# EDB Postgres Distributed - PGD

リリース *6.1.0*

**EnterpriseDB Corporation**

**2025 年 08 月 26 日**

# 目 次

**第 1 章 Introduction to PGD** **3**
1.1 なぜ PGD なのか？ . . . 3
1.2 PGD は何を有効にしますか? . . . 3
1.3 PGD Essential と PGD Expanded の違いは何ですか？ . . . 3

**第 2 章 Get started with PGD** **5**
2.1 EDB Postgres Distributed とは何ですか? . . . 5
2.2 EDB Postgres Distributed Essential とは何ですか? . . . 5
2.3 PGD Essential Standard アーキテクチャとは . . . 5
2.4 Docker Compose を使用して最初の PGD Essential クラスターを作成する . . . 5
2.4.1 An introduction to PGD Essential . . . 5
2.4.2 Creating your first cluster (PGD Essential) . . . 7
2.4.3 First steps with your Quickstart PGD Cluster . . . 10
2.4.4 Expanded Examples and Use Cases . . . 25

**第 3 章 Essential How-To** **27**
3.1 概要 . . . 27
3.2 基本的な機能 . . . 27
3.3 基本的なハウツーガイド . . . 28
3.3.1 PGD Essential architectures . . . 28
3.3.2 Installing and configuring EDB Postgres Distributed 6 . . . 33
3.3.3 Connections . . . 45
3.3.4 Using PGD CLI . . . 46
3.3.5 Durability in PGD Essential . . . 53
3.3.6 Autopartitioning . . . 55
3.3.7 Production Best Practices . . . 56

**第 4 章 Expanded How-to** **59**
4.1 概要 . . . 59
4.2 拡張された機能 . . . 59
4.2.1 PGD Architectures . . . 59
4.2.2 Installing and configuring EDB Postgres Distributed 6 . . . 65

**第 5 章 PGD concepts explained** **79**
5.1 PGD の概念 . . . 79
5.2 PGD 拡張された概念 . . . 79
5.2.1 Replication . . . 79
5.2.2 PGD Nodes and Groups . . . 81
5.2.3 Connection Management . . . 82
5.2.4 Locking . . . 82
5.2.5 Durability . . . 83

5.2.6 Commit Scopes . . . 83
5.2.7 Lag Control . . . 83
5.2.8 Expanded Commit Scopes . . . 83
5.2.9 Geo-Distributed Clusters . . . 83
5.2.10 Advanced Durability . . . 84
5.2.11 Conflict Management . . . 84

**第 6 章 PGD Reference** **85**
6.1 ファンクションとコマンド . . . 85
6.2 構成と管理 . . . 85
6.3 高度な機能 . . . 86
6.3.1 PGD overview . . . 86
6.3.2 DDL replication . . . 94
6.3.3 Decoding worker . . . 115
6.3.4 CDC Failover support . . . 117
6.3.5 Parallel Apply . . . 121
6.3.6 Replication sets . . . 123
6.3.7 Security and roles . . . 132
6.3.8 Sequences . . . 141
6.3.9 Stream triggers . . . 151
6.3.10 Transaction streaming . . . 158
6.3.11 Explicit two-phase commit (2PC) . . . 161
6.3.12 Node types and capabilities . . . 162
6.3.13 Node management . . . 171
6.3.14 Connection Manager . . . 185
6.3.15 Postgres configuration . . . 192
6.3.16 Backup and recovery . . . 194
6.3.17 Monitoring . . . 199
6.3.18 AutoPartition in PGD . . . 214
6.3.19 Commit Scopes . . . 217
6.3.20 Conflict Management . . . 252
6.3.21 Testing and tuning PGD clusters . . . 289
6.3.22 Tables, views and functions reference . . . 291
6.3.23 EDB Postgres Distributed Command Line Interface (PGD CLI) . . . 468
6.3.24 Application use . . . 521

**第 7 章 Terminology** **533**
7.1 非同期レプリケーション . . . 533
7.2 コミットスコープ . . . 533
7.3 CAMO またはコミット・アット・モスト・ワンス . . . 533
7.4 競合 . . . 533
7.5 コンセンサス . . . 533
7.6 クラスター . . . 534
7.7 DDL データ定義言語 . . . 534
7.8 DML データ操作言語 . . . 534
7.9 熱心な . . . 534
7.10 最終的な一貫性 . . . 534

7.11 フェイルオーバー . . . 534
7.12 グループコミット . . . 534
7.13 即時一貫性 . . . 535
7.14 論理レプリケーション . . . 535
7.15 ノード . . . 535
7.16 ノードグループ . . . 535
7.17 PGD クラスター . . . 535
7.18 PGD ノード . . . 535
7.19 物理的レプリケーション . . . 536
7.20 Postgres クラスター . . . 536
7.21 クォーラム . . . 536
7.22 レプリケートされた利用可能なフォールトトレランス Raft . . . 536
7.23 読み取りスケーラビリティ . . . 536
7.24 サブスクリプション . . . 536
7.25 スイッチオーバー . . . 537
7.26 同期レプリケーション . . . 537
7.27 サブスクライバー専用ノード . . . 537
7.28 2 フェーズコミット 2PC . . . 537
7.29 垂直スケーリングまたはスケールアップ . . . 537
7.30 Witness ノード . . . 537
7.31 ライトリーダー . . . 538
7.32 ライター . . . 538

**第 8 章 PGD compatibility** **539**
8.1 PGD の PostgreSQL バージョンとの互換性 . . . 539
8.2 オペレーティングシステムおよびアーキテクチャとの PGD の互換性 . . . 539
8.2.1 Linux . . . 539

**第 9 章 EDB Postgres Distributed 6 release notes** **541**
9.1 EDB Postgres Distributed 6.1.0 release notes . . . 541
9.1.1 ハイライト . . . 541
9.1.2 機能 . . . 542
9.1.3 機能強化 . . . 543
9.1.4 の変更 . . . 543
9.1.5 バグ修正 . . . 544
9.2 EDB Postgres Distributed 6.0.2 release notes . . . 546
9.2.1 バグ修正 . . . 546
9.3 EDB Postgres Distributed 6.0.1 release notes . . . 546
9.3.1 ハイライト . . . 546
9.3.2 機能 . . . 547
9.3.3 機能強化 . . . 549
9.3.4 の変更 . . . 551
9.3.5 バグ修正 . . . 551

**第 10 章 Known issues and limitations** **555**
10.1 既知の問題 . . . 555
10.2 制限事項 . . . 556
10.2.1 ノード . . . 556

10.2.2 単一のインスタンス上の複数のデータベース . . . 556
10.2.3 耐久性オプショングループコミット/ CAMO . . . 557
10.2.4 一般的な耐久性の制限 . . . 557
10.2.5 グループコミット . . . 558
10.2.6 熱心な . . . 558
10.3 CAMO . . . 558
10.3.1 混合 PGD バージョン . . . 560
10.3.2 その他の制限 . . . 560

**第 11 章 Upgrading** **561**
11.1 PGD のアップグレード . . . 561
11.2 Postgres または Postgres と PGD メジャーバージョンのアップグレード . . . 561
11.3 その他のアップグレード . . . 561
11.3.1 Upgrading PGD clusters manually . . . 561
11.3.2 Supported PGD upgrade paths . . . 567
11.3.3 Compatibility changes . . . 567
11.3.4 Application schema upgrades . . . 567
11.3.5 In-place Postgres or Postgres and PGD major version upgrades . . . 569
11.3.6 Performing a Postgres major version rolling upgrade on a PGD cluster . . . 573

PGD 6.1 ドキュメントへようこそ。 PGD 6.1 は、Essential と Expanded の 2 つのエディションで入手できます。

# 第1章　Introduction to PGD

## 1.1 なぜ PGD なのか？

最新のデータアーキテクチャでは、要件が高可用性、ディザスターリカバリー、またはマルチリージョンのデータ分散であっても、データ管理への拡張可能なアプローチが必要です。PGD は、これらのニーズを満たすように設計されています。 PGD 6 では、PGD を使い始めやすいようにしながら、ユースケースがより複雑になるに応じて高度な機能を使用するための道を提供します。

## 1.2 PGD は何を有効にしますか?

PGD を使用すると、複数のリージョン、データセンター、またはクラウドプロバイダーにまたがる分散データベースアーキテクチャを構築できます。マルチマスターリプリケーションとデータ分散を提供します。Postgres データベースは、クラスター内のデータグループに展開でき、各ノード内のデータをマルチプルのノードに分散できます。

## 1.3 PGD Essential と PGD Expanded の違いは何ですか？

PGD Expanded は、PGD のフル機能バージョンです。これには、 PGD Essential のすべての機能に加えて、高度な競合管理、データ分散、大規模な展開のサポートなどの追加機能が含まれています。PGD Expanded は、PGD の最も高度な機能と機能を必要とするユーザーを対象に設計されています。

PGD Essential は、PGD Expanded の簡素化バージョンです。高度な機能や複雑な構成を必要とせずに、PGD をすばやく簡単に始めたいユーザー向けに設計されています。PGD Essential には、PGD のコア機能が含まれていますが、レプリケーションと可用性をシンプルにする方法でそれらを有効にします。したがって、PGD Expanded で使用可能なより高度な機能の一部は含まれていません。

PGD Essential は、クラスター内のデータノードの数を 4 に、グループの数を 2 に制限します。また、グループ内のノードの数を 4 に制限します。PGD Expanded には、これらの制限はありません。

Get Started with PGD の PGD の詳細については、こちらをご覧ください。

# 第2章　Get started with PGD

EDB Postgres Distributed のエディションの使用を開始するには、最初にローカルインストールと構成ガイドを試すことをお勧めします。

このガイドは、ソフトウェアのインストールと構成、および最初のクラスターを作成するのに役立ちます。

## 2.1 EDB Postgres Distributed とは何ですか?

EDB Postgres DistributedPGD は、PostgreSQL データベースの高可用性、スケーラビリティ、およびフォールトトレランスを提供する分散データベースソリューションです。これにより、連携して単一の統合データベースシステムを提供できる PostgreSQL インスタンスのクラスターを作成できます。

## 2.2 EDB Postgres Distributed Essential とは何ですか?

EDB Postgres Distributed Essential は、最小限の複雑さでコア分散データベース機能を提供することに焦点を当てた PGD の合理化バージョンです。これは、基本的な高可用性およびディザスターリカバリー機能を必要とするユーザー向けに設計されています。フルバージョンの PGD Expanded が提供する高度な機能はありません。

## 2.3 PGD Essential Standard アーキテクチャとは

*An introduction to PGD Essential* の EDB Postgres Distributed Essential がどのようなものであるかを理解します。

## 2.4 Docker Compose を使用して最初の PGD Essential クラスターを作成する

Docker Compose ファイルを 3 つのノードで *Creating your first cluster (PGD Essential)* に使用します。これは、PGD Essential の使用を開始し、現実のシナリオでどのように動作するかを確認するための優れた方法であり、PGD Essential または PGD Expanded を使用して運用クラスターを展開するための足がかりとなります。

### 2.4.1 An introduction to PGD Essential

EDB Postgres Distributed PGD Essential は、PGD Expanded の簡素化バージョンであり、分散データベースをすばやく簡単に開始できるように設計されています。PGD のコア機能を提供し、高度な構成の複雑さなしで高可用性とディザスターリカバリーを可能にします。

PGD のコアは、データノード、PGD クラスターの一部である Postgres データベースです。PGD により、これらのデータベースはノード間でデータを効率的にレプリケートし、データを常に利用可能な最新の状態に保つことができます。PGD Essential は、セットアップと管理が簡単な標準アーキテクチャを提供することにより、このプロセスを簡素化します。

標準アーキテクチャは、 EDB Postgres 分散システムの基本的なアーキテクチャ要素である単一のデータグループを中心に構築されています。グループ内のノードは協力して、着信書き込みまたは読み取りトラフィックを処理するノードを選択し、ノードが利用可能な場合、またはグループの残りの部分と同期していない場合を特定します。グループは、ノードが同じデータセンター内にあり、クラスター内に 1 つのグループのみがある単一のロケーションで最も一般的に使用されます。ワンロケーションアーキテクチャとも呼ばれます。

### 標準/ワンロケーションアーキテクチャ

1 ロケーションアーキテクチャは、3 つのノードを持つ単一の PGD クラスターで構成されています。ノードは同じデータセンターまたはリージョンにあります。理想的には別のアベイラビリティーゾーンにありますが、これは必須ではありません。ノードは、高速ネットワークを使用して相互に接続されています。

ノードはデータグループとして構成されます。これは、同じグループ内で相互にデータを複製することを意味します。PGD はネットワーク内のマルチプルのライターを処理できますが、これにはより高度な競合管理が必要で、PGD Essential ではサポートされていません。

したがって、標準アーキテクチャでは、1 つのノードが書き込みリーダーノードとして指定され、すべての書き込みトランザクションを処理します。グループ内の他のノードは、書き込みリーダーからデータを複製する読み取り専用ノードです。

書き込みリーダーノードは、すべての書き込みを処理するためにグループ内のノードによって選択された 1 つのノードです。書き込みトランザクションを受け入れ、グループ内の他のノードに複製する責任があります。書き込みリーダーノードに障害が発生すると、グループ内の他のノードが新しい書き込みリーダーノードを選択します。

アプリケーションは、すべてのデータノードで実行される PGD の接続マネージャーポートを使用して、クラスター内の任意のノードに接続できます。読み取りおよび書き込みトランザクションを書き込みリーダーに自動的にルーティングします。読み取り専用トランザクションをグループ内の他のノードにルーティングすることもできます。

この図では、接続マネージャーポートを介して PGD クラスターに接続しているアプリケーションが表示されます。接続マネージャーは、読み取りおよび書き込みトランザクションをグループ内の適切なノードにルーティングします。書き込みリーダーは、すべての書き込みトランザクションを処理する責任を負い、AZ1 の上部に緑色で表示されます。

グループ内の他のノードは、書き込みリーダーからデータを複製する読み取り専用ノードです。読み取り専用ノードの Connection Manager 読み取り/書き込みポートに接続しているアプリケーションは、クエリと変更が書き込みリーダーにルーティングされます。常に、ノードは互いに通信してデータを複製し、同期していることを確認します。

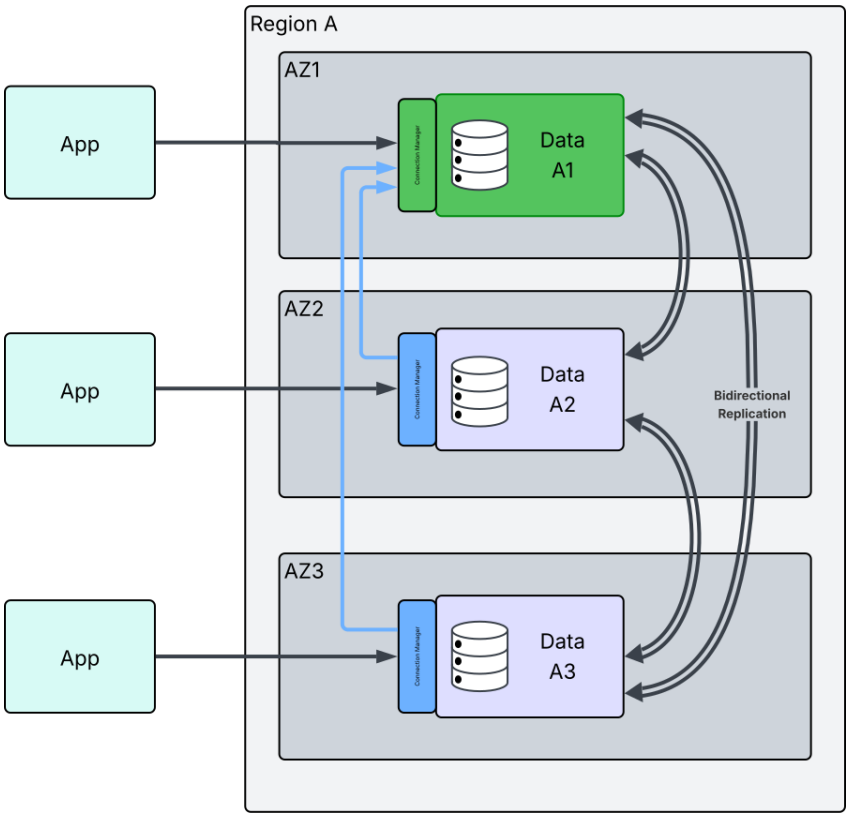


図 1: Standard architecture

### 2.4.2 Creating your first cluster (PGD Essential)

入門ガイドのこの部分は、Docker Compose を使用してローカルクラスターを作成するのに役立ちます。これは、EDB Postgres DistributedPGD Essential の機能と機能に慣れるための優れた方法です。

#### 前提条件

- ローカルマシンにインストールされている Docker および Docker Compose。

#### PGD Docker クイックスタートキットのインストール

最初の PGD クラスターを作成するには、EDB が提供する Docker Compose ファイルを使用できます。これにより、3 つのノードでローカルクラスターがセットアップされます。これは、テストと開発の目的に最適です。

1. ローカルマシンに Docker および Docker Compose がインストールされていることを確認します。Docker installation guide をまだ行っていない場合は、フォローできます。
2. ターミナルを開き、docker をインストールしたマシンで、PGD クラスターの新しいディレクトリを作成します。例

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

3. 次のコマンドを実行して、PGD Docker Compose ファイルをダウンロードします。

```
curl -L https://enterprisedb.com/docs/pgd/latest/get-started/assets/pgd_quickstart.sh␣
↪| bash
```

これにより、 PGD Docker クイックスタートキットがダウンロードされます。これには、Docker Compose ファイルおよび PGD Essential を開始するために必要なその他のファイルが含まれています。

4. ダウンロードが完了したら、PGD クラスターの環境を準備する必要があります。これは、次のコマンドを実行して実行します。

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

このコマンドは、PGD クラスターに必要なディレクトリとファイルを作成します。

5. 次に、PGD クラスターの Docker イメージをビルドする必要があります。これは、次のコマンドを実行して行うことができます。

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

このコマンドは、PGD クイックスタートクラスターに必要な Docker イメージをビルドします。

6. イメージがビルドされたら、Docker Compose を使用して PGD クラスターを起動できます。次のコマンドを実行します。

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

このコマンドは、Docker コンテナを起動し、バックグラウンドで実行されるデフォルト構成でローカルクラスターを作成します。

### PGD クラスターへのアクセス

1. コンテナが起動して実行されたら、次のコマンドを使用して PGD クラスターにアクセスできます。

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

このコマンドは、 `psql` コマンドラインインターフェイスを使用して、クラスターの最初のノードに直接接続します。

これは、メンテナンスおよび管理タスクのためにデータベースに接続する方法です。

アプリケーションとユーザーのアクセスには、通常、クラスター内のすべてのホストの TCP ポート 6432 で実行されている接続マネージャーを使用して接続します。

1. 次のコマンドを使用して、クラスター内の書き込みリーダーノードに接続できます。

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

`-h host-1` は、クラスター内のホストの名前に置き換えることができます。それらはすべて接続マネージャーを実行します。

ローカルマシンに psql クライアントをインストールしている場合は、次のコマンドを使用してクラスターに接続することもできます。

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

これは、ポート 6432 の host-3 コンテナで実行されている接続マネージャーに接続します。これは、クラスター内の書き込みリーダーノードにルーティングされます。

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

1. コンテナの外部から PGD CLI を使用するには、次のコマンドを実行できます。

```
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
```

この pgd コマンドは、クラスター内のノードとそのステータスをリストします。

host-1 コンテナでシェルを取得し、pgd コマンドを直接実行することもできます。

```
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
```

これにより、 PGD CLI にアクセスでき、host-1 コンテナで直接 PGD コマンドを実行できます。

### 次のステップ

最初の PGD クラスターを作成したため、次のトピックを検討できます。

- *Working with SQL and the PGD Cluster* SQL コマンドを使用してクラスターに接続および対話する方法を理解する。
- `COPY` コマンドまたは `pg_dump` および `pg_restore` を使用してクラスターに *Loading Data into your PGD Cluster* をインストールします。

- *Using PGD CLI* クラスターを監視および管理します。

### 2.4.3 First steps with your Quickstart PGD Cluster

最初の PGD クラスターが作成されたので、作業を開始できます。このガイドは、クラスターに接続し、データをロードし、基本的な SQL 操作を実行するのに役立ちます。

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

#### Working with SQL and the PGD Cluster

PGD クラスターを使用する最初のステップは、SQL を使用して接続することです。これは、 `psql` コマンドラインインターフェイスまたは PostgreSQL をサポートする他の SQL クライアントを使用して実行できます。

クイックスタートキットの Docker Compose セットアップを使用している場合、`psql` コマンドは既にインストールされ構成されています。つまり、ホストの 1 つでシェルを取得するだけで作業が開始できます。`host-1` コンテナにサインインして、クラスター内でコマンドを実行できます。

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

これにより、 `host-1` コンテナ内にシェルが提供され、PGD クラスターに対して PGD および SQL コマンドを実行できます。

#### PGD クラスター内の接続

PGD Essential では、メンテナンスタスクを実行する場合を除き、通常、クラスター内のすべてのホストの TCP ポート 6432 で実行されている接続マネージャーを使用してクラスターに接続します。

次のコマンドを使用して、クラスター内の書き込みリーダーノードに接続できます。

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

ユーザーなしで実行されている新しいクラスターは `postgres` スーパーユーザーを除き 、1 つのレプリケートデータベース `pgddb` があるため、次のコマンドを使用してクラスターに接続できます。

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

これは、ポート 6432 の `host-1` コンテナで実行されている接続マネージャーに接続し、クラスター内の書き込みリーダーノードにルーティングされます。`host-1` は、クラスター内のホストの名前に置き換えることができます。それらはすべて接続マネージャーを実行します。

次のコマンドを実行すると、クラスター内のどのノードに接続しているかを確認できます。

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

クラスター内で node-1 ノードを実行している host-1 コンテナに接続したため、これは驚くべきことではありません。

psql を終了し、次のように再接続すると

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

クラスター内の node-1 ノードに接続していることがわかります。

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

これは、クラスター内の書き込みリーダーノード node-1 にルーティングする接続マネージャーです。これを確認するには、次のことを実行できます。

```
\! 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      |
```

psql の\! コマンドを使用して、psql セッション内からシェルコマンドを直接実行できます。

### SQL での作業

クラスターに接続したので、SQL コマンドでの作業を開始できます。通常の PostgreSQL データベースを使用するのと同じように、テーブルを作成し、データを挿入し、クエリを実行できます。

たとえば、テーブルを作成し、いくつかのデータを挿入できます。

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

(次のページに続く)

(前のページからの続き)

```
(Alice, alice@example.com),
(Bob, bob@example.com);
```

次に、データを照会できます。

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

より複雑なクエリーを実行し、テーブルを結合し、PostgreSQL のすべての機能を使用することもできます。すべての SQL コマンドをカバーすることはこのガイドの範囲内ではありませんが、SQL 構文とコマンドの詳細については、 *PostgreSQL documentation* を参照してください。

### PGD との違い

重要なのは、これらの SQL コマンドがクラスター全体でレプリケートされることです。PGD がレプリケーションを処理します。たとえば、その `serial` キーはクラスター全体でグローバル一意キーに自動的に変換されているため、クラスター内の任意のノードにデータを挿入でき、それは他のすべてのノードにレプリケートされます。PGD Essential の場合、書き込みリーダーに接続する必要があるため、これはそれほど重要ではありませんが、PGD Expanded では、クラスター内の任意のノードに接続して SQL コマンドを実行できます。この自動変更により、サービスを心配せずにそれを行うことができます。競合または重複。PGD Essential を使用すると、将来も対応でき、SQL コマンドやアプリケーションコードを変更せずに、後で簡単に PGD Expanded に移行できます。

### 次のステップ

PGD クラスターに接続し、いくつかの SQL コマンドを実行したため、次のトピックを探索できます。

- *Loading Data into your PGD Cluster* 外部ソースからデータをインポートする方法を学習します。
- *Using PGD CLI* コマンドラインから PGD クラスターを管理します。

## Loading Data into your PGD Cluster

PGD は、その中核が Postgres データベースであるため、PostgreSQL データベースを使用するのと同じツールと方法を使用して、PGD クラスターにデータをロードできます。まず、このガイドでは、PGD クラスターにデータをロードするプロセスを説明します。

### オンライン CSV インポート

最初に、オンライン CSV ファイルから PGD クラスターにデータをインポートする方法を示します。この場合、それは Baseball Databank からの履歴野球データです。 psql の`\COPY` コマンドを使用して、URL から直接インポートします。 `\COPY` がしないことの 1 つはテーブルを作成することです。したがって、最初にテーブルを作成する必要があります。

psql を使用して PGD クラスターに接続します。docker compose exec host-1 psql を使用するか、psql がローカルにインストールされている場合は、それを使用してホストマシンのポート 6432 に接続します。

```
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)
);
```

これで、 \COPY コマンドを使用して CSV データを batters テーブルにインポートできます。

```
\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
```

このコマンドは、curl を使用して URL から CSV ファイルを取得し、\COPY コマンドに直接パイプします。このコマンドは、 batters テーブルにデータをインポートします。rotations(…) エントリーは、CSV データを行のどのフィールドに移動するかを定義します。DELIMITER ',' CSV HEADER オプションは、ファイルがスキップされるヘッダー行を含む CSV であることを指定します。

コマンドをコピーして、psql セッションに貼り付けます。すべてが正しく設定されている場合、データがエラーなしでインポートされるのが表示されるはずです。次のような、コピーされた行数を示す出力が表示されます。

```
COPY 110495
```

データが正しくロードされたことを確認するために、単純なクエリーを実行できます。

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

次のような結果が表示されるはずです。

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

これは、110,495 行が batters テーブルに正常にインポートされたことを確認します。

すぐにそれを使用して、1998 年のホームランリーダーが誰であったかを考えてみましょう

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

次のような出力が表示されるはずです。

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

そして、それを 1998 年の最高ランクのホームラン打者トップ 5 のコンテキストに置きたい場合は、次のことができます。

```
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;
```

次のような出力が表示されるはずです。

```
 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)
```

PGD を使用すると、ウィンドウファンクションのような高度な SQL 機能を含む PostgreSQL のフルパワー

を活用してデータを分析できますが、ノードがオフラインになった場合でもマルチプルのノードにわたって完全にレプリケートされ、可用性が高いという追加の利点があります。

### 次のステップ

いくつかのデータを PGD クラスターにロードしたので、次のトピックを探索できます。

- *Using PGD CLI* コマンドラインから PGD クラスターを管理します。

## Using PGD CLI

PGD CLI は、 EDB Postgres 分散 PGD クラスターを管理およびモニタリングするためのコマンドラインインターフェイスです。ノードの作成、ノードの参加、レプリケーションの管理など、クラスターでさまざまな操作を実行するためのコマンドセットを提供します。

*Quickstart Docker Compose kit* を使用している場合、それは既にインストールと構成されています。

PGD クラスターの最初のホストにログインします。

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

そして、 PGD CLI のバージョンを確認します。

```
pgd --version
__OUTPUT__
pgd-cli version 6.1.0
```

> **注釈**
>
> *docker compose exec* コマンドを使用して、PGD クラスターの最初のホストのコンテキストで実行することにより、コンテナの外部から次のコマンドを実行することもできます。

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

また、それらはすべて PGD CLI がインストールおよび構成されているため、クラスター内の任意のホストから pgd コマンドを実行できます。

pgd cluster show コマンドを使用して、クラスターの全体的なステータスを表示することから始めます。

```
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
```

(次のページに続く)

(前のページからの続き)

```
 group-1    | pgd          | data       | node-3    | data
 pgd        |              | global     |           |

#  Health

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

#  Clock Drift

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

このコマンドは、クラスター、そのノード、およびそれらの正常性ステータスの概要を提供します。また、ノード間のクロックドリフトも示します。これは、レプリケーションの一貫性にとって重要です。

pgd node show コマンドを使用して、個々のノードのステータスを表示することもできます。

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

(次のページに続く)

(前のページからの続き)

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

pgd CLI コマンドの構造は階層的で、コマンドは機能ごとにグループ化されています。次を実行して、使用可能なコマンドとその説明を表示できます。

```
pgd --help
__OUTPUT__
Manages PGD clusters

Usage: pgd [OPTIONS] <COMMAND>

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

Options:
  -V, --version  Print version

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

group 、node などのコマンドは、次の引数としてグループまたはノード名を取り、特定のコマンドが続きます。cluster 、groups 、nodes などのコマンドは、クラスターレベルで動作するか、すべてのグループまたはノードをリストするため、グループ名またはノード名を必要としません。

次のコマンドを実行して、特定のコマンドのヘルプを取得することもできます。

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

### クラスターステータスの表示

PGD クラスターの全体的なステータスを表示するために、既に pgd cluster show コマンドを使用しています。これは、すべてのクラスター情報を表示します。クラスターの正常性ステータスのみを表示するには、--health オプションを使用できます。

```
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
```

または、概要ステータスのみを表示する場合は、 --summary オプションを使用できます。

```
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     |           |
```

### グループとグループステータスの表示

クラスター内のすべてのグループのステータスを表示するには、pgd groups list コマンドを使用できます。

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

これで、トップレベルグループ pgd と、3 つのノードを含むデータグループ group-1 が表示されます。すべてのノードは、クラスター全体のすべてのアクティビティを調整するトップレベルグループのメンバーです。データグループ group-1 は、自分自身間でデータを複製し、グループ内の着信クエリーをグループ内の書き込みリーダーノードにルーティングする 3 つのデータノードのグループです。

pgd group show コマンドを使用して、グループの詳細を深く掘り下げることができます。

```
pgd group group-1 show
__OUTPUT__

#  Summary

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

#  Nodes

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

#  Options

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

(次のページに続く)

(前のページからの続き)

```
 route_writer_wait_flush           | false
 streaming_mode                    | default (inherited)
 use_https                         | true
```

このコマンドは、グループ、そのノード、およびそれらのステータスの概要を提供します。また、ルーティングが有効になっているかどうか、監視用の HTTP ポート、その他の構成設定などのグループオプションも表示します。

cluster コマンドと同様に、 --summary オプションを使用して、グループの概要のみを表示することもできます。

```
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      |
```

これで、グループがトップレベルグループ pgd の子であり、データグループであり、グループの書き込みリーダーノードが node-1 であることがわかります。このグループにはコミットスコープが設定されていません。これは、デフォルトのコミットスコープを使用していることを意味します。

--nodes オプションは、グループ内のノードを表示するために使用できます。

```
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
```

また、同様に、 --options オプションを使用してグループオプションを表示できます。

```
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
```

(次のページに続く)

(前のページからの続き)

```
 enable_wal_decoder                | false (inherited)
 http_port                         |  (inherited)
 location                          |
 num_writers                       | -1 (inherited)
 read_only_consensus_timeout       |  (inherited)
 read_only_max_client_connections  |  (inherited)
 read_only_max_server_connections  |  (inherited)
 read_only_port                    |  (inherited)
 read_write_consensus_timeout      |  (inherited)
 read_write_max_client_connections |  (inherited)
 read_write_max_server_connections |  (inherited)
 read_write_port                   |  (inherited)
 route_reader_max_lag              | -1
 route_writer_max_lag              | -1
 route_writer_wait_flush           | false
 streaming_mode                    | default (inherited)
 use_https                         | true
```

ご覧のとおり、オプションの多くは親グループであるトップレベルグループ pgd から継承されます。enable_raft および enable_routing オプションは true に設定されています。これは、グループがレプリケーションと書き込みリーダーノードへのルーティングクエリに Raft コンセンサスを使用していることを意味します。接続マネージャーポートを介して作成されます。

pgd group pgd show コマンドを使用して、親グループ pgd を見てみましょう。

```
pgd group pgd show
__OUTPUT__

#  Summary

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

これは、トップレベルグループ pgd がグローバルグループであることを示しています。これは、データグループではなく、独自のデータノードを持たないことを意味します。この場合、クラスター内のデータグループのアクティビティを調整するためにのみ使用されます。データノードがないため、書き込みリーダーはありません。

出力の次の部分には、グループ内のノードが表示されますが、空です。

```
#  Nodes
```

(次のページに続く)

(前のページからの続き)

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

pgd グループのオプションを次に示します。

```
#  Options

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

これらは、トップレベルグループ pgd のオプションです。これは、group-1 がオプションを継承する場所です。ただし、ここでは enable_routing オプションは false に設定されています。これは、トップレベルグループが独自のデータノードを持たないため、クエリーをデータノードにルーティングしないことを意味します。enable_raft オプションは true に設定されています。これは、トップレベルグループが Raft コンセンサスを使用してクラスターの管理を調整することを意味します。

オプションが設定されていない場合、デフォルト値が使用されます。たとえば、00:00:00 に設定される apply_delay オプション、つまりクラスターへの変更の適用に遅延がないことを意味します。

### ノードとノードのステータスの表示

クラスター内のすべてのノードのステータスを表示するには、pgd nodes list コマンドを使用できます。

```
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
```

pgd node show コマンドを使用して、特定のノードのステータスを表示することもできます。

```
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
```

ここで、ノード自分自身についての詳細を確認できます。ノードの名前とそれが属しているグループ、それがデータノードであること、グループにアクティブに参加していること、および実行中であることがわかります。ノード ID はノードの一意の識別子であり、Snowflake SeqID は、クラスター内のイベントを順序付けするために使用されます。最後に、そのデータベースが pgddb 、Quickstart Docker Compose キットで作成されたデフォルトデータベースであることがわかります。

ノードのオプションを次に示します。これらはこの特定のノードに固有です。

- route_dsn は、ノードの接続文字列であり、接続マネージャーがクエリーをこのノードにルーティングするために使用されます。
- route_fence は false に設定されています。これは、ノードにクエリのルーティングを防止するフェンス設定がないことを意味します。
- route_priority は-1 に設定されています。これは、ノードがルーティングクエリーに特定の優先度を持たないことを意味します。
- route_reads および route_writes は両方とも true に設定されています。これは、ノードが読み取りクエリと書き込みクエリの両方を処理できることを意味します。

これらは、接続マネージャーがクエリーをノードにルーティングするときに使用します。これらは、ダウンせずにアクティブなノードを制御する方法でもあります。route_fence を true に設定すると、接続マネージャーがクエリをこのノードにルーティングできなくなりますが、クラスターの一部でデータをレプリケートすることはできます。

### ノードオプションの設定

pgd node set コマンドを使用してノードのオプションを設定できます。たとえば、node-1 の route_fence オプションを true に設定するには、次のコマンドを実行できます。

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

node-1 の接続マネージャーに接続してみると

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

接続が得られます。ただし、ルーティングクエリーからフェンスされているため、node-1 ノードにルーティングされません。代わりに、グループ内の現在の書き込みリーダー node-2 にルーティングされます。

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

次を実行してフェンシングを終了して元に戻すと

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

node-1 ノードの接続マネージャーに再度接続できるようになりました。

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

そして、node-1 ノードに接続されたことがわかります。

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

(次のページに続く)

(前のページからの続き)

```
 node-1
(1 row)
```

### 2.4.4 Expanded Examples and Use Cases

PGD Essential は、高可用性および/またはディザスターリカバリーのユースケースを迅速に起動して実行するために必要なコア機能を提供しますが、PGD Essential で実装できる高度なユースケースが数多くあります。このセクションでは、これらの高度なユースケースのいくつかを実装する方法の例を提供します。

#### ユースケース

##### ユースケース 1 マルチマスターリプリケーション

デフォルトでは、 PGD Essential は PGD Connection Manager を使用して要求を適切なノードに送信します。このノードは書き込みリーダーであり、要求をそこに転送することにより、競合を迅速に解決できます。

PGD Expanded を使用すると、クラスター内の任意のノードに要求を送信できます。PGD は他のノードに変更をレプリケートします。構成可能な競合管理により、競合を解決する方法を選択できます。

##### ユースケース 2 データ分散

PGD Expanded を使用すると、サブスクライバー専用の読み取り専用ノードを含む、クラスター内の複数のノードにデータを分散できます。これらのノードは、複数のデータセンターまたはアベイラビリティーゾーンに配置できます。これにより、データベースの読み取り容量をホリゾンタルにスケーリングし、必要に応じてクラスターにノードを追加できます。

##### ユースケース 3: 地域分散

PGD Expanded では、マルチプルのリージョンにデータを分散し、クラスター内のすべてのノードにデータを複製できます。複数のデータグループをさまざまな場所に配置して、その場所での高可用性と復元力を保証できます。

##### ユースケース 4 階層化テーブル

PGD Expanded のオプショナル要素は、階層化テーブルを作成する機能です。これらのテーブルは、クラスター内でレプリケートされるホットデータと、Iceberg/Delta テーブルデータレイクに書き込まれるコールドデータ間のデータを階層化するために使用できます。階層テーブルは PGAA を使用するため、コールドデータは照会可能なままです。データベース内のテーブルであるかのようにデータレイクを照会できます。

# 第3章　Essential How-To

このセクションでは、PGD クラスターの展開と管理のための重要なハウツーガイドを提供します。これには、アーキテクチャ、展開、耐久性、自動パーティション、運用のベストプラクティス、および標準操作手順 SOP に関する情報が含まれています。

## 3.1 概要

PGD Essential は、PGD クラスターの展開と管理への簡素化されたアプローチを提供します。すばやく簡単に始められるように設計されており、ユースケースがより複雑になるにつれて、高度な機能を使用するための道を提供します。

PGD のコアは、データノード、PGD クラスターの一部である Postgres データベースです。PGD により、これらのデータベースはノード間でデータを効率的にレプリケートし、データを常に利用可能な最新の状態に保つことができます。PGD Essential は、セットアップと管理が簡単な標準アーキテクチャを提供することにより、このプロセスを簡素化します。

標準アーキテクチャは、 EDB Postgres 分散システムの基本的なアーキテクチャ要素である単一のデータグループを中心に構築されています。グループ内のノードは協力して、着信書き込みまたは読み取りトラフィックを処理するノードを選択し、ノードが利用可能な場合、またはグループの残りの部分と同期していない場合を特定します。グループは、ノードが同じデータセンター内にあり、クラスター内に 1 つのグループのみがある単一のロケーションで最も一般的に使用されます。ワンロケーションアーキテクチャとも呼ばれます。

## 3.2 基本的な機能

- Standard Architecture PGD Essential の標準アーキテクチャについて学びます。これは、同じデータセンターまたはリージョン内の 3 つのノードを持つ単一のデータグループで構成されます。
- Near/Far Architecture 異なる場所にある 2 つのデータグループで構成され、1 つのグループが書き込みを処理し、もう 1 つのグループが読み取りを処理する near/far アーキテクチャを理解します。
- Connection Management 接続マネージャーポートを使用して PGD クラスターに接続する方法を学習します。これは、読み取りおよび書き込みトランザクションを適切なノードに自動的にルーティングします。
- PGD CLI データグループ、ノード、接続の作成と管理を含む、PGD CLI を使用して PGD クラスターを管理する方法を説明します。
- Durability PGD Essential の耐久性機能を理解します。データは常に利用可能で最新の状態を保証します。

- Autopartition クラスター内のノード全体にデータを自動的にパーティション化して、パフォーマンスとスケーラビリティを向上させる autopartition 機能について説明します。

## 3.3 基本的なハウツーガイド

- *Simple PGD Essential Installation* PGD CLI を使用してシステムに PGD Essential をインストールするための詳細な手順を取得します。
- Production Best Practices パフォーマンスチューニングやモニタリングを含む、運用環境で PGD クラスターを展開および管理するためのベストプラクティスを取得します。

### 3.3.1 PGD Essential architectures

#### アーキテクチャの選択

PGD Essential には 2 つのサポートされているアーキテクチャがあります。Essential は、レプリケーションの 2 つの主要なユースケース、高可用性とディザスターリカバリーをサポートしています。どのアーキテクチャを選択するかは、ユースケースによって異なります。

それらは標準とニア/ファーです。

#### 標準アーキテクチャ - 高可用性の単一の場所に最適

標準または 1 つのロケーションアーキテクチャは、高可用性が必要な単一のロケーション用に設計されています。3 つのデータノードを中心に構築された Essential 標準アーキテクチャは、3 つのノードすべてでデータがレプリケートされ、障害が発生した場合にもシステムはデータを損失することなく動作を継続できます。

*PGD Essential Standard* アーキテクチャとは の詳細については、こちらをご覧ください。

#### ニア/ファーアーキテクチャ - ディザスターリカバリーに最適

Near/Far アーキテクチャは、適度に高可用性を必要とし、ディザスターから復旧できる必要がある単一の場所用に設計されています。これは、プライマリロケーションに 2 つのデータノードクラスターを配置し、セカンダリロケーションに単一のデータノードを配置することにより実行されます。

ニア/ファーアーキテクチャ - ディザスターリカバリーに最適 の詳細については、こちらをご覧ください。

> **注釈**
>
> マルチリージョン展開の場合 マルチリージョン展開の場合、

PGD Expanded で地域分散アーキテクチャを利用できます。これらのアーキテクチャは、複数のリージョンまたはデータセンターにデータを分散する必要があるユースケース向けに設計されています。競合解決、データ分散、大規模な展開のサポートなどの高度な機能を提供します。 PGD Expanded の詳細については、 Expanded how-to を参照してください。

### Standard PGD architecture

標準アーキテクチャは、コア PGD 機能を使用して、マルチマスターレプリケーション構成で 3 つのノードを構成します。つまり、各ノードはマスターノードとして動作し、そのデータを他のノードに論理的に複製します。PGD は、ノード上のデータ変更間の競合を処理できますが、Essential 標準アーキテクチャは PGD の統合接続マネージャーを使用して、すべての書き込みが単一のノード書き込みリーダーに送られるようにします。その特異なリーダーにデータのすべての更新を処理させることにより、競合が回避されます。変更は、クラスター内の他のノードに複製されます。

書き込みリーダーに障害が発生すると、クラスター内の残りのノードが新しい書き込みリーダーを選択し、これらのノードの接続マネージャーがフェイルオーバーして新しいリーダーに書き込みを送信します。障害が発生したノードがオンラインに戻ると、クラスターに再参加し、新しい書き込みリーダーからデータの複製を開始します。

Essential 標準アーキテクチャは、ユーザーエクスペリエンスに基づいて、展開と管理が簡単になるように作成されました。他の高可用性ソリューションとは異なり、Essential は PGD に基づいて構築されているため、より複雑なアーキテクチャへの移行はシンプルで簡単です。Expanded PGD に移動し、必要に応じて新しいデータグループをクラスターに追加します。

標準アーキテクチャの構成方法の詳細については、*Manually deploying PGD Essential standard architecture* を参照してください。

### Manually deploying PGD Essential standard architecture

PGD Essential 標準アーキテクチャの手動展開は、簡単なプロセスです。このアーキテクチャは、高可用性が必要で、ディザスターから復旧できる単一の場所向けに設計されています。これは、マルチマスターレプリケーション構成に 3 つのデータノードを持ち、1 つのノードが書き込みリーダーとして機能することにより実行されます。

#### PGD 構成

Essentials インストールガイドの手順を使用して、3 つのノードの各ノードに PGD をインストールします。具体的には

- Configure repositories PGD パッケージのインストールを有効にします。
- *Install PGD and Postgres* PGD パッケージをインストールします。
- *Configure the PGD cluster* PGD クラスターを構成します。

#### 実際の例

この例では、PGD Essential Standard アーキテクチャと次のパラメーターを使用して、EDB Postgres Extended Server で 3 ノード RHEL クラスターを作成します。

- 最初のノードは `node1` と呼ばれ、`host-1` にあります。
- 2 番目のノードは `node2` と呼ばれ、`host-2` にあります。
- 3 番目のノードは `node3` と呼ばれ、`host-3` にあります。
- クラスター名は `pgd` デフォルトの名前です。
- グループ名は `group1` です。

- Postgres のバージョンは 17 です。
- Postgres データディレクトリは/var/lib/edb-pge/17/main/ です。
- Postgres 実行可能ファイルは/usr/edb/pge17/bin/ にあります。
- Postgres データベースユーザーは postgres です。
- Postgres データベースポートは 5432 です。
- Postgres データベース名は pgddb です。

#### 最初のノードの場合

これは、ソフトウェアをインストールする 3 つのノードすべての共通設定です。

```
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
```

最初のノードで、次のコマンドはクラスターとグループを作成します。また、データディレクトリを作成し、データベースを初期化します。

```
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
```

#### 2 番目のノードの場合

2 番目のノードでソフトウェアのインストール手順を繰り返します。

次に、次のコマンドを実行してノードを初期化し、クラスターとグループに参加します。

```
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
↪"
```

### 3 番目のノードの場合

3 番目のノードでソフトウェアのインストール手順を繰り返します。

ノードを初期化し、クラスターとグループに参加するコマンドは、2 番目のノードに似ていますが、ホスト名とノード名が異なります。

```
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

ニア/ファーアーキテクチャでは、プライマリロケーションに 2 つのデータノード、セカンダリロケーションに 1 つのデータノードがあります。プライマリの場所は、データの大部分が保存され、ほとんどのクライアント接続が行われる場所です。セカンダリの場所はディザスターリカバリーに使用され、デフォルトではクライアント接続に使用されません。

データノードはすべて、標準のアーキテクチャと同様に、マルチマスターレプリケーション構成で構成されます。違いは、セカンダリロケーションのノードがクラスター内の他のノードからフェンスオフされ、デフォルトではクライアント接続を受け取らないことです。この構成では、セカンダリロケーションノードには、プライマリロケーションのデータの完全なレプリカがあります。

PGD コミットスコープを使用して、プライマリロケーションのデータノードは、プライマリロケーションの他のノードとセカンダリロケーションのノードにデータを同期レプリケートするように構成されます。これにより、データがプライマリロケーションにコミットされる前に、すべてのノードにレプリケートされます。ノードがダウンした場合、コミットスコープルールが状況を検出し、レプリケーションを非同期レプリケーションに低下させます。この動作により、システムは動作を継続できます。

プライマリロケーションで部分的な障害が発生した場合、システムは他のデータノードに切り替わり、同様にデータの完全なレプリカを使用して、動作を継続します。また、セカンダリの場所への複製も続行します。プライマリの場所で障害が発生したノードが復帰すると、再参加し、現在プライマリであるノードからデータの複製を開始します。

プライマリロケーションに完全な障害が発生した場合、セカンダリロケーションのデータベースにはデータの完全なレプリカがあります。障害に応じて、復旧のオプションには、セカンダリロケーションからプライマリロケーションの復元、またはセカンダリロケーションのバックアップからプライマリロケーションの復元が含まれます。セカンダリロケーションはクライアント接続を受け入れるように構成できますが、これはデフォルト構成ではないため、追加の再構成が必要です。

### Near/Far アーキテクチャでの同期レプリケーション

最良の結果を得るには、同期レプリケーションを使用してニア/ファーアーキテクチャを構成します。これにより、データがプライマリロケーションにコミットされる前に、セカンダリロケーションに複製されます。

同期レプリケーションを使用して near/far アーキテクチャを構成する方法の詳細については、*Manually Deploying PGD Essential near-far architecture* を参照してください。

### Manually Deploying PGD Essential near-far architecture

次の手順では、PGD Essential ニアファーアーキテクチャを手動で展開する方法について説明します。このアーキテクチャは、適度に高可用性を必要とし、ディザスターから復旧できる必要がある単一の場所用に設計されています。これは、プライマリロケーションに 2 つのデータノードクラスターを配置し、セカンダリロケーションに単一のデータノードを配置することにより実行されます。

これらの手順では、pgd コマンドラインツールを使用してクラスターを作成し、ノードを構成します。これらは、PGD Essential を既にインストールし、pgd コマンドラインツールにアクセスできることを前提としています。

プライマリロケーションは active ロケーションと呼ばれ、セカンダリロケーションは dr ロケーションと呼ばれます。

#### PGD 構成

プライマリロケーションは、独自のグループ「アクティブ」の 2 つのデータノードで構成されています。この場所は、大部分のクライアント接続が行われる場所です。

セカンダリロケーションは、独自のグループ dr 内の 1 つのデータノードで構成されています。

これらはすべて同じクラスターのメンバーです。

pgd-cli で作成したら、ノードのルーティングとフェンシングを構成する必要があります。

最初に、「active」グループと「dr」グループの両方でルーティングを無効にします。

```
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"
```

次に、「pgd」トップレベルグループでルーティングを有効にします。

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

最後に、「dr」グループのフェンシングを有効にします。

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

このアプローチにより、「dr」グループはクラスター内の他のノードからフェンスオフされ、デフォルトでクライアント接続を受け入れません。「active」グループは通常どおり動作し、「dr」グループにデータを複製し続けます。

### 3.3.2 Installing and configuring EDB Postgres Distributed 6

このセクションでは、 EDB Postgres Distributed 6 を手動で展開および構成する方法について説明します。

- *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*

#### 1 - Prerequisites for Essential installation

このガイドでは、システムに EDB Postgres DistributedPGD Essential をインストールする手順を説明します。

学習/テスト環境をインストールする場合は、 *PGD First Cluster* を使用することをお勧めします。

> **注釈**
>
> EDB Postgres DistributedPGD Expanded をインストールする場合は、 *Expanded installation guide* を参照してください。

##### プロビジョニングホスト

PGD を展開するプロセスの最初のステップは、ホストをプロビジョニングおよび構成することです。

Linux がインストールされているクラウドの仮想マシンインスタンス、Linux がインストールされているオンプレミス仮想マシン、または同じく Linux がインストールされているオンプレミスの物理ハードウェアに展開できます。

どの *supported Linux operating system* 、およびどの展開プラットフォームを選択しても、マシンのプロビジョニングの結果は、スーパーユーザー、管理者、または sudo 特権を持つユーザーに SSH を使用してアクセスできる Linux システムである必要があります。

プロビジョニングされた各マシンは、クラスターにプロビジョニングしている他のマシンに接続できる必要があります。

クラウド展開では、パブリックネットワークまたは VPC を介してこれを行うことができます。

オンプレミス展開は、ローカルネットワークを介して接続できる必要があります。

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

### 管理ユーザーを作成する

プロビジョニングされたインスタンスごとに管理ユーザーを構成することをお勧めします。管理ユーザーは、スーパーユーザーまたは sudo スーパーユーザーへの特権を持っている必要があります。また、証明書を使用してパスワードなしの SSH アクセス用に管理ユーザーを構成することをお勧めします。

### ネットワーク接続を確認する

作成した管理ユーザーを使用して、各マシンがプロビジョニングしている他のマシンと通信できることを確認します。

特に、 PostgreSQL TCP/IP ポート EDB Postgres Advanced Server の場合は 5444、EDB Postgres Extended およびコミュニティ PostgreSQL の場合は 5432、クラスター内のすべてのマシンで開いている必要があります。PGD 接続マネージャーは、クラスター内のすべてのノードからアクセスできることも必要です。デフォルトでは、接続マネージャーはポート 6432 または EDB Postgres Advanced Server の場合は 6444 を使用します。

### 実際の例

この一連の作業例では、Red Hat Enterprise Linux 9 を使用する 3 つのホストがプロビジョニングされました。

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

これらのホストはクラウドで構成されました。そのため、各ホストにはパブリック IP アドレスとプライベート IP アドレスの両方があります。クラスターのプライベート IP アドレスを使用します。

プライベート IP アドレスは次のとおりです。

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

クラスター例では、これらのプライベート IP アドレスを使用するように`/etc/hosts` も編集されました。

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

実稼働環境では、DNS を使用してホスト名を IP アドレスに解決する必要があります。

## Step 2 - Configure repositories

PGD データノードとして使用する各ホストで、データベースと PGD ソフトウェアをインストールする必要があります。

#### リポジトリの構成

次の環境変数を設定します。

#### EDB_SUBSCRIPTION_TOKEN

これは、EDB サブスクリプションに登録したときに受け取ったトークンです。EDB リポジトリへのアクセスを認証するために使用されます。

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

#### EDB_REPO_TYPE

これは、使用するパッケージマネージャーのタイプであり、インストーラーに、どのタイプのパッケージが必要であるかを通知します。これは、Ubuntu / Debian の場合は deb 、CentOS / RHEL の場合は rpm です。

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

#### リポジトリのインストール

構成する必要があるリポジトリは 2 つあります。1 つはデータベースソフトウェア用、もう 1 つは PGD ソフトウェア用です。

次のコマンドは、データベースの EDB リポジトリを使用するようにパッケージマネージャーを構成するスクリプトをダウンロードして実行します。

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

これにより、データベースソフトウェアのリポジトリがインストールされます。これには、EDB Postgres Extended Server およびその他の関連パッケージが含まれます。

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

このコマンドは、EDB リポジトリを使用するようにパッケージマネージャーを構成するスクリプトをダウンロードして実行します。また、必要な依存関係もインストールします。

#### 実際の例

この例では、標準サブスクリプションで PGD Essential を使用して EDB Postgres Extended Server 17 をインストールできる CentOS / RHEL システムにリポジトリを構成します。

#### 環境変数を設定する

```
export EDB_SUBSCRIPTION_TOKEN=XXXXXXXXXXXXXX
export EDB_REPO_TYPE=rpm
curl -1sSLf " https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/enterprise/
```

(次のページに続く)

(前のページからの続き)

```
↪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
```

次のステップは、 *Step 3 - Installing the database and pgd* です。

### Step 3 - Installing the database and pgd

PGD データノードとして使用する各ホストで、データベースと PGD ソフトウェアをインストールする必要があります。

*Step 2 - Configure repositories* を取得したら、パッケージマネージャーを使用してデータベースと PGD ソフトウェアをインストールできます。

#### データベースと PGD ソフトウェアのインストール

#### Postgres バージョンを設定する

環境変数を設定して、インストールする Postgres のバージョンを指定します。これは通常、Postgres 17 の 17 です。

```
export PG_VERSION=17
```

#### パッケージ名を設定する

環境変数を設定して、データベースと PGD ソフトウェアのパッケージ名を指定します。パッケージ名は、使用しているデータベースと使用しているプラットフォームによって異なります。

```
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 は、PGD Expanded でのみ動作可能です。

#### インストールコマンドを実行する

プラットフォームに適切なインストールコマンドを実行します。

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

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

このコマンドは、指定されたパッケージとそれに必要な依存関係をインストールします。インストールが完了すると、データベースと PGD ソフトウェアがシステムにインストールされます。

#### 実際の例

この例では、実際の例 で設定したリポジトリ構成を使用して、エンタープライズサブスクリプションを使用して CentOS / RHEL システムに PGD Essential を含む EDB Postgres Extended Server 17 をインストールします。

```
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
```

次のステップは、 *Step 4 - Configuring the cluster* です。

### Step 4 - Configuring the cluster

#### クラスターの構成

プロセスの次のステップは、データベースとクラスターを構成することです。

これには、各ホストにログインし、 pgd コマンドを実行してデータベースユーザーとしてクラスターを作成することが含まれます。

これらの手順は、使用しているプラットフォーム、および使用している Postgres のバージョンによって異なります。

#### クラスター名

クラスターの名前を選択する必要があります。これは、PGD CLI およびデータベースでクラスターを識別するために使用される名前です。例では<cluster-name> と呼ばれます。指定しない場合、デフォルト名は pgd です。

#### グループ名

グループの名前を選択する必要があります。これは、PGD CLI およびデータベースでグループを識別するために使用される名前です。例では<group-name> と呼ばれます。

グループ名はクラスター内で一意である必要があります。

#### ノード名

また、各ノードの名前を選択する必要があります。これは、PGD CLI およびデータベースでノードを識別するために使用される名前です。例では<node-name> と呼ばれます。これは、ノードが実行されているマシンの名前であるホスト名とは別です。

ノード名は、グループ内およびクラスター内で一意である必要があります。

### パスとユーザー

例で使用されるパスとユーザーは、使用している Postgres のバージョン、および使用しているプラットフォームによって異なります。

| |
|---|
| *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 は、PGD Expanded でのみ動作可能です。

### 各ホストで

上記のスクリプト/設定からコマンドを実行して、各ホストで Postgres ユーザーの環境変数とパスを設定します。これにより、pgd コマンドが Postgres 実行可能ファイルとデータディレクトリを見つけられるようになります。

1. 適切なユーザーを使用して、データベースユーザーとしてログインします。

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

1. Postgres バージョンの環境変数を設定します。 <version> を、17 など、使用している実際のバージョン番号に置き換えることを忘れないでください。

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

1. Postgres 実行可能ファイルをパスに追加します。

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

1. Postgres データディレクトリ環境変数を設定します。

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

1. Postgres パスワード環境変数を設定します。 <db-password> をデータベースユーザーに使用する実際のパスワードに置き換えることを忘れないでください。

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

### 最初のホストで

クラスター内の最初のホストは最初のノードでもあり、クラスターの作成を開始する場所になります。最初のホストで、次のコマンドを実行してクラスターを作成します。

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

このコマンドは、データディレクトリを作成してデータベースを初期化して、最初のノードにクラスターとグループを作成します。

#### 2 番目のホストで

2 番目のホストで、次のコマンドを実行してクラスターを作成します。

```
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>"
```

このコマンドは、2 番目のホストにノードを作成し、cluster-dsn 設定を使用してクラスターに参加し、最初のホストに接続します。

#### 3 番目のホストで

3 番目のホストで、次のコマンドを実行してクラスターを作成します。

```
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>"
```

このコマンドは、3 番目のホストにノードを作成し、cluster-dsn 設定を使用してクラスターに参加し、最初のホストに接続します。

#### 実際の例

この例では、前の手順で *Step 2 - Configure repositories* および *Step 3 - Installing the database and pgd* を作成した CentOS / RHEL システム上の EDB Postgres Extended Server 17 を使用して PGD Essential クラスターを構成します。

次に、 `node-1` 、`node-2` 、および `node-3` と呼ばれる 3 つのノードを使用して `pgd` と呼ばれるクラスターを作成します。

- グループ名は `group-1` になります。ホストは `host-1` 、`host-2` 、および `host-3` です。
- Postgres のバージョンは 17 です。
- データベースユーザーは `postgres` です。
- データベースポートは 5432 です。
- データベース名は `pgddb` です。
- Postgres 実行可能ファイルは`/usr/edb/pge17/bin/` にあります。
- Postgres データディレクトリは`/var/lib/edb-pge/17/main/` にあります。
- Postgres のパスワードは `secret` です。

Postgres バージョン環境変数 PG_VERSION は、前の手順で 17 に設定されていると仮定し、ユーザーを切り替えるときに環境変数を保存していることに注意してください。

#### 最初のホストで

```
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
```

#### 2 番目のホストで

```
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"
```

#### 3 番目のホストで

```
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"
```

次のステップは、 *Step 5 - Checking the cluster* です。

### Step 5 - Checking the cluster

#### クラスターの確認

クラスターが起動して実行されたら、いくつかの基本的なチェックを実行して、どの程度効率的に複製されているかを確認することは価値があります。

次の例は、これを行うための 1 つの簡単な方法を示していますが、実行するテストがユースケースに適切であることを確認する必要があります。

インストールおよび構成済みのノードで、ログインして `psql` を実行してデータベースに接続します。 EDB Postgres Advanced Server を使用している場合は、 `enterprisedb` ユーザーを使用します。それ以外の場合は、 `postgres` を使用します。

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

- **準備 クラスターの準備ができていることを確認します。**host-1/node-1 のデータベースにログインします。``*select bdr.wait_slot_confirm_lsn(NULL, NULL);*``**を実行します。**クエリーが返されると、クラスターの準備が整います。
- **データを作成する**クラスターがレプリケートしていることをテストする最も簡単な方法は、1 つのノードにログインし、テーブルを作成し、それに値を設定することです。*ノード 1 で、テーブルを作成します。

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

*ノード 1 で、テーブルにデータを入力します。

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

*ノード 1 で、パフォーマンスをモニタします。

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

*ノード 1 で、値列の合計を取得しますチェック用。

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

- **データの確認 ノード** *2* **にログインします。***host-2/node-2* **のデータベースにログインします。***ノード 2 で、値列の合計を取得しますチェック用。

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

**ノード** *1* **の結果と比較します。** ノード 3 にログインします。host-3/node-3 のデータベースにログインします。*ノード 3 で、値列の合計を取得しますチェック用。

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

*ノード 1 およびノード 2 の結果と比較します。

### 実際の例

#### 準備

host-1 の Postgres サーバーにログインします。

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

これは、PGD のノード 1 への接続です。

#### クラスターの準備ができていることを確認します

クラスターの準備ができていることを確認するには、次のコマンドを実行します。

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

このクエリは、クラスターが初期化でビジーな間はブロックし、クラスターの準備が整うと結果を返します。

別のウィンドウで、host-2 の Postgres サーバーにログインします。

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

### データを作成する

#### ノード 1 で、テーブルを作成します

実行します

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

#### ノード 1 で、テーブルにデータを設定します

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

このコマンドは、10000 行のランダム値のテーブルを生成します。

#### ノード 1 で、パフォーマンスを監視します

できるだけ早く実行します。

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

このコマンドは、そのデータが他の 2 つのノードに複製された速度に関する統計を表示します。

```
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)
```

そして、それは既に複製されています。

#### ノード 1 でチェックサムを取得します

実行します

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

このコマンドは、生成されたデータからいくつかの値を取得します。

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

### データを確認する

#### host-2 の Postgres サーバーにログインします

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

これは、PGD のノード 2 への接続です。

#### ノード 2 で、チェックサムを取得します

実行します

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

このコマンドは、生成されたデータのノード 2 の値を取得します。

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

#### ノード 1 の結果と比較します

値は同じです。

ノード 3 でプロセスを繰り返すか、任意のノードで新しいデータを生成し、他のノードにレプリケートされるのを確認できます。

#### host-3 の Postgres サーバーにログインします

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

これは、PGD のノード 3 への接続です。

#### ノード 3 で、チェックサムを取得します

実行します

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

このコマンドは、生成されたデータのノード 3 の値を取得します。

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

#### ノード 1 およびノード 2 の結果と比較します

値は同じです。

### 3.3.3 Connections

PGD Essential は、Postgres と同じ接続方法を使用します。違いは、クラスターへの接続のほとんどが、クラスター内のすべてのデータノードに組み込まれている接続マネージャーを経由することです。

データノードに直接接続することもできますが、メンテナンス以外の目的で、特定のノードのデータベースインスタンスで作業する場合はお勧めしません。

PGD Essential の場合、接続マネージャーを介してクラスターに接続する必要があります。PGD Essential は、展開と管理が簡単になるように設計されています。つまり、クラスターには、クラスターへのすべての書き込みを処理する書き込みリーダーノードがあることを意味します。接続マネージャーは、読み取り/書き込み接続を書き込みリーダーに転送します。クライアントまたはアプリケーションが行う必要があるのは、接続マネージャーのポートを使用することだけであり、接続マネージャーが残りを処理します。

接続マネージャーは、書き込みを書き込みリーダーに転送し、読み取りがクラスター内の正しいノードに転送されるようにします。データノードに直接接続する場合、これらの機能を利用できない場合があります。データを読み取ることのみが必要なアプリケーションの場合、接続マネージャーは、書き込みリーダーではないノードに読み取りを転送できます。これは、クラスターの負荷のバランスをとり、パフォーマンスを向上させるのに役立ちます。

#### 接続マネージャーを介して接続する

Postgres はポートと接続の構成が非常に柔軟であるため、簡単にするために、この例では Postgres と接続マネージャーのデフォルトのポート設定を使用します。Postgres のデフォルトポートは 5432 で、接続マネージャーのデフォルトポートは 6432 です。

接続文字列でそのポートを使用して、クラスターに接続できます。したがって、たとえば、psql コマンドラインツールを使用している場合、次のようにクラスターに接続できます。

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

host-1 は、接続しているノードのホスト名です。接続マネージャーは、接続をクラスター内の書き込みリーダーノードに転送します。

#### データノードに直接接続する

クラスター内のデータノードに直接接続できますが、それはお勧めしません。ただし、データノードに直接接続する必要がある場合は、次のコマンドを使用できます。

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

### 3.3.4 Using PGD CLI

PGD CLI は、PGD クラスターを管理するための強力なコマンドラインインターフェイスです。次のようなさまざまなタスクを実行するために使用できます。

- クラスターの状態の確認
- クラスター内のノードのリスト
- クラスター内のグループのリスト
- グループオプションの設定
- ライトリーダーの切り替え

installation guide を使用して PGD をインストールした場合、既に PGD CLI をインストールし、それを使用してクラスターを作成しています。

#### PGD CLI を使用する

PGD CLI コマンドは、構成ファイルを使用して、接続するホストを判断します。これをオーバーライドして代替構成ファイルを使用したり、サーバーを明示的にポイントしたりできる options があります。ただし、デフォルトでは、PGD CLI はプリセットの場所で構成ファイルを検索します。

データベースへの接続は、psql コマンドのような他のコマンドラインユーティリティが認証されるのと同じ方法で認証されます。

他のコマンドとは異なり、 PGD CLI は対話型でパスワードの入力を求めません。したがって、次のいずれかの方法を使用してパスワードを渡す必要があります。

- ホスト、ポート、データベース名、ユーザー名、パスワードを含むエントリを *.pgpass* に追加する

- PGPASSWORD 環境変数にパスワードを設定する
- 接続文字列にパスワードを含める

他のオプションはマルチプルのデータベースクラスターでうまく拡張できないか、パスワードの機密性が危険にさらされるため、最初のオプションをお勧めします。

### PGD CLI の構成と接続

- PGD CLI がインストールされていることを確認します。
- PGD CLI が既にインストールされている場合は、次のステップに移動します。
- システムの場合、そのシステムで *Step 2 - Configure repositories* ステップを繰り返します。
- 次に、そのプラットフォームに適切なパッケージインストールコマンドを実行します。
- RHEL および派生製品 `sudo dnf install edb-pgd6-cli`
- Debian、Ubuntu、およびデリバティブ `sudo apt-get install edb-pgd6-cli`
- 構成ファイルを作成します。 - これは、使用する PGD CLI のクラスターとエンドポイントを指定する YAML ファイルです。
- 構成ファイルのインストール.
- YAML 構成ファイルをデフォルトの構成ディレクトリ`/etc/edb/pgd-cli/` に `pgd-cli-config.yml` .
- としてコピーします PGD CLI を実行するシステムでこのプロセスを繰り返します。
- pgd-cli を実行します。

### PGD CLI を使用してクラスターを探索する

- `cluster show --health` コマンドでクラスターの状態を確認します。
- `nodes list` コマンドを使用して、クラスター内のノードを表示します。
- `groups list` コマンドでクラスター内のグループを表示する
- `group set-option` コマンドでグループオプションを設定します。
- `group set-leader` コマンドで書き込みリーダーを切り替えます。

これらのコマンドの詳細については、次の実際の例を参照してください。

また、他の構成オプションと完全なコマンドリファレンスの詳細については、PGD CLI documentation を参照してください。

### 実際の例

### PGD CLI がインストールされていることを確認します

この実際の例では、Postgres と PGD が既にインストールされているホスト 1 で PGD CLI を構成して使用します。 PGD CLI を再度インストールする必要はありません。

### オプションで構成ファイルを作成します

PGD CLI 構成ファイルは、クラスターオブジェクトを含む YAML ファイルです。これには 2 つのプロパティがあります。

- PGD クラスターのトップレベルグループの名前 `name` として
- データベースのエンドポイントの配列 `endpoints` として

```
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
```

この例のエンドポイントは `port=5444` を指定していることに注意してください。これは EDB Postgres Advanced Server インスタンスに必要です。 EDB Postgres Extended およびコミュニティ PostgreSQL の場合、これを省略できます。

PGD CLI 構成ディレクトリを作成します。

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

次に、構成を`/etc/edb/pgd-cli` ディレクトリの `pgd-cli-config.yml` ファイルに書き込みます。

この例では、host-1 でこれを実行してファイルを作成できます。

```
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
```

PGD CLI を使用する必要があるシステムでこのプロセスを繰り返すことができます。

### PGD CLI の実行

構成ファイルを配置し、enterprisedb system ユーザーとしてログインすると、pgd-cli を実行できます。たとえば、`nodes list` コマンドを使用して、クラスター内のノードとそのステータスをリストできます。

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

(次のページに続く)

(前のページからの続き)

```
node-2     group-1         data      ACTIVE     Up
node-3     group-1         data      ACTIVE     Up
```

### PGD CLI を使用してクラスターを探索する

PGD CLI が構成されたら、それを使用して、クラスターの PGD レベルのビューを取得できます。

#### クラスターの状態を確認する

*cluster show --health* コマンドは、クラスターの状態を表示する簡単な方法を提供します。

```
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
```

#### クラスター内のノードを表示する

前述のように、nodes list コマンドはクラスター内のノードをリストします。

```
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
```

このビューには、ノードがメンバーであるグループとその現在のステータスが表示されます。ノードで実行されている PGD および Postgres のバージョンを確認するには、`nodes list --versions` を使用します。

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

### クラスター内のグループを表示する

最後に、 PGD CLI の groups list コマンドは、構成されているグループなどを表示します。

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

このコマンドは次を示します。

- グループ
- そのタイプ
- その親グループ
- 各グループのノード数

### グループオプションを設定する

group set-option コマンドを使用して、PGD CLI を使用してグループオプションを設定することもできます。 group-1 グループの場所を London に設定する場合は、次を実行します。

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

group get-option コマンドを使用して、新しい場所を確認できます。

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

### 書き込みリーダーを設定する

グループ内の書き込みリーダーを変更して、ホストのメンテナンスを有効にする必要がある場合、PGD CLI は group set-leader コマンドを提供します。 group の後にグループ名を入力し、set leader の後にスイッチするノードの名前を入力します。

```
pgd group group-1 set-leader node-2
__OUTPUT__
Status Message
```

(次のページに続く)

(前のページからの続き)

```
- ----- ----------------------------
OK     Command executed successfully
```

`--summary option` で *group show command* を使用して、書き込みリーダーを検証できます。

```
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
```

PGD CLI で使用可能なコマンドの詳細については、PGD CLI command reference を参照してください。

### PGD エッセンシャルコンストレイン

#### クラスターアーキテクチャの検証

クラスターが以下にリストされている PGD Essential 制約を満たさない場合、`pgd cluster verify` コマンドは `Warning` を表示します。

- PGD Essential クラスターには、最大 3 つのデータノードが必要です。
- PGD Essential クラスターでは、グローバルグループのみにルーティングを有効にする必要があります。

```
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 互換性評価

`pgd assess` コマンドは、 PGD Essential の互換性について Postgres サーバーの以下にリストされている評価を実行しません。

- 複数の一意のインデックスを持つテーブル

- 行レベルのロックの使用
- ロックテーブルの使用法
- リッスン通知の使用法

PGD Essential クラスターのコマンド出力は以下のようになります。

```
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
```

### コミットスコープ管理コマンド

`pgd commit-scope create` 、`pgd commit-scope update` 、および `pgd commit-scope drop` コマンドは、PGD Essential クラスターではサポートされていません。コマンドは次のアドバイスを表示して終了します。

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

### グループオプションを設定する

`pgd group set-option` コマンドでは、 PGD Essential クラスターのグループオプション `enable_raft` 、`enable_routing` 、`enable_wal_decoder` 、および `streaming_mode` の更新はできません。コマンドは以下のアドバイスで終了します。

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

#### PGD ノードのセットアップ

`pgd node setup` コマンドは、`global` グループのルーティングを有効にし、`subgroup(s)` で同じを無効にします。

### 3.3.5 Durability in PGD Essential

デフォルトでは、 PGD Essential はノード間で非同期レプリケーションを使用しますが、同期レプリケーションを使用するように構成することもできます。これにより、データの耐久性と可用性の点で高い柔軟性が可能になります。非同期レプリケーションは、より低いレイテンシーとより高いスループットを提供しますが、同期レプリケーションは、パフォーマンスを犠牲にしてより強力な一貫性保証を提供します。PGD Essential では、コミットスコープを使用してレプリケーション戦略を選択できます。

#### コミットスコープ

コミットスコープは PGD Essential の強力な機能であり、データの耐久性と可用性を制御できます。これらにより、各トランザクションに必要な耐久性のレベルを指定できるため、アプリケーションのニーズに基づいてパフォーマンスと一貫性のバランスを保つことができます。PGD Essential には、トランザクションの持続性などを制御するために使用できる 4 つの事前定義されたコミットスコープがあります。

- ローカルプロテクト
- ラグプロテクト
- マジョリティプロテクト
- アダプティブプロテクト

PGD Essential の事前定義されたコミットスコープは、パフォーマンスとデータの安全性のバランスを提供するように設計されています。PGD Essential コミットスコープを追加、削除、または変更することはできません。PGD Expanded では、独自のコミットスコープを作成および管理できるため、耐久性保証の柔軟性と制御が可能になります。

#### ローカルプロテクト

これは、PGD Essential のデフォルトのコミットスコープです。耐久性保証のない非同期コミットを提供します。これは、トランザクションがローカルノードの WAL に書き込まれるとすぐに、クラスター内の他のノードからの確認を待たずにコミットされたと考えられることを意味します。

#### ラグプロテクト

このコミットスコープにより、遅延時間が指定された制限内この場合 30 秒であり、コミット遅延も指定された制限この場合 10 秒内である場合にのみトランザクションがコミットされたと見なされます。これは、ネットワークの問題またはノードに障害が発生した場合のデータの損失を防ぐのに役立ちます。

#### マジョリティプロテクト

このコミットスコープは、マジョリティオリジングループに基づいて耐久性保証を提供します。これにより、トランザクションは、オリジングループ内のノードの大部分によって確認された場合にのみコミットされたと見なされます。これは、ノードに障害またはネットワークの問題が発生した場合に、データの一貫性と耐久性を保証するのに役立ちます。

### アダプティブプロテクト

このコミットスコープは、より柔軟な耐久性保証を提供します。これにより、トランザクションはマジョリティオリジングループの同期コミットに基づいてコミットされたと考えられますが、指定されたタイムアウトこの場合 10 秒内にトランザクションを確認できない場合、非同期コミットに低下する可能性があります。これは、ネットワーク遅延またはノードの障害により、トランザクションの確認に遅延が発生する可能性があるシナリオで役立ちます。

コミットスコープの詳細については、Commit Scopes リファレンスセクションと Predefined Commit Scopes リファレンスページを参照してください。

## コミットスコープを使用する

PGD Essential でコミットスコープを使用するには、トランザクションの実行時に目的のコミットスコープを指定できます。これにより、アプリケーションのニーズに基づいてデータの耐久性と可用性を制御できます。たとえば、より高いレベルの耐久性を必要とするトランザクションに `lag protect` コミットスコープを使用し、耐久性よりパフォーマンスを優先するトランザクションに `local protect` コミットスコープを使用できます。

### トランザクション内

`SET LOCAL` コマンドを使用して、トランザクションのコミットスコープを指定できます。たとえば、トランザクションの `lag protect` コミットスコープを使用するには、次のコマンドを実行できます。

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

これにより、トランザクションが指定されたコミットスコープでコミットされることが保証され、目的のレベルの耐久性と可用性が提供されます。

### セッションの場合

`SET` コマンドを使用して、セッション全体のコミットスコープを設定することもできます。たとえば、セッション全体の `majority protect` コミットスコープを設定するには、次のコマンドを実行できます。

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

これにより、セッションで実行されるすべてのトランザクションが指定されたコミットスコープを使用し、目的のレベルの耐久性と可用性を提供します。

### グループの場合

`bdr.alter_node_group_option()` ファンクションを使用して、PGD グループのデフォルトのコミットスコープを設定することもできます。たとえば、PGD グループの `adaptive protect` コミットスコープを設定するには、次のコマンドを実行できます。

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

これにより、指定された PGD グループで実行されたすべてのトランザクションが指定されたコミットスコープを使用し、セッションまたはトランザクションレベルの設定でオーバーライドされない限り、目的のレベルの耐久性と可用性を提供します。

### 3.3.6 Autopartitioning

PGD の自動パーティショニングを使用すると、テーブルを複数のパーティションに分割できます。他のテーブル、これらのパーティションの作成と削除が必要です。自動パーティション化は、データを管理可能なチャンクに分割できるため、時間の経過とともに増加する大規模なテーブルの管理に役立ちます。たとえば、古いデータを独自のパーティションにアーカイブし、データが必要でなくなったらパーティションをアーカイブまたはドロップできます。

#### 自動パーティション化とレプリケーション

PGD 自動パーティションは、デフォルトで、 bdr.autopartition ファンクションを介してローカルに管理されます。この機能を使用すると、テーブルの自動レンジパーティションの定義を作成または変更できます。定義が存在しない場合、作成します。それ以外の場合、既存の定義を変更します。

#### レンジパーティショニング

PGD 自動パーティション化は、 RANGE キーワードを使用したレンジパーティション化をサポートしています。レンジパーティション化を使用すると、列の値の範囲に基づいてテーブルをパーティション化できます。たとえば、日付でテーブルをパーティション化できます。各パーティションには、特定の日付範囲のデータが含まれます。これは、データを管理可能なチャンクに分割できるため、時間の経過とともに増加する大規模なテーブルの管理に役立ちます。

たとえば、日付でパーティション化されたテーブルを作成できます。

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

次に、 bdr.autopartition ファンクションを使用して日次パーティションを作成し、データを 1 か月間保持できます。

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

この機能は、日ごとにパーティションを作成し、データを 30 日間保持します。30 日後、パーティションは自動的に削除されます。データベーステーブルを見ると、measurement テーブル用に作成されたパーティションが表示されます。

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

なぜこれほど多くのパーティションがあるのですか？デフォルトでは、自動パーティションは将来の使用に備えて 5 つの高度なパーティションを作成し、2 つのパーティションを除くすべてを使用するたびにそれを自動的に行うためです。これは、新しいデータに使用可能な少なくとも 2 つのパーティションがあることを意味します。bdr.autopartition ファンクションの minimum_advance_partitions および maximum_advance_partitions パラメーターを設定することにより、この動作を変更できます。

```
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);
```

### 3.3.7 Production Best Practices

運用環境に Postgres Distributed を展開する際に、従うべきベストプラクティスが多数あります。これらのプラクティスは、PGD クラスターの信頼性、パフォーマンス、およびセキュリティを保証するのに役立ちます。このセクションでは、運用環境に PGD を展開する際に考慮すべき重要なベストプラクティスのいくつかを概要します。

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

### Sizing

#### CPU/コアのサイジング

運用展開の場合、EDB は、Postgres データノードごとに少なくとも 4 コアを推奨します。Witness ノードはデータレプリケーション操作に参加せず、この要件を満たす必要はありません。サブグループ Raft なしで 1 つのコアで十分です。サブグループ Raft を使用する場合、2 つのコアで十分です。

ノードが昇格した場合のパフォーマンスの低下を回避するために、常にデータノードとまったく同じようにロジカルスタンバイのサイズを設定します。

特定のパフォーマンス要件の詳細なベンチマークを作成して、ワークロードに基づいて適切なサイジングを決定することをお勧めします。EDB プロフェッショナルサービスチームが、必要に応じてサポートします。

開発のために、2 コア未満の Postgres データノードを割り当てないでください。Barman ノードのサイズは、データベースのサイズとデータ変更率によって異なります。

Postgres データノードと Barman ノードは、仮想マシンまたはベアメタル展開モードで展開できます。ただし、復元性が低下するため、同じ物理ハードウェア上にある VM に複数のデータノードを展開しないでください。

### Time and PGD

#### 時計とタイムゾーン

EDB Postgres 分散は、マルチプルのタイムゾーンのノードで動作するように設計されており、真にワールドワイドなデータベースクラスターを実現します。個々のサーバーは一致するタイムゾーンで構成する必要はありませんが、`log_timezone = UTC` を使用して、人間が判読可能なサーバーログにアクセスしやすく比較しやすくすることをお勧めします。

NTP またはその他のソリューションを使用してサーバークロックを同期します。

他の一部のソリューションとは異なり、クロック同期はパフォーマンスにとって重要ではありません。クロックスキューはオリジンの競合検出に影響を与える可能性がありますが、EDB Postgres Distributed は、存在するスキューを報告および管理する制御を提供します。EDB Postgres Distributed は、Conflict detection で説明しているように、行バージョンの競合検出も提供します。

### Security Best Practices

# 第4章　Expanded How-to

## 4.1 概要

PGD Expanded は、完全な PGD 機能セットをユーザーに提供します。ここで、 PGD Essential は、PGD のベストプラクティス、制御、簡素化されたバージョンです。拡張バージョンは、高度なアーキテクチャ、カスタム構成、より複雑なユースケースを含む、PGD の機能と機能のフルセットを利用したいユーザーを対象としています。

PGD Expanded は、データベース環境の最高レベルの柔軟性と制御を必要とするユーザー向けに設計されています。ユーザーが展開をカスタマイズ、パフォーマンスを最適化できるようにする包括的なツールと機能のセットを提供します。

## 4.2 拡張された機能

PGD Expanded では次の機能が有効になっています。

- **マルチマスターレプリケーション** PGD Expanded はマルチマスターレプリケーションをサポートし、ユーザーは高可用性でフォールトトレラントのデータベース環境を作成できます。この機能により、ユーザーはクラスター内の任意のノードに書き込むことができ、最大限の柔軟性とスケーラビリティを提供します。
- **競合解決** PGD Expanded のマルチマスターレプリケーションのサポートには、高度な競合解決機能が含まれており、ユーザーはレプリケーション中に発生する可能性のある競合を処理できます。この機能により、クラスター内のすべてのノードでデータの一貫性が維持されます。
- **高度な耐久性** PGD Expanded は、PGD Essentials の事前定義されたコミットスコープを超えた柔軟性を提供する、カスタマイズ可能なコミットスコープを使用して、PGD の耐久性オプションのフルセットを開きます。この機能を使用すると、ユーザーは特定のニーズに基づいてデータベースのパフォーマンスと耐久性を最適化できます。
- **カスタム構成** PGD Expanded を使用すると、ユーザーは特定のニーズに合わせてデータベース構成をカスタマイズできます。PGD Essential は、制限された数のノードとグループで 2 つの基本アーキテクチャをサポートする場合、PGD Expanded 展開で使用できるノード、ノードタイプ、またはレプリケーション構成に制限はありません。

### 4.2.1 PGD Architectures

PGD 6 Expanded では、幅広いアーキテクチャでクラスターを展開できます。限られた数のグループで作成された 2 つのアーキテクチャに制限されている PGD 6 Essential とは異なり、 PGD 6 Expanded は、技術的に無制限のグループを含むマルチプルのアーキテクチャをサポートしています。

- *Always-On Architecture* 同じデータセンターまたはアベイラビリティーゾーン内の 2 つ以上のグループを持つ単一の PGD クラスター。このアーキテクチャは、高可用性とディザスターリカバリー用に設計されており、1 つのグループに障害が発生してもデータベースは動作し続けます。
- *PGD Essential architectures* 同じデータセンターまたはアベイラビリティーゾーンに 3 つのノードを持つ単一の PGD クラスター。PGD 6 Essential アーキテクチャ。
- *Multi-Location Architectures* 異なるデータセンターまたはアベイラビリティーゾーンにある 2 つ以上のグループを持つ単一の PGD クラスター。
- *Geo-Distributed Architectures* 異なるリージョンに 2 つ以上のグループを持つ単一の PGD クラスター。マルチロケーションアーキテクチャと似ていますが、レイテンシーが高く、潜在的なネットワーク分割の問題があります。

### Always-On Architecture

PGD のアーキテクチャは、組織のニーズを満たすために時間の経過とともに進化してきました。その中核は、Postgres データベースの高可用性とディザスターリカバリーを提供するように設計された Always-on アーキテクチャです。PGD 4 および 5 で定義された Always-on アーキテクチャは、PGD 6 の新しい機能と機能をサポートするように進化しました。

Always-on アーキテクチャは、EDB の Trused Postgres アーキテクチャを反映しています。これらはプラクティスをカプセル化し、複数の構成で可能な限り最高のサービスの可用性を達成するのに役立ちます。これらの構成は、単一ロケーションのアーキテクチャから、ハードウェア障害やデータセンターの障害から保護する複雑な分散システムにまで及びます。このアーキテクチャは、EDB Postgres Distributed のマルチマスター機能と、メンテナンス操作中でも 99.999%の可用性を達成する機能を活用します。

ここで説明する例を超えたアーキテクチャに EDB Postgres 分散を使用できます。ユースケース固有のバリエーションは、実稼働環境に正常に展開されました。ただし、これらのバリエーションは、最初に厳密なアーキテクチャレビューを受ける必要があります。

#### 標準 EDB Always-on アーキテクチャ

EDB は、目標復旧ポイント RPO および目標復旧時間 RTO の要件に応じて、さまざまなレベルの冗長性で単一またはマルチロケーションの展開をサポートするための標準化されたアーキテクチャを特定しました。

Always-on アーキテクチャは、基本的なビルディングブロックとして 3 つのデータベースノードグループを使用します。冗長性を高めるために 5 つのノードグループを使用することもできます。

EDB Postgres Distributed は、次の主要なビルディングブロックで構成されています。

- 双方向レプリケーション BDR—マルチマスターメッシュネットワークを作成する Postgres 拡張機能
- 接続マネージャー—アプリケーションが適切なデータノードに接続されていることを確認する接続ルーター。

すべての Always-on アーキテクチャは、増加する範囲の障害状況を保護します。たとえば、2 つのデータノードを持つ単一のアクティブなロケーションは、ローカルハードウェア障害から保護しますが、ロケーションデータセンターまたはアベイラビリティーゾーンの障害からの保護は提供しません。別の場所にあるバックアップを使用してそのアーキテクチャを拡張することにより、場所が壊滅的に損失した場合に備えて、ある程度の保護が保証されます。ただし、最初にバックアップからデータベースを復元する必要があるため、RTO 要件に違反する可能性があります。マルチマスターメッシュネットワークで接続された 2

番目のアクティブなロケーションを追加すると、ロケーションがオフラインになってもサービスは利用可能なままです。最後に、3 番目の場所これは監視専用の場所にすることができますを追加すると、1 つの場所がオフラインになった場合でもグローバル Raft 機能が動作できます。グローバル Raft は、主に管理コマンドを実行するために必要です。また、DDL やシーケンスの割り当てなどの一部の機能は、それなしでは動作しない場合がありますが、DML レプリケーションは、グローバル Raft なしでも動作し続けます。

データは少なくとも 1 つのローカルマスターに同期してストリーミングできるため、各アーキテクチャはゼロ RPO を提供でき、ローカルハードウェアに障害が発生した場合のデータ損失ゼロを保証します。

可用性保証を強化すると、ハードウェアとライセンス、ネットワーク要件、および操作の複雑性の追加コストが常に増加します。アーキテクチャを選択する前に、可用性とコンプライアンス要件を注意深く検討することが重要です。

### アーキテクチャの詳細

デフォルトでは、アプリケーショントランザクションは DML のクラスター全体のコンセンサス選択、挿入、更新、および削除を必要としないため、レイテンシーが低下し、パフォーマンスが向上します。ただし、新しいグローバルシーケンスの生成や分散 DDL の実行などの特定の操作の場合、EDB Postgres Distributed は Raft ベースのコンセンサスモデルを使用して決定を行うための奇数のノードが必要です。したがって、より単純なアーキテクチャでも、すべてのノードがデータを保存しているわけではない場合でも、常に 3 つのノードがあります。

アプリケーションは、マルチホスト接続文字列を介して標準の Always-on アーキテクチャに接続します。各接続マネージャーは、マルチホスト接続文字列の個別のエントリです。

他の接続メカニズムは実稼働環境に正常に展開されています。ただし、これらは標準の Always-on アーキテクチャの一部ではありません。

### 常時接続の単一ロケーション

- データノード 1 と 3 間の追加のレプリケーションは表示されていませんが、レプリケーションメッシュの一部として発生します
- ローカル障害から迅速に復元するための冗長ハードウェア *3 PGD* **ノード** 3 つのデータノードにすることができます推奨 *2* **つのデータノードと、データを保持しない** *1* **つの監視であることができます図示されていません**データノードの構成とインフラストラクチャの対称性は再ルーティングされたときにアプリケーションのワークロードを処理するために適切なリソースが利用できることを保証することが期待されます
- バックアップとリカバリーのための Barman 不図示**オフサイトはオプションですが推奨**マルチプルの PGD クラスターで共有できます
- モニタリング用の Postgres Enterprise ManagerPEM（図示されていません）*マルチプルの PGD クラスターで共有できます

### 常時接続のマルチロケーション

- アプリケーションは、各場所でアクティブ/アクティブにするか、アクティブ/パッシブまたは書き込みを行う 1 つの場所のみを使用するアクティブ DR にすることができます。
- データノード 1 と 3 間の追加のレプリケーションは表示されていませんが、レプリケーションメッシュの一部として発生します。

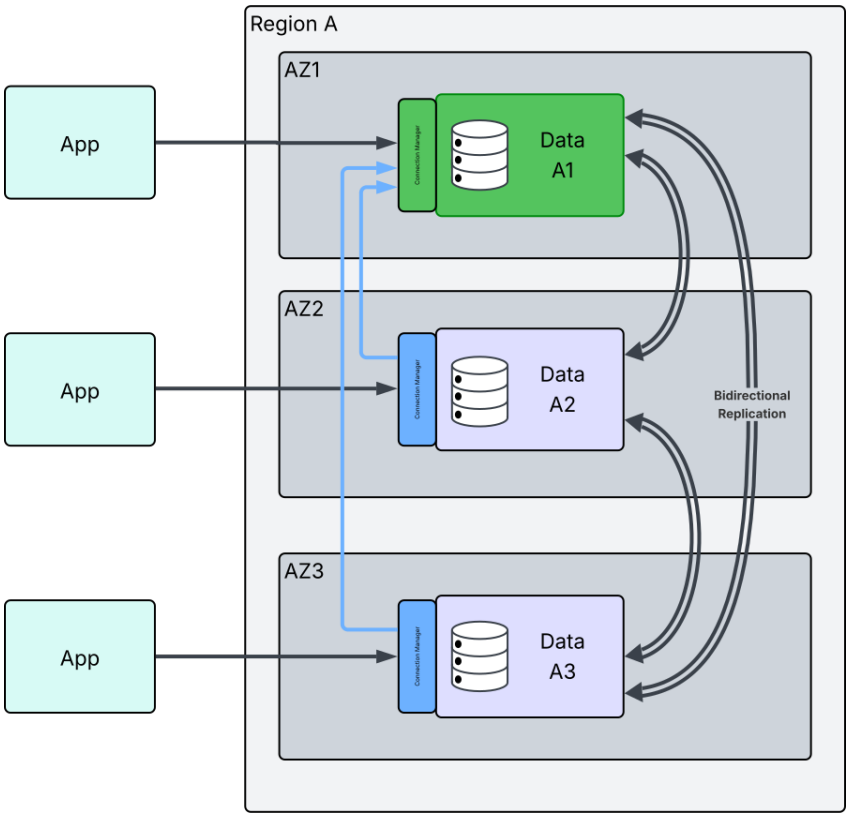


図 1: Always-on 1 Location, 3 Nodes Diagram

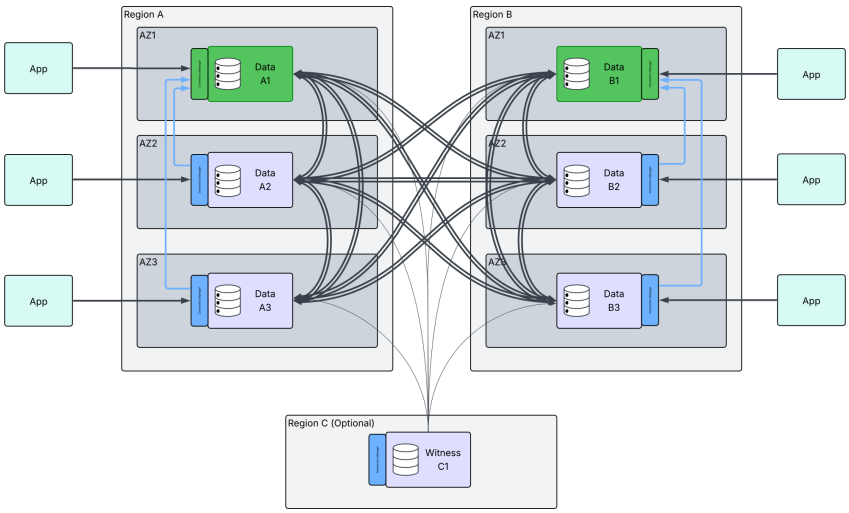


図 2: Always-on 2 Locations, 3 Nodes Per Location, Active/Active Diagram

- ローカル障害から迅速に復元するための冗長ハードウェア。**合計 *6* つの *PGD* ノード、各ロケーションに *3* つ** 3 つのデータノードにすることができます推奨 **2 つのデータノードと、データを保持しない *1* つの監視にすることができます不図示**データノードと場所の構成とインフラストラクチャの対称性は、次のとおりであると予想されます。再ルーティングされたときにアプリケーションのワークロードを処理するために適切なリソースが利用できることを確認
- バックアップとリカバリーの Barman 図示されていません。*マルチプルの PGD クラスターで共有できます
- モニタリングのための Postgres Enterprise ManagerPEM は示されていません。*マルチプルの PGD クラスターで共有できます
- オプションの監視ノードを 3 番目のリージョンに配置して、ロケーション障害の許容度を高める必要があります。*それ以外の場合、ロケーションに障害が発生すると、新しいノードや分散 DDL の追加など、グローバルコンセンサスを必要とするアクションがブロックされます。

### アーキテクチャの選択

すべてのアーキテクチャは以下を提供します。

- ハードウェア障害保護
- ゼロダウンタイムのアップグレード
- パブリック/プライベートクラウドでのアベイラビリティーゾーンのサポート

これらの基準を使用して、適切な Always-on アーキテクチャを選択します。

| | 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 |

### 標準アーキテクチャに柔軟性を追加する

必要なデータの復元力と、アプリケーションとデータを維持するユーザーへの近接性を提供するために、単一ロケーションアーキテクチャを必要なだけ多くの場所に展開できます。EDB Postgres 分散には、さまざまな競合処理アプローチが利用できますが、地理的に異なる場所からの書き込みアクティビティを許可する場合、予想される衝突数を最小限に抑えるように注意してください。

2 つの追加タイプのノードを使用して標準アーキテクチャを拡張することもできます。

- **サブスクライバー専用ノード**アプリケーションのワークロードの大部分が読み取り集中であり、書き込みが低い場合に、追加の読み取りスケーラビリティを実現し、データをユーザーの近くに配置するために使用できます。それらを活用して、レポート、アーカイブ、分析ニーズに合わせてデータのサブセットを公開することもできます。
- **論理スタンバイ** PGD クラスター内の別のノードからレプリケートデータを受信しますが、レプリケーションメッシュまたはコンセンサスに参加しません。これらには、他の PGD データノードとすべて同じデータが含まれており、データノードの 1 つがクラスターをフルキャパシティ/コンセンサスに戻すことができなかった場合、すぐにマスターに昇格できます。データセンター間のネットワークトラフィックが懸念される環境で使用できます。それ以外の場合、ロケーションごとに 3 つの PGD データノードが常に優先されます。

### Essential Architectures

PGD 6 Expanded は、Essential エディションの標準およびニアファーアーキテクチャを含む幅広いアーキテクチャをサポートしています。

Expanded を使用すると、Essential アーキテクチャを展開し、それにグループを追加したり、ニーズの拡大に応じてより複雑なアーキテクチャを構築したりできます。Essential アーキテクチャは、PGD のコア機能を提供しながら、展開と管理が簡単になるように設計されています。

Essential アーキテクチャについては、 Essential How-to で読むことができます。

### Multi-Location Architectures

PGD 6 Expanded は本質的に、データセンターやアベイラビリティーゾーンなどの複数の場所にまたがるアーキテクチャをサポートしています。これは Expanded エディションの重要な機能であり、障害を処理し、さまざまな地理的場所にわたって高可用性を維持できる堅牢で復元力の高い分散データベースを構築できます。

### Geo-Distributed Architectures

PGD は、複数の地理的および論理的な場所にまたがるクラスターをサポートします。これらのクラスターは、地理分散アーキテクチャと呼ばれます。

### 4.2.2 Installing and configuring EDB Postgres Distributed 6

このセクションでは、 EDB Postgres Distributed 6 を手動で展開および構成する方法について説明します。

- *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*

#### 1 - Prerequisites for Expanded installation

##### プロビジョニングホスト

PGD Expanded を展開するプロセスの最初のステップは、ホストをプロビジョニングおよび構成することです。

Linux がインストールされているクラウドの仮想マシンインスタンス、Linux がインストールされているオンプレミス仮想マシン、または同じく Linux がインストールされているオンプレミスの物理ハードウェアに展開できます。

どの *supported Linux operating system* 、およびどの展開プラットフォームを選択しても、マシンのプロビジョニングの結果は、スーパーユーザー、管理者、または sudo 特権を持つユーザーで SSH を使用してアクセスできる Linux システムである必要があります。

プロビジョニングされた各マシンは、クラスターにプロビジョニングしている他のマシンに接続できる必要があります。

クラウド展開では、パブリックネットワークまたは VPC を介してこれを行うことができます。

オンプレミス展開は、ローカルネットワークを介して接続できる必要があります。

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

##### 管理ユーザーを作成する

プロビジョニングされたインスタンスごとに管理ユーザーを構成することをお勧めします。管理ユーザーは、スーパーユーザーまたは sudo スーパーユーザーへの特権を持っている必要があります。また、証明書を使用してパスワードなしの SSH アクセス用に管理ユーザーを構成することをお勧めします。

#### ネットワーク接続を確認する

作成した管理ユーザーを使用して、各マシンがプロビジョニングしている他のマシンと通信できることを確認します。

特に、 PostgreSQL TCP/IP ポート EDB Postgres Advanced Server の場合は 5444、EDB Postgres Extended およびコミュニティ PostgreSQL の場合は 5432、クラスター内のすべてのマシンで開いている必要があります。PGD 接続マネージャーは、クラスター内のすべてのノードからアクセスできることも必要です。デフォルトでは、接続マネージャーはポート 6432 または EDB Postgres Advanced Server の場合は 6444 を使用します。

#### 実際の例

この一連の作業例では、Red Hat Enterprise Linux 9 を使用する 3 つのホストがプロビジョニングされました。

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

これらのホストはクラウドで構成されました。そのため、各ホストにはパブリック IP アドレスとプライベート IP アドレスの両方があります。クラスターのプライベート IP アドレスを使用します。

プライベート IP アドレスは次のとおりです。

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

クラスター例では、これらのプライベート IP アドレスを使用するように`/etc/hosts`も編集されました。

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

### Step 2 - Configure repositories

PGD データノードとして使用する各ホストで、データベースと PGD ソフトウェアをインストールする必要があります。

#### リポジトリの構成

次の環境変数を設定します。

#### EDB_SUBSCRIPTION_TOKEN

これは、EDB サブスクリプションに登録したときに受け取ったトークンです。EDB リポジトリへのアクセスを認証するために使用されます。

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

#### EDB_SUBSCRIPTION_PLAN

これは、EDB で持っているサブスクリプションのタイプです。 standard 、enterprise 、または community です。

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

#### EDB_REPO_TYPE

これは、使用するパッケージマネージャーのタイプであり、インストーラーに、どのタイプのパッケージが必要であるかを通知します。これは、Ubuntu / Debian の場合は deb 、CentOS / RHEL の場合は rpm です。

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

#### リポジトリのインストール

構成する必要があるリポジトリは 2 つあります。1 つはデータベースソフトウェア用、もう 1 つは PGD ソフトウェア用です。

次のコマンドは、データベースの EDB リポジトリを使用するようにパッケージマネージャーを構成するスクリプトをダウンロードして実行します。

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

次のコマンドは、PGD の EDB リポジトリを使用するようにパッケージマネージャーを構成するスクリプトをダウンロードして実行します。

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

#### 実際の例

この例では、エンタープライズサブスクリプションを使用して PGD Expanded で EDB Postgres Advanced Server 17 をインストールできる CentOS / RHEL システムにリポジトリを構成します。

#### 環境変数を設定する

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

#### リポジトリのインストール

```
#  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
```

次のステップは、 *Step 3 - Installing the database and pgd* です。

### Step 3 - Installing the database and pgd

PGD データノードとして使用する各ホストで、データベースと PGD ソフトウェアをインストールする必要があります。

*Step 2 - Configure repositories* を取得したら、パッケージマネージャーを使用してデータベースと PGD ソフトウェアをインストールできます。

#### データベースと PGD ソフトウェアのインストール

#### Postgres バージョンを設定する

環境変数を設定して、インストールする Postgres のバージョンを指定します。これは通常、Postgres 17 の 17 です。

```
export PG_VERSION=17
```

#### パッケージ名を設定する

環境変数を設定して、データベースと PGD ソフトウェアのパッケージ名を指定します。パッケージ名は、使用しているデータベースと使用しているプラットフォームによって異なります。

#### 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"
```

#### コミュニティ 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"
```

#### インストールコマンドを実行する

プラットフォームに適切なインストールコマンドを実行します。

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

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

このコマンドは、指定されたパッケージとそれに必要な依存関係をインストールします。インストールが完了すると、データベースと PGD ソフトウェアがシステムにインストールされます。

#### 実際の例

この例では、 実際の例 で設定したリポジトリ構成を使用して、CentOS / RHEL システムに PGD Expanded で EDB Postgres Extended Server 17 をインストールします。

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

### Step 4 - Configuring the cluster

#### クラスターの構成

プロセスの次のステップは、データベースとクラスターを構成することです。

これには、各ホストにログインし、 pgd コマンドを実行してデータベースユーザーとしてクラスターを作成することが含まれます。

これらの手順は、使用しているプラットフォーム、および使用している Postgres のバージョンによって異なります。

#### クラスター名

クラスターの名前を選択する必要があります。これは、PGD CLI およびデータベースでクラスターを識別するために使用される名前です。例では<cluster-name> と呼ばれます。指定しない場合、デフォルト名は pgd です。

### グループ名

グループの名前を選択する必要があります。これは、PGD CLI およびデータベースでグループを識別するために使用される名前です。例では<group-name> と呼ばれます。

グループ名はクラスター内で一意である必要があります。

### ノード名

また、各ノードの名前を選択する必要があります。これは、PGD CLI およびデータベースでノードを識別するために使用される名前です。例では<node-name> と呼ばれます。これは、ノードが実行されているマシンの名前であるホスト名とは別です。

ノード名は、グループ内およびクラスター内で一意である必要があります。

### パスとユーザー

例で使用されるパスとユーザーは、使用している Postgres のバージョン、および使用しているプラットフォームによって異なります。

Postgres バージョンを選択します

次に、プラットフォームを選択します。

| |
|---|
| *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/
```

(次のページに続く)

(前のページからの続き)

```
export PGDATA=/var/lib/edb/as$PG_VERSION/data/
export PGPORT=5444
```

次に、プラットフォームを選択します。

| |
|---|
| *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
```

次に、プラットフォームを選択します。

| |
|---|
| *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/
```

(次のページに続く)

(前のページからの続き)

```
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
```

### 各ホストで

上記のスクリプト/設定からコマンドを実行して、各ホストで Postgres ユーザーの環境変数とパスを設定します。これにより、pgd コマンドが Postgres 実行可能ファイルとデータディレクトリを見つけられるようになります。

1. 適切なユーザーを使用して、データベースユーザーとしてログインします。

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

1. Postgres バージョンの環境変数を設定します。 <version> を、17 など、使用している実際のバージョン番号に置き換えることを忘れないでください。

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

1. Postgres 実行可能ファイルをパスに追加します。

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

1. Postgres データディレクトリ環境変数を設定します。

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

1. Postgres パスワード環境変数を設定します。 <db-password> をデータベースユーザーに使用する実際のパスワードに置き換えることを忘れないでください。

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

#### 最初のホストで

クラスター内の最初のホストは最初のノードでもあり、クラスターの作成を開始する場所になります。最初のホストで、次のコマンドを実行してクラスターを作成します。

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

このコマンドは、データディレクトリを作成してデータベースを初期化して、最初のノードにクラスターとグループを作成します。

#### 2 番目のホストで

2 番目のホストで、次のコマンドを実行してクラスターを作成します。

```
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>"
```

このコマンドは、2 番目のホストにノードを作成し、cluster-dsn 設定を使用してクラスターに参加し、最初のホストに接続します。

#### 3 番目のホストで

3 番目のホストで、次のコマンドを実行してクラスターを作成します。

```
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>"
```

このコマンドは、3 番目のホストにノードを作成し、cluster-dsn 設定を使用してクラスターに参加し、最初のホストに接続します。

#### 実際の例

この例では、前の手順で *Step 2 - Configure repositories* および *Step 3 - Installing the database and pgd* を作成した CentOS / RHEL システム上の EDB Postgres Extended Server 17 を使用して PGD Essential クラスターを構成します。

次に、 `node-1` 、`node-2` 、および `node-3` と呼ばれる 3 つのノードを使用して `pgd` と呼ばれるクラスターを作成します。

- グループ名は `group-1` になります。ホストは `host-1` 、`host-2` 、および `host-3` です。
- Postgres のバージョンは 17 です。
- データベースユーザーは `postgres` です。
- データベースポートは 5432 です。
- データベース名は `pgddb` です。

- Postgres 実行可能ファイルは/usr/edb/pge17/bin/ にあります。
- Postgres データディレクトリは/var/lib/edb-pge/17/main/ にあります。
- Postgres のパスワードは secret です。

Postgres バージョン環境変数 PG_VERSION は、前の手順で 17 に設定されていると仮定し、ユーザーを切り替えるときに環境変数を保存していることに注意してください。

#### 最初のホストで

```
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
```

#### 2 番目のホストで

```
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"
```

#### 3 番目のホストで

```
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"
```

次のステップは、 *Step 5 - Checking the cluster* です。

### Step 5 - Checking the cluster

#### クラスターの確認

クラスターが起動して実行されたら、いくつかの基本的なチェックを実行して、どの程度効率的に複製されているかを確認することは価値があります。

次の例は、これを行うための 1 つの簡単な方法を示していますが、実行するテストがユースケースに適切であることを確認する必要があります。

インストールおよび構成済みのノードで、ログインして psql を実行してデータベースに接続します。 EDB Postgres Advanced Server を使用している場合は、 enterprisedb ユーザーを使用します。それ以外の場合は、 postgres を使用します。

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

このコマンドは、host-1/node-1 上のデータベースに **直接** 接続します。

### クイックテスト

- **準備 クラスターの準備ができていることを確認します。**host-1/node-1 のデータベースにログインします。 ``*select bdr.wait_slot_confirm_lsn(NULL, NULL);*`` **を実行します。**クエリーが返されると、クラスターの準備が整います。
- **データを作成する**クラスターがレプリケートしていることをテストする最も簡単な方法は、1 つのノードにログインし、テーブルを作成し、それに値を設定することです。*ノード 1 で、テーブルを作成します。

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

*ノード 1 で、テーブルにデータを入力します。

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

*ノード 1 で、パフォーマンスをモニタします。

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

*ノード 1 で、値列の合計を取得しますチェック用。

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

- **データの確認 ノード** *2* **にログインします。***host-2/node-2* **のデータベースにログインします。**ノード 2 で、値列の合計を取得しますチェック用。

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

**ノード** *1* **の結果と比較します。** ノード 3 にログインします。host-3/node-3 のデータベースにログインします。*ノード 3 で、値列の合計を取得しますチェック用。

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

*ノード 1 およびノード 2 の結果と比較します。

### 実際の例

#### 準備

host-1 の Postgres サーバーにログインします。

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

これは、PGD のノード 1 への接続です。

#### クラスターの準備ができていることを確認します

クラスターの準備ができていることを確認するには、次のコマンドを実行します。

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

このクエリは、クラスターが初期化でビジーな間はブロックし、クラスターの準備が整うと結果を返します。

別のウィンドウで、host-2 の Postgres サーバーにログインします。

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

#### データを作成する

#### ノード 1 で、テーブルを作成します

実行します

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

#### ノード 1 で、テーブルにデータを設定します

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

このコマンドは、10000 行のランダム値のテーブルを生成します。

#### ノード 1 で、パフォーマンスを監視します

できるだけ早く実行します。

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

このコマンドは、そのデータが他の 2 つのノードに複製された速度に関する統計を表示します。

```
pgddb=# select * from bdr.node_replication_rates;
__OUTPUT__
 peer_node_id | target_name | sent_lsn   | replay_lsn | replay_lag | replay_lag_bytes␣
```

(次のページに続く)

(前のページからの続き)

```
↪| 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)
```

そして、それは既に複製されています。

#### ノード 1 でチェックサムを取得します

実行します

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

このコマンドは、生成されたデータからいくつかの値を取得します。

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

### データを確認する

#### host-2 の Postgres サーバーにログインします

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

これは、PGD のノード 2 への接続です。

#### ノード 2 で、チェックサムを取得します

実行します

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

このコマンドは、生成されたデータのノード 2 の値を取得します。

```
pgddb=# select COUNT(*),SUM(value) from quicktest;
__OUTPUT__
```

(次のページに続く)

(前のページからの続き)

