EDB Postgres Distributed PGD

Release 6.2

EnterpriseDB Corporation

Dec 09, 2025

Contents

1 Introduction **3**
- 1.1 Why PGD? . . . 3
- 1.2 What does PGD enable? . . . 3
- 1.3 What are the differences between PGD Essential and PGD Expanded? . . . 3

2 Getting started **5**
- 2.1 Get started with PGD . . . 5
  - 2.1.1 What is EDB Postgres Distributed? . . . 5
  - 2.1.2 What is EDB Postgres Distributed Essential? . . . 5
  - 2.1.3 What is the PGD Essential Standard architecture . . . 5
  - 2.1.4 Create your first PGD Essential cluster with Docker Compose . . . 6
    - 2.1.4.1 An introduction to PGD Essential . . . 6
      - 2.1.4.1.1 Standard/One-location architecture . . . 6
    - 2.1.4.2 Creating your first cluster (PGD Essential) . . . 7
      - 2.1.4.2.1 Prerequisites . . . 7
      - 2.1.4.2.2 Install the PGD Docker Quickstart kit . . . 7
      - 2.1.4.2.3 Accessing the PGD Cluster . . . 12
      - 2.1.4.2.4 Next Steps . . . 13
    - 2.1.4.3 First steps with your Quickstart PGD Cluster . . . 13
      - 2.1.4.3.1 Working with SQL and the PGD Cluster . . . 14
      - 2.1.4.3.2 Loading Data into your PGD Cluster . . . 16
      - 2.1.4.3.3 Using PGD CLI . . . 18
    - 2.1.4.4 Expanded Examples and Use Cases . . . 26
      - 2.1.4.4.1 Use Cases . . . 26

3 How to **27**
- 3.1 Essential How-To . . . 27
  - 3.1.1 Overview . . . 27
  - 3.1.2 Essential features . . . 28
  - 3.1.3 Essential How-To Guides . . . 28
    - 3.1.3.1 PGD Essential architectures . . . 28
      - 3.1.3.1.1 Choosing an architecture . . . 28
      - 3.1.3.1.2 Standard architecture - Ideal for a highly available single location . . . 28
      - 3.1.3.1.3 Near/far architecture - Ideal for disaster recovery . . . 29
    - 3.1.3.2 Installing and configuring EDB Postgres Distributed 6 . . . 33
      - 3.1.3.2.1 1 - Prerequisites for Essential installation . . . 33
      - 3.1.3.2.2 Step 2 - Configure repositories . . . 34

3.1.3.2.3 Step 3 - Installing the database and pgd . . . 36
3.1.3.2.4 Step 4 - Configuring the cluster . . . 37
3.1.3.2.5 Step 5 - Checking the cluster . . . 41
3.1.3.3 Connections . . . 45
3.1.3.3.1 Connecting through the connection manager . . . 45
3.1.3.3.2 Connecting directly to a data node . . . 45
3.1.3.4 Using PGD CLI . . . 46
3.1.3.4.1 Using PGD CLI . . . 46
3.1.3.4.2 Worked example . . . 47
3.1.3.4.3 PGD Essential constraints . . . 50
3.1.3.5 Durability in PGD Essential . . . 52
3.1.3.5.1 Commit Scopes . . . 52
3.1.3.5.2 Using Commit Scopes . . . 53
3.1.3.6 Autopartitioning . . . 54
3.1.3.6.1 Autopartitioning and replication . . . 54
3.1.3.6.2 Range partitioning . . . 54
3.1.3.7 Production Best Practices . . . 55
3.1.3.7.1 Sizing . . . 55
3.1.3.7.2 Time and PGD . . . 56
3.1.3.7.3 Security Best Practices . . . 56
3.2 Expanded How-to . . . 56
3.2.1 Overview . . . 56
3.2.2 Expanded Features . . . 56
3.2.2.1 PGD Architectures . . . 57
3.2.2.1.1 Always-On Architecture . . . 57
3.2.2.1.2 Essential Architectures . . . 61
3.2.2.1.3 Multi-Location Architectures . . . 61
3.2.2.1.4 Geo-Distributed Architectures . . . 61
3.2.2.2 Installing and configuring EDB Postgres Distributed 6 . . . 61
3.2.2.2.1 1 - Prerequisites for Expanded installation . . . 61
3.2.2.2.2 Step 2 - Configure repositories . . . 63
3.2.2.2.3 Step 3 - Installing the database and pgd . . . 64
3.2.2.2.4 Step 4 - Configuring the cluster . . . 66
3.2.2.2.5 Step 5 - Checking the cluster . . . 70

4 In depth **75**
4.1 PGD concepts explained . . . 75
4.1.1 PGD concepts . . . 75
4.1.2 PGD Expanded concepts . . . 75
4.1.2.1 Replication . . . 76
4.1.2.1.1 How Replication Works . . . 76
4.1.2.1.2 Commit scopes and replication . . . 76
4.1.2.1.3 What is replicated? . . . 77
4.1.2.2 PGD Nodes and Groups . . . 77
4.1.2.2.1 Data Nodes . . . 77
4.1.2.2.2 Groups . . . 77
4.1.2.2.3 Witness Nodes . . . 77
4.1.2.2.4 Subscriber-only Nodes . . . 78
4.1.2.2.5 Logical Standby Nodes . . . 78
4.1.2.3 Connection Management . . . 78
4.1.2.4 Locking . . . 78
4.1.2.4.1 Kinds of Locks . . . 78
4.1.2.5 Durability . . . 79
4.1.2.6 Commit Scopes . . . 79

4.1.2.7 Lag Control . . . 79
4.1.2.7.1 Lag Control in PGD . . . 79
4.1.2.8 Expanded Commit Scopes . . . 80
4.1.2.9 Geo-Distributed Clusters . . . 80
4.1.2.10 Advanced Durability . . . 80
4.1.2.11 Conflict Management . . . 80

5 Reference **81**
5.1 PGD Reference . . . 81
5.1.1 Functions and Commands . . . 81
5.1.2 Configuration and Management . . . 81
5.1.3 Advanced Functionality . . . 82
5.1.3.1 PGD overview . . . 82
5.1.3.1.1 Architecture overview . . . 83
5.1.3.1.2 PGD overview - architecture and performance . . . 85
5.1.3.1.3 PGD compared . . . 86
5.1.3.2 Advanced functionality . . . 89
5.1.3.2.1 DDL replication . . . 89
5.1.3.2.2 Decoding worker . . . 106
5.1.3.2.3 CDC Failover support . . . 108
5.1.3.2.4 Parallel Apply . . . 112
5.1.3.2.5 Replication sets . . . 114
5.1.3.2.6 Security and roles . . . 121
5.1.3.2.7 Sequences . . . 130
5.1.3.2.8 Stream triggers . . . 137
5.1.3.2.9 Transaction streaming . . . 143
5.1.3.2.10 Explicit two-phase commit (2PC) . . . 146
5.1.3.3 Config and management . . . 147
5.1.3.3.1 Node types and capabilities . . . 147
5.1.3.3.2 Node management . . . 155
5.1.3.3.3 Connection Manager . . . 167
5.1.3.3.4 Postgres configuration . . . 174
5.1.3.3.5 Backup and recovery . . . 176
5.1.3.3.6 Monitoring . . . 182
5.1.3.3.7 AutoPartition in PGD . . . 195
5.1.3.3.8 Commit Scopes . . . 198
5.1.3.3.9 Conflict Management . . . 229
5.1.3.3.10 Testing and tuning PGD clusters . . . 261
5.1.3.4 Functions and command . . . 263
5.1.3.4.1 Tables, views and functions reference . . . 263
5.1.3.4.2 EDB Postgres Distributed Command Line Interface (PGD CLI) . . . 439
5.1.3.5 Guidance for dev . . . 500
5.1.3.5.1 Application use . . . 500
5.2 Terminology . . . 510
5.2.1 Asynchronous replication . . . 510
5.2.2 Commit scopes . . . 510
5.2.3 CAMO or commit-at-most-once . . . 510
5.2.4 Conflicts . . . 510
5.2.5 Consensus . . . 510
5.2.6 Cluster . . . 511
5.2.7 DDL (data definition language) . . . 511
5.2.8 DML (data manipulation language) . . . 511
5.2.9 Eager . . . 511
5.2.10 Eventual consistency . . . 511

5.2.11 Failover . . . 511
5.2.12 Group commit . . . 511
5.2.13 Immediate consistency . . . 511
5.2.14 Logical replication . . . 512
5.2.15 Node . . . 512
5.2.16 Node groups . . . 512
5.2.17 PGD cluster . . . 512
5.2.18 PGD node . . . 512
5.2.19 Physical replication . . . 512
5.2.20 Postgres cluster . . . 512
5.2.21 Quorum . . . 513
5.2.22 Replicated available fault tolerance (Raft) . . . 513
5.2.23 Read scalability . . . 513
5.2.24 Subscription . . . 513
5.2.25 Switchover . . . 513
5.2.26 Synchronous replication . . . 513
5.2.27 Subscriber-only nodes . . . 513
5.2.28 Two-phase commit (2PC) . . . 514
5.2.29 Vertical scaling or scale up . . . 514
5.2.30 Witness nodes . . . 514
5.2.31 Write leader . . . 514
5.2.32 Writer . . . 514

6 Appendix **515**
6.1 PGD compatibility . . . 515
6.1.1 PGD compatibility with PostgreSQL versions . . . 515
6.1.2 PGD compatibility with operating systems and architectures . . . 515
6.1.2.1 Linux . . . 516
6.2 EDB Postgres Distributed 6 release notes . . . 516
6.2.1 EDB Postgres Distributed 6.2.0 release notes . . . 516
6.2.1.1 Highlights . . . 517
6.2.1.2 Features . . . 517
6.2.1.3 Enhancements . . . 518
6.2.1.4 Bug Fixes . . . 519
6.2.2 EDB Postgres Distributed 6.1.0 release notes . . . 521
6.2.2.1 Highlights . . . 521
6.2.2.2 Features . . . 521
6.2.2.3 Enhancements . . . 522
6.2.2.4 Changes . . . 523
6.2.2.5 Bug Fixes . . . 523
6.2.3 EDB Postgres Distributed 6.1.1 release notes . . . 525
6.2.3.1 Bug Fixes . . . 525
6.2.4 EDB Postgres Distributed 6.1.2 release notes . . . 526
6.2.4.1 Bug Fixes . . . 526
6.2.5 EDB Postgres Distributed 6.0.1 release notes . . . 526
6.2.5.1 Highlights . . . 526
6.2.5.2 Features . . . 527
6.2.5.3 Enhancements . . . 529
6.2.5.4 Changes . . . 531
6.2.5.5 Bug Fixes . . . 532
6.2.6 EDB Postgres Distributed 6.0.2 release notes . . . 533
6.2.6.1 Bug Fixes . . . 533
6.3 Known issues and limitations . . . 534
6.3.1 Known issues . . . 534

6.3.2 Limitations . . . 535
6.3.2.1 Nodes . . . 535
6.3.2.2 Multiple databases on single instances . . . 535
6.3.2.3 Durability options (Group Commit/CAMO) . . . 536
6.3.2.4 General durability limitations . . . 536
6.3.2.5 Group Commit . . . 536
6.3.2.6 Eager . . . 537
6.3.3 CAMO . . . 537
6.3.3.1 Mixed PGD versions . . . 538
6.3.3.2 Other limitations . . . 538

7 Upgrading **539**
7.1 Upgrading PGD . . . 539
7.2 Upgrading Postgres or Postgres and PGD major versions . . . 539
7.3 Other upgrades . . . 540
7.3.1 Upgrading PGD clusters manually . . . 540
7.3.1.1 Upgrade planning . . . 540
7.3.1.1.1 Rolling upgrade considerations . . . 540
7.3.1.1.2 Rolling server software upgrades . . . 541
7.3.1.1.3 Rolling upgrade using node join . . . 541
7.3.1.1.4 Upgrading a CAMO-enabled cluster . . . 542
7.3.1.2 Upgrade preparation . . . 542
7.3.1.3 Server software upgrade . . . 542
7.3.1.3.1 BDR extension upgrade . . . 542
7.3.1.3.2 Fence the node . . . 542
7.3.1.3.3 Stop Postgres . . . 543
7.3.1.3.4 Upgrade packages . . . 543
7.3.1.3.5 Start Postgres . . . 543
7.3.1.3.6 Unfence the node . . . 543
7.3.1.3.7 Postgres upgrade . . . 543
7.3.1.3.8 Minor version Postgres upgrade . . . 543
7.3.1.3.9 Major version Postgres upgrade . . . 544
7.3.1.4 Upgrade check and validation . . . 544
7.3.1.5 PGD 5 - Moving from PGD Proxy to Connection Manager . . . 544
7.3.2 Supported PGD upgrade paths . . . 545
7.3.2.1 Upgrading to version {{version.full}} . . . 545
7.3.3 Compatibility changes . . . 545
7.3.3.1 Summary . . . 545
7.3.3.2 Connection Manager . . . 545
7.3.3.3 DDL Locking changes . . . 545
7.3.3.4 Join/part behavior . . . 546
7.3.3.4.1 Function and node state changes . . . 546
7.3.3.4.2 Sequences conversion . . . 546
7.3.3.4.3 Parting a node drops it . . . 546
7.3.3.5 Administration function changes . . . 546
7.3.3.6 Other . . . 547
7.3.3.6.1 Replica identity . . . 547
7.3.3.6.2 Run on (all) writes . . . 547
7.3.3.6.3 General UI improvements . . . 547
7.3.3.7 Changes to defaults . . . 547
7.3.3.7.1 Global lock timeout . . . 547
7.3.3.7.2 Auto sync . . . 547
7.3.4 Application schema upgrades . . . 547
7.3.4.1 Rolling application schema upgrades . . . 548

7.3.5 In-place Postgres or Postgres and PGD major version upgrades . . . 549
7.3.5.1 Terminology . . . 549
7.3.5.2 Precautions . . . 549
7.3.5.3 Usage . . . 550
7.3.5.3.1 `pgd node upgrade` command-line . . . 550
7.3.5.3.2 Synopsis . . . 550
7.3.5.3.3 Options . . . 550
7.3.5.3.4 Environment variables . . . 551
7.3.5.3.5 Example . . . 551
7.3.5.3.6 Steps performed . . . 552
7.3.5.3.7 PGD Postgres checks . . . 552
7.3.5.3.8 pg_upgrade steps . . . 552
7.3.5.3.9 PGD post-upgrade steps . . . 552
7.3.6 Performing a Postgres major version rolling upgrade on a PGD cluster . . . 553
7.3.6.1 Upgrading Postgres major versions . . . 553
7.3.6.1.1 Prepare the upgrade . . . 553
7.3.6.1.2 Perform the upgrade on each node . . . 554
7.3.6.2 Worked example: Upgrade PGD 4 to PGD 6 . . . 555
7.3.6.2.1 Overview . . . 555
7.3.6.2.2 Confirm the harp-proxy leader . . . 556
7.3.6.2.3 Fence the node . . . 556
7.3.6.2.4 Stop the Postgres service . . . 557
7.3.6.2.5 Stop HARP manager . . . 557
7.3.6.2.6 Remove and install packages . . . 557
7.3.6.2.7 Start the Postgres service . . . 557
7.3.6.2.8 Start HARP manager . . . 557
7.3.6.2.9 Unfence the node . . . 557
7.3.6.2.10 Repeat steps for all nodes . . . 557
7.3.6.2.11 Confirm cluster version . . . 557
7.3.6.2.12 Confirm SCRAM hashes . . . 557
7.3.6.2.13 Enable routing . . . 558
7.3.6.2.14 Switch to Connection Manager . . . 558
7.3.6.2.15 Stop harp manager and proxy services. . . . 558
7.3.6.3 Worked example: Upgrade PGD 5 to PGD 6 . . . 558
7.3.6.4 Prerequisites . . . 559
7.3.6.4.1 Install packages for the new server and PGD . . . 559
7.3.6.5 Pre-upgrade steps . . . 559
7.3.6.5.1 Version Check . . . 559
7.3.6.5.2 Move to Connection Manager . . . 559
7.3.6.5.3 Write leader node verification . . . 560
7.3.6.5.4 Fence the node . . . 560
7.3.6.5.5 Initialize the new Postgres instance . . . 560
7.3.6.5.6 Migrate configuration to the new Postgres version . . . 561
7.3.6.5.7 Execute the upgrade. . . . 562
7.3.6.6 Post-upgrade steps . . . 564
7.3.6.6.1 Update the Postgres service file . . . 564
7.3.6.6.2 Start the postgres service . . . 565
7.3.6.6.3 Verify the upgraded cluster versions . . . 565
7.3.6.6.4 Unfence the node . . . 566
7.3.6.6.5 Verify the Connection Manager is working . . . 566
7.3.6.6.6 Clean up and vacuum analyze . . . 566
7.3.6.6.7 Upgrade the remaining nodes . . . 566
7.3.6.6.8 Verify the final state of the cluster . . . 567
7.3.6.6.9 Verify the Connection Manager . . . 567

Welcome to the PGD {{version.full}} documentation. PGD {{version.full}} is available in two editions: Essential and Expanded.

CHAPTER 1

Introduction

1.1 Why PGD?

Modern data architectures require an extensible approach to data management, whether the requirement is for high availability, disaster recovery or multi-region data distribution. PGD is designed to meet these needs, and in PGD 6 we have made it easier to get started with PGD, while also providing a pathway to using advanced features as your use case becomes more complex.

1.2 What does PGD enable?

PGD enables you to build a distributed database architecture that can span multiple regions, data centers, or cloud providers. It provides multi-master replication and data distribution. Postgres databases can be deployed into data groups within the cluster and data within each node can be distributed across multiple nodes.

1.3 What are the differences between PGD Essential and PGD Expanded?

PGD Expanded is the full-featured version of PGD. It includes all the features of PGD Essential, as well as additional features such as advanced conflict management, data distribution, and support for large-scale deployments. PGD Expanded is designed for users who need the most advanced features and capabilities of PGD.

PGD Essential is a simplified version of PGD Expanded. It is designed for users who want to get started with PGD quickly and easily, without the need for advanced features or complex configurations. PGD Essential includes the core features of PGD but enables them in a way that makes replication and availability simple. It therefore does not include some of the more advanced features available in PGD Expanded.

PGD Essential limits the number of data nodes in a cluster to four and the number of groups to two. It also limits the number of nodes in a group to four. PGD Expanded does not have these limitations.

Learn more about PGD in Get Started with PGD .

CHAPTER 2

Getting started

2.1 Get started with PGD

To begin using any edition of EDB Postgres Distributed, we recommend you first try our local installation and configuration guide.

This guide will help you install and configure the software, and create your first cluster.

2.1.1 What is EDB Postgres Distributed?

EDB Postgres Distributed (PGD) is a distributed database solution that provides high availability, scalability, and fault tolerance for PostgreSQL databases. It allows you to create clusters of PostgreSQL instances that can work together to provide a single, unified database system.

2.1.2 What is EDB Postgres Distributed Essential?

EDB Postgres Distributed Essential is a streamlined version of PGD that focuses on delivering core distributed database functionality with minimal complexity. It is designed for users who need basic high availability and disaster recovery features without the advanced capabilities offered by PGD Expanded, the full version.

2.1.3 What is the PGD Essential Standard architecture

Get to know what EDB Postgres Distributed Essential is all about in *An introduction to PGD Essential* .

2.1.4 Create your first PGD Essential cluster with Docker Compose

Use the Docker Compose file to *Creating your first cluster (PGD Essential)* with three nodes. This is a great way to get started with PGD Essential and see how it works in a real-world scenario and a stepping stone to deploying a production cluster with PGD Essential or PGD Expanded.

2.1.4.1 An introduction to PGD Essential

EDB Postgres Distributed (PGD) Essential is a simplified version of PGD Expanded, designed to help you get started with distributed databases quickly and easily. It provides the core features of PGD, enabling high availability and disaster recovery without the complexity of advanced configurations.

At the core of PGD are data nodes, Postgres databases that are part of a PGD cluster. PGD enables these databases to replicate data efficiently between nodes, ensuring that your data is always available and up-to-date. PGD Essential simplifies this process by providing a standard architecture that is easy to set up and manage.

The standard architecture is built around a single data group, which is the basic architectural element for EDB Postgres Distributed systems. Within a group, nodes cooperate to select which nodes handle incoming write or read traffic, and identify when nodes are available or out of sync with the rest of the group. Groups are most commonly used on a single location where the nodes are in the same data center and where you have just the one group in the cluster, we also call it the one-location architecture.

2.1.4.1.1 Standard/One-location architecture

The one-location architecture consists of a single PGD cluster with three nodes. The nodes are located in the same data center or region. Ideally they are in different availability zones, but that isn't required. The nodes are connected to each other using a high-speed network.

The nodes are configured as a data group which means that they replicate data to each other within the same group. While PGD can handle multiple writers in a network, this requires more advanced conflict management and is not supported in PGD Essential.

Therefore, in the standard architecture, one node is designated as the write leader node, which handles all write transactions. The other nodes in the group are read-only nodes that replicate data from the write leader.

The write leader node is one node selected by the nodes in the group to handle all the writes. It is responsible for accepting write transactions and replicating them to the other nodes in the group. If the write leader node fails, the other nodes in the group will elect a new write leader node.

Applications can connect to any node in the cluster using PGD's Connection Manager ports which runs on every data node. It will automatically route read and write transactions to the write leader. It can also route read only transactions to the other nodes in the group.

Fig. 1: Standard architecture

In this diagram, you can see the applications connecting to the PGD cluster through the Connection Manager ports. The Connection Manager is responsible for routing the read and write transactions to the appropriate nodes in the group. The write leader is responsible for handling all write transactions and is shown in at the top in AZ1 in green.

The other nodes in the group are read-only nodes that replicate data from the write leader. Applications connecting to the read-only nodes Connection Manager read/write ports will have their queries and changes routed to the write leader. All the time, the nodes are talking to each other replicationing data to ensure they are in sync.

2.1.4.2 Creating your first cluster (PGD Essential)

This part of the Getting Started guide will help you create a local cluster using Docker Compose. This is a great way to get familiar with the EDB Postgres Distributed (PGD) Essential features and functionality.

2.1.4.2.1 Prerequisites

- Docker and Docker Compose installed on your local machine.

2.1.4.2.2 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/get-started/assets/pgd_quickstart.sh␣
↪| bash
```

```
#  pgd_quickstart.sh

# !/bin/bash

#  This script will unpack embedded files directly into the current working directory.

#  Designed for curl ... | bash execution.

#  Exit immediately if a command exits with a non-zero status.

#  Pipefail ensures that a pipelines return status is the value of the last␣
↪(rightmost) command

#  to exit with a non-zero status, or zero if all commands in the pipeline exit␣
↪successfully.

set -eo pipefail

echo "Starting PGD Docker Quickstart unpacker (via curl | bash)..."
echo ""
echo "================================================================================"
echo "WARNING: Files will be extracted directly into your current directory: $(pwd)"
echo "         This process requires the current directory to be EMPTY."
echo "         Affected files: Dockerfile.pge, docker-compose.yml, docker-entrypoint.
↪sh, qs.sh" # <--- UPDATED list
echo "================================================================================"
echo ""
```

(continues on next page)

(continued from previous page)

```
#  --- Check if current directory is empty ---

#  ls -A . lists all files and directories except . and ..

if [ -n "$(ls -A .)" ]; then
    echo "Error: The current directory $(pwd) is NOT empty." >&2
    echo "       Please run this script from an empty directory to prevent accidental␣
↪overwrites." >&2
    echo "       Aborting extraction." >&2
    exit 1
fi

echo "Current directory is empty. Proceeding with extraction."
echo "" # Newline for readability

#  --- No temporary directory creation or cd operations here ---

#  Files will be extracted directly into the current working directory.

echo "Extracting embedded files..."

