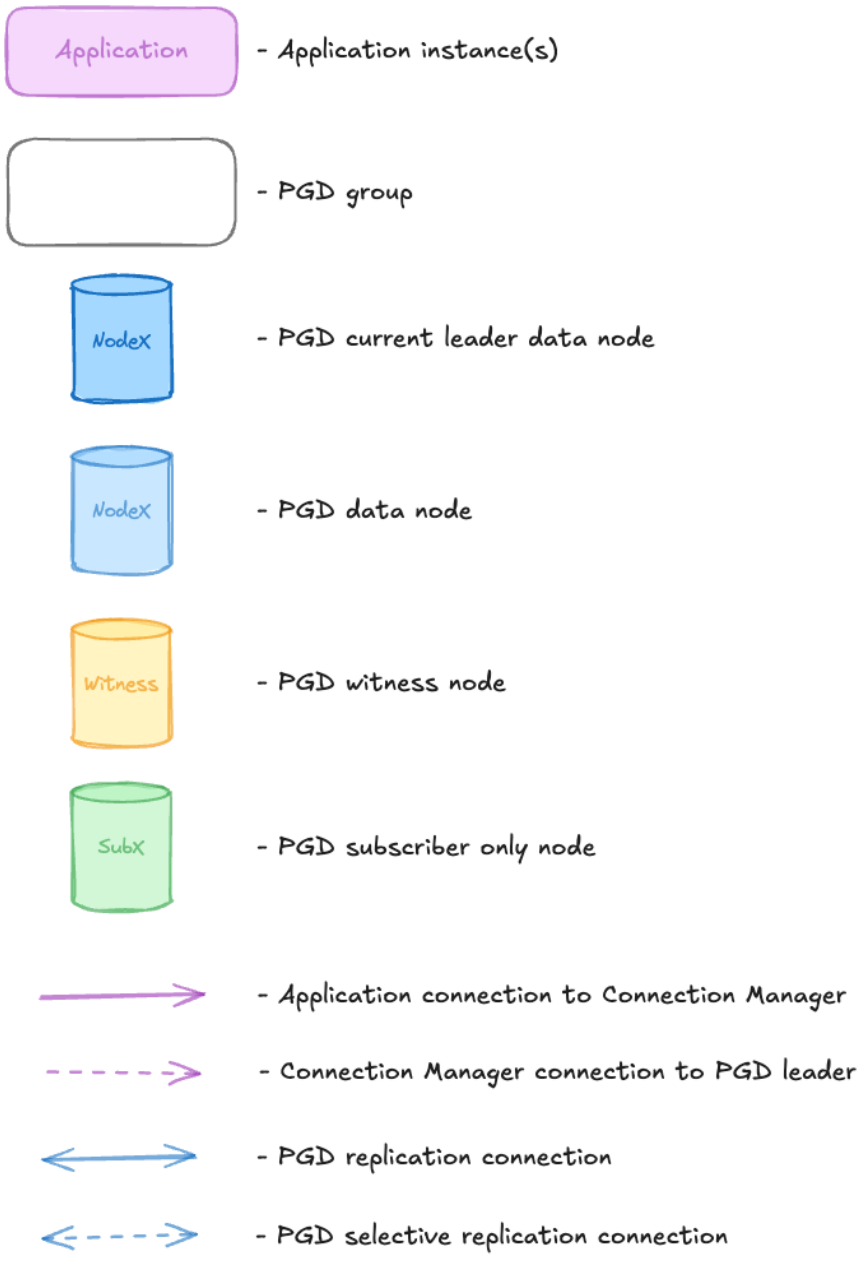


Fig. 9: Diagram legend showing node types, group containers, and connection types

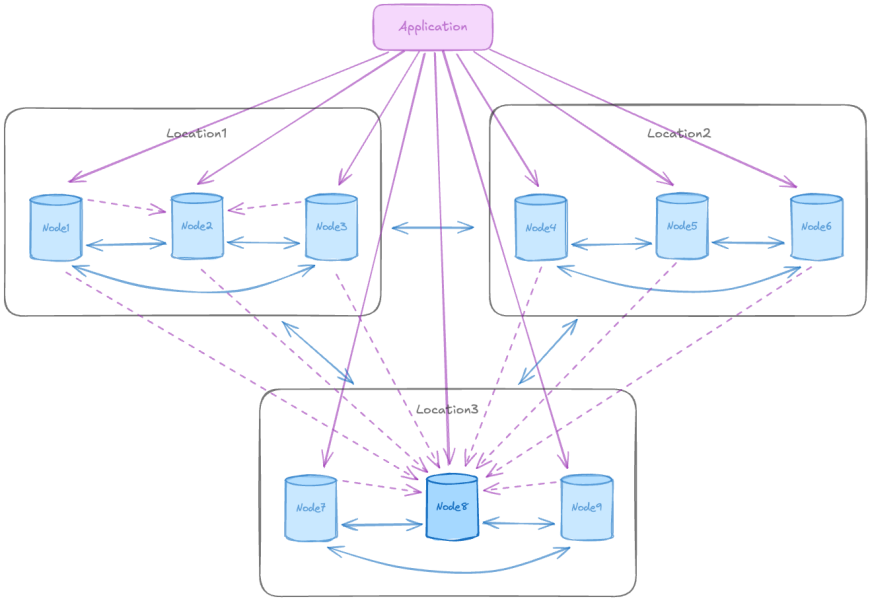


Fig. 10: Three data groups with global routing topology

### Recommended commit scopes for this pattern

- **Majority protect**: synchronous replication to the majority of the nodes within the group. The most commonly recommended scope for production use.
- **Quorum Commit**: coordinates the commit decision across all participating nodes before any node commits locally, ensuring the transaction is durable in all regions before it commits. Recommended for workloads requiring strict distributed consistency, such as payments and core banking. Use `MAJORITY FOR EACH GROUP IN CLUSTER QUORUM COMMIT ABORT ON (timeout = 6s)` , which guarantees that any new write leader, regardless of region, has the latest committed data acknowledged by the prior leader.
- **Adaptive protect**: synchronous replication to the majority of nodes within the group when available, degrading automatically to asynchronous replication after a configurable timeout. Recommended when using two data nodes and one witness per group.
- **Local protect**: asynchronous replication with durability only for the local node. Not recommended for production use.

Commit scopes make different trade-offs across performance, durability, consistency, and scalability.

### Single data group, global read scaling

The global read scaling pattern adds multiple subscriber-only groups to a central write group of three or more nodes. Connection Manager handles write leader routing within the write group automatically, while each subscriber-only group can have its own load balancer and scaling policy.

Subscriber-only nodes receive all changes but can't write or participate in consensus. Groups can be added and removed dynamically without affecting the write group, and the write group itself can be distributed across multiple regions or availability zones for higher availability.

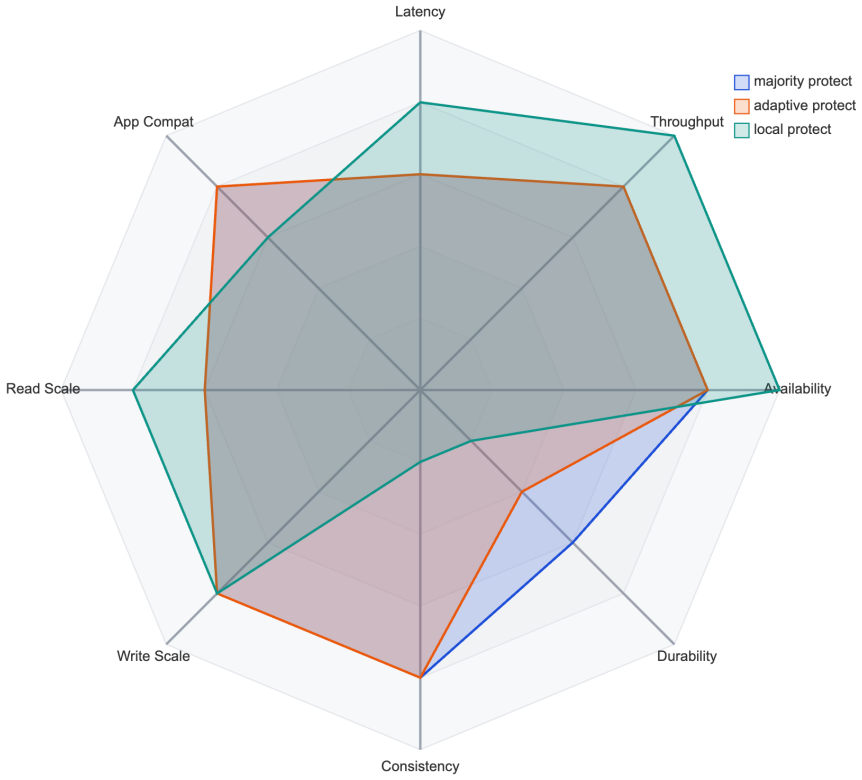


Fig. 11: Three data groups behavior diagram

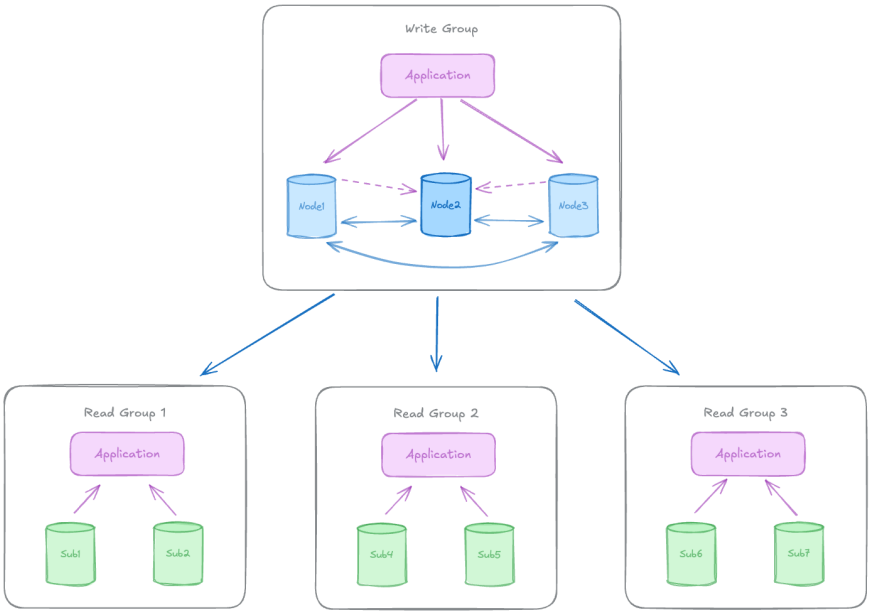


Fig. 12: Single data group global read scaling topology

### Diagram legend

Symbols represent node types, group containers, and connection types.

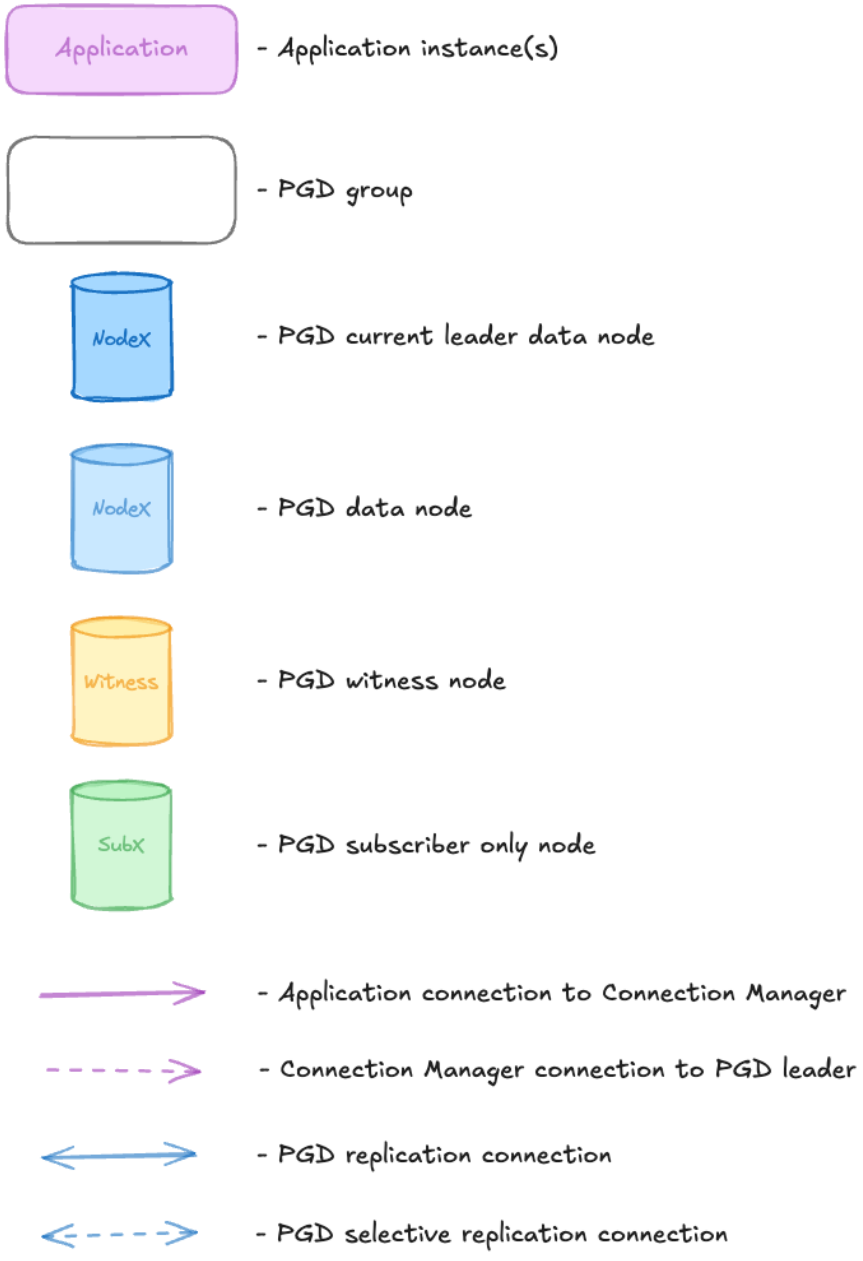


Fig. 13: Diagram legend showing node types, group containers, and connection types

### When to use this pattern

This pattern suits read-heavy workloads requiring workload isolation, with separate groups for analytics and reporting, API read traffic, configuration distribution, and regional read replicas. Each group scales independently based on its workload.

The advantages of this pattern include the following:

- Scales reads massively by adding nodes without affecting the write cluster.
- Offloads reporting and analytics from the production write path.
- Subscriber-only nodes can be placed in additional regions for local read latency.
- Supports dynamic scaling by adding and removing read nodes without cluster reconfiguration.
- Isolates heavy read workloads from transaction processing.

There are some limitations to keep in mind:

- Subscriber-only nodes have replication lag and aren't suitable for read-after-write consistency requirements.
- Applications must route connections explicitly, distinguishing between subscriber-only nodes and data nodes.
- Subscriber node lag or downtime increases WAL retention on data nodes, which affects disk usage.

### Recommended commit scopes for this pattern

- **Majority protect**: synchronous replication to the majority of the data nodes in the write group. The most commonly recommended scope for production use.
- **Adaptive protect**: synchronous replication to the majority of nodes in the write group when available, degrading automatically to asynchronous replication after a configurable timeout. Recommended when using two data nodes and one witness.
- **Local protect**: asynchronous replication with durability only for the local node. Not recommended for production use.

Commit scopes make different trade-offs across performance, durability, consistency, and scalability.

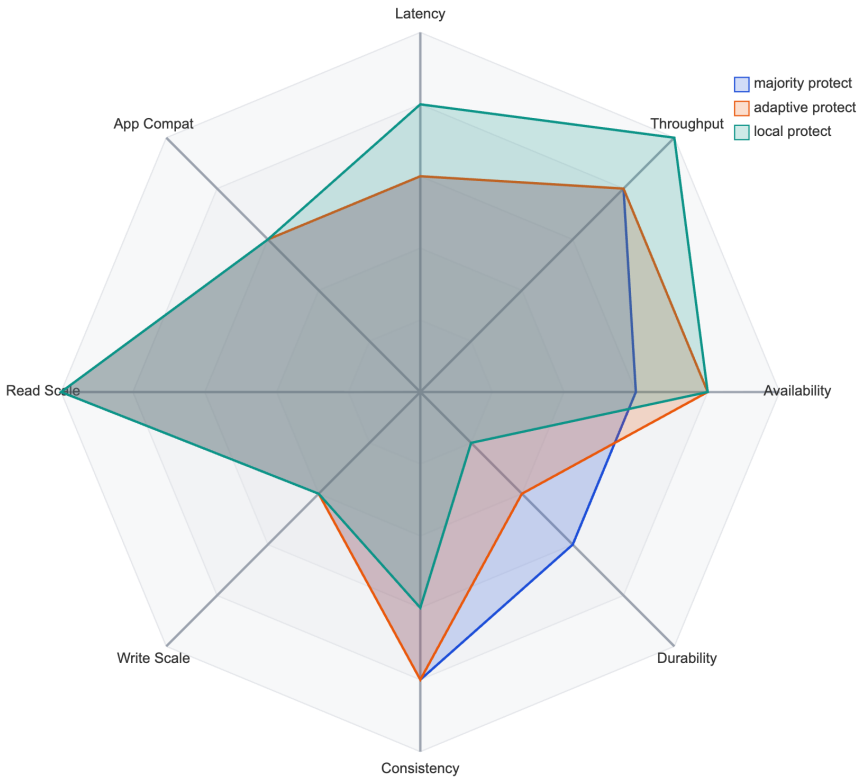


Fig. 14: Single data group global read scaling behavior diagram

### Primary active group, DR group

The primary active group, DR group pattern pairs a full-capacity primary site with two data nodes and a reduced-capacity disaster recovery (DR) site with one node. Connection Manager handles automatic failover between the two primary nodes, keeping the primary site available through single node failures without operator intervention.

The DR site receives asynchronous replication and remains on standby for failover. During DR failover, applications switch to the single DR node at reduced capacity until the primary is restored.

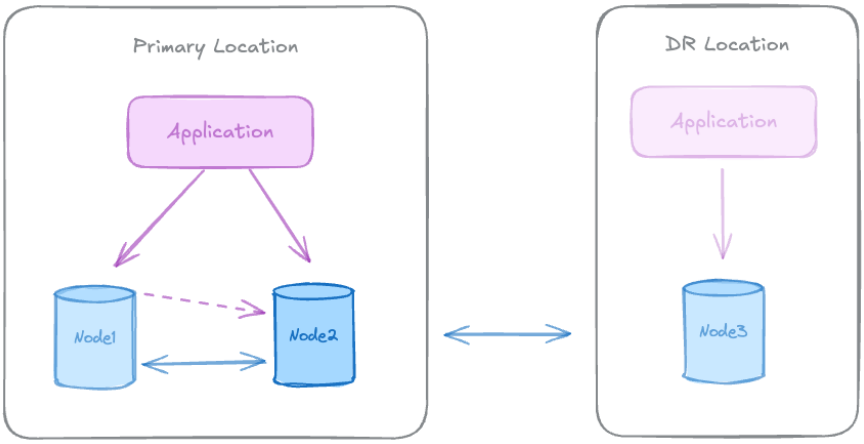


Fig. 15: Primary active group, DR group topology

The cluster transitions through four states depending on which nodes are available.

- Steady state: two nodes active in the primary, one node receiving replication in the DR location.
- Primary node failure: automatic failover to the second primary node in seconds.
- Primary data center (DC) failure: manual or orchestrated failover to the DR node in minutes.
- Recovery: rebuild the primary DC from the DR node, then fail back.

### Diagram legend

Symbols represent node types, group containers, and connection types.

### When to use this pattern

This pattern suits organizations needing disaster recovery with budget constraints for DR infrastructure. The primary site has full HA, and the DR site provides data protection and recovery capability at lower cost. It's common in deployments where full-capacity active-active isn't justified.

The advantages of this pattern include the following:

- Cheapest setup with DR, protecting data in a separate location.
- Full HA within the primary data center, surviving single node failure.
- DR node can be used for read-only reporting and analytics.
- Simpler than full multi-DC active-active.

There are some limitations to keep in mind:

- DR capacity is limited to a single node, with no HA in the DR site.

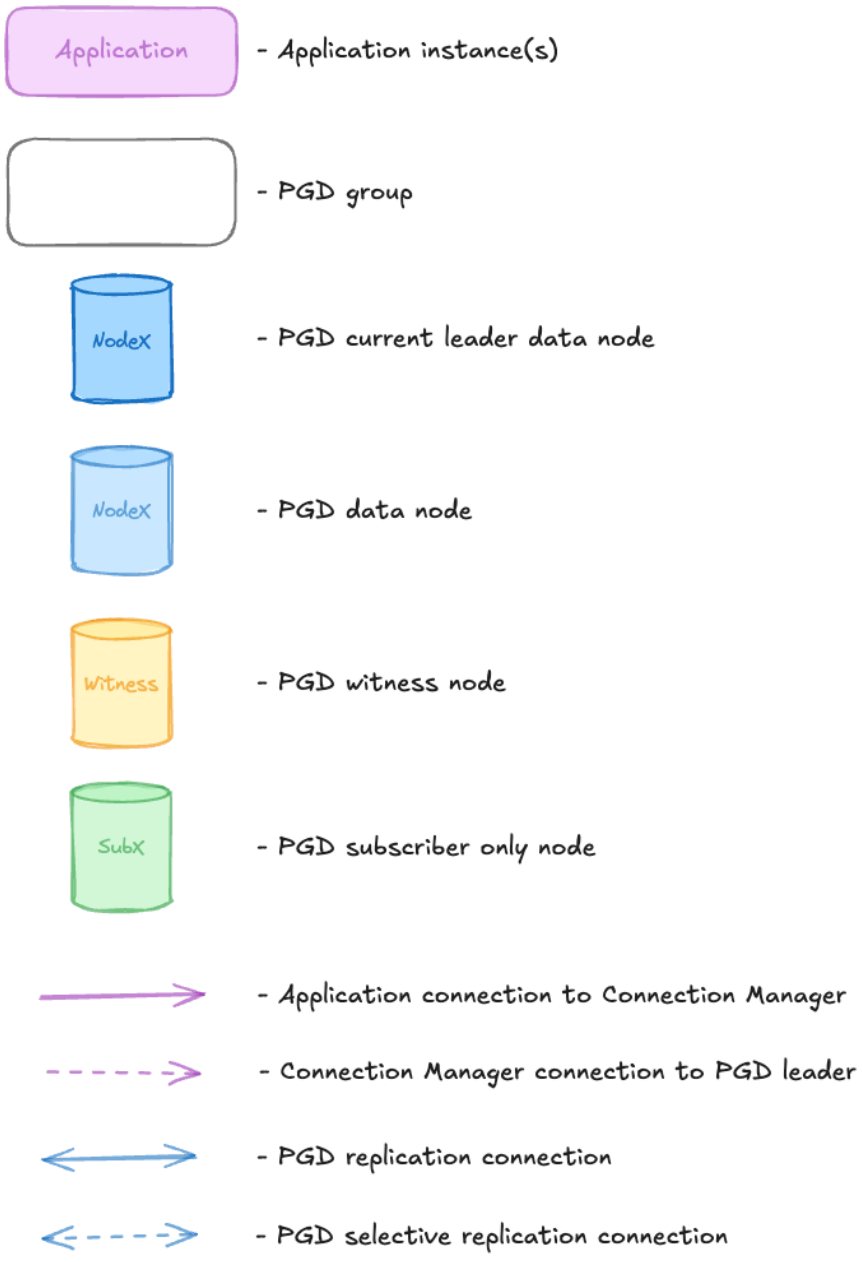


Fig. 16: Diagram legend showing node types, group containers, and connection types

- DR failover requires manual intervention or orchestration.
- Active-active workloads aren't supported.
- The primary group can have consistency issues on cascading failures or recovery, because replication degrades to asynchronous to preserve high availability.

### Recommended commit scopes for this pattern

- **Adaptive protect**: synchronous replication to both nodes in the primary group, degrading to asynchronous replication for availability on node loss. The most commonly recommended scope for production use.
- **Local protect**: asynchronous replication with durability only for the local node. Not recommended for production use.

Commit scopes make different trade-offs across performance, durability, consistency, and scalability.

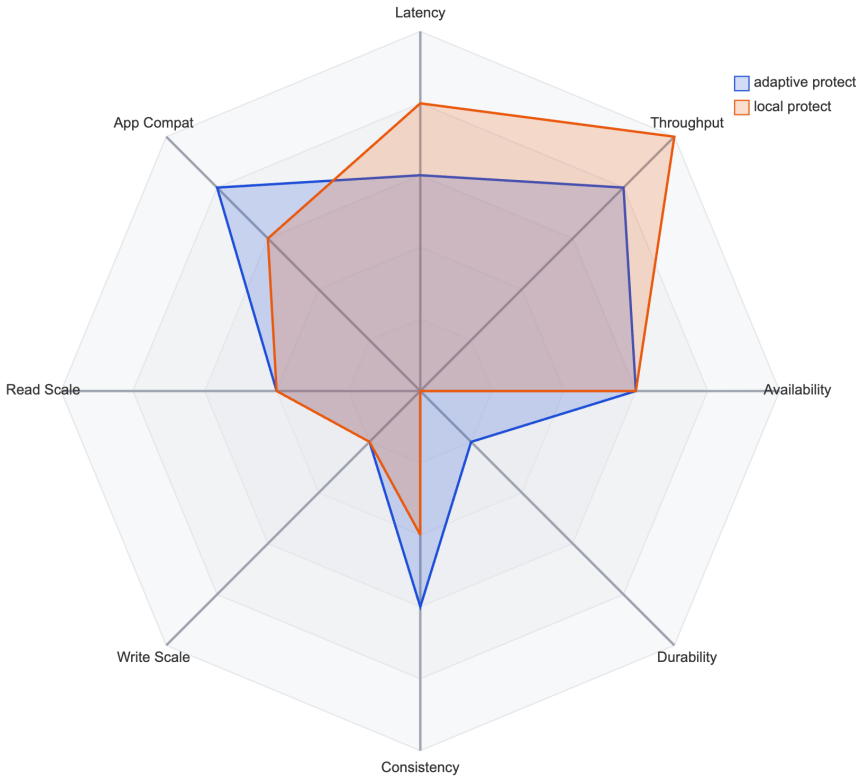


Fig. 17: Primary active group, DR group behavior diagram

### Multiple locations, data residency

The multiple locations, data residency pattern is a multi-region deployment where personal and sensitive data stays in its origin region and is never replicated cross-border. Each region runs its own independent Raft consensus group with two or more data nodes for local high availability (HA).

Only global reference data, including product catalogs, configuration, and anonymized analytics, replicates between regions, enabling compliance with GDPR, CCPA, and other data localization laws.

Selective replication works by classifying data into two categories.

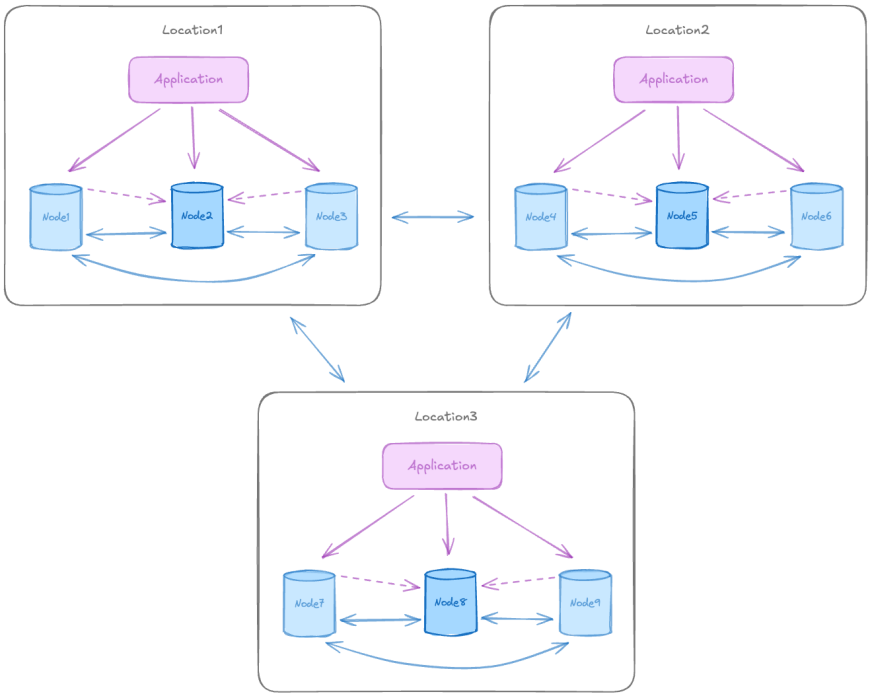


Fig. 18: Multiple locations, data residency topology

- **Local-only** (replicated only within the region): personally identifiable information (PII), customer data, financial records, and healthcare data.
- **Global** (replicated everywhere): product catalog, configuration, reference data, and anonymized analytics.

### Diagram legend

Symbols represent node types, group containers, and connection types.

### When to use this pattern

This pattern suits global enterprises that need geo-replication with strict regulatory requirements for data localization, including financial services where banking data must stay in-country, healthcare where patient data residency is required, government institutions, and multinational SaaS where customer data must stay in specific jurisdictions.

The advantages of this pattern include the following:

- Allows compliance with GDPR, CCPA, and data localization regulations.
- Avoids cross-border data transfer legal complexity.
- Provides low latency for regional users by keeping data in-region.
- Single global application codebase with regional data routing.
- Isolates regional failures so one region going down doesn't affect others.

There are some limitations to keep in mind:

- No cross-region view of all data.
- Schema design must separate local and global data clearly.
- Monitoring must track per-region compliance.

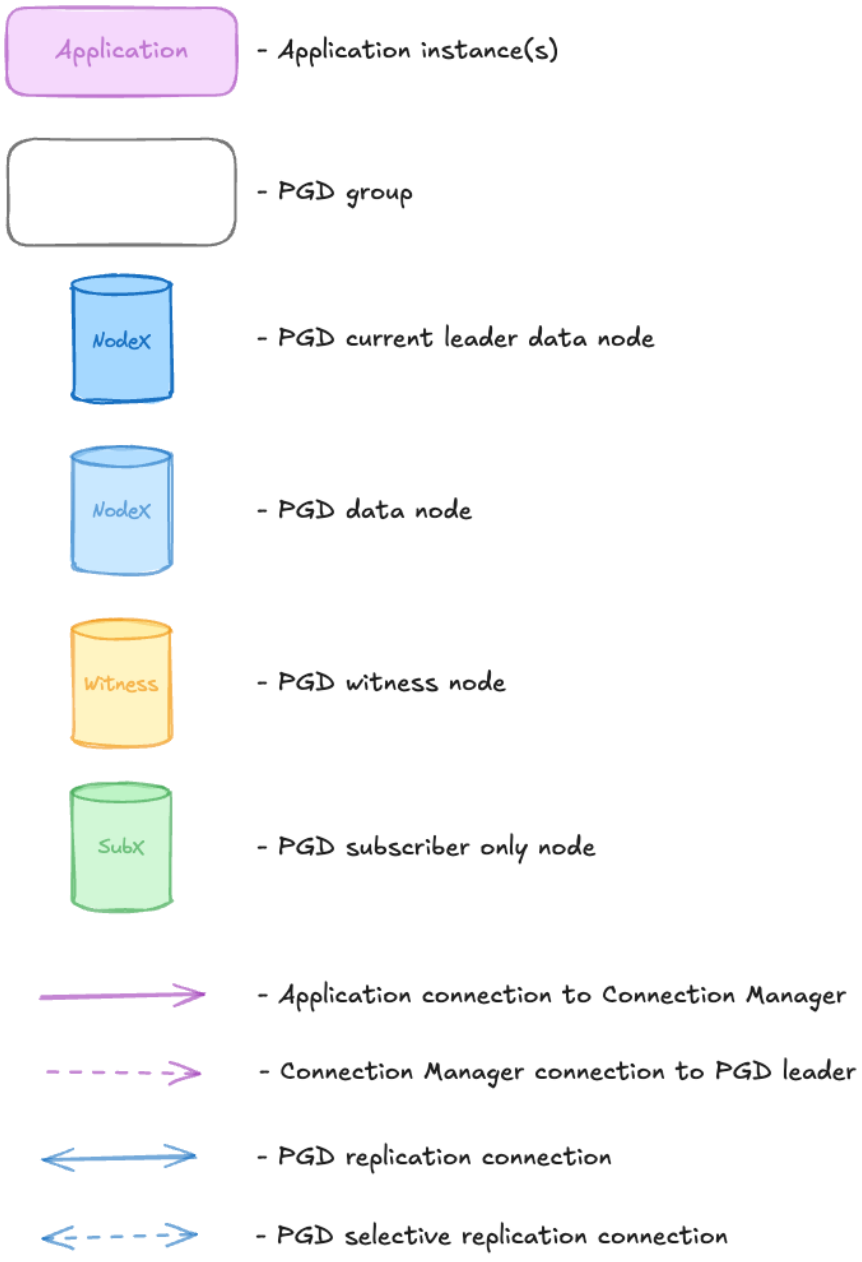


Fig. 19: Diagram legend showing node types, group containers, and connection types

### Recommended commit scopes for this pattern

- **Majority protect**: synchronous replication to the majority of the nodes in a region. The most commonly recommended scope for production use.
- **Adaptive protect**: synchronous replication to the majority of nodes when available, degrading automatically to asynchronous replication after a configurable timeout. Recommended when using two data nodes and one witness.
- **Local protect**: asynchronous replication with durability only for the local node. Not recommended for production use.

Commit scopes make different trade-offs across performance, durability, consistency, and scalability.

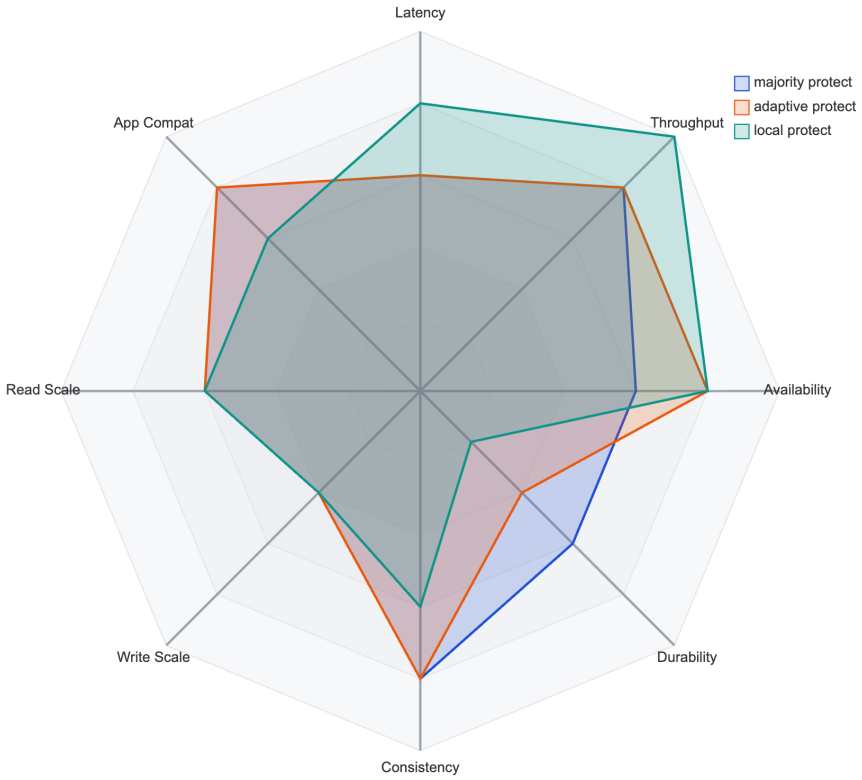


Fig. 20: Multiple locations, data residency behavior diagram

## 4.2 Choosing a Postgres distribution

EDB Postgres Distributed can be deployed with three different Postgres distributions: PostgreSQL, EDB Postgres Extended Server, or EDB Postgres Advanced Server. The availability of particular EDB Postgres Distributed features depends on the Postgres distribution being used. Therefore, it's essential to adopt the Postgres distribution best suited to your business needs. For example, if having the Commit At Most Once (CAMO) feature is mission critical to your use case, don't adopt open source PostgreSQL, which doesn't have the core capabilities required to handle CAMO.

The following table lists features of EDB Postgres Distributed that are dependent on the Postgres distribution and version.

| Feature | PostgreSQL | EDB Postgres Extended | EDB Postgres Advanced |
|---|---|---|---|
| *Replicated large objects* | Y | Y | Y |
| *Replicated large objects* | Y | Y | Y |
| *Statement-specific DDL replication concerns* | Y | Y | Y |
| *Replicated large objects* | Y | Y | Y |
| *Replicated large objects* | Y | Y | Y |
| *Replicated large objects* | Y | Y | Y |
| *Subscriber-only nodes* | Y | Y | Y |
| *Monitoring Parallel Apply* | Y | Y | Y |
| *Parallel Apply* | Y | Y | Y |
| Conflict-free replicated data types (CRDTs) | Y | Y | Y |
| *Special problems for column-level conflict resolution* | Y | Y | Y |
| *Transform triggers* | Y | Y | Y |
| *Conflict triggers* | Y | Y | Y |
| *Asynchronous replication* | Y | Y | Y |
| *Legacy synchronous replication using PGD* | Y | Y | Y |
| *SYNCHRONOUS COMMIT and GROUP COMMIT* | N | Y | Y |
| *Replicated large objects* | N | Y | Y |
| *Eager conflict resolution* | N | Y | Y |
| *LAG CONTROL* | N | Y | Y |
| *Decoding worker* | N | Y | Y |
| *Replicated large objects* | N | Y | Y |
| *Replicated large objects* | N | Y | Y |
| *Replicated large objects* | N | Y | Y |
| *Replicated large objects* | N | Y | Y |
| *Support for Transparent Data Encryption (TDE)* | N | 15+ | 15+ |

## 4.3 Choosing your deployment method

You can deploy and install EDB Postgres Distributed (PGD) products using the following methods:

- ```
  :ref:`Manual deployment and administration </pgd/current/lifecycle/deploying/
  ↪deploy-manual/>`  provides granular control by allowing you to manually␣
  ↪provision, install, and configure PGD directly on your chosen infrastructure.␣
  ↪This method involves manually setting up the underlying Postgres engine, and␣
  ↪configuring the node group settings. While this approach offers maximum␣
  ↪flexibility for bespoke environments, it requires a deep understanding of PGD's␣
  ↪configuration parameters and life-cycle management.
  ```

- ```
  `Trusted Postgres Architect <https://www.enterprisedb.com/docs/tpa/latest>`_ ␣
  ↪(TPA) is an orchestration tool that uses Ansible to build Postgres clusters␣
  ↪using a set of reference architectures that document how to set up and operate␣
  ```

(continues on next page)

(continued from previous page)

```
↪Postgres in various scenarios. TPA represents the best practices followed by␣
↪EDB, and its recommendations apply to quick testbed and production environments.
↪ TPA's flexibility allows deployments to virtual machines, AWS cloud instances,␣
↪or Linux host hardware.
```

- ```
  `EDB Postgres Distributed for Kubernetes <https://www.enterprisedb.com/docs/
  ↪postgres_distributed_for_kubernetes/latest/>`_  is a Kubernetes operator␣
  ↪designed, developed, and supported by EDB. It covers the full lifecycle of␣
  ↪highly available Postgres database clusters with a multi-master architecture,␣
  ↪using PGD replication. It's based on the open source CloudNativePG operator and␣
  ↪provides additional value, such as compatibility with Oracle using EDB Postgres␣
  ↪Advanced Server, Transparent Data Encryption (TDE) using EDB Postgres Extended␣
  ↪or Advanced Server, and additional supported platforms including IBM Power and␣
  ↪OpenShift.
  ```

- ```
  `Hybrid Manager <https://www.enterprisedb.com/docs/edb-postgres-ai/latest/hybrid-
  ↪manager/>`_  is a management component designed to orchestrate and monitor PGD␣
  ↪clusters across hybrid and multi-cloud environments. It simplifies the␣
  ↪complexity of managing PGD nodes that span different infrastructure providers,␣
  ↪providing a unified interface for lifecycle operations, health monitoring, and␣
  ↪configuration management. By bridging the gap between on-premises data centers␣
  ↪and public cloud deployments, Hybrid Manager ensures consistent operational␣
  ↪standards and high availability for geographically dispersed clusters.
  ```

## 4.4 Comparing replication solutions

Replication solutions differ significantly in how they handle multi-region deployments, conflict management, schema changes, and major version upgrades. Use this comparison to evaluate PGD against other options across those dimensions.

**Last updated: 2026-04-14**

| Product | Version |
|---|---|
| PGD Standalone | {Version 6.4} |
| PGD HM-managed | {Version 6.4} |
| PSR + EFM | 5.3 |

### 4.4.1 Deployment

| Product | On premis | Cloud | Multi-c | Hybrid (on-prem + cloud) | SLA | Native Kubernetes Operator |
|---|---|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ | ✓ | 99.999 | ✓ (PGD for Kubernetes (GA Q1 2025), based on CloudNativePG) |
| PGD HM-managed | ✓ | ✓ | ✓ | No | 99.995 | ✓ |

continues on next page

Table 4 – continued from previous page

| Product | On premis | Cloud | Multi-c | Hybrid (on-prem + cloud) | SLA | Native Kubernetes Operator |
|---|---|---|---|---|---|---|
| PSR + EFM | ✓ | ✓ | ✓ | ✓ | 99.99 | Partial (CloudNativePG, no native EFM integration) |

### 4.4.2 Performance

| Product | Read scalability | Horizontal scalability | Transaction streaming | Parallel apply (vertical scalability) | Logical Change Record (LCR) sharing |
|---|---|---|---|---|---|
| PGD Standalone | ✓ (subscriber only nodes) | No | ✓ | ✓ | ✓ (decodes WAL once, shares to all subscribers) |
| PGD HM-managed | No | No | ✓ | ✓ | ✓ |
| PSR + EFM | ✓ (physical standbys) | No | ✓ | No | No (decodes per subscriber) |

### 4.4.3 Durability

| Product | Asynchronous replication | Optional RPO limit for asynchronous replication | Synchronous replication | Consensus based replication | Per transaction durability setting |
|---|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ | ✓ | ✓ |
| PGD HM-managed | ✓ | ✓ | ✓ | ✓ | ✓ |
| PSR + EFM | ✓ | No | ✓ | No | ✓ |

### 4.4.4 Consistency

| Product | Automatic conflict management | Conflict avoidance types | Conflict avoidance at commit (pessimistic conflict handling) | Builtin distributed sequence | CRDT (Conflict-Free Replicated Data Types) |
|---|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ | ✓ (snowflake (bigint), galloc (int/bigint)) | ✓ (multiple nodes update same row simultaneously without global locking) |
| PGD HM-managed | ✓ | ✓ | ✓ | ✓ (snowflake (bigint), galloc (int/bigint)) | ✓ |
| PSR + EFM | N/A | N/A | N/A | N/A | No |

### 4.4.5 Data distribution

| Product | Data residency/selective replication | Cluster level Active-A (writers in different regions) | Regional Active-A (multiple writers within region) | Automatic partitioning | Offload cold data to cheaper storage |
|---|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ (not recommended unless specific setup) | ✓ | ✓ |
| PGD HM-managed | ✓ | ✓ | No | ✓ | ✓ |
| PSR + EFM | No | No | No | ✓ with EPAS | No |

### 4.4.6 Maintenance

| Product | Near-zero downtime major version upgrades by adding nodes | Near-zero downtime inplace major version upgrades | Rolling schema upgrades/green-blu (with application assistance) | Rolling maintenance operations |
|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ | ✓ |
| PGD HM-managed | ✓ | ✓ | ✓ | ✓ |
| PSR + EFM | ✓ (using logical replication) | No | No | No |

### 4.4.7 Connection Management

| Product | Automatic failover | Automatic connection failover for switchover | Cluster level connection routing | Region level connection routing | pgbouncer support | Built-in connection manager (no external consensus/proxy required) |
|---|---|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ (PGD 6.0+, eliminates HARP + etcd dependency) |
| PGD HM-managed | ✓ | ✓ | ✓ | ✓ | No | ✓ |
| PSR + EFM | ✓ | ✓ | ✓ | No | ✓ | No (requires pgbouncer / HAProxy) |

### 4.4.8 DDL support

| Product | General DDL replication | Granular (per-object) DDL locking | Create and drop objects | Add columns to table | Change column type | CREATE TABLE AS |
|---|---|---|---|---|---|---|
| PGD Standalone | ✓ | ✓ | ✓ | ✓ | ✓ (rewrite requires permit_unsafe_commands) | ✓ (with restrictions) |
| PGD HM-managed | ✓ | ✓ | ✓ | ✓ | ✓ (rewrite requires permit_unsafe_commands) | ✓ (with restrictions) |
| PSR + EFM | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

### 4.4.9 Postgres compatibility

| Product | Latest supported version | Works on standard PG | Supports TDE | Supports custom types (i.e. PostGIS) | Supports extensions | CDC failover support | Large Object support | Multiple DB support |
|---|---|---|---|---|---|---|---|---|
| PGD Standalone | 18 | ✓ | ✓ with EPAS/PGE | ✓ | Many | ✓ (PGD 5.7+) | No | No |
| PGD HM-managed | 18 | No | ✓ with EPAS/PGE | ✓ | Many | ✓ (PGD 5.7+) | No | No |
| PSR + EFM | 17 | ✓ | ✓ with EPAS/PGE | ✓ | All | ✓ | ✓ | ✓ |

### 4.4.10 Management

| Product | CLI | UI | Monitoring options |
|---|---|---|---|
| PGD Standalone | ✓ | ✓ (PEM) | Hybrid Manager, SQL, CLI, PEM |
| PGD HM-managed | ✓ | ✓ | Hybrid Manager, SQL, CLI, edbctl |
| PSR + EFM | ✓ | ✓ (PEM) | Hybrid Manager, SQL, CLI, PEM |

### 4.4.11 Licensing

| Product | Source available | Open source |
|---|---|---|
| PGD Standalone | No | No |
| PGD HM-managed | No | No |
| PSR + EFM | No | No |

## 4.5 Choosing between Physical Streaming Replication (PSR) and EDB Postgres Distributed (PGD)

EDB supports both high availability and distributed high availability. High availability is achieved through Physical Streaming Replication (PSR), which is typically managed by external tools like EDB Failover (EFM) or Patroni. Distributed high availability is achieved with EDB Postgres distributed (PGD).

Choosing the right replication strategy for Postgres depends on your application's specific availability, consistency, and geographic requirements. While PSR is effective for local high availability and standard DR scenarios, PGD is designed for "always-on" global architectures that require continuous write availability along with advanced consistency and durability guarantees.

### 4.5.1 PSR with EFM or Patroni

In a PSR setup, nodes are typically classified as either primary or standby. The primary serves as the central authority that handles all write operations and generates the Write-Ahead Log (WAL) stream. A standby acts as a resilient secondary node that remains in a continuous recovery state.

PSR works at the block level, creating a byte-for-byte copy of the primary on one or more standbys. Because PostgreSQL does not natively handle automatic failover, external orchestration tools are required.

#### PSR management tools

While there are many tools for managing PSR, the two highlighted here are supported by EDB because they are most widely deployed by EDB customers or the most extensively adopted open source solution.

- ** EDB Failover Manager (EFM): ** This enterprise-grade agent monitors cluster health, manages virtual IP (VIP) addresses, and automates the promotion of a standby to primary. This solution is ideal if you prefer native PSR, use bare metal and virtualized platforms where write operations are centralized on a single node. In this architecture, secondary regions typically serve as Disaster Recovery (DR) targets. Moving traffic between regions typically requires a formal, coordinated promotion process for both the database and application layer.
- ** Patroni: ** This is a popular open source orchestration tool for PSR with an active development community. Patroni relies on an external Distributed Consensus Store (DCS), most commonly `etcd` , to maintain leader election and cluster state. Like EFM, this solution is ideal if you already prefer PSR with an architecture that typically serves a primary and secondary region, requiring a coordinated manual or scripted process to shift operations between sites.

### 4.5.2 EDB Postgres Distributed (PGD)

PGD is an integrated extension stack that transforms PostgreSQL into a distributed, multi-master platform. It builds upon the **logical replication** features originally designed and contributed to the PostgreSQL core by the PGD architects, but adds the essential cluster-wide orchestration that the core feature lacks.

#### When to choose PGD

PGD is engineered for Tier 1 enterprise applications where downtime is not an option and data integrity must be guaranteed across a global footprint. Consider PGD if your requirements include:

- **Advanced durability and distributed consistency:** Unlike PSR, which offers limited and coarse-grained options for synchronicity, PGD provides true distributed consistency. Through commit scopes , you can precisely define the apply behavior for both local and remote nodes. This allows you to choose exactly how many nodes, and in which specific regions, must acknowledge a transaction before it is considered durable, allowing you to balance performance against data protection from a global down to a transaction level.
- **Always-on requirements (99.999%):** PGD offers zero-wait failover. Every node is writable; if one fails, the application simply reroutes to another active node without waiting for a promotion process. This eliminates the detection and promotion lag inherent in PSR, reducing your Recovery Time Objective (RTO) to the time it takes to reroute network traffic.
- **Zero-downtime operations:** PGD enables rolling upgrades and OS patches while the application remains online. Because the cluster is active-active, you can take nodes offline one by one for maintenance without losing write availability or requiring a complex switchover process. This architecture also allows you to perform heavy maintenance operations such as `REINDEX` or `VACUUM FULL` on a specific node without affecting the availability or performance of other nodes.
- **Global read and write scaling:** For workloads requiring massive read capacity, PGD supports subscriber only nodes. By placing writable nodes and high-scale subscribers physically close to your users within a mesh architecture, you eliminate cross-region latency and ensure snappy application response times regardless of geography. This approach allows you to scale reads while maintaining a high-performance write fabric across your primary regions.

Refer to *Comparing replication solutions* for a more detailed breakdown between PGD, PSR+EFM, pglogical2, and core logical replication in PostgreSQL.

To transition your current environment, follow the step-by-step guide Migrating from Postgres Physical Streaming Replication (PSR) to PGD .

## 4.6 Other considerations

There are a number of best practices to follow when deploying Postgres Distributed (PGD) in production. These practices help ensure the reliability, performance, and security of your PGD clusters. This section outlines some of the key best practices to consider when deploying PGD in a production environment.

- `:ref:`Sizing <Sizing>``
- `:ref:`Time and PGD <Time and PGD>``

### 4.6.1 Sizing

#### CPU/Core sizing

For production deployments, EDB recommends a minimum of 4 cores for each Postgres data node. Witness nodes don't participate in the data replication operation and don't have to meet this requirement. One core is enough without subgroup Raft. Two cores are enough when using subgroup Raft.

Always size logical standbys exactly like the data nodes to avoid performance degradations in case of a node promotion.

We recommend detailed benchmarking of your specific performance requirements to determine appropriate sizing based on your workload. The EDB Professional Services team is available to assist if needed.

For development purposes, don't assign Postgres data nodes fewer than two cores. The sizing of Barman nodes depends on the database size and the data change rate.

You can deploy Postgres data nodes and Barman nodes on virtual machines or in a bare metal deployment mode. However, don't deploy multiple data nodes on VMs that are on the same physical hardware, as that reduces resiliency.

### 4.6.2 Time and PGD

#### Clocks and timezones

EDB Postgres Distributed is designed to operate with nodes in multiple timezones, allowing a truly worldwide database cluster. Individual servers don't need to be configured with matching timezones, though we do recommend using `log_timezone = UTC` to ensure the human readable server log is more accessible and comparable.

Synchronize server clocks using NTP or other solutions.

Clock synchronization isn't critical to performance, as it is with some other solutions. Clock skew can affect origin conflict detection, though EDB Postgres Distributed provides controls to report and manage any skew that exists. EDB Postgres Distributed also provides row-version conflict detection, as described in *bdr_read_all_conflicts* .

CHAPTER 5

# Managing lifecycle operations

EDB Postgres Distributed (PGD) provides a suite of tools and methodologies for maintaining the health and evolution of your multi-master clusters. Managing lifecycle operations effectively ensures that your high-availability environment remains performant and secure through software updates, migrations, and configuration tuning.

- *Deploying and configuring PGD on Hybrid Manager* _ Instructions for provisioning PGD clusters across different environments, including manual installations, TPA-automated deployments, CloudNativePG deployments, and Hybrid Manager integrations.
- *Upgrading PGD* _ Detailed workflows for performing minor and major version upgrades for both the underlying PostgreSQL engine and the PGD extension with minimal downtime.
- *Testing and tuning PGD clusters* _ Guidance on benchmarking your distributed workloads using `pgd-bench` and optimizing system parameters for maximum throughput.
- *Backup and recovery* _ Best practices for securing your data, including physical and logical backup strategies tailored for distributed multi-master topologies.

## 5.1 Deploying and configuring EDB Postgres Distributed

This section covers how to deploy EDB Postgres Distributed and how to configure it.

There are four main ways to deploy PGD:

- Manual deployment and administration describes how to manually deploy and configure EDB Postgres Distributed on a set of servers.
- *Deploying and configuring PGD using TPA* describes how to use TPA to deploy and configure EDB Postgres Distributed to a Docker environment, Linux hosts, or AWS.
- *Deploying and configuring PGD on Kubernetes* describes how to deploy and configure EDB Postgres Distributed to a Kubernetes environment.
- *Deploying and configuring PGD on Hybrid Manager* describes how to deploy and configure EDB Postgres Distributed to Hybrid Manager.

### 5.1.1 Manually

This section covers how to manually deploy and configure EDB Postgres Distributed (PGD).

If you want to install a learning/test environment, we recommend using the Quickstart guide .

Manually deploying a PGD cluster consists of the following steps:

1. Meet the *Prerequisites* .
2. *Configuring PGD repositories* .
3. *Installing the database and pgd* .
4. *Configuring the cluster* .
5. *Checking the cluster* .
6. Learn how to *Connecting to your PGD cluster* .

#### Prerequisites

#### Provisioning hosts

The first step in the process of deploying PGD is to provision and configure hosts.

You can deploy to virtual machine instances in the cloud with Linux installed, on-premises virtual machines with Linux installed, or on-premises physical hardware, also with Linux installed.

Whichever supported Linux operating system and whichever deployment platform you select, the result of provisioning a machine must be a Linux system that you can access using SSH with a user that has superuser, administrator, or sudo privileges.

Each machine provisioned must be able to make connections to any other machine you're provisioning for your cluster.

On cloud deployments, you can do this over the public network or over a VPC.

On-premises deployments must be able to connect over the local network.

| Vendor | Platform | Guide |
|---|---|---|
| Amazon | AWS | Tutorial: Get started with Amazon EC2 Linux instances |
| Microsoft | Azure | *Replicated large objects* |
| Google | GCP | *Replicated large objects* |

#### Create an admin user

We recommend that you configure an admin user for each provisioned instance. The admin user must have superuser or `sudo` (to superuser) privileges. We also recommend that the admin user be configured for passwordless SSH access using certificates.

#### Ensure networking connectivity

With the admin user created, ensure that each machine can communicate with the other machines you're provisioning.

In particular, the PostgreSQL TCP/IP port (5444 for EDB Postgres Advanced Server, 5432 for EDB Postgres Extended and community PostgreSQL) must be open to all machines in the cluster. The PGD Connection Manager must also be accessible to all nodes in the cluster. By default, the Connection Manager uses port 6432 (or 6444 for EDB Postgres Advanced Server).

### Worked example

For this series of worked examples, three hosts with Red Hat Enterprise Linux 9 were provisioned:

- host-1
- host-2
- host-3

These hosts were configured in the cloud. As such, each host has both a public and private IP address. We will use the private IP addresses for the cluster.

The private IP addresses are:

- host-1: `192.168.254.166`
- host-2: `192.168.254.247`
- host-3: `192.168.254.135`

For the example cluster, `/etc/hosts` was also edited to use those private IP addresses:

```
192. 168.254.166 host-1
192. 168.254.247 host-2
192. 168.254.135 host-3
```

### Configuring PGD repositories

On each host which you want to use as a PGD data node, you need to install the database and the PGD software.

### Configure the PGD repositories

Set the following environment variables:

#### EDB_SUBSCRIPTION_TOKEN

This is the token you received when you registered for the EDB subscription. It is used to authenticate your access to the EDB repository.

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
```

#### EDB_SUBSCRIPTION_PLAN

This is the type of subscription you have with EDB. It can be `standard` , `enterprise` , or `community` .

```
export EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
```

#### EDB_REPO_TYPE

This is the type of package manager you use, which informs the installer which type of package you need. This can be `deb` for Ubuntu/Debian or `rpm` for CentOS/RHEL.

```
export EDB_REPO_TYPE=<your-repo-type>
```

### Install the repositories

There are two repositories you need to configure: one for the database software and one for the PGD software.

The following command will download and run a script that configures your package manager to use the EDB repository for databases.

```
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↪SUBSCRIPTION_PLAN/setup.$EDB_REPO_TYPE.sh" | sudo -E bash
```

The following command will download and run a script that configures your package manager to use the EDB repository for PGD.

```
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_
↪distributed/setup.$EDB_REPO_TYPE.sh" | sudo -E bash
```

### Worked example

In this example, we will configure the repositories on a CentOS/RHEL system that will allow us to install EDB Postgres Advanced Server 17 with PGD Expanded using an enterprise subscription.

### Set the environment variables

```
export EDB_SUBSCRIPTION_TOKEN=XXXXXXXXXXXXXX
export EDB_SUBSCRIPTION_PLAN=enterprise
export EDB_REPO_TYPE=rpm
```

### Install the repositories

```
#  For PGD Expanded, there are two repositories to install.

curl -1sSLf " https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↪SUBSCRIPTION_PLAN/setup.$EDB_REPO_TYPE.sh" | sudo -E bash
curl -1sSLf " https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_
↪distributed/setup.$EDB_REPO_TYPE.sh" | sudo -E bash
```

The next step is to *Installing the database and pgd* .

### Installing the database and pgd

On each host which you want to use as a PGD data node, you need to install the database and the PGD software.

After you have *Configuring PGD repositories* , you can install the database and PGD software using your package manager.

### Install the database and PGD software

### Set the Postgres version

Set an environment variable to specify the version of Postgres you want to install. This is typically `17` for Postgres 17.

```
export PG_VERSION=17
```

### Set the package names

Set an environment variable to specify the package names for the database and PGD software. The package names will vary depending on the database you are using and the platform you are on.

#### EDB Postgres Advanced Server

```
export EDB_PACKAGES="edb-as$PG_VERSION-server edb-pgd6-<edition>-epas$PG_VERSION"
```

Where `<edition>` is the PGD edition you want to install: `essential` or `expanded`.


```
export EDB_PACKAGES="edb-as$PG_VERSION-server edb-pgd6-<edition>-epas$PG_VERSION"
```

Where `<edition>` is the PGD edition you want to install: `essential` or `expanded`.


#### EDB Postgres Extended

```
export EDB_PACKAGES="edb-postgresextended-$PG_VERSION edb-pgd6-<edition>-pgextended
↪$PG_VERSION"
```

Where `<edition>` is the PGD edition you want to install: `essential` or `expanded`.

```
export EDB_PACKAGES="edb-postgresextended$PG_VERSION-server edb-postgresextended$PG_
↪VERSION-contrib edb-pgd6-<edition>-pgextended$PG_VERSION"
```

Where `<edition>` is the PGD edition you want to install: `essential` or `expanded`.

#### Community PostgreSQL

```
export EDB_PACKAGES="postgresql-$PG_VERSION edb-pgd6-<edition>-pg$PG_VERSION"
```

Where `<edition>` is the PGD edition you want to install: `essential` or `expanded`.

```
export EDB_PACKAGES="postgresql$PG_VERSION-server postgresql$PG_VERSION-contrib edb-
↪pgd6-<edition>-pg$PG_VERSION"
```

Where `<edition>` is the PGD edition you want to install: `essential` or `expanded`.

### Run the installation command

Run the installation command appropriate for your platform.

```
sudo apt install -y $EDB_PACKAGES
```

```
sudo dnf install -y $EDB_PACKAGES
```

This command will install the specified packages and any dependencies they require. Once the installation is complete, you will have the database and PGD software installed on your system.

### Worked example

In this example, we will install EDB Postgres Extended Server 17 with PGD Expanded on a CentOS/RHEL system using the repository configuration we set up in the *Worked example* .

```
export PG_VERSION=17
export EDB_PACKAGES="edb-as$PG_VERSION edb-pgd6-expanded-epas$PG_VERSION"
sudo dnf install -y $EDB_PACKAGES
```

### Configuring the cluster

### Configuring the cluster

The next step in the process is to configure the database and the cluster.

This involves logging into each host and running the `pgd` command to create the cluster as the database user.

These steps will vary according to which platform you are using and which version of Postgres you are using.

### Cluster name

You will need to choose a name for your cluster. This is the name that will be used to identify the cluster in the PGD CLI and in the database. It will be referred to as `<cluster-name>` in the examples. If not specified, the default name is `pgd` .

### Group names

You will also need to choose a name for the group. This is the name that will be used to identify the group in the PGD CLI and in the database. It will be referred to as `<group-name>` in the examples.

The group name must be unique within the cluster.

### Node names

You will also need to choose a name for each node. This is the name that will be used to identify the node in the PGD CLI and in the database. It will be referred to as `<node-name>` in the examples. This is separate from the host name, which is the name of the machine on which the node is running.

The node name must be unique within the group and within the cluster.

### Paths and users

The paths and users used in the examples will vary according to which version of Postgres you are using and which platform you are using.

Select your Postgres version:

Then select your platform:

| |
|---|
| *enterprisedb* |
| *5444* |
| */usr/lib/edb-as/$PG_VERSION/bin/* |
| */var/lib/edb-as/$PG_VERSION/main/* |

```
sudo -iu enterprisedb
export PG_VERSION=<version>
export PATH=$PATH:/usr/lib/edb-as/$PG_VERSION/bin/
export PGDATA=/var/lib/edb-as/$PG_VERSION/main/
export PGPORT=5444
```

| |
|---|
| *enterprisedb* |
| *5444* |
| */usr/edb/as$PG_VERSION/bin/* |
| */var/lib/edb/as$PG_VERSION/data/* |

```
sudo -iu enterprisedb
export PG_VERSION=<version>
export PATH=$PATH:/usr/edb/as$PG_VERSION/bin/
export PGDATA=/var/lib/edb/as$PG_VERSION/data/
export PGPORT=5444
```

Then select your platform:

| |
|---|
| *postgres* |
| *5432* |
| */usr/lib/edb-pge/$PG_VERSION/bin/* |
| */var/lib/edb-pge/$PG_VERSION/main/* |

```
sudo -iu postgres
export PG_VERSION=<version>
export PATH=$PATH:/usr/lib/edb-pge/$PG_VERSION/bin/
export PGDATA=/var/lib/edb-pge/$PG_VERSION/main/
export PGPORT=5432
```

| |
|---|
| *postgres* |
| *5432* |
| */usr/edb/pge$PG_VERSION/bin/* |
| */var/lib/edb-pge/$PG_VERSION/data/* |

```
sudo -iu postgres
export PG_VERSION=<version>
export PATH=$PATH:/usr/edb/pge$PG_VERSION/bin/
export PGDATA=/var/lib/edb-pge/$PG_VERSION/data/
export PGPORT=5432
```

Then select your platform:

| |
|---|
| *postgres* |
| *5432* |
| */usr/lib/postgresql/$PG_VERSION/bin/* |
| */var/lib/postgresql/$PG_VERSION/main/* |

```
sudo -iu postgres
export PG_VERSION=<version>
export PATH=$PATH:/usr/lib/postgresql/$PG_VERSION/bin/
export PGDATA=/var/lib/postgresql/$PG_VERSION/main/
export PGPORT=5432
```

| |
|---|
| *postgres* |
| *5432* |
| */usr/pgsql-$PG_VERSION/bin/* |
| */var/lib/pgsql/$PG_VERSION/data/* |

```
sudo -iu postgres
export PG_VERSION=<version>
export PATH=$PATH:/usr/pgsql-$PG_VERSION/bin/
export PGDATA=/var/lib/pgsql/$PG_VERSION/data/
export PGPORT=5432
```

### On each host

Run the commands from the script/settings above to set the environment variables and paths for the Postgres user on each host. This will ensure that the `pgd` command can find the Postgres executable files and data directory.

1. Using the appropriate user, log in as the database user.

```
sudo -iu <db-user>
```

1. Set the Postgres version environment variable. Don't forget to replace `<version>` with the actual version number you are using, such as `17` .

```
export PG_VERSION=<version>
```

1. Add the Postgres executable files to your path.

```
export PATH=$PATH:<executable-path>
```

1. Set the Postgres data directory environment variable.

```
export PGDATA=<data-directory>
```

1. Set the Postgres password environment variable. Don't forget to replace `<db-password>` with the actual password you want for the database user.

```
export PGPASSWORD=<db-password>
```

### On the first host

The first host in the cluster is also the first node and will be where we begin the cluster creation. On the first host, run the following command to create the cluster:

```
pgd node <first-node-name> setup --dsn "host=<first-host> user=<db-user> port=<db-
↪port> dbname=<dbname>" --group-name <group-name>
```

This command will create the data directory and initialize the database, then will create the cluster and the group on the first node.

### On the second host

On the second host, run the following command to create the cluster:

```
pgd node <second-node-name> setup --dsn "host=<second-host> user=<db-user> port=<db-
↪port> dbname=<db-name>" --cluster-dsn "host=<first-host> user=<db-user> port=<db-
↪port> dbname=<db-name>"
```

This command will create the node on the second host, and then join the cluster using the cluster-dsn setting to connect to the first host.

### On the third host

On the third host, run the following command to create the cluster:

```
pgd node <third-node-name> setup --dsn "host=<third-host> user=<db-user> port=<db-
↪port> dbname=<db-name>" --cluster-dsn "host=<first-host> user=<db-user> port=<db-
↪port> dbname=<db-name>"
```

This command will create the node on the third host, and then join the cluster using the cluster-dsn setting to connect to the first host.

### Worked example

In this example, we will configure the PGD Essential cluster with EDB Postgres Extended Server 17 on a CentOS/RHEL system that we *Configuring PGD repositories* and *Installing the database and pgd* in the previous steps.

We will now create a cluster called `pgd` with three nodes called `node-1` , `node-2` , and `node-3` .

- The group name will be `group-1` . The hosts are `host-1` , `host-2` , and `host-3` .
- The Postgres version is 17.
- The database user is `postgres` .
- The database port is 5432.
- The database name is `pgddb` .
- The Postgres executable files are in `/usr/edb/pge17/bin/` .
- The Postgres data directory is in `/var/lib/edb-pge/17/main/` .
- The Postgres password is `secret` .

(Note that we assume the Postgres version environment variable PG_VERSION is set to `17` from the previous step, and that we are preserving the environment variable when switching users.)

### On the first host

```
sudo  -iu postgres
export PG_VERSION=17
export PATH=$PATH:/usr/edb/pge$PG_VERSION/bin/
export PGDATA=/var/lib/edb-pge/$PG_VERSION/data/
export PGPASSWORD=secret
pgd node node-1 setup --dsn "host=host-1 user=postgres port=5432 dbname=pgddb" --
↪group-name group-1
```

### On the second host

```
sudo -iu postgres
export PG_VERSION=17
export PATH=$PATH:/usr/edb/pge$PG_VERSION/bin/
export PGDATA=/var/lib/edb-pge/$PG_VERSION/data/
export PGPASSWORD=secret
pgd node node-2 setup --dsn "host=host-2 user=postgres port=5432 dbname=pgddb" --
↪cluster-dsn "host=host-1 user=postgres port=5432 dbname=pgddb"
```

### On the third host

```
sudo -iu postgres
export PG_VERSION=17
export PATH=$PATH:/usr/edb/pge$PG_VERSION/bin/
export PGDATA=/var/lib/edb-pge/$PG_VERSION/data/
export PGPASSWORD=secret
pgd node node-3 setup --dsn "host=host-3 user=postgres port=5432 dbname=pgddb" --
↪cluster-dsn "host=host-1 user=postgres port=5432 dbname=pgddb"
```

The next step is to *Checking the cluster* .

## Checking the cluster

## Checking the cluster

With the cluster up and running, it's worthwhile to run some basic checks to see how effectively it's replicating.

The following example shows one quick way to do this, but you must ensure that any testing you perform is appropriate for your use case.

On any of the installed and configured nodes, log in and run `psql` to connect to the database. If you are using EDB Postgres Advanced Server, use the `enterprisedb` user, otherwise use `postgres` :

```
sudo -iu postgres psql pgddb
```

This command connects you *directly* to the database on host-1/node-1.

### Quick test

- **Preparation**
- Ensure the cluster is ready:
- Log in to the database on host-1/node-1.

- Run `select bdr.wait_slot_confirm_lsn(NULL, NULL);`.
- When the query returns, the cluster is ready.
- **Create data** The simplest way to test that the cluster is replicating is to log in to one node, create a table, and populate it.
- On node-1, create a table: `sql CREATE TABLE quicktest ( id SERIAL PRIMARY KEY, value INT );`
- On node-1, populate the table:

```
INSERT INTO quicktest (value) SELECT random()*10000 FROM generate_series(1,10000);
```

- On node-1, monitor performance:

```
select * from bdr.node_replication_rates;
```

- On node-1, get a sum of the value column (for checking):

```
select COUNT(*),SUM(value) from quicktest;
```

- **Check data**
- Log in to node-2. Log in to the database on host-2/node-2.
- On node-2, get a sum of the value column (for checking): `sql select COUNT(*),SUM(value) from quicktest;`
- Compare with the result from node-1.
- Log in to node-3. Log in to the database on host-3/node-3.
- On node-3, get a sum of the value column (for checking): `sql select COUNT(*),SUM(value) from quicktest;`
- Compare with the result from node-1 and node-2.

### Worked example

### Preparation

Log in to host-1's Postgres server.

```
ssh admin@host-1
sudo -iu postgres psql "host=host-1 port=5432 username=postgres dbname=pgddb"
```

This is your connection to PGD's node-1.

### Ensure the cluster is ready

To ensure that the cluster is ready to go, run:

```
select bdr.wait_slot_confirm_lsn(NULL, NULL)
```

This query blocks while the cluster is busy initializing and returns when the cluster is ready.

In another window, log in to host-2's Postgres server:

```
ssh admin@host-2
sudo -iu postgres psql "host=host-2 port=5432 username=postgres dbname=pgddb"
```

### Create data

#### On node-1, create a table

Run:

```
CREATE TABLE quicktest ( id SERIAL PRIMARY KEY, value INT );
```

#### On node-1, populate the table

```
INSERT INTO quicktest (value) SELECT random()*10000 FROM generate_series(1,10000);
```

This command generates a table of 10000 rows of random values.

#### On node-1, monitor performance

As soon as possible, run:

```
select * from bdr.node_replication_rates;
```

The command shows statistics about how quickly that data was replicated to the other two nodes:

```
pgddb=# select * from bdr.node_replication_rates;
__OUTPUT__
 peer_node_id | target_name | sent_lsn  | replay_lsn | replay_lag | replay_lag_bytes␣
↪| replay_lag_size | apply_rate | catchup_interv
al
- -------------+-------------+-----------+------------+------------+------------------
↪+-----------------+------------+---------------
- --
   1954860017 | node-3  | 0/DDAA908 | 0/DDAA908  | 00:00:00   |                0 | 0␣
↪bytes         |      13682 | 00:00:00
   2299992455 | node-2    | 0/DDAA908 | 0/DDAA908  | 00:00:00   |                0 |␣
↪0 bytes         |      13763 | 00:00:00
(2 rows)
```

And it's already replicated.

#### On node-1 get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets some values from the generated data:

```
pgddb=# select COUNT(*),SUM(value) from quicktest;
__OUTPUT__
 count  |    sum
- -------+-----------
 100000 | 498884606
(1 row)
```

### Check data

#### Log in to host-2's Postgres server

```
ssh admin@host-2
sudo -iu postgres psql "host=host-2 port=5432 username=postgres dbname=pgddb"
```

This is your connection to PGD's node-2.

#### On node-2, get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets node-2's values for the generated data:

```
pgddb=# select COUNT(*),SUM(value) from quicktest;
__OUTPUT__
 count  |    sum
- -------+-----------
 100000 | 498884606
(1 row)
```

#### Compare with the result from node-one

The values are identical.

You can repeat the process with node-3 or generate new data on any node and see it replicate to the other nodes.

#### Log in to host-3's Postgres server

```
ssh admin@host-3
sudo -iu enterprisedb psql pgddb
```

This is your connection to PGD's node-3.

#### On node-3, get a checksum

Run:

```
select COUNT(*),SUM(value) from quicktest;
```

This command gets node-3's values for the generated data:

```
pgddb=# select COUNT(*),SUM(value) from quicktest;
__OUTPUT__
 count  |    sum
- -------+-----------
 100000 | 498884606
(1 row)
```

#### Compare with the result from node-one and node-two

The values are identical.

#### Connecting to your PGD cluster

EDB Postgres Distributed (PGD) uses the same connection methods as Postgres. The difference is that most of your connections to the cluster go through the connection manager that's built into every data node in the cluster.

Although you can connect directly to the data nodes, we don't recommend it for anything other than maintenance when you want to work on a particular node's database instance.

In general, you must connect to the cluster through the connection manager. PGD is designed to be simple to deploy and manage, and that means the cluster has a write leader node that handles all the writes to the cluster. The connection manager is then responsible for directing your read-write connections to the write leader. All your client or application needs to do is to use the connection manager's port and the connection manager will handle the rest.

The connection manager is responsible for directing your writes to the write leader and ensuring that your reads are directed to the correct node in the cluster. If you connect directly to a data node, you may not be able to take advantage of these features. For applications that only need to read data, the connection manager can direct your reads to a node that isn't the write leader. This can help to balance the load on the cluster and improve performance.

#### Connecting through the connection manager

Postgres is very flexible for configuring ports and connections, so for simplicity, this example uses the default port settings for Postgres and the connection manager. The default port for Postgres is 5432, and the default port for the connection manager is 6432.

You can use that port in your connection strings to connect to the cluster. So, for example, if you're using the psql command line tool, you can connect to the cluster like this:

```
psql -h host-1 -p 6432 -U pgdadmin -d pgddb
```

Where `host-1` is the hostname of the node you're connecting to. The connection manager will then direct your connection to the write leader node in the cluster.

#### Connecting directly to a data node

You can connect directly to a data node in the cluster, but we don't recommend it. However, if you need to connect directly to a data node, you can use the following command:

```
psql -h host-1 -p 5432 -U pgdadmin -d pgddb
```

### 5.1.2 Deploying and configuring PGD using TPA

Trusted Postgres Architect (TPA) is the primary orchestration tool for deploying and managing highly available EDB Postgres Distributed (PGD) clusters on self-managed infrastructure. It uses Ansible to automate the installation, configuration, and optimization of the entire PGD stack. It provides additional value, such as compatibility with Oracle using EDB Postgres Advanced Server, Transparent Data Encryption (TDE) using EDB Postgres Extended or Advanced Server".

Refer to the TPA documentation for instructions on how to deploy a PGD cluster using TPA.

### 5.1.3 Deploying and configuring PGD on Kubernetes

EDB CloudNativePG Global Cluster is a Kubernetes operator designed, developed, and supported by EDB. It covers the full lifecycle of highly available Postgres database clusters with a multi-master architecture, using PGD replication. It's based on the open source CloudNativePG operator and provides additional value, such as compatibility with Oracle using EDB Postgres Advanced Server, Transparent Data Encryption (TDE) using EDB Postgres Extended or Advanced Server, and additional supported platforms including IBM Power and OpenShift.

Refer to the following resources to deploy a PGD cluster using EDB Postgres Distributed for Kubernetes:

- Quick start in the EDB CloudNativePG Global Cluster documentation works through the steps needed to:
-

**Installation and upgrade provides detailed instructions for installing and**
upgrading EDB CloudNativePG Global Cluster.

### 5.1.4 Deploying and configuring PGD on Hybrid Manager

EDB Postgres AI Hybrid Manager (HM) is the centralized control plane for the EDB Postgres AI platform, providing additional value such as Oracle compatibility via EDB Postgres Advanced Server and Transparent Data Encryption (TDE) using either EDB Postgres Extended or Advanced Server. It allows you to deploy and manage the full lifecycle of PGD-powered clusters through two primary configurations: Advanced High Availability (AHA) and Distributed High-Availability (DHA).

The primary distinction between these configurations lies in their scalability and group architecture:

- **Distributed High-Availability (DHA):** Designed for global scale and geographic redundancy, this configuration supports multiple data groups. This allows for distributed workloads across different locations or regions.
- **Advanced High Availability (AHA):** Designed for localized resilience, this configuration consists of a single data group. It is ideal for high availability within a single region or data center.

Detailed comparisons of these architectures, including resource requirements and recommended use cases, are provided within the Hybrid Manager console during the cluster creation process.

Refer to Creating a database cluster for step-by-step instructions on deploying these high-availability configurations.

## 5.2 Upgrading

While PGD and Postgres are closely related, they're separate products with separate upgrade paths. This section covers how to upgrade both PGD and Postgres.

### 5.2.1 Upgrading PGD

EDB Postgres Distributed is a flexible platform. This means that your upgrade path depends largely on how you installed PGD.

- ** *Upgrading PGD clusters manually* ** — If you manually installed and configured your PGD cluster, you can move a cluster between versions, both minor and major.
- ** *Supported PGD upgrade paths* ** — Several supported upgrade paths are available.
- ** *Compatibility changes* ** — If you're upgrading from PGD 3.x or 4.x to PGD 5.x or later, you need to understand the compatibility changes between versions.

### 5.2.2 Upgrading Postgres or Postgres and PGD major versions

- **

**In-place Postgres major version upgrades ** — How to use `pgd node upgrade` to manually** upgrade the Postgres version or Postgres and PGD major version on one or more nodes.

- ** *Performing a Postgres major version rolling upgrade on a PGD cluster* ** — How to perform a major version upgrade of Postgres on a cluster.

### 5.2.3 Other upgrades

- ** *Application schema upgrades* ** — A guide for safely upgrading your application's schema when running multiple distributed servers with PGD.

#### Upgrading PGD clusters manually

Because EDB Postgres Distributed consists of multiple software components, the upgrade strategy depends partially on the components that are being upgraded.

In general, you can upgrade the cluster with almost zero downtime by using an approach called *rolling upgrade*. Using this approach, nodes are upgraded one by one, and the application connections are switched over to already upgraded nodes.

You can also stop all nodes, perform the upgrade on all nodes, and only then restart the entire cluster. This approach is the same as with a standard PostgreSQL setup. This strategy of upgrading all nodes at the same time avoids running with mixed versions of software and therefore is the simplest. However, it incurs downtime and we don't recommend it unless you can't perform the rolling upgrade for some reason.

To upgrade an EDB Postgres Distributed cluster:

1. Plan the upgrade.
2. Prepare for the upgrade.
3. Upgrade the server software.
4. Check and validate the upgrade.

#### Upgrade planning

There are broadly two ways to upgrade each node:

- Upgrade nodes in place to the newer software version. See *Rolling server software upgrades* .
- Replace nodes with ones that have the newer version installed. See *Rolling upgrade using node join* .

You can use both of these approaches in a rolling manner.

#### Rolling upgrade considerations

While the cluster is going through a rolling upgrade, mixed versions of software are running in the cluster. For example, suppose nodeA has PGD 4.3.6, while nodeB and nodeC have 5.6.1. In this state, the replication and group management uses the protocol and features from the oldest version (4.3.6 in this example), so any new features provided by the newer version that require changes in the protocol are disabled. Once all nodes are upgraded to the same version, the new features are enabled.

Similarly, when a cluster with WAL-decoder-enabled nodes is going through a rolling upgrade, WAL decoder on a higher version of PGD node produces *logical change records (LCRs)* with a higher pglogical version. WAL decoder on a lower version of PGD node produces LCRs with a lower pglogical version. As a result, WAL senders on a higher version of PGD nodes aren't expected to use LCRs due to a mismatch in protocol versions. On a lower version of PGD

nodes, WAL senders can continue to use LCRs. Once all the PGD nodes are on the same PGD version, WAL senders use LCRs.

A rolling upgrade starts with a cluster with all nodes at a prior release. It then proceeds by upgrading one node at a time to the newer release, until all nodes are at the newer release. There must be no more than two versions of the software running at the same time. An upgrade must be completed, with all nodes fully upgraded, before starting another upgrade.

Where additional caution is required to reduce business risk, more time may be required to perform an upgrade. For maximum caution and to reduce the time required upgrading production systems, we suggest performing the upgrades in a separate test environment first.

Don't run with mixed versions of the software for any longer than is absolutely necessary to complete the upgrade. You can check on the versions in the cluster using the *pgd commit-scopes list* command.

The longer you run with mixed versions, the more likely you are to encounter issues, the more difficult it is to diagnose and resolve them. We recommend upgrading in off peak hours for your business, and over a short period of time.

While you can use a rolling upgrade for upgrading a major version of the software, we don't support mixing PostgreSQL, EDB Postgres Extended, and EDB Postgres Advanced Server in one cluster. So you can't use this approach to change the Postgres variant.

**Warning**

Downgrades of EDB Postgres Distributed aren't supported. They require that you manually rebuild the cluster.

### Rolling server software upgrades

A rolling upgrade is where the *Rolling server software upgrades* is upgraded sequentially on each node in a cluster without stopping the cluster. Each node is temporarily stopped from participating in the cluster and its server software is upgraded. Once updated, it's returned to the cluster, and it then catches up with the cluster's activity during its absence.

The actual procedure depends on whether the Postgres component is being upgraded to a new major version.

During the upgrade process, you can switch the application over to a node that's currently not being upgraded to provide continuous availability of the database for applications.

### Rolling upgrade using node join

The other method to upgrade the server software is to join a new node to the cluster and later drop one of the existing nodes running the older version of the software.

For this approach, the procedure is always the same. However, because it includes node join, a potentially large data transfer is required.

Take care not to use features that are available only in the newer Postgres version until all nodes are upgraded to the newer and same release of Postgres. This is especially true for any new DDL syntax that was added to a newer release of Postgres.

**Note**

*bdr_init_physical* makes a byte-by-byte copy of the source node so you can't use it while upgrading from one major Postgres version to another. In fact, currently *bdr_init_physical* requires that even the PGD version of the source and the joining node be exactly the same. You can't use it for rolling upgrades by way of joining a new node method. Instead, use a logical join.

### Upgrading a CAMO-enabled cluster

Upgrading a CAMO-enabled cluster requires upgrading CAMO groups one by one while disabling the CAMO protection for the group being upgraded and reconfiguring it using the new *Commit scopes* -based settings.

We recommended the following approach for upgrading two BDR nodes that constitute a CAMO pair to PGD 5.0:

1. Ensure `bdr.enable_camo` remains `off` for transactions on any of the two nodes, or redirect clients away from the two nodes. Removing the CAMO pairing while attempting to use CAMO leads to errors and prevents further transactions.
2. For BDR 4.x, deconfigure CAMO by using `bdr.remove_camo_pair` to uncouple the pair.
3. Upgrade the two nodes to PGD 5.0.
4. Create a dedicated node group for the two nodes and move them into that node group.
5. Create a *Commit scopes* for this node group and thus the pair of nodes to use CAMO.
6. Reactivate CAMO protection again either by setting a `default_commit_scope` or by changing the clients to explicitly set `bdr.commit_scope` instead of `bdr.enable_camo` for their sessions or transactions.
7. If necessary, allow clients to connect to the CAMO-protected nodes again.

### Upgrade preparation

Each major release of the software contains several changes that might affect compatibility with previous releases. These might affect the Postgres configuration, deployment scripts, as well as applications using PGD. We recommend considering these changes and making any needed adjustments in advance of the upgrade.

See individual changes mentioned in the release notes and any version-specific upgrade notes.

### Server software upgrade

Upgrading EDB Postgres Distributed on individual nodes happens in place. You don't need to back up and restore when upgrading the BDR extension.

### BDR extension upgrade

The BDR extension upgrade process consists of the following high-level steps:

### Fence the node

To make sure the node being upgraded does not become a write leader until the upgrade is complete, you should fence the node before initiating the upgrade.

### Stop Postgres

During the upgrade of binary packages, it's usually best to stop the running Postgres server first. Doing so ensures that mixed versions don't get loaded in case of an unexpected restart during the upgrade.

### Upgrade packages

The first step in the upgrade is to install the new version of the BDR packages. This installation installs both the new binary and the extension SQL script. This step is specific to the operating system.

### Start Postgres

Once packages are upgraded, you can start the Postgres instance. The BDR extension is upgraded upon start when the new binaries detect the older version of the extension.

### Unfence the node

You can unfence the node after the node upgrade is completed.

!!!Note

A PGD 4 cluster must be running PGD 4.4.1 before upgrading to PGD {Version 6.4}.

Upgrading from PGD 4.4.1 to PGD {Version 6.4} requires additional steps to move from Harp proxy to Connection Manager. For more information, see the worked example: *Upgrade PGD 4 to PGD {Version 6.4}* .

A PGD 5 cluster must be running PGD 5.9 before upgrading to PGD {Version 6.4}.

Upgrading from PGD 5.9 to PGD {Version 6.4} requires additional steps to move from PGD proxy to Connection Manager. For more information, see the worked example: *Upgrade PGD 5 to PGD {Version 6.4}* .

The process of in-place upgrade of Postgres depends on whether you're upgrading to a new minor version of Postgres or to a new major version of Postgres.

### Minor version Postgres upgrade

Upgrading to a new minor version of Postgres is similar to *BDR extension upgrade* . Stopping Postgres, upgrading packages, and starting Postgres again is typically all that's needed.

However, sometimes more steps, like reindexing, might be recommended for specific minor version upgrades. Refer to the release notes of the version of Postgres you're upgrading to.

### Major version Postgres upgrade

Upgrading to a new major version of Postgres is more complicated than upgrading to a minor version.

EDB Postgres Distributed provides a `pgd node upgrade` command line utility, which you can use to do in-place Postgres major version upgrades .

> **Note**
>
> When upgrading to a new major version of any software, including Postgres, the BDR extension, and others, it's always important to ensure your application is compatible with the target version of the software you're upgrading.

### Upgrade check and validation

After you upgrade your PGD node, you can verify the current version of the binary:

```
SELECT bdr.bdr_version();
```

Always check your *Monitoring the Connection Manager* after upgrading a node to confirm that the upgraded node is working as expected.

### PGD 5 - Moving from PGD Proxy to Connection Manager

Use the following steps to move from PGD Proxy to Connection Manager:

1. From one of the PGD 5.9 nodes, run the following query to ensure that SCRAM hashes of all user passwords are the same across all nodes:

- ```sql

  DO

  $$DECLARE rec RECORD; command TEXT; password TEXT; BEGIN FOR rec IN SELECT rolname, rolpassword F

  ; SELECT wait_slot_confirm_lsn(NULL, NULL); ```

- !!!Note

**No new users should be added to 5.9 after executing this. If they are added, run the query again. The above block does not change the passwords, it just ensure SCRAM hashes are same across the cluster on all nodes. .. ::**

2. Fence a node in this cluster with *pgd node <node-name> set-option route_fence true* so that it does not become the write leader. 3. Enable the GUC *bdr.enable_builtin_connection_manager* to *true*. 4. Restart the server. 5. Stop PGD Proxy running on the server. 6. Restart the server. It will start with the Connection Manager running on the default port. If the proxy read and write ports were different, the Connection Manager port read and write ports can be changed to be the same as proxy by *bdr.alter_node_group_option()* . 7. Unfence the node. This node can now accept connections from the user and route to the write leader via Connection Manager. 8. Repeat this for each node in the cluster. This will ensure all nodes are now routing via Connection Manager. 9. If you're also performing a major version upgrade from PGD 5 to PGD {Version 6.4}, proceed with the *rolling upgrade steps* .

### Supported PGD upgrade paths

### Upgrading to version {Version 6.4}

EDB Postgres Distributed uses semantic versioning . All changes within the same major version are backward compatible, lowering the risk when upgrading and allowing you to choose any later minor or patch release as the upgrade target.

The following upgrade paths are supported for PGD {Version 6.4}:

| Source version | Target version |
|---|---|
| 4.4.1 | {Version 6.4} |
| 5.9.1 | {Version 6.4} |
| 6.0.2 | {Version 6.4} |

### Compatibility changes

### Summary

There are some changes in PGD {Version 6.4} that are not compatible with 5.x and other previous versions. For users of 4.x, the version 5.x compatibility changes still apply.

### Connection Manager

PGD 6 introduces a new Connection Manager which replaces the PGD 5's proxy solution with a tightly integrated approach using a background worker to expose read-write, read-only and http-status network interfaces in PGD. This means that all the old proxy functions were removed, as there are no longer separate PGD-Proxy instances to manage and the connection manager uses group configuration.

Moreover, `bdr.proxy_conifg` table and `bdr.proxy_config_summary` view were removed. You should now use `bdr.node_group_config_summary` for the connection manager's configuration. This also includes all the other group settings, including routing.

For more details see *Connection Manager Authentication* .

### DDL Locking changes

There is a new global lock type, `ddl_locking = leader` that's used by PGD {Version 6.4} by default instead of the global DML lock. This lock ensures that a table is exclusively locked on all group leaders (as opposed to all nodes). For PGD Essential this means that in practice the lock is engaged immediately as there is only one leader, which means the system exhibits the same behavior as a single node Postgres setup. `LOCK TABLE` and `TRUNCATE TABLE` now use this new lock type by default (before the default behavior was to not lock outside of the origin node).

`LOCK TABLE` now uses DDL lock for `ACCESS SHARE` , `ROW SHARE` , `ROW EXCLUSIVE` , `SHARE UPDATE EXCLUSIVE` and DML lock for `SHARE ROW EXCLUSIVE` , `EXCLUSIVE` , and `ACCESS EXCLUSIVE` . This means that `SHARE+` blocks DMLs, but lower levels don't (which is more closely aligned with how these levels work in Postgres). `LOCK TABLE` used to always use DML lock.

### Join/part behavior

### Function and node state changes

For `bdr.part_node` :

- `wait_for_completion` works on the local node
- the `force` option is deprecated and has the same behavior as `bdr.drop_node(force:=true)`

For `bdr.drop_node` :

- the `cascade` option was removed
- `force` now always immediately drops the node locally and assumes previous cascade behavior as well

For `bdr.join_node_group` :

- the `pause_in_standby` option was removed
- now whether the node is in standby is determined by `node_kind` when running `bdr.create_node`

A new node state `PART_CLEANUP` was added, which waits for the group slot to move past any records relevant to the parting node and cleans up origin records related to the node.

### Sequences conversion

When creating a PGD cluster, existing *Sequences* are automatically converted to distributed ones.

### Parting a node drops it

When a node is parted, it is eventually dropped and the node will not be seen in the `PARTED` state at the end of the parting. Therefore, scripts that wait for the node to reach `PARTED` state may need change. Parting can also take longer than earlier versions since this operation waits till the group slot on all crosses of the last transaction from the parting node to ensure consistency.

### Administration function changes

- The `enable_proxy_routing` node group option is now `enable_routing` , although `bdr.alter_node_group_option` still supports the older syntax for backwards compatibility. The views only show the new name.
- The already deprecated `bdr.alter_node_group_config()` was removed. Use `bdr.alter_node_group_option()` instead.
- The `managed_locally` option in `bdr.autopartition()` was removed. Only locally managed partitions are now supported. At the same time `ap_is_local` was removed from `bdr.autopartition_rules` catalog as well.

### Other

### Replica identity

- Replica identity defaults to FULL now.
- This removes some edge cases in conflict detection between inserts, updates, and deletes across node crashes and recovery.

### Run on (all) writes

- Run on (all) writes is now not replicated.
- The `bdr.xact_replicate` is set to `off` by default, reducing possible accidental issues with replication of commands sent via the run on (all) nodes command.
- Some log messages were reworded to be easier to understand.
- Some internal log messages' log levels were lowered from `LOG` to `DEBUG` so they may no longer appear in logs.

### General UI improvements

- Workers renamed to not contain 'pglogical'
- Previously deprecated configuration parameters (GUCs) now removed (this includes, for example, all the pglogical prefixed GUCs).
- Conflicts are logged neither to Postgres log or `conflict_history` table by default

### Changes to defaults

### Global lock timeout

`bdr.global_lock_timeout` is set to 1 minute by default.

### Auto sync

`bdr.enable_auto_sync_reconcile` is set to ON by default. This feature is documented in *Relationship to automatic synchronization* . It enables changes from a down node to be synced to all other nodes.

### Application schema upgrades

Similar to the upgrade of EDB Postgres Distributed, there are two approaches to upgrading the application schema. The simpler option is to stop all applications affected, preform the schema upgrade, and restart the application upgraded to use the new schema variant. This approach imposes some downtime.

To eliminate this downtime, EDB Postgres Distributed offers useful tools to perform a rolling application schema upgrade.

The following recommendations and tips reduce the impact of the application schema upgrade on the cluster.

### Rolling application schema upgrades

By default, DDL is automatically sent to all nodes. You can control this behavior manually, as described in *Running DDL safely* . You can use this approach to create differences between database schemas across nodes.

PGD is designed to allow replication to continue even with minor differences between nodes. These features are designed to allow application schema migration without downtime or to allow logical standby nodes for reporting or testing.

Careful scripting is required to make this work correctly on production clusters. We recommend extensive testing.

See *Replicating between nodes with differences* for details.

When one node runs DDL that adds a new table, nodes that haven't yet received the latest DDL need to handle the extra table. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `skip` resolver in case of a `target_table_missing` conflict. Perform this configuration before adding tables to any node. This setting is intended to be permanent.

Execute the following query **separately on each node**. Replace `node1` with the actual node name.

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
       target_table_missing, skip);
```

When one node runs DDL that adds a column to a table, nodes that haven't yet received the latest DDL need to handle the extra columns. In view of this, the appropriate setting for rolling schema upgrades is to configure all nodes to apply the `ignore` resolver in case of a `target_column_missing` conflict. Perform this before adding columns to one node. This setting is intended to be permanent.

Execute the following query **separately on each node**. Replace `node1` with the actual node name.

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
       target_column_missing, ignore);
```

When one node runs DDL that removes a column from a table, nodes that haven't yet received the latest DDL need to handle the missing column. This situation causes a `source_column_missing` conflict, which uses the `use_default_value` resolver. Thus, columns that don't accept NULLs and don't have a DEFAULT value require a two-step process:

1. Remove the NOT NULL constraint, or add a DEFAULT value for a column on all nodes.
2. Remove the column.

You can remove constraints in a rolling manner. There's currently no supported way for handling adding table constraints in a rolling manner, one node at a time.

When one node runs a DDL that changes the type of an existing column, depending on the existence of binary coercibility between the current type and the target type, the operation might not rewrite the underlying table data. In that case, it's only a metadata update of the underlying column type. Rewriting a table is normally restricted. However, in controlled DBA environments, you can change the type of a column to an automatically castable one by adopting a rolling upgrade for the type of this column in a non-replicated environment on all the nodes, one by one. See *ALTER TABLE* for more details.

### In-place Postgres or Postgres and PGD major version upgrades

You can upgrade a PGD node to a newer major version of Postgres or a major version of Postgres and PGD using the command-line utility pgd node upgrade .

!!!Note In previous versions before 5.7.0, the command used for in-place major version upgrades was `bdr_pg_upgrade` . However, this command didn't have an option to upgrade both Postgres major versions and PGD versions simultaneously, as `pgd node upgrade` does.

`pgd node upgrade` is a wrapper around the standard pg_upgrade that adds PGD-specific logic to the process to ensure a smooth upgrade.

### Terminology

This terminology is used when describing the upgrade process and components involved:

*Postgres cluster* _ The database files, both executables and data, that make up a Postgres database instance on a system when run.

*Old Postgres cluster* _ The existing Postgres cluster to upgrade, the one from which to migrate data.

*New Postgres cluster* _ The new Postgres cluster that data is migrated to. This Postgres cluster must be one major version ahead of the old cluster.

### Precautions

Standard Postgres major version upgrade precautions apply, including the fact both Postgres clusters must meet all the requirements for pg_upgrade .

Additionally, don't use `pgd node upgrade` if other tools are using replication slots and replication origins. Only PGD slots and origins are restored after the upgrade.

You must meet several prerequisites for `pgd node upgrade` :

- Disconnect applications using the old Postgres cluster. You can, for example, redirect them to another node in the PGD cluster.
- Configure peer authentication for both Postgres clusters. `bdr_pg_upgrade` requires peer authentication.
- The same or newer version of PGD must be installed on the new cluster.
- The PGD version must be 4.1.0 or later.
- The new cluster must be in a shutdown state.
- You must install PGD packages in the new cluster.
- The new cluster must already be initialized and configured as needed to match the old cluster configuration.
- Databases, tables, and other objects must not exist in the new cluster.

> **Note**
>
> When upgrading to PGD 5.7.0+, you don't need to have both clusters run the same PGD version. The new cluster must be running 5.7.0+. In that case *pgd node upgrade* will upgrade the PGD version to 5.7.x and upgrade the Postgres major version.

We also recommend having the old Postgres cluster up prior to running `pgd node upgrade`. The CLI starts the old Postgres cluster if it's shut down.

### Usage

To upgrade to a newer major version of Postgres or Postgres and PGD, you must first install the new version packages.

#### `pgd node upgrade` command-line

`pgd node upgrade` passes all parameters to pg_upgrade. Therefore, you can specify any parameters supported by pg_upgrade.

### Synopsis

```
pgd node <NODE_NAME> upgrade [OPTION] ...
```

### Options

In addition to the options for pg_upgrade, you can pass the following parameters to `pgd node upgrade`.

### Required parameters

Specify these parameters either in the command line or, for all but the `--database` parameter, in their equivalent environment variable. They're used by `pgd node upgrade`.

- `-b, --old-bindir` _ Old Postgres cluster bin directory.
- `-B, --new-bindir` _ New Postgres cluster bin directory.
- `-d, --old-datadir` _ Old Postgres cluster data directory.
- `-D, --new-datadir` _ New Postgres cluster data directory.
- `--database` _ PGD database name.

### Optional parameters

These parameters are optional and are used by `pgd node upgrade`:

- `-p, --old-port` _ Old cluster port number.
- `-s, --socketdir` _ Directory to use for postmaster sockets during upgrade.
- `--check` _ Specify to only perform checks and not modify clusters.

### Other parameters

Any other parameter that's not one of the above is passed to pg_upgrade. pg_upgrade accepts the following parameters:

- `-j, --jobs` _ Number of simultaneous processes or threads to use.
- `-k, --link` _ Use hard links instead of copying files to the new cluster.
- `-o, --old-options` _ Option to pass to old postgres command. Multiple invocations are appended.
- `-O, --new-options` _ Option to pass to new postgres command. Multiple invocations are appended.
- `-N, --no-sync` _ Don't wait for all files in the upgraded cluster to be written to disk.
- `-P, --new-port` _ New cluster port number.
- `-r, --retain` _ Retain SQL and log files even after successful completion.
- `-U, --username` _ Cluster's install user name.
- `--clone` _ Use efficient file cloning.

### Environment variables

You can use these environment variables in place of command-line parameters:

- `PGBINOLD` _ Old Postgres cluster bin directory.
- `PGBINNEW` _ New Postgres cluster bin directory.
- `PGDATAOLD` _ Old Postgres cluster data directory.
- `PGDATANEW` _ New Postgres cluster data directory.
- `PGPORTOLD` _ Old Postgres cluster port number.
- `PGSOCKETDIR` _ Directory to use for postmaster sockets during upgrade.

### Example

Given a scenario where:

- Node name of the cluster you want to upgrade is kaolin.
- Old Postgres cluster bin directory is `/usr/lib/postgresql/16/bin` .
- New Postgres cluster bin directory is `/usr/lib/postgresql/17/bin` .
- Old Postgres cluster data directory is `/var/lib/postgresql/16/main` .
- New Postgres cluster data directory is `/var/lib/postgresql/17/main` .
- Database name is `bdrdb` .

You can use the following command to upgrade the cluster:

```
pgd node kaolin upgrade \
- -old-bindir /usr/lib/postgresql/16/bin \
- -new-bindir /usr/lib/postgresql/17/bin \
- -old-datadir /var/lib/postgresql/16/main \
- -new-datadir /var/lib/postgresql/17/main \
- -database bdrdb
```

### Steps performed

These steps are performed when running `pgd node upgrade` .

> **Note**
>
> When *--check* is supplied as an argument to *pgd node upgrade* , the CLI skips steps that modify the database.

### PGD Postgres checks

| Steps | *--check* supplied |
|---|---|
| Collecting pre-upgrade new cluster control data | *run* |
| Checking new cluster state is shutdown | *run* |
| Checking PGD versions | *run* |
| Starting old cluster (if shutdown) | *skip* |
| Connecting to old cluster | *skip* |
| Checking if bdr schema exists | *skip* |
| Turning DDL replication off | *skip* |
| Terminating connections to database | *skip* |
| Waiting for all slots to be flushed | *skip* |
| Disconnecting from old cluster | *skip* |
| Stopping old cluster | *skip* |
| Starting old cluster with PGD disabled | *skip* |
| Connecting to old cluster | *skip* |
| Collecting replication origins | *skip* |
| Collecting replication slots | *skip* |
| Disconnecting from old cluster | *skip* |
| Stopping old cluster | *skip* |

### pg_upgrade steps

Standard pg_upgrade steps are performed.

> **Note**
>
> If supplied, *--check* is passed to pg_upgrade.

### PGD post-upgrade steps

| Steps | *--check* supplied |
|---|---|
| Collecting old cluster control data | *skip* |
| Collecting new cluster control data | *skip* |
| Advancing LSN of new cluster | *skip* |
| Starting new cluster with PGD disabled | *skip* |
| Connecting to new cluster | *skip* |
| Creating replication origin, repeated for each origin | *skip* |
| Advancing replication origin, repeated for each origin | *skip* |

continues on next page

Table 10 – continued from previous page

| Steps | *--check* supplied |
|---|---|
| Creating replication slot, repeated for each slot | *skip* |
| Stopping new cluster | *skip* |

### Performing a Postgres major version rolling upgrade on a PGD cluster

### Upgrading Postgres major versions

Upgrading a Postgres database's major version to access improved features, performance enhancements, and security updates is a common administration task. Doing the same for an EDB Postgres Distributed (PGD) cluster is essentially the same process but performed as a rolling upgrade.

The rolling upgrade process allows updating individual cluster nodes to a new major Postgres version while maintaining cluster availability and operational continuity. This approach minimizes downtime and ensures data integrity by allowing the rest of the cluster to remain operational as each node is upgraded sequentially.

The following overview of the general instructions and worked examples help to provide a smooth and controlled upgrade process.

### Prepare the upgrade

To prepare for the upgrade, identify the subgroups and nodes you're trying to upgrade and note an initial upgrade order.

To do this, connect to one of the nodes using SSH and run the `pgd nodes list` command:

```
sudo -u postgres pgd nodes list
```

The `pgd nodes list` command shows you all the nodes in your PGD cluster and the subgroup to which each node belongs. Then you want to find out which node is the write leader in each subgroup:

```
sudo -u postgres pgd group <group_name> show --summary
```

This command shows you information about the pgd group tokened by your `<group_name>` running in your cluster, including which node is the write leader. To maintain operational continuity, you need to switch write leaders over to another node in their subgroup before you can upgrade them. To keep the number of planned switchovers to a minimum, when upgrading a subgroup of nodes, upgrade the writer leaders last.

To make sure the node being upgraded does not become a write leader until the upgrade is complete, you should fence the node before initiating the upgrade and then unfence the node after the node upgrade is completed.

Even though you verified which node is the current write leader for planning purposes, the write leader of a subgroup could change to another node at any moment for operational reasons before you upgrade that node. Therefore, you still need to verify that a node isn't the write leader just before upgrading that node.

You now have enough information to determine your upgrade order, one subgroup at a time, aiming to upgrade the identified write leader node last in each subgroup.

### Perform the upgrade on each node

**Important**

To help prevent data loss, before starting the upgrade process, ensure that your databases and configuration files are backed up.

Using the *Prepare the upgrade* , perform the following steps on each node while connected via SSH:

- **Confirm the current Postgres version**
- View versions from PGD:

```
sudo -u postgres pgd nodes list --versions.
```

- Ensure that the expected major version is running.
- **Verify that the target node isn't the write leader**
- Check whether the target node is the write leader for the group you're upgrading:

```
sudo -u postgres pgd group <group_name> show --summary
```

- If the target node is the current write leader for the group/subgroup you're upgrading, perform a planned switchover to another node:

```
sudo -u postgres pgd group <group_name> set-leader <new_leader_node_name>
```

- **Fence the node**
- To make sure the node being upgraded does not become a write leader until the upgrade is complete, you should fence the node before initiating the upgrade.
- **Stop Postgres on the target node**
- Stop the Postgres service on the current node:

```
sudo systemctl stop postgres
```

The target node is no longer actively participating as a node in the cluster.

- **Install PGD and utilities**
- Install PGD and its utilities compatible with the Postgres version you're upgrading to:

```
sudo apt install edb-pgd6-<edition>-pg<new_postgres_version_number>
```

- **Initialize the new Postgres instance**
- Create a directory to house the database files for the new version of PostgreSQL:

```
sudo mkdir -p /opt/postgres/datanew
```

- Ensure that the user postgres has ownership permissions to the directory using `chown` .
- Initialize a new PostgreSQL database cluster in the directory you just created. This step involves using the `initdb` command provided by the newly installed version of PostgreSQL. Include the `--data-checksums` flag to ensure the cluster uses data checksums.

```
sudo -u postgres <path_to_postgres_bin>/initdb -D /opt/postgres/datanew --data-check-
sums
```

Replace `<path_to_postgres_bin>` with the path to the bin directory of the newly installed PostgreSQL version.

You may need to run this command as the postgres user or another user with appropriate permissions.

- **Migrate configuration to the new Postgres version**
- Locate the following configuration files in your current PostgreSQL data directory:
- `postgresql.conf` _ The main configuration file containing settings related to the database system.
- `postgresql.auto.conf` _ Contains settings set by PostgreSQL, such as those modified by the `ALTER SYSTEM` command.

- `pg_hba.conf` _ Manages client authentication, specifying which users can connect to which databases from which hosts.
- The entire `conf.d` directory (if present) _ Allows for organizing configuration settings into separate files for better manageability.
- Copy these files and the `conf.d` directory to the new data directory you created for the upgraded version of PostgreSQL.

You may find it more efficient to copy just the uncommented lines from your existing configuration files into the new ones, checking the documentation for each setting as you go to ensure no further adjustments are needed.

- **Verify the Postgres service is inactive**
- Before proceeding, it's important to ensure that no PostgreSQL processes are active for both the old and the new data directories. This verification step prevents any data corruption or conflicts during the upgrade process.

Use the `sudo systemctl status postgres` command to verify that Postgres was stopped. If it isn't stopped, run `systemctl stop postgres` and verify again that it was stopped.

- **Swap PGDATA directories for version upgrade**
- Rename `/opt/postgres/data` to `/opt/postgres/dataold` and `/opt/postgres/datanew` to `/opt/postgres/data` .

This step readies your system for the next crucial phase: running pgd node upgrade to finalize the PostgreSQL version transition.

- **Verify upgrade feasibility**
- The `pgd node upgrade` tool offers a `--check` option designed to perform a preliminary scan of your current setup, identifying any potential issues that could hinder the upgrade process.

You need to run this check from an upgrade directory with ownership given to user postgres, such as `/home/upgrade/` , so that the upgrade log files created by `pgd node upgrade` can be stored. To initiate the safety check, append the `--check` option to your `pgd node upgrade` command.

This operation simulates the upgrade process without making any changes, providing insights into any compatibility issues, deprecated features, or configuration adjustments required for a successful upgrade.

- Address any warnings or errors indicated by this check to ensure an uneventful transition to the new version.
- **Execute the Postgres major version upgrade**
- Execute the upgrade process by running the `pgd node <node_name> upgrade` command without the `--check` option.
- It's essential to monitor the command output for any errors or warnings that require attention.
- The time the upgrade process take depends on the size of your database and the complexity of your setup.
- **Update the Postgres service configuration**
- Update the service configuration to reflect the new PostgreSQL version by updating the version number in the `postgres.service` file:

```
sudo sed -i -e 's/<old_version_number>/<new_version_number>/g' /etc/systemd/system/
postgres.service
```

- Refresh the system's service manager to apply these changes:

```
sudo systemctl daemon-reload
```

- **Restart Postgres**
- Proceed to restart the PostgreSQL service:

```
systemctl start postgres
```

- **Validate the new Postgres version**
- Verify that your PostgreSQL instance is now upgraded:

```
sudo -u postgres pgd nodes list --versions
```

- **Unfence the node**
- You can unfence the node after validating the upgrade.
- **Clean up post-upgrade**
- Run `vacuumdb` with the `ANALYZE` option immediately after the upgrade but before introducing a heavy production load. Running this command minimizes the immediate performance impact, preparing the database for more accurate testing.
- Remove the old version's data directory, `/opt/postgres/dataold` .

### Worked example: Upgrade PGD 4 to PGD 6

This worked example describes an in-place major version rolling upgrade from PGD 4 to PGD {Version 6.4}.

### Overview

A PGD 4 cluster using HARP Proxy based routing continues this routing method to all nodes until the entire cluster is upgraded to {Version 6.4}.0 or higher. HARP-proxy based routing functions the same within a mixed version cluster. HARP uses its own mechanism to elect a leader since a 4.x cluster does not have a write leader. Once the entire cluster is upgraded to version {Version 6.4}.0 or higher, Connection Manager is already enabled and ready. Applications then can be pointed to the Connection Manager port/node and it will start routing. Once confirmed to be working, proxy can be stopped.

!!!Note The worked example assumes Harp proxy is not co-located with a PGD node, as recommended in the PGD architecture. If you have a Harp proxy co-located with a PGD node, contact EDB Support for upgrade instructions.

### Confirm the harp-proxy leader

Start the upgrade on a node that isn't the harp-proxy leader. Confirm which node is the harp-proxy leader:

```
test-pgd6major-d1:~ $ harpctl get leader a
```

```
Cluster  Name                      Location Ready Fenced Allow Routing Routing Status␣
↪Role    Type Lock Duration
- ------  ----                       -------- ----- ------ ------------- --------------␣
↪----    ---- -------------
bdrgroup test-pgd6major-d2 a        true  false  true                          primary␣
↪bdr  6
```

### Fence the node

Fence off the node to be upgraded from HARP and then verify it was fenced, so it does not become the leader during the middle of upgrade:

```
test-pgd6major-d1:~ $ harpctl fence test-pgd6major-d1
```

```
INFO    cmd/fence.go:42 fence node test-pgd6major-d1
```

```
test-pgd6major-d1:~ $ harpctl get nodes
Cluster  Name                    Location Ready Fenced Allow Routing Routing Status␣
↪Role    Type Lock Duration
- ------  ----                     -------- ----- ------ ------------- --------------␣
↪----    ---- --------------
bdrgroup test-pgd6major-d1 a        false true   true          N/A            primary␣
↪bdr  6
bdrgroup test-pgd6major-d2 a        true  false  true          ok             primary␣
↪bdr  6
```

### Stop the Postgres service

On the fenced node, stop the Postgres service.

### Stop HARP manager

On the fenced node, stop HARP manager.

### Remove and install packages

On the fenced node, remove PGD 4.4 and the cli packages and install the PGD {Version 6.4} packages.

### Start the Postgres service

On the fenced node, start Postgres service. This performs an in-place upgrade of the PGD local node to PGD {Version 6.4} with Connection Manager enabled.

### Start HARP manager

On the fenced node, start HARP manager.

### Unfence the node

Unfence the upgraded node from HARP:

```
test-pgd6major-d1:~ $ harpctl unfence test-pgd6major-d1
```

### Repeat steps for all nodes

Repeat same steps on all other nodes.

### Confirm cluster version

Confirm the updated cluster version is version 6001 by running `bdr.group_raft_details` .

### Confirm SCRAM hashes

From any of the upgrades nodes, run the following query to ensure that SCRAM hashes are the same across all nodes for each user. This is required before applications switch to Connection Manager.

```
DO $$
DECLARE
    rec RECORD;
    command TEXT;
BEGIN
    FOR rec IN SELECT rolname,rolpassword FROM pg_authid WHERE rolcanlogin = true AND␣
↪rolpassword like SCRAM-SHA%
    LOOP
        command := ALTER ROLE  || quote_ident(rec.rolname) ||  WITH ENCRYPTED␣
↪PASSWORD  ||  || rec.rolpassword || ;
        EXECUTE command;
    END LOOP;
END;
$$;
SELECT wait_slot_confirm_lsn(NULL,NULL);
```

### Enable routing

Enable node group routing as per your global or local routing requirement. For local routing enable it on subgroups, for global routing enable it on the top group.

```
bdrdb=# SELECT bdr.alter_node_group_option(node_group_name := bdrgroup,config_key :=␣
↪enable_routing, config_value := true::TEXT);
```

Output:

```
 alter_node_group_option
- ------------------------
 (1 row)
```

### Switch to Connection Manager

It should now be safe to switch your application to Connection Manager.

### Stop harp manager and proxy services.

It should now be safe to stop any running harp manager and proxy services.

> **Note**
>
> This completes the worked example for an in-place major version rolling upgrade from PGD 4 to PGD {Version 6.4}.

### Worked example: Upgrade PGD 5 to PGD 6

This worked example describes an in-place major version rolling upgrade (PGD & Postgres) of a 3-node PGD 5, EPAS 13 cluster to PGD {Version 6.4}, EPAS 17 cluster using the `pgd node upgrade` command.

### Prerequisites

Ensure you have a 3-node PGD 5, EPAS 13 cluster up and running. This is a TPA-deployed cluster.

```
pgd-a1:/home/rocky $ pgd nodes list
```

Output:

```
Node Name  Group Name Node Kind Join State Node Status
- --------- ---------- --------- ---------- -----------
pgd-a1     group-a    data      ACTIVE     Up
pgd-a2     group-a    data      ACTIVE     Up
witness-a1 group-a    witness   ACTIVE     Up
```

### Install packages for the new server and PGD

Ensure that the packages for EPAS 17 and the corresponding PGD 6 packages are installed on all nodes in the cluster. To prevent binary conflicts, you must remove the PGD 5 packages, viz., edb-pgd5-cli and edb-bdr5-epas13, before installing the PGD 6 packages. The commands below were used for the RHEL 8 platform. Use the appropriate commands for your specific platform.

```
dnf remove edb-pgd5-cli
dnf install edb-as17-server edb-pgd6-essential-epas17 -y
```

### Pre-upgrade steps

#### Version Check

Check the current version of the cluster (optional).

```
pgd-a1:/home/rocky $ pgd nodes list --versions
```

Output:

```
Node Name  BDR Version  Postgres Version
- --------- ------------ ----------------
pgd-a1     5.9.0        13.22.28
pgd-a2     5.9.0        13.22.28
witness-a1 5.9.0        13.22.28
```

### Move to Connection Manager

PGD 5 uses PGD Proxy for routing. In PGD 6, PGD Proxy has been replaced with Connection Manager. When you upgrade from PGD 5 to PGD 6, use the following steps to move to Connection Manager. See *PGD 5 - Moving from PGD Proxy to Connection Manager* .

### Write leader node verification

Ensure that the node you want to upgrade is not the write leader node.

```
pgd-a1:/home/rocky $ pgd group group-a show --summary
```

Output:

```
Group Property    Value
- ---------------- -------
Group Name        group-a
Parent Group Name dc-1
Group Type        data
Write Leader      pgd-a2
Commit Scope
```

The current write leader is node pgd-a2, so we are good to upgrade node pgd-a1.

Switch the write leader to a different node if it is the node to be upgraded.

Use the `pgd group set-leader` command to switch the write leader if required:

```
witness-a1:/home/rocky $ /usr/edb/as17/bin/pgd group group-a set-leader pgd-a1
```

```
witness-a1:/home/rocky $ /usr/edb/as17/bin/pgd group group-a show --summary
```

Output:

```
 Group Property    | Value
- ------------------+----------
 Group Name        | group-a
 Parent Group Name | dc-1
 Group Type        | data
 Write Leader      | pgd-a1
 Commit Scope      |
```

### Fence the node

Fence a node in this cluster with `pgd node <node-name> set-option route_fence true` so that it does not become the write leader.

### Initialize the new Postgres instance

Execute the initdb utility to initialize the new server. Ensure the `--data-checksums` option is set.

```
/usr/edb/as17/bin/initdb -D /var/lib/edb/as17/data -E UTF8 --data-checksums
```

Create the new data dir if you don't want to use the default one. This example uses the default for simplicity.

### Migrate configuration to the new Postgres version

Copy the following files and directory (if present), to the new data directory you created for the upgraded version of PostgreSQL:

- `postgresql.conf`
- `postgresql.auto.conf`
- `pg_hba.conf`
- the `conf.d` directory

```
cp /opt/postgres/data/postgresql.conf /var/lib/edb/as17/data/
cp /opt/postgres/data/postgresql.auto.conf /var/lib/edb/as17/data/
cp /opt/postgres/data/pg_hba.conf /var/lib/edb/as17/data/
```

If you have a TPA-deployed cluster, copy the `conf.d` directory as well:

```
cp -r /opt/postgres/data/conf.d /var/lib/edb/as17/data/
```

For instance, you may encounter the `operator_precedence_warning` GUC, which can be ignored in the new configuration.

```
sudo systemctl stop postgres
sudo systemctl status postgres
```

!!!Note

systemctl commands were used in this example because the PostgreSQL instance is configured as a service. You might need to use the `pg_ctl` utility if your setup is different.

Before running the actual upgrade, perform a dry run to check the compatibility and upgrade feasibility. The `pgd node upgrade` tool has a `--check` option, which performs a dry run of some of the upgrade process. You can use this option to ensure the upgrade goes smoothly. Run the upgrade command with the `--check` option.

```
/usr/edb/as17/bin/pgd node pgd-a1 upgrade \
- -database bdrdb -B /usr/edb/as17/bin \
- -socketdir /tmp \
- -old-bindir /usr/edb/as13/bin \
- -old-datadir /opt/postgres/data \
- -new-datadir /var/lib/edb/as17/data \
- -username enterprisedb \
- -old-port 5444 \
- -new-port 5444 \
- -check
```

A successful check should return output as shown:

```
Performing BDR Postgres Checks
- ------------------------------
Getting old PG instance shared directory                    ok
Getting new PG instance shared directory                    ok
Collecting pre-upgrade new PG instance control data         ok
Checking new cluster state is shutdown                      ok
Checking BDR extension versions                             ok
Checking Postgres versions                                  ok

Finished BDR pre-upgrade steps, calling pg_upgrade
- ------------------------------------------------

Performing Consistency Checks
- -----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
```

(continues on next page)

(continued from previous page)

```
Checking for contrib/isn with bigint-passing mismatch         ok
Checking data type usage                                      ok
Checking for user-defined encoding conversions                ok
Checking for user-defined postfix operators                   ok
Checking for incompatible polymorphic functions               ok
Checking for not-null constraint inconsistencies              ok
Checking for presence of required libraries                   ok
Checking database user is the install user                    ok
Checking for prepared transactions                            ok
Checking for new cluster tablespace directories               ok

*Clusters are compatible*
```

### Execute the upgrade.

If the dry run check passed, you can execute the upgrade by running the command without the `--check` option

```
/usr/edb/as17/bin/pgd node pgd-a1 upgrade \
- -database bdrdb -B /usr/edb/as17/bin \
- -socketdir /tmp \
- -old-bindir /usr/edb/as13/bin \
- -old-datadir /opt/postgres/data \
- -new-datadir /var/lib/edb/as17/data \
- -username enterprisedb \
- -old-port 5444 \
- -new-port 5444
```

A successful upgrade should return output as shown:

```
Performing BDR Postgres Checks
- ------------------------------
Getting old PG instance shared directory                      ok
Getting new PG instance shared directory                      ok
Collecting pre-upgrade new PG instance control data           ok
Checking new cluster state is shutdown                        ok
Checking BDR extension versions                               ok
Checking Postgres versions                                    ok

Collecting Pre-Upgrade BDR Information
- -----------------------------------
Collecting pre-upgrade old PG instance control data           ok
Connecting to the old PG instance                             ok
Checking for BDR extension                                    ok
Checking BDR node name                                        ok
Terminating connections to database                           ok
Waiting for all slots to be flushed                           ok
Disconnecting from old cluster PG instance                    ok
Stopping old PG instance                                      ok
Starting old PG instance with BDR disabled                    ok
Connecting to the old PG instance                             ok
Collecting replication origins                                ok
Collecting replication slots                                  ok
```

(continues on next page)

(continued from previous page)

```
Disconnecting from old cluster PG instance                    ok
Stopping old PG instance                                      ok

Finished BDR pre-upgrade steps, calling pg_upgrade
- ------------------------------------------------

Performing Consistency Checks
- ----------------------------
Checking cluster versions                                     ok
Checking database user is the install user                    ok
Checking database connection settings                         ok
Checking for prepared transactions                            ok
Checking for contrib/isn with bigint-passing mismatch         ok
Checking data type usage                                      ok
Checking for user-defined encoding conversions                ok
Checking for user-defined postfix operators                   ok
Checking for incompatible polymorphic functions               ok
Checking for not-null constraint inconsistencies              ok
Creating dump of global objects                               ok
Creating dump of database schemas
                                                              ok
Checking for presence of required libraries                   ok
Checking database user is the install user                    ok
Checking for prepared transactions                            ok
Checking for new cluster tablespace directories               ok

If `pg_upgrade` fails after this point, you must re-initdb
the new cluster before continuing.

Performing Upgrade
- -----------------
Setting locale and encoding for new cluster                   ok
Analyzing all rows in the new cluster                         ok
Freezing all rows in the new cluster                          ok
Deleting files from new pg_xact                               ok
Copying old pg_xact to new server                             ok
Setting oldest XID for new cluster                            ok
Setting next transaction ID and epoch for new cluster         ok
Deleting files from new pg_multixact/offsets                  ok
Copying old pg_multixact/offsets to new server                ok
Deleting files from new pg_multixact/members                  ok
Copying old pg_multixact/members to new server                ok
Setting next multixact ID and offset for new cluster          ok
Resetting WAL archives                                        ok
Setting frozenxid and minmxid counters in new cluster         ok
Restoring global objects in the new cluster                   ok
Restoring database schemas in the new cluster
                                                              ok
Copying user relation files
                                                              ok
Setting next OID for new cluster                              ok
Sync data directory to disk                                   ok
```

(continues on next page)

(continued from previous page)

```
Creating script to delete old cluster                         ok
Checking for extension updates                                notice

Your installation contains extensions that should be updated
with the ALTER EXTENSION command.  The file
    update_extensions.sql
when executed by psql by the database superuser will update
these extensions.

Upgrade Complete
- ---------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/edb/as17/bin/vacuumdb -U enterprisedb --all --analyze-in-stages
Running this script will delete the old clusters data files:
    ./delete_old_cluster.sh

pg_upgrade complete, performing BDR post-upgrade steps
- ----------------------------------------------------
Collecting post-upgrade old PG instance control data          ok
Collecting post-upgrade new PG instance control data          ok
Checking LSN of the new PG instance                           ok
Starting new PG instance with BDR disabled                    ok
Connecting to the new PG instance                             ok
Creating replication origin bdr_bdrdb_dc_1_pgd_a2             ok
Advancing replication origin bdr_bdrdb_dc_1_pgd_a2 to 0/3...  ok
Creating replication origin bdr_bdrdb_dc_1_witness_a1         ok
Advancing replication origin bdr_bdrdb_dc_1_witness_a1 to...  ok
Creating replication slot bdr_bdrdb_dc_1                      ok
Creating replication slot bdr_bdrdb_dc_1_witness_a1           ok
Creating replication slot bdr_bdrdb_dc_1_pgd_a2               ok
Stopping new PG instance                                      ok
```

**Note**

You can use the *--link* option for a hard link. This option works if both the data dirs are on the same filesystem. For more information, see pg_upgrade in the PostgreSQL documentation.

### Post-upgrade steps

#### Update the Postgres service file

Update the server version, data directory, and binary directories of the new server in the PostgreSQL service file, located at `/etc/systemd/system/postgres.service` .

An example of what the updated service file looks like:

```
[Unit]
Description=Postgres 17 (TPA)
After=syslog.target
After=network.target
```

(continues on next page)

(continued from previous page)

```
[Service]
Type=simple
User=enterprisedb
Group=enterprisedb
OOMScoreAdjust=-1000
Environment=PG_OOM_ADJUST_VALUE=0
Environment=PGDATA=/var/lib/edb/as17/data
StandardOutput=syslog
ExecStart=/usr/edb/as17/bin/edb-postgres -D ${PGDATA} -c config_file=/var/lib/edb/
↪as17/data/postgresql.conf
ExecStartPost=+/bin/bash -c echo 0xff > /proc/$MAINPID/coredump_filter
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
Restart=no
LimitCORE=infinity

[Install]
WantedBy=multi-user.target
```

### Start the postgres service

Execute a daemon-reload and start the Postgres service:

```
systemctl daemon-reload
systemctl start postgres
```

> **Note**
>
> If your server was not running as a service, you can skip the service file update and start the server using the pg_ctl utility.

### Verify the upgraded cluster versions

Use the following command to verify the upgraded cluster versions:

```
pgd-a1:/home/rocky $ /usr/edb/as17/bin/pgd nodes list --versions
```

Output:

```
 Node Name  | BDR Version          | Postgres Version
- -----------+----------------------+------------------
 pgd-a1     | PGD {Version 6.4}.0 Essential | 17.6.0
 pgd-a2     | 5.9.0                | 13.22.28
 witness-a1 | 5.9.0                | 13.22.28
```

The BDR version for node **pgd-a1** was upgraded to {Version 6.4}.0 and the Postgres version to 17.6.0.

### Unfence the node

Unfence a node in this cluster with `pgd node <node-name> set-option route_fence false` so that it does not become the write leader.

### Verify the Connection Manager is working

Execute a query through the connection manager port (6444 by default) on the upgraded node:

```
pgd-a1:/home/rocky $ psql "host=pgd-a1 port=6444 dbname=bdrdb user=enterprisedb " -c
↪"select node_name from bdr.local_node_summary;"
```

Output:

```
 node_name
- ----------
 pgd-a2
(1 row)
```

### Clean up and vacuum analyze

As a best practice, run a vacuum over the database at this point. When the upgrade ran, you may have noticed the post-upgrade report included:

```
Upgrade Complete
- ----------------
Optimizer statistics are not transferred by pg_upgrade.
Once you start the new server, consider running:
    /usr/edb/as17/bin/vacuumdb -U enterprisedb --all --analyze-in-stages
Running this script will delete the old clusters data files:
    ./delete_old_cluster.sh
```

You can run the vacuum now. On the target node, run:

```
/usr/edb/as17/bin/vacuumdb -U enterprisedb --all --analyze-in-stages
```

If you're sure you don't need to revert this node, you can also clean up the old cluster's data files:

```
./delete_old_cluster.sh
```

### Upgrade the remaining nodes

You must perform these steps for every node in the cluster. The only difference will be the node name in the upgrade command. For quick reference, the commands for nodes **pgd-a2** and **witness-a1** are provided:

Node **pgd-a2**

```
/usr/edb/as17/bin/pgd node pgd-a2 upgrade \
- -database bdrdb -B /usr/edb/as17/bin \
- -socketdir /tmp \
- -old-bindir /usr/edb/as13/bin \
- -old-datadir /opt/postgres/data \
- -new-datadir /var/lib/edb/as17/data \
- -username enterprisedb \
```

(continues on next page)

(continued from previous page)

```
- -old-port 5444 \
- -new-port 5444
```

Node **witness-a1**

```
/usr/edb/as17/bin/pgd node witness-a1 upgrade \
- -database bdrdb -B /usr/edb/as17/bin \
- -socketdir /tmp \
- -old-bindir /usr/edb/as13/bin \
- -old-datadir /opt/postgres/data \
- -new-datadir /var/lib/edb/as17/data \
- -username enterprisedb \
- -old-port 5444 \
- -new-port 5444
```

### Verify the final state of the cluster

Use the following command to verify the node versions:

```
pgd-a2:/home/rocky $ /usr/edb/as17/bin/pgd nodes list --versions
```

Output:

```
 Node Name   | BDR Version           | Postgres Version
- -----------+-----------------------+------------------
 pgd-a1      | PGD {Version 6.4}.0 Essential   | 17.6.0
 pgd-a2      | PGD {Version 6.4}.0 Essential   | 17.6.0
 witness-a1 | PGD {Version 6.4}.0 Essential   | 17.6.0
```

All nodes of the cluster have been upgraded to PGD {Version 6.4}.0 and EPAS 17.6.0.

### Verify the Connection Manager

For every data node, use the following command to verify the Connection Manager:

```
pgd-a2:/home/rocky $ psql "host=pgd-a2 port=6444 dbname=bdrdb user=enterprisedb " -c
↪"select node_name from bdr.local_node_summary;"
```

Output:

```
 node_name
- ----------
 pgd-a1
(1 row)
```

```
pgd-a2:/home/rocky $ psql "host=pgd-a2 port=6445 dbname=bdrdb user=enterprisedb " -c
↪"select node_name from bdr.local_node_summary;"
```

Output:

```
 node_name
- ----------
```

(continues on next page)

(continued from previous page)

```
 pgd-a2
(1 row)
```

**Note**

This completes the worked example of an in-place major version rolling upgrade (PGD & Postgres) of a 3-node PGD 5, EPAS 13 cluster to PGD 6, EPAS 17 cluster.

## 5.3 Testing and tuning PGD clusters

You can test PGD applications using the following approaches:

### 5.3.1 pgd_bench

The Postgres benchmarking application pgbench was extended in the form of a new application: pgd_bench.

pgd_bench is a regular command-line utility that's added to the PostgreSQL bin directory. The utility is based on the PostgreSQL pgbench tool but supports benchmarking CAMO transactions and PGD-specific workloads.

Functionality of pgd_bench is a superset of pgbench functionality but requires the BDR extension to be installed to work properly.

Key differences include:

- Adjustments to the initialization (`-i` flag) with the standard pgbench scenario to prevent global lock timeouts in certain cases.
- `VACUUM` command in the standard scenario is executed on all nodes.
- pgd_bench releases are tied to the releases of the BDR extension and are built against the corresponding Postgres distribution. This information is reflected in the output of the `--version` flag.

The current version allows you to run failover tests while using CAMO or regular PGD deployments.

The following options were added:

```
- m, --mode=regular|camo|failover
mode in which pgbench should run (default: regular)
```

- Use `-m camo` or `-m failover` to specify the mode for pgd_bench. You can use the `-m failover` specification to test failover in regular PGD deployments.

```
- -retry
retry transactions on failover
```

- Use `--retry` to specify whether to retry transactions when failover happens with `-m failover` mode. This option is enabled by default for `-m camo` mode.

In addition to these options, you must specify the connection information about the peer node for failover in DSNform .

Here's an example in a CAMO environment:

```
pgd_bench -m camo -p $node1_port -h $node1_host bdrdemo \
    "host=$node2_host user=postgres port=$node2_port dbname=bdrdemo"
```

This command runs in CAMO mode. It connects to node1 and runs the tests. If the connection to node1 is lost, then pgd_bench connects to node2. It queries node2 to get the status of in-flight transactions. Aborted and in-flight transactions are retried in CAMO mode.

In failover mode, if you specify `--retry` , then in-flight transactions are retried. In this scenario, there's no way to find the status of in-flight transactions.

### 5.3.2 Notes on pgd_bench usage

- When using custom init-scripts, it's important to understand implications behind the DDL commands. We generally recommend waiting for the secondary nodes to catch up on the data-load steps before proceeding with DDL operations such as `CREATE INDEX` . The latter acquire global locks that can't be acquired until the data load is complete and thus might time out.
- No extra steps are taken to suppress client messages, such as `NOTICE` and `WARNING` messages emitted by PostgreSQL and or any possible extensions, including the BDR extension. It's your responsibility to suppress them by setting appropriate variables, such as `client_min_messages` , `bdr.camo_enable_client_warnings` , and so on.
- pgd_bench doesn't initiate SQL transactions for custom scripts. Scripts that are intended to run in an SQL transaction need to include the transaction start and end commands. If `pgd_bench` is executed with the `-m` /`--mode` option set to `camo` , any custom scripts provided must wrap the SQL commands in a transaction, otherwise CAMO functionality will not work as expected.

#### Performance testing and tuning

PGD allows you to issue write transactions onto multiple nodes. Bringing those writes back together onto each node has a performance cost.

First, replaying changes from another node has a CPU cost and an I/O cost, and it generates WAL records. The resource use is usually less than in the original transaction since CPU overhead is lower as a result of not needing to reexecute SQL. In the case of UPDATE and DELETE transactions, there might be I/O costs on replay if data isn't cached.

Second, replaying changes holds table-level and row-level locks that can produce contention against local workloads. The conflict-free replicated data types (CRDT) and column-level conflict detection (CLCD) features ensure you get the correct answers even for concurrent updates, but they don't remove the normal locking overheads. If you get locking contention, try to avoid conflicting updates, or keep transactions as short as possible. A heavily updated row in a larger transaction causes a bottleneck on performance for that transaction. Complex applications require some thought to maintain scalability.

If you think you're having performance problems, develop performance tests using the benchmarking tools. pgd_bench allows you to write custom test scripts specific to your use case so you can understand the overhead of your SQL and measure the impact of concurrent execution.

If PGD is running slow, then we suggest the following:

1. Write a custom test script for pgd_bench, as close as you can make it to the production system's problem case.
2. Run the script on one node to give you a baseline figure.
3. Run the script on as many nodes as occur in production, using the same number of sessions in total as you did on one node. This technique shows you the effect of moving to multiple nodes.
4. Increase the number of sessions for these two tests so you can plot the effect of increased contention on your application.
5. Make sure your tests are long enough to account for replication delays.
6. Ensure that replication delay isn't growing during your tests.

Use all of the normal Postgres tuning features to improve the speed of critical parts of your application.

## 5.4 Backup and recovery

PGD is designed to be a distributed, highly available system. If one or more nodes of a cluster are lost, the best way to replace them is to clone new nodes directly from the remaining nodes.

The role of backup and recovery in PGD is to provide for disaster recovery (DR), such as in the following situations:

- Loss of all nodes in the cluster
- Significant, uncorrectable data corruption across multiple nodes as a result of data corruption, application error, or security breach

### 5.4.1 Logical backup and restore

You can use pg_dump, sometimes referred to as *logical backup*, normally with PGD.

#### Temporary postgresql.conf settings

First, temporarily set the following settings in `postgresql.conf` :

```
#  Increase from the default of `1GB` to something large, but still a

#  fraction of your disk space since the non-WAL data must also fit.

#  This decreases the frequency of checkpoints.

max_wal_size = 100GB

#  Increase the amount of memory for building indexes. Default is

#  64MB. For example, 1GB assuming 128GB total RAM.

maintenance_work_mem = 1GB

#  Increase the receiver and sender timeout from 1 minute to 1hr to

#  allow large transactions through.

wal_receiver_timeout = 1h
wal_sender_timeout = 1h

#  Increase the number of writers to make better use of parallel

#  apply. Default is 2. Make sure this isnt overriden lower by the

#  node group config num_writers setting.

bdr.writers_per_subscription = 5

#  Increase Raft-related election timeouts with default values of 6s

#  and 3s.

bdr.raft_global_election_timeout = 20s
bdr.raft_group_election_timeout = 10s
```

(continues on next page)

(continued from previous page)

```
#  Increase the size of the shared memory queue used by the receiver to

#  send data to the writer process from the default 1MB.

bdr.writer_input_queue_size = 32MB
```

Additionally:

- Make sure the default bdr.streaming_mode = 'auto' is not overridden so that transactions are streamed.
- Make sure any session or postgresql.conf settings listed above are not overriden by node group-level settings in general.

Now continue with pg_dump and pg_restore.

### pg_dump / pg_restore

In order to reduce the risk of global lock timeouts, we recommend dumping pre-data, data, and post-data separately. For example:

```
pg_dump -h $PG_HOST -p $PG_PORT -U $PG_USER -d $PGD_DB -v --exclude-schema="bdr" --
↪exclude-extension="bdr" --section=pre-data -Fc -f pgd-pre-data.dump
pg_dump -h $PG_HOST -p $PG_PORT -U $PG_USER -d $PGD_DB -v --exclude-schema="bdr" --
↪exclude-extension="bdr" --section=data -Fc -f pgd-data.dump
pg_dump -h $PG_HOST -p $PG_PORT -U $PG_USER -d $PGD_DB -v --exclude-schema="bdr" --
↪exclude-extension="bdr" --section=post-data -Fc -f pgd-post-data.dump
```

And restore by directly executing these SQL files on a node (do not run these on the connection manager port):

```
PGOPTIONS="-cbdr.commit_scope=local" pg_restore -h $PG_HOST -p $PG_PORT -U $PG_USER -
↪d $PGD_DB --section=pre-data -f pgd-pre-data.dump
PGOPTIONS="-cbdr.commit_scope=local" pg_restore -h $PG_HOST -p $PG_PORT -U $PG_USER -
↪d $PGD_DB --section=data -f pgd-data.dump
psql -h $PG_HOST -p $PG_PORT -U $PG_USER -d $PGD_DB -c SELECT bdr.wait_slot_confirm_
↪lsn(NULL, NULL)
PGOPTIONS="-cbdr.commit_scope=local" pg_restore -h $PG_HOST -p $PG_PORT -U $PG_USER -
↪d $PGD_DB --section=post-data -f pgd-post-data.dump
psql -h $PG_HOST -p $PG_PORT -U $PG_USER -d $PGD_DB -c SELECT bdr.wait_slot_confirm_
↪lsn(NULL, NULL)
```

After which point the dump will be restored on all nodes in the cluster.

In contrast if you do not split sections out with a naive pg_dump and pg_restore, the restore will likely fail with a global lock timeout.

If you still get global lock timeouts with pg_restore, add `-cbdr.ddl_locking=off` to `PGOPTIONS` .

If you choose to run pg_restore with `-j` /`--jobs` you will need to increase `max_worker_processes` and `max_parallel_maintenance_workers` by the same amount.

### Prefer restoring to a single node cluster

Especially when initially setting up a cluster from a Postgres dump, we recommend you restore to a cluster with a single PGD node. Then run `pgd node setup` for each node you want in the cluster which will do a physical join that uses `bdr_init_physical` under the hood.

#### Sequences

pg_dump dumps both local and global sequences as if they were local sequences. This behavior is intentional, to allow a PGD schema to be dumped and ported to other PostgreSQL databases. This means that sequence-kind metadata is lost at the time of dump, so a restore effectively resets all sequence kinds to the value of `bdr.default_sequence_kind` at time of restore.

To create a post-restore script to reset the precise sequence kind for each sequence, you might want to use a SQL script like this:

```
SELECT SELECT bdr.alter_sequence_set_kind(||
        nspname||.||relname||,||seqkind||);
FROM bdr.sequences
WHERE seqkind != local;
```

If you run pg_dump using `bdr.crdt_raw_value = on` , then you can reload the dump only with `bdr.crdt_raw_value = on` .

Technical Support recommends the use of physical backup techniques for backup and recovery of PGD.

### 5.4.2 Physical backup and restore

You can take physical backups of a node in an EDB Postgres Distributed cluster using standard PostgreSQL software, such as Barman .

You can perform a physical backup of a PGD node using the same procedure that applies to any PostgreSQL node. A PGD node is just a PostgreSQL node running the BDR extension.

Consider these specific points when applying PostgreSQL backup techniques to PGD:

- PGD operates at the level of a single database, while a physical backup includes all the databases in the instance. Plan your databases to allow them to be easily backed up and restored.
- Backups make a copy of just one node. In the simplest case, every node has a copy of all data, so you need to back up only one node to capture all data. However, the goal of PGD isn't met if the site containing that single copy goes down, so the minimum is at least one node backup per site (with many copies, and so on).
- However, each node might have unreplicated local data, or the definition of replication sets might be complex so that all nodes don't subscribe to all replication sets. In these cases, backup planning must also include plans for how to back up any unreplicated local data and a backup of at least one node that subscribes to each replication set.