```
 count  |    sum
- -------+-----------
 100000 | 498884606
(1 row)
```

#### ノード 1 の結果と比較します

値は同じです。

ノード 3 でプロセスを繰り返すか、任意のノードで新しいデータを生成し、他のノードにレプリケートされるのを確認できます。

#### host-3 の Postgres サーバーにログインします

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

これは、PGD のノード 3 への接続です。

#### ノード 3 で、チェックサムを取得します

実行します

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

このコマンドは、生成されたデータのノード 3 の値を取得します。

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

#### ノード 1 およびノード 2 の結果と比較します

値は同じです。

# 第5章　PGD concepts explained

## 5.1 PGD の概念

- ユースケース *1* マルチマスターリプリケーション
- *PGD Nodes and Groups*
- *Connection Management*
- *Locking*
- *Durability in PGD Essential*
- コミットスコープ
- *Lag Control*

## 5.2 PGD 拡張された概念

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

### 5.2.1 Replication

EDB Postgres DistributedPGD の中心には、レプリケーションシステム BDR があります。BDR は Bi-Directional Replication の略で、マルチプルの書き込みノードを使用して分散 Postgres クラスターを作成できるマルチマスターレプリケーションシステムです。これは、クラスター内の任意のノードに書き込むことができ、変更はクラスター内の他のすべてのノードにレプリケートされることを意味します。

クラスター内の任意のノードに書き込むことができるからと、それを行う必要があるという意味ではありません。ほとんどの場合、クラスター内の単一のノードへの書き込みが必要になります。書き込みリーダーノードと呼ばれます。これは、クラスター内の他のすべてのノードへの変更のレプリケーションを調整する責任を負うノードです。実際、PGD Essential では、書き込みリーダーノードにのみ書き込むことができ、クラスター内の他のすべてのノードは読み取り専用です。

ただし、さまざまな場所に複数の書き込みノードがある地理分散クラスターを使用している場合など、クラスター内の複数のノードに書き込みが必要になる場合があります。これらの場合、BDR レプリケーションシステムを使用して、書き込みノード間で変更をレプリケートできます。これおよび他のシナリオは、PGD Expanded が設計されているものであり、これらのユースケースをサポートする追加の機能と機能をアクティブにします。

### レプリケーションの仕組み

PGD は、論理レプリケーションを使用して、クラスター内のノード間で変更をレプリケートします。これは、変更が物理レベルではなく論理レベルで複製されることを意味します。これにより、変更をレプリケートする方法の柔軟性が向上し、変更のより効率的なレプリケーションも可能になります。

クラスター内のテーブルに変更が行われると、最初に書き込みリーダーノードのログ先行書き込み WAL に書き込まれます。WAL は、データベースに行われたすべての変更のログであり、データベースの耐久性と一貫性を保証するために使用されます。変更が WAL に書き込まれると、クラスター内の他のすべてのノードに複製されます。

レプリケーションプロセスはデフォルトで非同期です。つまり、変更はクラスター内のすべてのノードにすぐにレプリケートされません。代わりに、変更はクラスター内の他のノードにバッチで送信されるため、より効率的なレプリケーションが可能になり、ネットワークの負荷が削減されます。

変更はクラスター内の他のノードが受信すると、データベースのローカルコピーに適用されます。このプロセスは WAL の再生と呼ばれ、クラスター内のすべてのノードがデータの一貫したビューを持つようにします。

### コミットスコープとレプリケーション

非同期レプリケーションはレプリケーションのデフォルトモードですが、唯一のモードではありません。PGD では、いわゆるコミットスコープを介して定義可能なレプリケーション構成ができます。コミットスコープは、トランザクションまたはグループ内のすべてのトランザクションに適用でき、クラスター内の他のノードに変更をレプリケートする方法を定義します。これにより、レプリケーションプロセスの動作を制御でき、パフォーマンスを最適化したり、変更が特定の方法でレプリケートされたことを確認したり、ネットワークとサーバーの不利な条件を正常に処理したりすることができます。

- PGD Expanded には、完全に定義可能なコミットスコープがあり、クラスター用のカスタムレプリケーション構成を作成できます。詳細については、*commit scopes in PGD Expanded* を参照してください。
- PGD Essential には、変更をレプリケートする方法を制御するために使用できる 4 つの事前定義されたコミットスコープがあります。詳細については、*commit scopes in PGD Essential* を参照してください。

### レプリケートとは何ですか?

PGD では、次の種類の変更がレプリケートされます。

- **データの変更**テーブルへの挿入、更新、および削除は、クラスター内のすべてのノードに複製されます。これは、DML データ操作言語レプリケーションと呼ばれます。
- **スキーマの変更**テーブルの作成や削除など、データベースの構造の変更も、クラスター内のすべてのノードにレプリケートされます。これは、DDL データ定義言語レプリケーションと呼ばれます。ただし、すべての DDL 変更がレプリケートされるわけではありません。たとえば、テーブルへの列の追加はレプリケートされますが、列の削除はレプリケートされません。
- **構成の変更**レプリケーション設定の変更など、データベースの構成の変更も、クラスター内のすべてのノードにレプリケートされます。

現在、PGD は、クラスターごとに 1 つの Postgres データベースのみをレプリケートします。これは、Postgres インスタンスに複数のデータベースがある場合、レプリケーション用に構成されたデータベースのみがクラ

スター内の他のノードにレプリケートされることを意味します。これは、PGD Essential と PGD Expanded の両方で同じです。

### 5.2.2 PGD Nodes and Groups

PGD クラスターは 1 つ以上のノードで構成され、各ノードは Postgres のインスタンスです。

クラスター内の各ノードは、BDR 拡張機能がインストールおよび構成された完全な Postgres インスタンスです。ノードには、クラスター内でさまざまなロールと責任を持つことができます。ノードは次に、ノード間のデータのレプリケーションを編成するために使用されるグループに編成されます。クラスター自分自身である「トップレベル」グループもあります。クラスター内のすべてのノードはこのグループのメンバーであり、クラスター内の他のすべてのグループの親です。

#### データノード

最初に知るノードの種類はデータノードです。これは、PGD クラスターの基本的なビルディングブロックです。クラスター内の他のデータノードとの間でデータを複製するように構成されています。グループではなく、クラスター。デザインでは、PGD クラスター内のすべてのノードは、クラスター内の他のすべてのノードにレプリケートします。

#### グループ

グループは、ノードが自分自身を **管理**する方法をローカライズするために使用されます。各グループは、グループメンバーから独自の RAFT リーダーを選択します。グループがデータノードで構成されるデータグループの場合、RAFT を使用してそのグループの書き込みリーダーノードを選択します。クライアント接続がそのグループ内のノードの接続マネージャーを介して着信すると、書き込みリーダーノードは、そのグループのすべての読み取り/書き込みクライアント接続が送信され、そのグループのすべての書き込み操作を処理するノードになります。

#### Witness ノード

Witness ノードはデータノードに似ていますが、データを複製または保存しません。彼らの役割は、データノードのグループが多数決を完了できないほど十分なノードを失った場合に、デッドロック打開投票を提供することです。

Witness ノードは通常のデータレプリケーションプロセスに参加しませんが、競合を解決し、ネットワーク分割またはその他の障害が発生した場合でもクラスターが利用可能な状態を維持するために使用できます。

#### サブスクライバー専用ノード

サブスクライバー専用ノードは、クラスター内のデータの読み取り専用レプリカを提供するために使用されます。PGD 6 では、加入者専用ノードをデータグループのメンバーまたは加入者専用グループのメンバーとして構成できます。後者には書き込みリーダーノードがなく、グループ内のすべてのノードは読み取り専用で、レプリケーションプロセスでいくつかのオプションの最適化が可能です。前者では、グループ内のデータの読み取り専用レプリカを許可しますが、レプリケーションプロセスでの最適化はできません。

サブスクライバ専用ノードを使用して、書き込みリーダーノードから読み取りクエリをオフロードできます。これは、書き込みリーダーノードのパフォーマンスを向上させ、負荷を削減するのに役立ちます。レ

ポートまたは分析の目的で、クラスター内のデータの読み取り専用レプリカを提供するために使用できます。接続マネージャーの読み取り専用接続文字列を使用して、このような読み取り専用ノードに接続できます。これにより、接続がクラスター内の読み取り専用ノードのプールに転送されます。

#### ロジカルスタンバイノード

論理スタンバイノードは、クラスター内のデータの読み取り専用レプリカを提供するために使用されます。これらは加入者専用ノードに似ていますが、より柔軟になるように設計されており、より幅広いシナリオで使用できます。

### 5.2.3 Connection Management

クライアントが分散クラスター内の適切なノードに接続できるように、 EDB Postgres DistributedPGD は、クライアントがニーズに基づいて適切なノードに接続できる接続管理システムを提供します。

このシステムは、クライアントがクラスターのパフォーマンスと可用性を維持しながら、必要なデータにアクセスできるように設計されています。プロキシシステムとは異なり、この接続管理システムはデータベースインスタンス自分自身に組み込まれているため、より効率的で信頼性の高い接続が可能になります。

実装の詳細については、 Connection Management feature in PGD の詳細をご覧ください。

### 5.2.4 Locking

クラスター内のさまざまな操作間の競合を防ぐため、PGD は分散ロックメカニズムを使用して、一度に 1 つのノードのみが特定の操作を実行できるようにします。

これは、マルチプルのノードが同じデータを同時に変更しようとする可能性がある分散環境で特に重要です。PGD Essential は単一書き込みノードクラスターであるため、いつでも書き込み操作を実行できるノードが 1 つだけであるため、同じ方法で分散ロックを扱う必要はありません。ただし、PGD Expaned には複数の書き込みノードがあるため、常に分散ロックを使用して整数を保証する必要があります。

#### ロックの種類

PGD は、数種類のロックを使用して、クラスター内のデータとリソースへの同時アクセスを管理します。

#### DDL ロック

DDL データ定義言語ロックは、テーブル、インデックス、スキーマなどのデータベースオブジェクトへのアクセスを管理するために使用されます。DDL 操作が実行されると、変更されるオブジェクトでロックが取得され、他の操作が変更を干渉しないようにします。これにより、データベースの構造の一貫性が保証され、同時 DDL 操作間の競合が防止されます。*DDL Locking reference* ドキュメントの DDL ロックの詳細については、こちらをご覧ください。

#### DML ロック

DML ロックは DDL ロックと密接に関連しており、DDL 操作と競合する DDL 操作も停止することにより、レプリケートされる DDL 操作に追加の保護レイヤーを追加するために使用されます。繰り返しますが、これはマルチ書き込みノードクラスターでのみ必要であり、PGD Essential では使用されません。

#### どのロックはいつ使用されますか?

PGD で使用されるロックは、実行される操作の種類とクラスターの構成によって異なります。通常、DDL ロックはスキーマの変更に使用され、DML ロックはデータ変更に使用されます。PGD で使用されるロックの完全なリストは、DDL command handling matrix ドキュメントにあります。

### 5.2.5 Durability

EDB Postgres DistributedPGD は、トランザクションの耐久性をどのように保証しますか？

持続性は、トランザクションがコミットされると、システム障害が発生した場合でもそのままであるという保証と定義できます。EDB Postgres 分散 PGD では、耐久性は、先行書き込みロギング WAL とレプリケーションの組み合わせ、クラスターで使用可能なコミットスコープとクラスター内のノードの構成と組み合わせて、実現されます。

### 5.2.6 Commit Scopes

### 5.2.7 Lag Control

ノードがクラスターの残りの部分よりも遅延している場合、データの一貫性と可用性に関する問題が発生する可能性があります。遅延制御は、遅延ノードがクラスターの全体的なパフォーマンスを混乱させないように保証することにより、この状況を管理するメカニズムです。

#### PGD のラグ制御

PGD で遅延が検出されると、Lag Control 機能がアクティブになります。この機能は、遅延ノードを管理し、クラスターの全体的なパフォーマンスを中断しないように設計されています。これは、透過的かつ一時的にクライアント接続を遅くすることにより、クライアントにコミット遅延を発生させます。これにより、遅延ノードは、他のノードのパフォーマンスに影響を与えることなく、クラスターの残りの部分に追いつくことができます。

詳細については、 *Lag Control feature in PGD* の詳細をご覧ください。

### 5.2.8 Expanded Commit Scopes

PGD Expanded では、PGD で使用可能な標準の事前定義されたコミットスコープよりも詳細なまたはよりカスタマイズされたコミットスコープを定義できます。この機能は、特定のコミット動作を必要とするアプリケーション、または複雑なトランザクションシナリオを管理する必要があるアプリケーションに特に役立ちます。

### 5.2.9 Geo-Distributed Clusters

地理分散クラスターは、複数の地理的場所にまたがる分散データベースシステムを作成できる PGD の強力な機能です。この設定は、さまざまなリージョンにわたる高可用性、低遅延、およびディザスターリカバリーを必要とするアプリケーションに特に役立ちます。この機能は複数の書き込みノードと複数の分散グループを必要とするため、PGD Expanded でのみ使用できます。

### 5.2.10 Advanced Durability

### 5.2.11 Conflict Management

PGD Expanded では、複数のライターが存在すると、競合の可能性、またはその可能性が発生します。別のノードから同じ行への変更は、いつでもノードに到着する可能性があります。PGD Expanded は、分散データベース環境で競合を処理する方法を定義できる競合管理システムを提供します。

競合管理の詳細については、 *Conflict Management reference* ドキュメントを参照してください。

# 第6章　PGD Reference

PGD Reference セクションでは、PGD の機能に関する詳細情報を提供します。

- Overview

PGD の特徴と機能の概要。

## 6.1 ファンクションとコマンド

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

## 6.2 構成と管理

- Connection Manager 接続マネージャーの構成と使用の詳細。PGD クラスターへのクライアント接続に単一のエンドポイントを提供します。
- Nodes

サブスクライバー専用ノードの詳細を含む、PGD クラスターに存在できるノードのタイプに関する情報。

- Node Management ノード、グループの追加と削除、クラスターステータスの検査など、PGD クラスター内のノードを管理する方法。
- Postgres Configuration

PGD クラスターで動作するように Postgres を構成する方法。

- *Backup and Restore*

PGD クラスターをバックアップおよび復元する方法。

- Monitoring ビルトイン監視ツールの使用や外部監視システムとの統合を含む、PGD クラスターを監視する方法。
- Commit Scopes

コミットスコープを使用して、分散環境でトランザクションの可視性を制御する方法。

- Conflict Management

競合解決戦略を含む、分散環境での競合を管理する方法。

- Autopartition

自動パーティションを使用して、PGD クラスター内のノードの冷却データを処理する方法。

- Testing and Tuning

最適なパフォーマンスを実現するための PGD クラスターのテストと調整に関するガイダンス。

- Upgrades

互換性とデータ移行に関する考慮事項を含む、PGD クラスターを新しいバージョンにアップグレードする方法。

## 6.3 高度な機能

- Application Usage アプリケーションで PGD の高度な機能を使用したい開発者向けのガイダンス。
- DDL 分散環境で DDL ファンクションを使用する方法。
- Decoding Worker PGD のレプリケーションでデコードワーカーを使用する方法。
- CDC Failover 一貫した変更データキャプチャ CDC のための PGD での論理スロットフェイルオーバーの使用。
- Parallel Apply パラレル適用を使用して分散トランザクションのパフォーマンスを向上させる方法。
- Replication Sets
- Security PGD のユーザー、ロール、および権限。
- Sequences アプリケーションで PGD の分散シーケンスを使用する方法。
- Stream Triggers

競合と変換に応答してアクションをトリガーします。

- Transaction Streaming

PGD でのリアルタイムデータレプリケーションにトランザクションストリーミングを使用する方法。

- Two-Phase Commit

PGD で明示的に 2 フェーズコミットをオプトインする方法。

### 6.3.1 PGD overview

EDB Postgres DistributedPGD は、高度な競合管理、データ損失保護、および throughput up to 5X faster than native logical replication を備えたマルチマスターレプリケーションとデータ分散を提供します。また、最大ファイブナインの高可用性を備えた分散 Postgres クラスターも有効にします。

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

### Architecture overview

EDB Postgres Distributed PGD は、PostgreSQL の機能を拡張し、マルチプルのノードにわたる高可用性とフォールトトレラントのデータベース展開を可能にする分散データベースソリューションです。PGD は、高度な競合管理、データ損失保護、最大ファイブナインの高可用性、およびネイティブ論理レプリケーションより最大 5 倍高速のスループットを備えたデータ分散を提供します。

PGD は、マルチマスター基盤双方向レプリケーション BDR 上に構築されており、Connection Manager を介してパフォーマンスと可用性が最適化されます。マルチマスター機能をより適切に利用するカスタム展開が必要な場合、接続マネージャーを使用せずに PGD を実行できます。接続マネージャーなしで実行されている場合、書き込みはノード間で分散されて相互に複製され、一貫性の維持には競合の解決が必要です。アーキテクチャのニーズによっては、これがより効率的な場合があります。ただし、接続マネージャーは、書き込みリーダーを使用して、競合と競合の低下を保証します。Raft は、システムがどのノードが Raft 選挙リーダーであり、どのノードが書き込みリーダーであるかを決定するなど、重要な決定を下すのを助けるために実装されています。

#### 高レベルアーキテクチャ

最高レベルでは、PGD は 2 つの主要コンポーネントで構成されています。双方向レプリケーション BDR と接続マネージャー。BDR は、異なる BDR 対応 Postgres インスタンス/ノード間のマルチマスターレプリケーションメッシュを有効にする Postgres 拡張機能です。接続マネージャーを介して接続する は、書き込みリーダーに要求を送信します。ノード間の競合のリスクの低下、より強い一貫性を保証します。

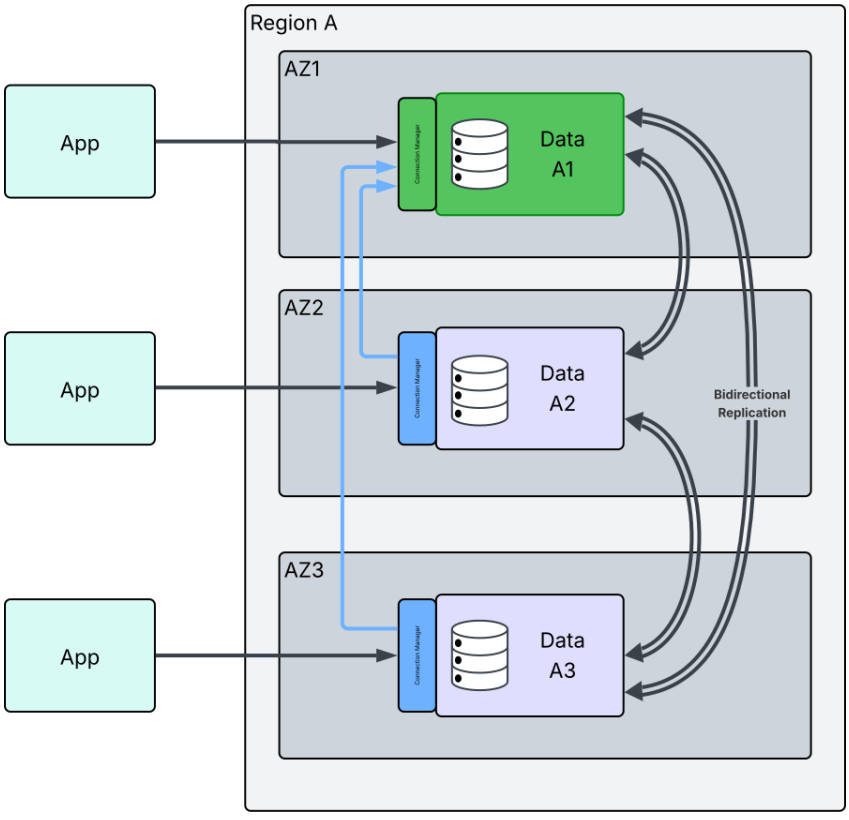


Diagram showing 3 application nodes and 3 PGD nodes. Traffic is being directed from each of the PGD nodes by the connection manatger to the write leader node.

変更は、すべてのノード間で行ごとに直接レプリケートされます。PGD の Logical replication はデフォルトで非同期であるため、最終的な一貫性のみが保証されます通常数秒以内。ただし、 commit scope オプショ

ンは、 CAMO 、 group および synchronous コミットを介して即時一貫性と耐久性を保証します。

Raft アルゴリズムは、リーダー Raft リーダーと書き込みリーダーの両方を選択し、クラスターに追加または削除するノードを決定するためのメカニズムを提供します。これにより、一般に、ノードに障害が発生した場合でも、分散システムの一貫性とフォールトトレラントが保証されます。

### アーキテクチャ要素

PGD は、分散データベースソリューションを提供するために連携するいくつかの主要なアーキテクチャ要素で構成されています。

- **PGD ノード**これらは、データを保存および管理する個々の Postgres インスタンスです。これらは、PGD クラスターの基本的なビルディングブロックです。- **グループ**デフォルトでは、すべてのノードは独自の Raft リーダーを持つ **トップレベルグループでのコミットスコープの作成** のメンバーでもありますが、書き込みリーダーはありません。PGD ノードはさらに *Groups and subgroups* に編成でき、管理性と高可用性が向上します。各グループには複数のノードを含めることができ、グループ内の冗長性とフェールオーバーが可能です。グループは、同じグループ内および異なるグループ間のノード間の組織的なレプリケーションとデータの一貫性を促進します。各グループには、独自の書き込みリーダーがあります。
- **レプリケーションメカニズム** PGD のレプリケーションメカニズムには、ノード全体で効率的なレプリケーションのための BDR が含まれ、マルチマスターレプリケーションを有効にします。BDR はデフォルトで非同期レプリケーションをサポートしますが、**耐久性オプショングループコミット**/ *CAMO* や **同期コミット** などのさまざまな同期レベルに構成して、データの耐久性を強化できます。
- **モニタリングツール** PGD でのパフォーマンス、状態、および使用状況をモニタするには、いくつかの便利なコマンドを提供する built-in command-line interface CLI を使用できます。例えば - pgd nodes list コマンドは、クラスター内のすべてのノードの概要を提供します。それらの状態とステータスを含みます。- *pgd cluster show --health* コマンドは、クラスターの状態を確認し、ノードのアクセス可能性、レプリケーションスロットの状態、およびその他の重要なメトリックを報告します。- pgd events show コマンドは、バックグラウンドワーカーエラーやノードメンバーシップの変更などの重要なイベントをリストするため、クラスター内の操作ステータスと問題を追跡するのに役立ちます。

さらに、 BDR 拡張機能を使用すると、*bdr.monitor_group_versions* ロールを使用して SQL を使用してクラスターをモニタリングできます。

### ノードタイプ

PGD のすべてのノードは事実上データノードです。これらは、クラスター内の目的が異なるだけです。

- **Data nodes ** データの保存と管理、読み取りおよび書き込み操作を処理し、レプリケーションに参加します。

次に、データノードのように構築されますが、特定の目的を持つ 3 種類のノードがあります。これらは次のとおりです。

- **Subscriber-only nodes ** 読み取り専用の目的でデータノードからの変更をサブスクライブします。レポートまたは分析で使用されます。
- **Witness nodes ** データを保存せずにコンセンサスプロセスに参加し、クォーラムの達成と高可用性の維持に役立ちます。

- **Logical standby nodes ** 必要に応じてデータノードに昇格できるスタンバイノードとして機能し、高可用性とディザスターリカバリーを保証します。

### ノードのロール

グループ内のデータノードは、特定の機能を有効にするために特定のロールを引き受けることもできます。これらのロールは一時的なもので、必要に応じてグループ内の他の対応可能なノードに転送できます。これらのロールには次のものが含まれます。

- **Raft leader** グループのノード間のコンセンサスを調停および管理します。
- **Write leader ** アプリケーションが接続マネージャーを介して接続するときにすべての書き込み操作を受け取ります。

### アーキテクチャの柔軟性

PGD は、さまざまなパフォーマンス、可用性、コンプライアンスのニーズを満たすようにアーキテクチャを展開、維持、およびスケーリングする方法に関する柔軟なオプションを提供します。

PGD は、Postgres のアップグレードとその他のシステムまたはアプリケーションレベルの変更の両方のブルー/グリーン展開を含むローリングメンテナンスをサポートしています。このアプローチにより、マイナーまたはメジャーバージョンのアップグレード、スキーマの変更、バキューム操作などの日常的なタスク中にデータベースが利用可能なままになります。システムは、アクティブなデータベースバージョンをシームレスに切り替え、ゼロダウンタイムを実現します。

PGD は、自動フェイルオーバーを提供して、高可用性を保証します。クラスター内のノードが使用できなくなると、別のノードがその責任を引き継ぎ、ダウンタイムを最小限に抑えます。また、PGD には自己修復機能が含まれており、障害が発生または切断されたノードはクラスターに再接続し、問題が解決されると通常のオペレーションを再開します。

PGD では選択的レプリケーションが可能であるため、ユーザーはデータのサブセットのみを特定のノードにレプリケートできます。この機能を使用して、ノード間の不要なデータトラフィックを削減することによりパフォーマンスを最適化したり、地理的データ制限などの規制要件を満たすことができます。たとえば、ヘルスケアアプリケーションは、地域のデータプライバシー法に準拠するために、特定のリージョン内の患者データのみを複製する場合があります。

コミットスコープを使用すると、PGD は構成可能な耐久性も提供します。したがって、耐久性をデフォルトの非同期動作から向上させ、さまざまな構成可能なコミットスコープを使用して調整できます。

- **Synchronous Commit ** 基になるオペレーションでの PostgreSQL の synchronous_commit オプションとよく似ています。COMMIT 時に少なくとも 1 つの他のノードに書き込む必要がありますが、すべてのノードを必要とするように調整できます。
- **CAMO ** Commit At Most Once の一意の ID で各トランザクションを追跡し、ノードのペアを使用してトランザクションの結果を確認することにより動作し、アプリケーションがトランザクションを再試行するかどうかを認識します。
- **Group Commit ** 実験的なコミットスコープ。その目標は、COMMIT 時にトランザクションを正常に確認するために複数の PGD ノードを必要とすることにより、一時的な停止の単一ノードに障害が発生した場合のデータ損失から保護することです。

- **Lag Control ** レプリケーションが設定された制限の外で実行されている場合別のノードにレプリケートされるのに時間がかかりすぎる場合、最初にトランザクションを受け取ったノードに遅延が注入され、他のノードが追いつくまで速度が低下します。

## PGD overview - architecture and performance

### アーキテクチャオプションとパフォーマンス

#### 常時接続アーキテクチャ

多数の異なるアーキテクチャを構成できますが、それぞれが異なるパフォーマンスとスケーラビリティ特性があります。

グループは、2 つ以上のノードサーバーで構成される基本的なビルディングブロックです。グループでは、各ノードは専用のルーターとバックアップを備えた異なるアベイラビリティーゾーンにあり、即時のスイッチオーバーと高可用性を提供します。各グループには、専用のレプリケーションセットが定義されています。グループがノードを失った場合、グループから既存のノードをコピーすることにより、簡単に修復または置換できます。

Always-on アーキテクチャは、1 つの場所にある 1 つのグループ、または 2 つの別の場所にある 2 つのグループから構築されます。各グループは高可用性を提供します。2 つのグループが遠隔地で活用される場合、それらは一緒にディザスターリカバリー DR も提供します。

テーブルは両方のグループにまたがって作成されるため、変更はローカルグループのノードだけでなく、すべてのノードに適用されます。

各グループの 1 つのノードがグループの書き込みリーダーとして選択されます。プロキシは、アプリケーションの書き込みと照会を書き込みリーダーに転送します。他のノードは書き込みリーダーのレプリカです。任意の時点で、書き込みリーダーが使用できない場合、グループ内の残りのノードはグループから新しい書き込みリーダーを選択し、プロキシはトラフィックをそのノードに転送します。スケーラビリティはこのアーキテクチャの目標ではありません。

書き込みは主に 1 つのノードのみに行われるため、ノード間の競合の可能性はほぼゼロに減少します。その結果、パフォーマンスへの影響が大幅に削減されます。

セカンダリアプリケーションはシャドウノードに対して実行される場合がありますが、メインアプリケーションがそのノードの使用を開始すると、これらは削減または中断されます。

将来的には、1 つのノードが他のグループのメインレプリケーターとして選択され、クラスターの成長に応じてレプリケーションの CPU オーバーヘッドを制限し、他のグループへの帯域幅を最小化します。

### サポートされている Postgres データベースサーバー

PGD は、 PostgreSQL 、 EDB Postgres Extended Server 、および EDB Postgres Advanced Server と互換性があり、BDR という名前の標準の Postgres 拡張機能として展開されます。サポートされているバージョンの組み合わせの詳細については、Compatibility を参照してください。

一部の主要な PGD 機能は、ターゲット Postgres データベースサーバーで使用可能な特定のコア機能に依存しています。したがって、PGD ユーザーは、ビジネスニーズに最適な Postgres データベースサーバーディストリビューションも採用する必要があります。たとえば、PGD 機能の Commit At Most OnceCAMO を使用することがユースケースにとってミッションクリティカルである場合、コミュニティ PostgreSQL ディストリビューションを採用しないでください。CAMO を処理するために必要なコア機能がありません。

PGD は、ネイティブに近い Postgres の互換性を提供します。ただし、一部のアクセスパターンは、単一のインスタンスで行う場合と同じようにマルチノード設定で必ずしも機能するとは限りません。マルチノード設定で安全に複製できるものには、いくつかの制限があります。Application usage は、アプリケーション開発の観点から PGD がどのように動作するかについて詳しく説明します。

### パフォーマンスに影響を与える特性

デフォルトでは、PGD はグループ内の各ノードに各テーブルのコピーを 1 つ保持し、変更はグループ内のすべてのノードに伝播します。

データのコピーはどこにでもあるため、SELECT はローカルノードにアクセスすることのみが必要です。読み取り専用クラスターでは、ノードのパフォーマンスはノード数の影響を受けず、長時間実行された SELECT クエリによって発生する他のノードでのレプリケーション競合の影響を受けません。したがって、ノードを追加すると、可能な合計 SELECT スループットが線形に増加します。

INSERT、UPDATE、および DELETEDML がローカルで実行される場合、変更はグループ内のすべてのノードに伝播します。DML apply のオーバーヘッドは、元の実行よりも少なくなります。したがって、複数のノードで純粋な書き込みワークロードを同時に実行する場合、マルチノードクラスターは、シングルノードよりも多くの TPS を処理できます。

競合処理にはコストがかかり、スループットを低下させます。スループットは、実際にアプリケーションが表示する競合の量によって異なります。競合が非常に低いアプリケーションは、単一ノードよりも優れたパフォーマンスを示します。競合の多いアプリケーションは、単一ノードよりもパフォーマンスが低下する場合があります。これらの結果は、マルチマスターテクノロジーと一貫性があり、PGD に固有ではありません。

同期レプリケーションオプションは、変更をマルチプルのノードに同時に送信できるため、レプリケーション遅延を最小限に抑えることができます。ノードを追加するということは、レプリケーションにより多くの CPU を使用することを意味するため、各ノードが追加されるとピーク TPS がわずかに減少します。

ワークロードがすべての CPU リソースを使用しようとすると、このリソースがレプリケーションを制約し、レプリケーションラグに影響を与える可能性があります。

要約すると、すべての書き込みがすべてのノードで再生されるため、ほとんどのテーブルがレプリケートされる場合、PGD グループにマスターノードを追加しても、大幅な書き込みスループットの増加はありません。PGD 書き込みは、一般に SQL を介して Postgres クライアントからの書き込みよりも効果的であるため、パフォーマンスを向上させることができます。読み取りスループットは、一般にノードの数に線形にスケールします。

### PGD compared

次の表では、EDB Postgres Distributed と他のレプリケーションソリューションを比較しています。

| | 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 |

次のページに続く

表 1 – 前のページからの続き

| | PGD Standalone | PGD Managed | PSR + EFM | pglogical 2 | PG Builtin Logical Replication |
|---|---|---|---|---|---|
| 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 |
| 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 | | | | | |

次のページに続く

表 1 – 前のページからの続き

| | PGD Standalone | PGD Managed | PSR + EFM | pglogical 2 | PG Builtin Logical Replication |
|---|---|---|---|---|---|
| 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 |
| 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 |

次のページに続く

表 1 – 前のページからの続き

| | PGD Standalone | PGD Managed | PSR + EFM | pglogical 2 | PG Builtin Logical Replication |
|---|---|---|---|---|---|
| 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_con | Yes - rewrite requires permit_unsafe_con | 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 |
| Licensing | | | | | |
| Source available | No | No | No | Yes | Yes |
| Open source | No | No | No | Yes | Yes |

### 6.3.2 DDL replication

DDL はデータ定義言語の略で、データベースオブジェクトを作成、変更、および削除する SQL 言語のサブセットです。

PGD は自動 DDL レプリケーションを提供し、特定の DDL 変更を簡単にします。自動レプリケーションを使用すると、DDL 変更をすべてのノードに手動で配布し、一貫性を確認する必要はありません。

このセクションでは、PGD で DDL レプリケーションがどのように処理されるかについて説明します。

- *DDL overview* は、PGD の DDL レプリケーションでできることの一般的な概要を提供します。
- *DDL locking details* は、DDL レプリケーションがロックを使用して DDL を安全にレプリケートする方法を調べます。
- *Managing DDL with PGD replication* は、クラスターのスムーズな実行に過度に影響を与えないように、DDL 変更のインパクトを制限する理由と方法に関するベストプラクティスアドバイスを提供します。
- *DDL and role manipulation statements* は、クラスター内の複数のデータベースにわたるロールの操作に関する問題をノートします。
- *Workarounds for DDL restrictions* は、DDL レプリケーションが列、制約、タイプの変更などの制限を提示する可能性がある状況を処理するためのさまざまなオプションを提供します。
- *PGD functions that behave like DDL* は、DDL と同様に動作するため、同様の方法で動作し、同様の制限の対象となる PGD ファンクションについて詳しく説明します。

### DDL overview

DDL はデータ定義言語の略で、データベースオブジェクトを作成、変更、および削除する SQL 言語のサブセットです。

### レプリケートされた DDL

操作の利便性と正確性を維持するために、PGD はほとんどの DDL アクションをレプリケートしますが、次の例外を除きます。

- 一時的な関係
- 特定の DDL ステートメントほとんどの場合、長時間実行
- ロックコマンド `LOCK`
- テーブルメンテナンスコマンド `VACUUM` 、 `ANALYZE` 、 `CLUSTER` 、 `REINDEX`
- オートバキュームのアクション

・操作コマンド `CHECKPOINT` 、 `ALTER SYSTEM`

- データベースまたはテーブルスペースに関連するアクション

自動 DDL レプリケーションにより、DDL 変更をすべてのノードに手動で配布し、一貫性を確認する必要がなく、特定の DDL 変更が簡単になります。

デフォルトのレプリケーションセットでは、DDL はデフォルトですべてのノードにレプリケートされます。

### PostgreSQL との違い

DDL レプリケーションに関して、PGD はスタンドアロンの PostgreSQL とは大きく異なります。それを同じに扱うことは、PGD で最も一般的な問題です。

テーブルレプリケーションとの主な違いは、DDL レプリケーションが DDL の結果をレプリケートしないことです。代わりに、ステートメントをレプリケートします。これは、ほとんどの場合で非常にうまく機能しますが、DDL をすべてのノードで同様に実行する必要があるという要件が導入されます。より微妙な点は、DDL は、拡張機能ビルトインではなくによって導入されたデータ型を含む、すべてのデータ型固有のパラメーター設定に関して不変である必要があることです。たとえば、DDL ステートメントは、各ノードで使用されるデフォルトのエンコーディングで正しく実行される必要があります。

### PGD システムでの DDL の実行

PGD グループは、スタンドアロンの PostgreSQL サーバーと同じではありません。これは、集中ロックやトランザクションコーディネーターを使用しない非同期マルチマスターレプリケーションに基づいています。これは、DDL の実行時に重要な意味を持ちます。

パラレルに実行される DDL は、PGD で引き続き実行されます。DDL の実行は、実行時に各ノードの並列オペレーションに影響を与えるパラメーターを尊重するため、ノード間の設定の違いに気づく場合があります。

競合する DDL の実行を防止します。そうしないと、DDL レプリケーションでエラーが発生し、レプリケーションが停止します。

PGD は、これらの問題に対する 3 つのレベルの保護を提供します。

`ddl_locking = 'all'` は最も厳密なオプションであり、DDL が任意のノードから同時に実行される可能性があり、正確性を保証したい場合に最適です。これがデフォルトです。

`ddl_locking = 'dml'` は、任意の時点で 1 つのノードから DDL を実行する場合にのみ安全なオプションです。DDL を実行する場所を完全に制御できる場合にのみこの設定を使用します。単一のノードから DDL を実行すると、ノード間の競合が保証されません。ノード内の競合は PostgreSQL によって既に処理されています。

`ddl_locking = 'off'` は最も厳密でないオプションであり、一般的な使用は危険です。このオプションは、ロックを完全にスキップし、パフォーマンスのオーバーヘッドを回避するため、新しい空のデータベーススキーマを作成するときに役立つオプションです。

これらのオプションは、bdr_superuser、スーパーユーザー、または `postgres.conf` 構成ファイルでのみ設定できます。

bdr.replicate_ddl_command を使用する場合、そのファンクションに渡される DDL コマンドにのみ指定された bdr.ddl_locking 設定を使用して、3 番目の引数でこのパラメーターを直接設定できます。

### DDL および混合 PostgreSQL バージョン

PGD は、クラスター内の異なるメジャー Postgres バージョン間の DDL レプリケーションをサポートしていません。クラスターは Postgres の同じメジャーバージョンを実行するため、ほとんどの場合、これは問題ではありません。ただし、これは、メジャーバージョンから別のメジャーバージョンへのクラスターのローリングアップグレードを実行する場合には当てはまりません。この場合、DDL レプリケーションは、すべてのノードが同じメジャーバージョンにアップグレードされるまでサポートされないため、使用しないでください。

スクリプトが DDL レプリケーションをトリガーする場合があるため、拡張機能を更新するときはアップグレードプロセスで特別な注意を払う必要があります。この場合、アップグレードが完了する前にスクリプ

トを実行する必要がある場合は、スクリプトの実行中に bdr.ddl_replication 設定を off に設定する必要があります。

### DDL replication options

bdr.ddl_replication パラメーターは、レプリケーション動作を指定します。

bdr.ddl_replication = on がデフォルトです。この設定は、DDL をデフォルトのレプリケーションセット、デフォルトではすべてのノードを意味します。デフォルト以外のレプリケーションセットは、*Replication sets* がない限り DDL をレプリケートしません

彼らのために定義されました。

ファンクション *bdr.replicate_ddl_command()* を使用して、DDL を特定のレプリケーションセットにレプリケートすることもできます。この機能は、ノードがダウンしているときに DDL コマンドを実行する場合に役立ちます。ノードまたはレップセットのサブセット、たとえば、site1 のすべてのノードに存在するインデックスまたはパーティションが必要な場合にも役立ちます。

```
SELECT bdr.replicate_ddl_command(
                CREATE INDEX CONCURRENTLY ON foo (col7);,
                ARRAY[site1],     -- the replication sets
                all);              -- ddl_locking to apply
```

お勧めしませんが、bdr.ddl_replication 構成パラメーターを使用して、自動 DDL レプリケーションをスキップし、各ノードで手動で実行できます。

```
SET bdr.ddl_replication = off;
```

設定すると、PGD は実行された DDL コマンドのグローバルロックとレプリケーションの両方をスキップします。次に、すべてのノードで DDL を手動で実行する必要があります。

> **警告**
>
> グローバルロックを使用せずに各ノードで DDL を手動で実行すると、競合する DDL または DML が同時に実行される場合、PGD グループ全体のレプリケートが停止される場合があります。

bdr_superuser またはスーパーユーザーのみが bdr.ddl_replication パラメーターを設定できます。postgres.conf 構成ファイルで設定することもできます。

### DDL locking details

グローバル DDL ロックとグローバル DML ロックの 2 種類のロックは、PGD でレプリケートされた DDL の正しさを強制します。

### グローバル DDL ロック

グローバル DDL ロックは、`ddl_locking = 'all'` の場合にのみ使用されます。この種類のロックは、各 DDL ステートメントの実行中にクラスターで他の DDL が実行されないようにします。この動作は、一般的な場合に完全な正しさを保証しますが、多くの単純な場合には厳密すぎます。PGD は、スキーマ変更が行われるトランザクションで初めて DDL 操作のグローバルロックを取得します。これにより、クラスター内の DDL 実行トランザクションが効果的にシリアル化されます。つまり、DDL の実行中は、ノード上の他の接続は、別のテーブルに影響を与える場合でも、別の DDL コマンドを実行できません。

DDL 操作のロックを取得するために、DDL を実行する PGD ノードは PGD グループ内の他のノードに接続し、DDL を実行する排他的権限を付与するように求めます。

ロック要求は通常のレプリケーションストリームによって送信され、ノードは同様にレプリケーションストリームによって応答します。したがって、ノードまたは少なくともノードの大部分が、レプリケーション遅延をほとんど発生せずに実行されることが重要です。そうしないと、ノードが DDL ロックを取得するまでに時間がかかる場合があります。ノードの大部分が同意すると、DDL 実行が実行されます。

DDL ロックの順序は、Raft プロトコルを使用して決定されます。1 つのノードで実行された DDL ステートメントは、他のすべてのノードで同じシーケンスで実行されます。

DDL を実行しているノードがクラスター内で実行された以前のすべての DDL の効果を確認したことを確認するために、以前の DDL を実行したノードに追いつくまで待機します。現在の DDL を実行しているノードが、以前の DDL を実行したノードと比較してレプリケーションが遅れている場合、ロックの取得に時間がかかる場合があります。したがって、単一のノード、または他のノードで発生したレプリケーションの変更にほぼ追いついたノードから DDL を実行することをお勧めします。

グローバル DDL ロックは、データおよび監視ノードの大部分によって付与される必要があります。ここで、マジョリティは適格ノードの N/2+1 です。サブスクライバー専用ノードは参加する資格がありません。

### グローバル DML ロック

グローバル DML ロックまたはリレーション DML ロックとして知られるこの種類のロックは、`ddl_locking = all` または `ddl_locking = dml` のいずれか、および DDL ステートメントがフライト中の DML ステートメントが失敗する可能性がある場合に使用されます。これらの障害は、一意制約、チェック制約、NOT NULL 制約などの制約を追加または変更するときに発生する可能性があります。リレーション DML ロックは、一度に 1 つのリレーションにのみ影響します。これらのロックにより、変更がキュー内にあるときに DDL が実行されず、エラーでレプリケーションが停止する可能性があります。

テーブルのグローバル DML ロックを取得するために、DDL を実行する PGD ノードは PGD グループ内の他のすべてのノードに接続し、書き込みに対してテーブルをロックするように依頼し、そのテーブルの保留中の変更がすべてドレインされるまで待機します。すべてのノードが完全に追いつくと、DML ロックの発信者は自由にテーブルのスキーマ変更を実行し、他のノードにレプリケートできます。

グローバル DML ロックは、各ノードのテーブルの EXCLUSIVE LOCK を保持するため、実行中にそのテーブルに対する DML、他の DDL、VACUUM、および index コマンドをブロックします。これは、通常 EXCLUSIVE LOCK 以上を使用しないコマンドにグローバル DML ロックが保持されている場合にも当てはまります。

保留中の DML 操作がドレインされるまでに長い時間がかかる場合があり、レプリケーションが現在遅延している場合はさらに長くなります。これは、データの変更とは異なり、行表現と制約に影響を与えるスキーマの変更は、構成されているすべてのノードに到達でき、現在の書き込みレートに適度に追随してい

るときにのみ実行できることを意味します。ノードがダウンしているときにこのような DDL コマンドを実行する必要がある場合は、最初にダウンしたノードを構成から削除します。

ロックが付与される前に、資格のあるすべてのデータノードがグローバル DML ロックを付与することに同意する必要があります。Witness およびサブスクライバ専用ノードは参加する資格がありません。

DDL ステートメントがレプリケートされない場合、グローバルロックは取得されません。

*PGD システムでの DDL の実行* で説明しているように、bdr.ddl_locking パラメーターでロック動作を指定します。

- `ddl_locking = all` は、グローバル DDL ロックを取得し、必要に応じてリレーション DML ロックを取得します。
- `ddl_locking = dml` は、グローバル DDL ロックをスキップし、必要に応じてリレーション DML ロックを取得します。
- `ddl_locking = leader` は、リーダーベースのグローバル DML ロックを有効にします。
- `ddl_locking = auto` 現在の `ddl_locking = leader` のように動作します。
- `ddl_locking = off` は、グローバル DDL ロックとリレーション DML ロックの両方をスキップします。

一部の PGD ファンクションは DDL 変更を行います。これらのファンクションには、DDL ロック動作が適用されます。これについては、各ファンクションのドキュメントに記載されています。

したがって、`ddl_locking = dml` は、競合する DDL が他のノードから実行されないことを保証できる場合にのみ安全です。この設定では、グローバル DDL ロックのみを必要とするステートメントは、グローバルロックをまったく使用しません。

`ddl_locking = off` は、DDL が実行されるデータベースオブジェクトで競合する DDL と競合する DML 操作がないことを保証できる場合にのみ安全です。ロックをオフにした後に問題が発生すると、データの進行中の変更が失われる可能性があります。ユーザーアプリケーションチームは、発生した問題を解決する必要があります。

場合によっては、同時に実行される DDL を適切にシリアル化できる場合があります。これらのシリアル化障害が発生した場合、DDL が再実行される場合があります。

DDL レプリケーションは、昇格するまでロジカルスタンバイノードでアクティブになりません。

一部の PGD 管理機能は DDL のように動作します。つまり、グローバルロックを取得しようとし、DDL レプリケーションがアクティブな場合、アクションがレプリケートされます。レプリケートファンクションの完全なリストは、*PGD functions that behave like DDL* にリストされています。

一時テーブルで実行される DDL は、グローバルロックを必要としません。

現在のトランザクションで作成されたオブジェクトの ALTER または DROP には、グローバル DML ロックは必要ありません。

グローバル DDL ロックとグローバル DML ロックのモニタリングは *Monitoring the Connection Manager* に表示されます。

## Managing DDL with PGD replication

### DDL のインパクトの最小化

DDL の影響を最小限に抑えることは、データベースにとって良い運用上のアドバイスです。PGD では、これらの点がさらに重要になります。

- DDL の影響を最小限に抑えるには、DDL を実行するトランザクションを短くします。大量の行変更と組み合わせず、長時間実行される外部キーまたはその他の制約の再チェックを避けてください。
- `ALTER TABLE` の場合、`ADD CONSTRAINT` を単独で使用するのではなく、`ADD CONSTRAINT NOT VALID` を使用し、その後に `VALIDATE CONSTRAINT` との別のトランザクションを使用します。 `VALIDATE CONSTRAINT` は、すべてのノードで再生されるまで待機するため、確認の受信に顕著な遅延が発生します。
- インデックスを作成するときは、可能な限り `CONCURRENTLY` オプションを使用します。

長時間実行 DDL を実行する別の方法は、DDL レプリケーションを無効にして、各ノードで個別に DDL ステートメントを実行することです。次の例に示すように、単一の SQL ステートメントを使用してこれを行うことができます。グローバルロックルールは引き続き適用されるため、このタイプの使用法で自分自身をロックアウトしないように注意してください。これは回避策です。

```
SELECT bdr.run_on_all_nodes($ddl$
        CREATE INDEX CONCURRENTLY index_a ON table_a(i);
$ddl$);
```

`CREATE INDEX CONCURRENTLY` はマルチトランザクションコマンドであるため、セッション全体で DDL レプリケーションを無効にする必要があることに注意して、`CREATE INDEX CONCURRENTLY` で *bdr.run_on_all_nodes()* テクニックを使用することをお勧めします。実行中の書き込みを防止するため、実稼働システムでは `CREATE INDEX` を避けてください。 `REINDEX` は、AccessExclusiveLocks を保持するため、使用を避けてください。

代わりに、`REINDEX CONCURRENTLY` または `reindexdb --concurrently` を使用します。

次のようなコマンドラインユーティリティを使用する場合、DDL レプリケーションを無効にできます。

```
$ export PGOPTIONS="-c bdr.ddl_replication=off"
$ pg_restore --section=post-data
```

複数の DDL ステートメントは、個々のステートメントとして起動されるよりも単一のトランザクションに束ねたほうがメリットがある場合があるため、DDL ロックは 1 回のみを取ります。テーブルレベルのロックが通常の操作を妨げる場合、これは望ましくない場合があります。

DDL がシステムを長時間保持している場合は、 psql の **Control-C** または `pg_cancel_backend()` を使用して、元のノードで DDL を安全にキャンセルできます。他のノードから DDL ロックをキャンセルすることはできません。

オプションのグローバルロックタイムアウト設定を使用して、グローバルロックにかかる時間を制御できます。bdr.global_lock_timeout は、グローバルロックの取得がキャンセルされるまでの待機時間を制限します。bdr.global_lock_statement_timeout は、グローバルロックを保持するトランザクション内のステートメントの実行タイム長を制限し、bdr.global_lock_idle_timeout は、グローバルロックを保持するトランザク

ションの最大許可アイドル時間ステートメント間の時間を設定します。値をゼロに設定することにより、これらのタイムアウトをすべて無効にできます。

元のノードで DDL 操作がコミットされると、キャンセルまたは中止することはできません。PGD グループは、グローバルロックを確認した他のノードに正常に適用され、それらが再生を確認するまで待機する必要があります。このため、DDL トランザクションは短くて速いままにしてください。

### ダウンしたノードでの DDL の処理

グローバル DDL ロックを開始するノードがグローバルロック DDL または DML を取得した後にダウンした場合、ロックはアクティブなままです。タイムアウトが設定されていても、グローバルロックはタイムアウトしません。ノードが復旧した場合、ノードは保持しているすべてのグローバルロックを解放します。

長時間または無期限にダウンしたままになる場合は、PGD グループからノードを削除してグローバルロックを解放します。これが、bdr_superuser として SET コマンドを使用して緊急 DDL を実行し、bdr.ddl_locking 値を更新する理由の 1 つです。

グローバルロックを確認した後、それを取得するコマンドが実行される前に、他のノードのいずれかがダウンした場合、ノードがアップしているかのように、ロックを要求するそのコマンドの実行は続行されます。

前述のように、グローバル DDL ロックはノードの大部分のみが応答する必要があるため、大部分が実行され到達可能である限り、クラスターの一部がダウンしている場合に機能します。ただし、クラスター全体が利用できない限り DML ロックは取得できません。

グローバル DDL またはグローバル DML ロックでは、別のノードがダウンすると、コマンドは通常に続行され、ロックが解放されます。

### ステートメント固有の DDL レプリケーションの問題

すべてのコマンドを自動的に複製できるわけではありません。bdr.ddl_replication をオフにして DDL レプリケーションをオフにしない限り、このようなコマンドは許可されません。

PGD は、データベースでアクティブなときに一部の DDL ステートメントの実行を防止します。これにより、正しくレプリケートできないステートメント、またはレプリケーションがまだサポートされていないステートメントを禁止することにより、システムの一貫性が保護されます。

ステートメントが PGD で許可されない場合、多くの場合、同じことを行う別の方法を見つけることができます。たとえば、揮発性のデフォルト値を含む列を追加する `ALTER TABLE` は実行できません。しかし、一般的には、機能する一連の独立した `ALTER TABLE` および `UPDATE` ステートメントとして言い換えることができます。

通常、サポートされていないステートメントは実行されず、`feature_not_supported` SQLSTATE `0A000` エラーが発生します。

一時オブジェクトを参照または依存する DDL は、PGD でレプリケートできず、DDL レプリケーションを有効にして実行するとエラーがスローされます。

### DDL command handling matrix

次の表では、許可されるユーティリティまたは DDL コマンド、レプリケートされるもの、およびレプリケート時に使用されるグローバルロックの種類について説明します。

`ALTER TABLE` のようなより複雑なステートメントの場合、これらは実行されるサブコマンドによって異なる場合があります。このようなすべてのコマンドには、次の表の下に詳細な説明があります。

### コマンドマトリックス

| 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 | アーキテクチャの詳細 |
| ALTER PUBLICATION | Y | Y | DDL |

次のページに続く

表 2 – 前のページからの続き

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| ALTER QUEUE | Y | Y | DDL |
| ALTER QUEUE TABLE | Y | Y | DDL |
| ALTER REDACTION POLICY | Y | Y | DDL |
| ALTER RESOURCE GROUP | Y | N | N |
| ALTER ROLE | Y | Y | DDL |
| ALTER ROUTINE | Y | Y | DDL |
| ALTER RULE | Y | Y | DDL |
| ALTER SCHEMA | Y | Y | DDL |
| ALTER SEQUENCE | アーキテクチャの詳細 | 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 | アーキテクチャの詳細 | Y | アーキテクチャの詳細 |
| 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 | N |

次のページに続く

表 2 – 前のページからの続き

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| COMMENT | Y | アーキテクチャの詳細 | DDL |
| COMMIT | Y | N | N |
| COMMIT PREPARED | Y | N | N |
| COPY | Y | N | N |
| COPY FROM | Y | N | N |
| CREATE ACCESS METHOD | Y | Y | DDL |
| CREATE AGGREGATE | Y | Y | DDL |
| CREATE CAST | Y | Y | DDL |
| CREATE COLLATION | Y | Y | DDL |
| CREATE CONSTRAINT | Y | Y | DDL |
| CREATE CONVERSION | Y | Y | DDL |
| CREATE DATABASE | Y | N | N |
| CREATE DATABASE LINK | Y | Y | DDL |
| CREATE DIRECTORY | Y | Y | DDL |
| CREATE DOMAIN | Y | Y | DDL |
| CREATE EVENT TRIGGER | Y | Y | DDL |
| CREATE EXTENSION | Y | Y | DDL |
| 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 | アーキテクチャの詳細 |

次のページに続く

表 2 – 前のページからの続き

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| 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 | アーキテクチャの詳細 | 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 | アーキテクチャの詳細 | Y | DDL |
| CREATE TABLESPACE | Y | Y | DDL |
| CREATE TEXT SEARCH CONFIGURATION | Y | Y | DDL |
| CREATE TEXT SEARCH DICTIONARY | Y | Y | DDL |
| CREATE TEXT SEARCH PARSER | Y | Y | DDL |
| CREATE TEXT SEARCH TEMPLATE | Y | Y | DDL |
| CREATE TRANSFORM | Y | Y | DDL |
| CREATE TRIGGER | Y | Y | DDL |
| 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 |

次のページに続く

表 2 – 前のページからの続き

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| 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 |
| DROP PUBLICATION | Y | Y | DDL |
| DROP QUEUE | Y | Y | DDL |

次のページに続く

表 2 – 前のページからの続き

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| 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 | アーキテクチャの詳細 | アーキテクチャの詳細 |
| FETCH | Y | N | N |
| GRANT | Y | アーキテクチャの詳細 | 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 | アーキテクチャの詳細 |

次のページに続く

表 2 – 前のページからの続き

| Command | Allowed | Replicated | Lock |
|---|---|---|---|
| MOVE | Y | N | N |
| NOTIFY | Y | N | N |
| PREPARE | Y | N | N |
| PREPARE TRANSACTION | Y | N | N |
| REASSIGN OWNED | Y | Y | DDL |
| REFRESH MATERIALIZED VIEW | Y | Y | DML |
| REINDEX | Y | N | N |
| RELEASE | Y | N | N |
| RESET | Y | N | N |
| REVOKE | Y | アーキテクチャの詳細 | DDL |
| REVOKE ROLE | Y | Y | DDL |
| ROLLBACK | Y | N | N |
| ROLLBACK PREPARED | Y | N | N |
| SAVEPOINT | Y | N | N |
| SECURITY LABEL | Y | アーキテクチャの詳細 | DDL |
| SELECT INTO | アーキテクチャの詳細 | Y | DDL |
| SET | Y | N | N |
| SET CONSTRAINTS | Y | N | N |
| SHOW | Y | N | N |
| START TRANSACTION | Y | N | N |
| TRUNCATE TABLE | Y | アーキテクチャの詳細 | アーキテクチャの詳細 |
| UNLISTEN | Y | N | N |
| VACUUM | Y | N | N |

### コマンドに関する注意事項

#### ALTER SEQUENCE

一般に `ALTER SEQUENCE` がサポートされていますが、グローバルシーケンスを使用する場合、一部のオプションは効果がありません。

`ALTER SEQUENCE ... RENAME` は、galloc シーケンスのみでサポートされていません。`ALTER SEQUENCE ... SET SCHEMA` は、galloc シーケンスのみでサポートされていません。

### ALTER TABLE

通常、 ALTER TABLE コマンドは許可されます。ただし、いくつかのサブコマンドはサポートされていません。

### ALTER TABLE は許可されないコマンド

ALTER TABLE の一部のバリアントは、現在 PGD ノードで使用できません。

- ALTER COLUMN ... SET STORAGE external —列がテーブルのレプリカ ID の列の 1 つである場合、拒否されます。コマンドが安全であることが確認できる場合は、bdr.permit_unsafe_commands を使用してこの動作をオーバーライドできます。
- RENAME —自動パーティション化されたテーブルの名前は変更できません。
- SET SCHEMA —自動パーティション化されたテーブルのスキーマは設定できません。
- ALTER TABLE ... ADD FOREIGN KEY —現在のユーザーが参照されるテーブルを読み取る権限がない場合、または参照されるテーブルで現在のユーザーがバイパスできない RLS 制限が有効になっている場合、サポートされません。

次の例は、timestamp 型の定数値を timestamptz 型の列に追加しようとするため、失敗します。 timestamp と timestamptz 間のキャストは、セッションのタイムゾーンに依存するため、不変ではありません。

```
ALTER TABLE foo
  ADD expiry_date timestamptz DEFAULT timestamp 2100-01-01 00:00:00 NOT NULL;
```

DML ロックを取得せずに、 CHECK や FOREIGN KEY 制約などの特定のタイプの制約を追加できます。ただし、これには、最初に NOT VALID 制約を作成し、次に ALTER TABLE ... VALIDATE CONSTRAINT コマンドを使用して別のトランザクションで制約を検証する 2 段階のプロセスが必要です。コンストレインの追加 を参照

詳細については、

### ALTER TABLE ロック

ALTER TABLE の次のバリアントは、DDL ロックのみを取得し、DML ロックは取得しません。

- ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT
- ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression
- ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT
- ALTER TABLE ... ALTER COLUMN ... TYPE リライトが必要ない場合
- 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 は引き続き DML ロックを取得します
- ALTER TABLE ... CLUSTER ON
- ALTER TABLE ... SET WITHOUT CLUSTER
- ALTER TABLE ... SET ( storage_parameter = value [, ... ] )
- ALTER TABLE ... RESET ( storage_parameter = [, ... ] )
- ALTER TABLE ... OWNER TO

ALTER TABLE の他のすべてのバリアントは、変更されるテーブルで DML ロックを取得します。ALTER TABLE の一部のバリアントには、以下に説明する制限があります。

### ALTER TABLE の例

さまざまなタイプの ALTER TABLE ... ALTER COLUMN TYPE ATCT 操作が可能です。一部の ATCT 操作は、基になる列タイプのメタデータのみを更新し、基になるテーブルデータの書き換えは必要ありません。これは通常、既存の列タイプとターゲットタイプがバイナリ強制可能な場合に当てはまります。

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(20));
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(128);
```

ただし、VARCHAR(128) から VARCHAR(20) に変更されたコマンドを元に戻すためにこの変更を行うことは、バイナリ強制不可であり、テーブルの書き換えが発生します。

```
CREATE TABLE sample (col1 BIGINT PRIMARY KEY, col2 VARCHAR(128), col3 INT);
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(256);
```

バイナリの強制性のため、列タイプを VARCHAR または TEXT データ型に変更することもできます。繰り返しますが、これは基になる列タイプの単なるメタデータの更新です。

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR;
ALTER TABLE sample ALTER COLUMN col2 TYPE TEXT;
```

### TABLE REWRITE の例

ユーザーは列のデータ型を変更したり、ヌル以外のデフォルトで列を追加したり、メンテナンスジョブを実行したりする場合があります。これらの操作は、テーブルの書き換えをトリガーする可能性がある PostgreSQL の DDL コマンドでサポートされています。テーブル書き換え操作には、ディスク上にテーブルの新しいバージョンの作成が含まれます。CLUSTER や VACUUM などの DDL 操作は、テーブル定義を変更せずにテーブルの書き換えを直接発生させます。ALTER TABLE…ALTER COLUMN TYPE のような DDL 操作は、列を変更し、テーブルの書き換えをトリガーする場合があります。既存のすべてのデータを古いバージョンから新しいものにコピーし、新しいバージョンの準備ができて一貫性があると、古いテーブルを新しいものに置き換えます。

典型的なテーブル書き換え ATCT 操作の例

次のテーブルを検討します。

```
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
);
```

そのテーブルを指定すると、次の ALTER TABLE コマンドはテーブルの書き換えをトリガーします。

列のデータ型を変更します。

```
ALTER TABLE foo ALTER c1 TYPE bigint;
```

USING 句を使用して列タイプを互換性のないタイプに変更します。

```
ALTER TABLE foo ALTER COLUMN name TYPE int USING name::integer;
```

揮発性列タイプ

```
ALTER TABLE foo ALTER COLUMN time_now TYPE timestamp;
```

変更可能なデフォルトを使用して列を追加します。

```
ALTER TABLE foo ADD COLUMN time_clock timestamptz NOT NULL DEFAULT clock_timestamp();
```

生成された列

```
ALTER TABLE foo ALTER COLUMN indian_currency TYPE numeric;
```

単一のトランザクションブロックで複数の ALTER TABLE DDL 操作を実行します。

```
BEGIN;
ALTER TABLE foo ALTER c2 TYPE bigint;
ALTER TABLE foo ALTER c3 TYPE bigint;
COMMIT;
```

PGD クラスターでは、これには書き換えが含まれるため、書き換えアクティビティは DML タイプのグローバルリーダーロックを取得し、したがって、クラスター内のすべてのグループリーダーが利用できる必要があります。

**注釈**

提供されている ALTER TABLE 例は、使用可能な ATCT 操作の完全なリストではありません。

ALTER TYPE はレプリケートされますが、PostgreSQL はこれらの依存関係を記録しないため、グローバル DML ロックはそのデータ型を使用するすべてのテーブルに適用されません。*Workarounds for DDL restrictions* を参照してください。

### コメントオン

COMMENT ON のすべてのバリアントが許可されますが、COMMENT ON TABLESPACE/DATABASE/LARGE OBJECT はレプリケートされません。

### CREATE PROFILE または ALTER PROFILE

ファンクションが SECURITY DEFINER の場合、プロファイルに関連付けられている PASSWORD_VERIFY_FUNCTION は IMMUTABLE である必要があります。このような CREATE PROFILE または ALTER PROFILE コマンドはレプリケートされますが、このプロファイルを使用する後続の CREATE USER または ALTER USER コマンドは、 writer ワーカーが cannot change current role within security-restricted operation エラーをスローするため、レプリケーションを中断します。

### CREATE SEQUENCE

一般に CREATE SEQUENCE がサポートされていますが、グローバルシーケンスを使用する場合、一部のオプションは効果がありません。

### CREATE TABLE AS と SELECT INTO

CREATE TABLE AS および SELECT INTO は、すべてのサブコマンドも許可される場合にのみ許可されます。

### EXPLAIN

一般に EXPLAIN は許可されますが、 EXPLAIN ANALYZE はデータベースに副作用を与える可能性があるため、いくつかの制限があります。

### EXPLAIN ANALYZE レプリケーション

EXPLAIN ANALYZE は、分析されたステートメントのレプリケーションルールに従います。

### EXPLAIN ANALYZE ロック

EXPLAIN ANALYZE は、分析されたステートメントのロックルールに従います。

### GRANT と REVOKE

通常、GRANT および REVOKE ステートメントはサポートされていますが、GRANT/REVOKE ON TABLESPACE/LARGE OBJECT はレプリケートされません。

### ロックテーブル

LOCK TABLE はレプリケートされませんが、bdr.lock_table_locking が on に設定されている場合、グローバル DML ロックを取得する場合があります。

bdr.global_lock_table() ファンクションを使用して、グローバル DML ロックを明示的に要求することもできます。

### セキュリティラベル

SECURITY LABEL のすべてのバリアントが許可されますが、SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT はレプリケートされません。

### TRUNCATE レプリケーション

TRUNCATE コマンドは、DDL ステートメントではなく DML としてレプリケートされます。テーブルの TRUNCATE がレプリケートされるかどうかは、影響を受ける各テーブルのレプリケーション設定によって異なります。

### TRUNCATE ロック

TRUNCATE は他の DDL と同じ方法でレプリケートされませんが、bdr.truncate_locking が on に設定されている場合、グローバル DML ロックを取得できます。

### DDL and role manipulation statements

ユーザーは PostgreSQL インスタンスのグローバルオブジェクトです。つまり、ユーザーは複数のデータベースにまたがりますが、PGD は個々のデータベースレベルで動作します。この動作のため、ロール操作ステートメントの処理には追加の検討が必要です。

PGD では、レプリケートされた DDL によって参照されるロールがすべてのノードに存在する必要があります。ロールの付与、パスワードなどは同じである必要はありませんが、存在する必要があります。

bdr.role_replication が有効デフォルトであり、ロール操作ステートメントが PGD 対応データベースで実行されている場合、PGD はロール操作ステートメントをレプリケートします。

ロール操作ステートメントには次のものが含まれます。

- CREATE ROLE
- ALTER ROLE
- DROP ROLE
- GRANT ROLE
- CREATE USER
- ALTER USER
- DROP USER
- CREATE GROUP
- ALTER GROUP
- DROP GROUP

一般に、次のいずれかです。

- bdr.role_replication = off でシステムを構成し、Ansible、Puppet、Chef などの外部オーケストレーションツールまたは bdr.replicate_ddl_command() によって明示的にレプリケートされて、すべてのロール変更ユーザーとグループを展開します。

- PostgreSQL インスタンス上の 1 つの PGD 対応データベースに bdr.role_replication = on があるようにシステムを構成し、そのデータベースですべてのロール管理 DDL を実行します。

すべてのロール管理コマンドを 1 つのデータベースで実行することをお勧めします。

ロールレプリケーションがオフになっている場合、管理者は、あるノードで DDL が使用するロールが他のノードにも存在することを確認する必要があります。それ以外の場合、PGD apply は、他のノードでロールが作成されるまで、エラーでストールします。

### Workarounds for DDL restrictions

PGD DDL オペレーション処理の制限の一部をワークアラウンドできます。多くの場合、操作をより小さな変更に分割すると、単一のステートメントとして許可されないか、過剰なロックが必要な目的の結果が生成される場合があります。

#### コンストレインの追加

DML ロックを必要とせずに CHECK および FOREIGN KEY 制約を追加できます。これには、2 段階のプロセスが含まれます。

- ALTER TABLE ... ADD CONSTRAINT ... NOT VALID
- ALTER TABLE ... VALIDATE CONSTRAINT

2 つの異なるトランザクションでこれらの手順を実行します。これらの手順は両方とも、テーブルでのみ DDL ロックを取得するため、1 つ以上のノードがダウンしている場合でも実行できます。ただし、コンストレインを検証するには、PGD は次のことを確認する必要があります。

- クラスター内のすべてのノードは ADD CONSTRAINT コマンドを参照します。
- コンストレインを検証するノードは、これらのノードに NOT VALID コンストレインを作成する前に、他のすべてのノードからレプリケーション変更を適用します。

したがって、新しいメカニズムでは、制約の検証中にすべてのノードがアップしている必要はありませんが、すべてのノードが ALTER TABLE .. ADD CONSTRAINT ... NOT VALID コマンドを適用し、十分に進行していることは必要です。PGD は、コンストレインを検証する前に、一貫した状態に到達するのを待機します。

新しい機能では、クラスターが Raft プロトコルバージョン 24 以降で実行されている必要があります。Raft プロトコルがまだアップグレードされていない場合、古いメカニズムが使用され、DML ロック要求が発生します。

### PGD functions that behave like DDL

次の PGD 管理機能は DDL のように動作します。これは、DDL レプリケーションがアクティブで、DDL フィルター設定で許可されている場合、グローバルロックを取得しようとし、アクションがレプリケートされることを意味します。詳細については、各機能のドキュメントを参照してください。

レプリケーションセットの管理

- 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

競合管理

- bdr.alter_table_conflict_detection
- `bdr.column_timestamps_enable` 非推奨。`bdr.alter_table_conflict_detection()` を使用します
- `bdr.column_timestamps_disable` 非推奨。`bdr.alter_table_conflict_detection()` を使用します

シーケンス管理

- bdr.alter_sequence_set_kind

ストリームトリガー

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

### 6.3.3 Decoding worker

PGD は、データが送信されるノードの数に関係なく、デコードを 1 回実行するデコードワーカープロセスを有効にするオプションを提供します。このオプションは、各 PGD ノードに新しいプロセス WAL デコーダーを導入します。接続ごとに 1 つの WAL 送信者プロセスがまだ存在しますが、これらのプロセスはデータの送受信のタスクを実行するだけです。まとめると、これらの変更は、大規模な PGD グループの CPU オーバーヘッドを削減し、WAL 送信プロセスが通信により多くの時間を費やすようになったため、より高いレプリケーションスループットを可能にします。

#### 有効にする

`enable_wal_decoder` は、各 PGD グループのオプションであり、現在デフォルトで無効になっています。bdr.alter_node_group_option() を使用して、PGD グループのデコードワーカーを有効または無効にできます。

デコードワーカーが有効になっている場合、PGD は論理変更レコード LCR ファイルを保存して、デコードの間、およびすべてのサブスクライブノードがデータを受信したときに変更のバッファリングを許可します。LCR ファイルは、各ローカルノードのデータディレクトリの `pg_logical` ディレクトリの下に保存されます。レプリケーション遅延が増加すると、LCR ファイルの数とサイズが変化するため、このプロセスも監視が必要です。どの PGD ノードでも必要とされない LCR は、定期的にクリーニングされます。2 つの連続するクリーンアップ間の間隔は、bdr.lcr_cleanup_interval によって制御されます。デフォルトは 3 分です。bdr.lcr_cleanup_interval が 0 の場合、クリーンアップは無効になります。

### 無効化

無効にすると、論理デコードは、各ノードをサブスクライブする各ノードの WAL 送信プロセスによって実行されます。この場合、LCR ファイルは書き込まれません。

PGD グループでデコードワーカーが有効になっている場合でも、次の GUC はノードごとに LCR の生成と使用を制御します。デフォルトでは、これらは `false` です。 LCR を生成および使用するには、PGD グループのデコードワーカーを有効にし、PGD グループの各ノードでこれらの GUC を `true` に設定します。

- bdr.enable_wal_decoder — `false` の場合、LCR を使用するすべての WAL 送信者は WAL を直接使用するように再起動します。`true` と PGD グループ構成を使用すると、デコードワーカープロセスが開始され、LCR および LCR を使用する WAL センダーが生成されます。
- bdr.receive_lcr —サブスクライブノードで `true` を実行すると、パブリッシャーノードの WAL 送信者に LCR を使用するように要求します利用可能な場合。

サブスクライバ専用ノードは、各ノードから変更を直接受信します。したがって、加入者専用ノードにサービスを提供する WAL 送信者は、LCR を使用できます。

LCR が生成されても、対応する WAL は、デコードワーカーが有効になっていない場合と同様に保持されます。将来的には、LCR が必要でない場合、LCR に対応する WAL を削除することが可能になる可能性があります。

参考までに、LCR ファイル名の最初の 24 文字は、WAL ファイル名のものと似ています。名前の最初の 8 文字は現在すべて「0」です。将来的には、WAL セグメントファイル名の最初の 8 文字と同様の TimeLineId を表すことが予想されます。次の 16 文字の名前のシーケンスは、WAL ストリームに対して LCR 変更を追跡するために使用される WAL セグメント番号に似ています。

ただし、論理変更は、それらが属しているトランザクションのコミット順序に従って並べ替えられます。したがって、LCR セグメント内のそれらの配置は、WAL セグメント内の対応する WAL の配置と一致しません。

最後の 16 文字のセットは、LCR セグメントのサブセグメント番号を表します。各 LCR ファイルはサブセグメントに対応します。LCR ファイルはバイナリで可変サイズです。`bdr.max_lcr_segment_file_size` デフォルトの 1GB を調整することにより、LCR ファイルの最大サイズを制御できます。

### トランザクションストリーミングでの使用

トランザクションストリーミングでの使用 とデコードワーカーを同時に有効にすることが可能です。トランザクションストリーミングは、WAL 送信者がコミットする前に部分的なトランザクションを送信でき、レプリケーションラグを削減できることを意味します。WAL デコーダーが部分トランザクションのデコードをサポートするようになったため、デコードワーカーは部分トランザクションをデコードし、LCR ファイルに保存できます。次に、LCR ファイルを使用して、サブスクライバノードにトランザクションを適用します。これにより、遅延が減少することにより CPU 使用率が削減され、「.spill」ファイルが生成されないため、ディスク領域の使用量が削減されます。

WAL デコーダーは常にトランザクションを LCR にストリーミングしますが、ダウンストリーム要求に基づいて、WAL 送信者はトランザクションをストリーミングするか、通常の `BEGIN..COMMIT` シナリオを模倣します。

この機能をサポートするために、システムは追加のストリーミングファイルを作成します。これらのファイルは、`STR_TXN_<file-name-format>` および `CAS_TXN_<file-name-format>` で始まる名前があり、ス

トリーミングされた各トランザクションは独自のペアを作成します。

WAL デコーダーを使用したトランザクションストリーミングを有効にするには、bdr.alter_node_group_option を使用して PGD グループの bdr.streaming_mode セットを`default’に設定します。

### 6.3.4 CDC Failover support

#### 背景

PGD の以前のバージョンでは、データベース内のデータに発生した論理変更のフィードを提供できるノードに論理レプリケーションスロットを作成できました。これらの論理レプリケーションスロットはノードに対してローカルであり、レプリケートされていません。特定のノードで変更をレプリケートするだけでなく、この動作は、クラスター内のノードフェールオーバーが発生した場合に課題を提示しました。そのシナリオでは、障害が発生したノードの論理レプリケーションのコンシューマには、別のノードに消費を継続するスロットのレプリカがありません。

この問題へのソリューションは、サブスクライバ専用ノードを中間として使用して設計できますが、論理レプリケーションのコストが大幅に増加します。

#### CDC フェイルオーバーのサポート

このニーズに対応するために、PGD は CDC フェールオーバーサポートを導入しました。これは、クラスター全体で自動論理スロットレプリケーションをアクティブにするオプションで有効な機能です。これにより、論理スロットのレプリケーションのコンシューマは、障害が発生したときに任意のノードから変更データを受信できます。

#### CDC フェールオーバーの仕組み

CDC フェールオーバーサポートが有効になっているノードで論理スロットが作成されると、スロットはクラスター全体でレプリケートされます。これは、クラスター内の任意のノードでスロットを使用できることを意味します。ノードに障害が発生すると、クラスター内の別のノードからスロットが消費される可能性があります。これにより、論理レプリケーションストリームを中断せずに継続できます。

ただし、スロットのコンシューマがクラスター内の別のノードに接続する場合、コンシューマが以前に持っていた接続は PGD によって閉じられます。この動作は、スロットがマルチプルのノードから同時に消費されないようにするためです。バックグラウンドで、PGD は Raft コンセンサスプロトコルを使用して、スロットが一度に 1 つのノードのみから消費されるようにしています。これは、一度に 1 つのスロットのみが消費されるという保証が、スプリットブレインシナリオでは成立しないことを意味します。

現在、CDC フェールオーバーのサポートは、トップグループオプションによって制御されるグローバルオプションです。failover_slot_scope トップグループオプションは現在、論理スロットのレプリケーションを無効にする local または global に設定できます。デフォルト。 global 設定は、PGD データベースで作成されたすべての非一時論理スロットのレプリケーションを有効にします。

一時論理スロットは、有効期間がそれらを作成したセッションにスコープされ、そのセッションが終了すると消失するため、レプリケートされません。

#### 少なくとも 1 回の配信保証

CDC フェールオーバーサポートは、コンシューマーがすべての変更を少なくとも 1 回受信できるようにするための手順を実行します。これは、配信が確認されるまでスロットを抑制することにより行われ、その時点でスロットは非同期方法ですべてのノードで進められます。スロットが消費されていたノードに障害が発生した場合、スロットは、コンシューマがクラスター内のノードに接続するまで保持されます。これにより、スロットが進行できます。

#### 1 回限りの配信

現在、正確に 1 回だけの配信を保証する方法はなく、使用するアプリケーションが、以前に完了したトランザクションの破棄を管理することを期待しています。

### CDC フェールオーバーサポートの有効化

CDC フェールオーバーのサポートを有効にするには、SQL コマンドを実行し、次のパラメーターを使用して bdr.alter_node_group_option ファンクションを呼び出します。

```
select bdr.alter_node_group_option(<top-level group name>,
     failover_slot_scope,
     global);
```

<top-level group name> は、クラスターのトップレベルグループの名前に置き換えます。名前がわからない場合は、bdr.node_group の node_group_parent_id が 0 に等しいグループです。

名前がわからない場合は、bdr.node_group で 0 に等しい node_group_parent_id を持つグループです。次のものも使用できます。

```
SELECT bdr.alter_node_group_option(
     node_group_name,
     failover_slot_scope,
     global)
    from bdr.node_group
    where node_group_parent_id=0;
```

このコマンドにより、正しいトップレベルグループのオプションを設定できます。

CDC フェールオーバーが有効になったら、新しいグローバルにレプリケートされたスロットを作成するには、以下を使用できます。

```
SELECT pg_create_logical_replication_slot(myslot,
                                          test_decoding);
```

オプションが global に設定される前に作成された論理レプリケーションスロットはレプリケートされません。新しいスロットのみが複製されます。

フェールオーバースロットは、レプリケーション接続で CREATE_REPLICATION_SLOT コマンドを使用して作成することもできます。

フェイルオーバースロットのステータスは、 bdr.failover_replication_slots テーブルで追跡されます。

### Postgres 17+での CDC フェールオーバーのサポート

Postgres 17 以降では、フェールオーバーのサポートが追加され、スタンバイを再開できるようになりました。このためには、failover という名前の pg_create_logical_replication_slot のオプションを使用します。この新しい設定では、 failover_slot_scope の設定に関係なく、 failover を true に設定する必要があります。

```
SELECT pg_create_logical_replication_slot(myslot,
                                          test_decoding,
                                          failover=>true);
```

### 一貫した初期スナップショットの取得

論理レプリケーションスロットが作成されると、一貫したスナップショットが Postgres によってエクスポートされます。このスナップショットを使用して、データの一貫した初期コピーを取得できます。PGD のフェイルオーバースロットメカニズムも同じ手順に従います。ただし、コンシューマは、スロットが最初に作成されたノードと同じノードからスナップショットを取得する必要があります。さらに、同じノードから初期レプリケーションを開始する必要があります。コンシューマがレプリケーションストリームを介して十分な変更を受信すると、フェールオーバースロットは failover_safe としてマークされます。スロットが failover_safe としてマークされると、コンシューマは PGD クラスター内の他のノードに安全にフェイルオーバーできます。ただし、他の考慮事項が適用されます。以下を参照してください。

スロットが failover_safe かどうかを確認するには、bdr.failover_replication_slots カタログを照会し、指定されたスロットの failover_safe 列の値を確認できます。

スロットが failover_safe とマークされる前に、コンシューマが他の PGD ノードに接続し、レプリケーションを開始しようとすると、PGD によって適切なエラーが発生します。

### 新しく参加したノードへのフェイルオーバー

新しいノードが PGD クラスターに参加すると、CDC フェールオーバースロットのデコードターゲットとして機能する準備がすぐに整わない場合があります。新しく参加したノードには、コンシューマがまだ使用していない変更をデコードするためのすべての WAL ファイルがない場合があります。このようなノードから消費すると、データの損失が発生する可能性があります。PGD は、レプリケーションの進行状況を内部で追跡し、安全なまで新しいノードがフェールオーバーターゲットになるのを防ぐことにより、このような状況を検出して防止します。コンシューマが、デコードターゲットとして機能する準備がまだ整っていないノードに接続しようとすると、適切なエラーが発生します。

### オリジンごとの進行状況の追跡

トランザクションは、PGD クラスター内の任意のノードから発生できます。コンシューマが PGD ノードに接続し、トランザクションのデコードを開始すると、そのノードで発生したトランザクションと、クラスター内の他のノードからレプリケートされたトランザクションの変更を受信する場合があります。コンシューマは、このようなすべての PGD ノードまたはオリジンにわたってレプリケーションの進行状況を追跡し、重複したトランザクションが正しく処理されることを確認することが求められます。これを促進するために、Postgres-Extended および EnterpriseDB Advance Server の test_decoding プラグインが機能強化され、トランザクションの起点情報が含まれるようになりました。コンシューマは、論理レプリケー

ションの開始時に include-origin オプションを on に設定することにより、オリジン情報を受信することを選択できます。

以下は、起点情報を含む test_decoding プラグインのサンプル出力です。

```
BEGIN 1723654 (origin 2) (origin_name bdr_bdrdemo_bdrgroup_node2) (origin_lsn 0/
↪1D948910)
table public.pgbench_accounts: UPDATE: old-key: aid[integer]:39958 bid[integer]:1␣
↪abalance[integer]:0 filler[character]:
                                        new-tuple: aid[integer]:39958 bid[integer]:1␣
↪abalance[integer]:-1783 filler[character]:
table public.pgbench_tellers: UPDATE: old-key: tid[integer]:6 bid[integer]:1␣
↪tbalance[integer]:0 new-tuple: tid[integer]:6 bid[integer]:1 tbalance[integer]:-
↪1783 filler[character]:null
table public.pgbench_branches: UPDATE: old-key: bid[integer]:1 bbalance[integer]:0␣
↪new-tuple: bid[integer]:1 bbalance[integer]:-1783 filler[character]:null
table public.pgbench_history: INSERT: tid[integer]:6 bid[integer]:1␣
↪aid[integer]:39958 delta[integer]:-1783 mtime[timestamp without time zone]:2025-01-
↪31 16:51:21.511571 filler[character]:null
COMMIT 1723654 (origin 2) (origin_name bdr_bdrdemo_bdrgroup_node2) (origin_lsn 0/
↪1D948910)
```

コンシューマはこの情報を利用して、オリジンごとの進行状況を追跡できます。

PGD は、bdr.logical_checkpoints カタログ内のすべてのノードにわたるレプリケーションの進行状況も記録し、コンシューマはカタログのデコードされた変更を受信し、その情報を使用してレプリケーションの進行状況を知ることができます。

```
⊠⊠ 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
```

この例では、ノード 370098259 がレプリケーションの進行状況を報告しています。コンシューマがこの変更レコードを受信すると、ノード 2228531844 および 4052927809 からそれぞれ 0/4836758 および 0/67F0AF8 までのすべてを受信したことを確認できます。

### 制限事項

CDC フェールオーバースロットのサポートには、特定の制限があります。

- CDC フェールオーバースロットのサポートには、 EDB Postgres DistributedPGD 5.7+の最新バージョンと Postgres Extended または EDB Postgres Advanced Server の最新のマイナーリリース 2025 年 2 月利用可能が必要です。
- CDC フェールオーバーのサポートはグローバルオプションであり、スロットごとに設定はできません。CDC フェールオーバーの有効なステータスの変更は、以前にプロビジョニングされたスロット

に影響を与えないため、これを有効にして `global` に設定し、レプリケートスロットを作成してから無効にして `local` に設定して、単一のレプリケートスロットを作成することが可能です。

- CDC フェールオーバーのサポートは、一時的なスロットではサポートされていません。
- CDC フェールオーバーのサポートは、 `failover` オプションを `false` に設定して作成されたスロットではサポートされていません。
- CDC フェールオーバーのサポートは、EDB Postgres Advanced Server および EDB Postgres Extended Server でのみ動作します。コミュニティ Postgres インストールではサポートされていません。
- オプションが有効になっている場合、既存のスロットはフェールオーバースロットに変換されません。
- `pg_logical_slot_get_changes()` などの Postgres のビルトイン機能は使用できますが、スロットが他の場所でデコードされていないことは保証されず、クラスター全体のレプリケーションの進行状況を正確に更新できません。したがって、デコードされた変更を受信するファンクションに依存しないことをお勧めします。

### 6.3.5 Parallel Apply

#### パラレル適用とは何ですか?

Parallel apply は、PGD ノードがサブスクリプションごとにマルチプルのライターを使用できるようにする PGD の機能です。この動作は、一般にサブスクリプションのスループットを向上させ、レプリケーションパフォーマンスを向上させます。

#### パラレル適用の構成

2 つの変数は PGD のパラレル適用を制御します bdr.max_writers_per_subscription

デフォルト 8 および bdr.writers_per_subscription

デフォルト 2。

```
bdr.max_writers_per_subscription = 8
bdr.writers_per_subscription = 2
```

この構成では、各サブスクリプションに 2 つのライターを与えます。ただし、状況によっては、システムはサブスクリプションに最大 8 つのライターを割り当てる場合があります。

bdr.max_writers_per_subscription の変更

有効にするにはサーバーを再起動する必要があります。

bdr.writers_per_subscription を変更できます

再起動なしの特定のサブスクリプションの場合

1. bdr.alter_subscription_disable を使用してサブスクリプションを停止します。
2. 新しい値の設定。
3. bdr.alter_subscription_enable を使用してサブスクリプションを再開します。

ただし、最初に、 `select * from bdr.subscription` を使用してサブスクリプションの名前を確立します。この例では、サブスクリプション名は `bdr_pgddb_bdrgroup_node2_node1` です。

```
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);
```

#### Parallel apply を使用する場合

Parallel apply はデフォルトで常にオンになっており、ほとんどの操作では、オンのままにすることをお勧めします。

#### モニタリング パラレル 適用

Parallel apply のデッドロックの緩和をサポートするために、PGD は bdr.stat_subscription に列を追加します。新しい列は `nprovisional_waits` 、`ntuple_waits` 、および `ncommmit_waits` です。これらは、Parallel apply が、以前はデッドロックであったものをどの程度適切に管理しているかを示すメトリックです。これらはシステム全体のパフォーマンスを反映するものではありません。

`nprovisional_waits` 値は、同時適用トランザクションによって実行されている同じタプルの操作の数を反映しています。これらは暫定的な待機であり、実際にはまだ待機していませんが、待機を開始する可能性があります。

タプルの書き込みが安全に適用できるまで待機する必要がある場合、`ntuple_waits` でカウントされます。コミットされる前に待機した完全に適用されたトランザクションは、`ncommit_waits` でカウントされます。

#### パラレル適用の無効化

Parallel apply を無効にするには、 bdr.writers_per_subscription を 1 に設定します。

#### デッドロックの軽減

Parallel apply が動作している場合、サブスクリプションからのトランザクションの変更は、マルチプルのライターによって書き込まれます。ただし、各ライタは、トランザクションの最終コミットが、オリジンノードで実行されたコミット順序に違反していないことを保証します。違反がある場合は、エラーが生成され、トランザクションをロールバックできます。

このメカニズムは、以前は、次のことがすべて true の場合、結果のエラーがデッドロックとして現れる可能性があることを意味しました。

- トランザクションはコミット保留中で、別のトランザクションが変更する必要がある行を変更します。
- 保留中のトランザクションが実行する前に、オリジンノードで他のトランザクションが実行されました。
- 保留中のトランザクションはロック要求を取り出しました。

さらに、エラーを処理すると、にかかった時間の組み合わせにより、レプリケーションラグが増加する可能性があります。

- デッドロックを検出するには
- クライアントがトランザクションをロールバックする場合
- すでに適用された変更の間接的なガベージコレクションの場合
- 作業をやり直すには

ここで、Parallel apply のデッドロックの緩和が役立ちます。いかなるトランザクションでも、Parallel apply は、現在のトランザクションが書き込む行タプルに既にスケジュールされているトランザクションを調べます。見つかった場合、行は、行に変更を適用する前に、他のトランザクションがコミットされるまで待機する必要があるとしてマークされます。このアプローチにより、行が正しい順序で書き込まれます。

#### パラレル適用のサポート

PGD 6 では、 Parallel apply は CAMO と連携します。 Group Commit または Eager Replication と互換性がないため、 Group Commit または Eager Replication を使用している場合は無効にします。

### 6.3.6 Replication sets

レプリケーションセットは、PGD ノードがサブスクライブできるテーブルのグループです。レプリケーションセットを使用して、各ノードが他のノードの正確なコピーである通常の対称マルチマスタートポロジよりも複雑なレプリケーショントポロジを作成できます。

すべての PGD グループは、グループと同じ名前のレプリケーションセットを作成します。このレプリケーションセットは、すべてのユーザーテーブルと DDL レプリケーションに使用されるデフォルトのレプリケーションセットです。すべてのノードがそれにサブスクライブされます。つまり、デフォルトでは、すべてのユーザーテーブルがすべてのノード間でレプリケートされます。

#### レプリケーションセットを使用する

bdr.create_replication_set を使用して、挿入、更新、削除、または切り捨てアクションを含めるかどうかを指定して、レプリケーションセットを作成できます。1 つのオプションでは既存のテーブルをセットに追加でき、2 番目のオプションでは作成時にテーブルを追加するかどうかを定義します。

テーブルを手動で定義して、レプリケーションセットに追加または削除することもできます。

レプリケーションセットに含まれるテーブルは、ノードがクラスターに参加したときおよびその後に維持されます。

ノードが参加したら、レプリケーションセットからテーブルを削除できますが、再同期操作を使用して新しいテーブルを追加する必要があります。

デフォルトでは、新しく定義されたレプリケーションセットは DDL または PGD 管理ファンクション呼び出しをレプリケートしません。bdr.replication_set_add_ddl_filter を使用する

複製するコマンドを定義します。

PGD は、すべてのノードでレプリケーションセット定義を作成します。次に、bdr.alter_node_replication_sets を使用して各レプリケーションセットをパブリッシュまたはサブスクライブするように各ノードを定義できます。

ファンクションを使用して、これらの定義を後で変更したり、レプリケーションセットを削除したりできます。

> **注釈**
>
> 選択的レプリケーションにデフォルトのレプリケーションセットを使用しないでください。クラスター内の PGD ノードにあるデフォルトのレプリケーションセットは、 DDL レプリケーションおよび管理ファンクション呼び出しにもデフォルトで使用されるため、ドロップまたは変更しないでください。

### パーティション分割テーブルの動作

PGD は、パーティション化テーブルを透過的にサポートします。つまり、パーティション化されたテーブルをレプリケーションセットに追加できます。

パーティションに関係する変更は、ダウンストリームにレプリケートされます。

> **注釈**
>
> パーティションがパーティションテーブルを介してレプリケートされる場合、パーティションで直接実行されるステートメントは、親テーブルで実行されたときと同じようにレプリケートされます。例外は `TRUNCATE` コマンドであり、これは常に影響を受けるテーブルまたはパーティションのリストを使用して複製します。

個々のパーティションをレプリケーションセットに追加できます。この場合、それらは通常のテーブルと同様に、つまり、ダウンストリームのパーティションと同じ名前のテーブルにレプリケートされます。パーティショニング定義がプロバイダーとサブスクライバーの両方で同じである場合、パーティショニングロジックを実行する必要がないため、この動作にはいくつかのパフォーマンスの利点があります。

> **注釈**
>
> ルートパーティションテーブルがレプリケーションセットの一部である場合、個々のパーティションのメンバーシップは無視されます。そのルートテーブルのメンバーシップのみが考慮されます。

### 外部キーを使用した動作

外部キー制約は、参照テーブルの各行が参照先テーブルの行と一致することを保証します。したがって、参照テーブルがレプリケーションセットのメンバーである場合、参照されるテーブルも同じレプリケーションセットのメンバーである必要があります。

PGD の現在のバージョンは、この条件をチェックまたは適用しません。レプリケーションセットにテーブルを追加するとき、データベース管理者は、外部キーによって参照されるすべてのテーブルも追加されていることを確認する必要があります。

次のクエリを使用して、この要件を満たさないすべての外部キーとレプリケーションセットをリストできます。参照テーブルはレプリケーションセットのメンバーですが、参照されるテーブルはレプリケーションセットのメンバーではありません。

```
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
);
```

このクエリの出力は次のようになります。

```
 relname | nspname |  conname   | set_name
- --------+---------+-----------+----------
 t2      | public  | t2_x_fkey | s2
(1 row)
```

この出力は、テーブル t2 がレプリケーションセット s2 のメンバーであるが、外部キー t2_x_fkey によって参照されるテーブルがそうでないことを意味します。

TRUNCATE CASCADE コマンドは、コマンドをレプリケートする前にレプリケーションセットのメンバーシップを考慮します。例

```
TRUNCATE table1 CASCADE;
```

これは、レプリケーションセットのみの一部であるすべてのテーブルでカスケードなしの TRUNCATE になります。

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

### レプリケーションセットのメンバーシップ

1 つ以上のレプリケーションセットにテーブルを追加または削除できます。そうすると、これらのテーブルの変更 DML のレプリケーションにのみ影響します。スキーマ変更 DDL は、DDL レプリケーションセットフィルターによって処理されます *DDL* レプリケーションフィルタリング を参照してください。

レプリケーションは、ノードレプリケーションセット構成を使用してレプリケーションセットのテーブルメンバーシップを使用して、レプリケートするアクションとそれらのレプリケート先のノードを決定します。決定は、すべてのメンバーシップとレプリケーションセットオプションの和集合を使用して行われま

す。テーブルが、INSERT アクションのみをレプリケートするレプリケーションセット A と、UPDATE アクションのみをレプリケートするレプリケーションセット B のメンバーであるとします。ターゲットノードがレプリケーションセット A および B の両方にサブスクライブされている場合、INSERT および UPDATE アクションは両方ともレプリケートされます。

bdr.replication_set_add_table および bdr.replication_set_remove_table を使用してメンバーシップを制御できます。

### レプリケーションセットのリスト

次のクエリーを使用して、既存のレプリケーションセットをリストできます。

```
SELECT set_name
FROM bdr.replication_sets;
```

このクエリを使用して、特定のレプリケーションセット内のすべてのテーブルをリストできます。

```
SELECT nspname, relname
FROM bdr.tables
WHERE set_name = myrepset;
```

外部キーを使用した動作 は、参照されるテーブルが参照テーブルと同じレプリケーションセットに含まれていないすべての外部キーをリストするクエリーを示しています。

次の SQL を使用して、現在のノードが発行およびサブスクライブするレプリケーションセットを表示します。

```
SELECT node_id,
       node_name,
       pub_repsets,
       sub_repsets
  FROM bdr.local_node_summary;
```

このコードは、次のような出力を生成します。

```
  node_id   | node_name |    pub_repsets     |    sub_repsets
- ----------+-----------+--------------------------------------
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
(1 row)
```

クラスター内のすべてのノードに対して同じクエリーを実行するには、次のクエリーを使用できます。このアプローチでは、すべてのノードに関連付けられたレプリケーションセットを同時に取得します。

```
WITH node_repsets AS (
  SELECT jsonb_array_elements(
    bdr.run_on_all_nodes($$
        SELECT
          node_id,
```

(次のページに続く)

(前のページからの続き)

```
          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;
```

これは、例を示します。

```
  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 レプリケーションフィルタリング

デフォルトでは、サポートされているすべての DDL のレプリケーションは、デフォルトの PGD グループレプリケーションセットを介して発生します。このレプリケーションは、PGD グループと同じ名前の DDL フィルターを使用して実現されます。このフィルタは、PGD グループの作成時に、デフォルトの PGD グループレプリケーションセットに追加されます。

既存のすべてのレプリケーションセットの DDL レプリケーションフィルターを変更することにより、この動作を調整できます。これらのフィルターは、レプリケーションセットのテーブルメンバーシップとは独立しています。データ変更と同様に、各 DDL ステートメントは、複数のレプリケーションセットの複数のフィルターと一致する場合でも、1 回だけレプリケートされます。

次のクエリーを使用して既存の DDL フィルターをリストできます。これは、フィルターごとに、コマンドタグとロール名に適用された正規表現を示します。

```
SELECT * FROM bdr.ddl_replication;
```

bdr.replication_set_add_ddl_filter および bdr.replication_set_remove_ddl_filter を使用して、DDL フィルターを操作できます。これらは DDL であると考えられるため、DDL レプリケーションとグローバルロックの対象になります。

### 選択的レプリケーションの例

この例では、テーブルを特定のノードのグループに選択的にレプリケートするように EDB Postgres 分散を構成します。

### クラスター構成

この例では、6 つのデータノード、data-a1 ～data-a3 および data-b1 ～data-b3 のクラスターが 2 つの場所にあることを前提としています。これらがメンバーである 2 つのロケーションは、region_a および region_b グループとして表されます。

また、お勧めのように、この例では記載されていませんが、 region-c に witness という名前の監視ノードがあります。クラスターの名前は sere です。

### 応募要項

この例は、音楽作品の公演に出席した人々の意見を記録するアプリケーションで動作します。出席者用のテーブル、作品用のテーブル、そしてオピニオンテーブルがあります。オピニオンテーブルには、各出席者が見た各作品、どこで、いつ見たか、および作品をどのように採点したかが記録されます。データ規制のため、この例では、意見データは、意見が記録されたリージョンにのみ存在する必要があることを前提としています。

### テーブルの作成

最初のステップは、適切なテーブルを作成することです。

```
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
);
```

### グループとレプリケーションセットの表示

デフォルトでは、 EDB Postgres 分散は、各テーブル全体を各すべてのノードにレプリケートするように構成されます。これは、レプリケーションセットを介して管理されます。

初期構成のデフォルトレプリケーションセットを表示するには、次のコマンドを実行します。

```
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
```

出力には、トップレベルのグループ sere があり、 sere という名前のデフォルトのレプリケーションセットがあることがわかります。3 つのサブグループには、それぞれ、サブグループと同じ名前のレプリケーションセットがあります。region_a グループには、region_a デフォルトレプリケーションセットがあります。

デフォルトでは、既存のすべてのテーブルと新しいテーブルは、トップレベルグループのレプリケーションセットのメンバーになります。

### レプリケーションセットへのテーブルの追加

次の手順は、リージョンを表すグループに属するレプリケーションセットにテーブルを追加することです。前述のように、新しいテーブルはすべて sere レプリケーションセットに自動的に追加されます。次を実行して、それを確認できます。

```
SELECT relname, set_name FROM bdr.tables ORDER BY relname, set_name;
```

```
 relname  | set_name
- ---------+----------
 attendee | sere
 opinion  | sere
 work     | sere
(3 rows)
```

opinion テーブルは、region_a でのみ、別に region_b でのみレプリケートする必要があります。これを行うには、各リージョンのレプリカセットにテーブルを追加します。

```
SELECT bdr.replication_set_add_table(opinion, region_a);
SELECT bdr.replication_set_add_table(opinion, region_b);
```

しかし、opinion はまだ sere レプリケーションセットのメンバーであるため、これで完了ではありません。テーブルが複数のレプリケーションセットのメンバーである場合、各テーブルでレプリケートされます。ただし、各行は各ターゲットノードで 1 回のみレプリケートされるため、これはパフォーマンスに影

響しません。opinion をすべてのノードでレプリケートしたくないため、トップレベルグループのレプリケーションセットから削除する必要があります。

```
SELECT bdr.replication_set_remove_table(opinion, sere);
```

これらの変更を確認できるようになりました。

```
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)
```

このプロセスにより、目的の選択的レプリケーションが提供されます。行われたかどうかを確認するには、次の手順を使用してテストします。

### 選択的レプリケーションのテスト

最初にいくつかのテストデータ 2 つの作品と出席者を作成します。 data-a1 に直接接続して、次のコードを実行します。

```
INSERT INTO work VALUES (1, Aida, Verdi);
INSERT INTO work VALUES (2, Lohengrin, Wagner);
INSERT INTO attendee (email) VALUES (gv@example.com);
```

これらのテーブルにデータが追加されたため、外部キー制約に違反せずに opinion テーブルに挿入できます。

```
INSERT INTO opinion (work_id, attendee_id, country, day, score)
SELECT work.id, attendee.id, Italy, 1871-11-19, 3
  FROM work, attendee
 WHERE work.title = Lohengrin
   AND attendee.email = gv@example.com;
```

挿入が完了したら、同じノードでデータベースの内容を検証できます。

```
SELECT a.email
, o.country
, o.day
, w.title
, w.author
, o.score
FROM opinion o
```

(次のページに続く)

(前のページからの続き)

```
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)
```

ここで、ノード data-a2 および data-a3 に接続し、同じクエリーを実行すると、同じ結果が得られます。データは region_a で複製されています。 data-b1 、data-b2 、または data-b3 に接続する場合、クエリーは行を返しません。これは、 attendee および work テーブルにデータが入力されていますが、選択する opinion 行がないためです。これは、 region_a での opinion のレプリケーションがそのリージョンでのみ発生するためです。

次に、 data-b1 に接続し、そこに意見を挿入します。

```
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;
```

この意見は region_b でのみ複製されます。 data-b1 、data-b2 、および data-b3 では、次を実行できます。

```
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)
```

各 region_b データノードで同じ結果が表示されます。 region_a ノードでクエリーを実行すると、この特定のエントリが表示されません。

最後に、 attendee テーブルがすべてのノードで同様に共有されていることに注意してください。任意のノードで、クエリを実行します。

```
SELECT * FROM attendee;
```

```
        id         |    email
- ------------------+----------------
 904252679641903104 | gv@example.com
 904261037006536704 | fb@example.com
(2 rows)
```

### 6.3.7 Security and roles

EDB Postgres Distributed を使用すると、デザインで保存されたデータへのアクセスを付与せずに PGD クラスターを管理できます。これは、ロールとシステムオブジェクトへの制御されたアクセスを使用して実現します。

- *Security and roles* は、PGD 機能へのアクセスを制御するために PGD が事前定義するロールを導入します。
- *Role management* は、マルチデータベースノードと新しいノードでロールが管理される方法について説明します。
- *PGD predefined roles* は、 PGD ロールの特定の特権を詳細に説明します。
- *Roles and replication* は、PGD レプリケーションがロールおよび特権とどのように相互作用するかについて説明します。
- *Access control* は、テーブル、ファンクション、カタログオブジェクト、トリガーが PGD ロールおよび Postgres 属性とどのように相互作用するかについて説明します。

#### Roles

PGD の構成と管理にはスーパーユーザーアクセスは必要ありません。スーパーユーザーアクセスを使用しないことをお勧めします。代わりに、PGD の管理に必要な特権は、次の事前定義されたロールに分割されます。

| Role | Description |
|---|---|
| *bdr_superuser* | すべての PGD テーブルとファンクションにアクセスする最高の特権ロール。 |
| *bdr_read_all_stats* | PGD の状態を理解するために十分な、テーブル、ビュー、およびファンクションへの読み取り専用アクセスを持つロール。 |
| *bdr_monitor* | bdr_read_all_stats の特権と、モニタリングのための追加の特権が含まれます。 |
| *bdr_application* | PGD を実行するアプリケーションに必要な最小限の特権。 |
| *pg_upgrade* | :ref:``bdr.conflict_history` 列<*bdr.conflict_history* 列>` のすべての競合を表示できます。 |

これらのロールは、 PostgreSQL の pg_ *predefinedroles* に似た名前が付けられます。

PGD bdr_ ロールは、BDR 拡張機能のインストール時に作成されます。各ロールが持つ特権の詳細については、*PGD predefined roles* を参照してください。

PGD の管理には、管理者がユーザーデータにアクセスする必要はありません。

競合に関する情報を保護するための取り決めについては、Logging conflicts to a table で説明されています。

bdr.conflict_history_summary ビューを使用して競合をモニタできます。

### Role management

ユーザーは、PostgreSQL インスタンスのグローバルオブジェクトです。`CREATE ROLE` コマンドまたはそのエイリアス `CREATE USER` は、PGD レプリケートデータベースで実行される場合、自動的にレプリケートされます。ロールまたはユーザーが非 PGD、非レプリケートデータベースで作成された場合、ロールはその PostgreSQL インスタンスにのみ存在します。`GRANT ROLE` および `DROP ROLE` は同じように動作し、PGD レプリケートデータベースに適用される場合にのみレプリケートします。

> **注釈**
>
> Postgres 用語でのユーザーは、単にログイン権限を持つロールであることに注意してください。

#### ロールルール - 未複製のロールはありません

非 PGD の非レプリケートデータベースにロールまたはユーザーを作成する場合、PGD レプリケートデータベースのオブジェクトをそのロールに依存させないことが特に重要です。PGD は、PGD レプリケートデータベースにないロールをレプリケートできないため、レプリケーションプロセスが中断します。

bdr.role_replication 設定をオフにすることにより、この自動レプリケーション動作を無効にできますが、それはお勧めしません。

#### 新しいノードのロール

bdr_init_physical を使用して追加された新しい PGD ノードは、PGD クラスターの他のノードからロールを自動的にレプリケートします。

PGD 6.0.1 以降、`bdr_init_physical` を使用せずに PGD ノードが PGD グループに手動で参加すると、既存のロールが新しく参加したノードに自動的にコピーされます。これは、グループに参加する前に、新しいノードでロールを手動で作成する必要がなくなることを意味します。

ロールが新しいノードにコピーされるときに、同じ名前の既存のロールまたはテーブルスペースが存在する場合、新しいノードの既存のロールまたはテーブルスペースは更新されて、ロールまたはテーブルスペースと同じ設定パスワードを共有します) 参加操作のソースノードで。

#### 接続とロール

新しい PGD ノードを割り当てるときに、 bdr.create_node の `local_dsn` 引数と bdr.join_node_group の `join_target_dsn` の DSN で指定されたユーザー

データベースオブジェクトを参照、作成、および管理するために頻繁に使用されます。

PGD は、これらの DSN で `SUPERUSER` 権限を持つロールを使用する場合でも、特権エスカレーション攻撃を防ぐように注意深く書き込まれています。

攻撃対象領域をさらに削減するために、これらの DSN でより制限されたユーザーを指定できます。少なくとも、このようなユーザーには、次の条件が満たされるように、すべてのノードで権限を付与する必要があります。

- ユーザは REPLICATION 属性を持っています。
- データベースに対する CREATE 権限が付与されています。
- bdr_superuser ロールを継承します。
- 複製するすべてのデータベースオブジェクトを直接、または所有者ロールの権限から所有します。

また、デフォルト以外の拡張機能 BDR 拡張機能を除くがソースノードに存在し、スーパーユーザーのみがインストールできる場合、スーパーユーザーは参加ターゲットノードでこれらの拡張機能を手動で作成する必要があります。それ以外の場合、参加プロセスは失敗します。

PostgreSQL 13 以降では、スーパーユーザーの権限を必要とし、手動でインストールする必要がある拡張機能を特定できます。ソースノードで、次を実行します。

```
SELECT name, (trusted IS FALSE AND superuser) AS superuser_only
  FROM pg_available_extension_versions
 WHERE installed AND name != bdr;
```

すべてのノードが参加したら、DML および DDL レプリケーションを引き続き許可するために、次の権限をさらに削減できます。

- ユーザは REPLICATION 属性を持っています。
- bdr_superuser ロールを継承します。

### PGD predefined roles

PGD の事前定義されたロールは、BDR 拡張機能のインストール時に作成されます。BDR 拡張機能がデータベースから削除された後、ロールは存在し続けます。ドロップが必要な場合は、手動でドロップする必要があります。

### bdr_superuser

このロールは、PGD 関連のすべてを管理できる管理ユーザー用です。データベースとテーブルアクセスを分離して管理できます。これを使用すると、PostgreSQL のスーパーユーザー権限を付与せずに PGD クラスターを管理できるユーザーができます。

#### 特権

- スキーマ BDR のすべてのテーブルのすべての特権
- スキーマ BDR のすべてのルーチンのすべての特権

### bdr_read_all_stats

このロールは、ユーザーまたはアプリケーションが PGD クラスターの統計と状態を観察するために必要なほとんどのテーブル、ビュー、およびファンクションへの読み取りアクセスを提供します。

#### 特権

次の SELECT 特権

- 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`

以下の EXECUTE 特権

- `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

このロールは、ユーザーまたはアプリケーションが PGD クラスターを監視するために必要なテーブル、ビュー、および機能への読み取りアクセスを提供します。これには、*bdr_read_all_stats* ロールのすべての特権が含まれています。

#### 特権

*bdr_read_all_stats* のすべての特権に加えて、次の追加特権。

次の SELECT 特権

- bdr.group_raft_details
- bdr.group_replslots_details
- bdr.group_subscription_summary
- bdr.group_versions_details
- `bdr.raft_instances`

次の EXECUTE 特権

- 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

このロールは、PGD 機能、オブジェクト、シーケンス、CRDT データ型、CAMO ステータス機能、トリガー管理機能などの機能へのアクセスを必要とするアプリケーション向けに設計されています。

#### 特権

次の EXECUTE 特権

- column_timestamps データ型のすべてのファンクション
- CRDT データ型のすべてのファンクション
- 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

これらの機能の多くは、使用する前に追加の特権を必要とします。たとえば、bdr.alter_sequence_set_kind を正常に実行するには、テーブルの所有者である必要があります。これらの追加ルールは、特定の機能ごとに説明します。

#### bdr_read_all_conflicts

PGD は競合を bdr.conflict_history にログ記録します

テーブル。競合はテーブルの所有者にのみ可視されるため、所有者が競合履歴を読み取るために追加の特権は必要ありません。

ただし、すべてのテーブルの競合を表示できるユーザーが便利な場合は、オプションでそのユーザーにロール bdr_read_all_conflicts を付与できます。

#### 特権

bdr.conflict_history に明示的なポリシーが設定され、このロールが bdr.conflict_history テーブルを読み取ることができます。

### Roles and replication

#### DDL および DML レプリケーションとユーザー

ユーザーが実行した DDL 変更は、各ノードでその同じユーザーとして適用されます。

テーブルへの DML 変更は、ターゲットノード上のテーブル所有ユーザーとして複製されます。

デフォルトでは、PGD はノード全体に同じ所有者を持つ新しいテーブルをレプリケートします。

#### テーブルの所有権の違い

同じユーザーが各ノードのテーブルを所有することをお勧めします。これはデフォルトの動作ですが、オーバーライドできます。その場合、考慮すべきことがいくつかあります。

テーブル A がノード 1 のユーザー X が所有し、ノード 2 のユーザー Y が所有している状況を考えます。ユーザー Y がユーザー X よりも高い特権を持っている場合、これは特権エスカレーションと見なされる場合があります。

ノードにはさまざまなユースケースがある可能性があるため、このシナリオを許可します。しかし、私たちはそれに対して警告します。テーブルの所有者が異なるノードに異なる場合、セキュリティ管理者にこの構成の計画と監査を手伝ってもらうことをお勧めします。

#### レプリケーションと行レベルのセキュリティ

行レベルのセキュリティポリシーが有効になっているテーブルでは、適用時にポリシーを強化せずに変更がレプリケートされます。この動作は、FORCE ROW LEVEL SECURITY が指定されている場合でも、NO FORCE ROW LEVEL SECURITY として適用される変更と同等です。これは希望するものでない場合は、すべての行のレプリケートを回避する row_filter を指定します。すべてのノードの行セキュリティポリシーは同一であるか、少なくとも互換性があることをお勧めしますが、これは強制しません。

#### bdr_superuser のロールとレプリケーション

ユーザー bdr_superuser は、PGD のレプリケーションを制御し、レプリケーションセットから任意のテーブルを追加または削除できます。bdr_superuser は、個々のテーブルに対する特権を必要とせず、お勧めしません。レプリケーションセットファンクションへのアクセスを制限する必要がある場合は、これらのファ

ンクションの制限バージョンを SECURITY DEFINER ファンクションとして実装し、適切なユーザーに付与できます。

#### 特権制限

PGD は追加の制限を適用し、TRIGGER または REFERENCES 特権のみに依存する DDL の使用を効果的に防止します。

GRANT ALL は引き続き TRIGGER と REFERENCES 特権の両方を付与するため、特権を明示的に記述することをお勧めします。たとえば、ALL の代わりに GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE を使用します。

#### 外部キー特権

ALTER TABLE ... ADD FOREIGN KEY は、ユーザーが参照されるテーブルに対する SELECT 特権を持っている場合、または参照先テーブルで現在のユーザーがバイパスできない RLS 制限が有効になっている場合にのみサポートされます。

これは、REFERENCES 特権だけでは PGD で外部キーを作成するのに十分でないことを意味します。REFERENCES 特権のみに依存すると、テーブルスキャンではなくトリガーを使用して検証チェックが実行されるため、通常は役立ちません。通常、コストが高すぎるため、正常に使用できません。

### Access control

#### カタログテーブル

システムカタログと情報スキーマテーブルは、PGD によってレプリケーションから常に除外されます。

さらに、拡張機能が所有するテーブルはレプリケーションから除外されます。

#### PGD ファンクションと演算子

すべての PGD ファンクションは bdr スキーマで公開されています。これらのファンクションへの呼び出しは、search_path に bdr を配置するのではなく、スキーマ修飾する必要があります。

すべての PGD 演算子は pg_catalog スキーマを介して使用できるため、ユーザーは search_path から public スキーマを問題なく除外できます。

#### カタログオブジェクトに対する特権の付与

管理者は、テーブル、ビュー、ファンクションなどのカタログオブジェクトに対する明示的な特権を付与してはなりません。*PGD predefined roles* で説明されているロールのいずれかを付与して、これらのオブジェクトへのアクセスを管理します。

この要件は、結合のどちらの側のノードも全く同じバージョンの PGD、したがって PGD カタログを持たない場合でも、ノードグループに参加できる柔軟性の結果です。

より正確には、個々のカタログオブジェクトの特権が明示的に付与された場合、参加するノードから抽出された対応する GRANT ステートメントが参加しているノードに適用されない可能性があるため、bdr.join_node_group() プロシージャーは失敗する可能性があります。

#### トリガー

PostgreSQL では、テーブルの所有者と TRIGGER 特権を付与された人は誰でもトリガーを作成できます。非テーブル所有者によって付与されたトリガーは、PGD でテーブル所有者として実行されます。これは、セキュリティ問題を発生させる可能性があります。TRIGGER 特権はめったに使用されず、PostgreSQL コアチームは、「個別の TRIGGER 権限は時代遅れだと考えられるものです。」と述べています。

PGD は、テーブルにトリガーを作成できる人に関するより厳密なルールを使用することにより、この問題を緩和します。

- スーパーユーザー トリガーを作成できます。
- bdr_superuser トリガーを作成できます。
- テーブルの所有者 PostgreSQL と同じルールに従ってトリガーを作成できます。トリガーによって使用されるファンクションの EXECUTE 特権が必要です。
- テーブルに対する TRIGGER 特権を持つユーザーテーブルと同じ所有者が所有するファンクションを使用し、標準の PostgreSQL ルールを満たす場合にのみトリガーを作成できます。具体的には、ファンクションに対する EXECUTE 特権が必要です。テーブルとファンクションの両方の所有者が同じで、所有者がユーザーにテーブルの TRIGGER 特権とファンクションの EXECUTE 特権の両方を付与することにした場合。そのユーザーがこのファンクションを使用してそのテーブルにトリガーを作成してもよいと想定されています。
- テーブルに対する TRIGGER 特権を持つユーザー *SECURITY DEFINER clause* で定義されたファンクションを使用してトリガーを作成することもできます

EXECUTE 権限がある場合。 SECURITY DEFINER 句により、ファンクションは常に標準の PostgreSQL と PGD の両方でファンクションの所有者として実行されます。

このロジックは、PostgreSQL では、トリガーの所有者がそれを作成したユーザーではなく、そのトリガーが使用するファンクションの所有者であるという事実に基づいて構築されています。

同じルールが既存のテーブルに適用され、既存のテーブルに、テーブルの所有者が所有せず、SECURITY DEFINER ファンクションを使用しないトリガーがある場合、レプリケーションセットに追加することはできません。

PGD レプリケーションは変更を適用する場合、システムレベルのデフォルトの search_path のみを使用します。他の search_path 設定を想定しているレプリカトリガー、ストリームトリガー、およびインデックス式ファンクションは、適用時に実行すると失敗します。これが発生しないようにするには、デフォルトの search_path のみを使用してオブジェクト参照を明確に解決するか、影響を受けるファンクションに `ALTER FUNCTION ... SET search_path = ...` を使用してファンクションの検索パスを設定します。デフォルトの search_path を使用する場合、schema.objectname などのオブジェクトへの完全修飾参照を常に使用します。

### 6.3.8 Sequences

多くのアプリケーションでは、一意のサロゲート ID をデータベースエントリに割り当てる必要があります。多くの場合、データベース `SEQUENCE` オブジェクトはこれらを生成するために使用されます。PostgreSQL では、これらは次のいずれかです。

- CREATE SEQUENCE コマンドを使用して手動で作成し、nextval() ファンクションを呼び出して取得したシーケンス
- serial および bigserial 列または GENERATED BY DEFAULT AS IDENTITY 列

ただし、PostgreSQL の標準シーケンスはマルチノード対応ではなく、ローカルノードでのみ一意の値を生成します。このようなシーケンスによって生成された一意の ID は、マルチマスターレプリケーションで破棄された INSERT アクションによる競合とデータ損失を発生させるため、これは重要です。

### PGD グローバルシーケンス

このため、 PGD は、 **グローバルシーケンス** と呼ばれる、PGD グループ全体の bigint または bigserial データ型のシーケンスを使用して、一意の ID を生成するアプリケーション透過的な方法を提供します。

PGD グローバルシーケンスは、アプリケーションがデータベースを使用して、ほとんどの場合に機能する非同期分散システムで一意の合成キーを生成する簡単な方法を提供しますが、必ずしもすべての場合ではありません。

PGD グローバルシーケンスを使用すると、挿入の競合に関する問題を回避できます。グローバルシーケンスを使用している列に PRIMARY KEY または UNIQUE 制約を定義すると、ノードは他のノードと同じ値を取得できません。PGD がノード間で挿入を同期すると、競合することはできません。

PGD グローバルシーケンスは PostgreSQL シーケンスを拡張するため、クラッシュセーフです。これらを使用するには、bdr_application ロールを付与される必要があります。

グローバルシーケンスにはさまざまなアルゴリズムが考えられます。

- SnowflakeId シーケンス
- グローバルに割り当てられたレンジシーケンス

SnowflakeId シーケンスは、どの時点でもノード間通信を必要としないアルゴリズムを使用して値を生成します。より速く、より堅牢で、値が作成されたときにタイムスタンプを記録する便利な特性があります。

SnowflakeId シーケンスには、64 ビット BIGINT データ型でのみ動作し、最大 19 桁の長さの値を生成するという制限があります。これは、JavaScript Number タイプなどの一部のホスト言語データ型で使用するには長すぎる場合があります。グローバルに割り当てられたシーケンスは、ノード間コンセンサスによって必要に応じて補充できるローカルな値の範囲を割り当て、BIGINT または INTEGER シーケンスに適しています。

*bdr.alter_sequence_set_kind()* を使用してグローバルシーケンスを作成できます

ファンクション。このファンクションは、標準の PostgreSQL シーケンスを取得し、PGD グローバルシーケンスとしてマークします。シーケンスを標準の PostgreSQL シーケンスに変換して戻すこともできます。

PGD は、構成変数 bdr.default_sequence_kind も提供します。この変数は、CREATE SEQUENCE コマンドが実行されたとき、または serial 、bigserial 、または GENERATED BY DEFAULT AS IDENTITY 列が作成されたときに作成するシーケンスの種類を決定します。有効な設定は次のとおりです。

- local —新しく作成されたシーケンスは、標準の PostgreSQL ローカルシーケンスです。
- galloc —常にグローバルに割り当てられたレンジシーケンスを作成します。

- `snowflakeid` —時間、ノード ID、およびカウンターコンポーネントで構成される BIGINT シーケンスのグローバルシーケンスを作成します。INTEGER シーケンスでは使用できませんので、 `bigserial` には使用できますが、 `serial` には使用できません。
- `timeshard` —SnowflakeId シーケンスの古いバージョン。下位互換性のためのみに提供されています。SnowflakeId が優先されます。
- `distributed` デフォルト bdr.default_sequence_kind にのみ使用できる特別な値。`int8` シーケンスつまり `bigserial` の場合は `snowflakeid` を選択し、`int4` つまり `serial` および `int2` シーケンスの場合は `galloc` シーケンスを選択します。

bdr.sequences ビューには、個々のシーケンス種類に関する情報が表示されます。

`currval()` および `lastval()` ファンクションは、すべてのタイプのグローバルシーケンスで正しく動作します。

### 自動シーケンス変換

PGD 6.0 以降では、ノードを PGD グループに参加するか、新しいグループを作成する行為も、ローカルシーケンスのグローバルシーケンスへの変換をトリガーします。bdr.default_sequence_kind を `distributed` に設定します。この設定は、ローカルシーケンスの変換に最適な種類のシーケンスを選択します。`bdr.default_sequence_kind` が `local` に設定されている場合、シーケンスはローカルシーケンスとして残されます。`galloc` への変換は、シーケンスがグループ内の他のシーケンスと競合しないことを保証する方法で実行されます。

ローカルシーケンスで開始し、後で galloc シーケンスに切り替える場合は、`bdr.default_sequence_kind` を `galloc` に設定し、変換する各シーケンスで `bdr.alter_sequence_set_kind()` ファンクションを実行します。ただし、シーケンスの開始値を手動で設定して、テーブル内の既存の値と競合しないようにする必要があることに注意してください。これ全般、特に *How to set a new start value for a sequence* の詳細については、Converting a local sequence to a galloc sequence を参照してください。

### SnowflakeId シーケンス

SnowflakeId シーケンスによって生成された ID は、緩やかな時間順序であるため、これらを使用して、標準の PostgreSQL シーケンスのように、データ挿入のおおよその順序を取得できます。1 つのノードでも、同じミリ秒以内に生成された値は順序が崩れる場合があります。緩い時間順序付けの特性は、レンジパーティションキーとしての使用に適していることを意味します。

SnowflakeId シーケンスは 1 つ以上のノードで動作し、ノード参加プロセスが完了した後にノード間通信を必要としません。したがって、拡張ネットワーク分割のリスクがある場合でも、それらを使用し続けることができます。これらは、レプリケーション遅延やノード間レイテンシーの影響を受けません。

SnowflakeId シーケンスは、標準のシーケンスとは異なる方法で一意の ID を生成します。このアルゴリズムは、シーケンス番号に 3 つのコンポーネントを使用します。シーケンスの最初のコンポーネントは、シーケンス番号生成時のタイムスタンプです。シーケンス番号の 2 番目のコンポーネントは、各 PGD ノードに割り当てられる一意の ID であり、これにより、異なるノードからの ID が常に異なることが保証されます。3 番目の成分は、ローカルシーケンスによって生成された数です。

競合がないことを保証には一意のノード ID をシーケンス番号に追加するだけで十分ですが、シーケンスの別の便利なプロパティも保持したいと思います。シーケンス番号の順序は、データがテーブルに挿入された順序にほぼ対応します。最初にタイムスタンプを配置することにより、これが保証されます。

SnowflakeId シーケンスには、いくつかの制限と注意事項が適用されます。

SnowflakeId シーケンスは 64 ビット幅で、`bigint` または `bigserial` が必要です。生成される値は最大 19 桁の長さです。実用的な 32 ビット `integer` バージョンがないため、 `serial` シーケンスでは使用できません。代わりに、グローバルに割り当てられたレンジシーケンスを使用します。

SnowflakeId の場合、特定のノードでミリ秒ごとに生成されるシーケンス値 4096 の制限があります 1 秒あたり約 400 万シーケンス値。シーケンス値の生成が特定のミリ秒内でラップアラウンドする場合、SnowflakeId シーケンスは次のミリ秒まで待機し、そのミリ秒の新しい値を取得します。

SnowflakeId シーケンスはタイムスタンプをシーケンス値にエンコードするため、指定された時間フレーム内でのみ新しいシーケンス値を生成できますシステムクロックによって異なります。使用できる最も古いタイムスタンプは 2016-10-07 で、これは SnowflakeId のエポック時間です。値は 2086 年に負の値にラップし、2156 年までに数値が完全になくなります。

タイムスタンプは SnowflakeId シーケンスの重要な部分であるため、Postgres プロセスの有効期間中に使用される最新のタイムスタンプより古いシーケンスを生成しないための追加の保護がありますただし、Postgres の再起動の間ではありません。

SnowflakeId シーケンスの入力として使用されるシーケンスの `INCREMENT` オプションは、事実上無視されます。これは、多くのオブジェクトリレーショナルマッパー ORM ツール、特に Hibernate と同様に、シーケンス ID キャッシュを行うアプリケーションに関連する可能性があります。シーケンスは時間ベースであるため、アプリケーションがキャッシュされた値で何かをできるようになるまでにシーケンスは新しい非衝突値に進むため、実際的な効果はほとんどありません。

同様に、基になるシーケンスの `START` 、`MINVALUE` 、`MAXVALUE` 、および `CACHE` 設定を変更することもできますが、変更するメリットはありません。シーケンスの下位 14 ビットが使用され、残りは破棄されるため、値範囲制限はファンクションの結果に影響を与えません。同じ理由で、`setval()` は SnowflakeId シーケンスには役立ちません。