#  --- Embedded Data Decoding and Extraction ---

#  Each files content is base64 encoded and embedded here as a here-document.

echo "  Extracting Dockerfile.pge..."
base64 -d <<EOF_DOCKERFILE_PGE_ > "Dockerfile.pge"
RlJPTSBkZWJpYW4KClJVTiBhcHQtZ2V0IHVwZGF0ZSAteSAmJiBhcHQtZ2V0IGluc3RhbGwgLXkgY3Vy-
bAoKQVJHIEVEQl9TVUJTQ1JJUFRJT05fVE9LRU49IiIKClJVTiBjdXJsIC0xc0xmICJodHRwc-
zovL2Rvd25sb2Fkcy5lbnRlcnByaXNlZGIuY29tLyR7RURCX1NVQlNDUklQVElPTl9UT0tFTn0vZW50ZXJwcm-
lzZS9zZXR1cC5kZWIuc2giIHwgYmFzaApSVU4gY3VybCAtMXNMZiAiaHR0cHM6Ly9kb3dubG9hZH-
MuZW50ZXJwcmlzZWRiLmNvbS8ke0VEQl9TVU-
JTQ1JJUFRJT05fVE9LRU59L3Bvc3RncmVzX2Rpc3RyaWJ1dGVkL3NldHVwLmRlYi5zaCIgfCBiYXNoCg-
pSVU4gYXB0LWdldCB1cGRhdGUgLXkgJiYgYXB0LWdld-
CBpbnN0YWxsIC15IGVkYi1wb3N0Z3Jlc2V4dGVuZGVkLTE3IGVkYi1wZ2Q2LWVzc2VudGl-
hbC1wZ2V4dGVuZGVkMTcgClJVTiBhcHQtZ2V0IGluc3RhbGwganEgLXkKUlVOIGFwdC1nZXQgaW5zdGFsb-
CAteSBpcHV0aWxzLXBpbmcKClJVTiBta2RpciAtcCAvdmFyL2xpYi9wb3N0Z3Jlc3FsCg-
pDT1BZIC4vZG9ja2VyLWVudHJ5cG9pbnQuc2ggL3Zhci9saWIvcG9zdGdyZXNxbC8KClJVTiBjaG93biAtU-
iBwb3N0Z3Jlczpwb3N0Z3JlcyAvdmFyL2xpYi9wb3N0Z3Jlc3FsCgpSVU4gbWtkaXIgLXAgL2V0Yy9lZ-
GIvcGdkLWNsaQoKUlVOIGNob3duIC1SIHBvc3RncmVzOnBvc3RncmVzIC9ldGMvZWRiL3BnZC1jbGkvC-
goKRU5WIFBBVEg9Ii91c3IvbGliL2VkYi1wZ2UvMTcvYmluOiR7UEFUSH0iCkVOViBQR0RBVEE9Ii92YX-
IvbGliL2VkYi1wZ2UvMTcvZGF0YSIKClJVTiBta2RpciAtcCAvdmFyL2xpYi9lZGItcGdlLzE3L2RhdGEKUl-
VOIGNob3duIC1SIHBvc3RncmVzOnBvc3RncmVzIC92YX-
IvbGliL2VkYi1wZ2UvMTcvZGF0YQoKVk9MVU1FIC92YXIvbGliL2VkYi1wZ2UvMTcvZGF0YQoKVVN-
FUiBwb3N0Z3JlcwpXT1JLRElSIC92YXIvbGliL3Bvc3RncmVzcWwKCkNNRCBbIi92YX-
IvbGliL3Bvc3RncmVzcWwvZG9ja2VyLWVudHJ5cG9pbnQuc2giXQo=
EOF_DOCKERFILE_PGE_
echo "  Extracting docker-compose.yml..."
base64 -d <<EOF_DOCKER_COMPOSE_YML_ > "docker-compose.yml"
c2VydmljZXM6CiAgaG9zdC0xOgogICAgaW1hZ2U6IHBnZAogICAgaG9zdG5hbWU6IGhvc3QtMQog-
ICAgZW52aXJvbm1lbnQ6CiAgICAgIFBHUEFTU1dPUkQ6IHNlY3JldAogICAgICBQR0RfSk9JTl9OT0RFX0RT-
TjogInBvcnQ9NTQzMiBkYm5hbWU9cGdkZGIgaG9zdD1ob3N0LTEgdXNlcj1wb3N0Z3JlcyIKICAgI-
HJlc3RhcnQ6IGFsd2F5cwogICAgdm9sdW1lczoKICAgICAgLSBwZ2RhdGEtaG9zdC0xOi92YX-
IvbGliL2VkYi1wZ2UvMTcvZGF0YQoKICBob3N0LTI6CiAgICBpbWFnZTogcGdkCiAgICBob3N0bmFtZTo-
gaG9zdC0yCiAgICBlbnZpcm9ubWVudDoKICAgICAgUEdQQVNTV09SRDogc2VjcmV0CiAgICAgIF-
BHRF9KT0lOX05PREVfRFNOOiAicG9ydD01NDMyIGRibmFtZT1wZ2RkYi-
```

(continues on next page)

(continued from previous page)

```
Bob3N0PWhvc3QtMSB1c2VyPXBvc3RncmVzIgogICAgcmVzdGFydDogYWx3YXlzCiAgICB2b2x1bWVzOgog-
ICAgICAtIHBnZGF0YS1ob3N0LTI6L3Zhci9saWIvZWRiLXBnZS8xNy9kYXRhCgogIGhvc3QtMzoKICAgIGltY-
WdlOiBwZ2QKICAgIGhvc3RuYW1lOiBob3N0LTMKICAgIGVudmlyb25tZW50Ogog-
ICAgICBQR1BBU1NXT1JEOiBzZWNyZXQ-
KICAgICAgUEdEX0pPSU5fTk9ERV9EU046ICJwb3J0PTU0MzIgZGJuYW1lPXBnZGRiIGhvc3Q9aG9zdC0xIHVz-
ZXI9cG9zdGdyZXMiCiAgICByZXN0YXJ0OiBhbHdheXMKICAgIHBvcnRzOgogICAgICAtICI2NDMyOjY0MzI-
iCiAgICAgIC0gIjY0MzM6NjQzMyIKICAgICAgLSAiNjQzNDo2NDM0IgogICAgdm9sdW1lczoKICAgICAgLS-
BwZ2RhdGEtaG9zdC0zOi92YXIvbGliL2VkYi1wZ2UvMTcvZGF0YQoKCnZvbHVtZXM6CiAgcGd-
kYXRhLWhvc3QtMToKICAgIGRyaXZlcjogbG9jYWwKICAgIGRyaXZlcl9vcHRzOgogICAgICBvOiBiaW5kCiAg-
ICAgIHR5cGU6IG5vbmUKICAgICAgZGV2aWNlOiAuL2hvc3QtMS12b2x1bWUKICBwZ2RhdGEtaG9zdC0yOgog-
ICAgZHJpdmVyOiBsb2NhbAogICAgZHJpdmVyX29wdHM6CiAgICAgIG86IGJpbmQKICAgICAgdHlwZ-
Togbm9uZQogICAgICBkZXZpY2U6IC4vaG9zdC0yLXZvbHVtZQogIHBnZGF0YS1ob3N0LTM6CiAg-
ICBkcml2ZXI6IGxvY2FsCiAgICBkcml2ZXJfb3B0czoKICAgICAgbzogYmluZAog-
ICAgICB0eXBlOiBub25lCiAgICAgIGRldmljZTogLi9ob3N0LTMtdm9sdW1lCiAgIAo=
EOF_DOCKER_COMPOSE_YML_
echo "  Extracting docker-entrypoint.sh..."
base64 -d <<EOF_DOCKER_ENTRYPOINT_SH_ > "docker-entrypoint.sh"
IyEvdXNyL2Jpbi9lbnYgYmFzaAojIFRoaXMgdmVyc2lvbiB3aWxsIGFzc3VtZSB0aGF0IGl0IGlzIFBHRQpz-
ZXQgLWV1Cgpwd2QKClBHRF9JTklUSUFMX05PREVfQ09VTlQ9IiR7UEdEX0lOSVRJQUxfTk9ERV9DT1VOV-
DotM30iClBHRF9IT1NUX05BTUU9IiR7UEdEX0hPU1RfTkFNRTotJChjYXQgL2V0Yy9ob3N0bmFtZSB8IHh-
hcmdzKX0iClBHRF9OT0RFX05BTUU9IiR7UEdEX05PREVfTkFNRTotJChjYXQgL2V0Yy9ob3N0bmFtZSB8IHh-
hcmdzIHwgc2VkIHMvaG9zdC0vbm9kZS0vKX0iClBHRF9OT0RFX0dST1VQP-
SIke1BHRF9OT0RFX0dST1VQOi1ncm91cC0xf-
SIKUEdEX0NMVVNURVJfTkFNRT0iJHtQR0RfQ0xVU1RFUl9OQU1FOi1wZ2R9IgoKUE9TVEdSRVN-
fREI9IiR7UE9TVEdSRVNfREI6LXBnZGRifSIKUE9TVEdSRVNfVVN-
FUj0iJHtQT1NUR1JFU19VU0VSOi1wb3N0Z3Jlc30iClBHREFUQT0iJHtQR0RBVEE6LS92YX-
IvbGliL2VkYi1wZ2UvMTcvbWFpbi99IgpQR0xPR0ZJTEU9IiR7UEdMT0dGSUxFOi0vdm-
FyL2xpYi9lZGItcGdlLzE3L2xvZ2ZpbGV9IgoKZWNobyAiQ29uZmlndXJpbmcgJHtQR0RfTk9ERV9OQU1Ff-
SAoJHtQR0RfSE9TVF9OQU1FfSkiCgpQR19QR0RfQ0xJX0NPTkZfQ09OVEVOVFM9JwpjbHVzdGVyO-
gogIG5hbWU6IHBnZAogIGVuZHBvaW50czoKICAgIC0gaG9zdD1ob3N0LTEgZGJuYW1lPXBnZGRiIHB-
vcnQ9NTQzMgogICAgLSBob3N0PWhvc3QtMiBkYm5hbWU9cGdkZGIgcG9ydD01NDMyCiAgI-
CAtIGhvc3Q9aG9zdC0zIGRibmFtZT1wZ2RkYiBwb3J0PTU0MzIKJwoKIyBUaGlzIHdpbGwgYmUgdXNlZCBieS-
BgcGdkIG5vZGUgc2V0dXBgLgpleHBvcnQgUEdQQVNTV09SRD0iJHtQR1BBU1NXT1JEOi0kUE9TVEdSRVN-
fUEFTU1dPUkR9IgoKZWNobyAiJFBHX1BHRF9DTElfQ09ORl9DT05URU5UUyIgPi-
AvZXRjL2VkYi9wZ2QtY2xpL3BnZC1jbGktY29uZmlnLnltbAoKaWYgISBbIC1zICIkUEdEQVRBL1BHX1ZFUl-
NJT04iIF07IHRoZW4KICAgIGVjaG8gIlByb3Zpc2lvbmluZyBQb3N0Z3Jlcy4iCgog-
ICAgUFJJTUFSWT0nZmFsc2UnCiAgICBjYXNlICIkUEdEX0pPSU5fTk9ERV9EU04iIGluCgkqIi-
Bob3N0PSRQR0RfSE9TVF9OQU1FICIqIHwgImhvc3Q9JFBHRF9IT1NUX05BTUUgIiogfCAqKSBQR1BBU1NXT-
Bob3N0PSRQR0RfSE9TVF9OQU1FICIkKCgkgICAgUFJJTUFSWT0ndHJ1ZScKCSAgICA7OwogICAgZXNhYwo-
KICAgIGlmIFsgJFBSSU1BUlkgPSAndHJ1ZScgXTsgdGhlbgoJZWNobyAiUHJvdmlzaW9uaW5nIF-
BHRCBub2RlIGFuZCBuZXcgZ3JvdXAuIgoJcGdkIG5vZGUgIiRQR0RfTk9ERV9OQU1FIiBzZXR1cCAtLXZl-
cmJvc2UgXAoJICAgIC0tZHNuICIkUEdEX0pPSU5fTk9ERV9EU04iIFwKCSAgICAtLWxpc3Rlbi1hZGRyI-
CIkUEdEX0hPU1RfTkFNRSxsb2NhbGhvc3QiIFwKCSAgICAtLWluaXRpYWwtbm9kZS1jb3VudCAiJFBHRF9JTk-
lUSUFMX05PREVfQ09VTlQiIFwKCSAgICAtLXBnZGF0YSAiJFBHREFUQSIgXAoJICAgIC0tbG9nLWZpbGUgI-
iRQR0xPR0ZJTEUiIFwKCSAgICAtLWNsdXN0ZXItbmFtZSAiJFBHRF9DTFVTVEVSX05BTUUiIFwKCSAgICAtL-
Wdyb3VwLW5hbWUgIiRQR0RfTk9ERV9HUk9VUCIgCiAgICBlbHNlCgllY2hvICJQcm92aXNpb25pbm-
cgUEdEIG5vZGUgdG8gam9pbiBleGlzdGluZyBncm91cC4iCgoJIyBJbiBjYXNlI-
HdlIG5lZWQgdG8gZG8gY2xlYW51cC4KCXBzcWwgIiRQR0RfSk9JTl9OT0RFX0RTTiIgLWMgJ1NFTEVD-
VCBiZHIucnVuX29uX2FsbF9ub2Rlcygkb25jKCkgIyBmRyLmRyb3Bfbm9kZSgnIickUEdEX05PRE-
VfTkFNRSciJywgZm9yY2UgOj0gdHJ1ZSkgJCQpOycKCSAgICAtLWxvZ2Qgbm9kZSAiJFBHRF9OT0RFX05BTUUiIHNld-
HVwIC0tdmVyYm9zZSBcCgkgICAgLS1kc24gImhvc3Q9JFBHRF9IT1NUX05BTUUgcG9ydD01NDMyI-
GRibmFtZT0kUE9TVEdSRVNfREIgdXNlcj0kUE9TVEdSRVNfVVNFUiIgXAoJICAgIC0tbGlzdGVuLWFkZHIgI-
iRQR0RfSE9TVF9OQU1FLGxvY2FsaG9zdCIgXAoJICAgIC0tcGdkYXRhICIkUEdEQVRBIiBcCgkg-
ICAgLS1sb2ctZmlsZSAiJFBHTE9HRklMRSIgXAoJICAgIC0tY2x1c3Rlci1kc24gI-
iRQR0RfSk9JTl9OT0RFX0RTTiIgXAoJICAgIC0tZ3JvdXAtbmFtZSAiJFBHRF9OT0RFX0dST1VQIiBcCgkg-
ICAgLS1jbHVzdGVyLW5hbWUgIiRQR0RfQ0xVU1RFUl9OQU1FIiB8fCAocm0gLXJmICIkUEdEQVRBIiAmJi-
BleGl0IDEpCiAgICBmaQpmaQoKcGdfY3RsIC1EICIkUEdEQVRBIiBzdG9wIHx8IGV-
```

(continues on next page)

(continued from previous page)

```
jaG8gIlBvc3RncmVzIG5vdCBydW5uaW5nIgoKZXhlYyBwb3N0Z3JlcyAtRCAiJFBHREFUQSIK
EOF_DOCKER_ENTRYPOINT_SH_
echo "  Extracting qs.sh..."
base64 -d <<EOF_QS_SH_ > "qs.sh"
IyEvYmluL2Jhc2gKIyBxcy5zaCAtIFBHRCBRdWlja3N0YXJ0IFV0aWxpdHkgU2NyaXB0CiMgVGhpcy-
BzY3JpcHQgcHJvdmlkZXMgY29tbWFuZHMgZm9yIGJ1aWxkaW5nLCBzdGFydGluZywgc3RvcHBpbmcsCiM-
gYW5kIGludGVyYWN0aW5nIHdpdGggdGhlIFBHRCBEb2NrZXIgZW52aXJvbm1lbnQuCgojIEV4aXQgaW1tZWR-
pYXRlbHkgaWYgYSBjb21tYW5kIGV4aXRzIHdpdGggYSBub24temVybyBzdGF0dXMuCnNld-
CAtZW8gcGlwZWZhaWwKCiMgLS0tIFV0aWxpdHkgRnVuY3Rpb25zIC0tLQoKY29tbWFuZF9leGlzdHMgKCkge-
wogICAgY29tbWFuZCAtdiAiJDEiID4vZGV2L251bGwgMj4mMQp9CgpjaGVja19kb2NrZXIoKSB7CiAgICBpZi-
AhIGNvbW1hbmRfZXhpc3RzIGRvY2tlcjsgdGhlbgogICAgICAgIGVjaG8gIkVycm9yOiBEb2NrZXI-
gaXMgbm90IGluc3RhbGxlZCBvciBub3QgaW4gUEFUSC4iID4mMgogICAgICAgIGVjaG8gIlBsZWFzZS-
BpbnN0YWxsIERvY2tlciB0byBwcm9jZWVkLiIgPiYyCiAgICAgICAgcmV0dXJuIDEKICAgIGZpCiAg-
ICBlY2hvICIgIERvY2tlciBmb3VuZC4iCiAgICByZXR1cm4gMAp9CgpjaGVja19kb2NrZXJfY29tcG9zZSg-
pIHsKICAgIGlmICEgZG9ja2VyIGNvbXBvc2UgdmVyc2lvbiA+L2Rldi9udWx-
sIDI+JjE7IHRoZW4KICAgICAgICBlY2hvICJFcnJvcjogRG9ja2VyIENvbXBvc2UgKHYyIG9yIG5ld2VyKSBp-
cyBub3QgaW5zdGFsbGVkIG9yIG5vdCBpbiBQQVRILiIgPiYyCiAgICAgICAgZWNobyAiUGxlYXN-
lIGluc3RhbGwgRG9ja2VyIENvbXBvc2UgdG8gcHJvY2VlZC4iID4mMgogICAgICAgIHJldHVybiAxCiAgICB-
maQogICAgZWNobyAiICBEb2NrZXIgQ29tcG9zZSBmb3VuZC4iCiAgICByZXR1cm4gMAp9CgpjaGVja19zd-
WJzY3JpcHRpb25fdG9rZW4oKSB7CiAgICBpZiBbIC16ICIkRURCX1NVQlNDUklQVElPTl9UT0tFTiIgXTs-
gdGhlbgogICAgICAgIGVjaG8gIkVycm9yOiBFRE-
JfU1VCU0NSSVBUSU9OX1RPS0VOIGVudmlyb25tZW50IHZhcmlhYmxlIGlzIG5vdCBzZXQuIiA+JjI-
KICAgICAgICBlY2hvICJQbGVhc2Ugc2V0IHRoaXMgdmFyaWFibGUgKGUuZy4sIGV4cG9ydCBFRE-
JfU1VCU0NSSVBUSU9OX1RPS0VOPSd5b3VyX3Rva2VuJykgYmVmb3JlIHJ1bm5pbmcgdGhlIGNvbW1hb-
mQuIiA+JjIKICAgICAgICByZXR1cm4gMQogICAgZmkKICAgIGVjaG8gIiAgRURCX1NVQlNDUk-
lQVElPTl9UT0tFTiBpcyBzZXQuIgogICAgcmV0dXJuIDAKfQoKIyAtLS0gQ29tbWFuZHMgLS0tCgpjbWR-
fcHJlcGFyZSgpIHsKICAgIGVjaG8gIlJ1bm5pbmcgJ3ByZXBhcmUnIGNvbW1hbmQ6IENyZWF0aW5nIHZvb-
HVtZSBkaXJlY3Rvcmllcy4uLiIKICAgIGxvY2FsIHZvbHVtZXM9KCJob3N0LTEtdm9sdW1lIiA-
iaG9zdC0yLXZvbHVtZSIgImhvc3QtMy12b2x1bWUiKQogICAgbG9jYWwgc3VjY2Vzcz0wCgogICAgZm9yI-
HZvbCBpbiAiJHt2b2x1bWVzW0BdfSI7IGRvCiAgICAgICAgaWYgWyAhIC1kICIkdm9sIiBdOyB0aGVuCiAg-
ICAgICAgICAgIGVjaG8gIiAgQ3JlYXRpbmcgZGlyZWN0b3J5OiAnJHZvbCcuLi4iCiAg-
ICAgICAgICAgIGlmICEgbWtkaXIgLXAgIiR2b2wiOyB0aGVuCiAgICAgICAgICAgICAgICBlY2hvICJFcn-
JvcjogRmFpbGVkIHRvIGNyZWF0ZSAnJHZvbCcuIiA+JjIKICAgICAgICAgICAgICAgIGVjaG8gIiAg-
ICAgICBZb3UgbWF5IG5lZWQgZWxldmF0ZWQgcGVybWlzc2lvbnMuIFRyeeTogc3VkbyAuL3FzLnNoIHByZXBhc-
mUiID4mMgogICAgICAgICAgICAgICAgc3VjY2Vzcz0xCiAg-
ICAgICAgICAgIGVsc2UKICAgICAgICAgICAgICAgIGVjaG8gI-
iAgU3VjY2Vzc2Z1bGx5IGNyZWF0ZWQgJyR2b2wnLiIKICAgICAgICAgICAgZmkKICAgICAgICBlbHNlCiAg-
ICAgICAgICAgIGVjaG8gIiAgRGlyZWN0b3J5ICckdm9sJyBhbHJlYWR5IGV4aXN0cy4gU2tpcHBpbmcuIgog-
ICAgICAgIGZpCiAgICBkb25lCiAgICByZXR1cm4gJHN1Y2Nlc3MKfQoKY21kX2J1aWxkKCkgewog-
ICAgZWNobyAiUnVubmluZyAnYnVpbGQnIGNvbW1hbmQ6IENoZWNraW5nIGVudmlyb25tZW50IGFuZCBidWl-
sZGluZyBQR0QgRG9ja2VyIGltYWdlLi4uIgogICAgZWNobyAiUGVyZm9ybWluZy-
BzeXN0ZW0gY2hlY2tzLi4uIgoKICAgIGNoZWNrX2RvY2tlciB8fCByZXR1cm4gMQog-
ICAgY2hlY2tfZG9ja2VyX2NvbXBvc2UgfHwgcmV0dXJuIDEKICAgIGNoZWNrX3N1YnNjcmlwdGlvbl90b2tl-
biB8fCByZXR1cm4gMQoKICAgIGVjaG8gIkFsbCBjaGVja3MgcGFzc2VkLiBCdWlsZGluZyBEb2NrZXI-
gaW1hZ2UgJ3BnZCcgZnJvbSBEb2NrZXJmaWxlLnBnZS4uLiIKCiAgICAjIFRoZSAnLicgY29udGV4dCByZWZl-
cnMgdG8gdGhlIGN1cnJlbnQgZGlyZWN0b3J5CiAgICBkb2NrZXIgYnVpbGQgLWYgRG9ja2VyZml-
sZS5wZ2UgLS1idWlsZC1hcmcgRURCX1NVQlNDUklQVElPTl9UT0tFTj0iJEVEQl9TVU-
JTQ1JJUFRJT05fVE9LRU4iIC10IHBnZCAuCgogICAgQlVJTERfU1RBVFVTPSQ/CiAgICBpZiBbICRCVUlMRF9TVEFUVVMgLWVxIDA
EOF_QS_SH_

echo "Files unpacked successfully into the current directory: $(pwd)."
echo ""

#  Make necessary scripts executable

if [ -f "qs.sh" ]; then # <--- UPDATED to qs.sh
    chmod +x qs.sh
```

(continues on next page)

(continued from previous page)

```
    echo "Made qs.sh executable."
fi

#  Assuming docker-entrypoint.sh might also need executable permissions if present

if [ -f "docker-entrypoint.sh" ]; then
    chmod +x docker-entrypoint.sh
    echo "Made docker-entrypoint.sh executable."
fi

echo ""
echo "You can now use the qs.sh command to manage your PGD Docker environment."
echo "For available commands, run: ./qs.sh help"
echo ""
echo "Common next steps:"
echo "1. Create volume directories: ./qs.sh prepare"
echo "2. Build the PGD Docker image: export EDB_SUBSCRIPTION_TOKEN=\"YOUR_EDB_TOKEN\";
↪ ./qs.sh build"
echo "3. Start the PGD services: ./qs.sh start"
echo ""
echo "Remember to clean up these files and created volumes manually when you are done.
↪ Example:"
echo "  ./qs.sh stop"
echo "  ./qs.sh cleanup"
echo "  rm Dockerfile.pge docker-compose.yml docker-entrypoint.sh qs.sh" # <---␣
↪UPDATED list
echo ""

#  The script exits here after printing instructions.

#  No automatic cleanup of extracted files as they are in the users CWD.
```

```
#  docker-compose.yml

services:
  host-1:
    hostname: host-1
    image: pgd
    environment:
      PGPASSWORD: secret
      PGD_JOIN_NODE_DSN: "port=5432 dbname=pgddb host=host-1 user=postgres"
    restart: always
    volumes:
      - ./host-1-data:/var/lib/postgresql/data

  host-2:
    hostname: host-2
    extends: host-1
    volumes:
     - ./host-2-data:/var/lib/postgresql/data

  host-3:
    hostname:  host-3
    extends: host-1
    volumes:
     - ./host-3-data:/var/lib/postgresql/data
```

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.

2.1.4.2.3 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.

2.1.4.2.4 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.

2.1.4.3 First steps with your Quickstart PGD Cluster

Now that you have created your first PGD cluster, you can start working with it. This guide will help you connect to the cluster, load data, and perform basic SQL operations.

- *Working with SQL and the PGD Cluster*
- *Loading Data into your PGD Cluster*
- *Using PGD CLI*

2.1.4.3.1 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
);
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.

2.1.4.3.2 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),
                        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.

2.1.4.3.3 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 *Quickstart Docker Compose kit* .

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.full}}.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
```

(continues on next page)

(continued from previous page)

```
 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
- ----------------+------------
 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
```

(continues on next page)

(continued from previous page)

```
  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␣
↪[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)
```

(continues on next page)

(continued from previous page)

```
 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)
```

(continues on next page)

(continued from previous page)

```
 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       |
```

(continues on next page)

(continued from previous page)

```
 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
 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
- ---------------+-----------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
 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)
```

2.1.4.4 Expanded Examples and Use Cases

While PGD Essential delivers the core functionality needed to get high availability and/or disaster recover use cases up and running quickly, there are many advanced use cases that can be implemented with PGD Essential. This section provides examples of how to implement some of these advanced use cases.

2.1.4.4.1 Use Cases

Use Case 1: Multi-Master Replication

By default, PGD Essential uses the PGD Connection Manager to send your requests to the right node. This node is the write leader and by directing your requests there, it allows conflicts to be rapidly resolved.

With PGD Expanded, you can send your requests to any node in the cluster, and PGD will replicate the changes to the other nodes. Configurable conflict management then allows you to choose how to resolve conflicts.

Use Case 2: Data Distribution

PGD Expanded allows you to distribute your data across multiple nodes in the cluster, including subscriber-only read-only nodes. These nodes can be located in multiple data centers or availability zones. This allows you to scale your database's read capacity horizontally, adding more nodes to the cluster as needed.

Use Case 3: Geo-Distribution

PGD Expanded allows you to distribute your data across multiple regions, replicating data to all the nodes in the cluster. Multiple Data groups can be located in different locations to ensure high availability and resilience in that location.

Use Case 4: Tiered Tables

An optional element of PGD Expanded is the ability to create tiered tables. These tables can be used to tier data between hot data, being replicated within the cluster and cold data being written to a Iceberg/Delta tables data lake. The cold data remains queryable as Tiered Tables uses PGAA which allows you to query the data lake as if it were a table in the database.

CHAPTER 3

How to

3.1 Essential How-To

This section provides essential how-to guides for deploying and managing your PGD cluster. It includes information on architectures, deployment, durability, autopartition, production best practices, and standard operating procedures (SOPs).