#### Restore

While you can take a physical backup with the same procedure as a standard PostgreSQL node, it's slightly more complex to restore the physical backup of a PGD node.

#### EDB Postgres Distributed cluster failure or seeding a new cluster from a backup

The most common use case for restoring a physical backup involves the failure or replacement of all the PGD nodes in a cluster, for instance in the event of a data center failure.

You might also want to perform this procedure to clone the current contents of a EDB Postgres Distributed cluster to seed a QA or development instance.

In that case, you can restore PGD capabilities based on a physical backup of a single PGD node, optionally plus WAL archives:

- If you still have some PGD nodes live and running, fence off the host you restored the PGD node to, so it can't connect to any surviving PGD nodes. This practice ensures that the new node doesn't confuse the existing cluster.

- Restore a single PostgreSQL node from a physical backup of one of the PGD nodes.
- If you have WAL archives associated with the backup, create a suitable `postgresql.conf` , and start PostgreSQL in recovery to replay up to the latest state. You can specify an alternative `recovery_target` here if needed.
- Start the restored node, or promote it to read/write if it was in standby recovery. Keep it fenced from any surviving nodes!
- Clean up any leftover PGD metadata that was included in the physical backup.
- Fully stop and restart the PostgreSQL instance.
- Add further PGD nodes with the standard procedure based on the `bdr.join_node_group()` function call.

### Cleanup of PGD metadata

To clean up leftover PGD metadata:

1. Drop the PGD node using bdr.drop_node .
2. Fully stop and restart PostgreSQL (important!).

### Cleanup of replication origins

You must explicitly remove replication origins with a separate step because they're recorded persistently in a system catalog. They're therefore included in the backup and in the restored instance. They aren't removed automatically when dropping the BDR extension because they aren't explicitly recorded as its dependencies.

To track progress of incoming replication in a crash-safe way, PGD creates one replication origin for each remote master node. Therefore, for each node in the previous cluster run this once:

```
SELECT pg_replication_origin_drop(bdr_dbname_grpname_nodename);
```

You can list replication origins as follows:

```
SELECT * FROM pg_replication_origin;
```

Those created by PGD are easily recognized by their name.

### Cleanup of replication slots

If a physical backup was created with `pg_basebackup` , replication slots are omitted from the backup.

Some other backup methods might preserve replications slots, likely in outdated or invalid states. Once you restore the backup, use these commands to drop all replication slots:

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

If you have a reason to preserve some slots, you can add a `WHERE slot_name LIKE 'bdr%'` clause, but this is rarely useful.

> **Warning**
>
> Never use these commands to drop replication slots on a live PGD node

### 5.4.3 Eventual consistency

The nodes in an EDB Postgres Distributed cluster are *eventually consistent* but not *entirely consistent*. A physical backup of a given node provides point-in-time recovery capabilities limited to the states actually assumed by that node.

The following example shows how two nodes in the same EDB Postgres Distributed cluster might not (and usually don't) go through the same sequence of states.

Consider a cluster with two nodes, `N1` and `N2` , that's initially in state `S` . If transaction `W1` is applied to node `N1` , and at the same time a non-conflicting transaction `W2` is applied to node `N2` , then node `N1` goes through the following states:

```
(N1)   S  -->  S + W1  -->  S + W1 + W2
```

Node `N2` goes through the following states:

```
(N2)   S  -->  S + W2  -->  S + W1 + W2
```

That is, node `N1` *never* assumes state `S + W2` , and node `N2` likewise never assumes state `S + W1` . However, both nodes end up in the same state `S + W1 + W2` . Considering this situation might affect how you decide on your backup strategy.

#### Point-in-time recovery (PITR)

The previous example showed that the changes are also inconsistent in time. `W1` and `W2` both occur at time `T1` , but the change `W1` isn't applied to `N2` until `T2` .

PostgreSQL PITR is designed around the assumption of changes arriving from a single master in COMMIT order. Thus, PITR is possible by scanning through changes until one particular point in time (PIT) is reached. With this scheme, you can restore one node to a single PIT from its viewpoint, for example, `T1` . However, that state doesn't include other data from other nodes that committed near that time but had not yet arrived on the node. As a result, the recovery might be considered to be partially inconsistent, or at least consistent for only one replication origin.

With PostgreSQL PITR, you can use the standard syntax:

```
recovery_target_time = T1
```

PGD allows for changes from multiple masters, all recorded in the WAL log for one node, separately identified using replication origin identifiers.

PGD allows PITR of all or some replication origins to a specific point in time, providing a fully consistent viewpoint across all subsets of nodes.

Thus for multi-origins, you can view the WAL stream as containing multiple streams all mixed up into one larger stream. There's still just one PIT, but that's reached as different points for each origin separately.

The WAL stream is read until requested origins have found their PIT. All changes are applied up until that point, except that any transaction records aren't marked as committed for an origin after the PIT on that origin is reached.

You end up with one LSN "stopping point" in WAL, but you also have one single timestamp applied consistently, just as you do with single-origin PITR.

Once you reach the defined PIT, a later one might also be set to allow the recovery to continue, as needed.

After the desired stopping point is reached, if the recovered server will be promoted, shut it down first. Move the LSN forward to an LSN value higher than used on any timeline on this server using `pg_resetwal` . This approach ensures that there are no duplicate LSNs produced by logical decoding.

In the specific example shown, `N1` is restored to `T1` . It also includes changes from other nodes that were committed by `T1` , even though they weren't applied on `N1` until later.

To request multi-origin PITR, use the standard syntax in the `postgresql.conf` file:

```
recovery_target_time = T1
```

You need to specify the list of replication origins that are restored to `T1` in one of two ways. You can use a separate `multi_recovery.conf` file by way of a new parameter, `recovery_target_origins` :

```
recovery_target_origins = *
```

Or you can specify the origin subset as a list in `recovery_target_origins` :

```
recovery_target_origins = 1,3
```

The local WAL activity recovery to the specified `recovery_target_time` is always performed implicitly. For origins that aren't specified in `recovery_target_origins` , recovery can stop at any point, depending on when the target for the list mentioned in `recovery_target_origins` is achieved.

In the absence of the `multi_recovery.conf` file, the recovery defaults to the original PostgreSQL PITR behavior that's designed around the assumption of changes arriving from a single master in COMMIT order.

**Note**

This feature is available only with EDB Postgres Extended. Barman doesn't create a *multi_recovery.conf* file.

### 5.4.4 Monitoring

Use the following queries to check on the progress of the restore process.

```
SELECT pg_size_pretty(pg_database_size(bdrdb));
```

The above query shows the database size on the restoring node. The size should grow as the restore makes progress and approaches the size of the original node. However, due to bloat, logical restores are always a little smaller than the original.

```
SELECT * FROM bdr.node_replication_rates;
```

The above query shows the rate of replication. However, the progress info can be misleading for big transactions; lag and progress will appear to stair-step.

```
SELECT
  application_name,
  state,
  wait_event_type,
  wait_event,
  now() - state_change AS state_change_ago
FROM
  pg_stat_activity
WHERE
  application_name LIKE %pg_restore%;
```

The above query shows information on what pg_restore is doing, if it's blocked/waiting (on what is waiting) or working, and changing its status continuously.

Check the following views to see issues with replication slots, accumulated lag, broken replication, etc.

- pg_catalog.pg_stat_replication_slots

- pg_catalog.pg_replication_slots
- bdr.node_slots

And use `bdr.stat_subscription` to see statistics for each subscription, for example to check on parallel apply or transaction stream.

## 5.5 Migrating to Postgres Distributed (PGD)

Moving from a traditional architecture to EDB Postgres Distributed (PGD) allows your organization to achieve high availability, geographically distributed workloads, and multi-master write capabilities.

The following guide details the tested and supported pathways for transitioning your existing infrastructure into a highly available PGD cluster.

- Migrate from Postgres Physical Streaming Replication (PSR) to PGD: Transition a standard primary-standby architecture to a PGD cluster using a seed-node approach to minimize downtime during data transfer and version upgrades.

### 5.5.1 Migrating from Postgres Physical Streaming Replication (PSR) to PGD

Moving from Postgres Physical Streaming Replication (PSR) to Postgres Distributed (PGD) represents a shift from a single-primary architecture to a multi-primary mesh network. This transition is typically driven by the need for higher availability and geographically distributed write capabilities.

See Choosing between PSR and PGD for more information.

The core of this migration strategy is the **seed node**, a new node that initially joins your existing PSR cluster as a standard physical standby. Once synchronized, it is decoupled from the physical stream and converted into a logical subscriber.

Logical replication allows for data to flow between different major versions of Postgres. Once the seed node is upgraded and running PGD, it serves as the "anchor" for the new cluster, allowing additional PGD nodes to join using a "physical join" from the seed node while still receiving fresh data from the original source primary.

By using this method, the original primary node remains the "source of truth" throughout the process, ensuring that the application can continue to run without interruption until the final cutover phase.

#### Prerequisites

- Nodes running PGE, EPAS, or Community Postgres version 14 or later currently utilizing physical replication (optionally managed by EFM).
- Two additional nodes with sufficient storage (the seed node and an additional node).
- Network connectivity between all existing and new nodes.

#### Migration overview

The migration consists of the following phases:

1. *Preparing your environment* Identify the source primary node and select a seed node to anchor the PGD cluster.
2. *Transferring data to the seed node* Establish a manual physical standby on the new node.
3. *Converting to logical replication* Transition the seed node from physical to logical streaming.
4. *Upgrading the seed node and installing PGD* Perform an in-place major version upgrade and install the PGD packages.
5. *Initializing PGD* Create the BDR extension and initialize the first PGD node group.

6. *Joining additional PGD nodes* Expand the cluster to ensure high availability.
7. *Application cutover* Switch traffic to the new PGD cluster.
8. *Performing cleanup* Decouple the clusters and remove legacy versions.

### Preparing your environment

Before beginning the migration from Physical Streaming Replication (PSR) to Postgres Distributed (PGD), you must identify the key roles of your nodes and establish your target topology. This ensures the environment can handle the temporary overhead of the migration and provides a clear path for the cluster transition.

### Choosing a source node

You must choose a source node from your existing cluster to provide the initial data and subsequent updates for the PGD cluster. Because logical replication cannot be cascaded from a standby node, only the current primary node is a viable source.

Consider the following:

- **Resource overhead:** During migration, the source node node must perform additional logical decoding for at least two PGD nodes, requiring extra CPU capacity.
- **Storage:** The source node will need to retain additional WAL data to keep replicas in sync, necessitating more disk space than standard operations.
- **Optimization:** If the current primary is underpowered, consider a switchover (promotion) to a more favorable node before starting the migration.
- **Failover handling:** If the primary node fails during the migration, you must abort and restart the process using a new primary as the source.

### Choosing a seed node

The seed node is a new node selected from the group that will form your PGD cluster. This node serves a unique transitional role during the migration:

1. It receives a full copy of the database from the source node.
2. It performs the in-place major version upgrade.
3. It initializes the final PGD cluster.

### Planning your PGD cluster topology

Establish your target topology in advance. Define locations, hostnames, PGD node names, and connection strings (DSNs) before proceeding. This guide assumes a two-node PGD start: the seed node (first node) and one additional node. Adding further nodes follows the same process as the second node.

Use the following table to map the required environment variables for your cluster, ensuring consistency across all configuration scripts.

| Variable | Description | Example |
|---|---|---|
| *${SEED_NODE_PGD_N* | The PGD-specific name assigned to the seed node. | node-1 |
| *${SEED_NODE_DSN}* | Connection string to reach the seed node. | host=node-1 port=5444 dbname=pgddb user=postgres |
| *${ADD_NODE_PGD_N* | The PGD-specific name for the second node. | node-2 |

continues on next page

Table 11 – continued from previous page

| Variable | Description | Example |
|---|---|---|
| *${ADD_NODE_DSN}* | Connection string to reach the additional node. | host=node-2 port=5432 dbname=pgddb user=postgres |
| *${PGD_CLUSTER_NAME}* | The name of the PGD node group containing all nodes. | pgd_cluster_main |

### Command environment setup

The migration steps specify which node to use for each command.

To ensure the migration commands execute correctly, verify your environment meets the following requirements:

- **Shell permissions:** Execute all commands as `root` unless the instructions explicitly specify using `su` .
- **Environment variables:** Ensure `${PGDATA}` is set to your Postgres data directory (e.g., `/var/lib/edb/as14/data` for EPAS 14). You may also need to set `${PGPORT}` .
- **Target database:** Connect to the specific database to migrate (not the default `postgres` or `edb` databases) by setting `${PGDATABASE}` before running `psql` .

Next step: *Transferring data to the seed node* .

### Transferring data to the seed node

In this phase, you initialize the seed node by creating a standby copy of the database on the seed node that mirrors the source node via physical replication. This standby serves as the seed node for the eventual PGD cluster. To minimize interference with the existing cluster—particularly if it is managed by Failover Manager (EFM)—this replica is added manually.

### Preparing the source node for logical replication

You must prepare the seed node to transition to logical replication. If not already enabled, set the `wal_level` to `logical` on the source node. Since this change requires a database restart, it is best to perform this step ahead of time.

1. Run the following command on the source node:

```
ALTER SYSTEM SET wal_level = logical;
```

1. Restart the source node Postgres service for the configuration change to take effect:

```
su -u enterprisedb --command "pg_ctl restart"
```

1. After the restart, verify that the `wal_level` is correct and that there are sufficient replication slots and worker processes available. For a setup with **M** existing standbys and **N** future PGD nodes, ensure these parameters are at least what is indicated below:

```
SHOW wal_level;                          -- Expected logical

SHOW max_replication_slots;              -- Must be at least M + N
SHOW max_wal_senders;                    -- Must be at least M + N
SHOW max_logical_replication_workers;    -- Must be at least N
```

### Allowing outbound logical replication

Configure the source environment to permit external logical connections and establish the necessary authentication credentials for the migration stream.

1. The source node requires a role with replication privileges to facilitate data migration. You can use an existing role or create a separate role, for example:

```
CREATE ROLE repl LOGIN NOSUPERUSER REPLICATION PASSWORD your_password_here;
```

EDB recommends using a `.pgpass` file to manage credentials securely without including passwords in connection strings (DSNs).

1. Verify connectivity from the PGD nodes using the following command to ensure they can connect without interactive password entry:

```
su -u enterprisedb -c "psql --no-password ${SOURCE_DSN} replication=1 \
--command IDENTIFY SYSTEM"
```

Ensure you test connectivity between the seed node and all future PGD nodes; this may require updating `pg_hba.conf` on the source node.

### Preparing the seed node

Install the same distribution and version of Postgres on the seed node to accommodate a physical backup. See Installing EPAS , Installing PGE , or *Installing Postgres* for details.

> **Note**
>
> Don't install EFM on the seed node, as it must be managed manually during the migration.

### Creating a physical backup

Once the source is configured, execute the following command from the seed node to pull a physical snapshot of the database from the source node. Ensure the destination `${PGDATA}` directory does not exist before starting the transfer.

```
su -u enterprisedb -c "pg_basebackup ${SOURCE_DSN}     \
  --pgdata=$(PGDATA)                                    \
  --write-recovery-conf                                 \
  --wal-method=stream                                   \
  --create-slot                                         \
  --slot=migration_phy_slot                             \
  --progress                                            \
  --verbose"
```

This command uses a spread checkpoint by default to minimize the I/O impact on the source node. For faster transfers at the cost of higher disk I/O, you can append `--checkpoint=fast` .

The transfer duration depends on the total volume of data and the network bandwidth between the source and seed nodes. For environments with limited bandwidth, you can enable compression using the `–compress-level= *0-9]* flag to optimize the transfer speed; refer to the official [pg_basebackup documentation <https://www.postgresql.org/docs/14/app-pgbasebackup.html>`_ for detailed configuration options.

### Configuring and starting the seed node

1. Once the transfer is complete, adjust the seed node's `postgresql.conf` and `pg_hba.conf` to reflect its local environment (hostnames, IP addresses, and paths). Validate that the replication settings on the seed node match the requirements for PGD:

```
SHOW wal_level;                          -- Expected: logical
SHOW max_replication_slots;              -- Should be at least M + N
SHOW max_wal_senders;                    -- Should be at least M + N
SHOW max_logical_replication_workers;    -- Should be at least N
```

1. Start the database on the seed node to initiate streaming:

```
su -u enterprisedb --command "pg_ctl start"
```

### Verifying physical replication

Monitor the replication status from both the source and seed nodes to ensure they are synchronized.

On the source node:

1. Check the migration slot status. It must show the slot assigned to the seed node with a low or decreasing `lag_size`:

```
SELECT slot_name, active, restart_lsn, confirmed_flush_lsn,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS lag_size
FROM pg_replication_slots
WHERE slot_name = migration_phy_slot;
```

1. Verify active connections. Confirm that the seed node's IP address appears in the replication statistics:

```
SELECT client_addr, state, sent_lsn, write_lsn, flush_lsn, replay_lsn,
       pg_size_pretty(pg_wal_lsn_diff(sent_lsn, replay_lsn)) AS replay_lag
FROM pg_stat_replication;
```

On the seed node:

1. Confirm the recovery mode. The node must be correctly acting as a standby:

```
SELECT pg_is_in_recovery();              -- Must return true
```

1. Verify the WAL receiver status. Check the health of the connection to the primary (source node):

```
SELECT slot_name, sender_host || : || sender_port AS sender, status,
now() - last_msg_send_time    AS last_msg_send_age,
now() - last_msg_receipt_time AS last_msg_receipt_age,
now() - latest_end_time       AS last_end_age,
pg_size_pretty(pg_wal_lsn_diff(latest_end_lsn, written_lsn)) AS write_lag_size,
pg_size_pretty(pg_wal_lsn_diff(latest_end_lsn, flushed_lsn)) AS flush_lag_size
FROM pg_stat_wal_receiver;
```

1. Compare receive vs. replay position. This identifies the replay lag _ the amount of data received but not yet applied to the database:

```
SELECT pg_size_pretty(pg_wal_lsn_diff(pg_last_wal_receive_lsn(),
                                      pg_last_wal_replay_lsn())) AS lag_bytes;
```

1. Monitor lag by time. Measure the time difference between the current time and the last replayed transition:

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

Next step: *Converting to logical replication* .

### Converting to logical replication

To facilitate a major version upgrade of your Postgres distribution and the subsequent installation of Postgres Distributed (PGD), you must convert the replication stream to the seed node to logical replication. For a seamless transition without data loss, logical replication must resume at the exact Log Sequence Number (LSN) within the WAL stream of the source node where physical replication was interrupted. This is known as the switch-over LSN.

### Creating a publication on the source node

Prepare the source node by defining which data will be replicated and creating a dedicated slot to hold WAL files.

1. Create a publication for all tables in the database:

```
CREATE PUBLICATION migration_seed_pub FOR ALL TABLES;
```

1. Verify the publications:

```
SELECT * FROM pg_publication;
SELECT * FROM pg_publication_tables;
```

1. Create a logical replication slot:

```
SELECT pg_create_logical_replication_slot(migration_node_${SEED_NODE_NAME}, pgoutput);
```

Creating the slot at this stage ensures that the source node begins retaining the necessary WAL data before the switch-over operation. Because the source node will eventually support multiple PGD nodes, each slot is named specifically to identify the target node it serves.

### Promoting the seed node

Promoting the seed node stops the incoming physical replication stream and converts the node into a standalone, writable instance.

> **Warning**
>
> To maintain data integrity, you must prevent any application writes to the seed node at this stage. Applications should continue to write exclusively to the source node.

On the seed node, run the following command:

```
su -u enterprisedb --command "pg_ctl promote"
```

### Enabling logical replication to the seed node

After promotion, you must align the new logical replication slot with the point where physical replication ended.

1. On the seed node, identify the last LSN replayed from the physical stream:

```
SELECT pg_last_wal_replay_lsn();
```

1. On the source node, manually move the logical slot forward to that LSN. Replace `${SWITCH_OVER_LSN}` with the value retrieved from the step above.

```
SELECT pg_replication_slot_advance(migration_node_${SEED_NODE_NAME}, ${SWITCH_OVER_
↪LSN});
```

1. Create a logical subscription on the seed node:

```
CREATE SUBSCRIPTION migration_seed_sub
        CONNECTION ${SOURCE_DSN}
        PUBLICATION migration_seed_pub
            WITH (
                enabled = false,
                copy_data = false,
                create_slot = false,
                slot_name = migration_node_${SEED_NODE_NAME}
                );
```

1. Verify that the subscription has correctly identified the tables for replication:

```
SELECT n.nspname AS schemaname,
    c.relname AS tablename,
    sr.srsubstate AS state,
    s.subname AS subscription_name
FROM pg_subscription_rel sr
JOIN pg_class c ON sr.srrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
JOIN pg_subscription s ON sr.srsubid = s.oid
WHERE s.subname = migration_seed_sub
ORDER BY n.nspname, c.relname;
```

1. Enable logical replication:

```
ALTER SUBSCRIPTION migration_seed_sub ENABLE;
```

### Cleaning up physical replication

1. Once the logical stream is verified and active, run the following command on the source node to remove the legacy physical replication components and free up resources:

```
SELECT pg_drop_replication_slot(migration_phy_slot);
```

1. On the seed node, run the following commands to remove any remaining configuration parameters for physical replication:

```
ALTER SYSTEM RESET primary_conninfo;
ALTER SYSTEM RESET primary_slot_name;
```

Next step: *Upgrading the seed node and installing PGD* .

### Upgrading the seed node and installing PGD

In this phase, you will perform a major version upgrade of your Postgres distribution on the seed node and prepare it for Postgres Distributed (PGD).

### Preparing the seed node for the upgrade

You must take the seed node offline to perform the major version upgrade. We highly recommend to disable the logical subscription before stopping the service, as subscription metadata must be manually refreshed after the upgrade.

1. On the seed node, disable logical replication:

```
ALTER SUBSCRIPTION migration_seed_sub DISABLE;
```

1. Stop the existing Postgres instance on seed node:

```
su -u enterprisedb --command "pg_ctl stop"
```

1. Install the packages for the target version on the seed node. Ensure the old version remains installed, as `pg_upgrade` requires binaries from both versions to perform the transition. See Installing EPAS , Installing PGE , or *Installing Postgres* for details.
2. Adjust your `${PATH}` environment variable to point to the new binaries (e.g., `/usr/edb/as17/bin/` ) and verify the version:

```
which pg_ctl
```

### Performing the in-place upgrade

Use the `--link` flag with `pg_upgrade` for an in-place upgrade that uses hard links instead of copying data files. This significantly reduces the required disk space and time, though it requires both the old (`${OLD_PGDATA}` ) and new (`${NEW_PGDATA}` ) directories to reside on the same filesystem.

1. On the seed node, prepare the new data directory:

```
OLD_PGDATA = ${PGDATA}
unset PGDATA
mkdir ${NEW_PGDATA}
chown enterprisedb: ${NEW_PGDATA}
chmod 700 ${NEW_PGDATA}
su enterprisedb -c "initdb -D ${NEW_PGDATA} -E utf-8"
```

1. On seed node, run `pg_upgrade` in check mode first:

```
mkdir /tmp/pg_upgrade
cd /tmp/pg_upgrade
su postgres -c "pg_upgrade \
--old-datadir ${OLD_PGDATA} \
--new-datadir ${NEW_PGDATA} \
--old-bindir /usr/edb/as14/bin/ \
--new-bindir /usr/edb/as17/bin/ \
--link \
--check"
```

1. If the check returns a positive result, perform the actual upgrade by executing the same command without the `--check` flag:

```
cd /tmp/pg_upgrade
su postgres -c "pg_upgrade \
--old-datadir ${OLD_PGDATA} \
--new-datadir ${NEW_PGDATA} \
--old-bindir /usr/edb/as14/bin/ \
--new-bindir /usr/edb/as17/bin/ \
--link"
```

1. Once completed, start the new version of your Postgres distribution:

```
su -u enterprisedb --command "pg_ctl start"
```

### Re-enabling the subscription

Major version upgrades do not carry over the internal table-mapping information for subscriptions; specifically, the catalog table `pg_subscription_rel` will be empty. Enabling the subscription immediately would cause the replication slot to be moved forward without synchronizing table data, resulting in data loss.

To avoid this, you must "bogus-configure" the subscription to prevent it from advancing the real slot while you refresh the table metadata. You can safely ignore errors being reported in the seed node logs about the slot not existing.

1. On the seed node, point to a bogus slot to safely refresh metadata:

```
ALTER SUBSCRIPTION migration_seed_sub SET (slot_name = bogus);
```

1. Enable and refresh the publication:

```
ALTER SUBSCRIPTION migration_seed_sub ENABLE;
ALTER SUBSCRIPTION migration_seed_sub REFRESH PUBLICATION WITH (copy_data = false);
ALTER SUBSCRIPTION migration_seed_sub DISABLE;
```

1. Verify that the table list in the subscription matches the configuration you obtained on *Enabling logical replication to the seed node* :

```
SELECT n.nspname AS schemaname,
    c.relname AS tablename,
    sr.srsubstate AS state,
    s.subname AS subscription_name
FROM pg_subscription_rel sr
JOIN pg_class c ON sr.srrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
JOIN pg_subscription s ON sr.srsubid = s.oid
WHERE s.subname = migration_seed_sub
ORDER BY n.nspname, c.relname;
```

1. Revert the subscription to the correct slot and resume replication at the right LSN:

```
ALTER SUBSCRIPTION migration_seed_sub
    SET (slot_name = migration_node_${SEED_NODE_NAME});
ALTER SUBSCRIPTION migration_seed_sub ENABLE;
```

At this stage, the seed node is running the new Postgres version and receiving logical updates from the source primary. Allow the system to settle and ensure the seed node has fully caught up before moving to initialize PGD.

Next step: *Initializing PGD* .

### Initializing PGD

Once the seed node is upgraded to the target Postgres version, you must install the Bi-Directional Replication (BDR) extension and initialize the cluster to transition the node into a distributed environment.

### Installing and configuring PGD

The seed node requires PGD packages that specifically match your Postgres distribution and version (e.g., `edb-pgd6-essential-epas17` for PGD Essential on EPAS 17). Ensure you have configured the appropriate PGD repository and PGD edition you want to install (Essential or Expanded) before proceeding. See Installing the database and pgd for details.

1. To enable the BDR extension, update your Postgres configuration to preload the library and enable commit timestamps on the seed node:

```
cat >> ${PGDATA}/postgresql.conf <<EOF
shared_preload_libraries = bdr
track_commit_timestamp = on
EOF
```

1. Restart Postgres to load the BDR extension:

```
su -u enterprisedb --command "pg_ctl restart"
```

### Initialize the PGD cluster

With the extension loaded, you can now create the BDR extension within the database you intend to replicate.

1. On the seed node, create the BDR extension:

```
CREATE EXTENSION bdr;
```

1. Validate BDR is loaded and check the version:

```
SELECT bdr.bdr_version();
```

1. Define the seed node as a BDR node and establish the initial node group that will form your cluster:

```
SELECT bdr.create_node(
    node_name := ${SEED_NODE_PGD_NAME},
    dsn := ${SEED_NODE_DSN_DSN}
);
```

1. Create the node group:

```
SELECT bdr.create_node_group(
    node_group_name := ${PGD_CLUSTER_NAME}
);
```

### Configuring PGD CLI

Use the PGD Command Line Interface (CLI) to manage and monitor the cluster. You must create a configuration file on **all** current and future PGD nodes that lists the connection endpoints for the entire cluster:

```
mkdir -p /etc/edb/pgd-cli
cat <<EOF > tee /etc/edb/pgd-cli/pgd-cli-config.yml
cluster:
  name: ${PGD_CLUSTER_NAME}
  endpoints:
    - ${SEED_NODE_DSN}
    - ${ADD_NODE_DSN}
      - ...
EOF
```

### Validating the PGD Installation

Use the PGD CLI to verify that the single-node setup is healthy and the CLI is correctly configured to communicate with the cluster. On the seed node, run:

```
pgd nodes list -- Expected to list exactly one node (itself).
pgd cluster show --health -- Expected to show OK for all checks.
```

Next step: *Joining additional PGD nodes* .

### Joining additional PGD nodes

In this phase, you expand the PGD cluster beyond a single node to enable high availability features and establish redundancy before the final application cutover. Since BDR does not automatically forward data arriving from external logical replication streams, the source node must feed each PGD node individually through its own dedicated logical replication stream. This architecture ensures data consistency across the entire cluster during the migration.

This procedure involves a fast data transfer through a physical copy that uses `pg_basebackup` under the hood, called a **physical join**. This process is automated through the PGD CLI.

### Installing the PGD packages

Additional nodes require PGD packages that match your Postgres distribution and version (e.g., `edb-pgd6-essential-epas17` for PGD Essential on EPAS 17). Ensure you have configured the appropriate PGD repository and PGD edition you want to install (Essential or Expanded) before proceeding. See Installing the database and pgd for details.

1. To enable the BDR extension, update your Postgres configuration to preload the library and enable commit timestamps on the seed node:

```
cat >> ${PGDATA}/postgresql.conf <<EOF
shared_preload_libraries = bdr
track_commit_timestamp = on
EOF
```

1. Restart Postgres to load the BDR extension:

```
su -u enterprisedb --command "pg_ctl restart"
```

### Creating additional replication slots on the source node

To add a new node, you must create a dedicated replication slot on the source node. To align the new node's data, use the seed node as a reference point and temporarily pause its replication to establish a consistent switch-over LSN.

1. On the source node, create an additional replication slot for the additional PGD node:

```
SELECT pg_create_logical_replication_slot(migration_node_${ADD_NODE_PGD_NAME},
    pgoutput);
```

1. On the seed node, pause logical replication from the source node:

```
ALTER SUBSCRIPTION migration_seed_sub DISABLE;
```

1. While replication is paused, align the new slot with the seed node's LSN to mark the exact switch-over point. On the source node, run the following to advance the slot

```
SELECT pg_replication_slot_advance(migration_node_${ADD_NODE_PGD_NAME},
    (SELECT confirmed_flush_lsn
        FROM pg_replication_slots
        WHERE slot_name = migration_node_${SEED_NODE_PGD_NAME}));
```

1. On the source node, verify that neither slot is advancing (lag should increase while LSN remains static):

```
SELECT slot_name, active, restart_lsn, confirmed_flush_lsn,
    pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS lag_size
FROM pg_replication_slots;
```

1. On the source node, check active connections from standbys. It mustn't show entries for the seed node nor the additional node:

```
SELECT client_addr, state, sent_lsn, write_lsn, flush_lsn, replay_lsn,
    pg_size_pretty(pg_wal_lsn_diff(sent_lsn, replay_lsn)) AS replay_lag
FROM pg_stat_replication;
```

### Joining the additional PGD node

PGD CLI automates the data transfer process through a physical join, which uses `pg_basebackup` internally to clone the seed node onto the new node.

1. On the additional PGD node, run:

```
su postgres -c pgd node ${ADD_NODE_PGD_NAME} setup       \
    --listen-addr="*"                                     \
    --initial-node-count=3                                \
    --pgdata=${PGDATA}                                    \
    --group-name ${PGD_CLUSTER_NAME}                      \
    --cluster-dsn "${SEED_NODE_DSN}"                      \
    --dsn "${ADD_NODE_DSN}"                               \
    --verbose
```

1. Once complete, verify that the new node is `ACTIVE` and visible from the seed node:

```
SELECT * FROM bdr.node_summary;
```

**Warning**

The PGD cluster is now active and will replicate all changes, including most DDL operations. Disable DDL replication (*SET bdr.ddl_replication = off* ) before performing node-specific operations to prevent unintended propagation. You must not apply any DML changes directly to any PGD node until the application switches over.

### Resuming replication from the source node

With both nodes synchronized to the same LSN, you can resume the logical data flow.

1. On the seed node, re-enable logical replication from the source node:

```
SET bdr.ddl_replication = off;
ALTER SUBSCRIPTION migration_seed_sub ENABLE;
```

1. On the additional node, the subscription is intentionally not copied over during a physical join. You must create it manually and point it to the dedicated slot on the source node:

```
SET bdr.ddl_replication = off;
CREATE SUBSCRIPTION migration_add_sub
    CONNECTION ${SOURCE_NODE_DSN}
    PUBLICATION migration_seed_pub    -- same publication, can be used multiple times
    WITH (
        enabled = false,
        copy_data = false,
        create_slot = false,
        slot_name = migration_node_${ADD_NODE_PGD_NAME}
    );
```

1. On the additional node, verify that the table list in the subscription matches your configuration you obtained on *Enabling logical replication to the seed node* :

```
SELECT n.nspname AS schemaname,
    c.relname AS tablename,
    sr.srsubstate AS state,
    s.subname AS subscription_name
FROM pg_subscription_rel sr
JOIN pg_class c ON sr.srrelid = c.oid
JOIN pg_namespace n ON c.relnamespace = n.oid
JOIN pg_subscription s ON sr.srsubid = s.oid
WHERE s.subname = migration_seed_sub
ORDER BY n.nspname, c.relname;
```

1. On the additional node, enable subscription to resume replication from the source node:

```
SET bdr.ddl_replication = off;
ALTER SUBSCRIPTION migration_seed_sub ENABLE;
```

### Verifying all replication streams

Ensure all nodes are receiving data from the source and are communicating within the PGD cluster. Note that data does not flow back from PGD to the source node.

1. On the source node, verify non-PGD logical subscription lag by running the following query:

```
SELECT slot_name, active, confirmed_flush_lsn, pg_size_pretty(pg_wal_lsn_diff(pg_
↪current_wal_lsn(), confirmed_flush_lsn)) as lag FROM pg_replication_slots;
```

Expect one entry per PGD node, and the lag to be minimal.

1. On each PGD node, verify the status of PGD mesh and replication. Expect very little traffic due to idle state of the PGD cluster, but no lag:

```
SELECT * FROM bdr.node_summary;
SELECT * FROM bdr.subscription_summary;
SELECT * FROM bdr.node_replication_rates;
```

Next step: *Application cutover* .

### Application cutover

With the PGD cluster fully synchronized, this phase details the final transition from your Postgres Streaming Replication (PSR) environment to the Postgres Distributed (PGD) cluster.

### Stopping application writes

To prevent data divergence, stop all write traffic to the legacy PSR cluster at the application level. Ensure no new transactions are being initiated on the source node.

### Performing lag verification

Ensure that the PGD cluster has ingested all outstanding changes from the source node before shifting traffic. Run the following command on the seed node to ensure that replication lag on the migration subscription is zero:

```
DO $$
DECLARE
   wait_lsn pg_lsn := pg_current_wal_lsn();
   current_flush_lsn pg_lsn;
BEGIN
  LOOP
    current_flush_lsn := (SELECT MAX(flush_lsn) FROM pg_stat_replication WHERE␣
↪application_name IN (migration_seed_sub, migration_add_sub));
    IF wait_lsn <= current_flush_lsn THEN
      EXIT;
    END IF;
    RAISE NOTICE Waiting for replication to catch up, current lag: %,  pg_wal_lsn_
↪diff(wait_lsn, current_flush_lsn);
    PERFORM pg_sleep(1);
  END LOOP;
END;$$;
```

### Disabling subscriptions

Once the lag is zero, disable the logical subscriptions on all PGD nodes that connect back to the original PSR source. On all PGD nodes, run:

```
SET bdr.ddl_replication = off;
ALTER SUBSCRIPTION migration_seed_sub DISABLE;
```

### Switching the application traffic

With the PGD cluster confirmed as fully up-to-date and isolated from the old source, you can now redirect traffic and resume operations.

Update the connection strings to point your application to the Connection Manager . The specific method (e.g., updating DNS, Kubernetes ConfigMaps, or `.env` files) depends on your specific application architecture.

Resume application traffic. The Connection Manager will now route requests to the correct PGD nodes.

### Performing validation

After the application is live on the PGD cluster, perform final verification checks to ensure the PGD mesh is healthy, receiving data from the application, and propagating across all nodes in the PGD cluster. This involves re-verifying the replication streams to confirm that writes are propagating across all nodes.

1. Confirm that all nodes are in an 'active' state and that internal subscriptions are streaming correctly. On any PGD node, run:

```
SELECT * FROM bdr.node_summary;
SELECT * FROM bdr.subscription_summary;
```

1. Check the replication rates to ensure the cluster is processing the new load without building up internal lag. On any PGD node, run:

```
SELECT * FROM bdr.node_replication_rates;
```

Next step: *Performing cleanup* .

### Performing cleanup

In this final phase you decommission the legacy replication infrastructure.

### Decoupling the source replication

After the application has been successfully switched to the PGD cluster, remove the migration links to fully decouple the environments.

To confirm that data is no longer flowing through the source node, you must disable the logical subscriptions on all PGD nodes that connect back to the original PSR source. On all PGD nodes, run:

```
DROP SUBSCRIPTION `subscription_name`;
```

Where `subscription_name` is the name of the subscription on each node.

### Cleaning up the seed node

After the migration is finalized and the cluster is stable, the seed node no longer requires the legacy version of Postgres. You can now remove the corresponding outdated packages from the node to free up resources and simplify future maintenance.

CHAPTER 6

# Configuration and Management

## 6.1 Node types and capabilities

A PGD cluster can contain several different types of node, each with its own role. This section describes the different types of node that can be configured in a PGD cluster.

- ```
  :ref:`Overview of durability options <Overview of durability options>`  is an␣
  ↪overview the kinds of node that can exist in PGD clusters and their associated␣
  ↪roles.
  ```

- ```
  :ref:`Witness nodes <Witness nodes>`  looks at the witness node, a special class␣
  ↪of PGD node, dedicated to establishing consensus in a group.
  ```

- ```
  :ref:`Logical standby nodes <Logical standby nodes>`  shows how to efficiently␣
  ↪keep a node on standby synchronized and ready to step in as a primary in the␣
  ↪case of failure.
  ```

- ```
  :ref:`Subscriber-only nodes and groups <Subscriber-only nodes and groups>`  looks␣
  ↪at how subscriber-only nodes work with subscriber-only groups, how they boost␣
  ↪read scalability and the different options for configuring them.
  ```

### 6.1.1 An overview of PGD Node types

#### Data nodes

A data node in PGD is a node that runs a Postgres instance. It replicates data to all other data nodes. It also participates in the cluster-wide Raft decision-making around locking and leadership. It can be a member of one or more groups and is, by default, a member of the "top level" group that spans all data nodes in the cluster.

The data node is also the foundation on which the other three nodes are built.

#### Witness nodes

A witness node behaves like a data node in that it participates in the cluster-wide Raft decision-making around locking and leadership. It doesn't replicate or store data, though. The purpose of a witness node is to be available to ensure that the cluster can achieve a majority it seeks a consensus. *Witness nodes* has more details.

#### Logical standby nodes

Logical standby nodes are nodes that receive the logical data changes from another node and replicate them locally. PGD can use a logical standby node to replace the node it's replicating if that node becomes unavailable, with some caveats. See *Logical standby nodes* for more details.

#### Subscriber-only nodes

A subscriber-only node is a data node that, as the name suggests, only subscribes to changes in the cluster but doesn't replicate changes to other nodes. You can use subscriber-only nodes as read-only nodes for applications. You create subscriber-only nodes by specifying a data node is `subscriber-only` when you create the node and then adding it to a subscriber-only group. See *Subscriber-only nodes and groups* for more details.

### 6.1.2 Witness nodes

A witness node is a lightweight node that functions as a data node but that doesn't store or replicate data. Use a witness node to allow a PGD cluster that uses Raft consensus to have an odd number of voting nodes and therefore be able to achieve a majority when making decisions.

#### Witness nodes within PGD groups or regions

One typical use of witness nodes is when a PGD group has two data nodes but resources aren't available for the recommended three data nodes. In this case, you can add a witness node to the PGD group to provide a third voting node to local Raft decision-making. These decisions are primarily about who will be electing a write leader for the proxies to use. With only two nodes, it's possible to have no consensus over which data node is write leader. With two data nodes and a witness, there are two candidates (the data nodes) and three voters (the data nodes and the witness). When a data node is down, then, there are still two voters that can select a write leader.

#### Witness node outside regions

At a higher level, you can use witness nodes when multiple PGD groups are mapped to different regions. For example, with three data nodes per region in two regions, while running normally, all six data nodes can participate in Raft decisions and obtain DDL and DML global locks. Even when a data node is down, there are sufficient data nodes to obtain a consensus. But if a network partition occurs and connectivity with the other region is lost, then now only three nodes out of six are available, which isn't enough for a consensus. To avoid this scenario, you can deploy a witness node in a third region as part of the PGD cluster. This witness node will allow a consensus to be achieved for most operational requirements of the PGD cluster while a region is unavailable.

### 6.1.3 Logical standby nodes

PGD allows you to create a *logical standby node*, also known as an offload node, a read-only node, receive-only node, or logical-read replicas. A master node can have zero, one, or more logical standby nodes.

> **Note**
>
> Logical standby nodes can be used in environments where network traffic between data centers is a concern. Otherwise, having more data nodes per location is always preferred.

Logical standby nodes are nodes that are held in a state of continual recovery, constantly updating until they're required. This behavior is similar to how Postgres physical standbys operate, while using logical replication for better performance. Logical standby nodes receive changes but don't send changes made locally to other nodes.

A logical standby is created by specifying the `node_kind` as `standby` when creating the node with bdr.create_node .

Later, if you want, use bdr.promote_node

to move the logical standby into a full, normal send/receive node.

A logical standby is sent data by one source node, defined by the DSN in bdr.join_node_group . Changes from all other nodes are received from this one source node, minimizing bandwidth between multiple sites.

For high availability, if the source node dies, one logical standby can be promoted to a full node and replace the source in a failover operation similar to single-master operation. If there are multiple logical standby nodes, the other nodes can't follow the new master, so the effectiveness of this technique is limited to one logical standby.

In case a new standby is created from an existing PGD node, the needed replication slots for operation aren't synced to the new standby until at least 16 MB of LSN has elapsed since the group slot was last advanced. In extreme cases, this might require a full 16 MB before slots are synced or created on the streaming replica. If a failover or switchover occurs during this interval, the streaming standby can't be promoted to replace its PGD node, as the group slot and other dependent slots don't exist yet.

The slot sync-up process on the standby solves this by invoking a function on the upstream. This function moves the group slot in the entire EDB Postgres Distributed cluster by performing WAL switches and requesting all PGD peer nodes to replay their progress updates. This behavior causes the group slot to move ahead in a short time span. This reduces the time required by the standby for the initial slot's sync-up, allowing for faster failover to it, if required.

On PostgreSQL, it's important to ensure that the slot's sync-up completes on the standby before promoting it. You can run the following query on the standby in the target database to monitor and ensure that the slots synced up with the upstream. The promotion can go ahead when this query returns `true` .

```
SELECT true FROM pg_catalog.pg_replication_slots WHERE
    slot_type = logical AND confirmed_flush_lsn IS NOT NULL;
```

You can also nudge the slot sync-up process in the entire PGD cluster by manually performing WAL switches and by requesting all PGD peer nodes to replay their progress updates. This activity causes the group slot to move ahead in a short time and also hastens the slot sync-up activity on the standby. You can run the following queries on any PGD peer node in the target database for this:

```
SELECT bdr.run_on_all_nodes(SELECT pg_catalog.pg_switch_wal());
SELECT bdr.run_on_all_nodes(SELECT bdr.request_replay_progress_update());
```

Use the monitoring query on the standby to check that these queries do help in faster slot sync-up on that standby.

A logical standby does allow write transactions. You can use this to great benefit, since it allows the logical standby to have additional indexes, longer retention periods for data, intermediate work tables, LISTEN/NOTIFY, temp tables, materialized views, and other differences.

Any changes made locally to logical standbys that commit before the promotion aren't sent to other nodes. All transactions that commit after promotion are sent onwards. If you perform writes to a logical standby, take care to quiesce the database before promotion.

You might make DDL changes to logical standby nodes, but they aren't replicated and they don't attempt to take global DDL locks. PGD functions that act similarly to DDL also aren't replicated. See *Running DDL safely* . If you made incompatible DDL changes to a logical standby, then the database is a *divergent node*. Promotion of a divergent node currently results in replication failing. As a result, plan to either ensure that a logical standby node is kept free of divergent changes if you intend to use it as a standby, or ensure that divergent nodes are never promoted.

### 6.1.4 Subscriber-only nodes and groups

Subscriber-only nodes and groups offer a powerful way to build read scaling into your PGD cluster.

- The *Overview of durability options* introduces how subscriber-only nodes and groups work in PGD.
- *Creating Subscriber-only groups and nodes* explains how to create a subscriber-only group and node.
- *Joining nodes to a Subscriber-only group* explains how to join a node to an existing subscriber-only group which has members.
- *Optimizing subscriber-only groups* provides details on how to configure the PGD subscriber-only optimized topology feature which uses a group leader for more efficient replication.

#### An overview of Subscriber-only nodes

##### Overview

While many use cases rely on accessing a database node which can handle queries and updates, there are also use cases which only require access to a node that can handle read-only database queries. Read scaling like this, by moving the read-only traffic away from active database nodes in the cluster, can improve the performance of the core cluster, whilst making database access more widely available.

##### Subscriber-only nodes

The basic idea of subscriber-only nodes is to provide a read-only node that you can use to offload read-only queries from the main cluster. The default topology of a PGD cluster is what's called a full mesh topology, where every node connects to every other node. This is the most robust and fault-tolerant way to connect nodes, but it can be inefficient for some use cases.

Subscriber-only nodes can be a member of a subscriber-only group or, with PGD 6 and later, they can be part of a data group.

##### Subscriber-only groups

Subscriber-only groups in PGD gather together subscriber-only nodes. Each group can address different regions or different application demands.

Unlike data groups, a subscriber-only group has no raft consensus mechanism of its own. This also means that a subscriber-only group can have as many subscriber-only nodes as your need.

Previous to PGD 6, the existence of a subscriber-only group didn't change the replication topology. All nodes in the subscriber-only group, by default, independently receive replicated changes from all other nodes in the cluster.

#### Optimizing subscriber-only groups

In PGD 6 and later, you can optionally optimize the topology of subscriber-only groups.

For clusters using proxies and raft-enabled groups for their data nodes, subscriber-only groups can use a more efficient model for receiving replicated changes.

The optimized topology option creates a group leader in each subscriber-only group, similar to a write leader in PGD Proxies. The group leader receives all the changes from the cluster and then replicates them to the other nodes in its group. See *Optimizing subscriber-only groups* for more information on this feature.

### Subscriber-only nodes and DDL

Subscriber-only nodes can execute locally issued DDL commands but they don't replicate those changes to other nodes in the cluster and will not attempt to acquire locks on the cluster.

### Creating Subscriber-only groups and nodes

The process of creating a Subscriber-only node or nodes starts with creating a Subscriber-only group to contain the node or nodes. Perform this step on an existing fully joined node in the PGD cluster.

### Creating a Subscriber-only group manually

To create a Subscriber-only group, you must specify the `node_group_type` as `subscriber-only` when creating the group. For example, here we are logged into the node "node-one" running on "host-one". It's a member of it's own data group and as for all nodes, a member of the top-level group, here called `topgroup` . Log into this node directly to create a new Subscriber-only group named `sogroup` with the following SQL command:

```
select bdr.create_node_group(sogroup, topgroup, false, subscriber-only);
```

or more explicitly with parameter names:

```
select bdr.create_node_group(node_group_name:=sogroup,
        parent_group_name:=topgroup,
        join_node_group:=false,
        node_group_type:=subscriber-only);
```

This creates a Subscriber-only group named `sogroup` which is a child of the `topgroup` group. The false parameter for `join_node_group` indicates that the node executing this command shouldn't join to the newly created group. Automatically joining the group is the default behavior, which in this case needs to be supressed.

### Adding a node to a new Subscriber-only group manually

You can now initialize a new data node and then add it to the Subscriber-only group. Create a data node and configure the bdr extension on it as you would for any other data node.

You now have to create this new node as a `subscriber-only` node. To do this, log into the new node and run the following SQL command:

```
select bdr.create_node(so-node-1, host=so-host-1 dbname=pgddb port=5444, subscriber-
↪only);
```

Then, log into that new node and add it to the `sogroup` group with the following SQL command:

```
select bdr.join_node_group(host=host-one dbname=pgddb port=5444,sogroup);
```

or more explicitly with parameter names:

```
select bdr.join_node_group(dsn:=host=host-one dbname=pgddb port=5444,
        node_group_name:=sogroup);
```

This instructs the new node to join the `sogroup` group. As it has no knowledge of the cluster topology, it will connect to the node specified in the DSN to receive the necessary information to join the group. In this example, this happens to be the same node as we used to create the subscriber-only group, but it could be any node that's fully joined to the cluster.

### Joining nodes to a Subscriber-only group

If you have no subscriber-only groups in your PGD cluster, you must create the groups following the process in *Creating Subscriber-only groups and nodes* . After you have created a subscriber-only group, you can join subscriber-only nodes to it.

### Joining a node to an existing subscriber-only group

Unlike joining a node to a *Creating Subscriber-only groups and nodes* , joining a node to an existing subscriber-only group is a simpler process.

First create the new node as a subscriber-only node. Run the following SQL command on the new node:

```
select bdr.create_node(so-node-2, host=so-host-2 dbname=pgddb port=5444, subscriber-
↪only);
```

or more explicitly with parameter names:

```
select bdr.create_node(node_name:=so-node-2,
        dsn:=host=so-host-2 dbname=pgddb port=5444,
        node_type:=subscriber-only);
```

This command creates a new node named `so-node-2` on host `so-host-2` and configures it as a subscriber-only node. The node won't be able to join the cluster until joins a group.

In *Creating Subscriber-only groups and nodes* , you created a group named `sogroup` and added a subscriber-only node called `so-node-1` on a host `shost-1` . It used a node in an existing data group to facilitate that join. But you can't use this new subscriber-only node to add another subscriber-only node. You must use any active data node that's fully joined to the cluster. In the *Creating Subscriber-only groups and nodes* examples, they use `host-one` in the cluster's data group for this task. You can use the following SQL command on `shost-2` to join it to the `sogroup` group:

```
select bdr.join_node_group(host=host-one dbname=pgddb port=5444,sogroup);
```

or more explicitly with parameter names:

```
select bdr.join_node_group(dsn:=host=host-one dbname=pgddb port=5444,
        node_group_name:=sogroup);
```

This command instructs the new node to join the `sogroup` group. As it has no knowledge of the cluster topology, it connects to the node specified in the DSN to receive the necessary information to join the group. That node must be fully joined to the cluster as it acts as the source of the request for the new node to join the group.

Once the new node has joined the group, it starts by first synchronizing and then begins to receive replication changes from the other nodes in the cluster.

> **Note**
>
> Unless, the group is using the *Optimizing subscriber-only groups* , in which case it replicates changes from a subscriber-only group leader in the subscriber-only group it has joined.

### Optimizing subscriber-only groups

With PGD 6 and later, it's possible to optimize the topology of *Overview of durability options* .

In this optimized topology, a small number of fully active nodes _ the write leaders of the data groups _ replicate changes to the group leaders of subscriber-only groups. These group leaders then replicate changes to the other members of its subscriber-only group.

### Requirements for the optimized topology

You can't enable this model if a cluster has any of the following:

- Data nodes that are directly members of the top-level group
- No data-node subgroups
- No data-node subgroups with proxy routing enabled

If any of these are the case, the nodes in subscriber-only groups revert to the full mesh topology.

To get the benefit of the new SO group and node replication, you must have your data nodes in subgroups, with proxy routing enabled on the subgroups.

### How the optimized topology works

For clusters using groups for their data nodes, subscriber-only groups can use a more efficient model. This model uses subscriber-only group leaders, similar to write leaders in PGD proxies.

Each subscriber-only group uses that group leader to replicate changes to other subscriber-only nodes in its group. The group leader acts as a replication proxy for incoming changes.

The write leader nodes in data groups replicate changes to the group leaders of the subscriber-only groups. Other nodes in the data groups only replicate with nodes in their data group and with data nodes in other data groups. They do not directly replicate their changes to the subscriber-only groups.

### Subscriber-only group leaders

With PGD 6 and later, each subscriber-only group gets assigned a group leader of its own. This is because subscriber-only groups don't have a group Raft consensus mechanism of their own. Instead, the cluster's top-level group uses its Raft consensus mechanism to handle selecting each subscriber-only group's group leader. This group leader selection is on by default in PGD 6, regardless of the topology optimization settings.

Group leaders in subscriber-only groups are regularly tested for connectivity and, if unavailable, the voting nodes of top-level group select a new subscriber-only node from the subscriber-only group to become group leader. The new group leader is then selected.

With optimized technology turned off, this election has no effect on the replication topology. Without the optimized topology, all data nodes replicate changes to all other nodes in the cluster.

### Group leaders in the optimized topology

With the optimized topology enabled, only the subscriber-only group's group leader receives changes from other data groups' write leaders in the cluster. The group leader takes on the responsibility of replicating those changes to the other nodes in the subscriber-only group.

The other voting nodes choose the group leader from a subscriber-only group's nodes. Once the group leader is selected, the whole cluster becomes aware of the change, and any data group's write leaders then replicate data only to this newly selected group leader node. Other data nodes in the data groups don't replicate data to the subscriber-only group's nodes.

This approach avoids the explosion of active connections that can happen when there are large numbers of SO nodes and reduces the amount of replication traffic.

The subscriber-only node and group form the building block for PGD tree topologies.

### Enabling the optimized model

By default, PGD 6 forces the full mesh topology. This means the optimization described here is off. To enable the optimized topology, you must have your data nodes in subgroups, with proxy routing enabled on the subgroups. You can then set the GUC bdr.force_full_mesh to `off` to allow the optimization to be activated.

> **Note**
>
> This GUC needs to be set in the *postgresql.conf* file on each data node and each node restarted for the change to take effect.

If any requirements of the optimized topology aren't met, the nodes in a subscriber-only group revert to the full mesh topology. When this happens, you'll find in the logs of the nodes in the cluster messages why the optimization wasn't possible, such as:

When a data node is part of the top-level node group:

```
node: <nodename> is part of top-level nodegroup: <toplevelgroupname>: changing to␣
↪full mesh".
```

When a data group doesn't have proxy routing enabled:

```
node: <nodename> is in nodegroup: <nodegroupname> that does not have proxy routing:␣
↪changing to full mesh.
```

## 6.2 Node management

All data nodes in a PGD cluster are members of one or more groups. By default, all data nodes are members of the top-level group, which spans all data nodes in the PGD cluster. Nodes can also belong to subgroups that can be configured to reflect logical or geographical organization of the PGD cluster.

You can manage nodes and groups using the various options available with nodes and subgroups.

- *Creating PGD nodes* covers the steps needed to create a new node in a PGD cluster.
- *Groups and subgroups* goes into more detail on how groups and subgroups work in PGD.
- *Creating and joining PGD groups* looks at how new PGD groups can be created and how to join PGD nodes to them.
- *Viewing PGD topology* details commands and SQL queries that can show the structure of a PGD cluster's nodes and groups.
- *Removing nodes and groups* shows the process to follow to safely remove a node from a group or a group from a cluster.
- *Node restart and down node recovery* details the steps needed to bring a node back into service after a failure or scheduled downtime and the impact it has on the cluster as it returns.
- *Relationship to automatic synchronization* looks at how the automatic sync feature works in PGD and how it can be used to keep nodes in sync with each other.
- *Node UUIDs* explains how the UUIDs of nodes are used in PGD and how they are generated.
- *Replication slots created by PGD* examines how the Postgres replication slots are consumed when PGD is operating.

### 6.2.1 Creating PGD nodes

#### It's just Postgres

A PGD node is just a Postgres instance with the BDR extension installed. The BDR extension enables bidirectional replication between nodes and is the foundation of PGD.

That means, in the most general terms, you can create a PGD node by installing Postgres and the BDR extension, and then configuring the node to connect to the other nodes in the PGD group. But there are some specifics to consider.

#### Which Postgres version?

PGD is built on top of Postgres, so the distribution and version of Postgres you use for your PGD nodes is important. The version of Postgres you use must be compatible with the version of PGD you are using. You can find the compatibility matrix in the release notes . Features and functionality in PGD may depend on the distribution of Postgres you are using. The EDB Postgres Advanced Server is the recommended distribution for PGD. PGD also supports EDB Postgres Extended Server and Community Postgres .

#### Installing Postgres

You must install your selected Postgres distribution on each node you are configuring. You can find installation instructions for each distribution in the EDB Postgres Advanced Server documentation , EDB Postgres Extended Server documentation , and the *Postgres installation documentation* . You can also refer to the *Installing the database and pgd* which covers the installation of Postgres.

#### Installing the BDR extension

The BDR extension is the key to PGD's distributed architecture. You need to install the BDR extension on each node in your PGD cluster. The BDR extension is available from the EDB Postgres Distributed repository. You need to add the EDB repository to your package management system on Linux and then install the PGD package. You can find the repository configuration instructions in the *Configuring PGD repositories* .

Once the repository is configured, you can install the PGD package with your package manager. The package name is `edb-pgd6-<edition>-pg<postgres_distribution><version>` where `<postgres_distribution>` is the Postgres distribution you are using and `<version>` is the version of Postgres you are using. For example, if you are using community Postgres 14 and want to install PGD Essential, the package name is `edb-pgd6-essential-pg14` . For details on package naming, see *Installing the database and pgd* .

#### Configuring the database for PGD

This process is specific to PGD and involves configuring the Postgres instance to work with the BDR extension and adjusting various settings to work with the PGD cluster. The steps are as follows:

- Add the BDR extension `$libdir/bdr` at the start of the `shared_preload_libraries` setting in `postgresql.conf` .
- Set the `wal_level` GUC variable to `logical` in `postgresql.conf` .
- Turn on commit timestamp tracking by setting `track_commit_timestamp` to `'on'` in `postgresql.conf` .
- Increase the maximum worker processes to 16 or higher by setting `max_worker_processes` to `'16'` in `postgresql.conf` .
- Add rules to `pg_hba.conf` to allow nodes to connect to each other. - Ensure that these lines are present in `pg_hba.conf` :

```
host all all all md5
host replication all all md5
```

- Add a `.pgpass` file to allow nodes to authenticate each other.
- Configure a user with sufficient privileges to log in to the other nodes.
- See

**The Password File in the Postgres documentation for more on the** `.pgpass` file.

Once these steps are complete, restart the Postgres instance to apply the changes.

#### Setting up the replicated database

Create a database within Postgres to use as PGD's replicated database. You can do this with the `CREATE DATABASE` command. The created database should be the name of the database that other nodes in the PGD cluster replicate. The convention is to name the database `pgddb` .

Log into that specific database instance you have configured and set up the BDR extension. You can do this by running the `CREATE EXTENSION bdr;` command as superuser. This command creates the BDR extension within that database.

#### Next steps

The node is now configured and ready to be join a group, or start a group, in the PGD cluster. You can find instructions for joining a node to a group in the *Creating and joining PGD groups* section.

### 6.2.2 Groups and subgroups

#### Groups

A PGD cluster's nodes are gathered in groups. A "top level" group always exists and is the group to which all data nodes belong to automatically. The "top level" group can also be the direct parent of sub-groups.

#### Sub-groups

A group can also contain zero or more subgroups. Subgroups can be used to represent data centers or locations allowing commit scopes to refer to nodes in a particular region as a whole. Connection Manager can also make use of subgroups to delineate nodes available to be write leader.

The `node_group_type` value specifies the type when the subgroup is created. Some sub-group types change the behavior of the nodes within the group. For example, a *Subscriber-only nodes* sub-group will make all the nodes within the group into subscriber-only nodes.

#### Configuring per-subgroup connection strings

In Kubernetes and cloud environments, nodes in the same subgroup often have access to a faster or cheaper local network path that isn't exposed as the node's primary `node_dsn` . Setting `node_group_dsn` on a node tells other nodes in the same subgroup to use that DSN instead. Nodes in other subgroups continue to use `node_dsn` .

Set `node_group_dsn` using *bdr.alter_node_option* :

```
SELECT bdr.alter_node_option(node_eu1, group_dsn, host=foo dbname=bdrdb port=5432);
```

Reset it by passing NULL:

```
SELECT bdr.alter_node_option(node_eu1, group_dsn, NULL);
```

The `node_dsn` endpoint must remain accessible by all nodes for initial setup. See *bdr.node_config_summary* for the currently configured and effective values per node.

### 6.2.3 Creating and joining PGD groups

#### Creating and joining PGD groups

For PGD, every node must connect to every other node. To make configuration easy, when a new node joins, it configures all existing nodes to connect to it. For this reason, every node, including the first PGD node created, must know the PostgreSQL connectionstring

that other nodes can use to connect to it. This connection string is sometimes referred to as a data source name (DSN).

Both formats of connection string are supported. So you can use either key-value format, like `host=myhost port=5432 dbname=mydb` , or URI format, like `postgresql://myhost:5432/mydb` .

The SQL function bdr.create_node_group()

creates the PGD group from the local node. Doing so activates PGD on that node and allows other nodes to join the PGD group, which consists of only one node at that point. At the time of creation, you must specify the connection string for other nodes to use to connect to this node.

When the first PGD node group is created, a default replication set is also created, and all existing tables are added into the set. After that, all tables will have the `bdr.default_replica_identity` value configured as the `REPLICA IDENTITY` for the table.

Once the node group is created, further nodes can be created using

bdr.create_node() , and then joined to the PGD group using the bdr.join_node_group() function.

Alternatively, use the command line utility bdr_init_physical to create a new node, using `pg_basebackup` . If using `pg_basebackup` , the bdr_init_physical utility can optionally specify the base backup of only the target database. The earlier behavior was to back up the entire database cluster. With this utility, the activity completes faster and also uses less space because it excludes unwanted databases. If you specify only the target database, then the excluded databases get cleaned up and removed on the new node.

When a new PGD node is joined to an existing PGD group or a node subscribes to an upstream peer, before replication can begin the system must copy the existing data from the peer nodes to the local node. This copy must be carefully coordinated so that the local and remote data starts out identical. It's not enough to use pg_dump yourself. The BDR extension provides built-in facilities for making this initial copy.

During the join process, the BDR extension synchronizes existing data using the provided source node as the basis and creates all metadata information needed for establishing itself in the mesh topology in the PGD group. If the connection between the source and the new node disconnects during this initial copy, restart the join process from the beginning.

The node that's joining the cluster must not contain any schema or data that already exists on databases in the PGD group. We recommend that the newly joining database be empty except for the BDR extension. However, it's important that all required database users and roles are created. Also, if a non-superuser is performing the joining operation, extensions that require superuser permission must be created manually. For more details, see *Connections and roles* .

Optionally, you can skip the schema synchronization using the `synchronize_structure` parameter of the bdr.join_node_group

function. In this case, the schema must already exist on the newly joining node.

We recommend that you select the source node that has the best connection (logically close, ideally with low latency and high bandwidth) as the source node for joining. Doing so lowers the time needed for the join to finish.

Coordinate the join procedure using the Raft consensus algorithm, which requires most existing nodes to be online and reachable.

The logical join procedure (which uses the bdr.join_node_group

function) performs data sync doing `COPY` operations and uses multiple writers (parallel apply) if those are enabled.

Node join can execute concurrently with other node joins for the majority of the time taken to join. However, only one regular node at a time can be in either of the states PROMOTE or PROMOTING. These states are typically fairly

short if all other nodes are up and running. Otherwise the join is serialized at this stage. The subscriber-only nodes are an exception to this rule, and they can be concurrently in PROMOTE and PROMOTING states as well, so their join process is fully concurrent.

The join process uses only one node as the source, so it can be executed when nodes are down if a majority of nodes are available. This approach can cause a complexity when running logical join. During logical join, the commit timestamp of rows copied from the source node is set to the latest commit timestamp on the source node. Committed changes on nodes that have a commit timestamp earlier than this (because nodes are down or have significant lag) can conflict with changes from other nodes. In this case, the newly joined node can be resolved differently to other nodes, causing a divergence. As a result, we recommend not running a node join when significant replication lag exists between nodes. If this is necessary, run LiveCompare on the newly joined node to correct any data divergence once all nodes are available and caught up.

`pg_dump` can fail when there's concurrent DDL activity on the source node because of cache-lookup failures. Since bdr.join_node_group uses pg_dump internally, it might fail if there's concurrent DDL activity on the source node. Retrying the join works in that case.

## 6.2.4 Viewing PGD topology

### Listing PGD groups

### Using pgd-cli

Use the pgd-cli `groups list` command to list all groups in the PGD cluster:

```
pgd groups list
```

```
Group Name  Parent Group Name Group Type     Nodes
----------- ----------------- --------------- -----
bdrgroup    bdrgroup          global          0
group_a     bdrgroup          data            4
group_b     bdrgroup          data            4
group_c     bdrgroup          data            1
group_so    bdrgroup          subscriber-only 1
```

### Using SQL

The following simple query lists all the PGD node groups of which the current node is a member. It currently returns only one row from bdr.local_node_summary .

```
SELECT node_group_name
FROM bdr.local_node_summary;
```

You can display the configuration of each node group using a more complex query:

```
SELECT g.node_group_name
, ns.pub_repsets
, ns.sub_repsets
, g.node_group_default_repset     AS default_repset
, node_group_check_constraints    AS check_constraints
FROM bdr.local_node_summary ns
JOIN bdr.node_group g USING (node_group_name);
```

#### Listing nodes in a PGD group

#### Using pgd-cli

Use the `nodes list` command to list all nodes in the PGD cluster:

```
pgd nodes list
```

```
Node Name           Group Name Node Kind       Join State Node Status
------------------- ---------- --------------- ---------- -----------
bdr-a1              group_a    data            ACTIVE     Up
bdr-a2              group_a    data            ACTIVE     Up
logical-standby-a1  group_a    standby         ACTIVE     Up
witness-a           group_a    witness         ACTIVE     Up
bdr-b1              group_b    data            ACTIVE     Up
bdr-b2              group_b    data            ACTIVE     Up
logical-standby-b1  group_b    standby         ACTIVE     Up
witness-b           group_b    witness         ACTIVE     Up
witness-c           group_c    witness         ACTIVE     Up
subscriber-only-c1  group_so   subscriber-only ACTIVE     Up
```

Use `grep` with the group name to filter the list to a specific group:

```
pgd nodes list | grep group_b
```

```
bdr-b1             group_b  data      ACTIVE     Up
bdr-b2             group_b  data      ACTIVE     Up
logical-standby-b1 group_b  standby   ACTIVE     Up
witness-b          group_b  witness   ACTIVE     Up
```

#### Using SQL

You can extract the list of all nodes in a given node group (such as `mygroup` ) from the bdr.node_summary ` view. For example:

```
SELECT node_name         AS name
, node_seq_id            AS ord
, peer_state_name        AS current_state
, peer_target_state_name AS target_state
, interface_connstr      AS dsn
FROM bdr.node_summary
WHERE node_group_name = mygroup;
```

### 6.2.5 Removing nodes and groups

#### Removing a node from a PGD group

Since PGD is designed to recover from extended node outages, you must explicitly tell the system if you're removing a node permanently. If you permanently shut down a node and don't tell the other nodes, then performance suffers and eventually the whole system stops working.

Node removal, also called *parting*, is done using the bdr.part_node()

function. You must specify the node name (as passed during node creation) to remove a node. You can call the bdr.part_node() function from any active node in the PGD group, including the node that you're removing.

Just like the join procedure, parting is done using Raft consensus and requires a majority of nodes to be online to work.

The parting process affects all nodes. The Raft leader manages a vote between nodes to see which node has the most recent data from the parting node. Then all remaining nodes make a secondary, temporary connection to the most recent node to allow them to catch up any missing data.

A parted node still is known to PGD but doesn't consume resources. A node might be added again under the same name as a parted node. In rare cases, you might want to clear all metadata of a parted node by using the function bdr.drop_node() .

#### Removing a whole PGD group

PGD groups usually map to locations. When a location is no longer being deployed, it's likely that the PGD group for the location also needs to be removed.

The PGD group that's being removed must be empty. Before you can remove the group, you must part all the nodes in the group.

### 6.2.6 Node restart and down node recovery

PGD is designed to recover from node restart or node disconnection. The disconnected node rejoins the group by reconnecting to each peer node and then replicating any missing data from that node.

When a node starts up, each connection begins showing up in bdr.node_slots with `bdr.node_slots.state = catchup` and begins replicating missing data. Catching up continues for a period of time that depends on the amount of missing data from each peer node and will likely increase over time, depending on the server workload.

If the amount of write activity on each node isn't uniform, the catchup period from nodes with more data can take significantly longer than other nodes. Eventually, the slot state changes to `bdr.node_slots.state = streaming` .

Nodes that are offline for longer periods, such as hours or days, can begin to cause resource issues for various reasons. Don't plan on extended outages without understanding the following issues.

Each node retains change information (using one replication slot

for each peer node) so it can later replay changes to a temporarily unreachable node. If a peer node remains offline indefinitely, this accumulated change information eventually causes the node to run out of storage space for PostgreSQL transaction logs (*WAL* in `pg_wal` ), and likely causes the database server to shut down with an error similar to this:

```
PANIC: could not write to file "pg_wal/xlogtemp.559": No space left on device
```

Or, it might report other out-of-disk related symptoms.

In addition, slots for offline nodes also hold back the catalog xmin, preventing vacuuming of catalog tables.

On EDB Postgres Extended Server and EDB Postgres Advanced Server, offline nodes also hold back freezing of data to prevent losing conflict-resolution data (see *bdr_read_all_conflicts* ).

Administrators must monitor for node outages (see *Monitoring the Connection Manager* ) and make sure nodes have enough free disk space. If the workload is predictable, you might be able to calculate how much space is used over time, allowing a prediction of the maximum time a node can be down before critical issues arise.

Don't manually remove replication slots created by PGD. If you do, the cluster becomes damaged and the node that was using the slot must be parted from the cluster, as described in *Replication slots created by PGD* .

While a node is offline, the other nodes might not yet have received the same set of data from the offline node, so this might appear as a slight divergence across nodes. The parting process corrects this imbalance across nodes.

During a phase of parting called part catchup, a node is selected that is furthest ahead from all other nodes with respect to the offline node. If other nodes are not equally caught up with respect to this furthest node, a sync is started with the

furthest-ahead node as source, offline node as origin and each of the nodes that are not equally caught up as targets. A sync is essentially a subscription on the target node to the source node (furthest ahead node), which forwards changes from the offline node (origin) to the target node.

Depending on how far behind other nodes are, this sync may take some time during parting. Once the sync is complete and all nodes equally caught up, parting moves on to part the node. Without this sync, if a forced part is done, the state of the cluster may not be consistent. This means data can diverge. The automatic sync feature ensures that when a node goes offline, this is detected and all nodes are equally caught up with respect to this offline node by a sync process. This ensures that we do not have to wait until node parting to ensure data consistency.

When a node is down for an extended period or lost permanently, the recommended strategy is to part the failing node and rejoin a new one in its place. The new node can use the same name as the original.

### 6.2.7 Automatic synchronization

#### Auto-triggering the Sync

The BDR manager process does the auto-triggering of sync requests. When there are no updates from a node for an interval of time greater than *bdr.raft_global_election_timeout* (default 6s), it is considered to be down.

Nodes are checked for their closeness to each other. If all nodes are equally caught up, no sync is needed. If not, the node that is furthest ahead from the "down" node is chosen as a source. Once a source is determined, for each target - nodes other than the origin and source - a sync request is set up. Witness and standby nodes do not need to be targets in the sync.

The view bdr.sync_node_requests_summary tracks the sync requests.

- `Origin` : origin node is the down node.
- `Source` : source node is the node furthest ahead from origin.
- `Target` : each of the other nodes that's behind the source with respect to the origin.
- `Sync_start_lsn` : Highest LSN received by the target from origin when sync started.
- `Sync_end_lsn` : Target LSN of the target node from the origin when the sync ended.
- `Sync_status` : status of the sync.
- `Sync_start_ts` : Time when sync started.

Once a sync request is entered in the catalog, it is carried forward to completion.

#### Cancellation

If the source node chosen is found to be down, the manager will cancel the sync operation. This is because some other node can be up which if not furthest, is at least further ahead than some targets. And it may be used to sync the nodes. Therefore the manager will cancel all sync operations which have the down node as source, and will choose another node that is not down as the source for sync. The state machine is described below for a successful sync as well as a cancelled sync.

The sync cancellation API, bdr.sync_node_cancel() is meant only to be used manually and only if the sync request gets stuck for any reason and is blocking normal functioning of the cluster.

```
select bdr.sync_node_cancel(origin, source)
```

This cancels all sync node requests for all targets that have the given origin and source. This can be invoked only from a write lead.

### Sync Request Life Cycle

A single sync request has an origin, source, target and a sync_end_lsn to reach. The sync request goes through various states and each state executes on a different node.

The states are as follows: **setup**: Executes on the write lead. It sets up the fields of the sync, except `sync_end_lsn` . **setup_source**: Executes on the source. It populates `sync_end_lsn` and creates a slot for the sync subscription. **setup_target**: This executes on the target node. In this state, the original subscription to the origin is disabled. A sync subscription is set up on the target which forwards the origin's changes from the source node to the target. **start**: This executes on the target. It monitors the progress of the target to see if `sync_end_lsn` is reached and if reached moves to synced state. **synced**: subscription has synced to `sync_end_lsn` . In this state the slot is dropped. **complete**: This state executes on the target. In this state, sync subscription is dropped on the target and original subscription is enabled. It then moves to to done state. **done**: This means sync is successful. **cancel start**: This executes on the target node. In this state, the sync subscription is disabled, in preparation for a drop later, and the original subscription to the origin is re-enabled. **cancel continue**: This executes on the target node. In this state, the sync subscription is dropped. **cancel done**: This executes on the source node. In this state, the slot is dropped. **failed**: A sync ends-up in this state if a cancellation happens and all cleanup is done. This means the sync could not happen and needs to be retried.

A cancellation of sync can also happen automatically if the chosen source node is found to be down. During cancellation the subscription and slot needs to be cleaned up, and the original subscription enabled. A sync request can be stalled if the source or target nodes are down.

### GUC

The GUC that controls automatic sync is bdr.enable_auto_sync_reconcile and it is set to true by default. To turn it off, it needs to be set to false on all nodes and the server restarted.

## 6.2.8 Node UUIDs

In PGD 6, each node now has a UUID that is used to identify the node in the cluster. This UUID is generated when the node is created and is unique to that node. The UUID can be found in various places in PGD, including:

- The `bdr.node` table, which contains information about each node in the cluster.
- The `bdr.node_summary` view, which provides a human-readable view of the nodes in the cluster.
- The `bdr.local_node` table, which contains information about the local node.
- The uuid values also appear in the naming of the replication slots that are created for each node.

Although used throughour PGD's node management, the use of UUIDs doesn't affect any existing functionality or features in PGD. The UUIDs are used internally to identify nodes and groups and don't change the way that users interact with PGD.

### Why UUIDs?

UUIDs are used in PGD to provide a unique identifier for each node in the cluster. Previous versions of PGD used the node name as an identifier, which could lead to conflicts if two nodes had the same name. By using UUIDs, PGD can ensure that each node has a unique identifier that will not change over time. This is especially important in a distributed system like PGD where nodes may be added or removed from the cluster frequently. The UUID ensures that although a new node may have the same name as an existing node, it has a different UUID and doesn't conflict with the existing node.

### How are UUIDs generated?

When a new node is created, a UUID is generated for that node. This UUID is created using the kernel's strong random number generator and guaranteed to be uniformly random. This guarantee ensures that the UUID is unique and can't be easily guessed. The generated UUID is then stored in the `bdr.node` table and is used to identify the node in the cluster.

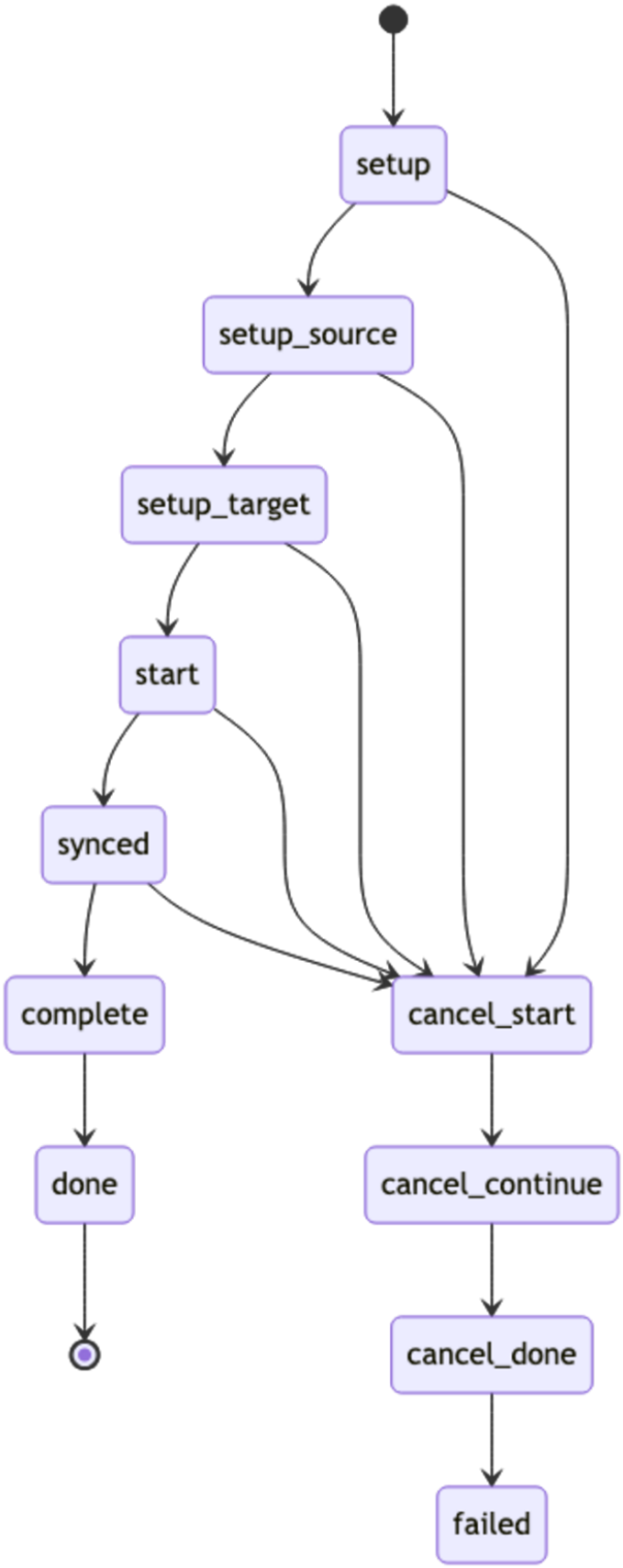


Fig. 1: Autosync Request Life Cycle

#### What happens if a node is removed and a replacement added?

If a node is removed from the cluster and a replacement node is added, the replacement node is assigned a new UUID. This ensures that the replacement node is treated as a separate entity in the cluster and doesn't conflict with the existing nodes. But PGD requires that the old node be fully parted from the cluster before it accepts the new node. The UUID of the replacement node is then used in the same way as the UUIDs of the other nodes in the cluster.

#### UUID-related changes in PGD 6

- The `generation` field in the `bdr.node` table, which was previously used to differentiate between nodes, is no longer used. It remains at 0 for all nodes.
- The `node_uuid` field in the `bdr.node` table is never null in PGD 6. It may be null in the future with a mixed version cluster.

### 6.2.9 Replication slots created by PGD

In previous versions of PGD, replication slots had human-readable names. PGD 6 has switched over to using UUIDs for nodes and groups to ensure better identification.

Replication slots are used by PostgreSQL to track the progress of replication. They're used to ensure that the data being replicated isn't lost and that the replication process is consistent. In PGD, replication slots are used to track the progress of replication from that node. There is one slot per downstream node. There's also a special replication slot used for tracking replication progress from a given node globally across all downstream nodes:

- One group slot, named `bdr_<topgroupuuid>_<dbhash>`
- N-1 node slots named `bdr_node_<targetnodeuuid>_<dbhash>` , where N is the total number of nodes in the cluster, including direct logical standbys, if any

Where `topgroupuuid` is the string representation of the top level-group's UUID (less the - characters) and `dbhash` is a hash of the database name. You can obtain the UUID of the top-level group using:

```
select node_group_uuid from bdr.node_group where node_group_parent_id=0;
```

And `dbhash` is a hash of the database name. You can obtain the hash using:

```
select to_hex(hashtext(pgddb));
```

And the `targetnodeuuid` is the string representation of the target node's UUID (less the - characters). You can obtain the UUID of the target node using:

```
select node_uuid from bdr.node where node_name=<target_node_name>;
```

The complete group slot name is returned by the function bdr.local_group_slot_name() .

> **Warning**
>
> Don't drop those slots. PGD creates and manages them and drops them when or if necessary.

- Avoid touching slots prefixed with `bdr_` slots directly.
- Don't start slot names with the prefix `bdr_` .

The group slot is used to track the progress of replication of the nodes in a PGD cluster that are replicating from the node. Each node in a PGD cluster has its own group slot, which is used to track the progress of replication from that node.

The group slot is used to:

- Join new nodes to the PGD group without having all existing nodes up and running (although the majority of nodes should be up). This process doesn't incur data loss in case the node that was down during join starts replicating again.
- Part nodes from the cluster consistently, even if some nodes haven't caught up fully with the parted node.
- Hold back the freeze point to avoid missing some conflicts.

The group slot is usually inactive and is fast forwarded only periodically in response to Raft progress messages from other nodes.

**Warning**

Don't drop the group slot. Although usually inactive, it's still vital to the proper operation of the EDB Postgres Distributed cluster. If you drop it, then some or all of PGD's features can stop working or have incorrect outcomes.

#### Other slot names

Other functionality within PGD makes use of replication slots.

For example, when a node is added to a group, a slot is created for that node to track its progress in the replication process.

This slot is named `bdr_node_<targetnodeuuid>_<dbhash>_tmp` .

There are also slots created for the analytics and decoding features of PGD. These slots have the following names.

| Slot type | Slot name |
|---|---|
| Forwarding slot, leader-to-leader slot | *bdr_node_<targetnodeuuid>_<originidhex>_<dbhash>* |
| Analytics slot | *bdr_analytics_<groupuuid>_<dbhash>* |
| Decoding slot | *bdr_decoder_<topgroupuuid>_<dbhash>* |

## 6.3 Connection Manager

PGD 6.0 introduces a new Connection Manager which replaces the PGD 5's proxy solution with a tightly integrated approach using a background worker to expose read-write, read-only and http-status network interfaces in PGD.

- *Overview of durability options* covers the features and benefits of Connection Manager.
- *Connection Manager Authentication* covers how authentication works with Connection Manager.
- *Connection pooling* covers the built-in connection pooling modes and how to configure them.
- *Load Balancing with Connection Manager* covers how to use load balancing with Connection Manager.
- *Configuring Connection Manager* details the configuration options available and how to set them.
- *Monitoring the Connection Manager* covers the tables and HTTP endpoints available for monitoring.

### 6.3.1 Connection Manager overview

#### About Connection Manager

Connection Manager is a new background worker for EDB Postgres Distributed (PGD) 6.0 that simplifies the process of connection to PGD clusters by providing a single point of entry for client applications. It replaces the PGD 5.x proxy

solution with a tightly integrated approach that exposes read-write, read-only, and HTTP status network interfaces in PGD.

Connection Manager is fully integrated into PGD and is designed to work seamlessly with the existing PGD architecture. Every PGD data node has a Connection Manager instance that listens for incoming connections and routes them to the appropriate node in the cluster, specifically the current write leader in the cluster. It also provides a read-only interface for applications that only need to read data from the cluster.

Starting in PGD 6.3.0, Connection Manager starts by default on subscriber-only nodes, ensuring that even nodes dedicated primarily to receiving replicated data have access to advanced connection pooling and session management features.

Connection Manager consolidates three capabilities that previously required separate tools. It routes writes to the current leader, distributes read traffic across nodes, and pools backend connections. Because it runs inside PGD, authentication is configured once in `pg_hba.conf` and applies to both the proxy layer and the database, with no separate user list to maintain.

### Using Connection Manager

Connection Manager follows the Postgres server's configuration by default. There are three ports, the read-write port, the read-only port, and the HTTP port. The read-write port is used for write operations, while the read-only port is used for read operations. The HTTP port is used for monitoring and management purposes.

The read-write port is, by default, set to the Postgres port + 1000 (usually 6432). The read-only port is set to the Postgres port + 1001 (usually 6433). The HTTP port is set to the Postgres port + 1002 (usually 6434).

To use Connection Manager, configure your client applications to connect to its read-write or read-only port on the data node. Connection Manager routes the connection to the appropriate node in the cluster.

Connecting to the read-only port restricts the connection to read-only operations, similar to SET TRANSACTION READ ONLY , except that it's not possible to change it to read-write. The transaction_read_only configuration parameter correctly reports `on` in these connections.

### Authentication

Connection Manager performs TLS termination and pre-authentication using the same `pg_hba.conf` configuration as Postgres, with no separate user list to maintain. See *Connection Manager Authentication* for supported methods and constraints.

### Connection pooling

Connection Manager includes built-in connection pooling, removing the need for a separate pooler such as pgBouncer in most deployments. Pooling reduces the number of backend Postgres connections by sharing them across many client connections, lowering memory and process overhead on the database server. See *Connection pooling* for reference details, *Configuring connection pooling* for DBA setup guidance, and *Using connection pooling in your application* for developer guidance.

### Load balancing

Connection Manager isn't a replacement for a load balancer. Use an external load balancer to distribute client traffic across Connection Manager instances running on different nodes. See *Load Balancing with Connection Manager* for configuration details and HAProxy examples.

### Configuration

Connection Manager is configured through PGD group options, including ports, connection limits, consensus timeouts, and pool mode. See *Configuring Connection Manager* for the full option reference.

### Monitoring

Connection Manager exposes connection counts and pool utilization through catalog views and HTTP endpoints. See *Monitoring the Connection Manager* for available metrics.

## 6.3.2 Connection Manager Authentication

Connection Manager's authentication is configured through Postgres's own pg_hba.conf file. Connection Manager uses the same authentication methods as Postgres.

### Connection Manager connection types

Connection Manager supports the following connection types in `pg_hba.conf` :

- `host` - TCP/IP connections
- `hostssl` - TCP/IP connections with SSL
- `hostnossl` - TCP/IP connections without SSL

### Connection Manager authentication methods

Connection Manager supports the following authentication methods in `pg_hba.conf` :

- `trust` - No authentication
- `reject` - Reject the connection
- `md5` - MD5 password authentication
- `scram-sha-256` - SCRAM-SHA-256 password authentication
- `cert` - SSL certificate authentication
- `ldap` - Authenticate using an LDAP server

> **Note**
>
> Connection Manager needs to be able to authenticate to the PGD nodes as the client user. Configure the *pg_hba.conf* file on each PGD node to accept connections originating from other PGD nodes for replication and internal communications.

When using a certificate authentication method, the Connection Manager presents its server key. You must configure the PGD node to accept this certificate from the Connection Manager address.

Connection Manager also supports regular expression matching for the `user` and `database` fields in `pg_hba.conf` . This allows you to specify a pattern for matching user and database names, making it easier to manage authentication for multiple users and databases.

Group membership checks are also supported. This allows you to specify a group of users that can connect to the database, rather than specifying each user individually.

Connection Manager supports `pg_ident.conf` for user identity mapping when using the `cert` authentication method.

#### Unsupported pg_hba.conf rules

Where a rule is not supported by Connection Manager, it will be logged as a warning and ignored.

#### LDAP authentication notes and constraints

Connection Manager supports the same LDAP authentication features as PostgreSQL, with the following key constraints:

- Connection Manager does not use the OpenLDAP library for communication with the LDAP server. As a result, standard OpenLDAP configuration files (such as `ldap.conf` or `.ldaprc` ) and environment variables are ignored.
- Connection Manager validates LDAP server certificates by default when using LDAPS or LDAP StartTLS. The necessary CA certificate for the LDAP server's SSL certificate must be in the system trust-store.
- Suppress validation of the LDAP server's SSL certificate by setting `LDAPTLS_REQCERT=never` environment variable. Other values for this variable are ignored.

### 6.3.3 Connection pooling

Configure connection pooling to reuse backend connections across clients, reducing server memory overhead and improving throughput for workloads with many short-lived sessions. Connection Manager's built-in pooling removes the need for external tools such as pgBouncer in most deployments.

See *Configuring connection pooling* for DBA setup and migration guidance, and *Using connection pooling in your application* for developer guidance.

#### Configuring the pool mode

Set the pool mode for a node group using `bdr.alter_node_group_option`:

```
SELECT bdr.alter_node_group_option(mygroup, server_pool_mode, transaction);
```

Or using the PGD CLI:

```
pgd group mygroup set-option server_pool_mode transaction
```

The `server_pool_mode` group option controls the pooling behavior.

| Mode | Behavior |
|---|---|
| *none* | No pooling. Each client connection gets a dedicated backend connection for its entire lifetime. The backend connection closes when the client disconnects. This mode is the default. |
| *session* | A backend connection is assigned to a client on first use and returned to the pool when the client disconnects. Connection Manager runs *Replicated large objects* before returning the connection for reuse, resetting session state. |
| *transaction* | A backend connection is assigned when a transaction begins and returned to the pool when the transaction ends. Between transactions, the client holds no backend connection, making it available to other clients. Connection Manager runs *Replicated large objects* before returning the connection for reuse, resetting session state, then re-applies the client's supported connection parameters for the next transaction. |

The current pool mode is visible in the `server_pool_mode` column of the `bdr.node_group_summary` view.

### Managing session parameters

Connection Manager forwards a specific set of connection parameters to the backend in all pool modes. The parameters it recognizes are `client_encoding` , `DateStyle` , `TimeZone` , `standard_conforming_strings` , `application_name` , `search_path` , and `extra_float_digits` . Additional parameters can be included via the `options` connection parameter using `-c name=value` syntax.

In `transaction` mode, where the backend connection can change between transactions, Connection Manager re-applies these parameters each time it assigns a new backend. Because only connection parameters are re-applied, avoid using `SET` commands to configure session parameters in transaction mode. Changes made with `SET` aren't preserved when the backend connection changes. See *Using connection pooling in your application* for connection string examples.

### Using prepared statements

Prepared statements sent via the extended query protocol work seamlessly across transactions in all pool modes. Connection Manager automatically detects missing prepared statements on the backend and re-prepares them on demand, transparently managing statement naming between the client and the backend.

In `transaction` mode, `DISCARD ALL` runs when the backend connection is returned to the pool at the end of each transaction, which deallocates any prepared statements created with `PREPARE` /`EXECUTE` SQL statements. Issue `PREPARE` /`EXECUTE` SQL statements inside the transaction where you use them, or use the extended query protocol instead. See *Using prepared statements* in the developer guide for specific steps.

### Reusing connections

In `session` and `transaction` modes, Connection Manager returns the backend connection to the pool when a client disconnects or a transaction ends. If a client triggers an unsupported feature or closes the connection with commands still pending, Connection Manager discards that backend connection rather than returning it to the pool. Other clients aren't affected.

### Unsupported features

Some Postgres features are incompatible with connection pooling and aren't supported.

- `SET` changes made during a session don't persist in `transaction` mode. Configure session parameters in the connection string instead. See *Managing session parameters* for details.
- `PREPARE` /`EXECUTE` /`DEALLOCATE` SQL statements aren't supported in `transaction` mode. `DISCARD ALL` runs when the backend connection is returned to the pool, deallocating any prepared statements. Issue these statements inside the transaction where you use them, or use the extended query protocol instead. See *Using prepared statements* for details.
- Holdable cursors (`WITH HOLD` ) aren't supported in `transaction` mode. They persist beyond the transaction boundary, which conflicts with returning the backend connection to the pool when the transaction ends.
- `LISTEN` isn't supported in `transaction` mode. A `LISTEN` subscription persists beyond a single transaction, which conflicts with returning the backend connection to the pool.
- Advisory locks held across transaction boundaries aren't supported in `transaction` mode.
- Temporary tables accessed across transactions aren't supported in `transaction` mode.
- Replication connections are rejected in all pool modes.

### 6.3.4 Load Balancing with Connection Manager

You can use an external load balancer to distribute traffic across multiple Connection Manager instances. Treat the Connection Manager ports as you would with any other Postgres port, but use the read-write ports for write traffic and the read-only ports for read traffic.

Depending on your load balancer, you have a number of options for how to configure it.

#### Connection Manager routing

If you want the Connection Manager to route traffic to the write leader, connect to the Connection Manager's read-write port. The Connection Manager routes the traffic to the write leader of the node group the node is a member of.

#### HAProxy example: Connection Manager routing

With HAProxy , you can use a configuration similar to the following example to load balance between three Connection Manager instances, each running on different hosts and ports. Read-write traffic to haproxy port 5010 is directed to Connection Manager's read-write port (default 6432) on any one of the nodes. Read-only traffic to haproxy port 5011 is directed to Connection Manager's read-only port (default 6433). Port 5011 is used for read-only traffic, which is distributed across the nodes in a round-robin fashion.

```
global
    maxconn 100

defaults
    log     global
    mode    tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen stats
    mode http
    bind 127.0.0.1:7000
    stats enable
    stats uri /

listen read-write
    bind *:5010
    option pgsql-check user postgres
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 host1:6432 maxconn 100 check port 6432
    server node2 host2:6432 maxconn 100 check port 6432
    server node3 host3:6432 maxconn 100 check port 6432

listen read-only
    balance roundrobin
    bind *:5011
    option pgsql-check user postgres
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 host1:6433 maxconn 100 check port 6433
    server node2 host2:6433 maxconn 100 check port 6433
    server node3 host3:6433 maxconn 100 check port 6433
```

### Direct routing

You may want to connect directly to the nodes in your cluster, bypassing Connection Manager. To leverage Connection Manager in this scenario, consult its HTTP endpoints for information about the nodes in your cluster, such as which node is the write leader. Then connect directly to that node. You can use the `/node/is-read-write` endpoint to check if a node is the write leader and the `/node/is-read-only` endpoint to get information about the read-only pool.

The main reason to configure the load balancer like this is if you want a layer 4 tcp proxy that looks more like a traditional Postgres connection, where TLS termination and authentication happen on the server. Connection Manager provides a layer 7 proxy, which handles TLS termination and authentication at the proxy level, that is, within Connection Manager.

The drawback to this approach is that you won't benefit from Connection Manager's features, especially its ability to enforce read-only connections to the read-only pool. Connections will be directed to the non-write-leader nodes rather than the write leader.

See Monitoring Connection Manager for more information on the available endpoints.

### HAProxy example: Direct routing

This example shows a configuration that uses HAProxy to route traffic directly to the nodes based on their health status. The configuration checks the health of the nodes by querying the `/node/is-read-write` and `/node/is-read-only` endpoints and routes traffic accordingly. As in the previous example, this example routes read-write traffic on port 5010 to the node that's the write leader and read-only traffic on port 5011 to the nodes that are read-only.

```
global
    maxconn 100

defaults
    log     global
    mode    tcp
    retries 2
    timeout client 30m
    timeout connect 4s
    timeout server 30m
    timeout check 5s

listen stats
    mode http
    bind 127.0.0.1:7000
    stats enable
    stats uri /

listen read-write
    bind *:5010
    option httpchk GET /node/is-read-write
    http-check expect string true
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 host1:5432 maxconn 100 check port 6434
    server node2 host2:5432 maxconn 100 check port 6434
    server node3 host3:5432 maxconn 100 check port 6434

listen read-only
    balance roundrobin
```

(continues on next page)

(continued from previous page)

```
    bind *:5011
    option httpchk GET /node/is-read-only
    http-check expect string true
    default-server inter 3s fall 3 rise 2 on-marked-down shutdown-sessions
    server node1 host1:5432 maxconn 100 check port 6434
    server node2 host2:5432 maxconn 100 check port 6434
    server node3 host3:5432 maxconn 100 check port 6434
```

This configuration checks the health of the nodes by querying the `/node/is-read-write` and `/node/is-read-only` endpoints. It routes traffic only to nodes that are healthy and available for read-write or read-only operations, respectively.

### 6.3.5 Configuring Connection Manager

#### Configuring Connection Manager

Connection Manager takes its configuration from the PGD Group options for the group the node is a member of.

These can be configured using the pgd group set-option command or the *Group configuration using bdr.alter_node_group_option()* SQL function.

The following options are available for configuring Connection Manager:

| Option | Default | Description |
|---|---|---|
| read_write_port | Postgres's port + 1000 (usually 6432) | which port to listen on for read-write connections |
| read_only_port | read_write_port + 1 (usually 6433) | which port to listen on for read-only connections |
| http_port | read_write_port + 2 (usually 6434) | which HTTP port to listen for REST API calls (for integration purposes) |
| use_https | false | whether HTTP listener should use HTTPS, if enabled, the server certificate is used to TLS |
| server_pool_mode | none | connection pooling mode. Valid values are *none* (no pooling), *session*, and *transaction*. See *Connection pooling* . |
| read_write_max_client_connections | max_connection | maximum read-write client connections allowed, defaults to max_connections |
| read_write_max_server_connections | max_connections | maximum read-write connections that will be opened to server |
| read_only_max_client_connections | max_connections | maximum read-only client connections allowed |
| read_only_max_server_connections | max_connections | maximum read-only connections that will be opened to server |
| read_write_consensus_timeout | 0 (immediate action) | how long to wait on loss of consensus before read-write connections are no longer accepted |
| read_only_consensus_timeout | 0 (immediate action) | how long to wait on loss of consensus before read-only connections are no longer accepted. |

**Note**

To enable HTTPS in Connection Manager, you must set *use_https* option to *true* and configure *ssl_cert_file* and *ssl_key_file* in *postgresql.conf* or via the command line. HTTPS can be active even if global server SSL (*ssl = on*

) is disabled. If either certificate file is missing, Connection Manager defaults to plaintext HTTP regardless of the *use_https* setting.

### 6.3.6 Monitoring the Connection Manager

You can view the status of the Connection Manager and its connections through SQL queries and HTTP endpoints.

#### Available SQL tables and views

The Connection Manager provides a number of tables and views that can be used to monitor the status of the Connection Manager and its connections. These include:

- bdr.stat_activity _ which is information from `pg_stat_activity` enhanced with addition columns regarding the `connection_manager_client_addr` and `connection_manager_client_port` is the connection has come through the connection manager, and `session_read_only` if it has connected through the read-only port.
- bdr.stat_connection_manager _ which is a view that provides statistics about the Connection Manager's status.
- bdr.stat_connection_manager_connections _ which is a view that provides statistics about the Connection Manager's connections.
- bdr.stat_connection_manager_node_stats _ which is a view that provides statistics about the Connection Manager on each of the data nodes.
- bdr.stat_connection_manager_hba_file_rules _ which is a view that shows which HBA file rules for the connection manager are being used on this node.

#### Available HTTP/HTTPS endpoints

The Connection Manager can be monitored through the HTTP API.

Endpoints returning true/false will also return a 200 status code for true and a 503 status code for false.

The following endpoints are available:

| Endpoint | Description |
|---|---|
| /connection/is-live | Is the connection manager live (listening), always returns "true", if the manager is not running, the client will simply fail to open the connection/url. |
| /connection/is-ready | Is the connection manager ready to accept connections, returns true(200)/false(503). |
| /connection/is-ready-ro | Is the connection manager ready to accept read-only connections, returns true(200)/false(503). |
| /connection/is-ready-rw | Is the connection manager ready to accept read-write connections, returns true(200)/false(503). |
| /node/is-read-write | Is this PGD node, not the connection manager but the PGD node itself, a read-write node (is it write leader), returns true(200)/false(503). |
| /node/is-read-only | Is this PGD node, not the connection manager but the PGD node itself, a read-only node (not the write leader), returns true(200)/false(503). |
| /group/read-write-info | Returns information about the read-write pool on this instance of connection manager - a list of nodes in the pool in JSON format with node id, node name, node host, node port and node dbname. For the read-write pool, the pool only contains one entry. |
| /group/read-only-info | Returns information about the read-only pool on this instance of connection manager - a list of nodes in the pool in JSON format with node id, node name, node host, node port and node dbname. |

Below is an example of a response body from the `/group/read-write-info` endpoint:

```
[
  {
    "id": 683485707,
    "name": "node-1",
    "host": "host-1",
    "port": 5432,
    "dbname": "pgddb"
  }
]
```

#### Logging

All Connection Manager log messages are written to the PostgreSQL log.

The behavior of `%r` and `%h` escape sequences in log_line_prefix has been altered to log "proxy_address/client_address" and "proxy_port/client_port" respectively.

This is achieved by the proxy setting a GUC for the server connections it uses. As users can override this GUC, any security context derived from the client_address will need to be verified by referring to the full session logs.

## 6.4 Postgres configuration

Several Postgres configuration parameters affect PGD nodes. You can set these parameters differently on each node, although we don't generally recommend it.

For PGD's own settings, see the PGD settings reference .

### 6.4.1 Postgres settings

To run correctly, PGD requires these Postgres settings:

- `wal_level` _ Must be set to `logical` , since PGD relies on logical decoding.
- `shared_preload_libraries` _ Must include `bdr` to enable the extension. Most other extensions can appear before or after the `bdr` entry in the comma-separated list. Don't include `pglogical` in this list.
- `track_commit_timestamp` _ Must be set to `on` for conflict resolution to retrieve the timestamp for each conflicting row.

PGD requires these PostgreSQL settings to be set to appropriate values, which vary according to the size and scale of the cluster:

- `logical_decoding_work_mem` _ Memory buffer size used by logical decoding. Transactions larger than this size overflow the buffer and are stored temporarily on local disk. Default is 64MB, but you can set it much higher.
- `max_worker_processes` _ PGD uses background workers for replication and maintenance tasks, so you need enough worker slots for it to work correctly. The formula for the correct minimal number of workers for each database is to add together these values:
- One per PostgreSQL instance
- One per database on that instance
- Four per PGD-enabled database
- One per peer node in the PGD group

- The number of peer nodes times the (number of writers (bdr.num_writers) plus one) You might need more worker processes temporarily when a node is being removed from a PGD group.
- `max_wal_senders` _ Two needed for every peer node.
- `max_replication_slots` _ Two needed for every peer node.
- `max_active_replication_origins` _ Three needed for every peer node. Available for PostgreSQL 18 and later.
- `wal_sender_timeout` and `wal_receiver_timeout` _ The default of one minute is usually sufficient, but large transactions may require longer than this amount of time to process. Since the WAL sender must process the full size of the transaction before transmitting it to a waiting replication connection, Postgres can see that as a timeout. If the problem is actually due to a large transaction, raising wal_sender_timeout to a higher value, like `3600s` or higher, and reloading the server could solve the problem. Additionally, this setting determines how quickly a node considers its CAMO partner as disconnected or reconnected. See *CAMO failure scenarios* for details.

In normal running for a group with N peer nodes, PGD requires N slots and WAL senders. During synchronization, PGD temporarily uses another N-1 slots and WAL senders, so be careful to set the parameters high enough for this occasional peak demand.

With Parallel Apply turned on, the number of slots must be increased to N slots from the formula * writers. This is because `max_replication_slots` also sets the maximum number of replication origins, and some of the functionality of Parallel Apply uses an extra origin per writer.

When the *Decoding worker* is enabled, this process requires one extra replication slot per PGD group.

Changing the `max_worker_processes` , `max_wal_senders` , and `max_replication_slots` parameters requires restarting the local node.

A legacy synchronous replication mode is supported using the following parameters. See *Commit scopes* for details and limitations.

- `synchronous_commit` and `synchronous_standby_names` _ Affects the durability and performance of PGD replication. in a similar way to physical replication .

### 6.4.2 Max prepared transactions

#### max_prepared_transactions

Needs to be set high enough to cope with the maximum number of concurrent prepared transactions across the cluster due to explicit two-phase commits, CAMO, or Eager transactions. Exceeding the limit prevents a node from running a local two-phase commit or CAMO transaction and prevents all Eager transactions on the cluster. This parameter can be set only at Postgres server start.

### 6.4.3 Considerations for global configuration

While certain PostgreSQL configuration parameters are useful for standalone instances, they can negatively impact PGD cluster stability, background workers, and replication processes.

The following parameters should not be enabled globally in `postgresql.conf` , nor should they be included as options in PGD node data source names (DSNs):

- `idle_session_timeout`
- `transaction_timeout`

Set these parameters on a per-session basis instead to make sure that they do not interfere with system-level operations.

## 6.5 Monitoring

Monitoring replication setups is important to ensure that your system:

- Performs optimally
- Doesn't run out of disk space
- Doesn't encounter other faults that might halt operations

It's important to have automated monitoring in place to ensure that the administrator is alerted and can take proactive action when issues occur. For example, the administrator can be alerted if replication slots start falling badly behind.

EDB provides Postgres Enterprise Manager (PEM), which supports PGD starting with version 8.1. See *Monitoring EDB PostgresDistributed* for more information.

Alternatively, tools or users can make their own calls into information views and functions provided by the BDR extension. See *Monitoring through SQL*

for details.

### 6.5.1 Monitoring through SQL

EDB Postgres Distributed provides several monitoring and statistics views that are specific to its distributed nature. The standard Postgres monitoring is also useful for monitoring EDB Postgres Distributed.

#### Monitoring overview

A PGD group consists of multiple servers, often referred to as nodes. Monitor all of the nodes to ensure the health of the whole group.

The bdr_monitor role can execute the `bdr.monitor` functions to provide an assessment of PGD health using one of three levels:

- `OK` — Often shown as green.
- `WARNING` — Often shown as yellow.
- `CRITICAL` — Often shown as red.
- `UNKNOWN` — For unrecognized situations, often shown as red.

PGD also provides dynamic catalog views that show the instantaneous state of various internal metrics. It also provides metadata catalogs that store the configuration defaults and configuration changes the user requests. Some of those views and tables are accessible by bdr_monitor or bdr_read_all_stats, but some contain user or internal information that has higher security requirements.

PGD allows you to monitor each of the nodes individually or to monitor the whole group by access to a single node. If you want to monitor each node individually, connect to each node and issue monitoring requests. If you want to monitor the group from a single node, then use the views starting with `bdr.group` since these requests make calls to other nodes to assemble a group-level information set.

If you were granted access to the `bdr.run_on_all_nodes()` function by bdr_superuser, then you can make your own calls to all nodes.

#### Monitoring node join and removal

By default, the node management functions wait for the join or part operation to complete. You can turn waiting off using the respective `wait_for_completion` function argument. If waiting is turned off, then to see when a join or part operation finishes, check the node state indirectly using `bdr.node_summary` and `bdr.event_summary` .

When called, the helper function `bdr.wait_for_join_completion()` causes a PostgreSQL session to pause until all outstanding node join operations area complete.

This example shows the output of a `SELECT` query from `bdr.node_summary` . It indicates that two nodes are active and another one is joining.

```
#  SELECT node_name, interface_connstr, peer_state_name,

#      node_seq_id, node_local_dbname

#  FROM bdr.node_summary;

- [ RECORD 1 ]-----+----------------------------------------
node_name          | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name    | ACTIVE
node_seq_id        | 1
node_local_dbname | postgres
- [ RECORD 2 ]-----+----------------------------------------
node_name          | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name    | ACTIVE
node_seq_id        | 2
node_local_dbname | postgres
- [ RECORD 3 ]-----+----------------------------------------
node_name          | node3
interface_connstr | host=localhost dbname=postgres port=7434
peer_state_name    | JOINING
node_seq_id        | 3
node_local_dbname | postgres
```

Also, the table bdr.node_catchup_info gives information on the catch-up state, which can be relevant to joining nodes or parting nodes.

When a node is parted, some nodes in the cluster might not receive all the data from that parting node. So parting a node creates a temporary slot from a node that already received that data and can forward it.

The `catchup_state` can be one of the following:

```
10 = setup
20 = start
30 = catchup
40 = done
```

### Monitoring the manager worker

The manager worker is responsible for many background tasks, including the managing of all the other workers. As such it is important to know what it's doing, especially in cases where it might seem stuck.

Accordingly, the bdr.stat_worker view provides per worker statistics for PGD workers, including manager workers. With respect to ensuring manager workers do not get stuck, the current task they are executing would be reported in their `query` field prefixed by "pgd manager:".

The `worker_backend_state` field for manager workers also reports whether the manager is idle or busy.

### Monitoring Routing

Routing is a critical part of PGD for ensuring a seemless application experience and conflict avoidance. Routing changes should happen quickly, including the detections of failures. At the same time we want to have as few disruptions as possible. We also want to ensure good load balancing for use-cases where it's supported.

Monitoring all of these is important for noticing issues, debugging issues, as well as informing more optimal configurations. Accoringly, there are two main views for monitoring statistics to do with routing:

- `bdr.stat_routing_state <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrstat_routing_state>`_ for monitoring the state of the connection routing with Connection Manager uses to route the connections.

- `bdr.stat_routing_candidate_state <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrstat_routing_candidate_state>`_ for information about routing candidate nodes from the point of view of the Raft leader (the view is empty on other nodes).

### Monitoring Replication Peers

You use two main views for monitoring of replication activity:

- `bdr.node_slots <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrnode_slots>`_ for monitoring outgoing replication

- `bdr.subscription_summary <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrsubscription_summary>`_ for monitoring incoming replication

You can also obtain most of the information provided by `bdr.node_slots` by querying the standard PostgreSQL replication monitoring views pg_catalog.pg_stat_replication

and pg_catalog.pg_replication_slots .

Each node has one PGD group slot that must never have a connection to it and is very rarely be marked as active. This is normal and doesn't imply something is down or disconnected. See *Replication slots created by PGD* in Node Management.

### Monitoring outgoing replication

You can use another view for monitoring of outgoing replication activity:

- `bdr.node_replication_rates <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrnode_replication_rates>`_ for monitoring outgoing replication

The bdr.node_replication_rates view gives an overall picture of the outgoing replication activity along with the catchup estimates for peer nodes, specifically.

```
#  SELECT * FROM bdr.node_replication_rates;

- [ RECORD 1 ]----+-----------
peer_node_id     | 112898766
```

(continues on next page)

(continued from previous page)

```
target_name      | node1
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
replay_lag       | 00:00:00
replay_lag_bytes | 0
replay_lag_size  | 0 bytes
apply_rate       | 822
catchup_interval | 00:00:00
- [ RECORD 2 ]----+-----------
peer_node_id     | 312494765
target_name      | node3
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
replay_lag       | 00:00:00
replay_lag_bytes | 0
replay_lag_size  | 0 bytes
apply_rate       | 853
catchup_interval | 00:00:00
```

The `apply_rate` refers to the rate in bytes per second. It's the rate at which the peer is consuming data from the local node. The `replay_lag` when a node reconnects to the cluster is immediately set to zero. This information will be fixed in a future release. As a workaround, we recommend using the `catchup_interval` column that refers to the time required for the peer node to catch up to the local node data. The other fields are also available from the bdr.node_slots

view.

Administrators can query bdr.node_slots for outgoing replication from the local node. It shows information about replication status of all other nodes in the group that are known to the current node as well as any additional replication slots created by PGD on the current node.

```
#  SELECT node_group_name, target_dbname, target_name, slot_name, active_pid,

#      catalog_xmin, client_addr, sent_lsn, replay_lsn, replay_lag,

#      replay_lag_bytes, replay_lag_size

#  FROM bdr.node_slots;

- [ RECORD 1 ]---+----------------------------
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node3
slot_name       | bdr_postgres_bdrgroup_node3
active_pid      | 15089
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 120
replay_lag_size | 120 bytes
- [ RECORD 2 ]---+----------------------------
```

(continues on next page)

(continued from previous page)

```
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node2
slot_name       | bdr_postgres_bdrgroup_node2
active_pid      | 15031
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 84211
replay_lag_size | 82 kB
```

Because PGD is a mesh network, to get the full view of lag in the cluster, you must execute this query on all nodes participating.

`replay_lag_bytes` reports the difference in WAL positions between the local server's current WAL write position and `replay_lsn` , the last position confirmed replayed by the peer node. `replay_lag_size` is a human-readable form of the same. It's important to understand that WAL usually contains a lot of writes that aren't replicated but still count in `replay_lag_bytes` , including, for example:

- `VACUUM` activity
- Index changes
- Writes associated with other databases on the same node
- Writes for tables that are not part of a replication set

So the lag in bytes reported here isn't the amount of data that must be replicated on the wire to bring the peer node up to date, only the amount of server-side WAL that must be processed.

Similarly, `replay_lag` isn't a measure of how long the peer node takes to catch up or how long it takes to replay from its current position to the write position at the time `bdr.node_slots` was queried. It measures the delay between when the peer confirmed the most recent commit and the current wall-clock time. We suggest that you monitor `replay_lag_bytes` and `replay_lag_size` or `catchup_interval` in `bdr.node_replication_rates` , as this column is set to zero immediately after the node reconnects.

The lag in both bytes and time doesn't advance while logical replication is streaming a transaction. It changes only when a commit is replicated. So the lag tends to "sawtooth," rising as a transaction is streamed and then falling again as the peer node commits it, flushes it, and sends confirmation. The reported LSN positions "stair-step" instead of advancing smoothly, for similar reasons.

When replication is disconnected (`active = 'f'` ), the `active_pid` column is `NULL` , as is `client_addr` and the other fields that make sense only with an active connection. The `state` field is `'disconnected'` . The `_lsn` fields are the same as the `confirmed_flush_lsn` , since that's the last position that the client is known for certain to have replayed to and saved. The `_lag` fields show the elapsed time between the most recent confirmed flush on the client and the current time. The `_lag_size` and `_lag_bytes` fields report the distance between `confirmed_flush_lsn` and the local server's current WAL insert position.

**Note**

It's normal for *restart_lsn* to be behind the other *lsn* columns. This doesn't indicate a problem with replication or a peer node lagging. The *restart_lsn* is the position that PostgreSQL's internal logical decoding must be reading WAL at if interrupted. It generally reflects the position of the oldest transaction that's not yet replicated and flushed. A very old *restart_lsn* can make replication slow to restart after disconnection and force retention of more WAL than

is desirable, but it's otherwise harmless. If you're concerned, look for very long-running transactions and forgotten prepared transactions.

### Monitoring incoming replication

You can monitor incoming replication (also called subscriptions) at a high level by querying the `bdr.subscription_summary` view. This query shows the list of known subscriptions to other nodes in the EDB Postgres Distributed cluster and the state of the replication worker:

```
#  SELECT node_group_name, origin_name, sub_enabled, sub_slot_name,

#      subscription_status

#  FROM bdr.subscription_summary;

- [ RECORD 1 ]-------+---------------------------
node_group_name     | bdrgroup
origin_name         | node2
sub_enabled         | t
sub_slot_name       | bdr_postgres_bdrgroup_node1
subscription_status | replicating
- [ RECORD 2 ]-------+---------------------------
node_group_name     | bdrgroup
origin_name         | node3
sub_enabled         | t
sub_slot_name       | bdr_postgres_bdrgroup_node1
subscription_status | replicating
```

You can further monitor subscriptions by monitoring subscription summary statistics through bdr.stat_subscription , and by monitoring the subscription replication receivers and subscription replication writers, using bdr.stat_receiver and bdr.stat_writer , respectively.

### Monitoring WAL senders using LCR

If the *Decoding worker* is enabled, you can monitor information about the current logical change record (LCR) file for each WAL sender using the function bdr.wal_sender_stats() . For example:

```
postgres=# SELECT * FROM bdr.wal_sender_stats();
   pid   | is_using_lcr |       decoder_slot_name       |              lcr_file_name
- --------+--------------+-------------------------------+---------------------------
↪--------------
 2059904 | f            |                               |
 2059909 | t            | bdr_postgres_bdrgroup_decoder |␣
↪000000000000000000000014000000000000000000
 2059916 | t            | bdr_postgres_bdrgroup_decoder |␣
↪000000000000000000000014000000000000000000
(3 rows)
```

If `is_using_lcr` is `FALSE` , `decoder_slot_name` /`lcr_file_name` is `NULL` . This is the case if the decoding worker isn't enabled or the WAL sender is serving a *Logical standby nodes* .

Also, you can monitor information about the decoding worker using the function bdr.get_decoding_worker_stat() . For example:

```
postgres=# SELECT * FROM bdr.get_decoding_worker_stat();
   pid   | decoded_upto_lsn | waiting | waiting_for_lsn
- --------+------------------+---------+-----------------
 1153091 | 0/1E5EEE8        | t       | 0/1E5EF00
(1 row)
```

### Monitoring PGD replication workers

All PGD workers show up in the system view `bdr.stat_activity` , which has the same columns and information content as pg_stat_activity . So this view offers these insights into the state of a PGD system:

- The wait_event column has enhanced information, if the reason for waiting is related to PGD.
- The `query` column is blank in PGD workers, except when a writer process is executing DDL, or for when a manager worker is active (in which case the entry in the `query` column will be prefixed with "`pgd manager:`").

The `bdr.workers` view shows PGD worker-specific details that aren't available from `bdr.stat_activity` .

The view `bdr.event_summary` shows the last error (if any) reported by any worker that has a problem continuing the work. This information is persistent, so it's important to note the time of the error and not just its existence. Most errors are transient, and PGD workers will retry the failed operation.

### Monitoring PGD writers

Another system view, `bdr.writers` , monitors writer activities. This view shows only the current status of writer workers. It includes:

- `sub_name` to identify the subscription that the writer belongs to
- `pid` of the writer process
- `streaming_allowed` to know if the writer supports applying in-progress streaming transactions
- `is_streaming` to know if the writer is currently applying a streaming transaction
- `commit_queue_position` to check the position of the writer in the commit queue

PGD honors commit ordering by following the same commit order as happened on the origin. In case of parallel writers, multiple writers might apply different transactions at the same time. The `commit_queue_position` shows the order in which they will commit. Value `0` means that the writer is the first one to commit. Value `-1` means that the commit position isn't yet known, which can happen for a streaming transaction or when the writer isn't currently applying any transaction.

### Monitoring commit scopes

Commit scopes are our durability and consistency configuration framework. As such, they affect the performance of transactions, so it is important to get statistics on them. Moreover, because in failure scenarios transactions might appear to be stuck due to the commit scope configuration, we need insight into what commit scope is being used, what it's waiting on, and so on.

Accordingly, these two views show relevant statistics about commit scopes:

- `bdr.stat_commit_scope <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrstat_commit_scope>`_ for cumulative statistics for each commit scope, including degrade event counters (`ndegrades` , `nconfig_degrades` ) and the timestamp of the last configuration state change (`last_state_change_time` ).

- `bdr.stat_commit_scope_state <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible/#bdrstat_commit_scope_state>`_ for information about the current use of commit scopes by backend processes.

See *Monitoring degrade events* for more information about monitoring degrade events.

### Monitoring global locks

The global lock, which is currently used only for DDL replication, is a heavyweight lock that exists across the whole PGD group.

There are currently two types of global locks:

- DDL lock, used for serializing all DDL operations on permanent (not temporary) objects (that is, tables) in the database
- DML relation lock, used for locking out writes to relations during DDL operations that change the relation definition

You can create either or both entry types for the same transaction, depending on the type of DDL operation and the value of the `bdr.ddl_locking` setting.

Global locks held on the local node are visible in the bdr.global_locks view. This view shows the type of the lock. For relation locks, it shows the relation that's being locked, the PID holding the lock (if local), and whether the lock was globally granted. In case of global advisory locks, `lock_type` column shows `GLOBAL_LOCK_ADVISORY` , and `relation` column shows the advisory keys on which the lock is acquired.

This example shows the output of `bdr.global_locks` while running an `ALTER TABLE` statement with `bdr.ddl_locking = 'all'`:

```
#  SELECT lock_type, relation, pid FROM bdr.global_locks;

- [ RECORD 1 ]--------------
lock_type | GLOBAL_LOCK_DDL
relation  | [NULL]
pid       | 15534
- [ RECORD 2 ]--------------
lock_type | GLOBAL_LOCK_DML
relation  | someschema.sometable
pid       | 15534
```

See Catalogs for details on all fields, including lock timing information.

### Monitoring conflicts

Replication *bdr_read_all_conflicts* can arise when multiple nodes make changes that affect the same rows in ways that can interact with each other. Monitor the PGD system to identify conflicts and, where possible, make application changes to eliminate the conflicts or make them less frequent.

By default, all conflicts are logged to `bdr.conflict_history` . Since this log contains full details of conflicting data, the rows are protected by row-level security to ensure they're visible only by owners of replicated tables. Owners should expect conflicts and analyze them to see which, if any, might be considered as problems to resolve.

For monitoring purposes, use bdr.conflict_history_summary , which doesn't contain user data. This example shows a query to count the number of conflicts seen in the current day using an efficient query plan:

```
SELECT count(*)
FROM bdr.conflict_history_summary
WHERE local_time > date_trunc(day, current_timestamp)
  AND local_time < date_trunc(day, current_timestamp + 1 day);
```

### Apply statistics

PGD collects statistics about replication apply, both for each subscription and for each table.

Two monitoring views exist: bdr.stat_subscription for subscription statistics and bdr.stat_relation for relation statistics. These views both provide:

- Number of INSERTs/UPDATEs/DELETEs/TRUNCATEs replicated
- Block accesses and cache hit ratio
- Total I/O time for read/write
- Number of in-progress transactions streamed to file
- Number of in-progress transactions streamed to writers
- Number of in-progress streamed transactions committed/aborted

For relations only, bdr.stat_relation also includes:

- Total time spent processing replication for the relation
- Total lock wait time to acquire lock (if any) for the relation (only)

For subscriptions only, bdr.stat_subscription includes:

- Number of COMMITs/DDL replicated for the subscription
- Number of times this subscription has connected upstream

Tracking of these statistics is controlled by the PGD GUCs

bdr.track_subscription_apply and bdr.track_relation_apply , respectively.

The following shows the example output from these:

```
#  SELECT sub_name, nconnect, ninsert, ncommit, nupdate, ndelete, ntruncate, nddl

FROM bdr.stat_subscription;
- [ RECORD 1 ]-----------------------------------
sub_name  | bdr_regression_bdrgroup_node1_node2
nconnect  | 3
ninsert   | 10
ncommit   | 5
nupdate   | 0
ndelete   | 0
ntruncate | 0
nddl      | 2
```

In this case, the subscription connected three times to the upstream, inserted 10 rows, and performed two DDL commands inside five transactions.

You can reset the stats counters for these views to zero using the functions bdr.reset_subscription_stats and bdr.reset_relation_stats .

PGD also monitors statistics regarding subscription replication receivers and subscription replication writers for each subscription, using bdr.stat_receiver and bdr.stat_writer , respectively.

### Standard PostgreSQL statistics views

Statistics on table and index usage are normally updated by the downstream master. This is essential for the correct function of

autovacuum . If there are no local writes on the downstream master and statistics haven't been reset, these two views show corresponding results between upstream and downstream:

- `pg_stat_user_tables`
- `pg_statio_user_tables`

> **Note**
>
> We don't necessarily expect the upstream table statistics to be *similar* to the downstream ones. We only expect them to *change* by the same amounts. Consider the example of a table whose statistics show 1M inserts and 1M updates. When a new node joins the PGD group, the statistics for the same table in the new node show 1M inserts and zero updates. However, from that moment, the upstream and downstream table statistics change by the same amounts because all changes on one side are replicated to the other side.

Since indexes are used to apply changes, the identifying indexes on the downstream side might appear more heavily used with workloads that perform `UPDATE` and `DELETE` than non-identifying indexes are.

The built-in index monitoring views are:

- `pg_stat_user_indexes`
- `pg_statio_user_indexes`

All these views are discussed in detail in the PostgreSQL documentation on the statistics views .

### Monitoring PGD versions

PGD allows running different Postgres versions as well as different BDR extension versions across the nodes in the same cluster. This capability is useful for upgrading.

The view bdr.group_versions_details uses the function bdr.run_on_all_nodes() to retrieve Postgres and BDR extension versions from all nodes at the same time. For example:

```
pgddb=# SELECT node_name, postgres_version, bdr_version
        FROM bdr.group_versions_details;
 node_name | postgres_version | bdr_version
- ----------+------------------+-------------
 node1     | 15.2.0           | 5.0.0
 node2     | 15.2.0           | 5.0.0
```

The recommended setup is to try to have all nodes running the same (and latest) versions as soon as possible. We recommend that the cluster doesn't run different versions of the BDR extension for too long.

For monitoring purposes, we recommend the following alert levels:

- status=UNKNOWN, message=This node is not part of any PGD group
- status=OK, message=All nodes are running same PGD versions
- status=WARNING, message=There is at least 1 node that is not accessible

- status=WARNING, message=There are node(s) running different PGD versions when compared to other nodes

The described behavior is implemented in the function bdr.monitor_group_versions() , which uses PGD version information returned from the view `bdr.group_version_details` to provide a cluster-wide version check. For example:

```
pgddb=# SELECT * FROM bdr.monitor_group_versions();
 status |                message
- -------+----------------------------------------
 OK     | All nodes are running same BDR versions
```

### Monitoring Raft consensus

Raft consensus must be working cluster-wide at all times. The impact of running an EDB Postgres Distributed cluster without Raft consensus working might be as follows:

- The replication of PGD data changes might still work correctly.
- Global DDL/DML locks doesn't work.
- Galloc sequences eventually run out of chunks.
- Eager Replication doesn't work.
- Cluster maintenance operations (join node, part node, promote standby) are still allowed, but they might not finish (hanging instead).
- Node statuses might not be correctly synced among the PGD nodes.
- PGD group replication slot doesn't advance LSN and thus keeps WAL files on disk.

The view bdr.group_raft_details uses the functions bdr.run_on_all_nodes() and

bdr.get_raft_status() to retrieve Raft consensus status from all nodes at the same time. For example:

```
pgddb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
  node_id    | node_name | node_group_name |     state      | leader_id
- -----------+-----------+-----------------+----------------+------------
 1148549230 | node1     | top_group       | RAFT_LEADER    | 1148549230
 3367056606 | node2     | top_group       | RAFT_FOLLOWER | 1148549230
```

Raft consensus is working correctly if all of these conditions are met:

- A valid state (`RAFT_LEADER` or `RAFT_FOLLOWER` ) is defined on all nodes.
- Only one of the nodes is the `RAFT_LEADER` .
- The `leader_id` is the same on all rows and must match the `node_id` of the row where `state = RAFT_LEADER` .

From time to time, Raft consensus starts a new election to define a new `RAFT_LEADER` . During an election, there might be an intermediary situation where there's no `RAFT_LEADER` , and some of the nodes consider themselves as `RAFT_CANDIDATE` . The whole election can't take longer than `bdr.raft_global_election_timeout` (by default it's set to 6 seconds). If the query above returns an in-election situation, then wait for `bdr.raft_global_election_timeout` , and run the query again. If after `bdr.raft_global_election_timeout` has passed and some the listed conditions are still not met, then Raft consensus isn't working.

Raft consensus might not be working correctly on only a single node. For example, one of the nodes doesn't recognize the current leader and considers itself as a `RAFT_CANDIDATE` . In this case, it's important to make sure that:

- All PGD nodes are accessible to each other through both regular and replication connections (check file `pg_hba.conf` ).

- PGD versions are the same on all nodes.
- `bdr.raft_global_election_timeout` is the same on all nodes.

In some cases, especially if nodes are geographically distant from each other or network latency is high, the default value of `bdr.raft_global_election_timeout` (6 seconds) might not be enough. If Raft consensus is still not working even after making sure everything is correct, consider increasing `bdr.raft_global_election_timeout` to 30 seconds on all nodes.

Given how Raft consensus affects cluster operational tasks, and also as Raft consensus is directly responsible for advancing the group slot, monitoring alert levels are defined as follows:

- status=UNKNOWN, message=This node is not part of any PGD group
- status=OK, message=Raft Consensus is working correctly
- status=WARNING, message=There is at least 1 node that is not accessible
- status=WARNING, message=There are node(s) as RAFT_CANDIDATE, an election might be in progress
- status=WARNING, message=There is no RAFT_LEADER, an election might be in progress
- status=CRITICAL, message=There is a single node in Raft Consensus
- status=CRITICAL, message=There are node(s) as RAFT_CANDIDATE while a RAFT_LEADER is defined
- status=CRITICAL, message=There are node(s) following a leader different than the node set as RAFT_LEADER

The described behavior is implemented in the function bdr.monitor_group_raft() , which uses Raft consensus status information returned from the view bdr.group_raft_details

to provide a cluster-wide Raft check. For example:

```
pgddb=# SELECT * FROM bdr.monitor_group_raft();
node_group_name | status |               message
- ---------------/--------+-------------------------------------
mygroup          | OK     | Raft Consensus is working correctly
```

Two further views that can give a finer-grained look at the state of Raft consensus are bdr.stat_raft_state , which provides the state of the Raft consensus on the local node, and bdr.stat_raft_followers_state , which provides a view when on the Raft leader (it is empty on other nodes) regarding the state of the followers of that Raft leader.

### Monitoring replication slots

Each PGD node keeps:

- One replication slot per active PGD peer
- One group replication slot

For example:

```
pgddb=# SELECT slot_name, database, active, confirmed_flush_lsn
FROM pg_replication_slots ORDER BY slot_name;
        slot_name         | database | active | confirmed_flush_lsn
- ------------------------+----------+--------+---------------------
 bdr_pgddb_bdrgroup       | pgddb    | f      | 0/3110A08
 bdr_pgddb_bdrgroup_node2 | pgddb    | t      | 0/31F4670
 bdr_pgddb_bdrgroup_node3 | pgddb    | t      | 0/31F4670
 bdr_pgddb_bdrgroup_node4 | pgddb    | t      | 0/31F4670
```

Peer slot names follow the convention `bdr_<DATABASE>_<GROUP>_<PEER>` , while the PGD group slot name follows the convention `bdr_<DATABASE>_<GROUP>` . You can access the group slot using the function bdr.local_group_slot_name() .

Peer replication slots must be active on all nodes at all times. If a peer replication slot isn't active, then it might mean either:

- The corresponding peer is shut down or not accessible.
- PGD replication is broken.

Grep the log file for `ERROR` or `FATAL` , and also check `bdr.event_summary` on all nodes. The root cause might be, for example, an incompatible DDL was executed with DDL replication disabled on one of the nodes.

The PGD group replication slot is, however, inactive most of the time. PGD maintains this slot and advances its LSN when all other peers already consumed the corresponding transactions. Consequently, it's not necessary to monitor the status of the group slot.

The function bdr.monitor_local_replslots() provides a summary of whether all PGD node replication slots are working as expected. This summary is also available on subscriber-only nodes that are operating as subscriber-only group leaders in a PGD cluster when *Optimizing subscriber-only groups* is enabled. For example:

```
pgddb=# SELECT * FROM bdr.monitor_local_replslots();
 status |                    message
- -------+-------------------------------------------------
 OK     | All BDR replication slots are working correctly
```

One of the following status summaries is returned:

| Status | Message |
|---|---|
| UNKNOWN | This node is not part of any BDR group |
| OK | All BDR replication slots are working correctly |
| OK | This node is part of a subscriber-only group |
| CRITICAL | There is at least 1 BDR replication slot which is inactive |
| CRITICAL | There is at least 1 BDR replication slot which is missing |

### Monitoring transaction COMMITs

By default, PGD transactions are committed only to the local node. In that case, a transaction's `COMMIT` is processed quickly.

PGD's *Commit scopes* feature offers a range of synchronous transaction commit scopes that allow you to balance durability, consistency, and performance for your particular queries. You can monitor these transactions by examining the bdr.stat_activity catalog. The processes report different `wait_event` states as a transaction is committed. This monitoring only covers transactions in progress and doesn't provide historical timing information.

## 6.6 AutoPartition in PGD

PGD AutoPartition allows you to split tables into several partitions, other tables, and creating and dropping partitions as needed. Autopartitioning is useful for managing large tables that grow over time as it allows you to separate the data into manageable chunks. You can create new partitions regularly and then drop them when the data retention period expires.

This capability uses features of PGD, such as low-conflict locking of creating and dropping partitions.

You perform PGD management primarily by using functions that can be called by SQL. All functions in PGD are exposed in the `bdr` schema. Unless you put it into your search_path, you need to schema qualify the name of each function.

### 6.6.1 Auto creation of partitions

PGD AutoPartition uses the bdr.autopartition()

function to create or alter the definition of automatic range partitioning for a table. If no definition exists, it's created. Otherwise, later executions will alter the definition.

PGD AutoPartition in PGD 5.5 and later leverages underlying Postgres features that allow a partition to be attached or detached/dropped without locking the rest of the table. Versions of PGD earlier than 5.5 don't support this feature and lock the tables.

An error is raised if the table isn't RANGE partitioned or a multi-column partition key is used.

By default, AutoPartition manages partitions locally. Managing partitions locally is useful when the partitioned table isn't a replicated table. In that case, you might not need or want to have all partitions on all nodes. For example, the built-in bdr.conflict_history

table isn't a replicated table. It's managed by AutoPartition locally. Each node creates partitions for this table locally and drops them once they're old enough.

Also consider:

- Activities are performed only when the entry is marked `enabled = on .`
- We recommend that you don't manually create or drop partitions for tables managed by AutoPartition. Doing so can make the AutoPartition metadata inconsistent and might cause it to fail.

### 6.6.2 AutoPartition examples

Daily partitions, keep data for one month:

```
CREATE TABLE measurement (
logdate date not null,
peaktemp int,
unitsales int
) PARTITION BY RANGE (logdate);

bdr.autopartition(measurement, 1 day, data_retention_period := 30 days);
```

Create five advance partitions when there are only two more partitions remaining. Each partition can hold 1 billion orders.

```
bdr.autopartition(Orders, 1000000000,
        partition_initial_lowerbound := 0,
        minimum_advance_partitions := 2,
        maximum_advance_partitions := 5
    );
```

### 6.6.3 RANGE-partitioned tables

PGD autopartitioning supports range partitioning using the `RANGE` keyword. Range partitioning allows you to partition a table based on the ranges of values in a column. For example, you can partition a table by date, where each partition contains data for a specific date range. This is useful for managing large tables that grow over time, as it allows you to separate the data into manageable chunks.

A new partition is added for every `partition_increment` range of values. Lower and upper bound are `partition_increment` apart. For tables with a partition key of type `timestamp` or `date` , the `partition_increment` must be a valid constant of type `interval` . For example, specifying `1 Day` causes a new partition to be added each day, with partition bounds that are one day apart.

If the partition column is connected to a `snowflakeid` , `timeshard` , or `ksuuid` sequence, you must specify the `partition_increment` as type `interval` . Otherwise, if the partition key is integer or numeric, then the `partition_increment` must be a valid constant of the same datatype. For example, specifying `1000000` causes new partitions to be added every 1 million values.

If the table has no existing partition, then the specified `partition_initial_lowerbound` is used as the lower bound for the first partition. If you don't specify `partition_initial_lowerbound` , then the system tries to derive its value from the partition column type and the specified `partition_increment` . For example, if `partition_increment` is specified as `1 Day` , then `partition_initial_lowerbound` is set to CURRENT DATE. If `partition_increment` is specified as `1 Hour` , then `partition_initial_lowerbound` is set to the current hour of the current date. The bounds for the subsequent partitions are set using the `partition_increment` value.

The system always tries to have a certain minimum number of advance partitions. To decide whether to create new partitions, it uses the specified `partition_autocreate_expression` . This can be an expression that can be evaluated by SQL that's evaluated every time a check is performed. For example, for a partitioned table on column type `date` , suppose `partition_autocreate_expression` is specified as `DATE_TRUNC('day',CURRENT_DATE)` , `partition_increment` is specified as `1 Day` , and `minimum_advance_partitions` is specified as `2` . New partitions are then created until the upper bound of the last partition is less than `DATE_TRUNC('day', CURRENT_DATE) + '2 Days'::interval` .

The expression is evaluated each time the system checks for new partitions.

For a partitioned table on column type `integer` , you can specify the `partition_autocreate_expression` as `SELECT max(partcol) FROM schema.partitioned_table` . The system then regularly checks if the maximum value of the partitioned column is within the distance of `minimum_advance_partitions * partition_increment` of the last partition's upper bound. Create an index on the `partcol` so that the query runs efficiently. If you don't specify the `partition_autocreate_expression` for a partition table on column type `integer` , `smallint` , or `bigint` , then the system sets it to `max(partcol)` .

If the `data_retention_period` is set, partitions are dropped after this period. To minimize locking, partitions are dropped at the same time as new partitions are added. If you don't set this value, you must drop the partitions manually.

The `data_retention_period` parameter is supported only for timestamp-based (and related) partitions. The period is calculated by considering the upper bound of the partition. The partition is dropped if the given period expires, relative to the upper bound.

### 6.6.4 Stopping automatic creation of partitions

Use bdr.drop_autopartition()

to drop the autopartitioning rule for the given relation. All pending work items for the relation are deleted, and no new work items are created.

### 6.6.5 Waiting for partition creation

Partition creation is an asynchronous process. AutoPartition provides a set of functions to wait for the partition to be created, locally or on all nodes.

Use bdr.autopartition_wait_for_partitions()

to wait for the creation of partitions on the local node. The function takes the partitioned table name and a partition key column value and waits until the partition that holds that value is created.

The function waits only for the partitions to be created locally. It doesn't guarantee that the partitions also exist on the remote nodes.

To wait for the partition to be created on all PGD nodes, use the

bdr.autopartition_wait_for_partitions_on_all_nodes()

function. This function internally checks local as well as all remote nodes and waits until the partition is created everywhere.

### 6.6.6 Finding a partition

Use the bdr.autopartition_find_partition()

function to find the partition for the given partition key value. If a partition to hold that value doesn't exist, then the function returns NULL. Otherwise it returns the Oid of the partition.

### 6.6.7 Enabling or disabling autopartitioning

Use bdr.autopartition_enable()

to enable autopartitioning on the given table. If autopartitioning is already enabled, then no action occurs. Similarly, use bdr.autopartition_disable()

to disable autopartitioning on the given table.

### 6.6.8 Dropping or detaching a partition

By default, partitions of tables managed by `autopartition()` are detached and dropped once the `data_retention_period` expires. The `drop_after_retention_period` parameter can control this behavior so you can choose to keep those partitions. When set to `false` , the partitions are only detached from the parent table, but not dropped. This allows users to potentially re-attach the partition to some other table or process them before finally dropping. The default value of the parameter is `true` so in any existing or new autopartitioned tables, the partitions are dropped after `data_retention_period` expires. To handle partitions that are detached but not dropped, manually re-attach those partitions to some other table or drop them.

## 6.7 Commit Scopes

Fully managable and configurable commit scopes are a feature of PGD Expanded.

PGD Expanded offers a range of synchronous modes to complement its default asynchronous replication. You use commit scopes to configure these synchronous modes. Commit scopes are rules that define how PGD handles synchronous operations and when the system considers a transaction committed.

PGD Essential offers a limited set of commit scopes that are pre-defined and cannot be changed.