### タイムシャードシーケンス

タイムシャードシーケンスは、既存のインストールとの下位互換性のために提供されていますが、新しいアプリケーションの使用はお勧めしません。代わりに SnowflakeId シーケンスを使用することをお勧めします。

Timeshard は SnowflakeId に非常に似ていますが、制限が異なり、保護が少なく、パフォーマンスが低下します。

Timeshard と SnowflakeId の違いは次のとおりです。

- Timeshard は、ミリ秒ごとに最大 16384 秒ごとに約 1600 万を生成できます。これは、SnowflakeId を超えています。ただし、特定のミリ秒内のラップアラウンドに対する保護はありません。Timeshard シーケンスを使用するスキーマは、特定の列に timeshard 値を使用する場合、`UNIQUE` 制約の使用を保護する必要があります。
- Timeshard シーケンスのタイムスタンプコンポーネントは、2050 年に値がなくなり、bigint と組み合わせて使用すると、値は 2033 年に負の数にラップします。これは、2033 年以降に生成されたシーケンスが負の値を持つことを意味します。これは SnowflakeId よりもかなり短い時間であり、これが SnowflakeId が優先される主な理由です。
- Timeshard シーケンスは、時折のディスク書き込みが必要です標準のローカルシーケンスと同様に。

SnowflakeId はメモリで計算されるため、SnowflakeId シーケンスは、一般に timeshard シーケンスよりも少し高速です。

### ログに記録されていないシーケンスと PGD

Postgres 15 以降、ログに記録されないシーケンスを作成できるようになりました。これらは関連しており、WAL に書き込まれず、レプリケートされないログ非記録テーブルに似ています。PGD とログなしシーケンスのコンテキストでは、ログなしの PGD シーケンスを持つことは合理的な構成ではなく、ノードに障害が発生した場合に予期しない問題を引き起こす可能性があります。したがって、ログのない PGD 配列の作成、または PGD 配列のログのない配列への変換を防止します。

### グローバルに割り当てられたレンジシーケンス

グローバルに割り当てられたレンジまたは `galloc` シーケンスは、値のレンジチャンクを各ノードに割り当てます。ローカルレンジが使い果たされると、他のノード間のコンセンサスによって、新しいレンジがグローバルに割り当てられます。この動作はキー空間を効率的に使用しますが、現在割り当てられているローカルレンジが使い果たされたときにシーケンスジェネレーターが進行するには、ローカルノードがクラスター内のノードの大部分に接続している必要があります。

SnowflakeId シーケンスとは異なり、`galloc` シーケンスは、PostgreSQL が提供するすべてのシーケンスデータ型 `smallint` 、`integer` 、および `bigint` をサポートします。これは、64 ビットシーケンスに問題がある環境で `galloc` シーケンスを使用できることを意味します。例には、53 ビット値のみをサポートするため、またはシーケンスが制限されたスペースで出力に表示される場合、JavaScript での整数の使用が含まれます。

各投票ノードによって割り当てられる範囲は、シーケンスが使用しているデータ型に基づいて、現在事前に決定されています。

- smallint — 1 000 数字
- 整数 — 1 000 000 番号
- bigint — 1 000 000 000 数字

各ノードは、seq_chunk_size の 2 つのチャンクを割り当てます。1 つは現在の使用に、1 つは将来の使用のために予約されたチャンクです。したがって、1 つのノードから生成される値は単調増加します。ただし、グローバルに見ると、生成された値はまったく順序付けられていません。これは、b ツリーインデックスへの影響によりパフォーマンスの低下を引き起こす可能性があり、通常は、生成された値がレンジパーティションキーとして役に立たないことを意味します。

`galloc` シーケンスの主な欠点は、割り当てられたレンジが使い果たされると、シーケンスジェネレーターがノード間通信を必要とするローカルノードの次のレンジに関するコンセンサスを求めなければならないことです。これにより、PGD グループの大部分がアクセスできない場合、遅延または操作の問題が発生する可能性があります。(これは、後のリリースで回避される可能性があります。)

`CACHE` 、`START` 、`MINVALUE` 、および `MAXVALUE` オプションは、`galloc` シーケンスで正常に動作します。ただし、シーケンスを `galloc` 種類に変換する前に、それらを設定する必要があります。 `INCREMENT BY` オプションも正常に動作します。ただし、各シーケンスデータ型に割り当てられた上記の範囲以上の増分値を割り当てることはできません。`setval()` は、`galloc` シーケンスのグローバル状態をリセットしません。使用しないでください。

galloc シーケンスには、いくつかの制限が適用されます。PGD は、特別な PGD カタログ bdr.sequence_alloc の galloc シーケンスを追跡します。このカタログは、galloc シーケンスに現在割り当てられているチャンクを追跡するために必要です。シーケンス名と名前空間はこのカタログに保存されます。シーケンスチャンクの割り当ては Raft によって管理されますが、シーケンス名前/名前空間の変更はレプリケーションストリームによって管理されます。したがって、PGD は現在 galloc シーケンスの名前の変更、別の名前空間への移動、または galloc シーケンスを含む名前空間の名前の変更をサポートしていません。アプリケーションスキーマをデザインするときは、この制限に注意してください。

### ローカルシーケンスの galloc シーケンスへの変換

ローカルシーケンスを galloc に変換する前に、いくつかの前提条件に注意する必要があります。

#### 1. シーケンスと列のデータ型が一致することを確認します

シーケンスのデータ型が、使用される列のデータ型と一致することを確認します。たとえば、bigint シーケンスを作成し、 integer 列のデフォルトをそのシーケンスによって返される nextval() に割り当てることができます。 bigint の場合 1 000 000 000 のブロックで割り当てられる galloc シーケンスでは、3 つ以上のノードが使用されている場合、nextval() によって返される値がすぐに int4 範囲を超えます。

この例は、何が起こり得るかを示しています。

```
CREATE SEQUENCE int8_seq;

SELECT sequencename, data_type FROM pg_sequences;
 sequencename | data_type
- -------------+-----------
 int8_seq     | bigint
(1 row)

CREATE TABLE seqtest (id INT NOT NULL PRIMARY KEY);

ALTER SEQUENCE int8_seq OWNED BY seqtest.id;

SELECT bdr.alter_sequence_set_kind(public.int8_seq::regclass, galloc, 1);
 alter_sequence_set_kind
- ------------------------

(1 row)

ALTER TABLE seqtest ALTER COLUMN id SET DEFAULT nextval(int8_seq::regclass);
```

2 つのノードで INSERT INTO seqtest VALUES(DEFAULT) を実行した後、テーブルには次の値が含まれます。

```
SELECT * FROM seqtest;
     id
- -----------
          2
```

(次のページに続く)

(前のページからの続き)

```
 2000000002
(2 rows)
```

ただし、3 番目のノードで同じ操作を試行すると、シーケンスが値 4000000002 を生成したため、integer out of range エラーで失敗します。

> **Tip**
>
> PostgreSQL からシーケンスの現在のデータ型を取得できます *pg_sequences* ビュー。現在の値が新しいデータ型の最大値を超えない限り、ALTER SEQUENCE ... AS ... たとえば ALTER SEQUENCE public.sequence AS integer を使用してシーケンスのデータ型を変更できます。

### 2. シーケンスの新しい開始値を設定します

シーケンス種類が galloc に変更されると、書き換えられ、ローカルシーケンスの定義された開始値からリスタートします。これが実稼働データベースの既存のシーケンスで発生した場合、現在の値を照会し、開始値を適切に設定する必要があります。このユースケースを支援するために、PGD ではファンクション *bdr.alter_sequence_set_kind()* で開始値を渡すことができます。すでにオフセットを使用しており、複数のノードからの書き込みがある場合は、最大使用値を確認し、少なくとも次の値までシーケンスをリスタートする必要があります。

```
- - 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);
```

ユーザーはシーケンスをロックできないため、 $MARGIN 値を残して、max() 値が照会されている間操作を続行できるようにする必要があります。

bdr.sequence_alloc テーブルは、チャンクサイズと、クラスター全体に割り当てられたレンジに関する情報を提供します。

この例では、シーケンスは 333 で始まり、クラスターには 2 つのノードがあります。割り当て数は 4 で、ノードごとに 2 であり、チャンクサイズは 1000000 で、整数シーケンスに関連しています。

```
SELECT * FROM bdr.sequence_alloc
    WHERE seqid = public.categories_category_seq::regclass;
          seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |      ␣
↪seq_last_alloc
- ----------------------+----------------+---------------------+-------------+------
```

(次のページに続く)

(前のページからの続き)

```
↪-----------------------
 categories_category_seq |        1000000 |             4000333 |           4 | 2020-
↪05-21 20:02:15.957835+00
(1 row)
```

各ノードの特定のシーケンスに現在割り当てられているレンジを表示するには、ファンクション bdr.galloc_chunk_info を実行します。

- ノード Node1 は、 333 〜2000333 のレンジを使用しています。

```
SELECT * FROM bdr.galloc_chunk_info(categories_category_seq);
 chunk_start | chunk_end
- ------------+-----------
         334 |   1000333
     1000334 |   2000333
(2 rows)
```

- ノード Node2 は、 2000334 〜4000333 のレンジを使用しています。

```
SELECT * FROM bdr.galloc_chunk_info(categories_category_seq);
 chunk_start | chunk_end
- ------------+-----------
     2000334 |   3000333
     3000334 |   4000333
```

ノードはチャンクを終了すると、新しいチャンクのコンセンサスを求め、最初に利用可能なものを取得します。この例では、4000334〜5000333 です。これは新しい予約チャンクであり、古い予約チャンクの消費を開始します。

### UUID、KSUUID、およびその他のアプローチ

PGD で使用できるグローバルシーケンスを使用せずに、他の方法でグローバルに一意の ID を生成できます。例

- UUID とその PGD バリアント KSUUID
- ノードごとに異なるオフセットを持つローカルシーケンスつまり手動
- 外部調整されたナチュラルキー

PGD アプリケーションは、他の方法を安全に使用できません。PGD はノード間で行ロックを取得しないため、シーケンス生成に SELECT ... FOR UPDATE 、UPDATE ... RETURNING ... または同様のものに依存するカウンターテーブルベースのアプローチは、PGD では正しく動作しません。同じ値が複数のノードで生成されます。同じ理由で、「ギャップレス」シーケンス生成の通常の戦略は PGD では機能しません。ほとんどの場合、アプリケーションは、2 フェーズコミットを使用して外部ソースからギャップレスである必要があるシーケンスの生成を調整します。または、PGD グループ内の 1 つのノードでのみ生成されます。

### KSUUID v2 ファンクション

PGD アプリケーションは、他の方法を安全に使用できません。PGD はノード間で行ロックを取得しないため、シーケンス生成に `SELECT ... FOR UPDATE`、`UPDATE ... RETURNING ...` または同様のものに依存するカウンターテーブルベースのアプローチは、PGD では正しく動作しません。同じ値が複数のノードで生成されます。同じ理由で、「ギャップレス」シーケンス生成の通常の戦略は PGD では機能しません。ほとんどの場合、アプリケーションは、2 フェーズコミットを使用して外部ソースからギャップレスである必要があるシーケンスの生成を調整します。または、PGD グループ内の 1 つのノードでのみ生成されます。

### UUID

`UUID` キーは、代わりにシーケンスを完全に回避し、128 ビットのユニバーサル一意識別子を使用します。これらはランダムまたは擬似乱数値であり、非常に大きいため、同じ値を 2 回生成することはほぼ不可能です。`UUID` キーを使用する場合、ノードが継続的に通信する必要はありません。

万が一、衝突が発生した場合、競合検出は、挿入された 2 つのレコードの新しい方を選択して保持します。競合ログが有効になっている場合、このようなイベントが記録されます。ただし、約 2^64 キーが生成された後にのみ衝突が実際に可能になるため、発生する可能性は非常に低いです。

`UUID` キーの主な欠点は、領域とネットワークの点でやや非効率であることです。これらは、主キーとしてだけでなく、外部キーで参照される場合、および回線上で送信される場合にも、より多くの領域を消費します。また、すべてのアプリケーションが `UUID` キーにうまく対応できるわけではありません。

### KSUUID

PGD は、`UUID` データの K ソート可能なバリアントを操作するためのファンクションを提供します。KSUUID として知られ、PostgreSQL 標準の `UUID` データ型を使用して保存できる値を生成します。 `KSUUID` 値は、`UUID` 標準に従って、タイムスタンプとランダムデータの両方を保存するという点で `UUIDv1` に似ています。違いは、 `KSUUID` が K ソート可能であること、つまりタイムスタンプで弱くソート可能であることを意味します。これにより、よりコンパクトな `btree` インデックスが生成されるため、データベースキーとしてより役立ちます。この動作により、検索の効率が向上し、結果データの自然な時間ソートが可能になります。`UUIDv1` とは異なり、`KSUUID` 値には、生成されたコンピューターの MAC が含まれないため、それらの使用にセキュリティ上の懸念はありません。

あらゆる場合に `KSUUID` v2 を推奨するようになりました。通常の比較演算子で生成された値を直接並べ替えることができます。

PGD には `KSUUID` の 2 つのバージョン v1 および v2 があります。古い `KSUUID` v1 は非推奨ですが、既存のインストールをサポートするために維持されます。新しいインストールには使用しないでください。v1 と v2 の内部コンテンツには互換性がありません。そのため、それらを操作する機能も互換性がありません。`KSUUID` の v2 も、`UUID` バージョン番号を保存しなくなりました。

PGD リファレンスの KSUUID v2 functions および KSUUID v1 functions を参照してください。

### ステップシーケンスとオフセットシーケンス

オフセットステップシーケンスでは、各ノードで通常の PostgreSQL シーケンスが使用されます。各シーケンスは同じ量だけインクリメントし、異なるオフセットで開始します。たとえば、ステップ 1000 では、ノード 1 のシーケンスは 1001、2001、3001 などを生成します。node2 のシーケンスは、1002、2002、3002 などを生成します。このスキームは、ノードが長期間通信できない場合でもうまく機能します。ただし、

デザイナーはスキーマを確立するときにノードの最大数を指定する必要があり、ノードごとの構成が必要です。間違いにより、シーケンスのオーバーラップが簡単に発生する可能性があります。

次のように 1 つのノードで目的のシーケンスを作成することにより、PGD でこのアプローチを構成するのは比較的簡単です。

```
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);
```

次に、 `setval()` を呼び出す各ノードで、各ノードに異なるオフセット開始値を与えます。例

```
- - On node 1
SELECT setval(some_seq, 1);

- - On node 2
SELECT setval(some_seq, 2);

 -- ... etc
```

将来的に変更することは困難であり、中断を伴うため、追加するすべてのノードのための領域を残すように十分な大きさの `INCREMENT` を許可してください。

`bigint` 値を使用する場合、10000 以上のオフセットを使用する場合でも、キーの枯渇に関する実質的な懸念はありません。枯渇に近づく可能性があるまでには、数百のマシンが毎秒数百万の挿入を実行して、数百年かかります。

PGD は、現在、このようなステップ/オフセットシーケンスでノードごとのオフセットを構成するための自動化を提供していません。

### 複合キー

ステップ/オフセットシーケンスのバリアントは、`PRIMARY KEY (node_number, generated_value)` で構成される複合キーを使用することです。ノード番号は通常、各ノードで異なる番号を返すファンクションから取得されます。DDL レプリケーションを一時的に無効にし、定数 SQL ファンクションを作成することにより、このようなファンクションを作成できます。または、レプリケーションセットの一部ではない 1 行テーブルを使用して、各ノードに異なる値を保存できます。

### 関連項目

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

### 6.3.9 Stream triggers

PGD は、ダウンストリーム/ターゲットノードでの追加データ処理に使用できる新しいタイプのトリガーを導入しました。

- 競合トリガー
- トランスフォームトリガー

これらのタイプのトリガーは、まとめて **ストリームトリガー** と呼ばれます。

ストリームトリガーは、構文がトリガーのようなものになるように設計されています。これらは PostgreSQL BEFORE トリガーアーキテクチャを活用し、PostgreSQL BEFORE トリガーと同様のパフォーマンス特性を持つ可能性があります。

通常の PostgreSQL トリガーと同様に、複数のトリガー定義で 1 つのトリガーファンクションを使用できます。トリガーファンクションは、`CREATE FUNCTION ... RETURNS TRIGGER` 形式で定義されたプログラムです。トリガーの作成に、`CREATE TRIGGER` コマンドを使用する必要はありません。代わりに、特別な PGD ファンクション *bdr.create_conflict_trigger()* および *bdr.create_transform_trigger()* を使用してストリームトリガーを作成します。

トリガーは、作成されると、カタログテーブル `pg_trigger` に表示されます。ストリームトリガーは `tgisinternal = true` および `tgenabled = 'D'` としてマークされ、名前サフィックス'_bdrc'または'_bdrt'が付いています。`bdr.triggers` ビューは、テーブルに関連するトリガー、実行されているプロシージャーの名前、それをトリガーするイベント、およびトリガータイプに関する情報を提供します。

ストリームトリガーは、通常の SQL 処理では有効になっていません。このため、`ALTER TABLE ... ENABLE TRIGGER` は、特定の名前バリアントと ALL バリアントの両方でストリームトリガーに対してブロックされます。このメカニズムにより、トリガーは通常の SQL トリガーとして実行されません。

これらのトリガーは、ダウンストリームまたはターゲットノードで実行されます。元のノードで実行するオプションはありません。ただし、オリジンで `row_filter` 式の使用を検討することができます。

また、ストリームトリガーの実行中に適用される DML は、他の PGD ノードにレプリケートされず、標準のローカルトリガーの実行をトリガーしません。これは意図的なものです。たとえば、これを使用して、ストリームトリガーによってキャプチャされた変更または競合を、クラッシュセーフでそのノードに固有のテーブルにログ記録できます。実用的な例については、**ストリームトリガーの例** を参照してください。

#### 適用中のトリガー実行

変換トリガーは、トリガーテーブルに着信する変更ごとに 1 回実行されます。これらのトリガーは、一致するターゲット行を見つけようとする前に起動するため、非常に幅広い変換を効率的かつ一貫して適用できます。

次に、UPDATE および DELETE 変更の場合、ターゲット行を見つけます。ターゲット行がない場合、これらの変更タイプに対してそれ以上の処理は発生しません。

次に、テーブルレベルでレプリカトリガーとして明示的に有効にされていたノーマルトリガーを実行します。

```
ALTER TABLE tablename
ENABLE REPLICA TRIGGER trigger_name;
```

次に、潜在的な競合が存在するかどうかを判断します。その場合、そのテーブルに存在する競合トリガーを呼び出します。

### 欠落列の競合の解決

変換トリガーが実行される前に、PostgreSQL は着信タプルをターゲットテーブルの行タイプと照合しようとします。

入力行には存在するがターゲットテーブルには存在しない列は、target_column_missing タイプの競合をトリガーします。逆に、ターゲットテーブルには存在するが、入力行にない列は、source_column_missing 競合をトリガーします。これらの 2 つの競合タイプのデフォルトの解決策は、それぞれ ignore_if_null および use_default_value です。

これは、ローリングスキーマのアップグレードのコンテキストで関連しますたとえば、スキーマの新しいバージョンが新しい列を導入する場合。古いバージョンのスキーマから新しいバージョンにレプリケートする場合、ソース列が欠落しており、新しく導入された列にデフォルト値を移入する use_default_value ストラテジーが適切です。

ただし、新しいスキーマバージョンを持つノードから古いスキーマバージョンを持つノードに複製する場合、ターゲットテーブルから列が失われます。ignore_if_null リゾルバーは、ユーザーがアップグレードされたノードの新しい列に NULL 以外の値を含むタプルを挿入するとすぐにレプリケーションを中断するため、ローリングアップグレードには適していません。

この例を考慮すると、ローリングスキーマアップグレードの適切な設定は、target_column_missing 競合が発生した場合に ignore リゾルバーを適用するように各ノードを構成することです。

これは、次のクエリーで行うことができます。これは、各ノードで個別に実行する必要があります。node1 を実際のノード名に置き換えます。

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

### データ損失と発散リスク

競合リゾルバーを ignore に設定すると、データの損失とクラスターの発散が発生する可能性があります。

次の例を考えます。テーブル t はノード 1 および 2 に存在しますが、その列 col はノード 1 にのみ存在します。

競合リゾルバーが ignore に設定されている場合、ノード 1 に c が null でない行、たとえば (pk=1, col=100) が存在する場合があります。その行はノード 2 にレプリケートされ、列 c の値は破棄されますたとえば、(pk=1) 。

列 c がノード 2 のテーブルに追加される場合、最初に既存のすべての行で NULL に設定され、上記で検討される行は (pk=1, col=NULL) になります。 pk=1 を持つ行はすべてのノードで同一ではなくなり、したがって、クラスターは発散します。

ノード 2 にレプリケートされた行には col=NULL があるため、デフォルトの ignore_if_null リゾルバーはこのリスクの影響を受けません。

この例に基づいて、 ignore リゾルバーが使用されたローリングスキーマアップグレードの最後に、クラスター全体に対して LiveCompare を実行することをお勧めします。この実践は、発散を確実に検出および

修正するのに役立ちます。

### 行型の用語

PGD は次の行タイプを使用します。

- SOURCE_OLD は更新前の行、つまりキーです。
- SOURCE_NEW は、別のノードからの新しい行です。
- TARGET は、ノードに既に存在する行、つまり競合する行です。

### 競合トリガー

競合トリガーは、PGD によって競合が検出されると実行されます。彼らは、紛争が発生したときに何が起こるかを決定します。

- トリガーファンクションが行を返した場合、アクションはターゲットに適用されます。
- トリガーファンクションが NULL 行を返した場合、アクションはスキップされます。

たとえば、DELETE に対してトリガーが呼び出された場合、DELETE をスキップする場合、トリガーは NULL を返します。 DELETE を続行する場合は、行値を返します。SOURCE_OLD または TARGET のいずれかが機能します。競合するオペレーションが INSERT または UPDATE で、選択された解決が競合する行を削除することである場合、トリガーは明示的に削除を実行し、NULL を返す必要があります。トリガーファンクションは、選択した他の SQL アクションを実行できますが、これらのアクションはローカルにのみ適用され、レプリケートされません。

2 つ以上のノード間で実際のデータの競合が発生すると、2 つ以上の変更が同時に発生します。変更が適用されると、競合の解決は各ノードで独立して発生します。これは、競合の解決が各ノードで 1 回発生し、それらの間で大幅な時間差で発生する可能性があることを意味します。その結果、競合トリガーの複数の実行間の通信は不可能です。トリガーが関連するすべてのイベントに対してまったく同じ結果を与えることを確認するのは、競合トリガーの作成者の責任です。そうしないと、データの相違が発生します。

> **警告**
>
> - 単一のテーブルに複数の競合トリガーを指定できますが、それらは個別のイベントと一致する必要があります。つまり、各競合は、単一の競合トリガーのみと一致する必要があります。 - 同じテーブルの同じイベントに一致する複数のトリガーはお勧めしません。これらは一貫性のない動作を引き起こす可能性があり、将来のリリースでは許可されません。

同じ競合トリガーが複数のイベントと一致する場合、トリガーで TG_OP 変数を使用して、競合を生成した操作を特定できます。

デフォルトでは、PGD は行のレプリケーションオリジンの変更を監視することにより競合を検出します。したがって、1 つの変更のみが発生している場合でも、競合トリガーを呼び出すことができます。この場合、実際の競合は存在しないため、この競合検出メカニズムは偽陽性の競合を生成する可能性があります。競合トリガーは、これらをすべて同様に処理する必要があります。

場合によっては、タイムスタンプ競合検出は競合をまったく検出しない場合があります。たとえば、UPDATE の直後に DELETE が発生するコンカレント UPDATE /DELETE では、最初に UPDATE を、次に DELETE を参照

するノードは競合を認識しません。競合が発生しない場合、競合トリガーは呼び出されません。同じ状況で、行バージョンの競合検出を使用すると、競合が表示され、競合トリガーで処理できます。

トリガーファンクションは、操作タイプに応じて、追加の状態情報と競合に関係するデータ行にアクセスします。

- INSERT では、競合トリガーはソースから SOURCE_NEW 行と TARGET 行にアクセスできます。
- UPDATE では、競合トリガーはソースから SOURCE_OLD および SOURCE_NEW 行と TARGET 行にアクセスできます。
- DELETE では、競合トリガーはソースから SOURCE_OLD 行と TARGET 行にアクセスできます。

そのオペレーションに値が存在する場合、ファンクション bdr.trigger_get_row() を使用して、SOURCE_OLD 、SOURCE_NEW 、または TARGET 行を取得できます。

競合トリガーの変更はトランザクション的に発生し、構成変更のレプリケーション中にグローバル DML ロックによって保護されます。この動作は、ALTER TABLE の一部のバリアントの処理方法に似ています。

競合トリガー内で主キーが更新されると、実行タイミングの違いにより一意制約違反エラーが発生する場合があります。したがって、競合トリガーでの主キーの更新は避けてください。

### Transform トリガー

これらのトリガーは、特定のテーブルに対するデータストリームのすべての行に対して実行される点を除き、競合トリガーに似ています。戻り値と公開変数の動作は似ていますが、変換トリガーはターゲット行が識別される前に実行されるため、TARGET 行はありません。

PGD の各テーブルで複数の変換トリガーを指定できます。変換トリガーはアルファベット順に実行されます。

変換トリガーは行をフィルターして取り除くことができ、必要に応じて追加の操作を実行できます。列の値を変更したり、NULL に設定したりできます。戻り値によって、実行される次のアクションが決まります。

- トリガーファンクションが行を返した場合、ターゲットに適用されます。
- トリガーファンクションが NULL 行を返した場合、それ以上に実行するアクションはありません。未実行のトリガーは実行されません。
- トリガー機能は、自分の選択に応じて他のアクションを実行できます。

トリガーファンクションは、追加の状態情報と、競合に関係する行にアクセスします。

- INSERT では、変換トリガーはソースから SOURCE_NEW 行にアクセスできます。
- UPDATE では、変換トリガーはソースから SOURCE_OLD および SOURCE_NEW 行にアクセスできます。
- DELETE では、変換トリガーはソースから SOURCE_OLD 行にアクセスできます。

ファンクション bdr.trigger_get_row() を使用して、SOURCE_OLD または SOURCE_NEW 行を取得できます。このタイプのトリガーは、このようなターゲット行が存在する場合が識別される前に実行されるため、TARGET 行は使用できません。

変換トリガーは通常の BEFORE 行トリガーに非常に似ていますが、次の重要な違いがあります。

- 変換トリガーは、入ってくる変更ごとに呼び出されます。 UPDATE および DELETE では、BEFORE トリガーはまったく呼び出されず、テーブル内で一致する行が見つからない場合、DELETE が変更されます。
- 変換トリガーは、パーティションテーブルルーティングが発生する前に呼び出されます。
- 変換トリガーは、SOURCE_OLD を介してルックアップキーにアクセスできます。これは、通常の SQL トリガーでは使用できません。

#### 行の内容

SOURCE_NEW 、SOURCE_OLD 、および TARGET の内容は、オペレーション、テーブルの REPLICA IDENTITY 設定、およびターゲットテーブルの内容によって異なります。

TARGET 行は、競合トリガーでのみ使用できます。TARGET 行には、ターゲットテーブルで UPDATE または DELETE を適用するときに行が見つかった場合にのみデータが含まれます。行が見つからない場合、TARGET は NULL です。

#### 実行オーダー

トリガーの実行順序

- Transform トリガー —ターゲットへの入力行ごとに 1 回実行します。
- 通常トリガー —行ごとに 1 回実行します。
- 競合トリガー —競合が存在する行ごとに 1 回実行します。

#### ストリームトリガーの例

update_if_newer 競合リゾルバーと同様の動作を提供する競合トリガー

```
CREATE OR REPLACE FUNCTION update_if_newer_trig_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF (bdr.trigger_get_committs(TARGET) >
        bdr.trigger_get_committs(SOURCE_NEW)) THEN
    RETURN TARGET;
    ELSIF
        RETURN SOURCE;
    END IF;
END;
$$;
```

カウンター列にデルタ変更を適用し、他のすべての列に SOURCE_NEW を使用する競合トリガー。

```
CREATE OR REPLACE FUNCTION delta_count_trg_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    DELTA bigint;
    SOURCE_OLD record;
    SOURCE_NEW record;
    TARGET record;
BEGIN
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    TARGET := bdr.trigger_get_row(TARGET);

    DELTA := SOURCE_NEW.counter - SOURCE_OLD.counter;
    SOURCE_NEW.counter = TARGET.counter + DELTA;

    RETURN SOURCE_NEW;
END;
$$;
```

すべての変更を適用する代わりにログテーブルにログを記録する変換トリガー

```
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;
```

(次のページに続く)

(前のページからの続き)

```
$$;
```

この例は、信頼できるサイト、優先ノード、または Always Wins 解決としても知られる、信頼できるソースの競合検出を実装する競合トリガーを示しています。bdr.trigger_get_origin_node_id() ファンクションを使用して、3 つ以上のノードで動作するソリューションを提供します。

```
CREATE OR REPLACE FUNCTION test_conflict_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE  record;
    TARGET  record;

    TRUSTED_NODE    bigint;
    SOURCE_NODE     bigint;
    TARGET_NODE     bigint;
BEGIN
    TARGET := bdr.trigger_get_row(TARGET);
    IF (TG_OP = DELETE)
        SOURCE := bdr.trigger_get_row(SOURCE_OLD);
    ELSE
        SOURCE := bdr.trigger_get_row(SOURCE_NEW);
    END IF;

    TRUSTED_NODE := current_setting(customer.trusted_node_id);

    SOURCE_NODE := bdr.trigger_get_origin_node_id(SOURCE_NEW);
    TARGET_NODE := bdr.trigger_get_origin_node_id(TARGET);

    IF (TRUSTED_NODE = SOURCE_NODE) THEN
        RETURN SOURCE;
    ELSIF (TRUSTED_NODE = TARGET_NODE) THEN
        RETURN TARGET;
    ELSE
        RETURN NULL; -- do not apply the change
    END IF;
END;
$$;
```

### 6.3.10 Transaction streaming

論理レプリケーションでは、トランザクションはパブリッシャーで同時にデコードされますが、トランザクションがコミットされるまでサブスクライバーに送信されません。変更が `logical_decoding_work_mem` PostgreSQL 13 以降を超える場合、それらはディスクにあふれます。これは、特に大規模なトランザクションの場合、サブスクライバーに到達するまでにある程度の遅延が発生し、パブリッシャーで追加の I/O を必要とする場合があることを意味します。

PostgreSQL 14 以降、パブリッシャーでコミットされる前に、オプションでトランザクションをデコードして、サブスクライバーに送信できます。サブスクライバは、着信した変更をステージングファイルまたはファイルセットに保存し、トランザクションがコミットするときに適用します、またはトランザクションがアボートする場合は破棄します。これにより、トランザクションがコミットされるとすぐに、サブスクライバーにトランザクションを適用できるようになります。

#### PGD の機能強化

PostgreSQL のビルトイントランザクションストリーミングには、次の制限があります。

- パブリッシャーのディスクに変更をスピルする必要はありませんが、各サブスクライバーのディスクに変更を書き込む必要があります。
- トランザクションが中止される場合、作業各サブスクライバが受け取った変更および関連するストレージ I/O は無駄になります。

ただし、PGD はパラレル適用をサポートし、各サブスクライバーでマルチプルのライタプロセスを有効にします。この機能を活用して、次の機能強化を提供します。

- デコードされたトランザクションは、サブスクライバーのライタに直接ストリーミングできます。
- デコードされたトランザクションは、サブスクライバーのディスクに保存する必要はありません。
- サブスクライバーへのトランザクションの適用を開始する前に、トランザクションがコミットされるのを待機する必要はありません。

#### 注意事項

- Parallel apply を有効にする必要があります。
- 多数の小規模で競合するトランザクションで構成されるワークロードは、ライター間で頻繁なデッドロックを発生させる可能性があります。

> **注釈**
>
> ライタへの直接ストリーミングはまだ実験的な機能です。注意して使用してください。具体的には、ストリーミングのコミットタイムスタンプを利用できない可能性があるため、競合解決ではうまく機能しない可能性があります。トランザクションはオリジンでまだコミットされていない可能性があります。

#### 構成

2 つの場所でトランザクションストリーミングを構成します。

- ノードレベルで、 GUC bdr.default_streaming_mode を使用
- グループレベルで、ファンクション bdr.alter_node_group_option() を使用

#### bdr.default_streaming_mode を使用したノード構成

許可される値は次のとおりです。

- `off`
- `writer`
- `file`
- `auto`

デフォルト値は `auto` です。

この設定の変更を有効にするには、サブスクリプションごとに pglogical レシーバープロセスを再起動します。

これは、サーバーを再起動することで実現できます。

`bdr.default_streaming_mode` が `off` 以外の値に設定されている場合、サブスクライバーはパブリッシャーにトランザクションストリーミングを要求します。これが提供される方法は、グループ構成設定によっても異なります。詳細は、*bdr.default_streaming_mode を使用したノード構成* を参照してください。

#### bdr.alter_node_group_option() を使用したグループ構成

ファンクションでパラメーター `streaming_mode` を使用できます bdr.alter_node_group_option()

グループトランザクションストリーミング構成を設定します。

許可される値は次のとおりです。

- `off`
- `writer`
- `file`
- `auto`
- `default`

デフォルト値は `default` です。

現在の設定の値は、ビュー bdr.node_group の列 `node_group_streaming_mode` に含まれています。返される値は単一の char 型で、使用可能な値は `D default` 、`W writer` 、 `F file` 、 `A auto` 、および `O off` です。

### 構成設定の効果

トランザクションストリーミングは、GUC bdr.default_streaming_mode によってサブスクライバーレベルで制御されます。トランザクションストリーミングを無効にする off に設定されていない限り、サブスクライバーはトランザクションストリーミングを要求します。

パブリッシャーがトランザクションストリーミングを提供できる場合、トランザクションサイズが logical_decoding_work_mem で設定されたしきい値を超えるたびに、トランザクションをストリーミングします。通常、パブリッシャーは、トランザクションがファイルにストリーミングされるか、ライタにストリーミングされるかを制御できません。COPY などの一部の状況を除き、サブスクライバーがトランザクションをライタにストリーミングすることを示唆する場合があります可能であれば。

サブスクライバーは、パブリッシャーから受け取ったトランザクションをライタまたはファイルにストリーミングできます。決定はいくつかの要素に基づいて行われます。

- Parallel apply がオフ num_writers = 1 の場合、ファイルにストリーミングされます。ライタ 0 は常に非ストリーミングトランザクション用に予約されています。
- Parallel apply がオンになっているが、すべてのライターがすでにビジー状態である場合、新しいトランザクションはファイルにストリーミングされます。PGD ライタのステータスを確認するには、*PGD* ライターのモニタリング を参照してください。

ライタへのストリーミングが可能である場合、つまりフリーライタが利用可能な場合、トランザクションをライタにストリーミングするかファイルにストリーミングするかは、次の表にあるグループとノード設定の組み合わせに基づいて決定されます。

| 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 |

グループ構成が auto に設定されている場合、またはグループ構成が default でノード構成が auto である場合、パブリッシャーがこれを行うことをヒントにした場合にのみ、トランザクションはライタにストリーミングされます。

現在、パブリッシャーは、サブスクライバーが次のトランザクションタイプのライタにストリーミングすることを示唆しています。これらは競合しないことが知られており、ライタが安全に処理できます。

- COPY
- CREATE INDEX CONCURRENTLY

#### モニタリング

bdr.stat_subscription を使用して、トランザクションストリーミングの使用をモニタできます。

サブスクライバノード上のファンクション。

- nstream_writer —ライタにストリーミングされたトランザクションの数。
- nstream_file —ファイルにストリーミングされたトランザクションの数。
- nstream_commit —コミットされたストリーミングトランザクションの数。
- nstream_abort —中止されたストリーミングトランザクションの数。
- nstream_start —開始されたストリーミングトランザクションの数。
- nstream_stop —完全に受信されたストリーミングトランザクションの数。

### 6.3.11 Explicit two-phase commit (2PC)

> **注釈**
>
> 2 フェーズコミットは、 Group Commit または CAMO では使用できません。*Commit scope limitations* を参照してください。

アプリケーションは、PGD で 2 フェーズコミットを使用することを明示的に選択できます。Distributed Transaction Processing: The XA Specification を参照してください。

X/Open 分散トランザクション処理 DTP モデルは、3 つのソフトウェアコンポーネントを想定しています。

- トランザクション境界を定義し、トランザクションを構成するアクションを指定するアプリケーションプログラム AP
- 共有リソースへのアクセスを提供するデータベースやファイルアクセスシステムなどのリソースマネージャー RM
- トランザクションに識別子を割り当て、その進行状況を監視し、トランザクションの完了と障害からの回復に対して責任を負うトランザクションマネージャー TM と呼ばれる別のコンポーネント

PGD は、 PREPARE TRANSACTION および COMMIT PREPARED /ROLLBACK PREPARED コマンドを使用して明示的な外部 2PC をサポートします。外部的には、 EDB Postgres 分散クラスターは、単一のセッションのトランザクションマネージャーに対する単一のリソースマネージャーであるように見えます。

bdr.commit_scope が local の場合、トランザクションはローカルノードでのみ準備されます。コミットされると、変更がレプリケートされ、PGD がコミット後の競合解決を適用します。

明示的な 2 フェーズコミットでは、 local に設定して bdr.commit_scope を使用するのは意味がないと思われる場合があります。ただし、このオプションは、トランザクションのレイテンシーと堅牢性の間のトレードオフを制御できるように提供されています。

明示的な 2 フェーズコミットは、CAMO またはグローバルコミットスコープでは機能しません。将来のリリースでは、この組み合わせが有効になる可能性があります。

#### 使用する

ローカルコミットスコープの 2 フェーズコミットは、標準の PostgreSQL とまったく同じように動作します。ローカルコミットスコープを使用します。

```
BEGIN;
SET LOCAL bdr.commit_scope = local;

... other commands possible...
```

コミットの最初のフェーズを開始するには、クライアントはグローバルトランザクション ID を割り当てる必要があります。これは、トランザクションを識別する一意の文字列です。

```
PREPARE TRANSACTION some-global-id;
```

最初のフェーズが成功した後、すべてのノードが変更を適用し、トランザクションをコミットする準備が整います。次に、クライアントは同じノードから第 2 フェーズを呼び出す必要があります。

```
COMMIT PREPARED some-global-id;
```

### 6.3.12 Node types and capabilities

PGD クラスターには、それぞれに独自のロールを持つ複数の異なるタイプのノードを含めることができます。このセクションでは、PGD クラスターで構成できるさまざまなタイプのノードについて説明します。

- Overview は、PGD クラスターに存在できるノードの種類と、それに関連するロールの概要です。
- Witness nodes は、グループでコンセンサスを確立する専用の PGD ノードの特別なクラスである監視ノードを調べます。
- Logical standby nodes は、スタンバイのノードを効率的に同期させ、障害が発生した場合にプライマリとしてステップインする準備をする方法を示します。
- Subscriber-only nodes and groups は、加入者専用ノードが加入者専用グループと連携する方法、読み取りスケーラビリティを向上させる方法、およびそれらを構成するためのさまざまなオプションを検討します。

#### An overview of PGD Node types

##### データノード

PGD のデータノードは、Postgres インスタンスを実行するノードです。他のすべてのデータノードにデータを複製します。また、ロックとリーダーシップを中心にクラスター全体の Raft の意思決定にも参加します。1 つ以上のグループのメンバーになることができ、デフォルトでは、クラスター内のすべてのデータノードにまたがる「トップレベル」グループのメンバーです。

データノードは、他の 3 つのノードが構築される基盤でもあります。

### Witness ノード

監視ノードは、ロックとリーダーシップを中心にクラスター全体の Raft の意思決定に参加するという点でデータノードのように動作します。ただし、データを複製または保存しません。監視ノードの目的は、クラスターがコンセンサスを求めるマジョリティを達成できるように利用できることです。*Witness ノード* に詳細があります。

### ロジカルスタンバイノード

論理スタンバイノードは、別のノードから論理データの変更を受信し、ローカルに複製するノードです。PGD は、ロジカルスタンバイノードを使用して、複製しているノードが利用できなくなった場合にそのノードを置き換えることができますが、いくつかの注意事項があります。詳細については、**ロジカルスタンバイノード** を参照してください。

### サブスクライバー専用ノード

サブスクライバー専用ノードは、名前が示すように、クラスター内の変更をサブスクライブするのみが、他のノードに変更をレプリケートしないデータノードです。サブスクライバ専用ノードは、アプリケーションの読み取り専用ノードとして使用できます。ノードを作成するときにデータノードが `subscriber-only` を指定し、サブスクライバー専用グループに追加することにより、サブスクライバー専用ノードを作成します。詳細は、*Subscriber-only nodes and groups* を参照してください。

### Witness nodes

監視ノードは、データノードとして機能する軽量ノードですが、データを保存または複製しません。監視ノードを使用して、Raft コンセンサスを使用する PGD クラスターが奇数の投票ノードを持つようにできるため、決定を行うときにマジョリティを達成できます。

### PGD グループまたはリージョン内の Witness ノード

監視ノードの一般的な使用の 1 つは、PGD グループに 2 つのデータノードがあるが、推奨される 3 つのデータノードでリソースが利用できない場合です。この場合、監視ノードを PGD グループに追加して、ローカルの Raft の意思決定に 3 番目の投票ノードを提供できます。これらの決定は、主に使用するプロキシの書き込みリーダーを誰が選択するかに関するものです。2 つのノードのみでは、どのデータノードが書き込みリーダーであるかについてコンセンサスがない可能性があります。2 つのデータノードと監視者がある場合、2 人の候補者データノードと 3 人の有権者データノードと監視者が存在します。データノードがダウンしている場合、書き込みリーダーを選択できる投票者はまだ 2 人います。

### リージョン外の Witness ノード

より高いレベルでは、複数の PGD グループが異なるリージョンにマッピングされている場合、監視ノードを使用できます。たとえば、2 つのリージョンのリージョンごとに 3 つのデータノードがある場合、通常に実行されている間、6 つのデータノードすべてが Raft 決定に参加し、DDL および DML グローバルロックを取得できます。データノードがダウンしている場合でも、コンセンサスを取得するための十分なデータノードが存在します。しかし、ネットワーク分割が発生し、他のリージョンとの接続が失われると、6 つのノードのうち 3 つのノードのみが利用できますが、これはコンセンサスには十分ではありません。このシナリオを回避するには、PGD クラスターの一部として 3 番目のリージョンに監視ノードを展開できます。この監視ノードにより、リージョンが利用できないときに PGD クラスターのほとんどの操作要件でコンセンサスを達成できます。

### Logical standby nodes

PGD では、オフロードノード、読み取り専用ノード、受信専用ノード、または論理リードレプリカとしても知られる**ロジカルスタンバイノード**を作成できます。マスターノードには、0、1 つ、またはそれ以上のロジカルスタンバイノードを持つことができます。

> **注釈**
>
> 論理スタンバイノードは、データセンター間のネットワークトラフィックが懸念される環境で使用できます。それ以外の場合、ロケーションごとにより多くのデータノードを持つことが常に優先されます。

論理スタンバイノードは、継続的リカバリの状態で保持されるノードであり、必要になるまで常に更新されます。この動作は、パフォーマンスを向上させるために論理レプリケーションを使用しながら、Postgres フィジカルスタンバイが動作する方法に似ています。論理スタンバイノードは変更を受信しますが、ローカルで行われた変更を他のノードに送信しません。

ロジカルスタンバイは、bdr.create_node でノードを作成するときに `node_kind` を `standby` として指定することにより作成されます。

後で、必要に応じて、 bdr.promote_node を使用します

ロジカルスタンバイを完全な通常の送信/受信ノードに移動します。

ロジカルスタンバイは、bdr.join_node_group の DSN で定義された 1 つのソースノードからデータが送信されます。他のすべてのノードからの変更はこの 1 つのソースノードから受信され、マルチプルのサイト間の帯域幅が最小化されます。

高可用性を実現するために、ソースノードが停止した場合、1 つのロジカルスタンバイをフルノードに昇格させ、シングルマスター操作と同様のフェールオーバー操作でソースを置き換えることができます。複数のロジカルスタンバイノードが存在する場合、他のノードは新しいマスターをフォローできないため、この手法の有効性は 1 つのロジカルスタンバイに制限されます。

既存の PGD ノードから新しいスタンバイが作成される場合、グループスロットが最後に進められてから少なくとも 16MB の LSN が経過するまで、操作に必要なレプリケーションスロットは新しいスタンバイに同期されません。極端な場合、これには、スロットがストリーミングレプリカで同期または作成される前に、完全な 16 MB が必要になる場合があります。この間隔でフェールオーバーまたはスイッチオーバーが発生した場合、グループスロットおよびその他の依存スロットがまだ存在していないため、ストリーミングスタンバイを PGD ノードに代わって昇格させることはできません。

スタンバイのスロット同期プロセスは、上流のファンクションを呼び出すことによりこれを解決します。この機能は、WAL スイッチを実行し、すべての PGD ピアノードに進行状況の更新を再生するように要求することにより、EDB Postgres 分散クラスター全体のグループスロットを移動します。この動作により、グループスロットが短期間で前進します。これにより、スタンバイが初期スロットの同期に必要な時間が削減され、必要に応じてより迅速なフェイルオーバーが可能になります。

PostgreSQL では、スロットを昇格する前に、スタンバイでスロットの同期が完了することを確認することが重要です。ターゲットデータベースのスタンバイで次のクエリを実行して、スロットがアップストリームと同期していることを監視および確認できます。このクエリーが `true` を返すと、プロモーションを続行できます。

```
SELECT true FROM pg_catalog.pg_replication_slots WHERE
    slot_type = logical AND confirmed_flush_lsn IS NOT NULL;
```

WAL スイッチを手動で実行し、すべての PGD ピアノードに進行状況の更新を再生するように要求することにより、PGD クラスター全体のスロット同期プロセスをナッジすることもできます。このアクティビティにより、グループスロットが短時間で前進し、スタンバイでのスロット同期アップアクティビティも高速化されます。このターゲットデータベース内の任意の PGD ピアノードで次のクエリーを実行できます。

```
SELECT bdr.run_on_all_nodes(SELECT pg_catalog.pg_switch_wal());
SELECT bdr.run_on_all_nodes(SELECT bdr.request_replay_progress_update());
```

スタンバイでモニタリングクエリを使用して、これらのクエリがそのスタンバイでのスロットの同期の高速化に役立つことを確認します。

ロジカルスタンバイでは、書き込みトランザクションが許可されます。これにより、ロジカルスタンバイに追加のインデックス、データのより長い保持期間、中間作業テーブル、LISTEN/NOTIFY、一時テーブル、マテリアライズドビュー、およびその他の違いを持つことができるため、大きなメリットがあります。

プロモーション前にコミットするロジカルスタンバイにローカルに加えられた変更は、他のノードに送信されません。プロモーション後にコミットするすべてのトランザクションは以降に送信されます。ロジカルスタンバイへの書き込みを実行する場合、昇格する前にデータベースを静止するように注意してください。

ロジカルスタンバイノードに DDL 変更を行う場合がありますが、それらはレプリケートされず、グローバル DDL ロックを取得しようとしません。DDL と同様に動作する PGD ファンクションもレプリケートされません。**ステートメント固有の *DDL* レプリケーションの問題** を参照してください。ロジカルスタンバイに互換性のない DDL 変更を行った場合、データベースは**ダイバージェントノード**です。現在、ダイバージェントノードのプロモーションはレプリケーションに失敗します。その結果、ロジカルスタンバイノードをスタンバイとして使用する場合は、ロジカルスタンバイノードに発散的な変更がないようにするか、発散ノードが昇格しないように計画します。

### Subscriber-only nodes and groups

サブスクライバー専用ノードとグループは、PGD クラスターに読み取りスケーリングを構築する強力な方法を提供します。

- *Overview of durability options* は、PGD でサブスクライバ専用ノードとグループがどのように動作するかを紹介します。
- *Creating Subscriber-only groups and nodes* は、サブスクライバー専用のグループとノードを作成する方法について説明します。
- *Joining nodes to a Subscriber-only group* は、メンバーがいる既存のサブスクライバー専用グループにノードを参加させる方法について説明します。
- *Optimizing subscriber-only groups* は、より効率的なレプリケーションのためにグループリーダーを使用する PGD サブスクライバのみの最適化トポロジ機能の構成方法の詳細を提供します。

## An overview of Subscriber-only nodes

### 概要

多くのユースケースは、クエリと更新を処理できるデータベースノードへのアクセスに依存していますが、読み取り専用のデータベースクエリを処理できるノードへのアクセスのみを必要とするユースケースもあります。このような読み取りスケーリングは、クラスター内のアクティブなデータベースノードから読み取り専用トラフィックを移動することにより、コアクラスターのパフォーマンスを向上させながら、データベースアクセスをより広く利用できるようにします。

### サブスクライバー専用ノード

サブスクライバー専用ノードの基本的なアイデアは、メインクラスターから読み取り専用クエリをオフロードするために使用できる読み取り専用ノードを提供することです。PGD クラスターのデフォルトのトポロジは、いわゆるフルメッシュトポロジであり、すべてのノードが他のすべてのノードに接続します。これは、ノードを接続する最も堅牢でフォールトトレラントな方法ですが、一部のユースケースでは非効率である場合があります。

サブスクライバ専用ノードは、サブスクライバ専用グループのメンバーになることができますが、PGD 6 以降では、データグループの一部になることができます。

### 購読者専用グループ

PGD の加入者専用グループは、加入者専用ノードを収集します。各グループは、さまざまなリージョンまたはさまざまなアプリケーションの要求に対応できます。

データグループとは異なり、購読者専用グループには、独自の raft コンセンサスメカニズムがありません。これは、加入者専用グループが必要な数の加入者専用ノードを持つことができることも意味します。

PGD 6 より前は、サブスクライバー専用グループの存在によってレプリケーショントポロジが変更されませんでした。デフォルトでは、サブスクライバ専用グループ内のすべてのノードは、クラスター内の他のすべてのノードからレプリケートされた変更を個別に受信します。

### 加入者専用グループの最適化

PGD 6 以降では、オプションで加入者専用グループのトポロジを最適化できます。

データノードにプロキシと raft 対応グループを使用するクラスターの場合、サブスクライバー専用グループは、レプリケートされた変更を受信するためのより効率的なモデルを使用できます。

最適化されたトポロジオプションは、PGD プロキシの書き込みリーダーと同様に、各サブスクライバ専用グループにグループリーダーを作成します。グループリーダーは、クラスターからすべての変更を受信し、グループ内の他のノードにそれらを複製します。この機能の詳細については、*Optimizing subscriber-only groups* を参照してください。

### サブスクライバー専用ノードと DDL

サブスクライバー専用ノードは、ローカルに発行された DDL コマンドを実行できますが、これらの変更はクラスター内の他のノードにレプリケートされず、クラスターでロックを取得しようとしません。

### Creating Subscriber-only groups and nodes

サブスクライバー専用ノードを作成するプロセスは、ノードを含めるサブスクライバー専用グループを作成することから始まります。PGD クラスター内の既存の完全に参加したノードでこの手順を実行します。

#### 購読者専用グループの手動作成

購読者専用グループを作成するには、グループの作成時に node_group_type を subscriber-only として指定する必要があります。たとえば、ここでは、「host-one」で実行されているノード「node-one」にログインしています。これは、独自のデータグループのメンバーであり、すべてのノードと同様に、ここでは topgroup と呼ばれるトップレベルグループのメンバーです。このノードに直接ログインして、次の SQL コマンドを使用して sogroup という名前の新しいサブスクライバー専用グループを作成します。

```
select bdr.create_node_group(sogroup, topgroup, false, subscriber-only);
```

またはパラメーター名を使用してより明示的に

```
select bdr.create_node_group(node_group_name:=sogroup,
        parent_group_name:=topgroup,
        join_node_group:=false,
        node_group_type:=subscriber-only);
```

これにより、 topgroup グループの子である sogroup という名前のサブスクライバー専用グループが作成されます。join_node_group の false パラメーターは、このコマンドを実行するノードが、新しく作成されたグループに参加しないことを示します。グループへの自動的な参加はデフォルトの動作であり、この場合、抑制する必要があります。

#### 新しいサブスクライバー専用グループへのノードの手動追加

新しいデータノードを初期化して、サブスクライバー専用グループに追加できるようになりました。データノードを作成し、他のデータノードの場合と同様に bdr 拡張機能を構成します。

次に、この新しいノードを subscriber-only ノードとして作成する必要があります。これを行うには、新しいノードにログインし、次の SQL コマンドを実行します。

```
select bdr.create_node(so-node-1, host=so-host-1 dbname=pgddb port=5444, subscriber-
↪only);
```

次に、その新しいノードにログインし、次の SQL コマンドを使用して sogroup グループに追加します。

```
select bdr.join_node_group(host=host-one dbname=pgddb port=5444,sogroup);
```

またはパラメーター名を使用してより明示的に

```
select bdr.join_node_group(dsn:=host=host-one dbname=pgddb port=5444,
        node_group_name:=sogroup);
```

これは、新しいノードに sogroup グループに参加するように指示します。クラスタートポロジの知識がないため、DSN で指定されたノードに接続して、グループに参加するために必要な情報を受信します。この

例では、これはたまたまサブスクライバー専用グループの作成に使用したノードと同じノードですが、クラスターに完全に参加しているノードである可能性があります。

### Joining nodes to a Subscriber-only group

PGD クラスターにサブスクライバー専用グループがない場合、 *Creating Subscriber-only groups and nodes* のプロセスに従ってグループを作成する必要があります。加入者専用グループを作成したら、加入者専用ノードをそれに参加できます。

### 既存の加入者専用グループへのノードの参加

*Creating Subscriber-only groups and nodes* へのノードの参加とは異なり、既存の加入者専用グループへのノードの参加はより簡単なプロセスです。

最初に、新しいノードを加入者専用ノードとして作成します。新しいノードで次の SQL コマンドを実行します。

```
select bdr.create_node(so-node-2, host=so-host-2 dbname=pgddb port=5444, subscriber-
↪only);
```

またはパラメーター名を使用してより明示的に

```
select bdr.create_node(node_name:=so-node-2,
        dsn:=host=so-host-2 dbname=pgddb port=5444,
        node_type:=subscriber-only);
```

このコマンドは、ホスト so-host-2 に so-node-2 という名前の新しいノードを作成し、サブスクライバ専用ノードとして構成します。ノードは、グループに参加するまでクラスターに参加できません。

*Creating Subscriber-only groups and nodes* では、 sogroup という名前のグループを作成し、ホスト shost-1 に so-node-1 というサブスクライバ専用ノードを追加しました。既存のデータグループのノードを使用して、その結合を促進します。ただし、この新しい加入者専用ノードを使用して、別の加入者専用ノードを追加することはできません。クラスターに完全に参加しているアクティブなデータノードを使用する必要があります。*Creating Subscriber-only groups and nodes* の例では、このタスクのクラスターのデータグループで host-one を使用します。 shost-2 で次の SQL コマンドを使用して、sogroup グループに参加できます。

```
select bdr.join_node_group(host=host-one dbname=pgddb port=5444,sogroup);
```

またはパラメーター名を使用してより明示的に

```
select bdr.join_node_group(dsn:=host=host-one dbname=pgddb port=5444,
        node_group_name:=sogroup);
```

このコマンドは、新しいノードに sogroup グループに参加するように指示します。クラスタートポロジの知識がないため、DSN で指定されたノードに接続して、グループに参加するために必要な情報を受信します。そのノードは、新しいノードをグループに参加する要求のソースとして機能するため、クラスターに完全に参加する必要があります。

新しいノードがグループに参加すると、最初に同期することから開始され、クラスター内の他のノードからレプリケーション変更の受信を開始します。

> **注釈**
>
> グループが *Optimizing subscriber-only groups* を使用している場合を除き、参加しているサブスクライバ専用グループのサブスクライバ専用グループリーダーから変更をレプリケートします。

### Optimizing subscriber-only groups

PGD 6 以降では、 *Overview of durability options* のトポロジを最適化できるようになりました。

この最適化されたトポロジでは、少数の完全にアクティブなノード-データグループの書き込みリーダー-が加入者専用グループのグループリーダーに変更をレプリケートします。これらのグループリーダーは、加入者専用グループの他のメンバーに変更を複製します。

#### 最適化されたトポロジの要件

クラスターに次のいずれかがある場合、このモデルを有効にすることはできません。

- トップレベルグループの直接メンバーであるデータノード
- データノードサブグループなし
- プロキシルーティングが有効になっているデータノードサブグループはありません

これらのいずれかに該当する場合、加入者専用グループのノードはフルメッシュトポロジに戻ります。

新しい SO グループとノードレプリケーションの利点を得るには、データノードをサブグループに配置し、サブグループでプロキシルーティングを有効にする必要があります。

#### 最適化されたトポロジの仕組み

データノードにグループを使用するクラスターの場合、サブスクライバー専用グループは、より効率的なモデルを使用できます。このモデルは、PGD プロキシの書き込みリーダーと同様に、サブスクライバ専用のグループリーダーを使用します。

各加入者専用グループは、そのグループリーダーを使用して、グループ内の他の加入者専用ノードに変更を複製します。グループリーダーは、入ってくる変更のレプリケーションプロキシとして機能します。

データグループの書き込みリーダーノードは、加入者専用グループのグループリーダーに変更を複製します。データグループ内の他のノードは、自分のデータグループ内のノードおよび他のデータグループのデータノードとのみ複製します。変更を加入者専用グループに直接複製しません。

#### 加入者専用グループリーダー

PGD 6 以降では、各加入者専用グループに独自のグループリーダーが割り当てられます。これは、加入者専用グループには、独自のグループ Raft コンセンサスメカニズムがないためです。代わりに、クラスターのトップレベルグループは Raft コンセンサスメカニズムを使用して、各加入者専用グループのグループリーダーの選択を処理します。このグループリーダーの選択は、トポロジ最適化設定に関係なく、PGD 6 ではデフォルトでオンになっています。

加入者専用グループのグループリーダーは定期的に接続性がテストされ、使用できない場合、トップレベルグループの投票ノードは、加入者専用グループから新しい加入者専用ノードを選択して、グループリーダーになります。次に、新しいグループリーダーが選択されます。

最適化されたテクノロジーがオフになっているため、この選択はレプリケーショントポロジに影響を与えません。最適化されたトポロジがないと、すべてのデータノードがクラスター内の他のすべてのノードに変更をレプリケートします。

### 最適化されたトポロジのグループリーダー

最適化されたトポロジが有効になると、サブスクライバ専用グループのグループリーダーのみがクラスター内の他のデータグループの書き込みリーダーから変更を受け取ります。グループリーダーは、これらの変更を加入者専用グループ内の他のノードに複製する責任を負います。

他の投票ノードは、加入者専用グループのノードからグループリーダーを選択します。グループリーダーが選択されると、クラスター全体が変更を認識し、データグループの書き込みリーダーは、この新しく選択されたグループリーダーノードにのみデータを複製します。データグループ内の他のデータノードは、加入者専用グループのノードにデータを複製しません。

このアプローチは、多数の SO ノードが存在する場合に発生する可能性のあるアクティブな接続の爆発を回避し、レプリケーショントラフィックの量を削減します。

サブスクライバ専用ノードとグループは、PGD ツリートポロジのビルディングブロックを形成します。

### 最適化されたモデルの有効化

デフォルトでは、 PGD 6 はフルメッシュトポロジを強制します。これは、ここで説明する最適化がオフであることを意味します。最適化されたトポロジを有効にするには、データノードをサブグループに配置し、サブグループでプロキシルーティングを有効にする必要があります。次に、GUC bdr.force_full_mesh を `off` に設定して、最適化をアクティブにできるようにします。

> **注釈**
>
> この GUC は、各データノードと各ノードの `postgresql.conf` ファイルで設定する必要があります、変更を有効にするには再起動します。

最適化されたトポロジの要件が満たされない場合、加入者専用グループのノードはフルメッシュトポロジに戻ります。これが発生すると、クラスターメッセージ内のノードのログで、次のような最適化が不可能な理由を見つけます。

データノードがトップレベルのノードグループの一部である場合

```
node: <nodename> is part of top-level nodegroup: <toplevelgroupname>: changing to␣
↪full mesh".
```

データグループでプロキシルーティングが有効になっていない場合

```
node: <nodename> is in nodegroup: <nodegroupname> that does not have proxy routing:␣
↪changing to full mesh.
```

### 6.3.13 Node management

PGD クラスター内のすべてのデータノードは、1 つ以上のグループのメンバーです。デフォルトでは、すべてのデータノードは、PGD クラスター内のすべてのデータノードにまたがるトップレベルグループのメンバーです。ノードは、PGD クラスターの論理または地理的組織を反映するように構成できるサブグループに属することもできます。

ノードとサブグループで使用可能なさまざまなオプションを使用して、ノードとグループを管理できます。

- *Creating PGD nodes* は、PGD クラスターに新しいノードを作成するために必要な手順をカバーしています。
- *Groups and subgroups* は、PGD でグループとサブグループがどのように動作するかについて詳しく説明します。
- *Creating and joining PGD groups* は、新しい PGD グループを作成する方法と、PGD ノードをそれらに参加する方法を検討します。
- *Viewing PGD topology* は、PGD クラスターのノードとグループの構造を表示できるコマンドと SQL クエリーの詳細を示しています。
- *Removing nodes and groups* は、グループからノード、またはクラスターからグループを安全に削除するために実行するプロセスを示します。
- *Connection DSNs and SSL (TLS)* は、PGD クラスター内のノードに直接接続するために必要な DSN または接続文字列を導入します。また、SSL / TLS 証明書を使用して、サーバー間およびクライアント間の認証と暗号化を提供する方法についても説明します。
- *Node restart and down node recovery* は、障害またはスケジュールされたダウンタイムの後にノードをサービスに戻すために必要な手順と、それが返すクラスターに与える影響について詳しく説明します。
- *Automatic synchronization* は、PGD での自動同期機能がどのように動作し、それを使用してノード間の同期を維持する方法を検討します。
- *Node UUIDs* は、ノードの UUID が PGD で使用される方法と、それらがどのように生成されるかについて説明します。
- *Replication slots created by PGD* は、PGD が動作しているときに Postgres レプリケーションスロットがどのように消費されるかを調べます。

#### Creating PGD nodes

##### それはただの Postgres です

PGD ノードは、 BDR 拡張機能がインストールされた Postgres インスタンスです。BDR 拡張機能は、ノード間の双方向レプリケーションを有効にし、PGD の基盤です。

つまり、最も一般的な言葉では、Postgres と BDR 拡張機能をインストールし、PGD グループ内の他のノードに接続するようにノードを構成することにより、PGD ノードを作成できます。ただし、考慮すべき詳細がいくつかあります。

### Postgres のバージョンはどれですか?

PGD は Postgres の上に構築されているため、PGD ノードに使用する Postgres のディストリビューションとバージョンが重要です。使用する Postgres のバージョンは、使用している PGD のバージョンと互換性がある必要があります。互換性マトリックスは release notes で見つけることができます。PGD の機能は、使用している Postgres のディストリビューションに依存する場合があります。EDB Postgres Advanced Server は、PGD の推奨ディストリビューションです。 PGD は、EDB Postgres Extended Server および Community Postgres もサポートしています。

### Postgres のインストール

構成する各ノードに選択した Postgres ディストリビューションをインストールする必要があります。EDB Postgres Advanced Server documentation 、 EDB Postgres Extended Server documentation 、および *Postgres installation documentation* で、各ディストリビューションのインストール手順を見つけることができます。Postgres のインストールをカバーする *Step 3 - Installing the database and pgd* を参照することもできます。

### BDR 拡張機能のインストール

BDR 拡張機能は、PGD の分散アーキテクチャのキーとなります。PGD クラスター内の各ノードに BDR 拡張機能をインストールする必要があります。BDR 拡張機能は、 EDB Postgres Distributed リポジトリから入手できます。`postgres_distributed` リポジトリを Linux のパッケージ管理システムに追加し、`edb-bdr` パッケージをインストールする必要があります。リポジトリー構成手順は *PGD manual installation guide* にあります。

リポジトリの構成が完了したら、パッケージマネージャーを使用して BDR パッケージをインストールできます。パッケージ名は `edb-pgd6-<postgresversion>` で、`<postgresversion>` は、使用している Postgres のバージョンです。たとえば、Postgres 14 を使用している場合、パッケージ名は `edb-pgd6-14` です。

### PGD のデータベースの構成

このプロセスは PGD に固有であり、BDR 拡張機能と連携するように Postgres インスタンスを構成し、PGD クラスターと連携するようにさまざまな設定の調整が含まれます。手順は次のとおりです。

- `postgresql.conf` の `shared_preload_libraries` 設定の先頭に BDR 拡張`$libdir/bdr` を追加します。
- `postgresql.conf` で `wal_level` GUC 変数を `logical` に設定します。
- `postgresql.conf` で `track_commit_timestamp` を`'on'` に設定することにより、コミットタイムスタンプの追跡をオンにします。
- `postgresql.conf` で `max_worker_processes` ～`'16'` を設定することにより、最大ワーカープロセスを 16 以上に増やします。

**注釈**

`max_worker_processes` 値

***max_worker_processes*** **値は、クラスターのトポロジ、ピアの数、データベースの数、およびその他の要素から導き出されます。 必要な値を計算するには、** ***Postgres configuration/settings*** **を参照してください。値 16 は、この例で展開されるクラスターのサイズとして計算されました。大規模なクラスターの場合、増加する必要があります。 .. ::**

- EnterprisedDB / Postgres ユーザーにパスワードを設定します。

- ルールを `pg_hba.conf` に追加して、ノードが相互に接続できるようにします。 - これらの行が `pg_hba.conf` に存在することを確認します。

```
host all all all md5
host replication all all md5
```

- `.pgpass` ファイルを追加して、ノードが相互に認証できるようにします。
  - 他のノードにログインするための十分な特権を持つユーザーを構成します。
  - `.pgpass` ファイルの詳細については、Postgres ドキュメントの *The Password File* を参照してください。

これらの手順が完了したら、Postgres インスタンスを再起動して変更を適用します。

#### PGD ノードの初期化

構成したデータベースインスタンスにログインし、BDR 拡張機能をセットアップします。これは、データベースでスーパーユーザーとして `CREATE EXTENSION bdr;` コマンドを実行することにより実行できます。このコマンドは、BDR 拡張機能を作成します。

また、Postgres 内でデータベースを作成して、PGD のレプリケートデータベースとして使用する必要があります。これは、 `CREATE DATABASE` コマンドで行うことができます。作成されるデータベースは、PGD クラスター内の他のノードがレプリケートするデータベースの名前である必要があります。通常、データベースの名前は `pgddb` です。

#### 次のステップ

ノードの構成が完了し、PGD クラスターでグループに参加またはグループを開始する準備が整います。*Joining a node to a group* セクションで、ノードをグループに参加させる手順を見つけます。

### Groups and subgroups

#### グループ

PGD クラスターのノードはグループに集められます。「トップレベル」グループは常に存在し、すべてのデータノードが自動的に属するグループです。「トップレベル」グループは、サブグループの直接の親になることもできます。

#### サブグループ

グループには、0 個以上のサブグループを含めることもできます。サブグループは、データセンターまたは場所を表すために使用でき、コミットスコープが特定のリージョンのノードを全体として参照できます。接続マネージャーは、サブグループを利用して、書き込みリーダーになることができるノードを描写することもできます。

node_group_type 値は、サブグループの作成時のタイプを指定します。一部のサブグループタイプは、グループ内のノードの動作を変更します。たとえば、 サブスクライバー専用ノード サブグループは、グループ内のすべてのノードをサブスクライバ専用ノードにします。

### Creating and joining PGD groups

### PGD グループの作成と参加

PGD の場合、すべてのノードは他のすべてのノードに接続する必要があります。構成を簡単にするために、新しいノードが参加すると、既存のすべてのノードがそれに接続するように構成します。このため、最初に作成した PGD ノードを含むすべてのノードは、*PostgreSQL connectionstring*

他のノードが接続するために使用できるもの。この接続文字列は、データソース名 DSN と呼ばれることがあります。

両方の形式の接続文字列がサポートされています。したがって、host=myhost port=5432 dbname=mydb のようなキー-値形式、または postgresql://myhost:5432/mydb のような URI 形式を使用できます。

SQL ファンクション *bdr.create_node_group()*

ローカルノードから PGD グループを作成します。これにより、そのノードで PGD がアクティブになり、他のノードがその時点で 1 つのノードのみで構成される PGD グループに参加できるようになります。作成時に、このノードへの接続に使用する他のノードの接続文字列を指定する必要があります。

ノードグループが作成されると、それ以降のすべてのノードは、*bdr.join_node_group()* を使用して PGD グループに参加できます。

ファンクション。

または、コマンドラインユーティリティ bdr_init_physical を使用して、pg_basebackup を使用して新しいノードを作成します。 pg_basebackup を使用する場合、bdr_init_physical ユーティリティはオプションでターゲットデータベースのみのベースバックアップを指定できます。以前の動作は、データベースクラスター全体をバックアップすることでした。このユーティリティを使用すると、アクティビティがより速く完了し、不要なデータベースを除外するため、使用する領域が少なくなります。ターゲットデータベースのみを指定した場合、除外されたデータベースは新しいノードでクリーンアップおよび削除されます。

新しい PGD ノードが既存の PGD グループに参加するか、ノードが上流ピアをサブスクライブする場合、レプリケーションを開始する前に、システムは既存のデータをピアノードからローカルノードにコピーする必要があります。このコピーは、ローカルデータとリモートデータが同一に始まるように、注意深く調整する必要があります。pg_dump を自分自身で使用するだけでは十分ではありません。BDR 拡張機能は、この初期コピーを作成するためのビルトイン機能を提供します。

参加プロセス中に、BDR 拡張機能は提供されたソースノードをベースとして使用して既存のデータを同期し、PGD グループのメッシュトポロジで自分自身を確立するために必要なすべてのメタデータ情報を作成します。この初期コピー中にソースと新しいノード間の接続が切断された場合、参加プロセスを最初からリスタートします。

クラスターに参加するノードには、PGD グループのデータベースに既に存在するスキーマまたはデータが含まれていない必要があります。新しく参加するデータベースは、BDR 拡張機能を除き空であることをお勧めします。ただし、必要なデータベースユーザーとロールがすべて作成されていることが重要です。また、非スーパーユーザーが参加操作を実行している場合、スーパーユーザーの権限を必要とする拡張機能を手動で作成する必要があります。詳細については、接続とロール を参照してください。

オプションで、 bdr.join_node_group の synchronize_structure パラメーターを使用して、スキーマの同期をスキップできます。

ファンクション。この場合、スキーマは新しく参加するノードに既に存在する必要があります。

最適な接続論理的に近く、理想的には低遅延、高帯域幅を持つソースノードを、参加するソースノードとして選択することをお勧めします。そうすることで、結合が完了するまでに必要な時間が短縮されます。

Raft コンセンサスアルゴリズムを使用して参加手順を調整します。これには、既存のノードのほとんどがオンラインで到達可能である必要があります。

論理結合プロシージャ bdr.join_node_group を使用

ファンクションは、COPY 操作を実行してデータ同期を実行し、マルチプルのライターパラレル適用が有効になっている場合を使用します。

ノード参加は、参加にかかる時間の大部分で他のノード参加と同時に実行できます。ただし、一度に 1 つのレギュラーノードのみが PROMOTE または PROMOTING の状態になることができます。他のすべてのノードが起動して実行されている場合、これらの状態は通常かなり短いです。それ以外の場合、結合はこの段階でシリアル化されます。サブスクライバ専用ノードはこのルールの例外であり、それらは同時に PROMOTE および PROMOTING 状態になることもできるため、それらの参加プロセスは完全にコンカレントです。

参加プロセスはソースとして 1 つのノードのみを使用するため、ノードの大部分が利用可能な場合、ノードがダウンしているときに実行できます。このアプローチでは、論理結合を実行するときに複雑さが発生する可能性があります。論理結合中に、ソースノードからコピーされた行のコミットタイムスタンプは、ソースノードの最新のコミットタイムスタンプに設定されます。これより前のコミットタイムスタンプを持つノードでコミットされた変更は、ノードがダウンしているか、大幅な遅延があるため、他のノードからの変更と競合する可能性があります。この場合、新しく参加したノードは他のノードとは異なる方法で解決され、発散が発生する可能性があります。その結果、ノード間に重大なレプリケーションラグが存在する場合は、ノード参加を実行しないことをお勧めします。これが必要に応じて、すべてのノードが利用可能になって追いついた後、新しく参加したノードで LiveCompare を実行してデータの相違を修正します。

pg_dump は、ソースノードに同時の DDL アクティビティがある場合、キャッシュルックアップの失敗が原因で失敗する場合があります。bdr.join_node_group は内部で pg_dump を使用するため、ソースノードに同時の DDL アクティビティがある場合、失敗する可能性があります。その場合、結合を再試行すると機能します。

### Viewing PGD topology

#### PGD グループのリスト

#### pgd-cli の使用

pgd-cli groups list コマンドを使用して、PGD クラスター内のすべてのグループをリストします。

```
pgd groups list
```

```
Group Name  Parent Group Name Group Type     Nodes
----------- ----------------- -------------- -----
bdrgroup    bdrgroup          global         0
```

(次のページに続く)

(前のページからの続き)

```
group_a    bdrgroup          data             4
group_b    bdrgroup          data             4
group_c    bdrgroup          data             1
group_so   bdrgroup          subscriber-only 1
```

#### SQL を使用する

次の単純なクエリーは、現在のノードがメンバーであるすべての PGD ノードグループをリストします。現在、bdr.local_node_summary から 1 行のみを返します。

```
SELECT node_group_name
FROM bdr.local_node_summary;
```

より複雑なクエリーを使用して、各ノードグループの構成を表示できます。

```
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);
```

### PGD グループ内のノードのリスト

#### pgd-cli の使用

nodes list コマンドを使用して、PGD クラスター内のすべてのノードをリストします。

```
pgd nodes list
```

```
Node Name           Group Name Node Kind       Join State Node Status
------------------- ---------- --------------- ---------- -----------
bdr-a1              group_a    data            ACTIVE     Up
bdr-a2              group_a    data            ACTIVE     Up
logical-standby-a1  group_a    standby         ACTIVE     Up
witness-a           group_a    witness         ACTIVE     Up
bdr-b1              group_b    data            ACTIVE     Up
bdr-b2              group_b    data            ACTIVE     Up
logical-standby-b1  group_b    standby         ACTIVE     Up
witness-b           group_b    witness         ACTIVE     Up
witness-c           group_c    witness         ACTIVE     Up
subscriber-only-c1  group_so   subscriber-only ACTIVE     Up
```

グループ名で grep を使用して、リストを特定のグループにフィルタリングします。

```
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
```

#### SQL を使用する

bdr.node_summary ビューから、特定のノードグループ mygroup`などのすべてのノードのリストを抽出できます。例

```
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

#### PGD グループからのノードの削除

PGD は、長期のノードの停止から復旧するように設計されているため、ノードを完全に削除するかどうかをシステムに明示的に指示する必要があります。ノードを完全にシャットダウンし、他のノードに通知しない場合、パフォーマンスが低下し、最終的にシステム全体が動作を停止します。

ノードの削除は *parting* とも呼ばれ、*bdr.part_node()* を使用して実行されます

ファンクション。ノードを削除するには、ノードの作成中に渡されるノード名を指定する必要があります。*bdr.part_node()* ファンクションは、削除するノードを含む、PGD グループのアクティブなノードから呼び出すことができます。

参加手順と同様に、分割は Raft コンセンサスを使用して実行され、動作するにはノードの大部分がオンラインである必要があります。

分離プロセスはすべてのノードに影響を与えます。Raft リーダーはノード間の投票を管理して、どのノードが分離ノードからの最新のデータを持つかを確認します。次に、残りのすべてのノードが最新のノードへのセカンダリの一時的な接続を作成して、不足しているデータをキャッチアップできるようにします。

分離されたノードは引き続き PGD に知られていますが、リソースを消費しません。ノードは、分割されたノードと同じ名前で再度追加される場合があります。まれに、ファンクション *bdr.drop_node()* を使用して、分割されたノードのすべてのメタデータをクリアすることが必要になる場合があります。

### PGD グループ全体の削除

通常、PGD グループは場所にマッピングされます。ロケーションが展開されない場合、そのロケーションの PGD グループも削除する必要がある可能性があります。

削除される PGD グループは空である必要があります。グループを削除する前に、グループ内のすべてのノードを分割する必要があります。

### Connection DSNs and SSL (TLS)

ノードは `libpq` を使用して接続するため、ノードの DSN は *libpq* 接続文字列です。そのため、接続文字列には、SSL の接続パラメーターを含む許可された `libpq` 接続パラメーターを含めることができます。DSN は、指定されたノードに接続するクライアントからの接続文字列として機能する必要があります。クライアント証明書を使用するこのようなパラメーターセットの例は次のとおりです。

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

この設定では、ファイル `bdr_client.crt` 、`bdr_client.key` 、および `root.crt` が、適切な権限を持つ各ノードのデータディレクトリに存在する必要があります。`verify-full` モードの場合、サーバーの SSL 証明書がチェックされ、`root.crt` 認証局で直接または間接的に署名されていること、および接続で使用されるホスト名またはアドレスが証明書の内容と一致することを確認します。名前の場合、これはサブジェクトの別名、または証明書にそのような名前が存在しない場合、サブジェクトのコモンネーム CN フィールドと一致します。Postgres は現在 IP アドレスのサブジェクト代替名をサポートしていないため、名前ではなくアドレスによって接続が行われる場合、CN フィールドと一致する必要があります。

クライアント証明書の CN は、PGD 接続を行うユーザーの名前である必要があります。通常はユーザー postgres です。各ノードには、`pg_hba.conf` ファイルでの接続を許可する一致する行が必要です。例

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

別の設定は、クライアント証明書の代わりに `SCRAM-SHA-256` パスワードを使用し、証明書が適切に署名されている限りサーバー ID を検証しないことです。ここで、DSN パラメーターは次のとおりです。

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

対応する `pg_hba.conf` 行は次のとおりです。

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

このようなシナリオでは、postgres ユーザーには、正しいパスワードを含む *.pgpass* ファイルが必要です。

### Node restart and down node recovery

PGD は、ノードの再起動またはノードの切断から復旧するように設計されています。切断されたノードは、各ピアノードに再接続し、そのノードから欠落しているデータを複製することにより、グループに再参加します。

ノードが起動すると、各接続は `bdr.node_slots.state = catchup` を使用して bdr.node_slots に表示され始め、欠落データの複製を開始します。キャッチアップは、各ピアノードからの欠落データの量に応じて継続され、サーバーのワークロードに応じて、時間の経過とともに増加する可能性があります。

各ノードの書き込みアクティビティの量が均一でない場合、より多くのデータを持つノードからのキャッチアップ期間は他のノードよりも大幅に長くかかる場合があります。最終的に、スロットの状態は `bdr.node_slots.state = streaming` に変更されます。

時間や日などの長期間オフラインになっているノードは、さまざまな理由でリソースの問題を発生させる可能性があります。次の問題を理解することなく、長期の停止を計画しないでください。

各ノードは変更情報を保持します 1 つの replication slot を使用

ピアノードごとに) したがって、後で一時的に到達不能なノードへの変更を再生できます。ピアノードが無期限にオフラインになったままである場合、この蓄積された変更情報により、最終的にノードは PostgreSQL トランザクションログ `pg_wal` の *WAL* 用の記憶領域が不足し、これに似たエラーでデータベースサーバーがシャットダウンする可能性があります。

```
PANIC: could not write to file "pg_wal/xlogtemp.559": No space left on device
```

または、他のディスク不足関連の症状を報告する場合があります。

さらに、オフラインノードのスロットもカタログ xmin を抑制し、カタログテーブルのバキュームを防止します。

EDB Postgres Extended Server および EDB Postgres Advanced Server では、オフラインノードもデータのフリーズを抑制して、競合解決データの損失を防ぎます Origin conflict detection を参照してください。

管理者は、ノードの停止を監視し Monitoring を参照して、ノードに十分な空きディスク領域があることを確認する必要があります。ワークロードが予測可能な場合は、時間の経過とともに使用される領域の量を計算できるため、重要な問題が発生する前にノードがダウンできる最大時間を予測できます。

PGD によって作成されたレプリケーションスロットを手動で削除しないでください。そうした場合、クラスターは損傷し、*Replication slots created by PGD* で説明しているように、スロットを使用していたノードをクラスターから分離する必要があります。

ノードがオフラインである間、他のノードはオフラインノードから同じデータのセットをまだ受け取っていない可能性があるため、これはノード全体でのわずかな相違として現れる場合があります。分離プロセスは、ノード間のこの不均衡を修正します。

パートキャッチアップと呼ばれる分離のフェーズ中に、オフラインノードに関して他のすべてのノードから最も遠いノードが選択されます。他のノードがこの最も遠いノードに関して等しく追いついていない場合、最も先のノードをソースとして、オフラインノードをオリジンとして、等しく追いついていない各ノードをターゲットとして同期が開始されます。同期は、基本的にターゲットノードからソースノードの最前方ノードへのサブスクリプションであり、オフラインノードオリジンからターゲットノードに変更を転送します。

他のノードがどれだけ遅れているかによっては、この同期は別れるときに時間がかかる場合があります。同期が完了し、すべてのノードが等しく追いつくと、分割はノードの分割に進みます。この同期がなければ、強制的な部分が実行されると、クラスターの状態は一貫していない場合があります。これは、データが分岐する可能性があることを意味します。自動同期機能により、ノードがオフラインになると、これが検出され、すべてのノードが同期プロセスによってこのオフラインノードに関して等しくキャッチアップされます。これにより、ノードが分離するまで待機してデータの一貫性を保証する必要がありません。

### Automatic synchronization

#### 同期の自動トリガー

BDR マネージャープロセスは、同期要求の自動トリガーを実行します。bdr.replay_progress_frequency の 3 倍を超える時間ノードからの更新がない場合、ダウンしているとみなされます。

ノードは互いの近さについてチェックされます。すべてのノードが等しく追い付いている場合、同期は必要ありません。そうでない場合、「ダウン」ノードから最も前にあるノードがソースとして選択されます。ソースが決定されると、各ターゲット-オリジンとソース以外のノード-に対して、同期要求が設定されます。Witness ノードとスタンバイノードは、同期のターゲットである必要はありません。

ビュー bdr.sync_node_requests_summary は、同期要求を追跡します。

- `Origin` オリジンノードはダウンノードです。
- `Source` ソースノードは、オリジンから最も遠いノードです。
- `Target` オリジンに関してソースの背後にある他の各ノード。
- `Sync_start_lsn` 同期が開始されたときにオリジンからターゲットが受信した最大 LSN。
- `Sync_end_lsn` 同期が終了したときのオリジンからのターゲットノードのターゲット LSN。
- `Sync_status` 同期のステータス。
- `Sync_start_ts` 同期が開始された時刻。

同期要求がカタログに入力されると、完了まで転送されます。

#### キャンセル

選択したソースノードがダウンしていることが判明した場合、マネージャーは同期操作をキャンセルします。これは、他のノードが起動している可能性があるため、最も遠いとは言えませんが、少なくとも一部のターゲットよりも前にあります。また、ノードを同期するために使用される場合があります。したがって、マネージャーはダウンしたノードをソースとして持つすべての同期操作をキャンセルし、ダウンしていない別のノードを同期のソースとして選択します。成功した同期とキャンセルされた同期のステートマシンを以下に説明します。

同期キャンセル API *bdr.sync_node_cancel()* は、手動でのみ使用され、同期要求が何らかの理由でスタックし、クラスターの通常の機能をブロックしている場合にのみ使用されます。

```
select bdr.sync_node_cancel(origin, source)
```

これにより、指定されたオリジンとソースを持つすべてのターゲットのすべての同期ノード要求がキャンセルされます。これは、書き込みリードからのみ呼び出すことができます。

### 同期要求のライフサイクル

単一の同期要求には、オリジン、ソース、ターゲット、および到達する sync_end_lsn があります。同期要求はさまざまな状態を経て、各状態は別のノードで実行されます。

状態は次のとおりです。 **setup** 書き込みリードで実行します。sync_end_lsn を除く同期のフィールドを設定します。 **setup_source** ソースで実行します。 sync_end_lsn を設定し、同期サブスクリプションのスロットを作成します。**setup_target** これはターゲットノードで実行されます。この状態では、オリジンへの元のサブスクリプションは無効になっています。同期サブスクリプションはターゲットで設定され、オリジンの変更をソースノードからターゲットに転送します。**start** これはターゲットで実行されます。ターゲットの進行状況を監視して、sync_end_lsn に到達したかどうかを確認し、到達した場合は同期状態に移動します。**synced** サブスクリプションは sync_end_lsn に同期しました。この状態では、スロットはドロップされます。 **complete** この状態はターゲットで実行されます。この状態では、同期サブスクリプションがターゲットでドロップされ、元のサブスクリプションが有効になります。次に、done 状態に移行します。**done** これは、同期が成功したことを意味します。 **cancel start** これはターゲットノードで実行されます。この状態では、同期サブスクリプションは後でドロップする準備として無効になり、オリジンへの元のサブスクリプションが再度有効になります。**cancel resume** これはターゲットノードで実行されます。この状態では、同期サブスクリプションがドロップされます。**cancel denied** これはソースノードで実行されます。この状態では、スロットはドロップされます。**failed** キャンセルが発生し、すべてのクリーンアップが完了した場合、同期はこの状態で終了します。これは、同期が発生せず、再試行する必要があることを意味します。

同期のキャンセルは、選択したソースノードがダウンしていることが判明した場合に、自動的に発生する場合があります。キャンセル中に、サブスクリプションとスロットをクリーンアップし、元のサブスクリプションを有効にする必要があります。ソースまたはターゲットノードがダウンしている場合、同期要求はストールする可能性があります。

### GUC

自動同期を制御する GUC は bdr.enable_auto_sync_reconcile で、デフォルトで true に設定されています。オフにするには、すべてのノードで false に設定し、サーバーを再起動する必要があります。

### Node UUIDs

PGD 6 では、各ノードに、クラスター内のノードを識別するために使用される UUID があります。この UUID は、ノードの作成時に生成され、そのノードに一意です。UUID は、PGD のさまざまな場所で見つけることができます。

- bdr.node テーブル。クラスター内の各ノードに関する情報が含まれています。
- bdr.node_summary ビュークラスター内のノードの人が判読可能なビューを提供します。
- bdr.local_node テーブル、ローカルノードに関する情報が含まれています。
- uuid 値は、ノードごとに作成されるレプリケーションスロットの名前付けにも表示されます。

PGD のノード管理で使用されますが、UUID の使用は PGD の既存の機能に影響を与えません。UUID は、ノードとグループを識別するために内部で使用され、ユーザーが PGD と対話する方法を変更しません。

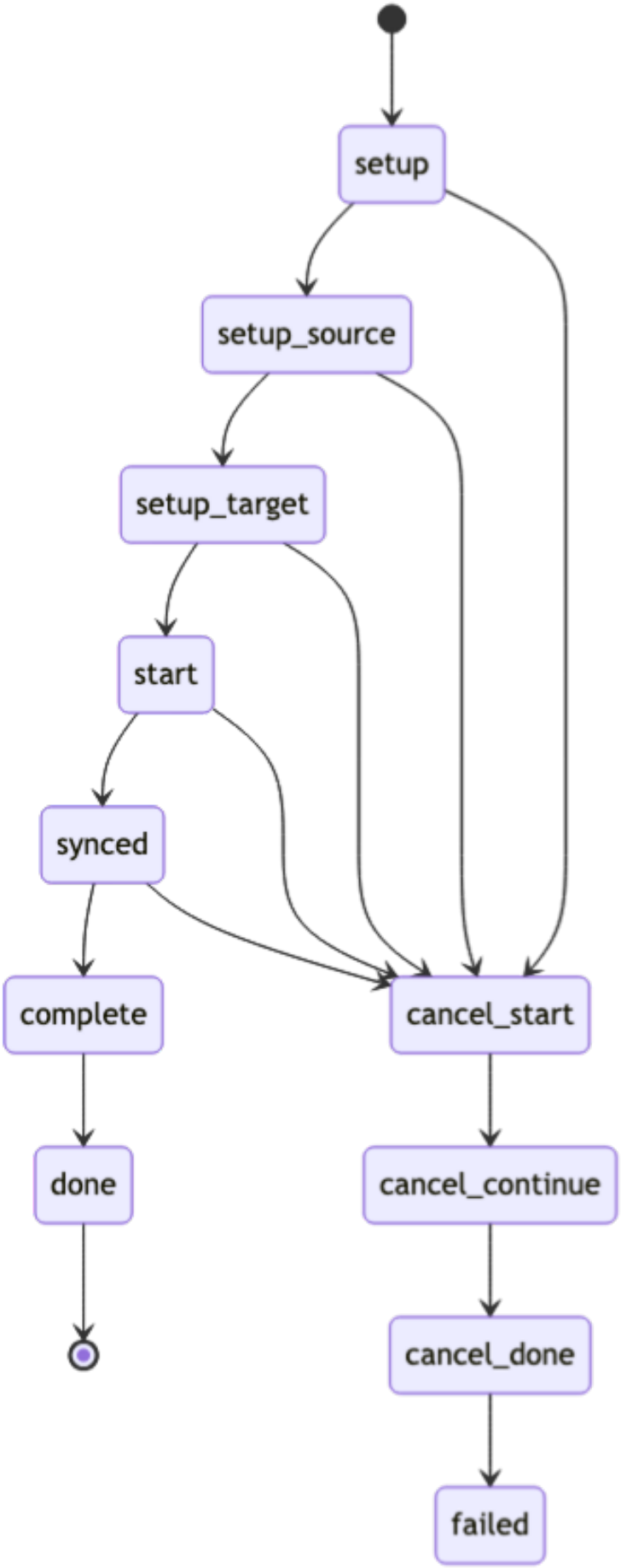


図 1: Autosync Request Life Cycle

#### なぜ UUID なのか?

UUID は PGD で使用され、クラスター内の各ノードに一意の識別子を提供します。PGD の以前のバージョンは、ノード名を識別子として使用していましたが、2 つのノードの名前が同じである場合、競合が発生する可能性がありました。UUID を使用することにより、PGD は各ノードに時間の経過とともに変更されない一意の識別子を持つことを保証できます。これは、クラスターにノードが頻繁に追加または削除される可能性がある PGD のような分散システムでは特に重要です。UUID により、新しいノードは既存のノードと同じ名前を持つ場合がありますが、異なる UUID を持ち、既存のノードと競合しないことが保証されます。

#### UUID はどのように生成されますか?

新しいノードが作成されると、そのノードの UUID が生成されます。この UUID は、カーネルの強力な乱数ジェネレーターを使用して作成され、一様にランダムであることが保証されています。この保証により、UUID は一意であり、簡単に推測できないことが保証されます。生成された UUID は `bdr.node` テーブルに保存され、クラスター内のノードを識別するために使用されます。

#### ノードが削除され、置換が追加された場合はどうなりますか?

ノードがクラスターから削除され、置換ノードが追加されると、置換ノードには新しい UUID が割り当てられます。これにより、置換ノードがクラスター内の別のエンティティとして扱われ、既存のノードと競合しないことが保証されます。ただし、PGD は、新しいノードを受け入れる前に、古いノードをクラスターから完全に分離する必要があります。置換ノードの UUID は、クラスター内の他のノードの UUID と同じ方法で使用されます。

#### PGD 6 の UUID 関連の変更

- 以前にノードを区別するために使用されていた `bdr.node` テーブルの `generation` フィールドは、使用されなくなりました。すべてのノードで 0 のままです。
- `bdr.node` テーブルの `node_uuid` フィールドは、PGD 6 で null になることはありません。将来、混合バージョンクラスターで null になる可能性があります。

### Replication slots created by PGD

PGD の以前のバージョンでは、レプリケーションスロットには人間が判読できる名前がありました。PGD 6 は、ノードとグループの UUID の使用に切り替えて、より良い識別を保証します。

レプリケーションスロットは、PostgreSQL がレプリケーションの進行状況を追跡するために使用します。これらは、レプリケートされるデータが失われず、レプリケーションプロセスの一貫性を保証するために使用されます。PGD では、レプリケーションスロットを使用して、そのノードからのレプリケーションの進行状況を追跡します。ダウンストリームノードごとに 1 つのスロットがあります。特定のノードからすべてのダウンストリームノードにわたってグローバルにレプリケーションの進行状況を追跡するために使用される特別なレプリケーションスロットもあります。

- `bdr_<topgroupuuid>_<dbhash>` という名前のグループスロット 1 つ
- `bdr_node_<targetnodeuuid>_<dbhash>` という名前の N-1 ノードスロット。N は、クラスター内のノードの合計数です。ダイレクトロジカルスタンバイがある場合は、

topgroupuuid は、トップレベルグループの UUID - 文字未満の文字列表現であり、 dbhash はデータベース名のハッシュです。次を使用して、トップレベルグループの UUID を取得できます。

```
select node_group_uuid from bdr.node_group where node_group_parent_id=0;
```

dbhash は、データベース名のハッシュです。次を使用してハッシュを取得できます。

```
select to_hex(hashtext(pgddb));
```

targetnodeuuid は、ターゲットノードの UUID の文字列表現です- 文字未満です。次を使用してターゲットノードの UUID を取得できます。

```
select node_uuid from bdr.node where node_name=<target_node_name>;
```

完全なグループスロット名は、ファンクション *bdr.local_group_slot_name()* によって返されます。

**警告**

これらのスロットはドロップしないでください。 PGD はそれらを作成および管理し、必要に応じて削除します。

- bdr_ スロットで始まるスロットに直接触れないでください。
- スロット名を接頭辞 bdr_ で始めないでください。

グループスロットは、ノードからレプリケートされている PGD クラスター内のノードのレプリケーションの進行状況を追跡するために使用されます。PGD クラスター内の各ノードには、独自のグループスロットがあり、そのノードからのレプリケーションの進行状況を追跡するために使用されます。

グループスロットは次の目的で使用されます。

- 既存のノードをすべて起動して実行せずに、新しいノードを PGD グループに参加させますが、ノードの大部分は起動している必要があります。参加中にダウンしたノードが再び複製を開始した場合、このプロセスではデータの損失は発生しません。
- 一部のノードが分離されたノードに完全に追いついていない場合でも、クラスターからノードを一貫して分離します。
- フリーズポイントを抑えて、いくつかの競合の見逃しを回避します。
- タイムスタンプベースのスナップショットの履歴スナップショットを保持します。

グループスロットは通常非アクティブで、他のノードからの Raft 進行状況メッセージに応答して定期的にのみ早送りされます。

**警告**

グループスロットをドロップしないでください。通常は非アクティブですが、 EDB Postgres Distributed クラスターの適切なオペレーションには依然として重要です。これをドロップすると、PGD の機能の一部またはすべてが動作を停止するか、誤った結果が発生する可能性があります。

#### その他のスロット名

PGD 内の他の機能は、レプリケーションスロットを利用します。

たとえば、ノードがグループに追加されると、そのノードのスロットが作成され、レプリケーションプロセスの進行状況を追跡します。

このスロットの名前は `bdr_node_<targetnodeuuid>_<dbhash>_tmp` です。

PGD の分析およびデコード機能用に作成されたスロットもあります。これらのスロットには次の名前があります。

| Slot type | Slot name |
|---|---|
| Forwarding slot, leader-to-leader slot | *bdr_node_<targetnodeuuid>_<originidhex>_<dbhash>* |
| Analytics slot | *bdr_analytics_<groupuuid>_<dbhash>* |
| Decoding slot | *bdr_decoder_<topgroupuuid>_<dbhash>* |

### 6.3.14 Connection Manager

PGD 6.0 では、 PGD 5 のプロキシソリューションを、バックグラウンドワーカーを使用して緊密に統合されたアプローチで置き換える新しい接続マネージャーを導入し、PGD の読み書き、読み取り専用、および http ステータスネットワークインターフェイスを公開します。

- *Overview of durability options* は、Connection Manager の新機能と利点をカバーしています。
- *Connection Manager Authentication* は、 Connection Manager での認証がどのように動作するかについて説明します。
- *Configuring Connection Manager* は、使用可能な構成オプションとその設定方法について詳しく説明します。
- *Load Balancing with Connection Manager* Connection Manager でロードバランシングを使用する方法。
- *Monitoring the Connection Manager* は、モニタリングに使用できるテーブルと HTTP エンドポイントをカバーします。

#### Connection Manager overview

##### 接続マネージャーについて

Connection Manager は、 EDB Postgres Distributed PGD 6.0 の新しいバックグラウンドワーカーであり、クライアントアプリケーションの単一のエントリポイントを提供することにより、PGD クラスターへの接続プロセスを簡素化します。PGD 5.x プロキシソリューションを、PGD の読み書き、読み取り専用、および HTTP ステータスネットワークインターフェイスを公開する緊密に統合されたアプローチに置き換えます。

Connection Manager は PGD に完全に統合されており、既存の PGD アーキテクチャとシームレスに動作するように設計されています。すべての PGD データノードには、着信接続をリッスンし、クラスター内の適切なノード、特にクラスター内の現在の書き込みリーダーにルーティングする Connection Manager インスタンスがあります。また、クラスターからデータを読み取ることのみが必要なアプリケーションに読み取り専用インターフェイスも提供します。

### 接続マネージャーを使用する

接続マネージャーは、デフォルトで Postgres サーバーの構成に従います。読み取り/書き込みポート、読み取り専用ポート、および HTTP ポートの 3 つのポートがあります。読み取り/書き込みポートは書き込み操作に使用され、読み取り専用ポートは読み取り操作に使用されます。HTTP ポートは、監視と管理の目的で使用されます。

読み取り/書き込みポートは、デフォルトで、Postgres ポート+1000 通常は 6432 に設定されます。読み取り専用ポートは、Postgres ポート+1001 通常は 6433 に設定されます。HTTP ポートは Postgres ポート+1002 通常は 6434 に設定されます。

接続マネージャーを使用するには、データノードで実行されている接続マネージャーインスタンスの読み取り書き込みまたは読み取り専用ポートに接続するようにクライアントアプリケーションを構成する必要があります。接続マネージャーは、接続をクラスター内の適切なノードにルーティングします。

接続マネージャーはロードバランサーの代替ではないことに注意してください。ロードバランサーと連携して動作し、PGD クラスターへの接続を管理するための完全なソリューションを提供するように設計されています。接続マネージャーは、PGD クラスターへの接続を管理するシンプルで効率的な方法を提供し、ロードバランサーはロードバランシングやフェイルオーバーなどの追加機能を提供します。詳細については、*Load Balancing with Connection Manager* を参照してください。

### 読み取り専用接続

接続マネージャーが提供する読み取り専用ポートにクライアントを接続すると、*SET TRANSACTION READ ONLY* を使用するのと同様の方法で、その接続が読み取り専用操作に制限されますが、読み取り/書き込みに変更できない点が異なります。*transaction_read_only* GUC は、これらの接続で on を正しくレポートします。

### TLS と認証

接続マネージャーは、TLS 終端と事前認証を実行します。これらの構成は Postgres から直接取得されます-pg_hba.conf とサーバーキー構成は透過的に使用されます。詳細については、*Connection Manager Authentication* を参照してください。

## Connection Manager Authentication

Connection Manager の認証は、Postgres 独自の pg_hba.conf ファイルを介して構成されます。 Connection Manager は、Postgres と同じ認証方法を使用します。

### Connection Manager 接続タイプ

接続マネージャーは、pg_hba.conf で次の接続タイプをサポートしています。

- `host` - TCP/IP 接続
- `hostssl` - SSL を使用した TCP/IP 接続
- `hostnossl` - SSL を使用しない TCP/IP 接続

### Connection Manager の認証方法

接続マネージャーは、pg_hba.conf で次の認証方法をサポートしています。

- `trust` - 認証なし
- `reject` - 接続を拒否する
- `md5` - MD5 パスワード認証
- `scram-sha-256` - SCRAM-SHA-256 パスワード認証
- `cert` - SSL 証明書認証
- `pam` - プラグ可能認証モジュール PAM 認証

### Connection Manager の認証オプション

接続マネージャーは、 pg_hba.conf の `user` および `database` フィールドの正規表現マッチングもサポートしています。これにより、ユーザー名とデータベース名を照合するためのパターンを指定できるため、複数のユーザーとデータベースの認証の管理が簡単になります。

グループメンバーシップチェックもサポートされています。これにより、各ユーザーを個別に指定するのではなく、データベースに接続できるユーザーのグループを指定できます。

### サポートされていない pg_hba.conf ルール

ルールが接続マネージャーでサポートされていない場合、警告としてログに記録され、無視されます。

## Configuring Connection Manager

### 接続マネージャーの構成

Connection Manager は、ノードがメンバーであるグループの PGD Group オプションから構成を取得します。

これらは、bdr.alter_node_group_option コマンド、または pgd group set-option コマンドを使用して構成できます。

接続マネージャーの構成には、次のオプションが使用できます。

| Option | Default | Description |
|---|---|---|
| listen_address | Postgres's listen address | クライアント接続をリッスンするローカルアドレス |
| read_write_port | Postgres's port + 1000 (usually 6432) | 読み取り/書き込み接続をリッスンするポート |
| read_only_port | read_write_port + 1 (usually 6433) | 読み取り専用接続をリッスンするポート |
| http_port | read_write_port + 2 (usually 6434) | REST API 呼び出しをリッスンする http ポート統合の目的 |

次のページに続く

表 6 – 前のページからの続き

| Option | Default | Description |
|---|---|---|
| use_https | | http リスナーが HTTPS を使用するかどうか。有効になっている場合、サーバー証明書は TLS に使用されます |
| read_write_max_client_cor | max_connection | 許可される最大読み取り/書き込みクライアント接続、デフォルトは max_connections |
| read_write_max_server_co | max_connections | サーバーに対して開かれる最大読み取り/書き込み接続 |
| read_only_max_client_con | max_connections | 許可される最大読み取り専用クライアント接続 |
| read_only_max_server_cor | max_connections | サーバーに対して開かれる読み取り専用接続の最大数 |
| read_write_consensus_time | 0 (immediate action) | 読み取り/書き込み接続が受け入れられなくなるまでにコンセンサスが失われるまで待機する時間 |
| read_only_consensus_time | 0 (immediate action) | 読み取り専用接続が受け入れられなくなるまでに、コンセンサス損失を待機する時間。 |

### Load Balancing with Connection Manager

外部ロードバランサーを使用して、複数の Connection Manager インスタンスにトラフィックを分散できます。接続マネージャーポートを他の Postgres ポートと同様に扱いますが、書き込みトラフィックには読み取り書き込みポートを使用し、読み取りトラフィックには読み取り専用ポートを使用します。

ロードバランサーによっては、構成方法について多くのオプションがあります。

### Connection Manager ルーティング

Connection Manager でトラフィックを書き込みリーダーにルーティングする場合は、Connection Manager の読み取り/書き込みポートに接続します。接続マネージャーは、ノードがメンバーであるノードグループの書き込みリーダーにトラフィックをルーティングします。

### HAProxy の例 Connection Manager ルーティング

HAProxy では、次の例のような構成を使用して、それぞれが異なるホストとポートで実行されている 3 つの Connection Manager インスタンス間でロードバランシングができます。haproxy ポート 5010 への読み取り書き込みトラフィックは、いずれかのノード上の Connection Manager の読み取り書き込みポートデフォルト 6432 に転送されます。haproxy ポート 5011 への読み取り専用トラフィックは、Connection Manager の読み取り専用ポートデフォルト 6433 に転送されます。ポート 5011 は読み取り専用トラフィックに使用され、ラウンドロビン方式でノード全体に分散されます。

```
global
    maxconn 100

defaults
    log     global
    mode    tcp
    retries 2
    timeout client 30m
```

(次のページに続く)

(前のページからの続き)

```
    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
```

### ダイレクトルーティング

接続マネージャーをバイパスして、クラスター内のノードに直接接続することができます。このシナリオで接続マネージャーを活用するには、その HTTP エンドポイントを参照して、クラスター内のノードに関する情報を参照してください。どのノードが書き込みリーダーであるかなど。次に、そのノードに直接接続します。`/node/is-read-write` エンドポイントを使用して、ノードが書き込みリーダーであるかどうかを確認し、`/node/is-read-only` エンドポイントを使用して読み取り専用プールに関する情報を取得できます。

このようにロードバランサーを構成する主な理由は、サーバーで TLS 終端と認証が発生する従来の Postgres 接続に似たレイヤー 4 TCP プロキシが必要な場合です。接続マネージャーは、プロキシレベル、つまり接続マネージャー内で TLS 終端と認証を処理するレイヤー 7 プロキシを提供します。

このアプローチの欠点は、接続マネージャーの機能、特に読み取り専用プールへの読み取り専用接続を強制する機能の恩恵を受けられないことです。接続は、書き込みリーダーではなく、非書き込みリーダーノードに転送されます。

使用可能なエンドポイントの詳細については、*Monitoring Connection Manager* を参照してください。

### HAProxy の例 ダイレクトルーティング

この例は、HAProxy を使用して、正常性ステータスに基づいてトラフィックをノードに直接ルーティングする構成を示しています。構成は、/node/is-read-write および/node/is-read-only エンドポイントを照会することによりノードの状態を確認し、それに応じてトラフィックをルーティングします。前の例と同様に、この例では、ポート 5010 の読み取り書き込みトラフィックを書き込みリーダーであるノードにルーティングし、ポート 5011 の読み取り専用トラフィックを読み取り専用のノードにルーティングします。

```
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
    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
```

この構成は、 /node/is-read-write および/node/is-read-only エンドポイントを照会することにより、

ノードの状態を確認します。正常で、それぞれ読み取り書き込みまたは読み取り専用操作に使用可能なノードにのみトラフィックをルーティングします。

### Monitoring the Connection Manager

SQL クエリーと HTTP エンドポイントを介して、接続マネージャーとその接続のステータスを表示できます。

#### 使用可能な SQL テーブルとビュー

接続マネージャーは、接続マネージャーとその接続のステータスを監視するために使用できる多数のテーブルとビューを提供します。これらには次のものが含まれます。

- bdr.stat_activity — これは、`connection_manager_client_addr` および `connection_manager_client_port` に関する追加列で強化された `pg_stat_activity` からの情報であり、接続が接続マネージャーを介して行われたものであり、読み取り専用ポートを介して接続された場合は `session_read_only` です。
- bdr.stat_connection_manager —これは、接続マネージャーのステータスに関する統計を提供するビューです。
- bdr.stat_connection_manager_connections —これは、接続マネージャーの接続に関する統計を提供するビューです。
- bdr.stat_connection_manager_node_stats —これは、各データノードの接続マネージャーに関する統計を提供するビューです。
- bdr.stat_connection_manager_hba_file_rules —これは、接続マネージャーのどの HBA ファイルルールがこのノードで使用されているかを示すビューです。

#### 使用可能な HTTP / HTTPS エンドポイント

接続マネージャーは、HTTP API を介して監視できます。

true/false を返すエンドポイントは、true の場合 200 ステータスコード、false の場合 503 ステータスコードも結果ます。

次のエンドポイントが使用できます。

| Endpoint | Description |
|---|---|
| /connection/is-live | 接続マネージャーはライブリッスンしていますか、常に「true」を返します。マネージャーが実行されていない場合、クライアントは単に接続/URL を開くことに失敗します |
| /connection/is-ready | 接続マネージャーの準備ができていますか、true(200)/false(503) を返します |
| /node/is-read-write | この PGD ノードは、接続マネージャーではなく、PGD ノード自分自身、読み取り書き込みノードですか、書き込みリーダーですか、true(200)/false(503) を返します |

次のページに続く

表 7 – 前のページからの続き

| Endpoint | Description |
|---|---|
| /node/is-read-only | この PGD ノードは、接続マネージャーではなく、PGD ノード自分自身であり、読み取り専用ノード書き込みリーダーではなく、true(200)/false(503) ノードを返します |
| /group/read-write-info | 接続マネージャーのこのインスタンスの読み取り/書き込みプールに関する情報を返します。ノード ID、ノード名、ノードホスト、ノードポート、ノード dbname を含む JSON 形式のプール内のノードのリスト。読み取り/書き込みプールの場合、プールには 1 つのエントリのみが含まれます。 |
| /group/read-only-info | 接続マネージャーのこのインスタンスの読み取り専用プールに関する情報を返します。ノード ID、ノード名、ノードホスト、ノードポート、ノード dbname を含む JSON 形式のプール内のノードのリスト。 |

以下は、`/group/read-write-info` エンドポイントからの応答本文の例です。

```
[
  {
    "id": 683485707,
    "name": "node-1",
    "host": "host-1",
    "port": 5432,
    "dbname": "pgddb"
  }
]
```

#### ロギング

すべての Connection Manager ログメッセージは PostgreSQL ログに書き込まれます。

log_line_prefix の%r および%h エスケープシーケンスの動作が、それぞれ「proxy_address/client_address」および「proxy_port/client_port」をログに記録するように変更されました。

これは、使用するサーバー接続の GUC をプロキシ設定することにより実現されます。ユーザーはこの GUC をオーバーライドできるため、client_address から派生したセキュリティコンテキストは、完全なセッションログを参照して検証する必要があります。

### 6.3.15 Postgres configuration

いくつかの Postgres 構成パラメーターは PGD ノードに影響を与えます。ノードごとにこれらのパラメーターを異なるように設定できますが、一般的にはお勧めしません。

PGD 独自の設定については、 *PGD settings reference* を参照してください。

### Postgres 設定

PGD を正しく実行するには、次の Postgres 設定が必要です。

- `wal_level` — PGD は論理デコードに依存するため、 `logical` に設定する必要があります。
- `shared_preload_libraries` —拡張機能を有効にするには、`bdr` を含める必要があります。他のほとんどの拡張機能は、コンマ区切りリストの `bdr` エントリーの前後に表示できます。その 1 つの例外は `pgaudit` であり、これはリストの `bdr` より前に表示する必要があります。また、このリストには `pglogical` を含めないでください。
- `track_commit_timestamp` —競合する各行のタイムスタンプを取得するには、競合を解決するために on に設定する必要があります。

PGD では、これらの PostgreSQL 設定を適切な値に設定する必要があります。これは、クラスターのサイズとスケールによって異なります。

- `logical_decoding_work_mem` —論理デコードによって使用されるメモリバッファサイズ。このサイズよりも大きいトランザクションはバッファをオーバーフローし、ローカルディスクに一時的に保存されます。デフォルトは 64MB ですが、かなり高く設定できます。
- `max_worker_processes` —PGD はレプリケーションとメンテナンスタスクにバックグラウンドワーカーを使用するため、正常に動作するには十分なワーカースロットが必要です。各データベースのワーカーの正しい最小数の式は、次の値を合計することです。
  - PostgreSQL インスタンスごとに 1 つ
  - そのインスタンスのデータベースごとに 1 つ
  - PGD 対応データベースごとに 4 つ
  - PGD グループのピアノードごとに 1 つ
  - ピアノードの数倍 (ライターの数 bdr.num_writers) プラス 1 ノードが PGD グループから削除されるときに、一時的により多くのワーカープロセスが必要になる場合があります。
- `max_wal_senders` —各ピアノードに 2 つ必要です。
- `max_replication_slots` —各ピアノードに 2 つ必要です。
- `wal_sender_timeout` および `wal_receiver_timeout` —通常はデフォルトの 1 分で十分ですが、大規模なトランザクションの場合、この時間よりも長く処理が必要になる場合があります。WAL 送信者は、待機しているレプリケーション接続に送信する前に、トランザクションのフルサイズを処理する必要があるため、Postgres はそれをタイムアウトとして認識できます。問題が実際に大規模なトランザクションによるものである場合は、wal_sender_timeout を `3600s` 以上のような高い値に引き上げ、サーバーをリロードすることにより問題が解決される可能性があります。さらに、この設定は、ノードが CAMO パートナーを切断または再接続したとみなすまでの時間を決定します。詳細は、*CAMO failure scenarios* を参照してください。

N 個のピアノードを持つグループの通常の実行では、PGD には N 個のスロットと WAL 送信機が必要です。同期中に、PGD は別の N-1 スロットと WAL センダーを一時的に使用するため、この時折のピーク要求に十分な高いパラメーターを設定するように注意してください。

Parallel apply がオンの場合、スロット数はフォーミュラ*ライターから N スロットに増やす必要があります。これは、`max_replication_slots` もレプリケーションオリジンの最大数を設定し、Parallel apply の

一部の機能はライタごとに追加のオリジンを使用するためです。

*Decoding worker* が有効になっている場合、このプロセスには PGD グループごとに 1 つの追加のレプリケーションスロットが必要です。

`max_worker_processes`、`max_wal_senders`、および `max_replication_slots` パラメーターを変更するには、ローカルノードを再起動する必要があります。

古い同期レプリケーションモードは、次のパラメーターを使用してサポートされています。詳細と制限については、**コミットスコープ** を参照してください。

- `synchronous_commit` および `synchronous_standby_names` —PGD レプリケーションの耐久性とパフォーマンスに影響します。physical replication と同様の方法。

#### 準備されたトランザクションの最大数

#### max_prepared_transactions

明示的な 2 フェーズコミット、CAMO、または Eager トランザクションのため、クラスター全体で同時に準備されたトランザクションの最大数に対処するために十分に高く設定する必要があります。制限を超過すると、ノードはローカル 2 フェーズコミットまたは CAMO トランザクションを実行できず、クラスター上のすべての Eager トランザクションを防止します。このパラメーターは、Postgres サーバーの起動時にのみ設定できます。

### 6.3.16 Backup and recovery

PGD は、分散された高可用システムであるように設計されています。クラスターの 1 つ以上のノードが失われた場合、それらを置き換える最良の方法は、残りのノードから直接新しいノードのクローンを作成することです。

PGD におけるバックアップとリカバリーの役割は、次の状況などのディザスターリカバリー DR を提供することです。

- クラスター内のすべてのノードの損失
- データの破損、アプリケーションエラー、またはセキュリティ侵害の結果、マルチプルのノードにわたる重大な修正不可能なデータの破損

#### バックアップ

#### pg_dump

pg_dump は、 **論理バックアップ**と呼ばれることもあります、通常は PGD で使用できます。

pg_dump は、ローカルシーケンスとグローバルシーケンスの両方をローカルシーケンスであるかのようにダンプします。この動作は意図的なもので、PGD スキーマをダンプして他の PostgreSQL データベースに移植できるようにします。これは、ダンプ時にシーケンス種類のメタデータが失われることを意味するため、リストアはすべてのシーケンス種類をリストア時に `bdr.default_sequence_kind` の値に効果的にリセットします。

シーケンスごとに正確なシーケンス種類をリセットする復元後スクリプトを作成するには、次のような SQL スクリプトを使用することができます。

```
SELECT SELECT bdr.alter_sequence_set_kind(||
        nspname||.||relname||,||seqkind||);
FROM bdr.sequences
WHERE seqkind != local;
```

bdr.crdt_raw_value = on を使用して pg_dump を実行した場合、bdr.crdt_raw_value = on でのみダンプをリロードできます。

テクニカルサポートでは、PGD のバックアップとリカバリーに物理的バックアップ技術を使用することをお勧めします。

### 物理的バックアップ

Barman などの標準の PostgreSQL ソフトウェアを使用して、 EDB Postgres 分散クラスター内のノードの物理バックアップを取得できます。

PostgreSQL ノードに適用されるのと同じ手順を使用して、PGD ノードの物理バックアップを実行できます。PGD ノードは、 BDR 拡張機能を実行する単なる PostgreSQL ノードです。

PostgreSQL バックアップ技術を PGD に適用するときは、次の特定の点を考慮してください。

- PGD は単一のデータベースのレベルで動作しますが、物理バックアップにはインスタンス内のすべてのデータベースが含まれます。データベースを簡単にバックアップおよび復元できるように計画します。
- バックアップは、1 つのノードのみのコピーを作成します。最も単純なケースでは、すべてのノードにすべてのデータのコピーがあるため、すべてのデータをキャプチャするには 1 つのノードのみをバックアップする必要があります。ただし、その単一のコピーを含むサイトがダウンした場合、PGD の目標は満たされないため、最小はサイトごとに少なくとも 1 つのノードバックアップ多数のコピーなど。
- ただし、各ノードにはレプリケーションされていないローカルデータがある場合、またはレプリケーションセットの定義が複雑な場合があるため、すべてのノードがすべてのレプリケーションセットをサブスクライブしない場合があります。これらの場合、バックアップ計画には、レプリケートされていないローカルデータと、各レプリケーションセットをサブスクライブする少なくとも 1 つのノードのバックアップをバックアップする方法に関する計画も含める必要があります。

### 最終的な一貫性

EDB Postgres Distributed クラスター内のノードは **結果的に一貫性** ですが、**完全に一貫性**ではありません。特定のノードの物理バックアップは、そのノードが実際に想定する状態に制限されたポイントインタイムリカバリ機能を提供します。

次の例は、同じ EDB Postgres 分散クラスター内の 2 つのノードがどのようにして同じ状態のシーケンスを通過しないか、通常は通過しないかを示しています。

2 つのノード N1 および N2 を含み、最初は S 状態にあるクラスターを検討します。トランザクション W1 がノード N1 に適用され、同時に競合しないトランザクション W2 がノード N2 に適用される場合、ノード N1 は次の状態を経ます。