3.1.1 Overview

PGD Essential offers a simplified approach to deploying and managing your PGD cluster. It is designed to help you get started quickly and easily, while also providing a pathway to using advanced features as your use case becomes more complex.

At the core of PGD are data nodes, Postgres databases that are part of a PGD cluster. PGD enables these databases to replicate data efficiently between nodes, ensuring that your data is always available and up-to-date. PGD Essential simplifies this process by providing a standard architecture that is easy to set up and manage.

The standard architecture is built around a single data group, which is the basic architectural element for EDB Postgres Distributed systems. Within a group, nodes cooperate to select which nodes handle incoming write or read traffic, and identify when nodes are available or out of sync with the rest of the group. Groups are most commonly used on a single location where the nodes are in the same data center and where you have just the one group in the cluster, we also call it the one-location architecture.

3.1.2 Essential features

- Standard Architecture : Learn about the standard architecture for PGD Essential,which consists of a single data group with three nodes in the same data center or region.
- Near/Far Architecture : Understand the near/far architecture, which consists of two data groups in different locations, with one group handling writes and the other group handling reads.
- Connection Management : Learn how to connect to your PGD cluster using the Connection Manager ports, which automatically route read and write transactions to the appropriate nodes.
- PGD CLI : Discover how to use the PGD CLI to manage your PGD cluster, including creating and managing data groups, nodes, and connections.
- Durability : Understand the durability features of PGD Essential, which ensure that your data is always available and up-to-date.
- Autopartition : Learn about the autopartition feature, which automatically partitions your data across nodes in the cluster for improved performance and scalability.

3.1.3 Essential How-To Guides

- *Simple PGD Essential Installation* : Get step-by-step instructions for installing PGD Essential on your system using the PGD CLI.
- Production Best Practices : Get best practices for deploying and managing your PGD cluster in a production environment, including performance tuning and monitoring.

3.1.3.1 PGD Essential architectures

3.1.3.1.1 Choosing an architecture

There are two supported architectures for PGD Essential. Essential supports the two major use cases for replication: high availability and disaster recovery. The architecture you choose depends on your use case.

They are standard and near/far.

3.1.3.1.2 Standard architecture - Ideal for a highly available single location

The standard, or one-location, architecture is designed for a single location that needs to be highly available. Built around three data nodes, the Essential standard architecture ensures that data is replicated across all three nodes and that, in the event of a failure, the system can continue to operate without data loss.

Learn more about the *What is the PGD Essential Standard architecture* .

3.1.3.1.3 Near/far architecture - Ideal for disaster recovery

The Near/Far architecture is designed for a single location that needs to be reasonably highly available and needs to be able to recover from a disaster. It does this by having a two-data-node cluster in the primary location and a single data node in a secondary location.

Learn more about the *Near/far architecture - Ideal for disaster recovery* .

Standard PGD architecture

Using core PGD capabilites, the standard architecture configures the three nodes in a multi-master replication configuration. That is, each node operates as a master node and logically replicates its data to the other nodes. While PGD is capable of handling conflicts between data changes on nodes, the Essential standard architecture uses PGD's integrated connection manager to ensure that all writes are directed to a single node, the write leader. Conflicts are avoided by allowing that singular leader to handle all updates to the data. Changes are then replicated to the other nodes in the cluster.

If the write leader fails, the remaining nodes in the cluster will elect a new write leader, and the connection managers in those nodes then failover to send writes to the new leader. When the failed node comes back online, it rejoins the cluster and begins replicating data from the new write leader.

The Essential standard architecture was created to be easy to deploy and manage, based on user experience. Unlike other high availability solutions, because Essential is built on PGD, moving to a more complex architecture is simple and straightforward. Move to Expanded PGD, and then add new data groups to the cluster as needed.

See *Manually deploying PGD Essential standard architecture* for more information on how to configure the standard architecture.

Manually deploying PGD Essential standard architecture

Manually deploying the PGD Essential standard architecture is a straightforward process. This architecture is designed for a single location that needs to be highly available and can recover from a disaster. It does this by having three data nodes in a multi-master replication configuration, with one node acting as the write leader.

PGD configuration

Install PGD on each of the three nodes using the instructions in the Essentials install guide. Specifically:

- Configure repositories to enable installation of the PGD packages.
- *Install PGD and Postgres* to install the PGD packages.
- *Configure the PGD cluster* to configure the PGD cluster.

Worked example

This example create a three-node RHEL cluster with EDB Postgres Extended Server, using the PGD Essential Standard architecture and the following parameters:

- The first node is called `node1` and is located on `host-1` .
- The second node is called `node2` and is located on `host-2` .
- The third node is called `node3` and is located on `host-3` .
- the cluster name is `pgd` (the default name).
- The group name is `group1` .
- The Postgres version is `17` .
- The Postgres data directory is `/var/lib/edb-pge/17/main/` .
- The Postgres executable files are in `/usr/edb/pge17/bin/` .
- The Postgres database user is `postgres` .
- The Postgres database port is `5432` .
- The Postgres database name is `pgddb` .

For the first node

This is the common setup for all three nodes, installing the software:

```
export EDB_SUBSCRIPTION_TOKEN=XXXXXXXXXXXXXX
export EDB_SUBSCRIPTION_PLAN=enterprise
export EDB_REPO_TYPE=rpm
curl -1sSLf " https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/$EDB_
↪SUBSCRIPTION_PLAN/setup.$EDB_REPO_TYPE.sh" | sudo -E bash
export PG_VERSION=17
export PGD_EDITION=essential
export EDB_PACKAGES="edb-as$PG_VERSION-server edb-pgd6-$PGD_EDITION-epas$PG_VERSION"
sudo dnf install -y $EDB_PACKAGES
```

On the first node, the following command creates the cluster and the group. It also creates the data directory and initializes the database.

```
sudo su - postgres
export PATH=$PATH:/usr/edb/pge17/bin/
pgd node node1 setup "host=host-1 user=postgres port=5432 dbname=pgddb" --pgdata /var/
↪lib/edb-pge/17/main/ --group-name group1 --cluster-name pgd --create-group --
↪initial-node-count 3
```

For the second node

Repeat the software installation steps on the second node.

Then run the following command to initialize the node and join the cluster and group:

```
sudo su - postgres
export PATH=$PATH:/usr/edb/pge17/bin/
pgd node node2 setup "host=host-2 user=postgres port=5432 dbname=pgddb" --pgdata /var/
↪lib/edb-pge/17/main/ --cluster-dsn "host=host-1 user=postgres port=5432 dbname=pgddb
↪"
```

For the third node

Repeat the software installation steps on the third node.

The command to initialize the node and join the cluster and group is similar to the second node but with a different host and node name:

```
sudo su - postgres
export PATH=$PATH:/usr/edb/pge17/bin/
pgd node node3 setup "host=host-3 user=postgres port=5432 dbname=pgddb" --pgdata /var/
↪lib/edb-pge/17/main/ --cluster-dsn "host=host-1 user=postgres port=5432 dbname=pgddb
↪"
```

Near/far architecture

In the near/far architecture, there are two data nodes in the primary location and one data node in a secondary location. The primary location is where the majority of the data is stored and where most of the client connections are made. The secondary location is used for disaster recovery and isn't used for client connections by default.

The data nodes are all configured in a multi-master replication configuration, just like the standard architecture. The difference is that the node at the secondary location is fenced off from the other nodes in the cluster and doesn't receive client connections by default. In this configuration, the secondary location node has a complete replica of the data in the primary location.

Using a PGD commit scope, the data nodes in the primary location are configured to synchronously replicate data to the other node in the primary location and to the node in the secondary location. This ensures that the data is replicated to all nodes before it's committed to on the primary location. In the case of a node going down, the commit scope rule detects the situation and degrades the replication to asynchronous replication. This behavior allows the system to continue to operate.

In the event of a partial failure at the primary location, the system switches to the other data node, also with a complete replica of the data, and continues to operate. It also continues replication to the secondary location. When the failed node at the primary location comes back, it rejoins and begins replicating data from the node that's currently primary.

In the event of a complete failure in the primary location, the secondary location's database has a complete replica of the data. Depending on the failure, options for recovery include restoring the primary location from the secondary location or restoring the primary location from a backup of the secondary location. The secondary location can be configured to accept client connections, but this isn't the default configuration and requires some additional reconfiguration.

Synchronous replication in near/far architecture

For best results, configure the near/far architecture with synchronous replication. This ensures that the data is replicated to the secondary location before it's committed to the primary location.

See *Manually Deploying PGD Essential near-far architecture* for more information on how to configure the near/far architecture with synchronous replication.

Manually Deploying PGD Essential near-far architecture

The following instructions describe how to manually deploy the PGD Essential near-far architecture. This architecture is designed for a single location that needs to be reasonably highly available and needs to be able to recover from a disaster. It does this by having a two-data-node cluster in the primary location and a single data node in a secondary location.

These instructions use the pgd command line tool to create the cluster and configure the nodes. They assume that you have already installed PGD Essential and have access to the pgd command line tool.

The primary location is referred to as the `active` location and the secondary location as the `dr` location.

PGD configuration

The primary location is configured with two data nodes, in their own group "active". This location is where the majority of the client connections will be made.

The secondary location is configured with one data node, in its own group "dr".

They are all members of the same cluster.

Once created with pgd-cli, the routing and fencing of the nodes needs to be configured.

First, disable the routing on both the "active" and "dr" groups:

```
pgd group dr set-option enable_routing off --dsn "host=localhost port=5432␣
↪dbname=pgddb user=pgdadmin"
pgd group active set-option enable_routing off --dsn "host=localhost port=5432␣
↪dbname=pgddb user=pgdadmin"
```

Then, enable the routing on the "pgd" top-level group:

```
pgd group pgd set-option enable_routing on --dsn "host=localhost port=5432␣
↪dbname=pgddb user=pgdadmin"
```

Finally, enable the fencing on the "dr" group:

```
pgd group dr set-option enable_fencing on --dsn "host=localhost port=5432␣
↪dbname=pgddb user=pgdadmin"
```

This approach ensures that the "dr" group is fenced off from the other nodes in the cluster and doesn't receive client connections by default. The "active" group will continue to operate normally and will continue to replicate data to the "dr" group.

3.1.3.2 Installing and configuring EDB Postgres Distributed 6

This section covers how to manually deploy and configure EDB Postgres Distributed 6.

- *1 - Prerequisites for Essential installation*
- *Step 2 - Configure repositories*
- *Step 3 - Installing the database and pgd*
- *Step 4 - Configuring the cluster*
- *Step 5 - Checking the cluster*

3.1.3.2.1 1 - Prerequisites for Essential installation

This guide takes you through the steps to install EDB Postgres Distributed (PGD) Essential on your systems.

If you want to install a learning/test environment, we recommend using the *PGD First Cluster* .

Note: If you want to install EDB Postgres Distributed (PGD) Expanded, consult the *Expanded installation guide* .

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 | *Quickstart: Create a Linux virtual machine in the Azure portal*

Google | GCP | *Create a Linux VM instance in Compute Engine*

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 serie 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
```

In production environments, you should use DNS to resolve hostnames to IP addresses.

3.1.3.2.2 Step 2 - Configure 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 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_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 commands will download and run a script that configures your package manager to use the EDB repository for databases.

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

This will install the repository for the database software, which includes the EDB Postgres Extended Server and other related packages.

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

This command will download and run a script that configures your package manager to use the EDB repository. It will also install any necessary dependencies.

Worked example

In this example, we will configure the repositories on a CentOS/RHEL system that will allow us to install EDB Postgres Extended Server 17 with PGD Essential with a standard subscription.

Set the environment variables

```
export EDB_SUBSCRIPTION_TOKEN=XXXXXXXXXXXXXXX
export EDB_REPO_TYPE=rpm
curl -1sSLf " https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/enterprise/
↪setup.$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 *Step 3 - Installing the database and pgd* .

3.1.3.2.3 Step 3 - 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 *Step 2 - Configure 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.

```
export EDB_PACKAGES="edb-as$PG_VERSION-server edb-pgd6-essential-epas$PG_VERSION"
```

```
export EDB_PACKAGES="edb-as$PG_VERSION-server edb-pgd6-essential-epas$PG_VERSION"
```

```
export EDB_PACKAGES="edb-postgresextended-$PG_VERSION edb-pgd6-essential-pgextended
↪$PG_VERSION"
```

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

Community PostgreSQL is only operable with PGD 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 Essential on a CentOS/RHEL system using an enterprise subscription using the repository confiuguration we set up in the *Worked example* .

```
export PG_VERSION=17
  export EDB_PACKAGES="edb-postgresextended$PG_VERSION-server edb-postgresextended$PG_
↪VERSION-contrib edb-pgd6-essential-pgextended$PG_VERSION"
  sudo dnf install -y $EDB_PACKAGES
```

The next step is to *Step 4 - Configuring the cluster* .

3.1.3.2.4 Step 4 - 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.

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
```

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
```

Community PostgreSQL is only operable with PGD Expanded.

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 *Step 2 - Configure repositories* and *Step 3 - 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 *Step 5 - Checking the cluster* .

3.1.3.2.5 Step 5 - 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 "host=host-1 port=5432 username=postgres dbname=pgddb"
```

- **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:

```
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):

```
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):

```
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;
 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.

3.1.3.3 Connections

PGD Essential 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.

For PGD Essential, you must connect to the cluster through the connection manager. PGD Essential 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.

3.1.3.3.1 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.

3.1.3.3.2 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
```

3.1.3.4 Using PGD CLI

The PGD CLI is a powerful command line interface for managing your PGD cluster. It can be used to perform a variety of tasks, including:

- Checking the health of the cluster
- Listing the nodes in the cluster
- Listing the groups in the cluster
- Setting group options
- Switching the write leader

If you have used the installation guide to install PGD, you will have already installed PGD CLI and used it to create the cluster.

3.1.3.4.1 Using PGD CLI

The PGD CLI command uses a configuration file to work out the hosts to connect to. There are options that allow you to override this to use alternative configuration files or explicitly point at a server. But, by default, PGD CLI looks for a configuration file in preset locations.

The connection to the database is authenticated in the same way as other command line utilities, like the psql command, are authenticated.

Unlike other commands, PGD CLI doesn't interactively prompt for your password. Therefore, you must pass your password using one of the following methods:

- 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 database clusters, or they compromise password confidentiality.

Configuring and connecting PGD CLI

- Ensure PGD CLI is installed.
- If PGD CLI was already installed, move to the next step.
- For any system, repeat the *Step 2 - Configure repositories* step on that system.
- Then run the package installation command appropriate for that platform.
- RHEL and derivatives: `sudo dnf install edb-pgd6-cli`
- Debian, Ubuntu, and derivatives: `sudo apt-get install edb-pgd6-cli`
- Create a configuration file. - This is a YAML file that specifies the cluster and endpoints for PGD CLI to use.
- Install the configuration file.
- Copy the YAML configuration file to a default config directory `/etc/edb/pgd-cli/` as `pgd-cli-config.yml` .
- Repeat this process on any system where you want to run PGD CLI.
- Run pgd-cli.

Use PGD CLI to explore the cluster

- Check the health of the cluster with the `cluster show --health` command.
- Show the nodes in the cluster with the `nodes list` command.
- Show the groups in the cluster with the `groups list` command.
- Set a group option with the `group set-option` command.
- Switch write leader with the `group set-leader` command.

For more details about these commands, see the worked example that follows.

Also consult the PGD CLI documentation for details of other configuration options and a full command reference.

3.1.3.4.2 Worked example

Ensure PGD CLI is installed

In this worked example, you configure and use PGD CLI on host-1, where you've already installed Postgres and PGD. You don't need to install PGD CLI again.

(Optionally) Create a configuration file

The PGD CLI configuration file is a YAML file that contains a cluster object. This has two properties:

- The name of the PGD cluster's top-level group (as `name`)
- An array of endpoints of databases (as `endpoints`)

```
cluster:
  name: pgd
  endpoints:
    - host=host-1 dbname=pgddb port=5444
    - host=host-2 dbname=pgddb port=5444
    - host=host-3 dbname=pgddb port=5444
```

Note that the endpoints in this example specify `port=5444` . This is necessary for EDB Postgres Advanced Server instances. For EDB Postgres Extended and community PostgreSQL, you can omit this.

Create the PGD CLI configuration directory:

```
sudo mkdir -p /etc/edb/pgd-cli
```

Then, write the configuration to the `pgd-cli-config.yml` file in the `/etc/edb/pgd-cli` directory.

For this example, you can run this on host-1 to create the file:

```
cat <<EOF | sudo tee /etc/edb/pgd-cli/pgd-cli-config.yml
cluster:
  name: pgd
  endpoints:
    - host=host-1 dbname=pgddb port=5444
    - host=host-2 dbname=pgddb port=5444
    - host=host-3 dbname=pgddb port=5444
EOF
```

You can repeat this process on any system where you need to use PGD CLI.

Running PGD CLI

With the configuration file in place, and logged in as the enterprisedb system user, you can run pgd-cli. For example, you can use the `nodes list` command to list the nodes in your cluster and their status:

```
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
```

Using PGD CLI to explore the cluster

Once PGD CLI is configured, you can use it to get PGD-level views of the cluster.

Check the health of the cluster

The *cluster show --health* command provides a quick way to view the health of the cluster:

```
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
```

Show the nodes in the cluster

As previously seen, the nodes list command lists the nodes in the cluster:

```
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 view shows the group the node is a member of and its current status. To find out what versions of PGD and Postgres are running on the nodes, use `nodes list --versions`:

```
pgd nodes list --versions
__OUTPUT__
Node Name  BDR Version                  Postgres Version
```

(continues on next page)

(continued from previous page)

```
- --------- --------------------------  --------------------------------
node-1     5.7.0 (snapshot e2534db6d)   16.6 (Debian 16.6-1EDB.bullseye)
node-2     5.7.0 (snapshot e2534db6d)   16.6 (Debian 16.6-1EDB.bullseye)
node-3     5.7.0 (snapshot e2534db6d)   16.6 (Debian 16.6-1EDB.bullseye)
```

Show the groups in the cluster

Finally, the groups list command for PGD CLI shows which groups are configured, and more:

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

This command shows:

- The groups
- Their types
- Their parent group
- The number of nodes in each group

Set a group option

You can set group options using PGD CLI, too, using the group set-option command. If you wanted to set the `group-1` group's location to `London` , you would run:

```
pgd group group-1 set-option location London
__OUTPUT__
Status Message
- ------ ----------------------------
OK     Command executed successfully
```

You can verity the new location using the group get-option command:

```
pgd group group-1 get-option location
__OUTPUT__
Option Name Option Value
- ---------- ------------
location    London
```

Set the write leader

If you need to change write leader in a group, to enable maintenance on a host, PGD CLI offers the group set-leader command. You enter a group name after `group` and the name of the node you want to switch to after `set leader`:

```
pgd group group-1 set-leader node-2
__OUTPUT__
Status Message
- ------ ----------------------------
OK     Command executed successfully
```

You can verify the write leader using the *group show command* with the `--summary option`:

```
pgd group group-1 show --summary
__OUTPUT__
Group Property    Value
- ---------------- ------------
Group Name        group-1
Parent Group Name pgd
Group Type        data
Write Leader      node-2
Commit Scope
```

More details on the available commands in PGD CLI are available in the PGD CLI command reference .

3.1.3.4.3 PGD Essential constraints

Verify cluster architecture

The `pgd cluster verify` command will display a `Warning` if the cluster does not meet the below listed PGD Essential constraints.

- A PGD Essential cluster must have at most three data nodes.
- A PGD Essential cluster must have routing enabled for global group only.

```
pgd cluster verify --arch
__OUTPUT__
 Check                        | Status  | Groups
- ---------------------------+---------+--------
 Cluster has data nodes       | Ok      |
 Max data nodes in a cluster  | Warning | dc-1
 Witness nodes per group      | Ok      |
 Witness-only groups          | Ok      |
 Data nodes per group         | Ok      |
 Routing enabled groups       | Warning | dc-1
 Empty groups                 | Ok      |
 Nodes have node kind set     | Ok      |
```

PGD compatibility assessment

The `pgd assess` command will not perform the below listed assessments for a Postgres server for PGD Essential compatibility.

- Tables with Multiple Unique Indexes
- Row Level Lock Usage
- Lock Table Usage
- Listen Notify Usage

The command output for a PGD Essential cluster will look like below.

```
pgd assess --dsn "host=pgd-a2 port=5432 dbname=pgddb user=postgres "
__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
```

Commit Scope management commands

The `pgd commit-scope create`, `pgd commit-scope update`, and `pgd commit-scope drop` commands are not supported in a PGD Essential cluster. The command will exit with the following advice:

```
Operation not supported for PGD Essential version.
HINT: This limit doesnt exist in the PGD Expanded version
```

Set group options

The `pgd group set-option` command will not allow update to the group options `enable_raft` , `enable_routing`, `enable_wal_decoder` , and `streaming_mode` for the PGD Essential cluster. The command will exit with below advice.

```
Operation not supported for PGD Essential version.
HINT: This limit doesnt exist in the PGD Expanded version
```

PGD node setup

The `pgd node setup` command will enable routing for `global` group and disable the same for the `sub-group(s)`.

3.1.3.5 Durability in PGD Essential

By default PGD Essential uses asynchronous replication between its nodes, but it can be configured to use synchronous replication as well. This allows for a high degree of flexibility in terms of data durability and availability. Asynchronous replication offers lower latency and higher throughput, while synchronous replication provides stronger consistency guarantees at the cost of performance. PGD Essential allows you to choose the replication strategy through the use of commit scopes.

3.1.3.5.1 Commit Scopes

Commit scopes are a powerful feature of PGD Essential that allow you to control the durability and availability of your data. They enable you to specify the level of durability required for each transaction, allowing you to balance performance and consistency based on your application's needs. PGD Essential has four pre-defined commit scopes that you can use to control the durability of your transactions, among other things.

- local protect
- lag protect
- majority protect
- adaptive protect

The predefined commit scopes in PGD Essential are designed to provide a balance between performance and data safety. You cannot add, remove or modify a PGD Essential commit scope. In PGD Expanded, you can create and manage your own commit scopes, allowing for more flexibility and control over the durability guarantees.

local protect

This 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.

lag protect

This commit scope 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.

majority protect

This 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.

adaptive protect

This 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.

For more information on commit scopes, see the Commit Scopes reference section and the Predefined Commit Scopes reference page.

3.1.3.5.2 Using Commit Scopes

To use commit scopes in PGD Essential, you can specify the desired commit scope when executing a transaction. This allows you to control the durability and availability of your data based on your application's needs. For example, you can use the `lag protect` commit scope for transactions that require a higher level of durability, while using the `local protect` commit scope for transactions that prioritize performance over durability.

Within a transaction

You can specify the commit scope for a transaction using the `SET LOCAL` command. For example, to use the `lag protect` commit scope for a transaction, you can execute the following commands:

```
BEGIN;
SET LOCAL bdr.commit_scope = lag protect;
-- Your transaction statements here
COMMIT;
```

This will ensure that the transaction is committed with the specified commit scope, providing the desired level of durability and availability.

For a session

You can also set the commit scope for the entire session using the `SET` command. For example, to set the `majority protect` commit scope for the entire session, you can execute the following command:

```
SET bdr.commit_scope = majority protect;
```

This will ensure that all transactions executed in the session will use the specified commit scope, providing the desired level of durability and availability.

For a group

You can also set the default commit scope for a PGD group using the `bdr.alter_node_group_option()` function. For example, to set the `adaptive protect` commit scope for a PGD group, you can execute the following command:

```
SELECT bdr.alter_node_group_option(
        node_group_name:=mygroup,
        config_key:=default_commit_scope,
        config_value:=adaptive protect);
```

This will ensure 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.

3.1.3.6 Autopartitioning

Autopartitioning in PGD allows you to split tables into several partitions, other tables, creating and dropping those partitions are needed. Autopartitioning is useful for managing large tables that grow over time as it allows you to separate the data into manageable chunks. For example, you can archive older data into its own partition, and then archive or drop the partition when the data is no longer needed.

3.1.3.6.1 Autopartitioning and replication

PGD autopartitioning is managed, by default, locally through the `bdr.autopartition` function. This function allows you to create or alter the definition of automatic range partitioning for a table. If no definition exists, it creates one; otherwise, it alters the existing definition.

3.1.3.6.2 Range partitioning

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.

For example, you can create a table that is partitioned by date:

```
CREATE TABLE measurement (
    logdate date not null,
    peaktemp int,
    unitsales int
) PARTITION BY RANGE (logdate);
```

Then, you can use the `bdr.autopartition` function to create daily partitions and keep data for one month:

```
select bdr.autopartition(measurement, 1 day, data_retention_period := 30 days);
```

This function creates a partition for each day and retains the data for 30 days. After 30 days, the partitions are automatically dropped. If you look at the database tables you'll see the partitions created for the `measurement` table:

```
pgddb=# \dt
__OUTPUT__
                              List of relations
```

(continues on next page)

(continued from previous page)

```
 Schema |                  Name                   |       Type        |  Owner
- -------+-----------------------------------------+-------------------+----------
 public | measurement                             | partitioned table | postgres
 public | measurement_part_1231354915_2103027132  | table             | postgres
 public | measurement_part_1520219330_1231354915  | table             | postgres
 public | measurement_part_1670975046_3921991865  | table             | postgres
 public | measurement_part_2103027132_2095358725  | table             | postgres
 public | measurement_part_2877346473_1670975046  | table             | postgres
 public | measurement_part_3921991865_1520219330  | table             | postgres
(7 rows)
```

Why are there so many partitions? Because, by default, the autopartition creates five advance partitions, for future use and will automatically do that whenever it uses all but two of the partitions. This means that it will always have at least two partitions available for new data. You can change this behavior by setting the `minimum_advance_partitions` and `maximum_advance_partitions` parameters in the `bdr.autopartition` function.

```
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);
```

3.1.3.7 Production Best Practices

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.