### 6.7.1 Introducing

- ```
  :ref:`Overview of durability options <Overview of durability options>` ␣
  ↪introduces the concepts and some of the essential terminology that's used when␣
  ↪discussing synchronous commits.
  ```

- ```
  :ref:`Durability terminology <Durability terminology>`  lists terms used around␣
  ↪PGD's durability options, including how to refer to nodes in replication.
  ```

- ```
  :ref:`Commit scopes <Commit scopes>`  is a more in-depth look at the structure of␣
  ↪commit scopes and how to define them for your needs.
  ```
  ```

- :ref:`Predefined commit scopes <Predefined commit scopes>` lists the pre-defined commit scopes that are available in PGD Essential.
- :ref:`Commit scope groups <Commit scope groups>` introduces the notion of an origin group, and how to leverage these when defining commit scopes rules.
- :ref:`Commit scope rules <Commit scope rules>` looks at the syntax of and how to formulate a commit scope rule.
- :ref:`Comparing durability options <Comparing durability options>` compares how commit scope options behave with regard to durability.
- :ref:`Degrading commit scope rules <Degrading commit scope rules>` shows how to set up a commit scope rule that can gracefully degrade to a lower setting in case of timeouts with a stricter setting.

### 6.7.2 Commit scope kinds

- :ref:`SYNCHRONOUS COMMIT <SYNCHRONOUS COMMIT>` is a commit scope mechanism that works in a similar fashion to legacy synchronous replication, but from within the commit scope framework.
- :ref:`QUORUM COMMIT <QUORUM COMMIT>` is the commit scope kind for workloads requiring distributed transaction consistency. Quorum Commit guarantees that all participating nodes commit together or roll back together, preventing conflicts before they occur.
- :ref:`Commit At Most Once <Commit At Most Once>` focuses on the Commit At Most Once option, in which applications take responsibility for verifying that a transaction has been committed before retrying. This ensures that their commits only happen at most once.
- :ref:`LAG CONTROL <LAG CONTROL>` looks at the commit scope mechanism which dynamically throttle nodes according to the slowest node and regulates how far out of sync nodes may go when a database node goes out of service.
- :ref:`Group Commit (legacy) <Group Commit (legacy)>` focuses on the Group Commit option, where you can define a transaction as done when a group of nodes agrees it's done.

### 6.7.3 Working with commit scopes

- :ref:`Administering <Administering>` addresses how to manage a PGD cluster with commit scopes in use.
- :ref:`Legacy synchronous replication using PGD <Legacy synchronous replication using PGD>` shows how you can still access traditional Postgres synchronous operations under PGD.

- :ref:`Internal timing of operations <Internal timing of operations>` compares legacy replication with PGD's async and synchronous operations, especially the difference in the order by which transactions are flushed to disk or made visible.

## Overview of durability options

### Overview

EDB Postgres Distributed (PGD) allows you to choose from several replication configurations based on your durability, consistency, availability, and performance needs using *commit scopes*.

In its basic configuration, PGD uses asynchronous replication. However, commit scopes can change both the default and the per-transaction behavior.

It's also possible to configure the legacy Postgres synchronous replication using standard `synchronous_standby_names` in the same way as the built-in physical or logical replication. However, commit scopes provide much more flexibility and control over the replication behavior.

The different synchronization settings affect three properties of interest to applications that are related but can all be implemented individually:

- Durability: Writing to multiple nodes increases crash resilience and allows you to recover the data after a crash and restart.
- Visibility: With the commit confirmation to the client, the database guarantees immediate visibility of the committed transaction on some sets of nodes.
- Conflict handling: Conflicts can be handled optimistically postcommit, with conflicts resolved when the transaction is replicated based on commit timestamps. Or, they can be handled pessimistically precommit. The client can rely on the transaction to eventually be applied on all nodes without further conflicts or get an abort, directly informing the client of an error.

Commit scopes allow four kinds of controlling durability of the transaction:

- :ref:`SYNCHRONOUS COMMIT <SYNCHRONOUS COMMIT>` : This kind of commit scope allows for a behavior where the origin node awaits a majority of nodes to confirm and behaves more like a native Postgres synchronous commit.
- :ref:`QUORUM COMMIT <QUORUM COMMIT>` : Quorum Commit internally uses two-phase commit to guarantee that a transaction commits or rolls back in the entire cluster, effectively serializing conflicting transactions across the cluster.
- :ref:`Commit At Most Once <Commit At Most Once>` : This kind of commit scope works with the client to verify that a transaction was committed before retrying, preventing duplicate commits in failover scenarios.
- :ref:`LAG CONTROL <LAG CONTROL>` : This kind of commit scope controls how far behind nodes can be in terms of replication before allowing commit to proceed.

Synchronous Commit and CAMO each support *Degrading commit scope rules* , for even further control of durability.

### Durability terminology

### Durability terminology

This page covers terms and definitions directly related to PGD's durability options. For other terms, see *Postgres Distributed terminology* .

### Nodes

PGD nodes take different roles during the replication of a transaction. These are implicitly assigned per transaction and are unrelated even for concurrent transactions.

- The *origin* is the node that receives the transaction from the client or application. It's the node processing the transaction first, initiating replication to other PGD nodes and responding back to the client with a confirmation or an error.
- The *origin node group* is a PGD group which includes the origin.
- A *partner* node is a PGD node expected to confirm transactions according to Group Commit requirements.
- A *commit group* is the group of all PGD nodes involved in the commit, that is, the origin and all of its partner nodes, which can be just a few or all peer nodes.

### Commit scopes

Commit scopes give applications granular control about durability and consistency of EDB Postgres Distributed.

A commit scope is a set of rules that describes the behavior of the system as transactions are committed. The actual behavior depends on which a kind of commit scope a commit scope's rule uses *SYNCHRONOUS COMMIT* , *QUORUM COMMIT* , *Commit At Most Once* , *LAG CONTROL* , or a combination of these.

While most commit scope kinds control the processing of the transaction, Lag Control is the exception as it dynamically regulates the performance of the system in response to replication operations being slow or queued up. It is typically used, though, in combination with other commit scope kinds

### Commit scope structure

Every commit scope has a name (a `commit_scope_name` ).

Each commit scope has one or more rules.

Each rule within the commit scope has an `origin_node_group` which together uniquely identify the commit scope rule.

The `origin_node_group` is a PGD group and it defines the nodes which will apply this rule when they are the originators of a transaction.

Finally there is the rule which defines what kind of commit scope or combination of commit scope kinds should be applied to those transactions.

So if a commit scope has a rule that reads:

origin_node_group := 'example_bdr_group', rule := 'MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT',

Then, the rule is applied when any node in the `example_bdr_group` issues a transaction.

The rule itself specifies how many nodes of a specified group will need to confirm the change. `MAJORITY ORIGIN GROUP` is the commit scope group, and `SYNCHRONOUS COMMIT` is the commit scope kind. This translates to requiring that a majority of nodes in the origin node's group must confirm the change before the change can be considered as committed.

### How a commit scope is selected

When any change takes place, PGD looks up which commit scope should be used for the transaction or node.

If a transaction specifies a commit scope, that scope will be used.

If not specified, the system will search for a default commit scope. Default commit scopes are a group level setting. The system consults the group tree. Starting at the bottom of the group tree with the node's group and working up, it searches for any group which has a default_commit_scope setting defined. This commit scope will then be used.

If no default_commit_scope is found then the node's GUC, bdr.commit_scope is used. And if that isn't set or is set to `local` then no commit scope applies and PGD's async replication is used.

A commit scope will not be used if it is not local and the node where the commit is being run on is not directly or indirectly related to the origin_node_group.

### Creating a Commit Scope

Use `bdr.create_commit_scope` to add our example rule to a commit scope. For example:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := example_bdr_group,
    rule := MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT,
    wait_for_ready := true
);
```

This will add the rule `MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT` for any transaction originating from the `example_bdr_group` to a scope called `example_scope` .

If no rules previously existed in `example_scope` , then adding this rule would make the scope exist.

When a rule is added, the origin_node_group must already exist. If it does not, the whole add operation will be discarded with an error.

The rule will then be evaluated. If the rule mentions groups that don't exist or the settings on the group are incompatible with other configuration setting on the group's nodes, a warning will be emitted, but the rule will be added.

Once the rule is added, the commit scope will be available for use.

The `wait_for_ready` controls whether the `bdr.create_commit_scope()` call blocks until the rule has been added to the relevant nodes. The setting defaults to true and can be omitted.

### Using a commit scope

To use our example scope, we can set `bdr.commit_scope` within a transaction

```
BEGIN;
SET LOCAL bdr.commit_scope = example_scope;
...
COMMIT;
```

You must set the commit scope before the transaction writes any data.

You can set a commit scope as a default for a group or subgroup using `bdr.alter_node_group_option` :

```
SELECT bdr.alter_node_group_option(
  node_group_name := example_bdr_group,
  config_key := default_commit_scope,
```

(continues on next page)

(continued from previous page)

```
  config_value := example_scope
);
```

To completely clear the default for a group or subgroup, set the `default_commit_scope` value to `local` :

```
SELECT bdr.alter_node_group_option(
  node_group_name := example_bdr_group,
  config_key := default_commit_scope,
  config_value := local
);
```

You can also make this change using PGD CLI:

```
pgd set-group-options example-bdr-group --option default_commit_scope=example_scope
```

And you can clear the default using PGD CLI by setting the value to `local` :

```
pgd set-group-options example-bdr-group --option default_commit_scope=local
```

Finally, you can set the default commit_scope for a node using:

```
SET bdr.commit_scope = example_scope;
```

Set `bdr.commit_scope` to `local` to use the PGD default async replication.

### Predefined commit scopes

EDB Postgres Distributed (PGD) provides the following predefined commit scopes available for use:

- *local protect*
- *lag protect*
- *majority protect*
- *adaptive protect*

### local protect

```
ASYNCHRONOUS COMMIT
```

The `local protect` commit scope is the default commit scope for PGD Essential. It provides asynchronous commit with no durability guarantees. This means that transactions are considered committed as soon as they are written to the local node's WAL, without waiting for any confirmation from other nodes in the cluster.

This commit scope is suitable for scenarios where high availability and low latency are more important than data durability. However, it does not provide any guarantees against data loss in case of node failures or network issues.

### lag protect

```
MAJORITY ORIGIN GROUP LAG CONTROL (max_lag_time = 30s, max_commit_delay = 10s)
```

The `lag protect` commit scope provides a durability guarantee based on the lag time of the majority origin group. It ensures that transactions are considered committed only when the lag time is within a specified limit (30 seconds in this case) and the commit delay is also within a specified limit (10 seconds in this case). This helps to prevent data loss in case of network issues or node failures.

This commit scope is useful in scenarios where data consistency and durability are important, but some latency is acceptable. It allows for a balance between performance and data safety by ensuring that transactions are not considered committed until they have been confirmed by the majority of nodes in the origin group within the specified lag and commit delay limits.

### majority protect

```
MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT
```

The `majority protect` commit scope provides a durability guarantee based on the majority origin group. It ensures that transactions are considered committed only when they are confirmed by the majority of nodes in the origin group. This helps to ensure data consistency and durability in case of node failures or network issues.

This commit scope is suitable for scenarios where data consistency and durability are critical, and it provides a higher level of protection against data loss compared to the `local protect` commit scope. However, it may introduce some latency due to the need for confirmation from multiple nodes before considering a transaction as committed.

### adaptive protect

```
MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT DEGRADE ON (timeout = 10s, require_write_
↪lead = true) TO ASYNCHRONOUS COMMIT
```

The `adaptive protect` commit scope provides a more flexible durability guarantee. It allows transactions to be considered committed based on the majority origin group synchronous commit, but it can degrade to asynchronous commit if the transaction cannot be confirmed within a specified timeout (10 seconds in this case). This is useful in scenarios where network latency or node failures may cause delays in confirming transactions.

This commit scope is suitable for scenarios where data consistency and durability are important, but some flexibility is needed to handle potential delays. It provides a balance between performance and data safety by allowing transactions to be considered committed even if they cannot be confirmed by the majority of nodes within the specified timeout, while still providing a higher level of protection compared to the `local protect` commit scope.

### Examples

- Set the `lag protect` commit scope for a transaction using the `SET LOCAL` command:

```
BEGIN;
SET LOCAL bdr.commit_scope = lag protect;
-- Your transaction statements here
COMMIT;
```

This ensures that the transaction is committed with the specified commit scope, providing the desired level of durability and availability.

- Set the `majority protect` commit scope for the entire session using the `SET` command:

```
SET bdr.commit_scope = majority protect;
```

This ensures that all transactions executed in the session will use the specified commit scope, providing the desired level of durability and availability.

- Set the `adaptive protect` commit scope for a PGD group using the `bdr.alter_node_group_option()` function:

```
SELECT bdr.alter_node_group_option(
        node_group_name:=mygroup,
        config_key:=default_commit_scope,
        config_value:=adaptive protect);
```

This ensures that all transactions executed in the specified PGD group will use the specified commit scope, providing the desired level of durability and availability, unless overridden by a session or transaction-level setting.

### Commit scope groups

Before a transaction commits in PGD, one or more nodes must confirm it. The commit scope group is the part of a commit scope rule that specifies which nodes those are.

For example, for the following commit scope definition:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT,
    wait_for_ready := true
);
```

In the rule `MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT`, `SYNCHRONOUS COMMIT` is the commit scope kind, which controls how confirmation works. `MAJORITY ORIGIN GROUP` is the commit scope group, meaning a majority of nodes in the same subgroup as the origin node must confirm. For a full explanation of commit scope rule syntax, see *Commit scope rules* .

A commit scope group is made up of two parts: a quantifier (such as `MAJORITY` ), which sets the threshold, and a group specifier (such as `ORIGIN GROUP` ), which identifies which nodes to count.

### Quantifiers

A quantifier sets the confirmation threshold, determining how many nodes from the group specifier must confirm before the transaction can commit.

- `ALL` requires every node in the group to confirm. Any single node failure prevents new commits until the node recovers or the timeout expires.
- `MAJORITY` requires more than half of the nodes to confirm (2 out of 3, or 3 out of 5). This quantifier is the standard choice for high-availability configurations, as it tolerates the failure of fewer than half the nodes while maintaining quorum.
- `ANY n` requires at least `n` nodes to confirm. `ANY 1` means at least one node; `ANY 2` means at least two. This gives fine-grained control and is useful when you need at least one confirmation from a remote region without requiring a full majority there.

### Group specifiers

A group specifier identifies which nodes are eligible to confirm the transaction.

### Explicit group name

Use an explicit group name to target a specific, named PGD group that exists in the cluster. The behavior is fixed and predictable regardless of which node originates the transaction. For example, `MAJORITY (region_a)` , `ALL (region_a)` , or `ANY 2 (region_a)` .

### CLUSTER

`CLUSTER` selects from all data nodes in the cluster, across all subgroups. Use `CLUSTER` when you want the quorum to span the entire cluster rather than a single subgroup. For example, `MAJORITY CLUSTER` or `ALL CLUSTER`.

### ORIGIN_GROUP

`ORIGIN_GROUP` resolves dynamically to the subgroup that contains the node originating the transaction, adapting to wherever the transaction is being written rather than naming a fixed group. For example, `MAJORITY ORIGIN GROUP` or `ALL ORIGIN GROUP`.

> **Note**
>
> *ORIGIN_GROUP* and *ORIGIN GROUP* are interchangeable.

`ORIGIN_GROUP` is the standard choice for deployments with local routing, where each subgroup has its own write leader and applications connect to their local region. Rather than writing a separate rule for each subgroup, a single rule using `ORIGIN_GROUP` applies correctly regardless of which subgroup is the origin.

For example, when a transaction originates from a node in `region_a`, `ORIGIN_GROUP` resolves to `region_a`. When it originates from `region_b`, it resolves to `region_b`. The same commit scope works correctly in both cases.

### NOT ORIGIN_GROUP

`NOT ORIGIN_GROUP` resolves dynamically to all subgroups that do not contain the origin node. It's the complement of `ORIGIN_GROUP` and is designed to be used alongside it in multi-region rules. For example, `ANY 1 NOT ORIGIN_GROUP` or `ALL NOT ORIGIN_GROUP`.

`NOT ORIGIN_GROUP` isn't typically used alone. Its purpose is to add a cross-region confirmation requirement to a rule that already covers the local region via `ORIGIN_GROUP`, as in `MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT AND ANY 1 NOT ORIGIN GROUP SYNCHRONOUS COMMIT`.

### FOR EACH GROUP

`FOR EACH GROUP` with an explicit list requires the quantifier to be satisfied independently within each named group. For example, `MAJORITY FOR EACH GROUP (region_a, region_b)` means a majority of nodes in `region_a` and a majority of nodes in `region_b` must both confirm before the transaction commits.

`FOR EACH GROUP` is stricter than `MAJORITY (region_a, region_b)`, which pools nodes from both groups and requires a majority of the combined total. `FOR EACH GROUP` requires the threshold to be met within every group individually.

Use `FOR EACH GROUP` when each region must independently confirm the transaction, such as when regulatory requirements or availability guarantees require data to be durable in every region before a commit is acknowledged.

### FOR EACH GROUP IN CLUSTER

`FOR EACH GROUP IN CLUSTER` is the dynamic equivalent of `FOR EACH GROUP` with an explicit list. Rather than naming subgroups individually, it applies the quantifier to every subgroup in the cluster automatically. For example, `MAJORITY FOR EACH GROUP IN CLUSTER` or `ANY 1 FOR EACH GROUP IN CLUSTER`.

As subgroups are added or removed, the rule adapts without requiring a change to the commit scope definition. Use `FOR EACH GROUP IN CLUSTER` in clusters with three or more regions where the set of subgroups may change over time, or where you want a single commit scope definition to remain valid as the cluster grows.

### Combining groups

Combine group expressions with `AND` to require confirmation from multiple groups. Each expression is evaluated independently, and all must be satisfied before the transaction commits.

### Requiring a local majority plus a remote confirmation

The most common multi-region pattern requires a majority of nodes in the local region plus at least one confirmation from any other region:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := ha_scope,
    origin_node_group := top_group,
    rule := MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT AND ANY 1 NOT ORIGIN GROUP␣
↪SYNCHRONOUS COMMIT,
    wait_for_ready := true
);
```

Regardless of which subgroup originates the transaction, `ORIGIN GROUP` resolves to that subgroup and `NOT ORIGIN GROUP` resolves to all others. The rule always requires a local majority plus at least one confirmation from outside the local subgroup, ensuring every committed transaction has been confirmed in at least two regions. A single-region outage can't cause data loss, because the remote confirmation guarantees the transaction is durable elsewhere.

### Requiring all local nodes plus a remote confirmation

When subgroups are fixed and you want to be explicit about which groups participate:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := explicit_scope,
    origin_node_group := region_a,
    rule := ALL (region_a) SYNCHRONOUS COMMIT AND ANY 1 (region_b) SYNCHRONOUS COMMIT,
    wait_for_ready := true
);
```

All nodes in `region_a` plus any one node in `region_b` must confirm. When using `ALL` with explicit groups (rather than `ORIGIN_GROUP` ), this is typically the right structure for two-data-center configurations where both centers must be represented in every commit.

For the corresponding commit scope kinds that use these groups, see *QUORUM COMMIT* , and *SYNCHRONOUS COMMIT* .

### Commit scope rules

Commit scope rules are at the core of the commit scope mechanism. They define what the commit scope enforces.

Commit scope rules are composed of one or more operations that work in combination. Use an AND between rules.

Each operation is made up of two or three parts: the commit scope group, an optional confirmation level, and the kind of commit scope, which can have its own parameters.

```
commit_scope_group [ confirmation_level ] commit_scope_kind
```

A full formal syntax diagram is available in the Commit scopes reference.

A typical commit scope rule, such as `MAJORITY ORIGIN GROUP QUORUM COMMIT ABORT ON (timeout = 6s)` , can be broken down into its components. `MAJORITY ORIGIN GROUP` is the commit scope group, specifying which

nodes need to respond and confirm they processed the transaction. In this example, a majority of nodes in the origin node's group must confirm.

No confirmation level is specified, which means that the default (`visible` ) is used. With `visible` , nodes confirm once all the transaction's changes are flushed to disk and are visible to other transactions.

The last part of this operation is the commit scope kind, which in this example is `QUORUM COMMIT` . `QUORUM COMMIT` is a synchronous two-phase commit that coordinates across all participating nodes before any node commits. The `ABORT ON` clause is mandatory and sets how long to wait for quorum before aborting the transaction.

### The commit scope group

There are three kinds of commit scope groups: `ANY` , `ALL` , and `MAJORITY` . They're all followed by a list of one or more groups in parentheses. This list of groups combines to make a pool of nodes this operation applies to. This list can be preceded by `NOT` , which inverts the pool to be all other groups that aren't in the list.

- `ANY n` is followed by an integer value, `n` . It translates to any `n` nodes in the listed groups' nodes.
- `ALL` is followed by the groups and translates to all nodes in the listed groups' nodes.
- `MAJORITY` is followed by the groups and translates to requiring a half, plus one, of the listed groups' nodes to confirm, to give a majority.
- `ANY n NOT` is followed by an integer value, `n` . It translates to any `n` nodes that aren't in the listed groups' nodes.
- `ALL NOT` is followed by the groups and translates to all nodes that aren't in the listed groups' nodes.
- `MAJORITY NOT` is followed by the groups and translates to requiring a half, plus one, of the nodes that aren't in the listed groups' nodes to confirm, to give a majority.

All of the above expressions only consider data nodes in the groups in their evaluation. Witness nodes and other non-data nodes are ignored.

### The confirmation level

PGD nodes can send confirmations for a transaction at different times. In increasing levels of protection, from the perspective of the confirming node, these are:

- `received` _ A remote PGD node confirms the transaction immediately after receiving it, prior to starting the local application.
- `replicated` _ Confirms after applying changes of the transaction but before flushing them to disk.
- `durable` _ Confirms the transaction after all of its changes are flushed to disk.
- `visible` (default) _ Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions.

In rules for commit scopes, you can append these confirmation levels to the node group definition in parentheses with `ON` , as follows:

- `ANY 2 (right_dc) ON replicated`
- `ALL (left_dc) ON visible` (default)
- `ALL (left_dc) ON received AND ANY 1 (right_dc) ON durable`

**Note**

If you're familiar with PostgreSQL's *synchronous_standby_names* feature, be aware that while the grammar for *synchronous_standby_names* and commit scopes can look similar, there's a subtle difference. The former doesn't

account for the origin node, but the latter does. For example, *synchronous_standby_names = 'ANY 1 (..)'* is equivalent to a commit scope of *ANY 2 (...)* . This difference makes reasoning about majority easier and reflects that the origin node also contributes to the durability and visibility of the transaction.

### The commit scope kinds

Currently, there are four commit scope kinds. The following is a summary, with links to more details.

#### SYNCHRONOUS COMMIT

Synchronous Commit is a commit scope option that's designed to behave like the native Postgres `synchronous_commit` option, but is usable from within the commit scope environment. It's a synchronous non-two-phase commit operation that supports an optional `DEGRADE ON` clause. The commit scope group that comes before this option controls the groups and confirmation requirements the `SYNCHRONOUS COMMIT` uses.

For more details, see *SYNCHRONOUS COMMIT* .

#### QUORUM COMMIT

Quorum Commit is a synchronous two-phase commit that coordinates across all participating nodes before any node commits. All participating nodes commit together or roll back together. Before any node can commit locally, Quorum Commit uses eager conflict resolution to prevent inconsistencies rather than resolving them after the fact. Supported commit scope groups are `ALL` , `MAJORITY ORIGIN GROUP` , `MAJORITY CLUSTER` , and `MAJORITY FOR EACH GROUP` .

For more details, see *QUORUM COMMIT* .

#### CAMO

Commit At Most Once, or CAMO, allows the client/application, origin node, and partner node to ensure that a transaction is committed to the database at most once. Because the client is involved in the process, an application will require modifications to participate in the CAMO process.

For more details, see *Commit At Most Once* .

#### LAG CONTROL

With Lag Control, when the system's replication performance exceeds specified limits, a commit delay can be automatically injected into client interaction with the database, providing a back pressure on clients. Lag Control has parameters to set the maximum commit delay that can be exerted. It also has limits in terms of time to process or queue size that trigger increases in that commit delay.

For more details, see *LAG CONTROL* .

Commit scope rules are composed of one or more operations that work in combination. Use an AND to form a single rule. For example:

```
MAJORITY (Region_A) SYNCHRONOUS COMMIT AND ANY 1 (Region_A) LAG CONTROL (MAX_LAG_SIZE␣
↪= 50MB)
```

The first operation sets up a synchronous commit against a majority of `Region_A` . The second operation adds lag control that starts pushing the commit delay up when any one of the nodes in `Region_A` has more than 50MB of lag. This combination of operations allows the lag control to operate when any node is lagging.

### Comparing durability options

### Comparison

Most options for synchronous replication available to PGD allow for different levels of synchronization, offering different tradeoffs between performance and protection against node or network outages.

The following list of *The confirmation level* explains what a user should expect to see when that confirmation level is in effect and how that can affect performance, durability, and consistency.

!!!Note *QUORUM COMMIT* supports only `ON DURABLE` and `ON VISIBLE`. `ON VISIBLE` is the default.

#### ON RECEIVED

**Expect**: The peer node has received the changes. Nothing has been updated in the peer nodes tuple store or written to storage.

Confirmation on reception means that the peer operating normally can eventually, apply the transaction without requiring any further communication, even in the face of a full or partial network outage. A crash of a peer node might still require retransmission of the transaction, as this confirmation doesn't involve persistent storage.

**For**: The origin node in the transaction only has to wait for the reception of the transaction. Where transactions are large, it may improve the TPS performance of the system.

**Against**: An increased likelihood of stale reads. Overall, ON RECEIVED is not robust because data can be lost when either a Postgres server or operating system crash occurs.

#### ON REPLICATED

**Expect**: The peer node has received the changes and applied them to the tuple store. The changes have been written to storage, but the storage has not been flushed to disk.

Confirmation on replication means the peer has received and applied the changes. Those changes have been written to storage, but will still be in operating system caches and buffers. The system has yet to persist them to disk.

**For**: This checkpoint is further down the timeline of transaction processing. The origin node only waits for the transaction to be applied, but not persisted.

**Against**: There's a slightly lower chance of stale reads over ON RECEIVED. Also, with ON REPLICATED data can survive a Postgres crash but will still not survive an operating system crash.

#### ON DURABLE

**Expect**: The peer node has received the changes, applied them to the tuple store and persisted the changes to storage. It has yet to make the changes available to other sessions.

Durable confirmation means that the transaction has been written and flushed to the peer node's storage. This protects against loss of data after a crash and recovery of the peer node. But, if a session commits a transaction with an ON DURABLE rule before disconnecting and reconnecting, the transaction's changes are not guaranteed to be visible to the reconnected session.

When used with the Quorum Commit commit scope kind, this also means the changes are visible.

**For**: More robust, able to recover without retransmission in the event of a crash.

**Against**: Doesn't guarantee consistency in cases of failover.

### ON VISIBLE

**Expect**: The peer node has received and applied the changes, persisted and flushed those changes to storage.

Confirmation of visibility means that the transaction was fully applied remotely. If a session commits a transaction with an ON VISIBLE rule before disconnecting and reconnecting, the transaction's changes are guaranteed to be visible to the reconnected session.

**For**: Robust and consistent.

**Against**: Lower performance.

### Degrading commit scope rules

`SYNCHRONOUS COMMIT` and `CAMO` each have the optional capability of degrading the requirements for transactions when particular performance thresholds are crossed. `GROUP COMMIT` cannot degrade, but can abort on timing out.

When a node is applying a transaction and that transaction times out, it can be useful to trigger a process of degrading the requirements of the transaction to be completed, rather than just rolling back.

`DEGRADE ON` offers a route for gracefully degrading the commit scope rule of a transaction. At its simplest, `DEGRADE ON` takes a timeout and a second set of commit scope operations that the commit scope can gracefully degrade to.

For instance, after 20ms or 30ms timeout, the requirements for satisfying a commit scope could degrade from `ALL (node_group_name) SYNCHRONOUS COMMIT` to `MAJORITY (node_group_name) SYNCHRONOUS COMMIT` , making the transactions apply more steadily.

You can also require that the write leader be the originator of a transaction in order for the degrade clause to be triggered. This can be helpful in "split brain scenarios" where you have, say, 2 data nodes and a witness node. Supposing there is a network split between the two data nodes and you have connections to both of the data nodes, only one of them will be allowed to degrade, because only one of them will be elected leader through the raft election with the witness node.

### Behavior

There are two parts to how the generalized `DEGRADE` clause behaves as it is applied to transactions.

Once during the commit, while the commit being processed is waiting for responses that satisfy the commit scope rule, PGD checks for a timeout and, if the timeout has expired, the commit being processed is reconfigured to wait for the commit scope rule in the `DEGRADE` clause. In fact, by this point, the commit scope rule in the `DEGRADE` clause might already be satisfied.

This mechanism alone is insufficient for the intended behavior, as this alone would mean that every transaction—even those that were certain to degrade due to connectivity issues—must wait for the timeout to expire before degraded mode kicks in, which would severely affect performance in such degrading-cluster scenarios.

To avoid this, the PGD manager process also periodically (every 5s) checks the connectivity and apply rate (the one in bdr.node_replication_rates ) and if there are commit scopes that would degrade at that point based on the current state of replication, they will be automatically degraded—such that any transaction using that commit scope when processing after that uses the degraded rule instead of waiting for timeout—until the manager process detects that replication is moving swiftly enough again.

### SYNCHRONOUS COMMIT and GROUP COMMIT

Both `SYNCHRONOUS COMMIT` and `GROUP COMMIT` have `timeout` and `require_write_lead` parameters, with defaults of `0` and `false` respectively. You should probably always set the `timeout` , as the default of `0` causes an instant degrade. You can also require that the write leader be the originator of the transaction in order to switch to degraded mode (again, default is `false` ). For `SYNCHRONOUS COMMIT` the `timeout` and `require_write_lead` apply to degrade, and for `GROUP COMMIT` these parameters apply to abort. A `GROUP COMMIT` commit scope cannot degrade and

a `SYNCHRONOUS COMMIT` commit scope cannot abort, since it is already committed on the primary prior to waiting for confirmations from other nodes.

`SYNCHRONOUS COMMIT` also has options regarding which rule you can degrade to—which depends on which rule you are degrading from.

First of all, you can degrade to asynchronous operation:

```
ALL (left_dc) SYNCHRONOUS COMMIT DEGRADE ON (timeout=20s) TO ASYNC
```

You can also degrade to a less restrictive commit group with the same commit scope kind (again as long as the kind is either `SYNCHRONOUS_COMMIT` or `GROUP COMMIT` ). For instance, you can degrade as follows:

```
ALL (left_dc) SYNCHRONOUS COMMIT DEGRADE ON (timeout=20s) TO MAJORITY (left_dc)␣
↪SYNCHRONOUS COMMIT
```

or as follows:

```
ANY 3 (left_dc) SYNCHRONOUS COMMIT DEGRADE ON (timeout=20s) TO ANY 2 (left_dc)␣
↪SYNCHRONOUS COMMIT
```

But you cannot degrade from `SYNCHRONOUS COMMIT` to `GROUP COMMIT` .

### CAMO

While *Commit At Most Once* supports both the same `timeout` and `require_write_lead` parameters (with the same defaults, `0` and `false` respectively), the options are simpler in that you can only degrade to asynchronous operation.

```
ALL (left_dc) CAMO DEGRADE ON (timeout=20ms, require_write_lead=true) TO ASYNC
```

Again, you should set the `timeout` parameter, as the default is `0` .

### Monitoring degrade events

Use the bdr.stat_commit_scope to track degrade events. Three key metrics provide visibility into the degrade behavior:

- `ndegrades` tracks per-transaction degrade events. It increments each time a backend hits the degrade timeout during commit and successfully degrades that specific transaction. This metric shows the total number of individual transactions that have experienced degradation.
- `nconfig_degrades` tracks configuration-level state changes to the degraded mode. It increments when the background worker (manager process) switches the commit scope's shared state to degraded, based on node availability checks. This indicates how many times the commit scope configuration itself entered the degraded state, affecting all subsequent transactions until recovered.
- `last_state_change_time` records the timestamp of the last configuration-level state change (either entering or recovering from the degraded state). Use this metric to identify precisely when the commit scope last transitioned between normal and degraded operation, adding correlation with other system events or outages.

### Example monitoring query

View commit scope statistics including degrade events and the latest configuration-level state change:

```
SELECT
  commit_scope_name,
  ndegrades,
  nconfig_degrades,
```

(continues on next page)

(continued from previous page)

```
  last_state_change_time,
  CASE
    WHEN last_state_change_time IS NULL THEN never degraded
    WHEN age(now(), last_state_change_time) < interval 5 seconds THEN recently changed
    ELSE stable for  || age(now(), last_state_change_time)::text
  END AS state_change_info
FROM bdr.stat_commit_scope
WHERE ndegrades > 0 OR nconfig_degrades > 0
ORDER BY last_state_change_time DESC NULLS LAST;
```

These metrics together provide complete visibility into both immediate per-transaction fallback and longer-term state management degrade behavior, along with temporal context for troubleshooting.

### Synchronous Commit

Commit scope kind: `SYNCHRONOUS COMMIT`

### Overview

PGD's `SYNCHRONOUS COMMIT` is a commit scope kind that works in a way that's more like PostgreSQL's synchronous_commit option in its underlying operation. Unlike the PostgreSQL option, though, it's configured as a commit scope and is easier to configure and interact with in PGD.

Unlike other commit scope kinds, such as `GROUP COMMIT` and `CAMO` , the transactions in a `SYNCHRONOUS COMMIT` operation aren't transformed into a two-phase commit (2PC) transaction. They work more like a Postgres `synchronous_commit` .

### Example

In this example, when this commit scope is in use, any node in the `left_dc` group uses `SYNCHRONOUS COMMIT` to replicate changes to the other nodes in the `left_dc` group. It looks for a majority of nodes in the `left_dc` group to confirm that they committed the transaction.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_sc_scope,
    origin_node_group := left_dc,
    rule := MAJORITY (left_dc) SYNCHRONOUS COMMIT,
    wait_for_ready := true
);
```

### Configuration

`SYNCHRONOUS COMMIT` supports the optional `DEGRADE ON` clause. See the SYNCHRONOUS COMMIT commit scope reference for specific configuration parameters or see *Degrading commit scope rules* regarding Degrade on options.

### Confirmation

| Confirmation level | PGD Synchronous Commit handling |
|---|---|
| *received* | A remote PGD node confirms the transaction once it's been fully received and is in the in-memory write queue. |
| *replicated* | Same behavior as *received*. |
| *durable* | Confirms the transaction after all of its changes are flushed to disk. Analogous to *synchronous_commit = on* in legacy synchronous replication. |
| *visible* (default) | Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions. Analogous to *synchronous_commit = remote_apply* in legacy synchronous replication. |

### Details

Currently `SYNCHRONOUS COMMIT` doesn't use the confirmation levels of the commit scope rule syntax.

In commit scope rules, the original keyword `SYNCHRONOUS_COMMIT` is now aliased to `SYNCHRONOUS COMMIT` . The use of a space instead of an underscore helps distinguish it from Postgres's native `SYNCHRONOUS_COMMIT` .

### Quorum Commit

Commit scope kind: `QUORUM COMMIT`

### Overview

Standard synchronous replication confirms that a remote node received data, but it doesn't prevent two nodes from independently committing conflicting changes. Two nodes can each read the same row, pass their local validation checks, and both commit successfully, leaving the cluster in an inconsistent state with no error reported on either side. In a payments system, for example, two nodes each checking an account balance locally can both approve simultaneous withdrawals that together exceed the account limit.

Quorum Commit prevents this divergence by requiring a majority of nodes to agree before any node commits. If the cluster can't reach agreement, the transaction is rolled back everywhere. The result is a single, consistent state across all nodes, with no divergence and no conflicts to resolve after the fact.

Quorum Commit supersedes Group Commit, which is retained for backward compatibility only. Unlike Group Commit, which offered flexible syntax that could be misconfigured to allow inconsistency, Quorum Commit restricts syntax to what can guarantee consistency.

### When to use Quorum Commit

Use Quorum Commit for workloads where inconsistencies can't be resolved after they occur. Payments and core banking are the clearest fit, where a transaction that can't be confirmed across the cluster doesn't commit anywhere. Telecom billing, regulated industries with strict audit and compliance requirements, and migrations from proprietary databases that depend on global consistency guarantees are equally good candidates. These workloads have traditionally required proprietary databases. Quorum Commit brings the same guarantees to Postgres.

### How Quorum Commit works

Quorum Commit requires all participating nodes to agree before any node commits, using eager conflict resolution to prevent inconsistencies rather than resolving them after the fact. All writes must go through a single write leader, managed by PGD's *Overview of durability options* . If fewer than a majority of nodes are available, the cluster can't reach quorum and manual intervention is required.

### Consistency guarantees

We recommend Postgres 17 or later to ensure all guarantees hold. Earlier versions are supported, but not all guarantees apply. When used with a commit scope covering a majority or more of the nodes in the routing group, Quorum Commit provides the following guarantees:

- **Committed means committed everywhere.** A transaction commits only when the required number of nodes have committed it. When the application receives a commit acknowledgment, all required nodes hold the same resulting data.
- **The write leader always reflects the latest committed state.** A read from the write leader after a commit always returns the latest results. This guarantee holds across write leader changes. Transactions visible on the old write leader are visible on the new one.
- **Aborted means aborted everywhere.** When a transaction aborts, its effects are rolled back on all nodes. No partial state is visible anywhere in the cluster.
- **Zero recovery time objective (RTO0) on write leader change.** When the write leader changes, the new write leader can immediately accept transactions. In-flight transactions from the old write leader are resolved through reconciliation and don't block the new leader's operations.
- **No data divergence.** No node diverges from the committed state of the cluster. Broken replication due to conflicting commits can't occur.

!!!Warning To make the guarantees above possible, Quorum Commit doesn't support *Degrading commit scope rules* clauses or *Commit scope groups* weaker than `MAJORITY` .

### Eager conflict resolution

Rather than detecting conflicts after the fact, Quorum Commit aborts one of the conflicting transactions during the quorum phase, before either commits. Eager conflict resolution is always active with Quorum Commit and can't be disabled. Configure your application to handle serialization errors at commit time and retry failed transactions. See *QUORUM COMMIT* in the developer guide for a transaction template that wraps commits in a retry loop and handles serialization errors.

### Transaction lifecycle

Quorum Commit uses *Explicit two-phase commit (2PC)* internally. No node commits until quorum is achieved. In the prepare phase, the origin node prepares the transaction locally, making the data durable but not yet visible to other transactions. The prepared transaction is replicated to all participating nodes, each of which prepares its own local copy.

The origin node then collects confirmations from the required commit scope group. Once it receives the required number, it makes the commit decision and propagates it to all participating nodes, which then all commit together. When Raft consensus is used for the commit decision, PGD's built-in Raft makes the decision rather than the origin node.

Configure *max_prepared_transactions* to at least as high as `max_connections` to handle all Quorum Commit transactions originating per node.

### Automatic reconciliation

In a Quorum Commit transaction, the origin node makes the commit decision once it receives the required confirmations, then propagates that decision to all nodes via replication. If the origin fails after preparing a transaction but before that decision reaches all nodes, the remaining nodes are left holding a prepared transaction with no outcome.

Those prepared transactions hold locks and other resources and must be resolved before the cluster can make progress. The new write leader acts as coordinator and must arrive at the correct commit or abort decision for each prepared

transaction. If the coordinator commits a transaction the origin had already aborted, or aborts one the origin had already committed, the cluster diverges.

PGD prevents this divergence by resolving these transactions automatically through reconciliation, enabled by default via *bdr.enable_auto_sync_reconcile* .

### Reconciliation phases

1. When a node hasn't been heard from for `bdr.raft_global_election_timeout` (default 6s), the PGD manager process creates a reconciliation request, adding an entry to `bdr.reconcile_2pc_requests` for that origin with state `STARTED` .
2. A new write leader is elected via Raft consensus, typically the node furthest ahead in confirmed LSN from the origin and most likely to already hold the prepared transaction and any pre-commit decisions. This write leader acts as the coordinator for reconciliation.
3. The coordinator determines the commit or abort decision for each orphaned prepared transaction.

**With ``commit_decision = group`` (default):** The coordinator examines the commit decision (CD) cache.

- If a pre-commit decision is found in the CD cache, the origin had already decided to commit before it failed. Some nodes may have already committed it locally. The coordinator commits the transaction to bring all nodes into agreement.
- If a commit or abort decision is found in the CD cache, a definitive decision was already recorded, either through an earlier reconciliation round or via Raft. The coordinator follows that decision, committing or aborting locally as indicated.
- If no decision is found, the coordinator queries all nodes and checks whether any node has a commit decision or the prepared transaction. If a commit decision is found anywhere, the transaction is committed. Otherwise, the coordinator counts confirmations and commits the transaction if quorum is reached according to the commit scope rules.

**With ``commit_decision = raft`` :** Raft acts as the sole arbiter. The coordinator aborts all unresolved transactions, and any prior Raft decision takes precedence. If no decision was recorded, the transaction is cleanly aborted everywhere.

1. The coordinator sends the decision via Raft, and each node commits or aborts the transaction accordingly.
2. The transaction is resolved on all nodes. Locks and resources held by the prepared transaction are released.

If the coordinator itself fails during reconciliation, the next write leader takes over. The process is idempotent and guaranteed not to cause decision divergence. If a decision was already sent via Raft, the new coordinator finds it in the CD cache and follows it.

### Monitoring reconciliation

After a node failure, query *bdr.reconcile_2pc_requests_summary* to check whether reconciliation is in progress and which node is acting as coordinator. For ongoing health monitoring, *bdr.stat_commit_scope* tracks `nreconciliations` , `ncommitted` , and `naborted` per scope.

### Relationship to automatic synchronization

Reconciliation runs alongside automatic synchronization automatically in the background. Reconciliation resolves the transaction outcomes first, and synchronization then propagates those outcomes to any nodes that missed them.

### Commit latency

The pre-commit coordination adds roughly two network round trips compared to asynchronous replication. If one or more nodes are lagging, the delay in getting confirmations can be significant. Before a peer node can confirm, it also needs to apply the transaction locally, which adds latency proportional to the transaction size.

Logical standby nodes, subscriber-only nodes, and nodes still joining or catching up aren't included in the quorum but eventually receive changes. Witness nodes can't participate in the quorum and don't receive data changes.

### Timeout handling

If quorum isn't reached within the configured timeout, the transaction is aborted and rolled back on all participating nodes. A partial commit isn't possible. Either all nodes commit, or none do. Once aborted, the transaction isn't resubmitted when the network recovers. The application must retry it.

If a node fails during the quorum phase, the remaining nodes wait until the timeout expires before aborting. If the failed node recovers before the timeout, the quorum attempt can still succeed. Disconnected nodes keep their transactions in a prepared state and reconcile with the majority decision once they reconnect.

### Using Quorum Commit

Quorum Commit scopes are created with `bdr.create_commit_scope` and configured for your topology and routing mode. Once created, a scope can be applied as the default for a node group or set per session or per transaction.

!!!Note All nodes in the cluster must be running PGD 6.4.0 or later, including during a rolling upgrade.

### Creating a commit scope

The Quorum Commit syntax is:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := <scope_name>,
    origin_node_group := <node_group>,
    rule := <commit_scope_group> [ ON { durable | visible } ] QUORUM COMMIT
        [ ( commit_decision = { group | raft } ) ]
        ABORT ON ( timeout = <value> ),
    wait_for_ready := true
);
```

The `rule` parameter accepts the following options:

- **``<commit_scope_group>`` :** Defines which nodes must confirm the transaction before it commits. Supported values are `ALL` , `MAJORITY ORIGIN GROUP` , `MAJORITY CLUSTER` , `MAJORITY CLUSTER WITH LEADER` , `MAJORITY FOR EACH GROUP (subgroup1, subgroup2, ...)` , or `MAJORITY FOR EACH GROUP IN CLUSTER` . For details on commit scope groups, see *Commit scope groups* .

`MAJORITY` with an explicit group list, `ANY n` , and `NOT` aren't supported.

- **``commit_decision`` :** Optional. Controls how the commit decision is reached, and affects how *Automatic reconciliation* behaves after a node failure.
- `group` (default): The origin node decides to commit once it receives the required confirmations from the commit scope group.
- `raft` : The commit decision goes through PGD's built-in Raft consensus. Slower than `group` , but required for certain multi-group topologies.
- **``ON { durable | visible }`` :** Optional. Sets the confirmation level, which is the point at which a remote node confirms it has processed the transaction.

- `durable` : Confirms after all changes are flushed to disk.
- `visible` (default): Confirms after all changes are flushed to disk and are visible to concurrent transactions.
- **``ABORT ON ( timeout = <value> )`` :** Mandatory. Sets how long to wait for quorum before aborting. If consensus isn't reached within this period, the transaction is aborted and rolled back on all nodes and won't be resubmitted when the network recovers. Set `<value>` to an interval, for example `10s` . For guidance on choosing a value, see *Choosing a timeout* .

### Configuring by topology

The right Quorum Commit configuration depends on your cluster topology and routing mode.

### Single-location cluster (3 data nodes)

A cluster with three data nodes in a single subgroup, with routing enabled at the subgroup level. This topology tolerates the failure of one node at a time. We recommend `MAJORITY ORIGIN GROUP` , which requires a majority (two of three) nodes to confirm before the transaction commits. One node can fail without blocking writes.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY ORIGIN GROUP QUORUM COMMIT ABORT ON (timeout = 6s),
    wait_for_ready := true
);
```

### Two-region cluster (3+3 data nodes)

A cluster with two subgroups of three data nodes each. The cluster tolerates the failure of up to two nodes, one per region.

With local routing enabled, `MAJORITY FOR EACH GROUP IN CLUSTER` requires a majority within each subgroup to confirm. A network partition between regions causes transactions to wait up to the configured timeout before aborting. When the partition heals, writes resume automatically. Durability is guaranteed in both regions before the transaction commits, not only in the origin region.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY FOR EACH GROUP IN CLUSTER QUORUM COMMIT ABORT ON (timeout = 6s),
    wait_for_ready := true
);
```

With global routing enabled, `MAJORITY CLUSTER WITH LEADER` requires a majority across the entire cluster, including the write leader, to confirm before the transaction commits.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY CLUSTER WITH LEADER QUORUM COMMIT ABORT ON (timeout = 6s),
    wait_for_ready := true
);
```

### Three-region cluster (3+3+3 data nodes)

A cluster with three subgroups of three data nodes each.

With global routing, `MAJORITY CLUSTER WITH LEADER` requires a majority of all nine nodes, including the write leader, to confirm. This topology tolerates the failure of up to four nodes, including the loss of an entire region.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY CLUSTER WITH LEADER QUORUM COMMIT ABORT ON (timeout = 6s),
    wait_for_ready := true
);
```

With local routing, `MAJORITY FOR EACH GROUP IN CLUSTER` requires a majority within each region to confirm, making the transaction durable in all three regions before it's considered committed. This topology tolerates the failure of up to three nodes, one per region.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY FOR EACH GROUP IN CLUSTER QUORUM COMMIT ABORT ON (timeout = 6s),
    wait_for_ready := true
);
```

!!!Note Only test failure scenarios that the cluster topology is designed to tolerate. For example, don't test the failure of two regions in a three-region cluster, or the failure of more than one node in a single-location cluster. PGD makes no guarantees for degraded clusters.

### Using Raft for the commit decision

Adding `commit_decision = raft` to any of the configurations above routes the commit decision through PGD's built-in Raft consensus rather than the origin node directly. After receiving the required confirmations, the transaction commits only after Raft confirms the decision across all nodes. The choice of `commit_decision` also affects how *Automatic reconciliation* works after a node failure. With `group` , the coordinator must query nodes and count confirmations to determine the outcome. With `raft` , Raft acts as the sole arbiter and the coordinator can resolve all unresolved transactions immediately, since any prior Raft decision takes precedence.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_group,
    rule := MAJORITY ORIGIN GROUP QUORUM COMMIT (commit_decision = raft) ABORT ON␣
↪(timeout = 6s),
    wait_for_ready := true
);
```

### Choosing a timeout

The `ABORT ON (timeout = <value>)` clause is mandatory for Quorum Commit. If quorum isn't reached within the timeout, the transaction is aborted and rolled back on all nodes and won't be resubmitted when connectivity recovers.

When using local routing, set the timeout to be shorter than `bdr.raft_group_election_timeout` (default 3s). If a write leader is unresponsive for longer than the election timeout, a new write leader is elected. Timing out the transaction before that point avoids races between an in-flight transaction and the incoming write leader.

When using global routing, set the timeout to be shorter than `bdr.raft_global_election_timeout` (default 6s). If a node isn't heard from for the global election timeout, reconciliation of its transactions begins. Setting a timeout of 6s or less ensures a transaction is cleanly aborted before write leader reconciliation starts.

### Setting a default commit scope

A commit scope can be set as the default for a node group using `bdr.alter_node_group_option` . All transactions on nodes in that group then use the commit scope automatically, ensuring consistent protection across all workloads.

```
SELECT bdr.alter_node_group_option(sg1, default_commit_scope, majority_qc);
```

!!!Important Quorum Commit doesn't guarantee consistency when transactions using it execute concurrently on the same tables as transactions that don't. Setting a default commit scope on the node group ensures all transactions get the consistency benefit.

### Writing applications for Quorum Commit

Quorum Commit requires a fast track writer enabled on every subscription. Before using a Quorum Commit scope with any transaction, enable the :ref:``enable_fast_track_writer` notes <*enable_fast_track_writer* notes>` option on the top-level group so the entire cluster has it enabled:

```
SELECT bdr.alter_node_group_option(topgroup, enable_fast_track_writer, true);
```

Because Quorum Commit uses eager conflict resolution, applications encounter serialization errors more often than with asynchronous replication. Wrap transactions in a retry loop to handle serialization errors:

```
BEGIN;
SET LOCAL bdr.commit_scope = banking_scope;
- - ... application queries ...
COMMIT;
```

If `COMMIT` returns a serialization error, re-execute the transaction from `BEGIN` . The retry rate depends on the workload. Large transactions that modify many rows or overlap heavily with concurrent transactions retry more often.

Set the timeout below `bdr.raft_global_election_timeout` (default 6s) when using global routing, or below `bdr.raft_group_election_timeout` (default 3s) when using local routing. For more details, see *Choosing a timeout* .

### Compatibility

Quorum Commit is compatible with *Parallel Apply* . You don't need to disable Parallel Apply when using Quorum Commit.

Quorum Commit is compatible with *Using with transaction streaming* in `file` and `off` modes only.

### Limitations

See the Quorum Commit section of *Known issues and limitations* .

### Commit At Most Once

Commit scope kind: `CAMO`

### Overview

The objective of the Commit At Most Once (CAMO) feature is to prevent the application from committing more than once.

Without CAMO, when a client loses connection after a `COMMIT` is submitted, the application might not receive a reply from the server and is therefore unsure whether the transaction committed.

The application can't easily decide between the two options of:

- Retrying the transaction with the same data, since this can in some cases cause the data to be entered twice
- Not retrying the transaction and risk that the data doesn't get processed at all

Either of those is a critical error with high-value data.

One way to avoid this situation is to make sure that the transaction includes at least one `INSERT` into a table with a unique index. However, that depends on the application design and requires application-specific error-handling logic, so it isn't effective in all cases.

The CAMO feature in PGD offers a more general solution and doesn't require an `INSERT` . When activated by `bdr.commit_scope` , the application receives a message containing the transaction identifier, if already assigned. Otherwise, the first write statement in a transaction sends that information to the client.

If the application sends an explicit `COMMIT` , the protocol ensures that the application receives the notification of the transaction identifier before the `COMMIT` is sent. If the server doesn't reply to the `COMMIT` , the application can handle this error by using the transaction identifier to request the final status of the transaction from another PGD node. If the prior transaction status is known, then the application can safely decide whether to retry the transaction.

CAMO works by creating a pair of partner nodes that are two PGD nodes from the same PGD group. In this operation mode, each node in the pair knows the outcome of any recent transaction executed on the other peer and especially (for our need) knows the outcome of any transaction disconnected during `COMMIT` . The node that receives the transactions from the application might be referred to as "origin" and the node that confirms these transactions as "partner." However, there's no difference in the CAMO configuration for the nodes in the CAMO pair. The pair is symmetric.

!!!Warning

CAMO requires changes to the user's application to take advantage of the advanced error handling. Enabling a parameter isn't enough to gain protection. Reference client implementations are provided to customers on request.

To use CAMO, an application must issue an explicit `COMMIT` message as a separate request, not as part of a multi-statement request. CAMO can't provide status for transactions issued from procedures or from single-statement transactions that use implicit commits.

### Configuration

See the CAMO commit scope reference for configuration parameters.

### Confirmation

| Confirmation Level | CAMO handling |
|---|---|
| *received* | Not applicable, only uses the default, *VISIBLE*. |
| *replicated* | Not applicable, only uses the default, *VISIBLE*. |
| *durable* | Not applicable, only uses the default, *VISIBLE*. |
| *visible* (default) | Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions. |

### Limitations

See the CAMO section of *Limitations* .

### Failure scenarios

Different failure scenarios occur in different configurations.

### Data persistence at receiver side

By default, a PGL writer operates in `bdr.synchronous_commit = off` mode when applying transactions from remote nodes. This holds true for CAMO as well, meaning that transactions are confirmed to the origin node possibly before reaching the disk of the CAMO partner. In case of a crash or hardware failure, a confirmed transaction might be unrecoverable on the CAMO partner by itself. This isn't an issue as long as the CAMO origin node remains operational, as it redistributes the transaction once the CAMO partner node recovers.

This in turn means CAMO can protect against a single-node failure, which is correct for local mode as well as or even in combination with remote write.

To cover an outage of both nodes of a CAMO pair, you can use `bdr.synchronous_commit = local` to enforce a flush prior to the pre-commit confirmation. This doesn't work with either remote write or local mode and has a performance impact due to I/O requirements on the CAMO partner in the latency sensitive commit path.

### Asynchronous mode

When the `DEGRADE ON ... TO ASYNC` clause is used in the commit scope, a node detects whether its CAMO partner is ready. If not, it temporarily switches to asynchronous (local) mode. When in this mode, a node commits transactions locally until switching back to CAMO mode.

This doesn't allow COMMIT status to be retrieved, but it does let you choose availability over consistency. This mode can tolerate a single-node failure. In case both nodes of a CAMO pair fail, they might choose incongruent commit decisions to maintain availability, leading to data inconsistencies.

For a CAMO partner to switch to ready, it needs to be connected, and the estimated catchup interval needs to drop below the `timeout` value of `TO ASYNC` . You can check the current readiness status of a CAMO partner with bdr.is_camo_partner_ready() , while bdr.node_replication_rates provides the current estimate of the catchup time.

The switch from CAMO-protected to asynchronous mode is only ever triggered by an actual CAMO transaction. This is true either because the commit exceeds the `timeout` value of `TO ASYNC` or, in case the CAMO partner is already known, disconnected at the time of commit. This switch is independent of the estimated catchup interval. If the CAMO pair is configured to require the current node to be the write lead of a group as configured through the `enable_routing` node group option. See *Commit scopes* for syntax. This can prevent a split brain situation due to an isolated node from switching to asynchronous mode. If `enable_routing` isn't set for the CAMO group, the origin node switches to asynchronous mode immediately.

The switch from asynchronous mode to CAMO mode depends on the CAMO partner node, which initiates the connection. The CAMO partner tries to reconnect at least every 30 seconds. After connectivity is reestablished, it might therefore take up to 30 seconds until the CAMO partner connects back to its origin node. Any lag that accumulated on the CAMO partner further delays the switch back to CAMO protected mode.

Unlike during normal CAMO operation, in asynchronous mode there's no added commit overhead. This can be problematic, as it allows the node to continuously process more transactions than the CAMO pair can normally process. Even if the CAMO partner eventually reconnects and applies transactions, its lag only ever increases in such a situation, preventing reestablishing the CAMO protection. To artificially throttle transactional throughput, PGD provides the bdr.camo_local_mode_delay setting, which allows you to delay a `COMMIT` in local mode by an arbitrary amount of time. We recommend measuring commit times in normal CAMO mode during expected workloads and configuring this delay accordingly. The default is 5 ms, which reflects a asynchronous network and a relatively quick CAMO partner response.

Consider the choice of whether to allow asynchronous mode in view of the architecture and the availability requirements. The following examples provide some detail.

### Example

This example considers a setup with two PGD nodes that are the CAMO partner of each other:

```
- - create a CAMO commit scope for a group over
- - a definite pair of nodes
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := camo_dc,
    rule := ALL (left_dc) CAMO DEGRADE ON (timeout=500ms) TO ASYNC
);
```

For this CAMO commit scope to be legal, the number of nodes in the group must equal exactly 2. Using ALL or ANY 2 on a group consisting of several nodes is an error because the unquantified group expression doesn't resolve to a definite pair of nodes.

### With asynchronous mode

If asynchronous mode is allowed, there's no single point of failure. When one node fails:

- The other node can determine the status of all transactions that were disconnected during `COMMIT` on the failed node.
- New write transactions are allowed. If the second node also fails, then the outcome of those transactions that were being committed at that time is unknown.

### Without asynchronous mode

If asynchronous mode isn't allowed, then each node requires the other node for committing transactions, that is, each node is a single point of failure. When one node fails:

- The other node can determine the status of all transactions that were disconnected during `COMMIT` on the failed node.
- New write transactions are prevented until the node recovers.

### Application use

### Overview and requirements

CAMO relies on a retry loop and specific error handling on the client side. There are three aspects to it:

- The result of a transaction's `COMMIT` needs to be checked and, in case of a temporary error, the client must retry the transaction.
- Prior to `COMMIT` , the client must retrieve a global identifier for the transaction, consisting of a node id and a transaction id (both 32-bit integers).
- If the current server fails while attempting a `COMMIT` of a transaction, the application must connect to its CAMO partner, retrieve the status of that transaction, and retry depending on the response.

The application must store the global transaction identifier only for the purpose of verifying the transaction status in case of disconnection during `COMMIT` . In particular, the application doesn't need another persistence layer. If the application fails, it needs only the information in the database to restart.

To illustrate this, this example shows a retry loop in a CAMO-aware client application, written in a C-like pseudo-code. It expects two DSNs, `origin_dsn` and `partner_dsn` , providing connection information. These usually are the same DSNs as used for the initial call to `bdr.create_node` and can be looked up in `bdr.node_summary` , column `interface_connstr` .

```
PGconn *conn = PQconnectdb(origin_dsn);
```

The process starts connecting to the origin node. Now enter the loop:

```
loop {
    PQexec(conn, "BEGIN");
```

Next, start the transaction and begin populating it with changes:

```
PQexec(conn, "INSERT INTO ...");
...
```

Once you're done, you need to make a record of the local node id and the transaction id. Both are available as parameters.

```
node_id = PQparameterStatus(conn, "bdr.local_node_id");
xid = PQparameterStatus(conn, "transaction_id");
```

Now it's ready to try to commit.

```
PQexec(conn, "COMMIT");
if (PQresultStatus(res) == PGRES_COMMAND_OK)
    return SUCCESS;
```

If the result is `PGRES_COMMAND_OK` , that's good, and you can move on. But if it isn't, you need to use CAMO to track the transaction to completion. The first question to ask is, "Was the connection bad?"

```
else if (PQstatus(res) == CONNECTION_BAD)
{
```

If it was a bad connection, then you can check on the CAMO partner node to see if the transaction made it there.

```
conn = PQconnectdb(partner_dsn);
if (!connectionEstablished())
    panic();
```

If you can't connect to the partner node, there's not a lot you can do. In this case, panic, or take similar actions.

But if you can connect, you can use bdr.logical_transaction_status() to find out how the transaction did. The code recorded the required values, node_id and xid (the transaction id), just before committing the transaction.

```
    sql = "SELECT bdr.logical_transaction_status($node_id, $xid)";
    txn_status = PQexec(conn, sql);
    if (txn_status == "committed")
        return SUCCESS;
    else
        continue;    // to retry the transaction on the partner
}
```

If the transaction reports it's been committed, then you can call this transaction a success. No more action is required. If, on the other hand, it doesn't report it's been committed, continue in the loop so the transaction can be retried on the partner node.

```
    else
    {
        if (isPermanentError())
            return FAILURE;
        else
        {
            sleep(increasing_retry_delay);

            continue;
        }
    }
}
```

If status of the transaction wasn't success or bad connection, check if the problem was a permanent error. If so, report a failure of the transaction. If not, you can still retry it. Have the code sleep for a period of time that increases with each retry, and then retry the transaction.

### Working with the CAMO partner

role assigned to them.

bdr.is_camo_partner_connected() allows checking the connection status of a CAMO partner node.

To check that the CAMO partner is ready, use the function bdr.is_camo_partner_ready . Underneath, this triggers the switch to and from local mode.

To find out more about the configured CAMO partner, use bdr.get_configured_camo_partner() . This function returns the local node's CAMO partner.

You can wait on the CAMO partner to process the queue with the function

bdr.wait_for_camo_partner_queue() . This function is a wrapper of bdr.wait_for_apply_queue . The difference is that bdr.wait_for_camo_partner_queue()

defaults to querying the CAMO partner node. It returns an error if the local node isn't part of a CAMO pair.

To check the status of a transaction that was being committed when the node failed, the application must use the function bdr.logical_transaction_status() .

You pass this function the node_id and transaction_id of the transaction you want to check on. You can use this function only on a node that's part of a CAMO pair.

In all cases, you must call the function within 15 minutes after of issuing the commit. The CAMO partner must regularly purge such meta-information and therefore can't provide correct answers for older transactions.

Before querying the status of a transaction, this function waits for the receive queue to be consumed and fully applied. This mechanism prevents early negative answers for transactions that were received but not yet applied.

Despite its name, it's not always a read-only operation. If the status is unknown, the CAMO partner decides whether to commit or abort the transaction, storing that decision locally to ensure consistency going forward.

The client must not call this function before attempting to commit on the origin. Otherwise the transaction might be forced to roll back.

### Connection pools and proxies

Consider the effect of connection pools and proxies when designing a CAMO cluster. A proxy might freely distribute transactions to all nodes in the commit group, that is, to both nodes of a CAMO pair.

Take care to ensure that the application fetches the proper node id. When using session pooling, the client remains connected to the same node, so the node id remains constant for the lifetime of the client session. However, with finer-grained transaction pooling, the client needs to fetch the node id for every transaction, as in the example that follows.

A client that isn't directly connected to the PGD nodes might not even notice a failover or switchover. But it can always use the `bdr.local_node_id` parameter to determine the node it's currently connected to. In the crucial situation of a disconnect during COMMIT, the proxy must properly forward that disconnect as an error to the client applying the CAMO protocol.

For CAMO in `received` mode, a proxy that potentially switches between the CAMO pairs must use the `bdr.wait_for_camo_partner_queue` function to prevent stale reads.

### CAMO limitations

CAMO limitations are covered in *Known issues and limitations* .

### Performance implications

CAMO extends the Postgres replication protocol by adding a message roundtrip at commit. Applications have a higher commit latency than with asynchronous replication, mostly determined by the round-trip time between involved nodes. Increasing the number of concurrent sessions can help to increase parallelism to obtain reasonable transaction throughput.

The CAMO partner confirming transactions must store transaction states. Compared to non-CAMO operation, this might require an added seek for each transaction applied from the origin.

### Client application testing

Proper use of CAMO on the client side isn't trivial. We strongly recommend testing the application behavior with the PGD cluster against failure scenarios, such as node crashes or network outages.

### Lag Control

Commit scope kind: `LAG CONTROL`

### Overview

Lag Control provides a mechanism where, if replication is running outside of limits set, a delay is injected into the origin node's client connections after processing transactions that make replicable updates. This delay is designed to slow the incoming transactions and bring replication back within the defined limits.

### Background

The data throughput of database applications on a PGD origin node can exceed the rate at which committed data can replicate to downstream peer nodes.

If this imbalance persists, it can put satisfying organizational objectives, such as RPO, RCO, and GEO, at risk.

- **Recovery point objective (RPO)** specifies the maximum-tolerated amount of data that can be lost due to unplanned events, usually expressed as an amount of time. In PGD, RPO determines the acceptable amount of committed data that hasn't been applied to one or more peer nodes.
- **Resource constraint objective (RCO)** acknowledges that finite storage is available. In PGD, the demands on these storage resources increase as lag increases.
- **Group elasticity objective (GEO)** ensures that any node isn't originating new data at a rate that can't be saved to its peer nodes.

To allow organizations to achieve their objectives, PGD offers Lag Control. This feature provides a means to precisely regulate the potential imbalance without intruding on applications. It does so by transparently introducing a delay to READ WRITE transactions that modify data. This delay, the PGD commit delay, starts at 0ms.

Using the LAG CONTROL commit scope kind, you can set a maximum time that commits can be delayed between nodes in a group, maximum lag time, or maximum lag size (based on the size of the WAL).

If the nodes can process transactions within the specified maximums on enough nodes, the PGD commit delay will stay at 0ms or be reduced toward 0ms. If the maximums are exceeded on enough nodes, though, the PGD commit delay on the originating node is increased. It will continue increasing until the Lag Control constraints are met on enough nodes again.

The PGD commit delay happens after a transaction has completed and released all its locks and resources. This timing of the delay allows concurrent active transactions to carry on observing and modifying the delayed transactions values and acquiring its resources.

Strictly speaking, the PGD commit delay isn't a per-transaction delay. It's the mean value of commit delays over a stream of transactions for a particular client connection. This technique allows the commit delay and fine-grained adjustments of the value to escape the coarse granularity of OS schedulers, clock interrupts, and variation due to system load. It also allows the PGD runtime commit delay to settle within microseconds of the lowest duration possible to maintain a lag measure threshold.

Postgres commit delay . They are unrelated and perform different functions. Don't substitute one for the other.

### Requirements

To get started using Lag Control:

- Determine the maximum acceptable commit delay time `max_commit_delay` that all database applications can tolerate.
- Decide on the lag measure to use. Choose either lag size `max_lag_size` or lag time `max_lag_time` .
- Decide on the groups or subgroups involved and the minimum number of nodes in each collection required to satisfy confirmation. This information forms the basis for the definition of a commit scope rule.

### Configuration

You specify Lag Control in a commit scope, which allows consistent and coordinated parameter settings across the nodes spanned by the commit scope rule. You can include a Lag Control specification in the default commit scope of a top group or as part of an origin group commit scope.

As in example, take a configuration with two datacenters, `left_dc` and `right_dc` , represented as subgroups:

```
SELECT bdr.create_node_group(
    node_group_name := left_dc,
    parent_group_name := top_group,
    join_node_group := false
);
SELECT bdr.create_node_group(
    node_group_name := right_dc,
    parent_group_name := top_group,
    join_node_group := false
);
```

The following code adds Lag Control rules for those two data centers, using individual rules for each subgroup:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s) AND␣
↪ANY 1 (right_dc) LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s),
    wait_for_ready := true
);
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) LAG CONTROL (max_commit_delay=0.250ms, max_lag_
↪size=100MB) AND ALL (right_dc) LAG CONTROL (max_commit_delay=0.250ms, max_lag_
↪size=100MB),
    wait_for_ready := true
);
```

You can add a Lag Control commit scope rule to existing commit scope rules that also include Group Commit and CAMO rule specifications.

The `max_commit_delay` is an interval, typically specified in milliseconds (1ms). Using fractional values for sub-millisecond precision is supported.

The `max_lag_size` is an integer that specifies the maximum allowed lag in terms of WAL bytes.

The `max_lag_time` is an interval, typically specified in seconds, that specifies the maximum allowed lag in terms of time.

The maximum commit delay (`max_commit_delay` ) is a ceiling value representing a hard limit, which means that a commit delay never exceeds the configured value.

The maximum lag size and time (`max_lag_size` and `max_lag_time` ) are soft limits that can be exceeded. When the maximum commit delay is reached, there's no additional back pressure on the lag measures to prevent their continued increase.

### Confirmation

| Confirmation level | Lag Control handling |
|---|---|
| *received* | Not applicable, only uses the default, *VISIBLE*. |
| *replicated* | Not applicable, only uses the default, *VISIBLE*. |
| *durable* | Not applicable, only uses the default, *VISIBLE*. |
| *visible* (default) | Not applicable, only uses the default, *VISIBLE*. |

### Transaction application

The PGD commit delay is applied to all READ WRITE transactions that modify data for user applications. This behavior implies that any transaction that doesn't modify data, including declared READ WRITE transactions, is exempt from the commit delay.

Asynchronous transaction commit also executes a PGD commit delay. This might appear counterintuitive, but asynchronous commit, by virtue of its performance, can be one of the greatest sources of replication lag.

Postgres and PGD auxillary processes don't delay at transaction commit. Most notably, PGD writers don't execute a commit delay when applying remote transactions on the local node. This is by design, as PGD writers contribute nothing to outgoing replication lag and can reduce incoming replication lag the most by not having their transaction commits throttled by a delay.

### Limitations

The maximum commit delay is a ceiling value representing a hard limit, which means that a commit delay never exceeds the configured value. Conversely, the maximum lag measures both by size and time and are soft limits that can be exceeded. When the maximum commit delay is reached, there's no additional back pressure on the lag measures to prevent their continued increase.

There's no way to exempt origin transactions that don't modify PGD replication sets from the commit delay. For these transactions, it can be useful to SET LOCAL the maximum transaction delay to 0.

### Caveats

Application TPS is one of many factors that can affect replication lag. Other factors include the average size of transactions for which PGD commit delay can be less effective. In particular, bulk load operations can cause replication lag to rise, which can trigger a concomitant rise in the PGD runtime commit delay beyond the level reasonably expected by normal applications, although still under the maximum allowed delay.

Similarly, an application with a very high OLTP requirement and modest data changes can be unduly restrained by the acceptable PGD commit delay setting.

In these cases, it can be useful to use the `SET [SESSION|LOCAL]` command to custom configure Lag Control settings for those applications or modify those applications. For example, bulk load operations are sometimes split into multiple smaller transactions to limit transaction snapshot duration and WAL retention size or establish a restart point if the bulk load fails. In deference to Lag Control, those transaction commits can also schedule very long PGD commit delays to allow digestion of the lag contributed by the prior partial bulk load.

### Meeting organizational objectives

In the example objectives listed earlier:

- RPO can be met by setting an appropriate maximum lag time.
- RCO can be met by setting an appropriate maximum lag size.
- GEO can be met by monitoring the PGD runtime commit delay and the PGD runtime lag measures,

As mentioned, when the maximum PGD runtime commit delay is pegged at the PGD-configured commit-delay limit, and the lag measures consistently exceed their PGD-configured maximum levels, this scenario can be a marker for PGD group expansion.

### Lag Control and extensions

The PGD commit delay is a post-commit delay. It occurs after the transaction has committed and after all Postgres resources locked or acquired by the transaction are released. Therefore, the delay doesn't prevent concurrent active transactions from observing or modifying its values or acquiring its resources. The same guarantee can't be made for external resources managed by Postgres extensions. Regardless of extension dependencies, the same guarantee can be made if the PGD extension is listed before extension-based resource managers in postgresql.conf.

### Group Commit (legacy)

Commit scope kind: `GROUP COMMIT`

### Overview

The goal of Group Commit is to protect against data loss in case of single node failures or temporary outages. You achieve this by requiring more than one PGD node to successfully confirm a transaction at COMMIT time. Confirmation can be sent at a number of points in the transaction processing but defaults to "visible" when the transaction has been flushed to disk and is visible to all other transactions.

### Example

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) GROUP COMMIT(commit_decision=raft) AND ANY 1 (right_dc)␣
↪GROUP COMMIT,
    wait_for_ready := true
);
```

This example creates a commit scope where all the nodes in the `left_dc` group and any one of the nodes in the `right_dc` group must receive and successfully confirm a committed transaction.

### Requirements

During normal operation, Group Commit is transparent to the application. Transactions that were in progress during failover need the reconciliation phase triggered or consolidated by either the application or a proxy in between. This activity currently happens only when either the origin node recovers or when it's parted from the cluster. This behavior is the same as with Postgres legacy built-in synchronous replication.

Transactions committed with Group Commit use *two-phasecommit* underneath. Therefore, configure `max_prepared_transactions` high enough to handle all such transactions originating per node.

### Limitations

See the Group Commit section of *Known issues and limitations* .

### Configuration

`GROUP_COMMIT` supports optional `GROUP COMMIT` parameters, as well as `ABORT ON` and `DEGRADE ON` clauses. For a full description of configuration parameters, see the GROUP_COMMIT commit scope reference or for more regarding `DEGRADE ON` options in general, see the *Degrading commit scope rules* section.

### Confirmation

| Confirmation level | Group Commit handling |
|---|---|
| *received* | A remote PGD node confirms the transaction immediately after receiving it, prior to starting the local application. |
| *replicated* | Confirms after applying changes of the transaction but before flushing them to disk. |
| *durable* | Confirms the transaction after all of its changes are flushed to disk. |
| *visible* (default) | Confirms the transaction after all of its changes are flushed to disk and it's visible to concurrent transactions. |

### Behavior

The behavior of Group Commit depends on the configuration applied by the commit scope.

### Commit decisions

You can configure Group Commit to decide commits in three different ways: `group` , `partner` , and `raft` .

The `group` decision is the default. It specifies that the commit is confirmed by the origin node upon receiving as many confirmations as required by the commit scope group. The difference is that the commit decision is made based on PREPARE replication while the durability checks COMMIT (PREPARED) replication.

The `partner` decision is what *Commit At Most Once* (CAMO) uses. This approach works only when there are two data nodes in the node group. These two nodes are partners of each other, and the replica rather than origin decides whether to commit something. This approach requires application changes to use the CAMO transaction protocol to work correctly, as the application is in some way part of the consensus. For more on this approach, see *Commit At Most Once* .

The `raft` decision uses PGDs built-in Raft consensus for commit decisions. Use of the `raft` decision can reduce performance. It's currently required only when using `GROUP COMMIT` with an ALL commit scope group.

Using an ALL commit scope group requires that the commit decision must be set to `raft` to avoid *Transaction reconciliation* issues.

### Conflict resolution

Conflict resolution can be `async` or `eager` .

Async means that PGD does optimistic conflict resolution during replication using the row-level resolution as configured for a given node. This happens regardless of whether the origin transaction committed or is still in progress. See *bdr_read_all_conflicts* for details about how the asynchronous conflict resolution works.

Eager means that conflicts are resolved eagerly (as part of agreement on COMMIT), and conflicting transactions get aborted with a serialization error. This approach provides greater isolation than the asynchronous resolution at the price of performance.

Using an ALL commit scope group requires that the *Commit decisions* must be set to `raft` to avoid reconciliation issues.

For details about how Eager conflict resolution works, see *Eager conflict resolution* .

### Aborts

To prevent a transaction that can't get consensus on the COMMIT from hanging forever, the `ABORT ON` clause allows specifying timeout. After the timeout, the transaction abort is requested. If the transaction is already decided to be committed at the time the abort request is sent, the transaction does eventually COMMIT even though the client might receive an abort message.

See also *Limitations* .

### Transaction reconciliation

A Group Commit transaction's commit on the origin node is implicitly converted into a two-phase commit.

In the first phase (prepare), the transaction is prepared locally and made ready to commit. The data is made durable but is uncomitted at this stage, so other transactions can't see the changes made by this transaction. This prepared transaction gets copied to all remaining nodes through normal logical replication.

The origin node seeks confirmations from other nodes, as per rules in the Group Commit grammar. If it gets confirmations from the minimum required nodes in the cluster, it decides to commit this transaction moving onto the second phase (commit). In the commit phase, it also sends this decision by way of replication to other nodes. Those nodes will also eventually commit on getting this message.

There's a possibility of failure at various stages. For example, the origin node may crash after preparing the transaction. Or the origin and one or more replicas may crash.

This leaves the prepared transactions in the system. The `pg_prepared_xacts` view in Postgres can show prepared transactions on a system. The prepared transactions might be holding locks and other resources. To release those locks and resources, either abort or commit the transaction. That decision must be made with a consensus of nodes.

When `commit_decision` is `raft` , then, Raft acts as the coordinator, and these transactions are eventually reconciled automatically.

When the `commit_decision` is `group` , then, transactions don't use Raft. Instead the write lead in the cluster performs the role of coordinator. This is because it's the node that's most ahead with respect to changes in its subgroup. It detects when a node is down and initiates reconciliation for such a node by looking for prepared transactions it has with the down node as the origin.

For all such transactions, it sees if the nodes as per the rules of the commit scope have the prepared transaction, it takes a decision. This decision is conveyed over Raft and needs the majority of the nodes to be up to do reconciliation.

This process happens in the background. There's no command for you to use to control or issue this.

### Eager conflict resolution

Eager conflict resolution (also known as Eager Replication) prevents conflicts by aborting transactions that conflict with each other with serializable errors during the COMMIT decision process.

You configure it using *Commit scopes* as one of the conflict resolution options for *DDL overview* .

### Usage

To enable Eager conflict resolution, the client needs to switch to a commit scope, which uses it at session level or for individual transactions as shown here:

```
BEGIN;

SET LOCAL bdr.commit_scope = eager_scope;

... other commands possible...
```

The client can continue to issue a `COMMIT` at the end of the transaction and let PGD manage the two phases:

```
COMMIT;
```

In this case, the `eager_scope` commit scope is defined something like this:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := eager_scope,
    origin_node_group := top_group,
    rule := ALL (top_group) GROUP COMMIT (conflict_resolution = eager, commit_
↪decision = raft) ABORT ON (timeout = 60s),
    wait_for_ready := true
);
```

The commit scope group for the Eager conflict resolution rule can only be `ALL` or `MAJORITY` . Where `ALL` is used, the `commit_decision` setting must also be set to `raft` .

### Error handling

Given that PGD manages the transaction, the client needs to check only the result of the `COMMIT` . This is advisable in any case, including single-node Postgres.

In case of an origin node failure, the remaining nodes eventually (after at least `ABORT ON timeout` ) decide to roll back the globally prepared transaction. Raft prevents inconsistent commit versus rollback decisions. However, this requires a majority of connected nodes. Disconnected nodes keep the transactions prepared to eventually commit them (or roll back) as needed to reconcile with the majority of nodes that might have decided and made further progress.

### Effects of Eager Replication in general

#### Increased abort rate

With single-node Postgres, or even with PGD in its default asynchronous replication mode, errors at `COMMIT` time are rare. The added synchronization step due to the use of a commit scope using `eager` for conflict resolution also adds a source of errors. Applications need to be prepared to properly handle such errors, usually by applying a retry loop.

The rate of aborts depends solely on the workload. Large transactions changing many rows are much more likely to conflict with other concurrent transactions.

### Effects of MAJORITY and ALL node replication in general

#### Increased commit latency

Adding a synchronization step due to the use of a commit scope means more communication between the nodes, resulting in more latency at commit time. When `ALL` is used in the commit scope, this also means that the availability of the system is reduced, since any node going down causes transactions to fail.

If one or more nodes are lagging behind, the round-trip delay in getting confirmations can be large, causing high latencies. ALL or MAJORITY node replication adds roughly two network round trips (to the furthest peer node in the worst case). Logical standby nodes and nodes still in the process of joining or catching up aren't included but eventually receive changes.

Before a peer node can confirm its local preparation of the transaction, it also needs to apply it locally. This further adds to the commit latency, depending on the size of the transaction. This setting is independent of the `synchronous_commit` setting.

### Administering

When running a PGD cluster with Group Commit, you need to be aware of some things when administering the system, such as how to safely shut down and restart nodes.

### Planned shutdown and restarts

When using Group Commit with receive confirmations, take care with planned shutdown or restart. By default, the apply queue is processed prior to shutting down. However, in the `immediate` shutdown mode, the queue is discarded at shutdown, leading to the stopped node "forgetting" transactions in the queue. A concurrent failure of the origin node can lead to loss of data, as if both nodes failed.

To ensure the apply queue gets flushed to disk, use either smart for maintenance tasks. This approach maintains the required synchronization level and prevents loss of data.

### Legacy synchronous replication using PGD

Postgres provides physical streamingreplication

(PSR), which is unidirectional but offers a synchronousvariant .

For backward compatibility, PGD still supports configuring synchronous replication with `synchronous_commit` and `synchronous_standby_names` . Consider using *QUORUM COMMIT* or *SYNCHRONOUS COMMIT*

instead.

Unlike PGD replication options, PSR sync persists first, replicating after the WAL flush of commit record.

### Usage

To enable synchronous replication using PGD, you need to add the application name of the relevant PGD peer nodes to `synchronous_standby_names` . The use of `FIRST x` or `ANY x` offers some flexibility if this doesn't conflict with the requirements of non-PGD standby nodes.

Once you've added it, you can configure the level of synchronization per transaction using `synchronous_commit` , which defaults to `on` . This setting means that adding the application name to `synchronous_standby_names` already enables synchronous replication. Setting `synchronous_commit` to `local` or `off` turns off synchronous replication.

Due to PGD applying the transaction before persisting it, the values `on` and `remote_apply` are equivalent for logical replication.

### Comparison

The following table summarizes what a client can expect from a peer node replicated to after receiving a COMMIT confirmation from the origin node the transaction was issued to. The Mode column takes on different meaning depending on the variant. For PSR and legacy synchronous replication with PGD, it refers to the `synchronous_commit` setting.

| Variant | Mode | Received | Visible | Durable |
|---|---|---|---|---|
| PSR Async | off (default) | no | no | no |
| PSR Sync | remote_write (2) | yes | no | no (3) |
| PSR Sync | on (2) | yes | no | yes |
| PSR Sync | remote_apply (2) | yes | yes | yes |
| PGD Legacy Sync (1) | remote_write (2) | yes | no | no |
| PGD Legacy Sync (1) | on (2) | yes | yes | yes |
| PGD Legacy Sync (1) | remote_apply (2) | yes | yes | yes |

- (1) Consider using Quorum Commit instead.*
- (2) Unless switched to local mode (if allowed) by setting `synchronous_replication_availability` to `async'` , otherwise the values for the asynchronous PGD default apply.*
- (3) Written to the OS, durable if the OS remains running and only Postgres crashes.*

### Postgres configuration parameters

The following table provides an overview of the configuration settings that you must set to a non-default value (req) and those that are optional (opt) but affect a specific variant.

| Setting (GUC) | Group Commit | Lag Control | PSR | Legacy Sync |
|---|---|---|---|---|
| synchronous_standby_names | n/a | n/a | req | req |

continues on next page

Table 11 – continued from previous page

| Setting (GUC) | Group Commit | Lag Control | PSR | Legacy Sync |
|---|---|---|---|---|
| synchronous_commit | n/a | n/a | opt | opt |
| synchronous_replication_availability | n/a | n/a | opt | opt |

### Migration to commit scopes

You configure the Group Commit feature of PGD independent of `synchronous_commit` and `synchronous_standby_names` . Instead, the `bdr.commit_scope` GUC allows you to select the scope per transaction. And instead of configuring `synchronous_standby_names` on each node individually, Group Commit uses globally synchronized commit scopes.

**Note**

While the grammar for *synchronous_standby_names* and commit scopes looks similar, the former doesn't account for the origin node, but the latter does. Therefore, for example, *synchronous_standby_names = 'ANY 1 (..)'* is equivalent to a commit scope of *ANY 2 (...)* . This choice makes reasoning about majority easier and reflects that the origin node also contributes to the durability and visibility of the transaction.

### Internal timing of operations

For a better understanding of how the different modes work, it's helpful to know that legacy Physical Streaming Replication (PSR) and PGD apply transactions in different ways. These ordering differences have practical implications for performance, durability, and consistency. For a broader comparison of when to choose PGD over PSR, see Choosing between PSR and PGD .

With Legacy PSR, the order of operations is:

1. Origin flushes a commit record to WAL, making the transaction visible locally.
2. Peer node receives changes and issues a write.
3. Peer flushes the received changes to disk.
4. Peer applies changes, making the transaction visible on the peer.

Note that the change is written to the disk before applying the changes.

With PGD, by default and with Lag Control, the order of operations is different. In these cases, the change becomes visible on the peer before the transaction is flushed to the peer's disk:

1. Origin flushes a commit record to WAL, making the transaction visible locally.
2. Peer node receives changes into its apply queue in memory.
3. Peer applies changes, making the transaction visible on the peer.
4. Peer persists the transaction by flushing to disk.

For PGD's Quorum Commit, Group Commit, and CAMO, the origin node waits for a certain number of confirmations prior to making the transaction visible locally. The order of operations is:

1. Origin flushes a prepare or precommit record to WAL.
2. Peer node receives changes into its apply queue in memory.
3. Peer applies changes, making the transaction visible on the peer.
4. Peer persists the transaction by flushing to disk.

5. Origin commits and makes the transaction visible locally.

The following table summarizes the differences.

| Variant | Order of apply vs persist | Replication before or after commit |
|---|---|---|
| PSR | persist first | after WAL flush of commit record |
| PGD Async | apply first | after WAL flush of commit record |
| PGD Lag Control | apply first | after WAL flush of commit record |
| PGD Quorum Commit | apply first | before COMMIT on origin |
| PGD Group Commit | apply first | before COMMIT on origin |
| PGD CAMO | apply first | before COMMIT on origin |

## 6.8 Conflict Management

EDB Postgres Distributed is an active/active or multi-master DBMS. If used asynchronously, writes to the same or related rows from multiple different nodes can result in data *bdr_read_all_conflicts* when using standard data types.

Conflicts aren't errors. In most cases, they're events that PGD can detect and resolve as they occur. Resolution depends on the nature of the application and the meaning of the data, so it's important that PGD provides the application a range of choices as to how to resolve them.

By default, conflicts are resolved at the row level. When changes from two nodes conflict, either the local or remote tuple is picked and the other is discarded. For example, the commit timestamps might be compared for the two conflicting changes and the newer one kept. This approach ensures that all nodes converge to the same result and establishes commit-order-like semantics on the whole cluster.

Column-level conflict detection and resolution is available with PGD, described in *CLCD* .

If you want to avoid conflicts, you can use *Group Commit (legacy)* with *Eager conflict resolution* or conflict-free data types (CRDTs), described in *Handling column-level conflicts using CRDT data types* . You can also use Connection Manager to route all writes to one write-leader, eliminating the chance for inter-nodal conflicts.

### 6.8.1 Conflicts

EDB Postgres Distributed is an active/active or multi-master DBMS. If used asynchronously, writes to the same or related rows from multiple different nodes can result in data conflicts when using standard data types.

Conflicts aren't errors. In most cases, they are events that PGD can detect and resolve as they occur. This section introduces the PGD functionality that allows you to manage that detection and resolution.

- *DDL overview* introduces the idea of conflicts in PGD and explains how they can happen.
- *Types of Conflict* lists and discusses the various sorts of conflicts you might run across in PGD.
- *Column-level conflict detection* introduces the mechanisms PGD provides for conflict detection.
- *Missing-column conflict resolution* explains how PGD resolves conflicts and how you can change the default behavior.
- *Conflict logging* points out where PGD keeps conflict logs and explains how you can perform conflict reporting.
- *Data verification with LiveCompare* explains how LiveCompare can help keep data consistent by pointing out conflicts as they arise.

### Overview

EDB Postgres Distributed is an active/active or multi-master DBMS. If used asynchronously, writes to the same or related rows from multiple different nodes can result in data conflicts when using standard data types.

Conflicts aren't errors. In most cases, they are events that PGD can detect and resolve as they occur. Resolving them depends on the nature of the application and the meaning of the data, so it's important for PGD to provide the application with a range of choices for how to resolve conflicts.

By default, conflicts are resolved at the row level. When changes from two nodes conflict, PGD picks either the local or remote tuple and the discards the other. For example, the commit timestamps might be compared for the two conflicting changes and the newer one kept. This approach ensures that all nodes converge to the same result and establishes commit-order-like semantics on the whole cluster.

Conflict handling is configurable, as described in *Missing-column conflict resolution* . PGD can detect conflicts and handle them differently for each table using conflict triggers, described in *Stream triggers* .

Column-level conflict detection and resolution is available with PGD, as described in *CLCD* .

By default, all conflicts are logged to bdr.conflict_history . If conflicts are possible, then table owners must monitor for them and analyze how to avoid them or make plans to handle them regularly as an application task. The LiveCompare tool is also available to scan regularly for divergence.

Some clustering systems use distributed lock mechanisms to prevent concurrent access to data. These can perform reasonably when servers are very close to each other but can't support geographically distributed applications where very low latency is critical for acceptable performance.

Distributed locking is essentially a pessimistic approach. PGD advocates an optimistic approach, which is to avoid conflicts where possible but allow some types of conflicts to occur and resolve them when they arise.

### How conflicts happen

Inter-node conflicts arise as a result of sequences of events that can't happen if all the involved transactions happen concurrently on the same node. Because the nodes exchange changes only after the transactions commit, each transaction is individually valid on the node it committed on. It isn't valid if applied on another node that did other conflicting work at the same time.

Since PGD replication essentially replays the transaction on the other nodes, the replay operation can fail if there's a conflict between a transaction being applied and a transaction that was committed on the receiving node.

Most conflicts can't happen when all transactions run on a single node because Postgres has inter-transaction communication mechanisms to prevent it. Examples of these mechanisms are `UNIQUE` indexes, `SEQUENCE` operations, row and relation locking, and `SERIALIZABLE` dependency tracking. All of these mechanisms are ways to communicate between ongoing transactions to prevent undesirable concurrency issues.

PGD doesn't have a distributed transaction manager or lock manager. That's part of why it performs well with latency and network partitions. As a result, transactions on different nodes execute entirely independently from each other when using the default, which is lazy replication. Less independence between nodes can avoid conflicts altogether, which is why PGD also offers Eager Replication for when this is important.

### Avoiding or tolerating conflicts

In most cases, you can design the application to avoid or tolerate conflicts.

Conflicts can happen only if things are happening at the same time on multiple nodes. The simplest way to avoid conflicts is to only ever write to one node or to only ever write to a specific row in a specific way from one specific node at a time.

This avoidance happens naturally in many applications. For example, many consumer applications allow only the owning user to change data, such as changing the default billing address on an account. Such data changes seldom

have update conflicts.

You might make a change just before a node goes down, so the change seems to be lost. You might then make the same change again, leading to two updates on different nodes. When the down node comes back up, it tries to send the older change to other nodes. It's rejected because the last update of the data is kept.

For `INSERT`/`INSERT` conflicts, use *global sequences* to prevent this type of conflict.

For applications that assign relationships between objects, such as a room-booking application, applying `update_if_newer` might not give an acceptable business outcome. That is, it isn't useful to confirm to two people separately that they have booked the same room. The simplest resolution is to use Eager Replication to ensure that only one booking succeeds. More complex ways might be possible depending on the application. For example, you can assign 100 seats to each node and allow those to be booked by a writer on that node. But if none are available locally, use a distributed locking scheme or Eager Replication after most seats are reserved.

Another technique for ensuring certain types of updates occur only from one specific node is to route different types of transactions through different nodes. For example:

- Receiving parcels on one node but delivering parcels using another node
- A service application where orders are input on one node and work is prepared on a second node and then served back to customers on another

Frequently, the best course is to allow conflicts to occur and design the application to work with PGD's conflict resolution mechanisms to cope with the conflict.

### Types of Conflict

#### PRIMARY KEY or UNIQUE conflicts

The most common conflicts are row conflicts, where two operations affect a row with the same key in ways they can't on a single node. PGD can detect most of those and applies the `update_if_newer` conflict resolver.

Row conflicts include:

- `INSERT` versus `INSERT`
- `UPDATE` versus `UPDATE`
- `UPDATE` versus `DELETE`
- `INSERT` versus `UPDATE`
- `INSERT` versus `DELETE`
- `DELETE` versus `DELETE`

The view `bdr.node_conflict_resolvers` provides information on how conflict resolution is currently configured for all known conflict types.

#### INSERT/INSERT conflicts

The most common conflict, `INSERT`/`INSERT`, arises where `INSERT` operations on two different nodes create a tuple with the same `PRIMARY KEY` values (or if no `PRIMARY KEY` exists, the same values for a single `UNIQUE` constraint).

PGD handles this situation by retaining the most recently inserted tuple of the two according to the originating node's timestamps. (A user-defined conflict handler can override this behavior.)

This conflict generates the `insert_exists` conflict type, which is by default resolved by choosing the newer row, based on commit time, and keeping only that one (`update_if_newer` resolver). You can configure other resolvers. See *Missing-column conflict resolution* for details.

To resolve this conflict type, you can also use column-level conflict resolution and user-defined conflict triggers.

You can effectively eliminate this type of conflict by using *global sequences* .

### INSERT operations that violate UNIQUE or EXCLUDE constraints

An `INSERT` /`INSERT` conflict can violate more than one `UNIQUE` constraint, one of which might be the `PRIMARY KEY` , or violate one or more `EXCLUDE` constraints.

In either of the following cases, applying the replication change produces a `multiple_unique_conflicts` conflict. Both of these cases result in a conflict against more than one other row.

- If a new row violates more than one `UNIQUE` constraint and that results in a conflict against more than one other row.
- If a new row violates more than one `EXCLUDE` constraint or a single `EXCLUDE` constraint.

In case of such a conflict, for replication to continue, you must remove some rows. Depending on the resolver setting for `multiple_unique_conflicts` , the apply process either exits with an error, skips the incoming row, or deletes some of the rows. The deletion tries to preserve the row with the correct `PRIMARY KEY` and delete the others.

!!!Warning

In case of multiple rows conflicting this way, if the result of conflict resolution is to proceed with the insert operation, some of the data is always deleted.

### UPDATE/UPDATE conflicts

Where two concurrent `UPDATE` operations on different nodes change the same tuple but not its `PRIMARY KEY` , an `UPDATE` /`UPDATE` conflict can occur on replay.

These can generate different conflict kinds based on the configuration and situation. If the table is configured with *Row version conflict detection* , then the original (key) row is compared with the local row. If they're different, the `update_differing` conflict is generated. When using *Origin conflict detection* , the origin of the row is checked. (The origin is the node that the current local row came from.) If that changed, the `update_origin_change` conflict is generated. In all other cases, the `UPDATE` is normally applied without generating a conflict.

Both of these conflicts are resolved the same way as `insert_exists` , described in *INSERT/INSERT conflicts* .

### UPDATE conflicts on the PRIMARY KEY

PGD can't currently perform conflict resolution where the `PRIMARY KEY` is changed by an `UPDATE` operation. You can update the primary key, but you must ensure that no conflict with existing values is possible.

Conflicts on the update of the primary key are *Divergent conflicts* and require manual intervention.

Updating a primary key is possible in Postgres, but there are issues in both Postgres and PGD.

To see how these updates behave, create a basic table with a primary key and insert an initial row:

```
CREATE TABLE pktest (pk integer primary key, val integer);
INSERT INTO pktest VALUES (1,1);
```

Updating a primary key column is technically possible. This command succeeds because the new value does not conflict with any existing data:

```
UPDATE pktest SET pk=2 WHERE pk=1;
```

However, suppose the table has multiple rows:

```
INSERT INTO pktest VALUES (3,3);
```

Some UPDATE operations succeed:

```
UPDATE pktest SET pk=4 WHERE pk=3;
```

Confirm the changes to the table by selecting all rows. Notice that the primary key values have successfully shifted:

```
SELECT * FROM pktest;
 pk | val
- ---+-----
  2 |   1
  4 |   3
(2 rows)
```

Other UPDATE operations fail with constraint errors:

```
UPDATE pktest SET pk=4 WHERE pk=2;
```

Select all rows from the table and obvserve a standard constraint violation error:

```
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
DETAIL:  Key (pk)=(4) already exists
```

Even without PGD, you must design your application to prevent these runtime errors, as Postgres strictly enforces primary key uniqueness.

Distributed environments like PGD increase the complexity of primary key updates. When multiple nodes attempt to modify keys simultaneously, the results depend on whether the changes overlap or conflict.

If two nodes update different rows to different new primary key values, PGD can replicate the changes successfully:

```
- -node1:
UPDATE pktest SET pk=pk+1 WHERE pk = 2;

After replication, both nodes reflect the incremented values, maintaining a␣
↪consistent state across the cluster:

- -node2:
UPDATE pktest SET pk=pk+1 WHERE pk = 4;
```

```
SELECT * FROM pktest;

 pk | val
- ---+-----
  3 |   1
  5 |   3
(2 rows)
```

Executing these next two changes concurrently causes a divergent error, since both changes are accepted. Both nodes accept their own local change, but applying the changes on the other node results in `update_missing` conflicts.

```
- -node1:
UPDATE pktest SET pk=1 WHERE pk = 3;
- -node2:
UPDATE pktest SET pk=2 WHERE pk = 3;
```

This scenario leaves the data different on each node:

```
- -node1:
SELECT * FROM pktest;
 pk | val
- ---+-----
  1 |   1
  5 |   3
(2 rows)

- -node2:
SELECT * FROM pktest;
 pk | val
- ---+-----
  2 |   1
  5 |   3
(2 rows)
```

You can identify and resolve this situation using LiveCompare .

Concurrent conflicts present problems. Executing these two changes concurrently isn't easy to resolve:

```
- -node1:
UPDATE pktest SET pk=6, val=8 WHERE pk = 5;
- -node2:
UPDATE pktest SET pk=6, val=9 WHERE pk = 5;
```

Both changes are applied locally, causing a divergence between the nodes. But the apply on the target fails on both nodes with a duplicate key-value violation error. This error causes the replication to halt and requires manual resolution.

You can avoid this duplicate key violation error, and replication doesn't break, if you set the conflict_type `update_pkey_exists` to `skip` , `update` , or `update_if_newer` . This approach can still lead to divergence depending on the nature of the update.

You can avoid divergence in cases where the same old key is being updated by the same new key concurrently by setting `update_pkey_exists` to `update_if_newer` . However, in certain situations, divergence occurs even with `update_if_newer` , namely when two different rows both are updated concurrently to the same new primary key.

As a result, we recommend strongly against allowing primary key UPDATE operations in your applications, especially with PGD. If parts of your application change primary keys, then to avoid concurrent changes, make those changes using *QUORUM COMMIT* , which uses eager conflict resolution to prevent concurrent conflicting commits before any node commits.

**Warning**

In case the conflict resolution of *update_pkey_exists* conflict results in update, one of the rows is always deleted.

### UPDATE operations that violate UNIQUE or EXCLUDE constraints

Like *INSERT operations that violate UNIQUE or EXCLUDE constraints* , when an incoming `UPDATE` violates more than one `UNIQUE` /`EXCLUDE` index (including the `PRIMARY KEY` ) or violates a single `EXCLUDE` index such that more than one row is in conflict, PGD raises a `multiple_unique_conflicts` conflict.

PGD supports deferred unique constraints. If a transaction can commit on the source, then it applies cleanly on target, unless it sees conflicts. However, you can't use a deferred primary key as a REPLICA IDENTITY, so the use cases are already limited by that and the warning about using multiple unique constraints.

### UPDATE/DELETE conflicts

One node can update a row that another node deletes at the same time. In this case an `UPDATE` /`DELETE` conflict can occur on replay.

If the deleted row is still detectable (the deleted row wasn't removed by `VACUUM` ), the `update_recently_deleted` conflict is generated. By default, the `UPDATE` is skipped, but you can configure the resolution for this. See *Missing-column conflict resolution* for details.

The database can clean up the deleted row by the time the `UPDATE` is received in case the local node is lagging behind in replication. In this case, PGD can't differentiate between `UPDATE` /`DELETE` conflicts and *INSERT/UPDATE conflicts* . It generates the `update_missing` conflict.

Another type of conflicting `DELETE` and `UPDATE` is a `DELETE` that comes after the row was updated locally. In this situation, the outcome depends on the type of conflict detection used. When using the default, *Origin conflict detection* , no conflict is detected, leading to the `DELETE` being applied and the row removed. If you enable *Row version conflict detection* , a `delete_recently_updated` conflict is generated. The default resolution for a `delete_recently_updated` conflict is to `skip` the deletion. However, you can configure the resolution or a conflict trigger can be configured to handle it.

### INSERT/UPDATE conflicts

When using the default asynchronous mode of operation, a node might receive an `UPDATE` of a row before the original `INSERT` was received. This can happen only when three or more nodes are active (see *Conflicts with three or more nodes* ).

When this happens, the `update_missing` conflict is generated. The default conflict resolver is `insert_or_skip` , though you can use `insert_or_error` or `skip` instead. Resolvers that do insert-or-action first try to `INSERT` a new row based on data from the `UPDATE` when possible (when the whole row was received). For reconstructing the row to be possible, the table either needs to have `REPLICA IDENTITY FULL` or the row must not contain any toasted data.

See *TOAST support details* for more info about toasted data.

### INSERT/DELETE conflicts

Similar to the `INSERT` /`UPDATE` conflict, the node might also receive a `DELETE` operation on a row for which it didn't yet receive an `INSERT` . This is again possible only with three or more nodes set up (see *Conflicts with three or more nodes* ).

PGD can't currently detect this conflict type. The `INSERT` operation doesn't generate any conflict type, and the `INSERT` is applied.

The `DELETE` operation always generates a `delete_missing` conflict, which is by default resolved by skipping the operation.

### DELETE/DELETE conflicts

A `DELETE` /`DELETE` conflict arises when two different nodes concurrently delete the same tuple.

This scenario always generates a `delete_missing` conflict, which is by default resolved by skipping the operation.

This conflict is harmless since both `DELETE` operations have the same effect. You can safely ignroe one of them.

### Conflicts with three or more nodes

If one node inserts a row that's then replayed to a second node and updated there, a third node can receive the `UPDATE` from the second node before it receives the `INSERT` from the first node. This scenario is an `INSERT` /`UPDATE` conflict.

These conflicts are handled by discarding the `UPDATE` , which can lead to different data on different nodes. These are *Divergent conflicts* .

This conflict type can happen only with three or more masters. At least two masters must be actively writing.

Also, the replication lag from node 1 to node 3 must be high enough to allow the following sequence of actions:

1. node 2 receives INSERT from node 1
2. node 2 performs UPDATE
3. node 3 receives UPDATE from node 2
4. node 3 receives INSERT from node 1

Using `insert_or_error` (or in some cases the `insert_or_skip` conflict resolver for the `update_missing` conflict type) is a viable mitigation strategy for these conflicts. However, enabling this option opens the door for `INSERT` /`DELETE` conflicts:

1. node 1 performs UPDATE
2. node 2 performs DELETE
3. node 3 receives DELETE from node 2
4. node 3 receives UPDATE from node 1, turning it into an INSERT

If these are problems, we recommend tuning freezing settings for a table or database so that they're correctly detected as `update_recently_deleted` .

Another alternative is to use *Eager conflict resolution* to prevent these conflicts. Eager conflict resolution is always active with *QUORUM COMMIT* and aborts conflicting transactions before either commits.

`INSERT` /`DELETE` conflicts can also occur with three or more nodes. Such a conflict is identical to `INSERT` /`UPDATE` except with the `UPDATE` replaced by a `DELETE` . This can result in a `delete_missing` conflict.

PGD could choose to make each `INSERT` into a check-for-recently deleted, as occurs with an `update_missing` conflict. However, the cost of doing this penalizes the majority of users, so at this time it instead logs `delete_missing` .

Future releases will automatically resolve `INSERT` /`DELETE` anomalies by way of rechecks using LiveCompare when `delete_missing` conflicts occur. Applications can perform these manually by checking the `bdr.conflict_history_summary` view.

These conflicts can occur in two main problem use cases:

- `INSERT` followed rapidly by a `DELETE` , as can be used in queuing applications
- Any case where the primary key identifier of a table is reused

Neither of these cases is common. We recommend not replicating the affected tables if these problem use cases occur.

PGD has problems with the latter case because PGD relies on the uniqueness of identifiers to make replication work correctly.

Applications that insert, delete, and then later reuse the same unique identifiers can cause difficulties. This is known as the ABA problem . PGD has no way of knowing whether the rows are the current row, the last row, or much older rows.

Unique identifier reuse is also a business problem, since it prevents unique identification over time, which prevents auditing, traceability, and sensible data quality. Applications don't need to reuse unique identifiers.

Any identifier reuse that occurs in the time interval it takes for changes to pass across the system causes difficulties. Although that time might be short in normal operation, down nodes can extend that interval to hours or days.

We recommend that applications don't reuse unique identifiers. If they do, take steps to avoid reuse in less than a year.

This problem doesn't occur in applications that use sequences or UUIDs.

### Foreign key constraint conflicts

Conflicts between a remote transaction being applied and existing local data can also occur for `FOREIGN KEY` (FK) constraints.

PGD applies changes with `session_replication_role = 'replica'` , so foreign keys aren't rechecked when applying changes. In an active/active environment, this situation can result in FK violations if deletes occur to the referenced table at the same time as inserts into the referencing table. This scenario is similar to an `INSERT` /`DELETE` conflict.

In single-master Postgres, any `INSERT` /`UPDATE` that refers to a value in the referenced table must wait for `DELETE` operations to finish before they can gain a row-level lock. If a `DELETE` removes a referenced value, then the `INSERT` /`UPDATE` fails the FK check.

In multi-master PGD. there are no inter-node row-level locks. An `INSERT` on the referencing table doesn't wait behind a `DELETE` on the referenced table, so both actions can occur concurrently. Thus an `INSERT` /`UPDATE` on one node on the referencing table can use a value at the same time as a `DELETE` on the referenced table on another node. The result, then, is a value in the referencing table that's no longer present in the referenced table.

In practice, this situation occurs if the `DELETE` operations occur on referenced tables in separate transactions from `DELETE` operations on referencing tables, which isn't a common operation.

In a parent-child relationship such as Orders -> OrderItems, it isn't typical to do this. It's more likely to mark an OrderItem as canceled than to remove it completely. For reference/lookup data, it's unusual to completely remove entries at the same time as using those same values for new fact data.

While dangling FKs are possible, the risk of this in general is very low. Thus PGD doesn't impose a generic solution to cover this case. Once you understand the situation in which this occurs, two solutions are possible.

The first solution is to restrict the use of FKs to closely related entities that are generally modified from only one node at a time, are infrequently modified, or where the modification's concurrency is application mediated. This approach avoids any FK violations at the application level.

The second solution is to add triggers to protect against this case using the PGD-provided functions `bdr.ri_fkey_trigger()` and `bdr.ri_fkey_on_del_trigger()` . When called as `BEFORE` triggers, these functions use `FOREIGN KEY` information to avoid FK anomalies by setting referencing columns to NULL, much as if you had a SET NULL constraint. This approach rechecks all FKs in one trigger, so you need to add only one trigger per table to prevent FK violation.

As an example, suppose you have two tables: Fact and RefData. Fact has an FK that references RefData. Fact is the referencing table, and RefData is the referenced table. You need to add one trigger to each table.

Add a trigger that sets columns to NULL in Fact if the referenced row in RefData was already deleted:

```
CREATE TRIGGER bdr_replica_fk_iu_trg
    BEFORE INSERT OR UPDATE ON fact
    FOR EACH ROW
    EXECUTE PROCEDURE bdr.ri_fkey_trigger();

ALTER TABLE fact
    ENABLE REPLICA TRIGGER bdr_replica_fk_iu_trg;
```

Add a trigger that sets columns to NULL in Fact at the time a DELETE occurs on the RefData table:

```
CREATE TRIGGER bdr_replica_fk_d_trg
    BEFORE DELETE ON refdata
```

(continues on next page)

(continued from previous page)

```
    FOR EACH ROW
    EXECUTE PROCEDURE bdr.ri_fkey_on_del_trigger();

ALTER TABLE refdata
    ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

Adding both triggers avoids dangling foreign keys.

### TRUNCATE conflicts

`TRUNCATE` behaves similarly to a `DELETE` of all rows but performs this action by physically removing the table data rather than row-by-row deletion. As a result, row-level conflict handling isn't available, so `TRUNCATE` commands don't generate conflicts with other DML actions, even when there's a clear conflict.

As a result, the ordering of replay can cause divergent changes if another DML is executed concurrently on other nodes to the `TRUNCATE` .

You can take one of the following actions:

- Ensure `TRUNCATE` isn't executed alongside other concurrent DML. Rely on LiveCompare to highlight any such inconsistency.
- Replace `TRUNCATE` with a `DELETE` statement with no `WHERE` clause. This approach is likely to have poor performance on larger tables.
- Set `bdr.truncate_locking = 'on'` to set the `TRUNCATE` command's locking behavior. This setting determines whether `TRUNCATE` obeys the `bdr.ddl_locking` setting. This isn't the default behavior for `TRUNCATE` since it requires all nodes to be up. This configuration might not be possible or wanted in all cases.

### Data conflicts for roles and tablespace differences

Conflicts can also arise where nodes have global (Postgres-system-wide) data, like roles, that differ. This conflict can result in operations—mainly `DDL` —that can run successfully and commit on one node but then fail to apply to other nodes.

For example, node1 might have a user named fred, and that user wasn't created on node2. If fred on node1 creates a table, the table is replicated with its owner set to fred. When the DDL command is applied to node2, the DDL fails because there's no user named fred. This failure generates an error in the Postgres logs.

Administrator intervention is required to resolve this conflict by creating the user fred in the database where PGD is running. You can set `bdr.role_replication = on` to resolve this in future.

### Lock conflicts and deadlock aborts

Because PGD writer processes operate much like normal user sessions, they're subject to the usual rules around row and table locking. This can sometimes lead to PGD writer processes waiting on locks held by user transactions or even by each other.

Relevant locking includes:

- Explicit table-level locking (`LOCK TABLE ...` ) by user sessions
- Explicit row-level locking (`SELECT ... FOR UPDATE/FOR SHARE` ) by user sessions
- Implicit locking because of row `UPDATE` , `INSERT` , or `DELETE` operations, either from local activity or from replication from other nodes

A PGD writer process can deadlock with a user transaction, where the user transaction is waiting on a lock held by the writer process and vice versa. Two writer processes can also deadlock with each other. Postgres's deadlock detector steps in and terminates one of the problem transactions. If the PGD writer process is terminated, it retries and generally succeeds.

All these issues are transient and generally require no administrator action. If a writer process is stuck for a long time behind a lock on an idle user session, the administrator can terminate the user session to get replication flowing again. However, this is no different from a user holding a long lock that impacts another user session.

Use of the log_lock_waits facility in Postgres can help identify locking related replay stalls.

### Divergent conflicts

Divergent conflicts arise when data that should be the same on different nodes differs unexpectedly. Divergent conflicts shouldn't occur, but not all such conflicts can be reliably prevented at the time of writing.

Changing the `PRIMARY KEY` of a row can lead to a divergent conflict if another node changes the key of the same row before all nodes have replayed the change. Avoid changing primary keys, or change them only on one designated node.

Divergent conflicts involving row data generally require administrator action to manually adjust the data on one of the nodes to be consistent with the other one. Such conflicts don't arise so long as you use PGD as documented and avoid settings or functions marked as unsafe.

The administrator must manually resolve such conflicts. You might need to use the advanced options such as `bdr.ddl_replication` and `bdr.ddl_locking` depending on the nature of the conflict. However, careless use of these options can make things much worse and create a conflict that generic instructions can't address.

### TOAST support details

Postgres uses out-of-line storage for larger columns called TOAST .

The TOAST values handling in logical decoding (which PGD is built on top of) and logical replication is different from inline data stored as part of the main row in the table.

The TOAST value is logged into the transaction log (WAL) only if the value changed. This can cause problems, especially when handling UPDATE conflicts, because an `UPDATE` statement that didn't change a value of a toasted column produces a row without that column. As mentioned in *INSERT/UPDATE conflicts* , PGD reports an error if an `update_missing` conflict is resolved using `insert_or_error` and there are missing TOAST columns.

However, more subtle issues than this one occur in case of concurrent workloads with asynchronous replication. (Eager transactions aren't affected.) Imagine, for example, the following workload on an EDB Postgres Distributed cluster with three nodes called A, B, and C:

1. On node A: txn A1 does an UPDATE SET col1 = 'toast data…' and commits first.
2. On node B: txn B1 does UPDATE SET other_column = 'anything else'; and commits after A1.
3. On node C: the connection to node A lags behind.
4. On node C: txn B1 is applied first, it misses the TOASTed column in col1, but gets applied without conflict.
5. On node C: txn A1 conflicts (on update_origin_change) and is skipped.
6. Node C misses the toasted data from A1 forever.

This scenario isn't usually a problem when using PGD. (It is when using either built-in logical replication or plain pglogical for multi-master.) PGD adds its own logging of TOAST columns when it detects a local `UPDATE` to a row that recently replicated a TOAST column modification and the local `UPDATE` isn't modifying the TOAST. Thus PGD prevents any inconsistency for toasted data across different nodes. This situation causes increased WAL logging when updates occur on multiple nodes, that is, when origin changes for a tuple. Additional WAL overhead is zero if all updates are made from a single node, as is normally the case with PGD AlwaysOn architecture.

**Note**

Running *VACUUM FULL* or *CLUSTER* on just the TOAST table without doing same on the main table removes metadata needed for the extra logging to work. This means that, for a short period after such a statement, the protection against these concurrency issues isn't present.

**Warning**

The additional WAL logging of TOAST is done using the *BEFORE UPDATE* trigger on standard Postgres. This trigger must be sorted alphabetically last based on trigger name among all *BEFORE UPDATE* triggers on the table. It's prefixed with *zzzz_bdr_* to make this easier, but make sure you don't create any trigger with a name that sorts after it. Otherwise you won't have the protection against the concurrency issues.

For the `insert_or_error` conflict resolution, the use of `REPLICA IDENTITY FULL` is still required.

None of these problems associated with toasted columns affect tables with `REPLICA IDENTITY FULL` . This setting always logs a toasted value as part of the key since the whole row is considered to be part of the key. PGD can reconstruct the new row, filling the missing data from the key row. As a result, using `REPLICA IDENTITY FULL` can increase WAL size significantly.

### Conflict detection

PGD provides these mechanisms for conflict detection:

- `:ref:`Origin conflict detection <Origin conflict detection>`  (default)`
- `:ref:`Row version conflict detection <Row version conflict detection>``
- `:ref:`Column-level conflict detection <Column-level conflict detection>``

### Origin conflict detection

Origin conflict detection uses and relies on commit timestamps as recorded on the node the transaction originates from. This requires clocks to be in sync to work correctly or to be within a tolerance of the fastest message between two nodes. If this isn't the case, conflict resolution tends to favor the node that's further ahead. You can manage clock skew between nodes using the parameters `bdr.maximum_clock_skew` and `bdr.maximum_clock_skew_action` .

**Note**

If a local node's clock lags behind a remote node's commit timestamp, transactions using group commit scopes will experience increased commit latency. This delay occurs regardless of the settings for *bdr.maximum_clock_skew* or *bdr.maximum_clock_skew_action* , as the local node must wait until its local timestamp is no longer older than the remote commit timestamp. To ensure optimal performance and minimize latency, system clocks must be strictly synchronized.

Conflicts are first detected based on whether the replication origin changed, so conflict triggers are called in situations that might not turn out to be conflicts. Hence, this mechanism isn't precise, since it can generate false-positive conflicts.

Origin info is available only up to the point where a row is frozen. Updates arriving for a row after it was frozen don't raise a conflict so are applied in all cases. This is the normal case when adding a new node by `bdr_init_physical` , so raising conflicts causes many false-positive results in that case.

A node that was offline that reconnects and begins sending data changes can cause divergent errors if the newly arrived updates are older than the frozen rows that they update. Inserts and deletes aren't affected by this situation.

We suggest that you don't leave down nodes for extended outages, as discussed in *Node restart and down node recovery* .

On EDB Postgres Extended Server and EDB Postgres Advanced Server, PGD holds back the freezing of rows while a node is down. This mechanism handles this situation gracefully so you don't need to change parameter settings.

On other variants of Postgres, you might need to manage this situation with some care.

Freezing normally occurs when a row being vacuumed is older than `vacuum_freeze_min_age` xids from the current xid, which means that you need to configure suitably high values for these parameters:

- `vacuum_freeze_min_age`
- `vacuum_freeze_table_age`
- `autovacuum_freeze_max_age`

Choose values based on the transaction rate, giving a grace period of downtime before removing any conflict data from the database node. For example, when `vacuum_freeze_min_age` is set to 500 million, a node performing 1000 TPS can be down for just over 5.5 days before conflict data is removed. The CommitTS data structure takes on-disk space of 5 GB with that setting, so lower transaction rate systems can benefit from lower settings.

Initially, recommended settings are:

```
#  1 billion = 10GB

autovacuum_freeze_max_age = 1000000000

vacuum_freeze_min_age = 500000000

#  90% of autovacuum_freeze_max_age

vacuum_freeze_table_age = 900000000
```

Note that:

- You can set `autovacuum_freeze_max_age` only at node start.
- You can set `vacuum_freeze_min_age` , so using a low value freezes rows early and can result in conflicts being ignored. You can also set `autovacuum_freeze_min_age` and `toast.autovacuum_freeze_min_age` for individual tables.
- Running the `CLUSTER` or `VACUUM FREEZE` commands also freezes rows early and can result in conflicts being ignored.

### Row version conflict detection

PGD provides the option to use row versioning and make conflict detection independent of the nodes' system clock.

Row version conflict detection requires that you enable three things. If any of these steps aren't performed correctly then *Origin conflict detection* is used.

- Enable `REPLICA IDENTITY FULL` on all tables that use row version conflict detection.
- Enable row version tracking on the table by using `bdr.alter_table_conflict_detection` . This function adds a column with a name you specify and an `UPDATE` trigger that manages the new column value. The column is created as `INTEGER` type.

Although the counter is incremented only on `UPDATE` , this technique allows conflict detection for both `UPDATE` and `DELETE` .

This approach resembles Lamport timestamps and fully prevents the ABA problem for conflict detection.

> **Note**
>
> The row-level conflict resolution is still handled based on the *Missing-column conflict resolution* configuration even with row versioning. The way the row version is generated is useful only for detecting conflicts. Don't rely on it as authoritative information about which version of row is newer.

To determine the current conflict detection strategy used for a specific table, refer to the column `conflict_detection` of the view `bdr.tables` .

To change the current conflict detection strategy, use bdr.alter_table_conflict_detection .

### Conflict resolution

Most conflicts can be resolved automatically. PGD defaults to a last-update-wins mechanism or, more accurately, the `update_if_newer` conflict resolver. This mechanism retains the most recently inserted or changed row of the two conflicting ones based on the same commit timestamps used for conflict detection. The behavior in certain corner-case scenarios depends on the settings used for `bdr.create_node_group` and alternatively for `bdr.alter_node_group` .

PGD lets you override the default behavior of conflict resolution by using bdr.alter_node_set_conflict_resolver .

### Conflict logging

Starting with PGD 6.0, conflict-logging is not enabled by default. You can enable conflict logging on a per-node basis. This allows you to log conflicts that occur on a specific node, which can be useful for debugging and monitoring purposes.

Run this command on the named node to enable logging of all conflicts on that particular node. If you want to enable logging on all nodes, run this command on each node in the PGD group.

```
SELECT bdr.alter_node_set_log_config(`nodename`, false, true, NULL, NULL);
```

The other parameters after the nodename control, respectively, specify whether to log conflicts into the PGD logfile, and whether to log conflicts into the `bdr.conflict_history` table. The last two parameters take arrays of strings which specify the conflict types and conflict resolutions to log. See Conflicts in the reference section for a full list of both. If you set these parameters to `NULL` , PGD will log all conflict types and resolutions.

PGD logs every conflict into the `bdr.conflict_history` table. You can change this behavior with more granularity using bdr.alter_node_set_log_config .

### Conflict reporting

You can summarize conflicts logged to tables in reports. Reports allow application owners to identify, understand, and resolve conflicts and introduce application changes to prevent them.

```
SELECT nspname, relname
, date_trunc(day, local_time) :: date AS date
, count(*)
FROM bdr.conflict_history
WHERE local_time > date_trunc(day, current_timestamp)
```

(continues on next page)

(continued from previous page)

```
GROUP BY 1,2,3
ORDER BY 1,2;
```

```
 nspname | relname |    date    | count
- --------+---------+------------+-------
 my_app  | test    | 2019-04-05 |     1
(1 row)
```

### Data verification with LiveCompare

LiveCompare is a utility program designed to compare any two databases to verify that they are identical.

LiveCompare is included as part of the PGD stack and can be aimed at any pair of PGD nodes. By default, it compares all replicated tables and reports differences. LiveCompare also works with non-PGD data sources such as Postgres and Oracle.

You can also use LiveCompare to continuously monitor incoming rows. You can stop and start it without losing context information, so you can run it at convenient times.

LiveCompare allows concurrent checking of multiple tables. You can configure it to allow checking of a few tables or just a section of rows in a table. Checks are performed by first comparing whole row hashes. If different, LiveCompare then compares whole rows. LiveCompare avoids overheads by comparing rows in useful-sized batches.

If differences are found, they can be rechecked over time, allowing for the delays of eventual consistency.

See the LiveCompare documentation for further details.

## 6.8.2 Column-level conflict detection

By default, conflicts are resolved at row level. When changes from two nodes conflict, either the local or remote tuple is selected and the other is discarded. For example, commit timestamps for the two conflicting changes might be compared and the newer one kept. This approach ensures that all nodes converge to the same result and establishes commit-order-like semantics on the whole cluster.

However, it might sometimes be appropriate to resolve conflicts at the column level rather than the row level, at least in some cases.

- *DDL overview* introduces column-level conflict resolution in contrast to row-level conflict resolution, suggesting where it might be a better fit than row-level conflict resolution.
- *Enabling and disabling column-level conflict resolution* provides an example of enabling column-level conflict resolution and explains how to list tables with column-level conflict resolution enabled.
- *Timestamps in column-level conflict resolution* explicates the difference between using `column_modify_timestamp` and `column_commit_timestamp` and shows how the timestamps associated with column-level conflict resolution can be selected and inspected.

### Overview

By default, conflicts are resolved at row level. When changes from two nodes conflict, either the local or remote tuple is selected and the other is discarded. For example, commit timestamps for the two conflicting changes might be compared and the newer one kept. This approach ensures that all nodes converge to the same result and establishes commit-order-like semantics on the whole cluster.

However, it might sometimes be appropriate to resolve conflicts at the column level rather than the row level, at least in some cases.

### When to resolve at the column level

Consider a simple example in which table t has two integer columns, a and b, and a single row `(1,1)` . On one node execute:

```
UPDATE t SET a = 100
```

On another node, before receiving the preceding `UPDATE` , concurrently execute:

```
UPDATE t SET b = 100
```

**Note**

The attributes modified by an *UPDATE* are determined by comparing the old and new row in a trigger. This means that if the attribute doesn't change a value, it isn't detected as modified even if it's explicitly set. For example, *UPDATE t SET a = a* doesn't mark *a* as modified for any row. Similarly, *UPDATE t SET a = 1* doesn't mark *a* as modified for rows that are already set to *1* .

This sequence results in an `UPDATE-UPDATE` conflict. With the

update_if_newer conflict resolution, the commit timestamps are compared, and the new row version is kept. Assuming the second node committed last, the result is `(1,100)` , which effectively discards the change to column a.

For many use cases, this behavior is desired and expected. However, for some use cases, this might be an issue. Consider, for example, a multi-node cluster where each part of the application is connected to a different node, updating a dedicated subset of columns in a shared table. In that case, the different components might conflict and overwrite changes.

For such use cases, it might be more appropriate to resolve conflicts on a given table at the column level. To achieve that, PGD tracks the timestamp of the last change for each column separately and uses that to pick the most recent value, essentially performing `update_if_newer` .

Applied to the previous example, the result is `(100,100)` on both nodes, despite neither of the nodes ever seeing such a row.

When thinking about column-level conflict resolution, it can be useful to see tables as vertically partitioned, so that each update affects data in only one slice. This approach eliminates conflicts between changes to different subsets of columns. In fact, vertical partitioning can even be a practical alternative to column-level conflict resolution.

Column-level conflict resolution requires the table to have `REPLICA IDENTITY FULL` . The bdr.alter_table_conflict_detection() function checks that and fails with an error if this setting is missing.

### Special problems for column-level conflict resolution

By treating the columns independently, it's easy to violate constraints in a way that isn't possible when all changes happen on the same node. Consider, for example, a table like this:

```
CREATE TABLE t (id INT PRIMARY KEY, a INT, b INT, CHECK (a > b));
INSERT INTO t VALUES (1, 1000, 1);
```

Assume one node does:

```
UPDATE t SET a = 100;
```

Another node concurrently does:

```
UPDATE t SET b = 500;
```

Each of those updates is valid when executed on the initial row and so passes on each node. But when replicating to the other node, the resulting row violates the `CHECK (a > b)` constraint, and the replication stops until the issue is resolved manually.

### Handling column-level conflicts using CRDT data types

By default, column-level conflict resolution picks the value with a higher timestamp and discards the other one. You can, however, reconcile the conflict in different, more elaborate ways. For example, you can use *Operation-based CRDT types (CmCRDT)* that allow merging the conflicting values without discarding any information.

### Enabling and disabling column-level conflict resolution

The bdr.alter_table_conflict_detection() function manages column-level conflict resolution.

### Using bdr.alter_table_conflict_detection to enable column-level conflict resolution

The bdr.alter_table_conflict_detection function takes a table name and column name as its arguments. The column is added to the table as a `column_modify_timestamp` column. The function also adds two triggers (BEFORE INSERT and BEFORE UPDATE) that are responsible for maintaining timestamps in the new column before each change.

```
db=# CREATE TABLE my_app.test_table (id SERIAL PRIMARY KEY, val INT);
CREATE TABLE

db=# ALTER TABLE my_app.test_table REPLICA IDENTITY FULL;
ALTER TABLE

db=# SELECT bdr.alter_table_conflict_detection(
db(# my_app.test_table::regclass,
db(# column_modify_timestamp, cts);
 alter_table_conflict_detection
- --------------------------------
 t

db=# \d my_app.test_table
                                      Table "my_app.test_table"
 Column |          Type           | Collation | Nullable |                       Default
- -------+-------------------------+-----------+----------+-----------------------------
↪----------------------
 id     | integer                 |           | not null | nextval(my_app.test_table_id_
↪seq::regclass)
 val    | integer                 |           |          |
 cts    | bdr.column_timestamps   |           | not null | s 1 775297963454602 0 0::bdr.
↪column_timestamps
Indexes:
    "test_table_pkey" PRIMARY KEY, btree (id)
Triggers:
    bdr_clcd_before_insert BEFORE INSERT ON my_app.test_table FOR EACH ROW EXECUTE␣
↪FUNCTION bdr.column_timestamps_current_insert()
    bdr_clcd_before_update BEFORE UPDATE ON my_app.test_table FOR EACH ROW EXECUTE␣
↪FUNCTION bdr.column_timestamps_current_update()
```

The new column specifies `NOT NULL` with a default value, which means that `ALTER TABLE ... ADD COLUMN` doesn't perform a table rewrite.

!!! Note

Avoid using columns with the *bdr.column_timestamps* data type for other purposes, as doing so can have negative effects. For example, it switches the table to column-level conflict resolution, which doesn't work correctly without the triggers.

### Listing tables with column-level conflict resolution

You can list tables having column-level conflict resolution enabled with the following query.

```
SELECT nc.nspname, c.relname
FROM pg_attribute a
JOIN (pg_class c JOIN pg_namespace nc ON c.relnamespace = nc.oid)
  ON a.attrelid = c.oid
JOIN (pg_type t  JOIN pg_namespace nt ON t.typnamespace = nt.oid)
  ON a.atttypid = t.oid
WHERE NOT pg_is_other_temp_schema(nc.oid)
  AND nt.nspname = bdr
  AND t.typname = column_timestamps
  AND NOT a.attisdropped
  AND c.relkind IN (r, v, f, p);
```

This query detects the presence of a column of type `bdr.column_timestamp` .

### Timestamps in column-level conflict resolution

Column-level conflict resolution depends on a timestamp column being included in the table.

### Comparing `column_modify_timestamp` and `column_commit_timestamp`

When you select one of the two column-level conflict detection methods, a column is added to the table that contains a mapping of modified columns and timestamps.

The column that stores timestamp mapping is managed automatically. Don't specify or override the value in your queries, as the results can be unpredictable. When possible, user attempts to override the value are ignored.

When enabling or disabling column timestamps on a table, the code uses DDL locking to ensure that there are no pending changes from before the switch. This approach ensures only conflicts with timestamps in both tuples or in neither of them are seen. Otherwise, the code might unexpectedly see timestamps in the local tuple and NULL in the remote one. It also ensures that the changes are resolved the same way (column-level or row-level) on all nodes.

### column_modify_timestamp

When `column_modify_timestamp` is selected as the conflict detection method, the timestamp assigned to the modified columns is the current timestamp, similar to the value you might get running `select_clock_timestamp()` .

This approach is simple and, for many cases, it's correct, for example, when the conflicting rows modify non-overlapping subsets of columns. Its simplicity can, though, lead to unexpected effects.

For example, if an `UPDATE` affects multiple rows, the clock continues ticking while the `UPDATE` runs. So each row gets a slightly different timestamp, even if they're being modified concurrently by the one `UPDATE` . This behavior, in

turn, means that the effects of concurrent changes might get "mixed" in various ways, depending on how the changes performed on different nodes interleaves.

Another possible issue is clock skew. When the clocks on different nodes drift, the timestamps generated by those nodes also drift. This clock skew can induce unexpected behavior such as newer changes being discarded because the timestamps are apparently switched around. However, you can manage clock skew between nodes using the parameters bdr.maximum_clock_skew and bdr.maximum_clock_skew_action .

As the current timestamp is unrelated to the commit timestamp, using it to resolve conflicts means that the result isn't equivalent to the commit order, which means it probably can't be serialized.

When using current timestamps to order changes or commits, the conflicting changes might have exactly the same timestamp because two or more nodes happened to generate the same timestamp. This risk isn't unique to column-level conflict resolution, as it can happen even for regular row-level conflict resolution. The node id is used as the tiebreaker in this situation. The higher node id wins. This approach ensures that the same changes are applied on all nodes.

#### column_commit_timestamp

You can also use the actual commit timestamp specified with `column_commit_timestamp` as the conflict detection method. This approach has the advantage of using the commit time, which is the same for all changes made in an `UPDATE` .

> **Note**
>
> Statement transactions might be added in the future, which will address issues with mixing effects of concurrent statements or transactions. Still, neither of these options can ever produce results equivalent to commit order.

### Inspecting column timestamps

The column storing timestamps for modified columns is maintained by triggers. Don't modify it directly. It can be useful to inspect the current timestamp's value, for example, while investigating how a conflict was resolved.

> **Note**
>
> The timestamp mapping is maintained by triggers, and the order in which triggers execute matters. If your custom triggers modify tuples and are executed after the pgl_clcd_ triggers, the modified columns aren't detected correctly. This can lead to incorrect conflict resolution. If you need to modify tuples in your triggers, make sure they're executed before the pgl_clcd_ triggers.

The following functions are useful for inspecting timestamps.

#### bdr.column_timestamps_to_text(bdr.column_timestamps)

This function returns a human-readable representation of the timestamp mapping and is used when casting the value to text:

```
db=# select cts::text from test_table;
                                                    cts
- ------------------------------------------------------------------------------------
↪----------------
 {source: current, default: 2018-09-23 19:24:52.118583+02, map: [2 : 2018-09-23␣
↪19:25:02.590677+02]}
(1 row)
```

### bdr.column_timestamps_to_jsonb(bdr.column_timestamps)

This function turns a JSONB representation of the timestamps mapping and is used when casting the value to jsonb:

```
db=# select jsonb_pretty(cts::jsonb) from test_table;
                    jsonb_pretty
- ---------------------------------------------------
 {                                                  +
     "map": {                                       +
         "2": "2018-09-23T19:24:52.118583+02:00"    +
     },                                             +
     "source": "current",                           +
     "default": "2018-09-23T19:24:52.118583+02:00"+
 }
(1 row)
```

### bdr.column_timestamps_resolve(bdr.column_timestamps, xid)

This function updates the mapping with the commit timestamp for the attributes modified by the most recent transaction if it already committed. This matters only when using the commit timestamp. For example, in this case, the last transaction updated the second attribute (with attnum = 2):

```
test=# select cts::jsonb from test_table;
                                                                    cts
- ---------------------------------------------------------------------------------------
↪---------------------------------------------------
 {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
↪"2018-09-23T19:29:55.581823+02:00", "modified": [2]}
(1 row)

db=# select bdr.column_timestamps_resolve(cts, xmin)::jsonb from test_table;
                                                 column_timestamps_resolve
- ---------------------------------------------------------------------------------------
↪----------------------------------
 {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
↪"2018-09-23T19:29:55.581823+02:00"}
(1 row)
```

## 6.8.3 Conflict-free replicated data types

Conflict-free replicated data types (CRDTs) support merging values from concurrently modified rows instead of discarding one of the rows as the traditional resolution does.

- *CRDTs Overview* provides an introduction to CRDTs, including how to use CRDTs in tables, configuration options, and examples of CRDTs.
- *Using CRDTs* investigates how to use CRDTs in tables, reviews some configuration options, and reviews some examples of CRDTs and how they work.
- *Operation-based and state-based CRDTs* reviews the differences between operation-based and state-based CRDTs.
- *CRDT Disk-space requirements* covers disk-size considerations for CRDTs, especially state-based CRDTs.
- *CRDTs vs conflict handling/reporting* explains how conflict handling and reporting works with CRDTs.

- *Resetting CRDT values* discusses the challenges of resetting CRDT values and provides some guidance on doing so successfully.
- *Implemented CRDTs* details each of the 6 available CRDTs available in PGD, with implementation examples.

### CRDTs Overview

### Introduction to CRDTs

Conflict-free replicated data types (CRDTs) support merging values from concurrently modified rows instead of discarding one of the rows as the traditional resolution does. Each CRDT type is implemented as a separate PostgreSQL data type with an extra callback added to the `bdr.crdt_handlers` catalog. The merge process happens inside the PGD writer on the apply side without any user action needed.

CRDTs require the table to have column-level conflict resolution enabled, as described in *Special problems for column-level conflict resolution* .

### CRDTs in PostgreSQL

*Implemented CRDTs* are installed as part of `bdr` into the `bdr` schema. For convenience, the basic operators ( `+` , `#` and `!` ) and a number of common aggregate functions ( `min` , `max` , `sum` , and `avg` ) are created in `pg_catalog` . Thus they are available without having to tweak `search_path` .

### Using CRDTs

### Using CRDTs in tables

To use CRDTs, you need to use a particular data type in CREATE/ALTER TABLE rather than standard built-in data types such as `integer` . For example, consider the following table with one regular integer counter and a single row:

### Non-CRDT example

```
CREATE TABLE non_crdt_example (
    id      integer             PRIMARY KEY,
    counter integer             NOT NULL DEFAULT 0
);

INSERT INTO non_crdt_example (id) VALUES (1);
```

Suppose you issue the following SQL on two different nodes at same time:

```
UPDATE non_crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

After both updates are applied, you can see the resulting values using this query:

```
SELECT * FROM non_crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
    1 |         1
(1 row)
```

This code shows that you lost one of the increments due to the `update_if_newer` conflict resolver.

### CRDT example

To use a CRDT counter data type instead, you would follow these steps:

Create the table but with a CRDT (`bdr.crdt_gcounter` ) as the counters data type.

```
CREATE TABLE crdt_example (
    id      integer             PRIMARY KEY,
    counter bdr.crdt_gcounter   NOT NULL DEFAULT 0
);
```

Configure the table for column-level conflict resolution:

```
ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_example,
            column_modify_timestamp, cts);
```

And then insert a row with a value for this example.

```
INSERT INTO crdt_example (id) VALUES (1);
```

If you now issue, as before, the same SQL on two nodes at same time.

```
UPDATE crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

Once the changes are applied, you find that the counter has managed to concurrenct updates.

```
SELECT id, counter FROM crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
    1 |         2
(1 row)
```

This example shows that the CRDT correctly allows the accumulator columns to work, even in the face of asynchronous concurrent updates that otherwise conflict.

### Configuration options

The `bdr.crdt_raw_value` configuration option determines whether queries return the current value or the full internal state of the CRDT type. By default, only the current numeric value is returned. When set to `true` , queries return representation of the full state. You can use the special hash operator ( `#` ) to request only the current numeric value without using the special operator (the default behavior). If the full state is dumped using `bdr.crdt_raw_value = on` , then the value can reload only with `bdr.crdt_raw_value = on` .

**Note**

The *bdr.crdt_raw_value* applies formatting only of data returned to clients, that is, simple column references in the select list. Any column references in other parts of the query (such as *WHERE* clause or even expressions in the select list) might still require use of the # operator.

### Different types of CRDTs

The `crdt_gcounter` type is an example of state-based CRDT types that work only with reflexive UPDATE SQL, such as `x = x + 1` , as the example shows.

Another class of CRDTs are *delta CRDT* types. These are a special subclass of *Operation-based CRDT types (CmCRDT)* .

With delta CRDTs, any update to a value is compared to the previous value on the same node. Then a change is applied as a delta on all other nodes.

```
CREATE TABLE crdt_delta_example (
    id       integer            PRIMARY KEY,
    counter  bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_delta_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_delta_example,
            column_modify_timestamp, cts);

INSERT INTO crdt_delta_example (id) VALUES (1);
```

Suppose you issue the following SQL on two nodes at same time:

```
UPDATE crdt_delta_example
   SET counter = 2          -- notice NOT counter = counter + 2
 WHERE id = 1;
```

After both updates are applied, you can see the resulting values using this query:

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
   id | counter
 -----+---------
    1 |       4
(1 row)
```

With a regular `integer` column, the result is `2` . But when you update the row with a delta CRDT counter, you start with the OLD row version, make a NEW row version, and send both to the remote node. There, compare them with the version found there (e.g., the LOCAL version). Standard CRDTs merge the NEW and the LOCAL version, while delta CRDTs compare the OLD and NEW versions and apply the delta to the LOCAL version.

### Query planning and optimization

An important question is how query planning and optimization works with these new data types. CRDT types are handled transparently. Both `ANALYZE` and the optimizer work, so estimation and query planning works fine without having to do anything else.

### Operation-based and state-based CRDTs

### Operation-based CRDT types (CmCRDT)

The implementation of operation-based types is trivial because the operation isn't transferred explicitly but computed from the old and new row received from the remote node.

Currently, these operation-based CRDTs are implemented:

- Delta counter (`crdt_delta_counter ) <Delta counter (*crdt_delta_counter* )>` _ `bigint` counter (increments/decrements)
- Delta sum (`crdt_delta_sum ) <Delta sum (*crdt_delta_sum* )>` _ `numeric` sum (increments/decrements)

These types leverage existing data types with a little bit of code to compute the delta. For example, `crdt_delta_counter` is a domain on a `bigint` .

This approach is possible only for types for which the method for computing the delta is known, but the result is simple and cheap (both in terms of space and CPU) and has a couple of added benefits. For example, it can leverage operators/syntax for the underlying data type.

The main disadvantage is that you can't reset this value reliably in an asynchronous and concurrent environment.

**Note**

Implementing more complicated operation-based types by creating custom data types is possible, storing the state and the last operation. (Every change is decoded and transferred, so multiple operations aren't needed). But at that point, the main benefits (simplicity, reuse of existing data types) are lost without gaining any advantage compared to state-based types (for example, still no capability to reset) except for the space requirements. (A per-node state isn't needed.)

### State-based CRDT types (CvCRDT)

State-based types require a more complex internal state and so can't use the regular data types directly the way operation-based types do.

Currently, four state-based CRDTs are implemented:

- Grow-only counter (`crdt_gcounter ) <Grow-only counter (*crdt_gcounter* )>` _ `bigint` counter (increment-only)
- Grow-only sum (`crdt_gsum ) <Grow-only sum (*crdt_gsum* )>` _ `numeric` sum/counter (increment-only)
- Positive-negative counter (`crdt_pncounter ) <Positive-negative counter (*crdt_pncounter* )>` _ `bigint` counter (increments/decrements)
- Positive-negative sum (`crdt_pnsum ) <Positive-negative sum (*crdt_pnsum* )>` _ `numeric` sum/counter (increments/decrements)

The internal state typically includes per-node information, increasing the on-disk size but allowing added benefits. The need to implement custom data types implies more code (in/out functions and operators).

The advantage is the ability to reliably reset the values, a somewhat self-healing nature in the presence of lost changes (which doesn't happen in a cluster that operates properly), and the ability to receive changes from other than source nodes.

Consider, for example, that a value is modified on node A, and the change gets replicated to B but not C due to network issue between A and C. If B modifies the value and this change gets replicated to C, it includes even the original change from A. With operation-based CRDTs, node C doesn't receive the change until the A-C network connection starts working again.

The main disadvantages of CvCRDTs are higher costs in terms of *State-based CRDT disk-space reqs* . A bit of information for each node is needed, including nodes that were already removed from the cluster. The complex nature of the state (serialized into varlena types) means increased CPU use.

### CRDT Disk-space requirements

An important consideration is the overhead associated with CRDT types, particularly the on-disk size.

### Operation-based CRDT disk-space reqs

For *operation-based types* , this is trivial because the types are merely domains on top of other types. They have the same disk space requirements no matter how many nodes are there:

- `crdt_delta_counter` _ Same as `bigint` (8 bytes)
- `crdt_delta_sum` _ Same as `numeric` (variable, depending on precision and scale)

There's no dependency on the number of nodes because operation-based CRDT types don't store any per-node information.

### State-based CRDT disk-space reqs

For *state-based types* , the situation is more complicated. All the types are variable length (stored essentially as a `bytea` column) and consist of a header and a certain amount of per-node information for each node that modified the value.

For the `bigint` variants, formulas computing approximate size are:

- `crdt_gcounter` _ `32B (header) + N * 12B (per-node)`
- `crdt_pncounter` - _ `48B (header) + N * 20B (per-node)`

`N` denotes the number of nodes that modified this value.