```
(N1)   S  -->  S + W1  -->  S + W1 + W2
```

ノード N2 は、次の状態を経ます。

```
(N2)   S  -->  S + W2  -->  S + W1 + W2
```

つまり、ノード N1 は決して*状態 S + W2 を引き受けることはなく、ノード N2 も同様に状態 S + W1 を引き受けることはありません。ただし、両方のノードは同じ状態 S + W1 + W2 になります。この状況を考慮すると、バックアップ戦略の決定方法に影響を与える可能性があります。

### ポイントインタイムリカバリー PITR

前の例では、変更も時間的に一貫していないことを示しました。 W1 と W2 は両方とも T1 で発生しますが、変更 W1 は T2 まで N2 に適用されません。

PostgreSQL PITR は、単一のマスターから COMMIT オーダーで変更が到着するという想定に基づいて設計されています。したがって、PITR は、特定の時点 PIT に到達するまで変更をスキャンすることにより可能です。このスキームを使用すると、T1 などの観点から 1 つのノードを単一の PIT に復元できます。ただし、その状態には、その時間近くにコミットしたがノードにまだ到着していない他のノードからの他のデータは含まれません。その結果、回復は部分的に一貫していない、または少なくとも 1 つのレプリケーションオリジンのみについて一貫していると考えられる場合があります。

PostgreSQL PITR では、標準の構文を使用できます。

```
recovery_target_time = T1
```

PGD では、マルチプルのマスターからの変更が可能で、すべては 1 つのノードの WAL ログに記録され、レプリケーションオリジン識別子を使用して個別に識別されます。

PGD は、すべてまたは一部のレプリケーションオリジンの特定の時点への PITR を可能にし、ノードのすべてのサブセットにわたって完全に一貫した観点を提供します。

したがって、マルチオリジンの場合、WAL ストリームを、複数のストリームがすべて 1 つの大きなストリームに混合されたものとして表示できます。PIT はまだ 1 つだけですが、それは各原点の異なるポイントとして個別に到達されます。

WAL ストリームは、要求されたオリジンが PIT を見つけるまで読み取られます。すべての変更は、その時点までに適用されますが、オリジンの PIT に到達した後、トランザクションレコードはオリジンのコミット済みとしてマークされません。

WAL には 1 つの LSN「停止ポイント」がありますが、シングルオリジン PITR の場合と同様に、単一のタイムスタンプが一貫して適用されます。

定義された PIT に到達すると、必要に応じてリカバリーを続行できるように新しい PIT が設定される場合があります。

目的の停止ポイントに到達した後、復旧したサーバーが昇格する場合は、最初にシャットダウンします。pg_resetwal を使用して、このサーバーのタイムラインで使用されるよりも高い LSN 値に LSN を移動します。このアプローチでは、論理デコードによって生成される重複 LSN がないことが保証されます。

この特定の例では、N1 は T1 に復元されます。また、後まで N1 に適用されなかった場合でも、 T1 によってコミットされた他のノードからの変更も含まれます。

マルチオリジン PITR を要求するには、 postgresql.conf ファイルの標準の構文を使用します。

```
recovery_target_time = T1
```

2 つの方法のいずれかで T1 に復元されるレプリケーションオリジンのリストを指定する必要があります。新しいパラメーター recovery_target_origins を介して別の multi_recovery.conf ファイルを使用できます。

```
recovery_target_origins = *
```

または、オリジンのサブセットを recovery_target_origins のリストとして指定できます。

```
recovery_target_origins = 1,3
```

指定された recovery_target_time へのローカル WAL アクティビティのリカバリーは、常に暗黙的に実行されます。recovery_target_origins で指定されていないオリジンの場合、recovery_target_origins で記載されているリストの目標が達成される時期に応じて、任意の時点でリカバリーを停止できます。

multi_recovery.conf ファイルが存在しない場合、リカバリーは、COMMIT オーダーで単一のマスターから到着する変更を想定して設計された元の PostgreSQL PITR 動作がデフォルトになります。

> **注釈**
>
> この機能は、 EDB Postgres Extended でのみ使用できます。 Barman は`multi_recovery.conf` ファイルを作成しません。

### リストア

標準の PostgreSQL ノードと同じ手順で物理バックアップを取ることができますが、PGD ノードの物理バックアップを復元するのは少し複雑です。

#### EDB Postgres Distributed クラスター障害またはバックアップからの新しいクラスターのシード

物理バックアップを復元するための最も一般的なユースケースには、データセンターに障害が発生した場合など、クラスター内のすべての PGD ノードの障害または交換が含まれます。

また、この手順を実行して、EDB Postgres 分散クラスターの現在のコンテンツのクローンを作成して、QA または開発インスタンスをシードすることもできます。

その場合、単一の PGD ノードとオプションで WAL アーカイブの物理バックアップに基づいて、PGD 機能を復元できます。

- いくつかの PGD ノードがまだライブで実行されている場合は、PGD ノードを復元したホストをフェンスオフして、残りの PGD ノードに接続できないようにします。この実践により、新しいノードが既存のクラスターを混乱させないようになります。
- PGD ノードの 1 つの物理バックアップから単一の PostgreSQL ノードを復元します。

- バックアップに関連付けられている WAL アーカイブがある場合は、適切な `postgresql.conf` を作成し、リカバリーで PostgreSQL を起動して最新の状態まで再生します。必要に応じて、ここで代替の `recovery_target` を指定できます。
- 復元されたノードを起動するか、スタンバイ復旧している場合は読み取り/書き込みに昇格させます。生き残ったノードからフェンスしてください。
- 物理バックアップに含まれる残りの PGD メタデータをクリーンアップします。
- PostgreSQL インスタンスを完全に停止して再起動します。
- `bdr.join_node_group()` ファンクション呼び出しに基づいた標準プロシージャで PGD ノードをさらに追加します。

### PGD メタデータのクリーンアップ

残った PGD メタデータをクリーンアップするには

1. bdr.drop_node を使用して PGD ノードを削除します。
2. PostgreSQL を完全に停止して再起動します重要！

### レプリケーション起点のクリーンアップ

レプリケーションオリジンはシステムカタログに永続的に記録されるため、別の手順で明示的に削除する必要があります。したがって、これらはバックアップと復元されたインスタンスに含まれます。これらは依存関係として明示的に記録されていないため、BDR 拡張機能をドロップするときに自動的に削除されません。

クラッシュセーフな方法で着信レプリケーションの進行状況を追跡するため、PGD はリモートマスターノードごとに 1 つのレプリケーションオリジンを作成します。したがって、以前のクラスターのノードごとに、これを 1 回実行します。

```
SELECT pg_replication_origin_drop(bdr_dbname_grpname_nodename);
```

次のようにして、レプリケーションオリジンをリストできます。

```
SELECT * FROM pg_replication_origin;
```

PGD によって作成されたものは、名前で簡単に認識されます。

### レプリケーションスロットのクリーンアップ

`pg_basebackup` で物理バックアップが作成された場合、レプリケーションスロットはバックアップから省略されます。

他のバックアップ方法には、レプリケーションスロットが古いまたは無効な状態で保持される場合があります。バックアップを復元したら、次のコマンドを使用してすべてのレプリケーションスロットをドロップします。

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

一部のスロットを保持する理由がある場合は、`WHERE slot_name LIKE 'bdr%'` 句を追加できますが、これはほとんど役に立ちません。

> **警告**
>
> これらのコマンドを使用して、ライブ PGD ノードでレプリケーションスロットをドロップしないでください

### 6.3.17 Monitoring

レプリケーションセットアップの監視は、システムが次のことを保証するために重要です。

- 最適なパフォーマンス
- ディスク領域が不足しない
- オペレーションを停止する可能性のある他の障害に遭遇しない

自動監視を導入して、問題が発生したときに管理者が警告を受け取り、プロアクティブなアクションを実行できるようにすることが重要です。たとえば、レプリケーションスロットが大幅に遅延し始めた場合、管理者に警告を与えることができます。

EDB は、バージョン 8.1 から PGD をサポートする Postgres Enterprise ManagerPEM を提供します。詳細については、*Monitoring EDB PostgresDistributed* を参照してください。

または、ツールまたはユーザーは、BDR 拡張機能が提供する情報ビューと機能を独自に呼び出すことができます。*Monitoring through SQL* を参照

詳細については。

#### Monitoring through SQL

EDB Postgres Distributed は、分散の性質に固有のいくつかのモニタリングおよび統計ビューを提供します。標準の Postgres モニタリングは、EDB Postgres Distributed をモニタリングするのにも役立ちます。

##### モニタリングの概要

PGD グループは、複数のサーバーで構成されており、多くの場合ノードと呼ばれます。すべてのノードを監視して、グループ全体の状態を確認します。

bdr_monitor ロールは、`bdr.monitor` ファンクションを実行して、3 つのレベルのいずれかを使用して PGD の状態の評価を提供できます。

- `OK` —多くの場合、緑色で表示されます。
- `WARNING` —多くの場合、黄色で表示されます。
- `CRITICAL` —多くの場合、赤で表示されます。
- `UNKNOWN` —認識されない状況の場合、多くの場合赤で表示されます。

PGD は、さまざまな内部メトリックの瞬間的な状態を表示する動的なカタログビューも提供します。また、ユーザーが要求した構成のデフォルトと構成の変更を保存するメタデータカタログも提供します。これらのビューとテーブルの一部は、bdr_monitor または bdr_read_all_stats からアクセスできますが、一部には、より高いセキュリティ要件が必要なユーザーまたは内部情報が含まれています。

PGD を使用すると、各ノードを個別に監視したり、単一のノードへのアクセスでグループ全体を監視したりできます。各ノードを個別に監視する場合は、各ノードに接続し、監視要求を発行します。これらの要求は他のノードを呼び出してグループレベルの情報セットをアセンブルするため、単一のノードからグループをモニタする場合は、bdr.group で始まるビューを使用します。

bdr_superuser によって bdr.run_on_all_nodes() ファンクションへのアクセスが付与されている場合、すべてのノードに独自の呼び出しを行うことができます。

### 監視ノードの参加と削除

デフォルトでは、ノード管理機能は join または part 操作が完了するまで待機します。それぞれの wait_for_completion ファンクション引数を使用して、待機をオフにできます。待機がオフになっている場合、join または part オペレーションが終了したときを確認するには、bdr.node_summary および bdr.event_summary を使用して間接的にノードの状態を確認します。

ヘルパーファンクション bdr.wait_for_join_completion() は、呼び出されると、すべての未処理のノード結合操作領域が完了するまで PostgreSQL セッションを一時停止します。

この例は、 bdr.node_summary からの SELECT クエリーの出力を示しています。これは、2 つのノードがアクティブで、別の 1 つが参加していることを示します。

```
#  SELECT node_name, interface_connstr, peer_state_name,

#      node_seq_id, node_local_dbname

#  FROM bdr.node_summary;

- [ RECORD 1 ]-----+----------------------------------------
node_name         | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name   | ACTIVE
node_seq_id       | 1
node_local_dbname | postgres
- [ RECORD 2 ]-----+----------------------------------------
node_name         | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name   | ACTIVE
node_seq_id       | 2
node_local_dbname | postgres
- [ RECORD 3 ]-----+----------------------------------------
node_name         | node3
interface_connstr | host=localhost dbname=postgres port=7434
peer_state_name   | JOINING
```

(次のページに続く)

(前のページからの続き)

```
node_seq_id       | 3
node_local_dbname | postgres
```

また、テーブル bdr.node_catchup_info は、ノードへの参加または分離に関連する可能性のあるキャッチアップ状態に関する情報を提供します。

ノードが分離されると、クラスター内の一部のノードがその分離したノードからすべてのデータを受信しない場合があります。したがって、ノードを分離すると、そのデータを既に受信し、転送できるノードから一時的なスロットが作成されます。

catchup_state は、次のいずれかです。

```
10 = setup
20 = start
30 = catchup
40 = done
```

### マネージャーワーカーの監視

マネージャーワーカーは、他のすべてのワーカーの管理を含む多くのバックグラウンドタスクを担当します。そのため、特に行き詰まっているように見える場合、それが何をしているのかを知ることが重要です。

したがって、bdr.stat_worker ビューは、マネージャーワーカーを含む PGD ワーカーのワーカーごとの統計を提供します。マネージャーワーカーがスタックしないように、実行中の現在のタスクは、“pgd manager:”の接頭辞が付いている query フィールドで報告されます。

マネージャーワーカーの worker_backend_state フィールドは、マネージャーがアイドル状態かビジー状態かを報告します。

### モニタリングルーティング

ルーティングは、シームレスなアプリケーションエクスペリエンスと競合の回避を保証するための PGD の重要な部分です。ルーティングの変更は、障害の検出を含めて、迅速に発生する必要があります。同時に、中断はできるだけ少なくしたいと考えています。また、サポートされているユースケースで、適切なロードバランシングを保証したいと考えています。

これらをすべて監視することは、問題に気づき、問題をデバッグし、より最適な構成を通知するために重要です。したがって、ルーティングに関連する監視統計には、2 つの主なビューがあります。

- bdr.stat_routing_state 接続マネージャーを使用した接続ルーティングの状態をモニタリングするための使用接続のルーティングに使用します。
- Raft リーダーの観点からのルーティング候補ノードに関する情報については bdr.stat_routing_candidate_state ビューは他のノードでは空です。

### レプリケーションピアのモニタリング

レプリケーションアクティビティの監視には、2 つのメインビューを使用します。

- bdr.node_slots 送信レプリケーションを監視するため
- bdr.subscription_summary 着信レプリケーションを監視するため

標準の PostgreSQL レプリケーション監視ビューを照会することにより、bdr.node_slots が提供する情報のほとんどを取得することもできます *pg_catalog.pg_stat_replication*

および *pg_catalog.pg_replication_slots* 。

各ノードには 1 つの PGD グループスロットがあります。これらのスロットには接続してはならず、アクティブとしてマークされることはほとんどありません。これは正常であり、何かがダウンまたは切断されていることを意味するものではありません。ノード管理の Replication slots を参照してください。

### 送信レプリケーションの監視

送信レプリケーションアクティビティの監視に別のビューを使用できます。

- bdr.node_replication_rates 送信レプリケーションを監視するため

bdr.node_replication_rates ビューは、送信レプリケーションアクティビティの全体像と、特にピアノードのキャッチアップ推定を提供します。

```
#  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
```

apply_rate は、1 秒あたりのバイト数のレートを参照します。これは、ピアがローカルノードからデータを消費している速度です。ノードがクラスターに再接続したときの replay_lag は、すぐにゼロに設定さ

れます。この情報は将来のリリースで修正される予定です。回避策として、ピアノードがローカルノードデータに追いつくまでに必要な時間を参照する catchup_interval 列を使用することをお勧めします。他のフィールドは、bdr.node_slots から利用できます。

ビュー。

管理者は、ローカルノードからの送信レプリケーションについて bdr.node_slots を照会できます。これは、現在のノードで知られているグループ内の他のすべてのノードのレプリケーションステータスと、現在のノードで PGD によって作成された追加のレプリケーションスロットに関する情報を表示します。

```
#  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
```

PGD はメッシュネットワークであるため、クラスター内の遅延の完全なビューを取得するには、参加しているすべてのノードでこのクエリを実行する必要があります。

replay_lag_bytes は、ローカルサーバーの現在の WAL 書き込み位置と、ピアノードによって再生確認された最後の位置である replay_lsn 間の WAL 位置の差を報告します。 replay_lag_size は、同じの人が判読可能な形式です。通常、WAL には、レプリケートされないが replay_lag_bytes でカウントされる大量の書き込みが含まれることを理解することが重要です。

- VACUUM アクティビティ
- インデックスの変更
- 同じノード上の他のデータベースに関連付けられた書き込み
- レプリケーションセットの一部ではないテーブルの書き込み

したがって、ここで報告されるバイト単位の遅延は、ピアノードを最新の状態にするためにワイヤで複製する必要があるデータの量ではなく、処理する必要があるサーバーサイドの WAL の量のみです。

同様に、 replay_lag は、ピアノードが追いつくまでに要した時間、または現在位置から bdr.node_slots が照会された時点の書き込み位置まで再生に要した時間を表すものではありません。ピアが最新のコミットを確認したときと現在の実測時間との間の遅延を測定します。この列は、ノードが再接続した直後にゼロに設定されるため、replay_lag_bytes と replay_lag_size または bdr.node_replication_rates の catchup_interval をモニタすることをお勧めします。

論理レプリケーションがトランザクションをストリーミングしている間、バイトと時間の両方の遅延は進行しません。コミットがレプリケートされた場合にのみ変更されます。したがって、遅延は「のこぎり波」の傾向があり、トランザクションがストリーミングされると上昇し、ピアノードがコミット、フラッシュ、確認を送信すると再び低下します。報告された LSN は、同様の理由で、スムーズに進むのではなく、「階段」を位置づけています。

レプリケーションが切断されている場合 active = 'f' 、active_pid 列は NULL 、client_addr およびアクティブな接続でのみ意味をなすその他のフィールドと同様に、NULL です。 state フィールドは'disconnected' です。_lsn フィールドは、クライアントが再生および保存したことが確認されている最後の位置であるため、confirmed_flush_lsn と同じです。 _lag フィールドは、クライアントで確認された最新のフラッシュから現在までの経過時間を示します。_lag_size および_lag_bytes フィールドは、confirmed_flush_lsn とローカルサーバーの現在の WAL 挿入位置間の距離をレポートします。

**注釈**

*restart_lsn* が他の `lsn` 列の後ろにあるのは通常です。これは、レプリケーションまたはピアノードの遅延に関する問題を示しているわけではありません。 *restart_lsn* は、PostgreSQL の内部論理デコードが中断された場合に WAL を読み取る位置です。これは、通常、まだレプリケートおよびフラッシュされていない最も古いトランザクションの位置を反映します。非常に古い `restart_lsn` は、切断後のレプリケーションの再起動を遅くし、望ましいよりも多くの WAL の保持を強制する場合がありますが、それ以外の場合は無害です。心配する場合は、非常に長時間実行されているトランザクションと忘れられた準備済みトランザクションを探してください。

### 着信レプリケーションの監視

bdr.subscription_summary ビューを照会することにより、受信レプリケーションサブスクリプションとも呼ばれますを高レベルでモニタできます。このクエリは、EDB Postgres 分散クラスター内の他のノードへの既知のサブスクリプションのリストとレプリケーションワーカーの状態を表示します。

```
#  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
```

bdr.stat_subscription を介してサブスクリプションの概要統計をモニタリングし、それぞれ bdr.stat_receiver および bdr.stat_writer を使用してサブスクリプションレプリケーションレシーバーとサブスクリプションレプリケーションライターをモニタリングすることにより、サブスクリプションをさらにモニタできます。

### LCR を使用した WAL 送信者のモニタリング

*Decoding worker* が有効になっている場合、ファンクション *bdr.wal_sender_stats()* を使用して、各 WAL センダーの現在の論理変更レコード LCR ファイルに関する情報をモニタできます。例

```
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)
```

is_using_lcr が FALSE の場合、decoder_slot_name /lcr_file_name は NULL です。これは、デコードワーカーが有効になっていない場合、または WAL 送信者が logical standby を提供している場合に当てはまります。

また、ファンクション *bdr.get_decoding_worker_stat()* を使用して、デコードワーカーに関する情報をモニタできます。例

```
postgres=# SELECT * FROM bdr.get_decoding_worker_stat();
   pid   | decoded_upto_lsn | waiting | waiting_for_lsn
- --------+------------------+---------+-----------------
 1153091 | 0/1E5EEE8        | t       | 0/1E5EF00
(1 row)
```

### PGD レプリケーションワーカーのモニタリング

すべての PGD ワーカーは、*pg_stat_activity* と同じ列と情報コンテンツがあるシステムビュー bdr.stat_activity に表示されます。したがって、このビューは、PGD システムの状態に関する次の洞察を提供します。

- 待機の理由が PGD に関連している場合、wait_event 列に強化された情報があります。
- ライタプロセスが DDL を実行している場合、またはマネージャーワーカーがアクティブな場合を除き、PGD ワーカーの query 列は空白です。この場合、 query 列のエントリには「pgd manager:」の接頭辞が付けられます。

bdr.workers ビューには、 bdr.stat_activity からは使用できない PGD ワーカー固有の詳細が表示されます。

bdr.event_summary ビューは、作業の続行に問題があるワーカーによって報告された最後のエラーがある場合を表示します。この情報は永続的なため、エラーの存在だけでなく、エラーの時間を記録することが重要です。ほとんどのエラーは一時的なもので、PGD ワーカーは失敗した操作を再試行します。

### PGD ライターのモニタリング

別のシステムビュー bdr.writers は、ライタアクティビティを監視します。このビューには、ライタワーカーの現在のステータスのみが表示されます。それには以下が含まれます

- sub_name ライタが属しているサブスクリプションを識別する
- ライタプロセスの pid
- streaming_allowed ライタが進行中のストリーミングトランザクションの適用をサポートしているかどうかを知るため
- is_streaming ライタが現在ストリーミングトランザクションを適用しているかどうかを知るため
- commit_queue_position コミットキュー内のライタの位置を確認する

PGD は、オリジンで発生したのと同じコミット順序に従って、コミット順序を尊重します。並列ライターの場合、複数のライターが異なるトランザクションを同時に適用する場合があります。commit_queue_position は、コミットする順序を示します。値 0 は、ライタが最初にコミットすることを意味します。値-1 は、コミット位置がまだ不明であることを意味します。これは、ストリーミングトランザクション、またはライタが現在トランザクションを適用していない場合に発生します。

### コミットスコープのモニタリング

コミットスコープは、耐久性と一貫性の構成フレームワークです。そのため、それらはトランザクションのパフォーマンスに影響を与えるため、それらに関する統計を取得することが重要です。さらに、障害シナリオでは、コミットスコープの構成が原因でトランザクションがスタックしているように見える場合があるため、どのコミットスコープが使用されているか、何を待機しているかなどに関する洞察が必要です。

したがって、これらの 2 つのビューは、コミットスコープに関する関連統計を示します。

- bdr.stat_commit_scope 各コミットスコープの累積統計。
- bdr.stat_commit_scope_state バックエンドプロセスによるコミットスコープの現在の使用に関する情報。

### グローバルロックの監視

グローバルロックは、現在 DDL レプリケーションにのみ使用されていますが、PGD グループ全体に存在する重量ロックです。

現在、2 種類のグローバルロックがあります。

- DDL ロック、データベース内の一時的なものではない永続的なオブジェクトつまりテーブルに対するすべての DDL 操作をシリアル化するために使用されます
- DML リレーションロック、リレーション定義を変更する DDL 操作中にリレーションへの書き込みをロックアウトするために使用されます

DDL 操作のタイプと bdr.ddl_locking 設定の値に応じて、同じトランザクションに一方または両方のエントリータイプを作成できます。

ローカルノードで保持されているグローバルロックは、bdr.global_locks ビューに表示されます。このビューには、ロックのタイプが表示されます。リレーションロックの場合、ロックされている関係、ロックを保持している PID ローカルの場合、およびロックがグローバルに付与されたかどうかを示します。グローバルアドバイザリロックの場合、lock_type 列は GLOBAL_LOCK_ADVISORY を示し、 relation 列は、ロックが取得されるアドバイザリキーを示します。

この例は、 bdr.ddl_locking = 'all' を使用して ALTER TABLE ステートメントを実行中の bdr.global_locks の出力を示しています。

```
#  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
```

ロックタイミング情報を含むすべてのフィールドの詳細については、Catalogs を参照してください。

### 競合のモニタリング

レプリケーション conflicts は、複数のノードが相互作用できる方法で同じ行に影響を与える変更を行う場合に発生する可能性があります。PGD システムをモニタして競合を特定し、可能であればアプリケーションを変更して競合を排除するか、競合の頻度を減らします。

デフォルトでは、すべての競合は bdr.conflict_history ログに記録されます。このログには、競合するデータの完全な詳細が含まれているため、行は行レベルのセキュリティで保護され、レプリケートテーブルの所有者にのみ表示されます。所有者は競合を予想し、それらを分析して、存在する場合、どれが解決すべき問題と考えられるかを確認する必要があります。

モニタリングには、ユーザーデータを含まない bdr.conflict_history_summary を使用します。この例は、効率的なクエリプランを使用して、その日に発生した競合の数をカウントするクエリを示しています。

```
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);
```

### 統計を適用する

PGD は、サブスクリプションごととテーブルごとの両方で、レプリケーションの適用に関する統計を収集します。

2 つのモニタリングビューが存在します。サブスクリプション統計の場合は bdr.stat_subscription 、リレーション統計の場合は bdr.stat_relation です。これらのビューは両方とも次を提供します。

- レプリケートされた INSERT / UPDATE / DELETE / TRUNCATE の数
- ブロックアクセスとキャッシュヒット率
- 読み取り/書き込みの合計 I/O 時間
- ファイルにストリーミングされた進行中のトランザクションの数
- ライターにストリーミングされる進行中のトランザクションの数
- コミット/中止された進行中のストリーミングトランザクションの数

リレーションのみの場合、 bdr.stat_relation には次のものも含まれます。

- リレーションのレプリケーションの処理に費やされた合計時間
- リレーションのみのロックがある場合を取得するまでの合計ロック待機時間

サブスクリプションの場合のみ、 bdr.stat_subscription には以下が含まれます。

- サブスクリプションに対してレプリケートされた COMMIT/ DDL の数
- このサブスクリプションがアップストリームに接続した回数

これらの統計の追跡は、それぞれ PGD GUC bdr.track_subscription_apply および bdr.track_relation_apply によって制御されます。

次に、これらの出力例を示します。

```
#  SELECT sub_name, nconnect, ninsert, ncommit, nupdate, ndelete, ntruncate, nddl

FROM bdr.stat_subscription;
- [ RECORD 1 ]----------------------------------
sub_name  | bdr_regression_bdrgroup_node1_node2
nconnect  | 3
ninsert   | 10
ncommit   | 5
nupdate   | 0
ndelete   | 0
ntruncate | 0
nddl      | 2
```

この場合、サブスクリプションは上流に 3 回接続し、10 行を挿入し、5 つのトランザクション内で 2 つの DDL コマンドを実行しました。

ファンクション bdr.reset_subscription_stats および bdr.reset_relation_stats を使用して、これらのビューの統計カウンターをゼロにリセットできます。

PGD は、それぞれ bdr.stat_receiver および bdr.stat_writer を使用して、各サブスクリプションのサブスクリプションレプリケーションレシーバーとサブスクリプションレプリケーションライターに関する統計も監視します。

#### 標準の PostgreSQL 統計ビュー

テーブルとインデックスの使用状況に関する統計は、通常、ダウンストリームマスターによって更新されます。これは、*autovacuum* の正しい機能のために重要です。ダウンストリームマスターにローカル書き込みがなく、統計がリセットされていない場合、これらの 2 つのビューは、アップストリームとダウンストリームで対応する結果を表示します。

- `pg_stat_user_tables`
- `pg_statio_user_tables`

> **注釈**
>
> 上流のテーブル統計がダウンストリームのテーブル統計と*同様*であることを必ずしも期待しているわけではありません。私たちはそれらが同じ量だけ*変更*されることを期待しています。統計が 100 万の挿入と 100 万の更新を示すテーブルの例を考えます。新しいノードが PGD グループに参加すると、新しいノードの同じテーブルの統計には、1M の挿入とゼロの更新が示されます。ただし、その瞬間から、一方の側ですべての変更がもう一方の側にレプリケートされるため、上流と下流のテーブル統計は同じ量だけ変更されます。

インデックスは変更を適用するために使用されるため、ダウンストリーム側の識別インデックスは、非識別インデックスよりも `UPDATE` および `DELETE` を実行するワークロードで頻繁に使用される場合があります。

ビルトインインデックスモニタリングビューは次のとおりです。

- `pg_stat_user_indexes`

- pg_statio_user_indexes

これらのビューはすべて、 *PostgreSQL documentation on the statistics views* で詳細に説明されています。

### PGD バージョンのモニタリング

PGD では、同じクラスター内のノード全体で異なる Postgres バージョンと異なる BDR 拡張バージョンを実行できます。この機能は、アップグレードに役立ちます。

ビュー bdr.group_versions_details は、ファンクション *bdr.run_on_all_nodes()* を使用して、すべてのノードから Postgres および BDR 拡張バージョンを同時に取得します。例

```
pgddb=# SELECT node_name, postgres_version, bdr_version
        FROM bdr.group_versions_details;
 node_name | postgres_version | bdr_version
- ----------+------------------+-------------
 node1     | 15.2.0           | 5.0.0
 node2     | 15.2.0           | 5.0.0
```

推奨されるセットアップは、すべてのノードでできるだけ早く同じ最新のバージョンを実行するようにすることです。クラスターは、BDR 拡張機能のさまざまなバージョンを長時間実行しないことをお勧めします。

モニタリングのために、次のアラートレベルをお勧めします。

- status = UNKNOWN、message =このノードは PGD グループの一部ではありません
- status = OK、message =すべてのノードが同じ PGD バージョンを実行しています
- status = WARNING、message =アクセスできないノードが少なくとも 1 つあります
- status = WARNING、message =他のノードと比較して、異なる PGD バージョンを実行しているノードがあります

説明した動作は *bdr.monitor_group_versions()* ファンクションに実装されています。これは、ビュー bdr.group_version_details から返された PGD バージョン情報を使用して、クラスター全体のバージョンチェックを提供します。例

```
pgddb=# SELECT * FROM bdr.monitor_group_versions();
 status |               message
- -------+----------------------------------------
 OK     | All nodes are running same BDR versions
```

### Monitoring Raft コンセンサス

Raft コンセンサスは、常にクラスター全体で動作している必要があります。Raft コンセンサスが機能せずに EDB Postgres 分散クラスターを実行した場合の影響は次のとおりです。

- PGD データ変更のレプリケーションは、引き続き正常に動作する場合があります。
- グローバル DDL/DML ロックは機能しません。
- Galloc シーケンスは、最終的にチャンクを使い果たします。

- Eager Replication が機能しない。
- クラスターメンテナンス操作ノード参加、部分ノード、プロモートスタンバイは引き続き許可されますが、完了しない代わりにハングする場合があります。
- PGD ノード間でノードのステータスが正しく同期されない場合があります。
- PGD グループレプリケーションスロットは LSN を進歩させないため、ディスク上に WAL ファイルを保持します。

ビュー bdr.group_raft_details は、ファンクション *bdr.run_on_all_nodes()* および *bdr.get_raft_status()* を使用して、すべてのノードから同時に Raft コンセンサスステータスを取得します。例

```
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
```

これらの条件がすべて満たされる場合、Raft コンセンサスは正常に動作します。

- 有効な状態 RAFT_LEADER または RAFT_FOLLOWER がすべてのノードで定義されています。
- ノードのうちの 1 つだけが RAFT_LEADER です。
- leader_id はすべての行で同じであり、state = RAFT_LEADER が存在する行の node_id と一致する必要があります。

時々、Raft コンセンサスは新しい選挙を開始して、新しい RAFT_LEADER を定義します。選挙中に、RAFT_LEADER が存在せず、一部のノードが自分自身を RAFT_CANDIDATE と見なす中間状況が発生する場合があります。選挙全体には bdr.raft_global_election_timeout より長くかかることはありませんデフォルトでは 6 秒に設定されています。上記のクエリーが選挙中の状況を結果た場合、bdr.raft_global_election_timeout を待って、クエリーを再度実行します。bdr.raft_global_election_timeout が経過した後、リストされた条件の一部がまだ満たされていない場合、Raft コンセンサスは機能していません。

Raft コンセンサスは、単一のノードでのみ正常に動作していない可能性があります。たとえば、ノードの 1 つが現在のリーダーを認識せず、自分自身を RAFT_CANDIDATE と見なします。この場合、次のことを確認することが重要です。

- すべての PGD ノードは、レギュラー接続とレプリケーション接続の両方を介して相互にアクセスできますチェックファイル pg_hba.conf 。
- PGD バージョンはすべてのノードで同じです。
- bdr.raft_global_election_timeout はすべてのノードで同じです。

場合によっては、特にノードが地理的に離れている場合、またはネットワーク遅延が高い場合、bdr.raft_global_election_timeout のデフォルト値 6 秒では十分でない場合があります。すべてが正しいことを確認した後でも Raft コンセンサスがまだ機能していない場合は、すべてのノードで bdr.raft_global_election_timeout を 30 秒に増加することを検討します。

Raft コンセンサスがクラスター操作タスクにどのような影響を与えるか、また Raft コンセンサスがグループスロットを前進させることに直接責任があるため、モニタリングアラートレベルは次のように定義されます。

- status = UNKNOWN、message =このノードは PGD グループの一部ではありません
- status=OK、message=Raft コンセンサスは正常に動作しています
- status = WARNING、message =アクセスできないノードが少なくとも 1 つあります
- status = WARNING、 message = RAFT_CANDIDATE としてノードがあります、選択が進行中の可能性があります
- status = WARNING、message = RAFT_LEADER がありません、選挙が進行中の可能性があります
- status=CRITICAL、message=Raft コンセンサスには単一のノードがあります
- status = CRITICAL、message = RAFT_CANDIDATE としてノードがありますが、RAFT_LEADER が定義されています
- status=CRITICAL、message=RAFT_LEADER として設定されたノードとは異なるリーダーをフォローしているノードがあります。

説明した動作はファンクション *bdr.monitor_group_raft()* に実装されます。これは、ビュー bdr.group_raft_details から返された Raft コンセンサスステータス情報を使用します

クラスター全体の Raft チェックを提供します。例

```
pgddb=# SELECT * FROM bdr.monitor_group_raft();
node_group_name | status |               message
- ---------------|--------+-------------------------------------
mygroup          | OK     | Raft Consensus is working correctly
```

Raft コンセンサスの状態をより詳細に提供できるさらに 2 つのビューは、ローカルノードでの Raft コンセンサスの状態を提供する bdr.stat_raft_state と、Raft リーダー上のビューを提供する bdr.stat_raft_followers_state です。他のノード) その Raft リーダーのフォロワーの状態に関するもの。

### レプリケーションスロットの監視

各 PGD ノードは以下を保持します。

- アクティブな PGD ピアごとに 1 つのレプリケーションスロット
- 1 つのグループレプリケーションスロット

例

```
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
```

(次のページに続く)

(前のページからの続き)

```
bdr_pgddb_bdrgroup_node3 | pgddb    | t       | 0/31F4670
bdr_pgddb_bdrgroup_node4 | pgddb    | t       | 0/31F4670
```

ピアスロット名は bdr_<DATABASE>_<GROUP>_<PEER> 規則に従いますが、PGD グループスロット名は bdr_<DATABASE>_<GROUP> 規則に従います。ファンクション *bdr.local_group_slot_name()* を使用してグループスロットにアクセスできます。

ピアレプリケーションスロットは、すべてのノードで常にアクティブである必要があります。ピアレプリケーションスロットがアクティブでない場合、それは次のいずれかを意味する可能性があります。

- 対応するピアはシャットダウンしているか、アクセスできません。
- PGD レプリケーションが壊れています。

ERROR または FATAL のログファイルを grep し、すべてのノードで bdr.event_summary を確認します。根本的な原因は、たとえば、ノードの 1 つで DDL レプリケーションを無効にして、互換性のない DDL が実行されたことが考えられます。

ただし、PGD グループレプリケーションスロットは、ほとんどの場合非アクティブです。他のすべてのピアが対応するトランザクションを既に消費している場合、PGD はこのスロットを維持し、その LSN を進めます。したがって、グループスロットのステータスを監視する必要はありません。

ファンクション *bdr.monitor_local_replslots()* は、すべての PGD ノードレプリケーションスロットが予想通りに動作しているかどうかの概要を提供します。この概要は、optimized topology が有効になっているときに PGD クラスターでサブスクライバ専用グループリーダーとして動作しているサブスクライバ専用ノードでも利用できます。例

```
pgddb=# SELECT * FROM bdr.monitor_local_replslots();
 status |                    message
- -------+-------------------------------------------------
 OK     | All BDR replication slots are working correctly
```

次のステータスの概要のいずれかが返されます。

| 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 |

### トランザクションの COMMIT の監視

デフォルトでは、PGD トランザクションはローカルノードにのみコミットされます。その場合、トランザクションの COMMIT は迅速に処理されます。

PGD の コミットスコープ 機能は、特定のクエリの耐久性、一貫性、パフォーマンスのバランスを取れますさまざまな同期トランザクションコミットスコープを提供します。bdr.stat_activity カタログを調べるこ

とにより、これらのトランザクションをモニタできます。トランザクションがコミットされると、プロセスはさまざまな `wait_event` 状態を報告します。このモニタリングは、進行中のトランザクションのみをカバーし、履歴タイミング情報は提供しません。

### 6.3.18 AutoPartition in PGD

PGD AutoPartition を使用すると、テーブルを複数のパーティションに分割できます。自動パーティション管理を使用して、テーブルを大きなサイズに簡単に拡張できます。この機能は、パーティションの作成と削除の低競合ロックなどの PGD の機能を使用します。

新しいパーティションを定期的に作成し、データ保持期間の有効期限が切れたら削除できます。

PGD 管理は、主に SQL で呼び出すことができるファンクションを使用して実行します。PGD のすべてのファンクションは `bdr` スキーマで公開されています。search_path に配置しない限り、各ファンクションの名前をスキーマ修飾する必要があります。

#### パーティションの自動作成

PGD AutoPartition は *bdr.autopartition()* を使用します

テーブルの自動レンジパーティションの定義を作成または変更するファンクション。定義が存在しない場合、作成されます。それ以外の場合、後の実行で定義が変更されます。

PGD 5.5 以降の PGD AutoPartition は、基になる Postgres 機能を活用して、テーブルの残りの部分をロックせずにパーティションをアタッチまたはデタッチ/ドロップできます。5.5 より前のバージョンの PGD はこの機能をサポートしておらず、テーブルをロックします。

テーブルが RANGE パーティション化されていない場合、または複数列パーティションキーが使用されている場合、エラーが発生します。

デフォルトでは、AutoPartition はパーティションをローカルに管理します。パーティションのローカルでの管理は、パーティション化されたテーブルがレプリケートテーブルでない場合に役立ちます。その場合、すべてのノードにすべてのパーティションを必要としないか、したくない場合があります。例、ビルトイン bdr.conflict_history

テーブルはレプリケートテーブルではありません。これは AutoPartition によってローカルに管理されます。各ノードはこのテーブルのパーティションをローカルに作成し、十分に古くなると削除します。

次のことも考慮します。

- アクティビティは、エントリーが `enabled = on` とマークされている場合にのみ実行されます。
- AutoPartition によって管理されるテーブルのパーティションを手動で作成または削除しないことをお勧めします。そうすると、AutoPartition メタデータの一貫性がなくなり、障害が発生する可能性があります。

#### AutoPartition の例

日次パーティション、データを 1 か月間保持します

```
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);
```

パーティションがあと 2 つだけ残っている場合、5 つの事前パーティションを作成します。各パーティションには、10 億の注文を保持できます。

```
bdr.autopartition(Orders, 1000000000,
        partition_initial_lowerbound := 0,
        minimum_advance_partitions := 2,
        maximum_advance_partitions := 5
    );
```

### RANGE パーティションテーブル

partition_increment 値の範囲ごとに新しいパーティションが追加されます。下限と上限は partition_increment 離れています。 timestamp または date タイプのパーティションキーを持つテーブルの場合、partition_increment は interval タイプの有効な定数である必要があります。たとえば、1 Day を指定すると、新しいパーティションが毎日追加され、パーティション境界は 1 日ごとに追加されます。

パーティション列が snowflakeid 、timeshard 、または ksuuid シーケンスに接続されている場合、partition_increment を interval タイプとして指定する必要があります。それ以外の場合、パーティションキーが整数または数値の場合、partition_increment は同じデータ型の有効な定数である必要があります。たとえば、 1000000 を指定すると、100 万の値ごとに新しいパーティションが追加されます。

テーブルに既存のパーティションがない場合、指定された partition_initial_lowerbound が最初のパーティションの下限として使用されます。partition_initial_lowerbound を指定しない場合、システムはパーティション列タイプと指定された partition_increment からその値を導き出そうとします。たとえば、 partition_increment が 1 Day として指定されている場合、partition_initial_lowerbound は CURRENT DATE に設定されます。partition_increment が 1 Hour として指定される場合、partition_initial_lowerbound は現在の日付の現在時間に設定されます。後続のパーティションの境界は、partition_increment 値を使用して設定されます。

システムは常に特定の最小数の事前パーティションを持とうとします。新しいパーティションを作成するかどうかを決定するために、指定された partition_autocreate_expression を使用します。これは、チェックが実行されるたびに評価される SQL によって評価できる式です。たとえば、列タイプ date のパーティションテーブルの場合、partition_autocreate_expression が DATE_TRUNC('day',CURRENT_DATE) として指定され、partition_increment が 1 Day として指定され、minimum_advance_partitions が 2 として指定されるとします。次に、最後のパーティションの上限が DATE_TRUNC('day', CURRENT_DATE) + '2 Days'::interval 未満になるまで、新しいパーティションが作成されます。

式は、システムが新しいパーティションをチェックするたびに評価されます。

列タイプ integer のパーティションテーブルの場合、partition_autocreate_expression を SELECT max(partcol) FROM schema.partitioned_table として指定できます。次に、システムは、パーティション化された列の最大値が最後のパーティションの上限の minimum_advance_partitions * partition_increment の距離内にあるかどうかを定期的にチェックします。クエリーが効率的に実行されるように、partcol にインデックスを作成します。カラムタイプ integer、smallint、または bigint のパーティションテーブルに partition_autocreate_expression を指定しない場合、システムは max(partcol) に設定します。

data_retention_period が設定されている場合、パーティションはこの期間の後にドロップされます。ロックを最小限に抑えるために、新しいパーティションが追加されると同時にパーティションが削除されます。この値を設定しない場合、パーティションを手動で削除する必要があります。

data_retention_period パラメーターは、タイムスタンプベースおよび関連するパーティションでのみサポートされています。期間は、パーティションの上限を考慮して計算されます。上限と比較して、指定された期間が満了すると、パーティションはドロップされます。

### パーティションの自動作成の停止

*bdr.drop_autopartition()* を使用する

指定されたリレーションの自動パーティションルールを削除します。リレーションの保留中の作業項目はすべて削除され、新しい作業項目は作成されません。

### パーティションの作成を待機しています

パーティションの作成は非同期プロセスです。AutoPartition は、ローカルまたはすべてのノードでパーティションが作成されるのを待機する一連の機能を提供します。

*bdr.autopartition_wait_for_partitions()* を使用する

ローカルノードでパーティションの作成を待機します。ファンクションは、パーティションテーブル名とパーティションキー列値を取得し、その値を保持するパーティションが作成されるまで待機します。

このファンクションは、パーティションがローカルに作成されるのをのみ待機します。パーティションがリモートノードにも存在することは保証されません。

すべての PGD ノードでパーティションが作成されるのを待機するには、*bdr.autopartition_wait_for_partitions_on_all_nodes()* を使用します

ファンクション。このファンクションは、ローカルノードとすべてのリモートノードを内部的にチェックし、パーティションがどこでも作成されるまで待機します。

### パーティションの検索

*bdr.autopartition_find_partition()* を使用します

指定されたパーティションキー値のパーティションを見つけるファンクション。その値を保持するパーティションが存在しない場合、ファンクションは NULL を結果ます。それ以外の場合、パーティションの Oid を結果ます。

#### 自動パーティショニングの有効化または無効化

*bdr.autopartition_enable()* を使用する

指定されたテーブルで自動パーティション化を有効にします。自動パーティション化が既に有効になっている場合、アクションは発生しません。同様に、*bdr.autopartition_disable()* を使用します

指定されたテーブルで自動パーティショニングを無効にします。

#### パーティションの削除またはデタッチ

デフォルトでは、`data_retention_period` の有効期限が切れると、`autopartition()` によって管理されるテーブルのパーティションがデタッチおよびドロップされます。`drop_after_retention_period` パラメーターはこの動作を制御できるため、これらのパーティションを保持することを選択できます。`false` に設定すると、パーティションは親テーブルからデタッチされるだけで、ドロップされません。これにより、ユーザーは最終的に削除する前にパーティションを他のテーブルに再アタッチしたり、処理したりできる可能性があります。パラメーターのデフォルト値は `true` であるため、既存または新しい自動パーティションテーブルでは、`data_retention_period` の有効期限が切れるとパーティションはドロップされます。デタッチされたがドロップされていないパーティションを処理するには、これらのパーティションを他のテーブルに手動で再アタッチするか、ドロップします。

### 6.3.19 Commit Scopes

完全に管理および構成可能なコミットスコープは、PGD Expanded の機能です。

PGD Expanded は、デフォルトの非同期レプリケーションを補完するさまざまな同期モードを提供します。コミットスコープを使用して、これらの同期モードを構成します。コミットスコープは、PGD が同期操作を処理する方法と、システムがトランザクションがコミットされたと判断するタイミングを定義するルールです。

PGD Essential は、事前定義され変更できない限定的なコミットスコープを提供します。

#### のご紹介

- Overview は、同期コミットを議論する際に使用される概念といくつかの重要な用語を紹介します。
- Durability terminology は、レプリケーションでノードを参照する方法を含む、PGD の耐久性オプションの周辺で使用される用語をリストします。
- Commit scopes は、コミットスコープの構造と、ニーズに合わせてそれらを定義する方法をより詳細に示しています。
- Predefined commit scopes は、 PGD Essential で使用可能な事前定義されたコミットスコープをリストします。
- Origin groups は、オリジングループの概念と、コミットスコープルールを定義するときにこれらを活用する方法を紹介します。
- Commit scope rules は、コミットスコープルールの構文と定式化方法を検討します。
- Comparing durability options は、耐久性に関してコミットスコープオプションがどのように動作するかを比較します。

- Degrading commit scope rules は、より厳密な設定でタイムアウトが発生した場合に、より低い設定に正常に低下できるコミットスコープルールを設定する方法を示します。

#### コミットスコープの種類

- Synchronous Commit は、従来の同期レプリケーションと同様の方法で動作するコミットスコープメカニズムですが、コミットスコープフレームワーク内からです。
- Group Commit は、 Group Commit オプションに焦点を当てており、ノードのグループが完了したことに同意したときにトランザクションを完了として定義できます。
- CAMO は、 Commit At Most Once オプションに焦点を当てています。このオプションでは、アプリケーションが再試行する前にトランザクションがコミットされたことを確認します。これにより、コミットが多くても 1 回のみ発生することが保証されます。
- Lag Control は、最も遅いノードに応じてノードを動的に調整し、データベースノードがサービス停止になったときに同期ノードがどの程度非同期になるかを調整するコミットスコープメカニズムを調べます。

#### コミットスコープを使用した作業

- Administering は、使用中のグループコミットを使用して PGD クラスターを管理する方法を解決します。
- Legacy synchronous replication は、PGD で従来の Postgres 同期操作にアクセスする方法を示しています。
- Internal timing of operations は、古いレプリケーションと PGD の非同期および同期操作、特にトランザクションがディスクにフラッシュされるまたは可視される順序の違いを比較します。

### Overview of durability options

#### 概要

EDB Postgres DistributedPGD では、 **コミットスコープ**を使用して、耐久性、一貫性、可用性、パフォーマンスのニーズに基づいて、いくつかのレプリケーション構成から選択できます。

基本構成では、PGD は非同期レプリケーションを使用します。ただし、コミットスコープは、デフォルトとトランザクションごとの動作の両方を変更できます。

ビルトインの物理または論理レプリケーションと同じ方法で、標準の `synchronous_standby_names` を使用して古い Postgres 同期レプリケーションを構成することもできます。ただし、コミットスコープは、レプリケーション動作の柔軟性と制御を提供します。

さまざまな同期設定は、関連しているがすべて個別に実装できるアプリケーションの 3 つのプロパティに影響を与えます。

- 耐久性複数のノードに書き込むと、クラッシュ耐性が向上し、クラッシュと再起動後にデータを復旧できます。
- 可視性クライアントへのコミット確認により、データベースは、一部のノードセットでコミットされたトランザクションの即時の可視性を保証します。

- 競合処理競合はコミット後に楽観的に処理でき、コミットタイムスタンプに基づいてトランザクションがレプリケートされるときに競合が解決されます。または、プリコミット時に悲観的に処理できます。クライアントは、最終的にすべてのノードに適用されるトランザクションに依存して、さらなる競合または中止を取得して、クライアントにエラーを直接通知できます。

コミットスコープでは、トランザクションの 4 種類の制御耐久性が許可されます。

- **同期コミット**この種のコミットスコープでは、オリジンノードが大部分のノードの確認を待機する動作が可能になり、ネイティブの Postgres 同期コミットのように動作します。
- **耐久性オプショングループコミット/** *CAMO* この種のコミットスコープは、トランザクションがコミット可能であると考えられる前に、どのおよびいくつのノードがコンセンサスに到達する必要があり、レプリケーションのどのステージでコミットされたと考えられるかを制御します。このオプションを使用すると、トランザクションの可視性の順序を制御することもできます。
- *Commit At Most Once* この種のコミットスコープはグループコミットのバリアントであり、クライアントは再試行する前にトランザクションがコミットされたことを確認する責任を負います。
- *Lag Control* この種のコミットスコープは、コミットを続行する前にレプリケーションの観点からノードがどれだけ遅れていてもよいかを制御します。

同期コミット、グループコミット、および CAMO は、それぞれ、耐久性のさらなる制御のために *Degrading commit scope rules* をサポートしています。

### Durability terminology

#### 耐久性の用語

このページでは、PGD の耐久性オプションに直接関連する用語と定義をカバーします。他の用語については、Terminology を参照してください。

#### ノード

PGD ノードは、トランザクションのレプリケーション中にさまざまなロールを引き受けます。これらはトランザクションごとに暗黙的に割り当てられ、同時トランザクションの場合でも関係ありません。

- *origin* は、クライアントまたはアプリケーションからトランザクションを受信するノードです。最初にトランザクションを処理するノードであり、他の PGD ノードへのレプリケーションを開始し、確認またはエラーでクライアントに応答します。
- **オリジンノードグループ** は、オリジンを含む PGD グループです。
- **パートナー**ノードは、グループコミット要件に従ってトランザクションを確認することが想定されている PGD ノードです。
- **コミットグループ**は、コミットに関係するすべての PGD ノードのグループです。つまり、オリジンとそのすべてのパートナーノード。いくつかのまたはすべてのピアノードにすることができます。

### Commit scopes

コミットスコープにより、アプリケーションは EDB Postgres Distributed の耐久性と一貫性に関する詳細な制御を提供します。

コミットスコープは、トランザクションがコミットされるときのシステムの動作を記述する一連のルールです。実際の動作は、コミットスコープのルールがどの種類のコミットスコープを使用するかによって異なります**同期コミット** 、 **耐久性オプショングループコミット**/ *CAMO* 、 *Commit At Most Once* 、 *Lag Control* 、またはこれらの組み合わせ。

ほとんどのコミットスコープの種類はトランザクションの処理を制御しますが、Lag Control は例外で、遅いまたはキューアップされているレプリケーション操作に応じてシステムのパフォーマンスを動的に調整します。ただし、通常は他のコミットスコープ種類と組み合わせて使用されます

#### コミットスコープ構造

すべてのコミットスコープには名前 `commit_scope_name` があります。

各コミットスコープには 1 つ以上のルールがあります。

コミットスコープ内の各ルールには、コミットスコープルールを一意に識別する `origin_node_group` があります。

`origin_node_group` は PGD グループであり、トランザクションのオリジネーターであるときにこのルールを適用するノードを定義します。

最後に、これらのトランザクションに適用されるコミットスコープまたはコミットスコープ種類の組み合わせを定義するルールがあります。

したがって、コミットスコープに次のルールがある場合

Origin_node_group :=‘example_bdr_group’、rule :=‘MAJORITY (example_bdr_group) GROUP COMMIT’,

次に、ルールは、`example_bdr_group` のノードがトランザクションを発行したときに適用されます。

ルール自分自身は、指定されたグループのノードの数を指定します変更-`MAJORITY (example_bdr_group)` -その後にコミットスコープの種類自分自身-`GROUP COMMIT` が続きます。これは、 `example_bdr_group` の任意の 2 つのノードが変更を確認する必要があることを意味します変更がコミットされたと考えられる前に。

#### コミットスコープの選択方法

変更が発生すると、PGD はトランザクションまたはノードに使用するコミットスコープを検索します。

トランザクションがコミットスコープを指定している場合、そのスコープが使用されます。

指定しない場合、システムはデフォルトのコミットスコープを検索します。デフォルトのコミットスコープはグループレベルの設定です。システムはグループツリーを参照します。ノードのグループを含むグループツリーの下部から始めて上へ、default_commit_scope 設定が定義されているグループを検索します。このコミットスコープが使用されます。

default_commit_scope が見つからない場合、ノードの GUC、bdr.commit_scope が使用されます。そして、それが設定されていないか、`local` に設定されている場合、コミットスコープは適用されず、PGD の非同期レプリケーションが使用されます。

コミットスコープはローカルでなく、コミットが実行されているノードが origin_node_group に直接的または間接的に関連していない場合、使用されません。

### コミットスコープの作成

bdr.create_commit_scope を使用して、ルール例をコミットスコープに追加します。例

```
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
);
```

これにより、example_bdr_group から発生したトランザクションのルール MAJORITY (example_bdr_group) GROUP COMMIT が example_scope と呼ばれるスコープに追加されます。

example_scope に以前にルールが存在しなかった場合、このルールを追加するとスコープが存在します。

ルールが追加されるときは、origin_node_group が既に存在する必要があります。そうしない場合、追加操作全体がエラーで破棄されます。

ルールが評価されます。ルールが存在しないグループを記載しているか、グループの設定がグループのノードの他の構成設定と互換性がない場合、警告が表示されますが、ルールは追加されます。

ルールが追加されると、コミットスコープが使用できるようになります。

wait_for_ready は、ルールが関連するノードに追加されるまで、bdr.create_commit_scope() 呼び出しをブロックするかどうかを制御します。設定はデフォルトで true であり、省略できます。

### コミットスコープを使用する

この例のスコープを使用するには、トランザクション内で bdr.commit_scope を設定できます

```
BEGIN;
SET LOCAL bdr.commit_scope = example_scope;
...
COMMIT;
```

トランザクションがデータを書き込む前に、コミットスコープを設定する必要があります。

bdr.alter_node_group_option を使用して、グループまたはサブグループのデフォルトとしてコミットスコープを設定できます。

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

グループまたはサブグループのデフォルトを完全にクリアするには、default_commit_scope 値を local に設定します。

```
SELECT bdr.alter_node_group_option(
  node_group_name := example_bdr_group,
  config_key := default_commit_scope,
  config_value := local
);
```

PGD CLI を使用してこの変更を行うこともできます。

```
pgd set-group-options example-bdr-group --option default_commit_scope=example_scope
```

また、PGD CLI を使用して値を local に設定することにより、デフォルトをクリアできます。

```
pgd set-group-options example-bdr-group --option default_commit_scope=local
```

最後に、次を使用してノードのデフォルトの commit_scope を設定できます。

```
SET bdr.commit_scope = example_scope;
```

PGD のデフォルトの非同期レプリケーションを使用するには、bdr.commit_scope から local を設定します。

### Origin groups

コミットスコープのルールは、トランザクションがコミットされるノード、つまりトランザクションのオリジンとして機能するノードに依存する場合があります。そのノードが属するグループツリーの下部のグループは、トランザクションの**オリジングループ**です。これをアプリケーションに対して透過的にするために、PGD ではコミットスコープがトランザクションのオリジングループに応じて異なるルールを定義できます。

たとえば、2 つのデータセンター左側 left_dc と右側 right_dc にまたがるノードを含む EDB Postgres 分散クラスターを検討します。トップレベルの PGD ノードグループの名前が top_group であると仮定します。次のコマンドを使用して、サブグループをセットアップし、ローカルデータセンター内のすべてのノードがトランザクションを確認する必要があるコミットスコープを作成できますが、リモートから 1 つのノードのみを確認します。

```
- - create sub-groups
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
```

(次のページに続く)

(前のページからの続き)

```
);

- - 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
);
```

次に、 left_dc の一部であるノードで example_scope を使用すると、最初のスコープが使用されます。right_dc の一部であるノードで同じスコープを使用すると、2 番目のスコープが使用されます。left_dc および right_dc オリジンルールを 1 つのコミットスコープ名で組み合わせることで、アプリケーションはどちらかのデータセンターで example_scope を使用するだけで、そのデータセンターに適切な動作を取得できます。

各グループは、 bdr.alter_node_group_option 管理インターフェイスを使用して指定されたデフォルトのコミットスコープを持つこともできます。

上記のスコープを、これらのグループのノードで発生したすべてのトランザクションのデフォルトのスコープにすると、次のようになります。

```
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

ORIGIN_GROUP を使用してコミットスコープルールを作成するときに、トランザクションのオリジングループを動的に参照することもできます。

これにより、上記の example_scope のような特定のコミットスコープルールを作成でき、2 つのルールではなく 1 つのルールを指定するだけで指定がさらに簡単になります。

たとえば、再び right_dc のノードから発生したトランザクションで、トランザクションがコミットされる前に right_dc のすべてのノードで確認し、left_dc の任意の 1 で確認したいとします。また、再び left_dc で発生したトランザクションの場合、トランザクションがコミットされる前に left_dc のすべてのノードと right_dc の任意の 1 ノードで確認したいとします。上記では、example_scope を定義するときにこのために次の 2 つのルールを使用しました。

```
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
);
```

ただし、 ORIGIN_GROUP では、次の単一ルールコミットスコープ example_scope_2 を追加して使用するだけで、上記の example_scope で使用した 2 つの個々のルールと同じ効果があります。

```
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
);
```

example_scope_2 では、トランザクションが left_dc から発生した場合、 ORIGIN_GROUP は left_dc にマップされ、NOT ORIGIN_GROUP は right_dc にマップされます。同様に、トランザクションが right_dc から発生した場合、 ORIGIN_GROUP は right_dc にマップされ、NOT ORIGIN_GROUP は left_dc にマップされます。したがって、1 つのルールを指定するだけで、2 つの効果が得られます。

さらにサブグループ、たとえば top_group の 3 番目の子 middle_dc を追加した場合、上記の example_scope_2 によると、left_dc から生成されたトランザクションの場合、 left_dc のすべてのノードに right_dc の任意の 1 を加え、middle_dc の任意の 1 を前に確認する必要があることに注意してください。トランザクションがコミットされます。もちろん、right_dc で発生したトランザクションの場

合、right_dc のすべてのノードと left_dc の任意の 1 ノード、および middle_dc の任意の 1 ノードは、トランザクションがコミットされる前に確認する必要があります。最後に、middle_dc は top_group の子であるため、 example_scope_2 は、 middle_dc で発生したトランザクションの場合、 middle_dc のすべてのノードと left_dc の任意の 1 ノード、および right_dc の任意の 1 ノードが、トランザクションがコミットされる前に確認する必要があることも意味します。

### Commit scope rules

コミットスコープルールは、コミットスコープメカニズムの中核です。これらは、コミットスコープが適用する内容を定義します。

コミットスコープルールは、組み合わせて動作する 1 つ以上の操作で構成されます。ルール間で AND を使用します。

各オペレーションは、コミットスコープグループ、オプションの確認レベル、および独自のパラメーターを持つことができるコミットスコープの種類の 2 つまたは 3 つの部分で構成されています。

```
commit_scope_group [ confirmation_level ] commit_scope_kind
```

完全な正式な構文図は、 Commit scopes リファレンスで入手できます。

ANY 2 (group) GROUP COMMIT などの一般的なコミットスコープルールは、そのコンポーネントに分割できます。ANY 2 (group) は、ルールに、どのノードが応答して、トランザクションを処理したことを確認する必要があるかを指定するコミットスコープグループです。この例では、名前付けグループの任意の 2 つのノードが確認する必要があります。

確認レベルが指定されていないため、デフォルトが使用されます。ルール全体を次のように考えることができます。

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

visible 設定は、すべてのトランザクションの変更がディスクにフラッシュされ、他のトランザクションから可視されると、ノードが確認できることを意味します。

このオペレーションの最後の部分はコミットスコープの種類です。この例では GROUP COMMIT です。 GROUP COMMIT は、名前付けグループ内の 2 つのノードが、トランザクションの変更をフラッシュして可視にしたことを確認したときに確認される同期 2 フェーズコミットです。

### コミットスコープグループ

コミットスコープグループには、 ANY 、ALL 、および MAJORITY の 3 種類があります。これらのすべての後には、かっこ内の 1 つ以上のグループのリストが続きます。グループのこのリストを組み合わせて、この操作が適用されるノードのプールを作成します。このリストには、NOT を付けることができます。これは、プールをリストにない他のすべてのグループに反転します。

- ANY n の後には、整数値 n が続きます。これは、リストされたグループのノード内の n ノードに変換されます。
- ALL の後にはグループが続き、リストされたグループのノード内のすべてのノードに変換されます。
- MAJORITY はグループが続き、マジョリティを与えるには、リストされたグループのノードの半分に 1 つを加えたものを必要とすることを意味します。

- `ANY n NOT` の後には、整数値 `n` が続きます。これは、リストされたグループのノードにない `n` ノードに変換されます。
- `ALL NOT` の後にはグループが続き、リストされたグループのノードにないすべてのノードに変換されます。
- `MAJORITY NOT` はグループが続き、マジョリティを与えるには、リストされたグループのノードにないノードの半分と 1 を必要とすることを意味します。

上記の式はすべて、評価時にグループ内のデータノードのみを考慮します。Witness ノードおよびその他の非データノードは無視されます。

### 確認レベル

PGD ノードは、さまざまなタイミングでトランザクションの確認を送信できます。保護レベルが増加すると、確認ノードの観点から、これらは次のとおりです。

- `received` —リモート PGD ノードは、トランザクションを受信したらすぐに、ローカルアプリケーションを開始する前に確認します。
- `replicated` —トランザクションの変更を適用した後、ディスクにフラッシュする前に確認します。
- `durable` —すべての変更がディスクにフラッシュされた後、トランザクションを確認します。
- `visible` デフォルトすべての変更がディスクにフラッシュされ、同時トランザクションに可視された後、トランザクションを確認します。

コミットスコープのルールでは、次のように `ON` のかっこ内のノードグループ定義にこれらの確認レベルを追加できます。

- `ANY 2 (right_dc) ON replicated`
- `ALL (left_dc) ON visible` デフォルト
- `ALL (left_dc) ON received AND ANY 1 (right_dc) ON durable`

> **注釈**
>
> PostgreSQL の `synchronous_standby_names` 機能に慣れている場合は、 *synchronous_standby_names* とコミットスコープの文法は似ているように見えますが、微妙な違いがあることに注意してください。前者は起点ノードを考慮しませんが、後者は考慮します。たとえば、 *synchronous_standby_names = 'ANY 1 (..)'* は、*ANY 2 (...)* のコミットスコープと同等です。この違いにより、マジョリティに関する推論が簡単になり、起点ノードがトランザクションの耐久性と可視性にも貢献することを反映しています。

### コミットスコープの種類

現在、4 つのコミットスコープ種類があります。以下は概要と、詳細へのリンクです。

### 同期コミット

同期コミットは、ネイティブの Postgres synchronous_commit オプションのように動作するように設計されたコミットスコープオプションですが、コミットスコープ環境内から使用できます。GROUP COMMIT とは異なり、これは同期非 2 フェーズコミット操作です。GROUP COMMIT と同様に、オプションの DEGRADE ON 句をサポートします。このオプションの前にあるコミットスコープグループは、SYNCHRONOUS COMMIT が使用するグループと確認要件を制御します。

詳細については、 同期コミット を参照してください。

### グループコミット

グループコミットは、コミットスコープグループの要件に従って確認される同期の 2 フェーズコミットです。GROUP COMMIT には、以下を制御するオプションがあります。

- 中断に対するトランザクションを追跡するかどうかブール値、デフォルトはオフ
- 競合を解決する方法 async または eager 、デフォルトは async
- コンセンサスを取得する方法 group 、partner または raft 、デフォルト group

詳細については、 耐久性オプショングループコミット/ *CAMO* を参照してください。

### CAMO

Commit At Most Once または CAMO により、クライアント/アプリケーション、オリジンノード、およびパートナーノードは、トランザクションが最大 1 回データベースにコミットされることを保証できます。クライアントはプロセスに関係するため、CAMO プロセスに参加するにはアプリケーションを変更する必要があります。

詳細については、 *Commit At Most Once* を参照してください。

### ラグコントロール

ラグ制御を使用すると、システムのレプリケーションパフォーマンスが指定された制限を超えると、データベースとのクライアントの相互作用にコミット遅延を自動的に挿入し、クライアントにバックプレッシャーを提供できます。Lag Control には、適用できる最大コミット遅延を設定するパラメーターがあります。また、そのコミット遅延の増加をトリガーするキューサイズまたは処理時間の面でも制限があります。

詳細については、 *Lag Control* を参照してください。

### 結合ルール

コミットスコープルールは、組み合わせて動作する 1 つ以上の操作で構成されます。AND を使用して 1 つのルールを形成します。例

```
MAJORITY (Region_A) SYNCHRONOUS COMMIT AND ANY 1 (Region_A) LAG CONTROL (MAX_LAG_SIZE␣
↪= 50MB)
```

最初の操作は、Region_A の大部分に対する同期コミットを設定します。2 番目の操作は、 Region_A のノードのいずれかが 50MB を超える遅延がある場合に、コミット遅延の押し上げを開始する遅延制御を追加します。この操作の組み合わせにより、ノードに遅延が発生しているときに遅延制御が動作できます。

### Comparing durability options

#### 比較

PGD で使用可能な同期レプリケーションのオプションのほとんどは、さまざまなレベルの同期を許可し、パフォーマンスとノードまたはネットワークの停止に対する保護の間のさまざまなトレードオフを提供します。

次の 確認レベル リストは、その確認レベルが有効なときにユーザーが表示することを予期すべきものと、それがパフォーマンス、耐久性、一貫性にどのように影響を与える可能性があるかを説明します。

#### 受信時

**Expect** ピアノードは変更を受信しました。ピアノードのタプルストアでは何も更新されず、ストレージに書き込まれません。

受信時の確認は、通常に動作しているピアが、完全または部分的なネットワークの停止に直面しても、最終的にそれ以上の通信を必要とせずにトランザクションを適用できることを意味します。この確認には永続ストレージが関係しないため、ピアノードがクラッシュしてもトランザクションの再送信が必要になる場合があります。

**の場合**トランザクションの起点ノードは、トランザクションの受信を待機するだけです。トランザクションが大きい場合、システムの TPS パフォーマンスが向上する場合があります。

**対** 古い読み取りの可能性の増加。Postgres サーバーまたはオペレーティングシステムのクラッシュが発生すると、データが失われる可能性があるため、全体的に ON RECEIVED は堅牢ではありません。

#### レプリケートについて

**Expect** ピアノードは変更を受信し、タプルストアに適用しました。変更はストレージに書き込まれましたが、ストレージはディスクにフラッシュされていません。

レプリケーションの確認は、ピアが変更を受信して適用したことを意味します。これらの変更はストレージに書き込まれますが、オペレーティングシステムのキャッシュとバッファーに残っています。システムはそれらをディスクにまだ永続化していません。

**For** このチェックポイントは、トランザクション処理のタイムラインのさらに下方にあります。オリジンノードは、トランザクションが適用されるのを待機するだけで、永続化されません。

**対** ON RECEIVED よりも古い読み取りの可能性がわずかに低くなります。また、 ON REPLICATED では、データは Postgres のクラッシュから生き残ることができますが、オペレーティングシステムのクラッシュからは生き残れません。

### ON DURABLE

**Expect** ピアノードは変更を受信し、タプルストアに適用し、変更をストレージに永続化しました。変更はまだ他のセッションで利用できるようにしていません。

永続的な確認は、トランザクションが書き込まれ、ピアノードのストレージにフラッシュされたことを意味します。これにより、ピアノードのクラッシュと復旧後のデータの損失から保護されます。ただし、セッションが切断して再接続する前に ON DURABLE ルールを使用してトランザクションをコミットした場合、トランザクションの変更は再接続されたセッションに可視される保証はありません。

Group Commit コミットスコープ種類と使用した場合、これは変更が可視されることも意味します。

**の場合**より堅牢で、クラッシュが発生した場合に再送信なしで回復できます。

**Against** フェイルオーバーが発生した場合の一貫性を保証しません。

#### オンビジブル

**Expect** ピアノードは変更を受信して適用し、それらの変更を永続化してストレージにフラッシュしました。

可視性の確認は、トランザクションが完全にリモートで適用されたことを意味します。切断して再接続する前に、セッションが ON VISIBLE ルールを使用してトランザクションをコミットした場合、トランザクションの変更は再接続されたセッションに可視されることが保証されます。

**対象** 堅牢で一貫性があります。

**対** パフォーマンスが低下します。

### Degrading commit scope rules

`SYNCHRONOUS COMMIT` および `CAMO` には、それぞれ、特定のパフォーマンスしきい値を超えたときにトランザクションの要件を低下させるオプション機能があります。`GROUP COMMIT` は機能を低下させることはできませんが、タイムアウトで中断することはできます。

ノードがトランザクションを適用しており、そのトランザクションがタイムアウトした場合、単にロールバックするのではなく、完了するトランザクションの要件を低下させるプロセスをトリガーすると役立つ場合があります。

`DEGRADE ON` は、トランザクションのコミットスコープルールを正常に低下させるためのルートを提供します。最も単純では、`DEGRADE ON` は、タイムアウトと、コミットスコープが正常に低下できるコミットスコープ操作の 2 番目のセットを取得します。

たとえば、20 ミリ秒または 30 ミリ秒のタイムアウト後、コミットスコープを満たすための要件は `ALL (node_group_name) SYNCHRONOUS COMMIT` から `MAJORITY (node_group_name) SYNCHRONOUS COMMIT` に低下し、トランザクションがより安定して適用されるようになります。

degrade 句がトリガーされるように、書き込みリーダーがトランザクションのオリジネーターであることを必要とすることもできます。これは、たとえば 2 つのデータノードと 1 つの監視ノードがある「スプリットブレインシナリオ」で役立つ場合があります。2 つのデータノードの間でネットワークが分割され、両方のデータノードへの接続があると仮定すると、監視ノードとの raft 選挙を介してリーダーに選ばれるのは一方だけであるため、一方のみが機能低下することは許可されます。

#### 動作

一般化された `DEGRADE` 句がトランザクションに適用されたときにどのように動作するかについては、2 つの部分があります。

コミット中に 1 回、処理中のコミットがコミットスコープルールを満たす応答を待機している間、PGD はタイムアウトをチェックし、タイムアウトが満了している場合、処理中のコミットは `DEGRADE` のコミットスコープルールを待機するように再構成されます。条項。実際、この時点で、`DEGRADE` 句のコミットスコープルールは既に満たされている可能性があります。

このメカニズムだけでは、目的の動作には不十分です。これだけでは、すべてのトランザクション接続の問題により低下することが確実なトランザクションも含めて、低下モードがキックインする前にタイムアウトが満了するのを待機する必要があり、パフォーマンスに重大な影響を与えるためです。このような劣化クラスターシナリオ。

これを回避するために、PGD マネージャープロセスは定期的に 5 秒ごとに接続と適用レート bdr.node_replication_rates のものをチェックし、レプリケーションの現在の状態に基づいて、その時点で低下するコミットスコープがある場合、それらは自動的に低下—その後の処理時にそのコミットスコープを使用するトランザクションは、タイムアウトを待機する代わりに低下ルールを使用します—レプリケーションが再び十分に迅速に移動していることをマネージャープロセスが検出するまで。

### 同期コミットとグループコミット

SYNCHRONOUS COMMIT と GROUP COMMIT は両方とも timeout および require_write_lead パラメーターを持ち、デフォルトはそれぞれ 0 および false です。デフォルトの 0 では即時低下が発生するため、おそらく常に timeout を設定する必要があります。デグレードモードに切り替えるために、書き込みリーダーがトランザクションのオリジネーターであることを要求することもできます。ここでも、デフォルトは false です。 SYNCHRONOUS COMMIT の場合、 timeout および require_write_lead はデグレードに適用され、GROUP COMMIT の場合、これらのパラメーターはアボートに適用されます。他のノードからの確認を待機する前にプライマリで既にコミットされているため、GROUP COMMIT コミットスコープはデグレードできず、SYNCHRONOUS COMMIT コミットスコープはアボートできません。

SYNCHRONOUS COMMIT には、どのルールに低下できるかに関するオプションもあります。これは、どのルールから低下するかによって異なります。

まず、非同期操作に低下させることができます。

```
ALL (left_dc) SYNCHRONOUS COMMIT DEGRADE ON (timeout=20s) TO ASYNC
```

同じコミットスコープ種類のより制限の低いコミットグループに低下することもできます。これも種類が SYNCHRONOUS_COMMIT または GROUP COMMIT のいずれかである場合に限ります。たとえば、次のように機能を低下させることができます。

```
ALL (left_dc) SYNCHRONOUS COMMIT DEGRADE ON (timeout=20s) TO MAJORITY (left_dc)␣
↪SYNCHRONOUS COMMIT
```

または次のように。

```
ANY 3 (left_dc) SYNCHRONOUS COMMIT DEGRADE ON (timeout=20s) TO ANY 2 (left_dc)␣
↪SYNCHRONOUS COMMIT
```

ただし、 SYNCHRONOUS COMMIT から GROUP COMMIT に低下することはできません。

### CAMO

*Commit At Most Once* は、同じ timeout および require_write_lead パラメーター同じデフォルトでそれぞれ 0 および false をサポートしますが、オプションは非同期オペレーションに低下するだけであるという点で簡単です。

```
ALL (left_dc) CAMO DEGRADE ON (timeout=20ms, require_write_lead=true) TO ASYNC
```

繰り返しになりますが、デフォルトは 0 であるため、 timeout パラメーターを設定する必要があります。

### Synchronous Commit

コミットスコープの種類 SYNCHRONOUS COMMIT

#### 概要

PGD の SYNCHRONOUS COMMIT は、基になるオペレーションでの PostgreSQL の *synchronous_commit* オプションに似た方法で動作するコミットスコープの種類です。ただし、PostgreSQL オプションとは異なり、コミットスコープとして構成され、PGD での構成と対話が簡単です。

GROUP COMMIT や CAMO などの他のコミットスコープの種類とは異なり、SYNCHRONOUS COMMIT オペレーションのトランザクションは 2 フェーズコミット 2PC トランザクションに変換されません。これらは Postgres synchronous_commit のように動作します。

#### 例

この例では、このコミットスコープが使用されている場合、left_dc グループ内のノードは SYNCHRONOUS COMMIT を使用して、left_dc グループ内の他のノードに変更をレプリケートします。 left_dc グループ内のノードの大部分を検索して、トランザクションをコミットしたことを確認します。

```
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
);
```

#### 構成

SYNCHRONOUS COMMIT は、オプションの DEGRADE ON 句をサポートしています。特定の構成パラメーターについては SYNCHRONOUS COMMIT コミットスコープリファレンスを参照してください。または、オプションの低下に関しては *Degrading commit scope rules* を参照してください。

#### 確認

| 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. |

次のページに続く

表 9 – 前のページからの続き

| Confirmation Level | PGD Synchronous Commit handling |
|---|---|
| *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. |

### 詳細

現在、SYNCHRONOUS COMMIT は、コミットスコープルール構文の確認レベルを使用しません。

コミットスコープルールでは、元のキーワード SYNCHRONOUS_COMMIT は SYNCHRONOUS COMMIT にエイリアスになりました。アンダースコアの代わりにスペースを使用すると、Postgres のネイティブ SYNCHRONOUS_COMMIT と区別するのに役立ちます。

## Group Commit

コミットスコープの種類 GROUP COMMIT

### 概要

グループコミットの目標は、単一ノードの障害または一時的な停止が発生した場合のデータの損失から保護することです。これは、COMMIT 時にトランザクションを正常に確認するために複数の PGD ノードを必要とすることにより実現します。確認はトランザクション処理のさまざまなポイントで送信できますが、トランザクションがディスクにフラッシュされ、他のすべてのトランザクションから見える場合、デフォルトの「可視」になります。

> **警告**
>
> グループコミットは、現在プレビューと評価を目的とした実験機能として提供されています。貴重な機能を提供しますが、実稼働環境に適さない既知の制限と課題があります。これらの制限が将来のリリースで解決されるまで、運用シナリオでこの機能を使用しないことをお勧めします。

### 例

```
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
);
```

この例では、 left_dc グループのすべてのノードと right_dc グループのいずれかのノードがコミットされたトランザクションを受信して正常に確認する必要があるコミットスコープを作成します。

### 要件

通常のオペレーションでは、グループコミットはアプリケーションに対して透過的です。フェールオーバー中に進行していたトランザクションには、アプリケーションまたは間のプロキシによってトリガーまたは統合される調整フェーズが必要です。このアクティビティは現在、元のノードが復旧した場合、またはクラスターから分離された場合にのみ発生します。この動作は、Postgres の古いビルトイン同期レプリケーションの場合と同じです。

グループコミットでコミットされたトランザクションは、その下で *two-phasecommit* を使用します。したがって、ノードごとに発生するこのようなすべてのトランザクションを処理できるように十分に高い `max_prepared_transactions` を構成します。

### 制限事項

Known Issues and Limitations の Group Commit セクションを参照してください。

### 構成

`GROUP_COMMIT` は、オプショナルの `GROUP COMMIT` パラメーター、`ABORT ON` および `DEGRADE ON` 句をサポートしています。構成パラメーターの完全な説明については、*GROUP_COMMIT* コミットスコープリファレンスを参照してください。または、`DEGRADE ON` オプション一般の詳細については、*Degrading commit scope rules* セクションを参照してください。

### 確認

| 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. |

### 動作

Group Commit の動作は、コミットスコープによって適用される構成によって異なります。

### 決定をコミットする

`group` 、`partner` 、および `raft` の 3 つの方法でコミットを決定するようにグループコミットを構成できます。

`group` 決定がデフォルトです。コミットスコープグループで必要な数の確認を受信したときに、元のノードによってコミットが確認されることを指定します。違いは、コミットの決定が PREPARE レプリケーションに基づいて行われるのに対し、耐久性は COMMIT (PREPARED) レプリケーションをチェックすることです。

`partner` 決定は、 *Commit At Most Once* CAMO が使用するものです。このアプローチは、ノードグループに 2 つのデータノードが存在する場合にのみ機能します。これらの 2 つのノードはお互いのパートナーで

あり、オリジンではなくレプリカが何かをコミットするかどうかを決定します。このアプローチでは、アプリケーションが何らかの形でコンセンサスの一部であるため、CAMO トランザクションプロトコルを使用して正常に動作するには、アプリケーションの変更が必要です。このアプローチの詳細については、*Commit At Most Once* を参照してください。

`raft` 決定は、コミット決定に PGD ビルトインの Raft コンセンサスを使用します。`raft` 決定を使用すると、パフォーマンスが低下する可能性があります。現在、ALL コミットスコープグループで `GROUP COMMIT` を使用する場合にのみ必要です。

ALL コミットスコープグループを使用するには、コミット決定を `raft` に設定して、**トランザクション調整** 問題を回避する必要があります。

#### 競合の解決

競合の解決は `async` または `eager` です。

非同期とは、PGD が、特定のノードに構成されている行レベルの解決を使用して、レプリケーション中にオプティミスティック競合解決を行うことを意味します。これは、元のトランザクションがコミットされたか、まだ進行中であるかに関係なく発生します。非同期競合解決の仕組みの詳細については、Conflicts を参照してください。

Eager とは、競合が COMMIT 契約の一部として熱心に解決され、競合するトランザクションがシリアル化エラーで中止されることを意味します。このアプローチは、パフォーマンスを犠牲にして非同期解決よりも高い分離を提供します。

ALL コミットスコープグループを使用するには、リコンシリエーションの問題を回避するために、**決定をコミットする** を `raft` に設定する必要があります。

Eager 競合解決の仕組みの詳細については、**熱心な競合解決** を参照してください。

#### アボート

COMMIT でコンセンサスを取得できないトランザクションが永久にハングしないように、`ABORT ON` 句でタイムアウトを指定できます。タイムアウトの後、トランザクションのアボートが要求されます。アボート要求が送信されたときにトランザクションがコミットされることが既に決定されている場合、クライアントがアボートメッセージを受信する場合でも、トランザクションは最終的に COMMIT します。

Limitations も参照してください。

#### トランザクション調整

元ノードでのグループコミットトランザクションのコミットは、暗黙的に 2 フェーズコミットに変換されます。

最初のフェーズ準備では、トランザクションがローカルで準備され、コミットの準備が整います。データは永続化されますが、この段階ではコミットされていないため、他のトランザクションはこのトランザクションによって行われた変更を認識できません。この準備されたトランザクションは、通常の論理レプリケーションを介して残りのすべてのノードにコピーされます。

オリジンノードは、グループコミット文法のルールに従って、他のノードからの確認を求めます。クラスター内の最小限必要なノードから確認を取得した場合、このトランザクションをコミットすることを決定

し、第 2 フェーズコミットに進みます。コミットフェーズでは、この決定をレプリケーションを介して他のノードに送信します。これらのノードは、最終的にこのメッセージの取得にコミットします。

さまざまな段階で失敗する可能性があります。たとえば、オリジンノードは、トランザクションの準備後にクラッシュする場合があります。または、オリジンと 1 つ以上のレプリカがクラッシュする場合があります。

これにより、準備されたトランザクションがシステムに残ります。Postgres の pg_prepared_xacts ビューは、システムで準備されたトランザクションを表示できます。準備されたトランザクションは、ロックおよびその他のリソースを保持している可能性があります。これらのロックとリソースを解放するには、トランザクションを中止またはコミットします。その決定は、ノードのコンセンサスで行う必要があります。

commit_decision が raft の場合、Raft はリコンシリエーターとして機能し、これらのトランザクションは最終的に自動的にリコンサイルされます。

commit_decision が group の場合、トランザクションは Raft を使用しません。代わりに、クラスター内の書き込みリードがリコンシリエーターの役割を実行します。これは、サブグループ内の変更に関して最も進んでいるノードであるためです。ノードがダウンしていることを検出し、ダウンしたノードをオリジンとして持っている準備されたトランザクションを探すことにより、このようなノードの調整を開始します。

このようなすべてのトランザクションについて、コミットスコープのルールに従ってノードに準備されたトランザクションがあるかどうかを確認し、決定を行います。この決定は Raft を介して伝えられ、ノードの大部分が調整を行うために立ち上がっている必要があります。

このプロセスはバックグラウンドで発生します。これを制御または発行するために使用するコマンドはありません。

### 熱心な競合解決

Eager 競合解決 Eager Replication としても知られ、COMMIT 決定プロセス中にシリアル化可能なエラーで互いに競合するトランザクションをアボートすることにより、競合を防止します。

概要 の競合解決オプションの 1 つとしてコミットスコープ を使用して構成します。

### 使用法

迅速な競合解決を有効にするには、クライアントをコミットスコープに切り替える必要があります。ここで示すように、セッションレベルまたは個々のトランザクションでそれを使用します。

```
BEGIN;

SET LOCAL bdr.commit_scope = eager_scope;

... other commands possible...
```

クライアントは、トランザクションの最後に COMMIT の発行を継続し、PGD に 2 つのフェーズを管理させることができます。

```
COMMIT;
```

この場合、 eager_scope コミットスコープは次のように定義されます。

```
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
);
```

Eager 競合解決ルールのコミットスコープグループは、ALL または MAJORITY のみです。 ALL を使用する場合、commit_decision 設定も raft に設定する必要があります。

#### エラー処理

PGD がトランザクションを管理することを考えると、クライアントは COMMIT の結果のみを確認する必要があります。これは、シングルノードの Postgres を含む、いかなる場合にもお勧めします。

オリジンノードに障害が発生した場合、残りのノードは最終的に少なくとも ABORT ON timeout の後、グローバルに準備されたトランザクションをロールバックすることを決定します。Raft は、一貫性のないコミットとロールバックの決定を防ぎます。ただし、これには接続されたノードの大部分が必要です。切断されたノードは、必要に応じて最終的にコミットまたはロールバックするようにトランザクションを準備し続け、決定してさらに進行した可能性のあるノードの大部分と調整します。

### Eager Replication の効果一般

#### アボート率の増加

シングルノード Postgres、またはデフォルトの非同期レプリケーションモードの PGD でも、COMMIT 時のエラーはほとんどありません。競合解決に eager を使用するコミットスコープの使用により追加された同期ステップも、エラーのソースも追加します。アプリケーションは、通常は再試行ループを適用することにより、このようなエラーを適切に処理するように準備する必要があります。

アボートの率は、ワークロードにのみ依存します。多くの行を変更する大規模なトランザクションは、他の同時トランザクションと競合する可能性が非常に高くなります。

### MAJORITY および ALL ノードレプリケーション一般の影響

#### コミットレイテンシーの増加

コミットスコープの使用により同期ステップを追加すると、ノード間の通信が増加し、コミット時のレイテンシーが増加することを意味します。ALL がコミットスコープで使用される場合、これは、ノードがダウンするとトランザクションが失敗するため、システムの可用性が低下することを意味します。

1 つ以上のノードが遅延している場合、確認を取得する際の往復遅延が大きくなり、高遅延が発生する可能性があります。ALL または MAJORITY ノードレプリケーションでは、約 2 つのネットワークラウンドトリップ、最悪の場合、最も遠いピアノードまでが追加されます。論理スタンバイノードと、まだ参加またはキャッチアップ中のノードは含まれませんが、最終的に変更を受け取ります。

ピアノードがトランザクションのローカルでの準備を確認する前に、それをローカルに適用する必要があります。これにより、トランザクションのサイズによっては、コミットレイテンシーがさらに増加します。この設定は synchronous_commit 設定とは無関係です。

### Commit At Most Once

コミットスコープの種類 CAMO

#### 概要

Commit At Most Once CAMO 機能の目的は、アプリケーションが複数回コミットしないようにすることです。

CAMO がないと、 COMMIT が送信された後にクライアントが接続を失うと、アプリケーションはサーバーから応答を受け取らない場合があるため、トランザクションがコミットしたかどうかがわかりません。

アプリケーションは、次の 2 つのオプションの間を簡単に決定できません。

- 場合によっては、データが 2 回入力される可能性があるため、同じデータでトランザクションを再試行します。
- トランザクションを再試行せず、データがまったく処理されないリスク

これらはいずれも、高値のデータに関する重大なエラーです。

この状況を回避する 1 つの方法は、トランザクションの一意のインデックスを持つテーブルに少なくとも 1 つの INSERT を含めることです。ただし、これはアプリケーションのデザインに依存し、アプリケーション固有のエラー処理ロジックが必要であるため、すべての場合に効果的ではありません。

PGD の CAMO 機能は、より一般的なソリューションを提供し、INSERT を必要としません。 bdr.commit_scope によってアクティブ化されると、アプリケーションはトランザクション識別子が既に割り当てられている場合を含むメッセージを受信します。それ以外の場合、トランザクションの最初の write ステートメントはその情報をクライアントに送信します。

アプリケーションが明示的な COMMIT を送信する場合、プロトコルは、COMMIT が送信される前にアプリケーションがトランザクション識別子の通知を受信することを保証します。サーバーが COMMIT に応答しない場合、アプリケーションはトランザクション識別子を使用して別の PGD ノードからトランザクションの最終ステータスを要求することにより、このエラーを処理できます。以前のトランザクションステータスがわかっている場合、アプリケーションはトランザクションを再試行するかどうかを安全に決定できます。

CAMO は、同じ PGD グループからの 2 つの PGD ノードであるパートナーノードのペアを作成することにより機能します。この操作モードでは、ペアの各ノードは、他のピアで実行された最近のトランザクションの結果を知っており、特にニーズに応じて、COMMIT 中に切断されたトランザクションの結果を知っています。アプリケーションからトランザクションを受信するノードは「オリジン」と呼ばれ、これらのトランザクションを確認するノードは「パートナー」と呼ばれます。ただし、CAMO ペアのノードの CAMO 構成に違いはありません。ペアは対称です。

> **警告**
>
> CAMO、高度なエラー処理を利用するには、ユーザーのアプリケーションを変更する必要があります。パラメーターを有効にするだけでは保護を得るには十分ではありません。リファレンスクライアント実装は、要求に応じてお客様に提供されます。

> **注釈**
>
> *CAMO* コミットスコープ種類のほとんどは、追加の `DEGRADE ON` 句を備えた `GROUP COMMIT (transaction_tracking = true, commit_decision = partner)` のエイリアスです。

### 要件

CAMO を使用するには、アプリケーションはマルチステートメント要求の一部としてではなく、別の要求として明示的な `COMMIT` メッセージを発行する必要があります。CAMO は、プロシージャーまたは暗黙的なコミットを使用する単一ステートメントトランザクションから発行されたトランザクションのステータスを提供できません。

### 構成

構成パラメーターについては、CAMO コミットスコープリファレンスを参照してください。

### 確認

| 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. |

### 制限事項

Limitations の CAMO セクションを参照してください。

### 障害シナリオ

さまざまな構成では、さまざまな障害シナリオが発生します。

### レシーバー側のデータの永続性

デフォルトでは、PGL ライタはリモートノードからトランザクションを適用するときに `bdr.synchronous_commit = off` モードで動作します。これは CAMO にも当てはまります。つまり、トランザクションは、CAMO パートナーのディスクに到達する前に、オリジンノードに確認されます。クラッシュまたはハードウェア障害が発生した場合、確認されたトランザクションは CAMO パートナーで自分自身では回復できない場合があります。これは、CAMO パートナーノードが復旧するとトランザクションを再配布するため、CAMO 起点ノードが動作している限り問題ではありません。

これは、CAMO が単一ノードの障害から保護できることを意味します。これは、ローカルモードおよびリモート書き込みと同様に、またはそれと組み合わせて正しいです。

CAMO ペアの両方のノードの停止をカバーするには、`bdr.synchronous_commit = local` を使用して、コミット前の確認の前にフラッシュを強制できます。これは、リモート書き込みモードでもローカルモー

ドでも動作せず、レイテンシーの影響を受けやすいコミットパスにある CAMO パートナーの I/O 要件により、パフォーマンスに影響を与えます。

### 非同期モード

コミットスコープで `DEGRADE ON ... TO ASYNC` 句が使用される場合、ノードは CAMO パートナーの準備ができているかどうかを検出します。そうでない場合、一時的に非同期ローカルモードに切り替わります。このモードの場合、ノードは CAMO モードに戻すまで、ローカルにトランザクションをコミットします。

これでは、COMMIT ステータスを取得することはできませんが、一貫性よりも可用性を選択できます。このモードは、単一ノードの障害を許容できます。CAMO ペアの両方のノードに障害が発生した場合、可用性を維持するために一致しないコミット決定を選択する可能性があり、データの不一貫性が発生します。

CAMO パートナーが準備ができているように切り替えるには、接続する必要があり、推定キャッチアップインターバルは `TO ASYNC` の `timeout` 値を下回る必要があります。*bdr.is_camo_partner_ready()* を使用して CAMO パートナーの現在の準備状況を確認できます。bdr.node_replication_rates は、キャッチアップ時間の現在の推定値を提供します。

CAMO 保護から非同期モードへの切り替えは、実際の CAMO トランザクションによってのみトリガーされます。これは、コミットが `TO ASYNC` の `timeout` 値を超えているため、または CAMO パートナーが既に知られている場合、コミット時に切断されたためです。このスイッチは、推定キャッチアップインターバルとは無関係です。CAMO ペアが、現在のノードが `enable_routing` ノードグループオプションで構成されたグループの書き込みリードであることを必要とするように構成されている場合。構文については、**コミットスコープ** を参照してください。これにより、分離されたノードが非同期モードに切り替わらないことによるスプリットブレイン状況を防ぐことができます。CAMO グループに `enable_routing` が設定されていない場合、オリジンノードはすぐに非同期モードに切り替わります。

非同期モードから CAMO モードへの切り替えは、接続を開始する CAMO パートナーノードに依存します。CAMO パートナーは、少なくとも 30 秒ごとに再接続を試行します。したがって、接続が再確立された後、CAMO パートナーが元のノードに接続するまでに最大 30 秒かかる場合があります。CAMO パートナーに蓄積された遅延は、CAMO 保護モードへの切り替えをさらに遅延させます。

通常の CAMO オペレーション中とは異なり、非同期モードでは、コミットオーバーヘッドの追加はありません。これは、CAMO ペアが通常処理できるよりも多くのトランザクションをノードが継続的に処理できるため、問題になる可能性があります。CAMO パートナーが最終的に再接続してトランザクションを適用した場合でも、このような状況では遅延が増加するだけであり、CAMO 保護の再確立を妨げます。トランザクションのスループットを人為的に調整するために、PGD は bdr.camo_local_mode_delay 設定を提供します。これにより、ローカルモードで `COMMIT` を任意の時間遅延できます。予想されるワークロード中に通常の CAMO モードでコミット時間を測定し、それに応じてこの遅延を構成することをお勧めします。デフォルトは 5 ミリ秒で、これは非同期ネットワークと比較的迅速な CAMO パートナーの応答を反映しています。

アーキテクチャと可用性要件を考慮して、非同期モードを許可するかどうかの選択を検討します。次の例は、詳細を提供します。

#### 例

この例では、互いに CAMO パートナーである 2 つの PGD ノードを使用したセットアップを検討します。

```
- - 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
);
```

この CAMO コミットスコープが正当であるためには、グループ内のノードの数が正確に 2 である必要があります。複数のノードで構成されるグループで ALL または ANY 2 を使用すると、非定量化グループ式が明確なペアに解決されないため、エラーになります。ノード。

#### 非同期モードの場合

非同期モードが許可されている場合、単一障害点はありません。1 つのノードに障害が発生した場合

- 他のノードは、障害が発生したノードで COMMIT 中に切断されたすべてのトランザクションのステータスを判断できます。
- 新しい書き込みトランザクションが許可されます。2 番目のノードにも障害が発生した場合、その時点でコミットされていたトランザクションの結果は不明です。

#### 非同期モードなし

非同期モードが許可されない場合、各ノードはトランザクションをコミットするために他のノードを必要とします。つまり、各ノードは単一障害点です。1 つのノードに障害が発生した場合

- 他のノードは、障害が発生したノードで COMMIT 中に切断されたすべてのトランザクションのステータスを判断できます。
- ノードが復旧するまで、新しい書き込みトランザクションは防止されます。

### アプリケーションの使用

#### 概要と要件

CAMO は、再試行ループとクライアント側での特定のエラー処理に依存しています。それには 3 つの側面があります。

- トランザクションの COMMIT の結果を確認する必要があり、一時的なエラーが発生した場合、クライアントはトランザクションを再試行する必要があります。
- COMMIT の前に、クライアントはノード ID とトランザクション ID 両方 32 ビット整数で構成されるトランザクションのグローバル識別子を取得する必要があります。
- トランザクションの COMMIT を試行中に現在のサーバーに障害が発生した場合、アプリケーションは CAMO パートナーに接続し、そのトランザクションのステータスを取得し、応答に応じて再試行する必要があります。

アプリケーションは、COMMIT 中に切断された場合にトランザクションステータスを確認する目的でのみグローバルトランザクション識別子を保存する必要があります。特に、アプリケーションは別の永続レイヤーを必要としません。アプリケーションに障害が発生した場合、データベースの情報のみを再起動します。

これを説明するために、この例では、C のような擬似コードで書かれた CAMO 対応クライアントアプリケーションの再試行ループを示します。接続情報を提供する 2 つの DSN origin_dsn および partner_dsn を想定しています。これらは通常、bdr.create_node への最初の呼び出しに使用される DSN と同じであり、 bdr.node_summary 、interface_connstr 列でルックアップできます。

```
PGconn *conn = PQconnectdb(origin_dsn);
```

プロセスは、オリジンノードへの接続を開始します。次に、ループに入ります。

```
loop {
    PQexec(conn, "BEGIN");
```

次に、トランザクションを開始し、変更の入力を開始します。

```
PQexec(conn, "INSERT INTO ...");
...
```

完了したら、ローカルノード ID とトランザクション ID の記録を作成する必要があります。どちらもパラメーターとして使用できます。

```
node_id = PQparameterStatus(conn, "bdr.local_node_id");
xid = PQparameterStatus(conn, "transaction_id");
```

これで、コミットする準備が整いました。

```
PQexec(conn, "COMMIT");
if (PQresultStatus(res) == PGRES_COMMAND_OK)
    return SUCCESS;
```

結果が PGRES_COMMAND_OK であれば、それは問題ありません。次に進んでください。しかし、そうでない場合は、CAMO を使用してトランザクションを完了まで追跡する必要があります。最初に尋ねる質問は「接続が悪かったですか?」

```
else if (PQstatus(res) == CONNECTION_BAD)
{
```

接続が悪い場合は、CAMO パートナーノードをチェックして、トランザクションがそこに到達したかどうかを確認できます。

```
conn = PQconnectdb(partner_dsn);
if (!connectionEstablished())
    panic();
```

パートナーノードに接続できない場合、できることはあまりありません。この場合、パニックになるか、同様の行動をとります。

ただし、接続できる場合は、*bdr.logical_transaction_status()* を使用して、トランザクションがどのように発生したかを確認できます。コードは、トランザクションをコミットする直前に、必要な値 node_id および xid トランザクション ID を記録しました。

```
    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
}
```

トランザクションがコミットされたと報告する場合、このトランザクションは成功したと言えます。これ以上のアクションは必要ありません。一方、コミットされたことを報告しない場合は、ループを続行して、パートナーノードでトランザクションを再試行できます。

```
    else
    {
        if (isPermanentError())
            return FAILURE;
        else
        {
            sleep(increasing_retry_delay);

            continue;
        }
    }
}
```

トランザクションのステータスが成功または不正な接続ではなかった場合、問題が永続的なエラーであったかどうかを確認します。その場合、トランザクションの失敗を報告します。そうでない場合は、再試行できます。再試行のたびに増加する期間コードをスリープさせて、トランザクションを再試行します。

### CAMO パートナーとの連携

ファンクション *bdr.is_camo_partner_connected()* を使用すると、ペアモードで構成された CAMO パートナーノードの接続ステータスを確認できます。現在、Eager Replication で使用される CAMO に相当するものはありません。

CAMO パートナーの準備ができていることを確認するには、ファンクション bdr.is_camo_partner_ready を使用します。その下で、これはローカルモードへの、またはローカルモードからのスイッチをトリガーします。

設定された CAMO パートナーの詳細を確認するには、*bdr.get_configured_camo_partner()* を使用します。このファンクションは、ローカルノードの CAMO パートナーを結果ます。

ファンクション *bdr.wait_for_camo_partner_queue()* を使用して、CAMO パートナーがキューを処理するのを待機できます。このファンクションは、bdr.wait_for_apply_queue のラッパーです。違いは *bdr.wait_for_camo_partner_queue()*

デフォルトでは CAMO パートナーノードを照会します。ローカルノードが CAMO ペアの一部でない場合、エラーを返します。

ノードに障害が発生したときにコミットされていたトランザクションのステータスを確認するには、アプリケーションは *bdr.logical_transaction_status()* ファンクションを使用する必要があります。

このファンクションに、チェックするトランザクションの node_id および transaction_id を渡します。ペアモードで CAMO を使用する場合、CAMO ペアの一部であるノードでのみこの機能を使用できます。 Eager Replication と併せて、すべてのノードで使用できます。

すべての場合、コミット発行後 15 分以内にファンクションを呼び出す必要があります。CAMO パートナーはこのようなメタ情報を定期的にパージする必要があるため、古いトランザクションに正しい回答を提供できません。

トランザクションのステータスを照会する前に、このファンクションは、受信キューが消費され完全に適用されるまで待機します。このメカニズムは、受信されたがまだ適用されていないトランザクションの早期の否定的な回答を防ぎます。

その名前にもかかわらず、それは常に読み取り専用操作であるわけではありません。ステータスが不明な場合、CAMO パートナーはトランザクションをコミットするか中止するかを決定し、その決定をローカルに保存して、今後の一貫性を保証します。

クライアントは、オリジンでコミットする前にこのファンクションを呼び出してはなりません。そうしないと、トランザクションが強制的にロールバックされる可能性があります。

### 接続プールとプロキシ

CAMO クラスターをデザインするときは、接続プールとプロキシの効果を考慮してください。プロキシは、コミットグループ内のすべてのノード、つまり CAMO ペアの両方のノード、または Eager All-Node Replication の場合はすべての PGD ノードにトランザクションを自由に分散することができます。

アプリケーションが適切なノード ID を取得するように注意してください。セッションプーリングを使用する場合、クライアントは同じノードに接続されたままであるため、ノード ID はクライアントセッションの有効期間を通じて一定のままです。ただし、より詳細なトランザクションプーリングでは、次の例のように、クライアントはすべてのトランザクションのノード ID を取得する必要があります。

PGD ノードに直接接続されていないクライアントは、フェールオーバーまたはスイッチオーバーに気付かない場合があります。ただし、`bdr.local_node_id` パラメーターを使用して、現在接続しているノードを判断することができます。COMMIT 中の切断という重要な状況では、プロキシはその切断をエラーとして CAMO プロトコルを適用するクライアントに適切に転送する必要があります。

`received` モードの CAMO の場合、CAMO ペア間を切り替える可能性のあるプロキシは、`bdr.wait_for_camo_partner_queue` ファンクションを使用して、古い読み取りを防ぐ必要があります。

### CAMO の制限

CAMO の制限については Known Issues and Limitations でカバーされています。

### パフォーマンスへの影響

CAMO は、コミット時にメッセージのラウンドトリップを追加することにより、Postgres レプリケーションプロトコルを拡張します。アプリケーションのコミットレイテンシーは非同期レプリケーションの場合よりも高く、これは主に、関係するノード間のラウンドトリップ時間によって決まります。同時セッションの数を増やすと、並列処理が増加して、合理的なトランザクションスループットを取得できます。

トランザクションを確認する CAMO パートナーは、トランザクションの状態を保存する必要があります。非 CAMO オペレーションと比較して、これは、オリジンから適用される各トランザクションの追加シークが必要になる場合があります。

### クライアントアプリケーションのテスト

クライアント側で CAMO を適切に使用することは簡単ではありません。ノードのクラッシュやネットワークの停止などの障害シナリオに対して、PGD クラスターを使用してアプリケーションの動作をテストすることを強くお勧めします。

## Lag Control

コミットスコープの種類 `LAG CONTROL`

### 概要

Lag Control は、レプリケーションが設定された制限の外部で実行されている場合、レプリカブル更新を行うトランザクションを処理した後、元のノードのクライアント接続に遅延が注入されるメカニズムを提供します。この遅延は、着信トランザクションを遅くし、定義された制限内にレプリケーションを戻すように設計されています。

### 背景

PGD 起点ノード上のデータベースアプリケーションのデータスループットは、コミットされたデータをダウンストリームのピアノードに複製できる速度を超える場合があります。

この不均衡が持続すると、RPO、RCO、GEO などの組織目標の充足が危険にさらされる可能性があります。

- **目標復旧ポイント RPO** は、計画外のイベントによって失われる可能性のあるデータの最大許容量を指定します。通常は時間として表されます。PGD では、RPO は、1 つ以上のピアノードに適用されていないコミットされたデータの許容量を決定します。
- **リソース制約目標 RCO** は、使用可能なストレージが有限であることを確認します。PGD では、遅延が増加すると、これらのストレージリソースの要求が増加します。
- **グループ弾性目標 GEO** は、ノードがピアノードに保存できないレートで新しいデータを生成していないことを保証します。

組織が目的を達成できるように、PGD はラグコントロールを提供します。この機能は、アプリケーションに侵入することなく、潜在的な不均衡を正確に調整する手段を提供します。これは、データを変更する READ WRITE トランザクションに遅延を透過的に導入することにより実現します。この遅延 PGD コミット遅延は 0 ミリ秒で始まります。

LAG CONTROL コミットスコープ種類を使用して、グループ内のノード間でコミットを遅延できる最大時間、最大遅延時間、または最大遅延サイズ WAL のサイズに基づいて、設定できます。

ノードが十分な数のノードで指定された最大値内でトランザクションを処理できる場合、PGD コミット遅延は 0 ミリ秒に留まるか、0 ミリ秒に向かって削減されます。ただし、十分な数のノードで最大値を超過する場合、元のノードでの PGD コミット遅延が増加します。十分な数のノードで Lag Control コンストレインが再び満たされるまで、増加し続けます。

PGD コミット遅延は、トランザクションが完了して、すべてのロックとリソースを解放した後に発生します。この遅延のタイミングにより、同時アクティブなトランザクションは、遅延されたトランザクション値の監視と変更、およびそのリソースの取得を続行できます。

厳密には、PGD コミット遅延はトランザクションごとの遅延ではありません。これは、特定のクライアント接続のトランザクションストリームにおけるコミット遅延の平均値です。この技術により、コミット遅延と値のきめの細かい調整が、OS スケジューラー、クロック割り込み、およびシステム負荷による変動の粗い粒度をエスケープできます。また、PGD ランタイムコミット遅延を可能な限り最小の期間のマイクロ秒以内に安定させて、遅延測定しきい値を維持できます。

### 要件

Lag Control の使用を開始するには

- すべてのデータベースアプリケーションが許容できる最大許容コミット遅延時間 `max_commit_delay` を決定します。
- 使用する遅延測定を決定します。ラグサイズ `max_lag_size` またはラグタイム `max_lag_time` のいずれかを選択します。
- 関係するグループまたはサブグループ、および確認を満たすために必要な各コレクションのノードの最小数を決定します。この情報は、コミットスコープルールの定義の基礎を形成します。

### 構成

コミットスコープでラグ制御を指定すると、コミットスコープルールがスパンするノード全体で一貫した調整されたパラメーター設定ができます。ラグコントロール仕様は、トップグループのデフォルトのコミットスコープに、またはオリジングループのコミットスコープの一部として含めることができます。

例のように、サブグループとして表される 2 つのデータセンター `left_dc` および `right_dc` を含む構成を取得します。

```
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
);
```

次のコードは、サブグループごとに個々のルールを使用して、これらの 2 つのデータセンターのラグ制御ルールを追加します。

```
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
);
```

Lag Control コミットスコープルールを、グループコミットおよび CAMO ルール仕様を含む既存のコミットスコープルールに追加できます。

`max_commit_delay` は間隔であり、通常はミリ秒 1ms 単位で指定されます。ミリ秒未満の精度での小数値の使用がサポートされています。

`max_lag_size` は、WAL バイト単位で最大許容遅延を指定する整数です。

`max_lag_time` は、通常秒単位で指定される間隔で、時間に関して最大許容遅延を指定します。

最大コミット遅延 `max_commit_delay` は、ハード制限を表す天井値であり、コミット遅延が構成値を超えないことを意味します。

最大ラグサイズと時間 `max_lag_size` および `max_lag_time` は、超過できるソフトリミットです。最大コミット遅延に到達すると、遅延対策に追加のバックプレッシャーはなく、継続的な増加を防ぎます。

#### 確認

| 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*. |

#### トランザクションアプリケーション

PGD コミット遅延は、ユーザーアプリケーションのデータを変更するすべての READ WRITE トランザクションに適用されます。この動作は、宣言された READ WRITE トランザクションを含む、データを変更しないトランザクションがコミット遅延から免除されることを意味します。

非同期トランザクションのコミットは、PGD コミット遅延も実行します。これは直観に反しているように見えるかもしれませんが、非同期コミットは、そのパフォーマンスのおかげで、レプリケーション遅延の

最大のソースの 1 つになる可能性があります。

Postgres および PGD 補助プロセスは、トランザクションのコミット時に遅延しません。最も注目的なのは、PGD ライターは、ローカルノードでリモートトランザクションを適用するときにコミット遅延を実行しないことです。PGD ライターは発信レプリケーションラグに何も寄与せず、遅延によってトランザクションコミットが調整されないことにより、着信レプリケーションラグを最も削減できるため、これはデザインです。

#### 制限事項

最大コミット遅延は、ハード制限を表す天井値であり、コミット遅延が構成された値を超えないことを意味します。逆に、最大遅延はサイズと時間の両方で測定され、超過できるソフト制限です。最大コミット遅延に到達すると、遅延対策に追加のバックプレッシャーはなく、継続的な増加を防ぎます。

PGD レプリケーションセットを変更しないオリジントランザクションをコミット遅延から免除する方法はありません。これらのトランザクションの場合、最大トランザクション遅延を 0 に SET LOCAL にすると役立ちます。

#### 注意事項

アプリケーション TPS は、レプリケーション遅延に影響を与える可能性のある多くの要因の 1 つです。他の要素には、PGD コミット遅延の効果があまりない可能性があるトランザクションの平均サイズが含まれます。特に、大量のロード操作はレプリケーション遅延の増加を引き起こす可能性があり、これは、最大許容遅延を下回っていますが、通常のアプリケーションで合理的に予想されるレベルを超えて PGD ランタイムコミット遅延の付随的な上昇をトリガーする可能性があります。

同様に、非常に高い OLTP 要件と適度なデータ変更があるアプリケーションは、許容可能な PGD コミット遅延設定によって不当に抑制される可能性があります。

これらの場合、 `SET [SESSION|LOCAL]` コマンドを使用して、これらのアプリケーションのラグ制御設定をカスタム構成するか、これらのアプリケーションを変更すると便利です。たとえば、大量のロード操作は複数の小さなトランザクションに分割されて、トランザクションのスナップショット期間と WAL 保持サイズを制限したり、大量のロードが失敗した場合にリスタートポイントを確立したりする場合があります。ラグ制御を参照して、これらのトランザクションコミットは、非常に長い PGD コミット遅延をスケジュールして、以前の部分的な大量ロードによって発生した遅延をダイジェストすることもできます。

#### 組織目標の達成

前にリストした目標の例では

- 適切な最大ラグタイムを設定することにより、RPO を満たすことができます。
- 適切な最大遅延サイズを設定することにより、RCO を満たすことができます。
- GEO は、PGD ランタイムコミット遅延と PGD ランタイムラグ対策をモニタリングすることにより満たすことができます。

前述のように、PGD ランタイムコミット遅延の最大値が PGD 構成のコミット遅延制限でペッグされ、遅延測定値が一貫して PGD 構成の最大レベルを超える場合、このシナリオは PGD グループ拡張のマーカーになる可能性があります。

### Lag Control と拡張機能

PGD コミット遅延は、コミット後の遅延です。これは、トランザクションがコミットされた後、トランザクションによってロックまたは取得されたすべての Postgres リソースが解放された後に発生します。したがって、遅延は、同時アクティブなトランザクションがその値を監視または変更する、またはそのリソースを取得することを妨げるものではありません。Postgres 拡張機能によって管理される外部リソースについては、同じ保証はできません。拡張機能の依存関係に関係なく、postgresql.conf で拡張機能ベースのリソースマネージャーの前に PGD 拡張機能がリストされている場合、同じ保証を行うことができます。

### Administering

グループコミットを使用して PGD クラスターを実行する場合、ノードを安全にシャットダウンおよび再起動する方法など、システムを管理するときにいくつかのことに注意する必要があります。

### 計画的なシャットダウンと再起動

受信確認でグループコミットを使用する場合、計画的なシャットダウンまたは再起動に注意してください。デフォルトでは、適用キューはシャットダウンする前に処理されます。ただし、`immediate` シャットダウンモードでは、シャットダウン時にキューは破棄されるため、停止したノードはキュー内のトランザクションを「忘れ」ます。元のノードに同時に障害が発生すると、両方のノードに障害が発生したかのように、データの損失が発生する可能性があります。

適用キューがディスクにフラッシュされるようにするには、メンテナンスタスクに *smart* のいずれかを使用します。このアプローチでは、必要な同期レベルを維持し、データの損失を防ぎます。

### Legacy synchronous replication using PGD

Postgres は *physical streamingreplication* を提供します

PSR、これは一方向ですが *synchronousvariant* を提供します。

下位互換性ために、PGD は `synchronous_commit` および `synchronous_standby_names` を使用した同期レプリケーションの構成を引き続きサポートしています。**耐久性オプショングループコミット/** *CAMO* または **同期コミット** の使用を検討します

代わりに。

PGD レプリケーションオプションとは異なり、PSR 同期は最初に持続し、コミットレコードの WAL フラッシュの後に複製します。

### 使用法

PGD を使用した同期レプリケーションを有効にするには、関連する PGD ピアノードのアプリケーション名を `synchronous_standby_names` に追加する必要があります。 `FIRST x` または `ANY x` を使用すると、非 PGD スタンバイノードの要件と競合しない場合、ある程度の柔軟性が提供されます。

追加したら、 `synchronous_commit` を使用してトランザクションごとの同期レベルを構成できます。デフォルトは `on` です。この設定は、アプリケーション名を `synchronous_standby_names` に追加することにより、同期レプリケーションが既に有効になっていることを意味します。`synchronous_commit` を `local` または `off` に設定すると、同期レプリケーションがオフになります。

PGD は永続化する前にトランザクションを適用するため、値 on および remote_apply は論理レプリケーションでは等価です。

### 比較

次の表は、トランザクションが発行されたオリジンノードから COMMIT 確認を受信した後、クライアントがレプリケートされたピアノードから期待できることをまとめています。Mode 列は、バリアントに応じて異なる意味を持ちます。PSR および PGD を使用した古い同期レプリケーションの場合、synchronous_commit 設定を参照します。

| 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)* **代わりにグループコミットの使用を検討します。**

- (2) synchronous_replication_availability を async' に設定してローカルモードに切り替えない限り、許可されている場合。それ以外の場合、非同期 PGD デフォルトの値が適用されます。

*(3) OS* **に書き込まれ、** *OS* **が実行されたままで** *Postgres* **のみがクラッシュした場合に持続します。**

### Postgres 構成パラメーター

次の表は、デフォルト以外の値 req に設定する必要がある構成設定とオプショナルですが特定のバリアントに影響を与える構成設定の概要を提供します。

| 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 |

### コミットスコープへの移行

synchronous_commit および synchronous_standby_names とは無関係に、PGD のグループコミット機能を構成します。代わりに、bdr.commit_scope GUC を使用すると、トランザクションごとにスコープを選択できます。また、各ノードで synchronous_standby_names を個別に構成するのではなく、グループコミットはグローバルに同期したコミットスコープを使用します。

**注釈**

*synchronous_standby_names* およびコミットスコープの文法は似ていますが、前者はオリジンノードを考慮していませんが、後者は考慮します。したがって、たとえば *synchronous_standby_names = 'ANY 1 (..)'* は、*ANY 2 (...)* のコミットスコープと同等です。この選択により、マジョリティに関する推論が簡単になり、起点ノードがトランザクションの耐久性と可視性にも貢献することを反映しています。

### Predefined Commit Scopes

PGD Essential と PGD Expanded は両方とも、使用可能な事前定義されたコミットスコープのセットを提供します。

2 つのエディションの違いは、 PGD Essential には変更できない事前定義されたコミットスコープの限定的なセットがあるのに対し、PGD Expanded は完全に管理および構成可能なコミットスコープを許可することです。PGD Essential の事前定義されたコミットスコープは、パフォーマンスとデータの安全性のバランスを提供するように設計されていますが、PGD Expanded の構成可能なコミットスコープは、耐久性保証のより高い柔軟性と制御を提供します。

#### ローカルプロテクト

```
ASYNCHRONOUS COMMIT
```

`local protect` コミットスコープは、PGD Essential のデフォルトのコミットスコープです。耐久性保証のない非同期コミットを提供します。これは、トランザクションがローカルノードの WAL に書き込まれるとすぐに、クラスター内の他のノードからの確認を待たずにコミットされたと考えられることを意味します。

このコミットスコープは、高可用性と低遅延がデータの耐久性よりも重要であるシナリオに適しています。ただし、ノードに障害またはネットワークの問題が発生した場合のデータ損失に対する保証はありません。

#### ラグプロテクト

```
MAJORITY ORIGIN GROUP LAG CONTROL (max_lag_time = 30s, max_commit_delay = 10s)
```

`lag protect` コミットスコープは、マジョリティオリジングループのラグタイムに基づいて、耐久性保証を提供します。これにより、遅延時間が指定された制限内この場合 30 秒であり、コミット遅延も指定された制限この場合 10 秒以内である場合にのみトランザクションがコミットされたと見なされます。これは、ネットワークの問題またはノードに障害が発生した場合のデータの損失を防ぐのに役立ちます。

このコミットスコープは、データの一貫性と耐久性が重要であるが、ある程度の遅延が許容されるシナリオで役立ちます。指定された遅延とコミット遅延制限内でオリジングループの大部分のノードによって確認されるまで、トランザクションがコミットされたとみなされないことを保証することにより、パフォーマンスとデータの安全性のバランスが可能です。