- *Sizing and Scaling PGD Clusters*
- *Time and PGD Clusters*
- Security Best Practices

3.1.3.7.1 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.

3.1.3.7.2 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 Conflict detection .

3.1.3.7.3 Security Best Practices

3.2 Expanded How-to

3.2.1 Overview

PGD Expanded offers the full PGD capability set to users; where PGD Essential is a best practice, controlled and simplified version of PGD. The expanded version is for users who want to take advantage of the full set of features and capabilities of PGD, including advanced architectures, custom configurations, and more complex use cases.

PGD Expanded is designed for users who need the highest level of flexibility and control over their database environments. It provides a comprehensive set of tools and features that allow users to customize their deployments and optimize their performance.

3.2.2 Expanded Features

The following features are enabled in PGD Expanded:

- **Multi-master replication**: PGD Expanded supports multi-master replication, allowing users to create a highly available and fault-tolerant database environment. This feature enables users to write to any node in the cluster, providing maximum flexibility and scalability.
- **Conflict resolution**: PGD Expanded's support for multi-master replication includes advanced conflict resolution capabilities, allowing users to handle conflicts that may arise during replication. This feature ensures that data consistency is maintained across all nodes in the cluster.
- **Advanced durability**: PGD Expanded opens up the full set of durability options in PGD with customizable commit scopes offering flexibility beyond PGD Essentials pre-defined commit scopes. This feature allows users to optimize their database performance and durability based on their specific needs.
- **Custom configurations**: PGD Expanded allows users to customize their database configurations to meet their specific needs. Where PGD Essential supports two basic architectures with limited numbers of nodes and groups, there are no restrictions on the number of nodes, node types, or replication configurations that can be used in a PGD Expanded deployment.

3.2.2.1 PGD Architectures

With PGD 6 Expanded, you can deploy a cluster in a wide range of architectures. Unlike PGD 6 Essential, which is limited to two architectures made with a limited number of groups, PGD 6 Expanded supports multiple architectures with technically unlimited groups, including:

- *Always-On Architecture* : A single PGD cluster with two or more groups in the same data center or availability zone. This architecture is designed for high availability and disaster recovery, ensuring that the database remains operational even if one group fails.
- *PGD Essential architectures* : A single PGD cluster with three nodes in the same data center or availability zone; The PGD 6 Essential architecture.
- *Multi-Location Architectures* : A single PGD cluster with two or more groups in different data centers or availability zones.
- *Geo-Distributed Architectures* : A single PGD cluster with two or more groups in different regions, like a multi-location architecture but with higher latency and potential network partitioning issues.

3.2.2.1.1 Always-On Architecture

PGD's architectures have evolved over time to meet the needs of organizations. At it's core is the Always-on architecture, which is designed to provide high availability and disaster recovery for Postgres databases. Defined in PGD 4 and 5, the Always-on architecture been evolved to support PGD 6's new features and capabilities.

Always-on architectures reflect EDB's Trusted Postgres architectures. They encapsulate practices and help you to achieve the highest possible service availability in multiple configurations. These configurations range from single-location architectures to complex distributed systems that protect from hardware failures and data center failures. The architectures leverage EDB Postgres Distributed's multi-master capability and its ability to achieve 99.999% availability, even during maintenance operations.

You can use EDB Postgres Distributed for architectures beyond the examples described here. Use-case-specific variations have been successfully deployed in production. However, these variations must undergo rigorous architecture review first.

Standard EDB Always-on architectures

EDB has identified a set of standardized architectures to support single- or multi-location deployments with varying levels of redundancy, depending on your recovery point objective (RPO) and recovery time objective (RTO) requirements.

The Always-on architecture uses three database node groups as a basic building block. You can also use a five-node group for extra redundancy.

EDB Postgres Distributed consists of the following major building blocks:

- Bi-Directional Replication (BDR) - A Postgres extension that creates the multi-master mesh network
- Connection Manager - A connection router that makes sure the application is connected to the right data nodes.

All Always-on architectures protect an increasing range of failure situations. For example, a single active location with two data nodes protects against local hardware failure but doesn't provide protection from location (data center or availability zone) failure. Extending that architecture with a backup at a different location ensures some protection in case of the catastrophic loss of a location. However, you still must restore the database from backup first, which might violate RTO requirements. Adding a second active location connected in a multi-master mesh network ensures that service remains available even if a location goes offline. Finally, adding a third location (this can be a witness-only location) allows global Raft functionality to work even if one location goes offline. The global Raft is primarily needed

to run administrative commands. Also, some features like DDL or sequence allocation might not work without it, while DML replication can continue to work even without global Raft.

Each architecture can provide zero RPO, as data can be streamed synchronously to at least one local master, guaranteeing zero data loss in case of local hardware failure.

Increasing the availability guarantee always drives added cost for hardware and licenses, networking requirements, and operational complexity. It's important to carefully consider the availability and compliance requirements before choosing an architecture.

Architecture details

By default, application transactions don't require cluster-wide consensus for DML (selects, inserts, updates, and deletes), allowing for lower latency and better performance. However, for certain operations, such as generating new global sequences or performing distributed DDL, EDB Postgres Distributed requires an odd number of nodes to make decisions using a Raft -based consensus model. Thus, even the simpler architectures always have three nodes, even if not all of them are storing data.

Applications connect to the standard Always-on architectures by way of multi-host connection strings, where each Connection Manager is a distinct entry in the multi-host connection string.

Other connection mechanisms have been successfully deployed in production. However, they aren't part of the standard Always-on architectures.

Always-on Single Location

- Additional replication between data nodes 1 and 3 isn't shown but occurs as part of the replication mesh
- Redundant hardware to quickly restore from local failures * 3 PGD nodes * Can be 3 data nodes (recommended) * Can be 2 data nodes and 1 witness that doesn't hold data (not depicted) * Configuration and infrastructure symmetry of data nodes is expected to ensure proper resources are available to handle application workload when rerouted
- Barman for backup and recovery (not depicted) * Offsite is optional but recommended * Can be shared by multiple PGD clusters
- Postgres Enterprise Manager (PEM) for monitoring (not depicted) * Can be shared by multiple PGD clusters

Always-on Multi-location

- Application can be Active/Active in each location or can be Active/Passive or Active DR with only one location taking writes.
- Additional replication between data nodes 1 and 3 isn't shown but occurs as part of the replication mesh.
- Redundant hardware to quickly restore from local failures. * 6 PGD nodes total, 3 in each location * Can be 3 data nodes (recommended) * Can be 2 data nodes and 1 witness which does not hold data (not depicted) * Configuration and infrastructure symmetry of data nodes and locations is expected to ensure proper resources are available to handle application workload when rerouted
- Barman for backup and recovery (not depicted). * Can be shared by multiple PGD clusters
- Postgres Enterprise Manager (PEM) for monitoring (not depicted). * Can be shared by multiple PGD clusters
- An optional witness node must be placed in a third region to increase tolerance for location failure. * Otherwise, when a location fails, actions requiring global consensus are blocked, such as adding new nodes and distributed DDL.

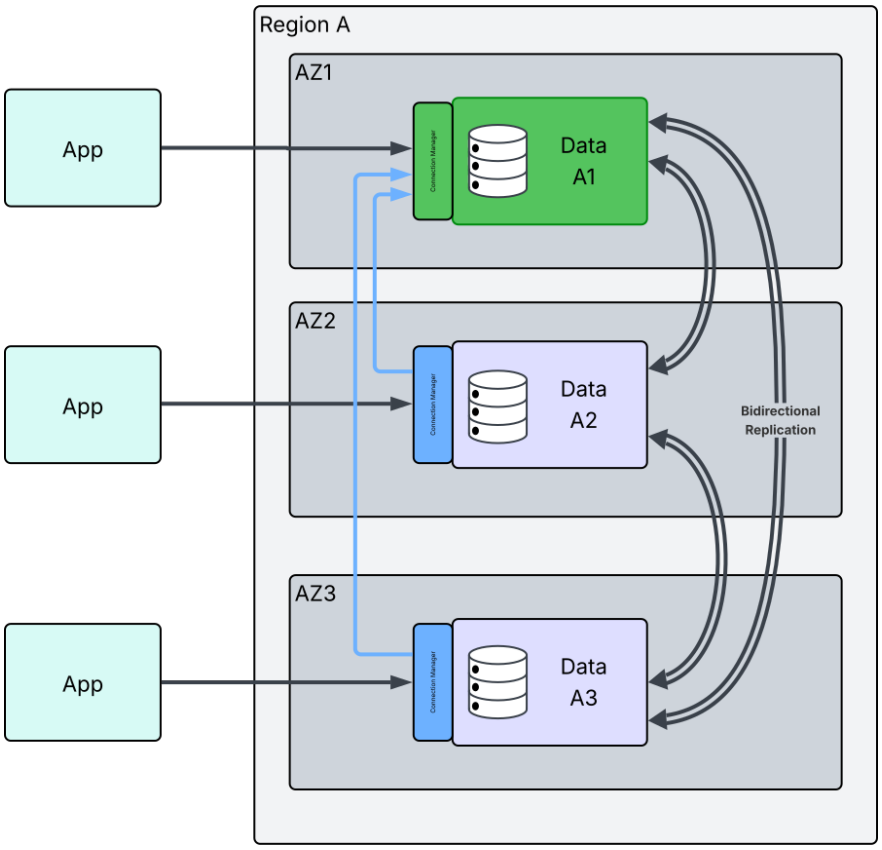

Fig. 1: Always-on 1 Location, 3 Nodes Diagram

Choosing your architecture

All architectures provide the following:

- Hardware failure protection
- Zero downtime upgrades
- Support for availability zones in public/private cloud

Use these criteria to help you to select the appropriate Always-on architecture.

	Single-data location	Two data locations	Two data locations + witness	Three or more data locations
Locations needed	1	2	3	3
Fast restoration of local HA after data node failure	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes
Data protection in case of location failure	No (unless offsite backup)	Yes	Yes	Yes
Global consensus in case of location failure	N/A	No	Yes	Yes
Data restore required after location failure	Yes	No	No	No
Immediate failover in case of location failure	No - requires data restore from backup	Yes - alternate Location	Yes - alternate Location	Yes - alternate Location
Cross-location network traffic	Only if backup is offsite	Full replication traffic	Full replication traffic	Full replication traffic
License cost	2 or 3 PGD data nodes	4 or 6 PGD data nodes	4 or 6 PGD data nodes	6+ PGD data nodes

Adding flexibility to the standard architectures

To provide the data resiliency needed and proximity to applications and to the users maintaining the data, you can deploy the single-location architecture in as many locations as you want. While EDB Postgres Distributed has a variety of conflict-handling approaches available, do take care to minimize the number of expected collisions if allowing write activity from geographically disparate locations.

You can also expand the standard architectures with two additional types of nodes:

- *Subscriber-only nodes*, which you can use to achieve additional read scalability and to have data closer to users when the majority of an application's workload is read intensive with infrequent writes. You can also leverage them to publish a subset of the data for reporting, archiving, and analytic needs.
- *Logical standbys*, which receive replicated data from another node in the PGD cluster but don't participate in the replication mesh or consensus. They contain all the same data as the other PGD data nodes and can quickly be promoted to a master if one of the data nodes fails to return the cluster to full capacity/consensus. You can use them in environments where network traffic between data centers is a concern. Otherwise, three PGD data nodes per location is always preferred.

3.2.2.1.2 Essential Architectures

PGD 6 Expanded supports a wide range of architectures, including the Essential editions standard and near-far architectures.

With Expanded, you can deploy an Essential architecture and then add more groups to it or build out a more complex architecture as your needs grow. The Essential architectures are designed to be simple to deploy and manage, while still providing the core features of PGD.

You can read about the Essential architectures in the Essential How-to .

3.2.2.1.3 Multi-Location Architectures

PGD 6 Expanded inherently supports architectures that span multiple locations, such as data centers or availability zones. This is a key feature of the Expanded edition, allowing you to build robust and resilient distributed databases that can handle failures and maintain high availability across different geographic locations.

3.2.2.1.4 Geo-Distributed Architectures

PGD supports clusters that span multiple geographic, as well as logical, locations. These clusters are known as geo-distributed architectures.

3.2.2.2 Installing and configuring EDB Postgres Distributed 6

This section covers how to manually deploy and configure EDB Postgres Distributed 6.

- *1 - Prerequisites for Essential installation*
- *Step 2 - Configure repositories*
- *Step 3 - Installing the database and pgd*
- *Step 4 - Configuring the cluster*
- *Step 5 - Checking the cluster*

3.2.2.2.1 1 - Prerequisites for Expanded installation

Provisioning hosts

The first step in the process of deploying PGD Expanded 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 | *Quickstart: Create a Linux virtual machine in the Azure portal*

Google | GCP | *Create a Linux VM instance in Compute Engine*

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 serie 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
```

3.2.2.2.2 Step 2 - Configure 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 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 repository/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 *Step 3 - Installing the database and pgd* .

3.2.2.2.3 Step 3 - 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 *Step 2 - Configure 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$PG_VERSION-server edb-pgd6-expanded-epas$PG_VERSION"
```


```
export EDB_PACKAGES="edb-as$PG_VERSION-server edb-pgd6-expanded-epas$PG_VERSION"
```


EDB Postgres Extended

```
export EDB_PACKAGES="edb-postgresextended-$PG_VERSION edb-pgd6-expanded-pgextended$PG_
↪VERSION"
```

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

Community PostgreSQL

```
export EDB_PACKAGES="postgresql-$PG_VERSION edb-pgd6-expanded-pg$PG_VERSION"
```

```
export EDB_PACKAGES="postgresql$PG_VERSION-server postgresql$PG_VERSION-contrib edb-
↪pgd6-expanded-pg$PG_VERSION"
```

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
```

3.2.2.2.4 Step 4 - 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 *Step 2 - Configure repositories* and *Step 3 - 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 *Step 5 - Checking the cluster* .

3.2.2.2.5 Step 5 - 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:

```
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):

```
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):

```
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.

CHAPTER 4

In depth

4.1 PGD concepts explained

4.1.1 PGD concepts

- *Use Case 1: Multi-Master Replication*
- *PGD Nodes and Groups*
- *Connection Management*
- *Locking*
- *Durability in PGD Essential*
- *Commit Scopes*
- *Lag Control*

4.1.2 PGD Expanded concepts

- *Expanded Commit Scopes*
- *Geo-Distributed Clusters*
- *Advanced Durability*
- *Conflict Management*

4.1.2.1 Replication

At the heart of EDB Postgres Distributed (PGD) is the replication system, BDR. BDR stands for Bi-Directional Replication, and it is a multi-master replication system that allows you to create a distributed Postgres cluster with multiple write nodes. This means that you can write to any node in the cluster, and the changes will be replicated to all other nodes in the cluster.

Just because you can write to any node in the cluster, it doesn't mean that you should. In most cases, you will want to write to a single node in the cluster, which is known as the write leader node. This is the node that is responsible for coordinating the replication of changes to all other nodes in the cluster. In fact, in PGD Essential, you can only write to the write leader node, and all other nodes in the cluster are read-only.

There are though some cases where you may want to write to multiple nodes in the cluster, such as when you are using a geo-distributed cluster with multiple write nodes in different locations. In these cases, you can use the BDR replication system to replicate changes between the write nodes. This, and other scenarios, are what PGD Expanded is designed for, and it activates additional features and functionality to support these use cases.

4.1.2.1.1 How Replication Works

PGD uses logical replication to replicate changes between nodes in the cluster. This means that changes are replicated at the logical level, rather than at the physical level. This allows for more flexibility in how changes are replicated, and it also allows for more efficient replication of changes.

When a change is made to a table in the cluster, it is first written to the write leader node's write-ahead log (WAL). The WAL is a log of all changes made to the database, and it is used to ensure durability and consistency of the database. Once the change is written to the WAL, it is then replicated to all other nodes in the cluster.

The replication process is asynchronous by default, which means that changes are not immediately replicated to all nodes in the cluster. Instead, changes are sent to the other nodes in the cluster in batches, which allows for more efficient replication and reduces the load on the network.

Once the changes are received by the other nodes in the cluster, they are applied to the local copy of the database. This process is known as replaying the WAL, and it ensures that all nodes in the cluster have a consistent view of the data.

4.1.2.1.2 Commit scopes and replication

Asynchronous replication is the default mode of replication, but not the only one. PGD allows for definable replication configuration through what are called commit scopes. A commit scope can be applied to a transaction or to all transactions in a group, and it defines how changes are replicated to other nodes in the cluster. This allows you to control how the replication process works, and it can be used to optimize performance, ensure that changes are replicated in a specific way or to handle adverse network and server conditions gracefully.

- PGD Expanded has fully definable commit scopes, which allow you to create custom replication configurations for your cluster. Read about the *commit scopes in PGD Expanded* for full details.
- PGD Essential has has four pre-defined commit scopes that you can use to control how changes are replicated. Read about the *commit scopes in PGD Essential* for full details.

4.1.2.1.3 What is replicated?

In PGD, the following types of changes are replicated:

- **Data changes**: Inserts, updates, and deletes to tables are replicated to all nodes in the cluster. This is called DML (Data Manipulation Language) replication.
- **Schema changes**: Changes to the structure of the database, such as creating or dropping tables, are also replicated to all nodes in the cluster. This is called DDL (Data Definition Language) replication. But not all DDL changes are replicated. For example, adding a column to a table is replicated, but dropping a column is not.
- **Configuration changes**: Changes to the configuration of the database, such as changing the replication settings, are also replicated to all nodes in the cluster.

Currently, PGD only replicates one Postgres database per cluster. This means that if you have multiple databases in your Postgres instance, only the database that is configured for replication will be replicated to the other nodes in the cluster. This is the same for both PGD Essential and PGD Expanded.

4.1.2.2 PGD Nodes and Groups

A PGD cluster is made up of one or more nodes, with each node being an instance of Postgres.

Each node in the cluster is a full Postgres instance with the BDR extension installed and configured. Nodes can have different roles and responsibilities within the cluster. Nodes are then organized into groups, which are used to organize the replication of data between the nodes. There's also the "top level" group, which is the cluster itself; every node in the cluster is also a member of this group, and it is the parent of all other groups in the cluster.

4.1.2.2.1 Data Nodes

The first node kind to know about is the data node. This is the basic building block of PGD clusters. It is configured to replicate data to and from the other data nodes in the cluster. Not the group, the cluster. By design, all nodes in a PGD cluster replicate to all other nodes in the cluster.

4.1.2.2.2 Groups

Groups are used to localize how the nodes **manage** themselves. Each group selects it's own Raft leader from the group members. If the group is a data group that is made up of data nodes it also uses Raft to elect a write leader node for that group. The write leader node will be sent all the read/write client connections for that group and will be the node that handles all write operations for that group, assuming that the client connections come in through the connection manager of a node in that group.

4.1.2.2.3 Witness Nodes

Witness nodes are like data nodes, but they do not replicate or store any data. Their role is to provide a deadlock breaking vote in the event of a group of data nodes losing sufficient nodes as to not be able to complete a majority vote.

Witness nodes do not participate in the normal data replication process, but they can be used to help resolve conflicts and ensure that the cluster remains available even in the face of network partitions or other failures.

4.1.2.2.4 Subscriber-only Nodes

Subscriber-only nodes are used to provide a read-only replica of the data in the cluster. In PGD 6, you can configure a subscriber-only node as a member of a data group or a member of a subscriber-only group. The latter has no write leader node, and all nodes in the group are read-only and allow for some optional optimizations in the replication process. The former allows for a read-only replica of the data in the group, but it does not allow for any optimizations in the replication process.

A subscriber-only node can be used to offload read queries from the write leader node, which can help to improve performance and reduce the load on the write leader node. It can also be used to provide a read-only replica of the data in the cluster for reporting or analytics purposes. You can connect to the read-only nodes like this using the connection manager's read-only connection string, which will direct the connections to the pool of read-only nodes in the cluster.

4.1.2.2.5 Logical Standby Nodes

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.

4.1.2.3 Connection Management

To ensure that clients can connect to the right nodes in the distributed cluster, EDB Postgres Distributed (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.

Read more about the Connection Management feature in PGD for full details of the implementation.

4.1.2.4 Locking

To prevent conflicts between various operations in the cluster, PGD uses a distributed locking mechanism to ensure the only one node can perform a specific operation at a time.

This is particularly important in a distributed environment where multiple nodes may attempt to modify the same data concurrently. As PGD Essential is a single-write node cluster, it does not have to deal with distributed locking in the same way, as there is only one node that can perform write operations at any time. PGD Expaned, however, has multiple write nodes, and so it must will always use distributed locking to ensure integerity.

4.1.2.4.1 Kinds of Locks

PGD uses several kinds of locks to manage concurrent access to data and resources in the cluster:

DDL locking

DDL (Data Definition Language) locks are used to manage access to database objects such as tables, indexes, and schemas. When a DDL operation is performed, a lock is acquired on the object being modified to prevent other operations from interfering with the change. This ensures that the structure of the database remains consistent and prevents conflicts between concurrent DDL operations. Read more about DDL locking in the *DDL Locking reference* documentation.

DML locking

DML locking is closly related to DDL locking and is used to add an extra layer of protection to a DDL operation being replicated by also halting any DML operations that would conflict with the DDL operation. Again, this is only needed in a multi-write node cluster, and is not used in PGD Essential.

Which locks are used when?

The locks used in PGD depend on the type of operation being performed and the configuration of the cluster. In general, DDL locks are used for schema changes, while DML locks are used for data modifications. A full list of the locks used in PGD can be found in the DDL command handling matrix documentation.

4.1.2.5 Durability

How does EDB Postgres Distributed (PGD) ensure durability of transactions?

Durability can be defined as the guarantee that once a transaction has been committed, it will remain so, even in the event of a system failure. In EDB Postgres Distributed (PGD), durability is achieved through a combination of write-ahead logging (WAL) and replication, in combination with the commit scopes available in the cluster and the configuration of the nodes in the cluster.

4.1.2.6 Commit Scopes

4.1.2.7 Lag Control

When a node is lagging behind the rest of the cluster, it can cause issues with data consistency and availability. Lag control is a mechanism to manage this situation by ensuring that the lagging node does not disrupt the overall performance of the cluster.

4.1.2.7.1 Lag Control in PGD

When lag is detected in PGD, the Lag Control feature is activated. This feature is designed to manage the lagging node and ensure that it does not disrupt the overall performance of the cluster. It does this by transparently and temporarily slowing client connections, introducing a commit delay to clients. This allows the lagging node to catch up with the rest of the cluster without impacting the performance of the other nodes.

Read more about the *Lag Control feature in PGD* for full details.

4.1.2.8 Expanded Commit Scopes

PGD Expanded allows you to define commit scopes that are more granular or more customised than the standard pre-defined commit scopes available in PGD. This feature is particularly useful for applications that require specific commit behaviors or need to manage complex transaction scenarios.

4.1.2.9 Geo-Distributed Clusters

Geo-distributed clusters are a powerful feature of PGD which allow you to create a distributed database system that spans multiple geographic locations. This setup is particularly useful for applications that require high availability, low latency, and disaster recovery across different regions. As this feature needs multiple write nodes and multiple distributed groups, it is only available in PGD Expanded.

4.1.2.10 Advanced Durability

4.1.2.11 Conflict Management

With PGD Expanded, the presence of multiple writers leads to the possibility, or even the likelihood, of conflicts. Changes to the same rows from different nodes can arrive on a node at any time. PGD Expanded provides a conflict management system that allows you to define how conflicts are handled in your distributed database environment.

Read more about conflict management in the *Conflict Management reference* documentation.

CHAPTER 5

Reference

5.1 PGD Reference

The PGD Reference section provides detailed information about PGD's features.

- Overview An overview of the features and capabilities of PGD.

5.1.1 Functions and Commands

- *Tables, Views, and Functions*
- Command Line Interface (CLI)

5.1.2 Configuration and Management

- Connection Manager Details about configuring and using Connection Manager, which provides a single endpoint for client connections to a PGD cluster.
- Nodes

Information about the types of nodes that can exist in a PGD cluster, including details of Subscriber-only nodes.

- Node Management How to manage nodes in a PGD cluster, including adding and removing nodes, groups and examining cluster status.
- Postgres Configuration

How Postgres is configured to work in a PGD cluster.

- Backup and Restore

How to back up and restore a PGD cluster.

- Monitoring How to monitor a PGD cluster, including using the built-in monitoring tools and integrating with external monitoring systems.

- Commit Scopes

How to use commit scopes to control transaction visibility in a distributed environment.

- Conflict Management

How to manage conflicts in a distributed environment, including conflict resolution strategies.

- Autopartition

How to use autopartitioning to handle cooling data on nodes in a PGD cluster.

- Testing and Tuning

Guidance on testing and tuning a PGD cluster for optimal performance.

- Upgrades

How to upgrade a PGD cluster to a new version, including considerations for compatibility and data migration.

5.1.3 Advanced Functionality

- Application Usage Guidance for developers wanting to use PGD's advanced functionality in their applications.
- DDL How to use DDL functions in a distributed environment.
- Decoding Worker How to use the decoding worker in replication in PGD.
- CDC Failover Using logical slot failover in PGD for consistent change data capture (CDC).
- Parallel Apply How to use parallel apply for improved performance in distributed transactions.
- Replication Sets
- Security Users, roles, and permissions in PGD.
- Sequences How to use PGD's distributed sequences in your applications.
- Stream Triggers

Triggering actions in response to conflicts and transformations.

- Transaction Streaming

How to use transaction streaming for real-time data replication in PGD.

- Two-Phase Commit

How to opt in to explicity two-phase commit in PGD.

5.1.3.1 PGD 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.

- *Architecture overview*
- *PGD overview - architecture and performance*
- *PGD compared*

5.1.3.1.1 Architecture overview

EDB Postgres Distributed (PGD) is a distributed database solution that extends PostgreSQL's capabilities, enabling highly available and fault-tolerant database deployments across multiple nodes. PGD provides data distribution with advanced conflict management, data-loss protection, high availability up to five 9s, and throughput up to 5X faster than native logical replication.