For the `numeric` variants, there's no exact formula because both the header and per-node parts include `numeric` variable-length values. To give you an idea of how many such values you need to keep:

- `crdt_gsum`
- fixed: `20B (header) + N * 4B (per-node)`
- variable: `(2 + N) numeric` values
- `crdt_pnsum`
- fixed: `20B (header) + N * 4B (per-node)`
- variable: `(4 + 2 * N) numeric` values

> **Note**
>
> It doesn't matter how many nodes are in the cluster if the values are never updated on multiple nodes. It also doesn't matter whether the updates were concurrent (causing a conflict).

In addition, it doesn't matter how many of those nodes were already removed from the cluster. There's no way to compact the state yet.

### CRDTs vs conflict handling/reporting

### CRDT types versus conflicts handling

As tables can contain both CRDT and non-CRDT columns (most columns are expected to be non-CRDT), you need to do both the regular conflict resolution and CRDT merge.

The conflict resolution happens first and is responsible for deciding the tuple to keep (applytuple) and the one to discard. The merge phase happens next, merging data for CRDT columns from the discarded tuple into the applytuple.

> **Note**
>
> This handling makes CRDT types somewhat more expensive compared to plain conflict resolution because the merge needs to happen every time. This is the case even when the conflict resolution can use one of the fast paths (such as those modified in the current transaction).

### CRDT types versus conflict reporting

By default, detected conflicts are individually reported. Without CRDT types, this makes sense because the conflict resolution essentially throws away half of the available information (local or remote row, depending on configuration). This presents a data loss.

CRDT types allow both parts of the information to be combined without throwing anything away, eliminating the data loss issue. This approach makes the conflict reporting unnecessary.

For this reason, conflict reporting is skipped when the conflict can be fully resolved by CRDT merge. Each column must meet at least one of these two conditions:

- The values in local and remote tuple are the same (NULL or equal).
- It uses a CRDT data type and so can be merged.

> **Note**
>
> Conflict reporting is also skipped when there are no CRDT columns but all values in local/remote tuples are equal.

### Resetting CRDT values

Resetting CRDT values is possible but requires special handling. The asynchronous nature of the cluster means that different nodes might see the reset operation at different places in the change stream no matter how it's implemented. Different nodes might also initiate a reset concurrently, that is, before observing the reset from the other node.

In other words, to make the reset operation behave correctly, it needs to be commutative with respect to the regular operations. Many naive ways to reset a value that might work well on a single-node fail for this reason.

### Challenges when resetting CRDT values

For example, the simplest approach to resetting a value might be:

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

With state-based CRDTs this doesn't work. It throws away the state for the other nodes but only locally. It's added back by merge functions on remote nodes, causing diverging values and eventually receiving it back due to changes on the other nodes.

With operation-based CRDTs, this might seem to work because the update is interpreted as a subtraction of `-cnt` . But it works only in the absence of concurrent resets. Once two nodes attempt to do a reset at the same time, the delta is applied twice, getting a negative value (which isn't expected from a reset).

It might also seem that you can use `DELETE + INSERT` as a reset, but this approach has a couple of weaknesses, too. If the row is reinserted with the same key, it's not guaranteed that all nodes see it at the same position in the stream of operations with respect to changes from other nodes. PGD specifically discourages reusing the same primary key value since it can lead to data anomalies in concurrent cases.

### How to reliably handle resetting CRDT values

State-based CRDT types can reliably handle resets using a special `!` operator like this:

```
UPDATE tab SET counter = !counter WHERE ...;
```

"Reliably" means the values don't have the two issues of multiple concurrent resets and divergence.

Operation-based CRDT types can be reset reliably only using a commit scope that performs Eager Replication, such as *QUORUM COMMIT* , since this avoids multiple concurrent resets. You can also use a Quorum Commit scope to set either kind of CRDT to a specific value.

### Implemented CRDTs

Currently, six CRDT data types are implemented:

- Grow-only counter and sum
- Positive-negative counter and sum
- Delta counter and sum

The counters and sums behave mostly the same, except that the counter types are integer based (`bigint` ), while the sum types are decimal-based (`numeric` ).

You can list the currently implemented CRDT data types with the following query:

```
SELECT n.nspname, t.typname
FROM bdr.crdt_handlers c
JOIN (pg_type t JOIN pg_namespace n ON t.typnamespace = n.oid)
  ON t.oid = c.crdt_type_id;
```

### Grow-only counter (`crdt_gcounter` )

- Supports only increments with nonnegative values (`value + int` and `counter + bigint` operators).
- You can obtain the current value of the counter either using `#` operator or by casting it to `bigint` .
- Isn't compatible with simple assignments like `counter = value` (which is common pattern when the new value is computed somewhere in the application).
- Allows simple reset of the counter using the `!` operator ( `counter = !counter` ).
- You can inspect the internal state using `crdt_gcounter_to_text` .

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_gcounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- error: negative value

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
```

(continues on next page)

(continued from previous page)

```
UPDATE crdt_test SET cnt = cnt + 1 WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET cnt = cnt - 1 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET cnt = cnt + (-1) WHERE id = 1;

-- reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

-- get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

-- show internal structure of counters
SELECT id, bdr.crdt_gcounter_to_text(cnt) FROM crdt_test;
```

### Grow-only sum (`crdt_gsum`)

- Supports only increments with nonnegative values (`sum + numeric`).
- You can obtain the current value of the sum either by using the `#` operator or by casting it to `numeric`.
- Isn't compatible with simple assignments like `sum = value`, which is the common pattern when the new value is computed somewhere in the application.
- Allows simple reset of the sum using the `!` operator (`sum = !sum`).
- Can inspect internal state using `crdt_gsum_to_text`.

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    gsum     bdr.crdt_gsum NOT NULL DEFAULT 0.0
);

INSERT INTO crdt_test VALUES (1, 0.0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- error: negative value

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

-- increment sum
UPDATE crdt_test SET gsum = gsum + 11.5 WHERE id = 1;
UPDATE crdt_test SET gsum = gsum + 120.33 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET gsum = gsum - 15.2 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET gsum = gsum + (-1.56) WHERE id = 1;
```

(continues on next page)

(continued from previous page)

```
- - reset sum
UPDATE crdt_test SET gsum = !gsum WHERE id = 1;

- - get current sum value
SELECT id, gsum::numeric, gsum FROM crdt_test;

- - show internal structure of sums
SELECT id, bdr.crdt_gsum_to_text(gsum) FROM crdt_test;
```

### Positive-negative counter (`crdt_pncounter`)

- Supports increments with both positive and negative values (through `counter + int` and `counter + bigint` operators).
- You can obtain the current value of the counter either by using the `#` operator or by casting to `bigint`.
- Isn't compatible with simple assignments like `counter = value`, which is the common pattern when the new value is computed somewhere in the application.
- Allows simple reset of the counter using the `!` operator (`counter = !counter`).
- You can inspect the internal state using `crdt_pncounter_to_text`.

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_pncounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

- - get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

- - show internal structure of counters
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;

- - reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;
```

(continues on next page)

(continued from previous page)

```
- - get current counter value after the reset
SELECT id, cnt::bigint, cnt FROM crdt_test;
```

### Positive-negative sum (`crdt_pnsum`)

- Supports increments with both positive and negative values through `sum + numeric`.
- You can obtain the current value of the sum either by using then `#` operator or by casting to `numeric`.
- Isn't compatible with simple assignments like `sum = value`, which is the common pattern when the new value is computed somewhere in the application.
- Allows simple reset of the sum using the `!` operator (`sum = !sum`).
- You can inspect the internal state using `crdt_pnsum_to_text`.

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    pnsum    bdr.crdt_pnsum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- initialized to -45.31

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment sums
UPDATE crdt_test SET pnsum = pnsum + 1.44     WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum + 12.20    WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum + (-24.34) WHERE id = 3;

- - decrement sums
UPDATE crdt_test SET pnsum = pnsum - 7.3      WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum - 192.83   WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum - (-12.22) WHERE id = 3;

- - get current sum value
SELECT id, pnsum::numeric, pnsum FROM crdt_test;

- - show internal structure of sum
SELECT id, bdr.crdt_pnsum_to_text(pnsum) FROM crdt_test;

- - reset sum
UPDATE crdt_test SET pnsum = !pnsum WHERE id = 1;

- - get current sum value after the reset
SELECT id, pnsum::numeric, pnsum FROM crdt_test;
```

### Delta counter (`crdt_delta_counter`)

- Is defined a `bigint` domain, so works exactly like a `bigint` column.
- Supports increments with both positive and negative values.
- Is compatible with simple assignments like `counter = value`, which is common when the new value is computed somewhere in the application.
- There's no simple way to reset the value reliably.

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

- - get current counter value
SELECT id, cnt FROM crdt_test;
```

### Delta sum (`crdt_delta_sum`)

- Is defined as a `numeric` domain so works exactly like a `numeric` column.
- Supports increments with both positive and negative values.
- Is compatible with simple assignments like `sum = value`, which is common when the new value is computed somewhere in the application.
- There's no simple way to reset the value reliably.

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    dsum     bdr.crdt_delta_sum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129.824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4.531);  -- initialized to -4531
```

(continues on next page)

(continued from previous page)

```
- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET dsum = dsum + 1.32   WHERE id = 1;
UPDATE crdt_test SET dsum = dsum + 12.01  WHERE id = 2;
UPDATE crdt_test SET dsum = dsum + (-2.4) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET dsum = dsum - 7.33   WHERE id = 1;
UPDATE crdt_test SET dsum = dsum - 19.83  WHERE id = 2;
UPDATE crdt_test SET dsum = dsum - (-1.2) WHERE id = 3;

- - get current counter value
SELECT id, cnt FROM crdt_test;
```

CHAPTER 7

# Advanced functionality

## 7.1 DDL replication

DDL stands for data definition language, the subset of the SQL language that creates, alters, and drops database objects.

PGD provides automatic DDL replication, which makes certain DDL changes easier. With automatic replication, you don't have to manually distribute the DDL change to all nodes and ensure that they're consistent.

This section looks at how DDL replication is handled in PGD.

- *DDL overview* provides a general outline of what PGD's DDL replication is capable of.
- *DDL locking details* examines how DDL replication uses locks to safely replicate DDL.
- *Managing DDL with PGD replication* gives best practice advice on why and how to limit the impact of DDL changes so they don't overly affect the smooth running of the cluster.
- *DDL and role manipulation statements* notes issues around manipulating roles over multiple databases in a cluster.
- *Workarounds for DDL restrictions* gives a range of options for handling situations where DDL replication may present restrictions, such as altering columns, constraints, and types.
- *PGD functions that behave like DDL* details the PGD functions that behave like DDL and therefore behave in a similar way and are subject to similar restrictions.

### 7.1.1 DDL overview

DDL stands for data definition language, the subset of the SQL language that creates, alters, and drops database objects.

#### Replicated DDL

For operational convenience and correctness, PGD replicates most DDL actions, with these exceptions:

- Temporary relations
- Certain DDL statements (mostly long running)

- Locking commands (`LOCK` )
- Table maintenance commands (`VACUUM` , `ANALYZE` , `CLUSTER` )
- Actions of autovacuum
- Operational commands (`CHECKPOINT` , `ALTER SYSTEM` )
- Actions related to databases

Automatic DDL replication makes certain DDL changes easier without having to manually distribute the DDL change to all nodes and ensure that they're consistent.

In the default replication set, DDL is replicated to all nodes by default.

### Differences from PostgreSQL

PGD is significantly different from standalone PostgreSQL when it comes to DDL replication. Treating it the same is the most common issue with PGD.

The main difference from table replication is that DDL replication doesn't replicate the result of the DDL. Instead, it replicates the statement. This works very well in most cases, although it introduces the requirement that the DDL must execute similarly on all nodes. A more subtle point is that the DDL must be immutable with respect to all datatype-specific parameter settings, including any datatypes introduced by extensions (not built in). For example, the DDL statement must execute correctly in the default encoding used on each node.

### Executing DDL on PGD systems

A PGD group isn't the same as a standalone PostgreSQL server. It's based on asynchronous multi-master replication without central locking and without a transaction coordinator. This has important implications when executing DDL.

DDL that executes in parallel continues to do so with PGD. DDL execution respects the parameters that affect parallel operation on each node as it executes, so you might notice differences in the settings between nodes.

It's essential to prevent the execution of conflicting DDL statements. Otherwise, DDL replication conflicts occur and replication stops.

PGD offers the following levels of protection against DDL conflicts:

`bdr.ddl_locking = 'all'` is the strictest option and is best when DDL might execute from any node concurrently and you want to ensure correctness.

`bdr.ddl_locking = 'leader'` locks on write-leaders only and doesn't require a majority of nodes to participate in the locking operation.

`bdr.ddl_locking = 'auto'` (the default) automatically selects a `leader` lock if possible, otherwise falls back to `all` .

`bdr.ddl_locking = 'dml'` is an option that is safe only when you execute DDL from one node at any time. Use this setting only if you can completely control where DDL is executed. Executing DDL from a single node ensures that there are no inter-node conflicts. Intra-node conflicts are already handled by PostgreSQL.

`bdr.ddl_locking = 'off'` is the least strict option and is dangerous in general use. This option skips locks altogether, avoiding any performance overhead, which makes it a useful option when creating a new and empty database schema.

These options can be set only by the bdr_superuser, by the superuser, or in the `postgres.conf` configuration file.

When using the bdr.replicate_ddl_command , you can set this parameter directly with the third argument, using the specified bdr.ddl_locking setting only for the DDL commands passed to that function.

#### DDL and mixed PostgreSQL versions

PGD does not support DDL replication between different major Postgres versions in a cluster. Most of the time, this is not an issue because clusters will be running the same major version of Postgres. This is not the case though when performing a rolling upgrade of a cluster from one major version to another. In this case, DDL replication is not supported until all nodes have been upgraded to the same major version and should not be used.

Special care should be taken in the upgrade process when updating extensions as the scripts may trigger DDL replication. In this case, if the scripts must be run before upgrading is complete, `bdr.ddl_replication` setting should be set to `off` while running the script.

### 7.1.2 DDL replication options

The bdr.ddl_replication parameter specifies replication behavior.

`bdr.ddl_replication = on` is the default. This setting replicates DDL to the default replication set, which by default means all nodes. Non-default replication sets don't replicate DDL unless they have a *Replication sets*

defined for them.

You can also replicate DDL to specific replication sets using the function bdr.replicate_ddl_command() . This function can be helpful if you want to run DDL commands when a node is down. It's also helpful if you want to have indexes or partitions that exist on a subset of nodes or rep sets, for example, all nodes at site1.

```
SELECT bdr.replicate_ddl_command(
                CREATE INDEX CONCURRENTLY ON foo (col7);,
                ARRAY[site1],     -- the replication sets
                all);              -- ddl_locking to apply
```

While we don't recommend it, you can skip automatic DDL replication and execute it manually on each node using the bdr.ddl_replication configuration parameter.

```
SET bdr.ddl_replication = off;
```

When set, it makes PGD skip both the global locking and the replication of executed DDL commands. You must then run the DDL manually on all nodes.

> **Warning**
>
> Executing DDL manually on each node without global locking can cause the whole PGD group to stop replicating if conflicting DDL or DML executes concurrently.

Only the bdr_superuser or superuser can set the `bdr.ddl_replication` parameter. It can also be set in the `postgres.conf` configuration file.

### 7.1.3 DDL locking details

Two kinds of locks enforce correctness of replicated DDL with PGD: the global DDL lock and the global DML lock.

#### The global DDL lock

A global DDL lock is used only when `ddl_locking = 'all'` . This kind of lock prevents any other DDL from executing on the cluster while each DDL statement runs. This behavior ensures full correctness in the general case but is too strict for many simple cases. PGD acquires a global lock on DDL operations the first time in a transaction where schema changes are made. This effectively serializes the DDL-executing transactions in the cluster. In other words,

while DDL is running, no other connection on any node can run another DDL command, even if it affects different tables.

To acquire a lock on DDL operations, the PGD node executing DDL contacts the other nodes in a PGD group and asks them to grant it the exclusive right to execute DDL.

The lock request is sent by the regular replication stream, and the nodes respond by the replication stream as well. So it's important that nodes (or at least a majority of the nodes) run without much replication delay. Otherwise it might take a long time for the node to acquire the DDL lock. Once the majority of nodes agree, the DDL execution is carried out.

The ordering of DDL locking is decided using the Raft protocol. DDL statements executed on one node are executed in the same sequence on all other nodes.

To ensure that the node running a DDL has seen effects of all prior DDLs run in the cluster, it waits until it catches up with the node that ran the previous DDL. If the node running the current DDL is lagging behind in replication with respect to the node that ran the previous DDL, then it might take a long time to acquire the lock. Hence it's preferable to run DDLs from a single node or the nodes that have nearly caught up with replication changes originating at other nodes.

A global DDL lock must be granted by a majority of data and witness nodes, where a majority is N/2+1 of the eligible nodes. Subscriber-only nodes aren't eligible to participate.

### The global DML lock

Known as a global DML lock or relation DML lock, this kind of lock is used when either `ddl_locking = all` or `ddl_locking = dml` , and the DDL statement might cause in-flight DML statements to fail. These failures can occur when you add or modify a constraint, such as a unique constraint, check constraint, or NOT NULL constraint. Relation DML locks affect only one relation at a time. These locks ensure that no DDL executes while changes are in the queue that might cause replication to halt with an error.

To acquire the global DML lock on a table, the PGD node executing the DDL contacts all other nodes in a PGD group, asking them to lock the table against writes and waiting while all pending changes to that table are drained. Once all nodes are fully caught up, the originator of the DML lock is free to perform schema changes to the table and replicate them to the other nodes.

The global DML lock holds an EXCLUSIVE LOCK on the table on each node, so it blocks DML, other DDL, VACUUM, and index commands against that table while it runs. This is true even if the global DML lock is held for a command that normally doesn't take an EXCLUSIVE LOCK or higher.

Waiting for pending DML operations to drain can take a long time and even longer if replication is currently lagging. This means that, unlike with data changes, schema changes affecting row representation and constraints can be performed only while all configured nodes can be reached and are keeping up reasonably well with the current write rate. If such DDL commands must be performed while a node is down, first remove the down node from the configuration.

All eligible data nodes must agree to grant a global DML lock before the lock is granted. Witness and subscriber-only nodes aren't eligible to participate.

If a DDL statement isn't replicated, no global locks are acquired.

Specify locking behavior with the bdr.ddl_locking parameter, as explained in *Executing DDL on PGD systems* :

- `ddl_locking = all` takes global DDL lock and, if needed, takes relation DML lock.
- `ddl_locking = leader` enables leader-based global DML locking.
- `ddl_locking = auto` selects `leader` if possible, otherwise `all` .
- `ddl_locking = dml` skips global DDL lock and, if needed, takes relation DML lock.
- `ddl_locking = off` skips both global DDL lock and relation DML lock.

Some PGD functions make DDL changes. For those functions, DDL locking behavior applies, which is noted in the documentation for each function.

Thus, `ddl_locking = dml` is safe only when you can guarantee that no conflicting DDL is executed from other nodes. With this setting, the statements that require only the global DDL lock don't use the global locking at all.

`ddl_locking = off` is safe only when you can guarantee that there are no conflicting DDL and no conflicting DML operations on the database objects DDL executes on. If you turn locking off and then experience difficulties, you might lose in-flight changes to data. The user application team needs to resolve any issues caused.

In some cases, concurrently executing DDL can properly be serialized. If these serialization failures occur, the DDL might reexecute.

DDL replication isn't active on logical standby nodes until they're promoted.

Some PGD management functions act like DDL, meaning that they attempt to take global locks, and their actions are replicated if DDL replication is active. The full list of replicated functions is listed in *PGD functions that behave like DDL* .

DDL executed on temporary tables never need global locks.

ALTER or DROP of an object created in the current transaction doesn't required global DML lock.

Monitoring of global DDL locks and global DML locks is shown in *Monitoring the Connection Manager* .

### 7.1.4 Managing DDL with PGD replication

#### Minimizing the impact of DDL

Minimizing the impact of DDL is good operational advice for any database. These points become even more important with PGD:

- To minimize the impact of DDL, make transactions performing DDL short. Don't combine them with lots of row changes, and avoid long-running foreign key or other constraint rechecks.
- For `ALTER TABLE` , use `ADD CONSTRAINT NOT VALID` followed by another transaction with `VALIDATE CONSTRAINT` rather than using `ADD CONSTRAINT` alone. `VALIDATE CONSTRAINT` waits until replayed on all nodes, which gives a noticeable delay to receive confirmations.
- When indexing, use the `CONCURRENTLY` option whenever possible.

An alternative way of executing long-running DDL is to disable DDL replication and then to execute the DDL statement separately on each node. You can still do this using a single SQL statement, as shown in the following example. Global locking rules still apply, so be careful not to lock yourself out with this type of usage, which is more of a workaround.

```
SELECT bdr.run_on_all_nodes($ddl$
        CREATE INDEX CONCURRENTLY index_a ON table_a(i);
$ddl$);
```

We recommend using the bdr.run_on_all_nodes() technique with `CREATE INDEX CONCURRENTLY` , noting that DDL replication must be disabled for the whole session because `CREATE INDEX CONCURRENTLY` is a multi-transaction command. Avoid `CREATE INDEX` on production systems since it prevents writes while it executes. Avoid using `REINDEX` because of the AccessExclusiveLocks it holds.

Instead, use `REINDEX CONCURRENTLY` (or `reindexdb --concurrently` ).

You can disable DDL replication when using command-line utilities like this:

```
$ export PGOPTIONS="-c bdr.ddl_replication=off"
$ pg_restore --section=post-data
```

Multiple DDL statements might benefit from bunching into a single transaction rather than fired as individual statements, so take the DDL lock only once. This might not be desirable if the table-level locks interfere with normal operations.

If DDL is holding up the system for too long, you can safely cancel the DDL on the originating node with **Control-C** in psql or with `pg_cancel_backend()` . You can't cancel a DDL lock from any other node.

You can control how long the global lock takes with optional global locking timeout settings. bdr.global_lock_timeout limits how long the wait for acquiring the global lock can take before it's canceled. bdr.global_lock_statement_timeout limits the runtime length of any statement in transaction that holds global locks, and bdr.global_lock_idle_timeout sets the maximum allowed idle time (time between statements) for a transaction holding any global locks. You can disable all of these timeouts by setting their values to zero.

Once the DDL operation has committed on the originating node, you can't cancel or abort it. The PGD group must wait for it to apply successfully on other nodes that confirmed the global lock and for them to acknowledge replay. For this reason, keep DDL transactions short and fast.

### Handling DDL with down nodes

If the node initiating the global DDL lock goes down after it acquired the global lock (either DDL or DML), the lock stays active. The global locks don't time out, even if timeouts were set. In case the node comes back up, it releases all the global locks that it holds.

If it stays down for a long time or indefinitely, remove the node from the PGD group to release the global locks. This is one reason for executing emergency DDL using the `SET` command as the bdr_superuser to update the bdr.ddl_locking value.

If one of the other nodes goes down after it confirmed the global lock but before the command acquiring it executed, the execution of that command requesting the lock continues as if the node were up.

As mentioned earlier, the global DDL lock requires only a majority of the nodes to respond, and so it works if part of the cluster is down, as long as a majority is running and reachable. But the DML lock can't be acquired unless the whole cluster is available.

With global DDL or global DML lock, if another node goes down, the command continues normally, and the lock is released.

### Statement-specific DDL replication concerns

Not all commands can be replicated automatically. Such commands are generally disallowed, unless DDL replication is turned off by turning bdr.ddl_replication off.

PGD prevents some DDL statements from running when it's active on a database. This protects the consistency of the system by disallowing statements that can't be replicated correctly or for which replication isn't yet supported.

If a statement isn't permitted under PGD, you can often find another way to do the same thing. For example, you can't do an `ALTER TABLE` , which adds a column with a volatile default value. But generally you can rephrase that as a series of independent `ALTER TABLE` and `UPDATE` statements that work.

Generally, unsupported statements are prevented from executing, raising a `feature_not_supported` (SQLSTATE `0A000` ) error.

Any DDL that references or relies on a temporary object can't be replicated by PGD and throws an error if executed with DDL replication enabled.

### 7.1.5 DDL command handling matrix

The following table describes the utility or DDL commands that are allowed, the ones that are replicated, and the type of global lock they take when they're replicated.

For some more complex statements like `ALTER TABLE` , these can differ depending on the subcommands executed. Every such command has detailed explanation under the following table.

#### Command matrix

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| ALTER AGGREGATE | Y | Y | DDL |
| ALTER CAST | Y | Y | DDL |
| ALTER COLLATION | Y | Y | DDL |
| ALTER CONVERSION | Y | Y | DDL |
| ALTER DATABASE | Y | N | N |
| ALTER DATABASE LINK | Y | Y | DDL |
| ALTER DEFAULT PRIVILEGES | Y | Y | DDL |
| ALTER DIRECTORY | Y | Y | DDL |
| ALTER DOMAIN | Y | Y | DDL |
| ALTER EVENT TRIGGER | Y | Y | DDL |
| ALTER EXTENSION | Y | Y | DDL |
| ALTER FOREIGN DATA WRAPPER | Y | Y | DDL |
| ALTER FOREIGN TABLE | Y | Y | DDL |
| ALTER FUNCTION | Y | Y | DDL |
| ALTER INDEX | Y | Y | DDL |
| ALTER LANGUAGE | Y | Y | DDL |
| ALTER LARGE OBJECT | N | N | N |
| ALTER MATERIALIZED VIEW | Y | *At-least-once delivery guarantees* | DML |
| ALTER OPERATOR | Y | Y | DDL |
| ALTER OPERATOR CLASS | Y | Y | DDL |
| ALTER OPERATOR FAMILY | Y | Y | DDL |
| ALTER PACKAGE | Y | Y | DDL |
| ALTER POLICY | Y | Y | DDL |
| ALTER PROCEDURE | Y | Y | DDL |
| ALTER PROFILE | Y | Y | *DDL locking details* |
| ALTER PUBLICATION | Y | Y | DDL |
| ALTER QUEUE | Y | Y | DDL |
| ALTER QUEUE TABLE | Y | Y | DDL |
| ALTER REDACTION POLICY | Y | Y | DDL |
| ALTER RESOURCE GROUP | Y | N | N |
| ALTER ROLE | Y | Y | DDL |
| ALTER ROUTINE | Y | Y | DDL |

continues on next page

Table 1 – continued from previous page

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| ALTER RULE | Y | Y | DDL |
| ALTER SCHEMA | Y | Y | DDL |
| ALTER SEQUENCE | *DDL locking details* | Y | DML |
| ALTER SERVER | Y | Y | DDL |
| ALTER SESSION | Y | N | N |
| ALTER STATISTICS | Y | Y | DDL |
| ALTER SUBSCRIPTION | Y | Y | DDL |
| ALTER SYNONYM | Y | Y | DDL |
| ALTER SYSTEM | Y | N | N |
| ALTER TABLE | *DDL locking details* | Y | *DDL locking details* |
| ALTER TABLESPACE | Y | Y | DDL |
| ALTER TEXT SEARCH CONFIGURATION | Y | Y | DDL |
| ALTER TEXT SEARCH DICTIONARY | Y | Y | DDL |
| ALTER TEXT SEARCH PARSER | Y | Y | DDL |
| ALTER TEXT SEARCH TEMPLATE | Y | Y | DDL |
| ALTER TRIGGER | Y | Y | DDL |
| ALTER TYPE | Y | Y | DDL |
| ALTER USER MAPPING | Y | Y | DDL |
| ALTER VIEW | Y | Y | DDL |
| ANALYZE | Y | N | N |
| BEGIN | Y | N | N |
| CHECKPOINT | Y | N | N |
| CLOSE | Y | N | N |
| CLOSE CURSOR | Y | N | N |
| CLOSE CURSOR ALL | Y | N | N |
| CLUSTER | Y | N | DML |
| COMMENT | Y | *DDL locking details* | DDL |
| COMMIT | Y | N | N |
| COMMIT PREPARED | Y | N | N |
| COPY | Y | N | N |
| COPY FROM | Y | N | N |
| CREATE ACCESS METHOD | Y | Y | DDL |
| CREATE AGGREGATE | Y | Y | DDL |
| CREATE CAST | Y | Y | DDL |
| CREATE COLLATION | Y | Y | DDL |
| CREATE CONSTRAINT | Y | Y | DDL |
| CREATE CONVERSION | Y | Y | DDL |
| CREATE DATABASE | Y | N | N |
| CREATE DATABASE LINK | Y | Y | DDL |
| CREATE DIRECTORY | Y | Y | DDL |
| CREATE DOMAIN | Y | Y | DDL |
| CREATE EVENT TRIGGER | Y | Y | DDL |
| CREATE EXTENSION | Y | Y | DDL |

continues on next page

Table 1 – continued from previous page

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| CREATE FOREIGN DATA WRAPPER | Y | Y | DDL |
| CREATE FOREIGN TABLE | Y | Y | DDL |
| CREATE FUNCTION | Y | Y | DDL |
| CREATE INDEX | Y | Y | DML |
| CREATE LANGUAGE | Y | Y | DDL |
| CREATE MATERIALIZED VIEW | Y | *At-least-once delivery guarantees* | DML |
| CREATE OPERATOR | Y | Y | DDL |
| CREATE OPERATOR CLASS | Y | Y | DDL |
| CREATE OPERATOR FAMILY | Y | Y | DDL |
| CREATE PACKAGE | Y | Y | DDL |
| CREATE PACKAGE BODY | Y | Y | DDL |
| CREATE POLICY | Y | Y | DML |
| CREATE PROCEDURE | Y | Y | DDL |
| CREATE PROFILE | Y | Y | *DDL locking details* |
| CREATE PUBLICATION | Y | Y | DDL |
| CREATE QUEUE | Y | Y | DDL |
| CREATE QUEUE TABLE | Y | Y | DDL |
| CREATE REDACTION POLICY | Y | Y | DDL |
| CREATE RESOURCE GROUP | Y | N | N |
| CREATE ROLE | Y | Y | DDL |
| CREATE ROUTINE | Y | Y | DDL |
| CREATE RULE | Y | Y | DDL |
| CREATE SCHEMA | Y | Y | DDL |
| CREATE SEQUENCE | *DDL locking details* | Y | DDL |
| CREATE SERVER | Y | Y | DDL |
| CREATE STATISTICS | Y | Y | DDL |
| CREATE SUBSCRIPTION | Y | Y | DDL |
| CREATE SYNONYM | Y | Y | DDL |
| CREATE TABLE | Y | Y | DDL |
| CREATE TABLE AS | *DDL locking details* | Y | DDL |
| CREATE TABLESPACE | Y | Y | DDL |
| CREATE TEXT SEARCH CONFIGURATION | Y | Y | DDL |
| CREATE TEXT SEARCH DICTIONARY | Y | Y | DDL |
| CREATE TEXT SEARCH PARSER | Y | Y | DDL |
| CREATE TEXT SEARCH TEMPLATE | Y | Y | DDL |
| CREATE TRANSFORM | Y | Y | DDL |
| CREATE TRIGGER | Y | Y | DDL |

continues on next page

Table 1 – continued from previous page

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| CREATE TYPE | Y | Y | DDL |
| CREATE TYPE BODY | Y | Y | DDL |
| CREATE USER MAPPING | Y | Y | DDL |
| CREATE VIEW | Y | Y | DDL |
| DEALLOCATE | Y | N | N |
| DEALLOCATE ALL | Y | N | N |
| DECLARE CURSOR | Y | N | N |
| DISCARD | Y | N | N |
| DISCARD ALL | Y | N | N |
| DISCARD PLANS | Y | N | N |
| DISCARD SEQUENCES | Y | N | N |
| DISCARD TEMP | Y | N | N |
| DO | Y | N | N |
| DROP ACCESS METHOD | Y | Y | DDL |
| DROP AGGREGATE | Y | Y | DDL |
| DROP CAST | Y | Y | DDL |
| DROP COLLATION | Y | Y | DDL |
| DROP CONSTRAINT | Y | Y | DDL |
| DROP CONVERSION | Y | Y | DDL |
| DROP DATABASE | Y | N | N |
| DROP DATABASE LINK | Y | Y | DDL |
| DROP DIRECTORY | Y | Y | DDL |
| DROP DOMAIN | Y | Y | DDL |
| DROP EVENT TRIGGER | Y | Y | DDL |
| DROP EXTENSION | Y | Y | DDL |
| DROP FOREIGN DATA WRAPPER | Y | Y | DDL |
| DROP FOREIGN TABLE | Y | Y | DDL |
| DROP FUNCTION | Y | Y | DDL |
| DROP INDEX | Y | Y | DDL |
| DROP LANGUAGE | Y | Y | DDL |
| DROP MATERIALIZED VIEW | Y | *At-least-once delivery guarantees* | DML |
| DROP OPERATOR | Y | Y | DDL |
| DROP OPERATOR CLASS | Y | Y | DDL |
| DROP OPERATOR FAMILY | Y | Y | DDL |
| DROP OWNED | Y | Y | DDL |
| DROP PACKAGE | Y | Y | DDL |
| DROP PACKAGE BODY | Y | Y | DDL |
| DROP POLICY | Y | Y | DDL |
| DROP PROCEDURE | Y | Y | DDL |
| DROP PROFILE | Y | Y | DDL |
| DROP PUBLICATION | Y | Y | DDL |
| DROP QUEUE | Y | Y | DDL |
| DROP QUEUE TABLE | Y | Y | DDL |

continues on next page

Table 1 – continued from previous page

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| DROP REDACTION POLICY | Y | Y | DDL |
| DROP RESOURCE GROUP | Y | N | N |
| DROP ROLE | Y | Y | DDL |
| DROP ROUTINE | Y | Y | DDL |
| DROP RULE | Y | Y | DDL |
| DROP SCHEMA | Y | Y | DDL |
| DROP SEQUENCE | Y | Y | DDL |
| DROP SERVER | Y | Y | DDL |
| DROP STATISTICS | Y | Y | DDL |
| DROP SUBSCRIPTION | Y | Y | DDL |
| DROP SYNONYM | Y | Y | DDL |
| DROP TABLE | Y | Y | DML |
| DROP TABLESPACE | Y | Y | DDL |
| DROP TEXT SEARCH CONFIGURATION | Y | Y | DDL |
| DROP TEXT SEARCH DICTIONARY | Y | Y | DDL |
| DROP TEXT SEARCH PARSER | Y | Y | DDL |
| DROP TEXT SEARCH TEMPLATE | Y | Y | DDL |
| DROP TRANSFORM | Y | Y | DDL |
| DROP TRIGGER | Y | Y | DDL |
| DROP TYPE | Y | Y | DDL |
| DROP TYPE BODY | Y | Y | DDL |
| DROP USER MAPPING | Y | Y | DDL |
| DROP VIEW | Y | Y | DDL |
| EXECUTE | Y | N | N |
| EXPLAIN | Y | *DDL locking details* | *DDL locking details* |
| FETCH | Y | N | N |
| GRANT | Y | *DDL locking details* | DDL |
| GRANT ROLE | Y | Y | DDL |
| IMPORT FOREIGN SCHEMA | Y | Y | DDL |
| LISTEN | Y | N | N |
| LOAD | Y | N | N |
| LOAD ROW DATA | Y | Y | DDL |
| LOCK TABLE | Y | N | *DDL locking details* |
| MOVE | Y | N | N |
| NOTIFY | Y | N | N |
| PREPARE | Y | N | N |
| PREPARE TRANSACTION | Y | N | N |
| REASSIGN OWNED | Y | Y | DDL |
| REFRESH MATERIALIZED VIEW | Y | *At-least-once delivery guarantees* | DML |
| REINDEX | Y | Y | DDL |
| RELEASE | Y | N | N |

continues on next page

Table 1 – continued from previous page

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| RESET | Y | N | N |
| REVOKE | Y | *DDL locking details* | DDL |
| REVOKE ROLE | Y | Y | DDL |
| ROLLBACK | Y | N | N |
| ROLLBACK PREPARED | Y | N | N |
| SAVEPOINT | Y | N | N |
| SECURITY LABEL | Y | *DDL locking details* | DDL |
| SELECT INTO | *DDL locking details* | Y | DDL |
| SET | Y | N | N |
| SET CONSTRAINTS | Y | N | N |
| SHOW | Y | N | N |
| START TRANSACTION | Y | N | N |
| TRUNCATE TABLE | Y | *DDL locking details* | *DDL locking details* |
| UNLISTEN | Y | N | N |
| VACUUM | Y | N | N |

### Command notes

#### ALTER SEQUENCE

Starting with PGD 6.3.0, `ALTER SEQUENCE` is fully supported.

While PGD 6.2.0 introduced `RENAME` and `SET SCHEMA` functionality for `galloc` sequences, the feature did not work for sequences created in PGD versions earlier than 6.2.0 following an upgrade to 6.2.0. The reason for the failure was that PGD 6.2.0 introduced a global UUID identifier to track `galloc` sequences, which was generated at the time of sequence creation, but it was missing for sequences created on an earlier PGD version.

PGD 6.3.0 removes that limitation as BDR manager generates the UUID for sequences if it is missing, enabling the feature for sequences created in previous PGD versions.

#### ALTER TABLE

Generally, `ALTER TABLE` commands are allowed. However, several subcommands aren't supported.

#### ALTER TABLE disallowed commands

Some variants of `ALTER TABLE` currently aren't allowed on a PGD node:

- `ALTER COLUMN ... SET STORAGE external` _ Is rejected if the column is one of the columns of the replica identity for the table. You can override this behavior using `bdr.permit_unsafe_commands` if you're sure the command is safe.
- `RENAME` _ Can't rename an Autopartitioned table.
- `SET SCHEMA` _ Can't set the schema of an Autopartitioned table.
- `ALTER TABLE ... ADD FOREIGN KEY` _ Isn't supported if current user doesn't have permission to read the referenced table or if the referenced table has RLS restrictions enabled that the current user can't bypass.

The following example fails because it tries to add a constant value of type `timestamp` onto a column of type `timestamptz` . The cast between `timestamp` and `timestamptz` relies on the time zone of the session and so isn't immutable.

```
ALTER TABLE foo
  ADD expiry_date timestamptz DEFAULT timestamp 2100-01-01 00:00:00 NOT NULL;
```

You can add certain types of constraints, such as `CHECK` and `FOREIGN KEY` constraints, without taking a DML lock. But this requires a two-step process of first creating a `NOT VALID` constraint and then validating the constraint in a separate transaction with the `ALTER TABLE ... VALIDATE CONSTRAINT` command. See *Adding a CONSTRAINT* for more details.

### ALTER TABLE locking

The following variants of `ALTER TABLE` take only DDL lock and not a DML lock:

- `ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT`
- `ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression`
- `ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT`
- `ALTER TABLE ... ALTER COLUMN ... TYPE` if it doesn't require rewrite
- `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS`
- `ALTER TABLE ... VALIDATE CONSTRAINT`
- `ALTER TABLE ... ATTACH PARTITION`
- `ALTER TABLE ... DETACH PARTITION`
- `ALTER TABLE ... ENABLE TRIGGER` (`ENABLE REPLICA TRIGGER` still takes a DML lock)
- `ALTER TABLE ... CLUSTER ON`
- `ALTER TABLE ... SET WITHOUT CLUSTER`
- `ALTER TABLE ... SET ( storage_parameter = value [, ... ] )`
- `ALTER TABLE ... RESET ( storage_parameter = [, ... ] )`
- `ALTER TABLE ... OWNER TO`

All other variants of `ALTER TABLE` take a DML lock on the table being modified. Some variants of `ALTER TABLE` have restrictions, noted below.

### ALTER TABLE examples

Different types of `ALTER TABLE ... ALTER COLUMN TYPE` (ATCT) operations are possible. Some ATCT operations update only the metadata of the underlying column type and don't require a rewrite of the underlying table data. This is typically the case when the existing column type and the target type are binary coercible.

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(20));
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(128);
```

However, making this change to reverse the command that is changed from VARCHAR(128) to VARCHAR(20) isn't binary coercible and will result in table rewrite.

```
CREATE TABLE sample (col1 BIGINT PRIMARY KEY, col2 VARCHAR(128), col3 INT);
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(256);
```

You can also change the column type to VARCHAR or TEXT data types because of binary coercibility. Again, this is just a metadata update of the underlying column type.

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR;
ALTER TABLE sample ALTER COLUMN col2 TYPE TEXT;
```

### TABLE REWRITE examples:

Users may want to change data types of columns, add columns with a non-null default, or run maintenance jobs. These operations are supported by DDL commands in PostgreSQL which could trigger a table rewrite. The table rewrite operation involves creating a new version of the table on disk. DDL operations such as `CLUSTER` or `VACUUM` directly cause table rewriting without any change in table definition. DDL operations like `ALTER TABLE…ALTER COLUMN TYPE` might change columns and trigger a table rewrite. It copies all the existing data from the old version to the new one, and once the new version is ready and consistent, it replaces the old table with the new one.

Examples of typical table rewrite ATCT operations:

Consider the following table:

```
CREATE TABLE foo(
    c1 int,
    c2 int,
    c3 int,
    name text,
    time_now timestamptz,
    dollar numeric,
    indian_currency int GENERATED ALWAYS AS (dollar * 90.50) STORED
);
```

Given that table, the following ALTER TABLE commands will result in triggering a table rewrite.

Change column data type:

```
ALTER TABLE foo ALTER c1 TYPE bigint;
```

Change column type to incompatible type with USING clause:

```
ALTER TABLE foo ALTER COLUMN name TYPE int USING name::integer;
```

Volatile column types:

```
ALTER TABLE foo ALTER COLUMN time_now TYPE timestamp;
```

Add column with mutable default:

```
ALTER TABLE foo ADD COLUMN time_clock timestamptz NOT NULL DEFAULT clock_timestamp();
```

Generated columns:

```
ALTER TABLE foo ALTER COLUMN indian_currency TYPE numeric;
```

Run multiple `ALTER TABLE` DDL operations in single transaction block:

```
BEGIN;
ALTER TABLE foo ALTER c2 TYPE bigint;
ALTER TABLE foo ALTER c3 TYPE bigint;
COMMIT;
```

In a PGD cluster this involves a rewrite, therefore, the rewrite activity takes the global leader lock of type DML and thus requires that all group leaders in cluster are available.

!!!Note The `ALTER TABLE` examples provided aren't an exhaustive list of possibly allowable ATCT operations.

### ALTER TYPE

`ALTER TYPE` is replicated, but a global DML lock isn't applied to all tables that use that data type, since PostgreSQL doesn't record those dependencies. See *Workarounds for DDL restrictions* .

### COMMENT ON

All variants of `COMMENT ON` are allowed, but `COMMENT ON TABLESPACE/DATABASE/LARGE OBJECT` isn't replicated.

### CREATE PROFILE or ALTER PROFILE

The `PASSWORD_VERIFY_FUNCTION` associated with the profile should be `IMMUTABLE` if the function is `SECURITY DEFINER` . Such a `CREATE PROFILE` or `ALTER PROFILE` command will be replicated but subsequent `CREATE USER` or `ALTER USER` commands using this profile will break the replication due to the `writer` worker throwing the error: `cannot change current role within security-restricted operation` .

### CREATE SEQUENCE

Generally `CREATE SEQUENCE` is supported, but when using global sequences, some options have no effect.

### CREATE TABLE AS and SELECT INTO

`CREATE TABLE AS` and `SELECT INTO` are allowed only if all subcommands are also allowed.

### EXPLAIN

Generally `EXPLAIN` is allowed, but because `EXPLAIN ANALYZE` can have side effects on the database, there are some restrictions on it.

### EXPLAIN ANALYZE Replication

`EXPLAIN ANALYZE` follows replication rules of the analyzed statement.

### EXPLAIN ANALYZE Locking

`EXPLAIN ANALYZE` follows locking rules of the analyzed statement.

### GRANT and REVOKE

Generally `GRANT` and `REVOKE` statements are supported, however `GRANT/REVOKE ON TABLESPACE/LARGE OBJECT` aren't replicated.

### LOCK TABLE

LOCK TABLE isn't replicated, but it might acquire the global DML lock when `bdr.lock_table_locking` is set `on` .

You can also use The `bdr.global_lock_table()` function to explicitly request a global DML lock.

#### MATERIALIZED VIEW

`CREATE MATERIALIZED VIEW` populates data on the upstream node only; the data is then replicated. `REFRESH MATERIALIZED VIEW` also runs on the upstream node only; the result is replicated to downstream nodes.

Materialized view replication is available only on BDRPG, EDB Advanced Server (EDBAS), or 2QPG. A community Postgres build of PGD treats all four commands as unreplicated.

#### SECURITY LABEL

All variants of `SECURITY LABEL` are allowed, but `SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT` isn't replicated.

#### TRUNCATE Replication

`TRUNCATE` command is replicated as DML, not as a DDL statement. Whether the `TRUNCATE` on table is replicated depends on replication settings for each affected table.

#### TRUNCATE Locking

Even though `TRUNCATE` isn't replicated the same way as other DDL, it can acquire the global DML lock when `bdr.truncate_locking` is set to `on` .

### 7.1.6 DDL and role manipulation statements

Users are global objects in a PostgreSQL instance, which means they span multiple databases while PGD operates on an individual database level. Because of this behavior, role manipulation statement handling needs extra thought.

PGD requires that any roles that are referenced by any replicated DDL must exist on all nodes. The roles don't have to have the same grants, password, and so on, but they must exist.

PGD replicates role manipulation statements if bdr.role_replication is enabled (default) and role manipulation statements are run in a PGD-enabled database.

The role manipulation statements include the following:

- `CREATE ROLE`
- `ALTER ROLE`
- `DROP ROLE`
- `GRANT ROLE`
- `CREATE USER`
- `ALTER USER`
- `DROP USER`
- `CREATE GROUP`
- `ALTER GROUP`
- `DROP GROUP`

In general, either:

- Configure the system with `bdr.role_replication = off` , and deploy all role changes (user and group) by external orchestration tools like Ansible, Puppet, and Chef or explicitly replicated by `bdr.replicate_ddl_command()` .

- Configure the system so that exactly one PGD-enabled database on the PostgreSQL instance has `bdr.role_replication = on`, and run all role management DDL on that database.

We recommended that you run all role management commands in one database.

If role replication is turned off, then the administrator must ensure that any roles used by DDL on one node also exist on the other nodes. Otherwise PGD apply stalls with an error until the role is created on the other nodes.

### 7.1.7 Workarounds for DDL restrictions

You can work around some of the limitations of PGD DDL operation handling. Often splitting the operation into smaller changes can produce the desired result that either isn't allowed as a single statement or requires excessive locking.

#### Adding a CONSTRAINT

You can add `CHECK` and `FOREIGN KEY` constraints without requiring a DML lock. This involves a two-step process:

- `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID`
- `ALTER TABLE ... VALIDATE CONSTRAINT`

Execute these steps in two different transactions. Both of these steps take DDL lock only on the table and hence can be run even when one or more nodes are down. But to validate a constraint, PGD must ensure that:

- All nodes in the cluster see the `ADD CONSTRAINT` command.
- The node validating the constraint applied replication changes from all other nodes prior to creating the NOT VALID constraint on those nodes.

So even though the new mechanism doesn't need all nodes to be up while validating the constraint, it still requires that all nodes applied the `ALTER TABLE .. ADD CONSTRAINT ... NOT VALID` command and made enough progress. PGD waits for a consistent state to be reached before validating the constraint.

The new facility requires the cluster to run with Raft protocol version 24 and later. If the Raft protocol isn't yet upgraded, the old mechanism is used, resulting in a DML lock request.

### 7.1.8 PGD functions that behave like DDL

The following PGD management functions act like DDL. This means that, if DDL replication is active and DDL filter settings allow it, they attempt to take global locks, and their actions are replicated. For detailed information, see the documentation for the individual functions.

Replication set management:

- `bdr.create_replication_set <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-management#bdrcreate_replication_set>`_
- `bdr.alter_replication_set <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-management#bdralter_replication_set>`_
- `bdr.drop_replication_set <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-management#bdrdrop_replication_set>`_
- `bdr.replication_set_add_table <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-membership#bdrreplication_set_add_table>`_

- `bdr.replication_set_remove_table <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-membership#bdrreplication_set_remove_table>`_
- `bdr.replication_set_add_ddl_filter <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-ddl-filtering#bdrreplication_set_add_ddl_filter>`_
- `bdr.replication_set_remove_ddl_filter <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/repsets-ddl-filtering#bdrreplication_set_remove_ddl_filter>`_

Conflict management:

- `bdr.alter_table_conflict_detection <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/conflict_functions/#bdralter_table_conflict_detection>`_
- `bdr.column_timestamps_enable` (deprecated; use `bdr.alter_table_conflict_detection()` )
- `bdr.column_timestamps_disable` (deprecated; use `bdr.alter_table_conflict_detection()` )

Sequence management:

- `bdr.alter_sequence_set_kind <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/sequences#bdralter_sequence_set_kind>`_

Stream triggers:

- `bdr.create_conflict_trigger <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/interfaces#bdrcreate_conflict_trigger>`_
- `bdr.create_transform_trigger <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/interfaces#bdrcreate_transform_trigger>`_
- `bdr.drop_trigger <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/interfaces#bdrdrop_trigger>`_

## 7.2 Decoding worker

PGD provides an option to enable a decoding worker process that performs decoding once, no matter how many nodes are sent data. This option introduces a new process, the WAL decoder, on each PGD node. One WAL sender process still exists for each connection, but these processes now just perform the task of sending and receiving data. Taken together, these changes reduce the CPU overhead of larger PGD groups and also allow higher replication throughput since the WAL sender process now spends more time on communication.

### 7.2.1 Enabling

`enable_wal_decoder` is an option for each PGD group, which is currently disabled by default. You can use bdr.alter_node_group_option() to enable or disable the decoding worker for a PGD group.

When the decoding worker is enabled, PGD stores logical change record (LCR) files to allow buffering of changes between decoding and when all subscribing nodes received data. LCR files are stored under the `pg_logical` directory in each local node's data directory. The number and size of the LCR files varies as replication lag increases, so this process also needs monitoring. The LCRs that aren't required by any of the PGD nodes are cleaned periodically. The interval between two consecutive cleanups is controlled by bdr.lcr_cleanup_interval , which defaults to 3 minutes. The cleanup is disabled when bdr.lcr_cleanup_interval is 0.

### 7.2.2 Disabling

When disabled, logical decoding is performed by the WAL sender process for each node subscribing to each node. In this case, no LCR files are written.

Even though the decoding worker is enabled for a PGD group, following GUCs control the production and use of LCR per node. By default these are `false` . For production and use of LCRs, enable the decoding worker for the PGD group and set these GUCs to `true` on each of the nodes in the PGD group.

- bdr.enable_wal_decoder _ When `false` , all WAL senders using LCRs restart to use WAL directly. When `true` along with the PGD group config, a decoding worker process is started to produce LCR and WAL senders that use LCR.
-

**bdr.receive_lcr _ When `true` on the subscribing node, it requests**
WAL sender on the publisher node to use LCRs if available.

A subscriber-only node receives changes from respective nodes directly. Hence a WAL sender serving a subscriber-only node can use LCRs.

Even though LCRs are produced, the corresponding WALs are still retained similar to the case when a decoding worker isn't enabled. In the future, it might be possible to remove WAL corresponding the LCRs, if they aren't otherwise required.

For reference, the first 24 characters of an LCR file name are similar to those in a WAL file name. The first 8 characters of the name are currently all '0'. In the future, they're expected to represent the TimeLineId similar to the first 8 characters of a WAL segment file name. The following sequence of 16 characters of the name is similar to the WAL segment number, which is used to track LCR changes against the WAL stream.

However, logical changes are reordered according to the commit order of the transactions they belong to. Hence their placement in the LCR segments doesn't match the placement of corresponding WAL in the WAL segments.

The set of the last 16 characters represents the subsegment number in an LCR segment. Each LCR file corresponds to a subsegment. LCR files are binary and variable sized. You can control the maximum size of an LCR file by adjusting `bdr.max_lcr_segment_file_size` , which defaults to 1 GB.

#### Using with transaction streaming

It's possible to enable *Using with transaction streaming* and the decoding worker at the same time. Transaction streaming means that the WAL sender can send a partial transaction before it commits, reducing replication lag. The WAL decoder now supports the decoding of partial transactions, so the decoding worker can decode the partial transaction and store it in an LCR file. The LCR file is then used to apply the transaction on the subscriber node. This in turn reduces CPU usage, by reducing the lag, and reduces disk space usages, since ".spill" files are not generated.

The WAL decoder always streams the transactions to LCRs but based on downstream request the WAL sender either stream transaction or just mimics a normal `BEGIN..COMMIT` scenario.

To support this feature, the system creates additional streaming files. These files have names in that begin with `STR_TXN_<file-name-format>` and `CAS_TXN_<file-name-format>` and each streamed transaction creates their own pair.

To enable transaction streaming with the WAL decoder, set the PGD group's `bdr.streaming_mode` set to 'default' using bdr.alter_node_group_option .

## 7.3 CDC Failover support

### 7.3.1 Background

Earlier versions of PGD have allowed the creation of logical replication slots on nodes that can provide a feed of the logical changes happening to the data in the database. These logical replication slots have been local to the node and not replicated. Apart from only replicating changes on the particular node, this behavior has presented challenges when faced with node failover in the cluster. In that scenario, a consumer of the logical replication off a node that fails has no replica of the slot on another node to continue consuming from.

While solutions to this can be engineered using a subscriber-only node as an intermediary, it significantly raises the cost of logical replication.

### 7.3.2 CDC Failover support

To address this need, PGD introduced CDC Failover support. This is an optionally enabled feature that activates automatic logical slot replication across the cluster. This, in turn, allows a consumer of a logical slot's replication to receive change data from any node when a failure occurs.

#### How CDC Failover works

When a logical slot is created on a node with CDC Failover support enabled, the slot is replicated across the cluster. This means that the slot is available for consumption on any node in the cluster. When a node fails, the slot can be consumed from another node in the cluster. This allows for continuing the logical replication stream without interruption.

If, though, the consumer of the slot connects to a different node in the cluster, the previous connection the consumer had will be closed by PGD. This behavior is to ensure that the slot isn't being consumed from multiple nodes at the same time. In the background, PGD is using its Raft consensus protocol to ensure that the slot is being consumed from only one node at a time. This means that the guarantee of only one slot being consumed at a time doesn't hold in split-brain scenarios.

Currently CDC Failover support is a global option that's controlled by a top-group option. The `failover_slot_scope` top-group option can currently be set to (and defaults to) `local` , which disables replication of logical slots, or `global` . The `global` setting enables the replication of all non-temporary logical slots created in the PGD database.

Temporary logical slots aren't replicated, as they have a lifetime scoped to the session that created them and will go away when that session ends.

#### At-least-once delivery guarantees

CDC Failover support takes steps to ensure that the consumer receives all changes at least once. This is done by holding back slots until delivery has been confirmed, at which point the slot is then advanced on all nodes in an asynchronous manner. In the case of a failure on the node where the slot was being consumed, the slot is held until the consumer connects to a node in the cluster. This then allows the slot to progress.

#### Exactly-once delivery

Currently, there's no way to ensure exactly-once delivery, and we expect consuming applications to manage the discarding of previously completed transactions.

### 7.3.3 Enabling CDC Failover support

To enable CDC Failover support run the SQL command and call the bdr.alter_node_group_option function with the following parameters:

```
select bdr.alter_node_group_option(<top-level group name>,
     failover_slot_scope,
     global);
```

Replace `<top-level group name>` with the name of your cluster's top-level group. If you don't know the name, it's the group with a node_group_parent_id equal to 0 in bdr.node_group .

If you do not know the name, it is the group with a node_group_parent_id equal to 0 in bdr.node_group . You can also use:

```
SELECT bdr.alter_node_group_option(
     node_group_name,
     failover_slot_scope,
     global)
    from bdr.node_group
    where node_group_parent_id=0;
```

This command ensures you're setting the correct top-level group's option.

Once CDC Failover is enabled, to create a new globally replicated slot, you can use:

```
SELECT pg_create_logical_replication_slot(myslot,
                                          test_decoding);
```

Logical replication slots created before the option was set to `global` aren't replicated. Only new slots are replicated.

Failover slots can also be created with the `CREATE_REPLICATION_SLOT` command on a replication connection.

The status of failover slots is tracked in the bdr.failover_replication_slots table.

### 7.3.4 CDC Failover support with Postgres 17+

For Postgres 17 and later, support for failover was added to allow standbys to be resumed. Use an option in `pg_create_logical_replication_slot` named `failover` for this purpose. This new setting requires that, no matter what the setting of `failover_slot_scope` , you must also set `failover` to `true` .

```
SELECT pg_create_logical_replication_slot(myslot,
                                          test_decoding,
                                          failover=>true);
```

### 7.3.5 Obtaining Initial Consistent Snapshot

When a logical replication slot is created, a consistent snapshot is exported by Postgres. This snapshot can be used to obtain a consistent initial copy of the data. PGD's failover slot mechanism also follows the same procedure. But the consumer must obtain the snapshot from the same node where the slot was originally created. In addition, it must also start the initial replication from the same node. Once the consumer has received enough changes over the

replication stream, the failover slot is marked as `failover_safe` . Once the slot is marked as `failover_safe` , then the consumer can safely failover to some other node in the PGD cluster (other considerations apply though, see below).

To check if the slot is `failover_safe` or not, the user can query the `bdr.failover_replication_slots` catalog and check for the value of `failover_safe` column of the given slot.

If the consumer connects to some other PGD node and attempts to start replication before the slot is marked `failover_safe` , an appropriate error will be raised by PGD.

### 7.3.6 Failing Over to Newly Joined Nodes

When a new node joins the PGD cluster, it may not be immediately ready to serve as a decoding target for a CDC failover slot. The newly joined node may not have all the WAL files to decode the changes that the consumer has not yet consumed. Consuming from such a node may result in data loss. PGD detects and prevents such situations by internally tracking the replication progress and preventing a new node from being a failover target, until it's safe to do so. If the consumer tries to connect to a node that is not yet ready to serve as a decoding target, an appropriate error will be raised.

### 7.3.7 Tracking Per-Origin Progress

Transactions can originate from any node in the PGD cluster. When a consumer connects to a PGD node and starts decoding transactions, it may receive changes for the transactions originated on that node as well as transactions replicated from other nodes in the cluster. The consumer is expected to track replication progress across all such PGD nodes or origins and ensure that duplicate transactions are handled correctly. To facilitate this, the `test_decoding` plugin in Postgres-Extended and EnterpriseDB Advance Server has been enhanced to include the origin information of the transactions. Consumers can opt to receive origin information by setting `include-origin` option to `on` while starting the logical replication.

A sample output of `test_decoding` plugin with the origin information is produced below.

```
BEGIN 1723654 (origin 2) (origin_name bdr_bdrdemo_bdrgroup_node2) (origin_lsn 0/
↪1D948910)
table public.pgbench_accounts: UPDATE: old-key: aid[integer]:39958 bid[integer]:1␣
↪abalance[integer]:0 filler[character]:                                          ␣
↪                                      new-tuple: aid[integer]:39958 bid[integer]:1␣
↪abalance[integer]:-1783 filler[character]:
table public.pgbench_tellers: UPDATE: old-key: tid[integer]:6 bid[integer]:1␣
↪tbalance[integer]:0 new-tuple: tid[integer]:6 bid[integer]:1 tbalance[integer]:-
↪1783 filler[character]:null
table public.pgbench_branches: UPDATE: old-key: bid[integer]:1 bbalance[integer]:0␣
↪new-tuple: bid[integer]:1 bbalance[integer]:-1783 filler[character]:null
table public.pgbench_history: INSERT: tid[integer]:6 bid[integer]:1␣
↪aid[integer]:39958 delta[integer]:-1783 mtime[timestamp without time zone]:2025-01-
↪31 16:51:21.511571 filler[character]:null
COMMIT 1723654 (origin 2) (origin_name bdr_bdrdemo_bdrgroup_node2) (origin_lsn 0/
↪1D948910)
```

Consumers can make use of this information to track per-origin progress.

PGD also records replication progress across all nodes in the `bdr.logical_checkpoints` catalog and the consumer can receive decoded changes for the catalog and use that information to know the replication progress.

```
BEGIN 65720
id[name]:370098259-0-6056978 origin_node[oid]:370098259 origin_lsn[pg_lsn]:0/6056978␣
↪local_node[oid]:370098259 local_lsn[pg_lsn]:0/6056978 peer_count[integer]:2 peer_
```

(continues on next page)

(continued from previous page)

```
↪nodes[oid[]]:{2228531844,4052927809} peer_lsns[pg_lsn[]]:{0/4836758,0/67F0AF8}
COMMIT 65720
```

In this example, the node 370098259 is reporting the replication progress. When the consumer receives this change record, it can be sure of having received everything up to 0/4836758 and 0/67F0AF8 respectively from nodes 2228531844 and 4052927809.

### 7.3.8 Limitations

The CDC Failover Slot support comes with certain limitations:

- CDC Failover slot support requires the latest versions of EDB Postgres Distributed (PGD) 5.7+ and the latest minor releases of Postgres Extended or EDB Postgres Advanced Server (available Feb 2025).
- CDC Failover support is a global option and can't be set on a per-slot basis. Because changing the enabled status of CDC Failover doesn't affect previously provisioned slots, it's possible to enable it (set to `global` ), create a replicated slot, then disable it (set to `local` ) to create a singular replicated slot.
- CDC Failover support isn't supported on temporary slots.
- CDC Failover support isn't supported on slots created with the `failover` option set to `false` .
- CDC Failover support works with EDB Postgres Advanced Server and EDB Postgres Extended Server only. It isn't supported on community Postgres installations.
- Existing slots aren't converted into failover slots when the option is enabled.
- While Postgres's built-in functions such as `pg_logical_slot_get_changes()` can be used, they won't ensure that the slot isn't being decoded anywhere else and can't update replication progress accurately across the cluster. Therefore, we recommend that you don't rely on the function to receive decoded changes.

## 7.4 Parallel Apply

### 7.4.1 What is Parallel Apply?

Parallel Apply is a feature of PGD that allows a PGD node to use multiple writers per subscription. This behavior generally increases the throughput of a subscription and improves replication performance.

#### Configuring Parallel Apply

Two variables control Parallel Apply in PGD: bdr.max_writers_per_subscription

(defaults to 8) and bdr.writers_per_subscription

(defaults to 2).

```
bdr.max_writers_per_subscription = 8
bdr.writers_per_subscription = 2
```

This configuration gives each subscription two writers. However, in some circumstances, the system might allocate up to eight writers for a subscription.

Changing bdr.max_writers_per_subscription

requires a server restart to take effect.

You can change bdr.writers_per_subscription

for a specific subscription without a restart by:

1. Halting the subscription using bdr.alter_subscription_disable .
2. Setting the new value.
3. Resuming the subscription using bdr.alter_subscription_enable .

First though, establish the name of the subscription using `select * from bdr.subscription` . For this example, the subscription name is `bdr_pgddb_bdrgroup_node2_node1` .

```
SELECT bdr.alter_subscription_disable (bdr_pgddb_bdrgroup_node2_node1);

UPDATE bdr.subscription
SET num_writers = 4
WHERE sub_name = bdr_pgddb_bdrgroup_node2_node1;

SELECT bdr.alter_subscription_enable (bdr_pgddb_bdrgroup_node2_node1);
```

### When to use Parallel Apply

Parallel Apply is always on by default and, for most operations, we recommend leaving it on.

### Monitoring Parallel Apply

To support Parallel Apply's deadlock mitigation, PGD adds columns to bdr.stat_subscription . The new columns are `nprovisional_waits` , `ntuple_waits` , and `ncommmit_waits` . These are metrics that indicate how well Parallel Apply is managing what previously would have been deadlocks. They don't reflect overall system performance.

The `nprovisional_waits` value reflects the number of operations on the same tuples being performed by concurrent apply transactions. These are provisional waits that aren't actually waiting yet but could start waiting.

If a tuple's write needs to wait until it can be safely applied, it's counted in `ntuple_waits` . Fully applied transactions that waited before being committed are counted in `ncommit_waits` .

### Disabling Parallel Apply

To disable Parallel Apply, set bdr.writers_per_subscription to `1` .

### Deadlock mitigation

When Parallel Apply is operating, the transactional changes from the subscription are written by multiple writers. However, each writer ensures that the final commit of its transaction doesn't violate the commit order as executed on the origin node. If there's a violation, an error is generated and the transaction can be rolled back.

This mechanism previously meant that when the following are all true, the resulting error could manifest as a deadlock:

- A transaction is pending commit and modifies a row that another transaction needs to change.
- That other transaction executed on the origin node before the pending transaction did.
- The pending transaction took out a lock request.

Additionally, handling the error could increase replication lag due to a combination of the time taken:

- To detect the deadlock
- For the client to roll back its transaction
- For indirect garbage collection of the changes that were already applied
- To redo the work

This is where Parallel Apply's deadlock mitigation can help. For any transaction, Parallel Apply looks at transactions already scheduled for any row (tuple) that the current transaction wants to write. If it finds one, the row is marked as needing to wait until the other transaction is committed before applying its change to the row. This approach ensures that rows are written in the correct order.

#### Parallel Apply support

In PGD 6, Parallel Apply works with CAMO and Quorum Commit. It isn't compatible with Group Commit. You must disable Parallel Apply if Group Commit is in use.

## 7.5 Replication sets

A replication set is a group of tables that a PGD node can subscribe to. You can use replication sets to create more complex replication topologies than regular symmetric multi-master topologies where each node is an exact copy of the other nodes.

Every PGD group creates a replication set with the same name as the group. This replication set is the default replication set, which is used for all user tables and DDL replication. All nodes are subscribed to it. In other words, by default, all user tables are replicated between all nodes.

### 7.5.1 Using replication sets

You can create replication sets using bdr.create_replication_set , specifying whether to include insert, update, delete, or truncate actions. One option lets you add existing tables to the set, and a second option defines whether to add tables when they're created.

You can also manually define the tables to add or remove from a replication set.

Tables included in the replication set are maintained when the node joins the cluster and afterwards.

Once the node is joined, you can still remove tables from the replication set, but you must add new tables using a resync operation.

By default, a newly defined replication set doesn't replicate DDL or PGD administration function calls. Use bdr.replication_set_add_ddl_filter

to define the commands to replicate.

PGD creates replication set definitions on all nodes. Each node can then be defined to publish or subscribe to each replication set using bdr.alter_node_replication_sets .

You can use functions to alter these definitions later or to drop the replication set.

> **Note**
>
> Don't use the default replication set for selective replication. Don't drop or modify the default replication set on any of the PGD nodes in the cluster, as it's also used by default for DDL replication and administration function calls.

### 7.5.2 Behavior of partitioned tables

PGD supports partitioned tables transparently, meaning that you can add a partitioned table to a replication set.

Changes that involve any of the partitions are replicated downstream.

> **Note**
>
> When partitions are replicated through a partitioned table, the statements executed directly on a partition are replicated as they were executed on the parent table. The exception is the *TRUNCATE* command, which always replicates with the list of affected tables or partitions.

You can add individual partitions to the replication set, in which case they're replicated like regular tables, that is, to the table of the same name as the partition on the downstream. This behavior has some performance advantages if the partitioning definition is the same on both provider and subscriber, as the partitioning logic doesn't have to be executed.

> **Note**
>
> If a root partitioned table is part of any replication set, memberships of individual partitions are ignored. Only the membership of that root table is taken into account.

### 7.5.3 Behavior with foreign keys

A foreign-key constraint ensures that each row in the referencing table matches a row in the referenced table. Therefore, if the referencing table is a member of a replication set, the referenced table must also be a member of the same replication set.

The current version of PGD doesn't check for or enforce this condition. When adding a table to a replication set, the database administrator must make sure that all the tables referenced by foreign keys are also added.

You can use the following query to list all the foreign keys and replication sets that don't satisfy this requirement. The referencing table is a member of the replication set, while the referenced table isn't.

```
SELECT t1.relname,
       t1.nspname,
       fk.conname,
       t1.set_name
  FROM bdr.tables AS t1
  JOIN pg_catalog.pg_constraint AS fk
    ON fk.conrelid = t1.relid
   AND fk.contype = f
 WHERE NOT EXISTS (
  SELECT *
    FROM bdr.tables AS t2
   WHERE t2.relid = fk.confrelid
     AND t2.set_name = t1.set_name
);
```

The output of this query looks like this:

```
 relname | nspname |  conname   | set_name
- --------+---------+------------+----------
 t2      | public  | t2_x_fkey | s2
(1 row)
```

This output means that table `t2` is a member of replication set `s2` , but the table referenced by the foreign key `t2_x_fkey` isn't.

The `TRUNCATE CASCADE` command takes into account the replication set membership before replicating the command. For example:

```
TRUNCATE table1 CASCADE;
```

This becomes a TRUNCATE without cascade on all the tables that are part of the replication set only:

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

### 7.5.4 Replication set membership

You can add tables to or remove them from one or more replication sets. Doing so affects replication only of changes (DML) in those tables. Schema changes (DDL) are handled by DDL replication set filters (see *DDL replication filtering* ).

The replication uses the table membership in replication sets with the node replication sets configuration to determine the actions to replicate and the node to replicate them to. The decision is done using the union of all the memberships and replication set options. Suppose that a table is a member of replication set A that replicates only INSERT actions and replication set B that replicates only UPDATE actions. Both INSERT and UPDATE actions are replicated if the target node is also subscribed to both replication set A and B.

You can control membership using bdr.replication_set_add_table and bdr.replication_set_remove_table .

### 7.5.5 Listing replication sets

You can list existing replication sets with the following query:

```
SELECT set_name
FROM bdr.replication_sets;
```

You can use this query to list all the tables in a given replication set:

```
SELECT nspname, relname
FROM bdr.tables
WHERE set_name = myrepset;
```

*Behavior with foreign keys* shows a query that lists all the foreign keys whose referenced table isn't included in the same replication set as the referencing table.

Use the following SQL to show those replication sets that the current node publishes and subscribes from:

```
SELECT node_id,
       node_name,
       pub_repsets,
       sub_repsets
  FROM bdr.local_node_summary;
```

This code produces output like this:

```
  node_id   | node_name |    pub_repsets     |   sub_repsets
- ----------+-----------+--------------------+--------------------
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
(1 row)
```

To execute the same query against all nodes in the cluster, you can use the following query. This approach gets the replication sets associated with all nodes at the same time.

```
WITH node_repsets AS (
  SELECT jsonb_array_elements(
    bdr.run_on_all_nodes($$
        SELECT
          node_id,
          node_name,
          pub_repsets,
          sub_repsets
        FROM bdr.local_node_summary;
    $$)::jsonb
  ) AS j
)
SELECT j->response->command_tuples->0->>node_id AS node_id,
       j->response->command_tuples->0->>node_name AS node_name,
       j->response->command_tuples->0->>pub_repsets AS pub_repsets,
       j->response->command_tuples->0->>sub_repsets AS sub_repsets
FROM node_repsets;
```

This shows, for example:

```
  node_id   | node_name |    pub_repsets     |    sub_repsets
- ----------+-----------+--------------------------------------
 933864801  | s02db01   | {bdrglobal,bdrs02} | {bdrglobal,bdrs02}
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
 3898940082 | s01db02   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
 1102086297 | s02db02   | {bdrglobal,bdrs02} | {bdrglobal,bdrs02}
(4 rows)
```

### 7.5.6 DDL replication filtering

By default, the replication of all supported DDL happens by way of the default PGD group replication set. This replication is achieved using a DDL filter with the same name as the PGD group. This filter is added to the default PGD group replication set when the PGD group is created.

You can adjust this behavior by changing the DDL replication filters for all existing replication sets. These filters are independent of table membership in the replication sets. Just like data changes, each DDL statement is replicated only once, even if it's matched by multiple filters on multiple replication sets.

You can list existing DDL filters with the following query, which shows, for each filter, the regular expression applied to the command tag and to the role name:

```
SELECT * FROM bdr.ddl_replication;
```

You can use bdr.replication_set_add_ddl_filter and bdr.replication_set_remove_ddl_filter to manipulate DDL filters. They're considered to be DDL and are therefore subject to DDL replication and global locking.

### 7.5.7 Selective replication example

This example configures EDB Postgres Distributed to selectively replicate tables to particular groups of nodes.

### Cluster configuration

This example assumes you have a cluster of six data nodes, `data-a1` to `data-a3` and `data-b1` to `data-b3` in two locations. The two locations they're members of are represented as `region_a` and `region_b` groups.

There's also, as we recommend, a witness node named `witness` in `region-c` that isn't mentioned in this example. The cluster is called `sere` .

### Application requirements

This example works with an application that records the opinions of people who attended performances of musical works. There's a table for attendees, a table for the works, and an opinion table. The opinion table records each work each attendee saw, where and when they saw it, and how they scored the work. Because of data regulation, the example assumes that opinion data must stay only in the region where the opinion was recorded.

### Creating tables

The first step is to create appropriate tables:

```
CREATE TABLE attendee (
   id bigserial PRIMARY KEY,
   email text NOT NULL
);

CREATE TABLE work (
    id int PRIMARY KEY,
    title text NOT NULL,
    author text NOT NULL
);

CREATE TABLE opinion (
    id bigserial PRIMARY KEY,
    work_id int NOT NULL REFERENCES work(id),
    attendee_id bigint NOT NULL REFERENCES attendee(id),
    country text NOT NULL,
    day date NOT NULL,
    score int NOT NULL
);
```

### Viewing groups and replication sets

By default, EDB Postgres Distributed is configured to replicate each table in its entirety to each and every node. This is managed through replication sets.

To view the initial configuration's default replication sets, run:

```
SELECT node_group_name, default_repset, parent_group_name
FROM bdr.node_group_summary;
```

```
 node_group_name | default_repset | parent_group_name
- ----------------+----------------+-------------------
 sere            | sere           |
 region_a        | region_a       | sere
 region_b        | region_b       | sere
 region_c        | region_c       | sere
```

In the output, you can see there's the top-level group, `sere` , with a default replication set named `sere` . Each of the three subgroups has a replication set with the same name as the subgroup. The `region_a` group has a `region_a` default replication set.

By default, all existing tables and new tables become members of the replication set of the top-level group.

### Adding tables to replication sets

The next step is to add tables to the replication sets belonging to the groups that represent the regions. As previously mentioned, all new tables are automatically added to the `sere` replication set. You can confirm that by running:

```
SELECT relname, set_name FROM bdr.tables ORDER BY relname, set_name;
```

```
 relname  | set_name
- ---------+----------
 attendee | sere
 opinion  | sere
 work     | sere
(3 rows)
```

You want the `opinion` table to be replicated only in `region_a` and, separately, only in `region_b` . To do that, you add the table to the replica sets of each region:

```
SELECT bdr.replication_set_add_table(opinion, region_a);
SELECT bdr.replication_set_add_table(opinion, region_b);
```

But you're not done, because `opinion` is still a member of the `sere` replication set. When a table is a member of multiple replication sets, it's replicated in each. This doesn't affect performance, though, as each row is replicated only once on each target node. You don't want `opinion` replicated across all nodes, so you need to remove it from the top-level group's replication set:

```
SELECT bdr.replication_set_remove_table(opinion, sere);
```

You can now review these changes:

```
SELECT relname, set_name FROM bdr.tables ORDER BY relname, set_name;
```

```
 relname  | set_name
- ---------+-------------------
 attendee | sere
 opinion  | region_a
 opinion  | region_b
 work     | sere
(4 rows)
```

This process should provide the selective replication you wanted. To verify whether it did, use the next step to test it.

### Testing selective replication

First create some test data: two works and an attendee. Connect directly to `data-a1` to run this next code:

```
INSERT INTO work VALUES (1, Aida, Verdi);
INSERT INTO work VALUES (2, Lohengrin, Wagner);
INSERT INTO attendee (email) VALUES (gv@example.com);
```

Now that there's some data in these tables, you can insert into the opinion table without violating foreign key constraints:

```
INSERT INTO opinion (work_id, attendee_id, country, day, score)
SELECT work.id, attendee.id, Italy, 1871-11-19, 3
  FROM work, attendee
 WHERE work.title = Lohengrin
   AND attendee.email = gv@example.com;
```

Once you've done the insert, you can validate the contents of the database on the same node:

```
SELECT a.email
, o.country
, o.day
, w.title
, w.author
, o.score
FROM opinion o
JOIN work w ON w.id = o.work_id
JOIN attendee a ON a.id = o.attendee_id;
```

```
    email       | country |    day     |   title   | author | score
- ---------------+---------+------------+-----------+--------+-------
 gv@example.com | Italy   | 1871-11-19 | Lohengrin | Wagner |     3
(1 row)
```

If you now connect to nodes data-a2 and data-a3 and run the same query, you get the same result. The data is being replicated in region_a . If you connect to data-b1 , data-b2 , or data-b3 , the query returns no rows. That's because, although the attendee and work tables are populated, there's no opinion row to select. That, in turn, is because the replication of opinion on region_a happens only in that region.

Now connect to data-b1 and insert an opinion there:

```
INSERT INTO attendee (email) VALUES (fb@example.com);

INSERT INTO opinion (work_id, attendee_id, country, day, score)
SELECT work.id, attendee.id, Germany, 1850-08-27, 9
  FROM work, attendee
 WHERE work.title = Lohengrin
   AND attendee.email = fb@example.com;
```

This opinion is replicated only on region_b . On data-b1 , data-b2 , and data-b3 , you can run:

```
SELECT a.email
, o.country
, o.day
, w.title
, w.author
, o.score
FROM opinion o
JOIN work w ON w.id = o.work_id
JOIN attendee a ON a.id = o.attendee_id;
```

```
    email       | country |    day     |   title    | author | score
- ---------------+---------+------------+------------+--------+-------
 fb@example.com | Germany | 1850-08-27 | Lohengrin | Wagner |     9
(1 row)
```

You see the same result on each of the `region_b` data nodes. Run the query on `region_a` nodes, and you don't see this particular entry.

Finally, notice that the `attendee` table is shared identically across all nodes. On any node, run the query:

```
SELECT * FROM attendee;
```

```
         id          |    email
- --------------------+----------------
 904252679641903104 | gv@example.com
 904261037006536704 | fb@example.com
(2 rows)
```

## 7.6 Security and roles

EDB Postgres Distributed allows a PGD cluster to be administered without giving access to the stored data by design. It achieves this through the use of roles and controlled access to system objects.

- *Security and roles* introduces the roles that PGD predefines for controlling access to PGD functionality.
- *Role management* discusses how roles are managed on multi-database nodes and new nodes.
- *PGD predefined roles* details the specific privileges of the PGD roles.
- *Roles and replication* explains how PGD replication interacts with roles and privileges.
- *Access control* explains how tables, functions, catalog objects and triggers interact with PGD roles and Postgres attributes.

### 7.6.1 Roles

Configuring and managing PGD doesn't require superuser access and we recommend that you don't use superuser access. Instead, the privileges required to administer PGD are split across the following predefined roles.

| Role | Description |
|---|---|
| *bdr_superuser* | The highest-privileged role, having access to all PGD tables and functions. |
| *bdr_read_all_stats* | The role having read-only access to the tables, views, and functions, sufficient to understand the state of PGD. |
| *bdr_monitor* | Includes the privileges of bdr_read_all_stats, with some extra privileges for monitoring. |
| *bdr_application* | The minimal privileges required by applications running PGD. |
| *Replicated large objects* | Can view all conflicts in :ref:``bdr.conflict_history` columns<*bdr.conflict_history* columns>` . |

These roles are named to be analogous to PostgreSQL's `pg_`

predefinedroles .

The PGD `bdr_` roles are created when the BDR extension is installed. See *PGD predefined roles* for more details of the privileges each role has.

Managing PGD doesn't require that administrators have access to user data.

Arrangements for securing information about conflicts are discussed in Logging conflicts to a table .

You can monitor conflicts using the bdr.conflict_history_summary view.

### 7.6.2 Role management

Users are global objects in a PostgreSQL instance. A `CREATE ROLE` command or its alias `CREATE USER` is replicated automatically if it's executed in a PGD replicated database. If a role or user is created in a non-PGD, unreplicated database, the role exists only for that PostgreSQL instance. `GRANT ROLE` and `DROP ROLE` work the same way, replicating only if applied to a PGD-replicated database.

> **Note**
>
> Remember that a user in Postgres terms is simply a role with login privileges.

#### Role rule - No un-replicated roles

If you do create a role or user in a non-PGD, unreplicated database, it's especially important that you do not make an object in the PGD-replicated database rely on that role. It will break the replication process, as PGD cannot replicate a role that is not in the PGD-replicated database.

You can disable this automatic replication behavior by turning off the bdr.role_replication setting, but we don't recommend that.

#### Roles for new nodes

New PGD nodes that are added using bdr_init_physical will automatically replicate the roles from other nodes of the PGD cluster.

Starting with PGD 6.0.1, when a PGD node is manually joined to a PGD group without using `bdr_init_physical`, existing roles are automatically copied to the newly joined node. This means that you no longer need to create roles manually on the new node before joining it to the group.

When roles are copied to a new node, if there are existing roles (or tablespaces) with the same name, the new node's existing roles (or tablespaces) will be updated to share the same settings (including passwords) as the roles (or tablespaces) on the source node in the join operation.

#### Connections and roles

When allocating a new PGD node, the user supplied in the DSN for the `local_dsn` argument of bdr.create_node and the `join_target_dsn` of bdr.join_node_group

are used frequently to refer to, create, and manage database objects.

PGD is carefully written to prevent privilege escalation attacks even when using a role with `SUPERUSER` rights in these DSNs.

To further reduce the attack surface, you can specify a more restricted user in these DSNs. At a minimum, such a user must be granted permissions on all nodes, such that following stipulations are satisfied:

- The user has the `REPLICATION` attribute.
- It's granted the `CREATE` permission on the database.
- It inherits the bdr_superuser role.
- It owns all database objects to replicate, either directly or from permissions from the owner roles.

Also, if any non-default extensions (excluding the BDR extension) are present on the source node, and any of these can be installed only by a superuser, a superuser must create these extensions manually on the join target node. Otherwise the join process will fail.

In PostgreSQL 13 and later, you can identify the extensions requiring superuser permission and that must be manually installed. On the source node, execute:

```
SELECT name, (trusted IS FALSE AND superuser) AS superuser_only
  FROM pg_available_extension_versions
 WHERE installed AND name != bdr;
```

Once all nodes are joined, to continue to allow DML and DDL replication, you can further reduce the permissions to the following:

- The user has the `REPLICATION` attribute.
- It inherits the bdr_superuser role.

### 7.6.3 PGD predefined roles

PGD predefined roles are created when the BDR extension is installed. After BDR extension is dropped from a database, the roles continue to exist. You need to drop them manually if dropping is required.

#### bdr_superuser

This role is for an admin user that can manage anything PGD related. It allows you to separate management of the database and table access. Using it allows you to have a user that can manage the PGD cluster without giving them PostgreSQL superuser privileges.

#### Privileges

- ALL PRIVILEGES ON ALL TABLES IN SCHEMA BDR
- ALL PRIVILEGES ON ALL ROUTINES IN SCHEMA BDR

#### bdr_read_all_stats

This role provides read access to most of the tables, views, and functions that users or applications may need to observe the statistics and state of the PGD cluster.

#### Privileges

`SELECT` privilege on:

- `bdr.autopartition_partitions <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-internal#bdrautopartition_partitions>`_
- `bdr.autopartition_rules <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-internal#bdrautopartition_rules>`_
- `bdr.ddl_epoch <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-internal#bdrddl_epoch>`_
- `bdr.ddl_replication <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/pgd-settings#bdrddl_replication>`_

- `bdr.global_consensus_journal_details <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrglobal_consensus_journal_details>`_
- `bdr.global_lock <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrglobal_lock>`_
- `bdr.global_locks <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrglobal_locks>`_
- `bdr.group_camo_details <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrgroup_camo_details>`_
- `bdr.local_consensus_state <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrlocal_consensus_state>`_
- `bdr.local_node_summary <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrlocal_node_summary>`_
- `bdr.node <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode>`_
- `bdr.node_catchup_info <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_catchup_info>`_
- `bdr.node_catchup_info_details <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_catchup_info_details>`_
- `bdr.node_conflict_resolvers <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_conflict_resolvers>`_
- `bdr.node_group <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_group>`_
- `bdr.node_local_info <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_local_info>`_
- `bdr.node_peer_progress <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_peer_progress>`_
- `bdr.node_replication_rates <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_replication_rates>`_
- `bdr.node_slots <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_slots>`_
- `bdr.node_summary <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrnode_summary>`_

- `bdr.replication_sets <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrreplication_sets>`_
- bdr.replication_status
- `bdr.sequences <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrsequences>`_
- `bdr.stat_activity <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrstat_activity>`_
- `bdr.stat_relation <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrstat_relation>`_
- `bdr.stat_subscription <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrstat_subscription>`_ _deprecated_
-

bdr.state_journal_details

- `bdr.subscription <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrsubscription>`_
- `bdr.subscription_summary <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrsubscription_summary>`_
- `bdr.tables <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrtables>`_
- `bdr.taskmgr_local_work_queue <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrtaskmgr_local_work_queue>`_
- `bdr.taskmgr_work_queue <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrtaskmgr_work_queue>`_
-

bdr.worker_errors _deprecated_

- `bdr.workers <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrworkers>`_
- `bdr.writers <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrwriters>`_
- `bdr.xid_peer_progress`

EXECUTE privilege on:

- `bdr.bdr_edition` _deprecated_

- `bdr.bdr_version <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrbdr_version>`_
- `bdr.bdr_version_num <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrbdr_version_num>`_
- `bdr.decode_message_payload <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrdecode_message_payload>`_
- `bdr.get_consensus_status <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrget_consensus_status>`_
- `bdr.get_decoding_worker_stat <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrget_decoding_worker_stat>`_
- `bdr.get_global_locks <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrget_global_locks>`_
- `bdr.get_min_required_replication_slots <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrget_min_required_replication_slots>`_
- `bdr.get_min_required_worker_processes <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrget_min_required_worker_processes>`_
- `bdr.get_raft_status <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrget_raft_status>`_
- `bdr.get_relation_stats <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrget_relation_stats>`_
- `bdr.get_slot_flush_timestamp <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrget_slot_flush_timestamp>`_
- `bdr.get_sub_progress_timestamp`
- `bdr.get_subscription_stats <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrget_subscription_stats>`_
- `bdr.lag_control <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrlag_control>`_
- `bdr.lag_history <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrlag_history>`_

- `bdr.node_catchup_state_name <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrnode_catchup_state_name>`_
- `bdr.node_kind_name <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrnode_kind_name>`_
- `bdr.peer_state_name <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrpeer_state_name>`_
- `bdr.show_subscription_status <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrshow_subscription_status>`_
- `bdr.show_workers <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrshow_workers>`_
- `bdr.show_writers <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrshow_writers>`_
- `bdr.stat_get_activity <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrstat_get_activity>`_
- `bdr.wal_sender_stats <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrwal_sender_stats>`_
- `bdr.worker_role_id_name <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrworker_role_id_name>`_

### bdr_monitor

This role provides read access to any tables, views, and functions that users or applications may need to monitor the PGD cluster. It includes all the privileges of the *bdr_read_all_stats* role.

### Privileges

All privileges from *bdr_read_all_stats* plus the following additional privileges:

`SELECT` privilege on:

- `bdr.group_raft_details <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrgroup_raft_details>`_
- `bdr.group_replslots_details <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrgroup_replslots_details>`_
- `bdr.group_subscription_summary <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrgroup_subscription_summary>`_

- `bdr.group_versions_details <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/catalogs-visible#bdrgroup_versions_details>`_
- `bdr.raft_instances`

EXECUTE privilege on:

- `bdr.get_raft_instance_by_nodegroup <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrget_raft_instance_by_nodegroup>`_
- `bdr.monitor_camo_on_all_nodes <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrmonitor_camo_on_all_nodes>`_
- `bdr.monitor_group_raft <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrmonitor_group_raft>`_
- `bdr.monitor_group_versions <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrmonitor_group_versions>`_
- `bdr.monitor_local_replslots <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrmonitor_local_replslots>`_
- `bdr.monitor_raft_details_on_all_nodes <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrmonitor_raft_details_on_all_nodes>`_
- `bdr.monitor_replslots_details_on_all_nodes <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrmonitor_replslots_details_on_all_nodes>`_
- `bdr.monitor_subscription_details_on_all_nodes <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrmonitor_subscription_details_on_all_nodes>`_
- `bdr.monitor_version_details_on_all_nodes <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrmonitor_version_details_on_all_nodes>`_
- `bdr.node_group_member_info <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrnode_group_member_info>`_

### bdr_application

This role is designed for applications that require access to PGD features, objects, and functions such as sequences, CRDT datatypes, CAMO status functions, or trigger management functions.

### Privileges

`EXECUTE` privilege on:

- All functions for column_timestamps datatypes
- All functions for CRDT datatypes
- `bdr.alter_sequence_set_kind <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/sequences#bdralter_sequence_set_kind>`_
- `bdr.create_conflict_trigger <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/interfaces#bdrcreate_conflict_trigger>`_
- `bdr.create_transform_trigger <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/interfaces#bdrcreate_transform_trigger>`_
- `bdr.drop_trigger <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/interfaces#bdrdrop_trigger>`_
- `bdr.get_configured_camo_partner <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrget_configured_camo_partner>`_
- `bdr.global_lock_table <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrglobal_lock_table>`_
- `bdr.is_camo_partner_connected <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdris_camo_partner_connected>`_
- `bdr.is_camo_partner_ready <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdris_camo_partner_ready>`_
- `bdr.logical_transaction_status <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrlogical_transaction_status>`_
- bdr.ri_fkey_trigger
- `bdr.seq_nextval <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrseq_nextval>`_
- `bdr.seq_currval <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrseq_currval>`_
- `bdr.seq_lastval <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions-internal#bdrseq_lastval>`_
- `bdr.trigger_get_committs <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/rowfunctions#bdrtrigger_get_committs>`_

- `bdr.trigger_get_conflict_type <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/rowfunctions#bdrtrigger_get_conflict_type>`_
- `bdr.trigger_get_origin_node_id <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/rowfunctions#bdrtrigger_get_origin_node_id>`_
- `bdr.trigger_get_row <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/rowfunctions#bdrtrigger_get_row>`_
- `bdr.trigger_get_type <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/rowfunctions#bdrtrigger_get_type>`_
- `bdr.trigger_get_xid <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/streamtriggers/rowfunctions#bdrtrigger_get_xid>`_
- `bdr.wait_for_camo_partner_queue <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrwait_for_camo_partner_queue>`_
- `bdr.wait_slot_confirm_lsn <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrwait_slot_confirm_lsn>`_
- `bdr.wait_node_confirm_lsn <https://www.enterprisedb.com/docs/pgd/latest/reference/tables-views-functions/functions#bdrwait_node_confirm_lsn>`_

Many of these functions require additional privileges before you can use them. For example, you must be the table owner to successfully execute `bdr.alter_sequence_set_kind` . These additional rules are described with each specific function.

#### bdr_read_all_conflicts

PGD logs conflicts into the bdr.conflict_history

table. Conflicts are visible only to table owners, so no extra privileges are required for the owners to read the conflict history.

However, if it's useful to have a user that can see conflicts for all tables, you can optionally grant the role `bdr_read_all_conflicts` to that user.

#### Privileges

An explicit policy is set on bdr.conflict_history that allows this role to read the `bdr.conflict_history` table.

### 7.6.4 Roles and replication

#### DDL and DML replication and users

DDL changes executed by a user are applied as that same user on each node.

DML changes to tables are replicated as the table-owning user on the target node.

By default, PGD replicates new tables with the same owner across nodes.

#### Differing table ownership

We recommend for the same user to own the table on each node. That's the default behavior, but you can override it. If you do, there are some things to take into account.

Consider a situation where table A is owned by user X on node1 and owned by user Y on node2. If user Y has higher privileges than user X, this might be viewed as a privilege escalation.

Since nodes can have different use cases, we do allow this scenario. But we also warn against it. If tables have different owners on different nodes, we recommend that a security administrator help to plan and audit this configuration.

#### Replication and row-level security

On tables with row-level security policies enabled, changes are replicated without reenforcing policies on apply. This behavior is equivalent to the changes being applied as `NO FORCE ROW LEVEL SECURITY`, even if `FORCE ROW LEVEL SECURITY` is specified. If this isn't what you want, specify a row_filter that avoids replicating all rows. We recommend that the row security policies on all nodes be identical or at least compatible, but we don't enforce this.

#### bdr_superuser role and replication

The user bdr_superuser controls replication for PGD and can add or remove any table from any replication set. bdr_superuser doesn't need any privileges over individual tables, nor do we recommend it. If you need to restrict access to replication set functions, you can implement restricted versions of these functions as `SECURITY DEFINER` functions and grant them to the appropriate users.

#### Privilege restrictions

PGD enforces additional restrictions, effectively preventing the use of DDL that relies solely on TRIGGER or REFERENCES privileges.

`GRANT ALL` still grants both TRIGGER and REFERENCES privileges, so we recommend that you state privileges explicitly. For example, use `GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE` instead of `ALL`.

#### Foreign key privileges

`ALTER TABLE ... ADD FOREIGN KEY` is supported only if the user has SELECT privilege on the referenced table or if the referenced table has RLS restrictions enabled that the current user can't bypass.

This means that the REFERENCES privilege alone isn't sufficient to allow creating a foreign key with PGD. Relying solely on the REFERENCES privilege isn't typically useful since it makes the validation check execute using triggers rather than a table scan. It's typically too expensive to use successfully.

### 7.6.5 Access control

#### Catalog tables

System catalog and information schema tables are always excluded from replication by PGD.

In addition, tables owned by extensions are excluded from replication.

#### PGD functions and operators

All PGD functions are exposed in the `bdr` schema. Any calls to these functions must be schema qualified, rather than putting `bdr` in the search_path.

All PGD operators are available by way of the `pg_catalog` schema to allow users to exclude the `public` schema from the search_path without problems.

### Granting privileges on catalog objects

Administrators must not grant explicit privileges on catalog objects such as tables, views, and functions. Manage access to those objects by granting one of the roles described in *PGD predefined roles* .

This requirement is a consequence of the flexibility that allows joining a node group even if the nodes on either side of the join don't have the exact same version of PGD and therefore of the PGD catalog.

More precisely, if privileges on individual catalog objects were explicitly granted, then the `bdr.join_node_group()` procedure might fail because the corresponding GRANT statements extracted from the node being joined might not apply to the node that's joining.

### Triggers

In PostgreSQL, both the owner of a table and anyone who was granted the TRIGGER privilege can create triggers. Triggers granted by the non-table owner execute as the table owner in PGD, which might cause a security issue. The TRIGGER privilege is seldom used, and PostgreSQL Core Team has said, "The separate TRIGGER permission is something we consider obsolescent."

PGD mitigates this problem by using stricter rules on who can create a trigger on a table:

- superuser: Can create trigggers.
- bdr_superuser: Can create triggers.
- Owner of the table: Can create triggers according to same rules as in PostgreSQL (must have EXECUTE privilege on the function used by the trigger).
- Users who have TRIGGER privilege on the table: Can create a trigger only if they use a function that's owned by the same owner as the table and they satisfy standard PostgreSQL rules. Specifically, they must have EXECUTE privilege on the function. If both table and function have the same owner, and the owner decides to give a user both TRIGGER privilege on the table and EXECUTE privilege on thae function. It's assumed that it's okay for that user to create a trigger on that table using this function.
- Users who have TRIGGER privilege on the table: Can also create triggers using functions that are defined with the

SECURITY DEFINER clause

if they have EXECUTE privilege on them. The SECURITY DEFINER clause makes the function always execute as the owner of the function both in standard PostgreSQL and PGD.

This logic is built on the fact that, in PostgreSQL, the owner of the trigger isn't the user who created it but the owner of the function used by that trigger.

The same rules apply to existing tables, and if the existing table has triggers that aren't owned by the owner of the table and don't use SECURITY DEFINER functions, you can't add it to a replication set.

When PGD replication applies changes it uses the system-level default search_path only. Replica triggers, stream triggers, and index expression functions that assume other search_path settings will then fail when they execute on apply. To ensure this doesn't occur, resolve object references clearly using either the default search_path only, or set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected. When using the default search_path, always use fully qualified references to objects, for example, schema.objectname.

## 7.7 Sequences

Many applications require that unique surrogate ids be assigned to database entries. Often the database `SEQUENCE` object is used to produce these. In PostgreSQL, these can be either:

- A manually created sequence using the `CREATE SEQUENCE` command and retrieved by calling the `nextval()` function

- `serial` and `bigserial` columns or, alternatively, `GENERATED BY DEFAULT AS IDENTITY` columns

However, standard sequences in PostgreSQL aren't multi-node aware and produce values that are unique only on the local node. This is important because unique ids generated by such sequences cause conflict and data loss by means of discarded `INSERT` actions in multi-master replication.

role assigned to them.

For this reason, PGD provides an application-transparent way to generate unique ids using sequences on bigint or bigserial datatypes across the whole PGD group, called *global sequences*.

PGD global sequences provide an easy way for applications to use the database to generate unique synthetic keys in an asynchronous distributed system that works for most—but not necessarily all—cases.

Using PGD global sequences allows you to avoid the problems with insert conflicts. If you define a `PRIMARY KEY` or `UNIQUE` constraint on a column that's using a global sequence, no node can ever get the same value as any other node. When PGD synchronizes inserts between the nodes, they can never conflict.

PGD global sequences extend PostgreSQL sequences, so they are crash-safe. To use them, you must be granted the bdr_application role.

There are various possible algorithms for global sequences:

- SnowflakeId sequences
- Globally allocated range sequences

SnowflakeId sequences generate values using an algorithm that doesn't require inter-node communication at any point. It's faster and more robust and has the useful property of recording the timestamp when the values were created.

SnowflakeId sequences have the restriction that they work only for 64-bit BIGINT datatypes and produce values up to 19 digits long. This might be too long for use in some host language datatypes, such as JavaScript Number types. Globally allocated sequences allocate a local range of values that can be replenished as needed by inter-node consensus, making them suitable for either BIGINT or INTEGER sequences.

You can create a global sequence using the bdr.alter_sequence_set_kind()

function. This function takes a standard PostgreSQL sequence and marks it as a PGD global sequence. It can also convert the sequence back to the standard PostgreSQL sequence.

PGD also provides the configuration variable bdr.default_sequence_kind . This variable determines the kind of sequence to create when the `CREATE SEQUENCE` command is executed or when a `serial` , `bigserial` , or `GENERATED BY DEFAULT AS IDENTITY` column is created. Valid settings are:

- `local` — Newly created sequences are the standard PostgreSQL (local) sequences.
- `galloc` — Always creates globally allocated range sequences.
- `snowflakeid` — Creates global sequences for BIGINT sequences that consist of time, nodeid, and counter components. You can't use it with INTEGER sequences (so you can use it for `bigserial` but not for `serial` ).
- `timeshard` — The older version of SnowflakeId sequence. Provided for backward compatibility only. The SnowflakeId is preferred.
- `distributed` (default) — A special value that you can use only for bdr.default_sequence_kind . It selects `snowflakeid` for `int8` sequences (that is, `bigserial` ) and `galloc` sequence for `int4` (that is, `serial` ) and `int2` sequences.

The bdr.sequences view shows information about individual sequence kinds.

The `currval()` and `lastval()` functions work correctly for all types of global sequences.

### 7.7.1 Automatic sequence conversion

In PGD 6.0 and later, the act of joining a node to a PGD group or creating a new grou also triggers a conversion of any local sequences into global sequences. Set bdr.default_sequence_kind to `distributed` . This setting then selects the best kind of sequence to convert the local sequences into. If `bdr.default_sequence_kind` is set to `local` , the sequences are left as local sequences. Conversions to `galloc` are performed in a way that ensures that the sequence doesn't conflict with any other sequences in the group.

If you decide to start with local sequences and later switch to galloc sequences, you can do so by setting `bdr.default_sequence_kind` to `galloc` and then running the `bdr.alter_sequence_set_kind()` function on each sequence you want to convert. Be aware, though, that you need to manually set the starting values of the sequences to ensure that they don't conflict with any existing values in the table. See *Converting a local sequence to a galloc sequence* for more information about this in general and specifically How to set a new start value for a sequence .

#### SnowflakeId sequences

The ids generated by SnowflakeId sequences are loosely time ordered so you can use them to get the approximate order of data insertion, like standard PostgreSQL sequences. Values generated within the same millisecond might be out of order, even on one node. The property of loose time ordering means they're suitable for use as range-partition keys.

SnowflakeId sequences work on one or more nodes and don't require any inter-node communication after the node-join process completes. So you can continue to use them even if there's the risk of extended network partitions. They aren't affected by replication lag or inter-node latency.

SnowflakeId sequences generate unique ids in a different way from standard sequences. The algorithm uses three components for a sequence number. The first component of the sequence is a timestamp at the time of sequence number generation. The second component of the sequence number is the unique id assigned to each PGD node, which ensures that the ids from different nodes are always different. The third component is the number generated by the local sequence.

While adding a unique node id to the sequence number is enough to ensure there are no conflicts, you also want to keep another useful property of sequences. The ordering of the sequence numbers roughly corresponds to the order in which data was inserted into the table. Putting the timestamp first ensures this.

A few limitations and caveats apply to SnowflakeId sequences.

SnowflakeId sequences are 64 bits wide and need a `bigint` or `bigserial` . Values generated are up to 19 digits long. There's no practical 32-bit `integer` version, so you can't use it with `serial` sequences. Use globally allocated range sequences instead.

For SnowflakeId, there's a limit of 4096 sequence values generated per millisecond on any given node (about 4 million sequence values per second). In case the sequence value generation wraps around within a given millisecond, the SnowflakeId sequence waits until the next millisecond and gets a fresh value for that millisecond.

Since SnowflakeId sequences encode timestamps into the sequence value, you can generate new sequence values only within the given time frame (depending on the system clock). The oldest timestamp that you can use is 2016-10-07, which is the epoch time for the SnowflakeId. The values wrap to negative values in the year 2086 and completely run out of numbers by 2156.

Since timestamp is an important part of a SnowflakeId sequence, there's additional protection from generating sequences with a timestamp older than the latest one used in the lifetime of a Postgres process (but not between Postgres restarts).

The `INCREMENT` option on a sequence used as input for SnowflakeId sequences is effectively ignored. This might be relevant for applications that do sequence ID caching, like many object-relational mapper (ORM) tools, notably Hibernate. Because the sequence is time based, this has little practical effect since the sequence advances to a new noncolliding value by the time the application can do anything with the cached values.

Similarly, you might change the `START` , `MINVALUE` , `MAXVALUE` , and `CACHE` settings on the underlying sequence, but there's no benefit to doing so. The sequence's low 14 bits are used and the rest is discarded, so the value-range limits

don't affect the function's result. For the same reason, `setval()` isn't useful for SnowflakeId sequences.

### Timeshard sequences

Timeshard sequences are provided for backward compatibility with existing installations but aren't recommended for new application use. We recommend using the SnowflakeId sequence instead.

Timeshard is very similar to SnowflakeId but has different limits, fewer protections, and slower performance.

The differences between timeshard and SnowflakeId are as follows:

- Timeshard can generate up to 16384 per millisecond (about 16 million per second), which is more than SnowflakeId. However, there's no protection against wraparound within a given millisecond. Schemas using the timeshard sequence must protect the use of the `UNIQUE` constraint when using timeshard values for a given column.
- The timestamp component of timeshard sequence runs out of values in the year 2050 and, if used in combination with bigint, the values wrap to negative numbers in the year 2033. This means that sequences generated after 2033 have negative values. This is a considerably shorter time span than SnowflakeId and is the main reason why SnowflakeId is preferred.
- Timeshard sequences require occasional disk writes (similar to standard local sequences). SnowflakeIds are calculated in memory so the SnowflakeId sequences are in general a little faster than timeshard sequences.

### Unlogged sequences and PGD

Since Postgres 15, it has been possible to create unlogged sequences. These are related and similar to unlogged tables, which aren't written to the WAL and aren't replicated. In the context of PGD and unlogged sequences, it isn't a sensible configuration to have an unlogged PGD sequence and it could cause unexpected problems in the event of a node failure. Therefore, we prevent the creation of unlogged PGD sequences or the conversion of a PGD sequence to an unlogged sequence.

### Globally allocated range sequences

The globally allocated range (or `galloc` ) sequences allocate ranges (chunks) of values to each node. When the local range is used up, a new range is allocated globally by consensus among the other nodes. This behavior uses the key space efficiently but requires that the local node be connected to a majority of the nodes in the cluster for the sequence generator to progress when the currently assigned local range is used up.

Unlike SnowflakeId sequences, `galloc` sequences support all sequence data types provided by PostgreSQL: `smallint` , `integer` , and `bigint` . This means that you can use `galloc` sequences in environments where 64-bit sequences are problematic. Examples include using integers in JavaScript, since that supports only 53-bit values, or when the sequence is displayed on output with limited space.

The range assigned by each voting node is currently predetermined based on the datatype the sequence is using:

- smallint — 1 000 numbers
- integer — 1 000 000 numbers
- bigint — 1 000 000 000 numbers

Each node allocates two chunks of seq_chunk_size—one for the current use plus a reserved chunk for future use—so the values generated from any one node increase monotonically. However, viewed globally, the values generated aren't ordered at all. This might cause a loss of performance due to the effects on b-tree indexes and typically means that generated values aren't useful as range-partition keys.

The main downside of the `galloc` sequences is that, once the assigned range is used up, the sequence generator has to ask for consensus about the next range for the local node that requires inter-node communication. This might lead to delays or operational issues if the majority of the PGD group isn't accessible. (This might be avoided in later releases.)

The `CACHE` , `START` , `MINVALUE` , and `MAXVALUE` options work correctly with `galloc` sequences. However, you need to set them before transforming the sequence to the `galloc` kind. The `INCREMENT BY` option also works correctly. However, you can't assign an increment value that's equal to or more than the above ranges assigned for each sequence datatype. `setval()` doesn't reset the global state for `galloc` sequences. Don't use it.

PGD tracks `galloc` sequences in the `bdr.sequence_alloc` catalog to manage currently allocated chunks. While Raft manages chunk allocation, changes to a sequence's name or namespace are traditionally handled via the replication stream. Starting with PGD 6.3.0, `ALTER SEQUENCE ... RENAME` and `ALTER SEQUENCE ... SET SCHEMA` are fully supported across all sequence types. For `galloc` sequences, this support relies on a global UUID identifier (`sequuid` ) in the `bdr.sequence_kind catalog` , which allows PGD to track the sequence consistently across name or namespace changes.

See *ALTER SEQUENCE* for more details on managing these objects.

### Converting a local sequence to a galloc sequence

Before transforming a local sequence to galloc, you need to take care of several prerequisites.

#### 1. Verify that sequence and column data type match

Check that the sequence's data type matches the datatype of the column with which it will be used. For example, you can create a `bigint` sequence and assign an `integer` column's default to the `nextval()` returned by that sequence. With galloc sequences, which for `bigint` are allocated in blocks of 1 000 000 000, this quickly results in the values returned by `nextval()` exceeding the `int4` range if more than two nodes are in use.

This example shows what can happen:

```
CREATE SEQUENCE int8_seq;

SELECT sequencename, data_type FROM pg_sequences;
 sequencename | data_type
- -------------+-----------
 int8_seq     | bigint
(1 row)

CREATE TABLE seqtest (id INT NOT NULL PRIMARY KEY);

ALTER SEQUENCE int8_seq OWNED BY seqtest.id;

SELECT bdr.alter_sequence_set_kind(public.int8_seq::regclass, galloc, 1);
 alter_sequence_set_kind
- ------------------------

(1 row)

ALTER TABLE seqtest ALTER COLUMN id SET DEFAULT nextval(int8_seq::regclass);
```

After executing `INSERT INTO seqtest VALUES(DEFAULT)` on two nodes, the table contains the following values:

```
SELECT * FROM seqtest;
     id
- -----------
          2
 2000000002
(2 rows)
```

However, attempting the same operation on a third node fails with an `integer out of range` error, as the sequence generated the value `4000000002` .

**Tip**

You can retrieve the current data type of a sequence from the PostgreSQL

pg_sequences

view. You can modify the data type of a sequence with `ALTER SEQUENCE ... AS ...` , for example, `ALTER SEQUENCE public.sequence AS integer` , as long as its current value doesn't exceed the maximum value of the new data type.

### 2. Set a new start value for the sequence

When the sequence kind is altered to `galloc` , it's rewritten and restarts from the defined start value of the local sequence. If this happens on an existing sequence in a production database, you need to query the current value and then set the start value appropriately. To help with this use case, PGD lets you pass a starting value with the function bdr.alter_sequence_set_kind() . If you're already using offset and you have writes from multiple nodes, you need to check what's the greatest used value and restart the sequence to at least the next value:

```
- - determine highest sequence value across all nodes
SELECT max((x->response->command_tuples->0->>nextval)::bigint)
    FROM jsonb_array_elements(
        bdr.run_on_all_nodes(
            ESELECT nextval(\public.sequence\);
            )::jsonb) AS x;

- - turn into a galloc sequence
SELECT bdr.alter_sequence_set_kind(public.sequence::regclass, galloc, $MAX + $MARGIN);
```

Since users can't lock a sequence, you must leave a `$MARGIN` value to allow operations to continue while the `max()` value is queried.

The bdr.sequence_alloc table gives information on the chunk size and the ranges allocated around the whole cluster.

In this example, the sequence starts at `333` , and the cluster has two nodes. The number of allocation is 4, which is 2 per node, and the chunk size is 1000000, which is related to an integer sequence.

```
SELECT * FROM bdr.sequence_alloc
    WHERE seqid = public.categories_category_seq::regclass;
          seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |      ␣
↪seq_last_alloc
- ------------------------+----------------+---------------------+-------------+------
↪------------------------
 categories_category_seq |        1000000 |             4000333 |           4 | 2020-
↪05-21 20:02:15.957835+00
(1 row)
```

To see the ranges currently assigned to a given sequence on each node, execute the function bdr.galloc_chunk_info .

- Node `Node1` is using range from `333` to `2000333` .

```
SELECT * FROM bdr.galloc_chunk_info(categories_category_seq);
 chunk_start | chunk_end
```

(continues on next page)

(continued from previous page)

```
- ------------+-----------
         334 |   1000333
     1000334 |   2000333
(2 rows)
```

- Node `Node2` is using range from `2000334` to `4000333` .

```
SELECT * FROM bdr.galloc_chunk_info(categories_category_seq);
 chunk_start | chunk_end
- ------------+-----------
     2000334 |   3000333
     3000334 |   4000333
```

When a node finishes a chunk, it asks a consensus for a new one and gets the first available. In the example, it's from 4000334 to 5000333. This is the new reserved chunk and starts to consume the old reserved chunk.

### 7.7.2 UUIDs, KSUUIDs, and other approaches

You can generate globally unique ids in other ways without using the global sequences that can be used with PGD. For example:

- UUIDs and their PGD variant, KSUUIDs
- Local sequences with a different offset per node (i.e., manual)
- An externally coordinated natural key

PGD applications can't use other methods safely. Counter-table-based approaches relying on `SELECT ... FOR UPDATE` , `UPDATE ... RETURNING ...` or similar for sequence generation don't work correctly in PGD because PGD doesn't take row locks between nodes. The same values are generated on more than one node. For the same reason, the usual strategies for "gapless" sequence generation don't work with PGD. In most cases, the application coordinates generating sequences that must be gapless from some external source using two-phase commit. Or it generates them only on one node in the PGD group.

### 7.7.3 KSUUID v2 functions

PGD applications can't use other methods safely. Counter-table-based approaches relying on `SELECT ... FOR UPDATE` , `UPDATE ... RETURNING ...` or similar for sequence generation don't work correctly in PGD because PGD doesn't take row locks between nodes. The same values are generated on more than one node. For the same reason, the usual strategies for "gapless" sequence generation don't work with PGD. In most cases, the application coordinates generating sequences that must be gapless from some external source using two-phase commit. Or it generates them only on one node in the PGD group.

#### UUIDs

`UUID` keys instead avoid sequences entirely and use 128-bit universal unique identifiers. These are random or pseudo-random values that are so large that it's nearly impossible for the same value to be generated twice. There's no need for nodes to have continuous communication when using `UUID` keys.

In the unlikely event of a collision, conflict detection chooses the newer of the two inserted records to retain. Conflict logging, if enabled, records such an event. However, it's exceptionally unlikely to ever occur, since collisions become practically likely only after about 2^64 keys are generated.

The main downside of `UUID` keys is that they're somewhat inefficient in terms of space and the network. They consume more space not only as a primary key but also where referenced in foreign keys and when transmitted on the wire. Also, not all applications cope well with `UUID` keys.

### KSUUIDs

PGD provides functions for working with a K-sortable variant of `UUID` data. Known as KSUUID, it generates values that can be stored using the PostgreSQL standard `UUID` data type. A `KSUUID` value is similar to `UUIDv1` in that it stores both timestamp and random data, following the `UUID` standard. The difference is that `KSUUID` is K-sortable, meaning that it's weakly sortable by timestamp. This makes it more useful as a database key, as it produces more compact `btree` indexes. This behavior improves the effectiveness of search and allows natural time-sorting of result data. Unlike `UUIDv1` , `KSUUID` values don't include the MAC of the computer on which they were generated, so there are no security concerns from using them.

We now recommend `KSUUID` v2 in all cases. You can directly sort values generated with regular comparison operators.

There are two versions of `KSUUID` in PGD: v1 and v2. The legacy `KSUUID` v1 is deprecated but is kept to support existing installations. Don't use it for new installations. The internal contents of v1 and v2 aren't compatible. As such, the functions to manipulate them also aren't compatible. The v2 of `KSUUID` also no longer stores the `UUID` version number.

See KSUUID v2 functions and KSUUID v1 functions in the PGD reference.

### Step and offset sequences

In offset-step sequences, a normal PostgreSQL sequence is used on each node. Each sequence increments by the same amount and starts at differing offsets. For example, with step 1000, node1's sequence generates 1001, 2001, 3001, and so on. node2's sequence generates 1002, 2002, 3002, and so on. This scheme works well even if the nodes can't communicate for extended periods. However, the designer must specify a maximum number of nodes when establishing the schema, and it requires per-node configuration. Mistakes can easily lead to overlapping sequences.

It's relatively simple to configure this approach with PGD by creating the desired sequence on one node, like this:

```
CREATE TABLE some_table (
    generated_value bigint primary key
);

CREATE SEQUENCE some_seq INCREMENT 1000 OWNED BY some_table.generated_value;

ALTER TABLE some_table ALTER COLUMN generated_value SET DEFAULT nextval(some_seq);
```

Then, on each node calling `setval()` , give each node a different offset starting value, for example:

```
- - On node 1
SELECT setval(some_seq, 1);

- - On node 2
SELECT setval(some_seq, 2);

 -- ... etc
```

Be sure to allow a large enough `INCREMENT` to leave room for all the nodes you might ever want to add, since changing it in the future is difficult and disruptive.

If you use `bigint` values, there's no practical concern about key exhaustion, even if you use offsets of 10000 or more. It would take hundreds of years, with hundreds of machines, doing millions of inserts per second, to have any chance of approaching exhaustion.

PGD doesn't currently offer any automation for configuring the per-node offsets on such step/offset sequences.

#### Composite keys

A variant on step/offset sequences is to use a composite key composed of `PRIMARY KEY (node_number, generated_value)` . The node number is usually obtained from a function that returns a different number on each node. You can create such a function by temporarily disabling DDL replication and creating a constant SQL function. Alternatively, you can use a one-row table that isn't part of a replication set to store a different value in each node.

### 7.7.4 See also

- Global Sequence management interfaces
- KSUUID v2 functions
- KSUUID v1 functions

## 7.8 Replicated large objects

PGD provides a replicated large object implementation that mirrors the

PostgreSQL large object interface . Unlike standard PostgreSQL large objects, which aren't replicated by logical replication, PGD large objects are logically replicated across all nodes in the cluster. Object identifiers (OIDs) are generated using the galloc sequence to ensure uniqueness across all nodes.

Use replicated large objects when your application or driver relies on the PostgreSQL large object interface and you need that data replicated across all nodes. Replicated large objects are commonly used when migrating an existing application to PGD that already uses large objects, or when using a driver or ORM that manages binary data through the large object API.

For new applications, `BYTEA` is simpler and requires no special configuration. See *Preparing your application for PGD* for guidance on choosing the right approach.

### 7.8.1 Using replicated large objects

PGD offers two ways to work with replicated large objects: calling the `bdr.lo_*` functions directly, or enabling transparent redirection so existing applications work without code changes.

#### Calling bdr.lo_* functions

PGD exposes equivalents to all PostgreSQL large object server-side functions in the `bdr` schema, for example `bdr.lo_create()` instead of `lo_create()` and `bdr.lo_open()` instead of `lo_open()` . See *Replicated large objects* for the full function reference.

#### Redirecting PostgreSQL large object calls

Set `bdr.override_large_objects = on` to redirect calls to standard PostgreSQL `lo_*` functions to their `bdr.lo_*` equivalents automatically, so existing applications use replicated large objects without code changes.

Redirection applies to SQL-level calls. libpq-based clients and drivers also issue large object calls at the protocol level, bypassing SQL. To redirect those too, connect to the *Connection Manager Authentication* read-write port (default 6432) instead of connecting directly to Postgres.

### 7.8.2 How large objects are stored

Replicated large objects are stored in *bdr.pgd_largeobject* , with permissions stored in *bdr.pgd_largeobject_metadata* . These tables serve the same purpose as `pg_catalog.largeobject` and `pg_catalog.largeobject_metadata` , but their contents are replicated to all nodes.

### 7.8.3 Limitations

- `ALTER LARGE OBJECT` isn't supported.
- Large objects created before enabling this feature are stored in `pg_catalog.largeobject` and aren't accessible through the `bdr.lo_*` interface.
- `\lo_list` in psql doesn't display PGD large objects because it reads `pg_catalog.largeobject` directly.

## 7.9 Stream triggers

PGD introduces new types of triggers that you can use for additional data processing on the downstream/target node:

- Conflict triggers
- Transform triggers

Together, these types of triggers are known as *stream triggers*.

Stream triggers are designed to be trigger-like in syntax. They leverage the PostgreSQL BEFORE trigger architecture and are likely to have similar performance characteristics as PostgreSQL BEFORE triggers.

Multiple trigger definitions can use one trigger function, just as with normal PostgreSQL triggers. A trigger function is a program defined in this form: `CREATE FUNCTION ... RETURNS TRIGGER`. Creating the trigger doesn't require use of the `CREATE TRIGGER` command. Instead, create stream triggers using the special PGD functions bdr.create_conflict_trigger() and

bdr.create_transform_trigger() .

Once created, the trigger is visible in the catalog table `pg_trigger`. The stream triggers are marked as `tgisinternal = true` and `tgenabled = 'D'` and have the name suffix '_bdrc' or '_bdrt'. The view `bdr.triggers` provides information on the triggers in relation to the table, the name of the procedure that's being executed, the event that triggers it, and the trigger type.

Stream triggers aren't enabled for normal SQL processing. Because of this, the `ALTER TABLE ... ENABLE TRIGGER` is blocked for stream triggers in both its specific name variant and the ALL variant. This mechanism prevents the trigger from executing as a normal SQL trigger.

These triggers execute on the downstream or target node. There's no option for them to execute on the origin node. However, you might want to consider the use of `row_filter` expressions on the origin.

Also, any DML that's applied while executing a stream trigger isn't replicated to other PGD nodes and doesn't trigger the execution of standard local triggers. This is intentional. You can use it, for example, to log changes or conflicts captured by a stream trigger into a table that's crash-safe and specific to that node. See *Stream triggers examples* for a working example.

### 7.9.1 Trigger execution during apply

Transform triggers execute first _ once for each incoming change in the triggering table. These triggers fire before we attempt to locate a matching target row, allowing a very wide range of transforms to be applied efficiently and consistently.

Next, for UPDATE and DELETE changes, we locate the target row. If there's no target row, then no further processing occurs for those change types.

We then execute any normal triggers that previously were explicitly enabled as replica triggers at table level:

```
ALTER TABLE tablename
ENABLE REPLICA TRIGGER trigger_name;
```

We then decide whether a potential conflict exists. If so, we then call any conflict trigger that exists for that table.

#### Missing-column conflict resolution

Before transform triggers are executed, PostgreSQL tries to match the incoming tuple against the row-type of the target table.

Any column that exists on the input row but not on the target table triggers a conflict of type `target_column_missing`. Conversely, a column existing on the target table but not in the incoming row triggers a `source_column_missing` conflict. The default resolutions for those two conflict types are respectively `ignore_if_null` and `use_default_value`.

This is relevant in the context of rolling schema upgrades, for example, if the new version of the schema introduces a new column. When replicating from an old version of the schema to a new one, the source column is missing, and the `use_default_value` strategy is appropriate, as it populates the newly introduced column with the default value.

However, when replicating from a node having the new schema version to a node having the old one, the column is missing from the target table. The `ignore_if_null` resolver isn't appropriate for a rolling upgrade because it breaks replication as soon as a user inserts a tuple with a non-NULL value in the new column in any of the upgraded nodes.

In view of this example, the appropriate setting for rolling schema upgrades is to configure each node to apply the `ignore` resolver in case of a `target_column_missing` conflict.

You can do this with the following query, which you must execute separately on each node. Replace `node1` with the actual node name.

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
    target_column_missing, ignore);
```

#### Data loss and divergence risk

Setting the conflict resolver to `ignore` can lead to data loss and cluster divergence.

Consider the following example: table `t` exists on nodes 1 and 2, but its column `col` exists only on node 1.

If the conflict resolver is set to `ignore`, then there can be rows on node 1 where `c` isn't null, for example, `(pk=1, col=100)`. That row is replicated to node 2, and the value in column `c` is discarded, for example, `(pk=1)`.

If column `c` is then added to the table on node 2, it's at first set to NULL on all existing rows, and the row considered above becomes `(pk=1, col=NULL)`. The row having `pk=1` is no longer identical on all nodes, and the cluster is therefore divergent.

The default `ignore_if_null` resolver isn't affected by this risk because any row replicated to node 2 has `col=NULL`.

Based on this example, we recommend running LiveCompare against the whole cluster at the end of a rolling schema upgrade where the `ignore` resolver was used. This practice helps to ensure that you detect and fix any divergence.

### 7.9.2 Terminology of row-types

PGD uses these row-types:

- `SOURCE_OLD` is the row before update, that is, the key.
- `SOURCE_NEW` is the new row coming from another node.
- `TARGET` is the row that exists on the node already, that is, the conflicting row.

### 7.9.3 Conflict triggers

Conflict triggers execute when a conflict is detected by PGD. They decide what happens when the conflict occurs.

- If the trigger function returns a row, the action is applied to the target.
- If the trigger function returns a NULL row, the action is skipped.

For example, if the trigger is called for a `DELETE` , the trigger returns NULL if it wants to skip the `DELETE` . If you want the `DELETE` to proceed, then return a row value: either `SOURCE_OLD` or `TARGET` works. When the conflicting operation is either `INSERT` or `UPDATE` , and the chosen resolution is to delete the conflicting row, the trigger must explicitly perform the deletion and return NULL. The trigger function can perform other SQL actions as it chooses, but those actions are only applied locally, not replicated.

When a real data conflict occurs between two or more nodes, two or more concurrent changes are occurring. When the changes are applied, the conflict resolution occurs independently on each node. This means the conflict resolution occurs once on each node and can occur with a significant time difference between them. As a result, communication between the multiple executions of the conflict trigger isn't possible. It's the responsibility of the author of the conflict trigger to ensure that the trigger gives exactly the same result for all related events. Otherwise, data divergence occurs.

If the same conflict trigger matches more than one event, you can use the `TG_OP` variable in the trigger to identify the operation that produced the conflict.

By default, PGD detects conflicts by observing a change of replication origin for a row. Hence, you can call a conflict trigger even when only one change is occurring. Since, in this case, there's no real conflict, this conflict detection mechanism can generate false-positive conflicts. The conflict trigger must handle all of those identically.

In some cases, timestamp conflict detection doesn't detect a conflict at all. For example, in a concurrent `UPDATE` /`DELETE` where the `DELETE` occurs just after the `UPDATE` , any nodes that see first the `UPDATE` and then the `DELETE` don't see any conflict. If no conflict is seen, the conflict trigger is never called. In the same situation but using row-version conflict detection, a conflict is seen, which a conflict trigger can then handle.

The trigger function has access to additional state information as well as the data row involved in the conflict, depending on the operation type:

- On `INSERT` , conflict triggers can access the `SOURCE_NEW` row from the source and `TARGET` row.
- On `UPDATE` , conflict triggers can access the `SOURCE_OLD` and `SOURCE_NEW` row from the source and `TARGET` row.
- On `DELETE` , conflict triggers can access the `SOURCE_OLD` row from the source and `TARGET` row.

You can use the function `bdr.trigger_get_row()` to retrieve `SOURCE_OLD` , `SOURCE_NEW` , or `TARGET` rows, if a value exists for that operation.

Changes to conflict triggers happen transactionally and are protected by global DML locks during replication of the configuration change. This behavior is similar to how some variants of `ALTER TABLE` are handled.

If primary keys are updated inside a conflict trigger, it can sometimes lead to unique constraint violations errors due to a difference in timing of execution. Hence, avoid updating primary keys in conflict triggers.

### 7.9.4 Transform triggers

These triggers are similar to conflict triggers, except they're executed for every row on the data stream against the specific table. The behavior of return values and the exposed variables is similar, but transform triggers execute before a target row is identified, so there's no `TARGET` row.

You can specify multiple transform triggers on each table in PGD. Transform triggers execute in alphabetical order.

A transform trigger can filter away rows, and it can do additional operations as needed. It can alter the values of any column or set them to `NULL` . The return value decides the next action taken:

- If the trigger function returns a row, it's applied to the target.

- If the trigger function returns a `NULL` row, there's no further action to perform. Unexecuted triggers never execute.
- The trigger function can perform other actions as it chooses.

The trigger function has access to additional state information as well as rows involved in the conflict:

- On `INSERT` , transform triggers can access the `SOURCE_NEW` row from the source.
- On `UPDATE` , transform triggers can access the `SOURCE_OLD` and `SOURCE_NEW` row from the source.
- On `DELETE` , transform triggers can access the `SOURCE_OLD` row from the source.

You can use the function `bdr.trigger_get_row()` to retrieve `SOURCE_OLD` or `SOURCE_NEW` rows. `TARGET` row isn't available, since this type of trigger executes before such a target row is identified, if any.

Transform triggers look very similar to normal BEFORE row triggers but have these important differences:

- A transform trigger gets called for every incoming change. BEFORE triggers aren't called at all for `UPDATE` and `DELETE` changes if a matching row in a table isn't found.
- Transform triggers are called before partition-table routing occurs.
- Transform triggers have access to the lookup key via `SOURCE_OLD` , which isn't available to normal SQL triggers.

### 7.9.5 Row contents

The `SOURCE_NEW` , `SOURCE_OLD` , and `TARGET` contents depend on the operation, REPLICA IDENTITY setting of a table, and the contents of the target table.

The TARGET row is available only in conflict triggers. The TARGET row contains data only if a row was found when applying `UPDATE` or `DELETE` in the target table. If the row isn't found, the TARGET is `NULL` .

### 7.9.6 Execution order

Execution order for triggers:

- Transform triggers _ Execute once for each incoming row on the target.
- Normal triggers _ Execute once per row.
- Conflict triggers _ Execute once per row where a conflict exists.

### 7.9.7 Stream triggers examples

A conflict trigger that provides similar behavior as the `update_if_newer` conflict resolver:

```
CREATE OR REPLACE FUNCTION update_if_newer_trig_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF (bdr.trigger_get_committs(TARGET) >
        bdr.trigger_get_committs(SOURCE_NEW)) THEN
    RETURN TARGET;
    ELSIF
        RETURN SOURCE;
    END IF;
END;
$$;
```

A conflict trigger that applies a delta change on a counter column and uses SOURCE_NEW for all other columns:

```
CREATE OR REPLACE FUNCTION delta_count_trg_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    DELTA bigint;
    SOURCE_OLD record;
    SOURCE_NEW record;
    TARGET record;
BEGIN
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    TARGET := bdr.trigger_get_row(TARGET);

    DELTA := SOURCE_NEW.counter - SOURCE_OLD.counter;
    SOURCE_NEW.counter = TARGET.counter + DELTA;

    RETURN SOURCE_NEW;
END;
$$;
```

A transform trigger that logs all changes to a log table instead of applying them:

```
CREATE OR REPLACE FUNCTION log_change
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE_NEW record;
    SOURCE_OLD record;
    COMMITTS timestamptz;
BEGIN
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    COMMITTS := bdr.trigger_get_committs(SOURCE_NEW);

    IF (TG_OP = INSERT) THEN
        INSERT INTO log SELECT I, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = UPDATE) THEN
        INSERT INTO log SELECT U, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = DELETE) THEN
        INSERT INTO log SELECT D, COMMITTS, row_to_json(SOURCE_OLD);
    END IF;

    RETURN NULL; -- do not apply the change
END;
$$;
```

This example shows a conflict trigger that implements trusted-source conflict detection, also known as trusted site, preferred node, or Always Wins resolution. It uses the `bdr.trigger_get_origin_node_id()` function to provide a solution that works with three or more nodes.

```
CREATE OR REPLACE FUNCTION test_conflict_trigger()
```

(continues on next page)

(continued from previous page)

```
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE  record;
    TARGET  record;

    TRUSTED_NODE    bigint;
    SOURCE_NODE     bigint;
    TARGET_NODE     bigint;
BEGIN
    TARGET := bdr.trigger_get_row(TARGET);
    IF (TG_OP = DELETE)
        SOURCE := bdr.trigger_get_row(SOURCE_OLD);
    ELSE
        SOURCE := bdr.trigger_get_row(SOURCE_NEW);
    END IF;

    TRUSTED_NODE := current_setting(customer.trusted_node_id);

    SOURCE_NODE := bdr.trigger_get_origin_node_id(SOURCE_NEW);
    TARGET_NODE := bdr.trigger_get_origin_node_id(TARGET);

    IF (TRUSTED_NODE = SOURCE_NODE) THEN
        RETURN SOURCE;
    ELSIF (TRUSTED_NODE = TARGET_NODE) THEN
        RETURN TARGET;
    ELSE
        RETURN NULL; -- do not apply the change
    END IF;
END;
$$;
```

## 7.10 Transaction streaming

With logical replication, transactions are decoded concurrently on the publisher but aren't sent to subscribers until the transaction is committed. If the changes exceed `logical_decoding_work_mem` (PostgreSQL 13 and later), they're spilled to disk. This means that, particularly with large transactions, there's some delay before they reach subscribers and might entail additional I/O on the publisher.

Beginning with PostgreSQL 14, transactions can optionally be decoded and sent to subscribers before they're committed on the publisher. The subscribers save the incoming changes to a staging file (or set of files) and apply them when the transaction commits (or discard them if the transaction aborts). This makes it possible to apply transactions on subscribers as soon as the transaction commits.

### 7.10.1 PGD enhancements

PostgreSQL's built-in transaction streaming has the following limitations:

- While you no longer need to spill changes to disk on the publisher, you must write changes to disk on each subscriber.
- If the transaction aborts, the work (changes received by each subscriber and the associated storage I/O) is wasted.

However, PGD supports Parallel Apply, enabling multiple writer processes on each subscriber. This capability is leveraged to provide the following enhancements:

- Decoded transactions can be streamed directly to a writer on the subscriber.
- Decoded transactions don't need to be stored on disk on subscribers.
- You don't need to wait for the transaction to commit before starting to apply the transaction on the subscriber.

#### Caveats

- You must enable Parallel Apply.
- Workloads consisting of many small and conflicting transactions can lead to frequent deadlocks between writers.

> **Note**
>
> Direct streaming to writer is still an experimental feature. Use it with caution. Specifically, it might not work well with conflict resolutions since the commit timestamp of the streaming might not be available. (The transaction might not yet have committed on the origin.)

### 7.10.2 Configuration

Configure transaction streaming in two locations:

- At node level, using the GUC bdr.default_streaming_mode
- At group level, using the function bdr.alter_node_group_option()

#### Node configuration using bdr.default_streaming_mode

Permitted values are:

- `off`
- `writer`
- `file`
- `auto`

Default value is `auto` .

To make a change to this setting take effect, restart the pglogical receiver process for each subscription.

You can achieve this with a server restart.

If `bdr.default_streaming_mode` is set to any value other than `off` , the subscriber requests transaction streaming from the publisher. How this is provided can also depend on the group configuration setting. See *Node configuration using bdr.default_streaming_mode* for details.

#### Group configuration using bdr.alter_node_group_option()

You can use the parameter `streaming_mode` in the function bdr.alter_node_group_option()

to set the group transaction streaming configuration.

Permitted values are:

- `off`
- `writer`

- `file`
- `auto`
- `default`

The default value is `default` .

The value of the current setting is contained in the column `node_group_streaming_mode` from the view bdr.node_group . The value returned is a single char type, and the possible values are `D` (`default` ), `W` (`writer` ), `F` (`file` ), `A` (`auto` ), and `O` (`off` ).

### Configuration setting effects

Transaction streaming is controlled at the subscriber level by the GUC `bdr.default_streaming_mode` . Unless set to `off` , which disables transaction streaming, the subscriber requests transaction streaming.

If the publisher can provide transaction streaming, it streams transactions whenever the transaction size exceeds the threshold set in `logical_decoding_work_mem` . The publisher usually has no control over whether the transactions are streamed to a file or to a writer. Except for some situations (such as COPY), it might hint for the subscriber to stream the transaction to a writer (if possible).

The subscriber can stream transactions received from the publisher to either a writer or a file. The decision is based on several factors:

- If Parallel Apply is off (`num_writers = 1` ), then it's streamed to a file. (writer 0 is always reserved for non-streamed transactions.)
- If Parallel Apply is on but all writers are already busy handling streamed transactions, then the new transaction is streamed to a file. See *Monitoring PGD writers* to check PGD writer status.

If streaming to a writer is possible (that is, a free writer is available), then the decision whether to stream the transaction to a writer or a file is based on the combination of group and node settings as per the following table.

| Group | Node | Streamed to |
|---|---|---|
| off | (any) | (none) |
| (any) | off | (none) |
| writer | file | file |
| file | writer | file |
| default | writer | writer |
| default | file | file |
| default | auto | writer |
| auto | (any) | writer |

If the group configuration is set to `auto` , or the group configuration is `default` and the node configuration is `auto` , then the transaction is streamed to a writer only if the publisher hinted to do this.

Currently the publisher hints for the subscriber to stream to the writer for the following transaction types. These are known to be conflict free and can be safely handled by the writer.

- `COPY`
- `CREATE INDEX CONCURRENTLY`

### 7.10.3 Monitoring

You can monitor the use of transaction streaming using the bdr.stat_subscription

function on the subscriber node.

- `nstream_writer` _ Number of transactions streamed to a writer.
- `nstream_file` _ Number of transactions streamed to file.
- `nstream_commit` _ Number of committed streamed transactions.
- `nstream_abort` _ Number of aborted streamed transactions.
- `nstream_start` _ Number of streamed transactions that were started.
- `nstream_stop` _ Number of streamed transactions that were fully received.

## 7.11 Explicit two-phase commit (2PC)

!!!Note

Two-phase commit isn't available with Quorum Commit, Group Commit, or CAMO. See *Commit scope limitations* .

The X/Open Distributed Transaction Processing (DTP) model envisions three software components:

- An application program (AP) that defines transaction boundaries and specifies actions that constitute a transaction
- Resource managers (RMs), such as databases or file-access systems, that provide access to shared resources
- A separate component called a transaction manager (TM) that assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for failure recovery

PGD supports explicit external 2PC using the `PREPARE TRANSACTION` and `COMMIT PREPARED` /`ROLLBACK PREPARED` commands. Externally, an EDB Postgres Distributed cluster appears to be a single resource manager to the transaction manager for a single session.

When `bdr.commit_scope` is `local` , the transaction is prepared only on the local node. Once committed, changes are replicated, and PGD then applies post-commit conflict resolution.

Using `bdr.commit_scope` set to `local` might not seem to make sense with explicit two-phase commit. However, the option is offered to allow you to control the tradeoff between transaction latency and robustness.

Explicit two-phase commit doesn't work with Quorum Commit, Group Commit, or CAMO, as those commit scope kinds use two-phase commit internally. Explicit two-phase commit also doesn't work well with *Parallel Apply* or *Using with transaction streaming* . Disable both before using explicit two-phase commit.

### 7.11.1 Use

Two-phase commits with a local commit scope work exactly like standard PostgreSQL. Use the local commit scope:

```
BEGIN;
SET LOCAL bdr.commit_scope = local;

... other commands possible...
```

To start the first phase of the commit, the client must assign a global transaction id, which can be any unique string identifying the transaction:

```
PREPARE TRANSACTION some-global-id;
```

After a successful first phase, all nodes have applied the changes and are prepared for committing the transaction. The client must then invoke the second phase from the same node:

```
COMMIT PREPARED some-global-id;
```

CHAPTER 8

# Guidance for DBAs

## 8.1 Encrypting columns in PGD

Column encryption protects sensitive fields in a table by converting them into ciphertext, a scrambled and unreadable form, while leaving the rest of the record in its original, human-readable form. Because encryption happens at the SQL level during `INSERT` or `UPDATE` , only ciphertext ever reaches the Write-Ahead Log (WAL) and gets replicated across PGD nodes, keeping sensitive data protected at rest, in backups, and in transit.

In a distributed cluster, column encryption also provides the following benefits:

- **Node independence**: Applications can provide the key to any node for consistent decryption.
- **Reduced attack surface**: Keys never enter the WAL in plaintext. Only ciphertext is replicated.
- **Memory isolation**: Keys are cleared when the session closes.

PGD supports column-level encryption through the following:

- ** pgcrypto **: A community Postgres extension, not shipped as an EDB package, but supported when used with PostgreSQL, EDB Postgres Extended (PGE), and EDB Postgres Advanced Server (EPAS).
- ** DBMS_CRYPTO **: An Oracle-compatibility package available exclusively on EPAS.

Both are fully compatible with PGD replication. When using `pgcrypto` or `DBMS_CRYPTO` , data is transformed using cryptographic functions before being written to the heap. pgcrypto stores the resulting ciphertext as `bytea` , and DBMS_CRYPTO uses `RAW` , `BLOB` , or `CLOB` , all of which PGD can replicate natively.

### 8.1.1 Comparing encryption methods

Column encryption is one of several approaches to protecting data in PGD. Each operates at a different level of the stack and guards against different threats.

Transparent Data Encryption (TDE) is an optional feature of PGE and EPAS that encrypts data at the storage level, making database files on disk unreadable without the encryption key. TDE operates transparently alongside PGD replication, but it offers no protection against a DBA who has access to the running database, who can still query all data in plaintext.

Data redaction is a feature of EPAS that masks sensitive values in query results based on policy. The underlying data on disk is unchanged, so it must be paired with TDE or column encryption to provide meaningful protection.

Column encryption protects at the field level, which is more granular than either of the above. A DBA without the session-level decryption key sees only ciphertext, even in a live query. The trade-off is that the application must manage keys and handle encryption and decryption explicitly.

| Technique | Protection level | Key managed by | Protects against | Visible to DBAs |
|---|---|---|---|---|
| **TDE** | Block/storage-level | Database/KMIP | Physical disk theft or OS-level access | Yes |
| **Data redaction** | Presentation-level | Database policy | Unauthorized app users | Yes |
| **Column encryption** | Field-level | Application | Compromised DBAs or system admins | No (ciphertext only) |

### 8.1.2 Planning for encryption in a distributed cluster

Encrypting columns in PGD introduces considerations that don't arise in a single-node setup. Before implementing column encryption, make sure you've accounted for indexing, CPU overhead, and the effect on storage and replication.

#### Handling indexing and search on encrypted columns

Both pgcrypto and DBMS_CRYPTO use probabilistic (randomized) encryption by default, meaning encrypting the same plaintext multiple times produces a different ciphertext each time. pgcrypto achieves this behavior via `pgp_sym_encrypt()`, and DBMS_CRYPTO does this via AES in CBC mode.

Randomized encryption has two key implications:

- **Index incompatibility**: A standard B-tree index on a ciphertext column can't support plaintext equality lookups.
- **Query failure**: The following common pattern returns zero results even if the record exists, because each call to `pgp_sym_encrypt()` produces different ciphertext that will never match what's stored in the column. No error is raised, making the failure easy to mistake for the record not existing:

```
SELECT * FROM customers
WHERE email = pgp_sym_encrypt(alice@example.com, your_key);
```

With pgcrypto specifically, if the column type is `text` rather than `bytea`, Postgres raises a type mismatch error instead:

```
ERROR: operator does not exist: text = bytea
```

To search encrypted columns by exact value, use one of these strategies:

- **Deterministic encryption**: Configure the encryption algorithm to always produce the same ciphertext for a given plaintext.
- **Blind index (recommended)**: Store an additional column containing a one-way cryptographic hash of the plaintext, using a function like `hmac()` from pgcrypto, and index that column instead. Queries hash the search value and compare against the index rather than the encrypted column.

For implementation details on both approaches, see the pgcrypto documentation or the *DBMS_CRYPTO reference*.

> **Warning**
>
> Both approaches reveal equality patterns: an attacker who can observe the stored values can infer when two rows contain the same data, and on low-variety fields like status flags or country codes, can deduce the plaintext entirely.

### Accounting for CPU overhead

Encryption and decryption are computationally expensive operations that run on the database server. Ensure your nodes have sufficient CPU headroom. You can significantly reduce this impact by using algorithms that leverage hardware acceleration, such as AES on processors that support the AES-NI instruction set.

### Planning for storage and WAL growth

Ciphertext is larger than the original plaintext, which increases table size, WAL volume, and the replication bandwidth required for PGD to synchronize nodes. Plan capacity accordingly.

## 8.1.3 Encrypting columns using pgcrypto

pgcrypto is a community Postgres extension that provides cryptographic functions including `pgp_sym_encrypt()` and `pgp_sym_decrypt()` . It works with PostgreSQL, PGE, and EPAS. Views handle transparent decryption and triggers handle automatic encryption on write, so applications can read and write plaintext without calling the cryptographic functions directly.

1. Create the extension:

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;
```

1. Create the `customers` table with the encrypted column defined as `bytea` :

```
CREATE TABLE customers (
    customer_id   uuid    DEFAULT uuidv7() PRIMARY KEY,
    username      TEXT    NOT NULL,
    email         BYTEA   NOT NULL
);
```

1. Create a view for transparent decryption:

```
-- Note: requires SET app.pgc_key = your_key to be run in the session first
CREATE OR REPLACE VIEW customers_view AS
SELECT
    customer_id,
    username,
    pgp_sym_decrypt(email, current_setting(app.pgc_key)) AS email
FROM customers;
```

1. Create a trigger to encrypt data automatically before it's written:

```
CREATE OR REPLACE FUNCTION encrypt_customer_pii_trigger()
RETURNS TRIGGER AS $$
BEGIN
    IF current_setting(app.pgc_key, true) IS NULL OR current_setting(app.pgc_key) = ␣
↪THEN
        RAISE EXCEPTION Encryption key not set. Please run SET app.pgc_key = ...;
    END IF;

    IF (TG_OP = INSERT OR NEW.email != OLD.email) THEN
        NEW.email := pgp_sym_encrypt(
            convert_from(NEW.email, utf8),
            current_setting(app.pgc_key),
            cipher-algo=aes256
```

(continues on next page)

(continued from previous page)

```
        );
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_encrypt_customer_pii
BEFORE INSERT OR UPDATE ON customers
FOR EACH ROW EXECUTE FUNCTION encrypt_customer_pii_trigger();
```

1. With the view and trigger in place, set the session key and interact with the table normally:

```
-- 1. Initialize session key
SET app.pgc_key = your_secret_key;

-- 2. Trigger handles encryption automatically
INSERT INTO customers (username, email) VALUES (dolores, dolores@example.com);
```

Querying the raw `customers` table reveals that the `email` column stores ciphertext, while `customers_view` returns the decrypted plaintext:

```
SELECT customer_id, username, email FROM customers;
__OUTPUT__
 customer_id                          | username | email
- -------------------------------------+----------+-------------------------------------
↪------------
 01966e42-1af7-7a2a-b4b9-9cf8e0d3c2a | dolores  | \xc30d04070302e4a7f23b8c12d956aa...
(1 row)
```

```
SELECT customer_id, username, email FROM customers_view;
__OUTPUT__
 customer_id                          | username | email
- -------------------------------------+----------+---------------------
 01966e42-1af7-7a2a-b4b9-9cf8e0d3c2a | dolores  | dolores@example.com
(1 row)
```

### 8.1.4 Encrypting columns using DBMS_CRYPTO

DBMS_CRYPTO is an Oracle-compatible package available exclusively in EPAS. It provides encryption and decryption functions that mirror the Oracle `DBMS_CRYPTO` API. As with pgcrypto, a view handles transparent decryption and a trigger handles automatic encryption on write.