#### マジョリティプロテクト

```
MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT
```

`majority protect` コミットスコープは、マジョリティオリジングループに基づいて耐久性保証を提供します。これにより、トランザクションは、オリジングループ内のノードの大部分によって確認された場合

にのみコミットされたと見なされます。これは、ノードに障害またはネットワークの問題が発生した場合に、データの一貫性と耐久性を保証するのに役立ちます。

このコミットスコープは、データの一貫性と耐久性が重要なシナリオに適しており、local protect コミットスコープと比較してデータ損失に対する高レベルの保護を提供します。ただし、トランザクションがコミットされたものと見なされる前にマルチプルのノードからの確認が必要であるため、多少のレイテンシーが発生する場合があります。

#### アダプティブプロテクト

```
MAJORITY ORIGIN GROUP SYNCHRONOUS COMMIT DEGRADE ON (timeout = 10s, require_write_
↪lead = true) TO ASYNCHRONOUS COMMIT
```

adaptive protect コミットスコープは、より柔軟な耐久性保証を提供します。これにより、トランザクションはマジョリティオリジングループの同期コミットに基づいてコミットされたと考えられますが、指定されたタイムアウトこの場合 10 秒内にトランザクションを確認できない場合、非同期コミットに低下する可能性があります。これは、ネットワーク遅延またはノードの障害により、トランザクションの確認に遅延が発生する可能性があるシナリオで役立ちます。

このコミットスコープは、データの一貫性と耐久性が重要であるが、潜在的な遅延に対処するためにある程度の柔軟性が必要であるシナリオに適しています。local protect コミットスコープと比較してより高いレベルの保護を提供しながら、指定されたタイムアウト内にノードの大部分が確認できない場合でもトランザクションがコミットされたと見なすことにより、パフォーマンスとデータの安全性のバランスを提供します。

### Internal timing of operations

さまざまなモードがどのように動作するかをより深く理解するために、従来の物理的ストリーミングレプリケーション PSR と PGD はさまざまな方法でトランザクションを適用することを知っておくと役立ちます。

レガシー PSR では、操作の順序は次のとおりです。

1. オリジンはコミットレコードを WAL にフラッシュし、トランザクションをローカルに認識できるようにします。
2. ピアノードは変更を受信し、書き込みを発行します。
3. ピアは、受信した変更をディスクにフラッシュします。
4. ピアは変更を適用し、ピアでトランザクションを認識できるようにします。

変更は、変更を適用する前にディスクに書き込まれることに注意してください。

PGD では、デフォルトでと Lag Control を使用すると、操作の順序が異なります。これらの場合、トランザクションがピアのディスクにフラッシュされる前に、変更はピアで可視されるようになります。

1. オリジンはコミットレコードを WAL にフラッシュし、トランザクションをローカルに認識できるようにします。
2. ピアノードは、メモリ内の適用キューに変更を受信します。
3. ピアは変更を適用し、ピアでトランザクションを認識できるようにします。
4. ピアはディスクにフラッシュすることによりトランザクションを永続化します。

PGD のグループコミットと CAMO の場合、オリジンノードはトランザクションをローカルに表示する前に、特定の数の確認を待機します。操作の順序は次のとおりです。

1. Origin は、準備または事前コミットレコードを WAL にフラッシュします。
2. ピアノードは、メモリ内の適用キューに変更を受信します。
3. ピアは変更を適用し、ピアでトランザクションを認識できるようにします。
4. ピアはディスクにフラッシュすることによりトランザクションを永続化します。
5. オリジンはコミットして、トランザクションをローカルに可視化します。

次の表は、違いをまとめたものです。

| 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 |

### 6.3.20 Conflict Management

EDB Postgres Distributed は、アクティブ/アクティブまたはマルチマスター DBMS です。非同期で使用される場合、標準のデータ型を使用する場合、複数の異なるノードから同じまたは関連する行への書き込みにより、データが *bdr_read_all_conflicts* になる場合があります。

競合はエラーではありません。ほとんどの場合、これらは発生時に PGD が検出および解決できるイベントです。解決策はアプリケーションの性質とデータの意味によって異なるため、PGD がアプリケーションにそれらを解決する方法に関して幅広い選択肢を提供することが重要です。

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、ローカルまたはリモートのタプルのいずれかが選択され、他方は破棄されます。たとえば、コミットタイムスタンプは、2 つの競合する変更と新しい方が保持されると比較される場合があります。このアプローチでは、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスを確立します。

列レベルの競合の検出と解決は、*CLCD* で説明している PGD で利用できます。

競合を回避したい場合は、Group Commit を Eager conflict resolution または競合のないデータ型 CRDT とともに使用できます。*CRDT* **データ型を使用した列レベルの競合の処理** で説明しています。接続マネージャーを使用して、すべての書き込みを 1 つの書き込みリーダーにルーティングし、ノード間の競合の可能性を排除することもできます。

#### Conflicts

EDB Postgres Distributed は、アクティブ/アクティブまたはマルチマスター DBMS です。非同期で使用される場合、標準のデータ型を使用するときに、複数の異なるノードから同じまたは関連する行への書き込みにより、データの競合が発生する可能性があります。

競合はエラーではありません。ほとんどの場合、これらは発生時に PGD が検出および解決できるイベントです。このセクションでは、その検出と解決を管理できる PGD 機能を紹介します。

- **概要** は、PGD での競合のアイデアを紹介し、それらがどのように発生する可能性があるかを説明します。
- *Types of Conflict* は、PGD で発生する可能性のあるさまざまな種類の競合をリストおよび議論します。
- *Column-level conflict detection* は、PGD が競合検出のために提供するメカニズムを導入します。
- **欠落列の競合の解決** は、PGD が競合を解決する方法と、デフォルトの動作を変更する方法を説明します。
- *Conflict logging* は、PGD が競合ログを保持する場所を指摘し、競合レポートを実行する方法を説明します。
- *Data verification with LiveCompare* は、LiveCompare が発生した競合を指摘することにより、データの一貫性を保つ方法を説明します。

### Overview

EDB Postgres Distributed は、アクティブ/アクティブまたはマルチマスター DBMS です。非同期で使用される場合、標準のデータ型を使用するときに、複数の異なるノードから同じまたは関連する行への書き込みにより、データの競合が発生する可能性があります。

競合はエラーではありません。ほとんどの場合、これらは発生時に PGD が検出および解決できるイベントです。それらの解決は、アプリケーションの性質とデータの意味によって異なります。したがって、PGD はアプリケーションに競合を解決する方法についての幅広い選択肢を提供することが重要です。

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、PGD はローカルまたはリモートタプルのいずれかを選択し、他方は破棄します。たとえば、コミットタイムスタンプは、2 つの競合する変更と新しい方が保持されると比較される場合があります。このアプローチでは、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスを確立します。

**欠落列の競合の解決** で説明しているように、競合処理は構成可能です。 PGD は、*Stream triggers* で説明しているように、競合トリガーを使用してテーブルごとに競合を検出し、それらをさまざまな方法で処理できます。

*CLCD* で説明しているように、列レベルの競合の検出と解決は、PGD で利用できます。

デフォルトでは、すべての競合は bdr.conflict_history にログに記録されます。競合の可能性がある場合、テーブル所有者は競合を監視して回避方法を分析するか、アプリケーションタスクとして定期的に処理する計画を立てる必要があります。LiveCompare ツールは、発散がないか定期的にスキャンすることもできます。

一部のクラスタリングシステムは、分散ロックメカニズムを使用して、データへの同時アクセスを防止します。これらはサーバーが互いに非常に近い場合に合理的なパフォーマンスを発揮しますが、許容可能なパフォーマンスには非常に低い遅延が重要である地理的に分散したアプリケーションをサポートできません。

分散ロックは本質的に悲観的なアプローチです。PGD は、可能な限り競合を回避するが、一部の種類の競合の発生を許可し、発生した場合にそれらを解決する楽観的なアプローチを提唱しています。

### 競合が発生する仕組み

ノード間の競合は、関係するすべてのトランザクションが同じノードで同時に発生した場合に発生することはできない一連のイベントの結果として発生します。ノードはトランザクションがコミットした後にのみ変更を交換するため、各トランザクションはコミットしたノードで個別に有効です。他の競合する作業を同時に実行した別のノードに適用された場合、有効ではありません。

PGD レプリケーションは基本的に他のノードでトランザクションを再生するため、適用されるトランザクションと受信ノードでコミットされたトランザクションとの間に競合がある場合、再生操作は失敗する可能性があります。

Postgres にはそれを防ぐトランザクション間通信メカニズムがあるため、すべてのトランザクションが単一のノードで実行されている場合、ほとんどの競合は発生しません。これらのメカニズムの例は、`UNIQUE` インデックス、`SEQUENCE` 操作、行およびリレーションロック、`SERIALIZABLE` 依存関係追跡などです。これらのメカニズムはすべて、進行中のトランザクション間で通信して、望ましくない同時実行の問題を防ぐ方法です。

PGD には、分散トランザクションマネージャーもロックマネージャーもありません。これが、レイテンシーとネットワークパーティションで良好なパフォーマンスを発揮する理由の一部です。その結果、異なるノード上のトランザクションは、デフォルトの遅延レプリケーションを使用する場合、互いに完全に独立して実行されます。ノード間の独立性が低いほど、競合を完全に回避できます。これが、これが重要な場合に PGD が Eager Replication も提供する理由です。

### 競合の回避または許容

ほとんどの場合、競合を回避または許容するようにアプリケーションをデザインできます。

競合は、マルチプルのノードで同時に発生している場合にのみ発生します。競合を回避する最も簡単な方法は、1 つのノードにのみ書き込むか、一度に 1 つの特定のノードから特定の方法で特定の行のみに書き込むことです。

この回避は、多くのアプリケーションで自然に発生します。たとえば、多くのコンシューマアプリケーションでは、所有ユーザーのみがアカウントのデフォルトの請求先住所の変更など、データを変更できます。このようなデータの変更では、更新の競合が発生することはほとんどありません。

ノードがダウンする直前に変更を加える可能性があるため、変更は失われるようです。次に、同じ変更を再度行うと、別のノードで 2 つの更新が発生する場合があります。ダウンしたノードが復旧すると、古い変更を他のノードに送信しようとします。データの最終更新が保持されるため、リジェクトされます。

`INSERT` / `INSERT` 競合の場合、global sequences を使用してこのタイプの競合を防止します。

部屋予約アプリケーションなどのオブジェクト間の関係を割り当てるアプリケーションの場合、`update_if_newer` を適用すると、許容できるビジネス結果が得られない場合があります。つまり、同じ部屋を予約したことを 2 人に個別に確認することは役に立ちません。最も簡単な解決策は、Eager Replication を使用して、1 つの予約のみが成功するようにすることです。アプリケーションによっては、より複雑な方法が可能である場合があります。たとえば、各ノードに 100 席を割り当て、そのノードのライタがそれらを予約できるようにします。ただし、ローカルで利用できるものがない場合は、ほとんどのシートが予約された後に分散ロックスキームまたは Eager Replication を使用します。

特定の種類の更新が特定のノードからのみ発生するようにする別の手法は、異なるノードを介して異なる種類のトランザクションをルーティングすることです。例

- 1 つのノードで小包を受信するが、別のノードを使用して小包を配送する
- 1 つのノードで注文が入力され、2 番目のノードで作業が準備され、別のノードで顧客に提供されるサービスアプリケーション

多くの場合、最良のコースは、競合の発生を許可し、競合に対処するために PGD の競合解決メカニズムと連携するようにアプリケーションをデザインすることです。

### Types of Conflict

#### 主キーまたは一意の競合

最も一般的な競合は行の競合であり、2 つの操作が単一のノードではできない方法で同じキーを持つ行に影響を与えます。PGD はこれらのほとんどを検出でき、`update_if_newer` 競合リゾルバーを適用します。

行の競合には次のものが含まれます。

- `INSERT` vs `INSERT`
- `UPDATE` vs `UPDATE`
- `UPDATE` vs `DELETE`
- `INSERT` vs `UPDATE`
- `INSERT` vs `DELETE`
- `DELETE` vs `DELETE`

`bdr.node_conflict_resolvers` ビューは、すべての既知の競合タイプに対して競合解決が現在どのように構成されているかに関する情報を提供します。

#### INSERT/INSERT の競合

最も一般的な競合 `INSERT` /`INSERT` は、2 つの異なるノードでの `INSERT` 操作が、同じ `PRIMARY KEY` 値または `PRIMARY KEY` が存在しない場合、単一の `UNIQUE` コンストレインの同じ値を持つタプルを作成する場合に発生します。

PGD は、元のノードのタイムスタンプに従って、2 つのうちの最後に挿入されたタプルを保持することにより、この状況を処理します。ユーザー定義の競合ハンドラーは、この動作をオーバーライドできます。

この競合は `insert_exists` 競合タイプを生成します。これは、デフォルトで解決されます。コミット時間に基づいて、新しい行を選択し、その 1 つだけ `update_if_newer` リゾルバー。他のリゾルバーを構成できます。詳細は、欠落列の競合の解決 を参照してください。

この競合タイプを解決するには、列レベルの競合解決とユーザー定義の競合トリガーを使用することもできます。

global sequences を使用して、このタイプの競合を効果的に排除できます。

### UNIQUE または EXCLUDE 制約に違反する INSERT 操作

INSERT /INSERT 競合は、複数の UNIQUE 制約のうちの 1 つが PRIMARY KEY である可能性がある、または 1 つ以上の EXCLUDE 制約に違反する可能性があります。

次の場合、レプリケーション変更を適用すると、multiple_unique_conflicts 競合が生成されます。これらの場合は両方とも、複数の他の行との競合が発生します。

- 新しい行が複数の UNIQUE 制約に違反し、その結果、他の複数の行と競合する場合。
- 新しい行が複数の EXCLUDE 制約または単一の EXCLUDE 制約に違反する場合。

このような競合が発生した場合、レプリケーションを続行するには、いくつかの行を削除する必要があります。multiple_unique_conflicts のリゾルバー設定によっては、適用プロセスはエラーで終了するか、着信行をスキップするか、一部の行を削除します。削除では、正しい PRIMARY KEY を含む行を保持し、他の行を削除しようとします。

> **警告**
>
> この方法で競合する複数の行がある場合、競合解決の結果、挿入操作を続行すると、常にデータの一部が削除されます。

conflict trigger を使用して別の動作を定義することもできます。

### UPDATE/UPDATE の競合

異なるノードでの 2 つの同時 UPDATE 操作が同じタプルを変更するが、その PRIMARY KEY を変更しない場合、再生時に UPDATE /UPDATE 競合が発生する可能性があります。

これらは、構成と状況に基づいて、さまざまな種類の競合を生成する可能性があります。テーブルが 行バージョンの競合検出 で構成されている場合、元のキー行がローカル行と比較されます。それらが異なる場合、update_differing 競合が生成されます。 オリジンの競合検出を使用する場合、行の起点がチェックされます。オリジンは、現在のローカル行が発生したノードです。これが変更されると、update_origin_change 競合が生成されます。その他の場合、UPDATE は通常、競合を生成せずに適用されます。

これらの競合は両方とも、*INSERT/INSERT* の競合 で説明している insert_exists と同じ方法で解決されます。

### 主キーの UPDATE の競合

PGD は、現在、 PRIMARY KEY が UPDATE オペレーションによって変更される競合解決を実行できません。主キーを更新できますが、既存の値と競合しないことを確認する必要があります。

主キーの更新の競合は 多様な競合 であり、手動介入が必要です。

Postgres で主キーの更新は可能ですが、Postgres と PGD の両方に問題があります。

単純なスキーマは、次のことを説明する例を提供します。

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

主キー列の更新が可能であるため、次の SQL は成功します。

```
UPDATE pktest SET pk=2 WHERE pk=1;
```

ただし、テーブルに複数の行があるとします。

```
INSERT INTO pktest VALUES (3,3);
```

一部の UPDATE 操作は成功します。

```
UPDATE pktest SET pk=4 WHERE pk=3;

SELECT * FROM pktest;
 pk | val
- ---+-----
  2 |   1
  4 |   3
(2 rows)
```

他の UPDATE 操作は制約エラーで失敗します。

```
UPDATE pktest SET pk=4 WHERE pk=2;
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
DETAIL:  Key (pk)=(4) already exists
```

したがって、主キーを更新する Postgres アプリケーションの場合、PGD がなくても、実行時エラーを回避するように注意してください。

PGD では、複数の場所から同時に UPDATE 操作が許可される場合、状況はより複雑になります。

これらの 2 つの変更を同時に実行すると、動作します。

```
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)
```

両方の変更が受け入れられるため、これらの次の 2 つの変更を同時に実行すると、発散エラーが発生します。ただし、他のノードに変更を適用すると、update_missing 競合が発生します。

```
node1: UPDATE pktest SET pk=1 WHERE pk = 3;
node2: UPDATE pktest SET pk=2 WHERE pk = 3;
```

このシナリオでは、各ノードで異なるデータのままになります。

```
node1:
SELECT * FROM pktest;
 pk | val
- ---+-----
  1 |   1
  5 |   3
(2 rows)

node2:
SELECT * FROM pktest;
 pk | val
- ---+-----
  2 |   1
  5 |   3
(2 rows)
```

LiveCompare を使用して、この状況を特定して解決できます。

同時競合により問題が発生します。これら 2 つの変更を同時に実行することは、解決するのが簡単ではありません。

```
node1: UPDATE pktest SET pk=6, val=8 WHERE pk = 5;
node2: UPDATE pktest SET pk=6, val=9 WHERE pk = 5;
```

どちらの変更もローカルに適用されるため、ノード間の相違が発生します。しかし、ターゲットへの適用は、両方のノードで失敗し、重複キー値違反エラーが発生します。このエラーにより、レプリケーションが停止され、手動の解決が必要です。

conflict_type update_pkey_exists を skip 、update 、または update_if_newer に設定すると、この重複キー違反エラーを回避でき、レプリケーションは中断されません。このアプローチでも、更新の性質によっては、発散が発生する可能性があります。

update_pkey_exists から update_if_newer を設定することにより、同じ古いキーが同じ新しいキーによって同時に更新されている場合、発散を回避できます。ただし、特定の状況では、update_if_newer でも発散が発生します。つまり、2 つの異なる行が両方とも同じ新しい主キーに同時に更新される場合。

その結果、アプリケーション、特に PGD で主キーの UPDATE 操作を許可しないことを強くお勧めします。アプリケーションの一部が主キーを変更する場合、同時変更を回避するには、Eager Replication を使用してこれらの変更を行います。

> **警告**
>
> *update_pkey_exists* 競合の競合解決の結果が更新された場合、常にいずれかの行が削除されます。

### UNIQUE または EXCLUDE 制約に違反する UPDATE 操作

*UNIQUE または EXCLUDE 制約に違反する INSERT 操作* と同様に、着信 UPDATE が複数の UNIQUE /EXCLUDE インデックス PRIMARY KEY を含むに違反するか、複数の行が競合するように単一の EXCLUDE インデックスに違反する場合、PGD は multiple_unique_conflicts 競合を発生させます。

PGD は、遅延一意コンストレインをサポートしています。トランザクションがソースでコミットできる場合、競合が発生しない限り、ターゲットにクリーンに適用されます。ただし、遅延主キーをレプリカ ID として使用することはできないため、ユースケースはそれと複数の一意の制約の使用に関する警告によって既に制限されています。

### UPDATE/DELETE の競合

1 つのノードは、別のノードが同時に削除する行を更新できます。この場合、再生時に UPDATE /DELETE 競合が発生する可能性があります。

削除された行がまだ検出可能な場合 削除された行が VACUUM によって削除されなかった場合、update_recently_deleted 競合が生成されます。デフォルトでは、UPDATE はスキップされますが、この解像度を構成できます。詳細は、**欠落列の競合の解決** を参照してください。

ローカルノードのレプリケーションが遅延している場合、データベースは、UPDATE を受信するまでに削除された行をクリーンアップできます。この場合、PGD は UPDATE /DELETE 競合と *INSERT / UPDATE の競合* を区別できません。update_missing 競合が生成されます。

競合する DELETE および UPDATE の別のタイプは、行がローカルに更新された後に来る DELETE です。この状況では、結果は使用される競合検出の種類によって異なります。デフォルトの**オリジンの競合検出** を使用する場合、競合は検出されないため、 DELETE が適用され、行が削除されます。 **行バージョンの競合検出** を有効にすると、delete_recently_updated 競合が生成されます。delete_recently_updated 競合のデフォルトの解決策は、skip 削除です。ただし、解決を構成するか、それを処理するように競合トリガーを構成できます。

### INSERT / UPDATE の競合

デフォルトの非同期オペレーションモードを使用している場合、ノードは元の INSERT を受信する前に行の UPDATE を受信する場合があります。これは、3 つ以上のノードがアクティブな場合にのみ発生する可能性があります *3 つ以上のノードとの競合* を参照してください。

これが発生すると、update_missing 競合が生成されます。デフォルトの競合リゾルバーは insert_or_skip ですが、代わりに insert_or_error または skip を使用できます。insert-or-action を実行するリゾルバーは、可能な場合行全体が受信されたとき、UPDATE からのデータに基づいて、新しい行を INSERT 試行します。行の再構成を可能にするには、テーブルに REPLICA IDENTITY FULL が必要であるか、行にトーストされたデータが含まれていない必要があります。

トーストされたデータの詳細については、*TOAST サポートの詳細* を参照してください。

### INSERT/DELETE の競合

INSERT /UPDATE 競合と同様に、ノードは INSERT をまだ受け取っていない行で DELETE オペレーションを受信する場合があります。これも、3 つ以上のノードがセットアップされている場合にのみ可能です *3 つ以上のノードとの競合* を参照してください。

PGD は現在この競合タイプを検出できません。 INSERT オペレーションは競合タイプを生成せず、 INSERT が適用されます。

DELETE オペレーションは常に delete_missing 競合を生成します。これは、デフォルトでオペレーションをスキップすることにより解決されます。

### DELETE/DELETE の競合

DELETE / DELETE 競合は、2 つの異なるノードが同じタプルを同時に削除すると発生します。

このシナリオでは、常に delete_missing 競合が生成されます。これは、デフォルトで、オペレーションをスキップすることにより解決されます。

両方の DELETE 操作は同じ効果があるため、この競合は無害です。それらのいずれかを無視しても安全です。

### 3 つ以上のノードとの競合

1 つのノードが行を挿入し、それが 2 番目のノードに再生され、そこで更新されると、3 番目のノードは、最初のノードから INSERT を受信する前に、2 番目のノードから UPDATE を受信できます。このシナリオは、INSERT /UPDATE 競合です。

これらの競合は、UPDATE を破棄することにより処理されます。これにより、異なるノードに異なるデータが発生する可能性があります。これらは**多様な競合** です。

この競合タイプは、3 つ以上のマスターでのみ発生する可能性があります。少なくとも 2 人のマスターがアクティブに書き込みを行っている必要があります。

また、ノード 1 からノード 3 へのレプリケーションラグは、次の一連のアクションを許可するように十分に高い必要があります。

1. ノード 2 はノード 1 から INSERT を受信します
2. ノード 2 は UPDATE を実行します
3. ノード 3 はノード 2 から UPDATE を受信します
4. ノード 3 はノード 1 から INSERT を受信します

insert_or_error または場合によっては、 update_missing 競合タイプの insert_or_skip 競合リゾルバーを使用することは、これらの競合の実行可能な緩和戦略です。ただし、このオプションを有効にすると、INSERT /DELETE 競合の扉が開きます。

1. ノード 1 が UPDATE を実行します
2. ノード 2 が DELETE を実行します
3. ノード 3 はノード 2 から DELETE を受信します
4. ノード 3 はノード 1 から UPDATE を受信し、INSERT に変換します

これらが問題である場合は、テーブルまたはデータベースのフリーズ設定を調整して、update_recently_deleted として正しく検出されるようにすることをお勧めします。

*Eager Replication* **の効果一般** を使用して、これらの競合を防ぐこともできます。

INSERT /DELETE 競合は、3 つ以上のノードで発生する場合もあります。このような競合は、UPDATE が DELETE に置き換えられることを除き、 INSERT /UPDATE と同じです。これにより、delete_missing 競合が発生する可能性があります。

PGD は、 update_missing 競合で発生するように、各 INSERT を最近削除されたチェックにすることを選択できます。ただし、これを行うコストは大部分のユーザーにペナルティを与えるため、現時点では代わりに delete_missing ログに記録されます。

将来のリリースでは、delete_missing 競合が発生したときに LiveCompare を使用した再チェックにより、INSERT /DELETE 異常を自動的に解決します。アプリケーションは、bdr.conflict_history_summary ビューを確認することにより、これらを手動で実行できます。

これらの競合は、2 つの主な問題のユースケースで発生する可能性があります。

- INSERT の後にすぐに DELETE が続きます、キューイングアプリケーションで使用できます
- テーブルの主キー識別子が再利用される場合

これらのケースはいずれも一般的なものではありません。これらの問題のあるユースケースが発生した場合、影響を受けるテーブルをレプリケートしないことをお勧めします。

PGD は識別子の一意に依存してレプリケーションが正しく動作するため、後者の場合に問題があります。

同じ一意の識別子を挿入、削除、および後で再利用するアプリケーションは、問題を発生する可能性があります。これは、ABA problem と呼ばれます。PGD は、行が現在の行、最後の行、またはかなり古い行であるかを知る方法がありません。

一意の識別子の再利用は、時間の経過とともに一意の識別を妨げ、監査、トレーサビリティ、および賢明なデータ品質を妨げるため、ビジネス上の問題でもあります。アプリケーションは一意の識別子を再利用する必要はありません。

変更がシステム全体に渡されるまでにかかる時間間隔で発生する識別子の再利用は、問題を発生させます。この時間は通常のオペレーションでは短いかもしれませんが、ダウンノードではその間隔を数時間または数日に延長できます。

アプリケーションは一意の識別子を再利用しないことをお勧めします。その場合、1 年未満は再利用を避けるための措置を講じます。

この問題は、シーケンスまたは UUID を使用するアプリケーションでは発生しません。

### 外部キーコンストレインの競合

適用されるリモートトランザクションと既存のローカルデータの競合は、FOREIGN KEY FK コンストレインでも発生する可能性があります。

PGD は session_replication_role = 'replica' を使用して変更を適用するため、変更を適用するときに外部キーは再チェックされません。アクティブ/アクティブ環境では、参照テーブルへの挿入と同時に参照先テーブルで削除が発生すると、この状況により FK 違反が発生する可能性があります。このシナリオは、INSERT /DELETE 競合に似ています。

シングルマスター Postgres では、参照先テーブルの値を参照する INSERT /UPDATE は、行レベルのロックを取得する前に、DELETE 操作が完了するまで待機する必要があります。 DELETE が参照値を削除する場合、INSERT /UPDATE は FK チェックに失敗します。

マルチマスター PGD において。ノード間の行レベルのロックはありません。参照テーブルの INSERT は、参照テーブルの DELETE の背後で待機しないため、両方のアクションが同時に発生できます。したがって、参照テーブルのあるノードの INSERT /UPDATE は、別のノードの参照先テーブルの DELETE として同時に値を使用できます。したがって、結果は、参照先テーブルに存在しない参照テーブルの値です。

実際、この状況は、 DELETE 操作が、一般的な操作ではない参照テーブルでの DELETE 操作とは別のトランザクションの参照対象テーブルで発生した場合に発生します。

Orders -> OrderItems のような親子関係では、これは一般的ではありません。OrderItem を完全に削除するよりも、キャンセル済みとしてマークする可能性が高くなります。参照/ルックアップデータの場合、新しいファクトデータに同じ値を使用すると同時にエントリを完全に削除することはまれです。

ぶら下がった FK が可能ですが、一般的にこのリスクは非常に低いです。したがって、PGD は、この場合をカバーする汎用ソリューションを課しません。これが発生する状況を理解すると、2 つの解決策が考えられます。

最初の解決策は、一般的に一度に 1 つのノードのみから変更される、変更の頻度が低い、または変更のコンカレンシーがアプリケーションを介する場合に、密接に関連するエンティティに FK の使用を制限することです。このアプローチでは、アプリケーションレベルでの FK 違反を回避します。

2 番目の解決策は、 PGD 提供のファンクション bdr.ri_fkey_trigger() および bdr.ri_fkey_on_del_trigger() を使用して、この場合から保護するトリガーを追加することです。BEFORE トリガーとして呼び出された場合、これらのファンクションは FOREIGN KEY 情報を使用して、SET NULL 制約があるかのように参照列を NULL に設定することにより FK 異常を回避します。このアプローチでは、1 つのトリガーですべての FK を再チェックするため、FK 違反を防ぐには、テーブルごとにトリガーを 1 つだけ追加する必要があります。

例として、Fact と RefData の 2 つのテーブルがあるとします。Fact には、RefData を参照する FK があります。Fact は参照テーブルであり、RefData は参照されるテーブルです。各テーブルにトリガーを 1 つ追加する必要があります。

RefData で参照されている行が既に削除されている場合、Fact に列を NULL に設定するトリガーを追加します。

```
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;
```

RefData テーブルで DELETE が発生したときに、Fact の列を NULL に設定するトリガーを追加します。

```
CREATE TRIGGER bdr_replica_fk_d_trg
    BEFORE DELETE ON refdata
    FOR EACH ROW
    EXECUTE PROCEDURE bdr.ri_fkey_on_del_trigger();
```

(次のページに続く)

(前のページからの続き)

```
ALTER TABLE refdata
    ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

両方のトリガーを追加すると、ダングリング外部キーが回避されます。

### TRUNCATE の競合

TRUNCATE は、すべての行の DELETE と同様に動作しますが、行ごとに削除するのではなく、テーブルデータを物理的に削除することによりこのアクションを実行します。その結果、行レベルの競合処理は利用できないため、明確な競合が存在する場合でも、TRUNCATE コマンドは他の DML アクションとの競合を生成しません。

その結果、TRUNCATE への他のノードで別の DML が同時に実行される場合、再生の順序により発散的な変更が発生する可能性があります。

次のいずれかのアクションを実行できます。

- TRUNCATE が他のコンカレント DML とともに実行されていないことを確認します。LiveCompare を利用して、このような不一致を強調表示します。
- TRUNCATE を、 WHERE 句のない DELETE ステートメントに置き換えます。このアプローチは、大規模なテーブルではパフォーマンスが低下する可能性があります。
- Set bdr.truncate_locking = 'on' TRUNCATE コマンドのロック動作を設定します。この設定は、TRUNCATE が bdr.ddl_locking 設定に従うかどうかを決定します。すべてのノードが起動している必要があるため、これは TRUNCATE のデフォルト動作ではありません。この構成は、すべての場合に使用できないか、必要な場合があります。

### ロールとテーブルスペースの違いのデータの競合

競合は、ノードにロールのようなグローバル Postgres システム全体のデータが異なる場合にも発生する可能性があります。この競合により、操作主に DDL が発生し、1 つのノードで正常に実行してコミットできますが、他のノードへの適用に失敗する場合があります。

たとえば、ノード 1 には fred という名前のユーザーがあり、そのユーザーはノード 2 で作成されたものではない場合があります。ノード 1 の fred がテーブルを作成すると、テーブルは所有者を fred に設定して複製されます。DDL コマンドがノード 2 に適用されると、fred という名前のユーザーが存在しないため、DDL は失敗します。この障害により、Postgres ログにエラーが生成されます。

PGD が実行されているデータベースにユーザー fred を作成することにより、この競合を解決するには、管理者の介入が必要です。将来的にこれを解決するように bdr.role_replication = on を設定できます。

### ロックの競合とデッドロックアボート

PGD ライタプロセスは通常のユーザーセッションと同じように動作するため、行とテーブルのロックに関する通常のルールの対象となります。これにより、PGD ライタプロセスがユーザートランザクションまたは相互に保持するロックで待機する場合があります。

関連するロックには次のものが含まれます。

- ユーザーセッションによる明示的なテーブルレベルのロック LOCK TABLE ...

- ユーザーセッションによる明示的な行レベルのロック SELECT ... FOR UPDATE/FOR SHARE
- ローカルアクティビティまたは他のノードからのレプリケーションからの行 UPDATE 、INSERT 、または DELETE 操作のための暗黙的ロック

PGD ライタプロセスは、ユーザートランザクションとデッドロックになる場合があります。ユーザートランザクションは、ライタプロセスが保持するロックで待機しています。その逆も可能です。2 つのライタプロセスが互いにデッドロックする場合もあります。Postgres のデッドロック検出機能は、問題のあるトランザクションのいずれかに介入して終了します。PGD ライタプロセスが終了すると、再試行され、通常は成功します。

これらの問題はすべて一時的なもので、通常、管理者のアクションは必要ありません。ライタプロセスがアイドルなユーザーセッションのロックの背後で長時間スタックしている場合、管理者はユーザーセッションを終了して、レプリケーションフローを再度取得できます。ただし、これは、ユーザーが別のユーザーセッションに影響を与える長いロックを保持していることと変わりません。

Postgres の *log_lock_waits* 機能を使用すると、ロック関連の再生ストールを特定するのに役立ちます。

### 多様な競合

発散競合は、別のノードで同じであるはずのデータが予期せず異なる場合に発生します。多様な競合は発生すべきではありませんが、執筆時点でこのような競合すべてを確実に防止できるわけではありません。

すべてのノードが変更を再生する前に別のノードが同じ行のキーを変更した場合、行の PRIMARY KEY を変更すると、発散競合が発生する可能性があります。主キーの変更を避けるか、1 つの指定ノードでのみ変更します。

行データを含む多様な競合では、一般に、一方のノードのデータを他方のノードと一貫性があるように手動で調整する管理者のアクションが必要です。文書に従って PGD を使用し、安全でないとマークされた設定または機能を回避する限り、このような競合は発生しません。

管理者は、このような競合を手動で解決する必要があります。競合の性質によっては、bdr.ddl_replication や bdr.ddl_locking などの詳細オプションの使用が必要になる場合があります。ただし、これらのオプションを不注意に使用すると、状況をさらに悪化させ、汎用命令では対処できない競合が発生する可能性があります。

### TOAST サポートの詳細

Postgres は、 TOAST と呼ばれる大きな列にアウトオブラインストレージを使用します。

論理デコーディング PGD がその上に構築されるおよび論理レプリケーションでの TOAST 値の処理は、テーブルのメイン行の一部として保存されるインラインデータとは異なります。

TOAST 値は、値が変更された場合にのみトランザクションログ WAL に記録されます。トーストされた列の値を変更しなかった UPDATE ステートメントは、その列のない行を生成するため、特に UPDATE の競合を処理する場合に問題が発生する可能性があります。*INSERT / UPDATE の競合* で述べたように、PGD は、insert_or_error を使用して update_missing 競合が解決され、欠落している TOAST 列がある場合にエラーを報告します。

ただし、非同期レプリケーションと同時ワークロードの場合、これよりも微妙な問題が発生します。Eager トランザクションは影響を受けません。たとえば、A、B、および C と呼ばれる 3 つのノードを持つ EDB Postgres 分散クラスターでの次のワークロードを想像してください。

1. ノード A で txn A1 は UPDATE SETcol1 = ’toast data…’ を実行し、最初にコミットします。
2. ノード B で txn B1 は UPDATE SET other_column = ‘anything else’ ; そして、A1 の後にコミットします。
3. ノード C では、ノード A への接続が遅れます。
4. ノード C では txn B1 が最初に適用されますが、col1 の TOASTed 列を見逃しますが、競合せずに適用されます。
5. ノード C では、txn A1 は update_origin_change で競合し、スキップされます。
6. ノード C は、A1 からのトーストされたデータを永久に逃します。

PGD を使用する場合、このシナリオは通常問題になりません。PGD は、マルチマスターにビルトイン論理レプリケーションまたはプレーン pglogical のいずれかを使用する場合です。PGD は、 TOAST 列変更を最近レプリケートし、ローカル UPDATE がそうでない行にローカル UPDATE を検出すると、TOAST 列の独自のロギングを追加します。 t TOAST を変更するしたがって、PGD は、異なるノード間でトーストされたデータの不一貫性を防ぎます。この状況により、マルチプルのノードで更新が発生した場合、つまりタプルのオリジンが変更された場合、WAL ログの増加が発生します。通常、PGD AlwaysOn アーキテクチャの場合と同様に、すべての更新が単一のノードから行われる場合、追加の WAL オーバーヘッドはゼロです。

**注釈**

メインテーブルで同じことをせずに TOAST テーブルのみで `VACUUM FULL` または `CLUSTER` を実行すると、追加のロギングが機能するために必要なメタデータが削除されます。これは、このようなステートメントの後の短期間、これらの同時実行問題に対する保護が存在しないことを意味します。

**警告**

TOAST の追加 WAL ロギングは、標準の Postgres の `BEFORE UPDATE` トリガーを使用して実行されます。このトリガーは、テーブル上のすべての `BEFORE UPDATE` トリガーの中で、トリガー名に基づいて、アルファベット順に最後にソートする必要があります。これを簡単にするために `zzzz_bdr_` という接頭辞が付いていますが、その後にソートされる名前でトリガーを作成しないようにしてください。そうしないと、同時実行の問題から保護されません。

insert_or_error 競合解決の場合、引き続き REPLICA IDENTITY FULL の使用が必要です。

トーストされた列に関連するこれらの問題は、 REPLICA IDENTITY FULL を使用するテーブルには影響しません。この設定では、行全体がキーの一部と考えられるため、キーの一部としてトースト値が常にログに記録されます。PGD は、新しい行を再構成し、キー行から欠落しているデータを埋めることができます。その結果、REPLICA IDENTITY FULL を使用すると、WAL サイズが大幅に増加する可能性があります。

### Conflict detection

PGD は、競合検出のための次のメカニズムを提供します。

- オリジンの競合検出 デフォルト
- 行バージョンの競合検出

- *Column-level conflict detection*

### オリジンの競合検出

オリジン競合検出は、トランザクションの発生元のノードに記録されたコミットタイムスタンプを使用および依存します。これには、クロックが正しく動作するか、2 つのノード間の最速のメッセージの許容範囲内に同期している必要があります。これが当てはまらない場合、競合解決はより前にあるノードを優先する傾向があります。パラメーター bdr.maximum_clock_skew および bdr.maximum_clock_skew_action を使用して、ノード間のクロックスキューを管理できます。

行オリジンは、track_commit_timestamp = on の場合にのみ使用できます。

競合は、レプリケーション元が変更されたかどうかに基づいて最初に検出されるため、競合と判明しない可能性のある状況で競合トリガーが呼び出されます。したがって、このメカニズムは正確ではありません、偽陽性の競合を生成する可能性があります。

元の情報は、行がフリーズされた時点まででのみ利用できます。凍結された後に行に到着する更新は競合を発生させないため、すべての場合に適用されます。これは bdr_init_physical によって新しいノードを追加する場合の通常のケースであるため、競合が発生すると、その場合に多くの偽陽性の結果が発生します。

オフラインだったノードが再接続してデータ変更の送信を開始すると、新しく到着した更新が更新する凍結行よりも古い場合、発散エラーが発生する可能性があります。挿入と削除は、この状況の影響を受けません。

*Node restart and down node recovery* で説明しているように、長時間の停止のためにノードをダウンしたままにしないことをお勧めします。

EDB Postgres Extended Server および EDB Postgres Advanced Server では、ノードがダウンしている間、PGD は行のフリーズを抑制します。このメカニズムはこの状況を適切に処理するため、パラメーター設定を変更する必要はありません。

Postgres の他のバリアントでは、この状況をある程度注意して管理する必要がある場合があります。

フリーズは、通常、バキュームされる行が現在の xid から vacuum_freeze_min_age xid より古い場合に発生します。これは、これらのパラメーターに適切に高い値を構成する必要があることを意味します。

- vacuum_freeze_min_age
- vacuum_freeze_table_age
- autovacuum_freeze_max_age

トランザクションレートに基づいて値を選択し、データベースノードから競合データを削除する前にダウンタイムの猶予期間を与えます。たとえば、vacuum_freeze_min_age が 5 億に設定されている場合、1000TPS を実行するノードは、競合データが削除されるまで 5.5 日強の間ダウンする可能性があります。CommitTS データ構造は、その設定で 5 GB のディスク上の領域を必要とするため、トランザクションレートの低いシステムはより低い設定からメリットを得ることができます。

最初に、推奨される設定は次のとおりです。

```
#  1 billion = 10GB
```

(次のページに続く)

(前のページからの続き)

```
autovacuum_freeze_max_age = 1000000000

vacuum_freeze_min_age = 500000000

#  90% of autovacuum_freeze_max_age

vacuum_freeze_table_age = 900000000
```

次のことに注意してください。

- autovacuum_freeze_max_age はノード起動時にのみ設定できます。
- vacuum_freeze_min_age を設定できるため、低い値を使用すると行が早期にフリーズし、競合が無視される場合があります。個々のテーブルに autovacuum_freeze_min_age および toast.autovacuum_freeze_min_age を設定することもできます。
- CLUSTER または VACUUM FREEZE コマンドを実行しても、行が早期にフリーズされ、競合が無視される場合があります。

### 行バージョンの競合検出

PGD は、行バージョン管理を使用し、ノードのシステムクロックとは無関係に競合検出を行うオプションを提供します。

行バージョンの競合検出には、3 つのことを有効にする必要があります。これらの手順のいずれかが正しく実行されない場合、**オリジンの競合検出** が使用されます。

- 行バージョンの競合検出を使用するすべてのテーブルで REPLICA IDENTITY FULL を有効にします。
- bdr.alter_table_conflict_detection を使用して、テーブルの行バージョンの追跡を有効にします。このファンクションは、指定した名前の列と、新しい列値を管理する UPDATE トリガーを追加します。列は INTEGER タイプとして作成されます。

カウンタは UPDATE でのみインクリメントされますが、この技術により、UPDATE と DELETE の両方の競合検出が可能になります。

このアプローチはランポートタイムスタンプに似ており、競合検出の ABA 問題を完全に防止します。

> **注釈**
>
> 行レベルの競合解決は、行バージョン管理を使用しても **欠落列の競合の解決** 構成に基づいて処理されます。行バージョンの生成方法は、競合を検出する場合にのみ役立ちます。行のどのバージョンが新しいかに関する信頼できる情報として依存しないでください。

特定のテーブルに使用されている現在の競合検出ストラテジーを判断するには、ビュー bdr.tables の列 conflict_detection を参照してください。

現在の競合検出ストラテジーを変更するには、 bdr.alter_table_conflict_detection を使用します。

### Conflict resolution

ほとんどの競合は自動的に解決できます。PGD は、last-update-wins メカニズム、より正確には update_if_newer 競合リゾルバーにデフォルトで設定されます。このメカニズムでは、競合検出に使用される同じコミットタイムスタンプに基づいて、2 つの競合する行の最後に挿入または変更された行を保持します。特定のコーナーケースのシナリオでの動作は、bdr.create_node_group または bdr.alter_node_group に使用される設定によって異なります。

PGD では、 bdr.alter_node_set_conflict_resolver を使用して競合解決のデフォルト動作をオーバーライドできます。

### Conflict logging

PGD 6.0 以降、競合ログはデフォルトで有効になっていません。ノードごとに競合ログを有効にできます。これにより、特定のノードで発生した競合のログを記録できます。これは、デバッグやモニタリングの目的に役立ちます。

名前付けノードでこのコマンドを実行して、その特定のノードでのすべての競合のログを有効にします。すべてのノードでロギングを有効にする場合は、PGD グループの各ノードでこのコマンドを実行します。

```
SELECT bdr.alter_node_set_log_config(`nodename`, false, true, NULL, NULL);
```

ノード名コントロールの後の他のパラメーターは、それぞれ、競合を PGD ログファイルに記録するかどうか、および競合を bdr.conflict_history テーブルに記録するかどうかを指定します。最後の 2 つのパラメーターは、ログに記録する競合タイプと競合の解決を指定する文字列の配列を受け取ります。両方の完全なリストについては、リファレンスセクションの Conflicts を参照してください。これらのパラメーターを NULL に設定すると、PGD はすべての競合タイプと解決策をログに記録します。

PGD は、すべての競合のログを bdr.conflict_history テーブルに記録します。bdr.alter_node_set_log_config を使用して、この動作をより詳細に変更できます。

#### 競合報告

レポートのテーブルに記録された競合を要約できます。レポートにより、アプリケーションの所有者は競合を特定、理解、解決し、競合を防ぐためのアプリケーションの変更を導入できます。

```
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 は、2 つのデータベースを比較して、それらが同一であることを確認するように設計されたユーティリティプログラムです。

LiveCompare は PGD スタックの一部として含まれ、PGD ノードの任意のペアを対象とできます。デフォルトでは、レプリケートされたすべてのテーブルを比較し、違いを報告します。LiveCompare は、Postgres や Oracle などの非 PGD データソースでも動作します。

LiveCompare を使用して、着信行を継続的に監視することもできます。コンテキスト情報を失うことなく停止および開始できるため、都合の良いときに実行できます。

LiveCompare では、複数のテーブルを同時にチェックできます。いくつかのテーブル、またはテーブル内の行のセクションのみのチェックを許可するように構成できます。チェックは、最初に行全体のハッシュを比較することにより実行されます。異なる場合、LiveCompare は行全体を比較します。LiveCompare は、有用なサイズのバッチで行を比較することにより、オーバーヘッドを回避します。

違いが見つかった場合は、時間の経過とともに再チェックできるため、最終的な一貫性の遅延が発生します。

詳細については、 LiveCompare ドキュメントを参照してください。

### Column-level conflict detection

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、ローカルまたはリモートのタプルのいずれかが選択され、他方は破棄されます。たとえば、2 つの競合する変更のコミットタイムスタンプを比較し、新しい方が保持される場合があります。このアプローチでは、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスを確立します。

ただし、少なくとも一部の場合、行レベルではなく列レベルで競合を解決することが適切な場合があります。

- **概要** は、行レベルの競合解決とは対照的に列レベルの競合解決を導入し、行レベルの競合解決よりも適している場合を提案します。
- *Enabling and disabling column-level conflict resolution* は、列レベルの競合解決を有効にする例を提供し、列レベルの競合解決を有効にしてテーブルをリストする方法について説明します。
- *Timestamps in column-level conflict resolution* は、 `column_modify_timestamp` と `column_commit_timestamp` の使用の違いを説明し、列レベルの競合解決に関連付けられたタイムスタンプを選択および検査する方法を示します。

### Overview

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、ローカルまたはリモートのタプルのいずれかが選択され、他方は破棄されます。たとえば、2 つの競合する変更のコミットタイムスタンプを比較し、新しい方が保持される場合があります。このアプローチでは、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスを確立します。

ただし、少なくとも一部の場合、行レベルではなく列レベルで競合を解決することが適切な場合があります。

### 列レベルで解決する場合

テーブル t に 2 つの整数列 a および b と、単一の行 (1,1) がある単純な例を考えます。 1 つのノードで次を実行します。

```
UPDATE t SET a = 100
```

別のノードでは、前述の UPDATE を受信する前に、次のことを同時に実行します。

```
UPDATE t SET b = 100
```

> **注釈**
>
> *UPDATE* によって変更される属性は、トリガー内の古い行と新しい行を比較することにより決定されます。これは、属性が値を変更しない場合、明示的に設定されていても変更として検出されないことを意味します。たとえば、 *UPDATE t SET a = a* は、 *a* をどの行でも変更されたものとしてマークしません。同様に、*UPDATE t SET a = 1* は、既に `1` に設定されている行の `a` を変更済みとしてマークしません。

このシーケンスにより、UPDATE-UPDATE 競合が発生します。*update_if_newer* 競合解決では、コミットタイムスタンプが比較され、新しい行バージョンが保持されます。2 番目のノードが最後にコミットしたと仮定すると、結果は (1,100) であり、列 a への変更を効果的に破棄します。

多くのユースケースでは、この動作は望ましく、予想されます。ただし、一部のユースケースでは、これが問題になる場合があります。たとえば、アプリケーションの各部分が別のノードに接続され、共有テーブルの列の専用サブセットを更新するマルチノードクラスターを考えます。その場合、さまざまなコンポーネントが競合し、変更を上書きする可能性があります。

このようなユースケースでは、特定のテーブルの競合を列レベルで解決することがより適切である場合があります。これを行うために、PGD は各列の最後の変更のタイムスタンプを個別に追跡し、それを使用して最新の値を選択し、基本的に update_if_newer を実行します。

前の例に適用すると、どのノードもそのような行を認識しないにもかかわらず、結果は両方のノードで (100,100) になります。

列レベルの競合の解決について考えるとき、テーブルを垂直にパーティション化して、各更新が 1 つのスライスのデータのみに影響を与えるようにすると便利です。このアプローチにより、列のさまざまなサブセットへの変更間の競合が排除されます。実際、垂直パーティショニングは、列レベルの競合解決の実用的な代替品です。

列レベルの競合解決には、テーブルに REPLICA IDENTITY FULL が必要です。*bdr.alter_table_conflict_detection()* ファンクションはそれを確認し、この設定が存在しない場合はエラーで失敗します。

#### 列レベルの競合解決の特別な問題

列を独立して扱うことにより、すべての変更が同じノードで発生する場合、不可能な方法でコンストレインに違反するのが簡単になります。例、次のようなテーブルを考えます。

```
CREATE TABLE t (id INT PRIMARY KEY, a INT, b INT, CHECK (a > b));
INSERT INTO t VALUES (1, 1000, 1);
```

1 つのノードが次のことを行うと仮定します。

```
UPDATE t SET a = 100;
```

別のノードは同時に次のことを行います。

```
UPDATE t SET b = 500;
```

これらの更新はそれぞれ、最初の行で実行されたときに有効であるため、各ノードに渡されます。しかし、他のノードにレプリケートする場合、結果の行は CHECK (a > b) 制約に違反し、問題が手動で解決されるまでレプリケーションは停止します。

### CRDT データ型を使用した列レベルの競合の処理

デフォルトでは、列レベルの競合解決は、より高いタイムスタンプを持つ値を選択し、他の値を破棄します。ただし、さまざまなより複雑な方法で競合を調整することはできます。たとえば、情報を破棄せずに競合する値をマージできる **オペレーションベースの** *CRDT* **タイプ** *CmCRDT* を使用できます。

### Enabling and disabling column-level conflict resolution

*bdr.alter_table_conflict_detection()* ファンクションは、列レベルの競合解決を管理します。

#### bdr.alter_table_conflict_detection を使用して列レベルの競合解決を有効にする

bdr.alter_table_conflict_detection ファンクションは、テーブル名と列名を引数として受け取ります。列は column_modify_timestamp 列としてテーブルに追加されます。また、ファンクションは、各変更の前に新しい列のタイムスタンプを維持する 2 つのトリガー BEFORE INSERT および BEFORE UPDATE も追加します。

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

(次のページに続く)

(前のページからの続き)

```
 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()
```

新しい列は、デフォルト値で NOT NULL を指定します。これは、ALTER TABLE ... ADD COLUMN がテーブルリライトを実行しないことを意味します。

> **注釈**
>
> *bdr.column_timestamps* データ型の列を他の目的で使用することは避けてください。悪影響を与える可能性があります。たとえば、テーブルを列レベルの競合解決に切り替えますが、これはトリガーがなしでは正しく機能しません。

### 列レベルの競合解決を使用したテーブルのリスト

次のクエリーを使用して、列レベルの競合解決が有効になっているテーブルをリストできます。

```
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);
```

このクエリーは、bdr.column_timestamp 型の列の存在を検出します。

### Timestamps in column-level conflict resolution

列レベルの競合の解決は、テーブルに含まれるタイムスタンプ列によって異なります。

#### column_modify_timestamp と column_commit_timestamp の比較

2 つの列レベルの競合検出方法のいずれかを選択すると、変更された列とタイムスタンプのマッピングを含む列がテーブルに追加されます。

タイムスタンプマッピングを格納する列は自動的に管理されます。結果は予測できない場合があるため、クエリーで値を指定またはオーバーライドしないでください。可能な場合、ユーザーが値をオーバーライドしようとしても無視されます。

テーブルの列タイムスタンプを有効または無効にする場合、コードは DDL ロックを使用して、スイッチ前からの保留中の変更がないことを確認します。このアプローチでは、両方のタプルのタイムスタンプと競合のみが表示されないようにします。そうしないと、コードは予期せずローカルタプルのタイムスタンプとリモートタプルの NULL を認識する場合があります。また、変更がすべてのノードで同じ方法列レベルまたは行レベルで解決されることも保証します。

#### column_modify_timestamp

競合検出方法として column_modify_timestamp が選択されている場合、変更された列に割り当てられたタイムスタンプは、現在のタイムスタンプであり、select_clock_timestamp() を実行して取得する値と似ています。

このアプローチは簡単で、多くの場合、たとえば、競合する行が列の重複しないサブセットを変更する場合、それは正しいです。ただし、そのシンプルさは予期しない効果をもたらす可能性があります。

たとえば、 UPDATE が複数の行に影響を与える場合、UPDATE の実行中クロックは刻み続けます。したがって、各行は、UPDATE によって同時に変更されている場合でも、わずかに異なるタイムスタンプを取得します。この動作は、さまざまなノードで実行される変更がどのようにインターリーブされるかに応じて、同時の変更のエフェクトがさまざまな方法で「混合」される可能性があることを意味します。

もう 1 つ考えられる問題は、クロックスキューです。別のノードのクロックがドリフトすると、それらのノードによって生成されたタイムスタンプもドリフトします。このクロックスキューは、タイムスタンプが明らかに切り替わっているため、新しい変更が破棄されるなどの予期しない動作を誘発する可能性があります。ただし、パラメーター bdr.maximum_clock_skew および bdr.maximum_clock_skew_action を使用してノード間のクロックスキューを管理できます。

現在のタイムスタンプはコミットタイムスタンプと無関係であるため、競合の解決にそれを使用することは、結果がコミット順序と等価ではないこと、つまりシリアル化できないことを意味します。

現在のタイムスタンプを使用して変更またはコミットを注文する場合、2 つ以上のノードがたまたま同じタイムスタンプを生成したため、競合する変更のタイムスタンプがまったく同じになる場合があります。このリスクは、通常の行レベルの競合解決でも発生する可能性があるため、列レベルの競合解決に固有ではありません。この状況では、ノード ID がタイブレーカーとして使用されます。ノード ID が高い方が優先します。このアプローチでは、すべてのノードに同じ変更が適用されます。

### column_commit_timestamp

column_commit_timestamp で指定された実際のコミットタイムスタンプを競合検出方法として使用することもできます。このアプローチには、UPDATE で行われたすべての変更で同じであるコミット時間を使用する利点があります。

> **注釈**
>
> 将来的にステートメントトランザクションが追加される可能性があります。これにより、コンカレントステートメントまたはトランザクションの混合効果に関する問題が解決されます。それでも、これらのオプションはいずれも、コミットオーダーと同等の結果を生成することはできません。

### カラムのタイムスタンプの検査

変更された列のタイムスタンプを格納する列は、トリガーによって維持されます。直接変更しないでください。たとえば、競合がどのように解決されたかを調査しているときに、現在のタイムスタンプの値を検査すると役立つ場合があります。

> **注釈**
>
> タイムスタンプマッピングはトリガーによって維持され、トリガーの実行順序が重要です。カスタムトリガーがタプルを変更し、 pgl_clcd_トリガーの後に実行される場合、変更された列は正しく検出されません。これにより、誤った競合解決が発生する可能性があります。トリガー内のタプルを変更する必要がある場合は、 pgl_clcd_トリガーの前に実行されていることを確認します。

次の機能は、タイムスタンプを検査するために役立ちます。

### bdr.column_timestamps_to_text(bdr.column_timestamps)

このファンクションは、タイムスタンプマッピングの人間が判読可能な表現を返し、値をテキストにキャストするときに使用されます。

```
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)

このファンクションは、タイムスタンプマッピングの JSONB 表現を変換し、値を jsonb にキャストするときに使用されます。

```
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)

このファンクションは、最新のトランザクションが既にコミットしている場合、それによって変更された属性のコミットタイムスタンプを使用してマッピングを更新します。これは、コミットタイムスタンプを使用する場合にのみ重要です。たとえば、この場合、最後のトランザクションは 2 番目の属性を更新しました 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

競合のないレプリケートデータ型 CRDT は、従来の解決のように行の 1 つを破棄するのではなく、同時に変更された行からの値のマージをサポートします。

- *CRDTs Overview* は、テーブルで CRDT を使用する方法、構成オプション、CRDT の例を含む CRDT の概要を提供します。
- *Using CRDTs* は、テーブルで CRDT を使用する方法を調査し、いくつかの構成オプションをレビューし、CRDT のいくつかの例とその動作をレビューします。

- *Operation-based and state-based CRDTs* は、オペレーションベースと状態ベースの CRDT の違いをレビューします。
- *CRDT Disk-space requirements* は、CRDT、特に状態ベースの CRDT のディスクサイズの考慮事項をカバーしています。
- *CRDTs vs conflict handling/reporting* は、CRDT での競合処理と報告がどのように動作するかについて説明します。
- *Resetting CRDT values* は、CRDT 値をリセットする際の課題について説明し、これを正常に行うためのガイダンスを提供します。
- *Implemented CRDTs* は、PGD で使用可能な 6 つの CRDT のそれぞれを実装例とともに詳細に説明します。

### CRDTs Overview

#### CRDT の概要

競合のないレプリケートデータ型 CRDT は、従来の解決のように行の 1 つを破棄するのではなく、同時に変更された行からの値のマージをサポートします。各 CRDT タイプは、`bdr.crdt_handlers` カタログに追加される追加のコールバックを備えた別の PostgreSQL データタイプとして実装されます。マージプロセスは、適用側の PGD ライタ内で発生し、ユーザーアクションは必要ありません。

CRDT では、**列レベルの競合解決の特別な問題** で説明しているように、テーブルで列レベルの競合解決を有効にする必要があります。

PostgreSQL の## CRDT

*Implemented CRDTs* は、 `bdr` の一部として `bdr` スキーマにインストールされます。便宜上、基本的な演算子`+` 、`#` 、および`!` と多くの一般的な集計関数 `min` 、`max` 、`sum` 、および `avg` が `pg_catalog` で作成されます。したがって、`search_path` を調整せずに使用できます。

### Using CRDTs

#### テーブルで CRDT を使用する

CRDT を使用するには、 `integer` などの標準のビルトインデータ型ではなく、 CREATE/ALTER TABLE で特定のデータ型を使用する必要があります。たとえば、1 つの通常の整数カウンターと単一の行を含む次のテーブルを検討します。

#### 非 CRDT の例

```
CREATE TABLE non_crdt_example (
    id      integer             PRIMARY KEY,
    counter integer             NOT NULL DEFAULT 0
);

INSERT INTO non_crdt_example (id) VALUES (1);
```

2 つの異なるノードで次の SQL を同時に発行するとします。

```
UPDATE non_crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

両方の更新を適用した後、次のクエリを使用して結果の値を確認できます。

```
SELECT * FROM non_crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
    1 |         1
(1 row)
```

このコードは、 update_if_newer 競合リゾルバーが原因でインクリメントの 1 つが失われたことを示しています。

### CRDT の例

代わりに CRDT カウンターデータ型を使用するには、次の手順を実行します。

テーブルを作成しますが、カウンターデータ型として CRDT bdr.crdt_gcounter を使用します。

```
CREATE TABLE crdt_example (
    id      integer             PRIMARY KEY,
    counter bdr.crdt_gcounter   NOT NULL DEFAULT 0
);
```

列レベルの競合解決のためにテーブルを構成します。

```
ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_example,
            column_modify_timestamp, cts);
```

次に、この例の値を含む行を挿入します。

```
INSERT INTO crdt_example (id) VALUES (1);
```

ここで、以前と同様に、2 つのノードで同じ SQL を同時に発行するとします。

```
UPDATE crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

変更が適用されると、カウンターが更新を同時に実行できることがわかります。

```
SELECT id, counter FROM crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
```

(次のページに続く)

(前のページからの続き)

```
    1 |         2
(1 row)
```

この例は、CRDT が他の方法で競合する非同期同時更新に直面しても、アキュムレータ列が動作できることを示しています。

### 構成オプション

bdr.crdt_raw_value 構成オプションは、クエリが現在の値または CRDT タイプの完全な内部状態を返すかどうかを決定します。デフォルトでは、現在の数値のみが返されます。true に設定されている場合、クエリーは完全な状態の表現を結果ます。特殊なハッシュ演算子# を使用して、特殊な演算子デフォルト動作を使用せずに現在の数値のみを要求できます。bdr.crdt_raw_value = on を使用して完全な状態がダンプされた場合、値は bdr.crdt_raw_value = on でのみリロードできます。

> **注釈**
>
> *bdr.crdt_raw_value* は、クライアントに返されたデータ、つまり選択リストの単純な列参照のみのフォーマットを適用します。クエリーの他の部分の列参照たとえば `WHERE` 句または select リストの式などは、 # 演算子を使用する必要がある場合があります。

### さまざまな種類の CRDT

crdt_gcounter タイプは、例が示すように、 x = x + 1 などの再帰的 UPDATE SQL でのみ動作する状態ベースの CRDT タイプの例です。

CRDT の別のクラスは、*delta CRDT* タイプです。これらは **オペレーションベースの *CRDT* タイプ** *CmCRDT* の特別なサブクラスです。

デルタ CRDT では、値の更新は、同じノード上の以前の値と比較されます。次に、変更は他のすべてのノードにデルタとして適用されます。

```
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);
```

2 つのノードで次の SQL を同時に発行するとします。

```
UPDATE crdt_delta_example
   SET counter = 2          -- notice NOT counter = counter + 2
 WHERE id = 1;
```

両方の更新を適用した後、次のクエリを使用して結果の値を確認できます。

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
   id | counter
 -----+---------
    1 |       4
(1 row)
```

通常の integer 列の場合、結果は 2 になります。ただし、デルタ CRDT カウンターで行を更新する場合、古い行バージョンで開始し、新しい行バージョンを作成し、両方をリモートノードに送信します。そこで、そこにあるバージョンたとえばローカルバージョンなどと比較します。標準 CRDT は NEW と LOCAL バージョンをマージしますが、デルタ CRDT は OLD と NEW バージョンを比較し、デルタをローカルバージョンに適用します。

#### クエリのプランニングと最適化

重要な問題は、クエリのプランニングと最適化がこれらの新しいデータ型でどのように機能するかです。CRDT タイプは透過的に処理されます。 ANALYZE とオプティマイザーの両方が動作するため、他に何もせずに推定とクエリプランニングは正常に動作します。

### Operation-based and state-based CRDTs

#### オペレーションベースの CRDT タイプ CmCRDT

操作は明示的に転送されないが、リモートノードから受け取った古い行と新しい行から計算されるため、操作ベースの型の実装は簡単です。

現在、次のオペレーションベースの CRDT が実装されています。

- デルタカウンター `crdt_delta_counter <デルタカウンター *crdt_delta_counter*>` — bigint カウンターインクリメント/デクリメント
- デルタサム `crdt_delta_sum <デルタサム *crdt_delta_sum*>` — numeric sum インクリメント/デクリメント

これらの型は、デルタを計算する少しのコードで既存のデータ型を活用します。たとえば、crdt_delta_counter は bigint 上のドメインです。

このアプローチは、デルタを計算する方法が知られている型でのみ可能ですが、結果はシンプルで安価、スペースと CPU の両方の点で、いくつかの追加の利点があります。たとえば、基になるデータ型の演算子/構文を活用できます。

主な欠点は、非同期コンカレント環境ではこの値を確実にリセットできないことです。

> **注釈**

カスタムデータタイプを作成し、状態と最後のオペレーションを保存することにより、より複雑なオペレーションベースのタイプを実装できます。すべての変更はデコードされて転送されるため、複数の操作は必要ありません。しかし、その時点で、主な利点単純さ、既存のデータ型の再利用などは、領域要件を除き、状態ベースの型たとえば、まだリセット機能と比較して利点を得ることなく失われます。ノードごとの状態は必要ありません。

### 状態ベースの CRDT タイプ CvCRDT

状態ベースの型は、より複雑な内部状態を必要とするため、操作ベースの型のように通常のデータ型を直接使用できません。

現在、4 つの状態ベースの CRDT が実装されています。

- Grow 専用カウンター `crdt_gcounter <Grow 専用カウンター *crdt_gcounter*>` — `bigint` カウンターインクリメント専用
- Grow-only sum `crdt_gsum <Grow-only sum *crdt_gsum*>` — `numeric` sum/counter インクリメントのみ
- ポジティブネガカウンター `crdt_pncounter <ポジティブネガカウンター *crdt_pncounter*>` — `bigint` カウンターインクリメント/デクリメント
- 正負の合計 `crdt_pnsum <正負の合計 `crdt_pnsum`>` — `numeric` sum/counter インクリメント/デクリメント

内部状態には通常、ノードごとの情報が含まれるため、ディスク上のサイズは増加しますが、追加の利点が得られます。カスタムデータ型を実装する必要があるということは、より多くのコード in/out ファンクションと演算子を意味します。

利点は、値を確実にリセットする機能、変更が失われた場合のある程度の自己修復性正常に動作するクラスターでは発生しません、およびソースノード以外のノードから変更を受信できる機能です。

たとえば、ノード A で値が変更され、変更は B にレプリケートされますが、A と C 間のネットワークの問題により C にレプリケートされません。B が値を変更し、この変更が C にレプリケートされる場合、それには A からの元の変更。オペレーションベースの CRDT を使用すると、A-C ネットワーク接続が再び動作し始めるまで、ノード C は変更を受信しません。

CvCRDT の主な欠点は、**状態ベースの *CRDT* ディスク領域要求** の点でコストが高いことです。クラスターから既に削除されたノードを含む、ノードごとに少しの情報が必要です。状態 varlena タイプにシリアル化の複雑な性質は、CPU 使用量の増加を意味します。

### CRDT Disk-space requirements

重要な考慮事項は、CRDT タイプ、特にディスク上のサイズに関連するオーバーヘッドです。

### オペレーションベースの CRDT ディスク領域要求

*operation-based types* の場合、タイプは他のタイプの単なるドメインであるため、これは簡単です。ノードの数に関係なく、必要なディスク領域は同じです。

- `crdt_delta_counter` — `bigint` と同じ 8 バイト
- `crdt_delta_sum` — `numeric` と同じ変数、精度とスケールに依存します

操作ベースの CRDT タイプはノードごとの情報を保存しないため、ノードの数への依存関係はありません。

### 状態ベースの CRDT ディスク領域要求

*state-based types* の場合、状況はより複雑です。すべてのタイプは可変長であり、基本的に `bytea` 列として保存され、ヘッダーと、値を変更した各ノードの特定の量のノードごとの情報で構成されています。

`bigint` バリアントの場合、おおよそのサイズを計算する式は次のとおりです。

- `crdt_gcounter` — `32B (header) + N * 12B (per-node)`
- `crdt_pncounter` -— `48B (header) + N * 20B (per-node)`

`N` は、この値を変更したノードの数を示します。

`numeric` バリアントの場合、ヘッダー部分とノードごとの部分の両方に `numeric` 可変長値が含まれるため、正確な式はありません。保持する必要があるこのような値の数を知るには

- `crdt_gsum`
  - 固定 `20B (header) + N * 4B (per-node)`
  - 変数 `(2 + N)` `numeric` 値
- `crdt_pnsum`
  - 固定 `20B (header) + N *4B (per-node)`
  - 変数 `(4 + 2* N)` `numeric` 値

> **注釈**
>
> 複数のノードで値が更新されない場合、クラスター内のノードの数は関係ありません。また、更新が同時実行されたかどうか、競合が発生したかどうかも関係ありません。

さらに、これらのノードのうちのいくつがクラスターから既に削除されたかは関係ありません。状態を圧縮する方法はまだありません。

### CRDTs vs conflict handling/reporting

### CRDT タイプと競合処理

テーブルには CRDT 列と非 CRDT 列の両方を含めることができるため、ほとんどの列は非 CRDT であることが予想されるため、通常の競合解決と CRDT マージの両方を行う必要があります。

競合の解決は最初に発生し、保持するタプル applytuple と破棄するタプルを決定します。次にマージフェーズが発生し、CRDT 列のデータを破棄タプルから適用タプルにマージします。

> **注釈**
>
> この処理により、毎回マージが発生する必要があるため、単純な競合解決に比べて CRDT タイプのコストが多少高くなります。これは、競合解決で高速パスのいずれか現在のトランザクションで変更されたパスなどを使用できる場合でも、当てはまります。

### CRDT タイプと競合レポート

デフォルトでは、検出された競合は個別に報告されます。CRDT タイプがなければ、競合解決は本質的に使用可能な情報の半分を破棄するため、これは理にかなっています構成に応じてローカルまたはリモート行。これにより、データの損失が発生します。

CRDT タイプを使用すると、何も捨てずに情報の両方の部分を組み合わせることができ、データ損失の問題を排除します。このアプローチにより、競合報告が不要になります。

このため、CRDT マージによって競合を完全に解決できる場合、競合レポートはスキップされます。各列は、次の 2 つの条件の少なくとも 1 つを満たす必要があります。

- ローカルタプルとリモートタプルの値が同じ NULL または等しい。
- CRDT データ型を使用するため、マージできます。

> **注釈**
>
> CRDT 列はないが、ローカル/リモートタプルのすべての値が等しい場合にも、競合レポートはスキップされます。

### Resetting CRDT values

CRDT 値のリセットは可能ですが、特別な処理が必要です。クラスターの非同期の性質は、実装方法に関係なく、さまざまなノードが変更ストリームのさまざまな場所でリセット操作を認識する可能性があることを意味します。別のノードが同時にリセットを開始する場合があります。つまり、他のノードからリセットを観察する前に。

言い換えれば、リセット操作を正しく動作させるには、通常の操作に対して可換である必要があります。単一ノードでうまく機能する可能性のある値をリセットする多くの単純な方法は、この理由で失敗します。

### CRDT 値をリセットする際の課題

たとえば、値をリセットする最も簡単なアプローチは次のとおりです。

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

状態ベースの CRDT では、これは機能しません。他のノードの状態を破棄しますが、ローカルでのみです。リモートノードのマージファンクションによって追加され、値の発散を引き起こし、最終的に他のノードでの変更によりそれを受信します。

オペレーションベースの CRDT では、更新が-cnt の減算として解釈されるため、これは機能するように見える場合があります。ただし、それは同時リセットがない場合にのみ機能します。2 つのノードが同時にリセットを実行しようとすると、デルタが 2 回適用され、負の値を取得しますリセットからは予想されません。

DELETE + INSERT をリセットとして使用できるように見えるかもしれませんが、このアプローチにはいくつかの弱点もあります。行が同じキーで再挿入された場合、他のノードからの変更に関して操作ストリームの同じ位置ですべてのノードがそれを見るという保証はありません。PGD は、コンカレントケースでデータ異常を引き起こす可能性があるため、同じ主キー値の再利用を特にお勧めしません。

### CRDT 値のリセットを確実に処理する方法

状態ベースの CRDT タイプは、次のような特別な! 演算子を使用してリセットを確実に処理できます。

```
UPDATE tab SET counter = !counter WHERE ...;
```

「確実に」とは、値にマルチプルの同時リセットと発散という 2 つの問題がないことを意味します。

オペレーションベースの CRDT タイプは、*Eager Replication* の効果一般 を使用してのみ確実にリセットできます。これは、マルチプルの同時リセットを回避するためです。Eager Replication を使用して、いずれかの種類の CRDT を特定の値に設定することもできます。

### Implemented CRDTs

現在、6 つの CRDT データ型が実装されています。

- Grow 専用のカウンターと合計
- ポジティブ/ネガティブカウンタと合計
- デルタカウンターとサム

カウンタと合計は、カウンタタイプが整数ベース `bigint` であるのに対し、sum タイプが 10 進数ベース `numeric` であることを除き、ほとんど同じ動作をします。

次のクエリーを使用して、現在実装されている CRDT データ型をリストできます。

```
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 専用カウンター crdt_gcounter

- 非負の値のインクリメントのみをサポート `value + int` および `counter + bigint` 演算子。
- `#` 演算子を使用するか、`bigint` にキャストすることにより、カウンターの現在値を取得できます。
- `counter = value` のような単純な割り当てとは互換性がありません。これは、新しい値がアプリケーションのどこかで計算される場合の一般的なパターンです。
- `!` 演算子 `counter = !counter` を使用してカウンターを単純にリセットできます。
- `crdt_gcounter_to_text` を使用して内部状態を検査できます。

```
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
```

(次のページに続く)

(前のページからの続き)

```
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`

- 負でない値のインクリメントのみをサポート `sum + numeric` 。
- `#` 演算子を使用するか、`numeric` にキャストすることにより、合計の現在値を取得できます。
- `sum = value` のような単純な割り当てとは互換性がありません。これは、新しい値がアプリケーションのどこかで計算される場合の一般的なパターンです。
- `!` 演算子 `sum = !sum` を使用して合計を単純にリセットできます。
- `crdt_gsum_to_text` を使用して内部状態を検査できます。

```
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
```

(次のページに続く)

(前のページからの続き)

```
- - 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;
```

#### ポジティブネガカウンター crdt_pncounter

- 正と負の値の両方のインクリメントをサポート counter + int および counter + bigint 演算子を介して。
- # 演算子を使用するか、bigint にキャストすることにより、カウンターの現在値を取得できます。
- counter = value のような単純な割り当てとは互換性がありません。これは、新しい値がアプリケーションのどこかで計算される場合の一般的なパターンです。
- ! 演算子 counter = !counter を使用してカウンターを単純にリセットできます。
- crdt_pncounter_to_text を使用して内部状態を検査できます。

```
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
```

(次のページに続く)

(前のページからの続き)

```
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

- - get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

- - show internal structure of counters
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;

- - reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

- - get current counter value after the reset
SELECT id, cnt::bigint, cnt FROM crdt_test;
```

### 正負の合計 crdt_pnsum

- sum + numeric を介して正と負の値の両方のインクリメントをサポートします。
- then # 演算子を使用するか、numeric にキャストすることにより、合計の現在値を取得できます。
- sum = value のような単純な割り当てとは互換性がありません。これは、新しい値がアプリケーションのどこかで計算される場合の一般的なパターンです。
- ! 演算子 sum = !sum を使用して合計を単純にリセットできます。
- crdt_pnsum_to_text を使用して内部状態を検査できます。

```
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
```

(次のページに続く)

(前のページからの続き)

```
- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment sums
UPDATE crdt_test SET pnsum = pnsum + 1.44     WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum + 12.20    WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum + (-24.34) WHERE id = 3;

- - decrement sums
UPDATE crdt_test SET pnsum = pnsum - 7.3      WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum - 192.83   WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum - (-12.22) WHERE id = 3;

- - get current sum value
SELECT id, pnsum::numeric, pnsum FROM crdt_test;

- - show internal structure of sum
SELECT id, bdr.crdt_pnsum_to_text(pnsum) FROM crdt_test;

- - reset sum
UPDATE crdt_test SET pnsum = !pnsum WHERE id = 1;

- - get current sum value after the reset
SELECT id, pnsum::numeric, pnsum FROM crdt_test;
```

### デルタカウンター crdt_delta_counter

- bigint ドメインとして定義されているため、 bigint 列とまったく同様に動作します。
- 正と負の値の両方のインクリメントをサポートします。
- counter = value のような単純な割り当てと互換性があります。これは、新しい値がアプリケーションのどこかで計算される場合に一般的です。
- 値を確実にリセットする簡単な方法はありません。

```
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
```

(次のページに続く)

(前のページからの続き)

```
- - 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;
```

### デルタサム crdt_delta_sum

- numeric ドメインとして定義されるため、 numeric 列とまったく同様に動作します。
- 正と負の値の両方のインクリメントをサポートします。
- sum = value のような単純な割り当てと互換性があります。これは、新しい値がアプリケーションのどこかで計算される場合に一般的です。
- 値を確実にリセットする簡単な方法はありません。

```
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;
```

(次のページに続く)

(前のページからの続き)

```
- - 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;
```

### 6.3.21 Testing and tuning PGD clusters

次のアプローチを使用して PGD アプリケーションをテストできます。

#### pgd_bench

Postgres ベンチマークアプリケーション *pgbench* は、新しいアプリケーション pgd_bench の形式で拡張されました。

pgd_bench は、PostgreSQL bin ディレクトリに追加される通常のコマンドラインユーティリティです。このユーティリティは、PostgreSQL pgbench ツールに基づいていますが、CAMO トランザクションと PGD 固有のワークロードのベンチマークをサポートしています。

pgd_bench の機能は pgbench 機能のスーパーセットですが、正常に動作するには BDR 拡張機能をインストールする必要があります。

主な違いは次のとおりです。

- 特定の場合にグローバルロックタイムアウトを防ぐための標準の pgbench シナリオでの初期化`-i` フラグを調整。
- 標準シナリオの `VACUUM` コマンドは、すべてのノードで実行されます。
- pgd_bench リリースは BDR 拡張機能のリリースに関連付けられ、対応する Postgres ディストリビューションに対してビルドされます。この情報は、`--version` フラグの出力に反映されます。

現在のバージョンでは、CAMO または通常の PGD 展開を使用しながらフェールオーバーテストを実行できます。

次のオプションが追加されました。

```
- m, --mode=regular|camo|failover
mode in which pgbench should run (default: regular)
```

- `-m camo` または`-m failover` を使用して、pgd_bench のモードを指定します。 `-m failover` 仕様を使用して、通常の PGD 展開でフェイルオーバーをテストできます。

```
- -retry
retry transactions on failover
```

- `--retry` を使用して、 `-m failover` モードでフェイルオーバーが発生したときにトランザクションをリトライするかどうかを指定します。このオプションは`-m camo` モードではデフォルトで有効になっています。

これらのオプションに加えて、 *DSNform* でフェイルオーバーのピアノードに関する接続情報を指定する必要があります。

次に、 CAMO 環境の例を示します。