PGD is built on a multi-master foundation (bi-directional replication, or BDR) which is then optimized for performance and availability through Connection Manager. You can run PGD without Connection Manager if you need a custom deployment better utilizing the multi-master functionality. When running without Connection Manager, writes are distributed among the nodes and replicated to one another, and conflict resolution is relied upon for maintaining consistency. This can be more efficient depending on your architectural needs. However, Connection Manager ensures lower contention and conflict through the use of a write leader. Raft) is implemented to help the system make important decisions, like deciding which node is the Raft election leader and which node is the write leader.

High-level architecture

At the highest level, PGD comprises two main components: Bi-Directional Replication (BDR) and Connection Manager. BDR is a Postgres extension that enables a multi-master replication mesh between different BDR-enabled Postgres instances/nodes. *Connecting through the connection manager* sends requests to the write leader—ensuring a lower risk of conflicts (stronger consistency) between nodes.

Changes are replicated directly, row-by-row between all nodes. Logical replication in PGD is asynchronous by default, so only eventual consistency is guaranteed (within seconds usually). However, commit scope options offer immediate consistency and durability guarantees via CAMO , group and synchronous commits.

The Raft algorithm provides a mechanism for electing leaders (both Raft leader and write leader), deciding which nodes to add or subtract from the cluster. It generally ensures that the distributed system remains consistent and fault tolerant, even in the face of node failures.

Architectural elements

PGD comprises several key architectural elements that work together to provide its distributed database solution:

- **PGD nodes**: These are individual Postgres instances that store and manage data. They are the basic building blocks of a PGD cluster. - **Groups**: By default, all nodes are also members of a *Creating a Commit Scope with the top-level group* with its own Raft leader but without a write leader. PGD nodes can be further organized into *Groups and subgroups* , which enhance manageability and high availability. Each group can contain multiple nodes, allowing for redundancy and failover within the group. Groups facilitate organized replication and data consistency among nodes within the same group and across different groups. Each group has its own write leader.
- **Replication mechanisms**: PGD's replication mechanisms include BDR for efficient replication across nodes, enabling multi-master replication. BDR supports asynchronous replication by default but can be configured for varying levels of synchronicity, such as *Durability options (Group Commit/CAMO)* or *SYNCHRONOUS COMMIT* , to enhance data durability.
- **Monitoring tools**: To monitor performance, health, and usage with PGD, you can use its built-in command-line interface (CLI), which offers several useful commands. For example: - The pgd nodes list command provides a summary of all nodes in the cluster, including their state and status. - The *pgd cluster show --health* command checks the health of the cluster, reporting on node accessibility, replication slot health, and other critical metrics. - The pgd events show command lists significant events like background worker errors and node membership changes, which helps in tracking the operational status and issues within the cluster.

Furthermore, the BDR extension allows for monitoring your cluster using SQL using the *bdr.monitor_group_versions* role.

Node types

All nodes in PGD are effectively data nodes. They vary only in their purpose in the cluster.

- ** Data nodes **: Store and manage data, handle read and write operations, and participate in replication.

There are then three types of nodes which, although built like a data node, have a specific purpose. These are:

- ** Subscriber-only nodes **: Subscribe to changes from data nodes for read-only purposes. Used in reporting or analytics.
- ** Witness nodes **: Participate in the consensus process without storing data, aiding in achieving quorum and maintaining high availability.
- ** Logical standby nodes **: Act as standby nodes that can be promoted to data nodes if needed, ensuring high availability and disaster recovery.

Node roles

Data nodes in a group can also take on particular roles to enable particular features. These roles are transient and can be transferred to any other capable node in the group if needed. These roles can include:

- **Raft leader**: Arbitrates and manages consensus between a group's nodes.
- ** Write leader **: Receives all write operations when applications connect through the connection manager.

Architectural flexibility

PGD offers flexible options with how its architecture can be deployed, maintained, and scaled to meet various performance, availability, and compliance needs.

PGD supports rolling maintenance, including blue/green deployments for both Postgres upgrades and other system or application-level changes. This approach ensures that the database remains available during routine tasks, such as minor or major version upgrades, schema changes, and vacuuming operations. The system seamlessly switches between active database versions, achieving zero downtime.

PGD provides automatic failover to ensure high availability. If a node in the cluster becomes unavailable, another node takes over its responsibilities, minimizing downtime. Also, PGD includes self-healing capabilities, where nodes that have failed or disconnected reconnect to the cluster and resume normal operations once the issue is resolved.

PGD allows for selective replication, enabling users to replicate only a subset of data to specific nodes. This feature can be used to optimize performance by reducing unnecessary data traffic between nodes or to meet regulatory requirements, such as geographical data restrictions. For instance, a healthcare application might only replicate patient data within a specific region to comply with local data privacy laws.

With commit scopes, PGD also provides configurable durability. Accordingly, durability can be increased from the default asynchronous behavior and tuned using various configurable commit scopes:

- ** Synchronous Commit **: Works a lot like PostgreSQL's synchronous_commit option in its underlying operation. Requires writing to at least one other node at COMMIT time but can be tuned to require all nodes.
- ** CAMO ** (Commit At Most Once): Works by tracking each transaction with a unique ID and using a pair of nodes to confirm the transaction's outcome, ensuring the application knows whether to retry the transaction or not.

- ** Group Commit **: An experimental commit scope, the goal of which is to protect against data loss in case of single-node failures of temporary outages by requiring more than one PGD node to successfully confirm a transaction at COMMIT time.
- ** Lag Control **: If replication is running outside of set limits (taking too long for another node to be replicated to), a delay is injected into the node that originally received the transaction, slowing things down until other nodes have caught up.

5.1.3.1.2 PGD overview - architecture and performance

Architectural options and performance

Always-on architectures

A number of different architectures can be configured, each of which has different performance and scalability characteristics.

The group is the basic building block consisting of 2+ nodes (servers). In a group, each node is in a different availability zone, with a dedicated router and backup, giving immediate switchover and high availability. Each group has a dedicated replication set defined on it. If the group loses a node, you can easily repair or replace it by copying an existing node from the group.

The Always-on architectures are built from either one group in a single location or two groups in two separate locations. Each group provides high availability. When two groups are leveraged in remote locations, they together also provide disaster recovery (DR).

Tables are created across both groups, so any change goes to all nodes, not just to nodes in the local group.

One node in each group is selected as the group's write leader. Proxies then direct application writes and queries to the write leader. The other nodes are replicas of the write leader. If, at any point, the write leader is seen to be unavailable, the remaining nodes in the group select a new write leader from the group and the proxies direct traffic to that node. Scalability isn't the goal of this architecture.

Since writes are mainly to only one node, the possibility of contention between nodes is reduced to almost zero. As a result, performance impact is much reduced.

Secondary applications might execute against the shadow nodes, although these are reduced or interrupted if the main application begins using that node.

In the future, one node will be elected as the main replicator to other groups, limiting CPU overhead of replication as the cluster grows and minimizing the bandwidth to other groups.

Supported Postgres database servers

PGD is compatible with PostgreSQL , EDB Postgres Extended Server , and EDB Postgres Advanced Server and is deployed as a standard Postgres extension named BDR. See Compatibility for details about supported version combinations.

Some key PGD features depend on certain core capabilities being available in the target Postgres database server. Therefore, PGD users must also adopt the Postgres database server distribution that's best suited to their business needs. For example, if having the PGD feature Commit At Most Once (CAMO) is mission critical to your use case, don't adopt the community PostgreSQL distribution. It doesn't have the core capability required to handle CAMO.

PGD offers close-to-native Postgres compatibility. However, some access patterns don't necessarily work as well in multi-node setup as they do on a single instance. There are also some limitations in what you can safely replicate in

a multi-node setting. Application usage goes into detail about how PGD behaves from an application development perspective.

Characteristics affecting performance

By default, PGD keeps one copy of each table on each node in the group, and any changes propagate to all nodes in the group.

Since copies of data are everywhere, SELECTs need only ever access the local node. On a read-only cluster, performance on any one node isn't affected by the number of nodes and is immune to replication conflicts on other nodes caused by long-running SELECT queries. Thus, adding nodes increases linearly the total possible SELECT throughput.

If an INSERT, UPDATE, and DELETE (DML) is performed locally, then the changes propagate to all nodes in the group. The overhead of DML apply is less than the original execution. So if you run a pure write workload on multiple nodes concurrently, a multi-node cluster can handle more TPS than a single node.

Conflict handling has a cost that acts to reduce the throughput. The throughput then depends on how much contention the application displays in practice. Applications with very low contention perform better than a single node. Applications with high contention can perform worse than a single node. These results are consistent with any multimaster technology and aren't particular to PGD.

Synchronous replication options can send changes concurrently to multiple nodes so that the replication lag is minimized. Adding more nodes means using more CPU for replication, so peak TPS reduces slightly as each node is added.

If the workload tries to use all CPU resources, then this resource constrains replication, which can then affect the replication lag.

In summary, adding more master nodes to a PGD group doesn't result in significant write throughput increase when most tables are replicated because all the writes are replayed on all nodes. Because PGD writes are in general more effective than writes coming from Postgres clients by way of SQL, you can increase performance. Read throughput generally scales linearly with the number of nodes.

5.1.3.1.3 PGD compared

The following table compares EDB Postgres Distributed with other replication solutions.

	PGD Standalone	PGD Managed	PSR + EFM	pglogical 2	PG Builtin Logical Replication
Version at last update (2025-06-19)	6.x	6.x	4.10	2.4.5	17
Deployment					
On Premise	Yes	Yes	Yes	Yes	Yes
Multi-cloud	Yes	Yes	Yes	Yes	Yes
Hybrid (on-prem + cloud)	Yes	Coming Soon	Yes	Yes	Yes
SLA	99.999	99.995	99.99	N/A	N/A
Performance					
Read Scalability	Yes	No	Yes - Physical standbys	Yes - More nodes	Yes - More nodes
Horizontal Scalability	No	No	No	No	No

continues on next page

Table 1 – continued from previous page

	PGD Standalone	PGD Managed	PSR + EFM	pglogical 2	PG Builtin Logical Replication
Transaction Streaming	Yes	Yes	Yes	No	Yes
Parallel Apply (Vertical Scalability)	Yes	Yes	No	No	Partial - for large transactions only
Durability					
Asynchronous Replication	Yes	Yes	Yes	Yes	Yes
Optional RPO limit for asynchronous replication	Yes	Yes	No	No	No
Synchronous Replication	Yes	Yes	Yes	Yes	Yes
Consensus based replication	Yes	Yes	No	No	No
Per transaction durability setting	Yes	Yes	Yes	Yes	Yes
Consistenct					
Automatic conflict management	Yes	Yes	N/A	Yes	No
Conflict avoidance types	Yes	Yes	N/A	No	No
Conflict avoidance at commit (pessimistic conflict handling)	Yes	Yes	N/A	No	No
Builtin distributed sequence	Yes - snowflake (bigint) - galloc (int/bigint)	Yes - snowflake (bigint) - galloc (int/bigint)	N/A	No	No
Data Distibution					
Data residency/selective replication	Yes	Yes	No	Yes	Yes
Cluster level Active-Active (writers in different regions)	Yes	Yes	No	Manual setup	No
Regional Active-Active (multiple writers within region)	Yes - not recommended unless specific setup	No	No	Manual setup	No
Automatic partitioning	Yes	Yes	Yes with EPAS	No	No
Offload cold data to cheaper storage	Yes	Yes	No	No	No
Maintenance					
Near-zero downtime major version upgrades by adding nodes	Yes	Yes	Yes - using logical replication	Yes	Yes
Near-zero downtime inplace major version upgrades	Yes	Yes	No	No	No

continues on next page

Table 1 – continued from previous page

	PGD Standalone	PGD Managed	PSR + EFM	pglogical 2	PG Builtin Logical Replication
Rolling schema upgrades/green-blue (with application assistance)	Yes	Yes	No	No	No
Rolling maintenance operations	Yes	Yes	No	Yes	Yes
Connection Mgmt					
Automatic failover	Yes	Yes	Yes	N/A	N/A
Automatic connection failover for switchover	Yes	Yes	Yes	N/A	N/A
Cluster level connection routing	Yes	Yes	Yes	N/A	N/A
Region level connection routing	Yes	Yes	No	N/A	N/A
pgbouncer support	Yes	No	Yes	N/A	N/A
DDL Support					
General DDL replication	Yes	Yes	Yes	Manual	No
Granular (per-object) DDL locking	Yes	Yes	Yes	No	No
Create and drop objects	Yes	Yes	Yes	Manual	No
Add columns to table	Yes	Yes	Yes	Unsafe/manual	No
Change column type	Yes - rewrite requires permit_unsafe_commands	Yes - rewrite requires permit_unsafe_commands	Yes	Unsafe/manual	No
CREATE TABLE AS	Yes - with restrictions	Yes - with restrictions	Yes	Unsafe/manual	No
PG Compatibility					
Latest supported version	17	17	17	17	17
Works on standard PG	Yes	No	Yes	Yes	Yes
Supports TDE	Yes with EPAS/PGE	Yes with EPAS/PGE	Yes with EPAS/PGE	No	Yes with EPAS/PGE
Supports custom types (i.e. Postgis)	Yes	Yes	Yes	Yes	Yes
Supports extensions	Many	Many	All	Many	Many
CDC failover support	No	No	Yes	N/A	N/A
Large Object support	No	No	Yes	No	No
Multiple DB support	No	No	Yes	N/A	N/A
Management					
CLI	Yes	Yes	Yes	No	No
GUI	PEM	Yes	PEM	No	No
Monitoring options	• SQL - CLI - PEM	• SQL - CLI - UPM	• SQL - CLI - PEM	SQL	SQL

continues on next page

Table 1 – continued from previous page

	PGD Standalone	PGD Managed	PSR + EFM	pglogical 2	PG Builtin Logical Replication
Licensing					
Source available	No	No	No	Yes	Yes
Open source	No	No	No	Yes	Yes

5.1.3.2 Advanced functionality

5.1.3.2.1 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.

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` , `REINDEX`)
- Actions of autovacuum
- Operational commands (`CHECKPOINT` , `ALTER SYSTEM`)
- Actions related to databases or tablespaces

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.

Prevent the execution of conflicting DDL, otherwise DDL replication causes errors and the replication stops.

PGD offers three levels of protection against those problems:

`ddl_locking = 'all'` is the strictest option and is best when DDL might execute from any node concurrently and you want to ensure correctness. This is the default.

`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.

`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.

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.

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 = dml` skips global DDL lock and, if needed, takes relation DML lock.
- `ddl_locking = leader` enables leader-based global DML locking.
- `ddl_locking = auto` current behaves like `ddl_locking = leader` .
- `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* .

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.

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	Y	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	*Architecture 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

continues on next page

Table 2 – continued from previous page

Command	Allowed	Replicated	Lock
ALTER ROLE	Y	Y	DDL
ALTER ROUTINE	Y	Y	DDL
ALTER RULE	Y	Y	DDL
ALTER SCHEMA	Y	Y	DDL
ALTER SEQUENCE	*Architecture 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	*Architecture details*	Y	*Architecture 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	*Architecture 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

continues on next page

Table 2 – continued from previous page

Command	Allowed	Replicated	Lock
CREATE EVENT TRIGGER	Y	Y	DDL
CREATE EXTENSION	Y	Y	DDL
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	Y	DDL
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	*Architecture 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	*Architecture 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	*Architecture 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

continues on next page

Table 2 – continued from previous page

Command	Allowed	Replicated	Lock
CREATE TEXT SEARCH TEMPLATE	Y	Y	DDL
CREATE TRANSFORM	Y	Y	DDL
CREATE TRIGGER	Y	Y	DDL
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	Y	DDL
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

continues on next page

Table 2 – continued from previous page

Command	Allowed	Replicated	Lock
DROP PUBLICATION	Y	Y	DDL
DROP QUEUE	Y	Y	DDL
DROP QUEUE TABLE	Y	Y	DDL
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	*Architecture details*	*Architecture details*
FETCH	Y	N	N
GRANT	Y	*Architecture 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	*Architecture details*
MOVE	Y	N	N
NOTIFY	Y	N	N
PREPARE	Y	N	N
PREPARE TRANSACTION	Y	N	N
REASSIGN OWNED	Y	Y	DDL

continues on next page

Table 2 – continued from previous page

Command	Allowed	Replicated	Lock
REFRESH MATERIALIZED VIEW	Y	Y	DML
REINDEX	Y	N	N
RELEASE	Y	N	N
RESET	Y	N	N
REVOKE	Y	*Architecture details*	DDL
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	*Architecture details*	DDL
SELECT INTO	*Architecture details*	Y	DDL
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	*Architecture details*	*Architecture details*
UNLISTEN	Y	N	N
VACUUM	Y	N	N

Command notes

ALTER SEQUENCE

`ALTER SEQUENCE ... RENAME` and `ALTER SEQUENCE ... SET SCHEMA` are only fully supported for galloc sequences if `sequuid` (UUID) is present in the `bdr.sequence_kind` catalog.

If the required UUID is missing for a galloc sequence, the rename operation may fail. This typically occurs in clusters that have been upgraded from PGD versions prior to {{version.full}}.

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.

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`.

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.

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.

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
- bdr.alter_replication_set
- bdr.drop_replication_set
- bdr.replication_set_add_table
- bdr.replication_set_remove_table
- bdr.replication_set_add_ddl_filter
- bdr.replication_set_remove_ddl_filter

Conflict management:

- bdr.alter_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

Stream triggers:

- bdr.create_conflict_trigger
- bdr.create_transform_trigger
- bdr.drop_trigger

5.1.3.2.2 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.

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.

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 .

5.1.3.2.3 CDC Failover support

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.

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.

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.

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);
```

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.

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.

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]:
```

(continues on next page)

(continued from previous page)

```
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_
↪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.

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.

5.1.3.2.4 Parallel Apply

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. It isn't compatible with Group Commit or Eager Replication, so disable it if Group Commit or Eager Replication are in use.

5.1.3.2.5 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.

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.

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.

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 ...
```

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 .

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)
```

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.

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
```

(continues on next page)

(continued from previous page)

```
  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)
```

5.1.3.2.6 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.

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.
pg_upgrade	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.

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.

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
- bdr.autopartition_rules
- bdr.ddl_epoch
- bdr.ddl_replication
- bdr.global_consensus_journal_details

- bdr.global_lock
- bdr.global_locks
- bdr.group_camo_details
- bdr.local_consensus_state
- bdr.local_node_summary
- bdr.node
- bdr.node_catchup_info
- bdr.node_catchup_info_details
- bdr.node_conflict_resolvers
- bdr.node_group
- bdr.node_local_info
- bdr.node_peer_progress
- bdr.node_replication_rates
- bdr.node_slots
- bdr.node_summary
- bdr.replication_sets
- `bdr.replication_status`
- bdr.sequences
- bdr.stat_activity
- bdr.stat_relation
- bdr.stat_subscription _deprecated_
- *bdr.state_journal_details*
- bdr.subscription
- bdr.subscription_summary
- bdr.tables
- bdr.taskmgr_local_work_queue
- bdr.taskmgr_work_queue
- *bdr.worker_errors* _deprecated_
- bdr.workers
- bdr.writers
- *bdr.xid_peer_progress*

EXECUTE privilege on:

- *bdr.bdr_edition* _deprecated_
- bdr.bdr_version
- bdr.bdr_version_num
- bdr.decode_message_payload

- bdr.get_consensus_status
- bdr.get_decoding_worker_stat
- bdr.get_global_locks
- bdr.get_min_required_replication_slots
- bdr.get_min_required_worker_processes
- bdr.get_raft_status
- bdr.get_relation_stats
- bdr.get_slot_flush_timestamp
- *bdr.get_sub_progress_timestamp*
- bdr.get_subscription_stats
- bdr.lag_control
- bdr.lag_history
- bdr.node_catchup_state_name
- bdr.node_kind_name
- bdr.peer_state_name
- bdr.show_subscription_status
- bdr.show_workers
- bdr.show_writers
- bdr.stat_get_activity
- bdr.wal_sender_stats
- bdr.worker_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
- bdr.group_replslots_details
- bdr.group_subscription_summary
- bdr.group_versions_details
- *bdr.raft_instances*

`EXECUTE` privilege on:

- bdr.get_raft_instance_by_nodegroup

- bdr.monitor_camo_on_all_nodes
- bdr.monitor_group_raft
- bdr.monitor_group_versions
- bdr.monitor_local_replslots
- bdr.monitor_raft_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

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
- bdr.create_conflict_trigger
- bdr.create_transform_trigger
- bdr.drop_trigger
- bdr.get_configured_camo_partner
- bdr.global_lock_table
- bdr.is_camo_partner_connected
- bdr.is_camo_partner_ready
- bdr.logical_transaction_status
- `bdr.ri_fkey_trigger`
- bdr.seq_nextval
- bdr.seq_currval
- bdr.seq_lastval
- bdr.trigger_get_committs
- bdr.trigger_get_conflict_type
- bdr.trigger_get_origin_node_id
- bdr.trigger_get_row
- bdr.trigger_get_type

- bdr.trigger_get_xid
- bdr.wait_for_camo_partner_queue
- bdr.wait_slot_confirm_lsn
- bdr.wait_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.

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.

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.

5.1.3.2.7 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.

PGD global sequences

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.

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.

A few limitations apply to `galloc` sequences. PGD tracks `galloc` sequences in a special PGD catalog bdr.sequence_alloc . This catalog is required to track the currently allocated chunks for the `galloc` sequences. The sequence name and namespace is stored in this catalog. The sequence chunk allocation is managed by Raft, whereas any changes to the sequence name/namespace is managed by the replication stream. So PGD currently doesn't support renaming `galloc` sequences or moving them to another namespace or renaming the namespace that contains a `galloc` sequence. Be mindful of this limitation while designing application schema.

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
```

(continues on next page)

(continued from previous page)

```
- ------------------------

(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;
```

(continues on next page)

(continued from previous page)

```
          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
- ------------+-----------
         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.

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.

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 pseudorandom 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.

See also

- Global Sequence management interfaces
- KSUUID v2 functions
- KSUUID v1 functions

5.1.3.2.8 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.

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.

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.

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.

Warning:

- You can specify multiple conflict triggers on a single table, but they must match a distinct event. That is, each conflict must match only a single conflict trigger. - We don't recommend multiple triggers matching the same event on the same table. They might result in inconsistent behavior and will not be allowed in a future release.

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.

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.

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 .

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.

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
```

(continues on next page)

(continued from previous page)

```
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()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE  record;
    TARGET  record;
```

(continues on next page)

(continued from previous page)

```
    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;
$$;
```

5.1.3.2.9 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.

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.)

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`

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.

5.1.3.2.10 Explicit two-phase commit (2PC)

Note: Two-phase commit isn't available with Group Commit or CAMO. See *Commit scope limitations* .

An application can explicitly opt to use two-phase commit with PGD. See Distributed Transaction Processing: The XA Specification .

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 either CAMO or the global commit scope. Future releases might enable this combination.

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;
```

5.1.3.3 Config and management

5.1.3.3.1 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.

- Overview is an overview the kinds of node that can exist in PGD clusters and their associated roles.
- Witness nodes looks at the witness node, a special class of PGD node, dedicated to establishing consensus in a group.
- 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.
- 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.

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.

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.

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 *Statement-specific DDL replication concerns* . 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.

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.
```

5.1.3.3.2 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.
- *Connection DSNs and SSL (TLS)* introduces the DSNs or connection strings needed to connect directly to a node in a PGD cluster. It also covers how to use SSL/TLS certificates to provide authentication and encryption between servers and between clients.
- *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.
- *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.

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 *Step 3 - 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 `postgres_distributed` repository to your package management system on Linux and then install the `edb-bdr` package. You can find the repository configuration instructions in the *PGD manual installation guide* .

Once the repository is configured, you can install the BDR package with your package manager. The package name is `edb-pgd6-<postgresversion>` where `<postgresversion>` is the version of Postgres you are using. For example, if you are using Postgres 14, the package name is `edb-pgd6-14` .

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* . .. Note The *max_worker_processes* value:

  The *max_worker_processes* value is derived from the topology of the cluster, the number of peers, number of databases, and other factors. To calculate the needed value, see *Postgres configuration/settings* . The value of 16 was calculated for the size of cluster being deployed in this example. It must be increased for larger clusters. .. Note The *max_worker_processes* value:: - Set a password on the EnterprisedDB/Postgres user.
- 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.

Initializing a PGD node

Log into the database instance you have configured and set up the BDR extension. You can do this by running the `CREATE EXTENSION bdr;` command as super user in the database. This command creates the BDR extension.

You also need to 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` .

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 *Joining a node to a group* section.

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.

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 join 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.

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
```

(continues on next page)

(continued from previous page)

```
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;
```

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.

Connection DSNs and SSL (TLS)

Because nodes connect using `libpq`, the DSN of a node is a *libpq* connection string. As such, the connection string can contain any permitted `libpq` connection parameter, including those for SSL. The DSN must work as the connection string from the client connecting to the node in which it's specified. An example of such a set of parameters using a client certificate is:

```
sslmode=verify-full sslcert=bdr_client.crt sslkey=bdr_client.key
sslrootcert=root.crt
```

With this setup, the files `bdr_client.crt`, `bdr_client.key`, and `root.crt` must be present in the data directory on each node, with the appropriate permissions. For `verify-full` mode, the server's SSL certificate is checked to ensure that it's directly or indirectly signed with the `root.crt` certificate authority and that the host name or address used in the connection matches the contents of the certificate. In the case of a name, this can match a subject's alternative name or, if there are no such names in the certificate, the subject's common name (CN) field. Postgres doesn't currently support subject alternative names for IP addresses, so if the connection is made by address rather than name, it must match the CN field.