1. Create the `customers` table with the encrypted column defined as `RAW` :

```
CREATE TABLE customers (
    customer_id   uuid     DEFAULT uuidv7() PRIMARY KEY,
    username      TEXT     NOT NULL,
    email         RAW(128)
);
```

1. Create a view for transparent decryption:

```
CREATE OR REPLACE VIEW customers_view AS
SELECT
    customer_id,
    username,
    UTL_I18N.RAW_TO_CHAR(
        DBMS_CRYPTO.DECRYPT(
            src => email,
            typ => 258 + 256 + 4096, -- AES256 + CBC + PKCS7
            key => UTL_I18N.STRING_TO_RAW(current_setting(app.pgc_key), AL32UTF8)
        ),
        AL32UTF8
    ) AS email
FROM customers;
```

1. Create a trigger to encrypt data automatically before it's written:

```
CREATE OR REPLACE FUNCTION epas_encrypt_customer_pii_trigger()
RETURNS TRIGGER AS $$
DECLARE
    l_typ INTEGER := 258 + 256 + 4096; -- AES256 + CBC + PKCS7
    l_key RAW(32) := UTL_I18N.STRING_TO_RAW(current_setting(app.pgc_key), AL32UTF8);
BEGIN
    IF (TG_OP = INSERT OR NEW.email IS DISTINCT FROM OLD.email) THEN
        NEW.email := DBMS_CRYPTO.ENCRYPT(
            src => UTL_I18N.STRING_TO_RAW(convert_from(NEW.email, utf8), AL32UTF8),
            typ => l_typ,
            key => l_key
        );
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_epas_encrypt_customer_pii
BEFORE INSERT OR UPDATE ON customers
FOR EACH ROW EXECUTE FUNCTION epas_encrypt_customer_pii_trigger();
```

1. With the view and trigger in place, set the session key and interact with the table normally:

```
-- 1. Initialize session key
SET app.pgc_key = my_secure_32_byte_key_1234;

-- 2. Trigger handles encryption automatically
INSERT INTO customers (username, email) VALUES (dolores, dolores@example.com);
```

Querying the raw `customers` table reveals that the `email` column stores ciphertext as a `RAW` hex string, while `customers_view` returns the decrypted plaintext:

```
SELECT customer_id, username, email FROM customers;
__OUTPUT__
 customer_id                          | username | email
- ------------------------------------+----------+-----------------------------------
 ↪------------
```

(continues on next page)

(continued from previous page)

```
 01966e42-1af7-7a2a-b4b9-9cf8e0d3c2a | dolores   | A3F2C8D14E7B0591...
(1 row)
```

```
SELECT customer_id, username, email FROM customers_view;
__OUTPUT__
 customer_id                          | username | email
- -------------------------------------+----------+--------------------
 01966e42-1af7-7a2a-b4b9-9cf8e0d3c2a | dolores  | dolores@example.com
(1 row)
```

### 8.1.5 Managing encryption keys securely

PGD replicates data but not keys, which means all nodes must have access to the same key material. Follow these practices to manage keys safely:

- **Avoid hardcoding keys:** Never include keys in SQL scripts or `CREATE VIEW` definitions, as they're stored in `pg_catalog` and are visible to anyone with access to system catalogs.
- **Use a session-level configuration parameter to pass the key:** Setting it at the session level means the key exists only in memory and is cleared when the session closes. The parameter name can be anything in the `app.*` namespace. For example, `app.pgc_key` :

```
SET app.pgc_key = your_ultra_secure_aes_256_key_here;
```

**Important**

Configuration parameters are local to the specific node and session where the command was executed. Because PGD operates as a multi-node cluster, your application must ensure the *SET* command is issued to the specific node handling the current connection. If your application switches nodes (for example, via Connection Manager), the parameter must be re-set on the new node before attempting to read or write encrypted data. !!!

- **Use an external key management service for production deployments:** Services such as HashiCorp Vault or AWS KMS let the application fetch the key at runtime and provide it to PGD for the duration of the session, so keys are never persisted to local disk on any PGD node.

## 8.2 Using tiered storage

PGD supports tiered storage through integration with the Postgres Analytics Accelerator (PGAA) , a vectorized query engine and cloud-native storage extension for Postgres. PGAA manages data in object storage using columnar formats like Apache Iceberg® or Delta Lake, and exposes it as a queryable Postgres table, with no ETL pipelines required.

Together, PGD and PGAA manage the complete data lifecycle within a unified architecture, creating two distinct storage tiers:

- Hot tier (PGD): Local PostgreSQL storage for current operational data requiring sub-millisecond latency and full ACID compliance.
- Cold tier (PGAA): Cost-effective object storage using columnar formats like Apache Iceberg® or Delta Lake, optimized for high-speed analytical queries.

Together, they allow you to seamlessly transition data from high-performance transactional storage to a cost-effective data lake, all while maintaining transparent access through a unified Postgres interface. This architecture supports a variety of high-impact use cases:

- **Regulatory compliance:** Offload aged data and audit trails to maintain lean, high-performance systems while ensuring long-term retention.
- **Time-series & IoT:** Balance real-time dashboards using hot recent data with cost-effective cold historical analytics for sensor and event streams.
- **Financial services:** Manage massive volumes of required historical transaction records without inflating transactional storage costs.
- **Retail & seasonal business cycles**: Use adaptive policies to keep peak-season order data hot for customer service while tiering older records to cold storage.

### 8.2.1 Understanding table types

This architecture implements two distinct storage tiers: the hot tier (local SSD, sub-millisecond latency) and the cold tier (object storage, columnar format). Data exists in one of the following three distinct table types:

| Table type | Storage location | Behavior |
|---|---|---|
| Heap | Local disk only | Standard transactional table in the hot tier. |
| Hybrid Transactional and Analytics Processing (HTAP) | Local + object storage | A heap table mapped to an analytical copy. PGD replicates all changes in near real-time. |
| PGAA | Object storage only | A purely analytical cold table, available in the Delta or Iceberg format. |

### 8.2.2 Tiered storage strategies

There are three available methods to transition data between tiers based on your operational needs.

#### Implementing tiered tables

This method leverages **PGD AutoPartition** to automate the lifecycle of high-volume data. A heap table is transformed into a partitioned structure where:

- **Hot partitions:** Recent data remains as heap (or HTAP if `enable_replication` is enabled) for active processing.
- **Cold partition:** Once partitions cross an age threshold, PGD bulk-copies them to object storage, converts them to PGAA tables, and reclaims local disk space, maintaining the data queryable.

#### Replicating to analytics

This method continuously synchronizes a table with an analytical copy in object storage. A heap table is converted into an HTAP table. To the user, it appears as a standard transactional table, but an analytical copy is maintained in the background.

To ensure near real-time synchronization without taxing your transactional throughput, PGAA uses Iceberg Merge-on-Read (MoR). This records changes in small "delete files" rather than rewriting entire data blocks, keeping the data lake fresh with minimal overhead.

#### Offloading to analytics

This method offers a one-time, surgical way to immediately reclaim local disk space for HTAP tables, leaving only the copy in object storage (PGAA table). The local heap copy is truncated, and the table access method is altered to `PGAA`, which now points to object storage.

If you need to bring your data back for transactional writes, you can restore the table back from object storage into the local Postgres heap.

| Method | Purpose | Use case | Origin Table | Destination table |
|---|---|---|---|---|
| Tiered tables | Automated lifecycle management | High-volume time-series data where historical retention is critical and automated. | Heap | Heap or HTAP (hot) PGAA (cold) |
| Replication | Continuous synchronization | Replicating a transactional table in near real-time for general analytics. | Heap | HTAP |
| Offload | Selective archiving | One-time, surgical removal and archival of a specific table. | HTAP | PGAA |

### 8.2.3 Metadata management

PGAA supports two methods for managing object store metadata when integrating with PGD:

- **No-catalog offload**: Store metadata alongside data files using storage locations to connect directly to object storage.
- **Catalog-managed offload**: Decouple data from the metadata using a centralized catalog service like Iceberg REST.

#### Setting up PGAA

Tiered storage in PGD is powered by the Postgres Analytics Accelerator (PGAA) extension. PGAA provides the vectorized query engine and cloud-native storage handlers that allow PGD to offload data partitions to object storage and query them in place.

Before you can configure tiered tables, replicate to analytics, or offload data, you must install PGAA on every node in your cluster and define the object storage destination. Configuring the storage target at the node-group level ensures that every node writes to and reads from the same location.

#### Prerequisites

- **Cluster:** PGD version 6.1 or later with PGAA and PGFS extensions installed.
- **Storage locations:** Local, S3, GCP, or Azure storage using Iceberg or Delta Lake formats.
- **Catalog:** If using an external catalog service, only Iceberg REST catalogs are supported.
- **Permissions:** The database user must have `CREATE` , `ALTER` , and `EXECUTE` privileges for the PGD and PGAA functions.

> **Note**
>
> The credentials associated with a storage location or a catalog service must have both read and write permissions for the destination bucket. You can verify permissions by running the following functions, which return *NULL* if successful.

```
-   For a storage location:
```

```
SELECT pgaa.test_storage_location (my_storage_location, true);
```

```
-   For a catalog service:
```

```
SELECT pgaa.test_catalog(my_iceberg_catalog, test_writes := FALSE);
```

### Downloading and installing the PGAA extension

Install the PGAA extension on every node in your PGD cluster.

1. Download the package from the EDB repository:

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
export EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↪SUBSCRIPTION_PLAN/setup.deb.sh" | sudo -E bash
sudo apt-get install -y <postgresql-distribution>-pgaa
```

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
EDB_SUBSCRIPTION_PLAN=<your-subscription-plan>
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↪SUBSCRIPTION_PLAN/setup.rpm.sh" | sudo -E bash
sudo dnf install -y <postgresql-distribution>-pgaa
```

Where: - `$EDB_SUBSCRIPTION_TOKEN` is the token you received when you registered for the EDB subscription. - `$EDB_SUBSCRIPTION_PLAN` is either `standard` (for PGE or Community Postgres) or `enterprise` (for EPAS). - `<postgresql-distribution>` specifies your Postgres distribution (`edb-postgresextended`, `edb-as`, or `postgres`), and version.

1. Update your `postgresql.conf` file. Adding `pgaa` to `shared_preload_libraries` allows Postgres to manage the life cycle of the Seafowl query engine as a background worker.

```
shared_preload_libraries = pgaa
pgaa.autostart_seafowl = on
```

1. Restart your Postgres service to load the new library.

### Pointing to object storage

Define the destination for the cold data. Because you are working within a distributed PGD cluster, you must wrap these configuration calls in `bdr.replicate_ddl_command` to ensure the settings are propagated to all nodes in the group.

!!!Tip

To view your node group, use `SELECT node_group_id FROM bdr.node_group;` .

1. Create a storage location using the *PGFS extension* and set it as the analytics storage for your node group.

```
SELECT bdr.replicate_ddl_command($$
  SELECT pgfs.create_storage_location(
  my-storage-location,
  protocol://my-bucket-name/path/,
  {"region": "region-name"},
  {"access_key_id": "...", "secret_access_key": "..."}
  )
$$);
```

1. Clear `analytics_write_catalog`. Its value must be `NULL` before an `analytics_storage_location` can be assigned:

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_write_catalog, NULL);
```

1. Set the analytics storage location for your node group:

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_storage_location, my-
↪storage-location);
```

1. Configure a connection to an external catalog :

```
SELECT bdr.replicate_ddl_command($$
  SELECT pgaa.add_catalog(
    my-catalog-name,
    iceberg-rest,
    {
      "url": "https://my-catalog-rest-endpoint.com",
      "token": "MY_AUTH_TOKEN",
    }
  )
);
```

1. Clear `analytics_storage_location`. Its value must be `NULL` before an `analytics_write_catalog` can be assigned:

```
SELECT bdr.alter_node_group_option(my-pgd-group, analytics_storage_location, NULL);
```

1. Set the catalog as the default analytics write location:

```
SELECT bdr.replicate_ddl_command($$
  SELECT bdr.alter_node_group_option(my-pgd-group, analytics_write_catalog, my-
↪catalog-name)
);
```

### Next steps

Once you have defined the storage target for your PGD cluster, determine how your tables interact with that storage based on your requirements for performance and capacity:

- :ref:`Implementing tiered tables <Implementing tiered tables>` Establish a zero-touch data lifecycle by automatically transitioning partitions from local heap storage to your analytics storage targed based on an age threshold.
- :ref:`Replicating to analytics <Replicating to analytics>` Maintain a local transactional heap table while simultaneously synchronizing a columnar copy to your storage target for heavy analytical processing.
- :ref:`Offloading to analytics <Offloading to analytics>` Reclaim heap disk space from existing tables which are being replicated to analytics.

## Implementing tiered tables

Tiered tables provide a unified storage architecture by combining Postgres Distributed (PGD) AutoPartition with Postgres Analytics Accelerator (PGAA) to create an automated data lifecycle. When a standard heap table is converted to a tiered structure, the system manages the transition of data from transactional to analytical storage without manual intervention, providing zero-touch data lifecycle management that optimizes storage costs.

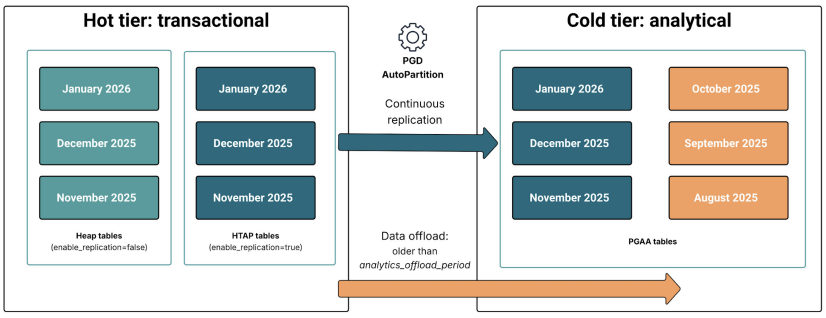


Fig. 1: tiered_tables

In this architecture, queries span both hot and cold data tiers seamlessly, providing a single point of access for the entire dataset regardless of its physical location. Recent data remains in local heap partitions for high-performance transactional processing, while older cold partitions are offloaded to object storage transparently once they cross a defined age threshold.

By calling `pgaa.convert_to_tiered_table()` , a standard heap table is transformed into a hybrid partitioned structure:

- **Hot tier:** Recent data partitions stay in local heap partitions, or become Hybrid Transactional and Analytics Processing (HTAP) tables if replication is enabled. For HTAP, PGAA uses Iceberg Merge-on-Read (MoR) to capture updates and deletes as lightweight "delete files." This keeps the cloud tier synchronized without the overhead of rewriting large data blocks.
- **Cold tier:** Once partitions cross the age threshold defined by `analytics_offload_period` , they are automatically converted to PGAA tables in the data lake. Local storage is reclaimed, but the data remains transparently queryable through the same table interface.

## Process overview

To convert a heap table into a tiered table, follow these steps:

1. *Setting up PGAA* by choosing either a storage location or a catalog service.
2. Use the `pgaa.convert_to_tiered_table()` function to *Converting a heap table to a tiered table* .
3. *Monitoring tiered tables* the automated offloading.
4. *Modifying and disabling tiered tables* settings as your storage needs evolve.

## Converting a heap table to a tiered table

The `pgaa.convert_to_tiered_table()` function initiates the automated offload process. If `enable_replication` is `true` , it also converts active partitions into HTAP tables for continuous replication. For example:

```
CALL pgaa.convert_to_tiered_table(
    relation := my_transactional_table::regclass,
```

(continues on next page)

(continued from previous page)

```
    range_partition_column := date,
    partition_increment := 1 month,
    analytics_offload_period := 1 year,
    initial_lower_bound := 2010-01-01,
    retention_period := 5 years,
    enable_replication := true,
    minimum_advance_partitions := 2,
    maximum_advance_partitions := 5,
    drop_after_retention_period := true,
    purge_analytics_target := true
);
```

Where:

- `relation` : The name of the heap table to convert to tiered table.
- `range_partition_column` : The column used for range partitioning. It must be `DATE` or `TIMESTAMP` , and it must also be included in the primary key.
- `partition_increment` : The time interval defining the size of each partition.
- `analytics_offload_period` : The threshold age at which a partition is moved from local storage to the cold tier in the data lake.
- `initial_lower_bound` : The starting value for the first partition's lower bound.
- `retention_period` : The period after which data is purged from the cold tier.
- `enable_replication` : If `true` , enables real-time replication to object storage for active partitions (as an HTAP table). If `false` , the hot partitions remain as heap tables.
- `minimum_advance_partitions` : Minimum number of future partitions to maintain.
- `maximum_advance_partitions` : Maximum number of future partitions to maintain.
- `drop_after_retention_period` : Whether to drop the partition locally after the retention period expires.
- `purge_analytics_target` : If `true` , any pre-existing tables in the object store directory or catalog entry are deleted. If set to `false` and pre-existing tables are detected, the command will fail with an error (Supported in PGD 6.3 and later).

**Note**

The column used for range partitioning must be a primary key.

Tables with sequences or foreign key constraints are not supported with tiered tables. You must drop the sequence or constraint before converting to a tiered table.

Tiered tables create a unified view that performs a `UNION` of all partitions, giving you a single query interface across the full data lifecycle. You can enable the `bdr.prefer_analytics_engine` parameter to route queries on the entire tiered table to the Seafowl engine:

```
SET bdr.prefer_analytics_engine = TRUE;
```

If `enable_replication` is `true` , this setting enables DirectScan for the entire partitioned table, allowing Seafowl to execute queries across both offloaded cold data and replicated hot data in a single pass.

> **Note**
>
> When using storage location-based replication, union views over partitioned tables are only supported by the Seafowl engine and are not accessible via Spark. When using a catalog service, union views are accessible via both Seafowl and Spark. !!!

### Monitoring tiered tables

The `pgaa.list_tiered_tables()` function provides a high-level overview of all tables currently undergoing automated lifecycle management. Its output is useful for identifying which heap tables have been successfully converted to tiered tables, what partitioning logic is being applied, and the current balance between between local storage and object storage (tiered and untiered data size).

```
SELECT * FROM pgaa.list_tiered_tables();
```

Query the `bdr.analytics_table` view to see the low-level mapping between your Postgres schema and the underlying analytical engine. This table tracks the relationship between your local node group and the remote analytics target.

```
SELECT * FROM bdr.analytics_table;
```

Use the following query to distinguish between hot and cold partitions:

```
SELECT
    relname AS partition_name,
    amname AS access_method
FROM pg_class c
JOIN pg_am am ON c.relam = am.oid
WHERE relname LIKE my-transactional-table%
  AND c.relkind = r
ORDER BY relname;
```

### Modifying and disabling tiered tables

You can modify their lifecycle policies or revert them to standard partitioned tables using a combination of PGD and PGAA commands.

**Adjusting lifecycle settings**

You can update the partitioning and offloading logic for an existing tiered table running the bdr.autopartition() function.

This is particularly useful if you need to shorten or extend the offload threshold. For example, to change `analytics_offload_period` from 1 month to 3 months, run:

```
SELECT bdr.replicate_ddl_command($$
  SELECT bdr.autopartition(
    relation := public.my-transactional-table,
    partition_increment := 1 month,
    partition_initial_lowerbound := 2020-01-01,
    analytics_offload_period := 3 months
  )
$$);
```

See AutoPartition in PGD for details.

!!!Warning

Modifying these settings only affects new partitions and does not reorganize existing data. If you increase the partition size, the system will not merge your current small partitions; it will only begin creating larger partitions once the current set is exhausted. While new offload timing applies to all partitions, the physical structure created under previous settings remains unchanged.

To change the current setting for `enable_replication` , use the `pgaa.enable_analytics_replication()` or `pgaa.disable_analytics_replication()` functions. When called on the parent table, these functions propagate the change to all existing and future child partitions:

```
CALL pgaa.enable_analytics_replication(my-transactional-table::regclass);
```

You can also enable or disable replication on individual partitions directly. However, you cannot disable replication on a child partition if it is already enabled on the parent table, and you cannot enable it on a child partition if the parent already has it enabled.

**Disabling tiered tables**

To revert to a standard partitioned heap table:

1. If `enable_replication` was active, stop replication on the parent table:

```
CALL pgaa.disable_analytics_replication(my-transactional-table::regclass);
```

1. Restore cold partitions. For each cold partition, run:

```
pgaa.restore_from_analytics(partition_name);
```

Or use `ALTER TABLE partition_name SET ACCESS METHOD heap;` .

1. Remove the tiered table flag from the parent table:

```
ALTER TABLE my-transactional-table SET (pgaa.tiered_table = false);
```

1. Stop AutoPartition (optional). To stop the table from creating new partitions entirely, run:

```
bdr.drop_autopartition(my-transactional-table);
```

1. Verify that the table is not listed as a tiered table anymore with `SELECT * FROM pgaa.list_tiered_tables()` .
2. Verify that all partitions list `heap` as their access method:

```
SELECT
    c.relname AS partition_name,
    a.amname AS access_method
FROM pg_class c
JOIN pg_am a ON c.relam = a.oid
WHERE c.relname LIKE my-transactional-table%
  AND c.relkind = r;
```

### Replicating to analytics

Use Postgres Analytics Accelerator (PGAA) within a Postgres Distributed (PGD) cluster to synchronize local heap tables with an analytical data lake by converting them to Hybrid Transactional and Analytics Processing (HTAP) tables.

In this state, every `INSERT` , `UPDATE` , and `DELETE` operation is captured and streamed in near real-time to object storage. While the table remains a standard transactional endpoint for applications, PGD maintains a high-fidelity analytical replica in the background, making data immediately available for BI tools and query engines without impacting production performance.

The key benefits of this process are:

- **Near real-time analytics:** Access the latest transactional data in your data lake without waiting for batch ETL processes.
- **Dual-state accessibility:** Maintain local high-performance storage for transactions while providing a columnar version for heavy analytical queries.
- **Unified management:** Manage your data lifecycle and replication settings directly through standard SQL and PGAA functions.
- **Efficiency with Merge-on-Read:** PGAA uses Iceberg Merge-on-Read (MoR) to synchronize data. Instead of rewriting large files every time a row changes, it simply records the change in a delete file. This keeps replication fast and significantly reduces the load on your object storage.

### Process overview

To enable continuous replication for a table, follow these steps:

1. *Setting up PGAA* by choosing either a storage location or a catalog service.
2. *Enabling replication* to start the data stream.
3. *Monitoring Replication Peers* status and health.
4. *Disabling replication* if you no longer require synchronization.

### Enabling replication

Use the `pgaa.enable_analytics_replication()` function to configure an existing heap table to stream data to object storage. The function converts the heap table to an analytical HTAP table and starts the logical replication worker.

```
CALL pgaa.enable_analytics_replication(my_table::regclass, purge_analytics_target :=␣
↪TRUE);
```

Set `purge_analytics_target:= TRUE` to purge existing data at the target location before replication begins (requires PGD 6.3 or later).

Alternatively, switch the replication state directly on the table object:

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = TRUE);
```

You can enable the `bdr.prefer_analytics_engine` parameter to route queries to the PGAA engine (Seafowl or Spark):

```
SET bdr.prefer_analytics_engine = TRUE;
```

If disabled, queries are executed by the standard Postgres engine against the local heap.

### Monitoring replication

Once replication is enabled, you can verify the HTAP status by checking that the table appears in both the local relation list and the analytics metadata.

View mappings between local tables and analytics targets:

```
SELECT * FROM bdr.analytics_table;
```

List all tables currently being managed by PGAA:

```
SELECT * FROM pgaa.list_analytics_tables();
```

### Disabling replication

Disabling replication stops the logical worker from capturing changes. The local table reverts to a standard heap table, and the remote copy in object storage becomes a static snapshot of the data.

Use the `pgaa.disable_analytics_replication()` function to stop the synchronization:

```
CALL pgaa.disable_analytics_replication(my_table::regclass);
```

Alternatively, use `ALTER TABLE` :

```
ALTER TABLE my_table SET (pgd.replicate_to_analytics = FALSE);
```

```
SET bdr.prefer_analytics_engine= FALSE;
```

**Note**

If you re-enable replication on a table where it was previously disabled, the system performs a full reload. This process effectively removes the existing analytics data in object storage, re-uploads the current local heap data in its entirety, and starts continuous synchronization. !!!

### Offloading to analytics

An Hybrid Transactional and Analytics Processing (HTAP) table uses Postgres Analytics Accelerator (PGAA) to continuously synchronize a Postgres Distributed (PGD) cluster with an analytical copy in object storage, enabling heavy analytics without impacting transaction performance at the cost of redundant storage in both the local heap and the data lake.

Offloading an HTAP table to analytics is a manual, surgical operation used to remove the local heap copy and reclaim disk space. When you offload a table, the system:

- Truncates the local heap table, immediately freeing up disk space on the PGD nodes.
- Stops the real-time replication to the data lake.
- Changes the table access method to PGAA, pointing the Postgres table definition directly to the existing data in object storage.

The result is a cold table that remains fully queryable through Postgres but consumes no local storage. If you require transactional writes again in the future, you can restore the table and stream the data back from the lake into the local heap.

### Process overview

To surgically offload a table to the data lake, follow these steps:

1. *Setting up PGAA* by choosing either a storage location or a catalog service.
2. *Offloading a table* .
3. *Restoring from analytics* back to heap if you require transactional access again.

### Offloading a table

Use the `pgaa.convert_to_analytics()` function to convert a synchronized HTAP table into a cold, analytical table and remove the data from local disk.

```
CALL pgaa.convert_to_analytics(my-table::regclass);
```

The function removes the local data and redefines the table's access method to PGAA to point solely to the external Iceberg or Delta files.

Alternatively, switch the access method with the `ALTER TABLE` command. Use the `pgaa.auto_truncate` option to ensure local disk space is freed immediately.

For storage locations:

```
ALTER TABLE public.my_table
  SET ACCESS METHOD pgaa,
  SET (
    pgaa.auto_truncate = true,
    pgaa.format = iceberg,
    pgaa.storage_location = my-storage-location,
    pgaa.path = public.my_table
  );
```

For a catalog-managed table:

```
ALTER TABLE public.my_catalog_managed_table
  SET ACCESS METHOD pgaa,
  SET (
      pgaa.auto_truncate = true,
      pgaa.format = iceberg,
      pgaa.managed_by = my_iceberg_catalog,
      pgaa.catalog_namespace = public,
      pgaa.catalog_table = my_remote_table_name
  );
```

### Restoring from analytics

If you need to resume transactional writes or bring the data back to local high-performance transactional storage, use the `pgaa.restore_from_analytics()` function:

```
CALL pgaa.restore_from_analytics(public.transactional_table);
```

The function checks the PGAA table's configuration, identifies its external Iceberg/Delta path in object storage, and rewrites the data back into the table's local storage structure. Once the data is locally present, the function sets the table's access method back to the local transactional endpoint again.

You can also use the `ALTER TABLE` command to restore a table from its cold state in the data lake back into a local transactional heap table. This manual method provides granular control over the transition back to local storage.

```
ALTER TABLE public.my_table
SET ACCESS METHOD heap;
```

**Warning**

Restoring a table to the heap access method does not remove the files from object storage. Manually delete the data via your object storage provider, or use a tool like PyIceberg to drop the table from the catalog.

## 8.3 Recovering a failed node

PGD automatically recovers failed nodes from brief outages, such as a restart or a transient network blip. The failed node reconnects, catches up with any changes it missed, and rejoins the cluster without intervention. You can confirm that the node transitions through `CATCHUP` back to `ACTIVE` by running `pgd nodes list` .

When automatic recovery doesn't succeed, you must manually recover the node. This typically happens when:

- The node's data directory is corrupted and Postgres won't start.
- The node has been offline long enough that its replication slot on the healthy nodes has been invalidated or is causing write-ahead log (WAL) to accumulate dangerously.
- The underlying hardware failed and the node needs to be rebuilt.
- The node is in a state (such as `JOINING` ) that it can't progress out of on its own.

Perform recovery by parting the failed node and bringing it back as a fresh member. This procedure applies to PGD clusters using any *Commit scopes* . It covers:

1. *Assessing the cluster state*
2. *Removing the failed nodes from the cluster*
3. *Recovering the failed nodes*
4. *Validating the recovered node*
5. *Verifying cluster health*

!!!Warning Don't leave failed nodes unattended for extended periods. Each healthy node holds WAL and replication data for offline peers, which can exhaust disk space and block catalog vacuuming. If a node is permanently lost or has been down too long, part it and rejoin a new one in its place. The replacement can reuse the original node's name.

Never manually remove replication slots created by PGD, as doing so can damage your cluster.

Before you begin, ensure you have:

- Access to the *Configuring PGD CLI* .
- Connection strings (DSN) for the healthy nodes.
- The *bdr_superuser* or equivalent to run `pgd` and SQL commands on the cluster.

### 8.3.1 Assessing the cluster state

Before taking any action, confirm the current state of all nodes and replication.

1. Check which nodes are `ACTIVE` and which are down or lagging:

```
pgd nodes list
```

Use the output to determine the type of failure. If fewer than half the nodes are down, you have a *Minority failure (quorum maintained)* and the cluster retains quorum. If half or more are down, you have a *Majority failure (quorum lost)* and the cluster has lost quorum. This determines which recovery approach to use in the next step.

1. Check for inactive or lagging replication slots:

```
pgd replication show --slots
```

Each healthy node holds a replication slot for offline peers, causing WAL to accumulate. A lagging slot indicates disk pressure; an invalidated slot means WAL has been discarded and the node may not be able to rejoin cleanly.

1. Identify the current write leader:

```
pgd group <group_name> show
```

### 8.3.2 Removing the failed nodes from the cluster

The approach differs depending on how many nodes have failed.

#### Minority failure (quorum maintained)

A minority failure is when fewer than half the nodes have failed. The remaining nodes still form a majority, so the cluster retains quorum, which means it can still reach agreement on writes and continues operating normally.

Use the PGD CLI to part each failed node — this is the recommended method for minority failure. Replace `<healthy_node_dsn>` with the connection string of a healthy node and `<failed_node_name>` with the name of the node to remove. If multiple nodes have failed, repeat the command for each one.

```
pgd --dsn "<healthy_node_dsn>" node <failed_node_name> part
```

For example:

```
pgd --dsn "host=node2.example.com dbname=bdrdb user=enterprisedb" node node1 part
```

Alternatively, use the `bdr.part_node` function from a healthy node, replacing `<failed_node_name>` with the name of the node to remove. If multiple nodes have failed, repeat the command for each one.

```
SELECT bdr.part_node(<failed_node_name>,
                     wait_for_completion => true,
                     force => false,
                     concurrent => true);
```

The parting operation initiates and the node is automatically removed once it's safe. The node transitions through the following states:

`ACTIVE` → `PART_START` → `PARTING` → `PART_CATCHUP` → `PART_TX_RESOLVE` → `PART_CLEANUP` → `PARTED` → Automatically dropped.

During the `PART_CATCHUP` phase, the remaining nodes synchronize any missing data from the parted node before the operation completes. This ensures no data loss and keeps the cluster consistent.

The automatic drop happens once all remaining nodes have consumed all changes from the parted node. You don't need to perform any manual catalog cleanup.

#### Majority failure (quorum lost)

A majority failure is when half or more of the nodes have failed. The remaining nodes can no longer form a majority, so the cluster loses quorum, which means it can no longer reach agreement on writes and write operations are blocked until the failed nodes are removed.

When a majority of nodes have failed, you can't use `pgd node part` because it requires cluster consensus. Instead, use `bdr.drop_node()` with `force => true` , wrapped in a transaction with a local commit scope. Run this on every remaining healthy node:

```
BEGIN;
SET LOCAL bdr.commit_scope = local;
SELECT bdr.drop_node(<failed_node_1>, force => true);
SELECT bdr.drop_node(<failed_node_2>, force => true);
COMMIT;
```

Unlike the standard parting process, forced drops are immediate, which means the nodes are removed without waiting for consensus.

!!!Note There is no PGD CLI equivalent for forced drop; you must use SQL. `bdr.part_node(force => true)` is deprecated in PGD 6 and is an alias for `bdr.drop_node(force => true)` .

#### Verifying the nodes have been removed from the cluster

Replace `<healthy_node_dsn>` with the connection string of a healthy node:

```
pgd --dsn "<healthy_node_dsn>" nodes list
```

For minority failure, wait for the automatic cleanup to complete before confirming the nodes are gone. For majority failure, which uses forced drop, the nodes are removed immediately.

### 8.3.3 Recovering the failed nodes

Repeat the following process for each failed node that needs to rejoin the cluster. A recovered node can reuse its original name as PGD allows parted or dropped node names to be reused.

#### Stopping the database and preparing the data directory

On the failed node:

1. Stop the Postgres service and verify it's stopped:

```
sudo systemctl stop postgresql.service
sudo systemctl status postgresql.service
```

1. Back up or move the existing data directory before clearing it:

```
cd /path/to/postgres/
mv data data_old
```

1. Clean the data directory, including any tablespace directories:

```
rm -rf <pgdata>/*
rm -rf /path/to/tablespace/*
```

#### Running pgd node setup

```
pgd node <new_node_name> setup \
  --pgdata <pgdata> \
  --dsn "host=localhost port=<port> dbname=<dbname> user=<username>" \
  --cluster-dsn "host=<healthy_node_host> port=5432 dbname=<dbname> user=<username>" \
  --log-file <log_file> \
  --group-name <group_name> \
  --initial-node-count <initial_node_count>
```

Where:

- `<new_node_name>` : Name of the recovered node. PGD allows reusing the original node name if the node is in a `PARTED` state.
- `--pgdata` : Data directory for the recovered node.
- `--dsn` : Connection string for the node being recovered.
- `--cluster-dsn` : Connection string to a healthy node in the cluster.
- `--log-file` : Path to the Postgres log file.
- `--group-name` : Node group name. Mandatory for the first node of a group. If not provided, the node joins the group of the active node.
- `--initial-node-count` : Number of nodes in the cluster (or planned). Used to calculate resource settings.

The command automatically initializes the data directory, joins the cluster, and starts the Postgres server. Look for `Node <new_node_name> added to the cluster successfully` in the output to confirm completion.

### 8.3.4 Validating the recovered node

Confirm the recovered node has fully rejoined the cluster. Verify the node's join state is `ACTIVE` :

```
pgd --output-format psql --dsn "<recovered_node_dsn>" node <new_node_name> show
```

Check that Raft consensus is established. All nodes must show `RAFT_LEADER` or `RAFT_FOLLOWER` :

```
pgd --output-format psql --dsn "<recovered_node_dsn>" raft show
```

Confirm all expected replication slots are present and active:

```
pgd --output-format psql --dsn "<recovered_node_dsn>" replication show --slots
```

Verify cluster visibility from both the recovered node and other healthy nodes. All nodes must appear as `ACTIVE` or `CATCHUP` :

```
pgd --output-format psql --dsn "<node_dsn>" nodes list
```

### 8.3.5 Verifying cluster health

While the node catches up with the cluster, monitor replication lag:

```
pgd --output-format psql --dsn "<node_dsn>" replication show --slots
```

If routing is enabled, verify routing information is available and correct on the recovered node:

```
pgd --output-format psql --dsn "<node_dsn>" node <new_node_name> show
```

Commit scope behavior resumes automatically once recovery is complete, and no manual reconfiguration is needed.

While the node is catching up, keep an eye on replication lag, disk space, Postgres logs, and network connectivity between nodes.

### 8.3.6 Troubleshooting

The following are common issues you may encounter during node recovery and how to resolve them.

**Recovery fails with a timeout**

Check network connectivity between nodes and ensure the source node is responding and healthy.

**Recovered node shows as ``JOINING`` instead of ``ACTIVE``**

Check the Postgres logs on the recovered node for errors, particularly any issues with the restore phase. Verify that replication slots exist on the other nodes by running `pgd replication show --slots` from a healthy node.

**Replication lag increases after recovery**

Check for resource constraints (CPU, disk I/O, memory), verify network bandwidth between nodes, and confirm that the Postgres configuration allows sufficient workers and replication slots (see *Postgres configuration parameters* ).

### 8.3.7 Post-recovery checklist

Use the following checklist to confirm the cluster is fully healthy before returning to normal operations.

- ☐ All recovered nodes show `join_state` as `ACTIVE` .
- ☐ Raft consensus is established — all nodes show `RAFT_LEADER` or `RAFT_FOLLOWER` .
- ☐ Replication slots are active on all nodes.
- ☐ All nodes see each other in `pgd nodes list` .
- ☐ No errors in Postgres logs on any node.
- ☐ Replication lag is within acceptable limits.

## 8.4 Developing applications for geo-replication

Learn what to take into consideration when your application runs against a geo-distributed PGD cluster. PGD handles replication and failover transparently, but geo-replication introduces behaviors at the application level that can require changes to how you connect, write, and retry.

### 8.4.1 When geo-replication affects your application

Identify the scenarios where geo-replication requires changes to your application:

- **Your application reads immediately after writing**: replication is asynchronous, so a read from a different node or location might not yet reflect a recent commit.
- **You can't afford to lose a write**: the default commit is asynchronous. If the node fails before the write replicates, the write is lost.
- **A failed commit can't safely be retried**: this risk exists on any database, but a geo-distributed cluster sees more in-flight failures during write leader elections. Because writes replicate across nodes, an interrupted commit may have already taken effect on some nodes, making blind retry potentially data-corrupting.

- **Multiple locations write to the same rows**: concurrent writes to the same row from different locations create conflicts. PGD resolves them automatically, but persistent conflict rates signal an access pattern that needs redesign.
- **You run DDL from application code**: schema changes acquire cluster-wide locks. DDL that completes in milliseconds on a local cluster can take seconds on a multi-region cluster and can block other transactions while it runs.

### 8.4.2 Make your application geo-replication ready

The same considerations apply whether you're building a new application or adapting an existing one.

#### Connect through Connection Manager

Use Connection Manager endpoints rather than connecting directly to individual nodes. Connection Manager routes write connections to the current write leader and handles failover automatically. Configure your connection string with multiple hosts for resilience.

See *Connecting through Connection Manager* .

#### Choose commit scopes per operation

The DBA defines the available commit scopes and sets a default at the group or subgroup level. Your application can override that default per session or per transaction. Use the group default for most writes, override to majority protect where durability matters, and use CAMO for non-idempotent operations such as payments and inventory decrements.

See *Choosing commit scopes* .

#### Design your schema to minimize conflicts

Conflicts occur when multiple locations write to the same data concurrently. Schema and access pattern choices at design time have more impact on conflict frequency than any configuration change after the fact.

See *Preventing conflicts* .

#### Configure selective replication

Use replication sets to control which tables replicate to which nodes, keeping sensitive or location-specific data within its region.

See *Configuring selective replication* .

#### Handle retries correctly

Idempotent operations, those that produce the same result whether applied once or many times, can retry freely on transient errors. Non-idempotent operations, such as payments and inventory decrements where a duplicate application causes incorrect results, need the CAMO workflow to determine transaction fate before retrying. Use exponential backoff and a circuit breaker to avoid overwhelming the cluster during failover.

See *Handling retries and failures* .

#### Run DDL safely

Schema changes in a geo-distributed cluster acquire locks across write leaders, which makes them slower and more disruptive than in a single-location setup. Avoid mixing DDL with DML and keep DDL transactions small.

See *Running DDL safely* .

### Connecting through Connection Manager

In a geo-distributed PGD cluster, *Connection Manager Authentication* handles routing automatically, directing write connections to the current write leader and read connections to a replica. Design your application to connect to Connection Manager's endpoints rather than directly to individual nodes, and configure your connection string to list multiple hosts for resilience.

### Splitting reads and writes

Connection Manager exposes two ports on each node. Direct traffic to the appropriate port based on your workload:

- **Read-write port**: Routes the connection to the current write leader of the node group. Use this port for all write operations and for reads that require up-to-date data.

```
host=node1.<location>,node2.<location>,node3.<location> port=6432 dbname=<dbname>␣
↪user=<username>
```

- **Read-only port**: Routes the connection to a replica. Use this port for reporting queries and reads that can tolerate slight replication lag, to keep read traffic off the write leader.

```
host=node1.<location>,node2.<location>,node3.<location> port=6433 dbname=<dbname>␣
↪user=<username>
```

Your DBA configures port numbers and they're consistent across all nodes in the cluster. Ask your DBA for the correct values.

List multiple nodes in both connection strings so that if one node is unavailable, the client driver tries the next one automatically.

Subscriber-only nodes never accept writes, so always connect to them through the read-only port.

### Configuring your connection strings

For geo-distributed deployments, configure your connection strings with the following in mind:

- **Connect to a local node.** Your application runs faster if you connect to a Connection Manager instance in the same location as your application servers, rather than routing to a remote node. Connection Manager then handles routing to the write leader if needed.
- **Set appropriate timeouts.** Cross-region network latency is higher than within a single data center. Use the standard Postgres `connect_timeout` parameter in your connection string to account for this. 5-10 seconds is typical for geo-distributed clusters.

### Handling failover

When the write leader fails, Connection Manager detects the failure and routes connections to the new write leader after election completes. In-flight connections drop during the election, so your application needs to account for the following:

- **Expect connection drops:** Any open transaction that was in-flight during failover is rolled back. Your application must catch connection errors and retry.
- **Don't assume transaction outcome:** Connection drops during `COMMIT` can happen on any database, but write leader elections make them more frequent in a geo-distributed cluster. If replication was in progress when the connection dropped, the transaction may have already committed on some nodes. For standard transactions, retry with idempotent operations. For non-idempotent critical transactions, use *Working with the CAMO partner* to determine the exact outcome.

- **Implement retry logic with backoff:** Retrying immediately after a connection failure risks hitting a node still in the process of failover. See *Handling retries and failures* for a retry strategy.
- **Plan for location-level failover:** Location-level failover is often handled at the infrastructure level through DNS or load balancer configuration. For applications that manage connections directly, we recommend including remote nodes as a secondary fallback in your connection string.

### Choosing commit scopes

Your DBA *Commit scopes* and sets a default at the group or subgroup level. Override the group default for operations that need stronger or weaker durability guarantees, setting the scope per session or per transaction. Choosing the right scope means understanding what each one costs in latency and what happens if the node fails before replication completes.

To see the default commit scope your DBA has configured for each node group:

```
SELECT node_group_name, default_commit_scope
FROM bdr.node_group_summary;
```

### Picking the right scope

Each scope makes a different promise about what happens after a write commits. The right choice depends on what your application can tolerate if a node fails before replication completes.

### Local protect

Use local protect for data where losing a recent write is acceptable, such as high-throughput event streams, analytics ingestion, or any workload where write latency matters more than durability. Transactions commit locally and replicate to other nodes asynchronously, so a node failure before replication completes means the write is lost. Reads on a different node can return slightly stale data while replication catches up.

### Majority protect

Use majority protect for writes that can't be lost but where added latency is acceptable, such as account updates, customer records, and most transactional data. The commit waits for a majority of nodes in the origin group to confirm before returning, so the write is durable across multiple nodes before the application proceeds. The cost is added round-trip latency per write, typically tens to hundreds of milliseconds for a multi-region cluster.

### Adaptive protect

Use adaptive protect for two-node-plus-witness groups where strict majority protect fails if one data node becomes unreachable. Adaptive protect behaves like majority protect when a majority of nodes are available, but degrades gracefully to asynchronous replication after a configurable timeout during transient network issues, at the cost of temporarily weaker durability.

### CAMO

Use CAMO for payments, fund transfers, and any operation where applying the same transaction twice causes incorrect results. The commit waits for a dedicated partner node and guarantees the transaction is applied exactly once. Even if the connection drops during commit, the application can query the transaction's fate and decide whether to retry. CAMO requires application code changes beyond just setting the scope. See *CAMO retry workflow* for the retry pattern.

CAMO requires exactly two data nodes in the CAMO group and adds higher commit latency than majority protect. It's not appropriate for high-throughput workloads where that latency is unacceptable.

### Quorum Commit

Use Quorum Commit for payments, core banking, and ledger systems where distributed consistency is required and conflicts can't be resolved after the fact. Unlike majority protect, which confirms durability, Quorum Commit coordinates the commit decision across all participating nodes before any node commits locally. If two transactions conflict, one is aborted during the commit phase before either commits.

Quorum Commit uses eager conflict resolution, so the application encounters serialization errors at commit time more often than with other scopes. For setup, configuration, and application patterns including retry loop examples, see *Using Quorum Commit* .

### Setting the scope

Use the group default for most operations, and override to a stronger scope only for writes that require it.

Set the scope per session when all operations on a connection share the same durability requirement, for example a dedicated connection used exclusively for payment processing. Set it per transaction when a single connection handles mixed workloads and only specific operations need a stronger guarantee.

- **Per session:** Set `bdr.commit_scope` at the session level to apply a scope to all transactions in that connection:

```
SET bdr.commit_scope = majority protect;
```

All subsequent transactions in the session use this scope until you change it or the session ends.

- **Per transaction:** Use `SET LOCAL` inside a transaction to override the scope for that transaction only. The session-level setting is restored when the transaction ends:

```
BEGIN;
SET LOCAL bdr.commit_scope = majority protect;
-- critical write operations
COMMIT;
```

!!!Important Quorum Commit doesn't guarantee consistency when transactions using it execute concurrently on the same tables as transactions that don't. If the node group default is a Quorum Commit scope, avoid overriding it to a weaker scope for transactions that touch the same tables.

To check which commit scope is active in the current session:

```
SHOW bdr.commit_scope;
```

### Other considerations

### Bulk operations

Avoid using a synchronous scope for high-throughput bulk operations. The added round-trip latency per transaction multiplies across large batches and can make bulk loads significantly slower. Split bulk operations by scope: use a strong scope for critical records and the default for the rest.

### Timed-out commits

If a commit scope requires remote confirmation and the remote location is unreachable, the transaction fails. Retry logic handles transient network failures, but timed-out commits require extra care: if a commit times out, the outcome is unknown. The transaction could have committed on some nodes before the timeout. For operations where duplicate application is not acceptable, use CAMO. See *CAMO retry workflow* .

### Reading your own writes

If your application writes on one connection and immediately reads on another, the read can land on a node that hasn't yet received the replication. To guarantee that a subsequent read reflects a prior write, use a synchronous commit scope on the write transaction. The commit doesn't return until the write has been replicated to the required nodes, so any subsequent read on those nodes sees the write:

```
BEGIN;
SET LOCAL bdr.commit_scope = majority protect;

- - Insert a new transaction for a known account
INSERT INTO transactions (transaction_id, account_id, direction, amount, balance_
↪after, description, channel, initiated_at)
VALUES (
    uuidv7(),
    <account_id>,
    D,
    250.00,
    <balance_after>,
    Online purchase,
    ONLINE,
    NOW()
)
RETURNING account_id, transaction_id, balance_after;

COMMIT;

- - This read now reflects the insert on any node in the majority
SELECT balance_after
FROM transactions
WHERE transaction_id = <returned-transaction-id>;
```

### Preventing conflicts

In a geo-distributed PGD cluster, all locations accept writes simultaneously. Conflicts arise when the same data is modified on different nodes before replication has a chance to propagate the change. Each node commits the change locally, then receives the conflicting change from the other node during replication.

PGD detects these collisions and resolves them automatically, but automatic resolution is a safety net, not a substitute for good design. Design your schema and access patterns to minimize how often conflicts occur.

### Understanding how PGD resolves conflicts

PGD resolves most conflicts automatically with no application involvement. The exception is conflicts with the `error` resolver set, which halt replication and require manual intervention. For all other conflict types, each has a deterministic default resolver. Knowing which resolver applies helps you identify when automatic resolution produces the expected outcome and when it doesn't. All conflicts are tracked in `bdr.conflict_history`:

- `update_update`: the same row was updated on two different nodes. Resolved by last-update-wins.
- `insert_insert`: the same primary key was inserted on two different nodes. Resolved by keeping the row with the later timestamp.
- `update_delete`: the row was updated on one node and deleted on another. By default, the delete wins.
- `delete_delete`: the row was deleted on both nodes simultaneously. Not a data concern, but logged.

For a full list of conflict types including more esoteric cases, see *bdr_read_all_conflicts* .

### Designing to minimize conflicts

Conflicts are a symptom of multiple locations writing to the same data concurrently. The most effective way to reduce them is to design your data model so that each piece of data has a clear owning location.

- **Use cluster-wide unique keys:** Auto-increment sequences are node-local and generate the same values on different nodes, causing `insert_insert` conflicts. Use PGD's global sequence types (SnowflakeId or galloc) for primary keys that are unique across the cluster. UUIDs generated client-side are also conflict-safe, though less space-efficient. See *Sequences* for PGD sequence options.
- **Partition data by location:** If each user or tenant always writes to one location, conflicts can't occur for that data. Assign each user, tenant, or data partition a home location and route write traffic there consistently. Data with high write rates and strict consistency requirements, such as account balances and inventory levels, is a good candidate for location-based partitioning. Data that is mostly read and rarely changes, such as product catalogs and lookup tables, can be written anywhere without conflict risk.
- **Use append-only patterns where possible:** Tables that only ever receive inserts (audit logs, event streams, time-series data) can't have update or delete conflicts. Design these tables to be insert-only and use separate summary tables if aggregation is needed.
- **Avoid shared counters:** A counter incremented by multiple locations is a conflict waiting to happen. Instead, record increments as individual rows and aggregate on read, or use PGD's CRDT data types, which are designed to merge correctly under concurrent updates. See *Handling column-level conflicts using CRDT data types* .
- **Avoid long-running transactions:** Transactions that hold locks or read snapshots for extended periods increase the window in which a conflicting write can arrive from another location. Keep transactions short.

### Monitoring for conflicts

All conflict types are tracked in `bdr.conflict_history` . Check `bdr.conflict_history_summary` regularly for tables with high conflict rates:

```
SELECT nspname, relname, conflict_type, count(*)
FROM bdr.conflict_history_summary
GROUP BY nspname, relname, conflict_type
ORDER BY count(*) DESC;
```

A persistent high conflict rate on a specific table points to an access pattern worth revisiting. Review the design patterns above or, if changing the access pattern isn't feasible, consider a custom conflict resolver. To inspect the actual row values involved in recent conflicts:

```
SELECT local_time, nspname, relname, conflict_type, conflict_resolution,
       key_tuple, local_tuple, remote_tuple, apply_tuple
FROM bdr.conflict_history
ORDER BY local_time DESC
LIMIT 20;
```

For tables where last-update-wins produces the wrong outcome and the access pattern can't be changed, ask your DBA to configure a custom conflict resolver. See *Configuring conflict resolvers* .

### Configuring selective replication

Use *Replication sets* to control which tables replicate to which nodes, keeping sensitive or location-specific data within its region.

Each node group has a pre-configured replication set with the same name. By default, all tables belong to the top-level group's replication set and replicate to every node in the cluster.

To see the group hierarchy and their associated replication sets:

```
SELECT node_group_name, default_repset, parent_group_name
FROM bdr.node_group_summary;
```

To see which replication sets your tables currently belong to:

```
SELECT relname, set_name FROM bdr.tables ORDER BY relname, set_name;
```

To restrict a table to specific locations, add it to those locations' replication sets and remove it from the top-level group's replication set:

```
SELECT bdr.replication_set_add_table(<table_name>, <location_a>);
SELECT bdr.replication_set_add_table(<table_name>, <location_b>);
SELECT bdr.replication_set_remove_table(<table_name>, <top_level_group>);
```

After this change, writes to the table on nodes in `<location_a>` replicate only to other nodes in `<location_a>` , and writes on `<location_b>` nodes replicate only within `<location_b>` .

Tables that remain in the top-level replication set continue to replicate everywhere. Use the top-level replication set for global reference data, such as product catalogs and configuration, that all locations need.

### Handling retries and failures

Connections to a geo-distributed cluster can drop during write leader elections, network partitions, or commit scope timeouts. Implement retry logic that handles these failures without applying the same operation twice, and use a circuit breaker to avoid overwhelming the cluster during recovery.

### Implementing retry logic

Any connection to a geo-distributed cluster can fail during a write leader election or network event. Wrap database operations in a retry loop with the following principles:

- **Retry on transient errors only:** Serialization failures ( `40001` ), deadlock errors ( `40P01` ), and connection errors are candidates for retry. Application-level errors or constraint violations aren't.
- **Use exponential backoff:** Retrying immediately after a failure can hit the cluster while it's still recovering. Start with a short delay, double it on each retry, and add a small random jitter to avoid thundering herd behavior.
- **Set a maximum retry count:** Don't retry indefinitely. After a reasonable number of attempts, fail and surface the error to the caller.
- **Make retried operations idempotent where possible:** Idempotent operations produce the same result whether applied once or many times. If the same transaction can be safely applied twice without incorrect results, retry without checking the prior outcome. If it can't, use *CAMO retry workflow* .

### CAMO retry workflow

CAMO is a commit scope configured by your DBA. If it isn't the group default, set it per session or per transaction as described in *Working with the CAMO partner* .

For non-idempotent operations such as payments, inventory decrements, and anything where applying the same transaction twice causes incorrect results, standard retry logic isn't safe. CAMO guarantees the transaction is applied at most once, giving the application a way to determine the definitive outcome of a disconnected transaction before deciding whether to retry.

Before issuing a CAMO transaction, confirm the CAMO partner is ready:

```
SELECT bdr.is_camo_partner_connected();
SELECT bdr.is_camo_partner_ready();
```

If the partner is not ready, CAMO can degrade to asynchronous mode, in which case the exactly-once guarantee isn't available.

For the full retry pattern and transaction status queries, see *Commit At Most Once* .

### Using a circuit breaker

A circuit breaker prevents your application from flooding a database that's in the middle of a failover with retry requests. Without one, a burst of retries during leader election can slow recovery and overwhelm the newly elected leader.

Implement a circuit breaker that tracks failure rate over a rolling window:

- **Closed state**: requests pass through normally.
- **Open state**: requests fail fast without hitting the database. Triggered when failures exceed a threshold within the window.
- **Half-open state**: after a recovery timeout, a single probe request goes through. If it succeeds, the circuit closes. If it fails, the circuit stays open.

The recovery timeout for a geo-distributed cluster's failover is typically 5-30 seconds depending on configuration. Set your circuit breaker's open duration to be at least that long.

### Running DDL safely

DDL operations in a geo-distributed cluster acquire locks across write leaders, held for the duration of the statement. Keep DDL transactions small, avoid mixing them with DML, and schedule them during low-traffic periods to minimize the impact on application throughput.

Lock acquisition time is proportional to round-trip latency across write leaders, so DDL that takes milliseconds on a local cluster can take seconds in a multi-region setup. Apply these practices to keep the impact minimal.

- **Make additive changes:** Adding a column, adding an index, or creating a new table are low-risk. Dropping columns, renaming columns, and changing data types require more care because they can break running application code on nodes that haven't received the schema change yet. In a rolling deployment, keep backward compatibility until all application versions that depend on the old schema are retired.
- **Add nullable columns first:** If a new column has a `NOT NULL` constraint, existing rows across the cluster need a default value or a backfill before the constraint can be enforced. Add the column as nullable first, backfill the data, then add the constraint in a subsequent migration.
- **Deploy schema changes before application code:** Roll out additive schema changes first, then deploy the application code that uses them. The schema is then in place across all nodes before any node starts sending queries that rely on it.

- **Keep DDL isolated:** Each DDL statement holds a cluster-wide DDL lock, preventing other concurrent DDL. Run DDL statements separately rather than combining them in a single transaction, and never mix DDL with DML. Combining schema changes with data manipulation extends the lock duration to cover the entire transaction.
- **Resolve long-running transactions before running DDL:** Long-running transactions block DDL from acquiring its cluster-wide lock. Check for and resolve any long-running queries or transactions before running DDL.
- **Have a rollback plan:** Not all DDL can be rolled back after it's committed. For destructive changes like `DROP TABLE` or `DROP COLUMN` , have a tested rollback procedure before running the change in production.

## 8.5 Setting up access controls

Encrypt and authenticate connections at every point in a PGD cluster to protect replication traffic, control application access, and prevent unauthorized nodes from joining. Access control operates at three levels:

- ** *Securing inter-node connections* ** are the connections PGD nodes make to each other for replication and Raft consensus. Securing these connections ensures that replication traffic is encrypted and that only authorized nodes can join the cluster.
- ** *Securing client application connections* ** covers how applications authenticate and connect to the cluster. Securing these connections controls how applications authenticate, which ports they connect to, and whether the HTTP management endpoint uses TLS.
- ** *Securing Connection Manager to node connections* ** covers how Connection Manager connects to backend nodes on behalf of clients. Each node must be configured to accept these connections.

### 8.5.1 Securing inter-node connections

PGD nodes connect directly to each other over the network to replicate changes. Without SSL, replication traffic is unencrypted and unauthenticated, leaving it exposed to interception and allowing unauthenticated peers to participate in replication. Configure SSL on each node to encrypt replication traffic and prevent unauthorized nodes from joining the cluster.

#### Setting up an authentication method

PGD supports two approaches for authenticating inter-node connections.

> **Note**
>
> Connection Manager reads the same *server.crt* configured for inter-node SSL. Using certificates that don't meet Connection Manager's requirements will prevent it from starting. See *Meeting SSL certificate requirements* for details.

As a best practice, use a dedicated database role for inter-node connections rather than the `postgres` superuser. Before joining nodes to the cluster, run the following on each node to create the role:

```
CREATE ROLE pgd_replication LOGIN IN ROLE bdr_superuser;
```

Use this role in the connection strings and `pg_hba.conf` rules shown in the following sections.

We recommend certificate authentication as it provides mutual authentication alongside encryption. If certificates aren't available in your environment, SCRAM-SHA-256 with Certificate Authority (CA) verification provides password-based authentication while still encrypting the connection.

### Authenticating with certificates

Certificate authentication is the recommended approach. Client certificates provide both encryption and mutual authentication. Each node presents a certificate to prove its identity, and all traffic is encrypted in transit.

### Provisioning certificates

Because each PGD node both accepts incoming replication connections and initiates outgoing ones, every node acts as both a server and a client. Each node needs server-side certificate files to accept connections and client-side certificate files to initiate them.

Make sure the following files are present in the data directory on each node. See SSL certificate setup in the PostgreSQL documentation for guidance on generating them.

- `server.crt` and `server.key` — the node's server certificate and private key.
- `bdr_client.crt` — the client certificate for the node.
- `bdr_client.key` — the private key for the client certificate. This file must not be world-readable.
- `root.crt` — the CA certificate used to verify both client and server certificates.

The certificates must also meet these requirements:

- The server's SSL certificate must be directly or indirectly signed by the CA in `root.crt` .
- The hostname or IP address in the connection must match the certificate's subject alternative name (SAN) or common name (CN).
- The CN of the client certificate must match the Postgres user making the PGD connection.

> **Note**
>
> Postgres doesn't currently support subject alternative names for IP addresses. If nodes connect by IP address rather than hostname, the IP address must match the CN field of the certificate.

### Configuring certificate authentication

1. Enable SSL in `postgresql.conf` on each node:

```
ssl = on
ssl_cert_file = server.crt
ssl_key_file = server.key
```

Restart Postgres on each node to apply the change:

```
pg_ctl restart -D "$PGDATA"
```

1. Check the current connection string for each node:

```
SELECT node_name, interface_connstr FROM bdr.node_summary;
```

1. Use *bdr.alter_node_interface()* to update each node's connection string with the SSL parameters. The function replaces the entire connection string, so include all existing parameters alongside the new SSL ones. Run it on every node in the group, including the node being changed. For example:

```
SELECT bdr.alter_node_interface(
  node-1,
  host=host-1 port=5432 dbname=pgddb user=pgd_replication sslmode=verify-full␣
↪sslcert=bdr_client.crt sslkey=bdr_client.key sslrootcert=root.crt
);
```

1. Add entries to each node's `pg_hba.conf` to permit inter-node connections:

```
hostssl all         pgd_replication <node_subnet>/24 cert
hostssl replication pgd_replication <node_subnet>/24 cert
```

Replace `<node_subnet>` with the actual subnet or IP range of your PGD nodes.

1. Reload Postgres on each node to apply the `pg_hba.conf` changes:

```
pg_ctl reload -D "$PGDATA"
```

### Authenticating with SCRAM-SHA-256 and CA verification

If client certificates aren't available, use password authentication while still encrypting the connection and verifying the server's certificate against a CA.

### Provisioning certificates and credentials

Make sure the following are in place on each node:

- `server.crt` and `server.key` in the data directory — the node's server certificate and private key. See

**SSL certificate setup in the**
PostgreSQL documentation for guidance on generating them.

- `root.crt` in the data directory — the CA certificate used to verify the server's SSL certificate.
- A

**.pgpass file in the home directory of the Postgres process**
user, containing the correct password for each node. The file must have permissions `0600` or stricter. If permissions are too open, the file is silently ignored.

### Configuring SCRAM-SHA-256 authentication

1. Enable SSL in `postgresql.conf` on each node:

```
ssl = on
ssl_cert_file = server.crt
ssl_key_file = server.key
```

Restart Postgres on each node to apply the change:

```
pg_ctl restart -D "$PGDATA"
```

1. Use *Node management interfaces* to update each node's connection string. The function replaces the entire connection string, so include all existing parameters alongside the new SSL ones. Run it on every node in the group, including the node being changed.

```
SELECT bdr.alter_node_interface(
  node-1,
  host=host-1 port=5432 dbname=pgddb user=pgd_replication sslmode=verify-ca␣
↪sslrootcert=root.crt
);
```

1. Add entries to each node's `pg_hba.conf` for inter-node connections:

```
hostssl all         pgd_replication <node_subnet>/24 scram-sha-256
hostssl replication pgd_replication <node_subnet>/24 scram-sha-256
```

Replace `<node_subnet>` with the actual subnet or IP range of your PGD nodes.

1. Reload Postgres on each node to apply the `pg_hba.conf` changes:

```
pg_ctl reload -D "$PGDATA"
```

### Verifying the configuration

After configuring SSL, confirm that replication connections are encrypted and that all nodes are active. Run these queries on any node in the cluster.

- Verify that replication connections are using SSL:

```
SELECT pid, ssl, version, cipher, bits
FROM pg_stat_ssl
JOIN pg_stat_replication USING (pid);
__OUTPUT__
pid | ssl | version |         cipher          | bits
-----+-----+---------+-------------------------+------
51 | t   | TLSv1.3 | TLS_AES_256_GCM_SHA384 |  256
50 | t   | TLSv1.3 | TLS_AES_256_GCM_SHA384 |  256
(2 rows)
```

A successful result shows one row per replication connection, with `ssl` set to `t` . This output confirms encryption is active, but doesn't show which authentication method is in use.

- Confirm the authentication method configured in `pg_hba.conf` :

```
SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules
WHERE auth_method IN (cert, scram-sha-256);
__OUTPUT__
type   |   database    | user_name  |  address   |  auth_method
---------+---------------+------------+------------+---------------
hostssl | {all}         | {pgd_replication} | 172.20.0.0 | scram-sha-256
hostssl | {replication} | {pgd_replication} | 172.20.0.0 | scram-sha-256
(2 rows)
```

The rows you added appear with the `auth_method` you configured, for example `cert` or `scram-sha-256` .

- Confirm all nodes are still replicating normally:

```
SELECT node_name, peer_state_name, peer_target_state_name
FROM bdr.node_summary;
```

(continues on next page)

(continued from previous page)

```
__OUTPUT__
node_name | peer_state_name | peer_target_state_name
----------+-----------------+-----------------------
node-1    | ACTIVE          | ACTIVE
node-3    | ACTIVE          | ACTIVE
node-2    | ACTIVE          | ACTIVE
(3 rows)
```

All nodes appear as `ACTIVE` in both state columns.

### 8.5.2 Securing client application connections

Client application connections to PGD pass through Connection Manager, which provides read-write and read-only routing endpoints. Securing this path requires configuration at two points: on the client-facing side, to control how applications authenticate and connect, and on the backend side, to control how Connection Manager authenticates to Postgres nodes on behalf of clients.

#### Securing client to Connection Manager connections

Enable SSL and configure authentication to encrypt application traffic and control which clients can connect to the cluster.

#### Enabling SSL

To use SSL for client connections, the following files must be present in the data directory on each node. See SSL certificate setup in the PostgreSQL documentation for guidance on generating them.

- `server.crt` and `server.key` — the node's server certificate and private key.

Enable SSL in `postgresql.conf` on each node:

```
ssl = on
ssl_cert_file = server.crt
ssl_key_file = server.key
```

Restart Postgres on each node to apply the change:

```
pg_ctl restart -D "$PGDATA"
```

#### Meeting SSL certificate requirements

Connection Manager uses rustls for TLS, which enforces stricter certificate requirements than OpenSSL. Certificates that don't meet these requirements will prevent Connection Manager from starting. The requirements apply to the same `server.crt` file used by Postgres. Some EDB packages are built with FIPS support enabled, in which case Connection Manager also applies FIPS 140-3 cryptographic requirements.

Server certificates must:

- Be X.509 v3
- Include `basicConstraints: CA:FALSE`
- Include a `subjectAltName` with the node's hostname or IP address
- Include `extKeyUsage: serverAuth`

Certificates must use RSA or ECDSA keys with SHA-256 or stronger. DSA keys and MD5 or SHA-1 signatures are not supported.

Unlike OpenSSL, rustls does not fall back to the common name (CN) for hostname validation. The `subjectAltName` extension is required.

### Configuring authentication

Connection Manager supports most of the authentication methods available in Postgres. Configure authentication by adding rules to pg_hba.conf on each node, then reload Postgres to apply them.

For the supported authentication methods, see *Connection Manager authentication methods* .

Add rules to `pg_hba.conf` on each node to control which clients can connect and how they authenticate. For example, to require password authentication from application servers:

```
hostssl   all   all   10.0.0.0/8   scram-sha-256
```

After editing `pg_hba.conf` , reload Postgres to apply the changes:

```
pg_ctl reload -D "$PGDATA"
```

**Note**

Where a *pg_hba.conf* rule is not supported by Connection Manager, it is logged as a warning and ignored. Connection Manager continues to operate using the rules it does support. Review the Postgres server log after any *pg_hba.conf* change to check for unsupported rules.

### Configuring endpoints

Connection Manager exposes three endpoints: a read-write endpoint that routes traffic to the primary node, a read-only endpoint that load balances across replicas, and an HTTP endpoint for the management API. Configure endpoint ports if the defaults conflict with other services in your environment. You can also cap connection counts to protect the database, or enable HTTPS on the HTTP endpoint.

| Option | Default | Purpose |
|---|---|---|
| *read_write_port* | Postgres port + 1000 (usually 6432) | Read-write connections |
| *read_only_port* | *read_write_port* + 1 (usually 6433) | Read-only connections |
| *http_port* | *read_write_port* + 2 (usually 6434) | HTTP management API (returns JSON) |

Configure these using `pgd group set-option` :

```
pgd group set-option --group-name <group_name> read_write_port <port>
```

### Enabling HTTPS on the HTTP endpoint

By default, the HTTP management endpoint uses plain HTTP. To enable HTTPS:

1. Set `use_https` to `true` :

```
pgd group set-option --group-name <group_name> use_https true
```

2. Connection Manager uses the same `server.crt` and `server.key` files configured in the *Enabling SSL* section. Make sure these files are present in the data directory on each node.

> **Note**
>
> HTTPS can be enabled even if *ssl = on* is not set globally in Postgres. If either certificate file is missing, Connection Manager falls back to plain HTTP regardless of the *use_https* setting.

### Setting connection limits

Connection Manager enforces separate limits for client and server connections on each endpoint, defaulting to `max_connections` from `postgresql.conf` . To override a limit:

```
pgd group set-option --group-name <group_name> read_write_max_client_connections
↪<value>
```

| Option | Default | Description |
|---|---|---|
| *read_write_max_client_c nections* | *max_connections* | Maximum client connections on the read-write port |
| *read_write_max_server_c nections* | *max_connections* | Maximum server connections for read-write traffic |
| *read_only_max_client_co nections* | *max_connections* | Maximum client connections on the read-only port |
| *read_only_max_server_c nections* | *max_connections* | Maximum server connections for read-only traffic |

> **Note**
>
> The server connection limits (*read_write_max_server_connections* and *read_only_max_server_connections* ) are currently applied per user. This is a known limitation that is planned to be lifted in a future release.

### Setting consensus timeout

By default, Connection Manager refuses connections immediately when it loses quorum, which protects against routing traffic to a cluster that can't guarantee consistency. Setting a consensus timeout gives the cluster a grace period to recover before connections are refused, which improves availability during transient failures at the cost of accepting connections while the cluster state is uncertain. Set this value only if your application can tolerate the uncertainty during the grace period.

```
pgd group set-option --group-name <group_name> read_write_consensus_timeout <seconds>
```

### Securing Connection Manager to node connections

When Connection Manager accepts a client connection, it connects to the appropriate backend node using the client's credentials. Each backend node must have a `pg_hba.conf` rule that permits this connection from Connection Manager.

SCRAM-SHA-256 is the recommended authentication method for these backend connections. Certificate authentication is possible, but complex to configure and manage, as Connection Manager can't present a per-user client certificate to the server.

1. Add a rule to `pg_hba.conf` on each node to allow Connection Manager to connect as client users:

```
hostssl all all <node_address>/32 scram-sha-256
```

Replace `<node_address>` with the address from which Connection Manager connects to Postgres on that node.

1. After editing `pg_hba.conf` , reload Postgres to apply the changes:

```
pg_ctl reload -D "$PGDATA"
```

## 8.6 Handling replication apply errors

When a replication apply failure occurs, PGD retries the failed transaction indefinitely by default. Persistent failures can overload the system with repeated apply attempts and fill logs with recurring error messages.

Use the *apply_error_policy* node group option to control what PGD does when a transaction fails to apply. Set it to stop retrying after a set number of attempts, record failed changes for review, and either disable the subscription or skip the failed transaction and continue replicating.

### 8.6.1 Choosing an error policy

Four error policies are available, configured at the node group level:

- **``keep_retrying``** retries the failed transaction indefinitely. PGD keeps retrying silently, nothing is recorded, and no subscription is disabled. Use for transient failures expected to self-heal, such as temporary lock contention. `keep_retrying` is the default.
- **``disable_after_retries``** retries up to `apply_error_max_retries` times, then disables the subscription and records the reason. Use when failures can't persist indefinitely but might still be transient.
- **``record_all_and_disable``** records the complete failed transaction, then disables the subscription. Use when you need a complete record of what failed before halting replication.
- **``record_all_and_skip``** records the complete failed transaction, skips it, and continues replicating. The subscription stays enabled, but failed changes accumulate silently in `bdr.pending_failed_changes` . Use for high-availability deployments where continuity matters more than guaranteeing every transaction is applied.

The `record_all_and_disable` and `record_all_and_skip` policies capture the full transaction by replaying it in capture-only mode before taking action, giving you a complete record to inspect and resolve later.

### 8.6.2 Configuring the error policy

Set the policy on the node group using `bdr.alter_node_group_option` . To find your node group name:

```
SELECT node_group_name FROM bdr.local_node_summary;
```

Apply your chosen policy:

```
SELECT bdr.alter_node_group_option(
    top_group,
    apply_error_policy,
    record_all_and_disable
);
```

Configure the supporting parameters as needed. For example:

```
- - Retry attempts before the policy takes action (default: 3)
SELECT bdr.alter_node_group_option(
    top_group,
    apply_error_max_retries,
    5
);

- - Maximum transactions record_all_and_skip can auto-skip before disabling.
- - Set to -1 for unlimited. Default is NULL (unlimited).
SELECT bdr.alter_node_group_option(
    top_group,
    apply_error_max_skips,
    100
);

- - Maximum bytes to capture per failed transaction (default: 10485760, which is 10␣
↪MB)
SELECT bdr.alter_node_group_option(
    top_group,
    apply_error_max_record_size,
    52428800
);
```

For the full list of apply error group options, see *Group configuration using bdr.alter_node_group_option()* .

### 8.6.3 Responding to a disabled subscription

When a subscription is disabled by the `disable_after_retries` or `record_all_and_disable` policy, replication stops. Follow these steps to investigate and resolve the failure.

1. Check which subscriptions are affected and why:

```
SELECT sub_name, disable_reason, effective_error_policy, pending_changes
FROM bdr.subscription_error_status
WHERE NOT sub_enabled;
```

The `disable_reason` column records why PGD disabled the subscription, and `pending_changes` shows how many failed changes are waiting for review. For the full column reference, see *bdr.subscription_error_status* .

1. Review the failed changes:

```
SELECT id, failed_txn_id, subscription_name, remote_xid, remote_commit_lsn,
       operation, qualified_table, sql_statement, error_message
FROM bdr.pending_failed_changes_sql
WHERE change_status = pending
ORDER BY id;
```

1. Optionally, analyze whether it's safe to skip. `bdr.format_skip_analysis` classifies the skip as `SAFE` , `UNSAFE` , or `UNKNOWN` and explains the reasoning:

```
SELECT bdr.format_skip_analysis(id)
FROM bdr.pending_failed_changes
```

(continues on next page)

(continued from previous page)

```
WHERE change_status = pending
LIMIT 1;
```

1. Preview the resolution using `bdr.resolve_failed_transaction` , which defaults to `dry_run := true` :

```
SELECT bdr.resolve_failed_transaction(
    bdr_mydb_mygroup_node1_node2,   -- subscription_name
    12345::xid,                     -- remote_xid
    0/1234567::pg_lsn,              -- remote_commit_lsn
    skipped,                        -- resolution: skipped, applied, or discarded
    true                              -- re_enable
);
```

Alternatively, pass `failed_txn_id` (the UUID from `bdr.pending_failed_changes` ) instead of the three-part identifier.

1. Execute the resolution by passing `dry_run := false` :

```
SELECT bdr.resolve_failed_transaction(
    bdr_mydb_mygroup_node1_node2,
    12345::xid,
    0/1234567::pg_lsn,
    skipped,
    true,    -- re_enable
    false    -- dry_run
);
```

To resolve without immediately re-enabling, pass `re_enable := false` and re-enable manually when ready:

```
SELECT bdr.alter_subscription_enable(bdr_mydb_mygroup_node1_node2);
```

1. Reset the retry and skip counters so the policy thresholds apply fresh to future failures:

```
SELECT bdr.alter_subscription_reset_error_counters(bdr_mydb_mygroup_node1_node2);
```

1. Once you've confirmed resolution, delete the resolved rows from `bdr.pending_failed_changes` to reclaim space. The resolution summary is preserved in `bdr.resolved_transactions` .

```
DELETE FROM bdr.pending_failed_changes
WHERE change_status IN (skipped, applied);
```

### Resolving individual changes

When a failed transaction contains a mix of changes, some safe to resolve and others not, use `bdr.skip_failed_change` and `bdr.apply_failed_change` to operate on individual rows rather than the transaction as a whole. Once you've resolved all changes individually, call `bdr.resolve_failed_transaction` to advance the subscription position and re-enable it.

### Handling truncated captures

When a failed transaction exceeds `apply_error_max_record_size` , PGD records as many changes as it can and sets `capture_truncated = true` in `bdr.pending_failed_changes` . Truncated transactions can't be reapplied using `bdr.apply_failed_transaction` because the record is incomplete.

To resolve a truncated transaction, use `bdr.reconstruct_failed_transaction` to review the partial SQL, manually apply the complete transaction on the target node using that output as a reference, then use `bdr.skip_failed_transaction` to mark the captured changes as skipped and re-enable the subscription.

To prevent truncation on large transactions, increase `apply_error_max_record_size` before the next failure.

### 8.6.4 Monitoring for accumulated failures

With `record_all_and_skip` , the subscription stays enabled and replication continues, but failed transactions accumulate silently. Check `bdr.subscription_error_status` regularly to catch them:

```
SELECT sub_name, total_skips, pending_changes
FROM bdr.subscription_error_status;
```

When `pending_changes` is non-zero, review and resolve the accumulated failures using the same steps in *Responding to a disabled subscription* , skipping the re-enable step since the subscription is still active.

To review past resolutions:

```
SELECT subscription_name, origin_node, remote_xid, resolution,
       insert_count, update_count, delete_count, error_message, resolved_at
FROM bdr.resolved_transactions_info
ORDER BY resolved_at DESC;
```

## 8.7 Configuring connection pooling

Enable *Connection Manager Authentication* 's connection pooling for a PGD node group to maximize database throughput by recycling server connections across clients, preventing connection spikes from applications with many short-lived connections. Connection Manager's built-in transaction pooling requires no additional software to deploy or maintain.

For guidance on writing applications that work correctly with each pool mode, see *Using connection pooling in your application* .

### 8.7.1 Prerequisites

- Connection Manager running on each data node (PGD 6.4 and later). See *Monitoring the Connection Manager* to verify it's running.
- `bdr_superuser` role or equivalent to run `bdr.alter_node_group_option`

### 8.7.2 Enabling connection pooling

Connection pooling is disabled by default, with `server_pool_mode` set to `none` . To enable it, set the mode to `session` or `transaction` for your node group.

Using SQL:

```
SELECT bdr.alter_node_group_option(mygroup, server_pool_mode, transaction);
```

Using the PGD CLI:

```
pgd group mygroup set-option server_pool_mode transaction
```

When enabling pooling, existing connections aren't affected and only new connections use the updated mode. When switching away from `transaction` mode, existing client connections with no open transaction are closed the next time they start a transaction.

Use `transaction` mode when connections are long-lived but transactions are short, with idle time between them, for example when client applications use their own connection pooling.

Use `session` mode for short-lived connections, or if your applications rely on session state that persists across transactions (such as `SET` , temporary tables, or advisory locks). For full details on each mode, see *Connection pooling* .

!!!Warning Transaction mode resets most session state at the end of each transaction. Before enabling it, check with your development team whether any applications rely on features that require session state to persist across transactions, such as `WITH HOLD` cursors, `LISTEN` , or `SET` changes. See *Using connection pooling in your application* for a full list of affected features.

Check the active pool mode.

```
SELECT node_group_name, server_pool_mode FROM bdr.node_group_summary;
```

### 8.7.3 Monitoring the pool

*Monitoring the Connection Manager* describes the available metrics and HTTP endpoints for tracking active connections and pool utilization.

### 8.7.4 Migrating from pgBouncer

Connection Manager provides built-in connection pooling, removing the need for a separate pooler such as pgBouncer. Its pooling modes are similar to pgBouncer's session and transaction pooling modes. Follow these steps to migrate.

1. Set `server_pool_mode` to match your current pgBouncer pool mode (`session` or `transaction` ).
2. Set `read_write_max_server_connections` to a similar value as pgBouncer's `pool_size` , and `read_write_max_client_connections` to a similar value as pgBouncer's `max_client_conn` . As a starting point, set `read_only_max_server_connections` and `read_only_max_client_connections` to the same values. Tune all four settings based on your actual workload.
3. Update application connection strings to point to Connection Manager's read-write port instead of pgBouncer.
4. Verify application behavior:
   - In `session` mode, application behavior is unchanged from a direct Postgres connection.
   - In `transaction` mode, pay particular attention to any use of `SET` commands, prepared statements, or `WITH HOLD` cursors, and adjust as needed. See *Using connection pooling in your application* .
1. Remove pgBouncer from your infrastructure once migration is verified.

Authentication configuration stays in `pg_hba.conf` on each PGD node. No separate user list or pooler configuration file is needed. Any user that can connect to Postgres through Connection Manager can use the pool, with no second place to grant or revoke access.

Connection Manager imposes additional restrictions compared to a direct Postgres connection in some scenarios. Test your application thoroughly before completing the migration.

CHAPTER 9

# Guidance for Developers

## 9.1 Designing applications for PGD

Design applications for PGD by accounting for replication behavior, write conflicts, and consistency tradeoffs that don't arise on a single-node database.

- *Preparing your application for PGD* applies to both new and existing applications, addressing workload patterns, conflict avoidance, connection routing, and the replication behaviors specific to PGD.
- *Assessing an existing application for PGD* identifies compatibility gaps between an existing Postgres deployment and PGD, from schema patterns flagged by `pgd assess` to backup strategy changes and incompatible replication configurations.

### 9.1.1 Preparing your application for PGD

PGD's distributed nature introduces design decisions that don't arise with single-node Postgres, whether you're porting an existing application or building from scratch. This guidance assumes the full database schema and all data are replicated to all nodes.

> **Note**
>
> PGD also supports selective replication, which lets you control which tables are replicated to which nodes. See *Configuring selective replication* .

#### Understanding your workload

Before making any design decisions, understand your application's read-to-write ratio. Reads can be routed to different nodes to spread load and fully offload it from write leader nodes.

If your application uses an object-relational mapper (ORM), review its emitted DDL to ensure generated tables have primary keys, configure separate connection pools to map to the Connection Manager's read-write and read-only ports, and audit generated SQL for the patterns described below.

### Planning your write strategy

Design your application to avoid write conflicts as the primary goal. Write transactions replicate to all data nodes in the cluster, so conflicts have a higher cost than on a single-node database.

If your application can cleanly partition write traffic by geography or user group with no overlapping writes between partitions, local routing with multiple write leaders in separate regions is the right fit. Keeping writes local to a region eliminates cross-region conflicts and improves throughput. If that clean separation can't be guaranteed, the conflict rate rises as write volume scales. In that case, global routing, where all writes go to a single write leader, is the safer choice. The routing configuration itself is a database administrator (DBA) task, but understanding your application's write patterns determines which model fits.

PGD uses global sequences by default. `int4` and `int2` sequences use `galloc` , which allocates ranges of values per node. `int8` sequences use `snowflakeid` . Both produce gaps in IDs. Don't assume gap-free sequences in your application. See *Sequences* for details on sequence kinds and their tradeoffs. For commutative updates that need to merge without conflict, use conflict-free replicated data types (CRDTs) .

For details on conflict detection and resolution, see *Conflict Management* .

### Planning your read strategy

Route read traffic to the Connection Manager's read-only port, which distributes connections across all nodes rather than directing them to the write leader. HAProxy can also balance read traffic across nodes using the Connection Manager's read-only port. See *Load Balancing with Connection Manager* for configuration examples.

For reads where a user must immediately see their own writes, such as a recently placed order or an updated account balance, route those reads to the write leader or use stricter *Commit scopes* to ensure the write is visible on enough nodes before it completes. For reads where stale data is acceptable, such as aggregate counts or social metrics, use the read-only port.

### Designing your schema for replication

Define a `PRIMARY KEY` or `UNIQUE` constraint on every table that takes `UPDATE` or `DELETE` operations. PGD uses the unique index behind the constraint for fast row lookups during replication apply.

For tables with no primary key and no unique constraint, PGD emits a warning when the table is created or added to replication. The table still replicates because `REPLICA IDENTITY FULL` is the default. Updates and deletes replicate by matching the full old row, but apply-side lookups are much slower. PGD uses any suitable index if one exists, or falls back to a sequential scan. A missing primary key is usually an oversight, and the scan cost compounds under write load. Don't rely on the default in production tables.

PGD handles updatable views transparently. Data manipulation language (DML) through an updatable view resolves to the underlying base table on the origin and is applied to the same base table on the target.

PGD supports partitioned tables transparently. Changes to any partition are replicated downstream automatically. For managing large tables that grow over time, *AutoPartition in PGD* automates the creation and dropping of partitions using low-conflict locking.

Ensure all collations used in the schema are available on all nodes, and that the default collation is the same across nodes. Row replication itself isn't affected, since PGD uses equality searches on replica identity values. However, unique indexes defined with nonmatching collation qualifiers can behave unexpectedly, and row filters that use collatable expressions can produce inconsistent results if collations differ between nodes.

### Accounting for replication behavior

PGD replicates only the final result of each statement, after all triggers and rules are processed. An `INSERT ... ON CONFLICT UPDATE` sends either an insert or an update, whichever occurred on the origin. If an update or delete affects zero rows, nothing is replicated.

The following behaviors differ from a single-node database and require attention during development.

- **Triggers** fire only on the originating node where the SQL was first executed, the same as on single-node Postgres. When PGD applies a replicated change on a target node, triggers on that table don't fire. Their effects are replicated instead.
- To also fire a trigger when changes apply on a target node, use `ALTER TABLE ... ENABLE ALWAYS TRIGGER`. To fire only at apply time, use `ALTER TABLE ... ENABLE REPLICA TRIGGER`.
- Statement-level triggers (`FOR EACH STATEMENT`) and per-column `UPDATE` triggers (`UPDATE OF column_name`) are never executed on apply, regardless of setting.
- Trigger functions that fire at apply time must fully qualify object references, or set the search path explicitly with `ALTER FUNCTION ... SET search_path = ...`. PGD apply uses the system-level default `search_path`, and functions that assume a different path fail on apply.
- Rules execute on the originating node only, the same as on single-node Postgres. PGD replicates the resulting changes, not the rule itself.
- Row-level locks from `SELECT ... FOR UPDATE/FOR SHARE` and advisory locks aren't replicated. `SERIALIZABLE` isolation is supported, but transactions aren't serialized across nodes, and concurrent transactions on multiple nodes can still conflict.
- Use `BYTEA` columns for binary data. They replicate fully, including the underlying `TOAST` storage, up to 1 GB. If your application requires the stream-oriented large object interface, PGD also supports *Replicated large objects*.
- Avoid volatile functions in `CHECK` constraints. PGD reexecutes `CHECK` constraints on apply by default, and if a volatile function returns a different result than it did on the origin, replication can break. This behavior is controlled by the *pgd group set-option* group option.
- Don't mark `REPLICA IDENTITY` columns as external.
- Foreign keys, including cascading ones, are enforced only on the writing node and aren't rechecked on apply. Use them for local consistency, but don't assume they prevent cross-node integrity violations during conflicts.

#### Avoiding data anomalies

PGD uses replica identity values to identify rows and table names to identify tables. Reusing an identifier after deletion, or a table name after dropping, creates ambiguity that PGD can't resolve. This ambiguity is the ABA problem, where PGD can't determine whether an identifier refers to the current object, the previous one, or an older one.

To avoid data anomalies and divergence:

- Use unique identifiers for rows on `INSERT`.
- Don't modify unique identifiers on `UPDATE`.
- Don't reuse deleted unique identifiers.
- Don't reuse dropped object names.

Violations can lead to data anomalies across nodes. While unlikely, they aren't impossible. A deleted row's identifier is safe to reuse only after the `DELETE` has replayed on all nodes, including any that were down. Under good conditions that takes under a second, but can take days if a node is severely degraded.

### 9.1.2 Assessing an existing application for PGD

Before migrating an existing application to PGD, assess its compatibility to identify schema patterns, backup gaps, and replication configurations that need to change. The `pgd assess` command is designed for Postgres deployments not yet running PGD, including single-instance setups and physical replication clusters.

#### Using pgd assess

If you have a running Postgres instance serving your application, use `pgd assess` to evaluate its compatibility with PGD. The command connects to your Postgres instance, checks the schema and internal state of the database for the most common application patterns, and provides recommendations on how to adapt them for PGD. It doesn't analyze live traffic or actual queries of your application.

```
pgd assess --dsn <connection_string>
```

For full details on the command and its output, see *Using pgd assess* .

#### Reviewing backup strategies

Identify what tools are currently used for backup. A PGD cluster spans multiple nodes and locations, increasing disk space and backup bandwidth requirements. Plan for at least one backup per group or location rather than backing up every node individually.

An alternative strategy is to back up a single node and rebuild the rest of the cluster from it. Restore to a single-node PGD cluster first, then run `pgd node setup` for each additional node to join them via a physical join. See *Backup and recovery* for details.

#### Reviewing replication and CDC setups

By default, outgoing logical replication to a data warehouse or event streaming pipeline is tied to a single PGD node. Logical replication slots aren't replicated across the cluster unless CDC Failover support is explicitly enabled. With CDC Failover support enabled, slots are replicated across nodes, but failover isn't fully transparent. The consumer must support reconnecting to a new node. See *CDC Failover support* for details.

Any synchronization with other databases through a two-phase commit (2PC) protocol or XA transactions isn't compatible with PGD.

## 9.2 Managing geo-replication

PGD supports clusters that span multiple geographic locations, including data centers, availability zones, and cloud regions, where each location can accept writes independently. With geo-replication, you can build resilient, globally distributed database systems that protect against location failures and keep data close to users.

### 9.2.1 When to use geo-replication

Geo-replication is the right choice when you need one or more of the following:

- **Location failure protection**: if a data center or availability zone goes offline, another location continues to serve reads and writes without data restore.
- **Data residency**: regulatory requirements mandate that certain data stays within a specific region or country.
- **Low-latency reads across regions**: users in different regions need fast access to data without round-tripping to a central location.
- **Disaster recovery with low Recovery Time Objective (RTO)**: you need near-instantaneous failover to another location without manual restore from backup.

### 9.2.2 Deployment workflow

Follow these steps when setting up a new geo-distributed PGD cluster.

### 1. Choose a deployment pattern

Decide on the number of active write locations, routing mode, and commit scope before configuring anything. The deployment pattern affects every subsequent decision.

See *Choosing a geo-replication deployment pattern* .

### 2. Plan your node distribution and create subgroups

Before provisioning nodes, measure network latency between locations and plan quorum so that no single location holds a majority of voting nodes. Determine whether you need a witness location. Then create a subgroup for each location under the top-level group and assign nodes to their subgroups.

See *Setting up your geo-replication topology* .

### 3. Configure routing

Decide between local routing (write leader per location) and global routing (single cluster-wide write leader). Local routing is the default and requires no changes. For global routing, enable it on the top-level group and disable it on subgroups.

Set route priority to control which nodes are preferred as write leaders during failover.

See *Configuring routing thresholds* .

### 4. Configure commit scopes

Choose a commit scope based on your durability requirements and configure it for each location subgroup.

See *Commit scopes* .

### 5. Tune replication and sequences

Tune parallel apply and streaming settings to reduce replication lag under high write volumes. Confirm that sequences are configured to generate cluster-wide unique values.

See *Replication and sequence tuning* .

### 6. Configure lag control

If cross-region replication lag is a concern, add Lag Control rules to your commit scope to throttle write throughput when remote nodes fall behind.

See *Controlling replication lag* .

### 7. Verify the cluster and prepare for operations

Once configuration is complete, verify that subgroups, routing, and commit scopes are set up correctly before going to production. The maintenance page also covers ongoing operational tasks such as rolling upgrades, DDL considerations, network partition handling, and conflict monitoring.

See *Performing cluster maintenance* .

## Choosing a geo-replication deployment pattern

Before configuring geo-replication, decide on a deployment pattern. The choice affects how you configure routing, how the cluster behaves during a location failure, and how much operational complexity you take on.

The full description of each pattern is in *Deployment patterns* . This page covers the decisions that matter specifically for geo-replication.

### Deciding on the number of active write locations

The number of active write locations is the most fundamental decision. More active write locations means lower write latency for users in each region, but cross-region write conflicts become inevitable and conflict resolution must be part of your application design.

PGD uses the Raft consensus algorithm to coordinate decisions across nodes, such as electing a write leader and performing DDL operations. Raft requires a quorum (a majority of nodes) to be available before it can make these decisions. In a geo-distributed cluster, this means that the number of active locations directly affects whether the cluster can maintain quorum during a location failure.

- **Two active write locations:** The simplest geo-replication setup. Each location serves local reads and writes, but losing one location breaks Raft majority, which blocks DDL and global sequences until the location recovers. Adding a witness node in a third location resolves this limitation without the cost of a full third active location. The relevant patterns are *Two data groups, active-active* and its witness variant.
- **Three or more active write locations:** Maintain natural Raft majority even when one location fails, eliminating the quorum concern. The trade-off is higher operational complexity. The relevant deployment patterns are *Three data groups, active-active-active* and *Multiple locations, data residency* for the variant where sensitive data must stay within its origin region.

### Choosing between local and global routing

PGD supports two routing modes for write traffic. You configure your preferred routing mode by setting the *enable_routing* configuration option for your node group.

- **Local routing:** Each location has its own write leader and routes writes locally. This approach gives the lowest write latency and means each location operates independently for most operations. However, cross-region conflicts are possible, and losing a location with only two groups breaks global consensus.
- **Global routing:** The entire cluster has a single write leader regardless of location. All write traffic is routed to that leader, eliminating cross-region conflicts at the cost of added latency for writes originating in other regions. Even if multiple locations are capable of accepting writes, global routing constrains writes to a single leader across those locations. This mode works with three or more active write locations, or with two data groups with a witness location. If the primary location fails, the two remaining locations still have Raft majority and can elect a new write leader.

For configuration steps, see *Configuring routing thresholds* .

### Enforcing data residency requirements

Regulations such as GDPR, CCPA, and national data localization laws require that personal or sensitive data stays within its origin region.

Data residency enforcement is built into the *Multiple locations, data residency* deployment pattern. With local routing, each region's subgroup has its own write leader and applications connect to a local endpoint, so sensitive data stays in-region by default. Cross-region replication is selective: only global reference data (product catalogs, configuration, anonymized analytics) replicates between groups. Personal data, financial records, and other regulated data remain local to their origin group.

You must design your schema to clearly separate local from global data, and configure your application to always write regional data to the correct local endpoint.

### Comparison

Each geo-replication deployment pattern makes different trade-offs across quorum resilience, routing options, conflict handling, and data residency.

| Deployment pattern | Active write locations | Quorum during location failure | Supports global routing | Conflict resolution required | Data residency enforcement |
|---|---|---|---|---|---|
| Two data groups, active-active | 2 | No (Yes with witness) | No (Yes with witness) | Yes | No |
| Three data groups, active-active-active | 3+ | Yes | Yes | Yes | No |
| Multiple locations, data residency | 2+ | Yes | Yes | Yes | Yes |

### Setting up your geo-replication topology

Once you have chosen a deployment pattern, the next step is to configure the cluster topology. This involves planning your node distribution, creating location subgroups, and assigning nodes to those subgroups.

### Planning your node distribution

Before configuring the cluster, measure network latency and bandwidth between your locations. Cross-location latency directly affects replication lag, synchronous commit performance, and DDL duration.

For quorum planning, ensure no single location holds a majority of voting nodes. For example, if three of five nodes sit in location A and two sit in location B, location A alone holds majority. Losing location A makes the cluster read-only even though location B is still running.

For two-location deployments, a witness node in a third location is the standard solution. With two data nodes per location plus one witness you have five voters total, so either location can fail and the remaining nodes still hold a majority. Each location must have at least two data nodes for local high availability.

### Creating location subgroups

PGD uses a group hierarchy to represent your physical topology. Each geographic location maps to a node subgroup, and all location subgroups sit under a single top-level group that spans the whole cluster. Create a subgroup for each location:

```
SELECT bdr.create_node_group(
    node_group_name := <location_a>,
    parent_group_name := <top_group>,
    join_node_group := false
);

SELECT bdr.create_node_group(
    node_group_name := <location_b>,
    parent_group_name := <top_group>,
    join_node_group := false
);
```

The `join_node_group := false` parameter creates the group without joining the current node to it. Nodes are joined separately in the next step.

Replace <location_a> and <location_b> with descriptive names for your locations (for example, us_east and eu_west ), and <top_group> with the name of your top-level PGD group.

To verify the subgroups were created:

```
SELECT node_group_name, parent_group_name
FROM bdr.node_group_summary
ORDER BY parent_group_name;
```

### Assigning nodes to subgroups

When joining a node to the cluster, specify the subgroup it belongs to:

```
SELECT bdr.join_node_group(
    node_group_name := <location_a>
);
```

Run this on each node so it is associated with its location subgroup. Connection Manager uses subgroup membership to route write traffic to the correct write leader for each location.

Once subgroups are in place, configure routing and route priority for your deployment pattern. See *Configuring routing thresholds* .

### Configuring routing

In a geo-distributed PGD cluster, routing determines which node accepts writes at any given time. PGD 6 uses Connection Manager, which runs inside each Postgres instance, to handle this. Applications that need to write must connect to Connection Manager's read-write port, which routes the connection to the current write leader of the node group.

How write leadership is structured across locations depends on whether you configure local or global routing.

- **Local routing:** Gives each location its own write leader. When an application connects to a node in location A, Connection Manager routes it to location A's write leader, keeping write latency low. All locations accept writes simultaneously, which means cross-location conflicts are possible.
- **Global routing:** Designates a single write leader for the entire cluster. Connection Manager on any node routes write connections to that cluster-wide leader, regardless of which location the application is in. This eliminates cross-location conflicts but means writes from remote locations incur cross-location latency.

Two node group *Postgres settings* control write leader election:

- enable_raft — whether the group participates in Raft consensus
- enable_routing — whether the group elects its own write leader

In PGD, the cluster has a single top-level group spanning all nodes across all locations, and a subgroup per location. The enable_routing setting determines whether routing is local or global:

| | Top-level group *enable_routing* | Subgroup *enable_routing* |
|---|---|---|
| Local routing (default) | *off* | *on* |
| Global routing | *on* | *off* |

enable_raft is always on for both the top-level group and subgroups and doesn't need to change between configurations.

Local routing is the default and requires no explicit configuration. For global routing, set `enable_routing=on` on the top-level group and `enable_routing=off` on each subgroup.

Global routing is only available on the *Three data groups, active-active-active* pattern, the *Multiple locations, data residency* pattern, or the *Two data groups, active-active* pattern with a witness location. Without a witness, losing one location in a two data groups setup breaks Raft majority, making it impossible to elect or maintain a cluster-wide write leader.

### Configuration examples

#### Local routing

Each location subgroup elects its own write leader. The top-level group does not participate in write leader election.

```
-- Top-level group does not elect a write leader
SELECT bdr.alter_node_group_option(top_group, enable_routing, false);

-- Each subgroup elects its own write leader
SELECT bdr.alter_node_group_option(location_a, enable_routing, true);
SELECT bdr.alter_node_group_option(location_b, enable_routing, true);
```

Connection Manager routes writes to the local write leader, keeping write latency low for each location.

#### Global routing

The top-level group elects a single write leader for the whole cluster. Location subgroups do not elect their own.

```
-- Top-level group elects the cluster-wide write leader
SELECT bdr.alter_node_group_option(top_group, enable_routing, true);

-- Subgroups do not elect their own write leaders
SELECT bdr.alter_node_group_option(location_a, enable_routing, false);
SELECT bdr.alter_node_group_option(location_b, enable_routing, false);
```

All writes are routed to the single cluster-wide write leader. Remote locations incur cross-location write latency but there are no cross-location conflicts.

### Tuning Connection Manager

Connection Manager has several settings that are important to review for geo-distributed deployments.

- `read_write_port` and `read_only_port` — the ports Connection Manager listens on for write and read traffic respectively. Ensure these are consistently configured across all locations.
- `read_write_max_client_connections` and `read_only_max_client_connections` — limit the number of client connections accepted. In geo-distributed clusters with high connection counts per location, tuning these prevents resource exhaustion.
- `read_write_consensus_timeout` — how long Connection Manager waits after losing consensus before it stops accepting write connections. The default is `0` , meaning it stops immediately. In a geo-distributed cluster, cross-location latency can cause brief consensus interruptions that are not true failures. Consider setting a small timeout to avoid unnecessary write disruption during transient network events.
- `read_only_consensus_timeout` — same as above for read-only connections.

See *Configuring Connection Manager* for the full list of options.

### Setting route priority

Route priority controls which nodes are preferred as write leaders within a group. Higher values mean higher priority. Assign higher priority to local nodes so that write leadership stays close to application traffic under normal conditions and only moves to a remote node when necessary:

```
SELECT bdr.alter_node_option(
    node_name := <node_name>,
    config_key := route_priority,
    config_value := 1
);
```

Document your priority scheme so the operations team understands expected failover behavior across locations.

### Verifying current settings

Check the current Raft and routing settings for all groups:

```
SELECT node_group_name,
       node_group_enable_raft,
       node_group_enable_routing
FROM bdr.node_group;
```

View current write leaders:

```
SELECT node_group_name, write_lead
FROM bdr.node_group_routing_summary;
```

### Configuring commit scopes for geo-replication

Commit scopes control how PGD confirms transactions and how many nodes must acknowledge a write before it's considered committed. In geo-replication, the choice directly affects write latency and how much data you risk losing if a location fails.

The deployment patterns that support geo-replication use the following commit scopes:

| Commit scope | Description | Latency | Data loss risk | When to use | Supported topology patterns | Notes |
|---|---|---|---|---|---|---|
| **Majority protect** | Waits for a majority of nodes in the origin group to confirm before committing. | Medium | Low | Zero data loss required and write latency is acceptable. Recommended for most production workloads. | Any group with 3+ nodes | Survives minority node failures. |

continues on next page

Table 3 – continued from previous page

| Commit scope | Description | Latency | Data loss risk | When to use | Supported topology patterns | Notes |
|---|---|---|---|---|---|---|
| **Local protect** | Commits locally without waiting for other nodes. Changes replicate asynchronously. | Lowest | Highest | Lowest possible write latency and some data loss on failure is acceptable. | Any topology | Not recommended for production use. Data loss possible if a node fails before replication. |
| **Adaptive protect** | Synchronous to a majority when available, degrades to asynchronous after a timeout (default 10s). | Variable | Low | Durability by default, but cluster must stay writable during partial failures. | Groups with 2 data nodes + witness | Balances durability and availability. Graceful degradation during node failures. |
| **CAMO** | Synchronous replication with exactly-once semantics using a dedicated partner node. | Higher | None | Exactly-once transaction requirements where duplicate detection isn't possible at the application level. | Groups with exactly 2 data nodes | Prevents duplicate transactions after failover. Higher latency. Requires application changes. |
| **Quorum Commit** | Coordinates the commit decision across all participating nodes using two-phase commit before any node commits locally. Conflicting transactions are detected and aborted during the commit phase. | High | None | Workloads requiring distributed transaction consistency where conflicts can't be resolved after the fact, such as payments, core banking, and ledger systems. | Any group with 3+ nodes (majority required) | No *DEGRADE TO* support. Requires write leader routing. Application must handle serialization errors and retry. |

### Choosing a commit scope

Choosing a commit scope is a critical decision that directly affects latency, durability, and behavior during failures.

Use the following questions to identify the right commit scope for your deployment. Your answers depend on your topology, your Recovery Point Objectives (RPO) requirements, and whether your workloads have mixed durability needs.

- Does the workload require strict distributed consistency where conflicting transactions can never both commit, such as payments or core banking?
- **Yes** — use Quorum Commit

- **No** — continue
- Can the application tolerate data loss if a location fails?
- **Yes** — majority protect without remote confirmation is sufficient
- **No** — add remote node confirmation to the rule
- Do you have 2 data nodes + witness per group?
- **Yes** — use adaptive protect
- **No** — use majority protect
- Is cross-location write latency acceptable for all transactions?
- **Yes** — add remote confirmation to the rule
- **No** — use local durability with lag control
- Do you have mixed workloads with different durability needs?
- **Yes** — use multiple scopes with session or transaction overrides
- **No** — use a single default scope

For workloads requiring exactly-once transaction semantics, CAMO provides additional guarantees but requires application-level changes to implement. See *Commit At Most Once* for details.

### Creating the commit scope

Once you have chosen a commit scope, use `bdr.create_commit_scope()` for each origin node group the rule applies to. For most geo-distributed deployments, `ORIGIN_GROUP` is the recommended approach. It resolves dynamically to the subgroup of the node originating the transaction, so you only need one rule for the entire cluster. For example, to create a majority protect scope:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := majority_protect,
    origin_node_group := <top_group>,
    rule := MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT,
    wait_for_ready := true
);
```

If your locations have asymmetric durability requirements, you can create a separate rule per location instead. See *Using Quorum Commit* for full details.

When using majority protect in a multi-group deployment, you can extend the `rule` parameter in `bdr.create_commit_scope()` with cross-location options to control how remote nodes participate in the commit:

- **Local durability only** — commits as soon as a majority of nodes in the origin group confirm. Changes replicate to remote locations asynchronously.

```
MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT
```

- **Remote confirmation** — adds a requirement for at least one remote node to confirm before committing. Ensures data survives a complete location failure at the cost of cross-location round-trip time on every write.

```
MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT AND ANY 1 NOT ORIGIN_GROUP SYNCHRONOUS COMMIT
```

- **Lag control** — commits locally at full speed but throttles transactions if remote lag exceeds the configured threshold. Prevents the remote location from falling too far behind without requiring synchronous confirmation.

```
MAJORITY ORIGIN_GROUP SYNCHRONOUS COMMIT AND ALL NOT ORIGIN_GROUP LAG CONTROL (max_
↪lag_time = 30s)
```

### Creating a Quorum Commit scope

Quorum Commit requires all participating nodes to agree before any node commits. Unlike other commit scope kinds, `DEGRADE TO` isn't supported, and eager conflict resolution is always active. The `ABORT ON` clause is mandatory.

```
SELECT bdr.create_commit_scope(
    commit_scope_name := quorum_scope,
    origin_node_group := <top_group>,
    rule := MAJORITY ORIGIN GROUP QUORUM COMMIT ABORT ON (timeout = 6s),
    wait_for_ready := true
);
```

For multi-region clusters, use `MAJORITY FOR EACH GROUP IN CLUSTER` to require a majority within each subgroup. For guidance on choosing the right configuration for your topology, see *QUORUM COMMIT* .

### Creating a CAMO scope

If you are using CAMO, the rule uses the `CAMO` keyword instead of `SYNCHRONOUS COMMIT` . CAMO requires a designated partner node within the same subgroup that tracks transaction outcomes, so that if the origin node fails during commit, the application can query the partner to determine whether the transaction was applied:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := camo_scope,
    origin_node_group := <location_a>,
    rule := ALL (<location_a>) CAMO DEGRADE ON (timeout=30s, require_write_lead=true)␣
↪TO ASYNC,
    wait_for_ready := true
);
```

The `require_write_lead=true` option prevents degradation to async unless the node is the current write leader. Without it, a non-leader node can degrade on timeout and commit locally, risking data loss if the write leader is still up and accepting writes in the same group.

CAMO has higher overhead than other commit scope kinds and requires application changes to handle transaction resolution after failover. See *Commit At Most Once* for full details.

### Setting the default commit scope

Set the commit scope as the default for each subgroup to apply it automatically to all transactions originating from that location.

```
SELECT bdr.alter_node_group_option(
    node_group_name := <location_a>,
    config_key := default_commit_scope,
    config_value := majority_protect
);

SELECT bdr.alter_node_group_option(
    node_group_name := <location_b>,
    config_key := default_commit_scope,
```

(continues on next page)

(continued from previous page)

```
    config_value := majority_protect
);
```

!!!Important Quorum Commit doesn't guarantee consistency when transactions using it execute concurrently on the same tables as transactions that don't. Setting a default commit scope on the node group ensures all transactions get the consistency benefit.

You can override the default for a specific session or transaction without changing the group configuration:

```
- - Session-level override
SET bdr.commit_scope = majority_protect;

- - Transaction-level override
BEGIN;
SET LOCAL bdr.commit_scope = camo_scope;
- - ...
COMMIT;
```

### Verifying the configuration

- Check that each subgroup has the correct default commit scope:

```
SELECT node_group_name, default_commit_scope
FROM bdr.node_group_summary;
```

- Check the commit scope rules:

```
SELECT commit_scope_name, commit_scope_origin_node_group, commit_scope_rule
FROM bdr.commit_scopes;
```

- Check the active commit scope for the current session:

```
SHOW bdr.commit_scope;
```

- If you are using CAMO, verify that the partner node is connected and ready before relying on it:

```
SELECT bdr.is_camo_partner_connected();
SELECT bdr.is_camo_partner_ready();
```

### Avoiding common pitfalls

- **Use majority protect for production workloads.** Local protect commits without waiting for any other node. A node failure before replication completes results in data loss.
- **Use local durability with lag control for high-latency connections.** Waiting for remote confirmation on every transaction causes write timeouts when cross-location latency is high.
- **Use multiple scopes for mixed workloads.** Non-critical writes shouldn't pay the same latency cost as critical ones. Override at the session or transaction level for workloads that don't need strong durability guarantees.
- **Add a witness node in a third location for two-location deployments.** Without one, losing a location breaks Raft quorum and the cluster can no longer elect a write leader.
- **Use CAMO for workloads that can't tolerate duplicate transactions.** Without it, a node failure during commit can result in the transaction being applied twice after failover.

- **Expect serialization errors with Quorum Commit.** Eager conflict resolution aborts one of any two conflicting transactions during the commit phase. Configure the application to catch serialization errors and retry.

### Controlling replication lag

Network latency between geographic locations means that replication across regions is inherently slower than within a single location. When left unmanaged, replication lag accumulates, putting your Recovery Point Objectives (RPO) at risk, consuming disk space on healthy nodes, and degrading cluster performance.

PGD provides two complementary tools for managing cross-region lag: Lag Control, which throttles write throughput to keep remote nodes from falling behind, and subscriber-only nodes, which offload regional read traffic without adding to the replication mesh.

### Using lag control

Lag Control is a commit scope kind that automatically injects a small delay into write transactions on the origin node when replication lag exceeds defined limits. This slows the rate of incoming writes, giving downstream nodes time to catch up.

The PGD commit delay starts at 0ms. If the lag on enough nodes stays within the configured limits, the delay remains at or near 0. If the limits are exceeded, the delay increases until the cluster catches up. The delay is applied after the transaction has committed and released its locks, so concurrent transactions aren't blocked.

### Configuring Lag Control

In most geo-replication deployments you already have a synchronous commit scope configured for the origin group. Add `LAG CONTROL` for the remote groups in the same rule using `AND` , so that synchronous durability applies locally while lag throttling applies cross-region:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := geo_scope_lag,
    origin_node_group := <top_group>,
    rule := ALL ORIGIN_GROUP SYNCHRONOUS COMMIT AND ANY 1 NOT ORIGIN_GROUP LAG␣
↪CONTROL (max_commit_delay=500ms, max_lag_time=30s),
    wait_for_ready := true
);
```

If you only want lag throttling without a synchronous commit requirement, use `LAG CONTROL` on both sides:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := geo_scope_lag_only,
    origin_node_group := <top_group>,
    rule := ALL ORIGIN_GROUP LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s)␣
↪AND ANY 1 NOT ORIGIN_GROUP LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s),
    wait_for_ready := true
);
```

The key parameters are:

- `max_commit_delay` : the maximum delay that can be injected into client write transactions. This is a hard ceiling; the delay never exceeds this value.
- `max_lag_time` : the acceptable replication lag expressed as time. This is a soft limit.
- `max_lag_size` : the acceptable replication lag expressed as WAL bytes. This is also a soft limit.

### When Lag Control is not enough

A commit delay that is consistently at the configured ceiling with lag still exceeding the limits indicates that write throughput is fundamentally too high for the available cross-region bandwidth. Consider:

- Increasing cross-region network bandwidth.
- Splitting large bulk load operations into smaller transactions with longer commit delays.
- Reviewing whether all tables need to be replicated cross-region.

### Configuring routing thresholds

Routing thresholds control when lagging nodes are removed from write and read routing. When a node's lag exceeds the configured threshold, Connection Manager stops routing traffic to it until it catches up. The default value for both thresholds is `-1` , which disables the check. Values are in bytes.

Set `route_writer_max_lag` to remove a node from write leader eligibility when its lag exceeds the threshold:

```
SELECT bdr.alter_node_group_option(
    node_group_name := <group_name>,
    config_key := route_writer_max_lag,
    config_value := 16777216
);
```

Set `route_reader_max_lag` to remove a node from read routing when its lag exceeds the threshold:

```
SELECT bdr.alter_node_group_option(
    node_group_name := <group_name>,
    config_key := route_reader_max_lag,
    config_value := 16777216
);
```

Check current thresholds:

```
SELECT node_group_name, route_writer_max_lag, route_reader_max_lag
FROM bdr.node_group_routing_config_summary;
```

### Tuning replication apply

Applying changes faster reduces lag under high write volumes. Configure the number of parallel apply workers (`num_writers` ), streaming mode (`streaming_mode` ), and intentional apply delay (`apply_delay` ) at the node group level. See *Replication and sequence tuning* for details.

### Monitoring replication lag

Use the following views to check current lag across the cluster.

- Check current replication rates and estimated catchup time per node:

```
SELECT target_name, replay_lag, replay_lag_bytes, apply_rate, catchup_interval
FROM bdr.node_replication_rates;
```

- Check subscription status and last applied transaction timestamp per node:

```
SELECT origin_name, target_name, subscription_status, last_xact_replay_timestamp
FROM bdr.subscription_summary;
```

- Check slot lag for detailed LSN-level information:

```
SELECT target_name, state, replay_lag_bytes, replay_lag_size
FROM bdr.node_slots
ORDER BY replay_lag_bytes DESC;
```

The accuracy of lag information depends on `bdr.replay_progress_frequency` , a configuration parameter that controls how often each node broadcasts its replication position to the rest of the cluster. The default is 60 seconds. Lower this value for more responsive lag monitoring, keeping in mind that more frequent updates add a small amount of network overhead.

### Subscriber-only nodes

If users in a region mostly read data and rarely write, adding subscriber-only nodes to that region can significantly reduce read latency without the cost of a full active location.

Subscriber-only nodes receive replicated changes from the cluster but don't originate writes and don't participate in the replication mesh or Raft consensus, making them lightweight and easy to add. See *Subscriber-only nodes and groups* for details.

### Adding a subscriber-only node

1. Create a subscriber-only subgroup under the top-level group:

```
SELECT bdr.create_node_group(
    node_group_name := <read_region>,
    parent_group_name := <top_group>,
    node_group_type := subscriber-only
);
```

1. Create a new node as a subscriber-only node:

```
SELECT bdr.create_node(
    node_name := <so_node>,
    dsn := host=<so_host> dbname=<dbname> port=5444,
    node_type := subscriber-only
);
```

1. Join the new node to the subscriber-only subgroup, using the DSN of an active data node (not another subscriber-only node) as the entry point:

```
SELECT bdr.join_node_group(
    dsn := host=<data_node_host> dbname=<dbname> port=5444,
    node_group_name := <read_region>
);
```

### Routing reads to subscriber-only nodes

Connection Manager can route read-only traffic to subscriber-only nodes. Configure your application's connection string to use the read-only endpoint of nodes in the read region.

See *Connection Manager Authentication* for details on configuring read-only routing.

### Limitations of subscriber-only nodes

- Subscriber-only nodes cannot accept writes. Attempts to write will fail.
- They don't participate in Raft consensus, so they don't count toward quorum for commit scope confirmations.
- Replication lag on subscriber-only nodes may be higher than on active nodes, depending on network conditions.

### Replication and sequence tuning

Geo-distributed clusters replicate changes across high-latency network links, which makes replication throughput and sequence uniqueness more important than in single-location setups. Configure these settings at the node group level to control how changes are applied on remote nodes and how sequences generate unique values across locations.

### Parallel Apply

*Parallel Apply* allows each node to apply incoming changes using multiple writer processes concurrently rather than sequentially, reducing apply lag under high write volumes. It's enabled by default with `bdr.writers_per_subscription` set to two writers per node. Use `num_writers` at the node group level to set the number of writers per subgroup, within the limit set by `bdr.max_writers_per_subscription`.

1. Increase throughput for your geo-distributed cluster by modifying the number of `num_writers` per location subgroup. Higher values improve throughput but increase memory usage. Repeat for each location subgroup.

```
SELECT bdr.alter_node_group_option(
    node_group_name := <location_a>,
    config_key := num_writers,
    config_value := 4
);
```

1. Set `streaming_mode` to `writer` so that large transactions are sent directly to a writer process rather than being written to disk first, reducing I/O overhead on the receiving node:

```
SELECT bdr.alter_node_group_option(
    node_group_name := <location_a>,
    config_key := streaming_mode,
    config_value := writer
);
```

The default is `auto`, which lets PGD choose between `file` and `writer` based on available resources. Setting `writer` explicitly is recommended for geo-distributed clusters with consistently high write volumes. The trade-off is additional resource usage. In `writer` mode, each large transaction requires a dedicated writer process. In `file` mode, PGD writes to temporary disk files before applying.

`streaming_mode = 'writer'` is incompatible with *Commit At Most Once*. If your cluster uses CAMO, leave `streaming_mode` at its default `auto`.

### Apply delay

Use `apply_delay` on subscriber-only subgroups to create a lagging read replica, giving you a recovery window against accidental data loss or logical corruption. The delay is measured from the transaction's original commit time at the origin. The default is `0s` (no delay).

```
SELECT bdr.alter_node_group_option(
    node_group_name := <delayed_read_replica_group>,
    config_key := apply_delay,
```

(continues on next page)

(continued from previous page)

```
    config_value := 30min
);
```

> **Note**
>
> Don't set *apply_delay* on data node groups or the top-level group, as doing so introduces intentional replication lag into your active cluster.

### Sequences

Standard Postgres sequences are local to each node and cause primary key conflicts when multiple locations write simultaneously. Configure your geo-distributed cluster to use PGD global sequences, which guarantee uniqueness across all nodes. The `bdr.default_sequence_kind` configuration parameter controls which type applies to newly created sequences and serial columns:

- `snowflakeid` — generates unique values without inter-node communication, using a timestamp-based algorithm. Works only for 64-bit `bigint` or `bigserial` columns. Resilient to network partitions and cross-region latency.
- `galloc` — allocates ranges of values from a cluster-wide counter, requiring quorum when a new range is needed. Supports `smallint` , `integer` , and `bigint` , but is sensitive to quorum availability.
- `distributed` (default) — automatically uses `snowflakeid` for `bigint` sequences and `galloc` for `smallint` and `integer` sequences.

For geo-distributed cluster, we recommend using the default `distributed` setting. It avoids the quorum dependency of `galloc` for large sequences where possible, while still supporting smaller integer types.

> **Note**
>
> Avoid setting *bdr.default_sequence_kind = local* in a geo-distributed cluster with multiple write locations. Local sequences do not guarantee cluster-wide uniqueness and cause insert conflicts in multi-master replication.

See *Sequences* for full details on global sequence types and behavior.

### Performing cluster maintenance

Verify your cluster configuration before going to production, and plan ongoing maintenance carefully. Operations that are straightforward in a single-location cluster can trigger cross-location failovers, introduce replication lag, or cause conflicts when multiple locations are active.

### Verifying the cluster configuration

After completing setup, verify that subgroups, routing, and commit scopes are configured correctly before going to production.

- Check subgroups and their routing settings:

```
SELECT node_group_name, node_group_enable_raft, node_group_enable_routing
FROM bdr.node_group;
```

- Check current write leaders:

```
SELECT node_group_name, write_lead
FROM bdr.node_group_routing_summary;
```

- Check commit scope configuration:

```
SELECT commit_scope_name, commit_scope_origin_node_group, commit_scope_rule
FROM bdr.commit_scopes;
```

### Managing database changes

Schema changes and routine maintenance behave differently in a geo-distributed cluster. DDL acquires cluster-wide locks, while operations like `VACUUM` and `ANALYZE` run locally on each node but can be affected by replication slot lag.

### Running DDL

DDL operations acquire cluster-wide locks, but the lock type and timing differ by operation. Schema changes such as `ALTER TABLE` and `CREATE INDEX` acquire global DDL locks simultaneously across all nodes, with lock acquisition time approximately equal to the round-trip time to the most distant node. Operations such as `DROP TABLE` acquire a global DML lock that also flushes replication, with completion time depending on current replication lag rather than network latency alone.

In both cases, long-running DDL blocks replication while the lock is held, and failed DDL may leave an inconsistent state across locations.

To reduce risk, keep DDL transactions minimal and avoid combining multiple DDL changes in a single transaction. Test DDL timing in a staging environment that simulates your actual cross-location latency. Schedule DDL during maintenance windows.

### Running VACUUM and ANALYZE

`VACUUM` and `ANALYZE` run locally on each node and are not replicated. Autovacuum operates independently on each node, so no special coordination is needed across locations.

The PGD-specific concern is replication slot lag. Slots for offline or lagging nodes hold back `catalog_xmin` , preventing VACUUM from cleaning catalog tables and causing bloat to accumulate on the origin node. Check slot lag regularly to catch this early:

```
SELECT target_name, state, replay_lag_bytes, replay_lag_size, xmin, catalog_xmin
FROM bdr.node_slots
ORDER BY replay_lag_bytes DESC;
```

### Managing write leadership

In a geo-distributed cluster, write leadership can move between locations due to node failure or planned operations. Configure failover detection to match your latency, maintain quorum before taking nodes offline, and use planned switchovers to transfer write leadership safely.

### Handling failover

When a write leader failure occurs, PGD detects it through TCP keepalives and connection timeouts, then elects a new write leader via Raft consensus. Tune the following parameters to match your cross-location latency and acceptable failover time:

- `bdr.global_keepalives_idle` , `bdr.global_keepalives_interval` , and `bdr.global_keepalives_count` control how quickly a failed connection is detected.

- `bdr.global_connection_timeout` sets the maximum time to wait when establishing a connection. In high-latency deployments, lower this value to detect failures faster without waiting for the full TCP timeout.
- `bdr.raft_global_election_timeout` controls how long before a new write leader is elected at the top-level group. `bdr.raft_group_election_timeout` applies to per-location subgroup elections. Increase these values if expected cross-location latency is causing false elections from network jitter.

### Maintaining quorum

Verify that your cluster maintains quorum (a majority of voting nodes) before taking any node offline. Without quorum, no new write leader is elected, the cluster enters a read-only state, and manual intervention is required. Check the current number of voting nodes before proceeding:

```
SELECT nvoting_nodes, is_voting FROM bdr.stat_raft_state;
```

Don't fence or stop nodes if doing so reduces the number of remaining voting nodes below a majority. In a two-location deployment, a witness node in a third location provides the extra vote needed to maintain quorum if one location fails. See *Planning your node distribution* for details.

### Performing a planned switchover

Before a planned switchover, verify that the target node is healthy and caught up. Ensure no long-running transactions are active on the current leader, and perform the switchover during a low-traffic period where possible.

1. Check that the target node's replication slot is active, streaming, and close to zero lag:

```
SELECT target_name, active, state, replay_lag_bytes, replay_lag_size
FROM bdr.node_slots
WHERE target_name = <target_node_name>;
```

The slot should show `active = true` , `state = 'streaming'` , and `replay_lag_bytes` close to zero. If the state is `catchup` or `disconnected` , or lag is high, wait before proceeding.

1. Initiate the switchover:

```
SELECT bdr.routing_leadership_transfer(
    node_group_name := <group_name>,
    leader_name := <target_node_name>
);
```

1. Confirm the cluster is in a healthy state:
2. Confirm the new write leader is elected:

```
SELECT node_group_name, write_lead
FROM bdr.node_group_routing_summary;
```

1. Confirm proxy routing has updated to the new leader:

```
SELECT node_group_name, write_lead_name, previous_write_lead_name
FROM bdr.stat_routing_state;
```

1. Confirm replication from the new leader to other nodes is streaming with low lag:

```
SELECT target_name, active, state, replay_lag_bytes, replay_lag_size
FROM bdr.node_slots
WHERE origin_name = <new_leader_node_name>;
```

All slots should show `state = 'streaming'` and `replay_lag_bytes` close to zero.

### Performing node maintenance

Follow a location-aware order for rolling operations and fence nodes before taking them offline.

### Rolling operations across locations

When performing upgrades or maintenance that affects multiple nodes, follow this order to minimize cross-location failovers and keep writes local as long as possible. This order assumes local routing, where each location has its own write leader. With global routing, there is a single write leader for the entire cluster, so skip step 2 and handle the global write leader last.

1. Start with non-leader nodes in the secondary location.
2. Move to the write leader in the secondary location (doing so causes a local failover within that location). Skip step 2 if using global routing.
3. Proceed to non-leader nodes in the primary location.
4. Handle the primary write leader last. Taking down the primary write leader is the operation most likely to cause a cross-location failover.

### Fencing nodes

Fencing removes a node from write leader eligibility without removing it from the cluster. Use fencing before any maintenance requiring a node restart, before upgrades, and when node health is questionable.

To fence a node:

```
SELECT bdr.alter_node_option(
    node_name := <node_name>,
    config_key := route_fence,
    config_value := true
);
```

A fenced node still participates in quorum and continues receiving replication, so it does not affect the cluster's ability to maintain consensus. However, fencing multiple nodes at the same time can reduce availability. Always unfence after maintenance completes.

To unfence:

```
SELECT bdr.alter_node_option(
    node_name := <node_name>,
    config_key := route_fence,
    config_value := false
);
```

### Monitoring the cluster and responding to incidents

Check replication slot health and conflict rates regularly as part of operations. Use the views below to assess impact when incidents occur and to verify recovery.

### Handling network partitions

When locations become partitioned, the side that retains quorum continues operating. The side without quorum becomes read-only. Replication slots on the isolated side pause and lag grows for the duration of the partition.

After the partition heals, verify recovery before resuming operations:

1. Monitor catchup progress using lag metrics in `bdr.node_slots` and wait for all slots to return to `streaming`.
2. Check conflict rates for the partition window. See *Monitoring conflicts*.

### Monitoring conflicts

Geo-replication increases conflict potential because the same row can be updated in different locations simultaneously. By default, PGD resolves conflicts using last-update-wins. Timestamp-based resolvers depend on clock synchronization across nodes. Keep clock skew minimal by running NTP or chronyd on all nodes.

Check `bdr.conflict_history_summary` regularly for tables with high conflict rates:

```
SELECT nspname, relname, conflict_type, count(*)
FROM bdr.conflict_history_summary
GROUP BY nspname, relname, conflict_type
ORDER BY count(*) DESC;
```

A persistent high conflict rate on a specific table points to an access pattern worth revisiting with the application team. Persistent conflicts are usually a design issue. Location-based data partitioning, append-only patterns, or switching to CRDT data types are more effective long-term than adjusting conflict resolution.

### Configuring conflict resolvers

If changing the access pattern isn't feasible and last-update-wins produces the wrong outcome for a specific table, configure a custom resolver using `bdr.alter_node_set_conflict_resolver()`. The available strategies are `update_if_newer` (default), `skip`, and `error`. The function call isn't replicated, so run it on each node in the cluster. Check the node name first using `SELECT node_name FROM bdr.node_summary`:

```
SELECT bdr.alter_node_set_conflict_resolver(
    node_name         => node1,
    conflict_type     => update_update,
    conflict_resolver => error
);
```

To verify the current resolver configuration across all conflict types:

```
SELECT * FROM bdr.node_conflict_resolvers;
```

See *Missing-column conflict resolution* for the full list of conflict types and resolvers.

## 9.3 Using connection pooling in your application

Adapt your application to work with *Connection Manager Authentication*'s connection pooling.

For DBA guidance on enabling and operating pooling, see *Configuring connection pooling*.

### 9.3.1 Understanding the active pool mode

Query `bdr.node_group_summary` to see the active pool mode.

```
SELECT node_group_name, server_pool_mode FROM bdr.node_group_summary;
```

The pool mode determines what session state persists between transactions.

| Feature | *none*/*session* | *transaction* |
|---|---|---|
| *SET* changes persist | Yes | No |
| *PREPARE*/*EXECUTE* statements persist | Yes | No |
| Temporary tables persist | Yes | No |
| Advisory locks across transactions | Yes | No |
| Holdable cursors (*WITH HOLD*) | Yes | No |
| *LISTEN* subscriptions | Yes | No |

In `none` and `session` modes, your application behaves the same as with a direct Postgres connection for most purposes. The exception is that Connection Manager forwards only a specific set of connection parameters to the backend. See *Managing session parameters* for details. The difference between `none` and `session` is at the infrastructure level: in `session` mode, backend connections are reused after clients disconnect, reducing server overhead, but this is invisible to the application.

In `transaction` mode, the backend connection is returned to the pool when each transaction ends and most session state is cleared. The next transaction can run on a different backend. Some Postgres features rely on session state that doesn't persist between transactions and don't work as a result. For a full list, see *Connection pooling's unsupported features* .

If your application depends on any of these features, ask your DBA to switch to `session` or `none` mode (see *Configuring connection pooling* ), or redesign the affected code paths to keep all state within a single transaction.

### 9.3.2 Adapting your application for transaction mode

Some application patterns that work with a direct Postgres connection need adjustment in transaction mode.

#### Managing session parameters

Connection Manager forwards a specific set of connection parameters to the backend and re-applies them on each new backend assignment in transaction mode. The parameters it recognizes are `client_encoding` , `DateStyle` , `TimeZone` , `standard_conforming_strings` , `application_name` , `search_path` , and `extra_float_digits` .

Set these in the connection string rather than with `SET` commands, so they apply to every backend the pool assigns.

```
host=myhost dbname=mydb options=-c search_path=myschema -c work_mem=256MB
```

Or in a URI.

```
postgresql://myhost/mydb?options=-c%20search_path%3Dmyschema%20-c%20work_mem%3D256MB
```

Parameters outside the recognized list aren't re-applied on backend changes unless passed via the `options` connection parameter.

### Using prepared statements

Connection Manager automatically detects missing prepared statements on the backend and re-prepares them on demand, so prepared statements sent via the extended query protocol work seamlessly across transactions. Most client libraries use the extended query protocol by default.

`DISCARD ALL` runs when the backend connection is returned to the pool at the end of each transaction, deallocating any prepared statements created with `PREPARE` /`EXECUTE` SQL statements. To avoid errors with these, follow these practices.

- Avoid `PREPARE` , `EXECUTE` , and `DEALLOCATE` SQL statements. Use the extended query protocol instead, which most client libraries do by default.
- If you need to use `PREPARE` /`EXECUTE` /`DEALLOCATE` SQL statements, issue them inside the transaction where you use them.

## 9.4 Application use

Developing an application with PGD is mostly the same as working with any PostgreSQL database. What's different, though, is that you need to be aware of how your application interacts with replication. You need to know how PGD behaves with applications, the SQL that is and isn't replicated, how different nodes are handled, and other important information.