```
pgd_bench -m camo -p $node1_port -h $node1_host bdrdemo \
    "host=$node2_host user=postgres port=$node2_port dbname=bdrdemo"
```

このコマンドは CAMO モードで実行されます。ノード 1 に接続し、テストを実行します。ノード 1 への接続が失われると、pgd_bench はノード 2 に接続します。ノード 2 を照会して、進行中のトランザクションのステータスを取得します。中止されたトランザクションおよび進行中のトランザクションは、CAMO モードで再試行されます。

フェイルオーバーモードで`--retry` を指定すると、進行中のトランザクションが再試行されます。このシナリオでは、進行中のトランザクションのステータスを見つける方法はありません。

### pgd_bench の使用に関する注意事項

- カスタム init スクリプトを使用する場合、DDL コマンドの背後にある意味を理解することが重要です。通常、`CREATE INDEX` などの DDL 操作を続行する前に、セカンダリノードがデータのロード手順に追いつくのを待機することをお勧めします。後者は、データのロードが完了するまで取得できないグローバルロックを取得するため、タイムアウトになる可能性があります。
- PostgreSQL および/または BDR 拡張機能を含む考えられる拡張機能によって発行される `NOTICE` および `WARNING` メッセージなどのクライアントメッセージを抑制するために追加の手順は実行されません。`client_min_messages` 、`bdr.camo_enable_client_warnings` などの適切な変数を設定してそれらを抑制するのはあなたの責任です。
- pgd_bench は、カスタムスクリプトの SQL トランザクションを開始しません。SQL トランザクションで実行するスクリプトには、transaction start および end コマンドを含める必要があります。 `-m` /`--mode` オプションを `camo` に設定して `pgd_bench` が実行された場合、提供されるカスタムスクリプトは SQL コマンドをトランザクションでラップする必要があります。そうしないと、CAMO 機能が予想通りに動作しません。

### パフォーマンスのテストとチューニング

PGD を使用すると、マルチプルのノードに書き込みトランザクションを発行できます。これらの書き込みを各ノードに戻すと、パフォーマンスのコストが発生します。

第一に、別のノードからの変更の再生には、CPU コストと I/O コストが発生し、WAL レコードが生成されます。SQL を再実行する必要がないため、CPU オーバーヘッドが低いため、通常、リソースの使用量は元のトランザクションよりも少なくなります。UPDATE および DELETE トランザクションの場合、データがキャッシュされていない場合、再生時に I/O コストが発生する可能性があります。

次に、変更の再生は、テーブルレベルおよび行レベルのロックを保持するため、ローカルワークロードに対する競合を生成する可能性があります。競合のないレプリケートデータ型 CRDT と列レベルの競合検出 CLCD 機能は、同時更新の場合でも正しい回答を取得しますが、通常のロックオーバーヘッドは削除され

ません。ロック競合が発生した場合は、競合する更新を回避するか、トランザクションをできるだけ短くしてください。大規模なトランザクションで頻繁に更新された行は、そのトランザクションのパフォーマンスのボトルネックを発生させます。複雑なアプリケーションでは、スケーラビリティを維持するためにいくつかの考慮が必要です。

パフォーマンスに問題があると思われる場合は、ベンチマークツールを使用してパフォーマンステストを開発します。pgd_bench を使用すると、ユースケースに固有のカスタムテストスクリプトを作成できるため、SQL のオーバーヘッドを理解し、同時実行の影響を測定できます。

PGD の実行が遅い場合、次のことをお勧めします。

1. 実稼働システムの問題ケースにできる限り近づけて、pgd_bench のカスタムテストスクリプトを作成します。
2. 1 つのノードでスクリプトを実行して、ベースライン図を取得します。
3. 1 つのノードで実行したのと同じ合計数のセッションを使用して、運用環境で発生するのと同じ数のノードでスクリプトを実行します。このテクニックは、複数のノードに移動した効果を示します。
4. これらの 2 つのテストのセッション数を増やして、アプリケーションでの競合の増加による影響をプロットできます。
5. テストがレプリケーションの遅延を考慮して十分な長さであることを確認します。
6. テスト中にレプリケーション遅延が増加していないことを確認します。

通常の Postgres チューニング機能をすべて使用して、アプリケーションの重要な部分の速度を向上させます。

### 6.3.22 Tables, views and functions reference

リファレンスセクションは、EDB Postgres Distributed で使用可能なすべてのファンクション、ビュー、およびコマンドの最終的なリストです。

#### User visible catalogs and views

- *bdr.camo_decion_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_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.* ライター
- *bdr.worker_tasks*

### System functions

#### バージョン情報ファンクション

- *bdr.bdr_version*
- *bdr.bdr_version_num*

#### システム情報ファンクション

- *bdr.get_relation_stats*
- *bdr.get_subscription_stats*

#### システムおよび進行状況情報パラメーター

- *bdr.local_node_id*
- *bdr.last_committed_lsn*
- *transaction_id*

#### ノードステータスファンクション

- *bdr.is_node_connected*
- *bdr.is_node_ready*

#### コンセンサスファンクション

- *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*

### ユーティリティファンクション

- *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*

### グローバルアドバイザリーロック

- *bdr.global_advisory_lock*
- *bdr.global_advisory_unlock*

### モニタリング機能

- *bdr.monitor_group_versions*
- *bdr.monitor_group_raft*
- *bdr.monitor_local_replslots*
- *bdr.wal_sender_stats*
- *bdr.get_decoding_worker_stat*
- *bdr.lag_control*

### ルーティング機能

- *bdr.routing_leadership_transfer*

#### CAMO ファンクション

- *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*

#### コミットスコープファンクション

- *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*

#### グローバルシーケンスパラメーター

- *bdr.default_sequence_kind*

#### DDL 処理

- *bdr.default_replica_identity*
- *bdr.ddl_replication*
- *bdr.role_replication*
- *bdr.ddl_locking*
- *bdr.truncate_locking*

#### グローバルロック

- *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*

### 汎用レプリケーション

- *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_lock_skew*
- *bdr.maximum_lock_skew_action*
- :ref:``bdr.accept_connections` <*bdr.accept_connections*>`
- *bdr.writer_input_queue_size*
- *bdr.writer_output_queue_size*
- *bdr.min_worker_backoff_migrate*

### CRDTs Overview

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

### コミットスコープ

- *bdr.commit_scopes*

### Commit At Most Once

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

### トランザクションストリーミングでの使用

- *bdr.default_streaming_mode* を使用したノード構成

### Lag Control

- *bdr.lag_control_max_commit_deploy*
- :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_deploy_adjust*
- *bdr.lag_control_sample_interval*
- *bdr.lag_control_commit_deploy_start*

### タイムスタンプベースのスナップショット

- *bdr.timestamp_snapshot_keep*

### モニタリングとロギング

- *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*

### 接続設定

- *bdr.global_connection_timeout*
- *bdr.global_keepalives*
- *bdr.global_keepalives_idle*
- *bdr.global_keepalives_interval*
- *bdr.global_keepalives_count*
- *bdr.global_tcp_user_timeout*

#### トポロジ設定

- *bdr.force_full_mesh*

#### 内部設定 - Raft タイムアウト

- *bdr.raft_global_election_timeout*
- *bdr.raft_group_election_timeout*
- *bdr.raft_response_timeout*

#### 内部設定 - その他の Raft 値

- *bdr.raft_keep_min_entries*
- *bdr.raft_log_min_apply_duration*
- *bdr.raft_log_min_message_duration*
- *bdr.raft_group_max_connections*

#### 内部設定 - その他の値

- *bdr.backwards_compatibility*
- *bdr.track_replication_estimates*
- *bdr.lag_tracker_apply_rate_weight*
- *bdr.enable_auto_sync_reconcile*

### Node management

- ノードの状態のリスト
- ノード管理コマンド
- *bdr_init_physical*
- *bdr_config*

### Node management interfaces

- *bdr.alter_node_group_option()* を使用したグループ構成
- *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*

### ルーティング機能

### コミットスコープ

- コミットスコープの構文
- *commit_scope_degrade_operation*
- コミットスコープターゲット
- *ORIGIN_GROUP*
- コミットスコープグループ
- 任意
- *ANY NOT*
- マジョリティプロテクト
- マジョリティではない
- *PGD Docker* クイックスタートキットのインストール
- すべてではない
- 確認レベル
- 受信時
- レプリケートについて
- *ON DURABLE*
- オンビジブル
- コミットスコープの種類
- 同期コミット
- *DEGRADE ON* パラメーター
- *commit_scope_degrade_operation*
- 耐久性オプショングループコミット/ *CAMO*
- *GROUP COMMIT* パラメーター

- *ABORT ON* パラメーター
- *DEGRADE ON* パラメーター
- *transaction_tracking* 設定
- *fence_resolution* 設定
- *commit_*決める設定
- *commit_scope_degrade_operation* 設定
- 耐久性オプショングループコミット/ *CAMO*
- *DEGRADE ON* パラメーター
- *Lag Control*
- *LAG CONTROL* パラメーター

### bdr_read_all_conflicts

- *Column-level conflict detection*
- 競合タイプのリスト
- 欠落列の競合の解決
- 競合リゾルバーのリスト
- デフォルトの競合リゾルバー
- 競合解決のリスト
- *Conflict logging*

### Column-level conflict functions

- *bdr.alter_table_conflict_detection* を使用して列レベルの競合解決を有効にする
- *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*

#### レプリケーションセットのメンバーシップ

- *bdr.replication_set_add_table*
- *bdr.replication_set_remove_table*

### DDL レプリケーションフィルタリング

- *bdr.replication_set_add_ddl_filter*
- *bdr.replication_set_remove_ddl_filter*

### Testing and tuning commands

- *pgd_bench*

### Sequences

#### シーケンスファンクション

- *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 ファンクション

- *bdr.gen_ksuuid_v2*
- *bdr.ksuuid_v2_cmp*
- *bdr.extract_timestamp_from_ksuuid_v2*

### KSUUID v1 ファンクション

- *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*>`
- 内部ファンクション
- *bdr.autopartition_create_partition*
- *bdr.autopartition_drop_partition*

## ストリームトリガーリファレンス

### 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

#### 一般的な内部ファンクション

- *bdr.bdr_get_commit_decions*
- *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*

### タスクマネージャー機能

- *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

- *bdr.alter_table_conflict_detection* を使用して列レベルの競合解決を有効にする
- *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*
- 競合タイプのリスト
- 欠落列の競合の解決
- 競合リゾルバーのリスト
- デフォルトの競合リゾルバー
- 競合解決のリスト

- *Conflict logging*

### User visible catalogs and views

カタログとビューは、アルファベット順にここにリストされています。

### bdr.camo_decion_journal

`bdr.logical_transaction_status` が呼び出された場合、フェイルオーバー後に CAMO パートナーノードによって解決された決定の永続ジャーナル。`bdr.node_pre_commit` とは異なり、これは通常の動作条件つまり、CAMO ペアの両方のノードが実行され接続されている下で処理されたトランザクションをカバーしません。このジャーナルのエントリは、自動的にクリーンアップされることはありません。これは、システムが依存しない診断ツールです。

#### bdr.camo_decision_journal 列

| Name | Type | Description |
|---|---|---|
| origin_node_id | oid | トランザクションが実行されたノードの OID |
| origin_xid | oid | リモートオリジンノードのトランザクション ID |
| decision | char | コミットの場合は'c'、アボートの場合は'a' |
| decision_ts | timestamptz | ディシジョンタイム |

### bdr.commit_scopes

`bdr.commit_scope` でグループコミットを有効にするために使用できるすべてのコミットスコープを保存するカタログ。

#### bdr.commit_scopes 列

| Name | Type | Description |
|---|---|---|
| commit_scope_id | oid | 参照するスコープの ID |
| commit_scope_name | name | 参照するスコープの名前 |
| commit_scope_origin_node_gr | oid | ID で参照される、ルールが適用されるノードグループ |
| sync_scope_rule | text | スコープの定義 |

### bdr.conflict_history

このテーブルは、競合がログに記録されるデフォルトのテーブルです。テーブルは列 `local_time` で RANGE パーティション化されており、Autopartition によって管理されます。デフォルトのデータ保持期間は 30 日です。

このテーブルへのアクセスは、テーブルの所有者が可能であり、行レベルのセキュリティによって制限されている、自分が所有するテーブルのすべての競合を表示できます。

詳細については、 Logging conflicts to a table を参照してください。

### bdr.conflict_history 列

| Name | Type | Description |
|---|---|---|
| sub_id | oid | この競合を生成したサブスクリプション。`bdr.subscription`テーブルに結合できます |
| origin_node_id | oid | 競合する変更を生成したノードの OID *pg_upgrade* カタログで見られる |
| local_xid | xid | 競合時のレプリケーションプロセスのローカルトランザクション |
| local_lsn | pg_lsn | 競合時のローカル LSN |
| local_time | timestamp with time zone | 紛争の現地時間 |
| remote_xid | xid | リモートノードオリジンで競合する変更を生成したトランザクション |
| remote_change_nr | oid | トランザクション内の変更のインデックス |
| remote_commit_lsn | pg_lsn | リモートノードオリジンで競合する変更を生成したトランザクションのコミット LSN |
| remote_commit_time | timestamp with time zone | リモートノードオリジンで競合する変更を生成したトランザクションのコミットタイムスタンプ |
| conflict_type | text | 検出された競合のタイプ |
| conflict_resolution | text | 選択された競合解決 |
| conflict_index | regclass | 競合するインデックスインデックスが削除されていない場合にのみ有効です |
| reloid | oid | 競合するリレーションインデックスがドロップされていない場合にのみ有効 |
| nspname | text | 競合時に競合が発生したリレーションのスキーマの名前 名前の変更に従いません |
| relname | text | 競合時に競合が発生したリレーションの名前名前の変更に従いません。 |
| key_tuple | json | 行のマッチングに使用されるキーの Json 表現 |
| remote_tuple | json | 競合する着信行の Json 表現 |
| local_tuple | json | ローカル競合行の Json 表現 |
| apply_tuple | json | 結果の適用された行の Json 表現 |
| local_tuple_xmin | xid | ローカル競合行を生成したトランザクション `local_tuple`が設定され、行が固定されていない場合 |
| local_tuple_node_id | oid | ローカル競合行を生成したノード `local_tuple`が設定され、行が固定されていない場合 |
| local_tuple_commit_time | timestamp with time zone | ローカル競合行の最後の既知の変更タイムスタンプ `local_tuple`が設定され、行が固定されていない場合。このコミットタイムスタンプは、このタプルを生成したノードに属します。 |

### bdr.conflict_history_summary

行の競合に関するユーザーが読み取り可能な詳細を含むビュー。

#### bdr.conflict_history_summary 列

| Name | Type | Description |
|---|---|---|
| nspname | text | スキーマの名前 |
| relname | text | テーブルの名前 |
| origin_node_id | oid | 競合する変更を生成したノードの OID *pg_upgrade* カタログで見られる |
| remote_commit_lsn | pg_lsn | リモートノードオリジンで競合する変更を生成したトランザクションのコミット LSN |
| remote_change_nr | oid | トランザクション内の変更のインデックス |
| local_time | timestamp with time zone | 紛争の現地時間 |
| local_tuple_commit_time | timestamp with time zone | ローカルコミットの時間 |
| remote_commit_time | timestamp with time zone | リモートコミットの時間 |
| conflict_type | text | 競合の種類 |
| conflict_resolution | text | 採択された決議 |

### bdr.consensus_kv_data

`bdr.consensus_kv_store()` および `bdr.consensus_kv_fetch()` インターフェイスによって使用される内部 Raft ベースの KV ストアの永続ストレージ。

#### bdr.consensus_kv_data 列

| Name | Type | Description |
|---|---|---|
| kv_key | text | 一意のキー |
| kv_val | json | json 形式の任意の値 |
| kv_create_ts | timestamptz | 最終書き込みタイムスタンプ |
| kv_ttl | int | ミリ秒単位の値の生存時間 |
| kv_expire_ts | timestamptz | 有効期限タイムスタンプ *kv_create_ts* + *kv_ttl* |

### bdr.crdt_handlers

この表は、すべての CRDT データ型のマージハンドラーファンクションをリストしています。

#### bdr.crdt_handlers 列

| Name | Type | Description |
|---|---|---|
| crdt_type_id | regtype | CRDT データ型 ID |
| crdt_merge_id | regproc | このデータ型のマージファンクション |

### bdr.ddl_replication

このビューには、現在の *DDL filters* によってセットアップされた DDL レプリケーション構成がリストされます。

#### bdr.ddl_replication 列

| Name | Type | Description |
|---|---|---|
| set_ddl_name | name | DDL フィルタの名前 |
| set_ddl_tag | text | 適用するコマンドタグ正規表現 |
| set_ddl_role | text | 適用されるロール正規表現 |
| set_name | name | このフィルタが定義されているレプリケーションセットの名前 |

### bdr.depend

このテーブルは、PGD カタログ内の内部オブジェクトの依存関係を追跡します。

### bdr.failover_replication_slots

このテーブルは、フェールオーバーサポートで使用されている論理レプリケーションスロットのステータスを追跡します。フェイルオーバーレプリケーションスロットの詳細については、CDC Failover support を参照してください。

#### bdr.failover_replication_slots 列

| Name | Type | Description |
|---|---|---|
| slot_name | name | レプリケーションスロットの名前 |
| slot_id | oid | レプリケーションスロットの ID |
| node_group_id | oid | ノードグループの ID |
| plugin | name | プラグインの名前 |
| twophase | boolean | スロットは 2 フェーズコミットに使用されますか |
| active_node | oid | アクティブノードの ID |
| active_pid | int | 現在スロットをデコードしているプロセスの PID |
| prev_node | oid | 以前のノードの ID |

### bdr.global_consensus_journal

このカタログテーブルは、グローバルコンセンサスを管理中に送信されたすべての Raft メッセージのログを記録します。

bdr.global_consensus_response_journal カタログと同様、ペイロードはバイナリエンコードされた形式で保存され、bdr.decode_message_payload() ファンクションでデコードできます。詳細については、*bdr.global_consensus_journal_details* ビューを参照してください。

#### bdr.global_consensus_journal 列

| Name | Type | Description |
|---|---|---|
| log_index | int8 | ジャーナルエントリの ID |
| term | int8 | Raft 用語 |
| origin | oid | 要求が発生したノードの ID |
| req_id | int8 | リクエストの ID |
| req_payload | bytea | 要求のペイロード |
| trace_context | bytea | 要求のトレースコンテキスト |

### bdr.global_consensus_journal_details

このビューは、 bdr.decode_message_payload() ファンクションを使用してペイロードをデコードして、送信された Raft メッセージと対応する応答を表示します。

#### bdr.global_consensus_journal_details 列

| Name | Type | Description |
|---|---|---|
| node_group_name | name | ノードグループの名前 |
| log_index | int8 | ジャーナルエントリの ID |
| term | int8 | Raft 用語 |
| request_id | int8 | リクエストの ID |
| origin_id | oid | 要求が発生したノードの ID |
| req_payload | bytea | 要求のペイロード |
| origin_node_name | name | 要求が発生したノードの名前 |
| message_type_no | oid | 要求の PGD メッセージタイプの ID |
| message_type | text | 要求の PGD メッセージタイプの名前 |
| message_payload | text | 要求の PGD メッセージペイロード |
| response_message_type_no | oid | 応答の PGD メッセージタイプの ID |
| response_message_type | text | 応答の PGD メッセージタイプの名前 |
| response_payload | text | 応答の PGD メッセージペイロード |
| response_errcode_no | text | 応答の SQLSTATE |

次のページに続く

表 25 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| response_errcode | text | 応答のエラーコード |
| response_message | text | 応答のエラーメッセージ |

### bdr.global_consensus_response_journal

このカタログテーブルは、グローバルコンセンサスを管理中に受信した Raft メッセージへのすべての応答を収集します。

`bdr.global_consensus_journal` カタログと同様、ペイロードはバイナリエンコードされた形式で保存され、`bdr.decode_message_payload()` ファンクションでデコードできます。詳細については、*bdr.global_consensus_journal_details* ビューを参照してください。

#### bdr.global_consensus_response_journal 列

| Name | Type | Description |
|---|---|---|
| log_index | int8 | ジャーナルエントリの ID |
| res_status | oid | 応答のステータスコード |
| res_payload | bytea | 応答のペイロード |
| trace_context | bytea | 応答のトレースコンテキスト |

### bdr.global_lock

このカタログテーブルには、サーバー再起動時にグローバルロック状態を回復するために必要な情報が保存されます。

`bdr.global_lock` の表示行は必ずしもすべてのグローバルロックアクティビティを反映しているわけではないため、使用状況をモニタリングする場合は、*bdr.global_locks* ビューが適しています。

このテーブルの内容は変更しないでください。重要な PGD カタログです。

#### bdr.global_lock 列

| Name | Type | Description |
|---|---|---|
| ddl_epoch | int8 | ロックの DDL エポック |
| origin_node_id | oid | グローバルロックが発生したノードの OID |
| lock_type | oid | ロックのタイプ DDL または DML |
| nspname | name | ロックされたリレーションのスキーマ名 |
| relname | name | ロックされたリレーションのリレーション名 |
| groupid | oid | トップレベルグループの OID アドバイザリーロックの場合 |
| key1 | integer | 最初の 32 ビットキーまたは 64 ビットキーの下位 32 ビットアドバイザリーロック用 |

次のページに続く

表 27 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| key2 | integer | 64 ビットキーの 2 番目の 32 ビットキー以上の 32 ビットアドバイザリロック用 |
| key_is_bigint | boolean | 64 ビット整数キーを使用する場合に true アドバイザリロックの場合 |

### bdr.global_locks

このノードのアクティブなグローバルロックを含むビュー。*bdr.global_locks* ビューは、PGD の共有メモリロック状態追跡を公開し、管理者に PGD のグローバルロックアクティビティと進行状況についてのより深い洞察を提供します。

グローバルロックの監視 を参照

グローバルロックの詳細については、

#### bdr.global_locks 列

| Name | Type | Description |
|---|---|---|
| origin_node_id | oid | グローバルロックが発生したノードの OID |
| origin_node_name | name | グローバルロックが発生したノードの名前 |
| lock_type | text | ロックのタイプ DDL または DML |
| relation | text | ロックされたリレーション名 DML ロックの場合またはキーアドバイザリロックの場合 |
| pid | int4 | ロックを保持しているプロセスの PID |
| acquire_stage | text | ロック取得プロセスの内部状態 |
| waiters | int4 | 同じグローバルロックを待機するバックエンドのリスト |
| global_lock_request_time | timestamptz | このグローバルロックの取得がオリジンノードによって開始された時間 |
| local_lock_request_time | timestamptz | ローカルノードがローカルロックの取得を試行し始めた時間 |
| last_state_change_time | timestamptz | `acquire_stage`が最後に変更された時間 |

カラムの詳細

- `relation` DML ロックの場合、`relation` は、DML ロックが取得されたリレーションを示します。グローバルアドバイザリロックの場合、`relation` 列は、実際に、ロックが取得された 2 つの 32 ビット整数または 1 つの 64 ビット整数を表示します。
- `origin_node_id` および `origin_node_name` これらがローカルノードの ID および名前と同じである場合、ローカルノードはグローバル DDL ロックのイニシエーター、つまり、取得トランザクションを実行しているノードです。これらのフィールドが別のノードを指定している場合、ローカルノードは代わりにローカル DDL ロックを取得して、リモートノードからのグローバル DDL ロック要求を満たそうとします。

- pid グローバル DDL ロックを要求したプロセスのプロセス ID。ローカルノードが要求ノードの場合。他のノードでは NULL。オリジンノードを照会して、ロッカー pid を特定します。
- global_lock_request_time グローバルロック要求イニシエーターがグローバルロックを取得するプロセスを開始したタイムスタンプ。現在のノードで不明な場合は null にすることができます。この時間は DDL ロック要求の先頭にスタンプされ、DDL エポック管理および保留中レプリケーションキューの必要なフラッシュに要した時間が含まれます。現在、オリジンノードでのみ知られています。
- local_lock_request_time ローカルノードがこのグローバルロックのローカルロックを取得しようと開始したタイムスタンプ。これには、重量セッションロックの取得に要した時間が含まれますが、DDL エポックまたはキューのフラッシュにかかる時間は含まれません。ローカルノードの再起動後にロックが再取得された場合、それがノードの再起動時間になります。
- last_state_change_time このグローバルロックエントリーの bdr.global_locks.acquire_stage フィールドが最後に変更されたタイムスタンプ。

### bdr.group_camo_details

bdr.run_on_all_nodes を使用して、すべてのノードから CAMO 関連の情報を収集します。

#### bdr.group_camo_details 列

| Name | Type | Description |
|---|---|---|
| node_id | text | 内部ノード ID |
| node_name | text | ノードの名前 |
| camo_partner | text | カモフラージュパートナーのノード名 |
| is_camo_partner_connecte | text | 接続状態 |
| is_camo_partner_ready | text | 準備状況 |
| camo_transactions_resolve | text | 保留中および未解決の CAMO トランザクションはありますか |
| apply_lsn | text | 再生されたとして報告された最新の位置可視 |
| receive_lsn | text | 受信した変更またはメッセージの最新の LSN 再起動の場合に遡ることができます |
| apply_queue_size | text | apply_lsn と receive_lsn のバイト差 |

### bdr.group_raft_details

bdr.run_on_all_nodes を使用して、すべてのノードから Raft コンセンサスステータスを収集します。

#### bdr.group_raft_details 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | 内部ノード ID |
| node_name | name | ノードの名前 |

次のページに続く

表 30 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| node_group_name | name | グループの名前は次の一部です |
| state | text | ノード上の Raft ワーカーの状態 |
| leader_id | oid | RAFT_LEADER のノード ID |
| current_term | int | Raft 選挙内部 ID |
| commit_index | int | Raft スナップショットの内部 ID |
| nodes | int | アクセス可能なノードの数 |
| voting_nodes | int | 投票ノードの数 |
| protocol_version | int | このノードのプロトコルバージョン |

### bdr.group_replslots_details

`bdr.run_on_all_nodes` を使用して、すべてのノードから PGD スロット情報を収集します。

#### `bdr.group_replslots_details` 列

| Name | Type | Description |
|---|---|---|
| node_group_name | text | PGD グループの名前 |
| origin_name | text | オリジンノードの名前 |
| target_name | text | ターゲットノードの名前 |
| slot_name | text | このサブスクリプションで使用されるオリジンノードのスロット名 |
| active | text | スロットはアクティブですか?接続は接続されていますか? |
| state | text | レプリケーションの状態キャッチアップ、ストリーミングなど、またはオフラインの場合は「切断」 |
| write_lag | interval | 報告された書き込みのおおよその遅延時間 |
| flush_lag | interval | 報告されたフラッシュのおおよその遅延時間 |
| replay_lag | interval | 報告されたリプレイのおおよそのラグタイム |
| sent_lag_bytes | int8 | send_lsn と現在の WAL 書き込み位置のバイト差 |
| write_lag_bytes | int8 | write_lsn と現在の WAL 書き込み位置のバイト差 |
| flush_lag_bytes | int8 | lush_lsn と現在の WAL 書き込み位置のバイト差 |
| replay_lag_byte | int8 | recovery_lsn と現在の WAL 書き込み位置のバイト差 |

### bdr.group_subscription_summary

`bdr.run_on_all_nodes` を使用して、すべてのノードからサブスクリプションステータスを収集します。

#### `bdr.group_subscription_summary` 列

| Name | Type | Description |
|---|---|---|
| origin_node_name | text | サブスクリプションの元の名前 |

次のページに続く

表 32 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| target_node_name | text | サブスクリプションのターゲットの名前 |
| last_xact_replay_timestamp | text | 最後に再生されたトランザクションのタイムスタンプ |
| sub_lag_seconds | text | now と last_xact_replay_timestamp 間のラグ |

### bdr.group_versions_details

bdr.run_on_all_nodes を使用して、すべてのノードから PGD 情報を収集します。

#### bdr.group_versions_details 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | 内部ノード ID |
| node_name | name | ノードの名前 |
| postgres_version | text | ノード上の PostgreSQL バージョン |
| bdr_version | text | ノード上の PGD バージョン |

### bdr.leader

クラスター内のサブグループ全体でリーダーノードを追跡します。クラスター内のすべての書き込みリーダーとサブスクライバー専用グループリーダーのステータスを表示します最適化されたトポロジが有効になっている場合。

#### bdr.leader 列

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | ノードグループの ID。 |
| leader_node_id | oid | リーダーノードの ID。 |
| generation | int | リーダーノードの生成。 Leader_kind は、セマンティクスを設定します。 |
| leader_kind | "char" | リーダーノードの種類。 |

Leader_kind 値は次のとおりです。

| Value | Description |
|---|---|
| W | プロキシルーティングごとに、リーダーを書き込みます。この場合、リーダーはサブグループの Raft インスタンスによって維持されます。 `generation`は、各 `bdr.node_group_routing_info`レコードの `write_leader_version`に対応します。 |

次のページに続く

表 35 – 前のページからの続き

| Value | Description |
|---|---|
| S | 購読者専用グループのリーダー。 SO グループのこの指定されたメンバーは、上流データノードをサブスクライブし、残りの SO グループメンバーに上流の変更を公開する任務を負います。リーダーは、トップレベルの Raft インスタンスによって維持されます。 `generation`は、リーダーが変更されると順次更新されます。 |

### bdr.local_consensus_snapshot

このカタログテーブルには、ローカルノードによって作成または受信されたコンセンサススナップショットが含まれています。

#### bdr.local_consensus_snapshot 列

| Name | Type | Description |
|---|---|---|
| log_index | int8 | ジャーナルエントリの ID |
| log_term | int8 | Raft 用語 |
| snapshot | bytea | Raft スナップショットデータ |

### bdr.local_consensus_state

このカタログテーブルには、ローカルノード上の Raft の現在の状態が保存されます。

#### bdr.local_consensus_state 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | ノードの ID |
| current_term | int8 | Raft 用語 |
| apply_index | int8 | Raft はインデックスを適用します |
| voted_for | oid | 今期にこのノードによって投じられた投票 |
| last_known_leader | oid | 最後に知られている Raft リーダーの node_id |

### bdr.local_node

このテーブルは、現在の Postgres インスタンスの現在のデータベース内のローカルノードを識別します。

#### bdr.local_node 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | ノードの ID |
| pub_repsets | text[] | 公開されたレプリケーションセット |

次のページに続く

表 38 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| sub_repsets | text[] | サブスクライブされたレプリケーションセット |
| node_uuid | uuid | ノードの UUID |

### bdr.local_node_summary

*bdr.node_summary* および `pub_repsets` および `sub_repsets` と同じ情報を含むビューですが、ローカルノードのみを対象とします。

### bdr.local_sync_status

サブスクリプションまたはテーブル同期プロセスのステータスに関する情報。

#### `bdr.local_sync_status` 列

| Name | Type | Description |
|---|---|---|
| sync_kind | char | 完了した同期の種類 |
| sync_subid | oid | 同期を実行するサブスクリプションの ID |
| sync_nspname | name | 同期テーブルのスキーマ名存在する場合 |
| sync_relname | name | 同期テーブルの名前存在する場合 |
| sync_status | char | 同期の現在の状態 |
| sync_remote_relid | oid | 上流に同期されたテーブルがある場合の ID |
| sync_end_lsn | pg_lsn | 同期状態が最後に変更された位置 |

### bdr.node

このテーブルは、クラスター内のすべての PGD ノードをリストしています。

ビュー *bdr.node_summary* は、`bdr.node` のほとんどの列の人が判読可能なバージョンを提供します。

#### `bdr.node` 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | ノードの ID |
| node_name | name | ノードの名前 |
| node_group_id | oid | ノードグループの ID |
| source_node_id | oid | ソースノードの ID |
| synchronize_structure | "char" | 結合中に実行されるスキーマの同期 |
| node_state | oid | ノードの一貫した状態 |
| target_state | oid | ノードが到達しようとしている状態参加または昇格中に |

次のページに続く

表 40 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| seq_id | int4 | 一意のシーケンス番号を生成するために使用されるノードのシーケンス識別子 |
| dbname | name | ノードのデータベース名 |
| node_dsn | char | ノードの接続文字列 |
| proto_version_ranges | int[] | ノードによってサポートされているプロトコルバージョンレンジ |
| generation | smallint | ノードが以前のノードと同じ名前で参加したときにインクリメントされるカウンター |
| node_kind | oid | ノード種類の ID |
| node_join_finished | boolean | 結合が完了したかどうかを確認する |
| node_uuid | uuid | ノードの UUID 一意 |

### bdr.node_catchup_info

このカタログテーブルは、各ノードに関連するキャッチアップ情報を記録します。結合または部分プロシージャに関連している場合。

#### bdr.node_catchup_info 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | ノードの ID |
| node_source_id | oid | データのソースとして使用されるノードの ID |
| slot_name | name | このソースに使用されるスロット |
| min_node_lsn | pg_lsn | ノードがピアノードからの直接再生に切り替えることができる最小 LSN |
| catchup_state | oid | キャッチアップ状態のステータスコード |
| origin_node_id | oid | トランザクションが必要なノードの ID |

ノード node_id は、分割ノード origin_node_id からの欠損データが必要な場合、転送によって既にそれ node_source_id を持つノードからそれを取得できます。このテーブルのレコードは、ノード node_id が EDB Postgres 分散クラスターのメンバーになるまで持続します。

### bdr.node_catchup_info_details

情報をより分かりやすく表示する bdr.node_catchup_info カタログのビュー

### bdr.node_conflict_resolvers

すべての既知の競合タイプの現在構成されている競合解決。

#### bdr.node_conflict_resolvers 列

| Name | Type | Description |
|---|---|---|
| conflict_type | text | 競合の種類 |
| conflict_resolver | text | この競合タイプに使用されるリゾルバー |

### bdr.node_group

このカタログテーブルには、すべての PGD ノードグループがリストされています。ユーザーが読み取り可能な詳細を含むビューについては、*bdr.node_group_summary* も参照してください。

#### bdr.node_group 列

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | ノードグループの ID。 |
| node_group_name | name | ノードグループの名前。 |
| node_group_default_repset | oid | このノードグループのデフォルトレプリケーションセット。 |
| node_group_default_repset | oid | このノードグループのデフォルトレプリケーションセット。 |
| node_group_parent_id | oid | 親グループの ID これがルートグループの場合は 0。 |
| node_group_flags | int | グループフラグ。 |
| node_group_uuid | uuid | グループの uuid。 |
| node_group_apply_delay | interval | プロバイダーからの変更を適用するまでに加入者が待機する時間。 |
| node_group_check_constra | bool | 適用プロセスがデータを適用するときにコンストレインをチェックするかどうか。 |
| node_group_num_writers | int | このノードグループをバッキングするサブスクリプションに使用するライターの数。 |
| node_group_enable_wal_d | bool | グループに enable_wal_decoder が設定されているかどうか。 |
| node_group_streaming_mo | char | トランザクションストリーミング設定 'O'-オフ、'F'-ファイル、 'W'-ライタ、 'A'-自動、 'D'-デフォルト。 |
| node_group_default_comm | oid | ノードグループのデフォルトのコミットスコープの ID。 |
| node_group_location | char | ノードグループに関連付けられた場所の名前。 |
| node_group_enable_routin | char | ノードグループが接続マネージャーを介したルーティングを許可するかどうか。 |
| node_group_enable_raft | bool | ノードグループが Raft コンセンサスを許可するかどうか。 |

### bdr.node_group_replication_sets

PGD グループ用に作成されるデフォルトのレプリケーションセットを示すビュー。bdr.replication_sets も参照してください。

#### bdr.node_group_replication_sets 列

| Name | Type | Description |
|---|---|---|
| node_group_name | name | PGD グループの名前 |
| def_repset | name | デフォルトの repset の名前 |
| def_repset_ops | text[] | デフォルトの repset によってレプリケートされたアクション |
| def_repset_ext | name | デフォルトの「外部」repset の名前通常は def_repset と同じです |
| def_repset_ext_ops | text[] | デフォルトの「外部」rep セットによってレプリケートされたアクション通常は def_repset_ops と同じです |

### bdr.node_group_summary

ノードグループに関するユーザーが読み取り可能な詳細を含むビュー。*bdr.node_group_config* も参照してください。

#### bdr.node_group_summary 列

| Name | Type | Description |
|---|---|---|
| node_group_name | name | ノードグループの名前 |
| default_repset | name | このノードグループのデフォルトレプリケーションセット |
| parent_group_name | name | 親グループの名前これがルートグループの場合は NULL |
| node_group_type | text | ノードグループのタイプ"global"、"data"、"shard"、または"subscriber-only"のいずれか |
| apply_delay | interval | プロバイダーからの変更を適用するまでに加入者が待機する時間 |
| check_constraints | boolean | 適用プロセスがデータの適用時にコンストレインをチェックするかどうか |
| num_writers | integer | このノードグループをバッキングするサブスクリプションに使用するライターの数 |
| enable_wal_decoder | boolean | グループに enable_wal_decoder セットが設定されているかどうか |
| streaming_mode | text | トランザクションストリーミング設定 "off"、"file"、"ライタ"、"auto"または"default" |
| default_commit_scope | name | ノードグループのデフォルトのコミットスコープの名前 |

次のページに続く

表 45 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| location | name | ノードグループに関連付けられた場所の名前 |
| enable_routing | boolean | ノードグループが接続マネージャーを介したルーティングを許可するかどうか |
| enable_raft | boolean | ノードグループが Raft コンセンサスを許可するかどうか |
| route_writer_max_lag | bigint | 受け入れられる最大書き込み遅延 |
| route_reader_max_lag | bigint | 受け入れられる最大読み取り遅延 |
| route_writer_wait_flush | boolean | フラッシュを待機する必要がある場合にスイッチ |

### bdr.node_local_info

ローカルノードに固有のノードごとの構成を保存するために使用されるカタログテーブルノードごとの構成のグローバルビューではなく。

#### bdr.node_local_info 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | ノードの OID ローカルノードを含む |
| applied_state | oid | ノード状態の内部 ID |
| ddl_epoch | int8 | ノードによって処理された最後のエポック番号 |
| slot_name | name | そのノードへの接続に使用されるスロットの名前ローカルノードの場合は NULL |
| origin_name | name | そのノードの レプリケーション起点のクリーンアップ の名前。ローカルノード、またはサブスクライバ専用ノードやスタンバイなどのデータノードではないノードの場合、NULL になります。 |

### bdr.node_log_config

競合ログ構成に関する情報を保存するカタログビュー。

#### bdr.node_log_config 列

| Name | Description |
|---|---|
| log_name | ロギング構成の名前 |
| log_to_file | サーバーログファイルにログを記録するかどうか |
| log_to_table | テーブルにログを記録するかどうか、およびどのテーブルがターゲットであるか |
| log_conflict_type | NULL がすべてを意味する場合、ログに記録する競合タイプ |
| log_conflict_res | NULL がすべてを意味する場合、どの競合解決をログに記録しますか |

### bdr.node_peer_progress

レプリケーションストリーム内のすべてのノードの進行状況を追跡するために使用されるカタログ。クラスター内のすべてのノードは、その進行状況を bdr.replay_progress_frequency ミリ秒ごとに他のすべてのノードに定期的にブロードキャストしますデフォルトは 60000 ミリ秒、つまり 1 分です。このリレーションには N * (N-1) 行が必要です。

モニタリングには、 bdr.node_slots ビューの方が興味深いかもしれません。 *Monitoring the Connection Manager* も参照してください。

#### bdr.node_peer_progress 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | この位置情報を報告した元ノードの OID |
| peer_node_id | oid | この位置情報が報告されたノードのピアリモートノードの OID |
| last_update_sent_time | timestamptz | レポートが元ノードによって送信された時刻 |
| last_update_recv_time | timestamptz | ローカルサーバーがレポートを受信した時刻 |
| last_update_node_lsn | pg_lsn | レポート時の元のノードの LSN |
| peer_position | pg_lsn | 元のノードから見たノードのピアの最新の LSN |
| peer_replay_time | timestamptz | 報告ノードによって認識されたピアの最新の再生時間 |
| last_update_horizon_xid | oid | 内部解像度ホライズン すべての下位 xid は報告ノードで解決されることが知られています |
| last_update_horizon_lsn | pg_lsn | 内部解像度ホライズン レポートノードの LSN に関して同じ |

### bdr.node_replication_rates

このビューには、特定のノードからの送信レプリケーションアクティビティに関する情報が含まれています。

#### bdr.node_replication_rates 列

| Column | Type | Description |
|---|---|---|
| peer_node_id | oid | この情報が報告されたノードのピアリモートノードの OID |
| target_name | name | ターゲットピアノードの名前 |
| sent_lsn | pg_lsn | 最新の送信位置 |
| replay_lsn | pg_lsn | 再生されたとして報告された最新の位置可視 |
| replay_lag | interval | 報告されたリプレイのおおよそのラグタイム |
| replay_lag_bytes | int8 | attempts_lsn とオリジンの現在の WAL 書き込み位置のバイト差 |
| replay_lag_size | text | repair_lsn と現在の WAL 書き込み位置間の人間が判読可能なバイトの差 |
| apply_rate | bigint | ピアノードで 1 秒ごとに適用される LSN |

次のページに続く

表 49 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| catchup_interval | interval | ピアノードがまだ適用されていないすべての変更に追いつくまでに必要なおおよその時間 |

**注釈**

*replay_lag* は、再接続後すぐにゼロに設定されます。回避策として、*replay_lag_bytes* 、*replay_lag_size* 、または `catchup_interval` を使用します。

### bdr.node_slots

このビューには、PGD が現在のデータベースで使用するレプリケーションスロットに関する情報が含まれています。

送信レプリケーションの監視 を参照

このビューのフィールドの使用と解釈に関するガイダンスについては。

#### bdr.node_slots 列

| Name | Type | Description |
|---|---|---|
| target_dbname | name | ターゲットノード上のデータベース名 |
| node_group_name | name | PGD グループの名前 |
| node_group_id | oid | PGD グループの OID |
| origin_name | name | オリジンノードの名前 |
| target_name | name | ターゲットノードの名前 |
| origin_id | oid | 元ノードの OID |
| target_id | oid | ターゲットノードの OID |
| local_slot_name | name | PGD に従ったレプリケーションスロットの名前 |
| slot_name | name | Postgres によるスロットの名前上記と同じ |
| is_group_slot | boolean | スロットがこのノードのノードグループのクラッシュリカバリスロットである場合は true ["Group Replication Slot"](nodes#Group Replication Slot) を参照してください。 |
| is_decoder_slot | boolean | このスロットはデコードワーカー機能によって使用されますか |
| plugin | name | このスロットを使用する論理デコードプラグイン pglogical_output または bdr である必要があります |
| slot_type | text | スロットのタイプ論理的である必要があります。 |
| datoid | oid | 現在のデータベースの OID |
| database | name | 現在のデータベースの名前 |
| temporary | bool | スロットは一時的なものですか |

次のページに続く

表 50 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| active | bool | スロットはアクティブですか?接続は接続されていますか? |
| active_pid | int4 | スロットに接続されたプロセスの PID |
| xmin | xid | スロットに必要な XID |
| catalog_xmin | xid | スロットに必要なカタログ XID |
| restart_lsn | pg_lsn | スロットがデコードをリスタートできる LSN |
| confirmed_flush_lsn | pg_lsn | 最新に確認された複製位置 |
| usesysid | oid | レプリケーションセッションが実行されているユーザーの sysid |
| usename | name | レプリケーションセッションが実行されているユーザーのユーザー名 |
| application_name | text | クライアント接続のアプリケーション名 `synchronous_standby_names`によって使用されます |
| client_addr | inet | クライアント接続の IP アドレス |
| client_hostname | text | クライアント接続のホスト名 |
| client_port | int4 | クライアント接続のポート |
| backend_start | timestamptz | 接続が開始されたとき |
| state | text | レプリケーションの状態キャッチアップ、ストリーミングなど、またはオフラインの場合は「切断」 |
| sent_lsn | pg_lsn | 最新の送信位置 |
| write_lsn | pg_lsn | 最新の立場を書面どおりに報告 |
| flush_lsn | pg_lsn | 最新の位置はディスクにフラッシュされたとして報告されました |
| replay_lsn | pg_lsn | 再生されたとして報告された最新の位置可視 |
| write_lag | interval | 報告された書き込みのおおよその遅延時間 |
| flush_lag | interval | 報告されたフラッシュのおおよその遅延時間 |
| replay_lag | interval | 報告されたリプレイのおおよそのラグタイム |
| sent_lag_bytes | int8 | send_lsn と現在の WAL 書き込み位置のバイト差 |
| write_lag_bytes | int8 | write_lsn と現在の WAL 書き込み位置のバイト差 |
| flush_lag_bytes | int8 | lush_lsn と現在の WAL 書き込み位置のバイト差 |
| replay_lag_bytes | int8 | recovery_lsn と現在の WAL 書き込み位置のバイト差 |
| sent_lag_size | text | send_lsn と現在の WAL 書き込み位置の間の人間が判読可能なバイトの差 |
| write_lag_size | text | write_lsn と現在の WAL 書き込み位置間の人間が判読可能なバイトの差 |
| flush_lag_size | text | flash_lsn と現在の WAL 書き込み位置の間の人間が判読可能なバイトの差 |
| replay_lag_size | text | repair_lsn と現在の WAL 書き込み位置間の人間が判読可能なバイトの差 |

**注釈**

*replay_lag* は、再接続後すぐにゼロに設定されます。回避策として、*replay_lag_bytes* または `re-

play_lag_size` を使用します。

### bdr.node_summary

このビューには、ローカルノードに知られているすべての PGD ノードに関する概要情報が含まれています。

#### bdr.node_summary 列

| Name | Type | Description |
|---|---|---|
| node_name | name | ノードの名前 |
| node_group_name | name | ノードが一部である PGD グループの名前 |
| interface_connstr | text | ノードへの接続文字列 |
| peer_state_name | text | 人間が読める形式のノードの一貫した状態 |
| peer_target_state_name | text | ノードが到達しようとしている状態参加または昇格中に |
| node_seq_id | int4 | 一意のシーケンス番号を生成するために使用されるノードのシーケンス識別子 |
| node_local_dbname | name | ノードのデータベース名 |
| node_id | oid | ノードの OID |
| node_group_id | oid | PGD ノードグループの OID |
| node_kind_name | oid | ノード種類名 |
| node_uuid | uuid | ノードの UUID |

### bdr.parted_origin_catchup_info

このテーブルは、分割されたオージンに関連する各ノードの関連キャッチアップ情報を記録します。

#### bdr.parted_origin_catchup_info 列

| Name | Type | Description |
|---|---|---|
| parting_peer_node_id | oid | 分割されたノードの ID |
| node_id | oid | ノードの ID |
| node_group_id | oid | ノードグループの ID |
| origin_catchup_lsn | pg_lsn | ノードがグループスロットが追いつくのを待機し、状態を DONE に移動する LSN |
| origin_catchup_state | oid | 分割されたオリジンキャッチアップのステータスコード |

ノード `node_id` は、そのグループスロットが記録された LSN `origin_catchup_lsn` に追いつくのを待機します。これは、そのグループスロットが PARTED ノード `parting_peer_node_id` から発生するすべてのトランザクションに追いつくようにするためです。

このテーブルのレコードは、パーティングノード `parting_peer_node_id` が自動的に削除されるまで持続します。

### bdr.parted_origin_catchup_info_details

このテーブルは、bdr.parted_origin_catchup_info のわかりやすいビューであり、分割されたオージンに関連する各ノードの関連キャッチアップ情報をこの場合はテキスト形式で示しています。

#### bdr.parted_origin_catchup_info_details 列

| Name | Type | Description |
|---|---|---|
| target_node_id | oid | ターゲットノードの ID |
| target_node_name | text | ターゲットノードの名前 |
| parting_node_id | oid | 分割されたノードの ID |
| parting_node_name | text | 分割されたノードの名前 |
| node_group_id | oid | ノードグループの ID |
| node_group_name | text | ノードグループの名前 |
| parting_catchup_lsn | pg_lsn | ノードがグループスロットが追いつくのを待機し、状態を DONE に移動する LSN |
| parting_catchup_state | oid | Parted Origin のキャッチアップステータスコード |
| parting_catchup_state_name | text | Parted Origin のキャッチアップステータステキスト |

ノード target_node_id は、そのグループスロットが記録された LSN parting_catchup_lsn に追いつくのを待機します。これは、そのグループスロットが PARTED ノード parting_node_id から発生するすべてのトランザクションに追いつくようにするためです。

このテーブルのレコードは、パーティングノード parting_node_id が自動的に削除されるまで持続します。

### bdr.queue

このテーブルには、レプリケートされた DDL ステートメントの履歴レコードが保存されます。

#### bdr.queue 列

| Name | Type | Description |
|---|---|---|
| queued_at | timestamptz | ステートメントがキューに入れられたとき |
| role | name | ステートメントを実行したロール |
| replication_sets | text[] | ステートメントはどのレプリケーションセットに公開されましたか |
| message_type | char | メッセージの種類。考えられる値<br />A - テーブル同期<br />D - DDL<br />S - シークエンス<br />T - 切り捨て<br />Q - SQL ステートメント |
| message | json | ステートメントのレプリケーションに必要なメッセージのペイロード |

### bdr.replication_set

レプリケーションセット構成を保存するテーブル。ユーザークエリーの場合、代わりに `bdr.replication_sets` ビューを確認することをお勧めします。

#### bdr.replication_set 列

| Name | Type | Description |
|---|---|---|
| set_id | oid | レプリケーションセットの OID |
| set_nodeid | oid | ノードの OID 現在は常にローカルノード oid |
| set_name | name | レプリケーションセットの名前 |
| replicate_insert | boolean | レプリケーションセットが INSERT をレプリケートするかどうかを示します |
| replicate_update | boolean | レプリケーションセットが UPDATE をレプリケートするかどうかを示します |
| replicate_delete | boolean | レプリケーションセットが DELETE をレプリケートするかどうかを示します |
| replicate_truncate | boolean | レプリケーションセットが TRUNCATE をレプリケートするかどうかを示します |
| set_isinternal | boolean | 予約済み |
| set_autoadd_tables | boolean | 新しいテーブルがこのレプリケーションセットに自動的に追加されるかどうかを示します |
| set_autoadd_seqs | boolean | 新しいシーケンスがこのレプリケーションセットに自動的に追加されるかどうかを示します |

### bdr.replication_set_table

レプリケーションセットテーブルのメンバーシップを保存するテーブル。ユーザークエリーの場合、代わりに `bdr.tables` ビューを確認することをお勧めします。

#### bdr.replication_set_table 列

| Name | Type | Description |
|---|---|---|
| set_id | oid | レプリケーションセットの OID |
| set_reloid | regclass | テーブルのローカル ID |
| set_att_list | text[] | 予約済み |
| set_row_filter | pg_node_tree | コンパイルされた行フィルタリング式 |

### bdr.replication_set_ddl

レプリケーションセット ddl レプリケーションフィルターを保存するテーブル。ユーザークエリーの場合、代わりに bdr.ddl_replication ビューを確認することをお勧めします。

#### bdr.replication_set_ddl 列

| Name | Type | Description |
|---|---|---|
| set_id | oid | レプリケーションセットの OID |
| set_ddl_name | name | DDL フィルタの名前 |
| set_ddl_tag | text | DDL フィルタのコマンドタグ |
| set_ddl_role | text | DDL を実行するロール |

### bdr.replication_sets

現在ノードで使用されていない場合でも、PGD グループで定義されたレプリケーションセットを表示するビュー。

#### bdr.replication_sets 列

| Name | Type | Description |
|---|---|---|
| set_id | oid | レプリケーションセットの OID |
| set_name | name | レプリケーションセットの名前 |
| replicate_insert | boolean | レプリケーションセットが INSERT をレプリケートするかどうかを示します |
| replicate_update | boolean | レプリケーションセットが UPDATE をレプリケートするかどうかを示します |
| replicate_delete | boolean | レプリケーションセットが DELETE をレプリケートするかどうかを示します |
| replicate_truncate | boolean | レプリケーションセットが TRUNCATE をレプリケートするかどうかを示します |
| set_autoadd_tables | boolean | 新しいテーブルがこのレプリケーションセットに自動的に追加されるかどうかを示します |
| set_autoadd_seqs | boolean | 新しいシーケンスがこのレプリケーションセットに自動的に追加されるかどうかを示します |

### bdr.schema_changes

スキーマ win PGD へのすべての変更を表示するシンプルなビュー。

#### bdr.schema_changes 列

| Name | Type | Description |
|---|---|---|
| schema_changes_ts | timestampstz | トリガーの ID |
| schema_changes_change | char | 変更タイプのフラグ |
| schema_changes_classid | oid | クラス ID |
| schema_changes_objectid | oid | オブジェクト ID |
| schema_changes_subid | smallint | サブスクリプション |
| schema_changes_descr | text | オブジェクトが変更されました |
| schema_changes_addrname | text[] | スキーマ変更の場所 |

### bdr.sequence_alloc

galloc シーケンスの割り当ての詳細を表示するビュー。

#### bdr.sequence_alloc 列

| Name | Type | Description |
|---|---|---|
| seqid | regclass | シーケンスの ID |
| seq_chunk_size | bigint | 値内のチャンクのシーケンス番号 |
| seq_allocated_up_to | bigint | NA |
| seq_nallocs | bigint | NA |
| seq_last_alloc | timestamptz | 最後に割り当てられたシーケンス |

### bdr.sequences

このビューには、内部 PGD 簿記のシーケンスを除き、すべてのシーケンスとその種類がリストされます。

#### bdr.sequences 列

| Name | Type | Description |
|---|---|---|
| nspname | name | シーケンスを含む名前空間 |
| relname | name | シーケンスの名前 |
| seqkind | text | シーケンスのタイプ'local'、'timeshard'、'galloc' |

### bdr.stat_activity

各バックエンドまたはワーカープロセスの動的アクティビティ。

これには、*pg_stat_activity* と同じ情報が含まれていますが、待機が PGD に関連し、次の接続マネージャー関連フィールドが追加されるときに `wait_event` が正しく設定される場合を除きます。

#### bdr.stat_activity 追加列

| Name | Type | Description |
|---|---|---|
| connection_manager_client_addr | inet | クライアント接続の IP アドレス |
| connection_manager_client_port | int | 接続マネージャーに接続したクライアントのソースポート接続マネージャーを介して接続が行われる場合 |
| connection_manager_client_hostn | text | クライアント接続のホスト名接続が接続マネージャーを介して行われる場合 |
| session_read_only | boolean | セッションが読み取り専用かどうか。接続マネージャーの読み取り専用ポートに接続されています |

### bdr.stat_commit_scope

各コミットスコープの統計を含むビュー。

#### bdr.stat_commit_scope 列

| Column | Type | Description |
|---|---|---|
| commit_scope_name | name | コミットスコープの名前 |
| group_name | name | コミットスコープが定義されているグループの名前 |
| ncalls | bigint | コミットスコープが使用された回数 |
| ncommits | bigint | コミットスコープで行われたコミットの成功した数 |
| naborts | bigint | 使用されるコミットスコープが最終的に中止された回数 |
| total_commit_time | double precision | コミットスコープを使用したコミットに費やされた合計時間ミリ秒単位 |
| min_commit_time | double precision | コミットスコープを使用してコミットに費やされた最小時間ミリ秒単位 |
| max_commit_time | double precision | コミットスコープを使用してコミットに費やした最大時間ミリ秒単位 |
| mean_commit_time | double precision | コミットスコープを使用したコミットに費やされた平均時間ミリ秒単位 |
| stats_reset | timestamp with time zone | ビュー内のすべての統計が最後にリセットされた時間 |

### bdr.stat_commit_scope_state

バックエンドによるコミットスコープの現在の使用に関する情報のビュー。

#### bdr.stat_commit_scope_state 列

| Column | Type | Description |
|---|---|---|
| pid | integer | バックエンドのプロセス ID |
| commit_scope_name | name | 使用されているコミットスコープの名前 |
| group_name | name | コミットスコープが定義されているグループの名前 |
| waiting_op_num | integer | まだ満たされていないコミットスコープ内の最初のオペレーションのインデックス |
| waiting_prepare_confirmations | integer | オペレーションに必要な PREPARE 確認の数 |
| waiting_commit_confirmations | integer | オペレーションに必要な COMMIT 確認の数 |
| waiting_lsn_confirmations | integer | オペレーションでまだ必要な LSN 確認の数 |

### bdr.stat_connection_manager

このノードの接続マネージャーの統計を含むビュー。

#### bdr.stat_connection_manager 列

| Column | Type | Description |
|---|---|---|
| ntotal_rw_conns | bigint | 読み取り/書き込み接続の合計数 |
| ntotal_ro_conns | bigint | 読み取り専用接続の合計数 |
| nactive_rw_conns | int | アクティブな読み取り/書き込み接続の数 |
| nactive_ro_conns | int | アクティブな読み取り専用接続の数 |

### bdr.stat_connection_manager_connections

接続マネージャーへの接続に関する情報を含むビュー。

#### bdr.stat_connection_manager_connections 列

| Column | Type | Description |
|---|---|---|
| connection_manager_client_addr | text | 接続マネージャーに接続したクライアントの IP アドレス。 |
| connection_manager_client_port | int | クライアントが接続マネージャーとの通信に使用している TCP ポート番号。 |

次のページに続く

表 66 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| connection_manager_addr | text | 接続マネージャーノードの IP アドレス。 |
| connection_manager_port | int | 接続マネージャーが Postgres ノードとの通信に使用している TCP ポート番号。 |
| session_read_only | boolean | セッションが読み取り専用かどうか。 |
| client_uses_tls | boolean | クライアントが TLS を使用して接続マネージャーノードに接続しているかどうか。 |

### bdr.stat_connection_manager_node_stats

このノードの接続マネージャーのサーバー接続統計に関する情報を含むビュー。

#### bdr.stat_connection_manager_node_stats 列

| Column | Type | Description |
|---|---|---|
| node_id | oid | ノードの OID |
| node_name | name | ノードの名前 |
| route_rw_connections | boolean | 読み取り/書き込み接続をこのノードにルーティングするかどうか |
| route_ro_connections | boolean | 読み取り専用接続をこのノードにルーティングするかどうか |
| ntotal_rw_conns | bigint | 読み取り/書き込み接続の合計数 |
| ntotal_ro_conns | bigint | 読み取り専用接続の合計数 |
| nactive_rw_conns | int | アクティブな読み取り/書き込み接続の数 |
| nactive_ro_conns | int | アクティブな読み取り専用接続の数 |

### bdr.stat_connection_manager_hba_file_rules

接続マネージャーが HBA ファイル pg_hba.conf から使用している有効なサポートされているルールのみを表示するビューと、それらのルールに関する情報。

#### bdr.stat_connection_manager_hba_file_rules 列

| Column | Type | Description |
|---|---|---|
| rule_number | integer | ルール番号。これは、認証中に一致が見つかるまで各ルールが検討される順序を示します。 |
| file_name | text | このルールを含むファイルの名前。 |
| line_number | integer | file_name で参照されるファイル内のこのルールの行番号。 |
| type | text | 接続の種類。 |
| database | text[] | このルールが適用されるデータベース名のリスト。 |

次のページに続く

表 68 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| user_name | text[] | このルールが適用されるユーザー名のリスト |
| address | text | ホスト名または IP アドレス、またはすべてのいずれか、samehost または samenet、またはローカル接続の場合は null。 |
| netmask | text | IP アドレスマスク、または該当しない場合は null。 |
| auth_method | text | 認証方法。 |
| auth_options | text | 認証方法に指定されたオプションがある場合。 |

### bdr.stat_raft_followers_state

Raft リーダーノード上の raft リーダーのフォロワーの状態のビュー他のノードでは空。

#### bdr.stat_raft_followers_state 列

| Column | Type | Description |
|---|---|---|
| group_name | name | この情報の対象となるグループ各グループには、個別のコンセンサスを構成できます。 |
| node_name | name | フォロワーノードの名前。 |
| sent_commit_index | bigint | 最新の Raft インデックスがフォロワーノードに送信されます。 |
| match_index | bigint | Raft インデックスは、フォロワーノードからの次の応答と一致すると予想されます。 |
| last_message_time | timestamp with time zone | フォロワーノードから見た最後のメッセージ任意の要求を含む。 |
| last_heartbeat_send_time | timestamp with time zone | リーダーが最後にハートビートをフォロワーノードに送信した時間。 |
| last_heartbeat_response_tir | timestamp with time zone | リーダーが最後にフォロワーノードからのハートビート応答を見た時間。 |
| approx_clock_drift_ms | bigint | ミリ秒単位のフォロワーノードに対してリーダーによって見られたおおよそのクロックドリフト。 |

### bdr.stat_raft_state

ローカルノードでの Raft コンセンサスの状態を説明するビュー。

#### bdr.stat_raft_state 列

| Column | Type | Description |
|---|---|---|
| group_name | name | この情報の対象となるグループ各グループには、個別のコンセンサスを構成できます。 |

次のページに続く

表 70 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| raft_stat | text | Raft のローカルノードの状態 'LEADER'、'CANDIDATE'、'FOLLOWER'、'STOPPED' |
| leader_name | name | Raft リーダーの名前がある場合 |
| voted_for_name | name | ローカルノードがリーダーとして最後に投票したノード |
| is_voting | boolean | Raft のローカルノード部分が投票しています |
| heartbeat_timeout_ms | bigint | ローカルノードのハートビートタイムアウト |
| heartbeat_elapsed_ms | bigint | ローカルノードがリーダーからのハートビートを確認してから経過したミリ秒数 |
| current_term | bigint | ローカルノードの現在の Raft ターム |
| commit_index | bigint | ローカルノードの現在の Raft コミットインデックス |
| apply_index | bigint | ローカルノードがカタログに適用された Raft コミットインデックス |
| last_log_term | bigint | 要求ログの最後の Raft ターム |
| last_log_index | bigint | 要求ログの最後の Raft インデックス |
| oldest_log_index | bigint | 要求ログに残っている最も古い Raft インデックス |
| newest_prunable_log_index | bigint | 要求ログから安全に削除できる最新の Raft インデックス |
| snapshot_term | bigint | 最後のスナップショットの Raft ターム |
| snapshot_index | bigint | 最後のスナップショットの Raft インデックス |
| nnodes | integer | Raft コンセンサス内のノードの数通常はグループ内のノードの数と同じである必要があります。 |
| nvoting_nodes | integer | Raft コンセンサスでの投票ノードの数 |

### bdr.stat_receiver

レプリケーションサブスクリプションレシーバープロセスに関する必要な情報をすべて含むビュー。

#### bdr.stat_receiver 列

| Column | Type | Description |
|---|---|---|
| worker_role | text | BDR ワーカーのロール常に'receiver' |
| worker_state | text | レシーバーワーカーの状態'running'、'down'、または 'disabled' のいずれかです。 |
| worker_pid | integer | レシーバーワーカーのプロセス ID |
| sub_name | name | レシーバーが属しているサブスクリプションの名前 |
| sub_slot_name | name | レシーバーが使用するレプリケーションスロット名 |
| source_name | name | このレシーバーのソースノード接続先。これは通常起点ノードと同じですが、フォワードモードサブスクリプションの場合は異なります。 |

次のページに続く

表 71 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| origin_name | name | このレシーバーのオリジンノード転送された変更を受信するノード。これは通常ソースノードと同じですが、フォワードモードサブスクリプションの場合は異なります。 |
| subscription_mode | char | サブスクリプションのモード、詳細については :ref:``bdr.subscription_summary` 列 <*bdr.subscription_summary* 列>` を参照してください。 |
| sub_replication_sets | text[] | このレシーバーがサブスクライブしているレプリケーションセット |
| sub_apply_delay | interval | 適用遅延インターバル |
| receive_lsn | pg_lsn | これまでに受け取った最後の変更の LSN |
| receive_commit_lsn | pg_lsn | これまでに受け取った最後のコミットの LSN |
| xact_apply_lsn | pg_lsn | 最後に適用されたトランザクション LSN |
| xact_flush_lsn | pg_lsn | 最後にフラッシュされたトランザクション LSN |
| xact_apply_timestamp | timestamp with time zone | 最後に適用されたトランザクションコミットのタイムスタンプ |
| worker_start | timestamp with time zone | 受信機が起動した時刻 |
| worker_xact_start | timestamp with time zome | レシーバーがローカル db トランザクションを開始した時間現在ローカルトランザクションを処理している場合、通常は NULL、詳細については、*pg_upgrade* の `xact_start`を参照してください。 |
| worker_backend_state_chai | timestamp with time zone | バックエンド状態変更のタイムスタンプ、詳細については、 *pg_upgrade* の `state_change`を参照してください |
| worker_backend_state | text | 現在のバックエンド状態、詳細については、*pg_upgrade* の `state`を参照してください。 |
| wait_event_type | text | レシーバーが現在待機している待機イベントのタイプある場合、詳細については、 *pg_upgrade* の `wait_event_type`を参照してください |
| wait_event | text | レシーバーが現在待機している正確なイベント存在する場合、詳細については *pg_upgrade* の `wait_event`を参照してください。 |

### bdr.stat_relation

各リレーションの適用統計を表示します。*bdr.track_relation_apply* で追跡が有効になっている場合にのみデータが含まれます

および特定のリレーションに対してデータがレプリケートされたかどうか。

`lock_acquire_time` は、 *bdr.track_apply_lock_timing* の場合にのみ更新されます

は on デフォルト `off` に設定されます。

*bdr.reset_relation_stats()* を呼び出すことにより、保存された関係統計をリセットできます。

#### bdr.stat_relation 列

| Column | Type | Description |
|---|---|---|
| nspname | name | リレーションのスキーマの名前 |
| relname | name | リレーションの名前 |
| relid | oid | リレーションの OID |
| total_time | double precision | リレーションのレプリケーションの処理に費やされた合計時間ミリ秒単位 |
| ninsert | bigint | リレーションに対してレプリケートされた挿入の数 |
| nupdate | bigint | リレーションに対して複製された更新の数 |
| ndelete | bigint | リレーションに対してレプリケートされた削除の数 |
| ntruncate | bigint | リレーションに対してレプリケートされたトランケートの数 |
| shared_blks_hit | bigint | リレーションの共有ブロックキャッシュヒットの合計数 |
| shared_blks_read | bigint | リレーションに対して読み取られた共有ブロックの合計数 |
| shared_blks_dirtied | bigint | リレーション用にダーティな共有ブロックの合計数 |
| shared_blks_written | bigint | リレーション用に書き込まれた共有ブロックの合計数 |
| blk_read_time | double precision | リレーションのブロックの読み取りに費やされた合計時間ミリ秒単位 `track_io_timing`が有効な場合、それ以外の場合はゼロ |
| blk_write_time | double precision | リレーションのブロックの書き込みに費やされた合計時間ミリ秒単位 `track_io_timing`が有効な場合、それ以外の場合はゼロ |
| lock_acquire_time | double precision | リレーションのロックの取得に費やされた合計時間ミリ秒単位 `bdr.track_apply_lock_timing`が有効な場合、それ以外の場合はゼロ |
| stats_reset | timestamp with time zone | 最後に統計をリセットした時間 `bdr.reset_relation_stats()`によって実行 |

### bdr.stat_routing_candidate_state

Raft リーダーのルーティング候補ノードに関する情報のビュー他のノードでは空。

#### bdr.stat_routing_candidate_state 列

| Column | Type | Description |
|---|---|---|
| node_group_name | name | この情報の対象となるグループ各グループは別個のルーティングプロキシを持つことができます。 |
| node_name | name | 候補ノード名 |

次のページに続く

表 73 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| node_route_fence | boolean | ノードはフェンスされています true の場合、リーダーまたは読み取り専用の接続ターゲットになることはできません。 |
| node_route_reads | boolean | ノードは読み取り専用の接続ターゲットとして考えられています |
| node_route_writes | boolean | ノードは書き込みリード候補として検討されています。 |
| last_message_time | timestamp with time zone | このノードによって見られた最後の Raft メッセージ任意の要求を含む時間ノードの活性度を確認するために使用されます |

### bdr.stat_routing_state

接続マネージャーが接続をルーティングするために使用する接続ルーティングの状態のビュー。

#### bdr.stat_routing_state 列

| Column | Type | Description |
|---|---|---|
| node_group_name | name | これはグループの情報です各グループは個別のルーティングプロキシを持つことができます。 |
| write_lead_name | name | 書き込みリードノードの名前 |
| previous_write_lead_name | name | 以前の書き込みリードノードの名前 |
| read_names | name[] | 読み取り専用接続がルーティングされるノードの配列 |
| write_candidate_names | name[] | フェールオーバーの場合に書き込みリードになるために必要なすべての基準と一致するノード |
| read_candidate_names | name[] | フェールオーバーの場合に読み取り専用の接続ターゲットになるために必要なすべての基準と一致するノード |

### bdr.stat_subscription

各サブスクリプションの適用統計を表示します。*bdr.track_subscription_apply* で追跡が有効になっている場合にのみデータが含まれます。

*bdr.reset_subscription_stats()* を呼び出すことにより、保存されたサブスクリプション統計をリセットできます。

#### bdr.stat_subscription 列

| Column | Type | Description |
|---|---|---|
| sub_name | name | サブスクリプションの名前 |

次のページに続く

表 75 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| subid | oid | サブスクリプションの OID |
| mean_apply_time | double precision | 適用トランザクションごとの平均時間ミリ秒 |
| nconnect | bigint | このサブスクリプションがアップストリームに接続した回数 |
| ncommit | bigint | このサブスクリプションが行ったコミット数 |
| nabort | bigint | ライタがこのサブスクリプションに対して実行した中止の数 |
| nerror | bigint | このサブスクリプションでライタがヒットしたエラーの数 |
| nskippedtx | bigint | このサブスクリプションのライタによってスキップされたトランザクションの数`skip_transaction`競合リゾルバーのため |
| ninsert | bigint | このサブスクリプションが行った挿入の数 |
| nupdate | bigint | このサブスクリプションが行った更新の数 |
| ndelete | bigint | このサブスクリプションが行った削除数 |
| ntruncate | bigint | このサブスクリプションが行った切り捨ての数 |
| nddl | bigint | このサブスクリプションが実行した DDL 操作の数 |
| ndeadlocks | bigint | デッドロックが発生したエラーの数 |
| nretries | bigint | ライタが実行した再試行数完全な再起動/再接続を行わずに |
| nstream_writer | bigint | ライタにストリーミングされたトランザクションの数 |
| nstream_file | bigint | ファイルにストリーミングされたトランザクションの数 |
| nstream_commit | bigint | コミットされたストリーミングトランザクションの数 |
| nstream_abort | bigint | アボートされたストリーミングトランザクションの数 |
| nstream_start | bigint | 処理された STREAM START メッセージの数 |
| nstream_stop | bigint | 処理された STREAM STOP メッセージの数 |
| nstream_commit | bigint | コミットされたストリーミングトランザクションの数 |
| nstream_abort | bigint | アボートされたストリーミングトランザクションの数 |
| nstream_prepare | bigint | 準備されたストリーミングトランザクションの数 |
| nstream_insert | bigint | 処理されたストリーミング挿入の数 |
| nstream_update | bigint | 処理されたストリーミング更新の数 |
| nstream_delete | bigint | 処理されたストリーミング削除の数 |
| nstream_truncate | bigint | 処理されたストリーミングトランケートの数 |
| shared_blks_hit | bigint | サブスクリプションによる共有ブロックキャッシュヒットの合計数 |
| shared_blks_read | bigint | サブスクリプションによって読み取られた共有ブロックの合計数 |
| shared_blks_dirtied | bigint | サブスクリプションによってダーティされた共有ブロックの合計数 |
| shared_blks_written | bigint | サブスクリプションによって書き込まれた共有ブロックの合計数 |

次のページに続く

表 75 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| blk_read_time | double precision | サブスクリプションがブロックの読み取りに費やした合計時間ミリ秒単位`track_io_timing`が有効な場合、それ以外の場合はゼロ |
| blk_write_time | double precision | サブスクリプションがブロックの書き込みに費やした合計時間ミリ秒単位`track_io_timing`が有効な場合、それ以外の場合はゼロ |
| connect_time | timestamp with time zone | 現在の上流接続が確立された時間、接続されていない場合は NULL |
| last_disconnect_time | timestamp with time zone | 最後のアップストリーム接続がドロップされた時刻 |
| start_lsn | pg_lsn | このサブスクリプションが上流からレプリケーションの開始を要求した LSN |
| retries_at_same_lsn | bigint | サブスクリプションが同じ LSN 値からリスタートされた試行回数 |
| curr_ncommit | bigint | 現在の接続が確立された後にこのサブスクリプションが実行したコミット数 |
| npre_commit_confirmation | bigint | CAMO パートナーによるプリコミット確認の数 |
| npre_commit | bigint | プリコミット数 |
| ncommit_prepared | bigint | 準備されたトランザクションのコミット数 |
| nabort_prepared | bigint | 準備されたトランザクションアボートの数 |
| nprovisional_waits | bigint | 同時適用トランザクションによる同じタプルの更新/削除操作の数。これらは暫定的な待機です。 *Parallel Apply* を参照 |
| ntuple_waits | bigint | 安全に適用されるまで待機した更新/削除操作の数。 *Parallel Apply* を参照 |
| ncommit_waits | bigint | コミットされる前に待機する必要があった、完全に適用されたトランザクションの数。 *Parallel Apply* を参照 |
| stats_reset | timestamp with time zone | 最後に統計をリセットした時間`bdr.reset_subscription_stats()`によって実行 |

### bdr.stat_worker

PGD マネージャーワーカーの概要情報とワーカーごとの統計を含むビュー。

#### `bdr.stat_worker` 列

| Column | Type | Description |
|---|---|---|
| worker_role | text | BDR ワーカーのロール |
| worker_pid | integer | ワーカーのプロセス ID |
| sub_name | name | ワーカーが関連しているサブスクリプションの名前がある場合 |

次のページに続く

表 76 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| worker_start | timestamp with time zone | ワーカーが開始した時刻 |
| worker_xact_start | timestamp with time zone | ワーカーがローカル db トランザクションを開始した時間。詳細については、 *pg_upgrade* の `xact_start`を参照してください |
| worker_xid | xid | ワーカーのトランザクション ID、詳細については、*pg_upgrade* の `backend_xid`を参照してください。 |
| worker_xmin | xid | ワーカーが必要とする最も古いトランザクション ID。詳細については、 *pg_upgrade* の `backend_xmin`を参照してください。 |
| worker_backend_state_chai | timestamp with time zone | バックエンド状態変更のタイムスタンプ詳細については、 *pg_upgrade* の `state_change`を参照してください |
| worker_backend_state | text | 現在のバックエンド状態、詳細については、*pg_upgrade* の `state`を参照してください |
| wait_event_type | text | ワーカーが現在待機している待機イベントのタイプ存在する場合、詳細については、 *pg_upgrade* の `wait_event_type`を参照してください。 |
| wait_event | text | ワーカーが待機している正確なイベント存在する場合、詳細については、 *pg_upgrade* の `wait_event`を参照してください。 |
| blocked_by_pids | integer[] | ワーカーをブロックする PID のリスト 存在する場合 |
| query | text | ワーカーによって現在実行中のクエリ |
| worker_query_start | timestamp with time zone | ワーカーによって実行された現在のクエリが開始されたタイムスタンプ |

### bdr.stat_writer

各サブスクリプションレプリケーションライタの概要情報と統計を含むビュー。サブスクリプションごとに複数のライターが存在する場合があります。

#### bdr.stat_writer 列

| Column | Type | Description |
|---|---|---|
| worker_role | text | BDR ワーカーのロール常に' ライタ' |
| worker_state | text | ワーカーの状態'running'、'down'、または'disabled' のいずれかです。 |
| worker_pid | integer | ライタのプロセス ID |
| sub_name | name | ライタが属しているサブスクリプションの名前 |
| writer_nr | integer | 同じサブスクリプションのライタグループのライターインデックス |
| nxacts | bigint | ライタが開始以降に処理したトランザクションの数 |
| ncommits | bigint | start 以降にライタが処理したコミットの数 |

次のページに続く

表 77 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| naborts | bigint | start 以降にライタが処理した中止の数 |
| commit_queue_position | integer | 同じライタグループ内の他のライターに対してトランザクションをシリアル化する場合、コミットキュー内の位置 |
| xact_source_xid | xid | ソースノードで現在処理されているトランザクションのトランザクション ID |
| xact_source_commit_lsn | pg_lsn | ソースノードで現在処理されているトランザクションの LSN |
| xact_nchanges | bigint | 現在処理されているトランザクションで書き込まれた変更の数 1000 変更ごとに更新 |
| xact_origin_node_name | name | 現在処理されているトランザクションのオリジンノード |
| xact_origin_lsn | pg_lsn | 現在処理されているトランザクションのオリジン LSN |
| xact_origin_timestamp | timestamp with time zone | 現在処理されているトランザクションのオリジンコミットタイムスタンプ |
| streaming_allowed | boolean | ライタは大規模なトランザクションの直接ストリームを受信できます |
| is_streaming | boolean | ライタは現在、大規模なトランザクションの直接ストリームを受信しています |
| nstream_file | bigint | ライタが処理したストリームファイルの数 |
| nstream_writer | bigint | ライタが処理した直接ストリーミングトランザクションの数 |
| worker_start | timestamp with time zone | ライタが書き始めた時期 |
| worker_xact_start | timestamp with time zone | ライタがローカル db トランザクションを開始した時間詳細については、*pg_upgrade* の xact_start を参照してください。 |
| worker_xid | xid | ワーカーのトランザクション ID 詳細については、*pg_upgrade* の `backend_xid`を参照してください。 |
| worker_xmin | xid | ワーカーが必要とする最も古いトランザクション ID 詳細については、 *pg_upgrade* の `backend_xmin`を参照してください。 |
| worker_backend_state_chai | timestamp with time zone | バックエンド状態変更のタイムスタンプ詳細については、*pg_upgrade* の `state_change`を参照してください。 |
| worker_backend_state | text | 現在のバックエンド状態詳細については、*pg_upgrade* の `state`を参照してください。 |
| wait_event_type | text | ライタが現在待機している待機イベントのタイプある場合、詳細については、 *pg_upgrade* の `event_type`を参照してください。 |
| wait_event | text | ライタが待機している正確なイベント存在する場合、詳細については *pg_upgrade* の `wait_event`を参照してください。 |
| blocked_by_pids | integer[] | ライタをブロックする PID のリスト 存在する場合 |

次のページに続く

表 77 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| query | text | ライタによって現在実行中のクエリ通常は DDL にのみ設定されます |
| worker_query_start | timestamp with time zone | ワーカーによって実行された現在のクエリが開始されたタイムスタンプ |
| command_progress_cmdtag | text | 進行状況追跡のあるコマンドの場合、ライタによって現在処理されているコマンドを示します。 'CREATE INDEX'、'CREATE INDEX CONCURRENTLY'、'REINDEX'、'REINDEX CONCURRENTLY'、'CLUSTER'、および'VACUUM FULL' のいずれかです。 |
| command_progress_relation | text | 進行状況追跡を使用するコマンドの場合、コマンドがどの関係を作業しているかを特定します |
| command_progress_phase | text | 進行状況追跡を含むコマンドの場合、コマンドが行っている現在のフェーズの名前、詳細については、Postgres ドキュメントの *pg_upgrade* を参照してください |
| command_progress_count | integer | 進行状況追跡を使用するコマンドの場合、このコマンドが通過したフェーズの数 |
| command_progress_phase_nr | integer | 進行状況追跡を使用するコマンドの場合、`command_progress_count`のフェーズの数 |
| command_progress_phase_tupl | real | 進行状況追跡を使用するコマンドの場合、コマンドの現在のフェーズで処理する必要がある行数フェーズが行を処理する場合 |
| command_progress_tuples_don | bigint | 進行状況追跡を使用するコマンドの場合、コマンドの現在のフェーズが既に処理した行数フェーズが処理行の場合 |

### bdr.subscription

このカタログテーブルには、ローカル PGD ノードが所有するすべてのサブスクリプションとそのモードがリストされています。

#### bdr.subscription 列

| Name | Type | Description |
|---|---|---|
| sub_id | oid | サブスクリプションの ID |
| sub_name | name | サブスクリプションの名前 |
| nodegroup_id | oid | ノードグループの ID |
| origin_node_id | oid | オリジンノードの ID |
| source_node_id | oid | ソースノードの ID |
| target_node_id | oid | ターゲットノードの ID |
| subscription_mode | char | サブスクリプションのモード |
| sub_enabled | bool | サブスクリプションが有効かどうかレプリケーションである必要があります。 |

次のページに続く

表 78 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| apply_delay | interval | このサブスクリプションの変更の適用はどのくらい遅れますか通常 0 |
| slot_name | name | このサブスクリプションで使用されるアップストリームのスロット |
| origin_name | name | このサブスクリプションで使用されるローカルオリジン |
| num_writers | int | このサブスクリプションが使用するライタプロセスの数 |
| streaming_mode | char | サブスクリプションのストリーミング構成 |
| replication_sets | text[] | このサブスクリプションによってレプリケートされたレプリケーションセット NULL = all |
| forward_origin | text[] | このサブスクリプションによって転送されたオリジン NULL = all |

### bdr.subscription_summary

このビューには、ローカルノードが他のノードに対して持っているすべての PGD サブスクリプションに関する概要情報が含まれています。

#### bdr.subscription_summary 列

| Name | Type | Description |
|---|---|---|
| node_group_name | name | ノードが一部である PGD グループの名前 |
| sub_name | name | サブスクリプションの名前 |
| origin_name | name | オリジンノードの名前 |
| target_name | name | ターゲットノードの名前通常はローカルノード |
| sub_enabled | bool | サブスクリプションは有効になっていますか |
| sub_slot_name | name | このサブスクリプションで使用されるオリジンノードのスロット名 |
| sub_replication_sets | text[] | サブスクライブされたレプリケーションセット |
| sub_forward_origins | text[] | サブスクリプションは、オリジン以外のノードから転送された変更を受け入れますか |
| sub_apply_delay | interval | オリジンと比較して、これだけの遅延トランザクション |
| sub_origin_name | name | このサブスクリプションで使用されるレプリケーション元の名前 |
| bdr_subscription_mode | char | サブスクリプションモード |
| subscription_status | text | サブスクリプションワーカーのステータス |
| node_group_id | oid | ノードが一部である PGD グループの OID |
| sub_id | oid | サブスクリプションの OID |
| origin_id | oid | 元ノードの OID |
| target_id | oid | ターゲットノードの OID |

次のページに続く

表 79 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| receive_lsn | pg_lsn | 受信した変更またはメッセージの最新の LSN これは、再起動の場合に後退する可能性があります |
| receive_commit_lsn | pg_lsn | 受け取った最後に COMMIT の最新の LSN これは、再起動の場合に後退する可能性があります |
| last_xact_replay_lsn | pg_lsn | このサブスクリプションで再生された最後のトランザクションの LSN |
| last_xact_flush_lsn | timestamptz | このサブスクリプションで再生され、永続的にディスクにフラッシュされる最後のトランザクションの LSN |
| last_xact_replay_timestamp | timestamptz | このサブスクリプションで再生された最後のトランザクションのタイムスタンプ |

### bdr.tables

このビューは、レプリケーションセットのテーブルメンバーシップに関する情報をリストします。テーブルが複数のレプリケーションセットに存在する場合、このテーブルに複数回表示されます。

#### bdr.tables 列

| Name | Type | Description |
|---|---|---|
| relid | oid | リレーションの OID |
| nspname | name | スキーマリレーションの名前は次のとおりです |
| relname | name | リレーションの名前 |
| set_name | name | レプリケーションセットの名前 |
| set_ops | text[] | レプリケートされた操作のリスト |
| rel_columns | text[] | レプリケートされた列のリスト NULL = すべての列* |
| row_filter | text | 行フィルタリング式 |
| conflict_detection | text | 使用される競合検出方法 row_origin デフォルト、row_version または column_level |

(*) これらの列は将来の使用のために予約されており、現在 NULL である必要があります。

### bdr.taskmgr_work_queue

タスクマネージャーによって作成および処理された作業項目が含まれています。作業項目は 1 つのノードでのみ作成され、別のノードで処理されます。

#### bdr.taskmgr_work_queue 列

| Column | Type | Description |
|---|---|---|
| ap_wq_workid | bigint | ワークアイテムの一意の ID |

次のページに続く

表 81 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| ap_wq_ruleid | int | autopartition_rules にリストされているルールの ID。ルールは bdr.autopartition コマンドを使用して指定されます |
| ap_wq_relname | name | タスクが属するリレーションの名前 |
| ap_wq_relnamespace | name | この作業項目のルールで指定されたテーブルスペースの名前 |
| ap_wq_partname | name | 作業項目によって作成されたパーティションの名前 |
| ap_wq_work_category | char | 作品カテゴリー; *c* パーティションの作成、*m* パーティションの移行 、 *d* パーティションのドロップ 、または *a* パーティションの変更 です。 |
| ap_wq_work_sql | text | 作業項目の SQL クエリー |
| ap_wq_work_depends | Oid[] | ワークアイテムが依存するノードの OID |

### bdr.taskmgr_workitem_status

各ノードでローカルに更新される作業項目のステータス。

#### bdr.taskmgr_workitem_status 列

| Column | Type | Description |
|---|---|---|
| ap_wi_workid | bigint | ワークアイテムの ID |
| ap_wi_nodeid | Oid | ワークアイテムが処理されているノードの OID |
| ap_wi_status | char | ステータス`q`キューイング、`c`完全、`f`failed、または`u`unknown のいずれかです |
| ap_wi_started_at | timestamptz | 作業項目の開始タイムスタンプ |
| ap_wi_finished_at | timestamptz | 作業項目の終了タイムスタンプ |

### bdr.taskmgr_local_work_queue

タスクマネージャーによって作成および処理された作業項目が含まれています。これは bdr.taskmgr_work_queue に似ていますが、これらの作業項目はローカル管理テーブル用である点が異なります。各ノードは、クラスター内の他のノードから独立して、独自のローカル作業項目を作成および処理します。

#### bdr.taskmgr_local_work_queue 列

| Column | Type | Description |
|---|---|---|
| ap_wq_workid | bigint | ワークアイテムの一意の ID |
| ap_wq_ruleid | int | autopartition_rules にリストされているルールの ID。ルールは bdr.autopartition コマンドを使用して指定されます |

次のページに続く

表 83 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| ap_wq_relname | name | タスクが属するリレーションの名前 |
| ap_wq_relnamespace | name | この作業項目のルールで指定されたテーブルスペースの名前。 |
| ap_wq_partname | name | 作業項目によって作成されたパーティションの名前 |
| ap_wq_work_category | char | カテゴリ; *c* パーティションの作成 、 *m* パーティションの移行 、 *d* パーティションのドロップ 、または *a* パーティションの変更 です。 |
| ap_wq_work_sql | text | 作業項目の SQL クエリー |
| ap_wq_work_depends | Oid[] | 常に NULL |

### bdr.taskmgr_local_workitem_status

ローカルに管理されるテーブルの作業項目のステータス。

#### bdr.taskmgr_local_workitem_status 列

| Column | Type | Description |
|---|---|---|
| ap_wi_workid | bigint | ワークアイテムの ID |
| ap_wi_nodeid | Oid | ワークアイテムが処理されているノードの OID |
| ap_wi_status | char | ステータス`q`キューイング、`c`完全、`f`failed、または`u`unknown のいずれかです |
| ap_wi_started_at | timestamptz | 作業項目の開始タイムスタンプ |
| ap_wi_finished_at | timestamptz | 作業項目の終了タイムスタンプ |

### bdr.trigger

このビューでは、作成されたすべてのストリームトリガーを表示できます。多くの場合、ここでのトリガーは `bdr.create_conflict_trigger` から作成されます。

#### bdr.trigger 列

| Name | Type | Description |
|---|---|---|
| trigger_id | oid | トリガーの ID |
| trigger_reloid | regclass | 関連するファンクションの名前 |
| trigger_pgtgid | oid | Postgres トリガー ID |
| trigger_type | char | トリガー呼び出しの種類 |
| trigger_name | name | トリガーの名前 |

### bdr.triggers

読みやすい列を含む bdr.trigger の展開ビュー。

| Name | Type | Description |
|---|---|---|
| trigger_name | name | トリガーの名前 |
| event_manipulation | text | オペレーション |
| trigger_type | bdr.trigger_type | トリガーの種類 |
| trigger_table | bdr.trigger_reloid | トリガーを呼び出すテーブル |
| trigger_function | name | 使用するファンクション |

### bdr.workers

実行中の PGD ワーカープロセスに関する情報。

これを pid を使用して bdr.stat_activity と結合して、PGD ワーカーの状態についてのさらに多くの洞察を得ることができます。

#### bdr.workers 列

| Name | Type | Description |
|---|---|---|
| worker_pid | int | ワーカープロセスのプロセス ID |
| worker_role | int | ワーカーロールの数値表現 |
| worker_role_name | text | ワーカーロールの名前 |
| worker_subid | oid | ワーカーが関連付けられている場合のサブスクリプション ID |

### bdr. ライター

PGD ライタプロセスに関する特定の情報。

#### bdr.writers 列

| Name | Type | Description |
|---|---|---|
| sub_name | name | サブスクリプションの名前 |
| pid | int | ワーカープロセスのプロセス ID |
| syncing_rel | int | 同期される関係の OID 存在する場合 |
| streaming_allowed | text | このライタはライタへの直接ストリーミングのターゲットになりますか |
| is_streaming | bool | このライタにストリーミングされているトランザクションはありますか |

次のページに続く

表 88 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| remote_xid | xid | 処理中のトランザクションのリモートトランザクション ID 存在する場合 |
| remote_commit_lsn | pg_lsn | 最後に処理されたコミットの LSN |
| commit_queue_position | int | 内部コミットキューの位置 |
| nxacts | bigint | このライタによって処理されたトランザクションの数 |
| ncommits | bigint | このライタがコミットしたトランザクションの数 |
| naborts | bigint | このライタによって中止されたトランザクションの数 |
| nstream_file | bigint | このライタによって処理されたストリームからファイルへのトランザクションの数 |
| nstream_writer | bigint | このライタによって処理されたライターへのストリーミングトランザクションの数 |
| xact_nchanges | bigint | このライタによって処理された変更の数 1000 行ごとに更新 |

### bdr.worker_tasks

bdr.worker_tasks ビューは、PGD の現在のワーカー起動レート制限状態と、バックグラウンドでのワーカーの起動と登録アクティビティに関するいくつかの基本的な統計を表示します。

ここにリストされている他のビューとは異なり、現在のデータベースと PGD ノードに固有ではありません。現在の PostgreSQL インスタンスのすべての PGD ノードの状態が表示されます。現在のデータベースに参加してフィルタリングします。

bdr.worker_tasks は、walsender と出力プラグインを追跡しません。

#### bdr.worker_tasks 列

| Column | Type | Description |
|---|---|---|
| task_key_worker_role | integer | ワーカーロール識別子 |
| task_key_worker_role_nam | text | ワーカーロール名 |
| task_key_dboid | oid | データベース識別子、使用可能な場合 |
| datname | name | データベースの名前使用可能な場合 |
| task_key_subid | oid | サブスクリプション識別子、可能な場合 |
| sub_name | name | サブスクリプションの名前使用可能な場合 |
| task_key_ext_libname | name | ライブラリの名前おそらく bdr |
| task_key_ext_funcname | name | ファンクションエントリーポイントの名前 |
| task_key_ext_workername | name | ワーカーに割り当てられた名前 |
| task_key_remoterelid | oid | リモート同期関係の識別子使用可能な場合 |
| task_pid | integer | ワーカーのプロセス ID |
| task_registered | timestamp with time zone | ワーカー登録タイムスタンプ |
| since_registered | interval | ワーカーが登録されてからのインターバル |

次のページに続く

表 89 – 前のページからの続き

| Column | Type | Description |
|---|---|---|
| task_attached | timestamp with time zone | ワーカー接続タイムスタンプ |
| since_attached | interval | ワーカーが接続してからのインターバル |
| task_exited | timestamp with time zone | ワーカー終了タイムスタンプ |
| since_exited | interval | ワーカーが終了してからのインターバル |
| task_success | boolean | ワーカーはまだ実行されていますか？ |
| task_next_launch_not_befo | timestamp with time zone | ワーカーが再度再起動されるときのタイムスタンプ |
| until_launch_allowed | interval | 次の起動までの残り時間 |
| task_last_launch_requestor | integer | 起動を要求したプロセス ID |
| task_last_launch_request_ti | timestamp with time zone | 要求が行われたときのタイムスタンプ |
| since_last_request | interval | 最後の要求からのインターバル |
| task_last_launch_request_a | boolean | 最後の要求は成功しましたか？ |
| task_nrequests | integer | 要求の数 |
| task_nregistrations | integer | 登録数 |
| task_prev_pid | integer | 前世代のプロセス ID |
| task_prev_registered | timestamp with time zone | 以前に登録されたタスクのタイムスタンプ |
| since_prev_registered | interval | 前回の登録からのインターバル |
| task_prev_launched | timestamp with time zone | 前回の起動のタイムスタンプ |
| since_prev_launched | interval | 前回の起動からのインターバル |
| task_prev_exited | timestamp with time zone | 前のタスクが終了した時のタイムスタンプ |
| since_prev_exited | interval | 前のタスクが終了してからの間隔 |
| task_first_registered | timestamp with time zone | 最初に登録が発生したときのタイムスタンプ |
| since_first_registered | interval | 最初の登録からのインターバル |

### System functions

主に SQL から呼び出すファンクションを使用して、PGD 管理を実行します。PGD のすべてのファンクションは `bdr` スキーマで公開されています。`search_path` に `bdr` を配置する代わりに、これらのファンクションへの呼び出しをスキーマ修飾します。

### バージョン情報ファンクション

#### bdr.bdr_version

このファンクションは、現在使用している BDR 拡張機能のバージョンのテキスト表現を取得します。

#### bdr.bdr_version_num

このファンクションは、現在使用されている BDR 拡張機能のバージョン番号を取得します。バージョン番号は単調増加であるため、この値をより小さいおよびより大きい比較に使用できます。

次の式は、メジャーバージョン、マイナーバージョン、およびパッチリリースで構成されるバージョン番号を単一の数値に結果ます。

```
MAJOR_VERSION  *10000 + MINOR_VERSION*  100 + PATCH_RELEASE
```

### システム情報ファンクション

#### bdr.get_relation_stats

リレーション情報を返します。

#### bdr.get_subscription_stats

現在のサブスクリプション統計を結果ます。

### システムおよび進行状況情報パラメーター

PGD は、 `SHOW` または `current_setting()` ファンクションなどを使用して SQL で直接照会できるいくつかのパラメーターを公開します。クライアントアプリケーションから `PQparameterStatus` または同等のものを使用することもできます。

#### bdr.local_node_id

セッションを初期化すると、これはクライアントが接続しているノード ID に設定されます。これにより、アプリケーションは、透明なプロキシの背後にある場合でも、接続されているノードを認識できます。

接続プールとプロキシ でも使用されます。

#### bdr.last_committed_lsn

非同期トランザクションの `COMMIT` ごとに、このパラメーターは更新されて、オリジンノードのコミットレコードの最後を指します。`bdr.wait_for_apply_queue` と組み合わせると、アプリケーションはマルチプルのノードにわたってカウサル読み取りを実行できます。つまり、トランザクションがリモートで可視されるまで待機します。

#### transaction_id

CAMO トランザクションが進行中の場合、transaction_id が更新されて、割り当てられたトランザクション ID が表示されます。このパラメーターは、PQparameterStatus または同等の使用を使用してのみ照会でき、SQL ではアクセスできません。 アプリケーションの使用 を参照

使用例

### ノードステータスファンクション

#### bdr.is_node_connected

##### 概要

```
bdr.is_node_connected(node_name name)
```

指定されたピアのワルセンダーがこのノードでアクティブかどうかを確認することにより、ブール値を返します。

#### bdr.is_node_ready

##### 概要

```
bdr.is_node_ready(node_name name, span interval DEFAULT NULL)
```

ラグが指定されたスパンより低いか、それ以外の場合は TO ASYNC の timeout より低いかどうかをチェックすることにより、ブール値を返します。

### コンセンサスファンクション

#### bdr.consensus_disable

サーバーが再起動するか、bdr.consensus_enable を使用して再有効化されるまで、ローカルノードのコンセンサスワーカーを無効にします。最初に発生した方。

> **警告**
>
> コンセンサスを無効にすると、PGD の一部の機能が無効になり、長時間無効になっていると EDB Postgres Distributed クラスターの可用性に影響します。この機能は、テクニカルサポートと協力する場合にのみ使用します。

#### bdr.consensus_enable

ローカルノードで無効になったコンセンサスワーカーが再度有効になりました。

### bdr.consensus_proto_version

ローカルノードが現在使用しているコンセンサスプロトコルのバージョンを返します。

PGD グループの再構成の内部メカニズムに必要です。

### bdr.consensus_snapshot_export

#### 概要

```
bdr.consensus_snapshot_export(version integer DEFAULT NULL)
```

ローカルノードの現在コミットおよび適用された状態から新しい PGD コンセンサススナップショットを生成し、bytea として結果ます。

デフォルトでは、サポートされている最新の Raft バージョンのスナップショットがエクスポートされます。ただし、明示的な version 番号を渡すことにより、それをオーバーライドできます。

エクスポートするノードは現在の Raft リーダーである必要はなく、リーダーの最新の状態で完全に最新の状態である必要はありません。ただし、bdr.consensus_snapshot_import() は、このようなスナップショットを受け入れない場合があります。

新しいスナップショットは、ローカルノードの bdr.local_consensus_snapshot テーブルに自動的に保存されません。呼び出し元にのみ返されます。

生成されたスナップショットは、エクスポートノードの Raft ログ位置の背後にある同じ PGD ノードグループ内の他のノードの bdr.consensus_snapshot_import() に渡される場合があります。

この機能が動作するには、ローカル PGD コンセンサスワーカーを無効にする必要があります。一般的な使用法は次のとおりです。

```
SELECT bdr.bdr_consensus_disable();
\copy (SELECT * FROM bdr.consensus_snapshot_export()) TO my_node_consensus_snapshot.
↪data
SELECT bdr.bdr_consensus_enable();
```

PGD コンセンサスワーカーが無効になっている間

- ノードでの DDL ロックの試行は失敗するか、タイムアウトします。
- galloc シーケンスは新しい値を取得しません。
- Eage および CAMO トランザクションの一時停止またはエラー。
- 分散コンセンサスシステムを必要とする他の機能が中断されます。必要なダウンタイムは、一般に非常に短時間です。

ユースケースによっては、 bdr.local_consensus_snapshot テーブルの snapshot フィールドから既に存在するスナップショットを抽出し、代わりにそれを使用することが現実的である場合があります。その際、コンセンサスワーカーを停止する必要はありません。

### bdr.consensus_snapshot_import

#### 概要

```
bdr.consensus_snapshot_import(snapshot bytea)
```

bdr.consensus_snapshot_export() によって、通常は同じ PGD ノードグループ内の別のノードからエクスポートされたコンセンサススナップショットをインポートします。

別のノード上の bdr.local_consensus_snapshot テーブルの snapshot フィールドから直接抽出されたスナップショットを使用することもできます。

この機能は、破損またはユーザーエラーが発生した場合に、PGD ノードのカタログ状態を既知の良好な状態にリセットするのに役立ちます。

スナップショットが生成されたときに、インポートノードの apply_index がスナップショットエクスポートノードの commit_index 以下の場合、スナップショットをインポートできます。bdr.get_raft_status() を参照してください。ログが既に進みすぎているためにスナップショットを受け入れることができないノードは、エラーを発生させ、変更は行いません。スナップショットがインポートされると、ノードは現在のリーダーから残りの変更を取得するため、インポートされたスナップショットは完全に最新の状態である必要はありません。

この機能が動作するには、インポートノードで PGD コンセンサスワーカーを無効にする必要があります。詳細については、bdr.consensus_snapshot_export() のノートを参照してください。

この機能を使用して、次のことを実行してローカルノードに新しい Raft スナップショットを生成させることができます。

```
SELECT bdr.consensus_snapshot_import(bdr.consensus_snapshot_export());
```

このアプローチでは、現在適用されているログの位置まで Raft ログが切り捨てられる場合があります。

### bdr.consensus_snapshot_verify

#### 概要

```
bdr.consensus_snapshot_verify(snapshot bytea)
```

bdr.consensus_snapshot_export() によってエクスポートされた指定されたコンセンサススナップショットを確認します。スナップショットヘッダーには、生成されたバージョンが含まれており、ノードは同じバージョンに対して検証しようとします。

スナップショットは、クラスター内の同じノードまたは他のノードでエクスポートされた可能性があります。スナップショットを検証するノードがエクスポートされたスナップショットのバージョンをサポートしていない場合、エラーが発生します。

#### bdr.get_consensus_status

現在のコンセンサス Raft ワーカーに関するステータス情報を結果ます。

#### bdr.get_raft_status

現在のコンセンサス Raft ワーカーに関するステータス情報を結果ます。bdr.get_consensus_status のエイリアス。

#### bdr.raft_leadership_transfer

##### 概要

```
bdr.raft_leadership_transfer(node_name text,
                             wait_for_completion boolean,
                             node_group_name text DEFAULT NULL)
```

node_name で指定されたノードに Raft リーダーになるように要求します。要求は任意の PGD ノードから開始でき、現在のリーダーに内部転送されて、指定されたノードにリーダーシップを転送します。指定されたノードは、完全な投票権を持つ ACTIVE PGD ノードである必要があります。

wait_for_completion が false の場合、要求はベストエフォートベースで提供されます。ノードが bdr.raft_global_lection_timeout 期間にリーダーになることができない場合、他の有効なノードが再びリーダーになります。また、リーダーシップは Raft プロトコルごとに時間の経過とともに変更される場合があります。true の戻り結果は、要求が正常に送信されたことのみを示します。

wait_for_completion が true の場合、ファンクションは指定されたノードが新しいリーダーになるまで待機し、要求されたノードが Raft リーダーになれなかった場合たとえば、ネットワークの問題のため、無限に待機します。したがって、無限ループを防ぐために、常に statement_timeout と wait_for_completion を設定することをお勧めします。

node_group_name はオプショナルであり、リーダーシップの移転が発生するノードグループの名前を指定するために使用できます。指定しない場合、デフォルトの NULL になり、クラスター内のトップレベルのグループとして解釈されます。node_group_name を指定すると、ファンクションは指定されたノードグループ内でのみリーダーシップを譲渡します。

##### ユーティリティファンクション

#### bdr.wait_slot_confirm_lsn

このセッションの最後の書き込みが 1 つまたはすべてのノードで再生されるまで待機できます。

スロットが特定の LSN を通過するまで待機します。位置が指定されない場合、現在の書き込み位置がローカルノードで使用されます。

スロット名が渡されない場合、すべての PGD スロットが LSN を通過するまで待機します。

ファンクションは、他のノードからの変更を 1000 ミリ秒ごとにポーリングします。

スロットが同時にドロップされた場合、そのスロットの待機は終了します。ノードが現在ダウンしており、スロットを更新していない場合、待機は継続されます。その場合、statement_timeout をより早く完了するように設定することができます。

Optimized Topology を使用している場合、代わりに bdr.wait_node_confirm_lsn を使用することをお勧めします。 )

#### 概要

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

#### 注

使用するには bdr_application 権限が必要です。

#### パラメーター

| Parameter | Description |
|---|---|
| *slot_name* | 待機するレプリケーションスロットの名前。 NULL の場合、すべての PGD スロットを待機します。 |
| *target_lsn* | 待機する LSN。 NULL の場合、ローカルノードの現在の書き込み LSN を使用します。 |

### bdr.wait_node_confirm_lsn

ノードが特定の LSN を通過するまで待機します。

この機能を使用すると、このセッションの最後の書き込みが 1 つまたはすべてのノードで再生されるまで待機できます。

ファンクションは、呼び出されると、ノードが特定の LSN を渡すのを待機します。LSN が指定されない場合、現在の wal_flush_lsn pg_current_wal_flush_lsn() ファンクションを使用しての位置がローカルノードで使用されます。ノード名パラメーターを指定すると、そのノードが LSN を渡すのを待機するようにファンクションに指示します。ノード名が NULL を渡すことにより指定されない場合、ファンクションはすべてのノードが LSN を渡すまで待機します。

*bdr.wait_slot_confirm_lsn* ではなく Optimized Topology を使用している場合、このファンクションを使用することをお勧めします。

これは、最適化トポロジでは、すべてのノードにレプリケーションスロットがあるわけではないため、ファンクション bdr.wait_slot_confirm_lsn が予想通りに動作しない場合があるためです。bdr.wait_node_confirm_lsn は、レプリケーションスロットのないノードで動作するように設計されており、代替戦略を使用してノードの進行状況を判断します。

ノードが現在ダウンしているか、更新していない場合、または単に接続できない場合、待機は無期限に継続されます。この状態を回避するには、statement_timeout を待機する準備ができている最大時間に設定します。

#### 概要

```
bdr.wait_node_confirm_lsn(node_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_name* | 待機するノードの名前。 NULL の場合、すべてのノードを待機します。 |
| *target_lsn* | 待機する LSN。 NULL の場合、ローカルノードで現在の wal_flush_lsn を使用します。 |

#### 注

使用するには bdr_application 権限が必要です。

### bdr.wait_for_apply_queue

ファンクション bdr.wait_for_apply_queue を使用すると、PGD ノードは、特定の PGD ノードから発生した特定のトランザクションのローカルアプリケーションを待機できます。そのピアノードからのすべてのトランザクションがローカルに適用された後にのみ返されます。アプリケーションまたはプロキシはこの機能を使用して、古い読み取りを防止できます。

便宜上、PGD は CAMO および CAMO パートナーノードにこのファンクションのバリアントを提供します。*bdr.wait_for_camo_partner_queue* を参照してください。

特定の LSN が指定された場合、それは、ピアが待機するリカバリストリームのポイントです。これは、以前のまたは同時接続でそのピアノードから取得した bdr.last_committed_lsn と使用できます。

指定された target_lsn が NULL の場合、このファンクションはローカル受信バッファをチェックし、指定されたピアノードから受け取った最後のトランザクションの LSN を使用し、既に受信したすべてのトランザクションが適用されるのを効果的に待機します。これは、ピアノードに障害が発生し、どのトランザクションが送信されたかが不明な場合に特に役立ちます。この場合、送信者側でまだ転送中またはバッファリングされているトランザクションは待機されません。

#### 概要

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *peer_node_name* | 着信トランザクションがキューに入れられて待機するピアノードの名前。 NULL の場合、すべてのピアノードの適用キューが使用されるまで待機します。 |

次のページに続く

表 92 – 前のページからの続き

| Parameter | Description |
|---|---|
| *target_lsn* | 待機するピアノードからのレプリケーションストリームの LSN。通常、ピアノードから `bdr.last_committed_lsn`を介して学習します。 |

### bdr.get_node_sub_receive_lsn

サブスクライバーでこのファンクションを使用して、特定のオリジンから受信した最後の LSN を取得できます。適用されるトランザクションに関連する LSN インクリメントのみを考慮して、フィルタリングしないか、フィルタリングすることができます。

このファンクションの出力と `bdr.get_node_sub_apply_lsn()` の出力の差は、対応する適用キューのサイズを測定します。

#### 概要

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_name* | LSN が取得されるレプリケーションストリームのソースであるノードの名前。 |
| *committed* | デフォルト true では、このファンクションは、最後の LSN 全体ではなく、受信したトランザクションのコミットのみを考慮します。これには、サブスクライバノードに影響を与えないアクションが含まれます。 |

### bdr.get_node_sub_apply_lsn

サブスクライバーでこのファンクションを使用して、特定のオリジンから受信および適用された最後の LSN を取得できます。

#### 概要

```
bdr.get_node_sub_apply_lsn(node_name name)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_name* | LSN が取得されるレプリケーションストリームのソースであるノードの名前。 |

### bdr.replicate_ddl_command

ノードのグループに DDL コマンドを複製する機能。

#### 概要

```
bdr.replicate_ddl_command(ddl_cmd text,
                          replication_sets text[],
                          ddl_locking text,
                          execute_locally bool)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *ddl_cmd* | 実行する DDL コマンド。 |
| *replication_sets* | `ddlcommand`を適用するレプリケーションセット名の配列。 NULL の場合、またはファンクションに `ddlcommand`のみが渡される場合、このパラメーターはアクティブな PGD グループのデフォルトレプリケーションセットに設定されます。 |
| *ddl_locking* | レプリケート中に *pg_upgrade* 値を設定する文字列。 `replicate_ddl_command`を実行しているローカルシステム上の `bdr.ddl_locking`の GUC 値のデフォルト。 |
| *execute_locally* | DDL コマンドをローカルで実行するかどうかを決定するブール値。デフォルト true。 |

#### 注

このファンクションの唯一の必須パラメーターは ddl_cmd です。

bdr.replicate_ddl_command() は常にコマンドをレプリケートし、*bdr.ddl_replication* の設定の影響を受けません。

### bdr.run_on_all_nodes

すべてのノードでクエリを実行するファンクション。

> **警告**
>
> このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザーの権限を使用して、リモートノードで任意のクエリーを実行します。この機能に特権を付与するときは注意してください。

### 概要

```
bdr.run_on_all_nodes(query text)
```

### パラメーター

| Parameter | Description |
|---|---|
| *query* | 実行する任意のクエリー。 |

### 注

このファンクションは、他のノードに接続してクエリーを実行し、各ノードから JSON 形式で結果を返します。複数の行が各ノードから返され、JSON 配列としてエンコードされる場合があります。ノードがダウンしているため接続できないなどのエラーは、応答フィールドに表示されます。明示的な statement_timeout またはその他のランタイムパラメーターが設定されていないため、デフォルトが使用されます。

このファンクションは、通常のレプリケーションを経由しません。すべての既知のノードへの直接クライアント接続を使用します。デフォルトでは、コマンドはクラスター内のすべてのノードに既に送信されているため、`bdr.ddl_replication = off` で接続が作成されます。

PGD 6 以降では、このファンクションは接続で `bdr.xact_replication=off` も設定して、コマンドが別のノードで実行された場合にのみトランザクションがローカルで実行されるようにします。

レプリケーションが中断され、ノード間で不一致が発生する危険があるため、この機能を使用する場合は注意してください。透過的 DDL レプリケーションまたは `bdr.replicate_ddl_command()` を使用して DDL をレプリケートします。DDL は、将来のリリースでブロックされる可能性があります。

### 例

例、次のクエリのように、モニタリングでこのファンクションを使用すると便利です。

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
      FROM bdr.node_slots
     WHERE origin_name IS NOT NULL
$$);
```

このクエリは、2 ノードクラスターで次のような結果を返します。

```
[
    {
        "dsn": "host=node1 port=5432 dbname=pgddb user=postgres ",
        "node_id": "2232128708",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
```

(次のページに続く)

(前のページからの続き)


```
                    "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

指定されたノードのリストでクエリを実行するファンクション。

> **警告**
>
> このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザーの権限を使用して、リモートノードで任意のクエリーを実行します。この機能に特権を付与するときは注意してください。

### 概要

```
bdr.run_on_nodes(node_names text[], query text)
```

### パラメーター

| Parameter | Description |
|---|---|
| *node_names* | クエリが実行されるノード名のテキスト ARRAY。 |
| *query* | 実行する任意のクエリー。 |

### 注

このファンクションは、他のノードに接続してクエリーを実行し、各ノードから JSON 形式で結果を返します。複数の行を各ノードから返すことができ、JSON 配列としてエンコードされます。ノードがダウンしているため接続できないなどのエラーは、応答フィールドに表示されます。明示的な statement_timeout またはその他のランタイムパラメーターが設定されていないため、デフォルトが使用されます。

このファンクションは、通常のレプリケーションを経由しません。すべての既知のノードへの直接クライアント接続を使用します。デフォルトでは、接続は `bdr.ddl_replication = off` で作成され、同じレプリケートされた DDL コマンドが複数のノードに送信される場合のレプリケーションの問題を回避します。

PGD 6 以降では、このファンクションは接続で `bdr.xact_replication=off` も設定して、コマンドが別のノードで実行された場合にのみトランザクションがローカルで実行されるようにします。

レプリケーションが中断され、ノード間で不一致が発生する危険があるため、この機能を使用する場合は注意してください。グローバルスキーマ変更の場合、DDL をレプリケートするには、透過的 DDL レプリケーションまたは `bdr.replicate_ddl_command()` のいずれかを使用します。

## bdr.run_on_group

ノードのグループでクエリを実行する機能。

> **警告**
>
> このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザーの権限を使用して、リモートノードで任意のクエリーを実行します。この機能に特権を付与するときは注意してください。

### 概要

```
bdr.run_on_group(node_group_name text, query text)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_group_name* | クエリが実行されるノードグループの名前。 |
| *query* | 実行する任意のクエリー。 |

#### 注

このファンクションは、他のノードに接続してクエリーを実行し、各ノードから JSON 形式で結果を返します。複数の行を各ノードから返すことができ、JSON 配列としてエンコードされます。ノードがダウンしているため接続できないなどのエラーは、応答フィールドに表示されます。明示的な statement_timeout またはその他のランタイムパラメーターが設定されていないため、デフォルトが使用されます。

このファンクションは、通常のレプリケーションを経由しません。すべての既知のノードへの直接クライアント接続を使用します。デフォルトでは、接続は `bdr.ddl_replication = off` で作成され、同じレプリケートされた DDL コマンドが複数のノードに送信される場合のレプリケーションの問題を回避します。

PGD 6 以降では、このファンクションは接続で `bdr.xact_replication=off` も設定して、コマンドが別のノードで実行された場合にのみトランザクションがローカルで実行されるようにします。

レプリケーションが中断され、グループ内のノード間で不一致が発生する危険があるため、この機能を使用する場合は注意してください。グローバルスキーマ変更の場合、DDL をレプリケートするには、透過的 DDL レプリケーションまたは `bdr.replicate_ddl_command()` のいずれかを使用します。

### bdr.global_lock_table

このファンクションは、指定されたテーブルのグローバル DML ロックを取得します。グローバル DML ロックの詳細については、DDL locking details を参照してください。

#### 概要

```
bdr.global_lock_table(relation regclass)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *relation* | ロックへの関係の名前または oid。 |

#### 注

このファンクションは、`ddl_locking` 設定とは無関係にグローバル DML ロックを取得します。

`bdr.global_lock_table` ファンクションには、`bdr.backwards_compatibility` が 30618 以下に設定されていない限り、ロックされた `relation` に対する `UPDATE`、`DELETE`、または `TRUNCATE` 特権が必要です。

### bdr.wait_for_xid_progress

このファンクションを使用して、特定のノードノード ID で識別される特定のトランザクション XID で識別されるが、クラスターで十分な進行を行うのを待機できます。進行状況は、ノードに適用されているトランザクションとして定義され、このノードは、トランザクションが適用される前に行われた他のすべてのレプリケーション変更を確認しています。

#### 概要

```
bdr.wait_for_xid_progress(origin_node_id oid, origin_topxid int4, allnodes boolean␣
↪DEFAULT true)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *origin_node_id* | トランザクションが発生したノードのノード ID。 |
| *origin_topxid* | トランザクションの XID。 |
| *allnodes* | `true`の場合、すべてのノードでトランザクションが進行するまで待機します。それ以外の場合、現在のノードのみを待機します。 |

#### 注

現在、これらのトランザクションのみが追跡されるため、DDL コマンドをレプリケートしたトランザクションにのみファンクションを使用できます。誤った `origin_node_id` または `origin_topxid` が指定された場合、ファンクションは永久にまたは `statement_timeout` が発生するまで待機する場合があります。

### bdr.local_group_slot_name

ローカルノードのグループスロットの名前を返します。

#### 例

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

### bdr.node_group_type

指定されたノードグループのタイプを返します。戻り値は、ノードグループの作成時に `bdr.create_node_group()` に渡されたものと同じですが、グループの作成時に `node_group_type` が NULL として渡された場合、 `global` が返されます。

#### 例

```
pgddb=# SELECT bdr.node_group_type(bdrgroup);
 node_group_type
- ----------------
 global
```

### bdr.alter_node_kind

PGD5 は、タスクマネージャーリーダーノードの概念を導入しました。ノードは PGD によって選択されますが、アップグレードされたクラスターの場合、クラスター内のすべてのノードで node_kind を適切に設定することが重要です。これは、最新の PGD バージョンにアップグレードした後、ノードごとに bdr.alter_node_kind() SQL ファンクションを呼び出すことにより手動で実行します。

#### 概要

```
bdr.alter_node_kind(node_name text,
                    node_kind text);
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_name* | 種類を変更するノードの名前。 |
| *node_kind* | ノードの種類。 |

### bdr.alter_subscription_skip_changes_upto

論理レプリケーションはバージョンをまたがってレプリケートでき、ロールのようなグローバル変更をレプリケートせず、選択的にレプリケートできるため、論理レプリケーション適用プロセスでエラーが発生し、変更の適用を停止する場合があります。

可能な場合は、ターゲット側に変更を加えてこのような問題を修正します。CREATE レプリケーションをブロックしている欠落テーブル、CREATE 必要なロール、GRANT 必要な権限など。ただし、その方法では問題を修正できない場合があり、トランザクションを完全にスキップする必要がある場合があります。変更はトランザクション全体としてスキップされます、オール・オア・ナッシング。スキップする場所を決定するには、次の例のように、ログ出力を使用してコミット LSN を見つけるか、論理デコード機能を使用して変更ストリームをピークにします。

トランザクションが 1 つの変更のみを行った場合を除き、多くの場合、ターゲット側でトランザクションのエフェクトを手動で適用する必要があります。そのため、以下の例に示すように、可能な限り問題のあるトランザクションを保存することが重要です。

pg_catalog.pg_logical_slot_get_binary_changes を使用して目的の LSN にスキップすることにより、bdr.alter_subscription_skip_changes_upto なしで変更をスキップすることが可能であるため、これは便利なファンクションです。論理デコードにおけるある種のエラーをバイパスする場合がありますが、より速いスキップを行います。

この機能は、無効になっているサブスクリプションでのみ動作します。

通常の手順のシーケンスは次のとおりです。

1. 問題のあるサブスクリプションと問題のあるコミットの LSN を特定します。
2. サブスクリプションを無効にします。
3. 可能であれば、ソースノードで pg_catalog.pg_logical_slot_peek_changes を使用してトランザクションのコピーを保存します。
4. ターゲットノード上の bdr.alter_subscription_skip_changes_upto 。
5. 必要に応じて、修復または同等の変更をターゲットに手動で適用します。
6. サブスクリプションを再度有効にします。

> **警告**
>
> このファンクションを使用すると、問題を悪化させるのは簡単です。それが唯一のオプションであると確信できない限り、何もしないでください。

### 概要

```
bdr.alter_subscription_skip_changes_upto(
  subname text,
  skip_upto_and_including pg_lsn
);
```

### 例

トランザクションの適用はエラーで失敗し、ターゲット側の変更などのインパクトの低い修正ではこの問題を解決できないと判断しました。トランザクションをスキップする必要があると判断します。

エラーログで、次の例のように、スキップするコミットレコード LSN を見つけます。

```
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)
```

ログのこの部分には、必要な情報があります。the_target_lsn: **0/300AC18** the_subscription: **bdr_regression_bdrgroup_node1_node2**

次に、サブスクリプションを無効にして、適用ワーカーがレプリケーションスロットに接続しようとしないようにします。

```
SELECT bdr.alter_subscription_disable(the_subscription);
```

トランザクションの一部だけをスキップすることはできません。それはオール・オア・ナッシングです。したがって、サブスクリプションのスロット名を使用して、最初にプロバイダー側でコピーして、レコードを保存することを強くお勧めします。

```
\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);
```

この例は、読みやすいように複数行に分割されていますが、単一行で発行します。\copy は、複数行のコマンドをサポートしていません。

peek を get に変更することにより、変更をスキップできますが、bdr....skip_changes_upto は、すべてのデータのデコードと出力を回避する高速なスキップを行います。

```
SELECT bdr.alter_subscription_skip_changes_upto(subscription_name,
    the_target_lsn);
```

ダンプされたトランザクションの内容をガイドとして使用して、同じ変更またはその修復バージョンをターゲットノードに手動で適用できます。

最後に、サブスクリプションを再度有効にします。

```
SELECT bdr.alter_subscription_enable(the_subscription);
```

### グローバルアドバイザリーロック

PGD は、グローバルアドバイザリーロックをサポートしています。これらのロックは、PGD でサポートされている勧告ロックがグローバルであることを除き、PostgreSQL で使用可能な勧告ロックと似ています。これらは DDL ロックと同様のセマンティクスに従います。したがって、アドバイザリロックはマジョリティコンセンサスによって取得され、すべてのノードの大部分が連携できる限り、1 つ以上のノードがダウンまたは遅延している場合でも使用できます。

現在、EXCLUSIVE ロックのみがサポートされています。したがって、別のノードまたは同じノード上の別のバックエンドがオブジェクトの勧告ロックを既に取得している場合、他のノードまたはバックエンドはロックが解放されるまで待機する必要があります。

アドバイザリーロックはトランザクション的な性質を持っています。したがって、トランザクションが終了する前に明示的に解放しない限り、トランザクションの終了時にロックは解放されます。この場合、リリースされるとすぐに利用可能になります。セッションレベルの勧告ロックは現在サポートされていません。

グローバル勧告ロックは再入可能です。したがって、同じリソースが 3 回ロックされている場合、3 回ロックを解除して、他のセッションで使用できるように解放する必要があります。

### bdr.global_advisory_lock

このファンクションは、提供されたオブジェクトの EXCLUSIVE ロックを取得します。ロックが利用できない場合、ロックが利用可能になるか、bdr.global_lock_timeout に到達するまで待機します。

#### 概要

```
bdr.global_advisory_lock(key bigint)
```

#### パラメーター

- key —アドバイザリロックが取得されるオブジェクト。

#### 概要

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *key1* | 複合キーの最初の部分。 |
| *key2* | 複合キーの 2 番目の部分。 |

### bdr.global_advisory_unlock

このファンクションは、アプリケーション定義のソースで以前に取得したロックを解放します。ロックは、アプリケーションによって同じトランザクションで取得されている必要があります。それ以外の場合、エラーが発生します。

#### 概要

```
bdr.global_advisory_unlock(key bigint)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *key* | アドバイザリロックが取得されるオブジェクト。 |

#### 概要

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *key1* | 複合キーの最初の部分。 |
| *key2* | 複合キーの 2 番目の部分。 |

### モニタリング機能

#### bdr.monitor_group_versions

クラスター全体のバージョンチェックを提供するために、このファンクションはビュー bdr.group_version_details から返された PGD バージョン情報を使用します。

#### 概要

```
bdr.monitor_group_versions()
```

#### 注

モニタリング パラレル 適用 で説明しているように、このファンクションは、フィールド status および message を持つレコードを結果ます。

このファンクションは bdr.run_on_all_nodes() を呼び出します。

#### bdr.monitor_group_raft

クラスター全体の Raft チェックを提供するために、このファンクションはビュー bdr.group_raft_details から返された PGD Raft 情報を使用します。

#### 概要

```
bdr.monitor_group_raft()
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_group_name* | 確認するノードグループ名。 |

#### 注

モニタリング パラレル 適用 で説明しているように、このファンクションは、フィールド status および message を持つレコードを結果ます。

このファンクションは bdr.run_on_all_nodes() を呼び出します。

### bdr.monitor_local_replslots

このファンクションは、ビュー pg_replication_slots スロットアクティブまたは非アクティブから返されたレプリケーションスロットステータス情報を使用して、PGD グループスロットを除くすべてのレプリケーションスロットを考慮したローカルチェックを提供します。

この機能は、optimized topology が有効になっているときに PGD クラスターでサブスクライバ専用グループリーダーとして動作しているサブスクライバ専用ノードのステータス情報も提供します。

#### 概要

```
bdr.monitor_local_replslots()
```

#### 注

このファンクションは、フィールド status および message を持つレコードを結果ます。

| 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 |

レプリケーションスロットの監視 で詳細な説明が得られます。

### bdr.wal_sender_stats

*Decoding worker* が有効になっている場合、このファンクションは、デコーダースロットと、各 WAL 送信者によって読み取られている現在の論理変更レコード LCR セグメントファイルに関する情報を表示します。

#### 概要

```
bdr.wal_sender_stats()
```

#### 出力列

| Column name | Description |
|---|---|
| *pid* | WAL 送信者の PID。 `pg_stat_replication`の `pid`列に対応します。 |
| *is_using_lcr* | WAL 送信者が LCR ファイルを送信しているかどうか。 |
| *decoder_slot_name* | デコーダーレプリケーションスロットの名前。 |
| *lcr_file_name* | 現在の LCR ファイルの名前。 |

### bdr.get_decoding_worker_stat

*Decoding worker* が有効になっている場合、このファンクションは、現在のデータベースに関連付けられているデコードワーカーの状態に関する情報を表示します。これは、ワーカーの進行状況のデコードに関するより詳細な情報も提供します `pg_replication_slots` を介して利用可能な情報。

#### 概要

```
bdr.get_decoding_worker_stat()
```

#### 出力列

| Column name | Description |
|---|---|
| *pid* | デコードワーカーの PID。 `pg_replication_slots`の `active_pid`列に対応します。 |
| *decoded_upto_lsn* | デコードワーカーがトランザクションログを読み取る LSN。 |
| *waiting* | デコードワーカーが新しい WAL を待機しているかどうか。 |
| *waiting_for_lsn* | 次に予想される WAL の LSN。 |

#### 注

詳細については、 *LCR を使用した WAL 送信者のモニタリング* を参照してください。

### bdr.lag_control

*Lag Control* が有効になっている場合、このファンクションは、ローカルノードと現在のデータベースの構成された遅延対策に適合するノードの数とコミット遅延に関する情報を表示します。

#### 概要

```
bdr.lag_control()
```

#### 出力列

| Column name | Description |
|---|---|
| *commit_scope_id* | コミットスコープの OID :ref:``bdr.commit_scopes` 列<*bdr.commit_scopes* 列>`を参照してください。 |
| *sessions* | 遅延制御エントリを参照するセッションの数。 |
| *current_commit_delay* | 現在のランタイムコミット遅延ミリ秒単位。 |
| *maximum_commit_delay* | 構成された最大コミット遅延ミリ秒単位。 |
| *commit_delay_adjust* | サンプルインターバル中に発生する可能性のあるランタイムコミット遅延の変更ミリ秒単位。 |

次のページに続く

表 109 – 前のページからの続き

| Column name | Description |
|---|---|
| *current_conforming_nodes* | 遅延対策に適合しているノードの現在の実行時数。 |
| *minimum_conforming_nodes* | 遅延対策に適合するために必要なノードの構成済みの最小数。これを下回ると、コミット遅延調整が適用されます。 |
| *lag_bytes_threshold* | コミット遅延が適用される遅延サイズキロバイト単位。 |
| *maximum_lag_bytes* | キロバイト単位の構成された最大遅延サイズ。 |
| *lag_time_threshold* | ミリ秒単位のコミット遅延が適用される遅延時間。 |
| *maximum_lag_time* | 設定された最大遅延時間ミリ秒単位。 |
| *sample_interval* | ミリ秒単位の遅延サンプル間の構成された最小時間と考えられるコミット遅延調整。 |

### ルーティング機能

#### bdr.routing_leadership_transfer

ルーティングリーダーを変更すると、ノードグループのリーダーシップが別のノードに転送されます。

#### 概要

```
bdr.routing_leadership_transfer(node_group_name text,
               leader_name text,
               transfer_method text DEFAULT strict,
               transfer_timeout interval DEFAULT 10s);
```

#### パラメーター

| Name | Type | Default | Description |
|---|---|---|---|
| *node_group_name* | text | | リーダーシップの譲渡が要求されたグループの名前。 |
| *leader_name* | text | | 書き込みリーダーになるノードの名前。 |
| *transfer_method* | text | *'strict'* | 転送の種類。 *'fast'* `または**デフォルトの**'strict'`で、最大遅延をチェックします。 |
| *transfer_timeout* | interval | '10s' | リーダーシップ転送のタイムアウト。デフォルトは 10 秒です。 |

### CAMO ファンクション

CAMO、クライアントがトランザクションの進行状況を追跡することにより、トランザクションのコミットに積極的に参加する必要があります。ここにリストされているファンクションはそのために使用され、*Commit At Most Once* で説明されています。

### bdr.is_camo_partner_connected

ペアモードで構成された CAMO パートナーノードの接続ステータスを確認できます。現在、熱心なレプリケーションで使用される CAMO に相当するものはありません。

#### 概要

```
bdr.is_camo_partner_connected()
```

#### 戻り値

CAMO パートナーが現在ローカルノード上の WAL 送信プロセスに接続しているため、トランザクションデータを受信し、確認を送り返すことができるかどうかを示すブール値。

### bdr.is_camo_partner_ready

ペアモードで構成された CAMO パートナーノードの準備状況を確認できます。その下で、これはローカルモードへの、またはローカルモードからのスイッチをトリガーします。

#### 概要

```
bdr.is_camo_partner_ready()
```

#### 戻り値

CAMO パートナーがタイムリーに、つまり `TO ASYNC` の `timeout` の有効期限が切れる前に、ローカルノードから発生したトランザクションを確認することが合理的に期待できるかどうかを示すブール値。

> **注釈**
>
> このファンクションは、過去または現在の状態を照会します。正の戻り値は、 CAMO パートナーが将来のトランザクションを確認できるかどうかを示しません。

### bdr.get_configured_camo_partner

この機能は、ローカルノードの CAMO パートナーペアモードで構成を表示します。

#### 概要

```
bdr.get_configured_camo_partner()
```

### bdr.wait_for_camo_partner_queue

このファンクションは、CAMO パートナーノードを照会することがデフォルトの `bdr.wait_for_apply_queue` のラッパーです。ローカルノードが CAMO ペアの一部でない場合、エラーを返します。

#### 概要

```
bdr.wait_for_camo_partner_queue()
```

### bdr.camo_transactions_resolved

このファンクションは、CAMO トランザクションが完全に解決されるまでの待機を開始します。

#### 概要

```
bdr.camo_transactions_resolved()
```

### bdr.logical_transaction_status

ノードに障害が発生したときにコミットされていたトランザクションのステータスを確認するには、アプリケーションがこのファンクションを使用し、トランザクションが発生したノードのノード ID と元ノードのトランザクション ID をパラメーターとして渡す必要があります。

#### 概要

```
bdr.logical_transaction_status(node_id OID, xid OID,
                               require_camo_partner boolean DEFAULT true)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_id* | トランザクションが発生する PGD ノードのノード ID。通常、 *pg_upgrade* `bdr.local_node_id`から `COMMIT`の前にクライアントによって取得されます。 |
| *xid* | オリジンノードのトランザクション ID。通常、*COMMIT* `から *:ref: `pg_upgrade <Monitoring through SQL>* `transaction_id`の前にクライアントによって取得されます。 |
| *require_camo_partner* | デフォルトで true で、構成チェックが有効になります。これらのチェックを無効にし、CAMO トランザクションではないトランザクションのステータスを照会するには、false に設定します。 |

#### 戻り値

ファンクションは、次のいずれかの結果を返します。

- `'committed'::TEXT` —トランザクションはコミットされ、CAMO ペアの両方のノードで可視され、最終的に他のすべての PGD ノードにレプリケートされます。クライアントが再試行する必要はありません。
- `'aborted'::TEXT` —トランザクションは中止され、他の PGD ノードにレプリケートされません。クライアントは再試行するか、トランザクションのコミットの失敗をエスカレートする必要があります。

- 'in progress'::TEXT —トランザクションはこのローカルノードでまだ進行中であり、まだコミットまたは中止されていません。トランザクションは COMMIT フェーズにあり、CAMO パートナーがコミットを確認または拒否するのを待機している可能性があります。推奨されるクライアントの反応は、元のノードから切断し、CAMO パートナーに再接続して、代わりにそれを照会することです。間にロードバランサーまたはプロキシがあり、クライアントがどのノードを照会するかを制御できない場合、クライアントはステータスが'committed' または'aborted' に切り替わるまで、繰り返しポーリングのみできます。

熱心な全ノードレプリケーションの場合、ピアノードはまだコミットまたは中止されていないトランザクションに対してこの結果を生成します。この場合、まだレプリケートされていないトランザクション、またはオリジンノードで開始されていないトランザクションでも、ピア PGD ノードで in progress 結果を生成する場合があります。ただし、クライアントは、オリジンでコミットする前にトランザクションステータスを照会してはなりません。

- 'unknown'::TEXT —指定されたトランザクションは、将来であるか、その特定のノードにまだレプリケートされていない、または過去すぎるため不明です。このようなトランザクションのステータスはまだ知られていない、または不明です。この戻り値は、クライアントによる不適切な使用の兆候です。

クライアントは、エラーが発生した場合にファンクション呼び出しを再試行する準備をする必要があります。

### コミットスコープファンクション

#### bdr.add_commit_scope

**非推奨**。代わりに bdr.create_commit_scope を使用してください。以前は、この機能を使用してコミットスコープをノードグループに追加しました。現在は非推奨であり、将来のリリースで削除されるまで警告を発し、その時点でエラーが発生します。

#### bdr.create_commit_scope

bdr.create_commit_scope は、指定されたコミットスコープ名とオリジンノードグループのルールを作成します。ルールが EDB Postgres 分散クラスター内のすべてのノードで同じである場合、トップレベルのノードグループに対してこのファンクションを 1 回呼び出すだけで、コミットスコープを完全に定義できます。

または、同じ commit_scope_name ではあるが、トランザクションのオリジンによって異なるオリジンノードグループとコミットスコープのルールを使用して、複数回呼び出すこともできます。

#### 概要

```
bdr.create_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT,
    wait_for_ready boolean DEFAULT true)
```

#### 注

bdr.create_commit_scope は、非推奨の bdr.add_commit_scope ファンクションを置き換えます。add_commit_scope とは異なり、同じ名前が使用される場合、既存のコミットスコープをサイレントに上書きしません。代わりに、エラーが報告されます。

### bdr.alter_commit_scope

bdr.alter_commit_scope を使用すると、コミットスコープ内のシングルオリジンノードグループの特定のルールを変更できます。

#### 概要

```
bdr.alter_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT)
```

### bdr.drop_commit_scope

コミットスコープ内の単一のルールを削除します。コミットスコープに複数のルールを定義する場合、ルールごとに 1 回このファンクションを呼び出して、コミットスコープ全体を完全に削除する必要があります。

#### 概要

```
bdr.drop_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME)
```

**注釈**

ノードグループによってデフォルトとして使用されているコミットスコープを削除することはできません。

### bdr.remove_commit_scope

**非推奨**。代わりに bdr.drop_commit_scope を使用してください。以前は、このファンクションはノードグループからコミットスコープを削除するために使用されていました。現在は非推奨であり、将来のリリースで削除されるまで警告を発し、その時点でエラーが発生します。

## PGD settings

PGD 固有の構成設定を設定できます。特に断りのない限り、値はいつでも設定できます。

#### 競合処理

##### bdr.default_conflict_detection

新しく作成されるテーブルのデフォルトの競合検出方法を設定します。*bdr.alter_table_conflict_detection()* と同じ値を受け入れます。

#### グローバルシーケンスパラメーター

##### bdr.default_sequence_kind

デフォルトの *Sequences* を設定します。

デフォルトは `distributed` で、これは、 `snowflakeid` が `int8` シーケンスつまり `bigserial` に使用され、`int4` つまり `serial` および `int2` シーケンスの `galloc` シーケンスに使用されることを意味します。

#### DDL 処理

##### bdr.default_replica_identity

新しく作成されるテーブルに `REPLICA IDENTITY` のデフォルト値を設定します。 `REPLICA IDENTITY` は、更新または削除された行を識別するためにログ先行書き込みに書き込まれる情報を定義します。

受け入れられる値は次のとおりです。

| Value | Description |
|---|---|
| *default* | 主キーの列の古い値がある場合は、それを記録しますこれはデフォルトの PostgreSQL 動作です。 |
| *full* | 行内のすべての列の古い値を記録します。 |
| *nothing* | 古い行に関する情報は記録しません。 |
| *auto* | テーブルは REPLICA IDENTITY FULL で作成されます。これはデフォルトの PGD 動作です。 |

詳細は、 *PostgreSQL documentation* を参照してください。

PGD は、 `PRIMARY KEY` または `UNIQUE` 制約のないテーブルで `UPDATE` および `DELETE` オペレーションをレプリケートできません。例外は、テーブルのレプリカ ID がテーブル固有の構成または `bdr.default_replica_identity` による `FULL` の場合です。

`bdr.default_replica_identity` が `default` で、テーブル定義に `UNIQUE` 制約が含まれている場合、`REPLICA IDENTITY` として自動的に選択されません。 `ALTER TABLE ... REPLICA IDENTITY ...` を使用して `REPLICA IDENTITY` を明示的に設定する必要があります。

テーブルのレプリカ ID を `full` に設定すると、テーブルのワイヤに書き込まれる WAL の量とレプリケートされるデータの量が増加します。

```
ALTER SYSTEM SET bdr.default_replica_identity="default";
```

デフォルトの引用符なしは、構成ファイルから設定の削除をトリガーする *ALTER SYSTEM* コマンドの特別な値であるため、引用符を含める必要があります。設定が削除されると、システムは PGD のデフォルト設

定である auto を使用します。

ノード間で DDL を自動的にレプリケートしますデフォルトは on です。

このパラメーターは、bdr_superuser または superuser ロールによってのみ設定できます。

DDL を実行するか、 bdr.ddl_replication = off を使用して PGD 管理機能を呼び出すと、管理者が介入できるまでレプリケーションが停止する状況が発生する場合があります。詳細は、ステートメント固有の *DDL* レプリケーションの問題 を参照してください。

bdr.ddl_replication が off に設定されている場合は常に、 LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、キャプチャされた DDL コマンドまたは PGD レプリケーションファンクションのレプリケーションがこの設定によりスキップされるたびに、WARNING-level メッセージが書き込まれます。

### bdr.role_replication

ノード全体で ROLE コマンドを自動的にレプリケートしますデフォルトは on です。スーパーユーザーのみがこのパラメーターを設定できます。この設定は、bdr.ddl_replication もオンになっている場合にのみ機能します。

外部方法を使用せずにこのパラメーターをオフにして、すべてのノードでロールが同期していることを確認すると、管理者が介入するまでレプリケート DDL がレプリケーションを中断する場合があります。

Role manipulation statements を参照してください

詳細については。

### bdr.ddl_locking

DDL のグローバルロックの操作モードを構成します。

このパラメーターは、bdr_superuser または superuser ロールによってのみ設定できます。

考えられるオプションは次のとおりです。

| Value | Description |
|---|---|
| *all* | すべての DDL 操作にグローバルロックを使用します。デフォルト |
| *leader* | リーダーベースのグローバル DML ロックを使用します。 |
| *auto* | 現在、`leader`とシノニムです。 |
| *dml* | リレーションのグローバル DML ロックを取得することにより書き込みを防止する必要がある DDL 操作にのみグローバルロックを使用します。 |
| *off* | DDL 操作にはグローバルロックを使用しないでください。 |

デフォルトは auto です。

bdr.ddl_replication が off に設定されている場合は常に、 LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定によりグローバルロックステップがスキップされるたびに、WARNING メッセージが書き込まれます。一部のステートメントでは、2 つの WARNING メッセージが発生するのが通常です。1 つは DML ロックをスキップし、もう 1 つは DDL ロックをスキップします。

下位互換性のため、bdr.ddl_locking はエイリアスをサポートしています。 on および true は、all のエイリアスです。 false は、off のエイリアスです。

グローバルロック も参照してください。

### bdr.truncate_locking

TRUNCATE コマンドのロック動作を設定しますデフォルトは on/true です。 on/true の場合、TRUNCATE は bdr.ddl_locking 設定に従います。

## グローバルロック

DDL ロックは、*bdr.ddl_locking* によって制御されます。その他のグローバルロック設定には次のものが含まれます。

### bdr.global_lock_max_locks

ノードで保持できるグローバルロックの最大数を設定しますデフォルトは 1000 です。Postgres サーバー起動時にのみ設定できます。

### bdr.global_lock_timeout

グローバルロックの待機の最大許可時間を設定しますデフォルトは 1 分です。値 0 はこのタイムアウトを無効にします。

### bdr.global_lock_statement_timeout

グローバルロックを保持するステートメントの最大許可期間を設定しますデフォルトは 60 分です。値 0 はこのタイムアウトを無効にします。

### bdr.global_lock_idle_timeout

グローバルロックを保持するトランザクションのアイドル時間の最大許可期間を設定しますデフォルトは 10 分です。値 0 はこのタイムアウトを無効にします。

### bdr.lock_table_locking

LOCK TABLE ステートメントのロック動作を設定しますデフォルトはオンです。LOCK TABLE ステートメントは有効にすると、クラスターでグローバル DML ロックを取得し、他のロックステートメントをブロックします。

| Value | Description |
|---|---|
| *on* | すべてのテーブルロックにグローバルロックを使用します。 デフォルト |
| *off* | テーブルロックにグローバルロックを使用しないでください。 |

#### bdr.predictive_checks

予測チェックのログレベルを設定します現在グローバルロックによってのみ使用されます。DEBUG、LOG、WARNING デフォルト、または ERROR です。予測チェックは、特定の操作を実行するときに予想されるクラスター状態の早期検証です。タイムアウトを待機するのではなく、早期フェイルのこれらの操作に使用できます。グローバルロック用語では、PGD は、接続されている十分なノードがあり、グローバルロックに必要なクォーラムを取得するための妥当な遅延制限内であることをチェックします。

### ノード管理

#### bdr.replay_progress_frequency

レプリケーション位置情報をクラスターの残りの部分に送信する間隔を設定しますデフォルトは 1 分です。

### 汎用レプリケーション

#### bdr.writers_per_subscription

サブスクリプションごとのライターのデフォルト数を設定します。PGD では、グループの bdr.alter_node_group_option を使用してこれを変更することもできます。

#### bdr.max_writers_per_subscription

サブスクリプションごとのライターの最大数 bdr.writers_per_subscription 設定の上限を設定します。

#### bdr.xact_replication

現在のトランザクションをレプリケートしますデフォルトは on です。

これをオフにすると、トランザクション全体がローカルのみになります。つまり、トランザクションは、PGD による論理デコードおよび論理デコードのその他のすべてのダウンストリームターゲットから可視されません。データは、ロジカルスタンバイノードを含む他のノードに転送されません。

このパラメーターは、bdr_superuser または superuser ロールによってのみ設定できます。

このパラメーターは、 bdr.permit_unsafe_commands = on を除き SET LOCAL コマンドを使用して現在のトランザクション内でのみ設定できます。

> **注釈**
>
> トランザクションレプリケーションを無効にしても WAL は生成されますが、これらの変更はオリジンでフィルタリングされます。

> **警告**
>
> *bdr.xact_replication* をオフにすると、ノード間でデータの不一貫性が発生します。ノード間のデータの相違からリカバリーするため、またはレプリケーションを続行するために単一ノードの変更が必要なレ

プリケーション状況にのみ使用します。自分の責任で使用してください。

### bdr.permit_unsafe_commands

一般的な使用には安全でないと考えられるコマンドの安全性チェックをオーバーライドします。

bdr_superuser または PostgreSQL スーパーユーザーが必要です。

**警告**

通常安全だと考えられていないコマンドは、一貫性のない結果を生成するか、レプリケーションを完全に中断する可能性があります。自分の責任で使用してください。

### bdr.batch_inserts

1 つのテーブルの挿入のバッチ処理をオンにする単一のトランザクションでの 1 つのテーブルへの連続した挿入の数。

この設定では、大規模なデータのレプリケーションを、挿入のセットとしてではなく、内部の COPY としてロードできます。これは、ノード参加中の初期データのコピー方法でもあります。

### bdr.maximum_lock_skew

`bdr.maximum_clock_skew_action` をトリガーする前に、受信トランザクションコミットタイムスタンプとサブスクライバーの現在時間の最大差を指定します。

現在再生されているトランザクションのタイムスタンプが、サブスクライバーの現在時刻と比較して将来的であるかどうかを確認します。その場合、その差が `bdr.maximum_clock_skew` より大きい場合、`bdr.maximum_clock_skew_action` 設定で指定されたアクションを実行します。

デフォルトは-1 で、これはクロックスキューを無視するチェックがオフになることを意味します。すべてのサーバーの時計が同期している場合、0 を設定するのが有効です。トランザクションが再生されているという事実は、過去にコミットされたことを意味します。

### bdr.maximum_lock_skew_action

`bdr.maximum_clock_skew` より高いクロックスキューが検出された場合に実行するアクションを指定します。

この設定には 2 つの値があります。

| Value | Description |
|---|---|
| *WARN* | このファクトに関する警告をログに記録します。警告は、サーバーログの氾濫を防ぐために、最大 1 分に 1 回ログに記録されます。 |
| *WAIT* | 現在のローカルタイムスタンプがリモートコミットタイムスタンプから `bdr.maximum_clock_skew`を引いたものより古くなくなるまで待機します。 |

### bdr.accept_connections

PGD への接続を有効または無効にしますデフォルトは on です。

bdr_superuser または PostgreSQL スーパーユーザーが必要です。

### bdr.writer_input_queue_size

レシーバーがライタプロセスにデータを送信するために使用する共有メモリキューのサイズを指定します。ライタプロセスが停止しているか、進行が遅い場合、キューがいっぱいになり、受信側プロセスも停止する可能性があります。したがって、このキューに十分な共有メモリを提供することが重要です。デフォルトは 1MB で、最大許可サイズは 1GB です。GUC の設定には任意のストレージサイズ指定子を使用できますが、デフォルトは KB です。

### bdr.writer_output_queue_size

レシーバーがライタプロセスからデータを受信するために使用する共有メモリキューのサイズを指定します。ライタは大量のデータを送信することは想定されていないため、比較的小さいサイズのキューで十分です。デフォルトは 32KB で、最大許可サイズは 1MB です。GUC の設定には任意のストレージサイズ指定子を使用できますが、デフォルトは KB です。

### bdr.min_worker_backoff_migrate

特定のワーカーが bdr.min_worker_backoff_delay ミリ秒ごとよりも頻繁に再起動されないようにすることにより、PGD バックグラウンドワーカー起動のレート制限が可能です。エラーが繰り返されると、バックオフが指数関数的に増加し、最大 bdr.max_worker_backoff_delay までジッターが追加されます。

時間単位のサフィックスがサポートされています。

> **注釈**
>
> この設定は現在レシーバーワーカーにのみ影響します。つまり、主にエラーまたは接続障害が発生したときにサブスクリプションが再接続を試行する速度に影響を与えます。

bdr.min_worker_backoff_delay のデフォルトは 1 秒です。bdr.max_worker_backoff_delay の場合、1 分です。

バックオフ遅延設定が変更され、PostgreSQL 構成がリロードされると、現在のすべてのバックオフがリセットを待機します。さらに、bdr.worker_task_reset_backoff_all() ファンクションは、管理者がすべてのバックオフインターバルを強制的にすぐに有効期限が切れるように提供されています。

共有メモリ内の追跡テーブルは、各種類のワーカーの最終起動時刻を記憶するために維持されます。この追跡テーブルは永続的ではありません。これは、汚れたバックエンド終了後のクラッシュリカバリー中のソフトリスタートを含む、PostgreSQL の再起動によってクリアされます。

ビュー *bdr.worker_tasks* を使用してこの状態を検査できるため、管理者は、現在有効になっているバックオフレート制限を確認できます。

レート制限の目的で、ワーカーはタスクによって分類されます。このキーは、ワーカーロール、データベース OID、サブスクリプション ID、サブスクリプションライタ ID、拡張ライブラリ名とファンクション名、

拡張機能提供のワーカー名、および同期ライターのリモートリレーション ID で構成されています。NULL は、特定の分類子が適用されない場合、たとえば、マネージャーワーカーがサブスクリプション ID がなく、レシーバーがライタ ID がない場合に使用されます。

### CRDT

#### bdr.crdt_raw_value

CRDT data types の出力フォーマットを設定します。

デフォルト出力この設定が off の場合、ベース CRDT タイプの現在値のみを結果ことができます、たとえば、crdt_pncounter の bigint。 on に設定されている場合、戻り値は CRDT 値の完全な表現を表します。これには、たとえば、マルチプルのノードからの状態を含めることができます。

### コミットスコープ

#### bdr.commit_scope

現在またはデフォルトの commit scope を設定しますデフォルトは空の文字列です。

### Commit At Most Once

#### bdr.camo_local_mode_deploy

CAMO の非同期モードで適用され、CAMO パートナーがトランザクションを確認する必要があるときに通常発生するオーバーヘッドをエミュレートするコミット遅延デフォルトは 5 ミリ秒です。この機能を無効にするには、0 に設定します。

#### bdr.camo_enable_client_warnings

CAMO プロパティを保証できないデータベースでアクティビティが実行された場合、警告を発しますデフォルトが有効になっています。十分な情報のあるユーザーは、この設定を無効にして、ログに記録される警告の量を減らすことができます。

### トランザクションストリーミング

#### bdr.default_streaming_mode

サブスクライバノードによるトランザクションストリーミングを制御します。使用可能な値は off 、writer 、file 、および auto です。デフォルト auto 。 off に設定されている場合、サブスクライバーはトランザクションストリーミングを要求しません。他の値に設定されている場合、サブスクライバーはトランザクションストリーミングを要求し、パブリッシャーはそれをサポートしている場合、およびグループレベルで構成されている場合にそれを提供します。詳細については、トランザクションストリーミングでの使用 を参照してください。

### 遅延制御

#### bdr.lag_control_max_commit_deploy

ミリ秒単位での許容できる最大許容コミット後遅延。

#### bdr.lag_control_max_lag_size

許容できる最大許容遅延サイズキロバイト単位。

#### bdr.lag_control_max_lag_time

ミリ秒単位の許容できる最大許容遅延時間。

#### bdr.lag_control_min_conforming_nodes

許容可能な遅延測定値を下回るように必要なノードの最小数。

#### bdr.lag_control_commit_deploy_adjust

最大コミット遅延時間の一部として測定されるコミット遅延のマイクロ調整。デフォルト値の 0.01%では、最大コミット遅延に到達するまでに 100 ネットインクリメントが必要です。

#### bdr.lag_control_sample_interval

ミリ秒単位の遅延サンプルとコミット遅延マイクロ調整間の最小時間。

#### bdr.lag_control_commit_deploy_start

コミット遅延インクリメントが適用され始める遅延しきい値。許容可能な遅延尺度の割合で表されます。デフォルト値の 1.0%では、許容可能な遅延測定値に違反するまでコミット遅延のインクリメントは開始されません。

より小さい分数を設定することにより、許容可能な遅延尺度に漸近するように早期に「遅延曲線を曲げる」ことにより、違反を防止できる場合があります。

### タイムスタンプベースのスナップショット

#### bdr.timestamp_snapshot_keep

タイムスタンプベースのスナップショット使用の有効なスナップショットを保持する時間デフォルトは `0`、過去のスナップショットを保持しないことを意味します。

### モニタリングとロギング

#### bdr.debug_level

PGD がデバッグメッセージを書き込むために使用するログレベルを定義します。デフォルト値は `debug2` です。詳細な PGD デバッグ出力を表示する場合は、`bdr.debug_level = 'log'` を設定します。

#### bdr.trace_level

`bdr.debug_level` と同様に、PGD トレースメッセージに使用するログレベルを定義します。 EDB Postgres 分散クラスターのすべてのノードでトレースを有効にすると、EDB サポートが問題を診断するのに役立つ場合があります。このパラメーターは、Postgres サーバーの起動時にのみ設定できます。

**警告**

*bdr.debug_level* または`bdr.trace_level` を値>= *log_min_messages* に設定すると、非常に大量のログ出力が生成される場合があります。ディスク領域の枯渇を防ぐためにログのフィルタリング、アーカイブ、およびローテーションの計画が策定されていない限り、運用環境では長期的に有効にしないでください。

### bdr.track_subscription_apply

トラックは、 bdr.stat_subscription デフォルトは on を使用した各サブスクリプションの統計を適用します。

### bdr.track_relation_apply

トラックは、 bdr.stat_relation デフォルトは `off` とのリレーションごとに統計を適用します。

### bdr.track_apply_lock_timing

bdr.stat_relation デフォルトは `off` とのリレーションの統計を追跡するときにロックタイミングを追跡します。

## デコードワーカー

### bdr.enable_wal_decoder

*Decoding worker* デフォルトは false を使用して、単一ノードで論理変更レコード LCR 送信を有効にします。true に設定すると、デコードワーカープロセスが開始され、WAL 送信側は生成した LCR を送信します。false に設定し直すと、LCR を使用する WAL 送信者が再起動され、WAL を直接使用します。

**注釈**

PGD グループ内のすべてのノードでこの設定を有効にし、グループで`enable_wal_decoder` オプションを true に設定する必要があります。

### bdr.receive_lcr

別のノードをサブスクライブする場合、この設定により、ノードはサブスクリプションの論理変更レコード LCR の使用を要求できますデフォルトは false です。下流ノードでこの設定が true の場合、ノードは上流ノードに送信するときに LCR を使用することを要求します。ノードで `bdr.enable_wal_decoder` を true に設定する場合、この設定も true に設定します。

**注釈**

PGD グループ内のすべてのノードでこの設定を有効にし、グループで`enable_wal_decoder` オプションを true に設定する必要があります。

### bdr.lcr_cleanup_interval

論理変更レコード LCR ファイルのクリーンアップ間隔デフォルト 3 分。*Decoding worker* が有効になっている場合、デコードワーカーは LCR ファイルをバッファとして保存します。これらのファイルは定期的にクリーンアップされ、この設定は 2 つの連続するクリーンアップ間の間隔を制御します。ゼロに設定すると、クリーンアップが無効になります。

### 接続設定

以下は、すべてのクロスノード `libpq` 接続に影響を与える接続設定のセットです。デフォルトはかなり保守的な値に設定されており、ほとんどの運用ニーズをカバーします。すべての変数には `SIGHUP` コンテキストがあります。これは、リロード時に変更が適用されることを意味します。

### bdr.global_connection_timeout

接続中に待機する最大時間秒単位でデフォルトは 15 秒です。10 進の整数、たとえば、10 として書き込みます。ゼロ、負の数、または指定なしは、無期限に待機することを意味します。最小許可タイムアウトは 2 秒であるため、値 1 は次のように解釈されます。

2.

### bdr.global_keepalives

TCP キープアライブを使用するかどうかを制御しますデフォルトは 1、オンを意味します。キープアライブが不要な場合は、これを 0、つまりオフに変更できます。このパラメーターは、Unix ドメインソケットによって作成された接続では無視されます。

### bdr.global_keepalives_idle

TCP がキープアライブメッセージをサーバーに送信するまでの非アクティブな秒数を制御しますデフォルトは 1 秒です。値 0 は、システムのデフォルトを使用します。このパラメータは、Unix ドメインソケットによって作成された接続、またはキープアライブが無効になっている場合に無視されます。`TCP_KEEPIDLE` または同等のソケットオプションが使用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

### bdr.global_keepalives_interval

サーバーによって確認されなかった TCP キープアライブメッセージを再送信する秒数を制御しますデフォルトは 2 秒です。値 0 は、システムのデフォルトを使用します。このパラメータは、Unix ドメインソケットによって作成された接続、またはキープアライブが無効になっている場合に無視されます。`TCP_KEEPINTVL` または同等のソケットオプションが使用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

### bdr.global_keepalives_count

サーバーへのクライアントの接続がデッドと見なされる前に、失われる可能性のある TCP キープアライブの数を制御しますデフォルトは 3 です。値 0 は、システムのデフォルトを使用します。このパラメータは、Unix ドメインソケットによって作成された接続、またはキープアライブが無効になっている場合に無視されます。`TCP_KEEPCNT` または同等のソケットオプションが使用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

#### bdr.global_tcp_user_timeout

接続が強制的に閉じられるまでに、送信されたデータが未確認のままでいられるミリ秒数を制御しますデフォルトは 5000、つまり 5 秒です。値 0 は、システムのデフォルトを使用します。このパラメーターは、Unix ドメインソケットによって作成された接続では無視されます。TCP_USER_TIMEOUT が利用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

### トポロジ設定

#### bdr.force_full_mesh

フルメッシュトポロジを強制しますデフォルトは on です。 off に設定されている場合、PGD はサブスクライバ専用グループに最適化されたトポロジを使用しようとします。この設定は、最適化されたトポロジの要件が満たされた場合にのみ有効です。詳細については、*Optimizing subscriber-only groups* を参照してください。

### 内部設定 - Raft タイムアウト

#### bdr.raft_global_election_timeout

ネットワーク障害を説明するために、Raft コンセンサスプロトコルは選択と要求のタイムアウトを実装します。この値は、要求がグローバルトップレベルグループに送信されるときに使用されます。デフォルトは 6 秒 6s です。

#### bdr.raft_group_election_timeout

ネットワーク障害を説明するために、Raft コンセンサスプロトコルは選択と要求のタイムアウトを実装します。この値は、要求がサブグループに送信されるときに使用されます。デフォルトは 3 秒 3s です。

#### bdr.raft_response_timeout

レスポンスの場合、 *bdr.raft_global_election_timeout* および *bdr.raft_group_election_timeout* の設定が必要に応じて使用されます。この変数を設定することにより、この動作をオーバーライドできます。bdr.raft_response_timeout の設定は、いずれの選択タイムアウト値よりも小さい必要があります。オーバーライドを無効にするには、この変数を-1 に設定します。デフォルトは-1 です。

### 内部設定 - その他の Raft 値

#### bdr.raft_keep_min_entries

ログ圧縮を行うときに Raft ログに保持するエントリの最小数デフォルトは 1000 、PGD 5.3 以前は 100 です。 0 の値は、ログ圧縮を無効にします。このパラメーターは、Postgres サーバーの起動時にのみ設定できます。

> **警告**
>
> ログ圧縮が無効になっている場合、ログのサイズは永久に増加します。

#### bdr.raft_log_min_apply_duration

ステートマシンを前進させるために、Raft は内部ログにエントリを追加します。通常オペレーションでは、追加には数ミリ秒しかかかりません。これにより、その追加アクションの期間に上限しきい値が設定され、それを超えると `INFO` メッセージがログに記録されます。これは問題を示している可能性があります。デフォルトは 3000 ミリ秒です。

#### bdr.raft_log_min_message_duration

コンセンサス要求をログに記録する場合。PGD コンセンサス要求の往復時間を測定し、時間がこのパラメーターデフォルト 5000 ミリ秒を超えた場合、`INFO` メッセージをログに記録します。

#### bdr.raft_group_max_connections

Postgres サーバーのすべての PGD グループの最大接続数デフォルトは 100 接続です。これらの接続は、グループのノード間で PGD コンセンサス要求を伝送します。このパラメーターは、Postgres サーバーの起動時にのみ設定できます。

### 内部設定 - その他の値

#### bdr.backwards_compatibility

`bdr.bdr_version_num` 、`30618` などで使用されるのと同じ数値形式で、下位互換性のあるバージョンを指定します。デフォルトは現在の PGD バージョンです。一般に不要な効果がある場合でも、以前の PGD バージョンの正確な動作を有効にします。これはリリースごとに異なるため、値が現在のバージョンと異なる場合を除き、構成ファイルで明示的に使用しないことをお勧めします。

#### bdr.track_replication_estimates

ピアノードの適用レートとキャッチアップ間隔の観点からレプリケーション推定を追跡します。CAMO のようなプロトコルは、この情報を使用して、ピアノードの準備状況を推定できます。このパラメーターはデフォルトで有効になっています。

#### bdr.lag_tracker_apply_rate_weight

PGD は、ローカルノードから WAL を適用するという点でピアノードがどれだけ遅れているかを監視し、遅延追跡の適用レートの移動平均を計算します。このパラメーターは、この移動平均計算で新しい計算値がどの程度寄与するかを指定します。デフォルトは 0.1 です。

#### bdr.enable_auto_sync_reconcile

有効にすると、ノードはダウンしたノードに対して最も前にあるノードからデータの自動同期を実行します。デフォルト 6.0.1 からがオンになっています。

### Node management

#### ノードの状態のリスト

| State | Description |
|---|---|
| *NONE* | ワーカーが起動したときにノードの状態は設定解除され、現在の既知の状態にすぐに設定されることが予想されます。 |
| *CREATED* | `bdr.create_node()`が実行されましたが、ノードはまだ EDB Postgres Distributed クラスターのメンバーではありません。 |
| *JOIN_START* | `bdr.join_node_group()`は、ローカルノードを既存の EDB Postgres Distributed クラスターへの参加を開始します。 |
| *JOINING* | ノード参加が開始され、現在初期同期フェーズにあり、ノードにスキーマとデータを作成しています。 |
| *CATCHUP* | 初期同期フェーズが完了しました。現在、結合は、結合が開始されてから上流のピアノードで実行されたトランザクションを取得および適用する最後のステップにあります。 |
| *STANDBY* | ノード参加は完了しましたが、変更のブロードキャストはまだ開始されていません。すべての結合はこの状態でしばらく時間を費やしますが、ロジカルスタンバイとして定義されている場合、ノードはこの状態を継続します。 |
| *PROMOTE* | ノードはロジカルスタンバイであり、`bdr.promote_node`が呼び出されてノード状態を `ACTIVE`に移動したところです。これらの 2 つの `PROMOTE`状態は、1 つのノードのみが `STANDBY`より高いが `ACTIVE`より低い状態になることができるという事実と一貫している必要があります。 |
| *PROMOTING* | ロジカルスタンバイからフル PGD ノードへの昇格が進行中です。 |
| *ACTIVE* | ノードは完全な PGD ノードであり、現在 `ACTIVE`です。これは、最も一般的なノードのステータスです。 |
| *PART_START* | ノードは `ACTIVE`または `STANDBY`であり、 EDB Postgres Distributed クラスターからノードを削除するために `bdr.part_node`が呼び出されたばかりです。 |
| *PARTING* | ノードは他のノードから切断し、コンセンサスまたはレプリケーションでそれ以上の役割を果たしません。 |
| *PART_CATCHUP* | 分割されていないノードは、最近分割されたノードから欠落しているデータを同期します。 |
| *PART_CLEANUP* | 非分割ノードは、すべてのノードのグループスロットが PARTED ノードから発生したすべてのトランザクションに追いつくまで待機します。 |
| *PARTED* | これで、すべてのノードでノード分離操作が完了しました。 |

一度に 1 つのノードのみが PROMOTE または PROMOTING の状態になります。

### ノード管理コマンド

PGD は、既存のノードの物理コピー `pg_basebackup` を使用してノードを PGD グループに追加するためのコマンドラインユーティリティも提供します。

### bdr_init_physical

これは、PostgreSQL の bin ディレクトリに追加される通常のコマンドです。

データディレクトリを指定する必要があります。このデータディレクトリが空の場合、`pg_basebackup -X stream` を使用して、高速なブロックレベルのコピー操作を使用してディレクトリを埋めます。

指定されたデータディレクトリが空でない場合、新しいノードのベースとして使用されます。最初は、キャッチアップを待機し、マスターノードに昇格してから、PGD グループに参加します。--standby オプションを使用する場合、ロジカルスタンバイノードに変わります。

このコマンドは、すべての PostgreSQL ネイティブ論理レプリケーションサブスクリプションをデータベースから削除または-S オプションを使用する場合を無効にします。また、レプリケーションオリジンとスロットも削除します。

### 概要