The CN of the client certificate must be the name of the user making the PGD connection, which is usually the user postgres. Each node requires matching lines permitting the connection in the `pg_hba.conf` file. For example:

```
hostssl all         postgres 10.1.2.3/24 cert
hostssl replication postgres 10.1.2.3/24 cert
```

Another setup might be to use `SCRAM-SHA-256` passwords instead of client certificates and not verify the server identity as long as the certificate is properly signed. Here the DSN parameters might be:

```
sslmode=verify-ca sslrootcert=root.crt
```

The corresponding `pg_hba.conf` lines are:

```
hostssl all         postgres 10.1.2.3/24 scram-sha-256
hostssl replication postgres 10.1.2.3/24 scram-sha-256
```

In such a scenario, the postgres user needs a *.pgpass* file containing the correct password.

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 Origin conflict detection).

Administrators must monitor for node outages (see Monitoring) 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.

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 3 times bdr.replay_progress_frequency , 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.

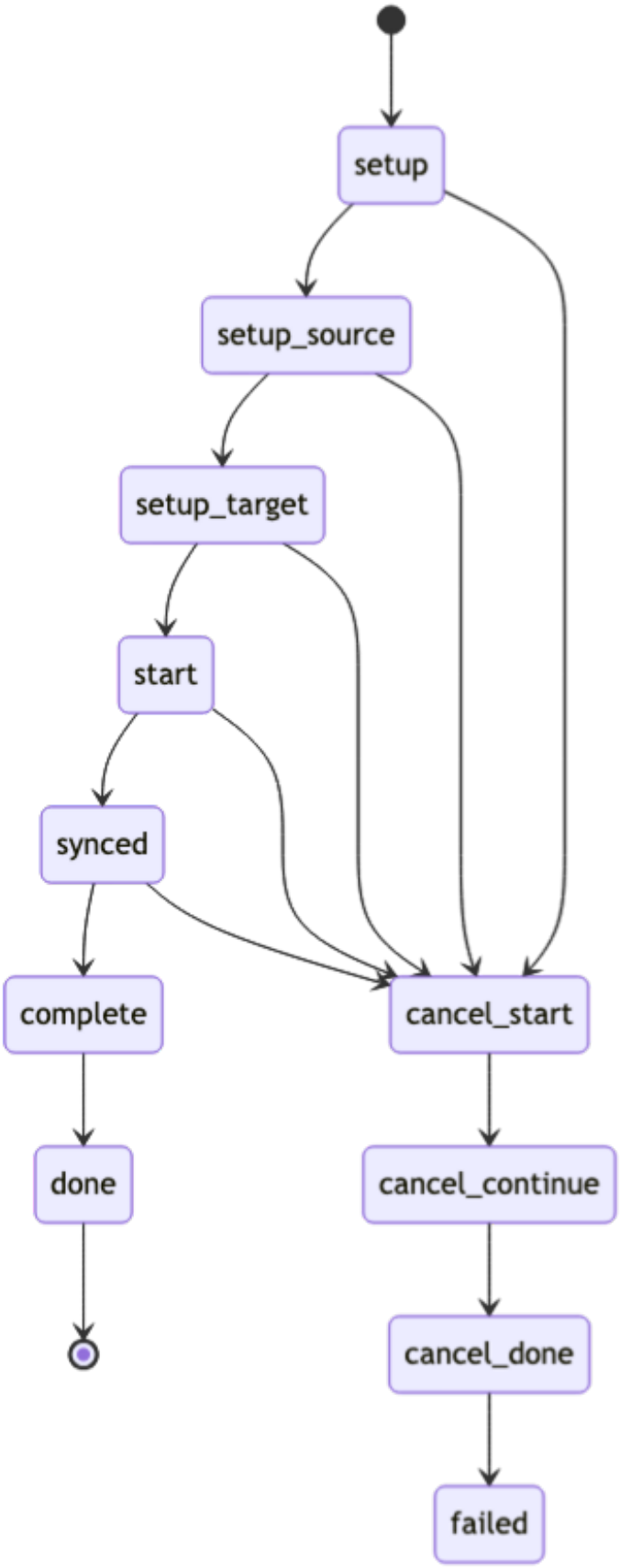

Fig. 1: Autosync Request Life Cycle

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.

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.

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.

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>*

5.1.3.3.3 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 new features and benefits of the Connection Manager.
- *Connection Manager Authentication* covers how authentication works with the Connection Manager.
- *Configuring Connection Manager* details the configuration options available and how to set them.
- *Load Balancing with Connection Manager* how to use load balancing with the Connection Manager.
- *Monitoring the Connection Manager* covers the tables and HTTP endpoints available for monitoring.

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.

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, you need to configure your client applications to connect to the read-write or read-only port of the Connection Manager instance running on the data node. The Connection Manager will then route the connection to the appropriate node in the cluster.

Note that the Connection Manager is not a replacement for a load balancer. It is designed to work in conjunction with a load balancer to provide a complete solution for managing connections to PGD clusters. The Connection Manager provides a simple and efficient way to manage connections to PGD clusters, while the load balancer provides additional features such as load balancing and failover. See *Load Balancing with Connection Manager* for more information.

Read-Only connections

Connecting a client to the read-only port provided by connection manager restricts that connection to read-only operations in a similar way to using *SET TRANSACTION READ ONLY* would, except that it's not possible to change it to read-write. The *transaction_read_only* GUC correctly reports `on` in these connections.

TLS and Authentication

The Connection Manager performs TLS termination and pre-authentication. The configuration for these is taken directly from Postgres - pg_hba.conf and server key configuration are used transparently. See *Connection Manager Authentication* for more information.

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.

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.

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 bdr.alter_node_group_option command, or using the pgd group set-option command.

The following options are available for configuring Connection Manager:

Option	Default	Description
listen_address	Postgres's listen address	which local addresses it should listen on for client connections
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		whether http listener should use HTTPS, if enabled, the server certificate is used to TLS
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.

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 checkuser 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 checkuser 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
    bind *:5011
```

(continues on next page)

(continued from previous page)

```
    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.

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 is ready, 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)node

continues on next page

Table 7 – continued from previous page

Endpoint	Description
/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.

5.1.3.3.4 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* .

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. One exception to that is `pgaudit` , which must appear in the list before `bdr` . Also, 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 .

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.

5.1.3.3.5 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

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
```

(continues on next page)

(continued from previous page)

```
#  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

#  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.

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

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.

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.

5.1.3.3.6 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.

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
```

(continues on next page)

(continued from previous page)

```
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 for monitoring the state of the connection routing with Connection Manager uses to route the connections.
- bdr.stat_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 for monitoring outgoing replication
- bdr.subscription_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 in Node Management.

Monitoring outgoing replication

You can use another view for monitoring of outgoing replication activity:

- bdr.node_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
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 ]---+----------------------------
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 .

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 for cumulative statistics for each commit scope.
- bdr.stat_commit_scope_state for information about the current use of commit scopes by backend processes.

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 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
```

(continues on next page)

(continued from previous page)

```
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
- ----------+------------------+-------------
```

(continues on next page)

(continued from previous page)

```
 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 optimized topology 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.

5.1.3.3.7 AutoPartition in PGD

PGD AutoPartition allows you to split tables into several partitions. It lets tables grow easily to large sizes using automatic partitioning management. This capability uses features of PGD, such as low-conflict locking of creating and dropping partitions.

You can create new partitions regularly and then drop them when the data retention period expires.

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.

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.

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
    );
```

RANGE-partitioned tables

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.

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.

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.

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.

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.

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.

5.1.3.3.8 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.

Introducing

- Overview introduces the concepts and some of the essential terminology that's used when discussing synchronous commits.
- Durability terminology lists terms used around PGD's durability options, including how to refer to nodes in replication.
- Commit scopes is a more in-depth look at the structure of commit scopes and how to define them for your needs.
- Predefined commit scopes lists the pre-defined commit scopes that are available in PGD Essential.
- Origin groups introduces the notion of an origin group, and how to leverage these when defining commit scopes rules.
- Commit scope rules looks at the syntax of and how to formulate a commit scope rule.
- Comparing durability options compares how commit scope options behave with regard to durability.
- 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.

Commit scope kinds

- Synchronous Commit is a commit scope mechanism that works in a similar fashion to legacy synchronous replication, but from within the commit scope framework.
- Group Commit focuses on the Group Commit option, where you can define a transaction as done when a group of nodes agrees it's done.
- CAMO 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.
- 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.

Working with commit scopes

- Administering addresses how to manage a PGD cluster with Group Commit in use.
- Legacy synchronous replication shows how you can still access traditional Postgres synchronous operations under PGD.
- 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:

- *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.
- *Durability options (Group Commit/CAMO)* : This kind of commit scope controls which and how many nodes have to reach a consensus before the transaction is considered to be committable and at what stage of replication it can be considered committed. This option also allows you to control the visibility ordering of the transaction.
- *Commit At Most Once* : This kind of commit scope is a variant of Group Commit, in which the client takes on the responsibility for verifying that a transaction was committed before retrying.
- *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, group 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 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* , *Durability options (Group Commit/CAMO)* , *Commit At Most Once* , *Lag Control* , or 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 (example_bdr_group) GROUP 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 (example_bdr_group)` - followed by the commit scope kind itself - `GROUP COMMIT` . This translates to requiring that any two nodes in `example_bdr_group` must confirm the change before the change can be considered as comitted.

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 (example_bdr_group) GROUP COMMIT,
    wait_for_ready := true
);
```

This will add the rule `MAJORITY (example_bdr_group) GROUP 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,
  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.

Origin groups

Rules for commit scopes can depend on the node the transaction is committed on, that is, the node that acts as the origin for the transaction. The bottom group of the group tree to which that node belongs is the transaction's *origin group*. To make this transparent for the application, PGD allows a commit scope to define different rules depending on the transaction's origin group.

For example, consider an EDB Postgres Distributed cluster with nodes spread across two data centers: a left (`left_dc`) and a right one (`right_dc`). Assume the top-level PGD node group is called `top_group` . You can use the following commands to set up subgroups and create a commit scope requiring all nodes in the local data center to confirm the transaction but only one node from the remote one:

```
- - create sub-groups
SELECT bdr.create_node_group(
    node_group_name := left_dc,
```

(continues on next page)

(continued from previous page)

```
    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
);

- - create a commit scope with individual rules
- - for each sub-group
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
);
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) GROUP COMMIT AND ALL (right_dc) GROUP COMMIT (commit_
↪decision=raft),
    wait_for_ready := true
);
```

Now, using the `example_scope` on any node that's part of `left_dc` uses the first scope. Using the same scope on a node that's part of `right_dc` uses the second scope. By combining the `left_dc` and `right_dc` origin rules under one commit scope name, an application can simply use `example_scope` on either data center and get the appropriate behavior for that data center.

Each group can also have a default commit scope specified using the `bdr.alter_node_group_option` admin interface.

Making the above scopes the default ones for all transactions originating on nodes in those groups looks like this:

```
SELECT bdr.alter_node_group_option(
  node_group_name := left_dc,
  config_key := default_commit_scope,
  config_value := example_scope
);
SELECT bdr.alter_node_group_option(
  node_group_name := right_dc,
  config_key := default_commit_scope,
  config_value := example_scope
);
```

ORIGIN_GROUP

You can also refer to the origin group of a transaction dynamically when creating a commit scope rule by using `ORIGIN_GROUP` .

This can make certain commit scopes rules like those above in `example_scope` , even easier to specify in that you can simply specify one rule instead of two.

For example, again suppose that for transactions originating from nodes in `right_dc` you want all nodes in `right_dc` to confirm and any 1 from `left_dc` to confirm before the transaction is committed. Also, again suppose that for transactions originating in `left_dc` you want all nodes in `left_dc` and any 1 in `right_dc` to confirm before the transaction is commited. Above we used these two rules for this when defining `example_scope` :

```
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
);
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) GROUP COMMIT AND ALL (right_dc) GROUP COMMIT (commit_
↪decision=raft),
    wait_for_ready := true
);
```

However, with `ORIGIN_GROUP` , just adding and using the following single-rule commit scope, `example_scope_2` , will have the same effect as the two individual rules we used above in `example_scope` :

```
SELECT bdr.create_commit_scope(
    commit_scope_name := example_scope_2,
    origin_node_group := top_group,
    rule := ALL ORIGIN_GROUP GROUP COMMIT (commit_decision=raft) AND ANY 1 NOT ORIGIN_
↪GROUP GROUP COMMIT;
    wait_for_ready := true
);
```

Under `example_scope_2` , when a transaction originates from `left_dc` , `ORIGIN_GROUP` maps to `left_dc` and `NOT ORIGIN_GROUP` maps to `right_dc` . Likewise, when a transaction originates from `right_dc` , `ORIGIN_GROUP` maps to `right_dc` and `NOT ORIGIN_GROUP` maps to `left_dc` . So by only specifying one rule, you get the effect of two.

Note that if you added more subgroups, for instance a third child of `top_group` , `middle_dc` , then according to `example_scope_2` above, for transactions originating from `left_dc` , all the nodes in `left_dc` must plus any 1 in `right_dc` and any 1 in `middle_dc` must confirm before the transaction is committed. Of course then for transactions originating in `right_dc` all the nodes in `right_dc` plus any 1 node in `left_dc` and any 1 node in `middle_dc` must confirm before the transaction is committed. Lastly, because `middle_dc` is a child of `top_group` , `example_scope_2` also means that for transactions originating in `middle_dc` , all the nodes in `middle_dc` plus any 1 node in `left_dc` and any 1 node in `right_dc` must confirm before the transaction is committed.

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 `ANY 2 (group) GROUP COMMIT`, can be broken down into its components. `ANY 2 (group)` is the commit scope group specifying, for the rule, which nodes need to respond and confirm they processed the transaction. In this example, any two nodes from the named group must confirm.

No confirmation level is specified, which means that the default is used. You can think of the rule in full, then, as:

```
ANY 2 (group) ON visible GROUP COMMIT
```

The `visible` setting means the nodes can confirm once all the transaction's changes are flushed to disk and visible to other transactions.

The last part of this operation is the commit scope kind, which in this example is `GROUP COMMIT`. `GROUP COMMIT` is a synchronous two-phase commit that's confirmed when any two nodes in the named group confirm they've flushed the transactions changes and made them visible.

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. Unlike `GROUP COMMIT` , it's a synchronous non-two-phase commit operation. Like `GROUP COMMIT` , it 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* .

GROUP COMMIT

Group Commit is a synchronous, two-phase commit that's confirmed according to the requirements of the commit scope group. `GROUP COMMIT` has options that control:

- Whether to track transactions over interruptions (Boolean, defaults to off)
- How to resolve conflicts (`async` or `eager` , defaults to `async`)
- How to obtain a consensus (`group` , `partner` or `raft` , defaults to `group`)

For more details, see *Durability options (Group Commit/CAMO)* .

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* .

Combining rules

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.

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 Group 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 .

Commit scope kind

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` .

Group Commit

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.

> **Warning:** Group commit is currently offered as an experimental feature intended for preview and evaluation purposes. While it provides valuable capabilities, it has known limitations and challenges that make it unsuitable for production environments. We recommend that customers avoid using this feature in production scenarios until these limitations are addressed in future releases.

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 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 reconciliator, 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 reconciliator. 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 *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.

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.

Note: The *CAMO* commit scope kind is mostly an alias for *GROUP COMMIT (transaction_tracking = true, commit_decision = partner)* with an additional *DEGRADE ON* clause.

Requirements

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

The function *bdr.is_camo_partner_connected()* allows checking the connection status of a CAMO partner node configured in pair mode. There currently is no equivalent for CAMO used with Eager Replication.

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 the node_id and transaction_id of the transaction you want to check on. With CAMO used in pair mode, you can use this function only on a node that's part of a CAMO pair. Along with Eager Replication, you can use it on all nodes.

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 or to all PGD nodes in case of Eager All-Node Replication.

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.

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.

Working with commit scopes

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 *Durability options (Group Commit/CAMO)* 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 Group 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
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.

Predefined Commit Scopes

Both PGD Essential and PGD Expanded provide a set of predefined commit scopes that are available for use.

The difference between the two editions is that PGD Essential has a limited set of predefined commit scopes that cannot be changed, while PGD Expanded allows for fully manageable and configurable commit scopes. The predefined commit scopes in PGD Essential are designed to provide a balance between performance and data safety, while the configurable commit scopes in PGD Expanded offer more flexibility and control over the durability guarantees.

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.

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.

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 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 Group Commit	apply first	before COMMIT on origin
PGD CAMO	apply first	before COMMIT on origin

5.1.3.3.9 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 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.

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.

- *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.

You can also define a different behavior using a conflict trigger .

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.

A simple schema provides an example that explains:

```
CREATE TABLE pktest (pk integer primary key, val integer);
INSERT INTO pktest VALUES (1,1);
```

Updating the Primary Key column is possible, so this SQL succeeds:

```
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;

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;
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
DETAIL:  Key (pk)=(4) already exists
```

So for Postgres applications that update primary keys, be careful to avoid runtime errors, even without PGD.

With PGD, the situation becomes more complex if UPDATE operations are allowed from multiple locations at same time.

Executing these two changes concurrently works:

```
node1: UPDATE pktest SET pk=pk+1 WHERE pk = 2;
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. 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 Eager Replication.

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 *Effects of Eager Replication in general* to prevent these conflicts.

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
    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:

- *Origin conflict detection* (default)
- *Row version 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`.

Row origins are available only if `track_commit_timestamp = on`.

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
```

(continues on next page)

(continued from previous page)

```
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)
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.

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.

- *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)
```

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
 -----+-----------
```

(continues on next page)

(continued from previous page)

```
    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 *Effects of Eager Replication in general* , since this avoids multiple concurrent resets. You can also use Eager Replication 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
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;

- - 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;
```

(continues on next page)

(continued from previous page)

```
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;

- - 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
```

(continues on next page)

(continued from previous page)

```
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

-- 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;
```

5.1.3.3.10 Testing and tuning PGD clusters

You can test PGD applications using the following approaches:

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.

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.

5.1.3.4 Functions and command

5.1.3.4.1 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_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_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*
- *bdr.node_replication_rates*
- *bdr.node_slots*
- *bdr.node_summary*
- *bdr.parted_origin_catchup_info*
- *bdr.parted_origin_catchup_info_details*
- *bdr.queue*
- *bdr.replication_set*
- *bdr.replication_set_table*

- *bdr.replication_set_ddl*
- *bdr.replication_sets*
- *bdr.schema_changes*
- *bdr.sequence_alloc*
- *bdr.sequences*
- *bdr.stat_activity*
- *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_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.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.wait_slot_confirm_lsn*
- *bdr.wait_node_confirm_lsn*
- *bdr.wait_for_apply_queue*
- *bdr.get_node_sub_receive_lsn*
- *bdr.get_node_sub_apply_lsn*
- *bdr.replicate_ddl_command*
- *bdr.run_on_all_nodes*
- *bdr.run_on_nodes*
- *bdr.run_on_group*
- *bdr.global_lock_table*
- *bdr.wait_for_xid_progress*
- *bdr.local_group_slot_name*
- *bdr.node_group_type*
- *bdr.alter_node_kind*
- *bdr.alter_subscription_skip_changes_upto*

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*

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*
- :ref:``bdr.global_lock_idle_timeout` <*bdr.global_lock_idle_timeout*>`
- *bdr.lock_table_locking*
- :ref:``bdr.predictive_checks` <*bdr.predictive_checks*>`

Node management

- *bdr.replay_progress_frequency*

Generic replication

- *bdr.writers_per_subscription*
- *bdr.max_writers_per_subscription*
- :ref:``bdr.xact_replication` <*bdr.xact_replication*>`
- *bdr.permit_unsafe_commands*
- *bdr.batch_inserts*
- *bdr.maximum_clock_skew*
- *bdr.maximum_clock_skew_action*
- :ref:``bdr.accept_connections` <*bdr.accept_connections*>`
- *bdr.writer_input_queue_size*
- *bdr.writer_output_queue_size*
- *bdr.min_worker_backoff_delay*

CRDTs Overview

- :ref:``bdr.crdt_raw_value` <*bdr.crdt_raw_value*>`

Commit Scopes

- *bdr.commit_scopes*

Commit At Most Once

- *bdr.camo_local_mode_delay*
- :ref:``bdr.camo_enable_client_warnings` <*bdr.camo_enable_client_warnings*>`

Using with transaction streaming

- *Node configuration using bdr.default_streaming_mode*

Lag Control

- *bdr.lag_control_max_commit_delay*
- :ref:``bdr.lag_control_max_lag_size` <*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*
- :ref:``bdr.trace_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*

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*

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*

Routing functions

Commit Scopes

- *Commit scope syntax*
- *commit_scope_degrade_operation*
- *Commit scope targets*
- *ORIGIN_GROUP*
- *Commit scope groups*
- *ANY*
- *ANY NOT*
- *majority protect*
- *MAJORITY NOT*
- *Install the PGD Docker Quickstart kit*

- *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*
- *Durability options (Group Commit/CAMO)*
- *GROUP COMMIT parameters*
- *ABORT ON parameters*
- *DEGRADE ON parameters*
- *transaction_tracking settings*
- *conflict_resolution settings*
- *commit_decision settings*
- *commit_scope_degrade_operation settings*
- *Durability options (Group Commit/CAMO)*
- *DEGRADE ON parameters*
- *Lag Control*
- *LAG CONTROL parameters*

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*

Autopartitioning

- *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*
- :ref:``bdr.autopartition_disable` <*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_routing_config_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.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_get_activity*
- *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.node_pre_commit` , 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_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 Logging conflicts to a table .

`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 *pg_upgrade* 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)

continues on next page

Table 14 – continued from previous page

Name	Type	Description
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 *pg_upgrade* 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
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
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
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_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
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
synchronize_structure	"char"	Schema synchronization done during the join

continues on next page

Table 36 – continued from previous page

Name	Type	Description
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
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
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
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)
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))
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

continues on next page

Table 46 – continued from previous page

Name	Type	Description
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 persists until the parting node (`parting_node_id`) is automatically removed.

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.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	
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` additional 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
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
total_commit_time	double precision	Total time spent committing using the commit scope, in milliseconds

continues on next page

Table 59 – continued from previous page

Column	Type	Description
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

continues on next page

Table 67 – continued from previous page

Column	Type	Description
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
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 *pg_upgrade* for more details
worker_backend_state_change	timestamp with time zone	Backend state change timestamp, see *state_change* in *pg_upgrade* for more details
worker_backend_state	text	Current backend state, see *state* in *pg_upgrade* for more details
wait_event_type	text	Type of wait event the receiver is currently waiting on (if any), see *wait_event_type* in *pg_upgrade* for more details
wait_event	text	Exact event the receiver is currently waiting on (if any, see *wait_event* in *pg_upgrade* for more details)

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 68 – 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_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

continues on next page

Table 69 – continued from previous page

Column	Type	Description
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)
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

continues on next page

Table 71 – continued from previous page

Column	Type	Description
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
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

continues on next page

Table 72 – continued from previous page

Column	Type	Description
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*
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 *pg_upgrade* for more details

continues on next page

Table 73 – continued from previous page

Column	Type	Description
worker_xid	xid	Transaction id of the worker, see *backend_xid* in *pg_upgrade* for more details
worker_xmin	xid	Oldest transaction id needed by the worker, see *backend_xmin* in *pg_upgrade* for more details
worker_backend_state_change	timestamp with time zone	Backend state change timestamp see *state_change* in *pg_upgrade* for more details
worker_backend_state	text	Current backend state see *state* in *pg_upgrade* 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 *pg_upgrade* for more details)
wait_event	text	The exact event the worker is waiting on, if any (see *wait_event* in *pg_upgrade* 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

continues on next page

Table 74 – continued from previous page

Column	Type	Description
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 *pg_upgrade* for more details)
worker_xid	xid	Transaction id of the worker (see *backend_xid* in *pg_upgrade* for more details)
worker_xmin	xid	Oldest transaction id needed by the worker (see *backend_xmin* in *pg_upgrade* for more details)
worker_backend_state_change	timestamp with time zone	Backend state change timestamp (see *state_change* in *pg_upgrade* for more details)
worker_backend_state	text	Current backend state (see *state* in *pg_upgrade* for more details)
wait_event_type	text	The type of wait event the writer is currently waiting on, if any (see *event_type* in *pg_upgrade* for more details)
wait_event	text	The exact event the writer is waiting on, if any (see *wait_event* in *pg_upgrade* 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
command_progress_cmdtag	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 *pg_upgrade* 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

continues on next page

Table 76 – continued from previous page

Name	Type	Description
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.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
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

continues on next page

Table 80 – continued from previous page

Column	Type	Description
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
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)

continues on next page

Table 85 – continued from previous page

Name	Type	Description
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

continues on next page

Table 86 – continued from previous page

Column	Type	Description
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

Disables the consensus worker on the local node until server restart or until it's reenabled using `bdr.consensus_enable` (whichever happens first).

> **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

Reenabled disabled consensus worker on local node.

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)
```

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 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 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)
```

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.

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.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 Optimized Topology , 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.

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 Optimized Topology 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_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.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.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.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.
ddl_locking	A string that sets the *pg_upgrade* 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_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.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.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.

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.local_group_slot_name

Returns the name of the group slot on the local node.

Example

```
pgddb=# SELECT bdr.local_group_slot_name();
 local_group_slot_name
- ----------------------
 bdr_pgddb_bdrgroup
```

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.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);
```

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 optimized topology 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.

continues on next page

Table 106 – continued from previous page

Column name	Description
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.

Synopsis

```
bdr.routing_leadership_transfer(node_group_name text,
              leader_name text,
              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		Name of node that will become write leader.
transfer_method	text	*'strict'*	Type of the transfer. It can be *'fast'* or the default, *'strict'*, which checks the maximum lag.
transfer_timeout	interval	'10s'	Timeout of the leadership transfer. Default is 10 seconds.

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 *pg_upgrade bdr.local_node_id*.
xid	The transaction id on the origin node, usually retrieved by the client before *COMMIT* from the *pg_upgrade 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.

continues on next page

Table 108 – continued from previous page

Parameter	Description

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.

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.