- ```
  :ref:`DML and DDL replication and nonreplication <DML and DDL replication and␣
  ↪nonreplication>`  shows the differences between the two classes of SQL␣
  ↪statements and how PGD handles replicating them. It also looks at the commands␣
  ↪PGD doesn't replicate at all.
  ```
- ```
  :ref:`Nodes with differences <Nodes with differences>`  examines how PGD works␣
  ↪with configurations where there are differing table structures and schemas on␣
  ↪replicated nodes. Also covered is how to compare between such nodes with␣
  ↪LiveCompare and how differences in PostgreSQL versions running on nodes can be␣
  ↪handled.
  ```
- ```
  :ref:`Internal timing of operations <Internal timing of operations>`  shows how␣
  ↪the asynchronous/synchronous replication might affect an application's view of␣
  ↪data and notes functions to mitigate stale reads.
  ```
- ```
  :ref:`Using extensions with PGD <Using extensions with PGD>`  explains how to␣
  ↪select, install, and configure extensions on PGD.
  ```
- ```
  :ref:`Use of table access methods (TAMs) in PGD <Use of table access methods␣
  ↪(TAMs) in PGD>`  (TAMs) notes the TAMs available with PGD and how to enable␣
  ↪them.
  ```
- ```
  :ref:`Feature compatibility <Feature compatibility>`  shows which server features␣
  ↪work with which commit scopes and which commit scopes can be daisy chained␣
  ↪together.
  ```

### 9.4.1 DML and DDL replication and nonreplication

The two major classes of SQL statement are DML and DDL.

- DML is the data modification language and is concerned with the SQL statements that modify the data stored in tables. It includes UPDATE, DELETE, and INSERT.
- DDL is the data definition language and is concerned with the SQL statements that modify how the data is stored. It includes CREATE, ALTER, and DROP.

PGD handles each class differently.

#### DML replication

PGD doesn't replicate the DML statement. It replicates the changes caused by the DML statement. For example, an UPDATE that changed two rows replicates two changes, whereas a DELETE that didn't remove any rows doesn't replicate anything. This means that the results of executing volatile statements are replicated, ensuring there's no divergence between nodes as might occur with statement-based replication.

#### DDL replication

DDL replication works differently from DML. For DDL, PGD replicates the statement, which then executes on all nodes. So a `DROP TABLE IF EXISTS` might not replicate anything on the local node, but the statement is still sent to other nodes for execution if DDL replication is enabled. For details, see *Running DDL safely* .

PGD works to ensure that intermixed DML and DDL statements work correctly, even in the same transaction.

#### Nonreplicated statements

Outside of those two classes are SQL commands that PGD, by design, doesn't replicate. None of the following user commands are replicated by PGD, so their effects occur on the local/origin node only:

- Cursor operations (DECLARE, CLOSE, FETCH)
- Execution commands (DO, CALL, PREPARE, EXECUTE, EXPLAIN)
- Session management (DEALLOCATE, DISCARD, LOAD)
- Parameter commands (SET, SHOW)
- Constraint manipulation (SET CONSTRAINTS)
- Locking commands (LOCK)
- Table maintenance commands (VACUUM, ANALYZE, CLUSTER)
- Async operations (NOTIFY, LISTEN, UNLISTEN)

Since the `NOTIFY` SQL command and the `pg_notify()` functions aren't replicated, notifications aren't reliable in case of failover. This means that notifications can easily be lost at failover if a transaction is committed just when the server crashes. Applications running `LISTEN` might miss notifications in case of failover.

This is true in standard PostgreSQL replication, and PGD doesn't yet improve on this.

CAMO and Eager Replication options don't allow the `NOTIFY` SQL command or the `pg_notify()` function.

### 9.4.2 Nodes with differences

#### Replicating between nodes with differences

By default, DDL is sent to all nodes. You can control this behavior, as described in *Running DDL safely* , and you can use it to create differences between database schemas across nodes. PGD is designed to allow replication to continue

even with minor differences between nodes. These features are designed to allow application schema migration without downtime or to allow logical standby nodes for reporting or testing.

Currently, replication requires the same table name on all nodes. A future feature might allow a mapping between different table names.

It's possible to replicate between tables with dissimilar partitioning definitions, such as a source that's a normal table replicating to a partitioned table, including support for updates that change partitions on the target. It can be faster if the partitioning definition is the same on the source and target since dynamic partition routing doesn't need to execute at apply time. For details, see *Replication sets* .

By default, all columns are replicated.

PGD replicates data columns based on the column name. If a column has the same name but a different data type, PGD attempts to cast from the source type to the target type, if casts were defined that allow that.

PGD supports replicating between tables that have a different number of columns.

If the target has missing columns from the source, then PGD raises a `target_column_missing` conflict, for which the default conflict resolver is `ignore_if_null` . This throws an error if a non-NULL value arrives. Alternatively, you can also configure a node with a conflict resolver of `ignore` . This setting doesn't throw an error but silently ignores any additional columns.

If the target has additional columns not seen in the source record, then PGD raises a `source_column_missing` conflict, for which the default conflict resolver is `use_default_value` . Replication proceeds if the additional columns have a default, either NULL (if nullable) or a default expression. If not, it throws an error and halts replication.

Transform triggers can also be used on tables to provide default values or alter the incoming data in various ways before apply.

If the source and the target have different constraints, then replication is attempted, but it might fail if the rows from source can't be applied to the target. Row filters can help here.

Replicating data from one schema to a more relaxed schema doesn't cause failures. Replicating data from a schema to a more restrictive schema can be a source of potential failures. The right way to solve this is to place a constraint on the more relaxed side, so bad data can't be entered. That way, no bad data ever arrives by replication, so it never fails the transform into the more restrictive schema. For example, if one schema has a column of type TEXT and another schema defines the same column as XML, add a CHECK constraint onto the TEXT column to enforce that the text is XML.

You can define a table with different indexes on each node. By default, the index definitions are replicated. To specify how to create an index on only a subset of nodes or just locally, see *Running DDL safely* .

Storage parameters, such as `fillfactor` and `toast_tuple_target` , can differ between nodes for a table without problems. An exception to that behavior is that the value of a table's storage parameter `user_catalog_table` must be identical on all nodes.

A table being replicated must be owned by the same user/role on each node. See *Security and roles* for details.

Roles can have different passwords for connection on each node, although by default changes to roles are replicated to each node. See *Running DDL safely* to specify how to alter a role password on only a subset of nodes or locally.

### Comparison between nodes with differences

LiveCompare is a tool for data comparison on a database against PGD and non-PGD nodes. It needs a minimum of two connections to compare against and reach a final result.

Starting with LiveCompare 1.3, you can configure with `all_bdr_nodes` set. This setting saves you from clarifying all the relevant DSNs for each separate node in the cluster. An EDB Postgres Distributed cluster has N amount of nodes with connection information, but it's only the initial and output connection that LiveCompare 1.3 and later needs to complete its job. Setting `logical_replication_mode` states how all the nodes are communicating.

All the configuration is done in a `.ini` file named `bdrLC.ini` , for example. Find templates for this configuration file in `/etc/2ndq-livecompare/` .

While LiveCompare executes, you see N+1 progress bars, N being the number of processes. Once all the tables are sourced, a time displays as the transactions per second (tps) was measured. This mechanism continues to count the time, giving you an estimate and then a total execution time at the end.

This tool offers a lot of customization and filters, such as tables, schemas, and replication_sets. LiveCompare can use stop-start without losing context information, so it can run at convenient times. After the comparison, a summary and a DML script are generated so you can review it. Apply the DML to fix any differences found.

#### Replicating between different release levels

The other difference between nodes that you might encounter is where there are different major versions of PostgreSQL on the nodes. PGD is designed to replicate between different major release versions. This feature is designed to allow major version upgrades without downtime.

PGD is also designed to replicate between nodes that have different versions of PGD software. This feature is designed to allow version upgrades and maintenance without downtime.

However, while it's possible to join a node with a major version in a cluster, you can't add a node with a minor version if the cluster uses a newer protocol version. Doing so returns an error.

Both of these features might be affected by specific restrictions. See Releasenotes for any known incompatibilities.

### 9.4.3 Timing considerations and synchronous replication

Being asynchronous by default, peer nodes might lag behind. This behavior makes it possible for a client connected to multiple PGD nodes or switching between them to read stale data.

A *queue waitfunction* is provided for clients or proxies to prevent such stale reads.

The synchronous replication features of Postgres are available to PGD as well. In addition, PGD provides multiple variants for more synchronous replication. See *Commit scopes* for an overview and comparison of all variants available and their different modes.

### 9.4.4 Using extensions with PGD

#### PGD and other PostgreSQL extensions

PGD is implemented as a PostgreSQL extension (see Supported Postgres database servers ). It takes advantage of PostgreSQL's expandability and flexibility to modify low-level system behavior to provide multi-master replication.

In principle, extensions provided by community PostgreSQL, EDB Postgres Advanced Server, and third-party extensions can be used with PGD. However, the distributed nature of PGD means that you need to carefully consider and plan the extensions you select and install.

#### Extensions providing logical decoding

Extensions providing logical decoding, such as wal2json , may in theory work with PGD. However, there's no support for failover, meaning any WAL stream being read from such an extension can be interrupted.

#### Extensions providing replication or HA functionality

Any extension extending PostgreSQL with functionality related to replication or HA/failover is unlikely to work well with PGD and may even be detrimental to the health of the PGD cluster. We recommend avoiding these.

### Supported extensions

These extensions are explicitly supported by PGD.

### EDB Advanced Storage table access methods

The *EDB Advanced Storage Pack* provides a selection of table access methods (TAMs) implemented as extensions. The following TAMs are certified for use with PGD:

- `:ref:`Autocluster <Using extensions with PGD>`
- `:ref:`Refdata <Using extensions with PGD>`

For more details, see *Use of table access methods (TAMs) in PGD* .

### pgaudit

PGD was modified to ensure compatibility with the pgaudit extension. See *Postgres settings* for configuration information.

### Installing extensions

PostgreSQL extensions provide SQL objects, such as functions, datatypes, and, optionally, one or more shared libraries. These must be loaded into the PostgreSQL backend before you can install and use the extension.

> **Warning**
>
> The relevant extension packages must be available on all nodes in the cluster. Otherwise extension installation can fail and impact cluster stability.

If PGD is deployed using Trusted Postgres Architect , configure extensions using that tool. For details, see Adding Postgres extensions .

The following is relevant for manually configured PGD installations.

### Configuring shared_preload_libraries

If an extension provides a shared library, include this library in the shared_preload_libraries configuration parameter before installing the extension.

`shared_preload_libraries` consists of a comma-separated list of extension names. It must include `bdr` . The order in which you specify other extensions generally doesn't matter.

Configure `shared_preload_libraries` on all nodes in the cluster before installing the extension with `CREATE EXTENSION` . You must restart PostgreSQL to activate the new configuration.

See also *Postgres settings* .

### Installing the extension

Install the extension using the `CREATE EXTENSION` command. You need to do this on only one node in the cluster. PGD's DDL replication will ensure that it propagates to all other nodes.

**Warning**

Do not attempt to install extensions manually on each node by, for example, disabling DDL replication before executing *CREATE EXTENSION* .

Do not use a command such as `bdr.replicate_ddl_command()` to execute `CREATE EXTENSION` .

### 9.4.5 Use of table access methods (TAMs) in PGD

The *EDB Advanced Storage Pack* provides a selection of table access methods (TAMs), available from EDB Postgres 15.0.

The following TAMs were certified for use with PGD 6.0:

- *Autocluster*
- *Refdata*

Usage of any other TAM is restricted until certified by EDB.

To use one of these TAMs on a PGD cluster, the appropriate extension library (`autocluster` and/or `refdata` ) must be added to the `shared_preload_libraries` parameter on each node, and the PostgreSQL server restarted.

Once the extension library is present in `shared_preload_libraries` on all nodes in the cluster, the extension itself can be created with `CREATE EXTENSION autocluster;` or `CREATE EXTENSION refdata;` . The `CREATE EXTENSION` command only needs to be executed on one node; it will be replicated to the other nodes in the cluster.

After you create the extension, use `CREATE TABLE test USING autocluster;` or `CREATE TABLE test USING refdata;` to create a table with the specified TAM. These commands replicate to all PGD nodes in the cluster.

For more information on these table access methods, see:

- *Autocluster example*
- *Refdata example*

### 9.4.6 Feature compatibility

#### Server feature/commit scope interoperability

Not all server features work with all commit scopes. This table shows the ones that interoperate.

| | Async | Parallel Apply | Transaction Streaming | Single Decoding Worker |
|---|---|---|---|---|
| *Group Commit* | N/A | Does not interoperate | Does not interoperate Attempting to use Group Commit and Transaction | Interoperates |
| *CAMO* | N/A | Interoperates | Does not interoperate | Does not interoperate |
| *Lag Control* | Interoperates | Interoperates | Interoperates | Interoperates |
| *Synchronous Commit* | N/A | Interoperates | Interoperates | Interoperates |

### Notes

N/A : The Async column in the table represents PGD without a synchronous commit scope in use. Lag Control isn't a synchronous commit scope. It's a controlling commit scope and is therefore available with asynchronous operations.

Attempting to use Group Commit and Transaction Streaming presents a warning. The warning suggests that you disable transaction streaming, and the transaction appears to take place. In the background, Group Commit was disabled to allow the transaction to occur.

### Commit scope/commit scope interoperability

Although you can't mix commit scopes, you can *combine rules* with an `AND` operator. This table shows where commit scopes can be combined.

| | *Group Commit* | *CAMO* | *Lag Control* | *Synchronous Commit* |
|---|---|---|---|---|
| *Group Commit* | N/A | Does not combine | Combines | Combines |
| *CAMO* | Does not combine | N/A | Combines | Does not combine |
| *Lag Control* | Combines | Combines | N/A | Combines |
| *Synchronous Commit* | Combines | Does not combine | Combines | N/A |

### Notes

Each commit scope implicitly works with itself.

CHAPTER 10

# Reference

## 10.1 PGD Reference

The reference section is a definitive listing of all functions, views, and commands available in EDB Postgres Distributed.

- 

Tables, Views, and Functions

- ```
  `Command Line Interface (CLI) <https://www.enterprisedb.com/docs/pgd/latest/
  ↪reference/cli>`_
  ```

### 10.1.1 Tables, views and functions reference

The reference section is a definitive listing of all functions, views, and commands available in EDB Postgres Distributed.

#### User visible catalogs and views

- *bdr.camo_decision_journal*
- *bdr.commit_decisions*
- *bdr.commit_scopes*
- *bdr.conflict_history*
- *bdr.conflict_history_summary*
- *bdr.consensus_kv_data*
- *bdr.crdt_handlers*
- *bdr.ddl_replication*
- *bdr.depend*
  ```

- *bdr.failover_replication_slots*
- *bdr.global_consensus_journal*
- *bdr.global_consensus_journal_details*
- *bdr.global_consensus_response_journal*
- *bdr.global_lock*
- *bdr.global_locks*
- *bdr.group_camo_details*
- *bdr.group_lease*
- *bdr.group_raft_details*
- *bdr.group_replslots_details*
- *bdr.group_subscription_summary*
- *bdr.group_versions_details*
- *bdr.leader*
- *bdr.local_consensus_snapshot*
- *bdr.local_consensus_state*
- *bdr.local_node*
- *bdr.local_node_summary*
- *bdr.local_sync_status*
- *bdr.node_config*
- *bdr.node_catchup_info*
- *bdr.node_catchup_info_details*
- *bdr.node_conflict_resolvers*
- *bdr.node_group_config*
- *bdr.node_group_replication_sets*
- *bdr.node_group_summary*
- *bdr.node_local_info*
- *bdr.node_log_config*
- *bdr.node_peer_progress_mapping*
- *bdr.node_replication_rates*
- *bdr.node_slots*
- *bdr.node_summary*
- *bdr.parted_origin_catchup_info*
- *bdr.parted_origin_catchup_info_details*
- *bdr.pending_failed_changes*
- *bdr.pending_failed_changes_sql*
- *bdr.pgd_largeobject*

- *bdr.pgd_largeobject_metadata*
- *bdr.queue*
- *bdr.replication_set*
- *bdr.replication_set_table*
- *bdr.replication_set_ddl*
- *bdr.replication_sets*
- *bdr.resolved_transactions*
- *bdr.resolved_transactions_info*
- *bdr.schema_changes*
- *bdr.sequence_alloc*
- *bdr.sequences*
- *bdr.stat_activity*
- *bdr.stat_commit_scope*
- *bdr.stat_commit_scope_state*
- *bdr.stat_connection_manager*
- *bdr.stat_connection_manager_connections*
- *bdr.stat_connection_manager_node_stats*
- *bdr.stat_connection_manager_hba_file_rules*
- *bdr.stat_raft_followers_state*
- *bdr.stat_raft_state*
- *bdr.stat_receiver*
- *bdr.stat_receiver_transactions*
- *bdr.stat_reconcile_commit_scope*
- *bdr.stat_relation*
- *bdr.stat_routing_candidate_state*
- *bdr.stat_routing_state*
- *bdr.stat_subscription*
- *bdr.stat_worker*
- *bdr.stat_writer*
- *bdr.subscription*
- *bdr.subscription_summary*
- *bdr.subscription_error_status*
- *bdr.tables*
- *bdr.taskmgr_work_queue*
- *bdr.taskmgr_workitem_status*
- *bdr.taskmgr_local_work_queue*

- *bdr.taskmgr_local_workitem_status*
- *bdr.trigger*
- *bdr.triggers*
- *bdr.workers*
- *bdr.writers*
- *bdr.worker_tasks*

### System functions

#### Version information functions

- *bdr.bdr_version*
- *bdr.bdr_version_num*

#### System information functions

- *bdr.get_relation_stats*
- *bdr.get_subscription_stats*

#### System and progress information parameters

- *bdr.local_node_id*
- *bdr.last_committed_lsn*
- *transaction_id*

#### Node status functions

- *bdr.is_node_connected*
- *bdr.is_node_ready*

#### Consensus function

- *bdr.consensus_disable*
- *bdr.consensus_enable*
- *bdr.consensus_proto_version*
- *bdr.consensus_snapshot_export*
- *bdr.consensus_snapshot_import*
- *bdr.consensus_snapshot_verify*
- *bdr.get_consensus_status*
- *bdr.get_raft_status*
- *bdr.raft_leadership_transfer*

### Utility functions

- *bdr.transfer_all_leaderships*
- *bdr.alter_node_kind*
- *bdr.alter_subscription_skip_changes_upto*
- *bdr.connection_manager_refresh_pools*
- *bdr.get_node_sub_apply_lsn*
- *bdr.get_node_sub_receive_lsn*
- *bdr.global_lock_table*
- *bdr.group_lease_override*
- *bdr.local_group_slot_name*
- *bdr.node_group_type*
- *bdr.replicate_ddl_command*
- *bdr.run_on_all_nodes*
- *bdr.run_on_group*
- *bdr.run_on_nodes*
- *bdr.stat_reset*
- *bdr.wait_for_apply_queue*
- *bdr.wait_for_xid_progress*
- *bdr.wait_node_confirm_lsn*
- *bdr.wait_slot_confirm_lsn*

### Global advisory locks

- *bdr.global_advisory_lock*
- *bdr.global_advisory_unlock*

### Monitoring functions

- *bdr.monitor_group_versions*
- *bdr.monitor_group_raft*
- *bdr.monitor_local_replslots*
- *bdr.wal_sender_stats*
- *bdr.get_decoding_worker_stat*
- *bdr.lag_control*

### Routing functions

- *bdr.routing_leadership_transfer*

### CAMO functions

- *bdr.is_camo_partner_connected*
- *bdr.is_camo_partner_ready*
- *bdr.get_configured_camo_partner*
- *bdr.wait_for_camo_partner_queue*
- *bdr.camo_transactions_resolved*
- *bdr.logical_transaction_status*

### Commit Scope functions

- *bdr.add_commit_scope*
- *bdr.create_commit_scope*
- *bdr.alter_commit_scope*
- *bdr.drop_commit_scope*
- *bdr.remove_commit_scope*

### Apply error policy functions

- *bdr.skip_failed_change*
- *bdr.skip_failed_transaction*
- *bdr.apply_failed_change*
- *bdr.apply_failed_transaction*
- *bdr.modify_failed_change*
- *bdr.reconstruct_change_sql*
- *bdr.reconstruct_failed_transaction*
- *bdr.resolve_failed_transaction*
- *bdr.analyze_skip_impact*
- *bdr.analyze_transaction_skip_impact*
- *bdr.format_skip_analysis*
- *bdr.alter_subscription_reset_error_counters*

### PGD settings

### CRDTs vs conflict handling/reporting

- *bdr.default_conflict_detection*

### Global sequence parameters

- *bdr.default_sequence_kind*

### DDL handling

- *bdr.default_replica_identity*
- *bdr.ddl_replication*
- *bdr.role_replication*
- *bdr.ddl_locking*
- *bdr.truncate_locking*

### Global locking

- *bdr.global_lock_max_locks*
- *bdr.global_lock_timeout*
- *bdr.global_lock_statement_timeout*
- *bdr.global_lock_idle_timeout*
- *bdr.lock_table_locking*
- *bdr.predictive_checks*

### Node management

- *bdr.replay_progress_frequency*

### Generic replication

- *bdr.writers_per_subscription*
- *bdr.max_writers_per_subscription*
- *bdr.xact_replication*
- *bdr.permit_unsafe_commands*
- *bdr.batch_inserts*
- *bdr.maximum_clock_skew*
- *bdr.maximum_clock_skew_action*
- *bdr.accept_connections*
- *bdr.writer_input_queue_size*
- *bdr.writer_output_queue_size*
- *bdr.min_worker_backoff_delay*

### CRDTs Overview

- *bdr.crdt_raw_value*

### Replicated large objects

- *bdr.override_large_objects*

### Commit scope rules

- *bdr.commit_scopes*

### Commit At Most Once

- *bdr.camo_local_mode_delay*
- *bdr.camo_enable_client_warnings*

### Using with transaction streaming

- *Node configuration using bdr.default_streaming_mode*

### LAG CONTROL

- *bdr.lag_control_max_commit_delay*
- *bdr.lag_control_max_lag_size*
- *bdr.lag_control_max_lag_time*
- *bdr.lag_control_min_conforming_nodes*
- *bdr.lag_control_commit_delay_adjust*
- *bdr.lag_control_sample_interval*
- *bdr.lag_control_commit_delay_start*

### Monitoring and logging

- *bdr.debug_level*
- *bdr.trace_level*
- *bdr.track_subscription_apply*
- *bdr.track_relation_apply*
- *bdr.track_apply_lock_timing*

### Decoding worker

- *bdr.enable_wal_decoder*
- *bdr.receive_lcr*
- *bdr.lcr_cleanup_interval*

### Connectivity settings

- *bdr.global_connection_timeout*
- *bdr.global_keepalives*
- *bdr.global_keepalives_idle*
- *bdr.global_keepalives_interval*
- *bdr.global_keepalives_count*
- *bdr.global_tcp_user_timeout*

### Topology settings

- *bdr.force_full_mesh*

### Analytics settings

- *bdr.prefer_analytics_engine*

### Internal settings - Raft timeouts

- *bdr.raft_global_election_timeout*
- *bdr.raft_group_election_timeout*
- *bdr.raft_response_timeout*

### Internal settings - Other Raft values

- *bdr.raft_keep_min_entries*
- *bdr.raft_log_min_apply_duration*
- *bdr.raft_log_min_message_duration*
- *bdr.raft_group_max_connections*

### Internal settings - Other values

- *bdr.backwards_compatibility*
- *bdr.track_replication_estimates*
- *bdr.lag_tracker_apply_rate_weight*
- *bdr.enable_auto_sync_reconcile*
- *bdr.allow_bdr_schema_changes*

### Node management

- *List of node states*
- *Node-management commands*
- *bdr_init_physical*
- *bdr_config*

### Node management interfaces

- *Group configuration using bdr.alter_node_group_option()*
- *bdr.alter_node_interface*
- *bdr.alter_node_option*
- *bdr.alter_subscription_enable*
- *bdr.alter_subscription_disable*
- *bdr.create_node*
- *bdr.create_node_group*

- *bdr.drop_node_group*
- *bdr.join_node_group*
- *bdr.part_node*
- *bdr.promote_node*
- *bdr.switch_node_group*
- *bdr.sync_node_cancel*
- *bdr.wait_for_join_completion*

### Configuring routing thresholds

### Commit scopes

- *Commit scope syntax*
- *commit_scope_degrade_operation*
- *Commit scope targets*
- *ORIGIN_GROUP*
- *Commit scope groups*
- *ANY*
- *ANY NOT*
- *Effects of MAJORITY and ALL node replication in general*
- *MAJORITY NOT*
- *ALTER TABLE disallowed commands*
- *ALL NOT*
- *The confirmation level*
- *ON RECEIVED*
- *ON REPLICATED*
- *ON DURABLE*
- *ON VISIBLE*
- *The commit scope kinds*
- *SYNCHRONOUS COMMIT*
- *DEGRADE ON parameters*
- *commit_scope_degrade_operation*
- *QUORUM COMMIT*
- *QUORUM COMMIT parameters*
- *QUORUM COMMIT ABORT ON parameters*
- *commit_decision settings*
- *Working with the CAMO partner*
- *DEGRADE ON parameters*
- *LAG CONTROL*

- *LAG CONTROL parameters*
- *SYNCHRONOUS COMMIT and GROUP COMMIT*
- *GROUP COMMIT parameters*
- *QUORUM COMMIT ABORT ON parameters*
- *DEGRADE ON parameters*
- *transaction_tracking settings*
- *conflict_resolution settings*
- *GROUP COMMIT commit_decision settings*
- *commit_scope_degrade_operation settings*

#### bdr_read_all_conflicts

- *Column-level conflict detection*
- *List of conflict types*
- *Missing-column conflict resolution*
- *List of conflict resolvers*
- *Default conflict resolvers*
- *List of conflict resolutions*
- *Conflict logging*

#### Column-level conflict functions

- *Using bdr.alter_table_conflict_detection to enable column-level conflict resolution*
- *bdr.alter_node_set_conflict_resolver*
- *bdr.alter_node_set_log_config*

#### Replication set management

- *bdr.create_replication_set*
- *bdr.alter_replication_set*
- *bdr.drop_replication_set*
- *bdr.alter_node_replication_sets*

#### Replication set membership

- *bdr.replication_set_add_table*
- *bdr.replication_set_remove_table*

#### DDL replication filtering

- *bdr.replication_set_add_ddl_filter*
- *bdr.replication_set_remove_ddl_filter*

### Testing and tuning commands

- *pgd_bench*

### Sequences

### Sequence functions

- *bdr.alter_sequence_set_kind*
- *bdr.extract_timestamp_from_snowflakeid*
- *bdr.extract_nodeid_from_snowflakeid*
- *bdr.extract_localseqid_from_snowflakeid*
- *bdr.timestamp_to_snowflakeid*
- *bdr.extract_timestamp_from_timeshard*
- *bdr.extract_nodeid_from_timeshard*
- *bdr.extract_localseqid_from_timeshard*
- *bdr.timestamp_to_timeshard*
- *bdr.galloc_chunk_info*

### KSUUID v2 functions

- *bdr.gen_ksuuid_v2*
- *bdr.ksuuid_v2_cmp*
- *bdr.extract_timestamp_from_ksuuid_v2*

### KSUUID v1 functions

- *bdr.gen_ksuuid_v2*
- *bdr.uuid_v1_cmp*
- *bdr.extract_timestamp_from_ksuuid_v2*

### Replicated large objects

### Large object interface

- *bdr.lo_creat*
- *bdr.lo_create*
- *bdr.lo_open*
- *bdr.lo_close*
- *bdr.loread*
- *bdr.lowrite*
- *bdr.lo_lseek*
- *bdr.lo_lseek64*
- *bdr.lo_tell*

- *bdr.lo_tell64*
- *bdr.lo_truncate*
- *bdr.lo_truncate64*
- *bdr.lo_unlink*
- *bdr.lo_import*
- *bdr.lo_import_with_oid*
- *bdr.lo_export*
- *bdr.lo_get*
- *bdr.lo_get_fragment*
- *bdr.lo_from_bytea*
- *bdr.lo_put*

### AutoPartition in PGD

- *bdr.autopartition*
- *bdr.drop_autopartition*
- *bdr.autopartition_wait_for_partitions*
- *bdr.autopartition_wait_for_partitions_on_all_nodes*
- *bdr.autopartition_find_partition*
- *bdr.autopartition_enable*
- *bdr.autopartition_disable*
- *Internal functions*
- *bdr.autopartition_create_partition*
- *bdr.autopartition_drop_partition*

### Stream triggers reference

### Stream triggers manipulation interfaces

- *bdr.create_conflict_trigger*
- *bdr.create_transform_trigger*
- *bdr.drop_trigger*

### Stream triggers row functions

- *bdr.trigger_get_row*
- *bdr.trigger_get_committs*
- *bdr.trigger_get_xid*
- *bdr.trigger_get_type*
- *bdr.trigger_get_conflict_type*
- *bdr.trigger_get_origin_node_id*
- *bdr.ri_fkey_on_del_trigger*

### Stream triggers row variables

- *TG_NAME*
- *TG_WHEN*
- *TG_LEVEL*
- *TG_OP*
- *TG_RELID*
- *TG_TABLE_NAME*
- *TG_TABLE_SCHEMA*
- *TG_NARGS*
- *TG_ARGV[]*

### Internal catalogs and views

- *bdr.autopartition_partitions*
- *bdr.autopartition_rules*
- *bdr.ddl_epoch*
- *bdr.event_history*
- *bdr.event_summary*
- *bdr.local_leader_change*
- *bdr.node_config*
- *bdr.node_config_summary*
- *bdr.node_group_config*
- *bdr.node_group_routing_config_summary*
- *bdr.node_group_routing_info*
- *bdr.node_group_routing_summary*
- *bdr.node_peer_progress_mapping*
- *bdr.node_routing_config_summary*
- *bdr.prepared_xacts_summary*
- *bdr.reconcile_2pc_requests*
- *bdr.reconcile_2pc_requests_summary*
- *bdr.sequence_kind*
- *bdr.sync_node_requests*
- *bdr.sync_node_requests_summary*

### Internal system functions

### General internal functions

- *bdr.bdr_get_commit_decisions*
- *bdr.bdr_track_commit_decision*

- *bdr.consensus_kv_fetch*
- *bdr.consensus_kv_store*
- *bdr.decode_message_payload*
- *bdr.decode_message_response_payload*
- *bdr.difference_fix_origin_create*
- *bdr.difference_fix_session_reset*
- *bdr.difference_fix_session_setup*
- *bdr.difference_fix_xact_set_avoid_conflict*
- *bdr.drop_node*
- *bdr.get_global_locks*
- *bdr.get_node_conflict_resolvers*
- *bdr.get_slot_flush_timestamp*
- *bdr.internal_alter_sequence_set_kind*
- *bdr.internal_replication_set_add_table*
- *bdr.internal_replication_set_remove_table*
- *bdr.internal_submit_join_request*
- *bdr.isolation_test_session_is_blocked*
- *bdr.local_node_info*
- *bdr.msgb_connect*
- *bdr.msgb_deliver_message*
- *bdr.node_catchup_state_name*
- *bdr.node_kind_name*
- *bdr.peer_state_name*
- *bdr.pg_xact_origin*
- *bdr.reconcile_2pc_status_name*
- *bdr.request_replay_progress_update*
- *bdr.reset_relation_stats*
- *bdr.reset_subscription_stats*
- *bdr.resynchronize_table_from_node*
- *bdr.seq_currval*
- *bdr.seq_lastval*
- *bdr.seq_nextval*
- *bdr.show_subscription_status*
- *bdr.show_workers*
- *bdr.show_writers*
- *bdr.sync_status_name*

### Task manager functions

- *bdr.taskmgr_set_leader*
- *bdr.taskmgr_get_last_completed_workitem*
- *bdr.taskmgr_work_queue_check_status*
- *bdr.get_min_required_replication_slots*
- *bdr.get_min_required_worker_processes*
- *bdr.stat_last_reset_time*
- *bdr.stat_get_activity*
- *bdr.stat_get_commit_scope*
- *bdr.stat_get_commit_scope_proc_wait*
- *bdr.stat_get_reconcile_commit_scope*
- *bdr.worker_role_id_name*
- *bdr.lag_history*
- *bdr.get_raft_instance_by_nodegroup*
- *bdr.monitor_camo_on_all_nodes*
- *bdr.monitor_raft_details_on_all_nodes*
- :ref:``bdr.monitor_replslots_details_on_all_nodes` <*bdr.monitor_replslots_details_on_all_nodes*>`
- *bdr.monitor_subscription_details_on_all_nodes*
- *bdr.monitor_version_details_on_all_nodes*
- *bdr.node_group_member_info*

### Column-level conflict functions

- *Using bdr.alter_table_conflict_detection to enable column-level conflict resolution*
- *bdr.alter_node_set_conflict_resolver*
- *bdr.alter_node_set_log_config*

### Column-level conflict functions

- *bdr.column_timestamps_create*

### bdr_read_all_conflicts

- *Column-level conflict detection*
- *List of conflict types*
- *Missing-column conflict resolution*
- *List of conflict resolvers*
- *Default conflict resolvers*
- *List of conflict resolutions*
- *Conflict logging*

### User visible catalogs and views

Catalogs and views are listed here in alphabetical order.

### bdr.camo_decision_journal

A persistent journal of decisions resolved by a CAMO partner node after a failover, in case `bdr.logical_transaction_status` was invoked. Unlike `bdr.commit_decisions` , this doesn't cover transactions processed under normal operational conditions (that is, both nodes of a CAMO pair are running and connected). Entries in this journal aren't ever cleaned up automatically. This is a diagnostic tool that the system doesn't depend on.

#### `bdr.camo_decision_journal` columns

| Name | Type | Description |
|---|---|---|
| origin_node_id | oid | OID of the node where the transaction executed |
| origin_xid | oid | Transaction ID on the remote origin node |
| decision | char | 'c' for commit, 'a' for abort |
| decision_ts | timestamptz | Decision time |

### bdr.commit_decisions

A view used internally on nodes configured as Commit At Most Once (CAMO) partners. Shows the decisions a CAMO partner took on transactions in the last 15 minutes.

#### `bdr.commit_decisions` columns

| Name | Type | Description |
|---|---|---|
| origin_node_id | oid | OID of the node where the transaction executed |
| origin_xid | xid | Transaction ID on the remote origin node |
| decision | char | Decision taken; can be`c` (commit), or *a* (abort). |
| local_xid | xid | Transaction ID on the local node |
| commit_ts | timestamptz | Commit timestamp of the transaction |
| decision_ts | timestamptz | Decision time |

### bdr.commit_scopes

Catalog storing all possible commit scopes that you can use for `bdr.commit_scope` to enable Group Commit.

#### `bdr.commit_scopes` columns

| Name | Type | Description |
|---|---|---|
| commit_scope_id | oid | ID of the scope to be referenced |
| commit_scope_name | name | Name of the scope to be referenced |
| commit_scope_origin_node_group | oid | Node group for which the rule applies, referenced by ID |
| sync_scope_rule | text | Definition of the scope |

### bdr.conflict_history

This table is the default table where conflicts are logged. The table is RANGE partitioned on column `local_time` and is managed by Autopartition. The default data retention period is 30 days.

Access to this table is possible by any table owner, who can see all conflicts for the tables they own, restricted by row-level security.

For details, see *bdr_read_all_conflicts* .

#### `bdr.conflict_history` columns

| Name | Type | Description |
|---|---|---|
| sub_id | oid | Subscription that produced this conflict; can be joined to *bdr.subscription* table |
| origin_node_id | oid | OID (as seen in the *Replicated large objects* catalog) of the node that produced the conflicting change |
| local_xid | xid | Local transaction of the replication process at the time of conflict |
| local_lsn | pg_lsn | Local LSN at the time of conflict |
| local_time | timestamp with time zone | Local time of the conflict |
| remote_xid | xid | Transaction that produced the conflicting change on the remote node (an origin) |
| remote_change_nr | oid | Index of the change within its transaction |
| remote_commit_lsn | pg_lsn | Commit LSN of the transaction which produced the conflicting change on the remote node (an origin) |
| remote_commit_time | timestamp with time zone | Commit timestamp of the transaction that produced the conflicting change on the remote node (an origin) |
| conflict_type | text | Detected type of the conflict |
| conflict_resolution | text | Conflict resolution chosen |
| conflict_index | regclass | Conflicting index (valid only if the index wasn't dropped since) |
| reloid | oid | Conflicting relation (valid only if the index wasn't dropped since) |
| nspname | text | Name of the schema for the relation on which the conflict has occurred at the time of conflict (doesn't follow renames) |
| relname | text | Name of the relation on which the conflict has occurred at the time of conflict (does not follow renames) |
| key_tuple | json | Json representation of the key used for matching the row |
| remote_tuple | json | Json representation of an incoming conflicting row |
| local_tuple | json | Json representation of the local conflicting row |
| apply_tuple | json | Json representation of the resulting (the one that has been applied) row |
| local_tuple_xmin | xid | Transaction that produced the local conflicting row (if *local_tuple* is set and the row isn't frozen) |
| local_tuple_node_id | oid | Node that produced the local conflicting row (if *local_tuple* is set and the row isn't frozen) |
| local_tuple_commit_time | timestamp with time zone | Last-known-change timestamp of the local conflicting row (if *local_tuple* is set and the row isn't frozen). This commit timestamp belongs to the node that produced this tuple. |

### bdr.conflict_history_summary

A view containing user-readable details on row conflict.

#### `bdr.conflict_history_summary` columns

| Name | Type | Description |
|---|---|---|
| nspname | text | Name of the schema |
| relname | text | Name of the table |
| origin_node_id | oid | OID (as seen in the *Replicated large objects* catalog) of the node that produced the conflicting change |
| remote_commit_lsn | pg_lsn | Commit LSN of the transaction which produced the conflicting change on the remote node (an origin) |
| remote_change_nr | oid | Index of the change within its transaction |
| local_time | timestamp with time zone | Local time of the conflict |
| local_tuple_commit_time | timestamp with time zone | Time of local commit |
| remote_commit_time | timestamp with time zone | Time of remote commit |
| conflict_type | text | Type of conflict |
| conflict_resolution | text | Resolution adopted |

### bdr.consensus_kv_data

A persistent storage for the internal Raft-based KV store used by `bdr.consensus_kv_store()` and `bdr.consensus_kv_fetch()` interfaces.

#### `bdr.consensus_kv_data` Columns

| Name | Type | Description |
|---|---|---|
| kv_key | text | Unique key |
| kv_val | json | Arbitrary value in json format |
| kv_create_ts | timestamptz | Last write timestamp |
| kv_ttl | int | Time to live for the value in milliseconds |
| kv_expire_ts | timestamptz | Expiration timestamp (*kv_create_ts* + *kv_ttl*) |

### bdr.crdt_handlers

This table lists merge ("handlers") functions for all CRDT data types.

#### `bdr.crdt_handlers` Columns

| Name | Type | Description |
|---|---|---|
| crdt_type_id | regtype | CRDT data type ID |

continues on next page

Table 7 – continued from previous page

| Name | Type | Description |
|---|---|---|
| crdt_merge_id | regproc | Merge function for this data type |

### bdr.ddl_replication

This view lists DDL replication configuration as set up by current *DDL filters* .

#### `bdr.ddl_replication` columns

| Name | Type | Description |
|---|---|---|
| set_ddl_name | name | Name of DDL filter |
| set_ddl_tag | text | Command tags it applies on (regular expression) |
| set_ddl_role | text | Roles it applies to (regular expression) |
| set_name | name | Name of the replication set for which this filter is defined |

### bdr.depend

This table tracks internal object dependencies inside PGD catalogs.

### bdr.failover_replication_slots

This table tracks the status of logical replication slots that are being used with failover support. For more information on failover replication slots, see *CDC Failover support* .

#### `bdr.failover_replication_slots` columns

| Name | Type | Description |
|---|---|---|
| slot_name | name | Name of the replication slot |
| slot_id | oid | ID of the replication slot |
| node_group_id | oid | ID of the node group |
| plugin | name | Name of the plugin |
| twophase | boolean | Is the slot used for two-phase commit |
| active_node | oid | ID of the active node |
| active_pid | int | PID of the process currently decoding the slot |
| prev_node | oid | ID of the previous node |

### bdr.global_consensus_journal

This catalog table logs all the Raft messages that were sent while managing global consensus.

As for the `bdr.global_consensus_response_journal` catalog, the payload is stored in a binary encoded format, which can be decoded with the `bdr.decode_message_payload()` function. See the *bdr.global_consensus_journal_details* view for more details.

#### `bdr.global_consensus_journal` columns

| Name | Type | Description |
|---|---|---|
| log_index | int8 | ID of the journal entry |
| term | int8 | Raft term |
| origin | oid | ID of node where the request originated |
| req_id | int8 | ID for the request |
| req_payload | bytea | Payload for the request |
| trace_context | bytea | Trace context for the request |

### bdr.global_consensus_journal_details

This view presents Raft messages that were sent and the corresponding responses, using the `bdr.decode_message_payload()` function to decode their payloads.

#### `bdr.global_consensus_journal_details` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | name | Name of the node group |
| log_index | int8 | ID of the journal entry |
| term | int8 | Raft term |
| request_id | int8 | ID of the request |
| origin_id | oid | ID of the node where the request originated |
| req_payload | bytea | Payload of the request |
| origin_node_name | name | Name of the node where the request originated |
| message_type_no | oid | ID of the PGD message type for the request |
| message_type | text | Name of the PGD message type for the request |
| message_payload | text | PGD message payload for the request |
| response_message_type_no | oid | ID of the PGD message type for the response |
| response_message_type | text | Name of the PGD message type for the response |
| response_payload | text | PGD message payload for the response |
| response_errcode_no | text | SQLSTATE for the response |
| response_errcode | text | Error code for the response |
| response_message | text | Error message for the response |

### bdr.global_consensus_response_journal

This catalog table collects all the responses to the Raft messages that were received while managing global consensus.

As for the `bdr.global_consensus_journal` catalog, the payload is stored in a binary-encoded format, which can be decoded with the `bdr.decode_message_payload()` function. See the *bdr.global_consensus_journal_details* view for more details.

#### `bdr.global_consensus_response_journal` columns

| Name | Type | Description |
|---|---|---|
| log_index | int8 | ID of the journal entry |

continues on next page

Table 12 – continued from previous page

| Name | Type | Description |
|---|---|---|
| res_status | oid | Status code for the response |
| res_payload | bytea | Payload for the response |
| trace_context | bytea | Trace context for the response |

### bdr.global_lock

This catalog table stores the information needed for recovering the global lock state on server restart.

For monitoring usage, the *bdr.global_locks* view is preferable because the visible rows in `bdr.global_lock` don't necessarily reflect all global locking activity.

Don't modify the contents of this table. It is an important PGD catalog.

#### `bdr.global_lock` columns

| Name | Type | Description |
|---|---|---|
| ddl_epoch | int8 | DDL epoch for the lock |
| origin_node_id | oid | OID of the node where the global lock has originated |
| lock_type | oid | Type of the lock (DDL or DML) |
| nspname | name | Schema name for the locked relation |
| relname | name | Relation name for the locked relation |
| groupid | oid | OID of the top level group (for Advisory locks) |
| key1 | integer | First 32-bit key or lower order 32-bits of 64-bit key (for advisory locks) |
| key2 | integer | Second 32-bit key or higher order 32-bits of 64-bit key (for advisory locks) |
| key_is_bigint | boolean | True if 64-bit integer key is used (for advisory locks) |

### bdr.global_locks

A view containing active global locks on this node. The *bdr.global_locks* view exposes PGD's shared-memory lock state tracking, giving administrators greater insight into PGD's global locking activity and progress.

See *Monitoring global locks*

for more information about global locking.

#### `bdr.global_locks` columns

| Name | Type | Description |
|---|---|---|
| origin_node_id | oid | OID of the node where the global lock has originated |
| origin_node_name | name | Name of the node where the global lock has originated |
| lock_type | text | Type of the lock (DDL or DML) |
| relation | text | Locked relation name (for DML locks) or keys (for advisory locks) |
| pid | int4 | PID of the process holding the lock |
| acquire_stage | text | Internal state of the lock acquisition process |
| waiters | int4 | List of backends waiting for the same global lock |

continues on next page

Table 14 – continued from previous page

| Name | Type | Description |
|---|---|---|
| global_lock_request_time | timestamptz | Time this global lock acquire was initiated by origin node |
| local_lock_request_time | timestamptz | Time the local node started trying to acquire the local lock |
| last_state_change_time | timestamptz | Time *acquire_stage* last changed |

Column details:

- `relation` : For DML locks, `relation` shows the relation on which the DML lock is acquired. For global advisory locks, `relation` column actually shows the two 32-bit integers or one 64-bit integer on which the lock is acquired.
- `origin_node_id` and `origin_node_name` : If these are the same as the local node's ID and name, then the local node is the initiator of the global DDL lock, that is, it is the node running the acquiring transaction. If these fields specify a different node, then the local node is instead trying to acquire its local DDL lock to satisfy a global DDL lock request from a remote node.
- `pid` : The process ID of the process that requested the global DDL lock, if the local node is the requesting node. Null on other nodes. Query the origin node to determine the locker pid.
- `global_lock_request_time` : The timestamp at which the global-lock request initiator started the process of acquiring a global lock. Can be null if unknown on the current node. This time is stamped at the beginning of the DDL lock request and includes the time taken for DDL epoch management and any required flushes of pending-replication queues. Currently only known on origin node.
- `local_lock_request_time` : The timestamp at which the local node started trying to acquire the local lock for this global lock. This includes the time taken for the heavyweight session lock acquire but doesn't include any time taken on DDL epochs or queue flushing. If the lock is reacquired after local node restart, it becomes the node restart time.
- `last_state_change_time` : The timestamp at which the `bdr.global_locks.acquire_stage` field last changed for this global lock entry.

### bdr.group_camo_details

Uses `bdr.run_on_all_nodes` to gather CAMO-related information from all nodes.

#### `bdr.group_camo_details` columns

| Name | Type | Description |
|---|---|---|
| node_id | text | Internal node ID |
| node_name | text | Name of the node |
| camo_partner | text | Node name of the camo partner |
| is_camo_partner_connected | text | Connection status |
| is_camo_partner_ready | text | Readiness status |
| camo_transactions_resolved | text | Are there any pending and unresolved CAMO transactions |
| apply_lsn | text | Latest position reported as replayed (visible) |
| receive_lsn | text | Latest LSN of any change or message received (can go backwards in case of restarts) |
| apply_queue_size | text | Bytes difference between apply_lsn and receive_lsn |

### bdr.group_lease

Catalog table that stores information about node group leases. Only one node at a time can hold a specific type of lease. Currently, the leasing facility is exclusively used when replicating to analytics using the Postgres Analytics Accelerator (PGAA) by specifying `kind = 'a'`.

#### `bdr.group_lease` columns

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | Identifier of the node group associated with the lease. |
| node_id | oid | Identifier of the node currently holding the lease. |
| received_ts | timestamp with time zone | The local node time when the last lease renewal request was received. |
| request_ts | timestamp with time zone | The timestamp from the lease holder's clock when the renewal request was made. |
| duration | integer | The duration of the lease in seconds. |
| kind | char | The specific type of lease (for example, 'a' for analytics). |

### bdr.group_raft_details

Uses `bdr.run_on_all_nodes` to gather Raft Consensus status from all nodes.

#### `bdr.group_raft_details` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | Internal node ID |
| node_name | name | Name of the node |
| node_group_name | name | Name of the group is part of |
| state | text | Raft worker state on the node |
| leader_id | oid | Node id of the RAFT_LEADER |
| current_term | int | Raft election internal ID |
| commit_index | int | Raft snapshot internal ID |
| nodes | int | Number of nodes accessible |
| voting_nodes | int | Number of nodes voting |
| protocol_version | int | Protocol version for this node |

### bdr.group_replslots_details

Uses `bdr.run_on_all_nodes` to gather PGD slot information from all nodes.

#### `bdr.group_replslots_details` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | text | Name of the PGD group |
| origin_name | text | Name of the origin node |
| target_name | text | Name of the target node |
| slot_name | text | Slot name on the origin node used by this subscription |

continues on next page

Table 18 – continued from previous page

| Name | Type | Description |
|---|---|---|
| active | text | Is the slot active (does it have a connection attached to it) |
| state | text | State of the replication (catchup, streaming, ...) or 'disconnected' if offline |
| write_lag | interval | Approximate lag time for reported write |
| flush_lag | interval | Approximate lag time for reported flush |
| replay_lag | interval | Approximate lag time for reported replay |
| sent_lag_bytes | int8 | Bytes difference between sent_lsn and current WAL write position |
| write_lag_bytes | int8 | Bytes difference between write_lsn and current WAL write position |
| flush_lag_bytes | int8 | Bytes difference between flush_lsn and current WAL write position |
| replay_lag_byte | int8 | Bytes difference between replay_lsn and current WAL write position |

### bdr.group_subscription_summary

Uses `bdr.run_on_all_nodes` to gather subscription status from all nodes.

#### `bdr.group_subscription_summary` columns

| Name | Type | Description |
|---|---|---|
| origin_node_name | text | Name of the origin of the subscription |
| target_node_name | text | Name of the target of the subscription |
| last_xact_replay_timestamp | text | Timestamp of the last replayed transaction |
| sub_lag_seconds | text | Lag between now and last_xact_replay_timestamp |

### bdr.group_versions_details

Uses `bdr.run_on_all_nodes` to gather PGD information from all nodes.

#### `bdr.group_versions_details` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | Internal node ID |
| node_name | name | Name of the node |
| postgres_version | text | PostgreSQL version on the node |
| bdr_version | text | PGD version on the node |

### bdr.leader

Tracks leader nodes across subgroups in the cluster. Shows the status of all write leaders and subscriber-only group leaders (when optimized topology is enabled) in the cluster.

#### `bdr.leader` columns

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | ID of the node group. |
| leader_node_id | oid | ID of the leader node. |
| generation | int | Generation of the leader node. Leader_kind sets semantics. |
| leader_kind | "char" | Kind of the leader node. |

Leader_kind values can be:

| Value | Description |
|---|---|
| W | Write leader, as per proxy routing. In this case leader is maintained by sub-group Raft instance. *generation* corresponds to *write_leader_version* of respective *bdr.node_group_routing_info* record. |
| S | Subscriber-only group leader. This designated member of a SO group subscribes to upstream data nodes and is tasked with publishing upstream changes to remaining SO group members. Leader is maintained by top-level Raft instance. *generation* is updated sequentially upon leader change. |

### bdr.local_consensus_snapshot

This catalog table contains consensus snapshots created or received by the local node.

#### `bdr.local_consensus_snapshot` columns

| Name | Type | Description |
|---|---|---|
| log_index | int8 | ID of the journal entry |
| log_term | int8 | Raft term |
| snapshot | bytea | Raft snapshot data |

### bdr.local_consensus_state

This catalog table stores the current state of Raft on the local node.

#### `bdr.local_consensus_state` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | ID of the node |
| current_term | int8 | Raft term |
| apply_index | int8 | Raft apply index |
| voted_for | oid | Vote cast by this node in this term |
| last_known_leader | oid | node_id of last known Raft leader |

### bdr.local_node

This table identifies the local node in the current database of the current Postgres instance.

#### `bdr.local_node` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | ID of the node |
| pub_repsets | text[] | Published replication sets |
| sub_repsets | text[] | Subscribed replication sets |
| node_uuid | uuid | UUID of the node |

### bdr.local_node_summary

A view containing the same information as *bdr.node_summary* (plus `pub_repsets` and `sub_repsets` ), but only for the local node.

### bdr.local_sync_status

Information about status of either subscription or table synchronization process.

#### `bdr.local_sync_status` columns

| Name | Type | Description |
|---|---|---|
| sync_kind | char | Kind of synchronization done |
| sync_subid | oid | ID of subscription doing the synchronization |
| sync_nspname | name | Schema name of the synchronized table (if any) |
| sync_relname | name | Name of the synchronized table (if any) |
| sync_status | char | Current state of the synchronization |
| sync_remote_relid | oid | ID of the synchronized table (if any) on the upstream |
| sync_end_lsn | pg_lsn | Position at which the synchronization state last changed |

### bdr.node

This table lists all the PGD nodes in the cluster.

The view *bdr.node_summary* provides a human-readable version of most of the columns from `bdr.node` .

#### `bdr.node` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | ID of the node |
| node_name | name | Name of the node |
| node_group_id | oid | ID of the node group |
| source_node_id | oid | ID of the source node |

continues on next page

Table 27 – continued from previous page

| Name | Type | Description |
|---|---|---|
| synchronize_structure | "char" | Schema synchronization done during the join |
| node_state | oid | Consistent state of the node |
| target_state | oid | State that the node is trying to reach (during join or promotion) |
| seq_id | int4 | Sequence identifier of the node used for generating unique sequence numbers |
| dbname | name | Database name of the node |
| node_dsn | char | Connection string for the node |
| proto_version_ranges | int[] | Supported protocol version ranges by the node |
| generation | smallint | Counter incremented when a node joins with the same name as a previous node |
| node_kind | oid | ID of the node kind |
| node_join_finished | boolean | Check if the join is finished |
| node_uuid | uuid | UUID of the node (UNIQUE) |

### bdr.node_catchup_info

This catalog table records relevant catchup information on each node, either if it is related to the join or part procedure.

#### `bdr.node_catchup_info` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | ID of the node |
| node_source_id | oid | ID of the node used as source for the data |
| slot_name | name | Slot used for this source |
| min_node_lsn | pg_lsn | Minimum LSN at which the node can switch to direct replay from a peer node |
| catchup_state | oid | Status code of the catchup state |
| origin_node_id | oid | ID of the node from which we want transactions |

If a node(node_id) needs missing data from a parting node(origin_node_id), it can get it from a node that already has it(node_source_id) by forwarding. The records in this table persists until the node(node_id) is a member of the EDB Postgres Distributed cluster.

### bdr.node_catchup_info_details

A view of `bdr.node_catchup_info` catalog which shows info in more friendly way

### bdr.node_conflict_resolvers

Currently configured conflict resolution for all known conflict types.

#### `bdr.node_conflict_resolvers` columns

| Name | Type | Description |
|---|---|---|
| conflict_type | text | Type of the conflict |

continues on next page

Table 29 – continued from previous page

| Name | Type | Description |
|---|---|---|
| conflict_resolver | text | Resolver used for this conflict type |

### bdr.node_group

This catalog table lists all the PGD node groups. See also *bdr.node_group_summary* for a view containing user-readable details.

#### `bdr.node_group` columns

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | ID of the node group. |
| node_group_name | name | Name of the node group. |
| node_group_default_repset | oid | Default replication set for this node group. |
| node_group_default_repset_ext | oid | Default replication set for this node group. |
| node_group_parent_id | oid | ID of parent group (0 if this is a root group). |
| node_group_flags | int | Group flags. |
| node_group_uuid | uuid | The uuid of the group. |
| node_group_apply_delay | interval | How long a subscriber waits before applying changes from the provider. |
| node_group_check_constraints | bool | Whether the apply process checks constraints when applying data. |
| node_group_num_writers | int | Number of writers to use for subscriptions backing this node group. |
| node_group_enable_wal_decoder | bool | Whether the group has enable_wal_decoder set. |
| node_group_streaming_mode | char | Transaction streaming setting: 'O' - off, 'F' - file, 'W' - writer, 'A' - auto, 'D' - default. |
| node_group_default_commit_scope | oid | ID of the node group's default commit scope. |
| node_group_location | char | Name of the location associated with the node group. |
| node_group_enable_routing | char | Whether the node group allows routing through Connection Manager. |
| node_group_enable_raft | bool | Whether the node group allows Raft Consensus. |

### bdr.node_group_replication_sets

A view showing default replication sets create for PGD groups. See also `bdr.replication_sets`.

#### `bdr.node_group_replication_sets` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | name | Name of the PGD group |
| def_repset | name | Name of the default repset |
| def_repset_ops | text[] | Actions replicated by the default repset |

continues on next page

Table 31 – continued from previous page

| Name | Type | Description |
|---|---|---|
| def_repset_ext | name | Name of the default "external" repset (usually same as def_repset) |
| def_repset_ext_ops | text[] | Actions replicated by the default "external" repset (usually same as def_repset_ops) |

### bdr.node_group_summary

A view containing user-readable details about node groups. See also *bdr.node_group_config* .

#### `bdr.node_group_summary` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | name | Name of the node group |
| default_repset | name | Default replication set for this node group |
| parent_group_name | name | Name of parent group (NULL if this is a root group) |
| node_group_type | text | Type of the node group (one of "global", "data", "shard" or "subscriber-only") |
| apply_delay | interval | How long a subscriber waits before applying changes from the provider |
| check_constraints | boolean | Whether the apply process checks constraints when applying data |
| num_writers | integer | Number of writers to use for subscriptions backing this node group |
| enable_wal_decoder | boolean | Whether the group has enable_wal_decoder set |
| streaming_mode | text | Transaction streaming setting: "off", "file", "writer", "auto" or "default" |
| default_commit_scope | name | Name of the node group's default commit scope |
| location | name | Name of the location associated with the node group |
| server_pool_mode | text | *Connection pooling* mode for this node group: *none*, *session*, or *transaction* |
| enable_routing | boolean | Whether the node group allows routing through connection manager |
| enable_raft | boolean | Whether the node group allows Raft Consensus |
| route_writer_max_lag | bigint | Maximum write lag accepted |
| route_reader_max_lag | bigint | Maximum read lag accepted |
| route_writer_wait_flush | boolean | Switch if we need to wait for the flush |

### bdr.node_local_info

A catalog table used to store per-node configuration that's specific to the local node (as opposed to global view of per-node configuration).

#### `bdr.node_local_info` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | The OID of the node (including the local node) |

continues on next page

Table 33 – continued from previous page

| Name | Type | Description |
|---|---|---|
| applied_state | oid | Internal ID of the node state |
| ddl_epoch | int8 | Last epoch number processed by the node |
| slot_name | name | Name of the slot used to connect to that node (NULL for the local node) |
| origin_name | name | Name of the *Cleanup of replication origins* for that node. It will be NULL for the local node or for nodes that are not data nodes such as subscriber-only nodes or standbys. |

### bdr.node_log_config

A catalog view that stores information on the conflict logging configurations.

#### `bdr.node_log_config` columns

| Name | Description |
|---|---|
| log_name | Name of the logging configuration |
| log_to_file | Whether it logs to the server log file |
| log_to_table | Whether it logs to a table, and which table is the target |
| log_conflict_type | Which conflict types it logs, if NULL means all |
| log_conflict_res | Which conflict resolutions it logs, if NULL means all |

### bdr.node_peer_progress

Catalog used to keep track of every node's progress in the replication stream. Every node in the cluster regularly broadcasts its progress every `bdr.replay_progress_frequency` milliseconds to all other nodes (default is 60000 ms, that is, 1 minute). Expect N * (N-1) rows in this relation.

You might be more interested in the `bdr.node_slots` view for monitoring purposes. See also *Monitoring the Connection Manager* .

#### `bdr.node_peer_progress` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | OID of the originating node that reported this position info |
| peer_node_id | oid | OID of the node's peer (remote node) for which this position info was reported |
| last_update_sent_time | timestamptz | Time at which the report was sent by the originating node |
| last_update_recv_time | timestamptz | Time at which the report was received by the local server |
| last_update_node_lsn | pg_lsn | LSN on the originating node at the time of the report |
| peer_position | pg_lsn | Latest LSN of the node's peer seen by the originating node |
| peer_replay_time | timestamptz | Latest replay time of peer seen by the reporting node |
| last_update_horizon_xid | oid | Internal resolution horizon: all lower xids are known resolved on the reporting node |
| last_update_horizon_lsn | pg_lsn | Internal resolution horizon: same in terms of an LSN of the reporting node |

### bdr.node_replication_rates

This view contains information about outgoing replication activity from a given node.

#### `bdr.node_replication_rates` columns

| Column | Type | Description |
|---|---|---|
| peer_node_id | oid | OID of node's peer (remote node) for which this info was reported |
| target_name | name | Name of the target peer node |
| sent_lsn | pg_lsn | Latest sent position |
| replay_lsn | pg_lsn | Latest position reported as replayed (visible) |
| replay_lag | interval | Approximate lag time for reported replay |
| replay_lag_bytes | int8 | Bytes difference between replay_lsn and current WAL write position on origin |
| replay_lag_size | text | Human-readable bytes difference between replay_lsn and current WAL write position |
| apply_rate | bigint | LSNs being applied per second at the peer node |
| catchup_interval | interval | Approximate time required for the peer node to catch up to all the changes that are yet to be applied |

**Note**

The *replay_lag* is set immediately to zero after reconnect. As a workaround, use *replay_lag_bytes* , *replay_lag_size* , or *catchup_interval* .

### bdr.node_slots

This view contains information about replication slots used in the current database by PGD.

See *Monitoring outgoing replication*

for guidance on the use and interpretation of this view's fields.

#### `bdr.node_slots` columns

| Name | Type | Description |
|---|---|---|
| target_dbname | name | Database name on the target node |
| node_group_name | name | Name of the PGD group |
| node_group_id | oid | OID of the PGD group |
| origin_name | name | Name of the origin node |
| target_name | name | Name of the target node |
| origin_id | oid | OID of the origin node |
| target_id | oid | OID of the target node |
| local_slot_name | name | Name of the replication slot according to PGD |
| slot_name | name | Name of the slot according to Postgres (same as above) |
| is_group_slot | boolean | True if the slot is the node-group crash recovery slot for this node (see ["Group Replication Slot"](nodes#Group Replication Slot)) |

continues on next page

Table 37 – continued from previous page

| Name | Type | Description |
|---|---|---|
| is_decoder_slot | boolean | Is this slot used by the decoding worker feature |
| plugin | name | Logical decoding plugin using this slot (should be pglogical_output or bdr) |
| slot_type | text | Type of the slot (should be logical) |
| datoid | oid | OID of the current database |
| database | name | Name of the current database |
| temporary | bool | Is the slot temporary |
| active | bool | Is the slot active (does it have a connection attached to it) |
| active_pid | int4 | PID of the process attached to the slot |
| xmin | xid | XID needed by the slot |
| catalog_xmin | xid | Catalog XID needed by the slot |
| restart_lsn | pg_lsn | LSN at which the slot can restart decoding |
| confirmed_flush_lsn | pg_lsn | Latest confirmed replicated position |
| usesysid | oid | sysid of the user the replication session is running as |
| usename | name | username of the user the replication session is running as |
| application_name | text | Application name of the client connection (used by *synchronous_standby_names*) |
| client_addr | inet | IP address of the client connection |
| client_hostname | text | Hostname of the client connection |
| client_port | int4 | Port of the client connection |
| backend_start | timestamptz | When the connection started |
| state | text | State of the replication (catchup, streaming, ...) or 'disconnected' if offline |
| sent_lsn | pg_lsn | Latest sent position |
| write_lsn | pg_lsn | Latest position reported as written |
| flush_lsn | pg_lsn | Latest position reported as flushed to disk |
| replay_lsn | pg_lsn | Latest position reported as replayed (visible) |
| write_lag | interval | Approximate lag time for reported write |
| flush_lag | interval | Approximate lag time for reported flush |
| replay_lag | interval | Approximate lag time for reported replay |
| sent_lag_bytes | int8 | Bytes difference between sent_lsn and current WAL write position |
| write_lag_bytes | int8 | Bytes difference between write_lsn and current WAL write position |
| flush_lag_bytes | int8 | Bytes difference between flush_lsn and current WAL write position |
| replay_lag_bytes | int8 | Bytes difference between replay_lsn and current WAL write position |
| sent_lag_size | text | Human-readable bytes difference between sent_lsn and current WAL write position |
| write_lag_size | text | Human-readable bytes difference between write_lsn and current WAL write position |
| flush_lag_size | text | Human-readable bytes difference between flush_lsn and current WAL write position |
| replay_lag_size | text | Human-readable bytes difference between replay_lsn and current WAL write position |

**Note**

The *replay_lag* is set immediately to zero after reconnect. As a workaround, use *replay_lag_bytes* or *replay_lag_size* .

### bdr.node_summary

This view contains summary information about all PGD nodes known to the local node.

#### `bdr.node_summary` columns

| Name | Type | Description |
|---|---|---|
| node_name | name | Name of the node |
| node_group_name | name | Name of the PGD group the node is part of |
| interface_connstr | text | Connection string to the node |
| peer_state_name | text | Consistent state of the node in human readable form |
| peer_target_state_name | text | State that the node is trying to reach (during join or promotion) |
| node_seq_id | int4 | Sequence identifier of the node used for generating unique sequence numbers |
| node_local_dbname | name | Database name of the node |
| node_id | oid | OID of the node |
| node_group_id | oid | OID of the PGD node group |
| node_kind_name | oid | Node kind name |
| node_uuid | uuid | UUID of the node |

### bdr.parted_origin_catchup_info

This table records relevant catchup information on each node related to parted orgins.

#### `bdr.parted_origin_catchup_info` columns

| Name | Type | Description |
|---|---|---|
| parting_peer_node_id | oid | ID of the parted node |
| node_id | oid | ID of the node |
| node_group_id | oid | ID of the node group |
| origin_catchup_lsn | pg_lsn | The LSN which the node will wait for its group slot to catch up to and then move its state to DONE |
| origin_catchup_state | oid | Status code of the parted origin catchup |

A node(`node_id` ) waits for its group slot to catch up with the recorded LSN, (`origin_catchup_lsn` ). This is to ensure it's group slot is caught up with all the transactions originating from PARTED node (`parting_peer_node_id` ).

The records in this table persists until the parting node (`parting_peer_node_id` ) is automatically removed.

### bdr.parted_origin_catchup_info_details

This table is a friendly view of `bdr.parted_origin_catchup_info` with relevant catchup information on each node related to parted orgins, in this case in text form.

#### `bdr.parted_origin_catchup_info_details` columns

| Name | Type | Description |
|---|---|---|
| target_node_id | oid | ID of the target node |
| target_node_name | text | Name of the target node |
| parting_node_id | oid | ID of the parted node |
| parting_node_name | text | Name of the parted node |
| node_group_id | oid | ID of the node group |
| node_group_name | text | Name of the node group |
| parting_catchup_lsn | pg_lsn | The LSN which the node will wait for its group slot to catch up to and then move its state to DONE |
| parting_catchup_state | oid | Parted origin's catchup status code |
| parting_catchup_state_name | text | Parted origin's catchup status text |

A node(`target_node_id` ) waits for its group slot to catch up with the recorded LSN, (`parting_catchup_lsn` ). This is to ensure it's group slot is caught up with all the transactions originating from PARTED node (`parting_node_id` ).

The records in this table persist until the parting node (`parting_node_id` ) is automatically removed.

### bdr.pending_failed_changes

A table storing captured changes from transactions that failed to apply during replication. Each row represents one DML or DDL operation from a failed transaction. Created by the apply error policy when the policy is set to `record_all_and_disable` or `record_all_and_skip` . For guidance on reviewing and resolving the contents of this table, see *Handling replication apply errors* .

#### `bdr.pending_failed_changes` columns

| Name | Type | Description |
|---|---|---|
| *id* | *bigint* | Primary key. |
| *failed_txn_id* | *uuid* | Unique identifier for the failed transaction. Use as a convenient alternative to *(subscription_name, remote_xid, remote_commit_lsn)* when calling resolution functions. |
| *sub_id* | *oid* | OID of the subscription that received the failed transaction. |
| *origin_node_id* | *oid* | OID of the node where the transaction originated. |
| *remote_xid* | *xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | *pg_lsn* | Commit LSN of the failed transaction on the origin node. |
| *remote_commit_end_lsn* | *pg_lsn* | LSN of the end of the commit record, used to advance the subscription position. |
| *remote_commit_time* | *timestamptz* | Commit timestamp on the origin node. |
| *change_seq* | *integer* | Order of the change within the transaction. |
| *schema_name* | *name* | Schema of the target table. |
| *table_name* | *name* | Name of the target table. |
| *operation* | *text* | Operation type: *INSERT*, *UPDATE*, *DELETE*, *SQL*, *DDL*, or *TRUNCATE*. |
| *key_columns* | *jsonb* | Primary key column values. |
| *old_values* | *jsonb* | Before-image column values for *UPDATE* and *DELETE* operations. |

continues on next page

Table 41 – continued from previous page

| Name | Type | Description |
|---|---|---|
| *new_values* | *jsonb* | After-image column values for *INSERT* and *UPDATE* operations, or the SQL text for *DDL* and *SQL* operations. |
| *change_status* | *text* | Resolution status: *pending*, *skipped*, *applied*, or *modified*. |
| *modified_values* | *jsonb* | User-edited column values set by *bdr.modify_failed_change*, used when the change is retried. |
| *error_message* | *text* | Error message from the failed apply. Populated on the last change of the failed transaction. |
| *error_detail* | *text* | Additional error detail. |
| *error_sqlstate* | *text* | SQLSTATE code of the error. |
| *capture_truncated* | *boolean* | Whether the change capture was truncated because the transaction exceeded *apply_error_max_record_size*. When *true*, some changes in the transaction may be missing and *bdr.apply_failed_transaction* will refuse to apply the transaction. Default is *false*. |
| *recorded_at* | *timestamptz* | Timestamp when the change was recorded. |
| *resolved_at* | *timestamptz* | Timestamp when the change was resolved. |
| *resolved_by* | *text* | How the change was resolved: *user*, *auto_skip*, or *retry*. |

### bdr.pending_failed_changes_sql

A view over `bdr.pending_failed_changes` that combines key columns with subscription and origin node names, and includes a reconstructed SQL statement for each change generated by `bdr.reconstruct_change_sql` . Unlike the base table, the view uses the `modified_values` column in place of `new_values` when present, so reconstructed SQL reflects any edits made by `bdr.modify_failed_change` .

#### `bdr.pending_failed_changes_sql` columns

| Name | Type | Description |
|---|---|---|
| *id* | *bigint* | Primary key of the row in *bdr.pending_failed_changes*. |
| *failed_txn_id* | *uuid* | UUID of the failed transaction. |
| *subscription_name* | *name* | Name of the subscription. |
| *origin_node* | *name* | Name of the origin node. |
| *remote_xid* | *xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | *pg_lsn* | Commit LSN of the failed transaction on the origin node. |
| *change_seq* | *integer* | Order of the change within the transaction. |
| *qualified_table* | *text* | Schema-qualified table name (*schema_name.table_name*). |
| *operation* | *text* | Operation type: *INSERT*, *UPDATE*, *DELETE*, *SQL*, *DDL*, or *TRUNCATE*. |
| *sql_statement* | *text* | Executable SQL representing the captured change, generated by *bdr.reconstruct_change_sql*. |
| *change_status* | *text* | Resolution status: *pending*, *skipped*, *applied*, or *modified*. |
| *error_message* | *text* | Error message from the failed apply, populated on the last change of the failed transaction. |
| *recorded_at* | *timestamptz* | Timestamp when the change was recorded. |

### bdr.pgd_largeobject

Stores the data pages of replicated large objects. This table is the PGD equivalent of pg_catalog.largeobject and is used by the *Replicated large objects* . Its contents are replicated to all nodes.

#### `bdr.pgd_largeobject` columns

| Name | Type | Description |
|---|---|---|
| *loid* | *oid* | OID of the large object this page belongs to. |
| *pageno* | *int* | Page number within the large object, starting from 0. |
| *data* | *bytea* | Data stored in this page. May be zero-length for sparse pages. |

### bdr.pgd_largeobject_metadata

Stores ownership and access control information for replicated large objects. This table is the PGD equivalent of pg_catalog.largeobject_metadata and is used by the *Replicated large objects* . Its contents are replicated to all nodes.

#### `bdr.pgd_largeobject_metadata` columns

| Name | Type | Description |
|---|---|---|
| *oid* | *oid* | OID of the large object. |
| *lomowner* | *oid* | OID of the owner of the large object. |
| *lomacl* | *aclitem[]* | Access privileges for the large object. |

### bdr.queue

This table stores the historical record of replicated DDL statements.

#### `bdr.queue` columns

| Name | Type | Description |
|---|---|---|
| queued_at | timestamptz | When was the statement queued |
| role | name | Which role has executed the statement |
| replication_sets | text[] | Which replication sets was the statement published to |
| message_type | char | Type of a message. Possible values: A - Table sync D - DDL S - Sequence T - Truncate Q - SQL statement |
| message | json | Payload of the message needed for replication of the statement |

### bdr.replication_set

A table that stores replication set configuration. For user queries, we recommend instead checking the `bdr.replication_sets` view.

#### `bdr.replication_set` columns

| Name | Type | Description |
|---|---|---|
| set_id | oid | OID of the replication set |
| set_nodeid | oid | OID of the node (always local node oid currently) |
| set_name | name | Name of the replication set |
| replicate_insert | boolean | Indicates if the replication set replicates INSERTs |
| replicate_update | boolean | Indicates if the replication set replicates UPDATEs |
| replicate_delete | boolean | Indicates if the replication set replicates DELETEs |
| replicate_truncate | boolean | Indicates if the replication set replicates TRUNCATEs |
| set_isinternal | boolean | Reserved |
| set_autoadd_tables | boolean | Indicates if new tables are automatically added to this replication set |
| set_autoadd_seqs | boolean | Indicates if new sequences are automatically added to this replication set |

### bdr.replication_set_table

A table that stores replication set table membership. For user queries, we recommend instead checking the `bdr.tables` view.

#### `bdr.replication_set_table` columns

| Name | Type | Description |
|---|---|---|
| set_id | oid | OID of the replication set |
| set_reloid | regclass | Local ID of the table |
| set_att_list | text[] | Reserved |
| set_row_filter | pg_node_tree | Compiled row filtering expression |

### bdr.replication_set_ddl

A table that stores replication set ddl replication filters. For user queries, we recommend instead checking the `bdr.ddl_replication` view.

#### `bdr.replication_set_ddl` Columns

| Name | Type | Description |
|---|---|---|
| set_id | oid | OID of the replication set |
| set_ddl_name | name | Name of the DDL filter |
| set_ddl_tag | text | Command tag for the DDL filter |
| set_ddl_role | text | Role executing the DDL |

### bdr.replication_sets

A view showing replication sets defined in the PGD group, even if they aren't currently used by any node.

#### `bdr.replication_sets` columns

| Name | Type | Description |
|---|---|---|
| set_id | oid | OID of the replication set |
| set_name | name | Name of the replication set |
| replicate_insert | boolean | Indicates if the replication set replicates INSERTs |
| replicate_update | boolean | Indicates if the replication set replicates UPDATEs |
| replicate_delete | boolean | Indicates if the replication set replicates DELETEs |
| replicate_truncate | boolean | Indicates if the replication set replicates TRUNCATEs |
| set_autoadd_tables | boolean | Indicates if new tables are automatically added to this replication set |
| set_autoadd_seqs | boolean | Indicates if new sequences are automatically added to this replication set |

### bdr.resolved_transactions

A table storing a summary record for each failed transaction resolved by `bdr.resolve_failed_transaction`. Use the table as a history of resolution actions after truncating `bdr.pending_failed_changes` to reclaim space.

#### `bdr.resolved_transactions` columns

| Name | Type | Description |
|---|---|---|
| *id* | *bigint* | Primary key. |
| *sub_id* | *oid* | OID of the subscription. |
| *origin_node_id* | *oid* | OID of the origin node. |
| *remote_xid* | *xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | *pg_lsn* | Commit LSN of the resolved transaction on the origin node. |
| *remote_commit_end_lsn* | *pg_lsn* | LSN of the end of the commit record. |
| *remote_commit_time* | *timestamptz* | Commit timestamp on the origin node. |
| *resolution* | *text* | Resolution applied: *skipped*, *applied*, or *discarded*. |
| *insert_count* | *integer* | Number of INSERT operations in the transaction. |
| *update_count* | *integer* | Number of UPDATE operations in the transaction. |
| *delete_count* | *integer* | Number of DELETE operations in the transaction. |
| *ddl_count* | *integer* | Number of DDL operations in the transaction. |
| *error_message* | *text* | Error message from the original failure. |
| *error_detail* | *text* | Additional error detail. |
| *error_sqlstate* | *text* | SQLSTATE code of the original error. |
| *recorded_at* | *timestamptz* | Timestamp when the failure was first captured. |
| *resolved_at* | *timestamptz* | Timestamp when the transaction was resolved. |

### bdr.resolved_transactions_info

A view over `bdr.resolved_transactions` that adds human-readable subscription and origin node names, and a computed total change count.

#### `bdr.resolved_transactions_info` columns

| Name | Type | Description |
|---|---|---|
| *id* | *bigint* | Primary key. |
| *subscription_name* | *name* | Name of the subscription. |
| *origin_node* | *name* | Name of the origin node. |
| *remote_xid* | *xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | *pg_lsn* | Commit LSN of the resolved transaction on the origin node. |
| *remote_commit_time* | *timestamptz* | Commit timestamp on the origin node. |
| *resolution* | *text* | Resolution applied: *skipped*, *applied*, or *discarded*. |
| *insert_count* | *integer* | Number of INSERT operations in the transaction. |
| *update_count* | *integer* | Number of UPDATE operations in the transaction. |
| *delete_count* | *integer* | Number of DELETE operations in the transaction. |
| *ddl_count* | *integer* | Number of DDL operations in the transaction. |
| *total_changes* | *integer* | Total number of changes (*insert_count* + *update_count* + *delete_count* + *ddl_count*). |
| *error_message* | *text* | Error message from the original failure. |
| *error_sqlstate* | *text* | SQLSTATE code of the original error. |
| *recorded_at* | *timestamptz* | Timestamp when the failure was first captured. |
| *resolved_at* | *timestamptz* | Timestamp when the transaction was resolved. |

### bdr.schema_changes

A simple view to show all the changes to schemas win PGD.

#### `bdr.schema_changes` columns

| Name | Type | Description |
|---|---|---|
| schema_changes_ts | timestampstz | ID of the trigger |
| schema_changes_change | char | Flag of change type |
| schema_changes_classid | oid | Class ID |
| schema_changes_objectid | oid | Object ID |
| schema_changes_subid | smallint | Subscription |
| schema_changes_descr | text | Object changed |
| schema_changes_addr-names | text[] | Location of schema change |

### bdr.sequence_alloc

A view to see the allocation details for galloc sequences.

#### `bdr.sequence_alloc` columns

| Name | Type | Description |
|---|---|---|
| seqid | regclass | ID of the sequence |
| seq_chunk_size | bigint | A sequence number for the chunk within its value |
| seq_allocated_up_to | bigint | |

continues on next page

Table 53 – continued from previous page

| Name | Type | Description |
|---|---|---|
| seq_nallocs | bigint | |
| seq_last_alloc | timestamptz | Last sequence allocated |

### bdr.sequences

This view lists all sequences with their kind, excluding sequences for internal PGD bookkeeping.

#### `bdr.sequences` columns

| Name | Type | Description |
|---|---|---|
| nspname | name | Namespace containing the sequence |
| relname | name | Name of the sequence |
| seqkind | text | Type of the sequence ('local', 'timeshard', 'galloc') |

### bdr.stat_activity

Dynamic activity for each backend or worker process.

This contains the same information as pg_stat_activity , except `wait_event` is set correctly when the wait relates to PGD and the following Connection Manager related fields are added:

#### `bdr.stat_activity` columns

| Name | Type | Description |
|---|---|---|
| connection_manager_client_addr | inet | IP address of the client connection |
| connection_manager_client_port | int | The source port of client connected to connection manager (if the connection is done through connection manager) |
| connection_manager_client_hostname | text | Hostname of the client connection (if the connection is done through connection manager) |
| session_read_only | boolean | Whether the session is a read-only; connected to read-only port of the connection manager |

### bdr.stat_commit_scope

A view containing statistics for each commit scope.

#### `bdr.stat_commit_scope` columns

| Column | Type | Description |
|---|---|---|
| commit_scope_name | name | Name of the commit scope |
| group_name | name | Name of group for which the commit scope is defined |
| ncalls | bigint | The number of times the commit scope was used |

continues on next page

Table 56 – continued from previous page

| Column | Type | Description |
|---|---|---|
| ncommits | bigint | The number of successful commits were made with the commit scope |
| naborts | bigint | The number of times the commit scope used was eventually aborted |
| ndegrades | bigint | The number of per-transaction degrade events where a backend hit the degrade timeout during commit |
| nconfig_degrades | bigint | The number of configuration-level state changes to degraded mode by the background worker |
| last_state_change_time | timestamp with time zone | The timestamp of the last configuration-level state change (entering or recovering from degraded state) |
| total_commit_time | double precision | Total time spent committing using the commit scope, in milliseconds |
| min_commit_time | double precision | Minimum time spent committing using the commit scope, in milliseconds |
| max_commit_time | double precision | Maximum time spend committing using the commit scope, in milliseconds |
| mean_commit_time | double precision | Mean time spent committing using the commit scope, in milliseconds |
| stats_reset | timestamp with time zone | Time at which all statistics in the view were last reset |

### bdr.stat_commit_scope_state

A view of information about the current use of commit scopes by backends.

#### `bdr.stat_commit_scope_state` columns

| Column | Type | Description |
|---|---|---|
| pid | integer | Process ID of the backend |
| commit_scope_name | name | Name of the commit scope being used |
| group_name | name | Name of group for which the commit scope is defined |
| waiting_op_num | integer | Index of the first operation in the commit scope that is not satisfied yet |
| waiting_prepare_confirmations | integer | The number of PREPARE confirmations that are still needed by the operation |
| waiting_commit_confirmations | integer | The number of COMMIT confirmations that are still needed by the operation |
| waiting_lsn_confirmations | integer | The number of LSN confirmations that are still needed by the operation |

### bdr.stat_connection_manager

A view contianing statistics for the connection manager on this node.

#### `bdr.stat_connection_manager` columns

| Column | Type | Description |
|---|---|---|
| ntotal_rw_conns | bigint | Total number of read-write connections |
| ntotal_ro_conns | bigint | Total number of read-only connections |
| nactive_rw_conns | int | Number of active read-write connections |
| nactive_ro_conns | int | Number of active read-only connections |

### bdr.stat_connection_manager_connections

A view containing information about the connections to the connection manager.

#### `bdr.stat_connection_manager_connections` columns

| Column | Type | Description |
|---|---|---|
| connection_manager_client_addr | text | IP address of the client connected to the connection manager. |
| connection_manager_client_port | int | TCP port number that the client is using for communication with the connection manager. |
| connection_manager_addr | text | IP address of the connection manager node. |
| connection_manager_port | int | TCP port number that the connection manager is using to communicate with the Postgres node. |
| session_read_only | boolean | Whether the session is read-only or not. |
| client_uses_tls | boolean | Whether the client is using TLS to connect to the connection manager node, or not. |

### bdr.stat_connection_manager_node_stats

A view containing information about server connection statistics for the connection manager on this node.

#### `bdr.stat_connection_manager_node_stats` columns

| Column | Type | Description |
|---|---|---|
| node_id | oid | OID of the node |
| node_name | name | Name of the node |
| route_rw_connections | boolean | Whether read-write connections are routed to this node |
| route_ro_connections | boolean | Whether read-only connections are routed to this node |
| ntotal_rw_conns | bigint | Total number of read-write connections |
| ntotal_ro_conns | bigint | Total number of read-only connections |
| nactive_rw_conns | int | Number of active read-write connections |
| nactive_ro_conns | int | Number of active read-only connections |

### bdr.stat_connection_manager_hba_file_rules

A view that shows only the only valid and supported rules the connection manager is using from the HBA file (pg_hba.conf) and information about those rules.

#### `bdr.stat_connection_manager_hba_file_rules` columns

| Column | Type | Description |
|---|---|---|
| rule_number | integer | Rule number. This indicates the order in which each rule is considered until a match is found during authentication. |
| file_name | text | Name of the file containing this rule. |
| line_number | integer | Line number of this rule in the file referenced in file_name. |
| type | text | Type of connection. |
| database | text[] | List of database names this rule applies to. |
| user_name | text[] | List of user names this rule applies to |
| address | text | Host name or IP address, or one of all, samehost, or samenet, or null for local connections. |
| netmask | text | IP address mask, or null if not applicable. |
| auth_method | text | Authentication method. |
| auth_options | text | Options specified for authentication method, if any. |

### bdr.stat_raft_followers_state

A view of the state of the raft leader's followers on the Raft leader node (empty on other nodes).

#### `bdr.stat_raft_followers_state` columns

| Column | Type | Description |
|---|---|---|
| group_name | name | The group this information is for (each group can have a separate consensus configured). |
| node_name | name | Name of the follower node. |
| sent_commit_index | bigint | Latest Raft index sent to the follower node. |
| match_index | bigint | Raft index we expect to match the next response from the follower node. |
| last_message_time | timestamp with time zone | Last message (any, including requests) seen from the follower node. |
| last_heartbeat_send_time | timestamp with time zone | Last time the leader sent heartbeat to the follower node. |
| last_heartbeat_response_time | timestamp with time zone | Last time the leader has seen a heartbeat response from the follower node. |
| approx_clock_drift_ms | bigint | Approximate clock drift seen by the leader against the follower node in milliseconds. |

### bdr.stat_raft_state

A view describing the state of the Raft consensus on the local node.

#### `bdr.stat_raft_state` columns

| Column | Type | Description |
|---|---|---|
| group_name | name | The group this information is for (each group can have a separate consensus configured) |
| raft_stat | text | State of the local node in the Raft ('LEADER', 'CANDIDATE', 'FOLLOWER', 'STOPPED') |
| leader_name | name | Name of the Raft leader, if any |
| voted_for_name | name | The node the local node voted for as leader last vote |
| is_voting | boolean | The local node part of Raft is voting |
| heartbeat_timeout_ms | bigint | The heartbeat timeout on the local node |
| heartbeat_elapsed_ms | bigint | The number of milliseconds that have elapsed since the local node has seen a heartbeat from the leader |
| current_term | bigint | The current Raft term the local node is at |
| commit_index | bigint | The current Raft commit index the local node is at |
| apply_index | bigint | The Raft commit index the local node applied to catalogs |
| last_log_term | bigint | Last Raft term in the request log |
| last_log_index | bigint | Last Raft index in the request log |
| oldest_log_index | bigint | Oldest Raft index still in the request log |
| newest_prunable_log_index | bigint | Newest Raft index that can be safely removed from the request log |
| snapshot_term | bigint | Raft term of the last snapshot |
| snapshot_index | bigint | Raft index of the last snapshot |
| nnodes | integer | Number of nodes in the Raft consensus (should normally be the same as the number of nodes in the group) |
| nvoting_nodes | integer | Number of voting nodes in the Raft consensus |

### bdr.stat_receiver

A view containing all the necessary info about the replication subscription receiver processes.

#### `bdr.stat_receiver` columns

| Column | Type | Description |
|---|---|---|
| worker_role | text | Role of the BDR worker (always 'receiver') |
| worker_state | text | State of receiver worker (can be 'running', 'down', or 'disabled') |
| worker_pid | integer | Process id of the receiver worker |
| sub_name | name | Name of the subscription the receiver belongs to |
| sub_slot_name | name | Replication slot name used by the receiver |
| source_name | name | Source node for this receiver (the one it connects to), this is normally the same as the origin node, but is different for forward mode subscriptions |

continues on next page

Table 64 – continued from previous page

| Column | Type | Description |
|---|---|---|
| origin_name | name | The origin node for this receiver (the one it receives forwarded changes from), this is normally the same as the source node, but is different for forward mode subscriptions |
| subscription_mode | char | Mode of the subscription, see :ref:``bdr.subscription_summary` columns<*bdr.subscription_summary* columns>` for more details |
| sub_replication_sets | text[] | Replication sets this receiver is subscribed to |
| sub_apply_delay | interval | Apply delay interval |
| receive_lsn | pg_lsn | LSN of the last change received so far |
| receive_commit_lsn | pg_lsn | LSN of the last commit received so far |
| xact_apply_lsn | pg_lsn | Last applied transaction LSN |
| xact_flush_lsn | pg_lsn | Last flushed transaction LSN |
| xact_apply_timestamp | timestamp with time zone | Last applied transaction (commit) timestamp |
| worker_start | timestamp with time zone | Time at which the receiver started |
| worker_xact_start | timestamp with time zome | Time at which the receiver started local db transaction (if it is currently processing a local transaction), usually NULL, see *xact_start* in *Replicated large objects* for more details |
| worker_backend_state_change | timestamp with time zone | Backend state change timestamp, see *state_change* in *Replicated large objects* for more details |
| worker_backend_state | text | Current backend state, see *state* in *Replicated large objects* for more details |
| wait_event_type | text | Type of wait event the receiver is currently waiting on (if any), see *wait_event_type* in *Replicated large objects* for more details |
| wait_event | text | Exact event the receiver is currently waiting on (if any, see *wait_event* in *Replicated large objects* for more details) |
| connection_dsn | text | The DSN currently used by the replication receiver to connect to the source node. Reflects *Replicated large objects* when set for nodes in the same subgroup, otherwise reflects *node_dsn*. |

### bdr.stat_receiver_transactions

A view containing real-time transactions status, detailling which transactions the receiver is actively processing, which are sent to writers, and whether the transaction is being streamed to a file or to a writer. It also reports the writer's progress on transactions currently being applied.

#### `bdr.stat_receiver_transactions` columns

| Column | Type | Description |
|---|---|---|
| sub_name | name | Name of the subscription |
| remote_node | bdr.regnode | Name of the upstream node sending the transactions |
| remote_xid | xid | Transaction identifier from the upstream node |
| remote_commit_timestamp | timestamp with time zone | Transaction commit timestamp from the upstream node |
| remote_commit_lsn | pg_lsn | Transaction commit LSN from the upstream node |

continues on next page

Table 65 – continued from previous page

| Column | Type | Description |
|---|---|---|
| origin_node_name | bdr.regnode | Origin node where the transaction originally committed |
| origin_lsn | pg_lsn | LSN of the transaction on the origin node |
| txn_start_received_at | timestamp with time zone | Timestamp when the transaction was first received at the downstream node |
| txn_last_change_received_at | timestamp with time zone | Timestamp when the last change for the transaction was received at the downstream node |
| txn_last_change_received_lsn | pg_lsn | LSN corresponding to the last change received |
| txn_source | text | Source of this transaction (replay/file/stream) |
| is_streaming | text | Whether the transaction is currently streaming from the upstream |
| txn_status | text | Status of the transaction |
| commit_position | integer | Position of this transaction in the commit queue |
| writer_pid | integer | PID of the writer process applying the transaction, if applicable |
| txn_size | text | Size of the transaction (may change if the transaction is still streaming) |
| txn_apply_progress | text | Progress of the transaction, if currently being applied by the writer |
| last_change_applied_type | text | Type of the last change applied by the writer |
| last_change_applied_lsn | pg_lsn | LSN of the last change applied by the writer |

### bdr.stat_reconcile_commit_scope

Displays statistics for each commit scope. You can reset the statistics using the *bdr.stat_reset* function.

#### `bdr.stat_reconcile_commit_scope` columns

| Column | Type | Description |
|---|---|---|
| commit_scope_name | name | The name of the commit scope being monitored. |
| origin_name | name | The name of the originating node for which reconciliation is performed. |
| nreconciles | bigint | The total number of transactions processed through the reconciliation phase. |
| ncommits | bigint | The total number of successful commits completed during reconciliation for this scope and origin node. |
| naborts | bigint | The total number of transactions that were aborted during the reconciliation phase. |
| ntimeouts | bigint | The number of aborts specifically caused by a reconciliation timeout. |
| total_reconcile_time | bigint | The cumulative time spent in the reconciliation phase (in milliseconds). |
| stats_reset | timestamptz | The timestamp when the statistics for this view were last reset. |

### bdr.stat_relation

Shows apply statistics for each relation. Contains data only if tracking is enabled with *bdr.track_relation_apply* and if data was replicated for a given relation.

`lock_acquire_time` is updated only if *bdr.track_apply_lock_timing* is set to `on` (default: `off` ).

You can reset the stored relation statistics by calling *bdr.reset_relation_stats()* .

#### `bdr.stat_relation` columns

| Column | Type | Description |
|---|---|---|
| nspname | name | Name of the relation's schema |
| relname | name | Name of the relation |
| relid | oid | OID of the relation |
| total_time | double precision | Total time spent processing replication for the relation, in milliseconds |
| ninsert | bigint | Number of inserts replicated for the relation |
| nupdate | bigint | Number of updates replicated for the relation |
| ndelete | bigint | Number of deletes replicated for the relation |
| ntruncate | bigint | Number of truncates replicated for the relation |
| shared_blks_hit | bigint | Total number of shared block cache hits for the relation |
| shared_blks_read | bigint | Total number of shared blocks read for the relation |
| shared_blks_dirtied | bigint | Total number of shared blocks dirtied for the relation |
| shared_blks_written | bigint | Total number of shared blocks written for the relation |
| blk_read_time | double precision | Total time spent reading blocks for the relation, in milliseconds (if *track_io_timing* is enabled, otherwise zero) |
| blk_write_time | double precision | Total time spent writing blocks for the relation, in milliseconds (if *track_io_timing* is enabled, otherwise zero) |
| lock_acquire_time | double precision | Total time spent acquiring locks on the relation, in milliseconds (if *bdr.track_apply_lock_timing* is enabled, otherwise zero) |
| stats_reset | timestamp with time zone | Time of the last statistics reset (performed by *bdr.reset_relation_stats()*) |

### bdr.stat_routing_candidate_state

A view of information about the routing candidate nodes on the Raft leader (empty on other nodes).

#### `bdr.stat_routing_candidate_state` columns

| Column | Type | Description |
|---|---|---|
| node_group_name | name | The group this information is for (each group can have a separate routing proxy) |
| node_name | name | Candidate node name |
| node_route_fence | boolean | The node is fenced (when true it cannot become leader or read-only connection target) |

continues on next page

Table 68 – continued from previous page

| Column | Type | Description |
|---|---|---|
| node_route_reads | boolean | The node is being considered as a read-only connection target |
| node_route_writes | boolean | The node is being considered as a write lead candidate. |
| last_message_time | timestamp with time zone | The time of the last Raft message (any, including requests) seen by this node (used to check liveness of node) |

### bdr.stat_routing_state

A view of the state of the connection routing which the Connection Manager uses to route the connections.

#### `bdr.stat_routing_state` columns

| Column | Type | Description |
|---|---|---|
| node_group_name | name | The group this is information for (each group can have a separate routing proxy) |
| write_lead_name | name | Name of the write lead node |
| previous_write_lead_name | name | Name of the previous write lead node |
| read_names | name[] | Array of nodes to which read-only connections are routed |
| write_candidate_names | name[] | Nodes that match all criteria needed to become write lead in case of failover |
| read_candidate_names | name[] | Nodes that match all criteria needed to become read-only connection targets in case of failover |

### bdr.stat_subscription

Shows apply statistics for each subscription. Contains data only if tracking is enabled with *bdr.track_subscription_apply* .

You can reset the stored subscription statistics by calling *bdr.reset_subscription_stats()* .

#### `bdr.stat_subscription` columns

| Column | Type | Description |
|---|---|---|
| sub_name | name | Name of the subscription |
| subid | oid | OID of the subscription |
| mean_apply_time | double precision | Average time per apply transaction, in milliseconds |
| nconnect | bigint | Number of times this subscription has connected upstream |
| ncommit | bigint | Number of commits this subscription did |
| nabort | bigint | Number of aborts writer did for this subscription |
| nerror | bigint | Number of errors writer has hit for this subscription |
| nskippedtx | bigint | Number of transactions skipped by writer for this subscription (due to *skip_transaction* conflict resolver) |
| ninsert | bigint | Number of inserts this subscription did |
| nupdate | bigint | Number of updates this subscription did |
| ndelete | bigint | Number of deletes this subscription did |

continues on next page

Table 70 – continued from previous page

| Column | Type | Description |
|---|---|---|
| ntruncate | bigint | Number of truncates this subscription did |
| nddl | bigint | Number of DDL operations this subscription has executed |
| ndeadlocks | bigint | Number of errors that were caused by deadlocks |
| nretries | bigint | Number of retries the writer did (without going for full restart/reconnect) |
| nstream_writer | bigint | Number of transactions streamed to writer |
| nstream_file | bigint | Number of transactions streamed to file |
| nstream_commit | bigint | Number of streaming transactions committed |
| nstream_abort | bigint | Number of streaming transactions aborted |
| nstream_start | bigint | Number of STREAM START messages processed |
| nstream_stop | bigint | Number of STREAM STOP messages processed |
| nstream_commit | bigint | Number of streaming transactions committed |
| nstream_abort | bigint | Number of streaming transactions aborted |
| nstream_prepare | bigint | Number of streaming transactions prepared |
| nstream_insert | bigint | Number of streaming inserts processed |
| nstream_update | bigint | Number of streaming updates processed |
| nstream_delete | bigint | Number of streaming deletes processed |
| nstream_truncate | bigint | Number of streaming truncates processed |
| shared_blks_hit | bigint | Total number of shared block cache hits by the subscription |
| shared_blks_read | bigint | Total number of shared blocks read by the subscription |
| shared_blks_dirtied | bigint | Total number of shared blocks dirtied by the subscription |
| shared_blks_written | bigint | Total number of shared blocks written by the subscription |
| blk_read_time | double precision | Total time the subscription spent reading blocks, in milliseconds (if *track_io_timing* is enabled, otherwise zero) |
| blk_write_time | double precision | Total time the subscription spent writing blocks, in milliseconds (if *track_io_timing* is enabled, otherwise zero) |
| connect_time | timestamp with time zone | Time when the current upstream connection was established, NULL if not connected |
| last_disconnect_time | timestamp with time zone | Time when the last upstream connection was dropped |
| start_lsn | pg_lsn | LSN from which this subscription requested to start replication from the upstream |
| retries_at_same_lsn | bigint | Number of attempts the subscription was restarted from the same LSN value |
| curr_ncommit | bigint | Number of commits this subscription did after the current connection was established |
| npre_commit_confirmations | bigint | Number of precommit confirmations by CAMO partners |
| npre_commit | bigint | Number of precommits |
| ncommit_prepared | bigint | Number of prepared transaction commits |
| nabort_prepared | bigint | Number of prepared transaction aborts |
| nprovisional_waits | bigint | Number of update/delete operations on same tuples by concurrent apply transactions. These are provisional waits. See *Parallel Apply* |
| ntuple_waits | bigint | Number of update/delete operations that waited to be safely applied. See *Parallel Apply* |
| ncommit_waits | bigint | Number of fully applied transactions that had to wait before being committed. See *Parallel Apply* |

continues on next page

Table 70 – continued from previous page

| Column | Type | Description |
|---|---|---|
| neager_lock_conflicts | bigint | Number of times a writer had to wait and possibly restart while applying a Quorum Commit transaction because another writer or backend held the lock. A high value relative to *ncommit_prepared* indicates a contentious workload. |
| neager_conflicts | bigint | Number of eager conflicts encountered, excluding *neager_lock_conflicts*. These are conflicts that cause the writer to abort the transaction and send negative feedback to the origin. |
| stats_reset | timestamp with time zone | Time of the last statistics reset (performed by *bdr.reset_subscription_stats()*) |

### bdr.stat_worker

A view containing summary information and per worker statistics for PGD manager workers.

#### `bdr.stat_worker` columns

| Column | Type | Description |
|---|---|---|
| worker_role | text | Role of the BDR worker |
| worker_pid | integer | Process id of the worker |
| sub_name | name | Name of the subscription the worker is related to, if any |
| worker_start | timestamp with time zone | Time at which the worker started |
| worker_xact_start | timestamp with time zone | Time at which the worker started the local db transaction, see *xact_start* in *Replicated large objects* for more details |
| worker_xid | xid | Transaction id of the worker, see *backend_xid* in *Replicated large objects* for more details |
| worker_xmin | xid | Oldest transaction id needed by the worker, see *backend_xmin* in *Replicated large objects* for more details |
| worker_backend_state_change | timestamp with time zone | Backend state change timestamp see *state_change* in *Replicated large objects* for more details |
| worker_backend_state | text | Current backend state see *state* in *Replicated large objects* for more details |
| wait_event_type | text | The type of wait event the worker is currently waitiing on, if any (see *wait_event_type* in *Replicated large objects* for more details) |
| wait_event | text | The exact event the worker is waiting on, if any (see *wait_event* in *Replicated large objects* for more details) |
| blocked_by_pids | integer[] | List of PIDs blocking the worker, if any |
| query | text | Query currently being run by the worker |
| worker_query_start | timestamp with time zone | Timestamp at which the current query run by the worker started |

### bdr.stat_writer

A view containing summary information and statistics for each subscription replication writer. There can be multiple writers for each subscription.

### `bdr.stat_writer` columns

| Column | Type | Description |
|---|---|---|
| worker_role | text | Role of the BDR worker (always 'writer') |
| worker_state | text | State of the worker (can be 'running', 'down', or 'disabled') |
| worker_pid | integer | Process id of the writer |
| sub_name | name | Name of the subscription the writer belongs to |
| writer_nr | integer | Writer index in the writer group for the same subscription |
| nxacts | bigint | The number of transactions the writer has processed since start |
| ncommits | bigint | The number of commits the writer processed since start |
| naborts | bigint | The number of aborts the writer processed since start |
| commit_queue_position | integer | Position in the commit queue, when serializing transactions against other writers in the same writer group |
| xact_source_xid | xid | Transaction id of the currently processed transaction on the source node |
| xact_source_commit_lsn | pg_lsn | LSN of the currently processed transaction on the source node |
| xact_nchanges | bigint | The number of changes in the currently processed transaction that have been written (updated every 1000 changes) |
| xact_origin_node_name | name | Origin node of the currently processed transaction |
| xact_origin_lsn | pg_lsn | Origin LSN of the currently processed transaction |
| xact_origin_timestamp | timestamp with time zone | Origin commit timestamp of the currently processed transaction |
| streaming_allowed | boolean | The writer can receive direct stream for large transactions |
| is_streaming | boolean | The writer is currently receiving a direct stream of a large transaction |
| nstream_file | bigint | The number of stream files the writer has processed |
| nstream_writer | bigint | The number of directly streamed transactions the writer has processed |
| worker_start | timestamp with time zone | The time at which the writer started |
| worker_xact_start | timestamp with time zone | The time at which the writer start the local db transaction (see xact_start in *Replicated large objects* for more details) |
| worker_xid | xid | Transaction id of the worker (see *backend_xid* in *Replicated large objects* for more details) |
| worker_xmin | xid | Oldest transaction id needed by the worker (see *backend_xmin* in *Replicated large objects* for more details) |
| worker_backend_state_change | timestamp with time zone | Backend state change timestamp (see *state_change* in *Replicated large objects* for more details) |
| worker_backend_state | text | Current backend state (see *state* in *Replicated large objects* for more details) |
| wait_event_type | text | The type of wait event the writer is currently waiting on, if any (see *event_type* in *Replicated large objects* for more details) |
| wait_event | text | The exact event the writer is waiting on, if any (see *wait_event* in *Replicated large objects* for more details) |
| blocked_by_pids | integer[] | List of PIDs blocking the writer, if any |
| query | text | Query currently being run by the writer (normally only set for DDL) |
| worker_query_start | timestamp with time zone | Timestamp at which the current query run by the worker started |

continues on next page

Table 72 – continued from previous page

| Column | Type | Description |
|---|---|---|
| command_progress_cmd-tag | text | For commands with progress tracking, identifies the command current processed by the writer (can be one of 'CREATE INDEX', 'CREATE INDEX CONCURRENTLY', 'REINDEX', 'REINDEX CONCURRENTLY', 'CLUSTER', and 'VACUUM FULL') |
| command_progress_relation | text | For commands with progress tracking, identifies therelation which the command is working on |
| command_progress_phase | text | For commands with progress tracking, name of the current phase the command is in, refer to *Replicated large objects* in the Postgres documentation for details |
| command_progress_count | integer | For commands with progress tracking, the number of phases this command has gone through |
| command_progress_phase_nr | integer | For commands with progress tracking, the number of the phase of *command_progress_count* |
| command_progress_phase_tuples_total | real | For commands with progress tracking, the number of rows the current phase of the command has to process (if the phase is process rows) |
| command_progress_tuples_done | bigint | For commands with progress tracking, the number of rows the current phase of the command has already processed (if the phase is process rows) |

### bdr.subscription

This catalog table lists all the subscriptions owned by the local PGD node and their modes.

#### `bdr.subscription` columns

| Name | Type | Description |
|---|---|---|
| sub_id | oid | ID of the subscription |
| sub_name | name | Name of the subscription |
| nodegroup_id | oid | ID of nodegroup |
| origin_node_id | oid | ID of origin node |
| source_node_id | oid | ID of source node |
| target_node_id | oid | ID of target node |
| subscription_mode | char | Mode of subscription |
| sub_enabled | bool | Whether the subscription is enabled (should be replication) |
| apply_delay | interval | How much behind should the apply of changes on this subscription be (normally 0) |
| slot_name | name | Slot on upstream used by this subscription |
| origin_name | name | Local origin used by this subscription |
| num_writers | int | Number of writer processes this subscription uses |
| streaming_mode | char | Streaming configuration for the subscription |
| replication_sets | text[] | Replication sets replicated by this subscription (NULL = all) |
| forward_origin | text[] | Origins forwarded by this subscription (NULL = all) |

### bdr.subscription_summary

This view contains summary information about all PGD subscriptions that the local node has to other nodes.

#### `bdr.subscription_summary` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | name | Name of the PGD group the node is part of |
| sub_name | name | Name of the subscription |
| origin_name | name | Name of the origin node |
| target_name | name | Name of the target node (normally local node) |
| sub_enabled | bool | Is the subscription enabled |
| sub_slot_name | name | Slot name on the origin node used by this subscription |
| sub_replication_sets | text[] | Replication sets subscribed |
| sub_forward_origins | text[] | Does the subscription accept changes forwarded from other nodes besides the origin |
| sub_apply_delay | interval | Delay transactions by this much compared to the origin |
| sub_origin_name | name | Replication origin name used by this subscription |
| bdr_subscription_mode | char | Subscription mode |
| subscription_status | text | Status of the subscription worker |
| node_group_id | oid | OID of the PGD group the node is part of |
| sub_id | oid | OID of the subscription |
| origin_id | oid | OID of the origin node |
| target_id | oid | OID of the target node |
| receive_lsn | pg_lsn | Latest LSN of any change or message received (this can go backwards in case of restarts) |
| receive_commit_lsn | pg_lsn | Latest LSN of last COMMIT received (this can go backwards in case of restarts) |
| last_xact_replay_lsn | pg_lsn | LSN of last transaction replayed on this subscription |
| last_xact_flush_lsn | timestamptz | LSN of last transaction replayed on this subscription that's flushed durably to disk |
| last_xact_replay_timestamp | timestamptz | Timestamp of last transaction replayed on this subscription |

### bdr.subscription_error_status

A view showing the current error policy configuration and error counter state for each subscription.

#### `bdr.subscription_error_status` columns

| Name | Type | Description |
|---|---|---|
| *sub_name* | *name* | Name of the subscription. |
| *sub_enabled* | *boolean* | Whether the subscription is currently enabled. |
| *disable_reason* | *text* | The reason the subscription was disabled, or *NULL* if it's enabled. |
| *effective_error_policy* | *text* | The active error policy for the subscription, derived from the node group configuration. |
| *max_retries* | *integer* | The configured maximum number of retries before the policy takes action. |

continues on next page

Table 75 – continued from previous page

| Name | Type | Description |
|---|---|---|
| *current_retries* | *integer* | Number of retry attempts for the current or most recent failure. |
| *max_skips* | *integer* | The configured maximum number of skips, or *NULL* for unlimited. |
| *total_skips* | *integer* | Total number of transactions skipped by the *record_all_and_skip* policy. |
| *pending_changes* | *bigint* | Count of changes in *bdr.pending_failed_changes* with status *pending* for this subscription. |

### bdr.tables

This view lists information about table membership in replication sets. If a table exists in multiple replication sets, it appears multiple times in this table.

#### `bdr.tables` columns

| Name | Type | Description |
|---|---|---|
| relid | oid | OID of the relation |
| nspname | name | Name of the schema relation is in |
| relname | name | Name of the relation |
| set_name | name | Name of the replication set |
| set_ops | text[] | List of replicated operations |
| rel_columns | text[] | List of replicated columns (NULL = all columns) (*) |
| row_filter | text | Row filtering expression |
| conflict_detection | text | Conflict detection method used: row_origin (default), row_version or column_level |

(*) These columns are reserved for future use and should currently be NULL

### bdr.taskmgr_work_queue

Contains work items created and processed by task manager. The work items are created on only one node and processed on different nodes.

#### `bdr.taskmgr_work_queue` columns

| Column | Type | Description |
|---|---|---|
| ap_wq_workid | bigint | Unique ID of the work item |
| ap_wq_ruleid | int | ID of the rule listed in autopartition_rules. Rules are specified using bdr.autopartition command |
| ap_wq_relname | name | Name of the relation the task belongs to |
| ap_wq_relnamespace | name | Name of the tablespace specified in rule for this work item |
| ap_wq_partname | name | Name of the partition created by the workitem |
| ap_wq_work_category | char | Work category; can be *c* (create partition), *m* (migrate partition), *d* (drop partition), or *a* (alter partition) |
| ap_wq_work_sql | text | SQL query for the work item |

continues on next page

Table 77 – continued from previous page

| Column | Type | Description |
|---|---|---|
| ap_wq_work_depends | Oid[] | OIDs of the nodes on which the work item depends |

### bdr.taskmgr_workitem_status

The status of the work items that is updated locally on each node.

#### `bdr.taskmgr_workitem_status` columns

| Column | Type | Description |
|---|---|---|
| ap_wi_workid | bigint | ID of the work item |
| ap_wi_nodeid | Oid | OID of the node on which the work item is being processed |
| ap_wi_status | char | Status; can be *q* (queued), *c* (complete), *f* (failed), or *u* (unknown) |
| ap_wi_started_at | timestamptz | Start timestamptz of work item |
| ap_wi_finished_at | timestamptz | End timestamptz of work item |

### bdr.taskmgr_local_work_queue

Contains work items created and processed by the task manager. This is similar to `bdr.taskmgr_work_queue` , except that these work items are for locally managed tables. Each node creates and processes its own local work items, independent of other nodes in the cluster.

#### `bdr.taskmgr_local_work_queue` columns

| Column | Type | Description |
|---|---|---|
| ap_wq_workid | bigint | Unique ID of the work item |
| ap_wq_ruleid | int | ID of the rule listed in autopartition_rules. Rules are specified using bdr.autopartition command |
| ap_wq_relname | name | Name of the relation the task belongs to |
| ap_wq_relnamespace | name | Name of the tablespace specified in rule for this work item. |
| ap_wq_partname | name | Name of the partition created by the workitem |
| ap_wq_work_category | char | Category; can be *c* (create partition), *m* (migrate partition), *d* (drop partition), or *a* (alter partition) |
| ap_wq_work_sql | text | SQL query for the work item |
| ap_wq_work_depends | Oid[] | Always NULL |

### bdr.taskmgr_local_workitem_status

The status of the work items for locally managed tables.

#### `bdr.taskmgr_local_workitem_status` columns

| Column | Type | Description |
|---|---|---|
| ap_wi_workid | bigint | ID of the work item |

continues on next page

Table 80 – continued from previous page

| Column | Type | Description |
|---|---|---|
| ap_wi_nodeid | Oid | OID of the node on which the work item is being processed |
| ap_wi_status | char | Status; can be *q* (queued), *c* (complete), *f* (failed), or *u* (unknown) |
| ap_wi_started_at | timestamptz | Start timestamptz of work item |
| ap_wi_finished_at | timestamptz | End timestamptz of work item |

### bdr.trigger

In this view, you can see all the stream triggers created. Often triggers here are created from `bdr.create_conflict_trigger`.

#### `bdr.trigger` columns

| Name | Type | Description |
|---|---|---|
| trigger_id | oid | ID of the trigger |
| trigger_reloid | regclass | Name of the relating function |
| trigger_pgtgid | oid | Postgres trigger ID |
| trigger_type | char | Type of trigger call |
| trigger_name | name | Name of the trigger |

### bdr.triggers

An expanded view of `bdr.trigger` with columns that are easier to read.

| Name | Type | Description |
|---|---|---|
| trigger_name | name | Name of the trigger |
| event_manipulation | text | Operations |
| trigger_type | bdr.trigger_type | Type of trigger |
| trigger_table | bdr.trigger_reloid | Table that calls the trigger |
| trigger_function | name | Function used |

### bdr.workers

Information about running PGD worker processes.

This can be joined with `bdr.stat_activity` using pid to get even more insight into the state of PGD workers.

#### `bdr.workers` Columns

| Name | Type | Description |
|---|---|---|
| worker_pid | int | Process ID of the worker process |
| worker_role | int | Numeric representation of worker role |
| worker_role_name | text | Name of the worker role |
| worker_subid | oid | Subscription ID if the worker is associated with one |

### bdr.writers

Specific information about PGD writer processes.

#### `bdr.writers` columns

| Name | Type | Description |
|---|---|---|
| sub_name | name | Name of the subscription |
| writer_nr | int | Number of changes applied by the writer in the current transaction |
| pid | int | Process ID of the worker process |
| syncing_rel | int | OID of the relation being synchronized (if any) |
| streaming_allowed | text | Can this writer be target of direct to writer streaming |
| is_streaming | bool | Is there transaction being streamed to this writer |
| remote_xid | xid | Remote transaction id of the transaction being processed (if any) |
| remote_commit_lsn | pg_lsn | LSN of last commit processed |
| commit_queue_position | int | Position in the internal commit queue |
| nxacts | bigint | Number of transactions processed by this writer |
| ncommits | bigint | Number of transactions committed by this writer |
| naborts | bigint | Number of transactions aborted by this writer |
| nstream_file | bigint | Number of streamed-to-file transactions processed by this writer |
| nstream_writer | bigint | Number of streamed-to-writer transactions processed by this writer |
| xact_nchanges | bigint | Number of changes processed by this writer (updated every 1000 rows) |
| xact_origin_node_id | oid | Node where the transaction originated |
| xact_origin_lsn | pg_lsn | LSN of the transaction at the origin |
| xact_origin_ts | timestamp with time zone | Commit timestamp of the transaction at the origin |
| xact_apply_source | text | Method used to apply the transaction (*replay*/*file*/*stream*) |
| xact_apply_size | bigint | Size of the transaction, if known |
| xact_apply_progress | bigint | Progress of the transaction so far |
| remote_change_type | "char" | Type of the last change applied by the writer |
| remote_change_lsn | pg_lsn | LSN of the last change applied by the writer |

### bdr.worker_tasks

The `bdr.worker_tasks` view shows PGD's current worker launch rate limiting state as well as some basic statistics on background worker launch and registration activity.

Unlike the other views listed here, it isn't specific to the current database and PGD node. State for all PGD nodes on the current PostgreSQL instance is shown. Join on the current database to filter it.

`bdr.worker_tasks` doesn't track walsenders and output plugins.

#### `bdr.worker_tasks` columns

| Column | Type | Description |
|---|---|---|
| task_key_worker_role | integer | Worker role identifier |
| task_key_worker_role_name | text | Worker role name |
| task_key_dboid | oid | Database identifier, if available |
| datname | name | Name of the database, if available |
| task_key_subid | oid | Subscription identifier, if available |
| sub_name | name | Name of the subscription, if available |
| task_key_ext_libname | name | Name of the library (most likely bdr) |
| task_key_ext_funcname | name | Name of the function entry point |
| task_key_ext_workername | name | Name assigned to the worker |
| task_key_remoterelid | oid | Identifier of the remote syncing relation, if available |
| task_pid | integer | Process ID of the worker |
| task_registered | timestamp with time zone | Worker registration timestamp |
| since_registered | interval | Interval since the worker registered |
| task_attached | timestamp with time zone | Worker attach timestamp |
| since_attached | interval | Interval since the worker attached |
| task_exited | timestamp with time zone | Worker exit timestamp |
| since_exited | interval | Interval since the worker exited |
| task_success | boolean | Is worker still running? |
| task_next_launch_not_before | timestamp with time zone | Timestamp when the worker will be restarted again |
| until_launch_allowed | interval | Time remaining for next launch |
| task_last_launch_requestor_pid | integer | Process ID that requested launch |
| task_last_launch_request_time | timestamp with time zone | Timestamp when the request was made |
| since_last_request | interval | Interval since the last request |
| task_last_launch_request_approved | boolean | Did the last request succeed? |
| task_nrequests | integer | Number of requests |
| task_nregistrations | integer | Number of registrations |
| task_prev_pid | integer | Process ID of the previous generation |
| task_prev_registered | timestamp with time zone | Timestamp of the previous registered task |
| since_prev_registered | interval | Interval since the previous registration |
| task_prev_launched | timestamp with time zone | Timestamp of the previous launch |
| since_prev_launched | interval | Interval since the previous launch |
| task_prev_exited | timestamp with time zone | Timestamp when the previous task exited |
| since_prev_exited | interval | Interval since the previous task exited |
| task_first_registered | timestamp with time zone | Timestamp when the first registration happened |
| since_first_registered | interval | Interval since the first registration |

### System functions

Perform PGD management primarily by using functions you call from SQL. All functions in PGD are exposed in the `bdr` schema. Schema qualify any calls to these functions instead of putting `bdr` in the `search_path` .

### Version information functions

#### bdr.bdr_version

This function retrieves the textual representation of the version of the BDR extension currently in use.

#### bdr.bdr_version_num

This function retrieves the version number of the BDR extension that is currently in use. Version numbers are monotonically increasing, allowing this value to be used for less-than and greater-than comparisons.

The following formula returns the version number consisting of major version, minor version, and patch release into a single numerical value:

```
MAJOR_VERSION * 10000 + MINOR_VERSION * 100 + PATCH_RELEASE
```

### System information functions

#### bdr.get_relation_stats

Returns the relation information.

#### bdr.get_subscription_stats

Returns the current subscription statistics.

### System and progress information parameters

PGD exposes some parameters that you can query directly in SQL using, for example, `SHOW` or the `current_setting()` function. You can also use `PQparameterStatus` (or equivalent) from a client application.

#### bdr.local_node_id

When you initialize a session, this is set to the node id the client is connected to. This allows an application to figure out the node it's connected to, even behind a transparent proxy.

It's also used with *Connection pools and proxies* .

#### bdr.last_committed_lsn

After every `COMMIT` of an asynchronous transaction, this parameter is updated to point to the end of the commit record on the origin node. Combining it with `bdr.wait_for_apply_queue` , allows applications to perform causal reads across multiple nodes, that is, to wait until a transaction becomes remotely visible.

#### transaction_id

If a CAMO transaction is in progress, `transaction_id` is updated to show the assigned transaction id. You can query this parameter only by using using `PQparameterStatus` or equivalent, and it isn't accessible in SQL. See *Application use*

for a usage example.

### Node status functions

#### bdr.is_node_connected

#### Synopsis

```
bdr.is_node_connected(node_name name)
```

Returns boolean by checking if the walsender for a given peer is active on this node.

#### bdr.is_node_ready

#### Synopsis

```
bdr.is_node_ready(node_name name, span interval DEFAULT NULL)
```

Returns boolean by checking if the lag is lower than the given span or lower than the `timeout` for `TO ASYNC` otherwise.

### Consensus function

#### bdr.consensus_disable

#### Synopsis

```
bdr.consensus_disable(node_group_name text DEFAULT NULL)
```

Disables the consensus worker on the local node until server restart or until it's reenabled using `bdr.consensus_enable` (whichever happens first).

The `node_group_name` parameter is accepted for interface consistency but is currently a placeholder. The function disables the consensus worker database-wide, affecting all Raft instances on the node regardless of the value passed.

> **Warning**
>
> Disabling consensus disables some features of PGD and affects availability of the EDB Postgres Distributed cluster if left disabled for a long time. Use this function only when working with Technical Support.

#### bdr.consensus_enable

#### Synopsis

```
bdr.consensus_enable(node_group_name text DEFAULT NULL)
```

Re-enables a disabled consensus worker on the local node.

The `node_group_name` parameter is accepted for interface consistency but is currently a placeholder. The function re-enables the consensus worker database-wide, affecting all Raft instances on the node regardless of the value passed.

#### bdr.consensus_proto_version

Returns currently used consensus protocol version by the local node.

Needed by the PGD group reconfiguration internal mechanisms.

### bdr.consensus_snapshot_export

#### Synopsis

```
bdr.consensus_snapshot_export(version integer DEFAULT NULL, raft_instance_id integer␣
↪DEFAULT NULL)
```

Generate a new PGD consensus snapshot from the currently committed-and-applied state of the local node and return it as bytea.

By default, a snapshot for the highest supported Raft version is exported. But you can override that by passing an explicit `version` number.

The `raft_instance_id` parameter identifies which Raft instance to export the snapshot from. When `NULL` (the default), the top-level Raft instance is used. To target a specific node group's Raft instance, pass its instance ID, which you can look up via `bdr.raft_instances` .

The exporting node doesn't have to be the current Raft leader, and it doesn't need to be completely up to date with the latest state on the leader. However, `bdr.consensus_snapshot_import()` might not accept such a snapshot.

The new snapshot isn't automatically stored to the local node's `bdr.local_consensus_snapshot` table. It's only returned to the caller.

The generated snapshot might be passed to `bdr.consensus_snapshot_import()` on any other nodes in the same PGD node group that's behind the exporting node's Raft log position.

The local PGD consensus worker must be disabled for this function to work. Typical usage is:

```
SELECT bdr.bdr_consensus_disable();
COPY (SELECT * FROM bdr.consensus_snapshot_export()) TO /var/lib/postgresql/my_node_
↪consensus_snapshot.data;
SELECT bdr.bdr_consensus_enable();
```

While the PGD consensus worker is disabled:

- DDL locking attempts on the node fail or time out.
- galloc sequences don't get new values.
- Eager and CAMO transactions pause or error.
- Other functionality that needs the distributed consensus system is disrupted. The required downtime is generally very brief.

Depending on the use case, it might be practical to extract a snapshot that already exists from the `snapshot` field of the `bdr.local_consensus_snapshot` table and use that instead. Doing so doesn't require you to stop the consensus worker.

### bdr.consensus_snapshot_import

#### Synopsis

```
bdr.consensus_snapshot_import(snapshot bytea, raft_instance_id integer DEFAULT NULL,␣
↪node_name bdr.regnode DEFAULT NULL)
```

Import a consensus snapshot that was exported by `bdr.consensus_snapshot_export()` , usually from another node in the same PGD node group.

It's also possible to use a snapshot extracted directly from the `snapshot` field of the `bdr.local_consensus_snapshot` table on another node.

This function is useful for resetting a PGD node's catalog state to a known good state in case of corruption or user error.

The `raft_instance_id` parameter identifies which Raft instance to import the snapshot into. When `NULL` (the default), the top-level Raft instance is used. To target a specific node group's Raft instance, pass its instance ID, which you can look up via `bdr.raft_instances` .

The `node_name` parameter attributes the snapshot to a specific source node, typically the leader node from which the snapshot was exported. When `NULL` (the default), no source node is attributed.

You can import the snapshot if the importing node's `apply_index` is less than or equal to the snapshot-exporting node's `commit_index` when the snapshot was generated. (See `bdr.get_raft_status()` .) A node that can't accept the snapshot because its log is already too far ahead raises an error and makes no changes. The imported snapshot doesn't have to be completely up to date, as once the snapshot is imported the node fetches the remaining changes from the current leader.

The PGD consensus worker must be disabled on the importing node for this function to work. See notes on `bdr.consensus_snapshot_export()` for details.

It's possible to use this function to force the local node to generate a new Raft snapshot by running:

```
SELECT bdr.consensus_snapshot_import(bdr.consensus_snapshot_export());
```

This approach might also truncate the Raft logs up to the current applied log position.

### bdr.consensus_snapshot_verify

### Synopsis

```
bdr.consensus_snapshot_verify(snapshot bytea)
```

Verify the given consensus snapshot that was exported by `bdr.consensus_snapshot_export()` . The snapshot header contains the version with which it was generated and the node tries to verify it against the same version.

The snapshot might have been exported on the same node or any other node in the cluster. If the node verifying the snapshot doesn't support the version of the exported snapshot, then an error is raised.

### bdr.get_consensus_status

Returns status information about the current consensus (Raft) worker.

### bdr.get_raft_status

Returns status information about the current consensus (Raft) worker. Alias for `bdr.get_consensus_status` .

### bdr.raft_leadership_transfer

### Synopsis

```
bdr.raft_leadership_transfer(node_name text,
                             wait_for_completion boolean,
                             node_group_name text DEFAULT NULL)
```

Request the node identified by `node_name` to be the Raft leader. The request can be initiated from any of the PGD nodes and is internally forwarded to the current leader to transfer the leadership to the designated node. The designated node must be an ACTIVE PGD node with full voting rights.

If `wait_for_completion` is false, the request is served on a best-effort basis. If the node can't become a leader in the `bdr.raft_global_lection_timeout` period, then some other capable node becomes the leader again. Also, the leadership can change over the period of time per Raft protocol. A `true` return result indicates only that the request was submitted successfully.

If `wait_for_completion` is `true` , then the function waits until the given node becomes the new leader and possibly waits infinitely if the requested node fails to become Raft leader (for example, due to network issues). We therefore recommend that you always set a `statement_timeout` with `wait_for_completion` to prevent an infinite loop.

The `node_group_name` is optional and can be used to specify the name of the node group where the leadership transfer happens. If not specified, it defaults to NULL, which is interpreted as the top-level group in the cluster. If the `node_group_name` is specified, the function transfers leadership only within the specified node group.

### Utility functions

#### bdr.transfer_all_leaderships

Transfers write and Raft leadership away from the local node on all groups where the local node currently holds write or Raft leadership. Use this function to drain all leaderships from a node before taking it offline.

#### Synopsis

```
bdr.transfer_all_leaderships(writeleader_timeout interval DEFAULT 10s)
```

#### Parameters

| Name | Type | Default | Description |
|---|---|---|---|
| *writeleader_timeout* | interval | *'10s'* | Timeout for each write leadership transfer. The write leader is auto-selected and retried until this timeout is reached. |

#### Return value

Returns `true` if all transfers succeeded or if the node held no leaderships, or `false` if any write leader transfer timed out.

If a write leader transfer times out, the function continues processing remaining write leaders and Raft transfers on a best-effort basis, and still returns `false` even if other transfers succeeded. Raft transfers are retried indefinitely until leadership changes or the query is cancelled, respecting the `statement_timeout` configuration parameter.

The function shares logic with `bdr.routing_leadership_transfer` and `bdr.raft_leadership_transfer` , completing both operations in a single call for faster results.

#### bdr.alter_node_kind

PGD5 introduced a concept of Task Manager Leader node. The node is selected by PGD, but for upgraded clusters, it's important to set the `node_kind` properly for all nodes in the cluster. Do this manually after upgrading to the latest PGD version by calling the `bdr.alter_node_kind()` SQL function for each node.

### Synopsis

```
bdr.alter_node_kind(node_name text,
                    node_kind text);
```

### Parameters

| Parameter | Description |
|---|---|
| *node_name* | Name of the node to change kind. |
| *node_kind* | Kind of the node. |

### bdr.alter_subscription_skip_changes_upto

Because logical replication can replicate across versions, doesn't replicate global changes like roles, and can replicate selectively, sometimes the logical replication apply process can encounter an error and stop applying changes.

Wherever possible, fix such problems by making changes to the target side. `CREATE` any missing table that's blocking replication, `CREATE` a needed role, `GRANT` a necessary permission, and so on. But occasionally a problem can't be fixed that way and it might be necessary to skip entirely over a transaction. Changes are skipped as entire transactions—all or nothing. To decide where to skip to, use log output to find the commit LSN, per the example that follows, or peek the change stream with the logical decoding functions.

Unless a transaction made only one change, you often need to manually apply the transaction's effects on the target side, so it's important to save the problem transaction whenever possible, as shown in the examples that follow.

It's possible to skip over changes without `bdr.alter_subscription_skip_changes_upto` by using `pg_catalog.pg_logical_slot_get_binary_changes` to skip to the LSN of interest, so this is a convenience function. It does do a faster skip, although it might bypass some kinds of errors in logical decoding.

This function works only on disabled subscriptions.

The usual sequence of steps is:

1. Identify the problem subscription and LSN of the problem commit.
2. Disable the subscription.
3. Save a copy of the transaction using `pg_catalog.pg_logical_slot_peek_changes` on the source node, if possible.
4. `bdr.alter_subscription_skip_changes_upto` on the target node.
5. Apply repaired or equivalent changes on the target manually, if necessary.
6. Reenable the subscription.

**Warning**

It's easy to make problems worse when using this function. Don't do anything unless you're certain it's the only option.

### Synopsis

```
bdr.alter_subscription_skip_changes_upto(
  subname text,
  skip_upto_and_including pg_lsn
);
```

### Example

Apply of a transaction is failing with an error, and you've determined that lower-impact fixes such as changes on the target side can't resolve this issue. You determine that you must skip the transaction.

In the error logs, find the commit record LSN to skip to, as in this example:

```
ERROR:  XX000: CONFLICT: target_table_missing; resolver skip_if_recently_dropped␣
↪returned an error: table does not exist
CONTEXT:  during apply of INSERT from remote relation public.break_me in xact with␣
↪commit-end lsn 0/300AC18 xid 131315
committs 2021-02-02 15:11:03.913792+01 (action #2) (effective sess origin id=2 lsn=0/
↪300AC18)
while consuming I message from receiver for subscription bdr_regression_bdrgroup_
↪node1_node2 (id=2667578509)
on node node2 (id=3367056606) from upstream node node1 (id=1148549230, reporiginid=2)
```

In this portion of log, you have the information you need: the_target_lsn: **0/300AC18** the_subscription: **bdr_regression_bdrgroup_node1_node2**

Next, disable the subscription so the apply worker doesn't try to connect to the replication slot:

```
SELECT bdr.alter_subscription_disable(the_subscription);
```

You can't skip only parts of the transaction: it's all or nothing. So we strongly recommend that you save a record of it by copying it out on the provider side first, using the subscription's slot name.

```
\copy (SELECT * FROM pg_catalog.pg_logical_slot_peek_changes(the_slot_name,
    the_target_lsn, NULL, min_proto_version, 1, max_proto_version, 1,
    startup_params_format, 1, proto_format, json))
 TO transaction_to_drop.csv WITH (FORMAT csv);
```

This example is broken into multiple lines for readability, but issue it in a single line. `\copy` doesn't support multi-line commands.

You can skip the change by changing `peek` to `get` , but `bdr....skip_changes_upto` does a faster skip that avoids decoding and outputting all the data:

```
SELECT bdr.alter_subscription_skip_changes_upto(subscription_name,
    the_target_lsn);
```

You can apply the same changes (or repaired versions of them) manually to the target node, using the dumped transaction contents as a guide.

Finally, reenable the subscription:

```
SELECT bdr.alter_subscription_enable(the_subscription);
```

### bdr.connection_manager_refresh_pools

This function tells the PGD connection manager to refresh its connection pools.

This function is useful for integration with other tools or when an immediate re-evaluation of connection pools is required. This function operates only on the local node; to refresh connection managers across the entire cluster, you must execute the function on each node individually. Execution requires the `bdr_superuser` role. Expand commentComment on line R861Resolved

#### Synopsis

```
SELECT bdr.connection_manager_refresh_pools();
```

> **Note**
>
> This function may terminate existing client connections if routing changes are detected (e.g., write leader changed, nodes added/removed from routing targets, or consensus was lost).

### bdr.get_node_sub_apply_lsn

You can use this function on a subscriber to get the last LSN that was received and applied from the given origin.

#### Synopsis

```
bdr.get_node_sub_apply_lsn(node_name name)
```

#### Parameters

| Parameter | Description |
|---|---|
| *node_name* | The name of the node that's the source of the replication stream whose LSN is being retrieved. |

### bdr.get_node_sub_receive_lsn

You can use this function on a subscriber to get the last LSN that was received from the given origin. It can be either unfiltered or filtered to take into account only relevant LSN increments for transactions to be applied.

The difference between the output of this function and the output of `bdr.get_node_sub_apply_lsn()` measures the size of the corresponding apply queue.

#### Synopsis

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

#### Parameters

| Parameter | Description |
|---|---|
| *node_name* | The name of the node that's the source of the replication stream whose LSN is being retrieved. |
| *committed* | The default (true) makes this function take into account only commits of transactions received rather than the last LSN overall. This includes actions that have no effect on the subscriber node. |

### bdr.global_lock_table

This function acquires a global DML locks on a given table. See *DDL locking details*

for information about global DML lock.

### Synopsis

```
bdr.global_lock_table(relation regclass)
```

### Parameters

| Parameter | Description |
|---|---|
| *relation* | Name or oid of the relation to lock. |

### Notes

This function acquires the global DML lock independently of the `ddl_locking` setting.

The `bdr.global_lock_table` function requires `UPDATE`, `DELETE`, or `TRUNCATE` privilege on the locked `relation` unless `bdr.backwards_compatibility` is set to 30618 or lower.

### Example

```
pgddb=# SELECT bdr.local_group_slot_name();
 local_group_slot_name
- ----------------------
 bdr_pgddb_bdrgroup
```

### bdr.group_lease_override

Manually transfers or cancels node group lease ownership. Under normal operating conditions, PGD nodes transfer leases automatically. However, if a lease holder crashes, stops, or becomes disconnected from consensus, other nodes cannot acquire the lease until the current lease expires. This function bypasses that waiting period.

**Danger**

Use this function only if you are certain the previous lease holder is not accessing the resource anymore. For example, manually transferring an analytics lease while the old leader is disconnected from consensus but still active, can result in both nodes writing to the same store, causing duplicate data.

### Parameters

| Parameter | Description |
|---|---|
| *node_group* | Name of the group where the lease override is applied. |
| *node* | Name of the node that will hold the lease. |
| *duration* | Duration of the lease in seconds. Providing a value of *0* or less cancels the current lease. |
| *kind* | The lease kind (for example, *analytics*). |
| *local_only* | When *false* (default), the override is sent over consensus to all nodes. When *true*, the override only affects the local node. Use *true* only if consensus is lost. |

## bdr.local_group_slot_name

Returns the name of the group slot on the local node.

## bdr.node_group_type

Returns the type of the given node group. Returned value is the same as what was passed to `bdr.create_node_group()` when the node group was created, except `global` is returned if the `node_group_type` was passed as NULL when the group was created.

### Example

```
pgddb=# SELECT bdr.node_group_type(bdrgroup);
 node_group_type
- -----------------
 global
```

## bdr.replicate_ddl_command

Function to replicate a DDL command to a group of nodes.

### Synopsis

```
bdr.replicate_ddl_command(ddl_cmd text,
                          replication_sets text[],
                          ddl_locking text,
                          execute_locally bool)
```

### Parameters

| Parameter | Description |
|---|---|
| *ddl_cmd* | DDL command to execute. |
| *replication_sets* | An array of replication set names to apply the *ddlcommand* to. If NULL (or the function is passed only the *ddlcommand*), this parameter is set to the active PGD groups's default replication set. |

continues on next page

Table 92 – continued from previous page

| Parameter | Description |
|---|---|
| *ddl_locking* | A string that sets the *Replicated large objects* value while replicating. Defaults to the GUC value for *bdr.ddl_locking* on the local system that's running *replicate_ddl_command*. |
| *execute_locally* | A Boolean that determines whether the DDL command executes locally. Defaults to true. |

### Notes

The only required parameter of this function is `ddl_cmd` .

`bdr.replicate_ddl_command()` always replicates the command and is unaffected by the setting of *bdr.ddl_replication*

.

### bdr.run_on_all_nodes

Function to run a query on all nodes.

> **Warning**
>
> This function runs an arbitrary query on a remote node with the privileges of the user used for the internode connections as specified in the node's DSN. Use caution when granting privileges to this function.

### Synopsis

```
bdr.run_on_all_nodes(query text)
```

### Parameters

| Parameter | Description |
|---|---|
| *query* | Arbitrary query to execute. |

### Notes

This function connects to other nodes and executes the query, returning a result from each of them in JSON format. Multiple rows might be returned from each node, encoded as a JSON array. Any errors, such as being unable to connect because a node is down, are shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults are used.

This function doesn't go through normal replication. It uses direct client connection to all known nodes. By default, the connection is created with `bdr.ddl_replication = off` , since the commands are already being sent to all of the nodes in the cluster.

In PGD 6 and later, this function also sets `bdr.xact_replication=off` on the connection to ensure that transaction run locally only when the command is executed on another node.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes. Use either transparent DDL replication or `bdr.replicate_ddl_command()` to replicate DDL. DDL might be blocked in a future release.

### Example

It's useful to use this function in monitoring, for example, as in the following query:

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
      FROM bdr.node_slots
     WHERE origin_name IS NOT NULL
$$);
```

This query returns something like this on a two-node cluster:

```
[
    {
        "dsn": "host=node1 port=5432 dbname=pgddb user=postgres ",
        "node_id": "2232128708",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
                    "origin_name": "node1",
                    "target_name": "node2",
                    "local_slot_name": "bdr_pgddb_bdrgroup_node2",
                    "replay_lag_size": "0 bytes"
                }
            ]
        },
        "node_name": "node1"
    },
    {
        "dsn": "host=node2 port=5432 dbname=pgddb user=postgres ",
        "node_id": "2058684375",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
                    "origin_name": "node2",
                    "target_name": "node1",
                    "local_slot_name": "bdr_pgddb_bdrgroup_node1",
                    "replay_lag_size": "0 bytes"
                }
            ]
        },
        "node_name": "node2"
    }
]
```

### bdr.run_on_group

Function to run a query on a group of nodes.

> **Warning**
>
> This function runs an arbitrary query on remote nodes with the privileges of the user used for the internode connections as specified in the node's DSN. Use caution when granting privileges to this function.

#### Synopsis

```
bdr.run_on_group(node_group_name text, query text)
```

#### Parameters

| Parameter | Description |
|---|---|
| *node_group_name* | Name of the node group where the query is executed. |
| *query* | Arbitrary query to execute. |

#### Notes

This function connects to other nodes and executes the query, returning a result from each of them in JSON format. Multiple rows can be returned from each node, encoded as a JSON array. Any errors, such as being unable to connect because a node is down, are shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults are used.

This function doesn't go through normal replication. It uses direct client connection to all known nodes. By default, the connection is created with `bdr.ddl_replication = off` to avoid replication issues when the same replicated DDL command is sent to multiple nodes.

In PGD 6 and later, this function also sets `bdr.xact_replication=off` on the connection to ensure that transactions run locally only when the command is executed on another node.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes in the group. For global schema changes, to replicate DDL, use either transparent DDL replication or `bdr.replicate_ddl_command()`.

### bdr.run_on_nodes

Function to run a query on a specified list of nodes.

> **Warning**
>
> This function runs an arbitrary query on remote nodes with the privileges of the user used for the internode connections as specified in the node's DSN. Use caution when granting privileges to this function.

### Synopsis

```
bdr.run_on_nodes(node_names text[], query text)
```

### Parameters

| Parameter | Description |
|---|---|
| *node_names* | Text ARRAY of node names where the query is executed. |
| *query* | Arbitrary query to execute. |

### Notes

This function connects to other nodes and executes the query, returning a result from each of them in JSON format. Multiple rows can be returned from each node, encoded as a JSON array. Any errors, such as being unable to connect because a node is down, are shown in the response field. No explicit statement_timeout or other runtime parameters are set, so defaults are used.

This function doesn't go through normal replication. It uses direct client connection to all known nodes. By default, the connection is created with `bdr.ddl_replication = off` to avoid replication issues when the same replicated DDL command is sent to multiple nodes.

In PGD 6 and later, this function also sets `bdr.xact_replication=off` on the connection to ensure that transactions run locally only when the command is executed on another node.

Be careful when using this function since you risk breaking replication and causing inconsistencies between nodes. For global schema changes, to replicate DDL, use either transparent DDL replication or `bdr.replicate_ddl_command()`.

## bdr.stat_reset

Resets statistics for the specified category. Running this function clears the value of the column `stats_reset` in the *bdr.stat_reconcile_commit_scope* view.

Requires `bdr_superuser` privileges.

### Parameters

| Parameter | Description |
|---|---|
| stats_tupe | The category of statistics to reset. Valid values are *commit_scope* and *reconcile_commit_scope*. If *NULL*, the function defaults to *commit_scope*. |

## bdr.wait_for_apply_queue

The function `bdr.wait_for_apply_queue` allows a PGD node to wait for the local application of certain transactions originating from a given PGD node. It returns only after all transactions from that peer node are applied locally. An application or a proxy can use this function to prevent stale reads.

For convenience, PGD provides a variant of this function for CAMO and the CAMO partner node. See *bdr.wait_for_camo_partner_queue*

.

In case a specific LSN is given, that's the point in the recovery stream from which the peer waits. You can use this with `bdr.last_committed_lsn` retrieved from that peer node on a previous or concurrent connection.

If the given `target_lsn` is NULL, this function checks the local receive buffer and uses the LSN of the last transaction received from the given peer node, effectively waiting for all transactions already received to be applied. This is especially useful in case the peer node has failed and it's not known which transactions were sent. In this case, transactions that are still in transit or buffered on the sender side aren't waited for.

### Synopsis

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

### Parameters

| Parameter | Description |
|---|---|
| *peer_node_name* | The name of the peer node from which incoming transactions are expected to be queued and to wait for. If NULL, waits for all peer node's apply queue to be consumed. |
| *target_lsn* | The LSN in the replication stream from the peer node to wait for, usually learned by way of *bdr.last_committed_lsn* from the peer node. |

## bdr.wait_for_xid_progress

You can use this function to wait for the given transaction (identified by its XID) originated at the given node (identified by its node id) to make enough progress on the cluster. The progress is defined as the transaction being applied on a node and this node having seen all other replication changes done before the transaction is applied.

### Synopsis

```
bdr.wait_for_xid_progress(origin_node_id oid, origin_topxid int4, allnodes boolean␣
↪DEFAULT true)
```

### Parameters

| Parameter | Description |
|---|---|
| *origin_node_id* | Node id of the node where the transaction originated. |
| *origin_topxid* | XID of the transaction. |
| *allnodes* | If *true*, wait for the transaction to progress on all nodes. Otherwise, wait only for the current node. |

### Notes

You can use the function only for those transactions that replicated a DDL command because only those transactions are tracked currently. If a wrong `origin_node_id` or `origin_topxid` is supplied, the function might wait forever or until `statement_timeout` occurs.

### bdr.wait_node_confirm_lsn

Wait until a node passes a certain LSN.

This function allows you to wait until the last write on this session was replayed to one or all nodes.

Upon being called, the function waits for a node to pass a certain LSN. If no LSN is supplied, the current wal_flush_lsn (using the `pg_current_wal_flush_lsn()` function) position is used on the local node. Supplying a node name parameter tells the function to wait for that node to pass the LSN. If no node name is supplied (by passing NULL), the function waits until all the nodes pass the LSN.

We recommend using this function if you are using *Optimizing subscriber-only groups*

instead of *bdr.wait_slot_confirm_lsn*

.

This is because in an optimized topology, not all nodes have replication slots, so the function `bdr.wait_slot_confirm_lsn` might not work as expected. `bdr.wait_node_confirm_lsn` is designed to work with nodes that don't have replication slots, using alternative strategies to determine the progress of a node.

If a node is currently down, isn't updating, or simply can't be connected to, the wait will continue indefinitely. To avoid this condition, set the statement_timeout to the maximum amount of time you are prepared to wait.

#### Synopsis

```
bdr.wait_node_confirm_lsn(node_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

#### Parameters

| Parameter | Description |
|---|---|
| *node_name* | Name of the node to wait for. If NULL, waits for all nodes. |
| *target_lsn* | LSN to wait for. If NULL, uses the current wal_flush_lsn on the local node. |

#### Notes

Requires `bdr_application` privileges to use.

### bdr.wait_slot_confirm_lsn

Allows you to wait until the last write on this session was replayed to one or all nodes.

Waits until a slot passes a certain LSN. If no position is supplied, the current write position is used on the local node.

If no slot name is passed, it waits until all PGD slots pass the LSN.

The function polls every 1000 ms for changes from other nodes.

If a slot is dropped concurrently, the wait ends for that slot. If a node is currently down and isn't updating its slot, then the wait continues. You might want to set `statement_timeout` to complete earlier in that case.

If you are using *Optimizing subscriber-only groups*

, we recommend using bdr.wait_node_confirm_lsn

instead.

### Synopsis

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

### Notes

Requires `bdr_application` privileges to use.

### Parameters

| Parameter | Description |
|---|---|
| *slot_name* | Name of the replication slot to wait for. If NULL, waits for all PGD slots. |
| *target_lsn* | LSN to wait for. If NULL, uses the current write LSN on the local node. |

## Global advisory locks

PGD supports global advisory locks. These locks are similar to the advisory locks available in PostgreSQL except that the advisory locks supported by PGD are global. They follow semantics similar to DDL locks. So an advisory lock is obtained by majority consensus and can be used even if one or more nodes are down or lagging behind, as long as a majority of all nodes can work together.

Currently only EXCLUSIVE locks are supported. So if another node or another backend on the same node has already acquired the advisory lock on the object, then other nodes or backends must wait for the lock to be released.

Advisory lock is transactional in nature. So the lock is released when the transaction ends unless you explicitly release it before the end of the transaction. In this case, it becomes available as soon as it's released. Session-level advisory locks aren't currently supported.

Global advisory locks are reentrant. So if the same resource is locked three times, you must then unlock it three times to release it for use in other sessions.

### bdr.global_advisory_lock

This function acquires an EXCLUSIVE lock on the provided object. If the lock isn't available, then it waits until the lock becomes available or the `bdr.global_lock_timeout` is reached.

### Synopsis

```
bdr.global_advisory_lock(key bigint)
```

### parameters

- `key` — The object on which an advisory lock is acquired.

### Synopsis

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

### Parameters

| Parameter | Description |
|---|---|
| *key1* | First part of the composite key. |
| *key2* | Second part of the composite key. |

### bdr.global_advisory_unlock

This function releases a previously acquired lock on the application-defined source. The lock must have been obtained in the same transaction by the application. Otherwise, an error is raised.