```
bdr_init_physical [OPTION] ...
```

### オプション

#### 一般オプション

- -D, --pgdata=DIRECTORY —新しいノードに使用するデータディレクトリ。空または存在しないディレクトリ、または pg_basebackup -X stream コマンド必須を使用して設定されたディレクトリのいずれかです。
- -l, --log-file=FILE — ロギングには FILE を使用します。デフォルトは bdr_init_physical_postgres.log です。
- -n, --node-name=NAME —新しく作成したノードの名前必須。
- --replication-sets=SETS —使用するレプリケーションセット名のコンマ区切りリストの名前。指定しない場合、すべてのレプリケーションセットが使用されます。
- --standby —完全な送信/受信ノードではなく、ロジカルスタンバイ受信専用ノードを作成します。
- --node-group-name —参加するグループ。デフォルトはソースノードと同じグループです。
- -s, --stop —初期化が完了したら、サーバーを停止します。
- -v —ロギングの詳細度を高めます。
- -L —空のまたは存在しないデータディレクトリ-D オプションで使用される場合、選択的な pg_basebackup を実行します。これは EDB Postgres Extended Server のみの機能です。
- -S —論理レプリケーションサブスクリプションをドロップする代わりに、無効にします。

#### 接続オプション

- -d, --remote-dsn=CONNSTR —リモートノードの接続文字列必須。
- --local-dsn=CONNSTR —ローカルノードの接続文字列必須。

#### 構成ファイルのオーバーライド

- --hba-conf —新しい pg_hba.conf へのパス。
- --postgresql-conf —新しい postgresql.conf へのパス。
- --postgresql-auto-conf —新しい postgresql.auto.conf へのパス。

#### 注

コマンドで指定されたレプリケーションセット名は、ノードが PGD グループに参加する前にデータディレクトリに存在するデータに影響を与えません。これは、bdr_init_physical が独自のベースバックアップを作成するか、既存のベースバックアップが新しい PGD ノードに昇格するかにかかわらず当てはまります。したがって、--replication-sets オプションは、ノードが PGD ノードグループに参加した後にパブリッシュおよびサブスクライブされたデータにのみ影響します。この動作は、bdr.join_node_group() を使用する場合など、論理結合でレプリケーションセットを使用する方法とは異なります。

オペレーターは、結合が完了した後、不要なテーブルを切り詰めることができます。bdr.tables カタログを参照して、レプリケーションセットのメンバーシップを判断し、サブスクライブされたレプリケーションセットのメンバーではないテーブルを特定します。次の理由から、テーブルを削除するのではなく、切り詰めることを強くお勧めします。

- DDL レプリケーションセットは行 DML レプリケーションセットと同じではないため、誤って他のノードでテーブルをドロップする可能性があります。
- 後でテーブルをレプリケーションセットに追加し、ノードの一部のサブセットにドロップした場合、レプリケーションセットに追加する前に、DDL 競合を作成せずにこれらのノードでのみ再作成する必要があります。

非レプリケートテーブルを切り捨て、存在するが空のままにする方が簡単かつ安全です。

### bdr_config

このコマンドラインユーティリティを使用すると、PGD インストールの構成を確認できます。これは、PostgreSQL に付属する pg_config ユーティリティに似ています。トラブルシューティングとサポートを支援するために使用できます。

#### 概要

```
bdr_config [OPTION] ...
```

#### オプション

| Option | Description |
|---|---|
| *--all* | 構成内のすべてのキーと値を表示します。 |
| *--version* | BDR バージョン関連のキーと値のみを表示します。これには、BDR 拡張機能のフルバージョン、それが実行されている Postgres バージョンとフレーバー、および BDRPG および BDR プラグイン API バージョンが含まれます。 |
| *--debug* | ビルド情報や機能の有効化を含む、 BDR デバッグキーと値のみを表示します。 |

例

```
$ /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

SQL インターフェイスを使用して、ノードを動的に追加および削除できます。

### bdr.alter_node_group_option

PGD ノードグループ構成を変更します。

#### 概要

```
bdr.alter_node_group_option(node_group_name text,
    config_key text,
    config_value text)
```

#### パラメーター

| Name | Description |
|---|---|
| *node_group_name* | 変更するグループの名前。 |
| *config_key* | 変更するノードグループのオプションのキー。 |
| *config_value* | 指定されたキーに設定する新しい値。 |

`config_value` は、オプションに適切なデータ型に解析されます。

この表は、この機能を使用して変更できるグループオプションを示しています。

| Name | Type | Description |
|---|---|---|
| *apply_delay* | *interval* | 着信変更を適用するまでノードが待機する時間。このオプションは、主に遅延加入者専用ノードを使用して特別なサブグループを設定する場合に役立ちます。これは、データノードを含むグループまたはトップレベルのグループでは設定しないでください。デフォルトは `0s`です。 |
| *check_constraints* | *boolean* | 適用プロセスがレプリケートデータを書き込むときに制約をチェックするかどうか。デフォルト値をそのまま使用することをお勧めします。そうしないと、データが失われる危険があります。有効な値は `on`または `off`です。デフォルトは `on`です。 |
| *default_commit_scope* | *text* | デフォルトで使用するコミットスコープ、最初は `local`コミットスコープです。このオプションは、トップレベルのノードグループにのみ適用されます。同じコミットスコープの異なるオリジングループに個々のルールを使用できます。詳細については、*Origin groups* を参照してください。 |

次のページに続く

表 119 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| *enable_routing* | *boolean* | ここで、グループ書き込みリーダーを介した ``Connection Manager` <https://www.enterprisedb.com/docs/pgd/latest/reference/connection-manager/>`__ は、特定のグループで有効になっています。有効な値は `on`または `off`です。デフォルトは、サブグループの場合は `on`、クラスターグループの場合は `off`です。 |
| *enable_raft* | *boolean* | グループが独自の Raft コンセンサスを持つかどうか。このオプションは `enable_routing`〜`on`を設定するために必要です。トップレベルグループのこのオプションは常に `on`です。有効な値は `on`または `off`です。サブグループのデフォルトは `on`です。 |
| *enable_wal_decoder* | *boolean* | デコードワーカープロセスを有効/無効にします。`streaming_mode`が既に有効になっている場合、デコードワーカープロセスを有効にすることはできません。有効な値は `on`または `off`です。デフォルトは `off`です。 |
| *location* | *text* | グループの場所に関する情報。このオプションは、モニタリングのための純粋なメタデータです。デフォルトは `'"`空の文字列です。 |
| *num_writers* | *integer* | このノードグループをバッキングするサブスクリプションの並列ライターの数。有効な値は `-1`または正の整数です。 *-1* `は、*GUC :ref:`pg_upgrade <PGD settings>*で指定された値が使用されることを意味します。 `-1`がデフォルトです。 |
| *route_reader_max_lag* | *integer* | ノードが実行可能な読み取り専用ノードと考えられるバイト単位の最大遅延。将来の使用のために現在予約されています。 |
| *route_writer_max_lag* | *integer* | 書き込みリーダーとして選択される新しい書き込み候補のバイト単位の最大遅延。これを通過する候補者がない場合、ライタは選択されません。デフォルトは `-1`です。 |
| *route_writer_wait_flush* | *boolean* | PGD がフラッシュを待機する必要があるかどうかを切り替えるかどうか。将来の使用のために現在予約されています。 |

次のページに続く

表 119 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| *streaming_mode* | *text* | 大規模なトランザクションのストリーミングを有効/無効にします。 *off* **に設定すると、ストリーミングが無効になります。他の値に設定されている場合、大規模なトランザクションは進行中にデコードされ、変更がダウンストリームに送信されます。値が** *file* **に設定されている場合、ストリーミングトランザクションの着信変更はファイルに保存され、上流でトランザクションがコミットされた後にのみ適用されます。値が** *writer* **に設定されている場合、着信変更はライターのいずれか利用可能な場合に直接送信されます。** *:ref:`Parallel Apply<Parallel Apply>`* が無効になっている場合、またはストリーミングトランザクションを処理できるライタがない場合、変更はファイルに書き込まれ、トランザクションがコミットされた後に適用されます。値が `auto` に設定されている場合、 PGD は、トランザクションプロパティと使用可能なリソースに応じて、*file* **と** *writer* **の間でインテリジェントに選択しようとします。** *WAL* **デコーダーが既に有効になっている場合、** *streaming_mode* **を有効にすることはできません。デフォルトは** *auto* **です。詳細については、** *:ref:`***トランザクションストリーミングでの使用<トランザクションストリーミングでの使用>** を参照してください。 |
| *failover_slot_scope* | *text* | PGD 5.7 以降のみ。論理スロットフェールオーバーサポートのスコープを設定します。有効な値は`global`または`local`です。デフォルトは`local`です。詳細については、 *CDC Failover support* を参照してください。 |

### 戻り値

`bdr.alter_node_group_option()` は、成功すると `VOID` を返します。

提供されたパラメーターのいずれかが無効な場合、 `ERROR` が発生します。

### 注

ビュー *bdr.node_group_summary* を介して、ノードグループオプションの現在の状態を調べることができます。

このファンクションは、デフォルトを変更するように要求をグループコンセンサスメカニズムに渡します。行われた変更は、コンセンサスメカニズムを使用してグローバルに複製されます。

ファンクションはトランザクションではありません。要求はバックグラウンドで処理されるため、ファンクション呼び出しをロールバックすることはできません。また、変更は現在のトランザクションにすぐに表示されない場合があります。

このファンクションはロックを保持しません。

### bdr.alter_node_interface

指定されたノードの接続文字列 DSN を変更します。

#### 概要

```
bdr.alter_node_interface(node_name text, interface_dsn text)
```

#### パラメーター

| Name | Description |
|---|---|
| *node_name* | 変更する既存のノードの名前。 |
| *interface_dsn* | ノードの新しい接続文字列。 |

#### 注

このファンクションを実行し、ローカルノードでのみ変更を加えます。これは、通常、変更されるノードを含む、PGD グループ内のすべてのノードで実行することを意味します。

このファンクションはトランザクションです。ロールバックできます。変更は現在のトランザクションに表示されます。

ファンクションはローカルノードでロックを保持します。

### bdr.alter_node_option

PGD ノードのルーティング構成を変更します。

#### 概要

```
bdr.alter_node_option(node_name text,
    config_key text,
    config_value text);
```

#### パラメーター

| Name | Description |
|---|---|
| *node_name* | 変更するノードの名前。 |
| *config_key* | 変更するノードのオプションのキー。 |
| *config_value* | 指定されたキーに設定する新しい値。 |

この機能を使用して変更できるノードオプションは次のとおりです。

| Config Key | Description |
|---|---|
| *route_priority* | 同じノードグループ内の他のノードに対するノードの相対ルーティング優先度。デフォルトは `'-1'`です。 |
| *route_fence* | ノードがルーティングからフェンスされているかどうか。 true の場合、ノードは接続マネージャーから接続を受信できません。レプリケーションは影響を受けません。デフォルトは `'f'`false です。 |
| *route_writes* | 書き込みをこのノードにルーティングできるかどうか、つまりノードが書き込みリーダーになることができるかどうか。デフォルトは、データノードの場合は `'t'*true* **であり、他のノードタイプの場合は**'f'`false です。 |
| *route_reads* | 読み取り専用接続をこのノードにルーティングできるかどうか。将来の使用のために現在予約されています。デフォルトは、データおよびサブスクライバ専用ノードの場合は `'t'*true*、**監視ノードおよびスタンバイノードの場合は**'f'`false です。 |
| *route_dsn* | このノードへの接続に使用するプロキシの dsn。このオプションはオプショナル。設定されていない場合、デフォルトでノードの `node_dsn`値が使用されます。 |

### bdr.alter_subscription_enable

指定されたサブスクリプションまたはローカル PGD ノードのすべてのサブスクリプションを有効にします。これは、再開サブスクリプションとも呼ばれます。サブスクリプションが既に有効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

#### 概要

```
bdr.alter_subscription_enable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false
)
```

#### パラメーター

| Name | Description |
|---|---|
| *subscription_name* | 有効にするサブスクリプションの名前。 NULL の場合デフォルト、ローカルノードのすべてのサブスクリプションが有効になります。 |
| *immediate* | アクションをすぐに強制するために使用され、有効なサブスクリプションに関連付けられているすべてのワーカーを起動します。このオプションが `true`の場合、トランザクションブロック内でこのファンクションを実行できません。 |

#### 注

この機能はレプリケートされず、ローカルノードサブスクリプション特定のノードまたはすべてのノードにのみ影響します。

このファンクションはトランザクションです。ロールバックできます。現在のトランザクションは、カタ

ログの変更を確認できます。サブスクリプションワーカーは、トランザクションがコミットされた後、バックグラウンドプロセスによって起動されます。

### bdr.alter_subscription_disable

指定されたサブスクリプションまたはローカル PGD ノードのすべてのサブスクリプションを無効にします。オプションで、無効なサブスクリプションに関連付けられているすべてのワーカーをすぐに停止することもできます。これは、サブスクリプションの一時停止とも呼ばれます。サブスクリプションが既に無効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

#### 概要

```
bdr.alter_subscription_disable(
    subscription_name name DEFAULT NULL,
    immediate boolean DEFAULT false,
    fast boolean DEFAULT true
)
```

#### パラメーター

| Name | Description |
|---|---|
| *subscription_name* | 無効にするサブスクリプションの名前。 NULL の場合デフォルト、ローカルノードのすべてのサブスクリプションが無効になります。 |
| *immediate* | アクションをすぐに強制し、無効になったサブスクリプションに関連付けられているすべてのワーカーを停止するために使用されます。このオプションが`true`の場合、トランザクションブロック内でこのファンクションを実行できません。 |
| *fast* | この引数は、`immediate`の動作に影響を与えます。 `true`デフォルトに設定されている場合、現在の作業が完了するのを待たずに、無効なサブスクリプションに関連付けられているすべてのワーカーを停止します。 |

#### 注

このファンクションはレプリケートされず、ローカルノードサブスクリプション特定のサブスクリプションまたはすべてのサブスクリプションにのみ影響します。

このファンクションはトランザクションです。ロールバックできます。現在のトランザクションは、カタログの変更を確認できます。ただし、サブスクリプションワーカーが停止するタイミングは、`immediate`の値に依存します。 `true` に設定されている場合、ワーカーは `COMMIT` を待たずに stop を受け取ります。`fast` 引数が `true` に設定されている場合、ワーカーの中断は、現在の作業が完了するまで待機しません。

### bdr.create_node

ノードを作成します。

#### 概要

```
bdr.create_node(node_name text,
                local_dsn text,
                node_kind DEFAULT NULL)
```

#### パラメーター

| Name | Description |
|---|---|
| *node_name* | 新しいノードの名前。データベースごとに 1 つのノードのみが許可されます。有効なノード名は、小文字、数字、ハイフン、アンダースコアで構成されます。 |
| *local_dsn* | ノードへの接続文字列。 |
| *node_kind* | *data* `デフォルト、`*standby*、*subscriber-only*、または `witness`のいずれか。このパラメーターを設定しない場合、または `NULL`を提供する場合、デフォルトの `data`ノード種類が使用されます。 |

#### 注

このファンクションは、関連するパブリック接続文字列を使用してローカルノードのレコードを作成します。ローカルレコードは 1 つだけであるため、作成されると、ファンクションは再度実行されるとエラーを報告します。

このファンクションはトランザクションファンクションです。ロールバックできます。これによる変更は、現在のトランザクションに表示されます。

ファンクションは、トランザクションが終了するまで、新しく作成されたノードのロックを保持します。

### bdr.create_node_group

PGD ノードグループを作成します。デフォルトでは、ローカルノードは唯一のメンバーとしてグループに参加します。bdr.join_node_group() を使用してグループにノードを追加できます。

#### 概要

```
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)
```

#### パラメーター

| Name | Description |
|---|---|
| *node_group_name* | 新しい PGD グループの名前。ノード名と同様に、有効なグループ名は小文字、数字、アンダースコアのみで構成されます。 |
| *parent_group_name* | ノードサブグループを作成する場合、これは親グループの名前である必要があります。クラスターのメインノードグループを作成するときに `NULL`デフォルトを提供します。 |
| *join_node_group* | ノードが作成中のグループに参加するかどうかを決定します。デフォルト値は`true`です。サブグループを作成するときに`false`を提供することは、たとえば、独立したリモートグループを作成する場合、ローカルノードが新しいグループに参加しないことを意味します。この場合、`parent_group_name`を指定する必要があります。 |
| *node_group_type* | 有効な値は`NULL`または`subscriber-only`です。 *NULL* **`デフォルトは、通常の汎用ノードグループを作成するためです。** *`subscriber-only`* **は、** *:ref:`Creating Subscriber-only groups and nodes<Creating Subscriber-only groups and nodes>* を作成するためのもので、そのメンバーはクラスター内の完全に参加したノードからのみ変更を受信しますが、他のノードに変更を送信することはありません。 |

#### 注

このファンクションは、ローカルノードで実行されているローカルコンセンサスワーカーに要求を渡します。

ファンクションはトランザクションではありません。グループの作成はバックグラウンドプロセスであるため、機能が完了すると、変更をロールバックすることはできません。また、変更は現在のトランザクションにすぐに表示されない場合があります。`bdr.wait_for_join_completion` を呼び出して、そうなるまで待機できます。

グループの作成では、ロックは保持されません。

### bdr.drop_node_group

空の PGD ノードグループを削除します。グループ内に参加したノードがある場合、ファンクションは失敗します。

#### 概要

```
bdr.drop_node_group(node_group_name text)
```

#### パラメーター

| Name | Description |
|---|---|
| *node_group_name* | ドロップする PGD グループの名前。 |

#### 注

このファンクションは、グループをドロップするように要求をグループコンセンサスメカニズムに渡します。ファンクションはトランザクションではありません。ドロッププロセスはバックグラウンドで発生し、ロールバックすることはできません。

### bdr.join_node_group

ローカルノードを既存の PGD グループに参加させます。

#### 概要

```
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
)
```

#### パラメーター

| Name | Description |
|---|---|
| *join_target_dsn* | ローカルノードを追加する PGD グループ内の既存のソースノードへの接続文字列を指定します。 |
| *node_group_name* | PGD グループのオプションの名前。デフォルトは NULL で、ソースノードに存在する情報からグループ名を検出しようとします。 |
| *wait_for_completion* | 戻る前に、参加プロセスが完了するのを待ちます。デフォルト `true` |
| *synchronize_structure* | 結合中にデータベース構造スキーマの同期を実行するかどうかを指定します。デフォルト設定の `all` は、完全なデータベース構造を同期します。 `none` は、構造体を同期しません。ただし、データは引き続き同期されます。つまり、データベース構造が参加ノードに既に存在する必要があります。デザインとして、スキーマもデータも監視ノードに同期されないことに注意してください。 |

`wait_for_completion` が `false` として指定された場合、ファンクション呼び出しは、参加プロシージャが開始されるとすぐに返されます。結合の進行状況はログファイルと bdr.event_summary で確認できます。

情報ビュー。ファンクション bdr.wait_for_join_completion() を呼び出すことができます

`bdr.join_node_group()` の後、結合操作が完了するまで待機します。`verbose_progress` を `true` に設定して bdr.wait_for_join_completion() が呼び出された場合、進行状況情報を出力できます。

#### 注

このファンクションは、join_target_dsn 接続文字列がポイントするノードを介して、グループコンセンサスメカニズムに要求を渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

ファンクションはトランザクション的ではなく、トランザクションで実行されるとエラーを生成します。参加プロセスはバックグラウンドで発生し、ロールバックすることはできません。wait_for_completion が true に設定されている場合、または後で bdr.wait_for_join_completion を呼び出した場合、変更はローカルセッションにのみ可視されます。

ノードは 1 つのグループのみに所属できるため、このファンクションは各ノードで 1 回のみ呼び出すことができます。

ノード参加は、PGD グループのロックを保持しません。

### bdr.part_node

PGD グループからノードを削除し、最終的にクラスター内のすべてのノードから分離されたノードのメタデータを削除します。

- ローカルノードの場合、リモートノードに関する情報を含むすべてのノードメタデータを削除します。
- リモートノードの場合、その特定のノードのメタデータのみを削除します。

この操作では、ノードからデータは削除されません。

削除するノードを含む、PGD グループ内の任意のアクティブノードからファンクションを呼び出すことができます。

削除されるノードから分割を実行すると、削除のステータスが誤って報告される、または全く報告されないリスクがあります。これは、削除の過程で、通信がクラスターの残りの部分から切断されるためです。削除は成功する場合がありますが、コマンドを発行したノードに、他のノードで失敗または成功したことを通知する方法はありません。同じ理由で、ファンクションは完了を待機するように設定できません。

ノードが自分自身を分離すると、クラスターの一部ではなくなるため、クラスター内の他のノードを分離することはできません。

ノードを使用してクラスターから自分自身を分離することは避けることをお勧めします。代わりに、完了を待機できるノードからノード分割操作を実行し、操作が完了した後にクラスターステータスを確認します。

**注釈**

ローカルノードを分離する場合、*wait_for_completion* を `false` に設定する必要があります。それ以外の場合、エラーが報告されます。

**警告**

このアクションは永続的です。ノードへのレプリケーションを一時的に停止する場合は、*bdr.alter_subscription_disable()* を使用します。

### 概要

```
bdr.part_node (
    node_name text,
    wait_for_completion boolean DEFAULT true,
    force boolean DEFAULT false
)
```

### パラメーター

| Name | Description |
|---|---|
| *node_name* | パートへの既存のノードの名前。 |
| *wait_for_completion* | `true`の場合、ノードがクラスターから完全に分離されるまで、ファンクションは返されません。それ以外の場合、ファンクションは分割プロシージャを開始し、待機せずにすぐに戻ります。ローカルノードで実行する場合、または`force`を使用する場合、常に`false`に設定します。 |
| *force* | ローカルノードでノードを強制的に削除します。これにより、コンセンサスに到達できない場合、またはノード分割プロセスがスタックしている場合、ノードの状態をローカルに設定します。 |

**警告**

*force = true* を使用すると、PGD グループが一貫性のない状態になる場合があります。他の方法でノードを削除できない障害から回復するためにのみ使用します。

### 注

このファンクションは、グループコンセンサスメカニズムに要求を渡して、特定のノードを分割します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。分離プロセスはバックグラウンドで発生し、ロールバックすることはできません。`wait_for_completion` が `true` に設定されている場合、分割プロセスによって行われた変更はローカルトランザクションにのみ可視されます。

`force` が `true` に設定されている場合、コンセンサスが失敗すると、このファンクションはローカルノードでのみ特定のノードの状態を設定します。このような場合、ファンクションはトランザクションであり、ファンクションがノードの状態を変更するため、ロールバックできます。`force` が `true` に設定された状態で既に分離処理中のノードでファンクションが呼び出された場合、指定されたノードをローカルに分離されたものとしてマークして、終了します。これは、クラスターでコンセンサスに到達できない場合つまりノードの大部分がダウンしている場合、または分割プロセスがスタックしている場合にのみ役立ちます。

ただし、書き込みを受信していた分割ノードが書き込みを行った場合、実際にはスタックせずに分割プロセスに時間がかかる場合があることを考慮することが重要です。他のノードは、特定のノードからの欠落データを再同期する必要があります。他のノードは、すべてのノードのグループスロットが PARTED ノードから発生したすべてのトランザクションに追いつくまで、待機する必要があります。

強制的な分離はこの再同期を完全にスキップし、他のノードを一貫性のない状態のままにする可能性があります。

分離プロセスはロックを保持しません。

### bdr.promote_node

ローカルロジカルスタンバイノードを PGD グループの完全なメンバーに昇格させます。

#### 概要

```
bdr.promote_node(wait_for_completion boolean DEFAULT true)
```

#### 注

このファンクションは、デフォルトを変更するように要求をグループコンセンサスメカニズムに渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

ファンクションはトランザクションではありません。プロモーションプロセスはバックグラウンドで発生し、ロールバックすることはできません。wait_for_completion が true に設定されている場合、または後で bdr.wait_for_join_completion を呼び出した場合、変更はローカルトランザクションにのみ可視されます。

プロモーションプロセスは、他のプロモーションに対してロックを保持します。このロックは他の bdr.promote_node 呼び出しをブロックしませんが、プロモーションのバックグラウンドプロセスが一度に複数のノードで進行することを防ぎます。

### bdr.switch_node_group

ローカルノードを現在のサブグループから同じ既存の PGD ノードグループ内の別のサブグループに切り替えます。

#### 概要

```
bdr.switch_node_group (
    node_group_name text,
    wait_for_completion boolean DEFAULT true
)
```

#### パラメーター

| Name | Description |
|---|---|
| *node_group_name* | PGD グループまたはサブグループの名前。 |
| *wait_for_completion* | 戻る前に、スイッチプロセスが完了するのを待ちます。デフォルト `true` |

wait_for_completion が false に設定されている場合、これは非同期呼び出しであり、スイッチングプロシージャが開始するとすぐに返されます。スイッチの進行状況は、ログと bdr.event_summary 情報ビューで、または bdr.switch_node_group() が戻った後に bdr.wait_for_join_completion() ファンクションを呼び出すことで確認できます。

#### 注

このファンクションは、要求をグループコンセンサスメカニズムに渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

ファンクションはトランザクションではありません。スイッチングプロセスはバックグラウンドで発生し、ロールバックすることはできません。wait_for_completion が true に設定されている場合、または後で bdr.wait_for_join_completion を呼び出した場合、変更はローカルトランザクションにのみ可視されます。

ローカルノードは、クラスターを分割する必要なく、現在のサブグループから同じ PGD ノードグループ内の別のサブグループにメンバーシップを変更します。ノードの種類は、ターゲットサブグループ内の既存のノードの種類と一致する必要があります。

ノードスイッチングは、PGD グループのロックを保持しません。

制限現在、この機能では、サブグループとその PGD ノードグループ間のスイッチングのみが許可されています。サブグループ間の移動を有効にするには、2 つの別個の呼び出しを行う必要があります。1) サブグループからノードグループにスイッチし、2) ノードグループから他のサブグループにスイッチします。

### bdr.sync_node_cancel

このファンクションは、指定されたオリジンおよびソースノードの同期要求をキャンセルします。

#### 概要

```
bdr.sync_node_cancel(origin text, source text)
```

#### パラメーター

| Name | Description |
|---|---|
| *origin* | 元ノードの名前。 |
| *source* | ソースノードの名前。 |

#### 注

このファンクションは、指定されたオリジンとソースを持つすべてのターゲットのすべての同期ノード要求をキャンセルします。書き込みリードからのみ呼び出すことができます。

### bdr.wait_for_join_completion

このファンクションは、ローカルノードの参加手順が完了するのを待機します。

#### 概要

```
bdr.wait_for_join_completion(verbose_progress boolean DEFAULT false)
```

### パラメーター

| Name | Description |
|---|---|
| *verbose_progress* | オプションで、参加プロシージャ中に実行される個々のステップに関する情報を出力します。 |

### 注

このファンクションは、ローカルノードのチェック状態が bdr.create_node_group 、bdr.join_node_group 、または bdr.promote_node で設定されたターゲット状態に到達するまで待機します。

## Routing functions

## Commit scopes

コミットスコープは、PGD システム内でトランザクションのコミットと競合を処理する方法を決定するルールです。それらの詳細については、Commit Scopes を参照してください。

次のファンクションを使用して、コミットスコープを操作できます。

- bdr.create_commit_scope
- bdr.alter_commit_scope
- bdr.drop_commit_scope

### コミットスコープの構文

コミットスコープルールの全体的な文法は次のように構成されます。

```
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:
    [ ON { received|replicated|durable|visible } ]
```

(次のページに続く)

(前のページからの続き)

```
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
```

node_group は、PGD データノードグループの名前です。

#### commit_scope_degrade_operation

commit_scope_degrade_operation は、定義されるルール全体としてより制限の少ないコミットスコープグループと同じコミットスコープ種類であるか、非同期 ASYNC のいずれかです。

例、*you can degrade* ALL SYNCHRONOUS COMMIT から MAJORITY SYNCHRONOUS COMMIT 、または MAJORITY SYNCHRONOUS COMMIT から ANY 3 SYNCHRONOUS COMMIT 、または ANY 3 SYNCHRONOUS COMMIT から ANY 2 SYNCHRONOUS COMMIT 。 SYNCHRONOUS COMMIT から ASYNC に低下することもできます。ただし、関係するコミットスコープグループに関係なく、SYNCHRONOUS COMMIT から GROUP COMMIT またはその逆に低下することはできません。

AND を使用して、それぞれが独自の低下句を持つルールを組み合わせることもできます。

```
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
```

#### コミットスコープターゲット

#### ORIGIN_GROUP

特定のグループをターゲットとする代わりに、トランザクションが発生する最下位グループを動的に参照する ORIGIN_GROUP を使用することもできます。したがって、トップレベルグループ top_group と、子として 2 つのサブグループ left_dc および right_dc がある場合、次のようなコミットスコープを追加します。

```
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
);
```

left_dc のノードで発生したトランザクションの場合、 left_dc のノードの大部分は、トランザクションがコミットされる前に同期的に確認する必要があることを意味します。さらに、同じルールは、right_dc

のノードから発生したトランザクションの場合、 right_dc のノードの大部分が、コミットされる前にトランザクションを同期的に確認する必要があることも意味します。これにより、left_dc 用と right_dc 用の 2 つの別個のルールをコミットスコープに追加する必要がなくなります。

### コミットスコープグループ

#### 任意

例 ANY 2 (left_dc)

このコミットスコープグループのトランザクションは、 left_dc グループ内の 2 つのノードがトランザクションを処理したことを確認した後、コミットされたとみなされます。

### ANY NOT

例 ANY 2 NOT (left_dc)

left_dc グループにない 2 つのノードがトランザクションを処理したことを確認した場合、このコミットスコープグループのトランザクションはコミットされたとみなされます。

#### マジョリティ

例 MAJORITY (left_dc)

left_dc グループ内のノードの大部分がトランザクションを処理したことを確認した場合、このコミットスコープグループのトランザクションはコミットされたとみなされます。

#### マジョリティではない

例 MAJORITY NOT (left_dc)

left_dc グループにないノードの大部分がトランザクションを処理したことを確認した場合、このコミットスコープグループのトランザクションはコミットされたとみなされます。

#### すべて

例 ALL (left_dc)

left_dc グループ内のすべてのノードがトランザクションを処理したことを確認した場合、このコミットスコープグループのトランザクションはコミットされたとみなされます。

ALL を 耐久性オプショングループコミット/ *CAMO* とともに使用する場合、リコンシリエーションの問題を回避するには、*commit_*決める設定 設定を raft に設定する必要があります。

#### すべてではない

例 ALL NOT (left_dc)

left_dc グループにないすべてのノードがトランザクションを処理したことを確認した場合、このコミットスコープグループのトランザクションはコミットされたとみなされます。

### 確認レベル

確認レベルは、リモート PGD ノードがトランザクションの処理中に特定のポイントに到達したことを確認する時点を設定します。

#### ON を受信しました

トランザクションは、ローカルアプリケーションを開始する前に、受信後すぐに確認されます。

#### ON が複製されました

トランザクションは、トランザクションの変更を適用した後、ディスクにフラッシュする前に確認されます。

#### ON 耐久性

トランザクションは、すべての変更がディスクにフラッシュされた後に確認されます。

#### ON 可視

これはデフォルトの可視性です。トランザクションは、すべての変更がディスクにフラッシュされ、同時トランザクションに可視された後に確認されます。

### コミットスコープの種類

コミットスコープの種類の詳細とそのパラメーターの詳細

- 同期コミット
- 耐久性オプショングループコミット/ *CAMO*
- *CAMO (Commit At Most Once)*
- *Lag Control*

> **注釈**
>
> パラメーター値 *Postgres GUC parameter value conventions* を使用して、Boolean、enum、int、および interval 値を指定します。

```
SYNCHRONOUS COMMIT [ DEGRADE ON (degrade_on_parameter = value ) TO commit_scope_
↪degrade_operation ]
```

### DEGRADE ON パラメーター

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | ミリ秒単位のタイムアウト他の単位を受け入れます。その後、オペレーションが低下します。 0 は設定されていないことを意味します。 |
| *require_write_lead* | Boolean | False | ノードが縮退動作に切り替えることができるように書き込みリードである必要があるかどうかを指定します。 |

これらは、コミットスコープルールがより制限の低いオペレーションモードに低下する条件を設定します。

### commit_scope_degrade_operation

`commit_scope_degrade_operation` は、より制限の少ないコミットスコープグループの `SYNCHRONOUS COMMIT` であるか、非同期 `ASYNC` である必要があります。

### グループコミット

ノードのコンセンサスによってコミットを確認でき、競合解決設定を制御し、**同期コミット** と同様に、オプションのルール低下パラメーターがあります。

```
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 パラメーター

| Parameter | Type | Default | Description |
|---|---|---|---|
| *transaction_tracking* | Boolean | Off/False | トランザクションのステータスを追跡するかどうかを指定します。 *transaction_tracking* **設定** を参照してください。 |
| *conflict_resolution* | enum | async | 競合を処理する方法を指定します。 *async`|`eager* 。 *fence_resolution* **設定** を参照してください。 |
| *commit_decision* | enum | group | COMMIT 決定を行う方法を指定します。 *group`|`partner`|`raft* 。 *commit_***決める設定** を参照してください。 |

### ABORT ON パラメーター

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | ミリ秒単位のタイムアウト他の単位を受け入れます。 0 は設定されていないことを意味します。 |
| *require_write_lead* | Boolean | False | CAMO のみ。設定されている場合、トランザクションをローカル非同期モードに切り替えるには、コンセンサス要求が必要です。 |

### DEGRADE ON パラメーター

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | ミリ秒単位のタイムアウト他の単位を受け入れます。その後、オペレーションが低下します。 0 は設定されていないことを意味します。 |
| *require_write_lead* | Boolean | False | ノードが縮退動作に切り替えることができるように書き込みリードである必要があるかどうかを指定します。 |

### transaction_tracking 設定

true に設定されている場合、2 フェーズコミットトランザクション

- ライタが PREPARE メッセージを処理しているときにコミット決定をルックアップします。
- 中断から回復するときは、中断前に準備されたトランザクションを検索します。見つかると、トランザクションのコミットスコープと、対応する RAFT コミット決定を検索します。ノードがトランザクションのオリジンであり、RAFT コミット決定がなく、コミットスコープで `transaction_tracking` がオンであると仮定します。その場合、コミットまたは中止されるまで、この未解決のトランザクションの RAFT コミット決定を定期的に検索します。

### fence_resolution 設定

値 `async` は、レプリケーション中に競合解決ポリシーを使用して競合を非同期的に解決することを意味します。

値 `eager` は、COMMIT 中に、競合するトランザクションのいずれかがアボートすることにより、競合が熱心に解決されることを意味します。

Eager は、 MAJORITY または ALL コミットスコープグループでのみ使用できます。

ALL コミットスコープグループで使用する場合、リコンシレーションの問題を回避するには、*commit_決める設定* を raft に設定する必要があります。

欠落列の競合の解決 を参照してください。

#### commit_決める設定

値 group は、前述の commit_scope_group 仕様が耐久性だけでなく COMMIT 決定にも影響を与えることを意味します。

値 partner は、パートナーノードがトランザクションをコミットできるかどうかを決定することを意味します。この値は、2 つのデータノードを持つグループでのみ許可されます。

値 raft は、決定が PGD のビルトイン Raft コンセンサスを利用することを意味します。選択したコミットスコープグループのすべてのノードがトランザクションを確認したら、PGD クラスター内のすべてのノードがトランザクションを記録したことを確認するために、全ノードの Raft で記録されます。

ALL コミットスコープグループが使用されている場合、このオプションは使用して、決定を巡ってノード間で相違がないことを保証する必要があります。このオプションはパフォーマンスが低い場合があります。

決定をコミットする を参照してください。

#### commit_scope_degrade_operation 設定

commit_scope_degrade_operation は、より制限の少ないコミットスコープグループの GROUP_COMMIT であるか、非同期 ASYNC である必要があります。

### CAMO

クライアントの協力により、フェールオーバーシナリオで同じトランザクションの複数の挿入を防止する保護を有効にします。

詳細は、 *Commit At Most Once* を参照してください。

```
CAMO [ DEGRADE ON ( degrade_on_parameter = value ) TO ASYNC ]
```

#### DEGRADE ON パラメーター

タイムアウト時に非同期操作に低下することを許可します。

| Parameter | Type | Default | Description |
|---|---|---|---|
| *timeout* | interval | 0 | ミリ秒単位のタイムアウト他の単位を受け入れますこの後、操作は非同期になります。 0 は設定されていないことを意味します。 |

次のページに続く

表 137 – 前のページからの続き

| Parameter | Type | Default | Description |
|---|---|---|---|
| *require_write_lead* | Boolean | False | ノードが非同期モードに切り替えることができるように書き込みリードである必要があるかどうかを指定します。 |

### ラグコントロール

レプリケーションラグによって制御される動的なレート制限の構成が可能になります。

詳細は、 *Lag Control* を参照してください。

```
LAG CONTROL [ ( lag_control_parameter = value [, ... ] ) ]
```

### LAG CONTROL パラメーター

| Parameter | Type | Default | Description |
|---|---|---|---|
| *max_lag_size* | int | 0 | 特定のノードが別のノードへのレプリケーション接続で持つことができる KB 単位の最大遅延。遅延が `max_commit_delay`によってスケールされたこの最大値を超えると、遅延制御はコミット遅延を調整します。 |
| *max_lag_time* | interval | 0 | 特定のダウンストリームノードへのレプリケーション接続に関して、特定のオリジンが持つことができるミリ秒単位の最大レプリケーションラグ。 |
| *max_commit_delay* | interval | 0 | 各コミットにかかる最大遅延をミリ秒の小数単位で構成します。0 に設定すると、Lag Control が無効になります。各コミット遅延調整の後たとえば、レプリケーションが `max_lag_size`または `max_lag_time`よりも遅延している場合、コミット遅延は `bdr.lag_control_commit_delay_adjust` GUC の重みで再計算されます。`max_commit_delay`は、コミット遅延の上限です。 |

- `max_lag_size` および `max_lag_time` が 0 に設定されている場合、LAG CONTROL は無効になります。
- `max_commit_delay` が設定されていないか、0 に設定されている場合、LAG CONTROL は無効にな

ります。

遅延サイズは、walsender の send_ptr から受信者の apply_ptr までのデルタから導き出されます。

遅延時間は次の式に従って計算されます。

```
lag_time = (lag_size / apply_rate) * 1000;
```

lag_size は、send_ptr と apply_ptr max_lag_size に使用されるデルタであり、 apply_rate は、単純化された式に従う加重指数移動平均です。

```
apply_rate = prev_apply_rate * (1 - apply_rate_weight) +
             ((apply_ptr_diff * apply_rate_weight) / diff_secs);
```

そこで

- prev_apply_rate は、新しいレートを再計算する前に、以前に構成された apply_rate でした。
- apply_rate_weight は、GUC bdr.lag_tracker_apply_rate_weight の値です。
- apply_ptr_diff は、現在の apply_ptr と、適用レートが最後に計算された時点での apply_ptr の差です。
- diff_secs は、最後に適用レートが計算されてからの秒単位のデルタです。

### Conflicts

#### 競合検出

#### 競合タイプのリスト

PGD は、次の競合タイプを認識し、 conflict_type パラメーターとして使用できます。

| Conflict type | Description |
|---|---|
| *insert_exists* | 着信挿入は、主キーまたは一意キー/インデックスを介して既存の行と競合します。 |
| *update_differing* | 着信更新のキー行はローカル行とは異なります。これは、 行バージョンの競合検出 を使用する場合にのみ発生する可能性があります。 |
| *update_origin_change* | 着信更新は、別のノードによって最後に変更された行を変更しています。 |
| *update_missing* | 着信更新は、存在しない行を変更しようとしています。 |
| *update_recently_deleted* | 着信更新は、最近削除された行を変更しようとしています。 |
| *update_pkey_exists* | 着信更新により、`PRIMARY KEY`が変更され、変更を適用しているノードに既に存在する値が変更されました。 |
| *multiple_unique_conflicts* | 受信行は、ターゲットテーブルの UNIQUE / EXCLUDE インデックスごとに複数の行と競合します。 |
| *delete_recently_updated* | 現在のノードまたは 行バージョンの競合検出 を使用している場合、行の最新の更新よりも古いコミットタイムスタンプを含む着信削除。 |
| *delete_missing* | 着信削除は、存在しない行を削除しようとしています。 |
| *target_column_missing* | ターゲットテーブルには、受信行に存在する 1 つ以上の列がありません。 |

次のページに続く

表 139 – 前のページからの続き

| Conflict type | Description |
|---|---|
| *source_column_missing* | 着信行には、ターゲットテーブルに存在する 1 つ以上の列がありません。 |
| *target_table_missing* | ターゲットテーブルがありません。 |
| *apply_error_ddl* | レプリケートされた DDL コマンドを適用するときに Postgres によってエラーがスローされました。 |

### 競合の解決

ほとんどの競合は自動的に解決できます。PGD は、last-update-wins メカニズム、より正確には `update_if_newer` 競合リゾルバーにデフォルトで設定されます。このメカニズムでは、競合検出に使用される同じコミットタイムスタンプに基づいて、2 つの競合する行の最後に挿入または変更された行を保持します。特定のコーナーケースのシナリオでの動作は、`bdr.create_node_group` または `bdr.alter_node_group` に使用される設定によって異なります。

PGD では、次のファンクションを使用して競合解決のデフォルト動作をオーバーライドできます。

### 競合リゾルバーのリスト

PGD ではいくつかの競合リゾルバーが使用でき、処理できる競合タイプのカバレッジが異なります。

| Resolver | Description |
|---|---|
| *error* | エラーをスローし、レプリケーションを停止します。 |
| *skip* | リモート変更の処理をスキップし、次の変更でレプリケーションを続行します。 *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*、および `source_column_missing`競合タイプに使用できます。 |
| *skip_if_recently_dropped* | 最近ダウンストリームで 1 日以内にドロップされたため、ダウンストリームが存在しないテーブル用の場合、リモート変更をスキップします。それ以外の場合はエラーをスローします。`target_table_missing`競合タイプに使用できます。この競合リゾルバーは、同じ名前のテーブルが削除された後すぐに再作成される場合、課題を引き起こす可能性があります。その場合、ノードのいずれかが、テーブルを再作成する DDL を参照する前に、再作成されたテーブルの DML を参照する場合があります。次に、テーブルが最近ドロップされたと仮定して、リモートデータを誤ってスキップし、データ損失を引き起こします。このリゾルバーを使用する場合、ドロップされた直後にオブジェクト名を再利用しないことをお勧めします。 |
| *skip_transaction* | 競合を生成したトランザクション全体をスキップします。 |
| *update_if_newer* | リモート行が競合するローカル行よりも後にコミットされた場合に更新されます元ノードの壁時計によって決定されます。タイムスタンプが同じ場合、ノード ID はタイブレーカーとして使用され、すべてのノードで同じ行が選択されるように保証されます。より高いノード ID が勝ちます。 *insert_exists*、*update_differing*、*update_origin_change*、および`update_pkey_exists`競合タイプに使用できます。 |

次のページに続く

表 140 – 前のページからの続き

| Resolver | Description |
|---|---|
| *update* | レプリケートされたアクションを常に実行します。 *insert_exists* *INSERT*`を `*UPDATE*`に変更します 、 `*update_differing*、*update_origin_change*、*update_pkey_exists*、および `delete_recently_updated` 削除を実行します。 |
| *insert_or_skip* | オリジンから送信された利用可能な情報から新しい行を構築し、 INSERT しようとします。完全な行を作成するための十分な情報がない場合、変更をスキップします。 `update_missing`および `update_recently_deleted`競合タイプに使用できます。 |
| *insert_or_error* | オリジンから送信された利用可能な情報から新しい行を構築し、挿入しようとします。完全な行を構築するために利用できる十分な情報がない場合、エラーをスローし、レプリケーションを停止します。完全な行を構築するために利用できる十分な情報がない場合、エラーをスローし、レプリケーションを停止します。 `update_missing`および `update_recently_deleted`競合タイプに使用できます。 |
| *ignore* | 欠落しているターゲット列を無視して、処理を続行します。 `target_column_missing`競合タイプに使用できます。 |
| *ignore_if_null* | リモート行の追加列に NULL 値が含まれる場合、欠落しているターゲット列を無視します。それ以外の場合は、エラーをスローし、レプリケーションを停止します。 `target_column_missing`競合タイプに使用できます。 |
| *use_default_value* | 欠落している列値をデフォルト列のデフォルトの場合は NULL を含むで埋めて、処理を続行します。デフォルトまたは制約違反つまり、 NOT NULL 列の NULL デフォルトの処理中にエラーが発生すると、レプリケーションが停止されます。 `source_column_missing`競合タイプに使用できます。 |

`insert_exists` 、`update_differing` 、`update_origin_change` 、`update_missing` 、`multiple_unique_conflicts` 、`update_recently_deleted` 、`update_pkey_exists` 、`delete_recently_updated` 、および `delete_missing` 競合タイプは、*Stream triggers* を使用するユーザー定義ロジックでも解決できます。

このマトリックスは、各競合リゾルバーが処理できる競合タイプを示しています。

| | insert exists | update differing | update origin change | update missing | update recently deleted | update pkey exists | delete recently updated | delete missing | target column missing | source column missing | target table missing | multiple unique conflicts |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 recently dropped | | | | | | | | | | | X | |
| update if newer | X | X | X | | | X | | | | | | |
| update | X | X | X | | | X | X | | | | | X |
| insert or skip | | | | X | X | | | | | | | |
| insert or error | | | | X | X | | | | | | | |
| ignore | | | | | | | | | X | | | |
| ignore if null | | | | | | | | | X | | | |
| use default value | | | | | | | | | | X | | |
| conflict trigger | X | X | X | X | X | X | X | X | | | | X |

### デフォルトの競合リゾルバー

| 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 |
| multi-ple_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 |

### 競合解決のリスト

競合解決は、競合リゾルバーが選択した解決の種類を表し、競合を解決するために実行された特定のアクションに対応します。

次の競合解決は、`conflict_resolution` パラメーターで現在サポートされています。

| Resolution | Description |
|---|---|
| *apply_remote* | リモート着信行が適用されました。 |
| *skip* | 行の処理はスキップされましたローカルでの変更は行われませんでした。 |
| *merge* | リモート行とローカル行からの情報をマージして、新しい行が作成されました。 |
| *user* | ユーザーコード競合トリガーは、ターゲットテーブルに書き込まれる行を生成しました。 |

### 競合ログ

バージョン 6.0 以降、PGD はデフォルトで `bdr.conflict_history` テーブルに競合をログに記録しません。これは、テーブルが大きくなり、パフォーマンスの問題を発生させる可能性があるためです。bdr.alter_node_set_log_config ファンクションを使用して、競合ログを有効にできます。この機能を使用すると、ログに記録する競合を詳細に制御できます。または、次のようにすべての競合をログに記録するように設定できます。

```
SELECT bdr.alter_node_set_log_config(`nodename`, false, true, NULL, NULL);
```

名前付けノードでこのコマンドを実行して、その特定のノードでのすべての競合のログを有効にします。

すべてのノードでロギングを有効にする場合は、PGD グループの各ノードでこのコマンドを実行します。

### Conflict functions

#### bdr.alter_table_conflict_detection

テーブルの所有者は、特定のテーブルの競合検出がどのように動作するかを変更できます。

##### 概要

```
bdr.alter_table_conflict_detection(relation regclass,
                                   method text,
                                   column_name name DEFAULT NULL)
```

##### パラメーター

- relation —新しい競合検出方法を設定する関係の名前。
- method —使用する競合検出方法。
- column_name —列検出データの保存に使用する列。これはスキップできます。その場合、列名は競合検出方法に基づいて選択されます。row_origin メソッドは、メタデータストレージに追加の列を必要としません。

競合検出として認識されている方法は次のとおりです。

- row_origin —タプルに加えられた以前の変更のオリジン Origin conflict detection を参照してください。これは、テーブルに追加の列を必要としないサポートされている唯一の方法です。
- row_version —行バージョン列 Row version conflict detection を参照してください。
- column_commit_timestamp —列ごとのコミットタイムスタンプ *CLCD* で説明。
- column_modify_timestamp —列ごとの変更タイムスタンプ *CLCD* で説明

##### 注

column_commit_timestamp と column_modify_timestamp の競合検出方法の違いの詳細については、*Timestamps in column-level conflict resolution* を参照してください。

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *ddl filters* 構成の影響を受けることを意味します。

ファンクションは、列レベルの競合解決が有効になっているリレーションで DML グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに可視されます。

bdr.backwards_compatibility が 30618 以下に設定されていない限り、relation の所有者のみが bdr.alter_table_conflict_detection ファンクションを実行できます。

> **警告**
>
> 追加の列を使用してメタデータを保存する方法から競合検出方法を変更すると、その列がドロップされます。

> **警告**
>
> このファンクションは、*bdr.camo_enable_client_warnings* で警告が無効になっていない限り、CAMO を無効にし、警告を発します。

### bdr.alter_node_set_conflict_resolver

この機能は、特定のノードでの競合解決の動作を設定します。

#### 概要

```
bdr.alter_node_set_conflict_resolver(node_name text,
                                     conflict_type text,
                                     conflict_resolver text)
```

#### パラメーター

- node_name —変更されるノードの名前。
- conflict_type —設定を適用する競合タイプ競合タイプのリスト を参照してください。
- conflict_resolver —指定された競合タイプに使用するリゾルバー競合リゾルバーのリスト を参照してください。

#### 注

現在、ローカルノードのみを変更できます。ファンクション呼び出しはレプリケートされません。複数のノードの設定を変更する場合は、各ノードでファンクションを実行する必要があります。

このファンクションによる構成変更は、 bdr.create_node_group または bdr.alter_node_group で指定された競合解決のデフォルト動作をオーバーライドします。

このファンクションはトランザクションです。変更はロールバックできます。それらは現在のトランザクションに表示されます。

### bdr.alter_node_set_log_config

ノードの競合ログ構成を設定します。

### 概要

```
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)
```

### パラメーター

- node_name —変更されるノードの名前。
- log_to_file —ノードログファイルに記録するかどうか。
- log_to_table — bdr.conflict_history テーブルにログを記録するかどうか。
- conflict_type —ログに記録する競合タイプ。NULL デフォルトはすべてを意味します。
- conflict_resolution —ログへの競合の解決。NULL デフォルトはすべてを意味します。

### 注

ローカルノードのみを変更できます。ファンクション呼び出しはレプリケートされません。複数のノードの設定を変更する場合は、各ノードでファンクションを実行する必要があります。

このファンクションはトランザクションです。変更はロールバックできます。それらは現在のトランザクションに表示されます。

### 競合ログ構成のリスト

bdr.node_log_config ビューには、すべてのロギング構成が表示されます。ここには、ロギング構成の名前、ログの場所、およびログに記録される競合のタイプと解決がリストされています。

### 競合のテーブルへのロギング

log_to_table が true に設定されている場合、競合はテーブルにログに記録されます。競合ログのターゲットテーブルは bdr.conflict_history です。

このテーブルは列 local_time でレンジパーティション化されています。テーブルはオートパーティションで管理されています。デフォルトでは、新しいパーティションは毎日作成され、過去 1 か月の競合が維持されます。その後、古いパーティションは削除されます。自動パーティションは、事前に 7〜14 のパーティションを作成します。bdr_superuser はこれらのデフォルトを変更できます。

PGD によって管理されるすべてのテーブルで生成された競合はこのテーブルに記録されるため、正当なユーザーのみが競合したデータを読み取ることができるようにすることが重要です。PGD は、 bdr.conflict_history テーブルで ROW LEVEL SECURITY ポリシーを定義することによりこれを行います。テーブルの所有者のみが、各テーブルの競合を読み取ることができます。基になるテーブルに RLS ポリシーが定義、有効化、および適用されている場合、所有者でも競合を読み取ることはできません。FORCE オプションで作成された RLS ポリシーは、テーブルの所有者にも適用されます。その場合、基になるテーブルの一部またはすべての行は、所有者でも読み取ることができない場合があります。したがって、PGD は、競合ログテーブルにより厳密なポリシーも適用します。

事前定義されたロール bdr_read_all_conflicts は、bdr_superuser ロールを付与せずに、bdr.conflict_history テーブルに記録された競合の詳細をすべて表示する必要があるユーザーに付与できます。

デフォルトのロール bdr_read_all_stats は、bdr.conflict_history_summary というカタログビューにアクセスします。このビューにはユーザーデータが含まれないため、ログに記録された競合を監視できます。

### Replication set management

#### bdr.create_replication_set

この機能は、レプリケーションセットを作成します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

##### 概要

```
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)
```

##### パラメーター

- set_name —新しいレプリケーションセットの名前。PGD グループ全体で一意である必要があります。
- replicate_insert —このレプリケーションセット内のテーブルへの挿入をレプリケートするかどうかを示します。
- replicate_update —このレプリケーションセット内のテーブルの更新をレプリケートするかどうかを示します。
- replicate_delete —このレプリケーションセット内のテーブルからの削除をレプリケートするかどうかを示します。
- replicate_truncate —このレプリケーションセット内のテーブルのトランケートをレプリケートするかどうかを示します。
- autoadd_tables —新しく作成される将来のテーブルをこのレプリケーションセットにレプリケートするかどうかを示します
- autoadd_existing —既存のすべてのユーザーテーブルをこのレプリケーションセットに追加するかどうかを示します。このパラメーターは、autoadd_tables が true に設定されている場合にのみ効果があります。

#### 注

デフォルトでは、新しいレプリケーションセットは DDL または PGD 管理ファンクション呼び出しをレプリケートしません。レプリケーションセットの DDL レプリケーションを設定する方法については、*DDL filters* を参照してください。既存の DDL フィルターは、すべての DDL および管理ファンクション呼び出しをレプリケートするデフォルトのグループレプリケーションセットに設定されます。これは、グループの作成時に作成されますが、PGD グループのデフォルトレプリケーションセットで DDL または PGD 管理ファンクション呼び出しをレプリケートしたくない場合にドロップできます。

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* の影響を受けることを意味します

構成。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

### bdr.alter_replication_set

この機能は、既存のレプリケーションセットのオプションを変更します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

#### 概要

```
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)
```

#### パラメーター

- set_name —既存のレプリケーションセットの名前。
- replicate_insert —このレプリケーションセット内のテーブルへの挿入をレプリケートするかどうかを示します。
- replicate_update —このレプリケーションセット内のテーブルの更新をレプリケートするかどうかを示します。
- replicate_delete —このレプリケーションセット内のテーブルからの削除をレプリケートするかどうかを示します。
- replicate_truncate —このレプリケーションセット内のテーブルのトランケートをレプリケートするかどうかを示します。

- autoadd_tables —新しく作成される将来のテーブルをこのレプリケーションセットに追加するかどうかを示します。

NULL デフォルトに設定されているオプションは、以前と同じままです。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* の影響を受けることを意味します

構成。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

### bdr.drop_replication_set

この機能は、既存のレプリケーションセットを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

#### 概要

```
bdr.drop_replication_set(set_name name)
```

#### パラメーター

- set_name —既存のレプリケーションセットの名前。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *ddl filters* の影響を受けることを意味します

構成。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

**警告**

少なくとも別のノードで使用されているレプリケーションセットを削除しないでください。削除すると、そのノードでレプリケーションが停止します。これが発生した場合は、影響を受けるノードをそのレプリケーションセットからサブスクライブ解除します。同じ理由で、参加されるノードがそのレプリケーションセットのメンバーである場合、参加操作が進行中のレプリケーションセットをドロップしないでください。レプリケーションセットのメンバーシップは、結合の開始時にのみチェックされます。

これは、レプリケーションセットの使用状況に関する情報が各ノードにローカルであるため、グループに参加する前にノードで構成できるために発生します。

alter_node_replication_sets を使用して、ノードのレプリケーションセットのサブスクリプションを管理できます。

### bdr.alter_node_replication_sets

この機能は、ノードが公開およびサブスクライブするレプリケーションセットを変更します。

#### 概要

```
bdr.alter_node_replication_sets(node_name name,
                                set_names text[])
```

#### パラメーター

- node_name —変更するノード。現在はローカルノードである必要があります。
- set_names —指定されたノードにレプリケートするレプリケーションセットの配列。空の配列を使用すると、グループのデフォルトレプリケーションセットが使用されます。

#### 注

このファンクションはローカルノードでのみ実行され、どのような方法でもレプリケートされません。

このファンクションはノードが参加する前に実行するように設計されているため、リストされたレプリケーションセットの存在はチェックされません。エラーを回避するには、レプリケーションセット名を正しく指定するように注意してください。

この動作により、PGD グループの一部であるノードでだけでなく、まだグループに参加していないノードでもファンクションを呼び出すことができます。このアプローチでは、結合中に同期されるデータを制限します。ただし、スキーマは、レプリケーションセット設定に関係なく、常に完全に同期されます。レプリケーションセットで指定されたテーブルだけでなく、すべてのテーブルがコピーされます。bdr.tables カタログテーブルを参照して、不要なテーブルを削除できます。(これらは PGD の後のバージョンでは自動的に削除される可能性があります。これは現在 true です。 *DDL filters*

それ以外の場合、構成は DDL のレプリケーションを防止します。

この呼び出しの後にノードがサブスクライブするレプリケーションセットは、他のノードからこのファンクションが実行されるノードに変更を実際にレプリケートするために他のノードによって公開されます。

### Replication set membership

### bdr.replication_set_add_table

このファンクションは、テーブルをレプリケーションセットに追加します。

このファンクションは、テーブルをレプリケーションセットに追加し、ファンクションの呼び出しを含むトランザクションのコミットから変更のレプリケートを開始します。ノード上のテーブルが持っている既

存のデータは同期されません。このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

### 概要

```
bdr.replication_set_add_table(relation regclass,
                              set_name name DEFAULT NULL,
                              columns text[] DEFAULT NULL,
                              row_filter text DEFAULT NULL)
```

### パラメーター

- relation —テーブルの名前または OID。
- set_name —レプリケーションセットの名前。NULL デフォルトの場合、PGD グループのデフォルトレプリケーションセットが使用されます。
- columns —将来の使用のために予約されています。現在は何もしないため、NULL である必要があります。
- row_filter —レプリケートされた行のフィルタリングに使用する SQL 式。この式が定義されていない場合つまり、デフォルトの NULL に設定されている場合、すべての行が送信されます。

row_filter は、NULL を使用したブール結果を生成する式を指定します。True または Unknown と評価される式は、行をレプリケートします。False 値は行をレプリケートしません。式にサブクエリーを含めたり、レプリケートされる現在の行の列以外の変数を参照したりすることはできません。システム列を参照することはできません。

row_filter は、ターゲットノードではなく、オリジンノードで実行されます。これにより、この特定のテーブルのレプリケーションに追加の CPU オーバーヘッドがかかりますが、フィルターされた行のデータの送信が完全に回避されます。したがって、ネットワーク帯域幅が削減され、ターゲットノードのオーバーヘッドが適用されます。

row_filter は、特定のテーブルの TRUNCATE コマンドを削除することはありません。レプリケーションセットレベルで TRUNCATE コマンドをフィルタリングできます。

テーブルの一部の列のみをレプリケートできます。 差分を含むノード間のレプリケーションを参照してください。

### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* の影響を受けることを意味します

構成。

row_filter が NULL でない場合、ファンクションはレプリケーションセットに追加されるリレーションで DML グローバルロックを取得します。それ以外の場合は、DDL グローバルロックのみを使用します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

### bdr.replication_set_remove_table

このファンクションは、レプリケーションセットからテーブルを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

#### 概要

```
bdr.replication_set_remove_table(relation regclass,
                                 set_name name DEFAULT NULL)
```

#### パラメーター

- relation —テーブルの名前または OID。
- set_name —レプリケーションセットの名前。NULL デフォルトの場合、PGD グループのデフォルトレプリケーションセットが使用されます。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* の影響を受けることを意味します

構成。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

### DDL replication filtering

*DDL* レプリケーションフィルタリング も参照してください。

### bdr.replication_set_add_ddl_filter

このファンクションは、 DDL フィルタをレプリケーションセットに追加します。

指定されたフィルターに一致する DDL は、そのセットをサブスクライブしているノードにレプリケートされます。この機能は、PGD 管理機能のレプリケーションにも影響します。

この機能は、ノードでの DDL の実行を妨げるものではありません。DDL を他のノードにレプリケートするかどうかのみを変更します。2 つのノードの間に、すべてのインデックスコマンドを除外するレプリケーションフィルタがあるとします。各ノードに直接接続し、そのノードで目的の DDL を実行することにより、インデックスコマンドを自由に実行できます。

DDL フィルターは command_tag および role_name を指定して、一部の DDL ステートメントのみのレプリケーションを許可できます。command_tag は、 *event triggers* で使用されるものと同じです

通常の PostgreSQL コマンドの場合。典型的な例は、ロジカルスタンバイ上の追加のインデックスコマンドが他のすべてのノードにレプリケートされないようにフィルタを作成することです。

修飾されたファンクション名と一致するタグ名を使用して、使用される PGD 管理機能をフィルタリングできます。たとえば、bdr.replication_set_add_table は、同じ名前のファンクションのコマンドタグです。この場合、このタグにより、bdr.* を使用してすべての PGD ファンクションをフィルタリングできます。

role_name は、コマンドを実行している現在のロールとの照合に使用されます。command_tag と role_name は両方とも正規表現として評価され、大文字と小文字が区別されます。

### 概要

```
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)
```

### パラメーター

- set_name —レプリケーションセットの名前。NULL の場合、PGD グループのデフォルトのレプリケーションセットが使用されます。
- ddl_filter_name —DDL フィルタの名前。この名前は、PGD グループ全体で一意である必要があります。
- command_tag —一致するコマンドタグの正規表現。NULL は、すべてと一致することを意味します
- role_name —ロール名と一致するための正規表現。NULL は、すべてのロールと一致することを意味します
- base_relation_name —将来の使用のために予約されています。NULL である必要があります。
- query_match —クエリーと一致するための正規表現。NULL は、すべてのクエリーと一致することを意味します。
- exclusive —true の場合、他の一致したフィルタは考慮されませんつまり、排他的フィルタのみが適用されます。複数の排他的フィルタが一致する場合、エラーがスローされます。このパラメーターは、メインレプリケーションセットを介してデフォルトのレプリケーションを維持しながら、特定のコマンドを特定のレプリケーションセットにルーティングするのに役立ちます。

### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* の影響を受けることを意味します

構成。これは、DDL フィルタ構成の変更のレプリケーションが既存の DDL フィルタ構成の影響を受けることも意味します。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

定義されたレプリケーションフィルタを表示するには、ビュー bdr.ddl_replication を使用します。

#### 例

PGD 管理機能のみを含めるには、次のようなフィルターを定義します。

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, mygroup_admin, $$bdr\..*$$);
```

インデックス DDL を除くすべてを除外するには

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, index_filter,
                ^(?!(CREATE INDEX|DROP INDEX|ALTER INDEX)).*);
```

テーブルとインデックスに対するすべての操作を含め、その他のすべてを除外するには、テーブル用とインデックス用の 2 つのフィルターを追加します。この例は、複数のフィルタが、許可されたすべての DDL コマンドの和集合を提供することを示しています。

```
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

このファンクションは、レプリケーションセットから DDL フィルターを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

#### 概要

```
bdr.replication_set_remove_ddl_filter(set_name name,
                                      ddl_filter_name text)
```

#### パラメーター

- set_name —レプリケーションセットの名前。NULL の場合、PGD グループのデフォルトのレプリケーションセットが使用されます。
- ddl_filter_name —削除する DDL フィルターの名前。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* 構成の影響を受けることを意味します。これは、DDL フィルタ構成の変更のレプリケーションが既存の DDL フィルタ構成の影響を受けることも意味します。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

### Testing and tuning commands

EDB Postgres Distributed には、PGD クラスターのテストと調整に役立つツールが含まれています。バックグラウンドについては、Testing and tuning を参照してください。

### pgd_bench

#### 概要

EDB Postgres Distributed 展開のベンチマークツール。

```
pgd_bench [OPTION]... [DBNAME] [DBNAME2]
```

DBNAME は、次のフォーマットの conninfo 文字列です。"host=10.1.1.2 user=postgres dbname=master"

pgd_bench のオプションと使用例については、*pgd_bench in Testing and tuning* を参照してください。

#### オプション

pgd_bench コマンドは、pgbench コマンドのラッパーとして実装されています。これは、同じオプションの多くを共有し、テストを実行するときに pgbench という名前のテーブルを作成したことを意味します。

pgd_bench に固有のオプションには次のものが含まれます。

#### 設定モード

-m または--mode

モードは regular 、camo 、または failover に設定できます。デフォルトは regular です。

- regular — pgd_bench を実行するために必要なノードは 1 つだけです。
- camo —CAMO パートナーとして機能するように 2 番目のノードを指定する必要があります。CAMO の設定が必要です。
- failover —フェールオーバーとして機能するように 2 番目のノードを指定する必要があります。

-m failover を使用する場合、追加オプション--retry を使用できます。このオプションは、フェイルオーバーが発生した場合にトランザクションを再試行するように pgd_bench に指示します。-m camo を使用する場合、 --retry オプションは自動的に有効になります。

-m camo を使用し、カスタムスクリプトを提供する場合、スクリプト内の SQL コマンドを SQL トランザクションコマンドでラップする必要があります。つまり、最初の SQL コマンドは BEGIN であり、最後の SQL コマンドは COMMIT である必要があります。

### GUC 変数の設定

-o または--set-option このオプションの後には、NAME=VALUE エントリが続きます。これらは、pgd_bench が接続する各サーバー、およびそれらのサーバーにのみ Postgres SET コマンドを使用して適用されます。

他のオプションは Postgres pgbench コマンドと同じです。詳細については、PostgreSQL *pgbench* ドキュメントを参照してください。

オプション pgd_bench および pgbench の完全なリストが以下にあります。

### 初期化オプション

- -i, --initialize —初期化モードを呼び出します。
- -I, --init-steps=[dtgGvpf]+ (デフォルト"dtgvp" ) —選択した初期化ステップを実行します。
  - d —既存の pgbench tables.
  - t —を削除します。標準の pgbench シナリオで使用されるテーブルを作成します。
  - g —クライアントサイドでデータを生成し、標準テーブルにロードし、既に存在するデータを置き換えます。
  - G —サーバーサイドでデータを生成し、標準テーブルにロードし、既に存在するデータを置き換えます。
  - v —標準テーブルで VACUUM を呼び出します。
  - p —標準のテーブルに主キーインデックスを作成します。
  - f —標準テーブル間に外部キー制約を作成します。
- -F, --fillfactor=NUM —フィルファクターを設定します。
- -n, --no-vacuum —初期化中に VACUUM を実行しないでください。
- -q, --quiet —クワイエットロギング 5 秒ごとに 1 メッセージ。
- -s, --scale=NUM —スケーリング係数。
- --foreign-keys —テーブル間に外部キーコンストレインを作成します。
- --index-tablespace=TABLESPACE —指定されたテーブルスペースにインデックスを作成します。
- --partition-method=(range|hash) —このメソッドで pgbench_accounts をパーティション化します。デフォルトは range です。
- --partitions=NUM — pgbench_accounts を NUM パーツにパーティション化します。デフォルトは 0 です。
- --tablespace=TABLESPACE —指定されたテーブルスペースにテーブルを作成します。
- --unlogged-tables —ログを取らないテーブルとしてテーブルを作成します。注ログに記録されていないテーブルはレプリケートされません。

### 実行するものを選択するオプション

- -b, --builtin=NAME[@W] —W で重み付けされたビルトインスクリプト NAME を追加します。デフォルトは 1 です。-b list を使用して、使用可能なスクリプトをリストします。
- -f, --file=FILENAME[@W] — W で重み付けスクリプト FILENAME を追加します。デフォルトは 1 です。
- -N, --skip-some-updates —pgbench_tellers および pgbench_branches の更新。 -b simple-update と同じ。
- -S, --select-only — SELECT のみのトランザクションを実行します。-b select-only と同じ。

### ベンチマークオプション

- -c, --client=NUM —同時データベースクライアントの数。デフォルトは 1 です。
- -C, --connect —トランザクションごとに新しい接続を確立します。
- -D, --define=VARNAME=VALUE —カスタムスクリプトで使用する変数を定義します。
- -j, --jobs=NUM —スレッド数。デフォルトは 1 です。
- -l, --log —トランザクション時間をログファイルに書き込みます。
- -L, --latency-limit=NUM —NUM ミリ秒を超えて続いたトランザクションを遅いものとしてカウントします。
- -m, --mode=regular|camo|failover —pgbench を実行するモード。デフォルトは regular です。
- -M, --protocol=simple|extended|prepared —クエリーを送信するためのプロトコル。デフォルトは simple です。
- -n, --no-vacuum —テストの前に VACUUM を実行しないでください。
- -o, --set-option=NAME=VALUE —ランタイム SET オプションを指定します。
- -P, --progress=NUM —NUM 秒ごとにスレッドの進行状況レポートを表示します。
- -r, --report-per-command —コマンドごとのレイテンシー、失敗、および再試行。
- -R, --rate=NUM — 1 秒ごとのトランザクションのターゲットレート。
- -s, --scale=NUM —このスケール係数を出力で報告します。
- -t, --transactions=NUM —各クライアントが実行するトランザクションの数。デフォルトは 10 です。
- -T, --time=NUM —ベンチマークテストの時間秒単位。
- -v, --vacuum-all —テストの前に 4 つの標準テーブルをすべてバキュームします。
- --aggregate-interval=NUM — NUM 秒以上のデータ。
- --failures-detailed —基本的なタイプごとにグループ化された障害を報告します。
- --log-prefix=PREFIX —トランザクション時間ログファイルの接頭辞。デフォルトは pgbench_log です。

- --max-tries=NUM —トランザクションを実行しようとする最大回数。デフォルトは 1 です。
- --progress-timestamp —進行状況には Unix エポックタイムスタンプを使用します。
- --random-seed=SEED —ランダムシード time 、rand 、integer を設定します。
- --retry —フェールオーバー時にトランザクションを再試行します。 -m とともに使用されます。
- --sampling-rate=NUM —ログに記録するトランザクションの割合、たとえば、1%の場合 0.01。
- --show-script=NAME —ビルトインスクリプトコードを表示して、終了します。
- --verbose-errors —すべてのエラーのメッセージを出力します。

#### 共通オプション

- -d, --debug —デバッグ出力を印刷します。
- -h, --host=HOSTNAME —データベースサーバホストまたはソケットディレクトリ。
- -p, --port=PORT —データベースサーバーのポート番号。
- -U, --username=USERNAME —指定されたデータベースユーザーとして接続します。
- -V, --version —バージョン情報を出力して終了します。
- -?, --help —ヘルプを表示して、終了します。

### Global sequence management interfaces

PGD は、標準の PostgreSQL シーケンスと PGD グローバルシーケンスの間で変換するためのインターフェイスを提供します。

次のファンクションは DDL であると考えられるため、DDL レプリケーションとグローバルロックが適用されます。

#### シーケンスファンクション

#### bdr.alter_sequence_set_kind

シーケンスの所有者がシーケンスの種類を設定できます。設定すると、seqkind は bdr.sequences ビューを介してのみ可視されます。他のすべての方法では、シーケンスは通常のシーケンスとして表示されます。

PGD はこのファンクションを DDL として扱うため、現在アクティブな場合、DDL レプリケーションとグローバルロックが適用されます。 ステートメント固有の *DDL* レプリケーションの問題を参照してください。

#### 概要

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text, start bigint DEFAULT NULL)
```

#### パラメーター

- seqoid —変更するシーケンスの名前または Oid。
- seqkind —標準の PostgreSQL シーケンスの場合は local 、グローバルに一意の PGD シーケンスの場合は snowflakeid または galloc 、または古いグローバル一意シーケンスの場合は timeshard 。
- start —galloc およびローカルシーケンスの新しい開始点を指定できます。

#### 注

シーケンス種類を galloc に変更すると、そのシーケンスの最初に割り当てられたレンジは、開始点としてシーケンス start 値を使用します。galloc に変更される前にシーケンスで使用された既存の値が存在する場合、次のコマンドを使用して、新しく生成された値が既存の値と競合しないように、開始ポイントを移動することをお勧めします。

```
ALTER SEQUENCE seq_name START starting_value RESTART
```

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *DDL filters* の影響を受けることを意味します

構成。

ファンクションはグローバル DDL ロックを取得します。また、シーケンスをローカルにロックします。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

bdr.backwards_compatibility が 30618 以下に設定されていない限り、シーケンスの所有者のみが bdr.alter_sequence_set_kind ファンクションを実行できます。

### bdr.extract_timestamp_from_snowflakeid

このファンクションは、snowflakeid シーケンスのタイムスタンプコンポーネントを抽出します。戻り値の型は timestamptz です。

#### 概要

```
bdr.extract_timestamp_from_snowflakeid(snowflakeid bigint)
```

#### パラメーター

- snowflakeid — snowflakeid シーケンスの値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.extract_nodeid_from_snowflakeid

このファンクションは、snowflakeid シーケンスの nodeid コンポーネントを抽出します。

#### 概要

```
bdr.extract_nodeid_from_snowflakeid(snowflakeid bigint)
```

#### パラメーター

- snowflakeid — snowflakeid シーケンスの値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.extract_localseqid_from_snowflakeid

このファンクションは、snowflakeid シーケンスのローカルシーケンス値コンポーネントを抽出します。

#### 概要

```
bdr.extract_localseqid_from_snowflakeid(snowflakeid bigint)
```

#### パラメーター

- snowflakeid — snowflakeid シーケンスの値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.timestamp_to_snowflakeid

このファンクションは、タイムスタンプ値をダミーの snowflakeid シーケンス値に変換します。

これは、 snowflakeid 列および特定のタイムスタンプのインデックス付き検索または値の比較を行う場合に役立ちます。

たとえば、 snowflakeid シーケンスを使用している列 id を持つテーブル foo があると、次のように昨日の午前 0 時からの変更数を取得できます。

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_snowflakeid(yesterday)
```

この方法で定式化された照会は、列 id のインデックススキャンを使用します。

#### 概要

```
bdr.timestamp_to_snowflakeid(ts timestamptz)
```

#### パラメーター

- ts — snowflakeid シーケンス生成に使用するタイムスタンプ。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.extract_timestamp_from_timeshard

このファンクションは、timeshard シーケンスのタイムスタンプコンポーネントを抽出します。戻り値の型は timestamptz です。

#### 概要

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

#### パラメーター

- timeshard_seq — timeshard シーケンスの値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.extract_nodeid_from_timeshard

このファンクションは、timeshard シーケンスの nodeid コンポーネントを抽出します。

#### 概要

```
bdr.extract_nodeid_from_timeshard(timeshard_seq bigint)
```

#### パラメーター

- timeshard_seq — timeshard シーケンスの値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.extract_localseqid_from_timeshard

このファンクションは、timeshard シーケンスのローカルシーケンス値コンポーネントを抽出します。

#### 概要

```
bdr.extract_localseqid_from_timeshard(timeshard_seq bigint)
```

#### パラメーター

- timeshard_seq — timeshard シーケンスの値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.timestamp_to_timeshard

このファンクションは、タイムスタンプ値をダミーの timeshard シーケンス値に変換します。

これは、 timeshard 列および特定のタイムスタンプのインデックス付き検索または値の比較を行う場合に役立ちます。

たとえば、 timeshard シーケンスを使用している列 id を持つテーブル foo があると、次のように昨日の午前 0 時からの変更数を取得できます。

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_timeshard(yesterday)
```

この方法で定式化された照会は、列 id のインデックススキャンを使用します。

#### 概要

```
bdr.timestamp_to_timeshard(ts timestamptz)
```

#### パラメーター

- ts — timeshard シーケンス生成に使用するタイムスタンプ。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.galloc_chunk_info

このファンクションは、ローカルノードで galloc シーケンスに割り当てられたレンジを取得します。

シーケンスがローカルノードでまだアクセスされていない場合、空の結果セットが返されます。

指定されたシーケンス名が galloc シーケンスでない場合、 ERROR が発生します。

#### 概要

```
bdr.galloc_chunk_info(seqname regclass)
```

#### パラメーター

- seqname - 照会する galloc シーケンスの名前

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### KSUUID v2 ファンクション

KSUUID v2 データ、K-Sortable UUID データを操作するためのファンクション。 *KSUUID in the sequences documentation* も参照してください。

### bdr.gen_ksuuid_v2

このファンクションは、引数として渡されたタイムスタンプの値、または NULL が渡された場合は現在のシステム時間を使用して、新しい KSUUID v2 値を生成します。システム時間を使用して KSUUID を自動的に生成する場合は、NULL 引数を渡します。

戻り値の型は UUID です。

#### 概要

```
bdr.gen_ksuuid_v2(timestamptz)
```

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.ksuuid_v2_cmp

このファンクションは、KSUUID v2 値を比較します。

最初の値の方が新しい場合は 1、2 番目の値が小さい場合は-1、または等しい場合は 0 を返します。

#### 概要

```
bdr.ksuuid_v2_cmp(uuid, uuid)
```

#### パラメーター

- UUID — KSUUID v2 と比較します。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.extract_timestamp_from_ksuuid_v2

このファンクションは、KSUUID v2 のタイムスタンプコンポーネントを抽出します。戻り値の型は timestamptz です。

#### 概要

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

#### パラメーター

- UUID — KSUUID タイムスタンプを抽出する v2 値。

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### KSUUID v1 ファンクション

KSUUID v1 データ、K-Sortable UUID データ v1 を操作するためのファンクション。非推奨 - 詳細については、*KSUUID in the sequences documentation* を参照してください。

### bdr.gen_ksuuid

このファンクションは、現在のシステム時間を使用して、新しい KSUUID v1 値を生成します。戻り値の型は UUID です。

#### 概要

```
bdr.gen_ksuuid()
```

#### 注

このファンクションは、ローカルノードでのみ実行されます。

### bdr.uuid_v1_cmp

このファンクションは、KSUUID v1 値を比較します。

最初の値の方が新しい場合は 1、2 番目の値が小さい場合は-1、または等しい場合は 0 を返します。

#### 概要

```
bdr.uuid_v1_cmp(uuid, uuid)
```

#### 注

このファンクションは、ローカルノードでのみ実行されます。

#### パラメーター

- UUID — KSUUID v1 と比較します。

### bdr.extract_timestamp_from_ksuuid

このファンクションは、KSUUID v1 または UUIDv1 値のタイムスタンプコンポーネントを抽出します。戻り値の型は timestamptz です。

#### 概要

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

#### パラメーター

- UUID — KSUUID タイムスタンプを抽出する v1 値。

#### 注

このファンクションはローカルノードで実行されます。

### Autopartition

オートパーティションを使用すると、テーブルを複数のパーティションに分割できます。詳細については、*Autopartitioning* を参照してください。

### bdr.autopartition

bdr.autopartition ファンクションは、テーブルの自動 RANGE パーティションを構成します。

#### 概要

```
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,
```

(次のページに続く)

(前のページからの続き)

```
        analytics_offload_period
        drop_after_retention_period);
```

### パラメーター

- relation —テーブルの名前または OID。
- partition_increment —次のパーティション作成までのインターバルまたはインクリメント。
- partition_initial_lowerbound —テーブルにパーティションがない場合、この下限と partition_increment 離れた上限を持つ最初のパーティションが作成されます。
- partition_autocreate_expression —新しいパーティションを作成する時期が来たかどうかを検出するために使用される式。
- minimum_advance_partitions —システムは、常に少なくとも minimum_advance_partitions パーティションを持とうとします。
- maximum_advance_partitions —事前パーティションの数が minimum_advance_partitions を下回った後に、一度に作成するパーティションの数。
- data_retention_period —古いパーティションが削除されるまでの間隔定義されている場合。この値は migrate_after_period より大きい必要があります。
- enabled —アクティビティを無効または一時停止し、後で再開または再有効化できます。
- analytics_offload_period —パーティションのオフロードのサポートを提供します。将来の使用のために予約されています。
- drop_after_retention_period —パーティションをドロップする代わりにデタッチできるようにします。ドロップする代わりにデタッチするには、これを false に設定します。

### 例

日次パーティション、データを 1 か月間保持します

```
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);
```

あと 2 つのパーティションのみが残っている場合、5 つの高度なパーティションを作成します。各パーティションには、10 億の注文を保持できます。

```
bdr.autopartition(Orders, 1000000000,
        partition_initial_lowerbound := 0,
        minimum_advance_partitions := 2,
```

(次のページに続く)

(前のページからの続き)

```
        maximum_advance_partitions := 5
    );
```

### bdr.drop_autopartition

bdr.drop_autopartition() を使用して、特定のリレーションの自動パーティションルールを削除します。リレーションの保留中の作業項目はすべて削除され、新しい作業項目は作成されません。

```
bdr.drop_autopartition(relation regclass);
```

#### パラメーター

- relation —テーブルの名前または OID。

### bdr.autopartition_wait_for_partitions

パーティションの作成は非同期プロセスです。AutoPartition は、ローカルまたはすべてのノードでパーティションが作成されるのを待機する一連の機能を提供します。

bdr.autopartition_wait_for_partitions() を使用して、ローカルノードでパーティションの作成を待機します。ファンクションは、パーティションテーブル名とパーティションキー列値を取得し、その値を保持するパーティションが作成されるまで待機します。

このファンクションは、パーティションがローカルに作成されるのをのみ待機します。パーティションがリモートノードにも存在することは保証されません。

すべての PGD ノードでパーティションが作成されるのを待機するには、bdr.autopartition_wait_for_partitions_on_all_nodes() ファンクションを使用します。このファンクションは、ローカルノードとすべてのリモートノードを内部的にチェックし、パーティションがどこでも作成されるまで待機します。

#### 概要

```
bdr.autopartition_wait_for_partitions(relation regclass, upperbound text);
```

#### パラメーター

- relation —テーブルの名前または OID。
- upperbound —パーティションキー列の値。

### bdr.autopartition_wait_for_partitions_on_all_nodes

#### 概要

```
bdr.autopartition_wait_for_partitions_on_all_nodes(relation regclass, upperbound␣
↪text);
```

#### パラメーター

- relation —テーブルの名前または OID。
- upperbound —パーティションキー列の値。

### bdr.autopartition_find_partition

bdr.autopartition_find_partition() ファンクションを使用して、指定されたパーティションキー値のパーティションを見つけます。その値を保持するパーティションが存在しない場合、ファンクションは NULL を結果ます。それ以外の場合、パーティションの Oid が返されます。

#### 概要

```
bdr.autopartition_find_partition(relname regclass, searchkey text);
```

#### パラメーター

- relname —パーティション分割テーブルの名前。
- searchkey —検索するパーティションキー値。

### bdr.autopartition_enable

bdr.autopartition_enable を使用して、指定されたテーブルで AutoPartitioning を有効にします。AutoPartitioning が既に有効になっている場合、アクションは発生しません。特定のテーブルで自動パーティションを無効にするには、:ref:``bdr.autopartition_disable` <*bdr.autopartition_disable*>` を参照してください。

#### 概要

```
bdr.autopartition_enable(relname regclass);
```

#### パラメーター

- relname — AutoPartitioning を有効にするリレーションの名前。

### bdr.autopartition_disable

bdr.autopartition_disable を使用して、指定されたテーブルで自動パーティションを無効にします。AutoPartitioning が既に無効になっている場合、アクションは発生しません。

#### 概要

```
bdr.autopartition_disable(relname regclass);
```

#### パラメーター

- relname — AutoPartitioning を無効にする関係の名前。

#### 内部ファンクション

### bdr.autopartition_create_partition

AutoPartition は、内部ファンクション bdr.autopartition_create_partition を使用して、親テーブルにスタンドアロンの AutoPartition を作成します。

#### 概要

```
bdr.autopartition_create_partition(relname regclass,
                                   partname name,
                                     lowerb text,
                                     upperb text,
                                     nodes oid[]);
```

#### パラメーター

- relname —アタッチする親テーブルの名前または Oid。
- partname —新しい AutoPartition の名前。
- lowerb —パーティションの下限。
- upperb —パーティションの上限。
- nodes —新しいパーティションが存在するノードのリスト。このパラメーターは PGD の内部であり、将来の使用のために予約されています。

#### 注

これは、AutoPartition がパーティション管理に使用する内部機能です。このファンクションを直接使用しないことをお勧めします。

### bdr.autopartition_drop_partition

AutoPartition は、内部ファンクション bdr.autopartition_drop_partition を使用して、データ保持ポリシーに従って、不要になったパーティションを削除します。パーティションテーブルのドロップが成功した場合、ファンクションは true を結果ます。

#### 概要

```
bdr.autopartition_drop_partition(relname regclass)
```

#### パラメーター

- `relname` —ドロップするパーティションテーブルの名前。

#### 注

このファンクションは、選択したパーティションテーブルで `DROP TABLE` を使用する前に、親テーブルに DDL ロックを配置します。この機能は、AutoPartition がパーティション管理に使用する内部ファンクションです。このファンクションを直接使用しないことをお勧めします。

### Stream triggers reference

> **参考**
>
> ストリームトリガーの概要については、 Stream Triggers 。

conflict triggers と transform triggers は、トリガー API が提供する事前定義された変数と PGD が提供する追加情報ファンクションを介して、行とメタデータに関する情報にアクセスします。

PL/pgSQL では、次の事前定義された変数とファンクションを使用できます。

- *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*

ストリームトリガーの作成とドロップは、操作インターフェイスを介して管理されます。

- *Stream triggers manipulation interfaces*
- *bdr.create_conflict_trigger*
- *bdr.create_transform_trigger*
- *bdr.drop_trigger*

### Stream triggers manipulation interfaces

ストリームトリガーは、 `REPLICA IDENTITY FULL` があるテーブル、または `TOAST` が適用される列のないテーブルにのみ作成できます。

### bdr.create_conflict_trigger

このファンクションは、新しい競合トリガーを作成します。

#### 概要

```
bdr.create_conflict_trigger(trigger_name text,
                            events text[],
                            relation regclass,
                            function regprocedure,
                            args text[] DEFAULT {})
```

#### パラメーター

- trigger_name —新しいトリガーの名前。
- events —このトリガーを起動するイベントの配列。有効な値は’INSERT ‘、’UPDATE ‘、および’DELETE ’です。
- relation —このトリガーを発射する関係。
- function —実行するファンクション。
- args —オプショナル。トリガーファンクションが実行時に受け取るパラメーターの配列 TG_ARGV 変数の内容を指定します。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *ddl filters* 構成の影響を受けることを意味します。

ファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

通常の PostgreSQL トリガーと同様に、bdr.create_conflict_trigger ファンクションには、relation の TRIGGER 特権とファンクションの EXECUTE 特権が必要です。これは、bdr.backwards_compatibility が 30619 以上の場合に適用されます。PGD では、追加のセキュリティルールが競合トリガーを含むすべてのトリガーに適用されます。*Security and roles* を参照してください。

### bdr.create_transform_trigger

このファンクションは、変換トリガーを作成します。

#### 概要

```
bdr.create_transform_trigger(trigger_name text,
                             events text[],
                             relation regclass,
```

(次のページに続く)

(前のページからの続き)

```
                                function regprocedure,
                                args text[] DEFAULT {})
```

#### パラメーター

- trigger_name —新しいトリガーの名前。
- events —このトリガーを起動するイベントの配列。有効な値は’INSERT ‘、’UPDATE ‘、および’DELETE ’です。
- relation —このトリガーを発射する関係。
- function —実行するファンクション。
- args —オプショナル。トリガーファンクションが実行時に受け取るパラメーターの配列 TG_ARGV 変数の内容を指定します。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *ddl filters* 構成の影響を受けることを意味します。

ファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

通常の PostgreSQL トリガーと同様に、bdr.create_transform_trigger ファンクションには、relation の TRIGGER 特権とファンクションの EXECUTE 特権が必要です。PGD では、追加のセキュリティルールが変換トリガーを含むすべてのトリガーに適用されます。*Security and roles* を参照してください。

### bdr.drop_trigger

このファンクションは、既存のストリームトリガー競合と変換の両方を削除します。

#### 概要

```
bdr.drop_trigger(trigger_name text,
                 relation regclass,
                 ifexists boolean DEFAULT false)
```

#### パラメーター

- trigger_name —既存のトリガーの名前。
- relation —トリガーが定義されている関係。
- ifexists — true に設定されている場合、このファンクションは不足しているトリガーを無視します。

#### 注

このファンクションは、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *ddl filters* 構成の影響を受けることを意味します。

ファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

このファンクションはトランザクションです。トランザクションの ROLLBACK を使用してエフェクトをロールバックできます。変更は現在のトランザクションに表示されます。

relation の所有者のみが bdr.drop_trigger ファンクションを実行できます。

### Stream triggers row functions

#### bdr.trigger_get_row

このファンクションは、識別子で指定されたトリガー行の内容を RECORD として結果ます。このファンクションは、不適切に呼び出された場合、つまり操作タイプ TG_OP が DELETE のときに SOURCE_NEW で呼び出された場合、NULL を結果ます。

#### 概要

```
bdr.trigger_get_row(row_id text)
```

#### パラメーター

- row_id —行の識別子。トリガータイプと操作に応じて、SOURCE_NEW 、SOURCE_OLD 、および TARGET のいずれかです。個々のトリガータイプの説明を参照してください。

#### bdr.trigger_get_committs

このファンクションは、識別子で指定されたトリガー行のコミットタイムスタンプを結果ます。行が固定されているか使用できないために使用できない場合、NULL を返します。行識別子 SOURCE_OLD の場合は、常に NULL を結果ます。

#### 概要

```
bdr.trigger_get_committs(row_id text)
```

#### パラメーター

- row_id —行の識別子。トリガータイプと操作に応じて、SOURCE_NEW 、SOURCE_OLD 、および TARGET のいずれかです。個々のトリガータイプの説明を参照してください。

#### bdr.trigger_get_xid

このファンクションは、識別子で指定された TARGET 行のローカルトランザクション ID を結果ます。行が固定されているか使用できないために使用できない場合、NULL を返します。 SOURCE_OLD および SOURCE_NEW 行識別子の場合は、常に NULL を結果ます。

競合トリガーでのみ使用できます。

#### 概要

```
bdr.trigger_get_xid(row_id text)
```

#### パラメーター

- row_id —行の識別子。トリガータイプと操作に応じて、SOURCE_NEW 、SOURCE_OLD 、および TARGET のいずれかです。個々のトリガータイプの説明を参照してください。

### bdr.trigger_get_type

このファンクションは、現在のトリガータイプ、CONFLICT または TRANSFORM を結果ます。ストリームトリガーの外部で呼び出された場合、null を返します。

#### 概要

```
bdr.trigger_get_type()
```

### bdr.trigger_get_conflict_type

このファンクションは、競合トリガー内で呼び出された場合、現在の競合タイプを結果ます。それ以外の場合は、NULL を返します。

Conflict types を参照

このファンクションの考えられる戻り値については、

#### 概要

```
bdr.trigger_get_conflict_type()
```

### bdr.trigger_get_origin_node_id

このファンクションは、引数として渡されたトリガー row_id の起点に対応するノード ID を結果ます。オリジンが有効でない場合、行がローカルに生成されたことを意味します、トリガー行引数に応じて、ソースまたはターゲットノードのノード ID を結果ます。行識別子 SOURCE_OLD の場合は、常に NULL を結果ます。この機能を使用して、競合トリガーを定義して、常に信頼できるソースノードを優先できます。

#### 概要

```
bdr.trigger_get_origin_node_id(row_id text)
```

#### パラメーター

- row_id —行の識別子。トリガータイプと操作に応じて、SOURCE_NEW 、SOURCE_OLD 、および TARGET のいずれかです。個々のトリガータイプの説明を参照してください。

### bdr.ri_fkey_on_del_trigger

BEFORE トリガーとして呼び出される場合、このファンクションは FOREIGN KEY 情報を使用して FK 異常を回避します。

#### 概要

```
bdr.ri_fkey_on_del_trigger()
```

### Stream triggers row variables

#### TG_NAME

データ型の名前。この変数には、実際に起動されたトリガーの名前が含まれています。実際のトリガー名には、トリガーの作成中に指定された名前と比較して、_bdrt または_bdrc サフィックストリガータイプにより異なります。

#### TG_WHEN

データ型テキスト。この変数は、競合トリガーと変換トリガーの両方で BEFORE を示します。 bdr.trigger_get_type() 情報ファンクションを呼び出すことにより、ストリームトリガータイプを取得できます。*bdr.trigger_get_type* を参照してください。

#### TG_LEVEL

データ型 text ROW の文字列。

#### TG_OP

データタイプ text トリガーが起動されたオペレーションを識別する INSERT 、UPDATE 、または DELETE の文字列。

#### TG_RELID

データ型 oid トリガー呼び出しの原因となったテーブルのオブジェクト ID。

#### TG_TABLE_NAME

データ型名 トリガーの呼び出しを発生させたテーブルの名前。

#### TG_TABLE_SCHEMA

データ型名トリガー呼び出しの原因となったテーブルのスキーマの名前。パーティションテーブルの場合、これはルートテーブルの名前です。

#### TG_NARGS

データ型整数 bdr.create_conflict_trigger() または bdr.create_transform_trigger() ステートメントのトリガーファンクションに指定された引数の数。

#### TG_ARGV[]

テキストのデータ型配列 bdr.create_conflict_trigger() または bdr.create_transform_trigger() ステートメントの引数。インデックスは 0 からカウントします。無効なインデックス 0 未満または TG_NARGS 以上の場合、 NULL 値になります。

### Internal catalogs and views

カタログとビューは、アルファベット順にここにリストされています。

### bdr.autopartition_partitions

自動パーティション機能によって作成されたパーティションに関する情報を保存する内部カタログテーブル。

#### bdr.autopartition_partitions 列

| Name | Type | Description |
|---|---|---|
| ap_parent_relid | oid | リレーションの OID |
| ap_part_relname | name | 作成したリレーションの名前 |
| ap_part_created_at | timestamp with time zone | 作成タイムスタンプ |
| ap_part_migrated_at | timestamp with time zone | 移行のタイムスタンプ |
| ap_part_dropped_at | timestamp with time zone | ドロップされたときのタイムスタンプ |

### bdr.autopartition_rules

自動パーティションルールに関する情報を保存する内部カタログテーブル。

#### bdr.autopartition_rules 列

| Name | Type | Description |
|---|---|---|
| ap_partition_relid | oid | NA |
| ap_partition_relname | name | NA |
| ap_partition_schemaname | name | NA |
| ap_partition_increment_kir | "char" | NA |
| ap_secondary_tablespace | oid | NA |
| ap_maximum_advance_par | integer | NA |

次のページに続く

表 145 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| ap_is_autoscaled | boolean | NA |
| ap_latest_partitions | integer | NA |
| ap_enabled | boolean | NA |
| ap_migrate_after_period | interval | NA |
| ap_data_retention_period | interval | NA |
| ap_last_triggered | timestamp with time zone | NA |
| ap_partition_increment_val | text | NA |
| ap_partition_autocreate_ex | text | NA |
| ap_partition_initial_lowerb | text | NA |
| ap_partition_last_upperbou | text | NA |
| ap_partition_min_upperbou | text | NA |

### bdr.ddl_epoch

DDL エポックごとの状態を保持する内部カタログテーブル。

#### bdr.ddl_epoch 列

| Name | Type | Description |
|---|---|---|
| ddl_epoch | int8 | 単調増加のエポック番号 |
| origin_node_id | oid | このエポックの作成を要求したノードの内部ノード ID |
| epoch_consume_timeout | timestamptz | このエポックのタイムアウト |
| epoch_consumed | boolean | ローカルノードがエポックを完全に処理するとすぐに true に切り替えます |
| epoch_consumed_lsn | boolean | ローカルノードがエポックを処理した LSN |

### bdr.event_history

特定の PGD ノードのクラスターメンバーシップイベントを追跡する内部カタログテーブル。具体的には、以下を追跡します。

- ノードがクラスターに参加します
- Raft の状態が変化するときは常に、ノードがコンセンサスプロトコルでのロールを変更するたびに、リーダー、フォロワー、または候補からリーダーへ。*Monitoring Raft* コンセンサス を参照
- ワーカーがエラーアウトしたとき bdr.workers を参照してください

および *PGD* レプリケーションワーカーのモニタリング )

#### bdr.event_history 列

| Name | Type | Description |
|---|---|---|
| event_node_id | oid | イベントが参照するノードの ID |
| event_type | int | イベントのタイプノード、raft、またはワーカー関連イベント |
| event_sub_type | int | イベントのサブタイプ、つまり、参加、状態変更、またはエラーであるかどうか |
| event_source | text | イベントがソースされたワーカープロセスの名前 |
| event_time | timestamptz | イベントが発生したタイムスタンプ |
| event_text | text | イベントのテキスト表現たとえば、ワーカーのエラー |
| event_detail | text | イベントのより詳細な説明今のところ、ワーカーエラーにのみ関連します |

### bdr.event_summary

より人に優しい形式で情報を表示する bdr.event_history カタログのビュー。具体的には、イベントタイプとサブタイプを整数ではなくテキスト表現として表示します。

### bdr.local_leader_change

これは、リーダー変更履歴の最近の部分のローカルキャッシュです。これは、bdr.leader と同じフィールドがありますが、現在のバージョンのみを追跡するマップではなく、node_group_id、leader_kind、generate の順序セットである点が異なります。

### bdr.node_config

ノードごとの構成オプションを含む内部カタログテーブル。

#### bdr.node_config 列

| Name | Type | Description |
|---|---|---|
| node_id | oid | ノード ID |
| node_route_priority | int | このノードに割り当てられた優先度 |
| node_route_fence | boolean | このノードをフェンスするように切り替えます |
| node_route_writes | boolean | 書き込みを許可するスイッチ |
| node_route_reads | boolean | 読み取りを許可するようにスイッチ |
| node_route_dsn | text | このノードのインターフェイス |

### bdr.node_config_summary

より人間が読みやすい形式で情報を表示する bdr.node_config カタログのビュー。

#### bdr.node_config_summary 列

| Name | Type | Description |
|---|---|---|
| node_name | text | このノードの名前 |
| node_id | oid | ノード ID |
| node_route_priority | int | このノードに割り当てられた優先度 |
| node_route_fence | boolean | このノードをフェンスするように切り替えます |
| node_route_writes | boolean | 書き込みを許可するスイッチ |
| node_route_reads | boolean | 読み取りを許可するようにスイッチ |
| node_route_dsn | text | このノードのインターフェイス |
| effective_route_dsn | text | このノードの完全な DSN |

### bdr.node_group_config

ノードごとのグループ構成オプションを含む内部カタログテーブル。

#### bdr.node_group_config 列

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | ノードグループ ID |
| route_writer_max_lag | bigint | 受け入れられる最大書き込み遅延 |
| route_reader_max_lag | bigint | 受け入れられる最大読み取り遅延 |
| route_writer_wait_flush | boolean | フラッシュを待機する必要がある場合にスイッチ |

### bdr.node_group_routing_config_summary

ノードグループごとのルーティング構成オプション。

#### bdr.node_group_routing_config_summary 列

| Name | Type | Description |
|---|---|---|
| node_group_name | name | ノードグループ名 |
| location | name | ノードグループのロケーション |
| enable_routing | boolean | グループルーティングは有効になっていますか？ |
| node_group_type | text | ノードグループタイプ"global"、"data"、または"subscriber-only"のいずれか |
| route_writer_max_lag | bigint | 受け入れられる最大書き込み遅延 |
| route_reader_max_lag | bigint | 受け入れられる最大読み取り遅延 |
| route_writer_wait_flush | boolean | フラッシュを待機する |

### bdr.node_group_routing_info

接続マネージャーの現在のルーティング情報を保持する内部カタログテーブル。

#### bdr.node_group_routing_info 列

| Name | Type | Description |
|---|---|---|
| node_group_id | oid | ノードグループ ID。 |
| write_node_id | oid | 現在の書き込みノード。 |
| prev_write_node_id | oid | 以前の書き込みノード。 |
| read_node_ids | oid[] | 読み取り専用ノード ID のリスト。 |
| record_version | bigint | レコードバージョン。ルーティングレコードが変更されるたびに 1 増加します。 |
| record_ts | timestamptz | Record_version の最後の更新のタイムスタンプ。 |
| write_leader_version | bigint | リーダーバージョンを書き込みます。 write_node_id が変更されるたびに record_version からコピーされます。 |
| write_leader_ts | timestamptz | 書き込みリーダーのタイムスタンプ。 write_node_id が変更されるたびに record_ts からコピーされます。 |
| read_nodes_version | bigint | ノードバージョンの読み取り。 read_node_ids リストが変更されるたびに、record_version からコピーされます。 |
| read_nodes_ts | timestamptz | ノードのタイムスタンプを読み取ります。 read_node_ids リストが変更されるたびに record_tw からコピーされます。 |

### bdr.node_group_routing_summary

情報をより分かりやすく表示する bdr.node_group_routing_info カタログのビュー。

#### bdr.node_group_routing_summary 列

| Name | Type | Description |
|---|---|---|
| node_group_name | name | ノードグループ名 |
| write_lead | name | 現在の書き込みリード |
| previous_write_lead | name | 以前の書き込みリード |
| read_nodes | name[] | 現在の読み取り専用ノード |

### bdr.node_routing_config_summary

ノードごとのルーティング構成オプションのわかりやすいビュー。oid および短いフィールド名ではなく、ノード名を表示します。

#### bdr.node_routing_config_summary 列

| Name | Type | Description |
|---|---|---|
| node_name | name | ノード名 |
| route_priority | int | このノードに割り当てられた優先度 |
| route_fence | boolean | このノードをフェンスするように切り替えます |
| route_writes | boolean | 書き込みを許可するスイッチ |
| route_reads | boolean | 読み取りを許可するようにスイッチ |
| route_dsn | text | このノードのインターフェイス |

### bdr.sequence_kind

各非ローカルシーケンスのタイプを格納する内部状態テーブル。診断にはビュー bdr.sequences をお勧めします。

#### bdr.sequence_kind 列

| Name | Type | Description |
|---|---|---|
| seqid | oid | シーケンスの内部 OID |
| seqkind | char | 内部シーケンス種類 `l`=local,`t`=timeshard,`s`=snowflakeid,`g`=galloc |

### bdr.sync_node_requests

ノード同期要求の状態を保存する内部状態テーブル。ビュー *bdr.sync_node_requests_summary* は、このテーブルの人間が判読可能な表現形式を提供します。

#### bdr.sync_node_requests 列

| Name | Type | Description |
|---|---|---|
| sn_origin_node_id | oid | 同期する変更がある使用不可ノード |
| sn_target_node_id | oid | 元のノードの変更を含むノード |
| sn_source_node_id | oid | 同期要求のターゲットノード |
| sn_sync_start_lsn | pg_lsn | 同期要求の開始 LSN |
| sn_sync_start_ts | timestamptz | 同期要求の開始タイムスタンプ |

次のページに続く

表 156 – 前のページからの続き

| Name | Type | Description |
|---|---|---|
| sn_sync_end_lsn | pg_lsn | 同期要求の終了 LSN |
| sn_sync_end_ts | timestamptz | 同期要求の終了タイムスタンプ |
| sn_sync_status | text | 同期要求のステータス |

### bdr.sync_node_requests_summary

基になる *bdr.sync_node_requests* の人が判読可能なバージョンを提供するビュー

テーブル。

#### bdr.sync_node_requests_summary 列

| Name | Type | Description |
|---|---|---|
| origin | text | 同期する変更がある使用不可ノード |
| source | text | 元のノードの変更を含むノード |
| target | text | 同期要求のターゲットノード |
| sync_start_lsn | pg_lsn | 同期要求の開始 LSN |
| sync_start_ts | timestamptz | 同期要求の開始タイムスタンプ |
| sync_end_lsn | pg_lsn | 同期要求の終了 LSN |
| sync_end_ts | timestamptz | 同期要求の終了タイムスタンプ |
| sync_status | text | 同期要求のステータス |

### Internal system functions

以下は内部システム機能です。多くは、さまざまなビューを作成するときに使用されます。ファンクションを直接使用せず、代わりにそれらが提供するビューを使用することをお勧めします。

#### 一般的な内部ファンクション

### bdr.bdr_get_commit_decions

共有メモリの状態を検査する便利なルーチン。

#### 概要

```
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

トランザクションのコミットステータスを共有メモリハッシュテーブルに保存します。この命令は、アップグレードスクリプトが bdr.node_pre_commit カタログに保存されたコミット決定を共有メモリハッシュテーブルに転送するために使用されます。トランザクションのコミットステータスは WAL に記録されるため、WAL からリロードできます。

#### 概要

```
bdr.bdr_track_commit_decision(OID, xid, xid, "char", timestamptz, boolean);
```

### bdr.consensus_kv_fetch

一貫した KV ストアから JSON 形式で値を取得します。

#### 概要

```
bdr.consensus_kv_fetch(IN key text) RETURNS jsonb
```

#### パラメーター

| Parameter | Description |
|---|---|
| *key* | 取得する任意のキー。 |

#### 注

このファンクションは内部ファンクションであり、主に HARP によって使用されます。

> **警告**
>
> ユーザアプリケーションではこのファンクションを使用しないでください。

### bdr.consensus_kv_store

一貫した KV ストアに値を保存します。

値の有効期限のタイムスタンプを返します。この機能は `ttl` に依存しています。 `ttl` が `NULL` の場合、このファンクションは `infinity` を結果ます。値が削除された場合、`-infinity` を返します。

#### 概要

```
bdr.consensus_kv_store(key text, value jsonb,
        prev_value jsonb DEFAULT NULL, ttl int DEFAULT NULL)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *key* | 挿入、更新、または削除する任意の一意のキー。 |
| *value* | 保存する JSON 値。 NULL の場合、既存のレコードは削除されます。 |
| *prev_value* | 設定されている場合、書き込み操作は、現在の値が`prev_value`と等しい場合にのみ実行されます。 |
| *ttl* | 新しい値の有効期間ミリ秒単位。 |

#### 注

これは内部ファンクションであり、主に HARP によって使用されます。

> **警告**
>
> ユーザアプリケーションではこの機能を使用しないでください。

### bdr.decode_message_payload

コンセンサスメッセージのペイロードをより人間が読みやすい出力にデコードする PGD メッセージペイロードファンクション。主に `bdr.global_consensus_journal_details` デバッグビューによって使用されます。

### bdr.decode_message_response_payload

コンセンサスメッセージへの応答のペイロードをより人間が読みやすい出力にデコードする PGD メッセージペイロードファンクション。主に `bdr.global_consensus_journal_details` デバッグビューによって使用されます。

### bdr.difference_fix_origin_create

引数として渡された指定された名前を使用して、レプリケーションオリジンを作成しますが、`bdr_` 接頭辞を追加します。オリジンの内部 ID を結果ます。このファンクションは、`pg_replication_origin_create()` と同じ機能がありますが、このファンクションは postgres スーパーユーザー権限ではなく bdr_superuser を必要とする点を除きます。

### bdr.difference_fix_session_reset

現在のセッションをオリジンから再生しないものとしてマークし、本質的に `bdr.difference_fix_session_setup()` のエフェクトをリセットします。void を返します。このファンクションは、`pg_replication_origin_session_reset()` と同じ機能がありますが、このファンクションは postgres スーパーユーザー権限ではなく bdr_superuser を必要とする点を除きます。

#### 概要

```
bdr.difference_fix_session_reset()
```

### bdr.difference_fix_session_setup

現在のセッションを現在のオリジンから再生するものとしてマークします。ファンクションは、事前に作成された bdr_local_only_origin ローカルレプリケーションオリジンをセッションに暗黙的に使用します。これにより、再生の進行状況を報告でき、void を返します。このファンクションは、pg_replication_origin_session_setup() と同じ機能がありますが、このファンクションは postgres スーパーユーザー権限ではなく bdr_superuser を必要とする点が異なります。ファンクションの以前の形式 bdr.difference_fix_session_setup(text) は非推奨であり、将来のリリースでは削除される予定です。

#### 概要

```
bdr.difference_fix_session_setup()
```

### bdr.difference_fix_xact_set_avoid_conflict

現在のトランザクションを、LSN ' 0/0' およびタイムスタンプ' 2000-01-01' でコミットしたトランザクションを再生するものとしてマークします。このファンクションは、pg_replication_origin_xact_setup('0/0', '2000-01-01') と同じ機能がありますが、このファンクションは postgres スーパーユーザー権限ではなく bdr_superuser を必要とする点を除きます。

#### 概要

```
bdr.difference_fix_xact_set_avoid_conflict()
```

### bdr.drop_node

ノードのメタデータを削除します。

ノードが PARTED になると、そのメタデータはクラスターのノードに存在したままになります。たとえば、ノードがドロップされるまで、ノードは PARTED としてマークされ、bdr.node_sumary 結果に残ります。

bdr.drop_node('some node', force := true) の呼び出しは、分離中にノードがスタックした場合に必要かつ適切です。そのノードにまだ同期する必要があるデータがある場合、ドロップされるノードからのデータの同期をスキップすることに注意してください。ノードスタックパーティングが既に再イメージ化または削除されている場合、それに対して bdr.drop_node を呼び出して害はありません。これは、クラスター内のすべてのノードでこのスタックしたノードに対して呼び出す必要があることに注意して、ノードがドロップされたという一貫したビューを持つようにします。

このファンクションは、特定のノードのメタデータをローカルデータベースから削除します。ノードは次のいずれかです。

- ローカルノード。この場合、リモートノードに関する情報を含むすべてのノードのメタデータを削除します。
- リモートノード。その場合、その特定のノードのメタデータのみを削除します。

この内部機能の使用は次の場合に制限されます。

- EDB テクニカルサポートから指示された場合。
- ドキュメントで特に指示されている場所。

bdr.part_node を使用して、PGD グループからノードを削除します。このファンクションは、ノードを PARTED 状態に設定し、ノード名の再利用を有効にします。