```
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 *Statement-specific DDL replication concerns* 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 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. (Default)
leader	Use leader-based global DML locking.
auto	Currently synonomous with`leader`.
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_replication` is set to `off` . Additionally, a `WARNING` message is written whenever any global locking steps are skipped due to this setting. It's normal for some statements to 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 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.

Commit scope

bdr.commit_scope

Sets the current (or default) commit scope (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.

bdr.track_subscription_apply

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.

---

bdr.receive_lcr

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.

---

bdr.lcr_cleanup_interval

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.

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.

bdr.raft_log_min_apply_duration

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, nodes perform automatic synchronization of data from a node that is furthest ahead with respect to the down node. Default (from 6.0.1) is on.

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.
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.

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

This is a regular command that's added to PostgreSQL's bin directory.

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
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*.
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 *Origin groups* for more details.
enable_routing	*boolean*	Where ``Connection Manager` <https://www.enterprisedb.com/docs/pgd/latest/reference/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.

continues on next page

Table 116 – continued from previous page

Name	Type	Description
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 *pg_upgrade* 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.
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* .
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* .

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 the PGD node routing configuration.

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.
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.

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*.
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*.

continues on next page

Table 125 – continued from previous page

Name	Description
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.

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.

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
)
```

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.

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.

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:
{ ANY num [NOT] commit_scope_target
  | MAJORITY [NOT] commit_scope_target
  | ALL [NOT] commit_scope_target }

confirmation_level:
```

(continues on next page)

(continued from previous page)

```
    [ 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 ]
  | 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, *you can degrade* 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,
    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 *Durability options (Group Commit/CAMO)* , the *commit_decision settings* 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.

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*
- *Durability options (Group Commit/CAMO)*
- *CAMO (Commit At Most Once)*
- *Lag Control*

```
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`).

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.)
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 *commit_decision settings* must be set to `raft` to avoid reconcilation issue.

See *Missing-column conflict resolution* .

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`).

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.

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.

continues on next page

Table 137 – continued from previous page

Resolver	Description
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 ex-ists	up-date dif-fer-ing	up-date ori-gin chang	up-date miss-ing	up-date re-cently delete	up-date pkey ex-ists	delete re-cently up-dated	delete miss-ing	tar-get col-umn miss-ing	source col-umn miss-ing	tar-get table miss-ing	mul-tiple 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

List of conflict resolutions

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.
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,
                                   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 Testing and tuning .

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 *Statement-specific DDL replication concerns* .

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.

Autopartition

Autopartition allows you to split tables into several partitions. For more information, see *Autopartitioning*.

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 :ref:``bdr.autopartition_disable` <*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

See also:

Stream Triggers for an introduction to Stream Triggers.

Both conflict triggers and transform triggers have access to information about rows and metadata by way of the pre-defined variables provided by the trigger API and additional information functions provided by PGD.

In PL/pgSQL, you can use the predefined variables and functions that follow:

- *Stream triggers row variables*
- *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*

Creating and dropping stream triggers is managed through the 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 Conflict types

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	

continues on next page

Table 142 – continued from previous page

Name	Type	Description
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

continues on next page

Table 144 – continued from previous page

Name	Type	Description
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

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
effective_route_dsn	text	Full DSN of this node

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
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_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.
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_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
node_name	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
route_dsn	text	Interface of this node

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`=time-shard,`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 dunction is used by the upgrade scripts to transfer commit decisions saved in bdr.node_pre_commit 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.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.
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_get_activity

Internal function underlying view `bdr.stat_activity` . Do not use directly. Use the bdr.stat_activity view instead.

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`.

5.1.3.4.2 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.

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
```

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 *Security Best Practices* . 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,
    "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 *Security Best Practices* . 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 *Security Best Practices* . 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

- *Creating your first cluster (PGD Essential)* : Cluster-level commands for managing the cluster.
- *pgd cluster show* : Show cluster-level information.
- *pgd cluster verify* : Verify cluster-level information.
- *Durability options (Group Commit/CAMO)* : 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 group status* : Group related commands for listing groups. - *pgd commit-scopes list* : List groups.
- *Nodes* : Node-level commands for managing nodes.
- *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.
- *pgd events* : Event log commands for viewing events. - *pgd cluster show* : Show events.
- *Use Case 1: Multi-Master Replication* : Replication related-commands for managing replication. - *pgd cluster show* : Show replication information.
- *Monitoring Raft consensus* : Raft related commands for managing Raft consensus. - *pgd cluster show* : Show information about Raft state.
- *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* : Commit Scopes related commands for PGD cluster.
- *pgd commit-scopes list* : List commit scopes information for the cluster.
- *PGD compatibility assessment* : 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"

continues on next page

Table 164 – continued from previous page

Short	Long	Description
	--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` commands are used to assess the suitability of a Postgres server instance for migration to the EDB Postgres Distributed cluster.

The command must be run with a DSN that connects 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. Not doing either one will result in an error.

The command will then check the Postgres server instance for compatibility with the EDB Postgres Distributed cluster, and will provide a report on the compatibility of the Postgres server instance.

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
- ---------------- ------ -----------------------------------------------
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
- ------------------------ ------ ------
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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
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

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
 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
```

(continues on next page)

(continued from previous page)

```
 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"
      },
      {
        "info": "Parent Group Name",
        "value": "pgdx6"
      },
      {
        "info": "Group Type",
        "value": "data"
      },
      {
        "info": "Write Leader",
        "value": "pgd-a3"
      },
      {
```

(continues on next page)

(continued from previous page)


```
      "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": ""
    },
    {
      "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",
```


(continues on next page)

(continued from previous page)


```
      "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",
      "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"
    },
```


(continues on next page)

(continued from previous page)


```
      {
        "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"
      },
      {
        "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.

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

continues on next page

Table 173 – continued from previous page

Option	Description
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.
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	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.

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.
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.

continues on next page

Table 175 – continued from previous page

Option	Description
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.
proxy_fallback_node_groups	The fallback node groups for the proxy in the group.

continues on next page

Table 177 – continued from previous page

Option	Description
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
 read_write_consensus_timeout       | 00:00:00
```

(continues on next page)

(continued from previous page)

```
 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

- *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__
[
  {
    "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.full}} 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.full}} 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.

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.

continues on next page

Table 184 – continued from previous page

Option	Description
--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.
--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"
      },
```

(continues on next page)

(continued from previous page)


```
      {
        "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",
        "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 PostgreSQL version on a node in the EDB Postgres Distributed cluster.

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
-p	--old-port	5432	PGPORTOLD	Old Postgres instance port
	--socketdir	/var/run/postgresql	PGSOCKETDIR	Directory to use for postmaster sockets during upgrade
	--new-socketdir	/var/run/postgresql	PGSOCKETDIRNEW	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

continues on next page

Table 185 – continued from previous page

Short	Long	Default	Env	Description
-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 .

Examples

In the following examples, "kaolin" is the name of the node to upgrade, from the Quickstart democluster.

Upgrade the PostgreSQL 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 PostgreSQL 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 PostgreSQL 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 PostgreSQL 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               ␣
```

(continues on next page)

(continued from previous page)

```
↪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 display the raft status in the EDB Postgres Distributed cluster.

Subcommands

- *pgd cluster show* : Show raft status for the cluster.

pgd raft show

Synopsis

The `pgd raft show` command is used to display the Raft status in the EDB Postgres Distributed cluster. In particular, it lists all nodes in all groups, including the top level group, and their Raft status - leader or follower, number of nodes in the group with them, number of voting nodes in the group, presence of a leader, and the term number.

Syntax

```
pgd raft show [OPTIONS]
```

Options

The following options are available for the `pgd raft show` command:

Short	Long	Description
	--group	Filter by group name.

See Global Options for common global options.

Examples

Show Raft status

```
pgd raft show
__OUTPUT__
Group Name    Node Name State         Leader Name Current Term Commit Index Nodes␣
↪Voting Nodes Protocol Version
- ------------ --------- ------------- ----------- ------------ ------------ ----- ---
↪--------- ----------------
dc1_subgroup  kaftan    RAFT_LEADER   kaftan      1            4            3     3  ␣
↪         0
dc1_subgroup  kaboom    RAFT_FOLLOWER kaftan      1            4            3     3  ␣
↪         0
dc1_subgroup  kaolin    RAFT_FOLLOWER kaftan      1            4            3     3  ␣
↪         0
democluster   kaftan    RAFT_LEADER   kaftan      0            335          3     3  ␣
↪         5007
democluster   kaboom    RAFT_FOLLOWER kaftan      0            335          3     3  ␣
↪         5007
democluster   kaolin    RAFT_FOLLOWER kaftan      0            335          3     3  ␣
↪         5007
```

Note that `dc1_subgroup` here is a data group with local routing, and `democluster` is the top level group with global routing.

The `Protocol Version` column shows the version of the Raft protocol in use. The `Commit Index` column shows the index of the last committed log entry. The `Nodes` column shows the total number of nodes in the group. The `Voting Nodes` column shows the number of nodes that participate in the Raft consensus. The `State` column shows the Raft state of the node - leader or follower. The `Leader Name` column shows the name of the leader node in the group. The `Current Term` column shows the current term number.

Show Raft status for a specific group

```
pgd raft show --group dc1_subgroup
__OUTPUT__
Group Name    Node Name State         Leader Name Current Term Commit Index Nodes␣
↪Voting Nodes Protocol Version
- ------------ --------- ------------- ----------- ------------ ------------ ----- ---
↪--------- ----------------
dc1_subgroup  kaftan    RAFT_LEADER   kaftan      1            4            3     3  ␣
↪         0
dc1_subgroup  kaboom    RAFT_FOLLOWER kaftan      1            4            3     3  ␣
↪         0
dc1_subgroup  kaolin    RAFT_FOLLOWER kaftan      1            4            3     3  ␣
↪         0
```

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.
	--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.

Symbol	Meaning
*	ok
~	warning (lag > 10M)
!	critical (lag > 100M OR slot is 'inactive' OR 'disconnected')
x	down / unreachable
-	n/a

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 *      -      *
```

(continues on next page)

(continued from previous page)

```
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

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 *      *      -
```

5.1.3.5 Guidance for dev

5.1.3.5.1 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.

- *Application behavior* looks at how PGD replication appears to an application, such as:
- The commands that are replicated
- The commands that run locally
- When row-level locks are acquired
- How and where triggers fire
- Large objects
- Toast
- *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.
- *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.
- *General rules for applications* offers some general rules for applications to avoid data anomalies.
- *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.
- *Using extensions with PGD* explains how to select, install, and configure extensions on PGD.
- *Use of table access methods (TAMs) in PGD* (TAMs) notes the TAMs available with PGD and how to enable them.
- *Feature compatibility* shows which server features work with which commit scopes and which commit scopes can be daisy chained together.

Application behavior

Much of PGD's replication behavior is transparent to applications. Understanding how it achieves that and the elements that aren't transparent is important to successfully developing an application that works well with PGD.

Replication behavior

PGD supports replicating changes made on one node to other nodes.

PGD, by default, replicates all changes from INSERT, UPDATE, DELETE, and TRUNCATE operations from the source node to other nodes. Only the final changes are sent, after all triggers and rules are processed. For example, `INSERT ... ON CONFLICT UPDATE` sends either an insert or an update, depending on what occurred on the origin. If an update or delete affects zero rows, then no changes are sent.

You can replicate INSERT without any preconditions.

For updates and deletes to replicate on other nodes, PGD must be able to identify the unique rows affected. PGD requires that a table have either a PRIMARY KEY defined, a UNIQUE constraint, or an explicit REPLICA IDENTITY defined on specific columns. If one of those isn't defined, a warning is generated, and later updates or deletes are explicitly blocked. If REPLICA IDENTITY FULL is defined for a table, then a unique index isn't required. In that case, updates and deletes are allowed and use the first non-unique index that's live, valid, not deferred, and doesn't have expressions or WHERE clauses. Otherwise, a sequential scan is used.

Truncate

You can use TRUNCATE even without a defined replication identity. Replication of TRUNCATE commands is supported, but take care when truncating groups of tables connected by foreign keys. When replicating a truncate action, the subscriber truncates the same group of tables that was truncated on the origin, either explicitly specified or implicitly collected by CASCADE, except in cases where replication sets are defined. See Replication sets for details and examples. This works correctly if all affected tables are part of the same subscription. But if some tables to truncate on the subscriber have foreign-key links to tables that aren't part of the same (or any) replication set, then applying the truncate action on the subscriber fails.

Row-level locks

Row-level locks taken implicitly by INSERT, UPDATE, and DELETE commands are replicated as the changes are made. Table-level locks taken implicitly by INSERT, UPDATE, DELETE, and TRUNCATE commands are also replicated. Explicit row-level locking (`SELECT ... FOR UPDATE/FOR SHARE`) by user sessions isn't replicated, nor are advisory locks. Information stored by transactions running in SERIALIZABLE mode isn't replicated to other nodes. The transaction isolation level of SERIALIAZABLE is supported, but transactions aren't serialized across nodes in the presence of concurrent transactions on multiple nodes.

If DML is executed on multiple nodes concurrently, then potential conflicts might occur if executing with asynchronous replication. You must either handle these or avoid them. Various avoidance mechanisms are possible, discussed in Conflicts .

Sequences

Sequences need special handling, described in Sequences . This is because in a cluster, sequences must be global to avoid nodes creating conflicting values. Global sequences are available with global locking to ensure integrity.

Binary objects

Binary data in BYTEA columns is replicated normally, allowing "blobs" of data up to 1 GB. Use of the PostgreSQL "large object" facility isn't supported in PGD.

Rules

Rules execute only on the origin node so aren't executed during apply, even if they're enabled for replicas.

Base tables only

Replication is possible only from base tables to base tables. That is, the tables on the source and target on the subscription side must be tables, not views, materialized views, or foreign tables. Attempts to replicate tables other than base tables result in an error. DML changes that are made through updatable views are resolved to base tables on the origin and then applied to the same base table name on the target.

Partitioned tables

PGD supports partitioned tables transparently, meaning that you can add a partitioned table to a replication set and changes that involve any of the partitions are replicated downstream.

Triggers

By default, triggers execute only on the origin node. For example, an INSERT trigger executes on the origin node and is ignored when you apply the change on the target node. You can specify for triggers to execute on both the origin node at execution time and on the target when it's replicated (*apply time*) by using `ALTER TABLE ... ENABLE ALWAYS TRIGGER`. Or, use the `REPLICA` option to execute only at apply time: `ALTER TABLE ... ENABLE REPLICA TRIGGER`.

Some types of trigger aren't executed on apply, even if they exist on a table and are currently enabled. Trigger types not executed are:

- Statement-level triggers (`FOR EACH STATEMENT`)
- Per-column UPDATE triggers (`UPDATE OF column_name [, ...]`)

PGD replication apply uses the system-level default search_path. Replica triggers, stream triggers, and index expression functions can assume other search_path settings that then fail when they execute on apply. To prevent this from occurring, use any of these techniques:

- Resolve object references clearly using only the default search_path.
- Always use fully qualified references to objects, for example, `schema.objectname`.
- Set the search path for a function using `ALTER FUNCTION ... SET search_path = ...` for the functions affected.

PGD assumes that there are no issues related to text or other collatable datatypes, that is, all collations in use are available on all nodes, and the default collation is the same on all nodes. Replicating changes uses equality searches to locate Replica Identity values, so this does't have any effect except where unique indexes are explicitly defined with nonmatching collation qualifiers. Row filters might be affected by differences in collations if collatable expressions were used.

Toast

PGD handling of very long "toasted" data in PostgreSQL is transparent to the user. The TOAST "chunkid" values likely differ between the same row on different nodes, but that doesn't cause any problems.

Other restrictions

PGD can't work correctly if Replica Identity columns are marked as external.

PostgreSQL allows CHECK() constraints that contain volatile functions. Since PGD reexecutes CHECK() constraints on apply, any subsequent reexecution that doesn't return the same result as before causes data divergence.

PGD doesn't restrict the use of foreign keys. Cascading FKs are allowed.

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 *Statement-specific DDL replication concerns* .

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, REINDEX)
- 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.

Nodes with differences

Replicating between nodes with differences

By default, DDL is sent to all nodes. You can control this behavior, as described in *Statement-specific DDL replication concerns* , 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 *Statement-specific DDL replication concerns* .

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 DDLreplication 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.

General rules for applications

Background

PGD uses replica identity values to identify the rows to change. Applications can cause difficulties if they insert, delete, and then later reuse the same unique identifiers. This is known as the ABAproblem . PGD can't know whether the rows are the current row, the last row, or much older rows.

Similarly, since PGD uses table names to identify the table against which changes are replayed, a similar ABA problem exists with applications that create, drop, and then later reuse the same object names.

Rules for applications

These issues give rise to some simple rules for applications to follow:

- Use unique identifiers for rows (INSERT).
- Avoid modifying unique identifiers (UPDATE).
- Avoid reusing deleted unique identifiers.
- Avoid reusing dropped object names.

In the general case, breaking those rules can lead to data anomalies and divergence. Applications can break those rules as long as certain conditions are met. However, use caution: while anomalies are unlikely, they aren't impossible. For example, you can reuse a row value as long as the DELETE was replayed on all nodes, including down nodes. This might normally occur in less than a second but can take days if a severe issue occurred on one node that prevented it from restarting correctly.

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.

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:

- *Autocluster*
- *Refdata*

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. However if you're using the pgaudit extension, `pgaudit` must appear in the list before `bdr` .

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` .

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*

Feature compatibility

Server feature/commit scope interoperability

Not all server features work with all commit scopes. This table shows the ones that interoperate.

	Async (default)	*Parallel Apply*	*Using with transaction streaming*	*pg_upgrade*
Durability options (Group Commit/CAMO)	Not applicable	Does not interoperate	Does not interoperate	Interoperates
Durability options (Group Commit/CAMO)	Not applicable	Interoperates	Does not interoperate	Does not interoperate
Lag Control	Interoperates	Interoperates	Interoperates	Interoperates
SYNCHRONOUS COMMIT	Not applicable	Interoperates	Interoperates	Interoperates

Legend: Not applicable Not applicable Does not interoperate Does not interoperate Interoperates Interoperates

Notes

Not applicable : 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.

	Durability options (Group Commit/CAMO)	*Durability options (Group Commit/CAMO)*	*Lag Control*	*SYNCHRONOUS COMMIT*
Durability options (Group Commit/CAMO)	Not applicable	Does not interoperate	Interoperates	Interoperates
Durability options (Group Commit/CAMO)	Does not interoperate	Not applicable	Interoperates	Does not interoperate
Lag Control	Interoperates	Interoperates	Not applicable	Interoperates
SYNCHRONOUS COMMIT	Interoperates	Does not interoperate	Interoperates	Not applicable

Legend: Not applicable Not applicable Does not interoperate Does not combine Interoperates Combines

Notes

Each commit scope implicitly works with itself.

5.2 Terminology

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

5.2.1 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 *Synchronous replication in near/far architecture* . 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.

5.2.2 Commit scopes

Rules for managing how transactions are committed between the nodes and groups of a PGD cluster. Used to configure *Synchronous replication in near/far architecture* , *Durability options (Group Commit/CAMO)* , *CAMO or commit-at-most-once* , *Eager* , Lag Control, and other PGD features.

5.2.3 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.

5.2.4 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.

5.2.5 Consensus

How *Monitoring Raft consensus* 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.

5.2.6 Cluster

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

5.2.7 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.

5.2.8 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.

5.2.9 Eager

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

5.2.10 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.

5.2.11 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.

5.2.12 Group commit

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

5.2.13 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.

5.2.14 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.

5.2.15 Node

A general term for an element of a distributed system. A node can play host to any service. In PGD, *PGD node setup* 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.

5.2.16 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.

5.2.17 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* .

5.2.18 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.

5.2.19 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.

5.2.20 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 *Accessing the PGD Cluster* .

5.2.21 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 Raft *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.

5.2.22 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.

5.2.23 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.

5.2.24 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.

5.2.25 Switchover

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

5.2.26 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.

5.2.27 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.

5.2.28 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.

5.2.29 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.

5.2.30 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.

5.2.31 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*.

5.2.32 Writer

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

CHAPTER 6

Appendix

6.1 PGD compatibility

6.1.1 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
EDB Postgres Distributed 6 release notes	18.1.0+
EDB Postgres Distributed 6 release notes	17.5.0+
EDB Postgres Distributed 6 release notes	16.9.0+
EDB Postgres Distributed 6 release notes	15.13.0+
EDB Postgres Distributed 6 release notes	14.18.0+

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

6.1.2 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.

6.1.2.1 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	
Ubuntu 22.04	Yes		
Ubuntu 24.04	Yes		
Debian 12	Yes		Yes
IBM Linux One	Yes		

Note: See *PGD compatibility* for previous versions of PGD.

6.2 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
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*

6.2.1 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.

6.2.1.1 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).

6.2.1.2 Features

Description	Addresses
<details><summary>Added TDE options to <code>pgd node upgrade</code> and <code>pgd node setup</code> commands.</summary> The <code>pgd node upgrade</code> command now supports in-place conversion of a non-TDE PGD node to a TDE-enabled node with the <code>--key-unwrap-command</code> and <code>--copy-by-block</code> options. The <code>pgd node setup</code> command now includes comprehensive options to configure TDE during the initial provisioning of a new cluster node. These new options are: <code>--data-encryption</code>, <code>--copy-key-from</code>, <code>--key-wrap-command</code>, <code>--key-unwrap-command</code>, <code>--no-key-wrap</code>, and <code>--data-encryption-keylen</code> (accepts the AES key length). </details>	
<details><summary>Added <code>bdr.stat_receiver_transactions</code> view.</summary> 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. </details>	
<details><summary>Added support for LDAP authentication to Connection Manager.</summary> Connection Manager now supports the same LDAP authentication features as PostgreSQL, with the exception that OpenLDAP configuration files and environment variables are not supported. </details>	
<details><summary>Added detailed monitoring columns to <code>bdr.writers</code> view.</summary> The <code>bdr.writers</code> 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. </details>	

6.2.1.3 Enhancements

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

6.2.1.4 Bug Fixes

Description	Addresses
<details><summary>Fixed an issue where the <code>pgd node upgrade</code> command failed with an <code>invalid_slots</code> error when upgrading to PostgreSQL 18 or later.</summary> The BDR extension has been updated to improve compatibility with the <code>pg_upgrade</code> utility, which ensures the upgrade process runs cleanly. </details>	
<details><summary>Fixed contradictory output for switchover with Connection Manager DSN.</summary> Resolved an issue where the <code>pgd group set-leader</code> 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. </details>	
<details><summary>Fixed unexpected subscription and slot creation involving subscriber-only and witness nodes.</summary> 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. </details>	
<details><summary>Improved Connection Manager routing change logic.</summary> 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. </details>	
<details><summary>Fixed an issue whereby a connection is silently dropped in the Connection Manager.</summary> The fix improves the handling of authentication failures in the Connection Manager. </details>	
<details><summary>Fixed an issue whereby Connection Manager crashed when full hostname is used in the <code>pg_hba.conf</code> <code>address</code> field.</summary> 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. </details>	
<details><summary>Fixed writer error <code>could not open temporary file</code> after restart.</summary> 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. </details>	53341
<details><summary>Fixed an issue to ensure that the values of <code>node_uuid</code> and <code>node_kind</code> of a node are preserved after joining a cluster in a different PGD version.</summary> This bug caused the <code>node_uuid</code> in system tables (<code>bdr.local_node</code> and <code>bdr.node</code>) to diverge after a full cluster upgrade. The fix ensures that the joining node's UUID and kind are correctly preserved throughout the process. </details>	
<details><summary>Allow rename of sequence or schema containing galloc sequences.</summary> 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 <code>bdr.sequence_kind</code> catalog, which remains constant despite any sequence or schema renames. </details>	

continues on next page

Table 6 – continued from previous page

Description	Addresses
<details><summary>Added version and distribution checks for physical join in <code>pgd node setup</code>.</summary> The <code>pgd node setup</code> command now performs stricter compatibility checks before allowing a physical join: - 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. - 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. - If the PostgreSQL distribution (flavor) of the local and remote nodes are not the same, the command exits with an error. </details>	
<details><summary>Fixed backward compatibility issue for <code>pgd group</code> commands.</summary> Resolved a backward compatibility issue where PGD 6.x CLI commands, such as <code>pgd group show</code>, reported the error <code>No query found for version</code> when invoked against an older PGD 5.x cluster. The fix ensures these commands function correctly across major PGD versions. </details>	
<details><summary>Fixed invalid system identifier check for <code>pgd node setup</code>.</summary> Resolved an issue where the <code>pgd node setup</code> command incorrectly reported a mismatch error when validating a valid base backup node of the given remote node. </details>	
<details><summary>Handle returned <code>CONTEXT</code> value for <code>bdr.run_on_x</code> functions.</summary> The PGD CLI now adjusts the database connection's verbosity to ignore the <code>CONTEXT</code> value, which previously added noise to the output, especially for errors from <code>bdr.run_on_x</code> functions. </details>	
<details><summary>Fixed the <code>bdr.msgb_connect</code> hang by introducing a connection timeout.</summary> Resolved an issue where the <code>bdr.msgb_connect</code> function could hang indefinitely due to a missing network timeout setting. The fix adds a timeout to the function. </details>	53442
<details><summary>Fixed <code>pgd node setup</code> to create a PGD node from an existing data directory.</summary> Resolved an issue which prevented the <code>pgd node setup</code> command from successfully creating and configuring a PGD node using an existing data directory. </details>	
<details><summary>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.</summary> In these cases, the configuration inherited from the top-level node group is automatically applied. </details>	
<details><summary>Fixed a unique constraint violation error that could occur during an <code>UPDATE</code> operation on tables configured with both primary and unique keys.</summary> Previously, an <code>UPDATE</code> conflict with an <code>update_missing</code> 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 <code>insert_exists</code> conflict and applying the appropriate resolution logic. </details>	50539, 51417, 54323
<details><summary>The <code>ALTER TABLE SET pgd.replicate_to_analytics</code> property can now be applied to individual partitions.</summary> This allows for granular control over which partitions replicate data to analytical targets. </details>	
Fixed an issue that caused consensus breakage when a duplicate value was assigned to either the <code>analytics_storage_location</code> or <code>analytics_write_catalog</code> settings.	