### Synopsis

```
bdr.global_advisory_unlock(key bigint)
```

### Parameters

| Parameter | Description |
|---|---|
| *key* | The object on which an advisory lock is acquired. |

### Synopsis

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

### Parameters

| Parameter | Description |
|---|---|
| *key1* | First part of the composite key. |
| *key2* | Second part of the composite key. |

## Monitoring functions

### bdr.monitor_group_versions

To provide a cluster-wide version check, this function uses PGD version information returned from the view `bdr.group_version_details`.

### Synopsis

```
bdr.monitor_group_versions()
```

### Notes

This function returns a record with fields `status` and `message`, as explained in *Monitoring Parallel Apply*.

This function calls `bdr.run_on_all_nodes()`.

### bdr.monitor_group_raft

To provide a cluster-wide Raft check, this function uses PGD Raft information returned from the view `bdr.group_raft_details`.

#### Synopsis

```
bdr.monitor_group_raft()
```

#### Parameters

| Parameter | Description |
|---|---|
| *node_group_name* | The node group name to check. |

#### Notes

This function returns a record with fields `status` and `message`, as explained in *Monitoring Parallel Apply*.

This function calls `bdr.run_on_all_nodes()`.

### bdr.monitor_local_replslots

This function uses replication slot status information returned from the view `pg_replication_slots` (slot active or inactive) to provide a local check considering all replication slots except the PGD group slots.

This function also provides status information on subscriber-only nodes that are operating as subscriber-only group leaders in a PGD cluster when *Optimizing subscriber-only groups* is enabled.

#### Synopsis

```
bdr.monitor_local_replslots()
```

#### Notes

This function returns a record with fields `status` and `message`.

| Status | Message |
|---|---|
| UNKNOWN | This node is not part of any BDR group |
| OK | All BDR replication slots are working correctly |
| OK | This node is part of a subscriber-only group |
| CRITICAL | There is at least 1 BDR replication slot which is inactive |
| CRITICAL | There is at least 1 BDR replication slot which is missing |

Further explaination is available in *Monitoring replication slots*.

### bdr.wal_sender_stats

If the *Decoding worker* is enabled, this function shows information about the decoder slot and current logical change record (LCR) segment file being read by each WAL sender.

### Synopsis

```
bdr.wal_sender_stats()
```

### Output columns

| Column name | Description |
|---|---|
| *pid* | PID of the WAL sender. (Corresponds to the *pid* column of *pg_stat_replication*). |
| *is_using_lcr* | Whether the WAL sender is sending LCR files. |
| *decoder_slot_name* | Name of the decoder replication slot. |
| *lcr_file_name* | Name of the current LCR file. |

## bdr.get_decoding_worker_stat

If the *Decoding worker* is enabled, this function shows information about the state of the decoding worker associated with the current database. This also provides more granular information about decoding worker progress than is available via `pg_replication_slots` .

### Synopsis

```
bdr.get_decoding_worker_stat()
```

### Output columns

| Column name | Description |
|---|---|
| *pid* | The PID of the decoding worker. (Corresponds to the column *active_pid* in *pg_replication_slots*.) |
| *decoded_upto_lsn* | LSN up to which the decoding worker read transactional logs. |
| *waiting* | Whether the decoding worker is waiting for new WAL. |
| *waiting_for_lsn* | The LSN of the next expected WAL. |

### Notes

For details, see *Monitoring WAL senders using LCR* .

## bdr.lag_control

If *LAG CONTROL* is enabled, this function shows information about the commit delay and number of nodes conforming to their configured lag measure for the local node and current database.

### Synopsis

```
bdr.lag_control()
```

### Output columns

| Column name | Description |
|---|---|
| *commit_scope_id* | OID of the commit scope (see :ref:``bdr.commit_scopes` columns<*bdr.commit_scopes* columns>` ). |
| *sessions* | Number of sessions referencing the lag control entry. |
| *current_commit_delay* | Current runtime commit delay, in fractional milliseconds. |
| *maximum_commit_delay* | Configured maximum commit delay, in fractional milliseconds. |
| *commit_delay_adjust* | Change to runtime commit delay possible during a sample interval, in fractional milliseconds. |
| *current_conforming_nodes* | Current runtime number of nodes conforming to lag measures. |
| *minimum_conforming_nodes* | Configured minimum number of nodes required to conform to lag measures, below which a commit delay adjustment is applied. |
| *lag_bytes_threshold* | Lag size at which a commit delay is applied, in kilobytes. |
| *maximum_lag_bytes* | Configured maximum lag size, in kilobytes. |
| *lag_time_threshold* | Lag time at which a commit delay is applied, in milliseconds. |
| *maximum_lag_time* | Configured maximum lag time, in milliseconds. |
| *sample_interval* | Configured minimum time between lag samples and possible commit delay adjustments, in milliseconds. |

### Routing functions

#### bdr.routing_leadership_transfer

Changing the routing leader transfers the leadership of the node group to another node. If `leader_name` is `NULL` , the system automatically selects the most appropriate node based on write leader progress.

### Synopsis

```
bdr.routing_leadership_transfer(node_group_name text,
              leader_name text DEFAULT NULL,
              transfer_method text DEFAULT strict,
              transfer_timeout interval DEFAULT 10s);
```

### Parameters

| Name | Type | Default | Description |
|---|---|---|---|
| *node_group_name* | text | | Name of group where the leadership transfer is requested. |
| *leader_name* | text | *NULL* | Name of node that will become write leader. If NULL, the system automatically selects the most appropriate candidate based on write leader progress. |
| *transfer_method* | text | *'strict'* | Type of the transfer. It can be *'fast'* or the default, *'strict'*, which checks the maximum lag. |

continues on next page

Table 109 – continued from previous page

| Name | Type | Default | Description |
|---|---|---|---|
| *transfer_timeout* | interval | *'10s'* | Timeout of the leadership transfer. Default is 10 seconds. When using auto-selection (*leader_name* is NULL), the system retries selecting the furthest-progressed node until the timeout is reached. |

### Notes

When `leader_name` is `NULL` , the function selects the best candidate node by identifying the node with the furthest write leader progress and retrying selection until a suitable candidate is available or the timeout is reached. Automatic selection is useful for automated failover scenarios where manual node selection isn't required. When using Quorum Commit as the commit scope kind, pass `NULL` for `leader_name` to trigger automatic selection of the most appropriate node.

### Selecting a candidate automatically

During automatic leader selection (when `leader_name` is `NULL` ):

- Route-fenced nodes (nodes with `route_fence` enabled via *bdr.alter_node_option* ) aren't considered candidates for leadership transfer.
- Nodes with `route_writes` set to `false` aren't considered candidates for leadership transfer.
- The `route_priority` option doesn't influence automatic selection. Selection is based solely on write leader progress among eligible nodes.

### Running in a mixed-version cluster

Automatic leader selection (passing `NULL` for `leader_name` ) requires PGD 6.4 and Raft protocol version 6004 or above.

When calling the function on pre-6.4 nodes, the function exits without performing any action, as the older function signature requires a non-`NULL` `leader_name` parameter.

When calling the function on 6.4 nodes, the behavior depends on the Raft protocol version:

- If Raft protocol version is below 6004 and a PGD 6.4 or above node is the Raft leader, the function returns `ERROR: leadership auto transfer is supported from raft protocol version 6004 and above` .
- If a pre-6.4 node is the Raft leader, the operation times out and returns `ERROR: switchover timed out` .

For reliable automatic leader selection, ensure all nodes in the cluster are running PGD 6.4 or later with Raft protocol version 6004 or above.

### CAMO functions

CAMO requires that a client actively participates in the committing of a transaction by following the transactions progress. The functions listed here are used for that purpose and explained in *Commit At Most Once* .

### bdr.is_camo_partner_connected

Allows checking of the connection status of a CAMO partner node configured in pair mode. There currently is no equivalent for CAMO used with eager replication.

#### Synopsis

```
bdr.is_camo_partner_connected()
```

#### Return value

A Boolean value indicating whether the CAMO partner is currently connected to a WAL sender process on the local node and therefore can receive transactional data and send back confirmations.

### bdr.is_camo_partner_ready

Allows checking of the readiness status of a CAMO partner node configured in pair mode. Underneath, this triggers the switch to and from local mode.

#### Synopsis

```
bdr.is_camo_partner_ready()
```

#### Return value

A Boolean value indicating whether the CAMO partner can reasonably be expected to confirm transactions originating from the local node in a timely manner, that is, before `timeout` for `TO ASYNC` expires.

> **Note**
>
> This function queries the past or current state. A positive return value doesn't indicate whether the CAMO partner can confirm future transactions.

### bdr.get_configured_camo_partner

This function shows the local node's CAMO partner (configured by pair mode).

#### Synopsis

```
bdr.get_configured_camo_partner()
```

### bdr.wait_for_camo_partner_queue

The function is a wrapper around `bdr.wait_for_apply_queue` defaulting to query the CAMO partner node. It returns an error if the local node isn't part of a CAMO pair.

### Synopsis

```
bdr.wait_for_camo_partner_queue()
```

### bdr.camo_transactions_resolved

This function begins a wait for CAMO transactions to be fully resolved.

### Synopsis

```
bdr.camo_transactions_resolved()
```

### bdr.logical_transaction_status

To check the status of a transaction that was being committed when the node failed, the application must use this function, passing as parameters the node id of the node the transaction originated from and the transaction id on the origin node.

### Synopsis

```
bdr.logical_transaction_status(node_id OID, xid OID,
                               require_camo_partner boolean DEFAULT true)
```

### Parameters

| Parameter | Description |
|---|---|
| *node_id* | The node id of the PGD node the transaction originates from, usually retrieved by the client before *COMMIT* from the *Replicated large objects* *bdr.local_node_id*. |
| *xid* | The transaction id on the origin node, usually retrieved by the client before *COMMIT* from the *Replicated large objects* *transaction_id*. |
| *require_camo_partner* | Defaults to true and enables configuration checks. Set to false to disable these checks and query the status of a transaction that wasn't a CAMO transaction. |

### Return value

The function returns one of these results:

- `'committed'::TEXT` — The transaction was committed, is visible on both nodes of the CAMO pair, and is eventually replicated to all other PGD nodes. No need for the client to retry it.
- `'aborted'::TEXT` — The transaction was aborted and isn't replicated to any other PGD node. The client needs to either retry it or escalate the failure to commit the transaction.
- `'in progress'::TEXT` — The transaction is still in progress on this local node and wasn't committed or aborted yet. The transaction might be in the COMMIT phase, waiting for the CAMO partner to confirm or deny the commit. The recommended client reaction is to disconnect from the origin node and reconnect to the CAMO partner to query that instead. With a load balancer or proxy in between, where the client lacks control over which node gets queried, the client can only poll repeatedly until the status switches to either `'committed'` or `'aborted'`.

For eager all-node replication, peer nodes yield this result for transactions that aren't yet committed or aborted. Even transactions not yet replicated (or not even started on the origin node) might yield an `in progress` result on a peer

PGD node in this case. However, the client must not query the transaction status prior to attempting to commit on the origin.

- `'unknown'::TEXT` — The transaction specified is unknown because it's either in the future, not replicated to that specific node yet, or too far in the past. The status of such a transaction isn't yet or is no longer known. This return value is a sign of improper use by the client.

The client must be prepared to retry the function call on error.

### Commit Scope functions

#### bdr.add_commit_scope

**Deprecated**. Use bdr.create_commit_scope instead. Previously, this function was used to add a commit scope to a node group. It's now deprecated and will emit a warning until it is removed in a future release, at which point it will raise an error.

#### bdr.create_commit_scope

`bdr.create_commit_scope` creates a rule for the given commit scope name and origin node group. If the rule is the same for all nodes in the EDB Postgres Distributed cluster, invoking this function once for the top-level node group is enough to fully define the commit scope.

Alternatively, you can invoke it multiple times with the same `commit_scope_name` but different origin node groups and rules for commit scopes that vary depending on the origin of the transaction.

#### Synopsis

```
bdr.create_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT,
    wait_for_ready boolean DEFAULT true)
```

#### Note

`bdr.create_commit_scope` replaces the deprecated bdr.add_commit_scope function. Unlike `add_commit_scope`, it doesn't silently overwrite existing commit scopes when the same name is used. Instead, an error is reported.

#### bdr.alter_commit_scope

`bdr.alter_commit_scope` allows you to change a specific rule for a single origin node group in a commit scope.

#### Synopsis

```
bdr.alter_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT)
```

### bdr.drop_commit_scope

Drops a single rule in a commit scope. If you define multiple rules for the commit scope, you must invoke this function once per rule to fully remove the entire commit scope.

#### Synopsis

```
bdr.drop_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME)
```

> **Note**
>
> Dropping a commit scope that's still used as default by a node group isn't allowed.

### bdr.remove_commit_scope

**Deprecated**. Use bdr.drop_commit_scope instead. Previously, this function was used to remove a commit scope from a node group. It's now deprecated and will emit a warning until it is removed in a future release, at which point it will raise an error.

### Apply error policy functions

These functions let you review, resolve, and analyze replication failures recorded by the apply error policy. See the `apply_error_policy` , `apply_error_max_retries` , `apply_error_max_skips` , and `apply_error_max_record_size` options in *Group configuration using bdr.alter_node_group_option()* for configuration details. For a step-by-step guide to configuring a policy and resolving failures, see *Handling replication apply errors* .

### bdr.skip_failed_change

Marks a single captured change in `bdr.pending_failed_changes` as skipped. The change status is updated to `skipped` but the row isn't deleted.

#### Synopsis

```
bdr.skip_failed_change(change_id bigint)
```

#### Parameters

| Parameter | Description |
|---|---|
| *change_id* | Primary key of the row in *bdr.pending_failed_changes* to skip. |

### bdr.skip_failed_transaction

Marks all pending changes in a failed transaction as skipped and optionally re-enables the subscription.

### Synopsis

```
bdr.skip_failed_transaction(
    subscription_name name,
    remote_xid xid,
    remote_commit_lsn pg_lsn,
    dry_run boolean DEFAULT false,
    re_enable boolean DEFAULT false
)

bdr.skip_failed_transaction(
    failed_txn_id uuid,
    dry_run boolean DEFAULT false,
    re_enable boolean DEFAULT false
)
```

### Parameters

| Parameter | Description |
|---|---|
| *subscription_name* | Name of the subscription. |
| *remote_xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | Commit LSN of the failed transaction on the origin node. |
| *failed_txn_id* | UUID of the failed transaction from *bdr.pending_failed_changes*. Use as a convenient alternative to specifying *subscription_name*, *remote_xid*, and *remote_commit_lsn*. |
| *dry_run* | When *true*, returns the count of changes that would be skipped without modifying anything. Default is *false*. |
| *re_enable* | When *true*, re-enables the subscription after skipping. Default is *false*. |

### Return value

Returns the number of changes skipped. Returns `0` if the transaction was already resolved (no error is raised).

### bdr.apply_failed_change

Retries applying a single captured change.

### Synopsis

```
bdr.apply_failed_change(change_id bigint)
```

### Parameters

| Parameter | Description |
|---|---|
| *change_id* | Primary key of the row in *bdr.pending_failed_changes* to apply. |

### bdr.apply_failed_transaction

Retries all pending changes for a failed transaction. Defaults to dry-run mode so you can preview the outcome before committing.

### Synopsis

```
bdr.apply_failed_transaction(
    subscription_name name,
    remote_xid xid,
    remote_commit_lsn pg_lsn,
    dry_run boolean DEFAULT true,
    re_enable boolean DEFAULT false
)

bdr.apply_failed_transaction(
    failed_txn_id uuid,
    dry_run boolean DEFAULT true,
    re_enable boolean DEFAULT false
)
```

### Parameters

| Parameter | Description |
|---|---|
| *subscription_name* | Name of the subscription. |
| *remote_xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | Commit LSN of the failed transaction on the origin node. |
| *failed_txn_id* | UUID of the failed transaction from *bdr.pending_failed_changes*. Use as a convenient alternative to specifying *subscription_name*, *remote_xid*, and *remote_commit_lsn*. |
| *dry_run* | When *true* (the default), shows what would happen without actually applying changes (rolls back at the end). Pass *false* to execute. |
| *re_enable* | When *true* and *dry_run* is *false*, re-enables the subscription after applying. Default is *false*. |

### Return value

Returns one row per change with the following columns.

| Column | Type | Description |
|---|---|---|
| *change_num* | *integer* | Order of the change within the transaction. |
| *change_id* | *bigint* | Primary key of the row in *bdr.pending_failed_changes*. |
| *operation* | *text* | Operation type: *INSERT*, *UPDATE*, *DELETE*, *SQL*, *DDL*, or *TRUNCATE*. |
| *schema_name* | *text* | Schema of the target table. |

continues on next page

Table 115 – continued from previous page

| Column | Type | Description |
|---|---|---|
| *table_name* | *text* | Name of the target table. |
| *status* | *text* | Result of the apply attempt. In dry-run mode, values are *would_apply*, *would_skip*, or *would_error*. When executed, values are *apply*, *skip*, or *error*. |
| *message* | *text* | Error message if the change failed, or *NULL* on success. |

If an error occurs during apply, the function returns results for all changes up to and including the failed one. If the transaction was already resolved, the function returns 0 rows without raising an error.

### bdr.modify_failed_change

Edits the column values of a single captured change before retrying the change. Updates the `modified_values` column in `bdr.pending_failed_changes` and sets the change status to `modified` .

#### Synopsis

```
bdr.modify_failed_change(change_id bigint, new_values jsonb)
```

#### Parameters

| Parameter | Description |
|---|---|
| *change_id* | Primary key of the row in *bdr.pending_failed_changes* to modify. |
| *new_values* | JSONB object containing the updated column values to use when the change is retried. |

### bdr.reconstruct_change_sql

Generates executable SQL for a single captured change. The `bdr.pending_failed_changes_sql` view uses this function internally.

#### Synopsis

```
bdr.reconstruct_change_sql(
    operation text,
    schema_name name,
    table_name name,
    key_columns jsonb,
    old_values jsonb,
    new_values jsonb
)
```

### Parameters

| Parameter | Description |
|---|---|
| *operation* | Operation type. Valid values are *INSERT*, *UPDATE*, *DELETE*, *SQL*, *DDL*, and *TRUNCATE*. |
| *schema_name* | Schema of the target table. |
| *table_name* | Name of the target table. |
| *key_columns* | JSONB object containing the primary key column values. |
| *old_values* | JSONB object containing the before-image column values, used for *UPDATE* and *DELETE* operations. |
| *new_values* | JSONB object containing the after-image column values for *INSERT* and *UPDATE* operations, or the SQL text for *DDL* and *SQL* operations. |

### Return value

Returns a `text` string containing the reconstructed SQL statement. DDL operations include a `SET search_path` prefix when a search path was in effect during the original execution.

## bdr.reconstruct_failed_transaction

Generates a complete SQL script for all captured changes in a failed transaction, wrapped in `BEGIN` and `COMMIT` .

### Synopsis

```
bdr.reconstruct_failed_transaction(
    subscription_name name,
    remote_xid xid,
    remote_commit_lsn pg_lsn
)

bdr.reconstruct_failed_transaction(failed_txn_id uuid)
```

### Parameters

| Parameter | Description |
|---|---|
| *subscription_name* | Name of the subscription. |
| *remote_xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | Commit LSN of the failed transaction on the origin node. |
| *failed_txn_id* | UUID of the failed transaction from *bdr.pending_failed_changes*. Use as a convenient alternative to specifying *subscription_name*, *remote_xid*, and *remote_commit_lsn*. |

### Return value

Returns a `text` SQL script containing all changes in the transaction. Use the script to review or manually replay the complete transaction.

### bdr.resolve_failed_transaction

Marks all pending changes in a failed transaction as resolved, advances the subscription's position past the failed transaction, and optionally re-enables the subscription. Defaults to dry-run mode so you can preview the outcome before committing.

### Synopsis

```
bdr.resolve_failed_transaction(
    subscription_name name,
    remote_xid xid,
    remote_commit_lsn pg_lsn,
    resolution text DEFAULT skipped,
    re_enable boolean DEFAULT false,
    dry_run boolean DEFAULT true
)

bdr.resolve_failed_transaction(
    failed_txn_id uuid,
    resolution text DEFAULT skipped,
    re_enable boolean DEFAULT false,
    dry_run boolean DEFAULT true
)
```

### Parameters

| Parameter | Description |
|---|---|
| *subscription_name* | Name of the subscription. |
| *remote_xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | Commit LSN of the failed transaction on the origin node. |
| *failed_txn_id* | UUID of the failed transaction from *bdr.pending_failed_changes*. Use as a convenient alternative to specifying *subscription_name*, *remote_xid*, and *remote_commit_lsn*. |
| *resolution* | Resolution action to apply. Valid values are *skipped*, *applied*, and *discarded*. Default is *skipped*. |
| *re_enable* | When *true*, re-enables the subscription after resolving. Default is *false*. |
| *dry_run* | When *true* (the default), returns a detailed preview of what the operation would do without making any changes. Pass *false* to execute. |

### Return value

In dry-run mode, returns a `text` report describing the planned changes. When `dry_run` is `false` , returns a `text` confirmation of the resolution applied.

### Notes

Calling this function also inserts a summary record into `bdr.resolved_transactions` . Once resolved, you can truncate `bdr.pending_failed_changes` to reclaim space while retaining a history of resolutions.

If the transaction was already resolved, the function raises an error with details about the prior resolution.

### bdr.analyze_skip_impact

Analyzes the safety and potential data impact of skipping a single captured change. Returns a risk assessment based on the error type, operation, and optionally the current local database state.

### Synopsis

```
bdr.analyze_skip_impact(
    change_id bigint,
    verify_state boolean DEFAULT true,
    include_transaction_analysis boolean DEFAULT true
)
```

### Parameters

| Parameter | Description |
|---|---|
| *change_id* | Primary key of the row in *bdr.pending_failed_changes* to analyze. |
| *verify_state* | When *true* (the default), queries the local database to verify current state before assessing safety. Pass *false* for a faster static analysis based on error code and operation type only. |
| *include_transaction_analysis* | When *true* (the default), examines all other changes in the same transaction to detect dependencies and generate warnings. |

### Return value

Returns a row with the following columns.

| Column | Type | Description |
|---|---|---|
| *skip_safety* | *text* | Safety classification: *safe*, *caution*, or *risky*. |
| *recommendation* | *text* | Human-readable recommendation. |
| *explanation* | *text* | Detailed explanation including error context and any state verification results. |
| *data_impact* | *text* | Data impact level: *none*, *possible_divergence*, or *certain_divergence*. |
| *suggested_action* | *text* | Suggested action: *skip*, *apply_modified*, or *manual_review*. |
| *dependency_warnings* | *text[]* | Array of warnings about related changes in the transaction that will also be affected, or *NULL* if none. |

### Safety classifications

| Safety | Description |
|---|---|
| *safe* | Skipping the change is unlikely to cause data divergence between nodes. |
| *caution* | Minor potential issues exist. Review the change before skipping. |
| *risky* | Skipping the change will likely cause data divergence between nodes. |

## bdr.analyze_transaction_skip_impact

Analyzes the skip safety for all pending changes in a failed transaction, returning one row per change.

### Synopsis

```
bdr.analyze_transaction_skip_impact(
    subscription_name name,
    remote_xid xid,
    remote_commit_lsn pg_lsn
)
```

### Parameters

| Parameter | Description |
|---|---|
| *subscription_name* | Name of the subscription. |
| *remote_xid* | Transaction ID on the origin node. |
| *remote_commit_lsn* | Commit LSN of the failed transaction on the origin node. |

### Return value

Returns one row per change in the transaction, with the following columns.

| Column | Type | Description |
|---|---|---|
| *change_seq* | *integer* | Order of the change within the transaction. |
| *operation* | *text* | Operation type: *INSERT*, *UPDATE*, *DELETE*, *SQL*, *DDL*, or *TRUNCATE*. |
| *qualified_table* | *text* | Schema-qualified table name. |
| *skip_safety* | *text* | Safety classification: *safe*, *caution*, or *risky*. |
| *recommendation* | *text* | Human-readable recommendation for the change. |
| *has_dependents* | *boolean* | Whether other changes in the transaction may depend on the outcome of this change. |

### bdr.format_skip_analysis

Returns a formatted, human-readable analysis of the skip safety for a single captured change.

#### Synopsis

```
bdr.format_skip_analysis(change_id bigint)
```

#### Parameters

| Parameter | Description |
|---|---|
| *change_id* | Primary key of the row in *bdr.pending_failed_changes* to analyze. |

#### Return value

Returns a `text` report including the safety classification, recommendation, explanation, and any state verification results.

### bdr.alter_subscription_reset_error_counters

Resets the error retry and skip counters for a subscription to their initial values. Use the reset after resolving replication failures to allow the subscription's error policy thresholds to apply fresh.

#### Synopsis

```
bdr.alter_subscription_reset_error_counters(subscription_name name)
```

#### Parameters

| Parameter | Description |
|---|---|
| *subscription_name* | Name of the subscription. |

## PGD settings

You can set PGD-specific configuration settings. Unless noted otherwise, you can set the values at any time.

### Conflict handling

#### bdr.default_conflict_detection

Sets the default conflict detection method for newly created tables. Accepts same values as *bdr.alter_table_conflict_detection()* .

### Global sequence parameters

#### bdr.default_sequence_kind

Sets the default *Sequences* .

The default is `distributed` , which means `snowflakeid` is used for `int8` sequences (that is, `bigserial` ) and `galloc` sequence for `int4` (that is, `serial` ) and `int2` sequences.

### DDL handling

#### bdr.default_replica_identity

Sets the default value for `REPLICA IDENTITY` on newly created tables. The `REPLICA IDENTITY` defines the information written to the write-ahead log to identify rows that are updated or deleted.

The accepted values are:

| Value | Description |
|---|---|
| *default* | Records the old values of the columns of the primary key, if any (this is the default PostgreSQL behavior). |
| *full* | Records the old values of all columns in the row. |
| *nothing* | Records no information about the old row. |
| *auto* | Tables are created with REPLICA IDENTITY FULL. This is the default PGD behavior. |

See the PostgreSQL documentation for more details.

PGD can't replicate `UPDATE` and `DELETE` operations on tables without a `PRIMARY KEY` or `UNIQUE` constraint. The exception is when the replica identity for the table is `FULL` , either by table-specific configuration or by `bdr.default_replica_identity` .

If `bdr.default_replica_identity` is `default` and there is a `UNIQUE` constraint included in the table definition, it won't be automatically picked up as `REPLICA IDENTITY` . You need to set the `REPLICA IDENTITY` explicitly using `ALTER TABLE ... REPLICA IDENTITY ...` .

Setting the replica identity of tables to `full` increases the volume of WAL written and the amount of data replicated on the wire for the table.

!!!warning On setting bdr.default_replica_identity to default

When setting `bdr.default_replica_identity` to `default` using `ALTER SYSTEM` , always quote the value, like this:

```
ALTER SYSTEM SET bdr.default_replica_identity="default";
```

You need to include the quotes because default, unquoted, is a special value to the ALTER SYSTEM command that triggers the removal of the setting from the configuration file. When the setting is removed, the system uses the PGD default setting, which is `auto` .

Automatically replicates DDL across nodes (default is `on` ).

This parameter can be set only by bdr_superuser or superuser roles.

Running DDL or calling PGD administration functions with `bdr.ddl_replication = off` can create situations where replication stops until an administrator can intervene. See *Running DDL safely* for details.

A `LOG` -level log message is emitted to the PostgreSQL server logs whenever `bdr.ddl_replication` is set to `off` . Additionally, a `WARNING`-level message is written whenever replication of captured DDL commands or PGD replication functions is skipped due to this setting.

### bdr.role_replication

Automatically replicates ROLE commands across nodes (default is `on` ). Only a superuser can set this parameter. This setting works only if `bdr.ddl_replication` is turned on as well.

Turning this parameter off without using external methods to ensure roles are in sync across all nodes might cause replicated DDL to interrupt replication until the administrator intervenes.

See *DDL and role manipulation statements*

for details.

### bdr.ddl_locking

Configures the operation mode of global locking for DDL.

This parameter can be set only by bdr_superuser or superuser roles.

Possible options are:

| Value | Description |
|---|---|
| *all* | Use global locking for all DDL operations. |
| *leader* | Use leader-based global DML locking. |
| *auto* | Selects *leader* if possible, otherwise *all*. (Default) |
| *dml* | Use global locking only for DDL operations that need to prevent writes by taking the global DML lock for a relation. |
| *off* | Don't use global locking for DDL operations. |

Default is `auto` .

A `LOG` -level log message is emitted to the PostgreSQL server logs whenever `bdr.ddl_locking` is set to `off` . Additionally, a `WARNING` message is written whenever any global locking steps are skipped due to this setting. Some statements result in two `WARNING` messages: one for skipping the DML lock and one for skipping the DDL lock.

For backward compatibility, `bdr.ddl_locking` supports aliases. `on` and `true` are an alias for `all` . `false` is an alias for `off` .

See also *Global locking* .

### bdr.truncate_locking

Sets the TRUNCATE command's locking behavior (default is `on` /`true` ). When `on` /`true` , TRUNCATE obeys the `bdr.ddl_locking` setting.

## Global locking

DDL locking is controlled by *bdr.ddl_locking* . Other global locking settings include the following.

### bdr.global_lock_max_locks

Sets the maximum number of global locks that can be held on a node (default is 1000). Can be set only at Postgres server start.

### bdr.global_lock_timeout

Sets the maximum allowed duration of any wait for a global lock (default is 1 minute). A value of zero disables this timeout.

### bdr.global_lock_statement_timeout

Sets the maximum allowed duration of any statement holding a global lock (default is 60 minutes). A value of zero disables this timeout.

### bdr.global_lock_idle_timeout

Sets the maximum allowed duration of idle time in a transaction holding a global lock (default is 10 minutes). A value of zero disables this timeout.

### bdr.lock_table_locking

Sets locking behavior for LOCK TABLE statement (default is on). When enabled, LOCK TABLE statement also takes a global DML lock on the cluster, blocking other locking statements.

| Value | Description |
|---|---|
| *on* | Use global locking for all table locks. (Default) |
| *off* | Don't use global locking for table locks. |

### bdr.predictive_checks

Sets the log level for predictive checks (currently used only by global locks). Can be `DEBUG` , `LOG` , `WARNING` (default), or `ERROR` . Predictive checks are early validations for expected cluster state when doing certain operations. You can use them for those operations for fail early rather than wait for timeouts. In global lock terms, PGD checks that there are enough nodes connected and withing reasonable lag limit for getting the quorum needed by the global lock.

## Node management

### bdr.replay_progress_frequency

Sets the interval for sending replication position info to the rest of the cluster (default is 1 minute).

## Generic replication

### bdr.writers_per_subscription

Sets the default number of writers per subscription. (In PGD, you can also change this with `bdr.alter_node_group_option` for a group.)

### bdr.max_writers_per_subscription

Maximum number of writers per subscription (sets upper limit for the `bdr.writers_per_subscription` setting).

### bdr.xact_replication

Replicates current transaction (default is `on` ).

Turning this off makes the whole transaction local only, which means the transaction isn't visible to logical decoding by PGD and all other downstream targets of logical decoding. Data isn't transferred to any other node, including logical standby nodes.

This parameter can be set only by the bdr_superuser or superuser roles.

This parameter can be set only inside the current transaction using the `SET LOCAL` command unless `bdr.permit_unsafe_commands = on` .

**Note**

Even with transaction replication disabled, WAL is generated, but those changes are filtered away on the origin.

**Warning**

Turning off *bdr.xact_replication* leads to data inconsistency between nodes. Use it only to recover from data divergence between nodes or in replication situations where changes on single nodes are required for replication to continue. Use at your own risk.

### bdr.permit_unsafe_commands

Overrides safety check on commands that are deemed unsafe for general use.

Requires bdr_superuser or PostgreSQL superuser.

**Warning**

The commands that are normally not considered safe can either produce inconsistent results or break replication altogether. Use at your own risk.

### bdr.batch_inserts

Number of consecutive inserts to one table in a single transaction that turns on batch processing of inserts for that table.

This setting allows replication of large data loads as COPY internally, rather than as a set of inserts. It's also how the initial data during node join is copied.

### bdr.maximum_clock_skew

Specifies the maximum difference between the incoming transaction commit timestamp and the current time on the subscriber before triggering `bdr.maximum_clock_skew_action` .

It checks if the timestamp of the currently replayed transaction is in the future compared to the current time on the subscriber. If it is, and the difference is larger than `bdr.maximum_clock_skew` , it performs the action specified by the `bdr.maximum_clock_skew_action` setting.

The default is `-1` , which means ignore clock skew (the check is turned off). It's valid to set 0 as when the clocks on all servers are synchronized. The fact that the transaction is being replayed means it was committed in the past.

### bdr.maximum_clock_skew_action

Specifies the action to take if a clock skew higher than `bdr.maximum_clock_skew` is detected.

There are two possible values for this setting:

| Value | Description |
|---|---|
| *WARN* | Log a warning about this fact. The warnings are logged once per minute at the maximum to prevent flooding the server log. |
| *WAIT* | Wait until the current local timestamp is no longer older than remote commit timestamp minus the *bdr.maximum_clock_skew*. |

### bdr.accept_connections

Enables or disables connections to PGD (default is `on` ).

Requires bdr_superuser or PostgreSQL superuser.

### bdr.writer_input_queue_size

Specifies the size of the shared memory queue used by the receiver to send data to the writer process. If the writer process is stalled or making slow progress, then the queue might get filled up, stalling the receiver process too. So it's important to provide enough shared memory for this queue. The default is 1 MB, and the maximum allowed size is 1 GB. While any storage size specifier can be used to set the GUC, the default is KB.

### bdr.writer_output_queue_size

Specifies the size of the shared memory queue used by the receiver to receive data from the writer process. Since the writer isn't expected to send a large amount of data, a relatively smaller sized queue is enough. The default is 32 KB, and the maximum allowed size is 1 MB. While any storage size specifier can be used to set the GUC, the default is KB.

### bdr.min_worker_backoff_delay

Allows for rate limiting of PGD background worker launches by preventing a given worker from being relaunched more often than every `bdr.min_worker_backoff_delay` milliseconds. On repeated errors, the backoff increases exponentially with added jitter up to a maximum of `bdr.max_worker_backoff_delay` .

Time-unit suffixes are supported.

> **Note**
>
> This setting currently affects only receiver worker, which means it primarily affects how fast a subscription tries to reconnect on error or connection failure.

The default for `bdr.min_worker_backoff_delay` is 1 second. For `bdr.max_worker_backoff_delay` , it's 1 minute.

If the backoff delay setting is changed and the PostgreSQL configuration is reloaded, then all current backoffs wait for reset. Additionally, the `bdr.worker_task_reset_backoff_all()` function is provided to allow the administrator to force all backoff intervals to immediately expire.

A tracking table in shared memory is maintained to remember the last launch time of each type of worker. This tracking table isn't persistent. It's cleared by PostgreSQL restarts, including soft restarts during crash recovery after an unclean backend exit.

You can use the view *bdr.worker_tasks* to inspect this state so the administrator can see any backoff rate limiting currently in effect.

For rate-limiting purposes, workers are classified by task. This key consists of the worker role, database OID, subscription ID, subscription writer ID, extension library name and function name, extension-supplied worker name, and the remote relation ID for sync writers. `NULL` is used where a given classifier doesn't apply, for example, when manager workers don't have a subscription ID and receivers don't have a writer ID.

### CRDTs

#### bdr.crdt_raw_value

Sets the output format of *Handling column-level conflicts using CRDT data types* .

The default output (when this setting is `off` ) is to return only the current value of the base CRDT type, for example, a bigint for `crdt_pncounter` . When set to `on` , the returned value represents the full representation of the CRDT value, which can, for example, include the state from multiple nodes.

### Large objects

#### bdr.override_large_objects

When set to `on` , transparently redirects PostgreSQL's built-in large object functions (`lo_create` , `lo_open` , `lo_read` , and so on) to their `bdr.lo_*` equivalents, so existing applications use replicated large objects without code changes.

Redirection applies to SQL-level calls. libpq-based clients and drivers also issue large object calls at the protocol level, bypassing SQL. To redirect those too, connect to the *Connection Manager Authentication* read-write port (default 6432) instead of connecting directly to Postgres.

The default is `off` . For more details, see *Replicated large objects* .

### Commit scope

#### bdr.commit_scope

Sets the current (or default) *Commit scopes* (default is an empty string).

### Commit At Most Once

#### bdr.camo_local_mode_delay

The commit delay that applies in CAMO's asynchronous mode to emulate the overhead that normally occurs with the CAMO partner having to confirm transactions (default is 5 ms). Set to `0` to disable this feature.

#### bdr.camo_enable_client_warnings

Emits warnings if an activity is carried out in the database for which CAMO properties can't be guaranteed (default is enabled). Well-informed users can choose to disable this setting to reduce the amount of warnings going into their logs.

### Transaction streaming

#### bdr.default_streaming_mode

Controls transaction streaming by the subscriber node. Possible values are: `off` , `writer` , `file` , and `auto` . Defaults to `auto` . If set to `off` , the subscriber doesn't request transaction streaming. If set to one of the other values, the subscriber requests transaction streaming and the publisher provides it if it supports them and if configured at group level. For more details, see *Using with transaction streaming* .

### Lag Control

#### bdr.lag_control_max_commit_delay

Maximum acceptable post-commit delay that can be tolerated, in fractional milliseconds.

#### bdr.lag_control_max_lag_size

Maximum acceptable lag size that can be tolerated, in kilobytes.

#### bdr.lag_control_max_lag_time

Maximum acceptable lag time that can be tolerated, in milliseconds.

#### bdr.lag_control_min_conforming_nodes

Minimum number of nodes required to stay below acceptable lag measures.

#### bdr.lag_control_commit_delay_adjust

Commit delay micro adjustment measured as a fraction of the maximum commit delay time. At a default value of 0.01%, it takes 100 net increments to reach the maximum commit delay.

#### bdr.lag_control_sample_interval

Minimum time between lag samples and commit delay micro adjustments, in milliseconds.

#### bdr.lag_control_commit_delay_start

The lag threshold at which commit delay increments start to be applied, expressed as a fraction of acceptable lag measures. At a default value of 1.0%, commit delay increments don't begin until acceptable lag measures are breached.

By setting a smaller fraction, it might be possible to prevent a breach by "bending the lag curve" earlier so that it's asymptotic with the acceptable lag measure.

### Monitoring and logging

#### bdr.debug_level

Defines the log level that PGD uses to write its debug messages. The default value is `debug2` . If you want to see detailed PGD debug output, set `bdr.debug_level = 'log'` .

### bdr.trace_level

Similar to `bdr.debug_level` , defines the log level to use for PGD trace messages. Enabling tracing on all nodes of an EDB Postgres Distributed cluster might help EDB Support to diagnose issues. You can set this parameter only at Postgres server start.

!!!Warning

Setting `bdr.debug_level` or `bdr.trace_level` to a value >= `log_min_messages` can produce a very large volume of log output. Don't enabled it long term in production unless plans are in place for log filtering, archival, and rotation to prevent disk space exhaustion.

Tracks apply statistics for each subscription with bdr.stat_subscription (default is `on` ).

### bdr.track_relation_apply

Tracks apply statistics for each relation with bdr.stat_relation (default is `off` ).

### bdr.track_apply_lock_timing

Tracks lock timing when tracking statistics for relations with bdr.stat_relation (default is `off` ).

### Decoding worker

### bdr.enable_wal_decoder

Enables logical change record (LCR) sending on a single node with a *Decoding worker* (default is false). When set to true, a decoding worker process starts, and WAL senders send the LCRs it produces. If set back to false, any WAL senders using LCR are restarted and use the WAL directly.

!!!Note

You also need to enable this setting on all nodes in the PGD group and set the `enable_wal_decoder` option to true on the group.

When subscribing to another node, this setting enables the node to request the use of logical change records (LCRs) for the subscription (default is false). When this setting is true on a downstream node, the node requests that upstream nodes use LCRs when sending to it. If you set `bdr.enable_wal_decoder` to true on a node, also set this setting to true.

!!!Note

You also need to enable this setting on all nodes in the PGD group and set the `enable_wal_decoder` option to true on the group.

Logical change record (LCR) file cleanup interval (default is 3 minutes). When the *Decoding worker* is enabled, the decoding worker stores LCR files as a buffer. These files are periodically cleaned, and this setting controls the interval between any two consecutive cleanups. Setting it to zero disables cleanup.

### Connectivity settings

The following are a set of connectivity settings affecting all cross-node `libpq` connections. The defaults are set to fairly conservative values and cover most production needs. All variables have `SIGHUP` context, meaning changes are applied upon reload.

#### bdr.global_connection_timeout

Maximum time to wait while connecting, in seconds (default is 15 seconds). Write as a decimal integer, for example, 10. Zero, negative, or not specified means wait indefinitely. The minimum allowed timeout is 2 seconds, therefore a value of 1 is interpreted as 2.

#### bdr.global_keepalives

Controls whether TCP keepalives are used (default is 1, meaning on). If you don't want keepalives, you can change this to 0, meaning off. This parameter is ignored for connections made by a Unix-domain socket.

#### bdr.global_keepalives_idle

Controls the number of seconds of inactivity after which TCP sends a keepalive message to the server (default is 1 second). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket or if keepalives are disabled. It's supported only on systems where `TCP_KEEPIDLE` or an equivalent socket option is available. On other systems, it has no effect.

#### bdr.global_keepalives_interval

Controls the number of seconds after which to retransmit a TCP keepalive message that isn't acknowledged by the server (default is 2 seconds). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket or if keepalives are disabled. It's supported only on systems where `TCP_KEEPINTVL` or an equivalent socket option is available. On other systems, it has no effect.

#### bdr.global_keepalives_count

Controls the number of TCP keepalives that can be lost before the client's connection to the server is considered dead (default is 3). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket or if keepalives are disabled. It's supported only on systems where `TCP_KEEPCNT` or an equivalent socket option is available. On other systems, it has no effect.

#### bdr.global_tcp_user_timeout

Controls the number of milliseconds that transmitted data can remain unacknowledged before a connection is forcibly closed (default is 5000, that is, 5 seconds). A value of zero uses the system default. This parameter is ignored for connections made by a Unix-domain socket. It's supported only on systems where `TCP_USER_TIMEOUT` is available. On other systems, it has no effect.

### Topology settings

#### bdr.force_full_mesh

Forces the full mesh topology (default is `on` ). When set to `off` , PGD will attempt to use the optimized topology for subscriber-only groups. This setting is only effective when the requirements for the optimized topology are met. See *Optimizing subscriber-only groups* for more information.

### Analytics settings

#### bdr.prefer_analytics_engine

Determines whether the system should route queries to the Postgres Analytics Accelerator (PGAA) engine (Seafowl or Spark), even when a local copy of the data exists in the Postgres heap. The default is `false` .

### Internal settings - Raft timeouts

#### bdr.raft_global_election_timeout

To account for network failures, the Raft consensus protocol implements timeouts for elections and requests. This value is used when a request is being sent to the global (top-level) group. The default is 6 seconds (6s).

#### bdr.raft_group_election_timeout

To account for network failures, the Raft consensus protocol implements timeouts for elections and requests. This value is used when a request is being sent to the sub-group. The default is 3 seconds (3s).

#### bdr.raft_response_timeout

For responses, the settings of *bdr.raft_global_election_timeout* and *bdr.raft_group_election_timeout* are used as appropriate. You can override this behavior by setting this variable. The setting of `bdr.raft_response_timeout` must be less than either of the election timeout values. Set this variable to -1 to disable the override. The default is -1.

### Internal settings - Other Raft values

#### bdr.raft_keep_min_entries

The minimum number of entries to keep in the Raft log when doing log compaction (default is `1000` ; PGD 5.3 and earlier: `100` ). The value of `0` disables log compaction. You can set this parameter only at Postgres server start.

!!!Warning

If log compaction is disabled, the log grows in size forever.

To move the state machine forward, Raft appends entries to its internal log. During normal operation, appending takes only a few milliseconds. This poses an upper threshold on the duration of that append action, above which an `INFO` message is logged. This can indicate a problem. Default is 3000 ms.

#### bdr.raft_log_min_message_duration

When to log a consensus request. Measures roundtrip time of a PGD consensus request and logs an `INFO` message if the time exceeds this parameter (default is 5000 ms).

#### bdr.raft_group_max_connections

The maximum number of connections across all PGD groups for a Postgres server (default is 100 connections). These connections carry PGD consensus requests between the groups' nodes. You can set this parameter only at Postgres server start.

### Internal settings - Other values

#### bdr.backwards_compatibility

Specifies the version to be backward compatible to, in the same numerical format as used by `bdr.bdr_version_num` , for example, `30618` . (Default is the current PGD version.) Enables exact behavior of a former PGD version, even if this has generally unwanted effects. Since this changes from release to release, we advise against explicit use in the configuration file unless the value is different from the current version.

### bdr.track_replication_estimates

Tracks replication estimates in terms of apply rates and catchup intervals for peer nodes. Protocols like CAMO can use this information to estimate the readiness of a peer node. This parameter is enabled by default.

### bdr.lag_tracker_apply_rate_weight

PGD monitors how far behind peer nodes are in terms of applying WAL from the local node and calculate a moving average of the apply rates for the lag tracking. This parameter specifies how much contribution newer calculated values have in this moving average calculation. Default is 0.1.

### bdr.enable_auto_sync_reconcile

When enabled, PGD automatically synchronizes data from the node furthest ahead when a node goes down, and automatically reconciles unresolved prepared transactions left by a failed origin node. Default (from 6.0.1) is on.

### bdr.allow_bdr_schema_changes

Permits manual changes to database objects in the `bdr` schema. Defaults to off. Change this parameter only when explicitly instructed to do so.

## Node management

### List of node states

| State | Description |
|---|---|
| *NONE* | Node state is unset when the worker starts, expected to be set quickly to the current known state. |
| *CREATED* | *bdr.create_node()* was executed, but the node isn't a member of any EDB Postgres Distributed cluster yet. |
| *JOIN_START* | *bdr.join_node_group()* begins to join the local node to an existing EDB Postgres Distributed cluster. |
| *JOINING* | The node join has started and is currently at the initial sync phase, creating the schema and data on the node. |
| *CATCHUP* | Initial sync phase is completed. Now the join is at the last step of retrieving and applying transactions that were performed on the upstream peer node since the join started. |
| *STANDBY* | Node join finished but hasn't yet started to broadcast changes. All joins spend some time in this state, but if defined as a logical standby, the node continues in this state. |
| *PROMOTE* | Node was a logical standby and *bdr.promote_node* was just called to move the node state to *ACTIVE*. These two *PROMOTE* states have to be coherent to the fact that only one node can be with a state higher than *STANDBY* but lower than *ACTIVE*. |
| *PROMOTING* | Promotion from logical standby to full PGD node is in progress. |
| *ACTIVE* | The node is a full PGD node and is currently *ACTIVE*. This is the most common node status. |
| *PART_START* | Node was *ACTIVE* or *STANDBY* and *bdr.part_node* was just called to remove the node from the EDB Postgres Distributed cluster. |
| *PARTING* | Node disconnects from other nodes and plays no further part in consensus or replication. |
| *PART_CATCHUP* | Nonparting nodes synchronize any missing data from the recently parted node. |
| *PART_CLEANUP* | Non-parting nodes wait until the group slots of all nodes are caught up with all the transactions that originated from the PARTED node. |

continues on next page

Table 131 – continued from previous page

| State | Description |
|---|---|
| *PARTED* | Node parting operation is now complete on all nodes. |
| *PARTED_CONCURRENT* | Node parting operation was started with concurrent option. |

Only one node at a time can be in either of the states `PROMOTE` or `PROMOTING` .

The `PARTED_CONCURRENT` state is used as target state to indicate that parting was done with a `concurrent = true` option.

### Node-management commands

PGD also provides a command-line utility for adding nodes to the PGD group using a physical copy (`pg_basebackup` ) of an existing node.

### bdr_init_physical

!!!warning Deprecated

This command is deprecated in favor of the using the pgd CLI command pgd node setup which offers a more flexible and powerful ways to create and manage nodes in a PGD group. `bdr_init_physical` will receive only bug fixes in the future and is not recommended for new installations.

`bdr_init_physical` requires that both the source node and the joining node have exactly the same PGD version. You can't use this command to join a node with a different PGD version to an existing cluster. This means that you can't use `bdr_init_physical` for rolling upgrades using the node join method. For rolling upgrades, use logical join instead. See *Rolling upgrade using node join* for more details.

You must specify a data directory. If this data directory is empty, use `pg_basebackup -X stream` to fill the directory using a fast block-level copy operation.

If the specified data directory isn't empty, it's used as the base for the new node. Initially, it waits for catchup and then promotes to a master node before joining the PGD group. The `--standby` option, if used, turns it into a logical standby node.

This command drops all PostgreSQL-native logical replication subscriptions from the database (or disables them when the `-S` option is used) as well as any replication origins and slots.

### Synopsis

```
bdr_init_physical [OPTION] ...
```

### Options

### General options

- `-D, --pgdata=DIRECTORY` — The data directory to use for the new node. It can be either an empty or nonexistent directory or a directory populated using the `pg_basebackup -X stream` command (required).
- `-l, --log-file=FILE` — Use FILE for logging. The default is `bdr_init_physical_postgres.log` .
- `-n, --node-name=NAME` — The name of the newly created node (required).
- `--replication-sets=SETS` — The name of a comma-separated list of replication set names to use. All replication sets are used if not specified.

- `--standby` — Create a logical standby (receive-only node) rather than full send/receive node.
- `--node-group-name` — Group to join. Defaults to the same group as source node.
- `-s, --stop` — Stop the server once the initialization is done.
- `-v` — Increase logging verbosity.
- `-L` — Perform selective pg_basebackup when used with an empty/nonexistent data directory (-D option). This is a feature of EDB Postgres Extended Server only.
- `-S` — Instead of dropping logical replication subscriptions, disable them.

### Connection options

- `-d, --remote-dsn=CONNSTR` — Connection string for remote node (required).
- `--local-dsn=CONNSTR` — Connection string for local node (required).

### Configuration files override

- `--hba-conf` — Path to the new `pg_hba.conf` .
- `--postgresql-conf` — Path to the new `postgresql.conf` .
- `--postgresql-auto-conf` — Path to the new `postgresql.auto.conf` .

### Notes

The replication set names specified in the command don't affect the data that exists in the data directory before the node joins the PGD group. This is true whether `bdr_init_physical` makes its own base backup or an existing base backup is being promoted to a new PGD node. Thus the `--replication-sets` option affects only the data published and subscribed to after the node joins the PGD node group. This behavior is different from the way replication sets are used in a logical join, as when using `bdr.join_node_group()` .

The operator can truncate unwanted tables after the join completes. Refer to the `bdr.tables` catalog to determine replication set membership and identify tables that aren't members of any subscribed-to replication set. We strongly recommend that you truncate the tables rather than drop them, because:

- DDL replication sets aren't necessarily the same as row (DML) replication sets, so you might inadvertently drop the table on other nodes.
- If you later want to add the table to a replication set and you dropped it on some subset of nodes, you need to re-create it only on those nodes without creating DDL conflicts before you can add it to any replication sets.

It's simpler and safer to truncate your nonreplicated tables, leaving them present but empty.

### bdr_config

This command-line utility allows you to examine the configuration of a PGD installation. It is analogous to the `pg_config` utility that comes with PostgreSQL. You can use it to assist in troubleshooting and support.

### Synopsis

```
bdr_config [OPTION] ...
```

### Options

| Option | Description |
|---|---|
| *--all* | Show all the keys and values in the configuration. |
| *--version* | Show only the BDR version related keys and values. This includes the full version of the BDR extension, the Postgres version and flavor it is running against, and the BDRPG and BDR plugin API versions. |
| *--debug* | Show only the BDR debug keys and values, including build information and feature enablement. |

### Example

```
$ /usr/lib/edb-as/16/bin/bdr_config --all
__OUTPUT__
BDR_VERSION_COMPLETE=5.6.0
BDR_VERSION_NUM=50600
PG_VERSION=16.4.1 (Debian 16.4.1~~snapshot11329862135.2980.1.88fbec6-1.bookworm)
PG_VERSION_NUM=160004
PG_FLAVOR=EPAS
BDRPG_API_VERSION_NUM=202309131
BDR_PLUGIN_API_VERSION=7011
USE_ASSERT_CHECKING=false
USE_VALGRIND=false
EXT_ENABLE_DTRACE=false
HAVE_LAG_CONTROL=true
HAVE_ASSESS_UPDATE_RI_HOOK=false
HAVE_BDRPG_PROBES=false
HAVE_CAMO=true
HAVE_DEADLOCK_DETECTOR_HOOK=true
HAVE_HEAP_UPDATE_HOOK=true
HAVE_LAG_TRACKER=true
HAVE_LCR=true
HAVE_LOG_TOAST_COLUMNS=false
HAVE_MISC_HOOKS=true
HAVE_MISSING_PARTITION_CONFLICT=true
HAVE_MULTI_PITR=false
HAVE_SELECTIVE_BASEBACKUP=false
HAVE_STREAMING_XACTS=true
HAVE_SYNC_COMMIT_HOOK=true
HAVE_TWOPHASE_DATA_HOOKS=true
HAVE_XLOG_FIND_NEXT_RECORD=true
HAVE_DETACH_CONCURRENTLY=true
HAVE_ANALYTICS=true
```

### Node management interfaces

You can add and remove nodes dynamically using the SQL interfaces.

### bdr.alter_node_group_option

Modifies a PGD node group configuration.

### Synopsis

```
bdr.alter_node_group_option(node_group_name text,
    config_key text,
    config_value text)
```

### Parameters

| Name | Description |
|---|---|
| *node_group_name* | Name of the group to change. |
| *config_key* | Key of the option in the node group to change. |
| *config_value* | New value to set for the given key. |

`config_value` is parsed into the data type appropriate for the option.

The table shows the group options that can be changed using this function.

| Name | Type | Description |
|---|---|---|
| *analytics_storage_format* | *text* | The columnar table format used by PGAA for the analytical data. Supported values are *iceberg* (default) or *delta*. |
| *analytics_storage_location* | *text* | The PGFS path where the analytical data files are stored. Used primarily when writing directly to object storage via the PGAA extension. |
| *analytics_write_catalog* | *text* | The name of the catalog service used by the PGAA extension to manage table metadata. |
| *apply_delay* | *interval* | How long nodes wait to apply incoming changes. This option is useful mainly to set up a special subgroup with delayed subscriber-only nodes. Don't set this on groups that contain data nodes or on the top-level group. Default is *0s*. |
| *apply_error_max_record_size* | *bigint* | Maximum number of bytes of change data to record per failed transaction when using a *record_all_and_disable* or *record_all_and_skip* policy. If a transaction exceeds this limit, only the changes captured up to the limit are stored and a warning is logged. Default is *10485760* (10 MB). |
| *apply_error_max_retries* | *integer* | Number of retry attempts before the error policy takes action. Applies to the *disable_after_retries*, *record_all_and_disable*, and *record_all_and_skip* policies. Default is *3*. |
| *apply_error_max_skips* | *integer* | Maximum number of transactions the *record_all_and_skip* policy can skip before disabling the subscription. Set to *-1* or *NULL* for unlimited skips. Default is *NULL*. |

continues on next page

Table 134 – continued from previous page

| Name | Type | Description |
|---|---|---|
| *apply_error_policy* | *text* | How replication apply errors are handled for subscriptions in this group. Valid values are *keep_retrying* (retry indefinitely), *disable_after_retries* (retry up to *apply_error_max_retries* times, then disable the subscription), *record_all_and_disable* (record the entire transaction by replaying it in capture-only mode, then disable the subscription), and *record_all_and_skip* (record the entire transaction, skip it, and continue replicating). Default is *keep_retrying*. |
| *check_constraints* | *boolean* | Whether the apply process checks the constraints when writing replicated data. We recommend keeping the default value or you risk data loss. Valid values are *on* or *off*. Default is *on*. |
| *default_commit_scope* | *text* | The commit scope to use by default, initially the *local* commit scope. This option applies only to the top-level node group. You can use individual rules for different origin groups of the same commit scope. See *Commit scope groups* for more details. |
| *enable_fast_track_writer* | *boolean* | Enables an additional writer for all subscriptions to process prepared transactions generated by Quorum Commit. Required for correct functioning with Quorum Commit commit scopes. Automatically enabled when a Quorum Commit scope is set as *default_commit_scope*. Default is *off*. See *Replicated large objects* . |
| *enable_routing* | *boolean* | Where :ref:``Connection Manager` </pgd/current/connection-manager/>` through the group write leader is enabled for a given group. Valid values are *on* or *off*. Default is *on* for subgroups and *off* for the cluster group. |
| *enable_raft* | *boolean* | Whether group has its own Raft consensus. This option is necessary for setting *enable_routing* to *on*. This option is always *on* for the top-level group. Valid values are *on* or *off*. Default is *on* for subgroups. |
| *enable_wal_decoder* | *boolean* | Enables/disables the decoding worker process. You can't enable the decoding worker process if *streaming_mode* is already enabled. Valid values are *on* or *off*. Default is *off*. |
| *location* | *text* | Information about group location. This option is purely metadata for monitoring. Default is '' (empty string). |
| *num_writers* | *integer* | Number of parallel writers for the subscription backing this node group. Valid values are *-1* or a positive integer. *-1* means the value specified by the GUC *Replicated large objects* is used. *-1* is the default. |
| *route_reader_max_lag* | *integer* | Maximum lag in bytes for a node to be considered a viable read-only node. Currently reserved for future use. |
| *route_writer_max_lag* | *integer* | Maximum lag in bytes of the new write candidate to be selected as write leader. If no candidate passes this, no writer is selected. Default is *-1*. |
| *route_writer_wait_flush* | *boolean* | Whether to switch if PGD needs to wait for the flush. Currently reserved for future use. |

continues on next page

Table 134 – continued from previous page

| Name | Type | Description |
|---|---|---|
| *server_pool_mode* | *text* | *Connection pooling* mode for Connection Manager. Valid values are *none* (no pooling, default), *session*, and *transaction* |
| *streaming_mode* | *text* | Enables/disables streaming of large transactions. When set to *off*, streaming is disabled. When set to any other value, large transactions are decoded while they're still in progress, and the changes are sent to the downstream. If the value is set to *file*, then the incoming changes of streaming transactions are stored in a file and applied only after the transaction is committed on upstream. If the value is set to *writer*, then the incoming changes are directly sent to one of the writers, if available. If *Parallel Apply* is disabled or no writer is free to handle streaming transactions, then the changes are written to a file and applied after the transaction is committed. If the value is set to *auto*, PGD tries to intelligently pick between *file* and *writer*, depending on the transaction property and available resources. You can't enable *streaming_mode* if the WAL decoder is already enabled. Default is *auto*. For more details, see *Using with transaction streaming* . |
| *use_https* | *boolean* | Whether the HTTP listener uses HTTPS. When enabled, the server certificate is used for TLS. Requires *ssl_cert_file* and *ssl_key_file* configured in *postgresql.conf* or via the command line. Changes take effect only after restarting the Connection Manager process. Default is *false*. See *Configuring Connection Manager* for more details. |
| *failover_slot_scope* | *text* | PGD 5.7 and later only. Sets the scope for Logical Slot Failover support. Valid values are *global* or *local*. Default is *local*. For more information, see *CDC Failover support* . |

For configuration examples and a resolution workflow for the `apply_error_*` options, see *Handling replication apply errors* .

#### `enable_fast_track_writer` notes

When `enable_fast_track_writer` is off and the application uses Quorum Commit via the `bdr.commit_scope` configuration parameter, enable this option explicitly on the top-level group so the entire cluster has it enabled:

```
SELECT bdr.alter_node_group_option(topgroup, enable_fast_track_writer, true);
```

Adding the fast track writer increases the number of replication slots required per node. Review the `max_replication_slots` calculation in *Postgres configuration parameters* and, for PostgreSQL 18 and later, `max_active_replication_origins` . Use *bdr:get_min_required_replication_slots()* to verify the values are high enough after enabling this option.

### Return value

`bdr.alter_node_group_option()` returns `VOID` on success.

An `ERROR` is raised if any of the provided parameters is invalid.

### Notes

You can examine the current state of node group options by way of the view *bdr.node_group_summary* .

This function passes a request to the group consensus mechanism to change the defaults. The changes made are replicated globally using the consensus mechanism.

The function isn't transactional. The request is processed in the background, so you can't roll back the function call. Also, the changes might not be immediately visible to the current transaction.

This function doesn't hold any locks.

## bdr.alter_node_interface

Changes the connection string (`DSN` ) of a specified node.

### Synopsis

```
bdr.alter_node_interface(node_name text, interface_dsn text)
```

### Parameters

| Name | Description |
|---|---|
| *node_name* | Name of an existing node to alter. |
| *interface_dsn* | New connection string for a node. |

### Notes

Run this function and make the changes only on the local node. This means that you normally execute it on every node in the PGD group, including the node that's being changed.

This function is transactional. You can roll it back, and the changes are visible to the current transaction.

The function holds lock on the local node.

## bdr.alter_node_option

Modifies per-node configuration options.

### Synopsis

```
bdr.alter_node_option(node_name text,
    config_key text,
    config_value text);
```

### Parameters

| Name | Description |
|---|---|
| *node_name* | Name of the node to change. |
| *config_key* | Key of the option in the node to change. |

continues on next page

Table 136 – continued from previous page

| Name | Description |
|---|---|
| *config_value* | New value to set for the given key. |

The node options you can change using this function are:

| Config Key | Description |
|---|---|
| *route_priority* | Relative routing priority of the node against other nodes in the same node group. Default is *'-1'*. |
| *route_fence* | Whether the node is fenced from routing. When true, the node can't receive connections from the Connection Manager. Replication is not impacted. Default is *'f'* (false). |
| *route_writes* | Whether writes can be routed to this node, that is, whether the node can become write leader. Default is *'t'* (true) for data nodes and *'f'* (false) for other node types. |
| *route_reads* | Whether read-only connections can be routed to this node. Currently reserved for future use. Default is *'t'* (true) for data and subscriber-only nodes, *'f'* (false) for witness and standby nodes. |
| *route_dsn* | The dsn for the proxy to use to connect to this node. This option is optional. If not set, it defaults to the node's *node_dsn* value. |
| *group_dsn* | Preferred connection string for nodes in the same subgroup. When set, nodes within the same subgroup connect using this DSN instead of the node's *node_dsn*. Set to NULL to reset to the default. Stored in :ref:``bdr.node_config` columns<*bdr.node_config* columns>` as *node_group_dsn*. |

### bdr.alter_subscription_enable

Enables either the specified subscription or all the subscriptions of the local PGD node. This is also known as resume subscription. No error is thrown if the subscription is already enabled. Returns the number of subscriptions affected by this operation.

### Synopsis

```
bdr.alter_subscription_enable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false
)
```

### Parameters

| Name | Description |
|---|---|
| *subscription_name* | Name of the subscription to enable. If NULL (the default), all subscriptions on the local node are enabled. |
| *immediate* | Used to force the action immediately, starting all the workers associated with the enabled subscription. When this option is *true*, you can't run this function inside of the transaction block. |

### Notes

This function isn't replicated and affects only local node subscriptions (either a specific node or all nodes).

This function is transactional. You can roll it back, and the current transaction can see any catalog changes. The subscription workers are started by a background process after the transaction has committed.

### bdr.alter_subscription_disable

Disables either the specified subscription or all the subscriptions of the local PGD node. Optionally, it can also immediately stop all the workers associated with the disabled subscriptions. This is also known as pause subscription. No error is thrown if the subscription is already disabled. Returns the number of subscriptions affected by this operation.

### Synopsis

```
bdr.alter_subscription_disable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false,
    fast boolean DEFAULT true
)
```

### Parameters

| Name | Description |
|---|---|
| *subscription_name* | Name of the subscription to disable. If NULL (the default), all subscriptions on the local node are disabled. |
| *immediate* | Used to force the action immediately, stopping all the workers associated with the disabled subscription. When this option is *true*, you can't run this function inside of the transaction block. |
| *fast* | This argument influences the behavior of *immediate*. If set to *true* (the default), it stops all the workers associated with the disabled subscription without waiting for them to finish current work. |

### Notes

This function isn't replicated and affects only local node subscriptions (either a specific subscription or all subscriptions).

This function is transactional. You can roll it back, and the current transaction can see any catalog changes. However, the timing of the subscription worker stopping depends on the value of `immediate` . If set to `true` , the workers receive the stop without waiting for the `COMMIT` . If the `fast` argument is set to `true` , the interruption of the workers doesn't wait for current work to finish.

### bdr.create_node

Creates a node.

### Synopsis

```
bdr.create_node(node_name text,
                local_dsn text,
                node_kind DEFAULT NULL)
```

### Parameters

| Name | Description |
|---|---|
| *node_name* | Name of the new node. Only one node is allowed per database. Valid node names consist of lowercase letters, numbers, hyphens, and underscores. |
| *local_dsn* | Connection string to the node. |
| *node_kind* | One of *data* (the default), *standby*, *subscriber-only*, or *witness*. If you don't set this parameter, or if you provide *NULL*, the default *data* node kind is used. |

### Notes

This function creates a record for the local node with the associated public connection string. There can be only one local record, so once it's created, the function reports an error if run again.

This function is a transactional function. You can roll it back and the changes made by it are visible to the current transaction.

The function holds lock on the newly created node until the end of the transaction.

## bdr.create_node_group

Creates a PGD node group. By default, the local node joins the group as the only member. You can add more nodes to the group with bdr.join_node_group() .

### Synopsis

```
bdr.create_node_group(node_group_name text,
                      parent_group_name text DEFAULT NULL,
                      join_node_group boolean DEFAULT true,
                      node_group_type text DEFAULT NULL)
```

### Parameters

| Name | Description |
|---|---|
| *node_group_name* | Name of the new PGD group. As with the node name, valid group names consist of only lowercase letters, numbers, and underscores. |
| *parent_group_name* | If a node subgroup is being created, this must be the name of the parent group. Provide *NULL* (the default) when creating the main node group for the cluster. |
| *join_node_group* | Determines whether the node joins the group being created. The default value is *true*. Providing *false* when creating a subgroup means the local node won't join the new group, for example, when creating an independent remote group. In this case, you must specify *parent_group_name*. |

continues on next page

Table 141 – continued from previous page

| Name | Description |
|---|---|
| *node_group_type* | The valid values are *NULL* or *subscriber-only*. *NULL* (the default) is for creating a normal, general-purpose node group. *subscriber-only* is for creating *Creating Subscriber-only groups and nodes* whose members receive changes only from the fully joined nodes in the cluster but that never send changes to other nodes. |

### Notes

This function passes a request to the local consensus worker that's running for the local node.

The function isn't transactional. The creation of the group is a background process, so once the function finishes, you can't roll back the changes. Also, the changes might not be immediately visible to the current transaction. You can call `bdr.wait_for_join_completion` to wait until they are.

The group creation doesn't hold any locks.

## bdr.drop_node_group

Drops an empty PGD node group. If there are any joined nodes in the group, the function will fail.

### Synopsis

```
bdr.drop_node_group(node_group_name text)
```

### Parameters

| Name | Description |
|---|---|
| *node_group_name* | Name of the PGD group to drop. |

### Notes

This function passes a request to the group consensus mechanism to drop the group. The function isn't transactional. The dropping process happens in the background, and you can't roll it back.

## bdr.join_node_group

Joins the local node to an already existing PGD group.

### Synopsis

```
bdr.join_node_group (
    join_target_dsn text,
    node_group_name text DEFAULT NULL,
    wait_for_completion boolean DEFAULT true,
    synchronize_structure text DEFAULT all
)
```

### Parameters

| Name | Description |
|---|---|
| *join_target_dsn* | Specifies the connection string to an existing (source) node in the PGD group you want to add the local node to. |
| *node_group_name* | Optional name of the PGD group. Defaults to NULL, which tries to detect the group name from information present on the source node. |
| *wait_for_completion* | Wait for the join process to complete before returning. Defaults to *true*. |
| *synchronize_structure* | Specifies whether to perform database structure (schema) synchronization during the join. *all*, the default setting, synchronizes the complete database structure. *none* does not synchronize any structure. However, data will still be synchronized, meaning the database structure must already be present on the joining node. Note that by design, neither schema nor data will ever be synchronized to witness nodes. |

If `wait_for_completion` is specified as `false` , the function call returns as soon as the joining procedure starts. You can see the progress of the join in the log files and the bdr.event_summary

information view. You can call the function bdr.wait_for_join_completion()

after `bdr.join_node_group()` to wait for the join operation to complete. It can emit progress information if bdr.wait_for_join_completion() is called with `verbose_progress` set to `true` .

### Notes

This function passes a request to the group consensus mechanism by way of the node that the `join_target_dsn` connection string points to. The changes made are replicated globally by the consensus mechanism.

The function isn't transactional and will emit an error if executed in a transaction. The joining process happens in the background and you can't roll it back. The changes are visible only to the local session if `wait_for_completion` is set to `true` or by calling `bdr.wait_for_join_completion` later.

A node can be part of only a single group, so you can call this function only once on each node.

Node join doesn't hold any locks in the PGD group.

To reuse a node name while a parting process is still in progress, set `concurrent = true` when using `bdr.part_node` (requires Raft protocol version 6003 or above). This parameter allows the original name to be reused once the parting node reaches the `PARTING` state, at which point the node is automatically renamed to its UUID.

It isn't possible to join an additional data node to a node group where the existing nodes are configured as CAMO partners. To add the node, drop the CAMO commit scope first.

## bdr.part_node

Removes (parts) the node from the PGD group and eventually removes the parted node's metadata from all nodes in the cluster.

- For the local node, it removes all the node metadata, including information about remote nodes.
- For remote nodes, it removes only the metadata for that specific node.

This operation doesn't remove data from the node.

You can call the function from any active node in the PGD group, including the node that you're removing.

It isn't possible to part a data node if the group's `default_commit_scope` or the local node's `bdr.commit_scope` configuration parameter is set to a CAMO commit scope. To remove the node, drop the CAMO commit scope first.

Executing parting from the node being removed runs the risk of incorrectly reporting, or never reporting, the status of the removal. This is because in the process of being removed, communications are cut off from the rest of the cluster. While the removal may succeed, there's no way to inform the node that issued the command that it failed or succeeded on the other nodes. The function can't be set to wait for completion either, for the same reason.

Once a node has parted itself, it can't part other nodes in the cluster as it's no longer part of the cluster.

We recommend avoiding using nodes to part themselves from the cluster. Instead, perform node parting operations from a node that can wait for completion and check the cluster status after the operation is complete.

**Note**

If you're parting the local node, you must set *wait_for_completion* to *false* . Otherwise, it reports an error.

**Warning**

This action is permanent. If you want to temporarily halt replication to a node, use *bdr.alter_subscription_disable()* .

### Synopsis

```
bdr.part_node (
    node_name text,
    wait_for_completion boolean DEFAULT true,
    force boolean DEFAULT false,
    concurrent boolean DEFAULT false
)
```

### Parameters

| Name | Description |
|---|---|
| *node_name* | Name of an existing node to part. |
| *wait_for_completion* | If *true*, the function doesn't return until the node is fully parted from the cluster. Otherwise, the function starts the parting procedure and returns immediately without waiting. Always set to *false* when executing on the local node or when using *force*. |
| *force* | Forces removal of the node on the local node. This sets the node state locally if consensus can't be reached or if the node-parting process is stuck. |
| *concurrent* | If *true*, allows reuse of parting node name as soon as node is in *PARTING* state or above, without waiting for the node to reach the *PARTED* state and eventually being dropped. Default is *false*. This parameter is available when the cluster is at Raft protocol version 6003 or above. |

**Warning**

Using *force* = *true* can leave the PGD group in an inconsistent state. Use it only to recover from failures in which you can't remove the node any other way.

### Notes

This function passes a request to the group consensus mechanism to part the given node. The changes made are replicated globally by the consensus mechanism. The parting process happens in the background, and you can't roll it back. The changes made by the parting process are visible only to the local transaction if `wait_for_completion` was set to `true` .

With `force` set to `true` , on consensus failure, this function sets the state of the given node only on the local node. In such a case, the function is transactional (because the function changes the node state) and you can roll it back. If the function is called on a node that's already in process of parting with `force` set to `true` , it also marks the given node as parted locally and exits. This is useful only when the consensus can't be reached on the cluster (that is, the majority of the nodes are down) or if the parting process is stuck.

But it's important to take into account that when the parting node that was receiving writes, the parting process can take a long time without actually being stuck. The other nodes need to resynchronize any missing data from the given node. The other nodes need to wait till group slots of all nodes are caught up to all the transactions originating from the PARTED node.

A forced parting completely skips this resynchronization and can leave the other nodes in an inconsistent state.

The parting process doesn't hold any locks.

When `concurrent = true` , the target state for a parting node is set to `PARTED_CONCURRENT` . Upon reaching the `PARTING` state, the node is renamed to its UUID. The process continues until the node reaches the `PARTED` state and is dropped. Consequently, all tracking for the node must use its UUID once it moves past the `PART_START` state.

We recommend setting `wait_for_completion = false` when using `concurrent = true` . This configuration allows the function to return with a `Notice` as soon as the original node reaches the `PARTING` state. At this precise moment, the node is renamed to its UUID, freeing the original name for immediate reassignment or reuse in the cluster.

### bdr.promote_node

Promotes a local logical standby node to a full member of the PGD group.

### Synopsis

```
bdr.promote_node(wait_for_completion boolean DEFAULT true)
```

### Notes

This function passes a request to the group consensus mechanism to change the defaults. The changes made are replicated globally by the consensus mechanism.

The function isn't transactional. The promotion process happens in the background, and you can't roll it back. The changes are visible only to the local transaction if `wait_for_completion` was set to `true` or by calling `bdr.wait_for_join_completion` later.

The promotion process holds lock against other promotions. This lock doesn't block other `bdr.promote_node` calls but prevents the background process of promotion from moving forward on more than one node at a time.

### bdr.switch_node_group

Switches the local node from its current subgroup to another subgroup in the same existing PGD node group.

### Synopsis

```
bdr.switch_node_group (
    node_group_name text,
    wait_for_completion boolean DEFAULT true
)
```

### Parameters

| Name | Description |
|---|---|
| *node_group_name* | Name of the PGD group or subgroup. |
| *wait_for_completion* | Wait for the switch process to complete before returning. Defaults to *true*. |

If `wait_for_completion` is set to `false` , this is an asynchronous call that returns as soon as the switching procedure starts. You can see progress of the switch in logs and the `bdr.event_summary` information view or by calling the `bdr.wait_for_join_completion()` function after `bdr.switch_node_group()` returns.

### Notes

This function passes a request to the group consensus mechanism. The changes made are replicated globally by the consensus mechanism.

The function isn't transactional. The switching process happens in the background and you can't roll it back. The changes are visible only to the local transaction if `wait_for_completion` was set to `true` or by calling `bdr.wait_for_join_completion` later.

The local node changes membership from its current subgroup to another subgroup in the same PGD node group without needing to part the cluster. The node's kind must match that of existing nodes in the target subgroup.

Node switching doesn't hold any locks in the PGD group.

Restrictions: currently, the function allows switching only between a subgroup and its PGD node group. To effect a move between subgroups you need to make two separate calls: 1) switch from subgroup to node group and, 2) switch from node group to other subgroup.

## bdr.sync_node_cancel

This function cancels a sync request for the specified origin and source nodes.

### Synopsis

```
bdr.sync_node_cancel(origin text, source text)
```

### Parameters

| Name | Description |
|---|---|
| *origin* | Name of the origin node. |
| *source* | Name of the source node. |

### Notes

This function cancels all sync node requests for all targets that have the given origin and source. You can invoke it only from a write lead.

### bdr.wait_for_join_completion

This function waits for the join procedure of a local node to finish.

### Synopsis

```
bdr.wait_for_join_completion(verbose_progress boolean DEFAULT false)
```

### Parameters

| Name | Description |
|---|---|
| *verbose_progress* | Optionally prints information about individual steps taken during the join procedure. |

### Notes

This function waits until the checks state of the local node reaches the target state, which was set by `bdr.create_node_group` , `bdr.join_node_group` , or `bdr.promote_node` .

## Routing functions

### Commit scopes

Commit scopes are rules that determine how transaction commits and conflicts are handled within a PGD system. You can read more about them in *Commit scopes* .

You can manipulate commit scopes using the following functions:

- bdr.create_commit_scope
- bdr.alter_commit_scope
- bdr.drop_commit_scope

### Commit scope syntax

The overall grammar for commit scope rules is composed as follows:

```
commit_scope:
    commit_scope_operation [AND ...]

commit_scope_operation:
    commit_scope_group confirmation_level commit_scope_kind

commit_scope_target:
    { (node_group [, ...])
      | ORIGIN_GROUP }

commit_scope_group:
```

(continues on next page)

(continued from previous page)

```
{ ANY num [NOT] commit_scope_target
  | MAJORITY [NOT] commit_scope_target
  | ALL [NOT] commit_scope_target }

confirmation_level:
    [ ON { received|replicated|durable|visible } ]

commit_scope_kind:
{ GROUP COMMIT [ ( group_commit_parameter = value [, ... ] ) ] [ ABORT ON ( abort_on_
↪parameter = value ) ] [ DEGRADE ON (degrade_on_parameter = value [, ... ] ) TO␣
↪commit_scope_degrade_operation ]
  | QUORUM COMMIT [ ( commit_decision = { group | raft } ) ] ABORT ON ( timeout =␣
↪value )
  | CAMO [ DEGRADE ON ( degrade_on_parameter = value [, ... ] ) TO ASYNC ]
  | LAG CONTROL [ ( lag_control_parameter = value [, ... ] ) ]
  | SYNCHRONOUS COMMIT [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_
↪degrade_operation ] }

commit_scope_degrade_operation:
   commit_scope_group confirmation_level commit_scope_kind
```

Where `node_group` is the name of a PGD data node group.

### commit_scope_degrade_operation

The `commit_scope_degrade_operation` is either the same commit scope kind with a less restrictive commit scope group as the overall rule being defined, or is asynchronous (`ASYNC` ).

For instance, *Degrading commit scope rules* from an `ALL SYNCHRONOUS COMMIT` to a `MAJORITY SYNCHRONOUS COMMIT` or a `MAJORITY SYNCHRONOUS COMMIT` to an `ANY 3 SYNCHRONOUS COMMIT` or even an `ANY 3 SYNCHRONOUS COMMIT` to an `ANY 2 SYNCHRONOUS COMMIT` . You can also degrade from `SYNCHRONOUS COMMIT` to `ASYNC` . However, you cannot degrade from `SYNCHRONOUS COMMIT` to `GROUP COMMIT` or the other way around, regardless of the commit scope groups involved.

It is also possible to combine rules using `AND` , each with their own degradation clause:

```
ALL ORIGIN GROUP SYNCHRONOUS COMMIT DEGRADE ON (timeout = 10s) TO MAJORITY ORIGIN␣
↪GROUP SYNCHRONOUS COMMIT AND ANY 1 NOT ORIGIN GROUP SYNCHRONOUS COMMIT DEGRADE ON␣
↪(timeout = 20s) TO ASYNC
```

### Commit scope targets

### ORIGIN_GROUP

Instead of targeting a specific group, you can also use `ORIGIN_GROUP` , which dynamically refers to the bottommost group from which a transaction originates. Therefore, if you have a top level group, `top_group` , and two subgroups as children, `left_dc` and `right_dc` , then adding a commit scope like:

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := top_level_group,
    rule := MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT,
```

(continues on next page)

(continued from previous page)

```
    wait_for_ready := true
);
```

would mean that for transactions originating on a node in `left_dc` , a majority of the nodes of `left_dc` would need to confirm the transaction synchronously before the transaction is committed. Moreover, the same rule would also mean that for transactions originating from a node in `right_dc` , a majority of nodes from `right_dc` are required to confirm the transaction synchronously before it is committed. This saves the need to add two seperate rules, one for `left_dc` and one for `right_dc` , to the commit scope.

### Commit scope groups

#### ANY

Example: `ANY 2 (left_dc)`

A transaction under this commit scope group will be considered committed after any two nodes in the `left_dc` group confirm they processed the transaction.

#### ANY NOT

Example: `ANY 2 NOT (left_dc)`

A transaction under this commit scope group will be considered committed if any two nodes that aren't in the `left_dc` group confirm they processed the transaction.

#### MAJORITY

Example: `MAJORITY (left_dc)`

A transaction under this commit scope group will be considered committed if a majority of the nodes in the `left_dc` group confirm they processed the transaction.

#### MAJORITY NOT

Example: `MAJORITY NOT (left_dc)`

A transaction under this commit scope group will be considered committed if a majority of the nodes that aren't in the `left_dc` group confirm they processed the transaction.

#### ALL

Example: `ALL (left_dc)`

A transaction under this commit scope group will be considered committed if all of the nodes in the `left_dc` group confirm they processed the transaction.

When `ALL` is used with *SYNCHRONOUS COMMIT and GROUP COMMIT* , the *bdr.commit_decisions* setting must be set to `raft` to avoid reconciliation issues.

#### ALL NOT

Example: `ALL NOT (left_dc)`

A transaction under this commit scope group will be considered committed if all of the nodes that aren't in the `left_dc` group confirm they processed the transaction.

### Confirmation level

The confirmation level sets the point in time when a remote PGD node confirms that it reached a particular point in processing a transaction. `QUORUM COMMIT` only supports `durable` and `visible` .

### ON received

A transaction is confirmed immediately after receiving it, prior to starting the local application.

### ON replicated

A transaction is confirmed after applying changes of the transaction but before flushing them to disk.

### ON durable

A transaction is confirmed after all of its changes are flushed to disk.

### ON visible

This is the default visibility. A transaction is confirmed after all of its changes are flushed to disk and it's visible to concurrent transactions.

### Commit Scope kinds

More details of the commit scope kinds and details of their parameters:

- *SYNCHRONOUS COMMIT*
- *QUORUM COMMIT*
- *CAMO (Commit At Most Once)*
- *LAG CONTROL*
- *SYNCHRONOUS COMMIT and GROUP COMMIT*

Postgres GUC parameter value conventions .

```
SYNCHRONOUS COMMIT [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_
↪degrade_operation ]
```

### DEGRADE ON parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | Timeout in milliseconds (accepts other units) after which operation degrades. (0 means not set.) |
| *require_write_lead* | Boolean | False | Specifies whether the node must be a write lead to be able to switch to degraded operation. |

These set the conditions on which the commit scope rule will degrade to a less restrictive mode of operation.

### commit_scope_degrade_operation

The `commit_scope_degrade_operation` must be `SYNCHRONOUS COMMIT` with a less restrictive commit scope group—or must be asynchronous (`ASYNC` ).

### QUORUM COMMIT

Provides distributed transaction consistency by coordinating commit decisions across all participating nodes before any node commits.

See *QUORUM COMMIT* for more details.

```
QUORUM COMMIT [ ( commit_decision = { group | raft } ) ] ABORT ON ( timeout = value )
```

Supported commit scope groups are `ALL` , `MAJORITY ORIGIN GROUP` , `MAJORITY CLUSTER` , and `MAJORITY FOR EACH GROUP` . `MAJORITY` with an explicit group list, `ANY n` , and `NOT` aren't supported.

Only the `durable` and `visible` confirmation levels are supported with `QUORUM COMMIT` . Using `received` or `replicated` results in an error.

### QUORUM COMMIT parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| *commit_decision* | enum | group | Specifies how the commit decision is made. (*group* I *raft*). See *commit_decision settings* . |

### QUORUM COMMIT ABORT ON parameters

`ABORT ON` is mandatory for Quorum Commit.

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | Time to wait for quorum before aborting. (0 means not set.) |

### commit_decision settings

The value `group` means the origin node decides to commit once it receives the required confirmations from the commit scope group.

The value `raft` means the commit decision goes through PGD's built-in Raft consensus. This is slower than `group` but may be required for certain multi-group topologies.

### CAMO

With the client's cooperation, enables protection to prevent multiple insertions of the same transaction in failover scenarios.

See *Commit At Most Once* for more details.

```
CAMO [ DEGRADE ON ( degrade_on_parameter = value ) TO ASYNC ]
```

### DEGRADE ON parameters

Allows degrading to asynchronous operation on timeout.

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | Timeout in milliseconds (accepts other units) after which operation becomes asynchronous. (0 means not set.) |
| *require_write_lead* | Boolean | False | Specifies whether the node must be a write lead to be able to switch to asynchronous mode. |

## LAG CONTROL

Allows the configuration of dynamic rate-limiting controlled by replication lag.

See *LAG CONTROL* for more details.

```
LAG CONTROL [ ( lag_control_parameter = value [, ... ] ) ]
```

### LAG CONTROL parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| *max_lag_size* | int | 0 | The maximum lag in kB that a given node can have in the replication connection to another node. When the lag exceeds this maximum scaled by *max_commit_delay*, lag control adjusts the commit delay. |
| *max_lag_time* | interval | 0 | The maximum replication lag in milliseconds that the given origin can have with regard to a replication connection to a given downstream node. |
| *max_commit_delay* | interval | 0 | Configures the maximum delay each commit can take, in fractional milliseconds. If set to 0, it disables Lag Control. After each commit delay adjustment (for example, if the replication is lagging more than *max_lag_size* or *max_lag_time*), the commit delay is recalculated with the weight of the *bdr.lag_control_commit_delay_adjust* GUC. The *max_commit_delay* is a ceiling for the commit delay. |

- If `max_lag_size` and `max_lag_time` are set to 0, the LAG CONTROL is disabled.
- If `max_commit_delay` is not set or set to 0, the LAG CONTROL is disabled.

The lag size is derived from the delta of the `send_ptr` of the walsender to the `apply_ptr` of the receiver.

The lag time is calculated according to the following formula:

```
lag_time = (lag_size / apply_rate) * 1000;
```

Where `lag_size` is the delta between the `send_ptr` and `apply_ptr` (as used for `max_lag_size` ), and `apply_rate` is a weighted exponential moving average, following the simplified formula:

```
apply_rate = prev_apply_rate * (1 - apply_rate_weight) +
             ((apply_ptr_diff * apply_rate_weight) / diff_secs);
```

Where:

- `prev_apply_rate` was the previously configured `apply_rate` , before recalculating the new rate.
- `apply_rate_weight` is the value of the GUC `bdr.lag_tracker_apply_rate_weight` .
- `apply_ptr_diff` is the difference between the current `apply_ptr` and the `apply_ptr` at the point in time when the apply rate was last computed.
- `diff_secs` is the delta in seconds from the last time the apply rate was calculated.

### GROUP COMMIT

Allows commits to be confirmed by a consensus of nodes, controls conflict resolution settings, and, like *SYNCHRONOUS COMMIT* , has optional rule-degredation parameters.

```
GROUP COMMIT [ ( group_commit_parameter = value [, ...] ) ] [ ABORT ON ( abort_on_
↪parameter = value ) ] [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_
↪degrade_operation ]
```

### GROUP COMMIT parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| *transaction_tracking* | Boolean | Off/False | Specifies whether to track status of transaction. See *transaction_tracking settings* . |
| *conflict_resolution* | enum | async | Specifies how to handle conflicts. (*async`I`eager* ). See *conflict_resolution settings* . |
| *commit_decision* | enum | group | Specifies how the COMMIT decision is made. (*group`I`partner`I`raft*). See *commit_decision settings* . |

### ABORT ON parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | Timeout in milliseconds (accepts other units). (0 means not set.) |

continues on next page

Table 154 – continued from previous page

| Parameter | Type | Default | Description |
|---|---|---|---|
| *require_write_lead* | Boolean | False | CAMO only. If set, then for a transaction to switch to local (async) mode, a consensus request is required. |

### DEGRADE ON parameters

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | Timeout in milliseconds (accepts other units) after which operation degrades. (0 means not set.) |
| *require_write_lead* | Boolean | False | Specifies whether the node must be a write lead to be able to switch to degraded operation. |

### transaction_tracking settings

When set to true, two-phase commit transactions:

- Look up commit decisions when a writer is processing a PREPARE message.
- When recovering from an interruption, look up the transactions prepared before the interruption. When found, it then looks up the commit scope of the transaction and any corresponding RAFT commit decision. Suppose the node is the origin of the transaction and doesn't have a RAFT commit decision, and `transaction_tracking` is on in the commit scope. In that case, it periodically looks for a RAFT commit decision for this unresolved transaction until it's committed or aborted.

### conflict_resolution settings

The value `async` means resolve conflicts asynchronously during replication using the conflict resolution policy.

The value `eager` means that conflicts are resolved eagerly during COMMIT by aborting one of the conflicting transactions.

Eager is only available with `MAJORITY` or `ALL` commit scope groups.

When used with the `ALL` commit scope group, the *bdr.commit_decisions* must be set to `raft` to avoid reconcilation issue.

See *Missing-column conflict resolution* .

### GROUP COMMIT commit_decision settings

The value `group` means the preceding `commit_scope_group` specification also affects the COMMIT decision, not just durability.

The value `partner` means the partner node decides whether transactions can be committed. This value is allowed only on groups with 2 data nodes.

The value `raft` means the decision makes use of PGD's built-in Raft consensus. Once all the nodes in the selected commit scope group have confirmed the transaction, to ensure that all the nodes in the PGD cluster have noted the transaction, it is noted with the all-node Raft.

This option must be used when the `ALL` commit scope group is being used to ensure no divergence between the nodes over the decision. This option may have low performance.

See *Commit decisions* .

### commit_scope_degrade_operation settings

The `commit_scope_degrade_operation` must be `GROUP_COMMIT` with a less restrictive commit scope group—or must be asynchronous (`ASYNC` ).

## Conflicts

### Conflict detection

### List of conflict types

PGD recognizes the following conflict types, which can be used as the `conflict_type` parameter:

| Conflict type | Description |
|---|---|
| *insert_exists* | An incoming insert conflicts with an existing row by way of a primary key or a unique key/index. |
| *update_differing* | An incoming update's key row differs from a local row. This can happen only when using *Row version conflict detection* . |
| *update_origin_change* | An incoming update is modifying a row that was last changed by a different node. |
| *update_missing* | An incoming update is trying to modify a row that doesn't exist. |
| *update_recently_deleted* | An incoming update is trying to modify a row that was recently deleted. |
| *update_pkey_exists* | An incoming update has modified the *PRIMARY KEY* to a value that already exists on the node that's applying the change. |
| *multiple_unique_conflicts* | An incoming row conflicts with multiple rows per UNIQUE/EXCLUDE indexes of the target table. |
| *delete_recently_updated* | An incoming delete with an older commit timestamp than the most recent update of the row on the current node or when using *Row version conflict detection* . |
| *delete_missing* | An incoming delete is trying to remove a row that doesn't exist. |
| *target_column_missing* | The target table is missing one or more columns present in the incoming row. |
| *source_column_missing* | The incoming row is missing one or more columns that are present in the target table. |
| *target_table_missing* | The target table is missing. |
| *apply_error_ddl* | An error was thrown by Postgres when applying a replicated DDL command. |

### Conflict resolution

Most conflicts can be resolved automatically. PGD defaults to a last-update-wins mechanism or, more accurately, the `update_if_newer` conflict resolver. This mechanism retains the most recently inserted or changed row of the two conflicting ones based on the same commit timestamps used for conflict detection. The behavior in certain corner-case scenarios depends on the settings used for `bdr.create_node_group` and alternatively for `bdr.alter_node_group` .

PGD lets you override the default behavior of conflict resolution by using the following function.

### List of conflict resolvers

Several conflict resolvers are available in PGD, with differing coverages of the conflict types they can handle:

| Resolver | Description |
|---|---|
| *error* | Throws an error and stops replication. |
| *skip* | Skips processing the remote change and continues replication with the next change. Can be used for *insert_exists*, *update_differing*, *update_origin_change*, *update_missing*, *update_recently_deleted*, *update_pkey_exists*, *delete_recently_updated*, *delete_missing*, *target_table_missing*, *target_column_missing*, and *source_column_missing* conflict types. |
| *skip_if_recently_dropped* | Skips the remote change if it's for a table that doesn't exist downstream because it was recently (within one day) dropped on the downstream. Throw an error otherwise. Can be used for the *target_table_missing* conflict type. This conflict resolver can pose challenges if a table with the same name is re-created shortly after it's dropped. In that case, one of the nodes might see the DMLs on the re-created table before it sees the DDL to re-create the table. It then incorrectly skips the remote data, assuming that the table is recently dropped, and causes data loss. We recommend that when using this resolver, you don't reuse the object names immediately after they're dropped. |
| *skip_transaction* | Skips the whole transaction that generated the conflict. |
| *update_if_newer* | Updates if the remote row was committed later (as determined by the wall clock of the originating node) than the conflicting local row. If the timestamps are same, the node id is used as a tie-breaker to ensure that same row is picked on all nodes (higher nodeid wins). Can be used for *insert_exists*, *update_differing*, *update_origin_change*, and *update_pkey_exists* conflict types. |
| *update* | Always performs the replicated action. Can be used for *insert_exists* (turns the *INSERT* into *UPDATE*), *update_differing*, *update_origin_change*, *update_pkey_exists*, and *delete_recently_updated* (performs the delete). |
| *insert_or_skip* | Tries to build a new row from available information sent by the origin and INSERT it. If there isn't enough information available to build a full row, skips the change. Can be used for *update_missing* and *update_recently_deleted* conflict types. |
| *insert_or_error* | Tries to build new row from available information sent by origin and insert it. If there isn't enough information available to build full row, throws an error and stops the replication. If there isn't enough information available to build full row, throws an error and stops the replication. Can be used for *update_missing* and *update_recently_deleted* conflict types. |
| *ignore* | Ignores any missing target column and continues processing.Can be used for the *target_column_missing* conflict type. |
| *ignore_if_null* | Ignores a missing target column if the extra column in the remote row contains a NULL value. Otherwise, throws an error and stops replication. Can be used for the *target_column_missing* conflict type. |
| *use_default_value* | Fills the missing column value with the default (including NULL if that's the column default) and continues processing. Any error while processing the default or violation of constraints (that is, NULL default on NOT NULL column) stops replication. Can be used for the *source_column_missing* conflict type. |

The `insert_exists` , `update_differing` , `update_origin_change` , `update_missing` , `multiple_unique_conflicts` , `update_recently_deleted` , `update_pkey_exists` , `delete_recently_updated` , and `delete_missing` conflict types can also be resolved by user-defined logic using *Stream triggers* .

This matrix shows the conflict types each conflict resolver can handle.

| | in-sert_e: ists | up-date_c fer-ing | up-date_c gin_ch | up-date_r ing | up-date_r cently_ | up-date_p ists | delete_ cently_ dated | delete_ ing | tar-get_co umn_n ing | source umn_n ing | tar-get_ta ble_mi ing | mul-ti-ple_unique_con-flicts |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| error | X | X | X | X | X | X | X | X | X | X | X | X |
| skip | X | X | X | X | X | X | X | X | X | X | X | X |
| skip_if_re-cently_drop | | | | | | | | | | | X | |
| up-date_if_new | X | X | X | | | X | | | | | | X |
| update | X | X | X | | | X | X | | | | | X |
| in-sert_or_skip | | | | X | X | | | | | | | |
| in-sert_or_er-ror | | | | X | X | | | | | | | |
| ignore | | | | | | | | | X | | | |
| ig-nore_if_null | | | | | | | | | X | | | |
| use_de-fault_value | | | | | | | | | | X | | |
| con-flict_trig-ger | X | X | X | X | X | X | X | X | | | | X |

### Default conflict resolvers

| Conflict type | Resolver |
|---|---|
| insert_exists | update_if_newer |
| update_differing | update_if_newer |
| update_origin_change | update_if_newer |
| update_missing | insert_or_skip |
| update_recently_deleted | skip |
| update_pkey_exists | update_if_newer |
| multiple_unique_conflicts | error |
| delete_recently_updated | skip |
| delete_missing | skip |
| target_column_missing | ignore_if_null |
| source_column_missing | use_default_value |
| target_table_missing (see note) | skip_if_recently_dropped |
| apply_error_ddl | error |

The conflict resolution represents the kind of resolution chosen by the conflict resolver and corresponds to the specific action that was taken to resolve the conflict.

The following conflict resolutions are currently supported for the `conflict_resolution` parameter:

| Resolution | Description |
|---|---|
| *apply_remote* | The remote (incoming) row was applied. |

continues on next page

Table 160 – continued from previous page

| Resolution | Description |
|---|---|
| *skip* | Processing of the row was skipped (no change was made locally). |
| *merge* | A new row was created, merging information from remote and local row. |
| *user* | User code (a conflict trigger) produced the row that was written to the target table. |

### Conflict logging

Starting with version 6.0, PGD doesn't log conflicts to the `bdr.conflict_history` table by default. This is because the table can grow large and cause performance issues. You can enable conflict logging by using the bdr.alter_node_set_log_config function. This function gives you granular control over which conflicts to log, or you can set it to log all conflicts like this:

```
SELECT bdr.alter_node_set_log_config(`nodename`, false, true, NULL, NULL);
```

Run this command on the named node to enable logging of all conflicts on that particular node. If you want to enable logging on all nodes, run this command on each node in the PGD group.

### Conflict functions

#### bdr.alter_table_conflict_detection

Allows the table owner to change how conflict detection works for a given table.

### Synopsis

```
bdr.alter_table_conflict_detection(relation regclass,
                                   method text,
                                   column_name name DEFAULT NULL)
```

### Parameters

- `relation` — Name of the relation for which to set the new conflict detection method.
- `method` — The conflict detection method to use.
- `column_name` — The column to use for storing the column detection data. This can be skipped, in which case the column name is chosen based on the conflict detection method. The `row_origin` method doesn't require an extra column for metadata storage.

The recognized methods for conflict detection are:

- `row_origin` — Origin of the previous change made on the tuple (see *Origin conflict detection* ). This is the only method supported that doesn't require an extra column in the table.
- `row_version` — Row version column (see *Row version conflict detection* ).
- `column_commit_timestamp` — Per-column commit timestamps (described in *CLCD* ).
- `column_modify_timestamp` — Per-column modification timestamp (described in *CLCD* ).

### Notes

For more information about the difference between `column_commit_timestamp` and `column_modify_timestamp` conflict detection methods, see *Timestamps in column-level conflict resolution* .

This function uses the same replication mechanism as `DDL` statements. This means the replication is affected by the *ddl filters* configuration.

The function takes a `DML` global lock on the relation for which column-level conflict resolution is being enabled.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction, and the changes are visible to the current transaction.

Only the owner of the `relation` can execute the `bdr.alter_table_conflict_detection` function unless `bdr.backwards_compatibility` is set to 30618 or less.

**Warning**

When changing the conflict detection method from one that uses an extra column to store metadata, that column is dropped.

**Warning**

This function disables CAMO and gives a warning, as long as warnings aren't disabled with *bdr.camo_enable_client_warnings* .

## bdr.alter_node_set_conflict_resolver

This function sets the behavior of conflict resolution on a given node.

### Synopsis

```
bdr.alter_node_set_conflict_resolver(node_name text,
                                     conflict_type text,
                                     conflict_resolver text)
```

### Parameters

- `node_name` — Name of the node that's being changed.
- `conflict_type` — Conflict type for which to apply the setting (see *List of conflict types* ).
- `conflict_resolver` — Resolver to use for the given conflict type (see *List of conflict resolvers* ).

### Notes

Currently you can change only the local node. The function call isn't replicated. If you want to change settings on multiple nodes, you must run the function on each of them.

The configuration change made by this function overrides any default behavior of conflict resolutions specified by `bdr.create_node_group` or `bdr.alter_node_group` .

This function is transactional. You can roll back the changes, and they are visible to the current transaction.

### bdr.alter_node_set_log_config

Set the conflict logging configuration for a node.

### Synopsis

```
bdr.alter_node_set_log_config(node_name text,
                              log_to_file bool DEFAULT true,
                              log_to_table bool DEFAULT true,
                              conflict_type text[] DEFAULT NULL,
                              conflict_resolution text[] DEFAULT NULL)
```

### Parameters

- `node_name` — Name of the node that's being changed.
- `log_to_file` — Whether to log to the node log file.
- `log_to_table` — Whether to log to the `bdr.conflict_history` table.
- `conflict_type` — Conflict types to log. NULL (the default) means all.
- `conflict_resolution` — Conflict resolutions to log. NULL (the default) means all.

### Notes

You can change only the local node. The function call isn't replicated. If you want to change settings on multiple nodes, you must run the function on each of them.

This function is transactional. You can roll back the changes, and they're visible to the current transaction.

### Listing conflict logging configurations

The view `bdr.node_log_config` shows all the logging configurations. It lists the name of the logging configuration, where it logs, and the conflict type and resolution it logs.

### Logging conflicts to a table

If `log_to_table` is set to true, conflicts are logged to a table. The target table for conflict logging is `bdr.conflict_history`.

This table is range partitioned on the column `local_time`. The table is managed by autopartition. By default, a new partition is created for every day, and conflicts of the last one month are maintained. After that, the old partitions are dropped. Autopartition creates between 7 and 14 partitions in advance. bdr_superuser can change these defaults.

Since conflicts generated for all tables managed by PGD are logged to this table, it's important to ensure that only legitimate users can read the conflicted data. PGD does this by defining ROW LEVEL SECURITY policies on the `bdr.conflict_history` table. Only owners of the tables are allowed to read conflicts on the respective tables. If the underlying tables have RLS policies defined, enabled, and enforced, then even owners can't read the conflicts. RLS policies created with the FORCE option also apply to owners of the table. In that case, some or all rows in the underlying table might not be readable even to the owner. So PGD also enforces a stricter policy on the conflict log table.

The predefined role bdr_read_all_conflicts can be granted to users who need to see all conflict details logged to the `bdr.conflict_history` table without also granting them bdr_superuser role.

The default role bdr_read_all_stats has access to a catalog view called `bdr.conflict_history_summary`. This view doesn't contain user data, allowing monitoring of any conflicts logged.

### Replication set management

#### bdr.create_replication_set

This function creates a replication set.

Replication of this command is affected by DDL replication configuration, including DDL filtering settings.

#### Synopsis

```
bdr.create_replication_set(set_name name,
                           replicate_insert boolean DEFAULT true,
                           replicate_update boolean DEFAULT true,
                           replicate_delete boolean DEFAULT true,
                           replicate_truncate boolean DEFAULT true,
                           autoadd_tables boolean DEFAULT false,
                           autoadd_existing boolean DEFAULT true)
```

#### Parameters

- `set_name` _ Name of the new replication set. Must be unique across the PGD group.
- `replicate_insert` _ Indicates whether to replicate inserts into tables in this replication set.
- `replicate_update` _ Indicates whether to replicate updates of tables in this replication set.
- `replicate_delete` _ Indicates whether to replicate deletes from tables in this replication set.
- `replicate_truncate` _ Indicates whether to replicate truncates of tables in this replication set.
- `autoadd_tables` _ Indicates whether to replicate newly created (future) tables to this replication set
- `autoadd_existing` _ Indicates whether to add all existing user tables to this replication set. This parameter has an effect only if `autoadd_tables` is set to `true` .

#### Notes

By default, new replication sets don't replicate DDL or PGD administration function calls. See *DDL filters* for how to set up DDL replication for replication sets. A preexisting DDL filter is set up for the default group replication set that replicates all DDL and admin function calls. It's created when the group is created but can be dropped in case you don't want the PGD group default replication set to replicate DDL or the PGD administration function calls.

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *DDL filters*

configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

#### bdr.alter_replication_set

This function modifies the options of an existing replication set.

Replication of this command is affected by DDL replication configuration, including DDL filtering settings.

### Synopsis

```
bdr.alter_replication_set(set_name name,
                          replicate_insert boolean DEFAULT NULL,
                          replicate_update boolean DEFAULT NULL,
                          replicate_delete boolean DEFAULT NULL,
                          replicate_truncate boolean DEFAULT NULL,
                          autoadd_tables boolean DEFAULT NULL)
```

### Parameters

- `set_name` _ Name of an existing replication set.
- `replicate_insert` _ Indicates whether to replicate inserts into tables in this replication set.
- `replicate_update` _ Indicates whether to replicate updates of tables in this replication set.
- `replicate_delete` _ Indicates whether to replicate deletes from tables in this replication set.
- `replicate_truncate` _ Indicates whether to replicate truncates of tables in this replication set.
- `autoadd_tables` _ Indicates whether to add newly created (future) tables to this replication set.

Any of the options that are set to NULL (the default) remain the same as before.

### Notes

This function uses the same replication mechanism as `DDL` statements. This means the replication is affected by the *DDL filters*

configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

### bdr.drop_replication_set

This function removes an existing replication set.

Replication of this command is affected by DDL replication configuration, including DDL filtering settings.

### Synopsis

```
bdr.drop_replication_set(set_name name)
```

### Parameters

- `set_name` _ Name of an existing replication set.

### Notes

This function uses the same replication mechanism as `DDL` statements. This means the replication is affected by the *ddl filters*

configuration.

The function takes a DDL global lock.

This function is transactional. You can roll back the effects with the ROLLBACK of the transaction. The changes are visible to the current transaction.

**Warning**

Don't drop a replication set that's being used by at least another node because doing so stops replication on that node. If that happens, unsubscribe the affected node from that replication set. For the same reason, don't drop a replication set with a join operation in progress when the node being joined is a member of that replication set. Replication set membership is checked only at the beginning of the join. This happens because the information on replication set usage is local to each node, so that you can configure it on a node before it joins the group.

You can manage replication set subscriptions for a node using alter_node_replication_sets .

### bdr.alter_node_replication_sets

This function changes the replication sets a node publishes and is subscribed to.

### Synopsis

```
bdr.alter_node_replication_sets(node_name name,
                                set_names text[])
```

### Parameters

- node_name _ The node to modify. Currently must be a local node.
- set_names _ Array of replication sets to replicate to the specified node. An empty array results in the use of the group default replication set.

### Notes

This function is executed only on the local node and isn't replicated in any manner.

The replication sets listed aren't checked for existence, since this function is designed to execute before the node joins. Be careful to specify replication set names correctly to avoid errors.

This behavior allows for calling the function not only on the node that's part of the PGD group but also on a node that hasn't joined any group yet. This approach limits the data synchronized during the join. However, the schema is always fully synchronized without regard to the replication sets setting. All tables are copied across, not just the ones specified in the replication set. You can drop unwanted tables by referring to the bdr.tables catalog table. (These might be removed automatically in later versions of PGD.) This is currently true even if the *DDL filters*

configuration otherwise prevents replication of DDL.

The replication sets that the node subscribes to after this call are published by the other nodes for actually replicating the changes from those nodes to the node where this function is executed.

### Replication set membership

### bdr.replication_set_add_table

This function adds a table to a replication set.

This function adds a table to a replication set and starts replicating changes from the committing of the transaction that contains the call to the function. Any existing data the table might have on a node isn't synchronized. Replication of this command is affected by DDL replication configuration, including DDL filtering settings.

### Synopsis

```
bdr.replication_set_add_table(relation regclass,
                              set_name name DEFAULT NULL,
                              columns text[] DEFAULT NULL,
                              row_filter text DEFAULT NULL)
```

### Parameters

- `relation` _ Name or Oid of a table.
- `set_name` _ Name of the replication set. If NULL (the default), then the PGD group default replication set is used.
- `columns` _ Reserved for future use (currently does nothing and must be NULL).
- `row_filter` _ SQL expression to use for filtering the replicated rows. If this expression isn't defined (that is, it's set to NULL, the default) then all rows are sent.

The `row_filter` specifies an expression producing a Boolean result, with NULLs. Expressions evaluating to True or Unknown replicate the row. A False value doesn't replicate the row. Expressions can't contain subqueries or refer to variables other than columns of the current row being replicated. You can't reference system columns.

`row_filter` executes on the origin node, not on the target node. This puts an additional CPU overhead on replication for this specific table but completely avoids sending data for filtered rows. Hence network bandwidth is reduced and overhead on the target node is applied.

`row_filter` never removes `TRUNCATE` commands for a specific table. You can filter away `TRUNCATE` commands at the replication set level.

You can replicate just some columns of a table. See *Replicating between nodes with differences* .

### Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *DDL filters*

configuration.

If the `row_filter` isn't NULL, the function takes a `DML` global lock on the relation that's being added to the replication set. Otherwise it takes just a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

### bdr.replication_set_remove_table

This function removes a table from the replication set.

Replication of this command is affected by DDL replication configuration, including DDL filtering settings.

### Synopsis

```
bdr.replication_set_remove_table(relation regclass,
                                 set_name name DEFAULT NULL)
```

### Parameters

- relation _ Name or Oid of a table.
- set_name _ Name of the replication set. If NULL (the default), then the PGD group default replication set is used.

### Notes

This function uses the same replication mechanism as DDL statements. This means the replication is affected by the *DDL filters*

configuration.

The function takes a DDL global lock.

This function is transactional. You can roll back the effects with the ROLLBACK of the transaction. The changes are visible to the current transaction.

### DDL replication filtering

See also *DDL replication filtering* .

### bdr.replication_set_add_ddl_filter

This function adds a DDL filter to a replication set.

Any DDL that matches the given filter is replicated to any node that's subscribed to that set. This function also affects replication of PGD admin functions.

This function doesn't prevent execution of DDL on any node. It only alters whether DDL is replicated to other nodes. Suppose two nodes have a replication filter between them that excludes all index commands. Index commands can still be executed freely by directly connecting to each node and executing the desired DDL on that node.

The DDL filter can specify a command_tag and role_name to allow replication of only some DDL statements. The command_tag is the same as those used by event triggers

for regular PostgreSQL commands. A typical example might be to create a filter that prevents additional index commands on a logical standby from being replicated to all other nodes.

You can filter the PGD admin functions used by using a tagname matching the qualified function name. For example, bdr.replication_set_add_table is the command tag for the function of the same name. In this case, this tag allows all PGD functions to be filtered using bdr.* .

The role_name is used for matching against the current role that's executing the command. Both command_tag and role_name are evaluated as regular expressions, which are case sensitive.

### Synopsis

```
bdr.replication_set_add_ddl_filter(set_name name,
                                   ddl_filter_name text,
                                   command_tag text,
```

(continues on next page)

(continued from previous page)

```
                                    role_name text DEFAULT NULL,
                                    base_relation_name text DEFAULT NULL,
                                    query_match text DEFAULT NULL,
                                    exclusive boolean DEFAULT FALSE)
```

### Parameters

- `set_name` _ Name of the replication set. If NULL then the PGD group default replication set is used.
- `ddl_filter_name` _ Name of the DDL filter. This name must be unique across the whole PGD group.
- `command_tag` _ Regular expression for matching command tags. NULL means match everything.
- `role_name` _ Regular expression for matching role name. NULL means match all roles.
- `base_relation_name` _ Reserved for future use. Must be NULL.
- `query_match` _ Regular expression for matching the query. NULL means match all queries.
- `exclusive` _ If true, other matched filters aren't taken into consideration (that is, only the exclusive filter is applied). When multiple exclusive filters match, an error is thrown. This parameter is useful for routing specific commands to a specific replication set, while keeping the default replication through the main replication set.

### Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *DDL filters*

configuration. This also means that replication of changes to DDL filter configuration is affected by the existing DDL filter configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

To view the defined replication filters, use the view `bdr.ddl_replication` .

### Examples

To include only PGD admin functions, define a filter like this:

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, mygroup_admin, $$bdr\..*$$);
```

To exclude everything except for index DDL:

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, index_filter,
               ^(?!(CREATE INDEX|DROP INDEX|ALTER INDEX)).*);
```

To include all operations on tables and indexes but exclude all others, add two filters: one for tables and one for indexes. This example shows that multiple filters provide the union of all allowed DDL commands:

```
SELECT bdr.replication_set_add_ddl_filter(bdrgroup,index_filter, ^((?!INDEX).)*$);
SELECT bdr.replication_set_add_ddl_filter(bdrgroup,table_filter, ^((?!TABLE).)*$);
```

### bdr.replication_set_remove_ddl_filter

This function removes the DDL filter from a replication set.

Replication of this command is affected by DDL replication configuration, including the DDL filtering settings.

### Synopsis

```
bdr.replication_set_remove_ddl_filter(set_name name,
                                      ddl_filter_name text)
```

### Parameters

- `set_name` _ Name of the replication set. If NULL then the PGD group default replication set is used.
- `ddl_filter_name` _ Name of the DDL filter to remove.

### Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *DDL filters* configuration. This also means that replication of changes to the DDL filter configuration is affected by the existing DDL filter configuration.

The function takes a `DDL` global lock.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

### Testing and tuning commands

EDB Postgres Distributed has tools that help with testing and tuning your PGD clusters. For background, see *Postgres Distributed terminology* .

### pgd_bench

### Synopsis

A benchmarking tool for EDB Postgres Distributed deployments.

```
pgd_bench [OPTION]... [DBNAME] [DBNAME2]
```

`DBNAME` can be a conninfo string of the format: `"host=10.1.1.2 user=postgres dbname=master"`

See *pgd_bench in Testing and tuning* for examples of pgd_bench options and usage.

### Options

The pgd_bench command is implemented as a wrapper around the pgbench command. This means that it shares many of the same options and created tables named `pgbench` as it performs its testing.

Options that are specific to pgd_bench include the following.

### Setting mode

`-m` or `--mode`

The mode can be set to `regular` , `camo` , or `failover` . The default is `regular` .

- `regular` — Only a single node is needed to run pgd_bench.
- `camo` — A second node must be specified to act as the CAMO partner. (CAMO must be set up.)
- `failover` — A second node must be specified to act as the failover.

When using `-m failover` , an additional option `--retry` is available. This option instructs pgd_bench to retry transactions when there's a failover. The `--retry` option is automatically enabled when `-m camo` is used.

When using `-m camo` and providing a custom script, the SQL commands in the script must be wrapped in SQL transaction commands. That is, the first SQL command must be `BEGIN` , and the final SQL command must be `COMMIT` .

### Setting GUC variables

`-o` or `--set-option`

This option is followed by `NAME=VALUE` entries, which are applied using the Postgres SET command on each server that pgd_bench connects to, and only those servers.

The other options are identical to the Postgres pgbench command. For details, see the PostgreSQL pgbench documentation.

The complete list of options (pgd_bench and pgbench) follow.

### Initialization options

- `-i, --initialize` — Invoke initialization mode.
- `-I, --init-steps=[dtgGvpf]+` (default `"dtgvp"` ) — Run selected initialization steps.
- `d` — Drop any existing pgbench tables.
- `t` — Create the tables used by the standard pgbench scenario.
- `g` — Generate data client-side and load it into the standard tables, replacing any data already present.
- `G` — Generate data server-side and load it into the standard tables, replacing any data already present.
- `v` — Invoke `VACUUM` on the standard tables.
- `p` — Create primary key indexes on the standard tables.
- `f` — Create foreign key constraints between the standard tables.
- `-F, --fillfactor=NUM` — Set fill factor.
- `-n, --no-vacuum` — Don't run `VACUUM` during initialization.
- `-q, --quiet` — Quiet logging (one message every 5 seconds).
- `-s, --scale=NUM` — Scaling factor.
- `--foreign-keys` — Create foreign key constraints between tables.
- `--index-tablespace=TABLESPACE` — Create indexes in the specified tablespace.
- `--partition-method=(range|hash)` — Partition `pgbench_accounts` with this method. The default is `range` .

- `--partitions=NUM` — Partition `pgbench_accounts` into `NUM` parts. The default is `0`.
- `--tablespace=TABLESPACE` — Create tables in the specified tablespace.
- `--unlogged-tables` — Create tables as unlogged tables. (Note: Unlogged tables aren't replicated.)

### Options to select what to run

- `-b, --builtin=NAME[@W]` — Add built-in script NAME weighted at W. The default is 1. Use `-b list` to list available scripts.
- `-f, --file=FILENAME[@W]` — Add script `FILENAME` weighted at W. The default is 1.
- `-N, --skip-some-updates` — Updates of pgbench_tellers and pgbench_branches. Same as `-b simple-update`.
- `-S, --select-only` — Perform SELECT-only transactions. Same as `-b select-only`.

### Benchmarking options

- `-c, --client=NUM` — Number of concurrent database clients. The default is 1.
- `-C, --connect` — Establish new connection for each transaction.
- `-D, --define=VARNAME=VALUE` — Define variable for use by custom script.
- `-j, --jobs=NUM` — Number of threads. The default is 1.
- `-l, --log` — Write transaction times to log file.
- `-L, --latency-limit=NUM` — Count transactions lasting more than NUM ms as late.
- `-m, --mode=regular|camo|failover` — Mode in which to run pgbench. The default is `regular`.
- `-M, --protocol=simple|extended|prepared` — Protocol for submitting queries. The default is `simple`.
- `-n, --no-vacuum` — Don't run `VACUUM` before tests.
- `-o, --set-option=NAME=VALUE` — Specify runtime `SET` option.
- `-P, --progress=NUM` — Show thread progress report every NUM seconds.
- `-r, --report-per-command` — Latencies, failures, and retries per command.
- `-R, --rate=NUM` — Target rate in transactions per second.
- `-s, --scale=NUM` — Report this scale factor in output.
- `-t, --transactions=NUM` — Number of transactions each client runs. The default is 10.
- `-T, --time=NUM` — Duration of benchmark test, in seconds.
- `-v, --vacuum-all` — Vacuum all four standard tables before tests.
- `--aggregate-interval=NUM` — Data over NUM seconds.
- `--failures-detailed` — Report the failures grouped by basic types.
- `--log-prefix=PREFIX` — Prefix for transaction time log file. The default is `pgbench_log`.
- `--max-tries=NUM` — Max number of tries to run transaction. The default is `1`.
- `--progress-timestamp` — Use Unix epoch timestamps for progress.
- `--random-seed=SEED` — Set random seed (`time`, `rand`, `integer`).
- `--retry` — Retry transactions on failover. Used with `-m`.

- `--sampling-rate=NUM` — Fraction of transactions to log, for example, 0.01 for 1%.
- `--show-script=NAME` — Show built-in script code, then exit.
- `--verbose-errors` — Print messages of all errors.

### Common options:

- `-d, --debug` — Print debugging output.
- `-h, --host=HOSTNAME` — Database server host or socket directory.
- `-p, --port=PORT` — Database server port number.
- `-U, --username=USERNAME` — Connect as specified database user.
- `-V, --version` — Output version information, then exit.
- `-?, --help` — Show help, then exit.

## Global sequence management interfaces

PGD provides an interface for converting between a standard PostgreSQL sequence and the PGD global sequence.

The following functions are considered to be `DDL` , so DDL replication and global locking applies to them.

## Sequence functions

### bdr.alter_sequence_set_kind

Allows the owner of a sequence to set the kind of a sequence. Once set, `seqkind` is visible only by way of the `bdr.sequences` view. In all other ways, the sequence appears as a normal sequence.

PGD treats this function as `DDL` , so DDL replication and global locking applies, if it's currently active. See *Running DDL safely* .

### Synopsis

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text, start bigint DEFAULT NULL)
```

### Parameters

- `seqoid` — Name or Oid of the sequence to alter.
- `seqkind` — `local` for a standard PostgreSQL sequence, `snowflakeid` or `galloc` for globally unique PGD sequences, or `timeshard` for legacy globally unique sequence.
- `start` — Allows specifying new starting point for galloc and local sequences.

### Notes

When changing the sequence kind to `galloc` , the first allocated range for that sequence uses the sequence start value as the starting point. When there are existing values that were used by the sequence before it was changed to `galloc` , we recommend moving the starting point so that the newly generated values don't conflict with the existing ones using the following command:

```
ALTER SEQUENCE seq_name START starting_value RESTART
```

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *DDL filters*

configuration.

The function takes a global `DDL` lock. It also locks the sequence locally.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Only the owner of the sequence can execute the `bdr.alter_sequence_set_kind` function, unless `bdr.backwards_compatibility` is set to 30618 or lower.

### bdr.extract_timestamp_from_snowflakeid

This function extracts the timestamp component of the `snowflakeid` sequence. The return value is of type `timestamptz`.

#### Synopsis

```
bdr.extract_timestamp_from_snowflakeid(snowflakeid bigint)
```

#### Parameters

- `snowflakeid` — Value of a `snowflakeid` sequence.

#### Notes

This function executes only on the local node.

### bdr.extract_nodeid_from_snowflakeid

This function extracts the nodeid component of the `snowflakeid` sequence.

#### Synopsis

```
bdr.extract_nodeid_from_snowflakeid(snowflakeid bigint)
```

#### Parameters

- `snowflakeid` — Value of a `snowflakeid` sequence.

#### Notes

This function executes only on the local node.

### bdr.extract_localseqid_from_snowflakeid

This function extracts the local sequence value component of the `snowflakeid` sequence.

### Synopsis

```
bdr.extract_localseqid_from_snowflakeid(snowflakeid bigint)
```

### Parameters

- `snowflakeid` — Value of a `snowflakeid` sequence.

### Notes

This function executes only on the local node.

### bdr.timestamp_to_snowflakeid

This function converts a timestamp value to a dummy `snowflakeid` sequence value.

This is useful for doing indexed searches or comparisons of values in the `snowflakeid` column and for a specific timestamp.

For example, given a table `foo` with a column `id` that's using a `snowflakeid` sequence, you can get the number of changes since yesterday midnight like this:

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_snowflakeid(yesterday)
```

A query formulated this way uses an index scan on the column `id` .

### Synopsis

```
bdr.timestamp_to_snowflakeid(ts timestamptz)
```

### Parameters

- `ts` — Timestamp to use for the `snowflakeid` sequence generation.

### Notes

This function executes only on the local node.

### bdr.extract_timestamp_from_timeshard

This function extracts the timestamp component of the `timeshard` sequence. The return value is of type `timestamptz` .

### Synopsis

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

### Parameters

- `timeshard_seq` — Value of a `timeshard` sequence.

### Notes

This function executes only on the local node.

### bdr.extract_nodeid_from_timeshard

This function extracts the nodeid component of the `timeshard` sequence.

### Synopsis

```
bdr.extract_nodeid_from_timeshard(timeshard_seq bigint)
```

### Parameters

- `timeshard_seq` — Value of a `timeshard` sequence.

### Notes

This function executes only on the local node.

### bdr.extract_localseqid_from_timeshard

This function extracts the local sequence value component of the `timeshard` sequence.

### Synopsis

```
bdr.extract_localseqid_from_timeshard(timeshard_seq bigint)
```

### Parameters

- `timeshard_seq` — Value of a `timeshard` sequence.

### Notes

This function executes only on the local node.

### bdr.timestamp_to_timeshard

This function converts a timestamp value to a dummy `timeshard` sequence value.

This is useful for doing indexed searches or comparisons of values in the `timeshard` column and for a specific timestamp.

For example, given a table `foo` with a column `id` that's using a `timeshard` sequence, you can get the number of changes since yesterday midnight like this:

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_timeshard(yesterday)
```

A query formulated this way uses an index scan on the column `id` .

### Synopsis

```
bdr.timestamp_to_timeshard(ts timestamptz)
```

### Parameters

- `ts` — Timestamp to use for the `timeshard` sequence generation.

### Notes

This function executes only on the local node.

### bdr.galloc_chunk_info

This function retrieves the ranges allocated to a galloc sequence on the local node.

An empty result set will be returned if the sequence has not yet been accessed on the local node.

An ERROR will be raised if the provided sequence name is not a galloc sequence.

### Synopsis

```
bdr.galloc_chunk_info(seqname regclass)
```

### Parameters

- `seqname` - the name of the galloc sequence to query

### Notes

This function executes only on the local node.

### KSUUID v2 functions

Functions for working with `KSUUID` v2 data, K-Sortable UUID data. See also *KSUUID in the sequences documentation* .

### bdr.gen_ksuuid_v2

This function generates a new `KSUUID` v2 value using the value of timestamp passed as an argument or current system time if NULL is passed. If you want to generate KSUUID automatically using the system time, pass a NULL argument.

The return value is of type UUID.

### Synopsis

```
bdr.gen_ksuuid_v2(timestamptz)
```

### Notes

This function executes only on the local node.

### bdr.ksuuid_v2_cmp

This function compares the `KSUUID` v2 values.

It returns 1 if the first value is newer, -1 if the second value is lower, or zero if they are equal.

#### Synopsis

```
bdr.ksuuid_v2_cmp(uuid, uuid)
```

#### Parameters

- `UUID` — `KSUUID` v2 to compare.

#### Notes

This function executes only on the local node.

### bdr.extract_timestamp_from_ksuuid_v2

This function extracts the timestamp component of `KSUUID` v2. The return value is of type `timestamptz` .

#### Synopsis

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

#### Parameters

- `UUID` — `KSUUID` v2 value to extract timestamp from.

#### Notes

This function executes only on the local node.

### KSUUID v1 functions

Functions for working with `KSUUID` v1 data, K-Sortable UUID data(v1). Deprecated - See *KSUUID in the sequences documentation* for details.

### bdr.gen_ksuuid

This function generates a new `KSUUID` v1 value, using the current system time. The return value is of type UUID.

#### Synopsis

```
bdr.gen_ksuuid()
```

#### Notes

This function executes only on the local node.

### bdr.uuid_v1_cmp

This function compares the `KSUUID` v1 values.

It returns 1 if the first value is newer, -1 if the second value is lower, or zero if they are equal.

#### Synopsis

```
bdr.uuid_v1_cmp(uuid, uuid)
```

#### Notes

This function executes only on the local node.

#### Parameters

- `UUID` — `KSUUID` v1 to compare.

### bdr.extract_timestamp_from_ksuuid

This function extracts the timestamp component of `KSUUID` v1 or `UUIDv1` values. The return value is of type `timestamptz` .

#### Synopsis

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

#### Parameters

- `UUID` — `KSUUID` v1 value to extract timestamp from.

#### Notes

This function executes on the local node.

## Large object functions

The `bdr.lo_*` functions are the replicated equivalents of the

PostgreSQL large object server-side functions . Each function operates on `bdr.pgd_largeobject` instead of `pg_catalog.largeobject` . Behavior and parameter semantics are identical to the PostgreSQL equivalents unless noted otherwise.

For an overview of replicated large objects, see *Replicated large objects* .

## Large object interface

### bdr.lo_creat

Creates a new large object. The `mode` argument is ignored but accepted for compatibility. Returns the OID of the new large object.

### Synopsis

```
bdr.lo_creat(mode int4) returns oid
```

### Parameters

- `mode` — Ignored. Accepted for compatibility with the PostgreSQL `lo_creat` interface.

## bdr.lo_create

Creates a new large object with the given OID. Returns the OID of the large object.

### Synopsis

```
bdr.lo_create(lobjId oid) returns oid
```

### Parameters

- `lobjId` — OID to assign to the new large object. Pass `0` to have an OID assigned automatically.

## bdr.lo_open

Opens an existing large object for reading or writing. Returns a large object file descriptor for use in subsequent operations.

### Synopsis

```
bdr.lo_open(lobjId oid, mode int4) returns int4
```

### Parameters

- `lobjId` — OID of the large object to open.
- `mode` — Access mode. Use `INV_READ` (0x40000) for read access, `INV_WRITE` (0x20000) for write access, or both combined.

## bdr.lo_close

Closes a large object file descriptor.

### Synopsis

```
bdr.lo_close(fd int4) returns int4
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .

### bdr.loread

Reads up to `len` bytes from the large object at the current seek position.

### Synopsis

```
bdr.loread(fd int4, len int4) returns bytea
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .
- `len` — Maximum number of bytes to read.

### bdr.lowrite

Writes data to the large object at the current seek position. Returns the number of bytes written.

### Synopsis

```
bdr.lowrite(fd int4, buf bytea) returns int4
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .
- `buf` — Data to write.

### bdr.lo_lseek

Moves the current seek position for the large object descriptor.

### Synopsis

```
bdr.lo_lseek(fd int4, offset int4, whence int4) returns int4
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .
- `offset` — Byte offset to seek to.
- `whence` — Seek mode: `SEEK_SET` (0) from the start, `SEEK_CUR` (1) from the current position, or `SEEK_END` (2) from the end.

### bdr.lo_lseek64

64-bit version of `bdr.lo_lseek` .

### Synopsis

```
bdr.lo_lseek64(fd int4, offset int64, whence int4) returns int64
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .
- `offset` — Byte offset to seek to.
- `whence` — Seek mode: `SEEK_SET` (0), `SEEK_CUR` (1), or `SEEK_END` (2).

## bdr.lo_tell

Returns the current seek position of the large object descriptor.

### Synopsis

```
bdr.lo_tell(fd int4) returns int4
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .

## bdr.lo_tell64

64-bit version of `bdr.lo_tell` .

### Synopsis

```
bdr.lo_tell64(fd int4) returns int64
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .

## bdr.lo_truncate

Truncates the large object to the given length in bytes.

### Synopsis

```
bdr.lo_truncate(fd int4, len int4) returns int4
```

### Parameters

- `fd` — File descriptor returned by `bdr.lo_open` .
- `len` — New length of the large object in bytes.

### bdr.lo_truncate64

64-bit version of `bdr.lo_truncate`.

#### Synopsis

```
bdr.lo_truncate64(fd int4, len int64) returns int4
```

#### Parameters

- `fd` — File descriptor returned by `bdr.lo_open`.
- `len` — New length of the large object in bytes.

### bdr.lo_unlink

Deletes the large object with the given OID.

#### Synopsis

```
bdr.lo_unlink(lobjId oid) returns int4
```

#### Parameters

- `lobjId` — OID of the large object to delete.

### bdr.lo_import

Imports a file from the server's filesystem as a new large object. Returns the OID of the new object. Requires superuser privilege.

#### Synopsis

```
bdr.lo_import(filename text) returns oid
```

#### Parameters

- `filename` — Absolute path of the file to import on the server's filesystem.

### bdr.lo_import_with_oid

Imports a file from the server's filesystem as a large object with a specific OID. Requires superuser privilege.

#### Synopsis

```
bdr.lo_import_with_oid(filename text, lobjId oid) returns oid
```

### Parameters

- `filename` — Absolute path of the file to import on the server's filesystem.
- `lobjId` — OID to assign to the new large object.

### bdr.lo_export

Exports a large object to a file on the server's filesystem. Requires superuser privilege.

### Synopsis

```
bdr.lo_export(lobjId oid, filename text) returns int4
```

### Parameters

- `lobjId` — OID of the large object to export.
- `filename` — Absolute path of the destination file on the server's filesystem.

### bdr.lo_get

Returns the entire contents of the large object as a `bytea` value.

### Synopsis

```
bdr.lo_get(lobjId oid) returns bytea
```

### Parameters

- `lobjId` — OID of the large object to read.

### bdr.lo_get_fragment

Returns a portion of the large object as a `bytea` value.

### Synopsis

```
bdr.lo_get_fragment(lobjId oid, offset int64, length int4) returns bytea
```

### Parameters

- `lobjId` — OID of the large object to read.
- `offset` — Byte offset within the large object to start reading from.
- `length` — Maximum number of bytes to return.

### bdr.lo_from_bytea

Creates a new large object from a `bytea` value. Returns the OID of the new object.

### Synopsis

```
bdr.lo_from_bytea(lobjId oid, data bytea) returns oid
```

### Parameters

- `lobjId` — OID to assign to the new large object. Pass `0` to have an OID assigned automatically.
- `data` — Initial contents of the large object.

### bdr.lo_put

Writes data into the large object starting at the given offset, extending the object if necessary.

### Synopsis

```
bdr.lo_put(lobjId oid, offset int64, data bytea) returns void
```

### Parameters

- `lobjId` — OID of the large object to write to.
- `offset` — Byte offset within the large object to start writing at.
- `data` — Data to write.

### Autopartition

Autopartition allows you to split tables into several partitions. For more information, see *AutoPartition in PGD* .

### bdr.autopartition

The `bdr.autopartition` function configures automatic RANGE partitioning of a table.

### Synopsis

```
bdr.autopartition(relation regclass,
        partition_increment text,
        partition_initial_lowerbound text DEFAULT NULL,
        partition_autocreate_expression text DEFAULT NULL,
        minimum_advance_partitions integer DEFAULT 2,
        maximum_advance_partitions integer DEFAULT 5,
        data_retention_period interval DEFAULT NULL,
        enabled boolean DEFAULT on,
        analytics_offload_period interval DEFAULT NULL,
        drop_after_retention_period boolean DEFAULT true);
```

### Parameters

- `relation` _ Name or Oid of a table.
- `partition_increment` _ Interval or increment to next partition creation.

- `partition_initial_lowerbound` _ If the table has no partition, then the first partition with this lower bound and `partition_increment` apart upper bound is created.
- `partition_autocreate_expression` _ The expression used to detect if it's time to create new partitions.
- `minimum_advance_partitions` _ The system attempts to always have at least `minimum_advance_partitions` partitions.
- `maximum_advance_partitions` _ Number of partitions to create in a single go after the number of advance partitions falls below `minimum_advance_partitions` .
- `data_retention_period` _ Interval until older partitions are dropped, if defined. This value must be greater than `migrate_after_period` .
- `enabled` _ Allows activity to be disabled or paused and later resumed or reenabled.
- `analytics_offload_period` _ Provides support for partition offloading. Reserved for future use.
- `drop_after_retention_period` _ Allows a partition to be detached instead of dropped. Set this to `false` to detach instead of drop.

### Examples

Daily partitions, keep data for one month:

```
CREATE TABLE measurement (
logdate date not null,
peaktemp int,
unitsales int
) PARTITION BY RANGE (logdate);

bdr.autopartition(measurement, 1 day, data_retention_period := 30 days);
```

Create five advance partitions when only two more partitions remain. Each partition can hold 1 billion orders.

```
bdr.autopartition(Orders, 1000000000,
        partition_initial_lowerbound := 0,
        minimum_advance_partitions := 2,
        maximum_advance_partitions := 5
    );
```

## bdr.drop_autopartition

Use `bdr.drop_autopartition()` to drop the autopartitioning rule for the given relation. All pending work items for the relation are deleted, and no new work items are created.

```
bdr.drop_autopartition(relation regclass);
```

### Parameters

- `relation` _ Name or Oid of a table.

### bdr.autopartition_wait_for_partitions

Partition creation is an asynchronous process. AutoPartition provides a set of functions to wait for the partition to be created, locally or on all nodes.

Use `bdr.autopartition_wait_for_partitions()` to wait for the creation of partitions on the local node. The function takes the partitioned table name and a partition key column value and waits until the partition that holds that value is created.

The function waits only for the partitions to be created locally. It doesn't guarantee that the partitions also exists on the remote nodes.

To wait for the partition to be created on all PGD nodes, use the `bdr.autopartition_wait_for_partitions_on_all_nodes()` function. This function internally checks local as well as all remote nodes and waits until the partition is created everywhere.

#### Synopsis

```
bdr.autopartition_wait_for_partitions(relation regclass, upperbound text);
```

#### Parameters

- `relation` _ Name or Oid of a table.
- `upperbound` _ Partition key column value.

### bdr.autopartition_wait_for_partitions_on_all_nodes

#### Synopsis

```
bdr.autopartition_wait_for_partitions_on_all_nodes(relation regclass, upperbound␣
↪text);
```

#### Parameters

- `relation` _ Name or Oid of a table.
- `upperbound` _ Partition key column value.

### bdr.autopartition_find_partition

Use the `bdr.autopartition_find_partition()` function to find the partition for the given partition key value. If partition to hold that value doesn't exist, then the function returns NULL. Otherwise Oid of the partition is returned.

#### Synopsis

```
bdr.autopartition_find_partition(relname regclass, searchkey text);
```

#### Parameters

- `relname` _ Name of the partitioned table.
- `searchkey` _ Partition key value to search.

### bdr.autopartition_enable

Use `bdr.autopartition_enable` to enable AutoPartitioning on the given table. If AutoPartitioning is already enabled, then no action occurs. See *bdr.autopartition_disable* to disable AutoPartitioning on the given table.

#### Synopsis

```
bdr.autopartition_enable(relname regclass);
```

#### Parameters

- `relname` _ Name of the relation to enable AutoPartitioning.

### bdr.autopartition_disable

Use `bdr.autopartition_disable` to disable AutoPartitioning on the given table. If AutoPartitioning is already disabled, then no action occurs.

#### Synopsis

```
bdr.autopartition_disable(relname regclass);
```

#### Parameters

- `relname` _ Name of the relation to disable AutoPartitioning.

### Internal functions

### bdr.autopartition_create_partition

AutoPartition uses an internal function `bdr.autopartition_create_partition` to create a standalone AutoPartition on the parent table.

#### Synopsis

```
bdr.autopartition_create_partition(relname regclass,
                                   partname name,
                                     lowerb text,
                                     upperb text,
                                     nodes oid[]);
```

#### Parameters

- `relname` _ Name or Oid of the parent table to attach to.
- `partname` _ Name of the new AutoPartition.
- `lowerb` _ Lower bound of the partition.
- `upperb` _ Upper bound of the partition.
- `nodes` _ List of nodes that the new partition resides on. This parameter is internal to PGD and reserved for future use.

### Notes

This is an internal function used by AutoPartition for partition management. We recommend that you don't use the function directly.

### bdr.autopartition_drop_partition

AutoPartition uses an internal function `bdr.autopartition_drop_partition` to drop a partition that's no longer required, as per the data-retention policy. If the partitioned table was successfully dropped, the function returns `true` .

### Synopsis

```
bdr.autopartition_drop_partition(relname regclass)
```

### Parameters

- `relname` _ The name of the partitioned table to drop.

### Notes

This function places a DDL lock on the parent table before using `DROP TABLE` on the chosen partition table. This function is an internal function used by AutoPartition for partition management. We recommend that you don't use the function directly.

### Stream triggers reference

!!!SeeAlso

*Stream triggers* for an introduction to Stream Triggers.

In PL/pgSQL, you can use the predefined variables and functions that follow:

- `:ref:`Stream triggers row variables <Stream triggers row variables>``
- `:ref:`Stream triggers row functions <Stream triggers row functions>``
  - `:ref:`bdr.trigger_get_row <bdr.trigger_get_row>``
  - `:ref:`bdr.trigger_get_committs <bdr.trigger_get_committs>``
  - `:ref:`bdr.trigger_get_xid <bdr.trigger_get_xid>``
  - `:ref:`bdr.trigger_get_type <bdr.trigger_get_type>``
  - `:ref:`bdr.trigger_get_conflict_type <bdr.trigger_get_conflict_type>``
  - `:ref:`bdr.trigger_get_origin_node_id <bdr.trigger_get_origin_node_id>``
  - `:ref:`bdr.ri_fkey_on_del_trigger <bdr.ri_fkey_on_del_trigger>``

Creating and dropping stream triggers is managed through the manipulation interfaces:

- :ref:`Stream triggers manipulation interfaces <Stream triggers manipulation interfaces>`
  - *bdr.create_conflict_trigger*
  - *bdr.create_transform_trigger*
  - *bdr.drop_trigger*

### Stream triggers manipulation interfaces

You can create stream triggers only on tables with `REPLICA IDENTITY FULL` or tables without any columns to which `TOAST` applies.

### bdr.create_conflict_trigger

This function creates a new conflict trigger.

### Synopsis

```
bdr.create_conflict_trigger(trigger_name text,
                            events text[],
                            relation regclass,
                            function regprocedure,
                            args text[] DEFAULT {})
```

### Parameters

- `trigger_name` _ Name of the new trigger.
- `events` _ Array of events on which to fire this trigger. Valid values are '`INSERT`', '`UPDATE`', and '`DELETE`'.
- `relation` _ Relation to fire this trigger for.
- `function` _ The function to execute.
- `args` _ Optional. Specifies the array of parameters the trigger function receives on execution (contents of `TG_ARGV` variable).

### Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *ddl filters* configuration.

The function takes a global DML lock on the relation on which the trigger is being created.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Similar to normal PostgreSQL triggers, the `bdr.create_conflict_trigger` function requires `TRIGGER` privilege on the `relation` and `EXECUTE` privilege on the function. This applies with a `bdr.backwards_compatibility` of 30619 or above. Additional security rules apply in PGD to all triggers including conflict triggers. See *Security and roles* .

### bdr.create_transform_trigger

This function creates a transform trigger.

### Synopsis

```
bdr.create_transform_trigger(trigger_name text,
                             events text[],
                             relation regclass,
                             function regprocedure,
                             args text[] DEFAULT {})
```

### Parameters

- `trigger_name` _ Name of the new trigger.
- `events` _ Array of events on which to fire this trigger. Valid values are 'INSERT', 'UPDATE', and 'DELETE'.
- `relation` _ Relation to fire this trigger for.
- `function` _ The function to execute.
- `args` _ Optional. Specify array of parameters the trigger function receives on execution (contents of `TG_ARGV` variable).

### Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *ddl filters* configuration.

The function takes a global DML lock on the relation on which the trigger is being created.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Similarly to normal PostgreSQL triggers, the `bdr.create_transform_trigger` function requires the `TRIGGER` privilege on the `relation` and `EXECUTE` privilege on the function. Additional security rules apply in PGD to all triggers including transform triggers. See *Security and roles* .

### bdr.drop_trigger

This function removes an existing stream trigger (both conflict and transform).

### Synopsis

```
bdr.drop_trigger(trigger_name text,
                 relation regclass,
                 ifexists boolean DEFAULT false)
```

### Parameters

- `trigger_name` _ Name of an existing trigger.
- `relation` _ The relation the trigger is defined for.
- `ifexists` _ When set to `true` , this function ignores missing triggers.

### Notes

This function uses the same replication mechanism as `DDL` statements. This means that the replication is affected by the *ddl filters* configuration.

The function takes a global DML lock on the relation on which the trigger is being created.

This function is transactional. You can roll back the effects with the `ROLLBACK` of the transaction. The changes are visible to the current transaction.

Only the owner of the `relation` can execute the `bdr.drop_trigger` function.

## Stream triggers row functions

### bdr.trigger_get_row

This function returns the contents of a trigger row specified by an identifier as a `RECORD` . This function returns `NULL` if called inappropriately, that is, called with `SOURCE_NEW` when the operation type (TG_OP) is `DELETE` .

### Synopsis

```
bdr.trigger_get_row(row_id text)
```

### Parameters

- `row_id` — Identifier of the row. Can be any of `SOURCE_NEW` , `SOURCE_OLD` , and `TARGET` , depending on the trigger type and operation. (See the descriptions of the individual trigger types.)

### bdr.trigger_get_committs

This function returns the commit timestamp of a trigger row specified by an identifier. If not available because a row is frozen or isn't available, returns `NULL` . Always returns `NULL` for row identifier `SOURCE_OLD` .

### Synopsis

```
bdr.trigger_get_committs(row_id text)
```

### Parameters

- `row_id` — Identifier of the row. Can be any of `SOURCE_NEW` , `SOURCE_OLD` , and `TARGET` , depending on trigger type and operation. (See the descriptions of the individual trigger types.)

### bdr.trigger_get_xid

This function returns the local transaction id of a TARGET row specified by an identifier. If not available because a row is frozen or isn't available, returns `NULL` . Always returns `NULL` for `SOURCE_OLD` and `SOURCE_NEW` row identifiers.

Available only for conflict triggers.

### Synopsis

```
bdr.trigger_get_xid(row_id text)
```

### Parameters

- `row_id` — Identifier of the row. Can be any of `SOURCE_NEW`, `SOURCE_OLD`, and `TARGET`, depending on trigger type and operation. (See the descriptions of the individual trigger types.)

### bdr.trigger_get_type

This function returns the current trigger type, which can be `CONFLICT` or `TRANSFORM`. Returns null if called outside a stream trigger.

### Synopsis

```
bdr.trigger_get_type()
```

### bdr.trigger_get_conflict_type

This function returns the current conflict type if called inside a conflict trigger. Otherwise, returns `NULL`.

See *Types of Conflict*

for possible return values of this function.

### Synopsis

```
bdr.trigger_get_conflict_type()
```

### bdr.trigger_get_origin_node_id

This function returns the node id corresponding to the origin for the trigger row_id passed in as argument. If the origin isn't valid (which means the row originated locally), returns the node id of the source or target node, depending on the trigger row argument. Always returns `NULL` for row identifier `SOURCE_OLD`. You can use this function to define conflict triggers to always favor a trusted source node.

### Synopsis

```
bdr.trigger_get_origin_node_id(row_id text)
```

### Parameters

- `row_id` — Identifier of the row. Can be any of `SOURCE_NEW`, `SOURCE_OLD`, and `TARGET`, depending on trigger type and operation. (See the descriptions of the individual trigger types.)

### bdr.ri_fkey_on_del_trigger

When called as a BEFORE trigger, this function uses FOREIGN KEY information to avoid FK anomalies.

### Synopsis

```
bdr.ri_fkey_on_del_trigger()
```

### Stream triggers row variables

#### TG_NAME

Data type name. This variable contains the name of the trigger actually fired. The actual trigger name has a `\_bdrt` or `\_bdrc` suffix (depending on trigger type) compared to the name provided during trigger creation.

#### TG_WHEN

Data type text. This variable says `BEFORE` for both conflict and transform triggers. You can get the stream trigger type by calling the `bdr.trigger_get_type()` information function. See *bdr.trigger_get_type* .

#### TG_LEVEL

Data type text: a string of `ROW` .

#### TG_OP

Data type text: a string of `INSERT` , `UPDATE` , or `DELETE` identifying the operation for which the trigger was fired.

#### TG_RELID

Data type oid: the object ID of the table that caused the trigger invocation.

#### TG_TABLE_NAME

Data type name: the name of the table that caused the trigger invocation.

#### TG_TABLE_SCHEMA

Data type name: the name of the schema of the table that caused the trigger invocation. For partitioned tables, this is the name of the root table.

#### TG_NARGS

Data type integer: the number of arguments given to the trigger function in the `bdr.create_conflict_trigger()` or `bdr.create_transform_trigger()` statement.

#### TG_ARGV[]

Data type array of text: the arguments from the `bdr.create_conflict_trigger()` or `bdr.create_transform_trigger()` statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to `TG_NARGS` ) result in a `NULL` value.

### Internal catalogs and views

Catalogs and views are listed here in alphabetical order.

#### bdr.autopartition_partitions

An internal catalog table that stores information about the partitions created by the autopartitioning feature.

#### `bdr.autopartition_partitions` columns

| Name | Type | Description |
|---|---|---|
| ap_parent_relid | oid | OID for relation |
| ap_part_relname | name | Name of created relation |
| ap_part_created_at | timestamp with time zone | Creation timestamp |
| ap_part_migrated_at | timestamp with time zone | Migration timestamp |
| ap_part_dropped_at | timestamp with time zone | Timestamp when dropped |

### bdr.autopartition_rules

An internal catalog table that stores information about the autopartitioning rules.

#### `bdr.autopartition_rules` columns

| Name | Type | Description |
|---|---|---|
| ap_partition_relid | oid | |
| ap_partition_relname | name | |
| ap_partition_schemaname | name | |
| ap_partition_increment_kind | "char" | |
| ap_secondary_tablespace | oid | |
| ap_maximum_advance_partitions | integer | |
| ap_is_autoscaled | boolean | |
| ap_latest_partitions | integer | |
| ap_enabled | boolean | |
| ap_migrate_after_period | interval | |
| ap_data_retention_period | interval | |
| ap_last_triggered | timestamp with time zone | |
| ap_partition_increment_value | text | |
| ap_partition_autocreate_expr | text | |
| ap_partition_initial_lowerbound | text | |
| ap_partition_last_upperbound | text | |
| ap_partition_min_upperbound | text | |

### bdr.ddl_epoch

An internal catalog table holding state per DDL epoch.

#### `bdr.ddl_epoch` columns

| Name | Type | Description |
|---|---|---|
| ddl_epoch | int8 | Monotonically increasing epoch number |
| origin_node_id | oid | Internal node ID of the node that requested creation of this epoch |
| epoch_consume_timeout | timestamptz | Timeout of this epoch |
| epoch_consumed | boolean | Switches to true as soon as the local node has fully processed the epoch |
| epoch_consumed_lsn | boolean | LSN at which the local node has processed the epoch |

### bdr.event_history

Internal catalog table that tracks cluster membership events for a given PGD node. Specifically, it tracks:

- Node joins (to the cluster)
- Raft state changes (that is, whenever the node changes its role in the consensus protocol - leader, follower, or candidate to leader); see *Monitoring Raft consensus*
- Whenever a worker has errored out (see bdr.workers

and *Monitoring PGD replication workers* )

#### `bdr.event_history` columns

| Name | Type | Description |
|---|---|---|
| event_node_id | oid | ID of the node to which the event refers |
| event_type | int | Type of the event (a node, raft, or worker-related event) |
| event_sub_type | int | Subtype of the event, that is, if it's a join, a state change, or an error |
| event_source | text | Name of the worker process where the event was sourced |
| event_time | timestamptz | Timestamp at which the event occurred |
| event_text | text | Textual representation of the event (for example, the error of the worker) |
| event_detail | text | A more detailed description of the event (for now, only relevant for worker errors) |

### bdr.event_summary

A view of the `bdr.event_history` catalog that displays the information in a more human-friendly format. Specifically, it displays the event types and subtypes as textual representations rather than integers.

### bdr.local_leader_change

This is a local cache of the recent portion of leader change history. It has the same fields as bdr.leader , except that it is an ordered set of (node_group_id, leader_kind, generation) instead of a map tracking merely the current version.

### bdr.node_config

An internal catalog table with per-node configuration options.

#### `bdr.node_config` columns

| Name | Type | Description |
|---|---|---|
| node_id | oid | Node ID |
| node_route_priority | int | Priority assigned to this node |
| node_route_fence | boolean | Switch to fence this node |
| node_route_writes | boolean | Switch to allow writes |
| node_route_reads | boolean | Switch to allow reads |
| node_route_dsn | text | Interface of this node |
| node_group_dsn | text | Preferred connection string for nodes in the same subgroup. When set, nodes within the same subgroup connect using this DSN instead of *node_dsn*. Defaults to empty. Set using *Replicated large objects* with key *group_dsn*. |

### bdr.node_config_summary

A view of the `bdr.node_config` catalog that displays the information in a more human-readable format.

#### `bdr.node_config_summary` columns

| Name | Type | Description |
|---|---|---|
| node_name | text | The name of this node |
| node_id | oid | Node ID |
| node_route_priority | int | Priority assigned to this node |
| node_route_fence | boolean | Switch to fence this node |
| node_route_writes | boolean | Switch to allow writes |
| node_route_reads | boolean | Switch to allow reads |
| node_route_dsn | text | Interface of this node |
| node_group_dsn | text | Preferred connection string for nodes in the same subgroup. When set, nodes within the same subgroup connect using this DSN instead of *node_dsn*. Defaults to empty. |
| effective_route_dsn | text | Full DSN of this node |
| effective_group_dsn | text | The effective group DSN for this node, derived from :ref:``node_group_dsn` <#bdrnode_config>` . |

### bdr.node_group_config

An internal catalog table with per-node group configuration options.

#### `bdr.node_group_config` columns

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | Node group ID. |

continues on next page

Table 167 – continued from previous page

| Name | Type | Description |
|---|---|---|
| route_writer_max_lag | bigint | Maximum write lag accepted. |
| route_reader_max_lag | bigint | Maximum read lag accepted. |
| route_writer_wait_flush | boolean | Switch if we need to wait for the flush. |
| leader_progress_tracking_enable | boolean | Enables periodic replication progress reports on each data nodes to the Raft leader. These reports ensure the most up-to-date node is selected during write leader elections. Defaults to *true*. |
| leader_progress_tracking_period | interval | The frequency at which replication progress reports are sent to the Raft leader. Defaults to *1s*. |

### bdr.node_group_routing_config_summary

Per-node-group routing configuration options.

#### `bdr.node_group_routing_config_summary` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | name | Node group name |
| location | name | Node group location |
| enable_routing | boolean | Group routing enabled? |
| node_group_type | text | Node group type (one of "global", "data", or "subscriber-only") |
| route_writer_max_lag | bigint | Maximum write lag accepted |
| route_reader_max_lag | bigint | Maximum read lag accepted |
| route_writer_wait_flush | boolean | Wait for flush |

### bdr.node_group_routing_info

An internal catalog table holding current routing information for connection manager.

#### `bdr.node_group_routing_info` columns

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | Node group ID. |
| write_node_id | oid | Current write node. |
| prev_write_node_id | oid | Previous write node. |
| read_node_ids | oid[] | List of read-only nodes IDs. |
| record_version | bigint | Record version. Incremented by 1 on every material change to the routing record. |
| record_ts | timestamptz | Timestamp of last update to record_version. |
| write_leader_version | bigint | Write leader version. Copied from record_version every time write_node_id is changed. |
| write_leader_ts | timestamptz | Write leader timestamp. Copied from record_ts every time write_node_id is changed. |
| read_nodes_version | bigint | Read nodes version. Copied from record_version every time read_node_ids list is changed. |

continues on next page

Table 169 – continued from previous page

| Name | Type | Description |
|---|---|---|
| read_nodes_ts | timestamptz | Read nodes timestamp. Copied from record_tw every time read_node_ids list is changed. |

### bdr.node_group_routing_summary

A view of `bdr.node_group_routing_info` catalog that shows the information in more friendly way.

#### `bdr.node_group_routing_summary` columns

| Name | Type | Description |
|---|---|---|
| node_group_name | name | Node group name |
| write_lead | name | Current write lead |
| previous_write_lead | name | Previous write lead |
| read_nodes | name[] | Current read-only nodes |

### bdr.node_peer_progress_mapping

A internal catalog used to track the relative progress of each node in the cluster against each other node to assist with group slot movement. This is an internal relation used by PGD to assist with node progress tracking. Its contents are primarily for system use and should not be used as a standalone reference to infer the exact state or health of replication progress.

#### `bdr.node_peer_progress_mapping` columns

| Name | Type | Description |
|---|---|---|
| peer_node_id | oid | OID of the remote peer node for which this progress information was reported. |
| local_lsn | pg_lsn | LSN on the local node at the time the progress was recorded. |
| remote_lsn | pg_lsn | LSN on the originating node at the time the progress was reported. |

### bdr.node_routing_config_summary

A friendly view of the per-node routing configuration options. Shows the node name rather than the oid and shorter field names.

#### `bdr.node_routing_config_summary` columns

| Name | Type | Description |
|---|---|---|
| gid | name | Node name |
| route_priority | int | Priority assigned to this node |
| route_fence | boolean | Switch to fence this node |
| route_writes | boolean | Switch to allow writes |
| route_reads | boolean | Switch to allow reads |

continues on next page

Table 172 – continued from previous page

| Name | Type | Description |
|---|---|---|
| route_dsn | text | Interface of this node |

### bdr.prepared_xacts_summary

An internal local node view that provides diagnostic information for prepared transactions implicitly created by an origin node when using group commit scopes.

#### `bdr.prepared_xacts_summary` columns

| Name | Type | Description |
|---|---|---|
| gid | name | Global identifier of the implicitly prepared transaction. |
| origin_node_id | oid | OID of the node where the transaction originated. |
| origin_xid | xid | Transaction ID of the prepared transaction on the origin node. |
| origin_name | name | Name of the origin node. |
| commit_scope_name | name | Name of the commit scope used for the transaction. |
| origin_node_group | name | Name of the node group associated with the origin node group for the commit scope. |
| cs_id | oid | Internal ID of the commit scope. |
| transaction_state | name | Current state of the prepared transaction. For example, *waiting for prepared confirmation*. |
| local_prepared_ts | timestamp with time zone | Timestamp when the transaction was prepared on the local node. |
| local_xid | xid | Transaction ID of the corresponding prepared transaction on the local node. |
| pid | int | PID of the process handling the prepared transaction, if active. |
| decision | char | The commit decision for the prepared transaction, if reached. |

### bdr.reconcile_2pc_requests

An internal state table storing the current state of 2 phase commit reconciliation requests. The view *bdr.reconcile_2pc_requests_summary* provides a human-readable representation of this table.

#### `bdr.reconcile_2pc_requests` columns

| Name | Type | Description |
|---|---|---|
| origin_id | oid | OID of the origin node for which the reconciliation request was created. |
| coord_id | oid | OID of the coordinator node acting as the reconciler. Remains empty until a coordinator is elected. |
| reconcile_status | integer | Status code of the reconciliation request. |
| origin_lsn | pg_lsn | LSN of the origin node as seen by the coordinator at the time of assignment. Remains empty until a coordinator is elected. |

continues on next page

Table 174 – continued from previous page

| Name | Type | Description |
|---|---|---|
| origin_commit_ts | timestamptz | Commit timestamp of the origin known to the coordinator at the time of assignment. Remains empty until a coordinator is elected. |
| reconcile_request_ts | timestamptz | The timestamp when the reconciliation request was registered for the origin ID. |

### bdr.reconcile_2pc_requests_summary

A view providing a human-readable summary of the reconciliation requests stored in *bdr.reconcile_2pc_requests* .

#### `bdr.reconcile_2pc_requests_summary` columns

| Name | Type | Description |
|---|---|---|
| origin | text | Name of the origin node for which the reconciliation request was created. |
| coordinator | text | Name of the coordinator node performing the reconciliation. Remains empty until a coordinator is elected. |
| origin_commit_ts | timestamptz | Commit timestamp of the origin known to the coordinator at the time of assignment. |
| origin_lsn | pg_lsn | LSN of the origin node as seen by the coordinator at the time of assignment. Remains empty until a coordinator is elected. |
| reconcile_request_ts | timestamptz | The timestamp when the reconciliation request was registered for the origin. Remains empty until a coordinator is elected. |
| reconcile_status | text | Textual status of the reconciliation request. For example: *leader elect*, *start*, *done*, *failed*. |

### bdr.sequence_kind

An internal state table storing the type of each non-local sequence. We recommend the view `bdr.sequences` for diagnostic purposes.

#### `bdr.sequence_kind` columns

| Name | Type | Description |
|---|---|---|
| seqid | oid | Internal OID of the sequence |
| seqkind | char | Internal sequence kind (`l`=local,`t`=timeshard,`s`=snowflakeid,`g`=galloc) |
| sequuid | name | Global identifier for the sequence. |

### bdr.sync_node_requests

An internal state table storing the state of node synchronization requests. The view *bdr.sync_node_requests_summary* provides a human-readable representation of this table.

#### `bdr.sync_node_requests` columns

| Name | Type | Description |
|---|---|---|
| sn_origin_node_id | oid | Unavailable node with changes to be synchronized |
| sn_target_node_id | oid | Node with the origin node's changes |
| sn_source_node_id | oid | Target node for the sync request |
| sn_sync_start_lsn | pg_lsn | Start LSN of the sync request |
| sn_sync_start_ts | timestamptz | Start timestamp of the sync request |
| sn_sync_end_lsn | pg_lsn | End LSN of the sync request |
| sn_sync_end_ts | timestamptz | End timestamp of the sync request |
| sn_sync_status | text | Status of the sync request |

### bdr.sync_node_requests_summary

A view providing a human-readable version of the underlying *bdr.sync_node_requests*

table.

#### `bdr.sync_node_requests_summary` columns

| Name | Type | Description |
|---|---|---|
| origin | text | Unavailable node with changes to be synchronized |
| source | text | Node with the origin node's changes |
| target | text | Target node for the sync request |
| sync_start_lsn | pg_lsn | Start LSN of the sync request |
| sync_start_ts | timestamptz | Start timestamp of the sync request |
| sync_end_lsn | pg_lsn | End LSN of the sync request |
| sync_end_ts | timestamptz | End timestamp of the sync request |
| sync_status | text | Status of the sync request |

## Internal system functions

The following are internal system functions. Many are used when creating various views. We recommend that you do not use the functions directly but instead use the views that they serve.

## General internal functions

### bdr.bdr_get_commit_decisions

Convenience routine to inspect shared memory state.

## Synopsis

```
bdr.bdr_get_commit_decisions(dbid OID,
            origin_node_id OID,
            origin_xid xid,
            local_xid xid,
            decision "char",
            decision_ts timestamptz,
            is_camo boolean)
```

### bdr.bdr_track_commit_decision

Save the transaction commit status in the shared memory hash table. This function is used by the upgrade scripts to transfer commit decisions saved in bdr.commit_decisions catalog to the shared memory hash table. The transaction commit status will also be logged to the WAL and hence can be reloaded from WAL.

#### Synopsis

```
bdr.bdr_track_commit_decision(OID, xid, xid, "char", timestamptz, boolean);
```

### bdr.consensus_kv_fetch

Fetch value from the consistent KV Store in JSON format.

#### Synopsis

```
bdr.consensus_kv_fetch(IN key text) RETURNS jsonb
```

#### Parameters

| Parameter | Description |
|---|---|
| *key* | An arbitrary key to fetch. |

#### Notes

This function is an internal function, mainly used by HARP.

> **Warning**
>
> Don't use this function in user applications.

### bdr.consensus_kv_store

Stores value in the consistent KV Store.

Returns the timestamp of the value expiration time. This function depends on `ttl` . If `ttl` is `NULL` , then this function returns `infinity` . If the value was deleted, it returns `-infinity` .

#### Synopsis

```
bdr.consensus_kv_store(key text, value jsonb,
        prev_value jsonb DEFAULT NULL, ttl int DEFAULT NULL)
```

#### Parameters

| Parameter | Description |
|---|---|
| *key* | An arbitrary unique key to insert, update, or delete. |
| *value* | JSON value to store. If NULL, any existing record is deleted. |
| *prev_value* | If set, the write operation is done only if the current value is equal to *prev_value*. |
| *ttl* | Time-to-live of the new value, in milliseconds. |

### Notes

This is an internal function, mainly used by HARP.

> **Warning**
>
> Don't use this function in user applications.

## bdr.decode_message_payload

PGD message payload function that decodes the payloads of consensus messages to a more human-readable output. Used primarily by the `bdr.global_consensus_journal_details` debug view.

## bdr.decode_message_response_payload

PGD message payload function that decodes the payloads of responses to consensus messages to a more human-readable output. Used primarily by the `bdr.global_consensus_journal_details` debug view.

## bdr.difference_fix_origin_create

Creates a replication origin with a given name passed as an argument but adding a `bdr_` prefix. Returns the internal id of the origin. This function has the same functionality as `pg_replication_origin_create()` except this function requires bdr_superuser rather than postgres superuser permissions.

## bdr.difference_fix_session_reset

Marks the current session as not replaying from any origin, essentially resetting the effect of `bdr.difference_fix_session_setup()` . It returns void. This function has the same functionality as `pg_replication_origin_session_reset()` except this function requires bdr_superuser rather than postgres superuser permissions.

### Synopsis

```
bdr.difference_fix_session_reset()
```

## bdr.difference_fix_session_setup

Marks the current session as replaying from the current origin. The function uses the pre-created `bdr_local_only_origin` local replication origin implicitly for the session. It allows replay progress to be reported and returns void. This function has the same functionality as `pg_replication_origin_session_setup()` except that this function requires bdr_superuser rather than postgres superuser permissions. The earlier form of the function, `bdr.difference_fix_session_setup(text)` , was deprecated and will be removed in a future release.

### Synopsis

```
bdr.difference_fix_session_setup()
```

### bdr.difference_fix_xact_set_avoid_conflict

Marks the current transaction as replaying a transaction that committed at LSN '0/0' and timestamp '2000-01-01'. This function has the same functionality as `pg_replication_origin_xact_setup('0/0', '2000-01-01')` except this function requires bdr_superuser rather than postgres superuser permissions.

### Synopsis

```
bdr.difference_fix_xact_set_avoid_conflict()
```

### bdr.drop_node

Drops a node's metadata.

After a node has been `PARTED` its metadata remains present in the cluster's nodes. For example, the node will remain in the `bdr.node_sumary` results, marked as `PARTED` , until the node is dropped.

Calling `bdr.drop_node('some node', force := true)` can be necessary and appropriate when a node becomes stuck while parting. Note that it skips past syncing any data out of the node being dropped, if there is any data on that node that still needs to be synced out. If a node stuck parting has already been reimaged or deleted, there is no harm in calling `bdr.drop_node` on it. Note that this must be called for this stuck node on all nodes in the cluster so they all have a consistent view that the node has been dropped.

This function removes the metadata for a given node from the local database. The node can be either:

- The local node, in which case it removes all the node metadata, including information about remote nodes.
- A remote node, in which case it removes only metadata for that specific node.

Use of this internal function is limited to:

- When you're instructed to by EDB Technical Support.
- Where you're specifically instructed to in the documentation.

Use bdr.part_node to remove a node from a PGD group. That function sets the node to `PARTED` state and enables reuse of the node name.

```
bdr.drop_node(node_name text, cascade boolean DEFAULT false, force boolean DEFAULT␣
↪false)
```

### Parameters

| Parameter | Description |
|---|---|
| *node_name* | Name of an existing node. |
| *cascade* | Deprecated, will be removed in a future release. |
| *force* | Circumvents all sanity checks and forces the removal of all metadata for the given PGD node despite a possible danger of causing inconsistencies. Only Technical Support uses a forced node drop in case of emergencies related to parting. |

### Notes

Before you run this function, part the node using `bdr.part_node()`.

This function removes metadata for a given node from the local database. The node can be the local node, in which case all the node metadata is removed, including information about remote nodes. Or it can be the remote node, in which case only metadata for that specific node is removed.

> **Note**
>
> PGD can have a maximum of 1024 node records (both ACTIVE and PARTED) at one time because each node has a unique sequence number assigned to it, for use by snowflakeid and timeshard sequences. PARTED nodes aren't automatically cleaned up. If this becomes a problem, you can use this function to remove those records.

### bdr.get_global_locks

Shows information about global locks held on the local node.

Used to implement the `bdr.global_locks` view to provide a more detailed overview of the locks.

### bdr.get_node_conflict_resolvers

Displays a text string of all the conflict resolvers on the local node.

### bdr.get_slot_flush_timestamp

Retrieves the timestamp of the last flush position confirmation for a given replication slot.

Used internally to implement the `bdr.node_slots` view.

### bdr.internal_alter_sequence_set_kind

A function previously used internally for replication of the various function calls. No longer used by the current version of PGD. Exists only for backward compatibility during rolling upgrades.

### bdr.internal_replication_set_add_table

A function previously used internally for replication of the various function calls. No longer used by the current version of PGD. Exists only for backward compatibility during rolling upgrades.

### bdr.internal_replication_set_remove_table

A function previously used internally for replication of the various function calls. No longer used by the current version of PGD. Exists only for backward compatibility during rolling upgrades.

### bdr.internal_submit_join_request

Submits a consensus request for joining a new node.

Needed by the PGD group reconfiguration internal mechanisms.

### bdr.isolation_test_session_is_blocked

A helper function, extending (and actually invoking) the original `pg_isolation_test_session_is_blocked` with an added check for blocks on global locks.

Used for isolation/concurrency tests.

### bdr.local_node_info

Displays information for the local node needed by the PGD group reconfiguration internal mechanisms.

The view `bdr.local_node_summary` provides similar information useful for user consumption.

### bdr.msgb_connect

Connects to the connection pooler of another node. Used by the consensus protocol.

### bdr.msgb_deliver_message

Sends messages to another node's connection pooler. Used by the consensus protocol.

### bdr.node_catchup_state_name

Converts catchup state code in name.

#### Synopsis

```
bdr.node_catchup_state_name(catchup_state oid);
```

#### Parameters

| Parameter | Description |
|---|---|
| *catchup_state* | Oid code of the catchup state. |

### bdr.node_kind_name

Returns human-friendly name of the node kind (data|standby|witness|subscriber-only).

### bdr.peer_state_name

Transforms the node state (`node_state` ) into a textual representation. Used mainly to implement the `bdr.node_summary` view.

### bdr.pg_xact_origin

Returns the origin id of a given transaction.

#### Synopsis

```
bdr.pg_xact_origin(xmin xid)
```

### Parameters

| Parameter | Description |
|---|---|
| *xid* | Transaction id whose origin is returned. |

### bdr.reconcile_2pc_status_name

Converts a numerical reconciliation status code into its corresponding textual representation. This function is primarily used to populate the *bdr.reconcile_2pc_requests_summary* view.

### Synopsis

```
bdr.reconcile_2pc_status_name(reconcile_status oid) RETURNS text
```

### Parameters

| Parameter | Description |
|---|---|
| *reconcile_status* | The OID code representing the specific reconciliation status to be converted. |

### bdr.request_replay_progress_update

Requests the immediate writing of a 'replay progress update' Raft message. Used mainly for test purposes but can also be used to test if the consensus mechanism is working.

### bdr.reset_relation_stats

Returns a Boolean result after resetting the relation stats, as viewed by `bdr.stat_relation` .

### bdr.reset_subscription_stats

Returns a Boolean result after resetting the statistics created by subscriptions, as viewed by `bdr.stat_subscription` .

### bdr.resynchronize_table_from_node

Resynchronizes the relation from a remote node.

### Synopsis

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

### Parameters

| Parameter | Description |
|---|---|
| *node_name* | The node from which to copy or resync the relation data. |

continues on next page

Table 185 – continued from previous page

| Parameter | Description |
|---|---|
| *relation* | The relation to copy from the remote node. |

### Notes

This function acquires a global DML lock on the relation, truncates the relation locally, and copies data into it from the remote node.

The relation must exist on both nodes with the same name and definition.

The following are supported:

- Resynchronizing partitioned tables with identical partition definitions
- Resynchronizing partitioned table to nonpartitioned table and vice versa
- Resynchronizing referenced tables by temporarily dropping and re-creating foreign key constraints

After running the function on a referenced table, if the referenced column data no longer matches the referencing column values, the function throws an error. After resynchronizing the referencing table data, rerun the function.

Furthermore, it supports resynchronization of tables with generated columns by computing the generated column values locally after copying the data from remote node.

Currently, row_filters are ignored by this function.

The `bdr.resynchronize_table_from_node` function can be executed only by the owner of the table, provided the owner has bdr_superuser privileges.

### bdr.seq_currval

Part of the internal implementation of global sequence manipulation.

Invoked automatically when `currval()` is called on a galloc or snowflakeid sequence.

### bdr.seq_lastval

Part of the internal implementation of global sequence manipulation.

Invoked automatically when `lastval()` is called on a galloc or snowflakeid sequence.

### bdr.seq_nextval

Part of the internal implementation of global sequence increments.

Invoked automatically when `nextval()` is called on a galloc or snowflakeid sequence

### bdr.show_subscription_status

Retrieves information about the subscription status. Used mainly to implement the `bdr.subscription_summary` view.

### bdr.show_workers

Information related to the bdr workers.

### Synopsis

```
bdr.show_workers(
    worker_pid int,
    worker_role int,
    worker_role_name text,
    worker_subid oid)
```

### bdr.show_writers

Function used in the `bdr.writers` view.

### bdr.sync_status_name

Converts sync state code into a textual representation. Used mainly to implement the *bdr.sync_node_requests_summary* view.

### Synopsis

```
bdr.sync_status_name(sync_state oid)
```

### Parameters

| Parameter | Description |
|---|---|
| *sync_state* | Oid code of the sync state. |

### Task manager functions

### bdr.taskmgr_set_leader

Requests the given `node` to be the task manager leader node. The leader node is responsible for creating new tasks. (Currently only autopartition makes use of this facility.) A witness node, a logical standby, or a subscriber-only node can't become a leader. Such requests will fail with an error.

### Synopsis

```
bdr.taskmgr_set_leader(node name, wait_for_completion boolean DEFAULT true);
```

### bdr.taskmgr_get_last_completed_workitem

Return the `id` of the last workitem successfully completed on all nodes in the cluster.

### Synopsis

```
bdr.taskmgr_get_last_completed_workitem();
```

### bdr.taskmgr_work_queue_check_status

Lets you see the status of the background workers that are doing their job to generate and finish the tasks.

The status can be seen through these views:

- `bdr.taskmgr_work_queue_local_status`
- `bdr.taskmgr_work_queue_global_status`

### Synopsis

```
bdr.taskmgr_work_queue_check_status(workid bigint
                            local boolean DEFAULT false);
```

### Parameters

| Parameter | Description |
|---|---|
| *workid* | The key of the task. |
| *local* | Check the local status only. |

### Notes

Taskmgr workers are always running in the background, even before the `bdr.autopartition` function is called for the first time. If an invalid `workid` is used, the function returns `unknown` . `In-progress` is the typical status.

### bdr.get_min_required_replication_slots

Internal function intended for use by PGD-CLI.

### bdr.get_min_required_worker_processes

Internal function intended for use by PGD-CLI.

### bdr.stat_last_reset_time

Internal function used to retrieve the most recent reset timestamp for a specific statistics category. For standard monitoring, each statistic that supports *bdr.stat_reset* includes a `stats_reset` column in the *bdr.stat_commit_scope* view, which provides this information in a more accessible format for user consumption.

### bdr.stat_get_activity

Internal function underlying view `bdr.stat_activity` . Do not use directly. Use the bdr.stat_activity view instead.

### bdr.stat_get_commit_scope

Internal function to fetch commit scope statistics. For standard monitoring and troubleshooting, use the *bdr.stat_commit_scope* view, which provides the same information in a more accessible format.

### bdr.stat_get_commit_scope_proc_wait

Internal function used to retrieve the wait state statistics for processes associated with specific commit scopes. For standard monitoring and troubleshooting, use the *bdr.stat_commit_scope* view, which provides a more accessible representation of this data.

### bdr.stat_get_reconcile_commit_scope

Internal function used to retrieve reconciliation statistics for commit scopes. For standard monitoring and troubleshooting, use the *bdr.stat_reconcile_commit_scope* view, which provides a more accessible and user-friendly representation of this data.

### bdr.worker_role_id_name

Internal helper function used when generating view `bdr.worker_tasks` . Do not use directly. Use the bdr.worker_tasks view instead.

### bdr.lag_history

Internal function used when generating view `bdr.node_replication_rates` . Do not use directly. Use the bdr.node_replication_rates view instead.

### bdr.get_raft_instance_by_nodegroup

Internal function used when generating view `bdr.group_raft_details` . Do not use directly. Use the bdr.group_raft_details view instead.

### bdr.monitor_camo_on_all_nodes

Internal function used when generating view `bdr.group_camo_details` . Do not use directly. Use the bdr.group_camo_details view instead.

### bdr.monitor_raft_details_on_all_nodes

Internal function used when generating view `bdr.group_raft_details` . Do not use directly. Use the bdr.group_raft_details view instead.

#### bdr.monitor_replslots_details_on_all_nodes

Internal function used when generating view `bdr.group_replslots_details` . Do not use directly. Use the bdr.group_replslots_details view instead.

### bdr.monitor_subscription_details_on_all_nodes

Internal function used when generating view `bdr.group_subscription_summary` . Do not use directly. Use the bdr.group_subscription_summary view instead.

### bdr.monitor_version_details_on_all_nodes

Internal function used when generating view `bdr.group_versions_details` . Do not use directly. Use the bdr.group_versions_details view instead.

#### bdr.node_group_member_info

Internal function used when generating view `bdr.group_raft_details` . Do not use directly. Use the bdr.group_raft_details view instead.

#### Column-level conflict functions

#### bdr.column_timestamps_create

This function creates column-level conflict resolution. It's called within `column_timestamp_enable` .

#### Synopsis

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestampstz)
```

#### Parameters

- `p_source` _ The two options are `current` or `commit` .
- `p_timestamp` _ Timestamp depends on the source chosen. If `commit` , then `TIMESTAMP_SOURCE_COMMIT` . If `current` , then `TIMESTAMP_SOURCE_CURRENT` .

### 10.1.2 EDB Postgres Distributed Command Line Interface (PGD CLI)

The EDB Postgres Distributed Command Line Interface (PGD CLI) is a tool for managing your EDB Postgres Distributed cluster. It's the key tool for inspecting and managing cluster resources.

It allows you to run commands against EDB Postgres Distributed clusters to:

- Determine the health of the cluster, inspect the cluster's configuration, and manage the cluster's resources.
- Inspect and manage the cluster's nodes and groups.
- Perform a write-leader change operation on the group.

You can also install it manually on Linux and macOS systems that can connect to a PGD cluster, including:

- HCP advanced and distributed high-availability clusters.
- PGD clusters deployed using the CloudNative Postgres Global Clusters operator.
- Manually deployed PGD clusters.

#### PGD Essential constraints

PGD Essential clusters have constraints that apply to certain CLI commands:

- **Cluster architecture**: A PGD Essential cluster must have at most three data nodes and must have routing enabled for the global group only. Use `pgd cluster verify --arch` to check compliance.
- **Compatibility assessment**: The `pgd assess` command skips checks for tables with multiple unique indexes, row level lock usage, lock table usage, and listen/notify usage.
- **Commit scope commands**: `pgd commit-scope create` , `pgd commit-scope update` , and `pgd commit-scope drop` are not supported.
- **Group options**: `pgd group set-option` cannot update `enable_raft` , `enable_routing` , `enable_wal_decoder` , or `streaming_mode` .
- **Node setup**: `pgd node setup` enables routing for the `global` group and disables it for subgroups.

### Installing PGD CLI

You can install PGD CLI on any system that can connect to the PGD cluster. Linux and macOS are currently supported platforms to install PGD CLI on.

### Installing PGD CLI on Linux

PGD CLI is available for most Linux distributions. You can install it from the EDB repositories, which you can access with your EDB account. PGD users and EDB Cloud Service users, including those on a free trial, have an EDB account and access to PGD CLI.

### Obtain your EDB subscription token

These repositories require a token to enable downloads from them. To obtain your token, log in to EDB Repos 2.0 . If this is your first time visiting the EDB Repos 2.0 page, you must select **Request Access** to generate your token. Once a generated token is available, select the **Copy** icon to copy it to your clipboard, or select the eye icon to view it.

### Set the EDB_SUBSCRIPTION_TOKEN environment variable

Once you have the token, execute the command shown for your operating system, substituting your token for `<your-token>` .

```
export EDB_SUBSCRIPTION_TOKEN=<your-token>
```

Then run the appropriate commands for your operating system.

### Debian or Ubuntu

On Debian or Ubuntu, you can install PGD CLI using the `apt` package manager.

```
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_
↪distributed/setup.deb.sh" | sudo -E bash
```

If this command returns an error like `curl: (22) The requested URL returned error: 404` , check that you entered the correct token.

When the command is successful, you'll see output like this:

```
Executing the  setup script for the enterprisedb/postgres_distributed repository ...
...
```

You can now install the PGD CLI package using the command:

```
sudo apt-get install edb-pgd6-cli
```

### RHEL, Rocky, AlmaLinux, or Oracle Linux

On RHEL, Rocky, AlmaLinux, or Oracle Linux, you can install PGD CLI using the `yum` package manager. You can also use the `dnf` package manager, which is the default package manager for RHEL 8 and later.

```
curl -1sSLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_
↪distributed/setup.rpm.sh" | sudo -E bash
```

If this command returns an error like `curl: (22) The requested URL returned error: 404` , check that you entered the correct token.

When the command is successful, you'll see output like this:

```
Executing the  setup script for the enterprisedb/postgres_distributed repository ...
...
```

You can now install the PGD CLI package using the command:

```
sudo dnf install edb-pgd6-cli
```

```
sudo yum install edb-pgd6-cli
```

### Installing PGD CLI on macOS

PGD CLI is available for macOS as a Homebrew formula. To install it, run the following commands:

```
brew tap enterprisedb/tap
brew install pgd-cli
```

**Note**

Standalone PGD CLI binaries for macOS are exclusively available for the Apple Silicon (ARM64) architecture. !!!

To verify the installation, run:

```
pgd --version
```

*Using PGD CLI*

### Using PGD CLI

### What is the PGD CLI?

The PGD CLI is a convenient way to connect to and manage your PGD cluster. To use it, you need a user with PGD superuser privileges or equivalent. The PGD user with superuser privileges is the *bdr_superuser role and replication* . An example of an equivalent user is edb_admin on an EDB Cloud Service distributed high-availability cluster.

- Adding an entry to your

**.pgpass , which includes the host, port,**
database name, user name, and password.

- Setting the password in the `PGPASSWORD` environment variable.
- Including the password in the connection string.

We recommend the first option, as the other options don't scale well with multiple databases, or they compromise password confidentiality.

Once you have installed pgd-cli , run the `pgd` command to access the PGD command line interface. The `pgd` command needs details about the host, port, and database to connect to, along with your username and password.

### Passing a database connection string

Use the `--dsn` flag to pass a database connection string to the `pgd` command. When you pass the connection string with the `--dsn` flag, you don't need a configuration file. The flag takes precedence even if a configuration file is present. For example:

```
pgd nodes list --dsn "host=bdr-a1 port=5432 dbname=pgddb user=enterprisedb"
```

The database conmnection string (DSN) can also be set using the `PGD_CLI_DSN` environment variable. For example:

```
export PGD_CLI_DSN="host=bdr-a1 port=5432 dbname=pgddb user=enterprisedb"
pgd nodes list
```

The `--dsn` flag takes precedence over the environment variable, so if both are set, the `--dsn` value is used.

See PGD CLI Command reference for a description of the command options.

### Specifying a configuration file

If a `pgd-cli-config.yml` file is in `/etc/edb/pgd-cli` or `$HOME/.edb/pgd-cli` , `pgd` uses it. You can override this behavior using the optional `-f` or `--config-file` flag. For example:

```
pgd nodes list -f /opt/my-config.yml
__OUTPUT__
Node Name Group Name   Node Kind Join State Node Status
- -------- ------------ --------- ---------- -----------
kaftan    dc1_subgroup data      ACTIVE     Up
kaolin    dc1_subgroup data      ACTIVE     Up
kaboom    dc1_subgroup data      ACTIVE     Up
```

### Specifying the output format

Use the `-o` or `--output` flag to change the default output format to JSON. For example:

```
pgd nodes list -o json
[
  {
    "node_name": "kaftan",
    "node_group_name": "dc1_subgroup",
    "node_kind_name": "data",
    "join_state": "ACTIVE",
    "node_status": "Up",
    "node_id": 3490219809,
    "node_seq_id": 2,
    "node_local_dbname": "pgddb"
  },
  {
    "node_name": "kaolin",
    "node_group_name": "dc1_subgroup",
    "node_kind_name": "data",
    "join_state": "ACTIVE",
    "node_status": "Up",
    "node_id": 2111777360,
    "node_seq_id": 1,
```

(continues on next page)

(continued from previous page)

```
    "node_local_dbname": "pgddb"
  },
  {
    "node_name": "kaboom",
    "node_group_name": "dc1_subgroup",
    "node_kind_name": "data",
    "join_state": "ACTIVE",
    "node_status": "Up",
    "node_id": 2710197610,
    "node_seq_id": 3,
    "node_local_dbname": "pgddb"
  }
]
```

The PGD CLI supports the following output formats.

| Setting | Format | Description |
|---|---|---|
| *simple* | Tabular | A simple tabular view. (Default). |
| *json* | JSON | Presents the raw data with no formatting. For some commands, the JSON output might show more data than the tabular output, such as extra fields and more detailed messages. |
| *psql* | PSQL | A tabular view in the style of PSQL output. format. |
| *modern* | Tabular | A tabular view which uses box characters to deliniate the table. |
| *markdown* | Markdown | A Markdown style output which may product long-form, non-tabular output for some commands such as *pgd assess*. |

### Accessing the command line help

To list the supported commands, enter:

```
pgd --help
```

For help with a specific command and its parameters, enter `pgd <command_name> --help` . For example:

```
pgd nodes list --help
```

### Configuring PGD CLI

PGD CLI can be installed on any system that can connect to the PGD cluster. To use PGD CLI, you need a user with PGD superuser privileges or equivalent. The PGD user with superuser privileges is the *bdr_superuser role and replication* . An example of an equivalent user is edb_admin on a EDB Cloud Service distributed high-availability cluster.

### PGD CLI and database connection strings

You might not need a database connection string. For example, when Trusted Postgres Architect installs the PGD CLI on a system, it also configures the connection to the PGD cluster, which means that the PGD CLI can connect to the cluster when run.

If you're installing PGD CLI manually, you must give PGD CLI a database connection string so it knows which PGD cluster to connect to.

- Adding an entry to your

**.pgpass , which includes the host, port,**
database name, user name, and password. - Setting the password in the `PGPASSWORD` environment variable. - Including the password in the connection string.

We recommend the first option, as the other options don't scale well with multiple databases, or they compromise password confidentiality.

Once you have that information, you can continue.

### Configuring the database to connect to

PGD CLI takes its database connection information from either the PGD CLI configuration file or the command line.

### Using database connection strings in the command line

You can pass the connection string directly to `pgd` using the `--dsn` option. For details, see the *Passing a database connection string* . For example:

```
pgd --dsn "host=kaboom port=5432 user=enterprisedb dbname=pgddb" nodes show --versions
```

### Using database connection strings in an environment variable

As an alternative to passing the connection string on the command line, you can set the `PGD_CLI_DSN` environment variable to the connection string. For example:

```
export PGD_CLI_DSN="host=kaboom port=5432 user=enterprisedb dbname=pgddb"
pgd nodes show --versions
```

### Using a configuration file

Use the `pgd-cli-config.yml` configuration file to specify the database connection string for your cluster. The configuration file must contain the database connection string for at least one PGD node in the cluster. The cluster name is optional and isn't validated.

For example:

```
cluster:
  name: cluster-name
  endpoints:
  - "host=host-1 port=5432 dbname=pgddb user=postgres"
  - "host=host-2 port=5432 dbname=pgddb user=postgres"
  - "host=host-3 port=5432 dbname=pgddb user=postgres"
```

By default, `pgd-cli-config.yml` is located in the `/etc/edb/pgd-cli` directory. The PGD CLI searches for `pgd-cli-config.yml` in the following locations. Precedence order is high to low.

1. `/etc/edb/pgd-cli` (default) 2. `$HOME/.edb/pgd-cli`

If your configuration file isn't in either of these directories, you can use the optional `-f` or `--config-file` flag on a `pgd` command to set the file to read as configuration. See the *Passing a database connection string* .

### Discovering connection strings

You can install PGD CLI on any system that can connect to the PGD cluster. To use PGD CLI, you need a user with PGD superuser privileges or equivalent. The PGD user with superuser privileges is the *bdr_superuser role and replication* . An example of an equivalent user is edb_admin on an EDB Cloud Service distributed high-availability cluster.

### PGD CLI and database connection strings

You might not need a database connection string. For example, when Trusted Postgres Architect installs the PGD CLI on a system, it also configures the connection to the PGD cluster. This means that PGD CLI can connect to the cluster when run.

### Getting your database connection string

Because of the range of different configurations that PGD supports, every deployment method has a different way of deriving a connection string for it. Generally, you can obtain the required information from the configuration of your deployment. You can then assemble that information into connection strings.

### For a cluster deployed with EDB CloudNative Postgres Global Cluster

If you are using EDB CloudNative Postgres Global Cluster (CNPG-GC), the connection string is derived from the configuration of the deployment. It is very flexible so there are multiple ways to obtain a connection string. It depends, in large part, on the configuration of the deployment's services :

- If you use the Node Service Template, direct connectivity to each node and proxy service is available.
- If you use the Group Service Template, there's a gateway service to each group.
- If you use the Proxy Service Template, a single proxy provides an entry point to the cluster for all applications.

** TODO [DOCS-1499] : remove proxy references when CNPG-GC is updated to use PGD6 CM **

Consult your configuration file to determine this information.

Establish a host name or IP address, port, database name, and username. The default database name is `pgddb` . The default username is enterprisedb for EDB Postgres Advanced Server and postgres for PostgreSQL and EDB Postgres Extended Server.

You can then assemble a connection string based on that information:

```
"host=<hostnameOrIPAddress> port=<portnumber> dbname=<databasename> user=<username>"
```

If the deployment's configuration requires it, add `sslmode=<sslmode>` .

### Command reference

The command name for the PGD command line interface is `pgd` .

### Synopsis

The EDB Postgres Distributed Command Line Interface (PGD CLI) is a tool to manage your EDB Postgres Distributed cluster. It allows you to run commands against EDB Postgres Distributed clusters. You can use it to inspect and manage cluster resources.

### Commands

- *Cluster configuration* : Cluster-level commands for managing the cluster.
- *pgd cluster show* : Show cluster-level information.
- *pgd cluster verify* : Verify cluster-level information.
- *Viewing groups and replication sets* : Group-level commands for managing groups.
- *pgd cluster show* : Show group-level information.
- *pgd group set-option* : Set group-level options.
- *pgd group get-option* : Get group-level options.
- *pgd group set-leader* : Set the write leader of a group (perform a switchover).
- *Viewing groups and replication sets* : Group related commands for listing groups. - *pgd commit-scopes list* : List groups.
- *Failing Over to Newly Joined Nodes* : Node-level commands for managing nodes.
- *Running pgd node setup* : Setup a node in the cluster.
- *pgd cluster show* : Show node-level information.
- *pgd node part* : Part a PGD node from an active cluster.
- *pgd group set-option* : Set node-level options.
- *pgd group get-option* : Get node-level options.
- *pgd node set-config* : Set node-level configuration.
- *pgd node get-config* : Get node-level configuration.
- *pgd node upgrade* : Perform a major version upgrade of a PGD Postgres node.
- *Node management* : Node related commands for listing nodes. - *pgd commit-scopes list* : List nodes.
- *Monitoring degrade events* : Event log commands for viewing events. - *pgd cluster show* : Show events.
- *EXPLAIN ANALYZE Replication* : Replication related-commands for managing replication. - *pgd cluster show* : Show replication information.
- *Using Raft for the commit decision* : Raft related commands for managing Raft consensus.
- *pgd cluster show* : Show information about Raft state.
- *pgd raft enable* : Enable the Raft consensus worker on one or more nodes.
- *pgd raft disable* : Disable the Raft consensus worker on one or more nodes.
- *pgd group set-leader* : Transfer Raft leadership to a specified node.
- *pgd raft sync-snapshot* : Synchronize Raft state across all nodes.
- *pgd commit-scope create* : Commit scope related commands for managing PGD commit scopes.
- *pgd cluster show* : Show information about a commit-scope.
- *pgd commit-scope create* : Create a commit-scope.
- *pgd commit-scope update* : Update a commit-scope.
- *pgd commit-scope drop* : Drop a commit-scope.
- *pgd commit-scopes list* : List commit scopes.
- *Assessing the cluster state* : Assesses a Postgres server's PGD compatibility.

- *pgd completion* : Generate shell completion scripts.

### Global Options

All commands accept the following global options:

| Short | Long | Description |
|---|---|---|
| *-f* | *--config-file* | Name/Path to config file. This is ignored if --dsn flag is present Default "/etc/edb/pgd-cli/pgd-cli-config.yml" |
| | *--dsn* | Database connection string For example "host=bdr-a1 port=5432 dbname=pgddb user=postgres" Also set by *PGD_CLI_DSN* environment variable. |
| *-h* | *--help* | Help for pgd - will show specific help for any command used |
| *-o* | *--output* | Output format: *json*, *psql*, *modern*, *markdown*, *simple* (see *Output formats* ) |

### Additional Options

Run `pgd -V` to see the version information for the pgd CLI.

### Output formats

Used with the `-o` /`--output` option:

| Format | Description |
|---|---|
| simple | Simple format - Output as a simple ASCII table (Default). |
| json | JSON format - Output as a JSON document, non-tabular |
| psql | PSQL format - Output as an ASCII table in the style of PSQL |
| modern | Modern format - Output as a table using box characters |
| markdown | Markdown table format - Output as a markdown compatible ASCII table |

## pgd assess

### Synopsis

The `pgd assess` command assesses the suitability of a Postgres server instance for migration to an EDB Postgres Distributed cluster. It's designed to run against standard Postgres instances that don't yet have the BDR extension installed.

The command must be run with a DSN that connects to the Postgres server instance you want to assess. Pass it with the `--dsn` option or set the `PGD_CLI_DSN` environment variable. Not providing either results in an error.

The command checks the Postgres instance for compatibility with EDB Postgres Distributed and produces a compatibility report. The number of checks varies by the edition detected on the target instance: standard Postgres and PGD Expanded instances get 11 checks, while PGD Essential instances get 7 (the 4 Expanded-specific checks are omitted).

### Syntax

```
pgd assess [OPTIONS]
```

### Options

The assess command has no command specific options. But it does require a DSN to connect to the Postgres server instance that you want to assess. This can be passed as the argument to the `--dsn` option, or by setting the `PGD_CLI_DSN` environment variable.

See also Global Options .

### Example

```
pgd assess
__OUTPUT__
Assessment                          | Result                       | Details
- ------------------------------------+------------------------------+------------------
 ↪----------------------------------------
 Multiple Databases                 | Compatible                   | Found only one␣
↪user database
 Materialized Views                 | Compatible                   | No materialized␣
↪views found
 EPAS Queue Tables                  | Compatible                   | No EPAS Queue␣
↪Tables found
 DDL Command Usage                  | Requires workload analysis | Cannot be checked␣
↪automatically at this time
 Advisory Lock Usage                | Potentially compatible     | No advisory lock␣
↪commands found in pg_stat_statements
 Large Objects                      | Compatible                   | No large objects␣
↪found
 Trigger/Reference Privileges       | Compatible                   | No triggers with␣
↪incompatible privileges found
 Tables with Multiple Unique Indexes | Compatible                   | No tables with␣
↪multiple unique indexes found
 LOCK TABLE Usage                   | Potentially compatible     | No LOCK TABLE␣
↪usage found in pg_stat_statements
 LISTEN/NOTIFY Usage                | Compatible                   | No LISTEN/NOTIFY␣
↪usage found in pg_stat_statements
 Row-Level Lock Usage               | Potentially compatible     | No row-level␣
↪locking commands found in pg_stat_statements
```

## pgd cluster

The `pgd cluster` commands are used to manage the EDB Postgres Distributed cluster.

### Subcommands

- *pgd cluster show* : Show cluster-level information.
- *pgd cluster verify* : Verify cluster-level information.

## pgd cluster show

### Synopsis

The `pgd cluster show` command is used to display the cluster-level information in the EDB Postgres Distributed cluster.

### Syntax

```
pgd cluster show [OPTIONS]
```

### Options

The following table lists the options available for the `pgd cluster show` command:

| Short | Long | Description |
|---|---|---|
| | --clock-drift | Only show detailed clock drift information. |
| | --summary | Only show cluster summary information. |
| | --health | Only show cluster health information. |

Only one of the above options can be specified at a time.

See also Global Options .

### Clock Drift

Please note that the current implementation of clock drift may return an inaccurate value if the cluster is under high load while running this command or has large number of nodes in it.

| Symbol | Meaning |
|---|---|
| * | ok |
| ~ | warning (drift > 2 seconds) |
| ! | critical (drift > 5 seconds) |
| x | down / unreachable |
| ? | unknown |
| - | not applicable |

### Examples

### Display the cluster information

```
pgd cluster show
__OUTPUT__

#  Summary

Group Name    Parent Group  Group Type Node Name Node Kind
- ------------ ------------- ---------- --------- ---------
democluster                 global
dc1_subgroup  democluster   data       kaboom    data
dc1_subgroup  democluster   data       kaftan    data
dc1_subgroup  democluster   data       kaolin    data

#  Health

Check             Status Details
- ---------------- ------ -----------------------------------------------
```

(continues on next page)

(continued from previous page)

```
Connections       Ok     All BDR nodes are accessible
Raft              Ok     Raft Consensus is working correctly
Replication Slots Ok     All PGD replication slots are working correctly
Clock Drift       Ok     Clock drift is within permissible limit
Versions          Ok     All nodes are running the same PGD version

#  Clock Drift

Reference Node Node Name Clock Drift
- ------------- --------- -----------
kaftan         kaboom    *
kaftan         kaolin    *
```

### pgd cluster verify

### Synopsis

The `pgd cluster verify` command is used to verify the configuration of an EDB Postgres Distributed cluster.

### Syntax

```
pgd cluster verify [OPTIONS]
```

### Options

The following table lists the options available for the `pgd cluster verify` command:

| Short | Long | Description |
|---|---|---|
| | --settings | Verify Postgres settings in the cluster |
| | --arch | Verify the cluster architecture |
| -v | --verbose | Display verbose output |

With no option set, the command verifies both `settings` and `arch` by default and the output is not verbose.

Use `--verbose` alongside `--arch` or `--settings` to include detailed information and specific advice about the cluster status.

Include the global option `-o markdown` to display the results in an inline Markdown format.

See also Global Options .

### Examples

### Verify the cluster settings and architecture

```
pgd cluster verify
__OUTPUT__

##  Architecture

Check                    Status Groups
```

(continues on next page)

(continued from previous page)

```
- ---------------------- ------ ------
Cluster has data nodes   Ok
Witness nodes per group  Ok
Witness-only groups      Ok
Data nodes per group     Ok
Empty groups             Ok
Nodes have node kind set Ok

#  Settings

Setting Name                      Status
- -------------------------------- ------
bdr.accept_connections           Ok
bdr.ddl_locking                  Ok
bdr.max_writers_per_subscription Ok
bdr.raft_group_max_connections   Ok
bdr.replay_progress_frequency    Ok
bdr.role_replication             Ok
bdr.start_workers                Ok
bdr.writers_per_subscription     Ok
bdr.xact_replication             Ok
max_connections                  Ok
max_prepared_transactions        Ok
max_replication_slots            Ok
max_wal_senders                  Ok
max_worker_processes             Ok
shared_preload_libraries         Ok
track_commit_timestamp           Ok
wal_level                        Ok
```

### Verify the cluster architecture with verbose output

```
pgd cluster verify --verbose --arch
__OUTPUT__
 Check                        | Status   | Groups | Details
- ---------------------------+---------+--------+-----------------------------------
↪-----------------
 Cluster has data nodes       | Ok       |        |
 Max data nodes in a cluster  | Ok       |        |
 Witness nodes per group      | Ok       |        |
 Witness-only groups          | Ok       |        |
 Data nodes per group         | Ok       |        |
 Routing enabled groups       | Warning  | dc-1   | Recommended to enable routing only␣
↪on global-group dc-1
 Empty groups                 | Ok       |        |
 Nodes have node kind set     | Ok       |        |
```

### Verify the cluster architecture, display output Markdown formatted

```
pgd cluster verify --arch -o markdown
__OUTPUT__
Cluster has data nodes
- ----------------------
Cluster must have at least one data node

Result: Ok

Max data nodes in a cluster
- --------------------------
A PGD Essential cluster must have at most three data nodes

Result: Ok

Witness nodes per group
- ----------------------
Each data group has at most one witness node

Result: Ok

Witness-only groups
- ------------------
Cluster has at most one witness-only group

Result: Ok

Data nodes per group
- -------------------
Each data group has at least one data node

Result: Ok

Routing enabled groups
- ---------------------
A PGD Essential cluster must have routing enabled for global group only

Following groups have problems:

 - dc-1

Details:

 - Recommended to enable routing only on global-group dc-1

Result: Warning

Empty groups
- -----------
There are no empty groups

Result: Ok
```

(continues on next page)

(continued from previous page)

```
Nodes have node kind set
- ------------------------
All nodes should have node_kind properly set

Result: Ok
```

### pgd commit-scope

The `pgd commit-scope` commands are used to display and manage the commit scopes in the EDB Postgres Distributed cluster.

### Subcommands

- *pgd cluster show* : Show information about a commit scope.
- *pgd commit-scope create* : Create a commit scope.
- *pgd commit-scope update* : Update a commit scope.
- *pgd commit-scope drop* : Drop a commit scope.
- *pgd commit-scopes list* : List commit scopes.

### pgd commit-scope create

### Synopsis

The `pgd commit-scope create` command is used to create a commit scope in the EDB Postgres Distributed cluster.

### Syntax

```
pgd commit-scope <COMMIT_SCOPE> create [OPTIONS] <RULE_DEFINITION> [GROUP_NAME]
```

Where `<COMMIT_SCOPE>` is the name of the commit scope to create.

The `<RULE_DEFINITION>` is the rule that defines the commit scope. The rule specifies the conditions that must be met for a transaction to be considered committed. See *Commit scopes* and *Commit scope rules* for more information on the rule syntax.

The optional `[GROUP_NAME]` is the name of the group to which the commit scope belongs. If omitted, it defaults to the top-level group.

### Options

No command specific options. See Global Options .

### Examples

### Creating a Commit Scope

The following example creates a commit scope named `abc1` with the rule `ANY 2 (dc1) on replicated group commit` on the `dc1_subgroup` group:

```
pgd commit-scope abc1 create "ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT" dc1_subgroup
__OUTPUT__
Command executed successfully
```

Verify the commit scope:

```
pgd commit-scope abc1 show
__OUTPUT__
Commit Scope Group Name   Rule Definition
- ----------- ------------ ----------------------------------------
abc1         dc1_subgroup ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT
```

### Creating a Commit Scope with the top-level group

The following example creates a commit scope named `abc2` with the rule `ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT` on the top-level group:

```
pgd commit-scope abc2 create "ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT"
__OUTPUT__
Command executed successfully
```

Verify the commit scope:

```
pgd commit-scope abc2 show
__OUTPUT__
Commit Scope Group Name  Rule Definition
- ----------- ----------- ----------------------------------------
abc2         democluster ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT
```

## pgd commit-scope drop

### Synopsis

The `pgd commit-scope drop` command is used to drop a commit scope from the EDB Postgres Distributed cluster.

### Syntax

```
pgd commit-scope <COMMIT_SCOPE> drop [OPTIONS] [GROUP_NAME]
```

Where `<COMMIT_SCOPE>` is the name of the commit scope to drop.

The optional `[GROUP_NAME]` is the name of the group to which the commit scope belongs. If omitted, it defaults to the top-level group. Note that the name of the group must match the group name the commit scope was created with.

### Options

No command specific options. See Global Options .

### Examples

### Drop a Commit Scope

The following example drops the commit scope named `abc2` from the top-level group:

```
pgd commit-scope abc2 drop
__OUTPUT__
Command executed successfully
```

### Drop a Commit Scope from a Group

The following example drops the commit scope named `abc1` from the `dc1_subgroup` group:

```
pgd commit-scope abc1 drop dc1_subgroup
__OUTPUT__
Command executed successfully
```

## pgd commit-scopes list

### Synopsis

The `pgd commit-scopes list` command is used to list the commit scopes information for the EDB Postgres Distributed cluster.

### Syntax

```
pgd commit-scopes list [OPTIONS] [GROUP_NAME]
```

Where `GROUP_NAME` is the name of the group for which you want to list the commit scopes. If not provided, all commit scopes in the cluster are listed.

### Options

No command specific options. See the *Global Options* for common global options.

### Examples

### List all commit-scopes

```
pgd commit-scopes list
__OUTPUT__
 Commit Scope     | Group Name | Rule Definition
- -----------------+------------+-------------------------------------------------------
↪-------------------------------------------------------------------
 local protect    | pgdx6      | ASYNCHRONOUS COMMIT
 lag protect      | pgdx6      | MAJORITY ORIGIN GROUP LAG CONTROL (max_lag_time =␣
↪30s, max_commit_delay = 10s)
 majority protect | pgdx6      | MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT
 adaptive protect | pgdx6      | MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT DEGRADE ON␣
↪(timeout = 10s, require_write_lead = true) TO ASYNCHRONOUS COMMIT
 subgroup protect | group-a    | ALL (group-a) SYNCHRONOUS COMMIT
```

### List all commit-scopes for the given group

```
pgd commit-scopes list group-a
__OUTPUT__
pgd commit-scopes list group-a
 Commit Scope     | Group Name | Rule Definition
- -----------------+------------+---------------------------------
 subgroup protect | group-a    | ALL (group-a) SYNCHRONOUS COMMIT
```

## pgd commit-scope show

### Synopsis

The `pgd commit-scope show` command is used to display information about a commit scope in the EDB Postgres Distributed cluster.

### Syntax

```
pgd commit-scope <COMMIT_SCOPE> show [OPTIONS]
```

Where `<COMMIT_SCOPE>` is the name of the commit scope for which you want to display information.

### Options

No command specific options. See Global Options .

### Example

### Showing a Commit Scope

The following example shows the information about the commit scope `abc1` :

```
pgd commit-scope abc1 show
__OUTPUT__
Commit Scope Group Name   Rule Definition
- ----------- ------------ ---------------------------------------
abc1         dc1_subgroup ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT
```

The `Group Name` column shows the name of the group to which the commit scope belongs. In this case, the commit scope belongs to the `dc1_subgroup` group.

The `Rule Definition` column shows the rule that defines the commit scope. In this case, the rule is `ANY 2 (dc1) SYCHRONOUS COMMIT` . The `dc1_subgroup` group is a replicated group, so the commit must be replicated to at least two nodes in the group and any two nodes within it must acknowledge the commit before it is considered committed.

## pgd commit-scope update

### Synopsis

The `pgd commit-scope update` command is used to update a commit scope in the EDB Postgres Distributed cluster.

### Syntax

```
pgd commit-scope <COMMIT_SCOPE> update [OPTIONS] <RULE_DEFINITION>[GROUP_NAME]
```

Where `<COMMIT_SCOPE>` is the name of the commit scope to update.

The `<RULE_DEFINITION>` is the rule that defines the commit scope. The rule specifies the conditions that must be met for a transaction to be considered committed. See *Commit scopes* and *Commit scope rules* for more information on the rule syntax.

The optional `[GROUP_NAME]` is the name of the group to which the commit scope belongs. If omitted, it defaults to the top-level group.

### Options

No command specific options. See Global Options .

### Examples

#### Updating a Commit Scope

The following example updates the commit scope `abc1` with the rule `ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT` :

```
pgd commit-scope abc1 update "ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT" dc1_subgroup
__OUTPUT__
Command executed successfully
```

#### Updating a Commit Scope in the Top-Level Group

The following example updates the commit scope `abc2` with the rule `ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT` in the top-level group:

```
pgd commit-scope abc2 update "ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT"
__OUTPUT__
Command executed successfully
```

## pgd completion

### Synopsis

The `pgd completion` commands are used to manage the completion settings for the EDB Postgres Distributed CLI.

### Syntax

```
pgd completion <SHELL>
```

Where `<SHELL>` is the shell for which to generate the autocompletion script.

Possible values for shell are `bash` , `fish` , `zsh` and `powershell` .

### Options

No command specific options. See Global Options .

### Example

```
pgd completion zsh
```

This command would normally be evaluated as part of a shell session's startup files. It generates a completion script for the Zsh shell and writes it to the standard output. Therfore you would add to your `.zshrc` file:

```
eval "$(pgd completion zsh)"
```

## pgd events

The `pgd events` commands are used to display the events in the EDB Postgres Distributed cluster.

### Subcommands

- *pgd cluster show* : Show events.

## pgd events show

### Synopsis

The `pgd events show` command is used to display the events in the EDB Postgres Distributed cluster. With no additional flags, the command displays the 20 most recent events for all nodes and groups.

### Syntax

```
pgd events show [OPTIONS]
```

### Options

The following table lists the options available for the `pgd events show` command:

| Short | Long | Description |
|---|---|---|
| | *--node <NODE_NAME>* | Only show events for the node with the specified name. |
| | *--group <GROUP_NAME>* | Only show events for the group with the specified name. |
| *-n* | *--limit <LIMIT>* | Limit the number of events to show. Defaults to 20. |

See also Global Options .

### Node States

| State | Description |
|---|---|
| NONE | Node state is unset when the worker starts, expected to be set quickly to the current known state. |
| CREATED | bdr.create_node() has been executed, but the node isn't a member of any EDB Postgres Distributed cluster yet. |
| JOIN_START | bdr.join_node_group() begins to join the local node to an existing EDB Postgres Distributed cluster. |
| JOINING | The node join has started and is currently at the initial sync phase, creating the schema and data on the node. |
| CATCHUP | Initial sync phase is complete; now the join is at the last step of retrieving and applying transactions that were performed on the upstream peer node since the join started. |
| STANDBY | Node join has finished, but not yet started to broadcast changes. All joins spend some time in this state, but if defined as a Logical Standby, the node will continue in this state. |
| PROMOTE | Node was a logical standby and we just called bdr.promote_node to move the node state to ACTIVE. These two PROMOTE states have to be coherent to the fact, that only one node can be with a state higher than STANDBY but lower than ACTIVE. |
| PROMOTING | Promotion from logical standby to full BDR node is in progress. |
| ACTIVE | The node is a full BDR node and is currently ACTIVE. This is the most common node status. |
| PART_START | Node was ACTIVE or STANDBY and we just called bdr.part_node to remove the node from the EDB Postgres Distributed cluster. |
| PARTING | Node disconnects from other nodes and plays no further part in consensus or replication. |
| PART_CATCHUP | Non-parting nodes synchronize any missing data from the recently parted node. |
| PARTED | Node parting operation is now complete on all nodes. |

Only one node at a time can be in either of the states PROMOTE or PROMOTING. STANDBY indicates that the node is in a read-only state.

### Examples

#### Display the last 5 events

```
$ pgd events show -n 5
__OUTPUT__
Event Time                      Event Observer Event Subject Event Source Event Type␣
↪Event Subtype Event Text    Event Detail
- ---------------------------- -------------- ------------- ------------ ---------- -
↪------------ ------------- ------------------------------------------------------------
↪---------------------------------------------
2025-02-21 17:44:00.444902 UTC kaolin         kaftan        consensus    ROUTING    ␣
↪STATE_CHANGE  WRITE_LEADER  dc1_subgroup
2025-02-21 17:44:00.445080 UTC kaolin         kaolin        consensus    ROUTING    ␣
↪STATE_CHANGE  RAFT_LEADER   {"raft_leader":"kaolin","group_name":"dc1_subgroup",
↪"read_nodes_version":1,"read_nodes":"kaboom,kaolin"}
2025-02-21 17:44:00.452029 UTC kaftan         kaftan        consensus    ROUTING    ␣
↪STATE_CHANGE  LEADER_UPDATE
2025-02-21 17:44:00.456483 UTC kaboom         kaboom        consensus    ROUTING    ␣
↪STATE_CHANGE  LEADER_UPDATE
2025-02-21 17:44:00.456667 UTC kaolin         kaolin        consensus    ROUTING    ␣
↪STATE_CHANGE  LEADER_UPDATE
```

## pgd group

The `pgd group` commands are used to manage the groups in the EDB Postgres Distributed cluster.

### Subcommands

- *pgd cluster show* : Show group-level information.
- *pgd group set-option* : Set group-level options.
- *pgd group get-option* : Get group-level options.
- *pgd group set-leader* : Set the write leader of a group (perform a switchover).

## pgd group show

### Synopsis

The `pgd group show` command is used to display group-level information in the EDB Postgres Distributed cluster.

### Syntax

```
pgd group <GROUP_NAME> show [OPTIONS]
```

Where `<GROUP_NAME>` is the name of the group for which you want to display information.

### Options

The following table lists the options available for the `pgd cluster show` command:

| Long | Description |
|---|---|
| --summary | Only Show summary of the group. |
| --options | Only show options of the group. |
| --nodes | Only show nodes of the group. |

Only one of the above options can be specified at a time.

See Global Options .

### Examples

### Show group information

```
pgd group group-a show
__OUTPUT__

#  Summary

 Group Property    | Value
- ------------------+---------
 Group Name        | group-a
 Parent Group Name | pgdx6
 Group Type        | data
```

(continues on next page)

(continued from previous page)

```
 Write Leader      | pgd-a3
 Commit Scope      |

#  Nodes

 Node Name | Node Kind | Join State | Node Status
- ----------+-----------+------------+-------------
 pgd-a2    | data      | ACTIVE     | Up
 pgd-a3    | data      | ACTIVE     | Up
 pgd-a1    | data      | ACTIVE     | Up

#  Options

 Option Name                        | Option Value  | Option Source
- -----------------------------------+---------------+---------------
 analytics_autoadd_tables           | false         | group
 analytics_storage_location         |               | default
 apply_delay                        | 00:00:00      | inherited
 check_constraints                  | true          | inherited
 default_commit_scope               | local protect | group
 enable_raft                        | true          | group
 enable_routing                     | true          | group
 enable_wal_decoder                 | false         | inherited
 http_port                          | 6434          | pg_config
 location                           | London        | group
 num_writers                        | 2             | pg_config
 read_only_consensus_timeout        | 00:00:00      | pg_config
 read_only_max_client_connections   | 250           | pg_config
 read_only_max_server_connections   | 250           | pg_config
 read_only_port                     | 6433          | pg_config
 read_write_consensus_timeout       | 00:00:00      | pg_config
 read_write_max_client_connections  | 250           | pg_config
 read_write_max_server_connections  | 250           | pg_config
 read_write_port                    | 6432          | pg_config
 route_reader_max_lag               | -1            | default
 route_writer_max_lag               | -1            | default
 route_writer_wait_flush            | false         | group
 streaming_mode                     | null          | group
 use_https                          | true          | group
```

### Show group summary information

```
pgd group group-a show --summary
__OUTPUT__
 Group Property    | Value
- ------------------+---------
 Group Name        | group-a
 Parent Group Name | pgdx6
 Group Type        | data
 Write Leader      | pgd-a3
 Commit Scope      |
```

### Show group nodes information

```
pgd group group-a show --nodes
__OUTPUT__
 Node Name | Node Kind | Join State | Node Status
- ----------+-----------+------------+-------------
 pgd-a1    | data      | ACTIVE     | Up
 pgd-a2    | data      | ACTIVE     | Up
 pgd-a3    | data      | ACTIVE     | Up
```

### Show group options information

```
pgd group group-a show --options
__OUTPUT__
 Option Name                        | Option Value  | Option Source
- -----------------------------------+---------------+---------------
 analytics_autoadd_tables           | false         | group
 analytics_storage_location         |               | default
 apply_delay                        | 00:00:00      | inherited
 check_constraints                  | true          | inherited
 default_commit_scope               | local protect | group
 enable_raft                        | true          | group
 enable_routing                     | true          | group
 enable_wal_decoder                 | false         | inherited
 http_port                          | 6434          | pg_config
 location                           | London        | group
 num_writers                        | 2             | pg_config
 read_only_consensus_timeout        | 00:00:00      | pg_config
 read_only_max_client_connections   | 250           | pg_config
 read_only_max_server_connections   | 250           | pg_config
 read_only_port                     | 6433          | pg_config
 read_write_consensus_timeout       | 00:00:00      | pg_config
 read_write_max_client_connections  | 250           | pg_config
 read_write_max_server_connections  | 250           | pg_config
 read_write_port                    | 6432          | pg_config
 route_reader_max_lag               | -1            | default
 route_writer_max_lag               | -1            | default
 route_writer_wait_flush            | false         | group
 streaming_mode                     | null          | group
 use_https                          | true          | group
```

### Show group information as JSON

```
pgd group group-a show -o json
__OUTPUT__
[
  {
    "Summary": [
      {
        "info": "Group Name",
        "value": "group-a"
      },
```

(continues on next page)

(continued from previous page)


```
      {
        "info": "Parent Group Name",
        "value": "pgdx6"
      },
      {
        "info": "Group Type",
        "value": "data"
      },
      {
        "info": "Write Leader",
        "value": "pgd-a3"
      },
      {
        "info": "Commit Scope",
        "value": ""
      }
    ]
  },
  {
    "Nodes": [
      {
        "join_state": "ACTIVE",
        "node_kind_name": "data",
        "node_name": "pgd-a2",
        "node_status": "Up"
      },
      {
        "join_state": "ACTIVE",
        "node_kind_name": "data",
        "node_name": "pgd-a3",
        "node_status": "Up"
      },
      {
        "join_state": "ACTIVE",
        "node_kind_name": "data",
        "node_name": "pgd-a1",
        "node_status": "Up"
      }
    ]
  },
  {
    "Options": [
      {
        "option_name": "analytics_autoadd_tables",
        "option_source": "group",
        "option_value": "false"
      },
      {
        "option_name": "analytics_storage_location",
        "option_source": "default",
        "option_value": ""
      },
```


(continues on next page)

(continued from previous page)


```
    {
      "option_name": "apply_delay",
      "option_source": "inherited",
      "option_value": "00:00:00"
    },
    {
      "option_name": "check_constraints",
      "option_source": "inherited",
      "option_value": "true"
    },
    {
      "option_name": "default_commit_scope",
      "option_source": "group",
      "option_value": "local protect"
    },
    {
      "option_name": "enable_raft",
      "option_source": "group",
      "option_value": "true"
    },
    {
      "option_name": "enable_routing",
      "option_source": "group",
      "option_value": "true"
    },
    {
      "option_name": "enable_wal_decoder",
      "option_source": "inherited",
      "option_value": "false"
    },
    {
      "option_name": "http_port",
      "option_source": "pg_config",
      "option_value": "6434"
    },
    {
      "option_name": "location",
      "option_source": "group",
      "option_value": "London"
    },
    {
      "option_name": "num_writers",
      "option_source": "pg_config",
      "option_value": "2"
    },
    {
      "option_name": "read_only_consensus_timeout",
      "option_source": "pg_config",
      "option_value": "00:00:00"
    },
    {
      "option_name": "read_only_max_client_connections",
```


(continues on next page)

(continued from previous page)


```
      "option_source": "pg_config",
      "option_value": "250"
    },
    {
      "option_name": "read_only_max_server_connections",
      "option_source": "pg_config",
      "option_value": "250"
    },
    {
      "option_name": "read_only_port",
      "option_source": "pg_config",
      "option_value": "6433"
    },
    {
      "option_name": "read_write_consensus_timeout",
      "option_source": "pg_config",
      "option_value": "00:00:00"
    },
    {
      "option_name": "read_write_max_client_connections",
      "option_source": "pg_config",
      "option_value": "250"
    },
    {
      "option_name": "read_write_max_server_connections",
      "option_source": "pg_config",
      "option_value": "250"
    },
    {
      "option_name": "read_write_port",
      "option_source": "pg_config",
      "option_value": "6432"
    },
    {
      "option_name": "route_reader_max_lag",
      "option_source": "default",
      "option_value": "-1"
    },
    {
      "option_name": "route_writer_max_lag",
      "option_source": "default",
      "option_value": "-1"
    },
    {
      "option_name": "route_writer_wait_flush",
      "option_source": "group",
      "option_value": "false"
    },
    {
      "option_name": "streaming_mode",
      "option_source": "group",
      "option_value": "null"
```


(continues on next page)

(continued from previous page)

```
      },
      {
        "option_name": "use_https",
        "option_source": "group",
        "option_value": "true"
      }
    ]
  }
]
```

### pgd group set-option

### Synopsis

The `pgd group set-option` command is used to set group-level options in the EDB Postgres Distributed cluster.

### Syntax

```
pgd group <GROUP_NAME> set-option [OPTIONS] <OPTION> <VALUE>
```

Where `<GROUP_NAME>` is the name of the group for which you want to get options.

And `<OPTION>` is the name of a specific group option you want to get and `<VALUE>` is the value you want it set to. The following options are available:

### Group options

| Option | Description |
|---|---|
| apply_delay | The delay in applying changes to the group. |
| check_constraints | Whether to check constraints in the group. |
| default_commit_scope | The default commit scope of the group. |
| enable_routing | Whether to enable routing in the group. |
| enable_raft | Whether to enable Raft in the group. |
| enable_wal_decoder | Whether to enable the WAL decoder in the group. |
| location | The location of the group. |
| num_writers | The number of writers in the group. |
| route_reader_max_lag | The maximum lag for the reader in the group. |
| route_writer_max_lag | The maximum lag for the writer in the group. |
| streaming_mode | The streaming mode of the group. |
| route_writer_wait_flush | The wait time for flushing the writer in the group. |
| default_seqkind | The default sequence kind of the group. |
| default_replica_identity | The default replica identity of the group. |
| conflict_detection_method | The conflict detection method of the group. |
| replay_progress_frequency | The replay progress frequency of the group. |
| batch_inserts | Whether to enable batch inserts in the group. |
| analytics_storage_location | The storage location for analytics in the group. |
| analytics_autoadd_tables | Whether to automatically add tables to analytics in the group. |

continues on next page

Table 197 – continued from previous page

| Option | Description |
|---|---|

### Group Connection Manager options

| Option | Description |
|---|---|
| read_write_port | which port to listen on for read-write connections |
| read_only_port | which port to listen on for read-only connections |
| http_port | which http port to listen for REST API calls (for integration purposes) |
| use_https | whether http listener should use HTTPS, if enabled, the server certificate is used to TLS |
| server_pool_mode | *Connection pooling* mode: *none* (no pooling), *session*, or *transaction* |
| read_write_max_client_connections | maximum read-write client connections allowed, defaults to max_connections |
| read_write_max_server_connections | maximum read-write connections that will be opened to server |
| read_only_max_client_connections | maximum read-only client connections allowed |
| read_only_max_server_connections | maximum read-only connections that will be opened to server |
| read_write_consensus_timeout | how long to wait on loss of consensus before read-write connections are no longer accepted |
| read_only_consensus_timeout | how long to wait on loss of consensus before read-only connections are no longer accepted. |

### Group Proxy options (for PGD 5.x only)

| Option | Description |
|---|---|
| proxy_listen_address | The listen address for the proxy in the group. |
| proxy_listen_addresses | The listen addresses for the proxy in the group. |
| proxy_listen_port | The listen port for the proxy in the group. |
| proxy_max_client_conn | The maximum number of client connections for the proxy in the group. |
| proxy_max_server_conn | The maximum number of server connections for the proxy in the group. |
| proxy_server_conn_timeout | The server connection timeout for the proxy in the group. |
| proxy_server_conn_keepali | The server connection keepalive for the proxy in the group. |
| proxy_fallback_node_groups | The fallback node groups for the proxy in the group. |
| proxy_fallback_node_group_timeout | The fallback node group timeout for the proxy in the group. |
| proxy_consensus_grace_period | The consensus grace period for the proxy in the group. |
| proxy_read_listen_address | The listen address for the read proxy in the group. |
| proxy_read_listen_addresses | The listen addresses for the read proxy in the group. |
| proxy_read_listen_port | The listen port for the read proxy in the group. |

continues on next page

Table 199 – continued from previous page

| Option | Description |
|---|---|
| proxy_read_max_client_co | The maximum number of client connections for the read proxy in the group. |
| proxy_read_max_server_co | The maximum number of server connections for the read proxy in the group. |
| proxy_read_server_conn_k | The server connection keepalive for the read proxy in the group. |
| proxy_read_server_conn_ti out | The server connection timeout for the read proxy in the group. |
| proxy_read_consen- sus_grace_period | The consensus grace period for the read proxy in the group. |

### Options

No command specific options. See Global Options .

### Examples

### Set the location of a group

```
pgd group dc1_subgroup set-option location London
__OUTPUT__
Command executed successfully
```

### Setting an option to a value with a space in it

```
pgd group dc1_subgroup set-option location "New York"
__OUTPUT__
Command executed successfully
```

### pgd group get-option

### Synopsis

The `pgd group get-option` command is used to get group-level options in the EDB Postgres Distributed cluster.

### Syntax

```
pgd group <GROUP_NAME> get-option [OPTIONS] <OPTION>
```

Where `<GROUP_NAME>` is the name of the group for which you want to get options.

And `<OPTION>` is the name of a specific group option you want to get. If you don't specify an option, the command lists all group options. The following options are available:

### Group options

| Option | Description |
|---|---|
| apply_delay | The delay in applying changes to the group. |
| check_constraints | Whether to check constraints in the group. |
| default_commit_scope | The default commit scope of the group. |

continues on next page

Table 200 – continued from previous page

| Option | Description |
|---|---|
| enable_routing | Whether to enable routing in the group. |
| enable_raft | Whether to enable Raft in the group. |
| enable_wal_decoder | Whether to enable the WAL decoder in the group. |
| location | The location of the group. |
| num_writers | The number of writers in the group. |
| route_reader_max_lag | The maximum lag for the reader in the group. |
| route_writer_max_lag | The maximum lag for the writer in the group. |
| streaming_mode | The streaming mode of the group. |
| route_writer_wait_flush | The wait time for flushing the writer in the group. |
| analytics_storage_location | The storage location for analytics in the group. |
| analytics_autoadd_tables | Whether to automatically add tables to analytics in the group. |

### Group Connection Manager options

| Option | Description |
|---|---|
| read_write_port | which port to listen on for read-write connections |
| read_only_port | which port to listen on for read-only connections |
| http_port | which http port to listen for REST API calls (for integration purposes) |
| use_https | whether http listener should use HTTPS, if enabled, the server certificate is used to TLS |
| read_write_max_client_connections | maximum read-write client connections allowed, defaults to max_connections |
| read_write_max_server_connections | maximum read-write connections that will be opened to server |
| read_only_max_client_connections | maximum read-only client connections allowed |
| read_only_max_server_connections | maximum read-only connections that will be opened to server |
| read_write_consensus_timeout | how long to wait on loss of consensus before read-write connections are no longer accepted |
| read_only_consensus_timeout | how long to wait on loss of consensus before read-only connections are no longer accepted. |

### Group Proxy options (for PGD 5.x only)

| Option | Description |
|---|---|
| proxy_listen_address | The listen address for the proxy in the group. |
| proxy_listen_addresses | The listen addresses for the proxy in the group. |
| proxy_listen_port | The listen port for the proxy in the group. |
| proxy_max_client_conn | The maximum number of client connections for the proxy in the group. |
| proxy_max_server_conn | The maximum number of server connections for the proxy in the group. |
| proxy_server_conn_timeout | The server connection timeout for the proxy in the group. |
| proxy_server_conn_keepali | The server connection keepalive for the proxy in the group. |

continues on next page

Table 202 – continued from previous page

| Option | Description |
|---|---|
| proxy_fallback_node_groups | The fallback node groups for the proxy in the group. |
| proxy_fallback_node_group_timeout | The fallback node group timeout for the proxy in the group. |
| proxy_consensus_grace_period | The consensus grace period for the proxy in the group. |
| proxy_read_listen_address | The listen address for the read proxy in the group. |
| proxy_read_listen_addresses | The listen addresses for the read proxy in the group. |
| proxy_read_listen_port | The listen port for the read proxy in the group. |
| proxy_read_max_client_co | The maximum number of client connections for the read proxy in the group. |
| proxy_read_max_server_co | The maximum number of server connections for the read proxy in the group. |
| proxy_read_server_conn_k | The server connection keepalive for the read proxy in the group. |
| proxy_read_server_conn_ti out | The server connection timeout for the read proxy in the group. |
| proxy_read_consensus_grace_period | The consensus grace period for the read proxy in the group. |

When a value is shown followed by `(inherited)` , this means the value is not specifically set on the group, but is inherited from a parent group.

### Options

No command specific options. See Global Options .

### Examples

#### List all group options

```
pgd group dc1_subgroup get-option
__OUTPUT__
 Option Name                        | Option Value
- -----------------------------------+---------------
 analytics_autoadd_tables           | false
 analytics_storage_location         |
 apply_delay                        | 00:00:00
 check_constraints                  | true
 default_commit_scope               | local protect
 enable_raft                        | true
 enable_routing                     | true
 enable_wal_decoder                 | false
 http_port                          | 6434
 location                           |
 num_writers                        | 2
 read_only_consensus_timeout        | 00:00:00
 read_only_max_client_connections   | 250
 read_only_max_server_connections   | 250
 read_only_port                     | 6433
```

(continues on next page)

(continued from previous page)

```
 read_write_consensus_timeout      | 00:00:00
 read_write_max_client_connections | 250
 read_write_max_server_connections | 250
 read_write_port                   | 6432
 route_reader_max_lag              | -1
 route_writer_max_lag              | -1
 route_writer_wait_flush           | false
 streaming_mode                    | null
 use_https                         | true
```

### Get a specific group option

```
pgd group dc1_subgroup get-option location
__OUTPUT__
 Option Name | Option Value
- ------------+--------------
 location    | London
```

### Get a specific group option as json

```
pgd group dc1_subgroup get-option location -o json
__OUTPUT__
[
  {
    "option_name": "location",
    "option_value": "London"
  }
]
```

## pgd group set-leader

### Synopsis

The `pgd group set-leader` command is used to set the write leader of a group in the EDB Postgres Distributed cluster.

This command performs a switchover operation.

### Syntax

```
pgd group <GROUP_NAME> set-leader [OPTIONS] <LEADER>
```

Where `<GROUP_NAME>` is the name of the group for which you want to set the write leader and `<LEADER>` is the name of the node that you want to set as the write leader.

### Options

The following table lists the options available for the `pgd group set-leader` command:

| Short | Long | Description |
|---|---|---|
| | *--strict* | Strict method (default). |
| | *--timeout* | Timeout period when method is strict. (Defaults to 30s (30 seconds)) |
| | *--fast* | Fast method. |

Strict method is the default method. The strict method waits for the new leader to be in sync with the old leader before switching the leader. The fast method is immediate as it does not wait for the new leader to be in sync with the old leader before switching the leader, ignoring `route_write_max_lag` .

See also Global Options .

### Examples

### Setting the write leader of a group

```
pgd group dc1_subgroup set-leader kaboom
__OUTPUT__
Command executed successfully
```

### Setting the write leader when node is already the leader

```
pgd group dc1_subgroup set-leader kaboom
__OUTPUT__
Node kaboom is already the write leader
```

## pgd groups

The `pgd groups` commands are used to display the groups in the EDB Postgres Distributed cluster.

### Subcommands

- *pgd commit-scopes list* : List groups.

## pgd groups list

### Synopsis

The `pgd groups list` command is used to display the groups in the EDB Postgres Distributed cluster.

### Syntax

```
pgd groups list [OPTIONS]
```

### Options

The following options are available for the `pgd groups list` command:

| Short | Long | Description |
|---|---|---|
| -v | --verbose | Display detailed information about the groups. |

See the *Global Options* for common global options.

### Examples

### List all groups

```
pgd groups list
__OUTPUT__
Group Name    Parent Group Name Group Type Nodes
- ------------ ----------------- ---------- -----
democluster                     global     0
dc1_subgroup  democluster       data       3
```

### List all groups with detailed information

```
pgd groups list --verbose
__OUTPUT__
Group Name    Parent Group Name Group Type Nodes Raft Leader Write Leader Commit␣
↪Scope Node Group ID
- ------------ ----------------- ---------- ----- ----------- ------------ -----------
↪- -------------
democluster                     global     0     kaftan                            ␣
↪ 150732310
dc1_subgroup  democluster       data       3     kaftan      kaboom                ␣
↪ 1302278103
```

### pgd node

The `pgd node` commands are used to manage the nodes in the EDB Postgres Distributed cluster.

### Subcommands

- *Running pgd node setup* : Configure PGD data nodes in a cluster.
- *pgd cluster show* : Show node-level information.
- *pgd node part* : Part a PGD node from an active cluster.
- *pgd group set-option* : Set node-level options.
- *pgd group get-option* : Get node-level options.
- *pgd node set-config* : Set node-level configuration.
- *pgd node get-config* : Get node-level configuration.
- *pgd node upgrade* : Perform a major version upgrade of a PGD Postgres node.

### pgd node get-config

#### Synopsis

The `pgd node get-config` command is used to get node-level configuration settings (GUC) in the EDB Postgres Distributed cluster.

#### Syntax

```
pgd node <NODE_NAME> get-config [OPTIONS] [CONFIG]
```

Where `<NODE_NAME>` is the name of the node for which you want to get options.

And `[CONFIG]` is the name of a specific GUC you want to get. If no option is specified, all GUCs are displayed.

#### Option

The following options are available for the `pgd node get-config` command:

| Short | Long | Description |
|---|---|---|
| -v | --verbose | Display additional information like min value, max value and description for the GUC(s) |

See Global Options .

#### Examples

#### Get a specific node config

```
pgd node pgd-a1 get-config bdr.batch_inserts
__OUTPUT__
 Config Name       | Config Value
- ------------------+--------------
 bdr.batch_inserts | 15
```

#### Get a specific node config with verbose output

```
pgd node pgd-a1 get-config max_wal_senders --verbose
__OUTPUT__
 Config Name     | Config Value | Min Value | Max Value | Description
- ----------------+--------------+-----------+-----------+----------------------------
↪-------------------------------------------
 max_wal_senders | 7            | 0         | 262143    | Sets the maximum number of␣
↪simultaneously running WAL sender processes.
```

#### Get a specific node config as json

```
 pgd node pgd-a1 get-config max_wal_senders -o json
__OUTPUT__
[
```

(continues on next page)

(continued from previous page)

```
  {
    "config_name": "max_wal_senders",
    "config_value": "6",
    "min_value": "0",
    "max_value": "262143",
    "description": "Sets the maximum number of simultaneously running WAL sender␣
↪processes."
  }
]
```

### pgd node get-option

### Synopsis

The `pgd node get-option` command is used to get node-level options in the EDB Postgres Distributed cluster.

### Syntax

```
pgd node <NODE_NAME> get-option [OPTIONS] [OPTION]
```

Where `<NODE_NAME>` is the name of the node for which you want to get options.

And `[OPTION]` is the name of a specific group option you want to get. If no option is specified, all options are displayed.

The following options are available:

### Node Options

| Option | Description | Type |
|---|---|---|
| route_priority | Relative routing priority of the node against other nodes in the same node group. Used only when electing a write leader. | integer |
| route_fence | Set to fence the node. When fenced, the node can't receive connections from the Connection Manager. It therefore can't become the write leader or be available in the read-only node pool. Replication is not impacted. | bool |
| route_writes | Determines whether writes can be routed to this node, that is, whether the node can become write leader. | bool |
| route_reads | Determines whether read-only connections can be routed to this node (PGD 5.5.0 and later). | bool |
| route_dsn | The dsn used by the Connection Manager to connect to this node. | string |

### Options

No command specific options. See Global Options .

### Examples

#### Get all node options

```
pgd node kaboom get-option
__OUTPUT__
Option Name    Option Value
- ------------ ------------------------------------------------
route_dsn      host=kaboom port=5444 dbname=pgddb user=postgres
route_fence    false
route_priority 100
route_reads    true
route_writes   true
```

#### Get a specific node option

```
pgd node kaboom get-option route_priority
__OUTPUT__
Option Name    Option Value
- ------------ ------------
route_priority 100
```

#### Get all node options as json

```
pgd node kaboom get-option -o json
__OUTPUT__
[
  {
    "option_name": "route_dsn",
    "option_value": "host=kaboom port=5444 dbname=pgddb user=postgres"
  },
  {
    "option_name": "route_fence",
    "option_value": "false"
  },
  {
    "option_name": "route_priority",
    "option_value": "100"
  },
  {
    "option_name": "route_reads",
    "option_value": "true"
  },
  {
    "option_name": "route_writes",
    "option_value": "true"
  }
]
```

### pgd node part

#### Synopsis

The `pgd node part` command removes a PGD node from an active cluster, initiating the parting operation.

The behavior of this command is conditional based on the cluster's Raft protocol version:

1. **Raft protocol version 6003 and above**

In this scenario, all PGD nodes in the cluster are running version {Version 6.4} or above.

The system immediately renames the node to its UUID and continues the complex parting operation in the background. This action allows you to rename the node's original name before the whole operation completes. After the initial part command returns, use the node's UUID-based name to track the progress of the parting node.

The `--no-wait` option is deprecated in this scenario.

1. **Raft protocol version below 6003**

In this scenario, not every PGD node in the cluster is running PGD version {Version 6.4} or above.

By default, the command waits for the entire part operation to complete. If you use `--no-wait` , the system schedules the part operation in the background and immediately returns an advisory message, without waiting for completion.

#### Syntax

```
pgd node <NODE_NAME> part [OPTIONS]
```

Where `<NODE_NAME>` is the name of the node to be parted.

#### Options

The following options are available for the `pgd node part` command:

| Option | Description | Note |
|---|---|---|
| --no-wait | Schedule the part operation in the background and immediately return, without waiting for the operation to complete. | Used for Raft version below 6003 |

See Global Options .

#### Examples

#### Part a node (Raft protocol version 6003 and above)

```
pgd node pgd-a1 part
__OUTPUT__
Starting a part node operation for node: pgd-a1
This may take some time, please wait...
NOTICE:  node pgd-a1 renamed to a867d475-7177-533a-a5b7-44badb2dfe4b and removal␣
↪started in the background
DETAIL:  node is parting nodegroup topgroup (51455156). node name pgd-a1 can be␣
↪reused if required.
```

### Part a node with `--no-wait` option (Raft protocol version below 6003)

```
pgd node pgd-a1 part --no-wait
__OUTPUT__
Starting a part node operation for node: pgd-a1
This may take some time, please wait...
NOTICE:  node pgd-a1 removal started in the background
DETAIL:  node is parting nodegroup topgroup (51455156)
```

### Part a node without `--no-wait` option (Raft protocol version below 6003)

```
pgd node pgd-a1 part
__OUTPUT__
Starting a part node operation for node: pgd-a1
This may take some time, please wait...
NOTICE: Node pgd-a1 has been removed from the BDR group
```

## pgd node set-config

### Synopsis

The `pgd node set-config` command is used to set node-level Postgres setting (GUC) in the EDB Postgres Distributed cluster. The command requires node `DSN` for `superuser` i.e. Postgres process owner user.

The command reloads the configuration on the node after setting the config and displays a warning of the GUC requires a server restart.

### Syntax

```
pgd node <NODE_NAME> set-config [OPTIONS] <CONFIG> <VALUE>
```

Where `<NODE_NAME>` is the name of the node for which you want to get options.

And `<CONFIG>` is the name of a specific node config you want to get and `<VALUE>` is the value you want it set to.

### Options

No command specific options. See Global Options .

### Examples

### Set a specific node config

```
pgd node pgd-a1 set-config bdr.batch_inserts 20 --dsn "host=pgd-a1 port=5432␣
↪user=postgres "
__OUTPUT__
Command executed successfully
```

### Set a specific node config with a space in the value

```
pgd node pgd-a2 set-config unix_socket_directories "/var/run/edb-pge, /tmp, /var/lib/
↪postgresql" --dsn "host=pgd-a2 port=5432 user=postgres "
__OUTPUT__
WARN: Config change requires a Postgres restart to take effect.
Command executed successfully
```

Please note the `WARN` message indicates that the configuration change will not take effect until the Postgres server is restarted.

### pgd node set-option

### Synopsis

The `pgd node set-option` command is used to set node-level options in the EDB Postgres Distributed cluster.

### Syntax

```
pgd node <NODE_NAME> set-option [OPTIONS] <OPTION> <VALUE>
```

Where `<NODE_NAME>` is the name of the node for which you want to get options.

And `<OPTION>` is the name of a specific node option you want to get and `<VALUE>` is the value you want it set to.

The following options are available:

### Node Options

| Option | Description | Type |
|---|---|---|
| route_priority | Relative routing priority of the node against other nodes in the same node group. Used only when electing a write leader. | integer |
| route_fence | Set to fence the node. When fenced, the node can't receive connections from the Connection Manager. It therefore can't become the write leader or be available in the read-only node pool. Replication is not impacted. | bool |
| route_writes | Determines whether writes can be routed to this node, that is, whether the node can become write leader. | bool |
| route_reads | Determines whether read-only connections can be routed to this node (PGD 5.5.0 and later). | bool |
| route_dsn | The dsn used by the Connection Manager to connect to this node. | string |

### Options

No command specific options. See Global Options .

### Examples

#### Set a specific node option

```
pgd node kaboom set-option route_priority 100
__OUTPUT__
Command executed successfully
```

#### Set a specific node option with a space in the value

```
pgd node kaboom set-option route_dsn "host=kaboom port=5444 dbname=pgddb user=postgres
↪"
__OUTPUT__
Command executed successfully
```

### pgd node setup

### Synopsis

The `pgd node setup` command is used to configure PGD data nodes in a cluster. It can be used to set up a new node, join an existing node to a cluster, or perform a logical join of a node to the cluster.

!!!important "Version requirement for physical joins"

When `pgd node setup` performs a physical join (copying data from the remote node when the local node isn't up and running), it requires that both the source node and the joining node have exactly the same PGD version. You can't use physical joins to join a node with a different PGD version to an existing cluster. For rolling upgrades, ensure you use logical joins instead. See *Rolling upgrade using node join* for more details.

The behavior of the command depends on the state of the local node and the remote node specified in the command.

If this is the first node in the cluster, `pgd node setup` will perform `initdb` and setup PGD node.

If this is not the first node, but the local node is not up and running, `pgd node setup` will perform a physical join of the node to the cluster. This will copy the data from the remote node to the local node as part of the initialization process, then join the local node to the cluster. This is the fastest way to load data into a new node.

If the local node is up and running and remote node also is reachable, `pgd node setup` will perform a logical join of the node to the cluster. This will create a new node in the cluster and start streaming replication from the remote node. This is the recommended way to add a new node to an existing cluster.

If the local node is up and running and remote node dsn is not provided, `pgd node setup` will do a node group switch if node not part of the given group.

### Users and roles

The `pgd node setup` command requires a superuser role to run. The superuser role is used to create the data directory and initialize the database. The superuser role must have the `CREATEDB` privilege to create the database.

The user specified in the `--dsn` option will be created if it does not exist. It will only be granted the `bdr_superuser` role which will allow it to administer PGD functionality. It will not, though have any other privileges on the database.

### Support for Transparent Data Encryption (TDE)

Transparent Data Encryption (TDE) is an optional feature available in EDB Postgres Advanced Server and EDB Postgres Extended Server versions 15 and later.

The `pgd node setup` command supports this feature when initializing a new database server or when joining an existing node to the cluster. The TDE options exposed by `pgd node setup` are similar to the initdb TDE options , with the following exceptions:

- The option `--data-encryption` is a boolean-only flag (it does not accept the AES key length value).
- Instead, use the option `--data-encryption-keylen` to specify the AES key length.

### Syntax

```
pgd node <NODE_NAME> setup [OPTIONS] -D <PG_DATA>
```

### Arguments

- `<NODE_NAME\>` The name of the node to be created. This is the name that will be used to identify the node in the cluster. It must be unique within the cluster.

### Options

| Option | Description |
|---|---|
| *--listen-addr <LISTEN_ADDR>* | The address that the configured node will listen on for incoming connections, and the address that other nodes will use to connect to this node. This is typically set to at least *localhost*, but can be set to any valid address. The default is *localhost*. The *host* value from the *--dsn* will also be appended to this list. |
| *--initial-node-count <INITIAL_NODE_COUNT>* | Number of nodes in the cluster (or planned to be in the cluster). Used to calculate various resource settings for the node. Default is 3. |
| *--bindir <BINDIR>* | <BINDIR> Specifies the directory where the binaries are located. Defaults to the directory where the running pgd binary is located. |
| *--log-file <LOG_FILE>* | Path to log file, used for postgres startup logs. Default is to write to a file in the current directory named *postgres-<port>.log* where the port value is fetched from the *port* attribute of *--dsn* option. |
| *-D, --pgdata <PG_DATA>* | Uses <PG_DATA> as the data directory of the node. (Also set with environment variable *PGDATA*). It must be a valid directory and must be writable by the user running the command. |
| *--superuser <SUPERUSER>* | Superuser name for *initdb*. Default is *postgres*. |
| *--node-kind <NODE_KIND>* | Specifies the kind of node to be created. Default is *data*. Supported values are *data*, *witness*, *subscriber-only*. |
| *--group-name <GROUP_NAME>* | Node group name. If not provided, the node will be added to the group of the active node. It is a mandatory argument for the first node of a group. |
| *--create-group* | Set this flag to create the given group, if it is not already present. This will be true by default for the first node. |
| *--cluster-name <CLUSTER_NAME>* | Name of the cluster to join the node to. When setting up cluster for the first time this will be used to create the *parent node group*. Defaults to *pgd* if not specified. |
| *--cluster-dsn <CLUSTER_DSN>* | A DSN which belongs to the active PGD cluster. This is not required when configuring the first node of a cluster, however is mandatory for subsequent nodes. Should point to the DSN of an existing active node. |

continues on next page

Table 209 – continued from previous page

| Option | Description |
|---|---|
| *--postgresql-conf <POSTGRESQL_CONF>* | Optional path of the *postgresql.conf* file to be used for the node. |
| *--postgresql-auto-conf <POSTGRESQL_AUTO_CONF>* | Optional path of the *postgresql.auto.conf* file to be used for the node. |
| *--hba-conf <HBA_CONF>* | Optional path of the *pg_hba.conf* file to be used for the node. |
| *--update-pgpass* | If set, the pgpass file for the new nodes password will be stored in the current user's *.pgpass* file. |
| *--verbose* | Print verbose messages. |
| *-y*, *--data-encryption* | Adds TDE when initializing a database server. |
| *--data-encryption-keylen <AES_KEYLEN>* | The AES key length for TDE. Default is *128*. Supported values are *128* and *256*. |
| *--key-wrap-command <KEY_WRAP_COMMAND>* | The wrapping/encryption command to protect the data encryption key. *<KEY_WRAP_COMMAND>* is customizable, but it must contain the placeholder *%p*. If you don't use this option, *pgd node setup* falls back to the environment variable *PGDATAKEYWRAPCMD*. |
| *--key-unwrap-command <KEY_UNWRAP_COMMAND>* | The unwrapping/decryption command to access the data encryption key. *<KEY_UNWRAP_COMMAND>* is customizable, but it must contain the placeholder *%p*. If you don't use this option, *pgd node setup* falls back to the environment variable *PGDATAKEYUNWRAPCMD*. |
| *--no-key-wrap* | Disable the key wrapping. This option is not recommended for production use. |
| *--copy-key-from <COPY_KEY_FROM>* | Copy an existing data encryption key from the provided location. Normally, encryption keys are stored in *pg_encryption/key.bin* |

See also Global Options .

The `pgd node setup` command attempts to find the database password non-interactively by checking the following methods of retrieval in order of precedence:

- Password set with the `password` parameter provided with the `--dsn` global option.
- Password set with the `PGPASSWORD` environment variable.
- Password file set with the `passfile` parameter provided with the `--dsn` global option.
- Password file set with the `PGPASSFILE` environment variable.
- Password file `.pgpass` in the `HOME` directory of the user invoking the command.

If the system can't find a password via any of these methods, the command prompts you to enter the password manually.

The `--update-pgpass` flag controls what happens after the system has successfully obtained a password. If set, it ensures that the password is saved for future non-interactive use. The password file to be updated follows this precedence order:

- The path specified by the `passfile` parameter within the connection string provided for the `--dsn option`.
- The file specified by the `PGPASSFILE` environment variable.
- The `.pgpass` file located in the `HOME` directory of the user invoking the command.

### Examples

In these examples, we will set up a cluster with on three hosts, `host-1` , `host-2` and `host-3` , to create three nodes: `node-1` , `node-2` , and `node-3` . The three nodes will be data nodes, and part of a cluster named `pgd` with the group name `group-1` .

We recommend that you export the PGPASSWORD environment variable to avoid having to enter the password for the `pgdadmin` user each time you run a command. You can do this with the following command:

```
export PGPASSWORD=pgdsecret
```

### Configuring the first node

```
pgd node node-1 setup --dsn "host=host-1 port=5432 user=pgdadmin dbname=pgddb" \
- -listen-addr "localhost,host-1" \
- -group-name group-1 --cluster-name pgd \
- D /var/lib/edb-pge/17/main
```

Stepping through the command, we are setting up `node-1` . The first option is the `--dsn` option, which is the connection string for the node. This is typically set to `host=hostname port=5432 user=pgdadmin dbname=pgd` , which is a typical connection string for a local Postgres instance.

The `--listen-address` option is used to specify the address that the node will listen on for incoming connections. In this case, we are setting it to `localhost,host-1` , which means that the node will listen on both the localhost and the `host-1` address.

This is the first node in the cluster, so we set the group name to `group-1` and the cluster name to `pgd` (which is actually the default). As this is the first node in the cluster, the `--create-group` option is automatically set.

Finally, we set the data directory for the node with the `-D` option; this is where the Postgres data files will be stored. In this example, we are using `/var/lib/edb-pge/17/main` as the data directory.

The command will create the data directory and initialize the database correctly for PGD. It will then start the node and make it available for new connections, including the other nodes joining the cluster.

### Configuring a second node

```
pgd node node-2 setup --dsn "host=host-2 port=5432 user=pgdadmin dbname=pgddb" \
- -listen-addr "localhost,host-2" \
- D /var/lib/edb-pge/17/main
- -cluster-dsn "host=host-1 port=5432 user=pgdadmin dbname=pgddb"
```

This command is similar to the first node, but we are setting up `node-2` . The `--dsn` option is the connection string for the node, which is typically set to `host=hostname port=5432 user=pgdadmin dbname=pgd` . The `cluster-dsn` must point to an active node, it can point to connection manager, or proxy endpoint etc., CLI will get the real DSN of the node behind it. In this case, we are setting it to `host=host-1 port=5432 user=pgdadmin dbname=pgd` , which is the connection string for the first node in the cluster.

### Configuring a third node

```
pgd node node-3 setup --dsn "host=host-3 port=5432 user=pgdadmin dbname=pgddb" \
- -listen-addr "localhost,host-3" \
- -cluster-dsn "host=host-1 port=5432 user=pgdadmin dbname=pgddb" \
- D /var/lib/edb-pge/17/main
```

This command is similar to the second node, but we are setting up `node-3`. The `--dsn` option is the connection string for the node, which is typically set to `host=hostname port=5432 user=pgdadmin dbname=pgd`. The `cluster-dsn` must point to an active node, it can point to connection manager, or proxy endpoint etc., CLI will get the real DSN of the node behind it. In this case, we are setting it to `host=host-1 port=5432 user=pgdadmin dbname=pgd`, which is the connection string for the first node in the cluster.

### Joining a parted and dropped node to the cluster

```
pgd node node-2 setup --dsn "host=host-2 port=5432 user=pgdadmin dbname=pgddb" \
- -listen-addr "localhost,host-2" \
- -cluster-dsn "host=host-1 port=5432 user=pgdadmin dbname=pgddb" \
- D /var/lib/edb-pge/17/main
```

This command is similar to the setting up the subsequent nodes, but we are setting up `node-2` again. The `--dsn` option is the connection string for the node, which is typically set to `host=hostname port=5432 user=pgdadmin dbname=pgd`. The `cluster-dsn` must point to an active node, it can point to connection manager, or proxy endpoint etc., CLI will get the real DSN of the node behind it. In this case, we are setting it to `host=host-1 port=5432 user=pgdadmin dbname=pgd`, which is the connection string for the first node in the cluster.

This is useful when a node has been `parted` and `dropped` from the cluster for some activity like maintenance and needs to be rejoined to the cluster. The command will perform a logical join of the node to the cluster, which will create a new node in the cluster and start streaming replication from the remote node.

### Configuring the first node with TDE options

```
pgd node node-1 setup \
- -dsn "host=host-1 port=5432 user=pgdadmin dbname=pgddb" \
- -listen-addr "localhost,host-1" \
- -group-name group-1 --cluster-name pgd \
- D /var/lib/edb-pge/17/main \
- -data-encryption \
- -key-wrap-command "openssl enc -e -aes-128-cbc -pbkdf2 -pass pass:secret -out %p" \
- -key-unwrap-command "openssl enc -d -aes-128-cbc -pbkdf2 -pass pass:secret -in %p"
```

The `pgd node setup` command for a TDE-enabled cluster is similar to the standard command, but requires three additional TDE-specific options: `--data-encryption`, `--key-wrap-command`, and `--key-unwrap-command`.

For simplicity, the values for `--key-wrap-command` and `--key-unwrap-command` in this example use a simple passphrase. However, you can configure more secure mechanisms, such as an external Key Management Service (KMS).

The example doesn't explicitly set the `--data-encryption-keylen` option, defaulting the key length to 128 bits.

### Configuring a second node with TDE options

```
pgd node node-2 setup \
- -dsn "host=host-2 port=5432 user=pgdadmin dbname=pgddb" \
- -listen-addr "localhost,host-2" \
- D /var/lib/edb-pge/17/main \
- -cluster-dsn "host=host-1 port=5432 user=pgdadmin dbname=pgddb" \
- -key-wrap-command "openssl enc -e -aes-128-cbc -pbkdf2 -pass pass:secret -out %p" \
- -key-unwrap-command "openssl enc -d -aes-128-cbc -pbkdf2 -pass pass:secret -in %p"
```

This command is similar to the setup used for the first node, retaining the required TDE options. The remaining

arguments are consistent with the standard command used when setting up subsequent nodes in a cluster. The values of the TDE options must be identical to those provided for the initial node.

### pgd node show

#### Synopsis

The `pgd node show` command is used to display node-level information in the EDB Postgres Distributed cluster.

#### Syntax

```
pgd node <NODE_NAME> show [OPTIONS]
```

Where `<NODE_NAME>` is the name of the node for which you want to display information.

#### Options

No command specific options. See Global Options .

#### Examples

#### Show node information

```
pgd node kaboom show
__OUTPUT__

#  Summary

Node Property   Value
- -------------- ------------
Node Name       kaboom
Group Name      dc1_subgroup
Node Kind       data
Join State      ACTIVE
Node Status     Up
Node ID         2710197610
Snowflake SeqID 2
Database        pgddb

#  Options

Option Name    Option Value
- ------------- -------------------------------------------------
route_dsn      host=kaboom port=5444 dbname=pgddb user=postgres
route_fence    false
route_priority 100
route_reads    true
route_writes   true
```

### Show node information as JSON

```
pgd node kaboom show -o json
__OUTPUT__
[
  {
    "Summary": [
      {
        "info": "Node Name",
        "value": "kaboom"
      },
      {
        "info": "Group Name",
        "value": "dc1_subgroup"
      },
      {
        "info": "Node Kind",
        "value": "data"
      },
      {
        "info": "Join State",
        "value": "ACTIVE"
      },
      {
        "info": "Node Status",
        "value": "Up"
      },
      {
        "info": "Node ID",
        "value": "2710197610"
      },
      {
        "info": "Snowflake SeqID",
        "value": "2"
      },
      {
        "info": "Database",
        "value": "pgddb"
      }
    ]
  },
  {
    "Options": [
      {
        "option_name": "route_dsn",
        "option_value": "host=kaboom port=5444 dbname=pgddb user=postgres "
      },
      {
        "option_name": "route_fence",
        "option_value": "false"
      },
      {
        "option_name": "route_priority",
```

(continues on next page)

(continued from previous page)

```
        "option_value": "100"
      },
      {
        "option_name": "route_reads",
        "option_value": "true"
      },
      {
        "option_name": "route_writes",
        "option_value": "true"
      }
    ]
  }
]
```

### pgd node upgrade

#### Synopsis

The `pgd node upgrade` command is used to upgrade the Postgres version on a node in the EDB Postgres Distributed cluster.

!!!note Starting from PGD 6.4, `pgd node setup` advances the `NextOID` counter before creating BDR objects in single-user mode, preventing those objects from receiving system-range OIDs (< 16384) that would cause upgrade failures. Before performing the upgrade, the command checks whether existing BDR objects were created with system-range OIDs. If detected, the command fails immediately with a clear error identifying the affected objects. See *Resolving system-range OID errors* for steps to recover.

#### Syntax

```
pgd node <NODE_NAME> upgrade [OPTIONS] --old-bindir <OLD_BINDIR> --new-bindir <NEW_
↪BINDIR> --old-datadir <OLD_DATADIR> --new-datadir <NEW_DATADIR> --database
↪<DATABASE> --username <USER_NAME>
```

Where `<NODE_NAME>` is the name of the node which you want to upgrade and `<OLD_BINDIR>` , `<NEW_BINDIR>` , `<OLD_DATADIR>` , `<NEW_DATADIR>` , `<DATABASE>` , and `<USER_NAME>` are the old and new Postgres instance bin directories, old and new Postgres instance data directories, database name, and cluster's install user name respectively.

#### Options

The following table lists the options available for the `pgd node upgrade` command:

| Short | Long | Default | Env | Description |
|---|---|---|---|---|
| -b | --old-bindir | | PGBINOLD | Old Postgres instance bin directory |
| -B | --new-bindir | | PGBINNEW | New Postgres instance bin directory |
| -d | --old-datadir | | PGDATAOLD | Old Postgres instance data directory |
| -D | --new-datadir | | PGDATANEW | New Postgres instance data directory |
| | --database | | PGDATABASE | PGD database name |

continues on next page

Table 210 – continued from previous page

| Short | Long | Default | Env | Description |
|---|---|---|---|---|
| -p | --old-port | 5432 | PGPORTOLD | Old Postgres instance port |
| | --socketdir | /var/run/post-gresql | PGSOCKET-DIR | Directory to use for postmaster sockets during upgrade |
| | --new-socketdir | /var/run/post-gresql | PGSOCKET-DIRNEW | Directory to use for postmaster sockets in the new cluster |
| | --check | | | Specify to only perform checks and not modify clusters |
| -j | --jobs | 1 | | Number of simultaneous processes or threads to use |
| -k | --link | | | Use hard links instead of copying files to the new cluster |
| | --old-options | | | Option to pass to old postgres command, multiple invocations are appended |
| | --new-options | | | Option to pass to new postgres command, multiple invocations are appended |
| -N | --no-sync | | | Don't wait for all files in the upgraded cluster to be written to disk |
| -P | --new-port | 5432 | PGPORTNEW | New Postgres instance port number |
| -r | --retain | | | Retain SQL and log files even after successful completion |
| -U | --username | | PGUSER | Cluster's install user name |
| | --clone | | | Use efficient file cloning |
| | --copy-by-block | | | Used to migrate data between clusters with different encryption settings. This option is supported for databases that use Transparent Data Encryption (TDE) |
| | --key-unwrap-comman | | PG-DATAKEYUN-WRAPCMD | Command to unwrap (decrypt) the data encryption key and access the files to copy. The command must be the same specified during the server initialization using *pgd node setup* |

See also Global Options .

### Resolving system-range OID errors

If the command fails with a system-range OID error, the cluster was set up with a version of `pgd node setup` earlier than 6.4 that didn't advance the `NextOID` counter before creating BDR objects in single-user mode. The affected objects can't be reassigned new OIDs in place.

To resolve the error, part the node, drop and recreate the BDR extension in a standard psql session (not single-user mode), then rejoin the cluster without synchronizing structure to preserve existing data.

1. Part the node from the cluster:

```
pgd node <node-name> part
```

2. In a psql session connected to the BDR database, drop the extension:

```
DROP EXTENSION bdr CASCADE;
```

3. In a standard psql session (not single-user mode), recreate the extension:

```
CREATE EXTENSION bdr;
```

4. Use *bdr.create_node()* and *bdr.join_node_group()* to recreate the node and rejoin the group, with `synchronize_structure` set to `'none'` .
5. Run `pgd node upgrade` again.

### Examples

In the following examples, "kaolin" is the name of the node to upgrade, from the Quickstart democluster.

#### Upgrade the Postgres version on a node

```
pgd node kaolin upgrade --old-bindir /usr/pgsql-16/bin --new-bindir /usr/pgsql-17/bin␣
↪--old-datadir /var/lib/pgsql/16/data --new-datadir /var/lib/pgsql/17/data --
↪database pgddb --username enterprisedb
```

#### Upgrade the Postgres version on a node with hard links

```
pgd node kaolin upgrade --old-bindir /usr/pgsql-16/bin --new-bindir /usr/pgsql-17/bin␣
↪--old-datadir /var/lib/pgsql/16/data --new-datadir /var/lib/pgsql/17/data --
↪database pgddb --username enterprisedb --link
```

#### Upgrade the Postgres version on a node with efficient file cloning

```
pgd node kaolin upgrade --old-bindir /usr/pgsql-16/bin --new-bindir /usr/pgsql-17/bin␣
↪--old-datadir /var/lib/pgsql/16/data --new-datadir /var/lib/pgsql/17/data --
↪database pgddb --username enterprisedb --clone
```

#### Upgrade the Postgres version on a node with a different port number

```
pgd node kaolin upgrade --old-bindir /usr/pgsql-16/bin --new-bindir /usr/pgsql-17/bin␣
↪--old-datadir /var/lib/pgsql/16/data --new-datadir /var/lib/pgsql/17/data --
↪database pgddb --username enterprisedb --old-port 5433 --new-port 5434
```

#### Upgrade the Postgres Extended version on a node with Transparent Data Encryption (TDE)

```
pgd node kaolin upgrade --database pgddb -B /usr/lib/edb-pge/16/bin --socketdir /var/
↪run/edb-pge/ --old-bindir /usr/lib/edb-pge/15/bin  --old-datadir /var/lib/edb-pge/
↪15/main --new-datadir /var/lib/edb-pge/16/main --username postgres --key-unwrap-
↪command "openssl enc -d -aes-128-cbc -pbkdf2 -pass pass:secret -in %p" --copy-by-
↪block
```

### pgd nodes

The `pgd nodes` commands are used to display the nodes in the EDB Postgres Distributed cluster.

### Subcommands

- *pgd commit-scopes list* : List nodes.

### pgd nodes list

### Synopsis

The `pgd nodes list` command is used to display the nodes in the EDB Postgres Distributed cluster. By default, this shows the node name, group name, node kind, join state of the node and whether it is up or down.

### Syntax

```
pgd nodes list [OPTIONS]
```

### Options

The following options are available for the `pgd nodes list` command:

| Short | Long | Description |
|---|---|---|
| | --versions | Display only version information about the nodes. For each node, the BDR version and Postgres version are shown. |
| -v | --verbose | Display detailed information about the nodes. For each node, this option addes the node id, Snowflake sequence id and database name. |
| | --group | Filter by group name. |

See Global Options for common global options.

### Examples

### List all nodes

```
pgd nodes list
__OUTPUT__
Node Name Group Name   Node Kind Join State Node Status
- -------- ------------ --------- ---------- -----------
kaftan    dc1_subgroup data      ACTIVE     Up
kaboom    dc1_subgroup data      ACTIVE     Up
kaolin    dc1_subgroup data      ACTIVE     Up
kapok     dc2_subgroup data      ACTIVE     Up
kaput     dc2_subgroup data      ACTIVE     Up
karakul   dc2_subgroup data      ACTIVE     Up
```

### List all nodes in a specific group

```
pgd nodes list --group dc1_subgroup
__OUTPUT__
Node Name Group Name   Node Kind Join State Node Status
- -------- ------------ --------- ---------- -----------
kaftan    dc1_subgroup data      ACTIVE     Up
kaboom    dc1_subgroup data      ACTIVE     Up
kaolin    dc1_subgroup data      ACTIVE     Up
```

### List all nodes with detailed information

```
pgd nodes list --verbose
__OUTPUT__
Node Name Group Name   Node Kind Join State Node Status Node ID    Snowflake SeqID␣
↪Database
- -------- ------------ --------- ---------- ----------- ---------- --------------- --
↪------
kaftan    dc1_subgroup data      ACTIVE     Up          3490219809 1               ␣
↪pgddb
kaboom    dc1_subgroup data      ACTIVE     Up          2710197610 2               ␣
↪pgddb
kaolin    dc1_subgroup data      ACTIVE     Up          2111777360 3               ␣
↪pgddb
kapok     dc2_subgroup data      ACTIVE     Up          1599108790 4               ␣
↪pgddb
kaput     dc2_subgroup data      ACTIVE     Up          1321108721 5               ␣
↪pgddb
karakul   dc2_subgroup data      ACTIVE     Up          1012686451 6               ␣
↪pgddb
```

### List all nodes version information

```
pgd nodes list --versions
__OUTPUT__
Node Name BDR Version Postgres Version
- -------- ----------- ----------------------------------
kaboom    5.7.0       15.12.0 (Debian 15.12.0-1.bullseye)
kaftan    5.7.0       15.12.0 (Debian 15.12.0-1.bullseye)
kaolin    5.7.0       15.12.0 (Debian 15.12.0-1.bullseye)
kapok     5.7.0       15.12.0 (Debian 15.12.0-1.bullseye)
kaput     5.7.0       15.12.0 (Debian 15.12.0-1.bullseye)
karakul   5.7.0       15.12.0 (Debian 15.12.0-1.bullseye)
```

## pgd raft

The `pgd raft` commands are used to monitor and manage the Raft consensus state within an EDB Postgres Distributed cluster.

### Subcommands

- *pgd raft disable* : Disables Raft consensus on the specified PGD node or group.
- *pgd raft enable* : Enables Raft consensus on the specified PGD node or group.
- *pgd group set-leader* : Manually triggers a leadership transfer within a Raft group.
- *pgd cluster show* : Show Raft status for the cluster.
- *pgd raft sync-snapshot* : Synchronize the Raft state across all nodes by exporting from the leader and importing on all nodes.

## pgd raft disable

### Synopsis

The `pgd raft disable` command stops the Raft consensus worker on one or more nodes in the EDB Postgres Distributed cluster. When the worker is disabled, the node stops participating in Raft elections, ceases to send or receive consensus log entries, and will not act as a write leader.

Enabling the worker allows the node to participate in Raft elections, log replication, and cluster-wide state changes.

> **Note**
>
> The consensus worker remains in the disabled state until it is explicitly enabled using *pgd raft enable* or until the Postgres node is restarted, whichever occurs first.

### Syntax

```
pgd raft disable [OPTIONS]
```

### Options

The following options are available for the `pgd raft disable` command:

| Short | Long | Description |
|---|---|---|
| | --node | Target a specific node by name to disable its Raft worker. |
| | --group | Target all nodes in a specific group. |
| | --all | Target all nodes in the cluster. |

See Global Options for common global options.

### Examples

#### Disable Raft on a specific node

```
pgd raft disable --node kaftan
__OUTPUT__
Raft disabled successfully on node kaftan
```

#### Disable Raft for all nodes in a group

```
pgd raft disable --group dc1_subgroup
__OUTPUT__
Raft disabled successfully on 3 nodes in group dc1_subgroup
```

#### Disable Raft for the entire cluster

```
pgd raft disable --all
__OUTPUT__
Raft disabled successfully on 6 nodes in the cluster
```

### pgd raft enable

#### Synopsis

The `pgd raft enable` command enables the Raft consensus worker on one or more nodes in the EDB Postgres Distributed cluster.

Enabling the worker allows the node to participate in Raft elections, log replication, and cluster-wide state changes.

#### Syntax

```
pgd raft enable [OPTIONS]
```

#### Options

The following options are available for the `pgd raft enable` command:

| Short | Long | Description |
|---|---|---|
| | --node | Target a specific node by name to enable its Raft worker. |
| | --group | Target all nodes in a specific group. |
| | --all | Target all nodes in the cluster. |

See Global Options for common global options.

#### Examples

#### Enable Raft on a specific node

```
pgd raft enable --node kaftan
__OUTPUT__
Raft enabled successfully on node kaftan
```

### Enable Raft for all nodes in a group

```
pgd raft enable --group dc1_subgroup
__OUTPUT__
Raft enabled successfully on 3 nodes in group dc1_subgroup
```

### Enable Raft for the entire cluster

```
pgd raft enable --all
__OUTPUT__
Raft enabled successfully on 6 nodes in the cluster
```

### pgd raft set-leader

### Synopsis

The `pgd raft set-leader` command manually triggers a leadership transfer within a Raft group. It instructs the current leader to step down and facilitates the election of the specified target node as the new Raft leader.

### Syntax

```
pgd raft set-leader [OPTIONS] <LEADER>
```

### Arguments

| Argument | Description |
|---|---|
| *<LEADER>* | The name of the node to become the new Raft leader. |

### Options

The following options are available for the `pgd raft set-leader` command:

| Short | Long | Description |
|---|---|---|
| | --group | The name of the group in which to change the leader. If omitted, the global group is used. |
| | --wait-for-completion | If set, the CLI will block and monitor the cluster state until the target node successfully assumes leadership or the operation times out. |

See Global Options for common global options.

### Examples

#### Change the leader of the global group

```
pgd raft set-leader kaboom
__OUTPUT__
Command submitted successfully
```

#### Attempt to set a node that is already the leader

```
pgd raft set-leader kaftan --group dc1_subgroup
__OUTPUT__
Node kaftan is already the raft leader
```

#### Change the leader and wait for completion

Using the `--wait-for-completion` flag sets a 60-second statement timeout. The CLI will attempt to verify the new leader using the existing connection, or retry with a new connection if a proxy (like Connection Manager) reroutes the traffic.

```
pgd raft set-leader kaolin --group dc1_subgroup --wait-for-completion
__OUTPUT__
Command executed successfully
```

### pgd raft show

### Synopsis

The `pgd raft show` command displays Raft consensus information for the EDB Postgres Distributed cluster. By default, it displays three sections:

- **State**: consensus state for all nodes, sourced from `bdr.stat_raft_state`.
- **Followers**: follower state as seen from the leader node, sourced from `bdr.stat_raft_followers_state`.
- **Journal**: global consensus journal details, sourced from `bdr.global_consensus_journal_details`.

Run the command without any of the section flags to display all three, or use `--state`, `--followers`, or `--journal` to display only that section. These flags are mutually exclusive.

### Syntax

```
pgd raft show [OPTIONS]
```

### Options

The following options are available for the `pgd raft show` command:

#### Section flags

| Short | Long | Description |
|---|---|---|
| | --state | Show only the State section (consensus state from *bdr.stat_raft_state*). |
| | --followers | Show only the Followers section (follower state from *bdr.stat_raft_followers_state*). |
| | --journal | Show only the Journal section (global journal details from *bdr.global_consensus_journal_details*). |

These flags are mutually exclusive.

### Section options

| Short | Long | Applies to | Default | Description |
|---|---|---|---|---|
| | --group | All states | | Filter output by group name. |
| -n | --limit | --journal | 20 | Limit the number of journal entries shown. |

The tabular output displays a subset of available fields. Use `--output json` or `--output markdown` for the full field set.

See Global Options for common global options.

### Examples

### Show Raft state for all nodes

```
pgd raft show --state
__OUTPUT__
Group Name    Node Name State         Leader Name Term Commit Index Apply Index Last␣
↪Log Index Is Voting Nodes Voting Nodes Protocol Version
- ------------ --------- ------------- ----------- ---- ------------ ----------- -----
↪--------- --------- ----- ------------ -----------------
dc1_subgroup  kaftan    RAFT_LEADER   kaftan      1    4            4           4     ␣
↪         true      3     3            0
dc1_subgroup  kaboom    RAFT_FOLLOWER kaftan      1    4            4           4     ␣
↪         true      3     3            0
dc1_subgroup  kaolin    RAFT_FOLLOWER kaftan      1    4            4           4     ␣
↪         true      3     3            0
democluster   kaftan    RAFT_LEADER   kaftan      0    335          335         335   ␣
↪         true      3     3            5007
democluster   kaboom    RAFT_FOLLOWER kaftan      0    335          335         335   ␣
↪         true      3     3            5007
democluster   kaolin    RAFT_FOLLOWER kaftan      0    335          335         335   ␣
↪         true      3     3            5007
```

In this output, `dc1_subgroup` is a data group with local routing, and `democluster` is the top-level group with global routing.

The `Term` column shows the current Raft term number. The `Commit Index` shows the index of the last committed log entry. `Apply Index` and `Last Log Index` show how far the node has applied and received log entries respectively. `Is Voting` indicates whether the node participates in Raft elections. `Nodes` shows the total number of nodes in the group. `Voting Nodes` shows the number of nodes that participate in Raft consensus. `Protocol Version` shows the version of the Raft protocol in use.

### Show follower state for a specific group

```
pgd raft show --followers --group dc1_subgroup
__OUTPUT__
Group Name    Node Name Sent Commit Index Match Index Clock Drift (ms)
- ------------ --------- ---------------- ----------- ----------------
dc1_subgroup  kaboom    4                4           -2
dc1_subgroup  kaolin    4                3           5
```

`Sent Commit Index` is the commit index the leader last sent to each follower. `Match Index` is the highest log index confirmed replicated on that follower. `Clock Drift (ms)` is the approximate clock difference between the leader and follower in milliseconds.

### Show journal entries with a custom limit

```
pgd raft show --journal --limit 5
__OUTPUT__
Group Name    Origin Log Index Term Message Type  Response Type
- ------------ ------ --------- ---- ------------ --------------------------
democluster   kaftan 335       0    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
democluster   kaftan 334       0    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
democluster   kaftan 333       0    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
democluster   kaftan 332       0    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
democluster   kaftan 331       0    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
```

### Show all sections for a specific group

```
pgd raft show --group dc1_subgroup
__OUTPUT__

#  State

Group Name    Node Name State         Leader Name Term Commit Index Apply Index Last␣
↪Log Index Is Voting Nodes Voting Nodes Protocol Version
- ------------ --------- ------------- ----------- ---- ------------ ----------- -----
↪--------- --------- ----- ------------ ----------------
dc1_subgroup  kaftan    RAFT_LEADER   kaftan      1    4            4           4     ␣
↪         true      3     3            0
dc1_subgroup  kaboom    RAFT_FOLLOWER kaftan      1    4            4           4     ␣
↪         true      3     3            0
dc1_subgroup  kaolin    RAFT_FOLLOWER kaftan      1    4            4           4     ␣
↪         true      3     3            0

#  Followers

Group Name    Node Name Sent Commit Index Match Index Clock Drift (ms)
- ------------ --------- ---------------- ----------- ----------------
dc1_subgroup  kaboom    4                4           -2
dc1_subgroup  kaolin    4                3           5

#  Journal
```

(continues on next page)

(continued from previous page)

```
Group Name    Origin Log Index Term Message Type  Response Type
- ------------ ------ --------- ---- ------------ --------------------------
dc1_subgroup  kaftan 4         1    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
dc1_subgroup  kaftan 3         1    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
dc1_subgroup  kaftan 2         1    CONFIG_CHANGE CONFIG_CHANGE_RESPONSE
```

### pgd raft sync-snapshot

#### Synopsis

The `pgd raft sync-snapshot` command synchronizes the Raft state across all nodes in the EDB Postgres Distributed cluster when that state has diverged.

The command runs pre-flight checks before making any changes: it validates the BDR version, identifies the Raft leader, verifies all nodes are reachable, and checks whether the state is already in sync. If the state is already consistent, the command exits without making changes unless `--ignore-sync-status` is specified.

When changes are needed, the command

1. Disables Raft on all nodes.
2. Exports the current Raft state from the leader node.
3. Imports the leader's Raft state on all follower nodes.
4. Re-enables Raft on all nodes and waits for a leader to be elected.

Subscriber-only groups don't have their own Raft instance and aren't valid targets for this command. To sync Raft state that includes subscriber-only nodes, run the command without `--group` .

#### Syntax

```
pgd raft sync-snapshot [OPTIONS]
```

#### Options

The following options are available for the `pgd raft sync-snapshot` command:

| Option | Default | Description |
|---|---|---|
| --group | | Target a specific group by name. Defaults to the top-level group if not specified. |
| --check | false | Perform pre-flight checks only without making any changes to the cluster. |
| --ignore-sync-status | false | Proceed with sync even if Raft state is already consistent across all nodes. |

See Global Options for common global options.

### Examples

#### Run pre-flight checks only

```
pgd raft sync-snapshot --check
__OUTPUT__
Performing Raft Sync-Snapshot Pre-flight Checks
- ------------------------------------------------
Checking BDR version                                          ok (6.4.0)
Resolving target group                                        ok (dc-1)
Identifying raft leader                                       ok (leader: pgd-a1)
Discovering cluster nodes                                     ok (3 nodes)
Connecting to pgd-a1 (leader)                                 ok
Connecting to pgd-a2                                          ok
Connecting to pgd-a3                                          ok
Validating node states                                        ok
Checking raft sync status                                     ok (out of sync)

Raft Sync-Snapshot Pre-flight Report
- ------------------------------------
Target group:    dc-1
Raft leader:     pgd-a1
Nodes in scope:  3 (pgd-a1, pgd-a2, pgd-a3)

All pre-flight checks passed. Ready for sync-snapshot.
```

#### Synchronize the global group

```
pgd raft sync-snapshot
__OUTPUT__
Performing Raft Sync-Snapshot Pre-flight Checks
- ------------------------------------------------
Checking BDR version                                          ok (6.4.0)
Resolving target group                                        ok (dc-1)
Identifying raft leader                                       ok (leader: pgd-a1)
Discovering cluster nodes                                     ok (3 nodes)
Connecting to pgd-a1 (leader)                                 ok
Connecting to pgd-a2                                          ok
Connecting to pgd-a3                                          ok
Validating node states                                        ok
Checking raft sync status                                     ok (out of sync)

Performing Raft Sync-Snapshot
- ---------------------------
Disabling raft on pgd-a1                                      WARNING:  disabling the␣
↪consensus
ok
Disabling raft on pgd-a2                                      ok (already disabled)
Disabling raft on pgd-a3                                      WARNING:  disabling the␣
↪consensus
ok
Exporting snapshot from leader (pgd-a1)                       ok
```

(continues on next page)

(continued from previous page)

```
Importing snapshot on pgd-a2                                    ok
Importing snapshot on pgd-a3                                    ok
Re-enabling raft on pgd-a1                                      ok
Re-enabling raft on pgd-a2                                      ok
Re-enabling raft on pgd-a3                                      ok
Waiting for raft recovery                                       ok (leader: pgd-a3)

Raft sync-snapshot completed successfully.
```

### pgd replication

The `pgd replication` commands are used to display the various aspects of replication status in the EDB Postgres Distributed cluster.

### Subcommands

- *pgd cluster show* : Show replication status for the cluster.

### pgd replication show

### Synopsis

The `pgd replication show` command is used to display the replication status in the EDB Postgres Distributed cluster.

By default, with no options, it produces reports on the following:

- Node Replication Progress: A matrix of the replication status between nodes.
- Replication Slots: The replication slots status for each node's slots.
- Subscriptions: The subscription status for each subscription between nodes.
- Analytics Replication: The analytics replication status for each node.

Options can be used to restrict the output to any one of the above reports. The `--verbose` option can be used to increase the detail in the default report to show the LSN and the replication lag for each node's connection to other nodes.

### Syntax

```
pgd replication show [OPTIONS]
```

### Options

The following options are available for the `pgd replication show` command:

| Short | Long | Description |
|---|---|---|
| | --nodes | Display only node to node replication status in a matrix format. |
| | --slots | Display the replication slots for each node. |

continues on next page

Table 219 – continued from previous page

| Short | Long | Description |
|---|---|---|
| | --subscriptions | Display the subscription status for each subscription between nodes. |
| | --analytics | Display the analytics replication status for each node. |
| -v | --verbose | Display detailed information about the replication status. |

See the *Global Options* for common global options.

### –slots

This shows Shows the status of BDR replication slots. Output with the verbose flag gives details such as is slot active, replication state (disconnected, streaming, catchup), and approximate lag.

In matrix view, sometimes byte lag is shown in parentheses. It is maxOf(WriteLag, FlushLag, ReplayLag, SentLag).

## Examples

### Display the replication status in the EDB Postgres Distributed cluster

```
pgd replication show
__OUTPUT__

#  Node Replication Progress

Node   kaboom kaftan kaolin
- ----- ------ ------ ------
kaboom -      *      *
kaftan *      -      *
kaolin *      *      -

#  Replication Slots

Group Name   Origin Node Target Node Slot Name                      Active State
↪Write Lag Replay Lag Sent Lag Bytes Write Lag Bytes Replay Lag Bytes
- ----------- ----------- ----------- ------------------------------ ------ ---------
↪--------- ---------- -------------- --------------- -----------------
dc1_subgroup kaboom      kaftan      bdr_pgddb_democluster15_kaftan t      streaming
↪00:00:00  00:00:00   0              0               0
dc1_subgroup kaboom      kaolin      bdr_pgddb_democluster15_kaolin t      streaming
↪00:00:00  00:00:00   0              0               0
dc1_subgroup kaftan      kaboom      bdr_pgddb_democluster15_kaboom t      streaming
↪00:00:00  00:00:00   0              0               0
dc1_subgroup kaftan      kaolin      bdr_pgddb_democluster15_kaolin t      streaming
↪00:00:00  00:00:00   0              0               0
dc1_subgroup kaolin      kaboom      bdr_pgddb_democluster15_kaboom t      streaming
↪00:00:00  00:00:00   0              0               0
dc1_subgroup kaolin      kaftan      bdr_pgddb_democluster15_kaftan t      streaming
↪00:00:00  00:00:00   0              0               0

#  Subscriptions
```

(continues on next page)

(continued from previous page)

```
Origin Node Target Node Last Applied Tx Timestamp      Last Applied Tx Age␣
↪Subscription Status
- ---------- ----------- ------------------------------ ------------------ ----------
↪---------
kaboom      kaftan      2025-02-21 19:18:12.661520 UTC 00:00:18.616        replicating
kaboom      kaolin      2025-02-21 19:18:12.661520 UTC 00:00:18.939        replicating
kaftan      kaboom      2025-02-21 19:18:12.658069 UTC 00:00:18.787        replicating
kaftan      kaolin      2025-02-21 19:18:12.658069 UTC 00:00:18.943        replicating
kaolin      kaboom      2025-02-21 19:18:12.663201 UTC 00:00:18.782        replicating
kaolin      kaftan      2025-02-21 19:18:12.663201 UTC 00:00:18.614        replicating

#  Analytics Replication Progress

Origin Node Replicating Node Replicated LSN Last Updated
- ---------- ---------------- -------------- ------------
```

### Display only the node to node replication status in a matrix format

```
pgd replication show --nodes
__OUTPUT__
Node   kaboom kaftan kaolin
- ----- ------ ------ ------
kaboom -      *      *
kaftan *      -      *
kaolin *      *      -
```