6.2.2 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.

6.2.2.1 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.

6.2.2.2 Features

Description	Addresses
<details><summary>Added CLI commands to get and update Postgres settings (GUC).</summary> Add the <code>pgd node get-config</code> and <code>pgd node set-config</code> commands to allow users to retrieve and modify Postgres settings (GUC) via the CLI. </details>	
<details><summary>New <code>drop_after_retention_period</code> argument added to the <code>bdr.autopartition</code> function</summary> <code>bdr.autopartition()</code> now takes an additional boolean argument <code>drop_after_retention_period</code> which can be set to false if the user does not want the partition to be dropped, but only detached. </details>	82332
<details><summary>Simplified Node Configuration with the PGD CLI</summary> The pgd command-line tool now includes new commands to make managing individual node configurations easier. You can now use <code>pgd node <node-name> get-config</code> and <code>pgd node <node-name> set-config</code> to retrieve and update specific Postgres settings (GUCs) for any node in your cluster directly from the CLI. </details>	
<details><summary>PGD CLI View Commit Scopes</summary> The PGD CLI now includes a new command, <code>pgd commit-scopes list</code>, 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. </details>	

continues on next page

Table 7 – continued from previous page

Description	Addresses
<details><summary>PGD CLI - Part a Node from a Cluster</summary> The PGD CLI now includes a new command, <code>pgd node part</code>, which provides a streamlined way to remove a node from an active cluster. This command automates the process of calling the <code>node_part</code> SQL function and reports on its progress, simplifying node management directly from the command line. </details>	
<details><summary>Leader DDL Lock</summary> 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 <code>bdr.ddl_locking</code> configuration parameter. </details>	
<details><summary>Replication of Roles to witness nodes</summary> 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. </details>	
<details><summary>Added command to list commit scopes for PGD cluster.</summary> Add a command <code>pgd commit-scopes list</code> to list commit scopes information for PGD cluster. </details>	
<details><summary>Added command to part a PGD node from cluster.</summary> Add a command <code>pgd node part</code> to part a PGD node from cluster. </details>	

6.2.2.3 Enhancements

Description	Addresses
<details><summary>Update the <code>pgd group get-option</code> and <code>pgd group show</code> commands to show default values.</summary> The <code>pgd group get-option</code> and <code>pgd group show</code> commands will now display the default values for all options. The <code>pgd group show</code> command will display the source of the option values as well. </details>	
<details><summary>Apply the PGD Essential constraints for CLI commands.</summary> The CLI will check for and enforce the PGD Essential constraints for the commands as applicable. See PGD Essential CLI Constraints for more details. </details>	
<details><summary>Improve Raft snapshot export/import to reset Raft completely</summary> 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. </details>	
<details><summary>EPAS Interval Partition Support for PGD</summary> 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. </details>	

continues on next page

Table 8 – continued from previous page

Description	Addresses
<details><summary>Improved max_connections Error Reporting</summary> 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. </details>	
<details><summary>Support for commands that generate Table Rewrites in an explicit transaction block</summary> 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. </details>	
<details><summary>Postgres Compatibility with PGD - Phase 5</summary> This is a placeholder for future updates regarding Postgres compatibility. </details>	

6.2.2.4 Changes

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

6.2.2.5 Bug Fixes

Description	Addresses
<details><summary>Fix the <code>pgd cluster verify</code> command warning for <code>witness-only</code> group.</summary> The <code>pgd cluster verify</code> command displayed warning for <code>Data nodes per group</code> check for <code>witness-only</code> group. The check is not applicable for <code>witness-only</code> groups and hence the status <code>Ok</code> will be displayed. </details>	
<details><summary>Fix the data dir check for the setup command.</summary> The <code>pgd node setup</code> command checks for the presence of <code>PG_VERSION</code> file in the given data dir option. However the code incorrectly uses the <code>bindir</code> option instead of <code>pgdata</code>. </details>	

continues on next page

Table 10 – continued from previous page

Description	Addresses
<details><summary>Fix for Mixed_version Clusters</summary> 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. </details>	46373
<details><summary>Connection Manager Behavior on Routing Changes</summary> 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. </details>	
<details><summary>Fix for node_pkey Errors When Rebuilding Parted Nodes</summary> 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 <code>bdr_init_physical</code> without encountering this issue. </details>	44467, 45084
<details><summary>PGD CLI cluster verify Command Fix for Witness-Only Groups</summary> 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. </details>	
<details><summary>Raft Fix for Parting Lagging Nodes</summary> 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. </details>	50099
<details><summary>Fix for PGD CLI Multi-Group Setup</summary> 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. </details>	
<details><summary>Fix for CLI setup Command GUC Update</summary> 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. </details>	
<details><summary>Fix for "Clock Has Moved Backwards" Error with Snowflake Sequences</summary> 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. </details>	49376, 43659
<details><summary>Fix a crash with pg_failover_slots</summary> 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 <code>pg_failover_slots</code> is active. </details>	

continues on next page

Table 10 – continued from previous page

Description	Addresses
<details><summary>Fix a crash on a stat view for connection manager stat</summary> The view <code>bdr.stat_connection_manager_hba_file_rules</code> had some parsing issues on the <code>pg_hba.conf</code> file specifically in the <code>Address/Mask</code> column for the "all" value. </details>	
<details><summary>Node kind check</summary> 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. </details>	
<details><summary>Skip the node GUCs update with <code>pgd node setup</code> command.</summary> The <code>pgd node setup</code> verifies the Postgres settings while setting up a PGD node. The command will now skip the node GUCs update as the <code>bdr_superuser</code> 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. </details>	

6.2.3 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.

6.2.3.1 Bug Fixes

Description	Addresses
<details><summary>Fix physical join when running 'pgd node setup'</summary> The <code>pgd node setup</code> 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 <code>pgd node setup</code> with earlier versions of the PGD CLI, we recommend performing a consistency check. </details>	
<details><summary>Fix handling of DDL commands that trigger table rewrites</summary> Executing specific DDL operations (such as <code>ALTER TABLE ... ADD COLUMN ... DEFAULT <volatile expression></code> or <code>ALTER TABLE ... ALTER COLUMN ... SET DATA TYPE</code> 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. </details>	
<details><summary>Connection Manager respects currentSchema JDBC parameter</summary> Previously, a JDBC client connected through the PGD Connection Manager would have the <code>currentSchema</code> parameter ignored. As a result, a connection's <code>search_path</code> would incorrectly default to <code>public</code>. This issue has now been resolved. </details>	

continues on next page

Table 11 – continued from previous page

Description	Addresses
<details><summary>prefer_analytics_engine works with Iceberg REST Catalog tables</summary> The <code>prefer_analytics_engine</code> flag has been updated to correctly function with tables that are replicated to Iceberg REST Catalog. Setting <code>bdr.prefer_analytics_engine = true</code> will now cause a query referencing a table with <code>pgaa.replicate_to_analytics = true</code> to be executed by the analytics engine when possible. </details>	

6.2.4 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.

6.2.4.1 Bug Fixes

Description	Addresses
<details><summary>Fixed ALTER TABLE failure during streamed transaction with table rewrite</summary> The <code>ALTER TABLE</code> 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. </details>	

6.2.5 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.

6.2.5.1 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.

6.2.5.2 Features

Description	Addresses
<details><summary>Built-in connection manager</summary> New built-in connection manager which handles routing of connections automatically and allows enforcing of read-only connections to non-leader. </details>	
<details><summary>CLI cluster setup</summary> The PGD CLI now allows initial cluster setup as well as adding nodes from command-line using <code>pgd node setup</code> command. </details>	
<details><summary>Set sequence kind on group create/join</summary> Transform the sequences in distributed based on the <code>bdr.default_sequence_kind</code> GUC when creating/joining a bdr group instead of when creating the node as done in older versions. </details>	
<details><summary>Set startvalue for distributed sequences automatically</summary> 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. </details>	
<details><summary>Enabling of automatic sync and reconciliation</summary> 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 documentation. </details>	
<details><summary>Add node_uuid column to bdr.node and bdr.local_node</summary> 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. </details>	

continues on next page

Table 13 – continued from previous page

Description	Addresses
<details><summary>Change replication origin and slot naming scheme</summary> 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: Slots Naming Convention - normal slot to a node => <code>bdr_node_<targetuuid>_<dbhash></code> - join slot for node => <code>bdr_node_<targetuuid>_<dbhash>_tmp</code> - group slot for a topgroup => <code>bdr_group_<topgroupuuid>_<dbhash></code> - slot for any forwarding + lead to lead => <code>bdr_node_<targetuuid>_<originidhex>_<dbhash></code> - analytics slot => <code>bdr_analytics_<groupuuid>_<dbhash></code> - decoding slot => <code>bdr_decoder_<topgroupuuid>_<dbhash></code> Origins Naming Convention: - normal origin to a node => <code>bdr_<originuuid>_<dbhash></code> - fwd origin to a source node => <code>bdr_<originuuid>_<sourceoidhex>_<dbhash></code> </details>	
<details><summary>Limit on the number of node groups allowed in the system for PGD Essential.</summary> 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. </details>	
<details><summary>Enforced PGD Essential limits - data node count</summary> Don't allow PGD Essential clusters to join more than 4 data nodes. </details>	
<details><summary>Added <code>bdr.wait_node_confirm_lsn()</code> function which waits until a given reaches a given LSN</summary> <code>bdr.wait_node_confirm_lsn(</code>) will look at the confirmed_flush_lsn of the given node when available, otherwise it will query <code>pg_replication_origin_progress()</code> of that node, and wait for the specified LSN to be reached by said node. </details>	
<details><summary>Subscriber-only nodes can now be added to data node groups</summary> 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. </details>	
<details><summary>Add <code>bdr.local_analytics_slot_name()</code> SQL function.</summary> Returns name of analytics slot. This merely produces the correct name irrespective of whether analytics feature is in use. </details>	
<details><summary>Add node_uuid column to <code>bdr.node_summary</code> view.</summary> Added to complement the addition of the node_uuid column to bdr.node and bdr.local_node </details>	

6.2.5.3 Enhancements

Description	Addresses
<details><summary>Multiple conflicting rows resolution</summary> Both <code>pk_exists</code> and <code>multiple_unique_conflicts</code> conflict types can now resolve more than one conflicting row by removing any old rows that are part of the conflict. The <code>multiple_unique_conflicts</code> now defaults to <code>update_if_newer</code> resolver, so it does not throw error by default anymore. </details>	
<details><summary>Improved <code>bdr.stat_activity</code> view</summary> The <code>backend_type</code> now shows consistent worker type for PGD workers without the extra process identification. The <code>wait_event_type</code> and <code>wait_event</code> 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. </details>	
<details><summary>The PARTED node is removed automatically from all nodes in the cluster.</summary> 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. - For local node, it will remove all the node metadata, including information about remote nodes. - For remote node, it removes only metadata for that specific node. Hence with this release - 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 - A node will not remain in PARTED state as the node is removed as soon as it moves to PARTED state. </details>	
<details><summary>The <code>--summary</code> and <code>--options</code> flags for <code>pgd node show</code> CLI command.</summary> Add the <code>--summary</code> and <code>--options</code> flags to <code>pgd node show</code> command to filter the output of the <code>pgd node show</code> command. This also maintains symmetry with other <code>show</code> commands. </details>	
<details><summary>More GUCs verfied in <code>pgd cluster verify</code> CLI command.</summary> Add the <code>bdr.lock_table_locking</code> and <code>bdr.truncate_locking</code> GUCs to list of GUCs verfied in <code>pgd cluster verify</code> command. </details>	
<details><summary>Table rewriting <code>ALTER TABLE... ALTER COLUMN</code> calls are now supported.</summary> 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 </code></pre> This also includes support for <code>ALTER TYPE</code> when using the <code>USING</code> 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; </code></pre> Table rewrites can hold an AccessExclusiveLock for extended periods on larger tables. </details>	
<details><summary>Restrictions on non-immutable <code>ALTER TABLE... ADD COLUMN</code> calls have been removed.</summary> The restrictions on non-immutable <code>ALTER TABLE... ADD COLUMN</code> calls have been removed. </details>	

continues on next page

Table 14 – continued from previous page

Description	Addresses
<details><summary>Synchronize roles and tablespaces during logical join</summary> 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. </details>	
<details><summary>Introduce <code>bdr.node_group_config_summary</code> view</summary> The new <code>bdr.node_group_config_summary</code> 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 <code>pg_settings</code> </details>	
<details><summary>Leader DML lock</summary> 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 <code>bdr.ddl_locking</code> configuration parameter. </details>	
<details><summary>Disabling bdr.xact_replication in run_on_* functions</summary> Functions <code>run_on_nodes</code>, <code>run_on_all_nodes</code> and <code>run_on_group</code> now sets <code>bdr.xact_replication</code> to <code>off</code> by default. </details>	
<details><summary>Replica Identity full by default</summary> The <code>auto</code> value for <code>bdr.default_replica_identity</code> 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 <code>bdr.default_replica_identity</code>. </details>	
<details><summary>Tablespace replication as a DDL operation is supported.</summary> Tablespace operations <code>CREATE/ALTER/DROP TABLESPACE</code> 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 <code>allow_in_place_tablespace</code>. </details>	
<details><summary>Improve the CLI debug messages.</summary> Improve the formating of the log messages to be more readable and symmetrical with Postgres log messages. </details>	
<details><summary>New column for <code>pgd cluster verify --settings</code> CLI command output.</summary> Add the <code>recommended_value</code> column to the result of the <code>pgd cluster verify --settings</code> command. The column will not be displayed in tabular output but will be displayed in JSON output. </details>	
<details><summary>Display sorted output for CLI.</summary> 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. </details>	

continues on next page

Table 14 – continued from previous page

Description	Addresses
<details><summary>Subscriber-only nodes replication.</summary> 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+. </details>	
<details><summary>automatic node sync and reconciliation is enabled by default.</summary> The GUC <code>bdr 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. </details>	</code>
<details><summary>Remove the deprecated legacy CLI commands.</summary> Remove the old (PGD 5 and below) CLI commands, which were deprecated but supported for backward compatibility. </details>	
<details><summary>Commit scope logic is now only run on data nodes.</summary> Previously, non-data nodes would attempt to handle, but not process commit scope logic, which could lead to confusing, albeit harmless log messages. </details>	
<details><summary>Explicitly log the start and stop of dump and restore operations.</summary> This provides greater visibility into the node cloning process and assists with debugging possible issues. </details>	

6.2.5.4 Changes

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

6.2.5.5 Bug Fixes

Description	Addresses
<details><summary>Fix the CLI <code>pgd cluster show</code> command issues on a degraded cluster.</summary> The <code>pgd cluster show</code> 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, <code>health</code> and <code>summary</code>, while reporting an appropriate error for <code>clock-drift</code>. </details>	
<details><summary>Fix the CLI <code>pgd node show</code> command issue if a non-existent node is specified.</summary> The <code>pgd node show</code> command crashed if a non-existent node is specified to the command. The command is fixed to fail gracefully with appropriate error message. </details>	
<details><summary>Fix the broken replication slot issue after rolling Postgres upgrade using <code>pgd node upgrade</code> command.</summary> 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. </details>	46412,48747
<details><summary>Fixed the timestamp parsing issue for <code>pgd replication show</code> CLI command.</summary> The <code>pgd replication show</code> command previously crashed when formatting EPAS timestamps. </details>	
<details><summary>Fixed issue where parting node may belong to a non-existing group</summary> 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. </details>	
<details><summary>num_writers should be positive or -1</summary> The num_writers option, used in bdr.alter_node_group_option() and bdr.alter_node_group_config() should be positive or -1. </details>	
<details><summary>Fix replication breakage with updates to non-unique indexes</summary> Fixes the case where an update to a table with non-unique indexes results in the ERROR <code>concurrent INSERT when looking for delete rows</code>, which breaks replication. </details>	43523,43802,45244,47815
<details><summary>Fix Raft leader election timeout/failure after upgrade</summary> Ensure that any custom value set in the deprecated GUC <code>bdr.raft_election_timeout</code> is applied to the replacement <code>bdr.raft_global_election_timeout</code> </details>	
<details><summary>Ensure that disables subscriptions on subscriber-only nodes are not re-enabled</summary> 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. </details>	46519
<details><summary>Subscriber-only nodes will not take a lock when running DDL</summary> 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. </details>	47233
<details><summary>Fixed hang in database system shutdown.</summary> Fixed non-transactional WAL message acknowledgment by downstream that could cause a WAL sender to never exit during fast database system shutdown. </details>	49022

continues on next page

Table 16 – continued from previous page

Description	Addresses
<details><summary>Fixed deadlock issue in bdr_init_physical.</summary> Fixed deadlock between bdr_init_physical cleaning unwanted node data and concurrent monitoring queries. </details>	46952
<details><summary>Fixed new cluster node consistency issue.</summary> 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. </details>	
<details><summary>Ensure correct sequence type is displayed in CREATE SEQUENCE warnings</summary> In some cases, warning messages referred to <code>timeshard</code> when the sequence was actually <code>snowflakeid</code>. </details>	

6.2.6 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.

6.2.6.1 Bug Fixes

Description	Addresses
<details><summary>Fixed an issue with the slot names generated by <code>node setup</code> command.</summary> The command generated the slot names as UUID values with underscore (<code>_</code>) which is not compatible with PGD slot names. </details>	
<details><summary>Fixed segmentation fault in bdr.stat_connection_manager_get_stats().</summary> The memory returned by the function was allocated in an incorrect memory context. </details>	
<details><summary>Ensure that GRANT check for BDR objects works for EPAS object types</summary> 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. </details>	49649
<details><summary>Renamed the check <code>Clock Skew</code> to <code>Clock Drift</code> for <code>pgd cluster show</code> command.</summary> The check is renamed to be consistent with the terminology used at other places in PGD. </details>	
<details><summary>Added a warning message if CLI fails to fetch value of a GUC for <code>pgd cluster verify</code> command.</summary> The command could crash if the CLI could not fetch a value for a GUC for some reason. </details>	

6.3 Known issues and limitations

6.3.1 Known issues

These are currently known issues in EDB Postgres Distributed {{version.full}}. 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.full}}.0 do not get replicated after upgrading. To prevent this issue, you can do one of the following:
- Before upgrading to PGD {{version.full}}.0., manually create the materialized view on each node. - OR
- After upgrading to PGD {{version.full}}.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 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 CAMO .
- 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.

6.3.2 Limitations

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

6.3.2.1 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 *Architecture overview* .

6.3.2.2 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.

6.3.2.3 Durability options (Group Commit/CAMO)

There are various limits on how the PGD durability options work. These limitations are a product of the interactions between Group Commit and CAMO, and how they interact with PGD features such as the *WAL decoder* and 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.

6.3.2.4 General durability limitations

- Legacy synchronous replication uses a mechanism for transaction confirmation different from the one used by CAMO, Eager, and Group Commit. The two aren't compatible, so don't use them together. Whenever you use 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, or Eager because those features use two-phase commit underneath.

6.3.2.5 Group Commit

Group Commit 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.

6.3.2.6 Eager

Eager is available through Group Commit. It avoids conflicts by eagerly aborting transactions that might clash. 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`.

6.3.3 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.

6.3.3.1 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.

6.3.3.2 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.

CHAPTER 7

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.

7.1 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.
- ** *PGD compatibility* ** — 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.

7.2 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.

7.3 Other upgrades

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

7.3.1 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.

7.3.1.1 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.

7.3.1.1.1 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.

7.3.1.1.2 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.

7.3.1.1.3 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.

7.3.1.1.4 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 scope -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 scope 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.

7.3.1.2 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 *EDB Postgres Distributed 6 release notes* and any version-specific upgrade notes.

7.3.1.3 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.

7.3.1.3.1 BDR extension upgrade

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

7.3.1.3.2 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.

7.3.1.3.3 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.

7.3.1.3.4 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.

7.3.1.3.5 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.

7.3.1.3.6 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.full}}.

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

A PGD 5 cluster must be running PGD 5.9 before upgrading to PGD {{version.full}}.

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

7.3.1.3.7 Postgres upgrade

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.

7.3.1.3.8 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.

7.3.1.3.9 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.

7.3.1.4 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 after upgrading a node to confirm that the upgraded node is working as expected.

7.3.1.5 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 FROM pg_authid WHERE rolcanlogin = true AND rolpassword like SCRAM-SHA% LOOP

          password := rec.rolpassword; command := ALTER ROLE || quote_ident(rec.rolname) || WITH ENCRYPTED PASSWORD || quote_literal(password); EXECUTE command;

      END LOOP;

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

-

  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. .. Note:: 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.full}}, proceed with the *rolling upgrade steps* .
  ```

7.3.2 Supported PGD upgrade paths

7.3.2.1 Upgrading to version {{version.full}}

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.full}}:

Source version	Target version
4.4.1	{{version.full}}.0
5.9.0	{{version.full}}.0
6.0.2	{{version.full}}.0

7.3.3 Compatibility changes

7.3.3.1 Summary

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

7.3.3.2 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 .

7.3.3.3 DDL Locking changes

There is a new global lock type, `ddl_locking = leader` that's used by PGD {{version.full}} 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.

7.3.3.4 Join/part behavior

7.3.3.4.1 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.

7.3.3.4.2 Sequences conversion

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

7.3.3.4.3 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.

7.3.3.5 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.

7.3.3.6 Other

7.3.3.6.1 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.

7.3.3.6.2 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.

7.3.3.6.3 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

7.3.3.7 Changes to defaults

7.3.3.7.1 Global lock timeout

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

7.3.3.7.2 Auto sync

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

7.3.4 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.

7.3.4.1 Rolling application schema upgrades

By default, DDL is automatically sent to all nodes. You can control this behavior manually, as described in DDL replication . 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.

7.3.5 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.

7.3.5.1 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.

7.3.5.2 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.

7.3.5.3 Usage

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

7.3.5.3.1 `pgd node upgrade` command-line

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

7.3.5.3.2 Synopsis

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

7.3.5.3.3 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.

7.3.5.3.4 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.

7.3.5.3.5 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
```

7.3.5.3.6 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.

7.3.5.3.7 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*

7.3.5.3.8 pg_upgrade steps

Standard pg_upgrade steps are performed.

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

7.3.5.3.9 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 3 – continued from previous page

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

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

7.3.6.1 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.

7.3.6.1.1 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.

7.3.6.1.2 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-bdr6-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-checksums
```

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.
- **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` .

7.3.6.2 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.full}}.

7.3.6.2.1 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.full}}.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.full}}.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.

7.3.6.2.2 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
```

7.3.6.2.3 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
```

7.3.6.2.4 Stop the Postgres service

On the fenced node, stop the Postgres service.

7.3.6.2.5 Stop HARP manager

On the fenced node, stop HARP manager.

7.3.6.2.6 Remove and install packages

On the fenced node, remove PGD 4.4 and the cli packages and install the PGD {{version.full}} packages.

7.3.6.2.7 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.full}} with Connection Manager enabled.

7.3.6.2.8 Start HARP manager

On the fenced node, start HARP manager.

7.3.6.2.9 Unfence the node

Unfence the upgraded node from HARP:

```
test-pgd6major-d1:~ $ harpctl unfence test-pgd6major-d1
```

7.3.6.2.10 Repeat steps for all nodes

Repeat same steps on all other nodes.

7.3.6.2.11 Confirm cluster version

Confirm the updated cluster version is version 6001 by running `bdr.group_raft_details`.

7.3.6.2.12 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
```

(continues on next page)

(continued from previous page)

```
    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);
```

7.3.6.2.13 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)
```

7.3.6.2.14 Switch to Connection Manager

It should now be safe to switch your application to Connection Manager.

7.3.6.2.15 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.full}}.

7.3.6.3 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.full}}, EPAS 17 cluster using the `pgd node upgrade` command.

7.3.6.4 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
```

7.3.6.4.1 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
```

7.3.6.5 Pre-upgrade steps

7.3.6.5.1 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
```

7.3.6.5.2 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* .

7.3.6.5.3 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      |
```

7.3.6.5.4 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.

7.3.6.5.5 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.

7.3.6.5.6 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
```

(continues on next page)

(continued from previous page)

```
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
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*
```

7.3.6.5.7 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
```

(continues on next page)

(continued from previous page)

```
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
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
```

(continues on next page)

(continued from previous page)

```
Sync data directory to disk                                 ok
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.

7.3.6.6 Post-upgrade steps

7.3.6.6.1 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

[Service]
Type=simple
```

(continues on next page)

(continued from previous page)

```
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
```

7.3.6.6.2 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.

7.3.6.6.3 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.full}}.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.full}}.0 and the Postgres version to 17.6.0.

7.3.6.6.4 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.

7.3.6.6.5 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)
```

7.3.6.6.6 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
```

7.3.6.6.7 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 \
```

(continues on next page)

(continued from previous page)

```
- -new-datadir /var/lib/edb/as17/data \
- -username enterprisedb \
- -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
```

7.3.6.6.8 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.full}}.0 Essential  | 17.6.0
 pgd-a2     | PGD {{version.full}}.0 Essential  | 17.6.0
 witness-a1 | PGD {{version.full}}.0 Essential  | 17.6.0
```

All nodes of the cluster have been upgraded to PGD {{version.full}}.0 and EPAS 17.6.0.

7.3.6.6.9 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
- -----------
 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.