```
bdr.drop_node(node_name text, cascade boolean DEFAULT false, force boolean DEFAULT␣
↪false)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_name* | 既存のノードの名前。 |
| *cascade* | 非推奨。将来のリリースで削除される予定です。 |
| *force* | 不一致を引き起こす可能性のある危険にもかかわらず、すべてのサニティチェックを回避し、指定された PGD ノードのすべてのメタデータの削除を強制します。テクニカルサポートのみが、分離に関連する緊急の場合に強制ノードドロップを使用します。 |

#### 注

このファンクションを実行する前に、bdr.part_node() を使用してノードを分割します。

このファンクションは、特定のノードのメタデータをローカルデータベースから削除します。ノードはローカルノードになることができ、その場合、リモートノードに関する情報を含むすべてのノードのメタデータが削除されます。または、リモートノードの場合、その特定のノードのメタデータのみが削除されます。

**注釈**

各ノードには、スノーフレーク ID および timeshard シーケンスで使用する一意のシーケンス番号が割り当てられているため、一度に最大 1024 のノードレコード ACTIVE と PARTED の両方を持つことができます。 PARTED ノードは自動的にクリーンアップされません。これが問題になる場合は、この機能を使用してこれらのレコードを削除できます。

### bdr.get_global_locks

ローカルノードで保持されているグローバルロックに関する情報を表示します。

bdr.global_locks ビューを実装して、ロックのより詳細な概要を提供するために使用されます。

### bdr.get_node_conflict_resolvers

ローカルノード上のすべての競合リゾルバーのテキスト文字列を表示します。

### bdr.get_slot_flush_timestamp

特定のレプリケーションスロットの最後のフラッシュ位置確認のタイムスタンプを取得します。

bdr.node_slots ビューを実装するために内部で使用されます。

### bdr.internal_alter_sequence_set_kind

さまざまなファンクション呼び出しのレプリケーションのために内部で使用されていたファンクション。PGD の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためのみに存在します。

### bdr.internal_replication_set_add_table

さまざまなファンクション呼び出しのレプリケーションのために内部で使用されていたファンクション。PGD の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためのみに存在します。

### bdr.internal_replication_set_remove_table

さまざまなファンクション呼び出しのレプリケーションのために内部で使用されていたファンクション。PGD の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためのみに存在します。

### bdr.internal_submit_join_request

新しいノードに参加するためのコンセンサス要求を送信します。

PGD グループの再構成の内部メカニズムに必要です。

### bdr.isolation_test_session_is_blocked

グローバルロック上のブロックのチェックを追加して、元の pg_isolation_test_session_is_blocked を拡張および実際に呼び出すヘルパーファンクション。

分離/同時実行テストに使用されます。

### bdr.local_node_info

PGD グループの再構成の内部メカニズムに必要なローカルノードの情報を表示します。

ビュー bdr.local_node_summary は、ユーザーの使用に役立つ同様の情報を提供します。

### bdr.msgb_connect

別のノードの接続プーラーに接続します。コンセンサスプロトコルによって使用されます。

### bdr.msgb_deliver_message

メッセージを別のノードの接続プーラーに送信します。コンセンサスプロトコルによって使用されます。

### bdr.node_catchup_state_name

name のキャッチアップ状態コードを変換します。

#### 概要

```
bdr.node_catchup_state_name(catchup_state oid);
```

#### パラメーター

| Parameter | Description |
|---|---|
| *catchup_state* | キャッチアップ状態の OID コード。 |

### bdr.node_kind_name

ノードの種類の人にわかりやすい名前データ|スタンバイ|ウィットネス|サブスクライバーのみを返します。

### bdr.peer_state_name

ノード状態 `node_state` をテキスト表現に変換します。主に `bdr.node_summary` ビューを実装するために使用されます。

### bdr.pg_xact_origin

指定されたトランザクションのオリジン ID を返します。

#### 概要

```
bdr.pg_xact_origin(xmin xid)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *xid* | オリジンが返されるトランザクション ID。 |

### bdr.request_replay_progress_update

’リプレイ進行状況の更新’Raft メッセージの即時書き込みを要求します。主にテスト目的で使用されますが、コンセンサスメカニズムが機能しているかどうかをテストするために使用することもできます。

### bdr.reset_relation_stats

bdr.stat_relation によって表示される、リレーション統計をリセットした後、ブール結果を返します。

### bdr.reset_subscription_stats

bdr.stat_subscription によって表示される、サブスクリプションによって作成された統計をリセットした後、ブール結果を返します。

### bdr.resynchronize_table_from_node

リモートノードから関係を再同期します。

#### 概要

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *node_name* | リレーションデータをコピーまたは再同期するノード。 |
| *relation* | リモートノードからのコピーに関する関係。 |

#### 注

このファンクションは、リレーションのグローバル DML ロックを取得し、リレーションをローカルで切り捨て、リモートノードからデータをコピーします。

リレーションは、同じ名前と定義を持つ両方のノードに存在する必要があります。

以下がサポートされています。

- 同一のパーティション定義を使用したパーティションテーブルの再同期
- パーティション化されたテーブルと非パーティションテーブルへの再同期、およびその逆の再同期
- 外部キー制約を一時的に削除および再作成することによる参照先テーブルの再同期

参照されるテーブルでファンクションを実行した後、参照される列データが参照列値と一致しない場合、ファンクションはエラーをスローします。参照テーブルデータを再同期した後、ファンクションを再実行します。

さらに、リモートノードからデータをコピーした後、生成された列の値をローカルに計算することにより、生成された列とテーブルの再同期をサポートします。

現在、 row_filters はこのファンクションによって無視されます。

bdr.resynchronize_table_from_node ファンクションは、テーブルの所有者が bdr_superuser 権限を持っている場合にのみ実行できます。

### bdr.seq_currval

グローバルシーケンス操作の内部実装の一部。

galloc または snowflakeid シーケンスで currval() が呼び出されるときに、自動的に呼び出されます。

### bdr.seq_lastval

グローバルシーケンス操作の内部実装の一部。

galloc または snowflakeid シーケンスで lastval() が呼び出されるときに、自動的に呼び出されます。

### bdr.seq_nextval

グローバルシーケンスインクリメントの内部実装の一部。

galloc または snowflakeid シーケンスで nextval() が呼び出されるときに自動的に呼び出されます

### bdr.show_subscription_status

サブスクリプションのステータスに関する情報を取得します。主に bdr.subscription_summary ビューを実装するために使用されます。

### bdr.show_workers

bdr ワーカーに関連する情報。

#### 概要

```
bdr.show_workers(
    worker_pid int,
    worker_role int,
    worker_role_name text,
    worker_subid oid)
```

### bdr.show_writers

bdr.writers ビューで使用されるファンクション。

### bdr.sync_status_name

同期状態コードをテキスト表現に変換します。主に *bdr.sync_node_requests_summary* を実装するために使用されます

ビュー。

#### 概要

```
bdr.sync_status_name(sync_state oid)
```

#### パラメーター

| Parameter | Description |
|---|---|
| *sync_state* | 同期状態の OID コード。 |

### タスクマネージャー機能

### bdr.taskmgr_set_leader

指定された node をタスクマネージャーリーダーノードになるように要求します。リーダーノードは、新しいタスクを作成します。現在、自動パーティションのみがこの機能を利用します。監視ノード、ロジカルスタンバイ、またはサブスクライバ専用ノードはリーダーになることはできません。このような要求はエラーで失敗します。

#### 概要

```
bdr.taskmgr_set_leader(node name, wait_for_completion boolean DEFAULT true);
```

### bdr.taskmgr_get_last_completed_workitem

クラスター内のすべてのノードで正常に完了した最後の作業項目の id を返します。

#### 概要

```
bdr.taskmgr_get_last_completed_workitem();
```

### bdr.taskmgr_work_queue_check_status

タスクを生成および終了するジョブを実行しているバックグラウンドワーカーのステータスを確認できます。

ステータスは次のビューから確認できます。

- bdr.taskmgr_work_queue_local_status
- bdr.taskmgr_work_queue_global_status

#### 概要

```
bdr.taskmgr_work_queue_check_status(workid bigint
                            local boolean DEFAULT false);
```

#### パラメーター

| Parameter | Description |
|---|---|
| *workid* | タスクのキー。 |
| *local* | ローカルステータスのみを確認します。 |

#### 注

Taskmgr ワーカーは、`bdr.autopartition` ファンクションが初めて呼び出される前でも、常にバックグラウンドで実行されています。無効な `workid` が使用される場合、ファンクションは `unknown` を結果ます。`In-progress` は一般的なステータスです。

### bdr.get_min_required_replication_slots

PGD-CLI によって使用される内部ファンクション。

### bdr.get_min_required_worker_processes

PGD-CLI によって使用される内部ファンクション。

### bdr.stat_get_activity

ビュー `bdr.stat_activity` の基になる内部ファンクション。直接使用しないでください。代わりに bdr.stat_activity ビューを使用します。

### bdr.worker_role_id_name

ビュー `bdr.worker_tasks` を生成するときに使用される内部ヘルパーファンクション。直接使用しないでください。代わりに bdr.worker_tasks ビューを使用します。

### bdr.lag_history

ビュー `bdr.node_replication_rates` を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.node_replication_rates ビューを使用します。

### bdr.get_raft_instance_by_nodegroup

ビュー `bdr.group_raft_details` を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_raft_details ビューを使用します。

### bdr.monitor_camo_on_all_nodes

ビュー `bdr.group_camo_details` を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_camo_details ビューを使用します。

#### bdr.monitor_raft_details_on_all_nodes

ビュー bdr.group_raft_details を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_raft_details ビューを使用します。

#### bdr.monitor_replslots_details_on_all_nodes

ビュー bdr.group_replslots_details を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_replslots_details ビューを使用します。

#### bdr.monitor_subscription_details_on_all_nodes

ビュー bdr.group_subscription_summary を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_subscription_summary ビューを使用します。

#### bdr.monitor_version_details_on_all_nodes

ビュー bdr.group_versions_details を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_versions_details ビューを使用します。

#### bdr.node_group_member_info

ビュー bdr.group_raft_details を生成するときに使用される内部ファンクション。直接使用しないでください。代わりに bdr.group_raft_details ビューを使用します。

#### Column-level conflict functions

#### bdr.column_timestamps_create

このファンクションは、列レベルの競合解決を作成します。column_timestamp_enable 内で呼び出されます。

##### 概要

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestampstz)
```

##### パラメーター

- p_source — 2 つのオプションは current または commit です。
- p_timestamp —タイムスタンプは、選択したソースによって異なります。commit の場合は TIMESTAMP_SOURCE_COMMIT 。 current の場合は TIMESTAMP_SOURCE_CURRENT 。

### 6.3.23 EDB Postgres Distributed Command Line Interface (PGD CLI)

EDB Postgres Distributed コマンドラインインターフェイス PGD CLI は、EDB Postgres 分散クラスターを管理するためのツールです。これは、クラスターリソースを検査および管理するための重要なツールです。

EDB Postgres 分散クラスターに対してコマンドを実行して、次のことを行うことができます。

- クラスターの状態を判断し、クラスターの構成を検査し、クラスターのリソースを管理します。

- クラスターのノードとグループを検査および管理します。
- グループに対して書き込みリーダー変更操作を実行します。

PGD クラスターに接続できる Linux および macOS システムに手動でインストールすることもできます。

- HCP の高度な分散高可用性クラスター。
- CloudNative Postgres Global Clusters オペレーターを使用して展開された PGD クラスター。
- 手動で展開された PGD クラスター。

### Installing PGD CLI

PGD CLI は、PGD クラスターに接続できる任意のシステムにインストールできます。Linux および macOS は、PGD CLI のインストールが現在サポートされているプラットフォームです。

### Installing PGD CLI on Linux

PGD CLI は、ほとんどの Linux ディストリビューションで利用できます。EDB アカウントでアクセスできる EDB リポジトリからインストールできます。PGD ユーザーと EDB Cloud Service ユーザー無料トライアルを含むユーザーは、EDB アカウントと PGD CLI にアクセスします。

#### EDB サブスクリプショントークンを取得する

これらのリポジトリからのダウンロードを有効にするには、トークンが必要です。トークンを取得するには、EDB Repos 2.0 にログインします。 EDB Repos 2.0 ページに初めてアクセスする場合は、 **Request Access** を選択してトークンを生成する必要があります。生成されたトークンが利用できるようになったら、**Copy** アイコンを選択してクリップボードにコピーするか、目のアイコンを選択して表示します。

#### EDB_SUBSCRIPTION_TOKEN 環境変数を設定する

トークンを取得したら、オペレーティングシステムに表示されているコマンドを実行して、<your-token> トークンを置き換えます。

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

次に、オペレーティングシステムに適切なコマンドを実行します。

#### Debian または Ubuntu

Debian または Ubuntu では、 apt パッケージマネージャーを使用して PGD CLI をインストールできます。

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

このコマンドが curl: (22) The requested URL returned error: 404 のようなエラーを返した場合、正しいトークンを入力したことを確認してください。

コマンドが成功すると、次のような出力が表示されます。

```
Executing the  setup script for the enterprisedb/postgres_distributed repository ...
...
```

次のコマンドを使用して PGD CLI パッケージをインストールできるようになりました。

```
sudo apt-get install edb-pgd6-cli
```

### RHEL、Rocky、AlmaLinux、または Oracle Linux

RHEL、Rocky、AlmaLinux、または Oracle Linux では、 yum パッケージマネージャーを使用して PGD CLI をインストールできます。 RHEL 8 以降のデフォルトパッケージマネージャーである dnf パッケージマネージャーを使用することもできます。

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

このコマンドが curl:  (22) The requested URL returned error:  404 のようなエラーを返した場合、正しいトークンを入力したことを確認してください。

コマンドが成功すると、次のような出力が表示されます。

```
Executing the  setup script for the enterprisedb/postgres_distributed repository ...
...
```

次のコマンドを使用して PGD CLI パッケージをインストールできるようになりました。

```
sudo dnf install edb-pgd6-cli
```

```
sudo yum install edb-pgd6-cli
```

### Installing PGD CLI on macOS

PGD CLI は、macOS で Homebrew フォーミュラとして利用できます。インストールするには、次のコマンドを実行します。

```
brew tap enterprisedb/tap
brew install pgd-cli
```

インストールを確認するには、次を実行します。

```
pgd --version
```

*Using PGD CLI*

## Using PGD CLI

### PGD CLI とは何ですか?

PGD CLI は、PGD クラスターに接続して管理する便利な方法です。これを使用するには、PGD スーパーユーザー権限または同等のユーザーが必要です。スーパーユーザー権限を持つ PGD ユーザーは *Security Best Practices* です。同等のユーザーの例は、EDB Cloud Service 分散高可用性クラスターの edb_admin です。

- *.pgpass* にエントリを追加します。これには、ホスト、ポート、データベース名、ユーザー名、パスワードが含まれます。
- PGPASSWORD 環境変数にパスワードを設定する。
- 接続文字列にパスワードを含めます。

他のオプションは複数のデータベースでうまく拡張できないか、パスワードの機密性が危険にさらされるため、最初のオプションをお勧めします。

installed pgd-cli を取得したら、 pgd コマンドを実行して PGD コマンドラインインターフェイスにアクセスします。pgd コマンドには、ユーザー名とパスワードとともに、接続するホスト、ポート、データベースに関する詳細が必要です。

### データベース接続文字列を渡す

--dsn フラグを使用して、データベース接続文字列を pgd コマンドに渡します。 --dsn フラグを使用して接続文字列を渡す場合、構成ファイルは必要ありません。構成ファイルが存在する場合でも、フラグが優先されます。例

```
pgd nodes list --dsn "host=bdr-a1 port=5432 dbname=pgddb user=enterprisedb"
```

データベース接続文字列 DSN は、 PGD_CLI_DSN 環境変数を使用して設定することもできます。例

```
export PGD_CLI_DSN="host=bdr-a1 port=5432 dbname=pgddb user=enterprisedb"
pgd nodes list
```

--dsn フラグは環境変数より優先されるため、両方が設定されている場合、--dsn 値が使用されます。

コマンドオプションの説明については、 PGD CLI Command reference を参照してください。

### 構成ファイルの指定

pgd-cli-config.yml ファイルが/etc/edb/pgd-cli または$HOME/.edb/pgd-cli にある場合、pgd はそれを使用します。オプションの-f または--config-file フラグを使用して、この動作をオーバーライドできます。例

```
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
```

### 出力形式の指定

-o または--output フラグを使用して、デフォルトの出力形式を JSON に変更します。例

```
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"
  }
]
```

PGD CLI は、次の出力形式をサポートしています。

| Setting | Format | Description |
|---|---|---|
| *simple* | Tabular | 単純な表形式ビュー。 (デフォルト)。 |

次のページに続く

表 166 – 前のページからの続き

| Setting | Format | Description |
|---|---|---|
| *json* | JSON | フォーマットなしの生データを表示します。一部のコマンドでは、JSON 出力には、追加のフィールドや詳細なメッセージなど、表形式出力より多くのデータが表示される場合があります。 |
| *psql* | PSQL | PSQL 出力のスタイルの表形式ビュー。フォーマット。 |
| *modern* | Tabular | ボックス文字を使用してテーブルを区切る表形式ビュー。 |
| *markdown* | Markdown | `pgd assess`などの一部のコマンドの長い形式の非表形式出力を生成するマークダウンスタイルの出力。 |

### コマンドラインヘルプへのアクセス

サポートされているコマンドをリストするには、次を入力します。

```
pgd --help
```

特定のコマンドとそのパラメーターのヘルプが必要な場合は、`pgd <command_name> --help` と入力してください。例

```
pgd nodes list --help
```

### Configuring PGD CLI

PGD CLI は、PGD クラスターに接続できる任意のシステムにインストールできます。PGD CLI を使用するには、PGD スーパーユーザー権限または同等のユーザーが必要です。スーパーユーザー権限を持つ PGD ユーザーは *Security Best Practices* です。同等のユーザーの例は、EDB Cloud Service 分散高可用性クラスターの edb_admin です。

### PGD CLI とデータベース接続文字列

データベース接続文字列は必要ない場合があります。たとえば、Trused Postgres Architect はシステムに PGD CLI をインストールすると、PGD クラスターへの接続も構成します。これは、 PGD CLI が実行時にクラスターに接続できることを意味します。

PGD CLI を手動でインストールする場合は、 PGD CLI にデータベース接続文字列を指定して、どの PGD クラスターに接続するかを認識する必要があります。

- *.pgpass* にエントリを追加します。これには、ホスト、ポート、データベース名、ユーザー名、パスワードが含まれます。- `PGPASSWORD` 環境変数にパスワードを設定する。 - 接続文字列にパスワードを含めます。

他のオプションは複数のデータベースでうまく拡張できないか、パスワードの機密性が危険にさらされるため、最初のオプションをお勧めします。

その情報を入手したら、続行できます。

### 接続するデータベースの構成

PGD CLI は、 PGD CLI 構成ファイルまたはコマンドラインからデータベース接続情報を取得します。

### コマンドラインでデータベース接続文字列を使用する

--dsn オプションを使用して、接続文字列を pgd に直接渡すことができます。詳細については、データベース接続文字列を渡す を参照してください。例

```
pgd --dsn "host=kaboom port=5432 user=enterprisedb dbname=pgddb" nodes show --versions
```

### 環境変数でデータベース接続文字列を使用する

コマンドラインで接続文字列を渡す別の方法として、PGD_CLI_DSN 環境変数を接続文字列に設定できます。例

```
export PGD_CLI_DSN="host=kaboom port=5432 user=enterprisedb dbname=pgddb"
pgd nodes show --versions
```

### 構成ファイルを使用する

pgd-cli-config.yml 構成ファイルを使用して、クラスターのデータベース接続文字列を指定します。構成ファイルには、クラスター内の少なくとも 1 つの PGD ノードのデータベース接続文字列が含まれている必要があります。クラスター名はオプショナルであり、検証されません。

例

```
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"
```

デフォルトでは、pgd-cli-config.yml は/etc/edb/pgd-cli ディレクトリにあります。 PGD CLI は、次の場所で pgd-cli-config.yml を検索します。優先順位は高から低です。

1. /etc/edb/pgd-cli デフォルト 2.$HOME/.edb/pgd-cli

構成ファイルがこれらのディレクトリのいずれにもない場合は、 pgd コマンドでオプションの-f または--config-file フラグを使用して、構成として読み取るようにファイルを設定できます。データベース接続文字列を渡す を参照してください。

### Discovering connection strings

PGD CLI は、PGD クラスターに接続できる任意のシステムにインストールできます。PGD CLI を使用するには、PGD スーパーユーザー権限または同等のユーザーが必要です。スーパーユーザー権限を持つ PGD ユーザーは *Security Best Practices* です。同等のユーザーの例は、EDB Cloud Service 分散高可用性クラスターの edb_admin です。

### PGD CLI とデータベース接続文字列

データベース接続文字列は必要ない場合があります。たとえば、Trused Postgres Architect はシステムに PGD CLI をインストールすると、PGD クラスターへの接続も構成します。これは、 PGD CLI が実行時にクラスターに接続できることを意味します。

### データベース接続文字列の取得

PGD はさまざまな構成をサポートするため、すべての展開方法には、接続文字列を取得する異なる方法があります。通常、展開の構成から必要な情報を取得できます。次に、その情報を接続文字列にアセンブルできます。

### EDB CloudNative Postgres グローバルクラスターで展開されたクラスターの場合

EDB CloudNative Postgres グローバルクラスター CNPG-GC を使用している場合、接続文字列は展開の構成から取得されます。非常に柔軟なため、接続文字列を取得する方法は複数あります。これは、主に、デプロイメントの services の構成に依存します。

・ノードサービステンプレートを使用する場合、各ノードとプロキシサービスへの直接接続が利用できます。

- グループサービステンプレートを使用する場合、各グループへのゲートウェイサービスがあります。
- プロキシサービステンプレートを使用する場合、単一のプロキシがすべてのアプリケーションのクラスターへのエントリポイントを提供します。

**TODO [DOCS-1499] CNPG-GC が PGD6 CM を使用するように更新されたときにプロキシ参照を削除します**

構成ファイルを参照して、この情報を判断します。

ホスト名または IP アドレス、ポート、データベース名、およびユーザー名を確立します。デフォルトのデータベース名は pgddb です。デフォルトのユーザー名は、EDB Postgres Advanced Server の場合は enterprisedb であり、PostgreSQL および EDB Postgres Extended Server の場合は postgres です。

次に、その情報に基づいて接続文字列をアセンブルできます。

```
"host=<hostnameOrIPAddress> port=<portnumber> dbname=<databasename> user=<username>"
```

デプロイメントの構成で必要な場合、 sslmode=<sslmode> を追加します。

## Command reference

PGD コマンドラインインターフェイスのコマンド名は pgd です。

### 概要

EDB Postgres Distributed コマンドラインインターフェイス PGD CLI は、 EDB Postgres 分散クラスターを管理するツールです。 EDB Postgres 分散クラスターに対してコマンドを実行できます。これを使用して、クラスターリソースを検査および管理できます。

### コマンド

- *Creating your first cluster (PGD Essential)* クラスターを管理するためのクラスターレベルのコマンド
- *pgd cluster show* クラスターレベルの情報を表示します。
- *pgd cluster verify* クラスターレベルの情報を確認します。
- **耐久性オプショングループコミット/** *CAMO* グループを管理するためのグループレベルのコマンド
- *pgd cluster show* グループレベルの情報を表示します
- *pgd group set-option* グループレベルのオプションを設定します
- *pgd group get-option* グループレベルのオプションを取得します
- *pgd group set-leader* グループの書き込みリーダーを設定しますスイッチオーバー)。
- **グループとグループステータスの表示** グループをリストするためのグループ関連コマンド。 - *pgd commit-scopes list* グループをリストします。
- **ノード** ノードを管理するためのノードレベルのコマンド。
- *PGD* **ノードのセットアップ** クラスター内のノードをセットアップします。
- *pgd cluster show* ノードレベルの情報を表示します。
- *pgd node part* アクティブなクラスターから PGD ノードを分割します。
- *pgd group set-option* オプション設定ノードレベル.
- *pgd group get-option* ノードレベルのオプションを取得します。
- *pgd node set-config* ノードレベルの構成を設定します。
- *pgd node get-config* ノードレベルの構成を取得します。
- *pgd node upgrade* PGD Postgres ノードのメジャーバージョンのアップグレードを実行します。
- *Node management* ノードをリストするためのノード関連のコマンド。 - *pgd commit-scopes list* ノードをリストします。
- *pgd events* イベントを表示するためのイベントログコマンド。 - *pgd cluster show* イベントを表示します。
- **ユースケース** *1* **マルチマスターリプリケーション**レプリケーション関連、レプリケーションを管理するためのコマンド。 - *pgd cluster show* レプリケーション情報を表示します。
- *Monitoring Raft* **コンセンサス** Raft コンセンサスを管理するための Raft 関連コマンド。 - *pgd cluster show* ラフトの状態に関する情報を表示します。
- *pgd commit-scope create* PGD コミットスコープを管理するためのコミットスコープ関連コマンド。
- *pgd cluster show* コミットスコープに関する情報を表示します。
- *pgd commit-scope create* コミットスコープを作成します。
- *pgd commit-scope update* コミットスコープを更新します。
- *pgd commit-scope drop* コミットスコープをドロップします。

- *pgd commit-scopes list* PGD クラスターのコミットスコープ関連のコマンド。
- *pgd commit-scopes list* クラスターのコミットスコープ情報をリストします。
- *PGD 互換性評価* Postgres サーバーの PGD 互換性を評価します。
- *pgd completion* シェル完了スクリプトを生成します。

#### グローバルオプション

すべてのコマンドは、次のグローバルオプションを受け入れます。

| Short | Long | Description |
|---|---|---|
| *-f* | *--config-file* | 構成ファイルへの名前/パス。これは--dsn フラグが存在する場合は無視されますデフォルト「/etc/edb/pgd-cli/pgd-cli-config.yml」 |
| | *--dsn* | データベース接続文字列 例 "host=bdr-a1 port=5432 dbname=pgddb user=postgres" `PGD_CLI_DSN`環境変数によっても設定されます。 |
| *-h* | *--help* | Help for pgd - 使用されるコマンドの特定のヘルプを表示します |
| *-o* | *--output* | 出力フォーマット *json*、*psql*、*modern*、*markdown*、*simple* 出力フォーマット を参照してください。 |

#### 追加オプション

`pgd -V` を実行して、pgd CLI のバージョン情報を確認します。

#### 出力フォーマット

`-o`/`--output` オプションとともに使用します。

| Format | Description |
|---|---|
| simple | 単純なフォーマット-単純な ASCII テーブルとして出力しますデフォルト。 |
| json | JSON 形式 - JSON ドキュメントとして出力します、非表形式 |
| psql | PSQL フォーマット-PSQL のスタイルの ASCII テーブルとして出力 |
| modern | 最新のフォーマット-ボックスキャラクターを使用して表として出力 |
| markdown | マークダウンテーブルフォーマット - マークダウン互換の ASCII テーブルとして出力 |

### pgd assess

#### 概要

`pgd assess` コマンドは、 EDB Postgres 分散クラスターへの移行に対する Postgres サーバーインスタンスの適合性を評価するために使用されます。

コマンドは、評価する Postgres サーバーインスタンスに接続する DSN で実行する必要があります。これは、`--dsn` オプションの引数として、または `PGD_CLI_DSN` 環境変数を設定して渡すことができます。いずれも実行しないとエラーになります。

次に、コマンドは Postgres サーバーインスタンスの EDB Postgres 分散クラスターとの互換性を確認し、Postgres サーバーインスタンスの互換性に関するレポートを提供します。

### 構文

```
pgd assess [OPTIONS]
```

### オプション

asset コマンドには、コマンド固有のオプションはありません。ただし、評価する Postgres サーバーインスタンスに DSN を接続する必要があります。これは、`--dsn` オプションの引数として、または `PGD_CLI_DSN` 環境変数を設定して渡すことができます。

Global Options も参照してください。

### 例

```bash pgd promote **OUTPUT** アセスメント |結果 |詳細——————————————— -+——— – ———— –+ ———————— - ———————— - 複数のデータベース|互換性のある | 1 つのユーザーデータベースのみが見つかりましたマテリアライズドビュー |互換性のある|マテリアライズドビューは見つかりませんでした EPAS キューテーブル | キューテーブル互換性のある | EPAS キューテーブルが見つかりません DDL コマンドの使用法ワークロード分析が必要です現時点では、自動的にチェックすることはできません。潜在的な互換性| pg_stat_statements でアドバイザリロックコマンドが見つかりませんラージオブジェクト | ラージオブジェクト互換性のある|大きなオブジェクトは見つかりませんでした トリガー/リファレンス権限|互換性のある |互換性のない特権を持つトリガーは見つかりませんでした複数の一意のインデックスを持つテーブル |互換性のある|複数の一意のインデックスを持つテーブルが見つかりませんでした LOCK TABLE 潜在的な互換性 | pg_stat_statements で LOCK TABLE の使用法が見つかりません LISTEN/NOTIFY 互換性のある | pg_stat_statements で LISTEN/NOTIFY の使用法が見つかりません行レベルのロックの使用法 | 行レベルのロックの使用法潜在的な互換性 | pg_stat_statements で行レベルのロックコマンドが見つかりません

## pgd cluster

`pgd cluster` コマンドは、 EDB Postgres 分散クラスターを管理するために使用されます。

### サブコマンド

- *pgd cluster show* クラスターレベルの情報を表示します。
- *pgd cluster verify* クラスターレベルの情報を確認します。

## pgd cluster show

### 概要

`pgd cluster show` コマンドは、 EDB Postgres 分散クラスターのクラスターレベルの情報を表示するために使用されます。
```

### 構文

```
pgd cluster show [OPTIONS]
```

### オプション

次の表は、`pgd cluster show` コマンドで使用可能なオプションをリストしています。

| Short | Long | Description |
|---|---|---|
| | --clock-drift | 詳細なクロックドリフト情報のみを表示します。 |
| | --summary | クラスターの概要情報のみを表示します。 |
| | --health | クラスターの状態に関する情報のみを表示します。 |

一度に指定できるのは、上記のオプションのうちの 1 つだけです。

Global Options も参照してください。

### クロックドリフト

クロックドリフトの現在の実装は、このコマンドの実行中にクラスターが高負荷になっている場合、または多数のノードが含まれている場合、不正確な値を返す場合があることに注意してください。

| Symbol | Meaning |
|---|---|
| * | ok |
| ~ | warning (drift > 2 seconds) |
| *!* | critical (drift > 5 seconds) |
| *x* | down / unreachable |
| *?* | unknown |
| - | not applicable |

### 例

### クラスター情報の表示

```
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
```

(次のページに続く)

(前のページからの続き)

```
#  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

#### 概要

`pgd cluster verify` コマンドは、 EDB Postgres 分散クラスターの構成を確認するために使用されます。

#### 構文

```
pgd cluster verify [OPTIONS]
```

#### オプション

次の表は、`pgd cluster verify` コマンドで使用可能なオプションをリストしています。

| Short | Long | Description |
|---|---|---|
| | --settings | クラスター内の Postgres 設定を確認します。 |
| | --arch | クラスターアーキテクチャの検証 |
| -v | --verbose | 詳細な出力を表示します。 |

オプションを設定しない場合、設定とアーチの両方がデフォルトで検証され、出力は詳細ではありません。

#### 例

##### クラスター設定とアーキテクチャの確認

```
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
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
```

### pgd commit-scope

`pgd commit-scope` コマンドは、 EDB Postgres 分散クラスターのコミットスコープを表示および管理するために使用されます。

### サブコマンド

- *pgd cluster show* コミットスコープに関する情報を表示します。
- *pgd commit-scope create* コミットスコープを作成します。
- *pgd commit-scope update* コミットスコープを更新します。
- *pgd commit-scope drop* コミットスコープを削除します。
- *pgd commit-scopes list* コミットスコープをリストします。

## pgd commit-scope create

### 概要

`pgd commit-scope create` コマンドは、 EDB Postgres 分散クラスターにコミットスコープを作成するために使用されます。

### 構文

```
pgd commit-scope <COMMIT_SCOPE> create [OPTIONS] <RULE_DEFINITION> [GROUP_NAME]
```

<COMMIT_SCOPE> は、作成するコミットスコープの名前です。

<RULE_DEFINITION> は、コミットスコープを定義するルールです。ルールは、トランザクションがコミットされたとみなされるために満たされる必要がある条件を指定します。ルール構文の詳細については、Commit Scopes および Commit Scope Rules を参照してください。

オプションの [GROUP_NAME] は、コミットスコープが属するグループの名前です。省略すると、デフォルトのトップレベルグループが設定されます。

### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

### 例

#### コミットスコープの作成

次の例では、 `dc1_subgroup` グループにルール `ANY 2 (dc1) on replicated group commit` を使用して `abc1` という名前のコミットスコープを作成します。

```
pgd commit-scope abc1 create "ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT" dc1_subgroup
__OUTPUT__
Command executed successfully
```

コミットスコープを確認します。

```
pgd commit-scope abc1 show
__OUTPUT__
Commit Scope Group Name   Rule Definition
```

(次のページに続く)

(前のページからの続き)

```
- ----------- ------------ ---------------------------------------
abc1         dc1_subgroup ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT
```

#### トップレベルグループでのコミットスコープの作成

次の例では、トップレベルグループにルール ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT を使用して abc2 という名前のコミットスコープを作成します。

```
pgd commit-scope abc2 create "ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT"
__OUTPUT__
Command executed successfully
```

コミットスコープを確認します。

```
pgd commit-scope abc2 show
__OUTPUT__
Commit Scope Group Name  Rule Definition
- ----------- ----------- ---------------------------------------
abc2         democluster ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT
```

### pgd commit-scope drop

#### 概要

pgd commit-scope drop コマンドは、 EDB Postgres 分散クラスターからコミットスコープをドロップするために使用されます。

#### 構文

```
pgd commit-scope <COMMIT_SCOPE> drop [OPTIONS] [GROUP_NAME]
```

<COMMIT_SCOPE> は、ドロップするコミットスコープの名前です。

オプションの [GROUP_NAME] は、コミットスコープが属するグループの名前です。省略すると、デフォルトのトップレベルグループが設定されます。グループの名前は、コミットスコープが作成されたグループ名と一致する必要があることに注意してください。

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### コミットスコープのドロップ

次の例では、abc2 という名前のコミットスコープをトップレベルグループから削除します。

```
pgd commit-scope abc2 drop
__OUTPUT__
Command executed successfully
```

### グループからコミットスコープを削除する

次の例は、 `dc1_subgroup` グループから `abc1` という名前のコミットスコープを削除します。

```
pgd commit-scope abc1 drop dc1_subgroup
__OUTPUT__
Command executed successfully
```

## pgd commit-scopes list

### 概要

`pgd commit-scopes list` コマンドは、EDB Postgres 分散クラスターのコミットスコープ情報をリストするために使用されます。

### 構文

```
pgd commit-scopes list [OPTIONS] [GROUP_NAME]
```

`GROUP_NAME` は、コミットスコープをリストするグループの名前です。指定しない場合、クラスター内のすべてのコミットスコープがリストされます。

edb_notranlate_1shell pgd commit-scopes list **OUTPUT** コミットスコープ|グループ名 |ルール定義 ———————— +———————— +—– ——————————————————— – ——————————————————— – — ローカルプロテクト | pgdx6 |非同期コミットラグプロテクト | pgdx6 |マジョリティオリジングループラグ制御 max_lag_time = 30s、max_commit_migrate = 10s マジョリティプロテクト | マジョリティプロテクト pgdx6 |マジョリティ オリジン グループ同期コミット 適応的保護 | pgdx6 |マジョリティオリジングループ同期コミット DEGRADE ON (timeout = 10s、 require_write_lead = true) TO 非同期コミットグループ a | ALL group-a 同期コミット

edb_notranlate_2shell pgd commit-scopes list group-a **OUTPUT** pgd commit-scopes list group-a コミットスコープ |グループ名 |ルール定義———————— +———————— +—– ———————— サブグループ保護 |グループ a | ALL group-a SYNCHRONOUS COMMIT ```

## pgd commit-scope show

### 概要

`pgd commit-scope show` コマンドは、EDB Postgres 分散クラスターのコミットスコープに関する情報を表示するために使用されます。

#### 構文

```
pgd commit-scope <COMMIT_SCOPE> show [OPTIONS]
```

<COMMIT_SCOPE> は、情報を表示するコミットスコープの名前です。

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### コミットスコープの表示

次の例は、コミットスコープ abc1 に関する情報を示しています。

```
pgd commit-scope abc1 show
__OUTPUT__
Commit Scope Group Name   Rule Definition
- ----------- ------------ ---------------------------------------
abc1         dc1_subgroup ANY 2 (dc1_subgroup) SYNCHRONOUS COMMIT
```

Group Name 列は、コミットスコープが属するグループの名前を示します。この場合、コミットスコープは dc1_subgroup グループに属します。

Rule Definition 列は、コミットスコープを定義するルールを示します。この場合、ルールは ANY 2 (dc1) SYCHRONOUS COMMIT です。 dc1_subgroup グループはレプリケートされたグループであるため、コミットはグループ内の少なくとも 2 つのノードにレプリケートされ、その内の 2 つのノードはコミットされたとみなされる前にコミットを確認する必要があります。

### pgd commit-scope update

#### 概要

pgd commit-scope update コマンドは、 EDB Postgres 分散クラスターのコミットスコープを更新するために使用されます。

#### 構文

```
pgd commit-scope <COMMIT_SCOPE> update [OPTIONS] <RULE_DEFINITION>[GROUP_NAME]
```

<COMMIT_SCOPE> は、更新するコミットスコープの名前です。

<RULE_DEFINITION> は、コミットスコープを定義するルールです。ルールは、トランザクションがコミットされたとみなされるために満たされる必要がある条件を指定します。ルール構文の詳細については、Commit Scopes および Commit Scope Rules を参照してください。

オプションの [GROUP_NAME] は、コミットスコープが属するグループの名前です。省略すると、デフォルトのトップレベルグループが設定されます。

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### コミットスコープの更新

次の例は、コミットスコープ abc1 をルール ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT で更新します。

```
pgd commit-scope abc1 update "ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT" dc1_subgroup
__OUTPUT__
Command executed successfully
```

#### トップレベルグループのコミットスコープの更新

次の例は、トップレベルグループのルール ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT でコミットスコープ abc2 を更新します。

```
pgd commit-scope abc2 update "ANY 1 (dc1_subgroup) SYNCHRONOUS COMMIT"
__OUTPUT__
Command executed successfully
```

### pgd completion

#### 概要

pgd completion コマンドは、 EDB Postgres 分散 CLI の完了設定を管理するために使用されます。

#### 構文

```
pgd completion <SHELL>
```

<SHELL> は、オートコンプリートスクリプトを生成するシェルです。

シェルの可能な値は bash 、 fish 、 zsh 、および powershell です。

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

```
pgd completion zsh
```

このコマンドは、通常、シェルセッションの起動ファイルの一部として評価されます。Zsh シェルの完了スクリプトを生成し、標準出力に書き込みます。したがって、 .zshrc ファイルに追加します。

```
eval "$(pgd completion zsh)"
```

### pgd events

pgd events コマンドは、 EDB Postgres 分散クラスターでイベントを表示するために使用されます。

#### サブコマンド

- *pgd cluster show* イベントを表示します。

### pgd events show

#### 概要

pgd events show コマンドは、 EDB Postgres 分散クラスターでイベントを表示するために使用されます。追加フラグがない場合、このコマンドは、すべてのノードとグループの最新の 20 件のイベントを表示します。

#### 構文

```
pgd events show [OPTIONS]
```

#### オプション

次の表は、pgd events show コマンドで使用可能なオプションをリストしています。

| Short | Long | Description |
|---|---|---|
| | *--node <NODE_NAME>* | 指定された名前のノードのイベントのみを表示します。 |
| | *--group <GROUP_NAME>* | 指定された名前のグループのイベントのみを表示します。 |
| *-n* | *--limit <LIMIT>* | 表示するイベントの数を制限します。デフォルト 20 |

Global Options も参照してください。

#### ノードの状態

| State | Description |
|---|---|
| NONE | ワーカーが起動したときにノードの状態は設定解除され、現在の既知の状態にすぐに設定されることが予想されます。 |
| CREATED | bdr.create_node() は実行されましたが、ノードはまだ EDB Postgres Distributed クラスターのメンバーではありません。 |
| JOIN_START | bdr.join_node_group() は、ローカルノードの既存の EDB Postgres Distributed クラスターへの参加を開始します。 |

次のページに続く

表 173 – 前のページからの続き

| State | Description |
|---|---|
| JOINING | ノード参加が開始され、現在初期同期フェーズにあり、ノードにスキーマとデータを作成しています。 |
| CATCHUP | 初期同期フェーズが完了しました。現在、結合は、結合が開始されてから上流のピアノードで実行されたトランザクションを取得および適用する最後のステップにあります。 |
| STANDBY | ノードの参加は完了しましたが、変更のブロードキャストはまだ開始されていません。すべての結合はこの状態でしばらく時間を費やしますが、論理スタンバイとして定義されている場合、ノードはこの状態で継続します。 |
| PROMOTE | ノードはロジカルスタンバイであり、bdr.promote_node を呼び出してノードの状態を ACTIVE に移動したところです。これらの 2 つの PROMOTE 状態は、1 つのノードのみが STANDBY より高いが ACTIVE より低い状態になることができるという事実と一貫している必要があります。 |
| PROMOTING | ロジカルスタンバイからフル BDR ノードへの昇格が進行中です。 |
| ACTIVE | ノードは完全な BDR ノードであり、現在アクティブです。これは、最も一般的なノードのステータスです。 |
| PART_START | ノードがアクティブまたはスタンバイで、 bdr.part_node を呼び出して EDB Postgres Distributed クラスターからノードを削除したところです。 |
| PARTING | ノードは他のノードから切断し、コンセンサスまたはレプリケーションでそれ以上の役割を果たしません。 |
| PART_CATCHUP | 非分割ノードは、最近分割されたノードからの欠落データを同期します。 |
| PARTED | これで、すべてのノードでノード分離操作が完了しました。 |

一度に 1 つのノードのみが PROMOTE または PROMOTING の状態になります。STANDBY は、ノードが読み取り専用状態にあることを示します。

### 例

#### 最新 5 件のイベントを表示します

```
$ 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
```

(次のページに続く)

(前のページからの続き)

```
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

pgd group コマンドは、 EDB Postgres 分散クラスター内のグループを管理するために使用されます。

#### サブコマンド

- *pgd cluster show* グループレベルの情報を表示します。
- *pgd group set-option* グループレベルのオプションを設定します。
- *pgd group get-option* グループレベルのオプションを取得します。
- *pgd group set-leader* グループの書き込みリーダーを設定しますスイッチオーバーを実行します。

### pgd group show

#### 概要

pgd group show コマンドは、 EDB Postgres 分散クラスターでグループレベルの情報を表示するために使用されます。

#### 構文

```
pgd group <GROUP_NAME> show [OPTIONS]
```

<GROUP_NAME> は、情報を表示するグループの名前です。

#### オプション

次の表は、pgd cluster show コマンドで使用可能なオプションをリストしています。

| Long | Description |
|---|---|
| --summary | グループの概要のみを表示します。 |
| --options | グループのオプションのみを表示します。 |
| --nodes | グループのノードのみを表示します。 |

一度に指定できるのは、上記のオプションのうちの 1 つだけです。

Global Options を参照してください。

### 例

#### グループ情報を表示する

```
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
 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
```

(次のページに続く)

(前のページからの続き)

```
 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
```

### グループの概要情報を表示する

```
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      |
```

### グループノード情報の表示

```
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
```

### グループオプション情報を表示する

```
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
```

(次のページに続く)

(前のページからの続き)

```
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
```

### グループ情報を JSON として表示する

```shell pgd group group-a show -o json **OUTPUT** [ { “summary”:[ { “info”:“Group Name”, “value”:“group-a” }, { “info”:“親グループ名”, “value”: “pgdx6” }, { “info”: “Group Type”, “value”: “data” }, { “info”: “Write Leader”, “value”: “pgd-a3” } , { “info”: “Commit Scope”, @@ -97,72 +178,144 @@ **OUTPUT** { “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_lay” , “option_source” “:” inherited” , “option_value” : “00:00:00” }, { “option_name”: “check_constraints”, “option_source”: “inherited”, “option_va lue”: “true” }, { “option_name”: “default_commit_scope” , “option_source” : “group” , “option_value” : “local promote” }, { “option_name” : “enable_raft” , “option_source” : “group” , “option_value” : “true” }, { “option_name” : “enable_routing” , “option_source”: “group”, “option_value”: “true” }, { “option_name”: “enable_wal_decoder”, “option_source”: “inherited” , “option_value” : “false” }, { “option_name” : “http_port” , “option_source” : “pg_config” , “option_value” : “6434” }, { “option_name” : “location” , “option_source” : “group” , “option_value” : “London” }, { “option_name” : “num_writers” , “option_source” : “pg_config” , “option_value” : “2” }, { “option_name” : “read_only_consensus_timeout” , “option_source” : “pg_config” , “option_value” : “00:00:00” }, { “option_name” : “read_only_max_client_connections” , “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”, “o ption_source”: “pg_config”, “option_value”: “6433” }, { “option_name”: “read_write_consensus_timeout”, “option_source”: “pg_config”, “option_value” : “00:00:00” }、: “read_write_max_client_connections” , “option_source” : “pg_config” , “option_value” : “250” }, { “option_name” : “read_write_max_server_connections” , “option_source” : “pg_config” , “option_value” : “250” }, { ” options_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

#### 概要

`pgd group set-option` コマンドは、EDB Postgres 分散クラスターでグループレベルのオプションを設定するために使用されます。

#### 構文

```
pgd group <GROUP_NAME> set-option [OPTIONS] <OPTION> <VALUE>
```

<GROUP_NAME> は、オプションを取得するグループの名前です。

<OPTION> は、取得する特定のグループオプションの名前であり、<VALUE> は、それに設定する値です。

次のオプションが使用できます。

#### グループオプション

| Option | Description |
|---|---|
| apply_delay | グループに変更を適用する際の遅延。 |
| check_constraints | グループ内のコンストレインをチェックするかどうか。 |
| default_commit_scope | グループのデフォルトのコミットスコープ。 |
| enable_routing | グループ内のルーティングを有効にするかどうか。 |
| enable_raft | グループで Raft を有効にするかどうか。 |
| enable_wal_decoder | グループで WAL デコーダーを有効にするかどうか。 |
| location | グループの場所。 |
| num_writers | グループ内のライターの数。 |
| route_reader_max_lag | グループ内のリーダーの最大遅延。 |
| route_writer_max_lag | グループ内のライタの最大遅延。 |
| streaming_mode | グループのストリーミングモード。 |
| route_writer_wait_flush | グループ内のライタをフラッシュするための待機時間。 |
| default_seqkind | グループのデフォルトのシーケンスの種類。 |
| default_replica_identity | グループのデフォルトのレプリカ ID。 |
| conflict_detection_method | グループの競合検出方法。 |
| replay_progress_frequency | グループの再生進行頻度。 |
| batch_inserts | グループ内のバッチ挿入を有効にするかどうか。 |
| analytics_storage_location | グループ内の分析の保存場所。 |

次のページに続く

表 175 – 前のページからの続き

| Option | Description |
|---|---|
| analytics_autoadd_tables | グループ内の分析にテーブルを自動的に追加するかどうか。 |

### グループ接続マネージャーオプション

| Option | Description |
|---|---|
| read_write_port | 読み取り/書き込み接続をリッスンするポート |
| read_only_port | 読み取り専用接続をリッスンするポート |
| http_port | REST API 呼び出しをリッスンする http ポート統合の目的 |
| use_https | http リスナーが HTTPS を使用するかどうか。有効になっている場合、サーバー証明書は TLS に使用されます |
| read_write_max_client_co | 許可される最大読み取り/書き込みクライアント接続、デフォルトは max_connections |
| read_write_max_server_cc | サーバーに対して開かれる最大読み取り/書き込み接続 |
| read_only_max_client_cor | 許可される最大読み取り専用クライアント接続 |
| read_only_max_server_co | サーバーに対して開かれる読み取り専用接続の最大数 |
| read_write_consensus_tim | 読み取り/書き込み接続が受け入れられなくなるまでにコンセンサスが失われるまで待機する時間 |
| read_only_consensus_time | 読み取り専用接続が受け入れられなくなるまでに、コンセンサス損失を待機する時間。 |

### グループプロキシオプション PGD 5.0〜5.8 のみ

| Option | Description |
|---|---|
| proxy_listen_address | グループ内のプロキシのリッスンアドレス。 |
| proxy_listen_addresses | グループ内のプロキシのリッスンアドレス。 |
| proxy_listen_port | グループ内のプロキシのリッスンポート。 |
| proxy_max_client_conn | グループ内のプロキシのクライアントの最大接続数。 |
| proxy_max_server_conn | グループ内のプロキシのサーバー接続の最大数。 |
| proxy_server_conn_timeou | グループ内のプロキシのサーバー接続タイムアウト。 |
| proxy_server_conn_keepal | グループ内のプロキシのサーバー接続キープアライブ。 |
| proxy_fallback_node_grou | グループ内のプロキシのフォールバックノードグループ。 |
| proxy_fallback_node_grou | グループ内のプロキシのフォールバックノードグループタイムアウト。 |
| proxy_consensus_grace_pe | グループ内のプロキシのコンセンサス猶予期間。 |
| proxy_read_listen_address | グループ内の読み取りプロキシのリッスンアドレス。 |
| proxy_read_listen_address | グループ内の読み取りプロキシのリッスンアドレス。 |
| proxy_read_listen_port | グループ内の読み取りプロキシのリッスンポート。 |
| proxy_read_max_client_cc | グループ内の読み取りプロキシのクライアントの最大接続数。 |
| proxy_read_max_server_c | グループ内の読み取りプロキシのサーバー接続の最大数。 |
| proxy_read_server_conn_k | グループ内の読み取りプロキシのサーバー接続キープアライブ。 |
| proxy_read_server_conn_t | グループ内の読み取りプロキシのサーバー接続タイムアウト。 |

次のページに続く

表 177 – 前のページからの続き

| Option | Description |
|---|---|
| proxy_read_consensus_gra | グループ内の読み取りプロキシのコンセンサス猶予期間。 |

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### グループの場所を設定する

```
pgd group dc1_subgroup set-option location London
__OUTPUT__
Command executed successfully
```

#### スペースを含む値にオプションを設定する

```
pgd group dc1_subgroup set-option location "New York"
__OUTPUT__
Command executed successfully
```

### pgd group get-option

#### 概要

`pgd group get-option` コマンドは、EDB Postgres 分散クラスターでグループレベルのオプションを取得するために使用されます。

#### 構文

```
pgd group <GROUP_NAME> get-option [OPTIONS] <OPTION>
```

`<GROUP_NAME>` は、オプションを取得するグループの名前です。

`<OPTION>` は、取得する特定のグループオプションの名前です。オプションを指定しない場合、すべてのグループオプションがリストされます。次のオプションが使用できます。

#### グループオプション

| Option | Description |
|---|---|
| apply_delay | グループに変更を適用する際の遅延。 |
| check_constraints | グループ内のコンストレインをチェックするかどうか。 |
| default_commit_scope | グループのデフォルトのコミットスコープ。 |
| enable_routing | グループ内のルーティングを有効にするかどうか。 |

次のページに続く

表 178 – 前のページからの続き

| Option | Description |
|---|---|
| enable_raft | グループで Raft を有効にするかどうか。 |
| enable_wal_decoder | グループで WAL デコーダーを有効にするかどうか。 |
| location | グループの場所。 |
| num_writers | グループ内のライターの数。 |
| route_reader_max_lag | グループ内のリーダーの最大遅延。 |
| route_writer_max_lag | グループ内のライタの最大遅延。 |
| streaming_mode | グループのストリーミングモード。 |
| route_writer_wait_flush | グループ内のライタをフラッシュするための待機時間。 |
| analytics_storage_location | グループ内の分析の保存場所。 |
| analytics_autoadd_tables | グループ内の分析にテーブルを自動的に追加するかどうか。 |

### グループ接続マネージャーオプション

| Option | Description |
|---|---|
| read_write_port | 読み取り/書き込み接続をリッスンするポート |
| read_only_port | 読み取り専用接続をリッスンするポート |
| http_port | REST API 呼び出しをリッスンする http ポート統合の目的 |
| use_https | http リスナーが HTTPS を使用するかどうか。有効になっている場合、サーバー証明書は TLS に使用されます |
| read_write_max_client_co | 許可される最大読み取り/書き込みクライアント接続、デフォルトは max_connections |
| read_write_max_server_cc | サーバーに対して開かれる最大読み取り/書き込み接続 |
| read_only_max_client_cor | 許可される最大読み取り専用クライアント接続 |
| read_only_max_server_co | サーバーに対して開かれる読み取り専用接続の最大数 |
| read_write_consensus_tim | 読み取り/書き込み接続が受け入れられなくなるまでにコンセンサスが失われるまで待機する時間 |
| read_only_consensus_time | 読み取り専用接続が受け入れられなくなるまでに、コンセンサス損失を待機する時間。 |

### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

### 例

#### すべてのグループオプションをリストする

```
pgd group dc1_subgroup get-option
__OUTPUT__
 Option Name                        | Option Value
- ----------------------------------+---------------
```

(次のページに続く)

(前のページからの続き)

```
 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
 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
```

#### 特定のグループオプションを取得する

```
pgd group dc1_subgroup get-option location
__OUTPUT__
 Option Name | Option Value
- ------------+--------------
 location    | London
```

#### 特定のグループオプションを json として取得する

```
pgd group dc1_subgroup get-option location -o json
__OUTPUT__
[
  {
    "option_name": "location",
    "option_value": "London"
  }
]
```

### pgd group set-leader

#### 概要

pgd group set-leader コマンドは、 EDB Postgres 分散クラスター内のグループの書き込みリーダーを設定するために使用されます。

このコマンドは、スイッチオーバー操作を実行します。

#### 構文

```
pgd group <GROUP_NAME> set-leader [OPTIONS] <LEADER>
```

<GROUP_NAME> は、書き込みリーダーを設定するグループの名前であり、<LEADER> は、書き込みリーダーとして設定するノードの名前です。

#### オプション

次の表は、pgd group set-leader コマンドで使用可能なオプションをリストしています。

| Short | Long | Description |
|---|---|---|
| | *--strict* | Strict メソッドデフォルト。 |
| | *--timeout* | メソッドが厳密な場合のタイムアウト期間。 (デフォルト 30s30 秒) |
| | *--fast* | 速い方法。 |

Strict メソッドがデフォルトのメソッドです。厳密な方法は、リーダーを切り替える前に、新しいリーダーが古いリーダーと同期するのを待機します。高速な方法は、route_write_max_lag を無視して、リーダーを切り替える前に新しいリーダーが古いリーダーと同期するのを待機しないため、即時です。

Global Options も参照してください。

#### 例

##### グループの書き込みリーダーの設定

```
pgd group dc1_subgroup set-leader kaboom
__OUTPUT__
Command executed successfully
```

##### ノードが既にリーダーである場合の書き込みリーダーの設定

```
pgd group dc1_subgroup set-leader kaboom
__OUTPUT__
Node kaboom is already the write leader
```

### pgd groups

pgd groups コマンドは、 EDB Postgres 分散クラスター内のグループを表示するために使用されます。

#### サブコマンド

- *pgd commit-scopes list* グループをリストします。

### pgd groups list

#### 概要

pgd groups list コマンドは、EDB Postgres 分散クラスター内のグループを表示するために使用されます。

#### 構文

```
pgd groups list [OPTIONS]
```

#### オプション

pgd groups list コマンドでは、次のオプションが使用できます。

| Short | Long | Description |
|---|---|---|
| -v | --verbose | グループに関する詳細情報を表示します。 |

共通のグローバルオプションについては、 グローバルオプション を参照してください。

#### 例

##### すべてのグループをリストする

```
pgd groups list
__OUTPUT__
Group Name    Parent Group Name Group Type Nodes
- ------------ ----------------- ---------- -----
democluster                     global     0
dc1_subgroup  democluster       data       3
```

##### すべてのグループと詳細情報をリストする

```
pgd groups list --verbose
__OUTPUT__
Group Name    Parent Group Name Group Type Nodes Raft Leader Write Leader Commit␣
↪Scope Node Group ID
- ------------ ----------------- ---------- ----- ----------- ------------ -----------
```

(次のページに続く)

(前のページからの続き)

```
↪- -------------
democluster                    global     0     kaftan
 150732310
dc1_subgroup  democluster      data       3     kaftan      kaboom
 1302278103
```

### pgd node

pgd node コマンドは、 EDB Postgres 分散クラスターのノードを管理するために使用されます。

#### サブコマンド

- *PGD ノードのセットアップ* クラスター内の PGD データノードを構成します。
- *pgd cluster show* ノードレベルの情報を表示します。
- *pgd node part* アクティブなクラスターから PGD ノードを分離します。
- *pgd group set-option* ノードレベルのオプションを設定します。
- *pgd group get-option* ノードレベルのオプションを取得します。
- *pgd node set-config* ノードレベルの構成を設定します。
- *pgd node get-config* ノードレベルの構成を取得します。
- *pgd node upgrade* PGD Postgres ノードのメジャーバージョンのアップグレードを実行します。

### pgd node get-config

#### 概要

pgd node get-config コマンドは、 EDB Postgres 分散クラスターのノードレベルの構成設定 GUC を取得するために使用されます。

#### 構文

```
pgd node <NODE_NAME> get-config [OPTIONS] [CONFIG]
```

<NODE_NAME> は、オプションを取得するノードの名前です。

[CONFIG] は、取得する特定の GUC の名前です。オプションを指定しない場合、すべての GUC が表示されます。

#### オプション

pgd node get-config コマンドでは、次のオプションが使用できます。

| Short | Long | Description |
|---|---|---|
| -v | --verbose | GUC の最小値、最大値、説明などの追加情報を表示します |

Global Options を参照してください。

### 例

#### 特定のノード構成を取得する

```
pgd node pgd-a1 get-config bdr.batch_inserts
__OUTPUT__
 Config Name       | Config Value
- ------------------+--------------
 bdr.batch_inserts | 15
```

#### 詳細な出力を使用して特定のノード構成を取得する

```
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.
```

#### 特定のノード構成を json として取得する

```
 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

#### 概要

`pgd node get-option` コマンドは、 EDB Postgres 分散クラスターのノードレベルのオプションを取得するために使用されます。

#### 構文

```
pgd node <NODE_NAME> get-option [OPTIONS] [OPTION]
```

`<NODE_NAME>` は、オプションを取得するノードの名前です。

`[OPTION]` は、取得する特定のグループオプションの名前です。オプションを指定しない場合、すべてのオプションが表示されます。

次のオプションが使用できます。

#### ノードオプション

| Option | Description | Type |
|---|---|---|
| route_priority | 同じノードグループ内の他のノードに対するノードの相対ルーティング優先度。書き込みリーダーを選択する場合にのみ使用されます。 | integer |
| route_fence | ノードをフェンスするように設定します。フェンスされると、ノードは接続マネージャーから接続を受信できません。したがって、書き込みリーダーになることも、読み取り専用ノードプールで使用することもできません。レプリケーションは影響を受けません。 | bool |
| route_writes | 書き込みをこのノードにルーティングできるかどうか、つまりノードが書き込みリーダーになることができるかどうかを決定します。 | bool |
| route_reads | 読み取り専用接続をこのノードにルーティングできるかどうかを決定します PGD 5.5.0 以降。 | bool |
| route_dsn | 接続マネージャーがこのノードに接続するために使用する dsn。 | string |

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### すべてのノードオプションの取得

```
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
```

#### 特定のノードオプションの取得

```
pgd node kaboom get-option route_priority
__OUTPUT__
Option Name    Option Value
- ------------ ------------
route_priority 100
```

#### すべてのノードオプションを json として取得

```
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

#### 概要

pgd node part は、アクティブなクラスターから PGD ノードを分離します。部分操作には時間がかかる場合があるため、コマンドはデフォルトで操作が完了するまで待機します。ただし、--no-wait オプションが設定されている場合、コマンドはパートオペレーションをスケジュールし、アドバイスで戻ります。

#### 構文

```
pgd node <NODE_NAME> part [OPTIONS]
```

<NODE_NAME> は、分離するノードの名前です。

#### オプション

pgd node part コマンドでは、次のオプションが使用できます。

| Long | Description |
|---|---|
| --no-wait | 部分操作が完了するまで待たないでください。 |

Global Options を参照してください。

#### 例

##### –no-wait オプションを使用してノードを分割する

```
pgd node pgd-a1 part --no-wait
__OUTPUT__
Initiating a part node operation for node: pgd-b1
The operation will be performed in background and may take some time
NOTICE:  node pgd-a2 removal started in the background
DETAIL:  node is parting nodegroup group-a (19805032)
```

##### –no-wait オプションを使用せずにノードを分割する

```
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

#### 概要

pgd node set-config コマンドは、EDB Postgres 分散クラスターでノードレベルの Postgres 設定 GUC を設定するために使用されます。このコマンドには、superuser つまり Postgres プロセス所有者ユーザーのノード DSN が必要です。

コマンドは、構成を設定した後、ノードに構成をリロードし、GUC はサーバーの再起動が必要であるという警告を表示します。

#### 構文

```
pgd node <NODE_NAME> set-config [OPTIONS] <CONFIG> <VALUE>
```

<NODE_NAME> は、オプションを取得するノードの名前です。

また、 <CONFIG> は、取得する特定のノード構成の名前であり、<VALUE> は、それに設定する値です。

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

##### 特定のノード構成を設定する

```
pgd node pgd-a1 set-config bdr.batch_inserts 20 --dsn "host=pgd-a1 port=5432␣
↪user=postgres "
__OUTPUT__
Command executed successfully
```

##### 値にスペースを使用して特定のノード構成を設定します

```
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
```

WARN メッセージは、Postgres サーバーが再起動されるまで構成の変更が有効にならないことを示していることに注意してください。

### pgd node set-option

#### 概要

pgd node set-option コマンドは、 EDB Postgres 分散クラスターでノードレベルのオプションを設定するために使用されます。

### 構文

```
pgd node <NODE_NAME> set-option [OPTIONS] <OPTION> <VALUE>
```

<NODE_NAME> は、オプションを取得するノードの名前です。

また、<OPTION> は、取得する特定のノードオプションの名前であり、<VALUE> は、それに設定する値です。

次のオプションが使用できます。

### ノードオプション

| Option | Description | Type |
|---|---|---|
| route_priority | 同じノードグループ内の他のノードに対するノードの相対ルーティング優先度。書き込みリーダーを選択する場合にのみ使用されます。 | integer |
| route_fence | ノードをフェンスするように設定します。フェンスされると、ノードは接続マネージャーから接続を受信できません。したがって、書き込みリーダーになることも、読み取り専用ノードプールで使用することもできません。レプリケーションは影響を受けません。 | bool |
| route_writes | 書き込みをこのノードにルーティングできるかどうか、つまりノードが書き込みリーダーになることができるかどうかを決定します。 | bool |
| route_reads | 読み取り専用接続をこのノードにルーティングできるかどうかを決定します PGD 5.5.0 以降。 | bool |
| route_dsn | 接続マネージャーがこのノードに接続するために使用する dsn。 | string |

### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

### 例

#### 特定のノードオプションを設定する

```
pgd node kaboom set-option route_priority 100
__OUTPUT__
Command executed successfully
```

#### 値にスペースを使用して特定のノードオプションを設定します

```
pgd node kaboom set-option route_dsn "host=kaboom port=5444 dbname=pgddb user=postgres
↪"
__OUTPUT__
Command executed successfully
```

### pgd node setup

#### 概要

pgd node setup コマンドは、クラスター内の PGD データノードを構成するために使用されます。新しいノードのセットアップ、既存のノードのクラスターへの参加、またはクラスターへのノードの論理参加を実行するために使用できます。

コマンドの動作は、コマンドで指定されたローカルノードとリモートノードの状態によって異なります。

これがクラスター内の最初のノードの場合、pgd node setup は initdb を実行し、PGD ノードをセットアップします。

これが最初のノードではなく、ローカルノードが起動して実行されていない場合、pgd node setup はクラスターへのノードの物理的参加を実行します。これにより、初期化プロセスの一部としてリモートノードからローカルノードにデータがコピーされ、ローカルノードがクラスターに参加します。これは、新しいノードにデータをロードする最も速い方法です。

ローカルノードが起動して実行されており、リモートノードにも到達可能である場合、pgd node setup はクラスターへのノードの論理参加を実行します。これにより、クラスター内に新しいノードが作成され、リモートノードからストリーミングレプリケーションが開始されます。これは、新しいノードを既存のクラスターに追加するための推奨される方法です。

ローカルノードがアップして実行されており、リモートノード dsn が提供されていない場合、pgd node setup は、ノードが指定されたグループの一部でない場合、ノードグループのスイッチを実行します。

#### ユーザーとロール

pgd node setup コマンドを実行するには、スーパーユーザーロールが必要です。スーパーユーザーロールは、データディレクトリを作成し、データベースを初期化するために使用されます。データベースを作成するには、スーパーユーザーロールに CREATEDB 特権が必要です。

--dsn オプションで指定されたユーザーが存在しない場合は作成されます。PGD 機能を管理できる bdr_superuser ロールのみが付与されます。ただし、データベースに対する他の特権はありません。

#### 構文

```
pgd node <NODE_NAME> setup [OPTIONS] -D <PG_DATA>
```

### 引数

- <NODE_NAME> 作成するノードの名前。これは、クラスター内のノードを識別するために使用される名前です。クラスター内で一意である必要があります。

### オプション

| Option | Description |
|---|---|
| *--listen-addr <LISTEN_ADDR>* | 構成されたノードが着信接続をリッスンするアドレス、および他のノードがこのノードに接続するために使用するアドレス。これは通常、少なくとも`localhost`に設定されますが、任意の有効なアドレスに設定できます。デフォルトは`localhost`です。 `--dsn`からの`host`値もこのリストに追加されます。 |
| *--initial-node-count <INITIAL_NODE_COUNT>* | クラスター内のノードまたはクラスター内に含まれる予定のノードの数。ノードのさまざまなリソース設定を計算するために使用されます。デフォルト 3 です。 |
| *--bindir <BINDIR>* | <BINDIR> バイナリが存在するディレクトリを指定します。デフォルトは、実行中の pgd バイナリが置かれているディレクトリです。 |
| *--log-file <LOG_FILE>* | ログファイルへのパス。postgres 起動ログに使用されます。デフォルトでは、*postgres-<port>.log* `という名前の現在のディレクトリにあるファイルに書き込みます。ここで、ポート値は`--dsn`オプションの`port`属性から取得されます。 |
| *-D, --pgdata <PG_DATA>* | ノードのデータディレクトリとして<PG_DATA>を使用します。環境変数`PGDATA`でも設定されます。これは有効なディレクトリであり、コマンドを実行するユーザーが書き込み可能である必要があります。 |
| *--superuser <SUPERUSER>* | `initdb`のスーパーユーザー名。デフォルトは`postgres`です。 |
| *--node-kind <NODE_KIND>* | 作成するノードの種類を指定します。デフォルトは`data`です。使用可能な値は`data`、*witness*、`subscriber-only`です。 |
| *--group-name <GROUP_NAME>* | ノードグループ名。指定しない場合、ノードはアクティブノードのグループに追加されます。これは、グループの最初のノードの必須引数です。 |
| *--create-group* | 指定されたグループがまだ存在しない場合、このフラグを設定して作成します。これは、最初のノードのデフォルトで true になります。 |
| *--cluster-name <CLUSTER_NAME>* | ノードに参加するクラスターの名前。クラスターを初めてセットアップするときに、これは`parent node group`を作成するために使用されます。指定しない場合、デフォルトの`pgd`。 |
| *--cluster-dsn <CLUSTER_DSN>* | アクティブな PGD クラスターに属する DSN。これは、クラスターの最初のノードを構成する場合には必要ありませんが、後続のノードには必須です。既存のアクティブノードの DSN を指す必要があります。 |
| *--postgresql-conf <POSTGRESQL_CONF>* | ノードに使用される`postgresql.conf`ファイルのオプションのパス。 |
| *--postgresql-auto-conf <POSTGRESQL_AUTO_CONF>* | ノードに使用される`postgresql.auto.conf`ファイルのオプションのパス。 |

次のページに続く

表 186 – 前のページからの続き

| Option | Description |
|---|---|
| *--hba-conf <HBA_CONF>* | ノードに使用される `pg_hba.conf` ファイルのオプションのパス。 |
| *--update-pgpass* | 設定されている場合、新しいノードパスワードの pgpass ファイルは、現在のユーザーの `.pgpass` ファイルに保存されます。 |
| *--verbose* | NA |

Global Options も参照してください。

### 例

これらの例では、3 つのホスト host-1 、host-2 、および host-3 でクラスターをセットアップして、3 つのノード node-1 、node-2 、および node-3 を作成します。3 つのノードはデータノードであり、グループ名が group-1 である pgd という名前のクラスターの一部です。

PGPASSWORD 環境変数をエクスポートして、コマンドを実行するたびに pgdadmin ユーザーのパスワードを入力する必要を回避することをお勧めします。これは、次のコマンドで行うことができます。

```
export PGPASSWORD=pgdsecret
```

### 最初のノードの構成

```
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
```

コマンドをステップ実行して、node-1 を設定しています。最初のオプションは--dsn オプションで、これはノードの接続文字列です。これは通常、host=hostname port=5432 user=pgdadmin dbname=pgd に設定されます。これは、ローカル Postgres インスタンスの一般的な接続文字列です。

--listen-address オプションは、ノードが着信接続をリッスンするアドレスを指定するために使用されます。この場合、localhost,host-1 に設定しています。これは、ノードがローカルホストと host-1 アドレスの両方でリッスンすることを意味します。

これはクラスター内の最初のノードであるため、グループ名を group-1 に設定し、クラスター名を pgd 実際にデフォルトに設定します。これはクラスター内の最初のノードであるため、--create-group オプションが自動的に設定されます。

最後に、 -D オプションを使用してノードのデータディレクトリを設定します。これは、Postgres データファイルが保存される場所です。この例では、データディレクトリとして/var/lib/edb-pge/17/main を使用しています。

このコマンドは、データディレクトリを作成し、データベースを PGD 用に正しく初期化します。次に、ノードを起動し、クラスターに参加する他のノードを含む新しい接続で利用できるようにします。

### 2 番目のノードの構成

```
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"
```

このコマンドは最初のノードに似ていますが、node-2 を設定しています。 --dsn オプションは、ノードの接続文字列であり、通常は host=hostname port=5432 user=pgdadmin dbname=pgd に設定されます。cluster-dsn は、アクティブなノードを指す必要があります。接続マネージャー、またはプロキシエンドポイントなどを指すことができます。CLI は、その背後にあるノードの実際の DSN を取得します。この場合、クラスター内の最初のノードの接続文字列である host=host-1 port=5432 user=pgdadmin dbname=pgd に設定しています。

### 3 番目のノードの構成

```
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
```

このコマンドは 2 番目のノードに似ていますが、node-3 を設定しています。 --dsn オプションは、ノードの接続文字列であり、通常は host=hostname port=5432 user=pgdadmin dbname=pgd に設定されます。cluster-dsn は、アクティブなノードを指す必要があります。接続マネージャー、またはプロキシエンドポイントなどを指すことができます。CLI は、その背後にあるノードの実際の DSN を取得します。この場合、クラスター内の最初のノードの接続文字列である host=host-1 port=5432 user=pgdadmin dbname=pgd に設定しています。

### 分割およびドロップされたノードのクラスターへの参加

```
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
```

このコマンドは、後続のノードの設定に似ていますが、 node-2 を再度設定します。 --dsn オプションは、ノードの接続文字列であり、通常は host=hostname port=5432 user=pgdadmin dbname=pgd に設定されます。 cluster-dsn は、アクティブなノードを指す必要があります。接続マネージャー、またはプロキシエンドポイントなどを指すことができます。CLI は、その背後にあるノードの実際の DSN を取得します。この場合、クラスター内の最初のノードの接続文字列である host=host-1 port=5432 user=pgdadmin dbname=pgd に設定しています。

これは、メンテナンスなどのアクティビティのためにノードがクラスターから parted および dropped され、クラスターに再参加する必要がある場合に役立ちます。コマンドは、クラスターへのノードの論理参加を実行します。これにより、クラスター内に新しいノードが作成され、リモートノードからストリーミングレプリケーションが開始されます。

### pgd node show

#### 概要

pgd node show コマンドは、 EDB Postgres 分散クラスターにノードレベルの情報を表示するために使用されます。

#### 構文

```
pgd node <NODE_NAME> show [OPTIONS]
```

<NODE_NAME> は、情報を表示するノードの名前です。

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### ノード情報の表示

```
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
```

ノード情報を JSON として表示する

```
pgd node kaboom show -o json
__OUTPUT__
[
  {
    "Summary": [
      {
        "info": "Node Name",
        "value": "kaboom"
      },
      {
        "info": "Group Name",
        "value": "dc1_subgroup"
      },
      {
        "info": "Node Kind",
        "value": "data"
      },
      {
        "info": "Join State",
        "value": "ACTIVE"
      },
      {
        "info": "Node Status",
        "value": "Up"
      },
      {
        "info": "Node ID",
        "value": "2710197610"
      },
      {
        "info": "Snowflake SeqID",
        "value": "2"
      },
      {
        "info": "Database",
        "value": "pgddb"
      }
    ]
  },
  {
    "Options": [
      {
        "option_name": "route_dsn",
        "option_value": "host=kaboom port=5444 dbname=pgddb user=postgres "
```

(次のページに続く)

(前のページからの続き)

```
      },
      {
        "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

#### 概要

pgd node upgrade コマンドは、EDB Postgres 分散クラスター内のノードの PostgreSQL バージョンをアップグレードするために使用されます。

#### 構文

```
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>
```

<NODE_NAME> はアップグレードするノードの名前であり、<OLD_BINDIR> 、<NEW_BINDIR> 、<OLD_DATADIR> 、<NEW_DATADIR> 、<DATABASE> 、および<USER_NAME> は、新旧の Postgres インスタンスの bin ディレクトリ、新旧の Postgres インスタンスデータディレクトリ、データベース名、およびクラスターのインストールです。それぞれユーザー名。

#### オプション

次の表は、pgd node upgrade コマンドで使用可能なオプションをリストしています。

| Short | Long | Default | Env | Description |
|---|---|---|---|---|
| -b | --old-bindir | | PGBINOLD | 古い Postgres インスタンスの bin ディレクトリ |
| -B | --new-bindir | | PGBINNEW | 新しい Postgres インスタンスの bin ディレクトリ |
| -d | --old-datadir | | PGDATAOLD | 古い Postgres インスタンスデータディレクトリ |
| -D | --new-datadir | | PGDATANEW | 新しい Postgres インスタンスデータディレクトリ |
| | --database | | PGDATABASE | PGD データベース名 |
| -p | --old-port | 5432 | PGPORTOLD | 古い Postgres インスタンスポート |
| | --socketdir | /var/run/postgresc | PGSOCKET-DIR | アップグレード中にポストマスターソケットに使用するディレクトリ |
| | --new-socketdir | /var/run/postgresc | PGSOCKET-DIRNEW | 新しいクラスターのポストマスターソケットに使用するディレクトリ |
| | --check | | | チェックのみを実行し、クラスターを変更しないように指定します |
| -j | --jobs | 1 | | 同時に使用するプロセスまたはスレッドの数 |
| -k | --link | | | 新しいクラスターにファイルをコピーする代わりにハードリンクを使用する |
| | --old-options | | | 古い postgres コマンドに渡すオプション、複数の呼び出しが追加されます |
| | --new-options | | | 新しい postgres コマンドに渡すオプション、複数の呼び出しが追加されます |
| -N | --no-sync | | | アップグレードされたクラスター内のすべてのファイルがディスクに書き込まれるのを待たない |
| -P | --new-port | 5432 | PGPORTNEW | 新しい Postgres インスタンスのポート番号 |
| -r | --retain | | | 正常に完了した後でも SQL ファイルとログファイルを保持する |
| -U | --username | | PGUSER | クラスターのインストールユーザー名 |
| | --clone | | | 効率的なファイルのクローン作成を使用する |

Global Options も参照してください。

#### 例

次の例では、“kaolin”は、クイックスタートデモクラスターからアップグレードするノードの名前です。

#### ノードの PostgreSQL バージョンをアップグレードする

```
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
```

#### ハードリンクを備えたノードの PostgreSQL バージョンをアップグレードする

```
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
```

#### 効率的なファイルクローン作成を使用してノードの PostgreSQL バージョンをアップグレードする

```
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
```

#### 別のポート番号を持つノードの PostgreSQL バージョンをアップグレードする

```
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
```

### pgd nodes

`pgd nodes` コマンドは、 EDB Postgres 分散クラスター内のノードを表示するために使用されます。

#### サブコマンド

- *pgd commit-scopes list* ノードをリストします。

### pgd nodes list

#### 概要

`pgd nodes list` コマンドは、 EDB Postgres 分散クラスター内のノードを表示するために使用されます。デフォルトでは、これはノード名、グループ名、ノードの種類、ノードの参加状態、およびアップまたはダウンを示します。

### 構文

```
pgd nodes list [OPTIONS]
```

### オプション

pgd nodes list コマンドでは、次のオプションが使用できます。

| Short | Long | Description |
|---|---|---|
| | --versions | ノードに関するバージョン情報のみを表示します。ノードごとに、 BDR バージョンと Postgres バージョンが表示されます。 |
| -v | --verbose | ノードに関する詳細情報を表示します。このオプションは、ノードごとに、ノード ID、Snowflake シーケンス ID、およびデータベース名を追加します。 |

共通のグローバルオプションについては、 グローバルオプション を参照してください。

### 例

#### すべてのノードをリストする

```
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
```

#### すべてのノードと詳細情報をリストする

```
pgd nodes list --verbose
__OUTPUT__
Node Name Group Name   Node Kind Join State Node Status Node ID    Snowflake SeqID␣
↪Database
- -------- ------------ --------- ---------- ----------- ---------- --------------- --
↪------
kaftan    dc1_subgroup data      ACTIVE     Up          3490219809 1               ␣
↪pgddb
kaboom    dc1_subgroup data      ACTIVE     Up          2710197610 2               ␣
↪pgddb
kaolin    dc1_subgroup data      ACTIVE     Up          2111777360 3               ␣
↪pgddb
```

#### すべてのノードのバージョン情報をリストする

```
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)
```

### pgd raft

pgd raft コマンドは、 EDB Postgres 分散クラスターの raft ステータスを表示するために使用されます。

#### サブコマンド

- *pgd cluster show* クラスターの raft ステータスを表示します。

### pgd raft show

#### 概要

pgd raft show コマンドは、 EDB Postgres 分散クラスターの Raft ステータスを表示するために使用されます。特に、トップレベルグループを含むすべてのグループのすべてのノードと、そのラフトステータスリーダーまたはフォロワー、それらが含まれるグループ内のノードの数、グループ内の投票ノードの数、リーダーの存在、およびターム番号。

#### 構文

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

#### オプション

コマンド固有のオプションはありません。 Global Options を参照してください。

#### 例

#### Show Raft ステータス

```
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
```

(次のページに続く)

(前のページからの続き)

```
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
```

ここでの dc1_subgroup はローカルルーティングを使用するデータグループであり、democluster はグローバルルーティングを使用するトップレベルグループであることに注意してください。

Protocol Version 列は、使用中の Raft プロトコルのバージョンを示します。 Commit Index 列は、最後にコミットされたログエントリのインデックスを示します。Nodes 列は、グループ内のノードの合計数を示します。 Voting Nodes 列は、Raft コンセンサスに参加するノードの数を示します。 State 列は、ノードの Raft 状態リーダーまたはフォロワーを示します。Leader Name 列は、グループ内のリーダーノードの名前を示します。Current Term 列には、現在の用語番号が表示されます。

### pgd replication

pgd replication コマンドは、 EDB Postgres 分散クラスターのレプリケーションステータスのさまざまな側面を表示するために使用されます。

#### サブコマンド

- *pgd cluster show* クラスターのレプリケーションステータスを表示します。

### pgd replication show

#### 概要

pgd replication show コマンドは、 EDB Postgres 分散クラスターでのレプリケーションステータスを表示するために使用されます。

オプションなしのデフォルトでは、次のレポートを生成します。

- ノードレプリケーションの進行状況ノード間のレプリケーションステータスのマトリックス。
- Replication Slots 各ノードのスロットのレプリケーションスロットステータス。
- サブスクリプションノード間の各サブスクリプションのサブスクリプションステータス。
- 分析レプリケーション 各ノードの分析レプリケーションステータス。

オプションを使用して、出力を上記のレポートのいずれかに制限できます。--verbose オプションを使用して、デフォルトレポートの詳細を増やし、各ノードの他のノードへの接続の LSN とレプリケーションラグを表示できます。

### 構文

```
pgd replication show [OPTIONS]
```

### オプション

`pgd replication show` コマンドでは、次のオプションが使用できます。

| Short | Long | Description |
|---|---|---|
| | --nodes | ノード間のレプリケーション状態のみをマトリックス形式で表示します。 |
| | --slots | 各ノードのレプリケーションスロットを表示します。 |
| | --subscriptions | ノード間の各サブスクリプションのサブスクリプションステータスを表示します。 |
| | --analytics | 各ノードの分析レプリケーションステータスを表示します。 |
| -v | --verbose | レプリケーションステータスに関する詳細情報を表示します。 |

共通のグローバルオプションについては、 グローバルオプション を参照してください。

#### –slots

これは、 BDR レプリケーションスロットのステータスを示します。verbose フラグを使用した出力は、スロットがアクティブか、レプリケーション状態切断、ストリーミング、キャッチアップ、おおよその遅延などの詳細を提供します。

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

マトリックスビューでは、バイトラグが括弧内に表示される場合があります。maxOf(WriteLag, FlushLag, ReplayLag, SentLag) です。

### 例

#### EDB Postgres Distributed クラスターでのレプリケーションステータスの表示

```
pgd replication show
__OUTPUT__
```

(次のページに続く)

(前のページからの続き)

```
#  Node Replication Progress

Node   kaboom kaftan kaolin
- ----- ------ ------ ------
kaboom -       **
kaftan  *-*
kaolin  **       -

#  Replication Slots

Group Name   Origin Node Target Node Slot Name                      Active State    ␣
↪Write Lag Replay Lag Sent Lag Bytes Write Lag Bytes Replay Lag Bytes
- ----------- ----------- ----------- ------------------------------ ------ ---------␣
↪--------- ---------- -------------- --------------- ----------------
dc1_subgroup kaboom      kaftan      bdr_pgddb_democluster15_kaftan t      streaming␣
↪00:00:00  00:00:00   0               0               0
dc1_subgroup kaboom      kaolin      bdr_pgddb_democluster15_kaolin t      streaming␣
↪00:00:00  00:00:00   0               0               0
dc1_subgroup kaftan      kaboom      bdr_pgddb_democluster15_kaboom t      streaming␣
↪00:00:00  00:00:00   0               0               0
dc1_subgroup kaftan      kaolin      bdr_pgddb_democluster15_kaolin t      streaming␣
↪00:00:00  00:00:00   0               0               0
dc1_subgroup kaolin      kaboom      bdr_pgddb_democluster15_kaboom t      streaming␣
↪00:00:00  00:00:00   0               0               0
dc1_subgroup kaolin      kaftan      bdr_pgddb_democluster15_kaftan t      streaming␣
↪00:00:00  00:00:00   0               0               0

#  Subscriptions

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

ノード間のレプリケーション状態のみをマトリックス形式で表示する

```
pgd replication show --nodes
__OUTPUT__
Node   kaboom kaftan kaolin
- ----- ------ ------ ------
kaboom -       **
kaftan  *-*
kaolin  **       -
```

### 6.3.24 Application use

PGD でのアプリケーションの開発は、PostgreSQL データベースでの作業とほぼ同じです。ただし、異なるのは、アプリケーションがレプリケーションとどのように相互作用するかを認識する必要があることです。PGD がアプリケーションでどのように動作するか、レプリケートされる SQL 、さまざまなノードがどのように処理されるか、およびその他の重要な情報を知る必要があります。

- *Application behavior* は、次のような PGD レプリケーションがアプリケーションにどのように表示されるかを確認します。
- レプリケートされるコマンド
- ローカルで実行されるコマンド
- 行レベルのロックが取得されたとき b_tran_7 トリガーが発生した方法と場所 b_tran_7 ラージオブジェクト
- トースト
- *DML and DDL replication and nonreplication* は、SQL ステートメントの 2 つのクラスの違いと、PGD がそれらのレプリケートを処理する方法を示します。また、PGD がまったくレプリケートしないコマンドも調べます。
- *Nodes with differences* は、レプリケートノードに異なるテーブル構造とスキーマが存在する構成で PGD がどのように動作するかを調べます。また、このようなノードを LiveCompare で比較する方法、およびノードで実行されている PostgreSQL バージョンの違いを処理する方法についても説明します。
- *General rules for applications* は、データ異常を回避するためのアプリケーションにいくつかの一般的なルールを提供します。
- *Internal timing of operations* は、非同期/同期レプリケーションがアプリケーションのデータのビューにどのような影響を与える可能性があるかを示し、古い読み取りを軽減する機能を示します。
- *Using extensions with PGD* は、PGD で拡張機能を選択、インストール、および構成する方法を説明します。
- *Use of table access methods (TAMs) in PGD* TAMs は、PGD で使用可能な TAM とそれらを有効にする方法をノートします。
- *Feature compatibility* は、どのサーバー機能がどのコミットスコープで動作するか、およびどのコミットスコープをデイジーチェーン接続できるかを示します。

### Application behavior

PGD のレプリケーション動作の多くは、アプリケーションに対して透過的です。PGD でうまく動作するアプリケーションの開発を成功させるには、それを実現する方法と透明でない要素を理解することが重要です。

### レプリケーション動作

PGD は、あるノードで行われた変更の他のノードへの複製をサポートしています。

PGD は、デフォルトで、INSERT、UPDATE、DELETE、および TRUNCATE 操作からのすべての変更をソースノードから他のノードにレプリケートします。すべてのトリガーとルールが処理された後、最終的な変更のみが送信されます。たとえば、`INSERT ... ON CONFLICT UPDATE` は、オリジンで発生した内容に応じて、挿入または更新を送信します。更新または削除が影響を与える行がない場合、変更は送信されません。

前提条件なしで INSERT をレプリケートできます。

他のノードで更新と削除をレプリケートするには、PGD が影響を受ける一意の行を識別できる必要があります。PGD では、テーブルに PRIMARY KEY が定義されている場合、UNIQUE 制約、または特定の列に明示的な REPLICA IDENTITY が定義されていることが必要です。これらのいずれかが定義されていない場合、警告が生成され、その後の更新または削除は明示的にブロックされます。テーブルに REPLICA IDENTITY FULL が定義されている場合、一意のインデックスは必要ありません。その場合、更新と削除が許可され、ライブ、有効、遅延せず、式または WHERE 句を含まない最初の非一意のインデックスを使用します。それ以外の場合、シークエンシャルスキャンが使用されます。

### Truncate

TRUNCATE は、レプリケーション ID が定義されていない場合でも使用できます。TRUNCATE コマンドのレプリケーションはサポートされていますが、外部キーで接続されたテーブルのグループを切り詰める場合には注意してください。truncate アクションをレプリケートする場合、サブスクライバーは、レプリケーションセットが定義されている場合を除き、CASCADE によって明示的に指定または暗黙的に収集された、オリジンで切り詰められたテーブルのグループを切り捨てます。詳細と例については、Replication sets を参照してください。これは、影響を受けるすべてのテーブルが同じサブスクリプションの一部である場合、正常に動作します。しかし、サブスクライバーで切り詰めるテーブルに、同じまたは任意のレプリケーションセットの一部ではないテーブルへの外部キーリンクがある場合、サブスクライバーに truncate アクションを適用すると失敗します。

### 行レベルのロック

INSERT、UPDATE、および DELETE コマンドによって暗黙的に取得された行レベルのロックは、変更が発生すると複製されます。INSERT、UPDATE、DELETE、および TRUNCATE コマンドによって暗黙的に取得されたテーブルレベルのロックもレプリケートされます。ユーザーセッションによる明示的な行レベルのロック `SELECT ... FOR UPDATE/FOR SHARE` はレプリケートされず、アドバイザリロックもレプリケートされません。SERIALIZABLE モードで実行されているトランザクションによって格納されている情報は、他のノードにレプリケートされません。SERIALAZABLE のトランザクション分離レベルがサポートされていますが、マルチプルのノードに同時トランザクションが存在する場合、トランザクションはノード全体でシリアル化されません。

DML がマルチプルのノードで同時に実行される場合、非同期レプリケーションで実行すると潜在的な競合

が発生する可能性があります。これらに対処するか、回避する必要があります。Conflicts で議論されているように、さまざまな回避メカニズムが可能です。

### シークエンス

シーケンスには、Sequences で説明している特別な処理が必要です。これは、クラスターでは、ノードが競合する値を作成するのを避けるためにシーケンスをグローバルにする必要があるためです。グローバルシーケンスは、完全性を保証するグローバルロックで使用できます。

### バイナリオブジェクト

BYTEA 列のバイナリデータは通常にレプリケートされ、最大 1 GB のデータの「blob」が許可されます。PostgreSQL の「ラージオブジェクト」機能の使用は、PGD ではサポートされていません。

### ルール

ルールはオリジンノードでのみ実行されるため、レプリカに対して有効になっている場合でも、適用中に実行されません。

### ベーステーブルのみ

レプリケーションは、ベーステーブルからベーステーブルへのみ可能です。つまり、サブスクリプション側のソースとターゲットのテーブルは、ビュー、マテリアライズドビュー、外部テーブルではなく、テーブルである必要があります。ベーステーブル以外のテーブルを複製しようとすると、エラーが発生します。更新可能なビューを介して行われた DML 変更は、オリジン上のベーステーブルに解決され、ターゲット上の同じベーステーブル名に適用されます。

### パーティション化されたテーブル

PGD は、パーティション化されたテーブルを透過的にサポートします。つまり、パーティション化されたテーブルをレプリケーションセットに追加でき、パーティションに関係する変更はダウンストリームにレプリケートされます。

### トリガー

デフォルトでは、トリガーは元のノードでのみ実行されます。たとえば、INSERT トリガーは元のノードで実行され、ターゲットノードに変更を適用するときに無視されます。`ALTER TABLE ... ENABLE ALWAYS TRIGGER` を使用して、実行時にオリジンノードで実行され、レプリケートされたとき**適用時** のターゲットの両方で実行するトリガーを指定できます。または、`REPLICA` オプションを使用して適用時にのみ実行します `ALTER TABLE ... ENABLE REPLICA TRIGGER`。

一部のトリガーは、テーブルに存在し、現在有効になっている場合でも、適用時に実行されません。実行されないトリガータイプは次のとおりです。

- ステートメントレベルのトリガー `FOR EACH STATEMENT`
- 列ごとの UPDATE トリガー `UPDATE OF column_name [, ...]`

PGD レプリケーション適用は、システムレベルのデフォルトの search_path を使用します。レプリカトリガー、ストリームトリガー、およびインデックス式ファンクションは、適用時に実行時に失敗する他の

search_path 設定を引き受けることができます。これの発生を防ぐには、次のいずれかのテクニックを使用します。

- デフォルトの search_path のみを使用して、オブジェクト参照を明確に解決します。
- オブジェクトへの完全修飾参照は常に使用します `schema.objectname` など。
- 影響を受けるファンクションに対して `ALTER FUNCTION ... SET search_path = ...` を使用して、ファンクションの検索パスを設定します。

PGD は、テキストまたはその他の照合可能なデータ型に関連する問題がないことを前提としています。つまり、使用中のすべての照合順序はすべてのノードで使用でき、デフォルトの照合順序はすべてのノードで同じです。変更のレプリケーションは、等価検索を使用してレプリカ ID 値を見つけるため、一意のインデックスが一致しない照合修飾子で明示的に定義されている場合を除き、これは効果がありません。照合可能な式が使用される場合、行フィルタは照合順序の違いの影響を受ける可能性があります。

#### トースト

PostgreSQL の非常に長い「トーストされた」データの PGD 処理は、ユーザーに対して透過的です。TOAST の「chunkid」値は、異なるノードの同じ行の間で異なる可能性がありますが、それは問題を発生させません。

#### その他の制限

レプリカ ID 列が外部としてマークされている場合、PGD は正しく動作できません。

PostgreSQL では、揮発性ファンクションを含む CHECK() 制約が許可されます。PGD は適用時に CHECK() 制約を再実行するため、以前と同じ結果を返さない後続の再実行はデータの相違を発生させます。

PGD は、外部キーの使用を制限しません。カスケード FK は許可されます。

### DML and DDL replication and nonreplication

SQL ステートメントの 2 つのメジャークラスは、 DML と DDL です。

- DML はデータ変更言語であり、テーブルに格納されているデータを変更する SQL ステートメントに関係します。これには、UPDATE、DELETE、および INSERT が含まれます。
- DDL はデータ定義言語であり、データの保存方法を変更する SQL ステートメントに関係します。これには、CREATE、ALTER、および DROP が含まれます。

PGD は、各クラスを個別に処理します。

#### DML レプリケーション

PGD は DML ステートメントをレプリケートしません。DML ステートメントによって発生した変更をレプリケートします。たとえば、2 つの行を変更した UPDATE は 2 つの変更をレプリケートしますが、行を削除しなかった DELETE は何もレプリケートしません。これは、volatile ステートメントの実行結果がレプリケートされ、ステートメントベースのレプリケーションで発生する可能性のあるノード間の相違がないことを意味します。

### DDL レプリケーション

DDL レプリケーションは DML とは異なります。DDL の場合、PGD はステートメントをレプリケートし、すべてのノードで実行します。したがって、`DROP TABLE IF EXISTS` はローカルノードで何もレプリケートしない場合がありますが、DDL レプリケーションが有効になっている場合、ステートメントは他のノードに送信されて実行されます。詳細については、ステートメント固有の *DDL* レプリケーションの問題 を参照してください。

PGD は、同じトランザクション内でも、混合された DML および DDL ステートメントが正しく動作するように機能します。

### レプリケートされていないステートメント

これらの 2 つのクラスの外部には、PGD がデザインでレプリケートしない SQL コマンドがあります。次のユーザーコマンドは PGD によってレプリケートされないため、その効果はローカル/オリジンノードでのみ発生します。

- カーソル操作 DECLARE、CLOSE、FETCH
- 実行コマンド DO、CALL、PREPARE、EXECUTE、EXPLAIN
- セッション管理 DEALLOCATE、DISCARD、LOAD
- パラメーターコマンド SET、SHOW

・コンストレイン操作 SET CONSTRAINTS

- ロックコマンド LOCK
- テーブルメンテナンスコマンド VACUUM、ANALYZE、CLUSTER、REINDEX
- 非同期操作 NOTIFY、LISTEN、UNLISTEN

`NOTIFY` SQL コマンドと `pg_notify()` ファンクションはレプリケートされないため、フェイルオーバーの場合の通知は信頼できません。これは、サーバーがクラッシュしたときにトランザクションがコミットされた場合、フェールオーバー時に通知が簡単に失われる可能性があることを意味します。`LISTEN` を実行しているアプリケーションは、フェイルオーバーが発生した場合の通知を見逃す場合があります。

これは標準の PostgreSQL レプリケーションに当てはまりますが、PGD はこれをまだ改善していません。

CAMO および Eager Replication オプションでは、`NOTIFY` SQL コマンドまたは `pg_notify()` ファンクションは使用できません。

## Nodes with differences

### 差分を含むノード間のレプリケーション

デフォルトでは、 DDL はすべてのノードに送信されます。ステートメント固有の *DDL* レプリケーションの問題 で説明しているように、この動作を制御でき、それを使用して、ノード間のデータベーススキーマ間の差異を作成できます。PGD は、ノード間のわずかな違いがあってもレプリケーションを続行できるように設計されています。これらの機能は、ダウンタイムなしでアプリケーションスキーマを移行できるように、またはロジカルスタンバイノードでレポートまたはテストをできるように設計されています。

現在、レプリケーションには、すべてのノードで同じテーブル名が必要です。将来の機能では、異なるテーブル名間のマッピングが可能になる可能性があります。

ターゲット上のパーティションを変更する更新のサポートを含む、パーティションテーブルにレプリケートする通常のテーブルのソースなど、異なるパーティション定義を持つテーブル間でレプリケートが可能です。パーティショニング定義がソースとターゲットで同じである場合、適用時に動的パーティションルーティングを実行する必要がないため、高速になります。詳細については、Replication sets を参照してください。

デフォルトでは、すべての列がレプリケートされます。

PGD は、列名に基づいてデータ列をレプリケートします。列の名前が同じで異なるデータ型がある場合、PGD はソースタイプからターゲットタイプにキャストを試行します（それを許可するキャストが定義されている場合）。

PGD は、列数が異なるテーブル間のレプリケーションをサポートしています。

ターゲットにソースからの欠落列がある場合、PGD は `target_column_missing` 競合を発生させます。デフォルトの競合リゾルバーは `ignore_if_null` です。これは、NULL 以外の値が到着した場合にエラーをスローします。または、`ignore` の競合リゾルバーを使用してノードを構成することもできます。この設定はエラーをスローしませんが、追加の列をサイレントに無視します。

ターゲットにソースレコードで見られない追加の列がある場合、PGD は `source_column_missing` 競合を発生させます。デフォルトの競合リゾルバーは `use_default_value` です。追加列にデフォルト NULL ヌル可能な場合またはデフォルト式がある場合、レプリケーションは続行されます。そうでない場合、エラーをスローし、レプリケーションを停止します。

変換トリガーはテーブルで使用して、デフォルト値を提供したり、適用前にさまざまな方法で受信データを変更したりすることもできます。

ソースとターゲットに異なる制約がある場合、レプリケーションが試行されますが、ソースからの行をターゲットに適用できない場合、失敗する場合があります。ここで行フィルタが役立ちます。

1 つのスキーマからより緩和されたスキーマにデータを複製しても、障害は発生しません。スキーマからより制限的なスキーマへのデータの複製は、潜在的な障害のソースになる可能性があります。これを解決する正しい方法は、より緩和な側にコンストレインを配置して、悪いデータを入力できないようにすることです。このようにして、レプリケーションによって不正なデータが到着することはなく、より制限的なスキーマへの変換に失敗することはありません。たとえば、あるスキーマに TEXT タイプの列があり、別のスキーマが XML と同じ列を定義している場合、TEXT 列に CHECK 制約を追加して、テキストが XML であることを強制します。

ノードごとに異なるインデックスを使用してテーブルを定義できます。デフォルトでは、インデックス定義はレプリケートされます。ノードのサブセットのみまたはローカルにインデックスを作成する方法を指定するには、**ステートメント固有の *DDL* レプリケーションの問題** を参照してください。

`fillfactor` や `toast_tuple_target` などのストレージパラメーターは、テーブルのノード間で問題なく異なる場合があります。その動作の例外は、テーブルのストレージパラメーター `user_catalog_table` の値がすべてのノードで同一である必要があることです。

レプリケートされるテーブルは、各ノードで同じユーザー/ロールが所有している必要があります。詳細は、Security and roles を参照してください。

ロールは、各ノードの接続に対して異なるパスワードを持つことができますが、デフォルトでは、ロール

の変更は各ノードにレプリケートされます。ノードのサブセットのみまたはローカルでロールパスワードを変更する方法を指定するには、DDLreplication を参照してください。

#### 違いがあるノード間の比較

LiveCompare は、PGD ノードおよび非 PGD ノードとデータベースのデータを比較するためのツールです。比較して最終結果に到達するには、少なくとも 2 つの接続が必要です。

LiveCompare 1.3 以降、`all_bdr_nodes` セットを使用して構成できます。この設定により、クラスター内の個別のノードごとに関連するすべての DSN を明確にする必要がありません。EDB Postgres 分散クラスターには、接続情報を備えた N 個のノードがありますが、LiveCompare 1.3 以降がジョブを完了するために必要な初期接続と出力接続のみです。`logical_replication_mode` 設定は、すべてのノードがどのように通信しているかを示します。

すべての構成は、たとえば、 `bdrLC.ini` という名前の`.ini` ファイルで実行されます。 `/etc/2ndq-livecompare/` でこの構成ファイルのテンプレートを見つけます。

LiveCompare の実行中、N+1 の進行状況バーが表示されます。N はプロセス数です。すべてのテーブルがソースされると、1 秒あたりのトランザクション tps が測定されたとして時間が表示されます。このメカニズムは時間をカウントし続け、推定を提供して、最後に合計実行時間を提供します。

このツールは、テーブル、スキーマ、replication_sets など、多くのカスタマイズとフィルターを提供します。LiveCompare は、コンテキスト情報を失うことなく stop-start を使用できるため、都合の良いときに実行できます。比較の後、概要と DML スクリプトが生成されるので、確認できます。DML を適用して、見つかった差異を修正します。

#### 異なるリリースレベル間のレプリケーション

ノード間のその他の違いは、ノードに PostgreSQL の異なるメジャーバージョンが存在するかどうかです。PGD は、異なるメジャーリリースバージョン間で複製するように設計されています。この機能は、ダウンタイムなしでメジャーバージョンのアップグレードを可能にするように設計されています。

PGD は、PGD ソフトウェアの異なるバージョンを持つノード間で複製するように設計されています。この機能は、ダウンタイムなしでバージョンのアップグレードとメンテナンスを可能にするように設計されています。

ただし、クラスター内のメジャーバージョンのノードを参加することは可能ですが、クラスターが新しいプロトコルバージョンを使用している場合、マイナーバージョンのノードを追加することはできません。そうするとエラーが返されます。

これらの機能は両方とも、特定の制限の影響を受ける可能性があります。既知の非互換性については、Releasenotes を参照してください。

### General rules for applications

#### 背景

PGD は、レプリカ ID 値を使用して、変更する行を識別します。アプリケーションが同じ一意の識別子を挿入、削除、および後で再利用すると、問題が発生する可能性があります。これは、ABAproblem と呼ばれます。PGD は、行が現在の行、最後の行、またはかなり古い行であるかどうかを知ることができません。

同様に、PGD はテーブル名を使用して、変更が再生されるテーブルを識別するため、同じオブジェクト名を作成、ドロップ、および後で再利用するアプリケーションにも同様の ABA 問題が存在します。

#### アプリケーションのルール

これらの問題により、アプリケーションが従うべき単純なルールがいくつか発生します。

- 行の一意の識別子を使用します INSERT
- 一意の識別子 UPDATE の変更を避けます。
- 削除された一意の識別子の再利用は避けてください。
- ドロップされたオブジェクト名の再利用は避けてください。

一般に、これらのルールに違反すると、データの異常と発散が発生する可能性があります。アプリケーションは、特定の条件が満たされる限り、これらのルールを破ることができます。ただし、注意してください。異常が発生する可能性は低いですが、不可能ではありません。たとえば、ダウンしたノードを含むすべてのノードで DELETE が再生された限り、行値を再利用できます。これは通常 1 秒以内に発生しますが、1 つのノードで重大な問題が発生し、正しく再起動できなかった場合には数日かかる場合があります。

### Timing considerations and synchronous replication

デフォルトでは非同期であるため、ピアノードは遅延する場合があります。この動作により、複数の PGD ノードに接続したクライアント、またはそれらをスイッチングしたクライアントが古いデータを読み取ることができます。

*queue waitfunction* は、クライアントまたはプロキシに提供され、このような古い読み取りを防止します。

Postgres の同期レプリケーション機能は、PGD でも利用できます。さらに、PGD は、より同期したレプリケーションのための複数のバリアントを提供します。使用可能なすべてのバリアントとそのさまざまなモードの概要と比較については、**コミットスコープ** を参照してください。

### Using extensions with PGD

#### PGD およびその他の PostgreSQL 拡張機能

PGD は、PostgreSQL 拡張機能として実装されています**サポートされている *Postgres* データベースサーバー** を参照してください。PostgreSQL の拡張性と柔軟性を利用して、低レベルのシステムの動作を変更して、マルチマスターレプリケーションを提供します。

原則として、コミュニティ PostgreSQL、 EDB Postgres Advanced Server が提供する拡張機能、およびサードパーティの拡張機能は PGD で使用できます。ただし、PGD の分散的な性質は、選択してインストールする拡張機能を注意深く検討および計画する必要があることを意味します。

#### 論理デコーディングを提供する拡張機能

wal2json などの論理デコードを提供する拡張機能は、理論的には PGD で動作すると考えられます。ただし、フェイルオーバーはサポートされていないため、このような拡張機能から読み取られる WAL ストリームが中断される可能性があることを意味します。

#### レプリケーションまたは HA 機能を提供する拡張機能

レプリケーションまたは HA /フェールオーバーに関連する機能で PostgreSQL を拡張する拡張機能は、PGD でうまく動作する可能性が低く、PGD クラスターの状態に悪影響を与える可能性があります。これらは避けることをお勧めします。

#### サポートされている拡張機能

これらの拡張機能は PGD によって明示的にサポートされています。

### EDB Advanced Storage テーブルのアクセス方法

*EDB Advanced Storage Pack* は、拡張機能として実装されるテーブルアクセス方法 TAM の選択を提供します。次の TAM は、PGD での使用が認定されています。

- *Autocluster*
- *Refdata*

詳細については、 *Use of table access methods (TAMs) in PGD* を参照してください。

### pgaudit

PGD は、 pgaudit 拡張機能との互換性を確保するために変更されました。構成情報については、*Postgres* **設定** を参照してください。

#### 拡張機能のインストール

PostgreSQL 拡張機能は、ファンクション、データ型、およびオプションで 1 つ以上の共有ライブラリなどの SQL オブジェクトを提供します。拡張機能をインストールして使用する前に、これらを PostgreSQL バックエンドにロードする必要があります。

> **警告**
>
> 関連する拡張機能パッケージは、クラスター内のすべてのノードで利用できる必要があります。そうしないと、拡張機能のインストールが失敗し、クラスターの安定性に影響を与える可能性があります。

Trusted Postgres Architect を使用して PGD が展開されている場合、そのツールを使用して拡張機能を構成します。詳細については、Adding Postgres extensions を参照してください。

以下は、手動で構成された PGD インストールに関連します。

#### shared_preload_libraries の構成

拡張機能が共有ライブラリを提供する場合、拡張機能をインストールする前にこのライブラリを shared_preload_libraries 構成パラメーターに含めます。

`shared_preload_libraries` は、拡張機能名のコンマ区切りのリストで構成されます。 `bdr` を含める必要があります。通常、他の拡張機能を指定する順序は重要ではありません。ただし、pgaudit 拡張機能を使用している場合、`pgaudit` は `bdr` の前にリストに表示される必要があります。

CREATE EXTENSION で拡張機能をインストールする前に、クラスター内のすべてのノードで shared_preload_libraries を構成します。新しい構成をアクティブにするには、PostgreSQL を再起動する必要があります。

Postgres settings も参照してください。

### 拡張機能のインストール

CREATE EXTENSION コマンドを使用して拡張機能をインストールします。これは、クラスター内の 1 つのノードでのみ行う必要があります。PGD の DDL レプリケーションは、他のすべてのノードに伝播します。

> **警告**
>
> *CREATE EXTENSION* を実行する前に DDL レプリケーションを無効にするなど、各ノードに拡張機能を手動でインストールしようとしないでください。

bdr.replicate_ddl_command() などのコマンドを使用して CREATE EXTENSION を実行しないでください。

## Use of table access methods (TAMs) in PGD

*EDB Advanced Storage Pack* は、 EDB Postgres 15.0 から利用可能なテーブルアクセス方法 TAM を提供します。

次の TAM は、PGD 6.0 での使用が認定されました。- *Autocluster*

- *Refdata*

EDB によって認定されるまで、他の TAM の使用は制限されます。

PGD クラスターでこれらの TAM のいずれかを使用するには、適切な拡張ライブラリ autocluster および/または refdata を各ノードの shared_preload_libraries パラメーターに追加し、PostgreSQL サーバーを再起動する必要があります。

クラスター内のすべてのノードの shared_preload_libraries に拡張ライブラリが存在すると、 CREATE EXTENSION autocluster; または CREATE EXTENSION refdata; を使用して拡張機能自分自身を作成できます。 CREATE EXTENSION コマンドは、1 つのノードでのみ実行する必要があります。クラスター内の他のノードに複製されます。

拡張機能を作成した後、 CREATE TABLE test USING autocluster; または CREATE TABLE test USING refdata; を使用して、指定された TAM でテーブルを作成します。これらのコマンドは、クラスター内のすべての PGD ノードに複製します。

これらのテーブルアクセス方法の詳細については、以下を参照してください。

- *Autocluster example*
- *Refdata example*

### Feature compatibility

#### サーバー機能/コミットスコープの相互運用性

すべてのサーバー機能がすべてのコミットスコープで動作するわけではありません。この表は、相互運用するものを示しています。

| | Async (default) | *Parallel Apply* | トランザクションストリーミングでの使用 | *pg_upgrade* |
|---|---|---|---|---|
| 耐久性オプショングループコミット/*CAMO* | Not applicable | Does not interoperate | Does not interoperate ☒ | Interoperates |
| 耐久性オプショングループコミット/*CAMO* | Not applicable | Interoperates | Does not interoperate | Does not interoperate |
| *Lag Control* | Interoperates | Interoperates | Interoperates | Interoperates |
| 同期コミット | Not applicable | Interoperates | Interoperates | Interoperates |

#### 注

該当なしテーブルの Async 列は、使用中の同期コミットスコープのない PGD を表します。Lag Control は同期コミットスコープではありません。これは制御コミットスコープであるため、非同期操作で使用できます。

☒グループコミットとトランザクションストリーミングを使用しようとすると、警告が表示されます。警告は、トランザクションストリーミングを無効にすることを示唆しており、トランザクションが発生しているように見えます。バックグラウンドでは、グループコミットが無効になって、トランザクションが発生できるようになりました。

#### コミットスコープ/コミットスコープの相互運用性

コミットスコープを混合することはできませんが、`AND` 演算子を使用して *combine rules* を行うことはできます。次の表は、コミットスコープを結合できる場所を示しています。

| | 耐久性オプショングループコミット/*CAMO* | 耐久性オプショングループコミット/*CAMO* | *Lag Control* | 同期コミット |
|---|---|---|---|---|
| 耐久性オプショングループコミット/*CAMO* | Not applicable | Does not combine | Combines | Combines |
| 耐久性オプショングループコミット/*CAMO* | Does not combine | Not applicable | Combines | Does not combine |
| *Lag Control* | Combines | Combines | Not applicable | Combines |
| 同期コミット | Combines | Does not combine | Combines | ☒ |

注

各コミットスコープは暗黙的に自分自身を操作します。

# 第7章　Terminology

この用語リストには、EDB Postgres Distributed に関連する馴染みのない用語が含まれています。

## 7.1 非同期レプリケーション

オリジナルノードでトランザクションが完了した後、他の PGD クラスターメンバーにデータをコピーするレプリケーションのタイプ。非同期レプリケーションは、*Near/Far* **アーキテクチャでの同期レプリケーション** よりも高いパフォーマンスと低いレイテンシーを提供できます。ただし、非同期レプリケーションでは、さまざまなクラスターメンバーに変更が表示されるまでにかかる時間に遅延が発生する場合があります。クラスターは **最終的な一貫性** になりますが、ノードが明らかに互いに同期していない可能性があります。

## 7.2 コミットスコープ

PGD クラスターのノードとグループ間でトランザクションがコミットされる方法を管理するためのルール。*Near/Far* **アーキテクチャでの同期レプリケーション** 、 **耐久性オプショングループコミット/** *CAMO* 、*CAMO* **またはコミット・アット・モスト・ワンス** 、 **熱心な** 、Lag Control、およびその他の PGD 機能を構成するために使用されます。

## 7.3 CAMO またはコミット・アット・モスト・ワンス

一部のアプリケーションの高価値トランザクションでは、アプリケーションが正確に 1 回だけ、フェールオーバーと再試行が発生した場合に 1 回だけ正常にコミットする必要があります。PGD でこれが発生するように保証には、CAMO を有効にして、アプリケーションがトランザクションにアクティブに参加できるようにします。

## 7.4 競合

データは PGD クラスターのノードにわたってレプリケートされるため、あるソースからの変更が別のソースからの変更と衝突する場合があります。これは競合であり、競合の解決で処理できます。競合の解決は、どのソースが正しいまたは優先されるかを決定するルールセットです。競合は、競合のないデータ型を使用して回避することもできます。

## 7.5 コンセンサス

*Monitoring Raft* **コンセンサス** がグループ全体の決定を行う方法。グループ内のノードの数を指定すると、Raft は、決定に投票するマジョリティのコンセンサスノードの数を 2 プラス 1 で割ったものを探します。たとえば、書き込みリーダーが選択される場合、グループ内のどのノードが書き込みリーダーになるかに

ついて Raft コンセンサスが求められます。投票メンバーのクォーラムがある場合にのみコンセンサスに到達できます。

## 7.6 クラスター

一般に、クラスターは、エンドユーザーに 1 つのシステムとして見えるように配置された複数のシステムのグループです。*PGD* **クラスターへのアクセス** および *Postgres* **クラスター** も参照してください。

## 7.7 DDL データ定義言語

データベースの構造の定義と管理を処理する SQL コマンドのサブセット。DDL ステートメントは、データベース内のオブジェクトつまりスキーマ、テーブル、インデックスを作成、変更、および削除できます。一般的な DDL コマンドは、CREATE、ALTER、および DROP です。

## 7.8 DML データ操作言語

データベースに保持されているデータの操作を処理する SQL コマンドのサブセット。DML ステートメントは、データベース内のテーブルの行を作成、変更、および削除できます。一般的な DML コマンドは、INSERT、UPDATE、および DELETE です。

## 7.9 熱心な

競合する可能性のある着信トランザクションを検出し、それらのいずれかを「迅速に」中止して一貫性を維持することにより競合を回避する同期コミットモード。

## 7.10 最終的な一貫性

異なるクラスターメンバーの同じ項目への変更を記述する分散コンピューティング一貫性モデルは、最終的に同じ値に収束します。競合解決と競合のないレプリケートデータ型を使用した非同期論理レプリケーションは、PGD の最終的一貫性を示します。

## 7.11 フェイルオーバー

高可用性データベースクラスターの障害を認識し、一貫性と可用性を維持するためのアクションを実行する自動プロセス。目標は、ダウンタイムとデータの損失を最小限に抑えることです。

## 7.12 グループコミット

複数の PGD ノードがコミット時にトランザクションを正常に受信および確認する必要がある同期コミットモード。

## 7.13 即時一貫性

すべてのレプリカが同期的かつ同時に更新される分散コンピューティングモデル。このモデルでは、書き込みが完了した後のすべての読み取りが、すべてのノードで同じ値を参照します。このアプローチの欠点は、パフォーマンスへの悪影響です。

## 7.14 論理レプリケーション

データベース内の変更をレプリケートするより効率的な方法。物理的ストリーミングレプリケーションは元のデータベースのディスクブロックを複製しますが、論理レプリケーションは代わりに、基になる物理的記憶形式とは無関係に変更を取得し、変更を表示するようにサブスクライブしたすべてのシステムに公開します。次に、各サブスクライバーは変更をローカルに適用します。論理レプリケーションは、ほとんどの DDL コマンドをサポートできません。

## 7.15 ノード

分散システムの要素の総称。ノードは任意のサービスのホストとして機能できます。PGD では、*PGD ノードのセットアップ* は Postgres データベース、BDR 拡張機能、および Connection Manager を実行します。

通常、高可用性を実現するために、各ノードは別の物理ハードウェアで実行されますが、常にそうとは限りません。

## 7.16 ノードグループ

PGD クラスター内の PGD ノードをグループに編成して、クラスターの論理操作を反映できます。たとえば、特定の物理的な場所にあるデータノードは、その場所の専用ノードグループの一部であることができます。

## 7.17 PGD クラスター

エンドユーザーには 1 つのシステムとして見えながら、単一障害点を回避するように配置された複数の冗長データベースシステムとプロキシのグループ。PGD クラスターは、Docker インスタンス、クラウドインスタンス、または「ベア」Linux ホスト、またはこれらのプラットフォームの組み合わせで実行できます。PGD クラスターには、バックアップノードを含めることもできます。クラスター内のデータノードは、トップレベルグループにグループ化され、さまざまなローカル*ノードグループ* にグループ化されます。

## 7.18 PGD ノード

PGD クラスターには、データベースを実行し、PGD クラスターに参加するノードがあります。一般的な PGD ノードは、Postgres データベース、 BDR 拡張機能、および Connection Manager を実行します。 PGD モードは **データノード**とも呼ばれ、データを保存することを示唆しています。ただし、一部の PGD ノード、特に *Witness ノード* はそれを行いません。

## 7.19 物理的レプリケーション

物理レプリケーションは、1 つ以上のスタンバイクラスターメンバーに変更されるときにデータベースディスクブロックの正確なコピーを作成することにより、サーバーをレプリケートする簡単に実装された方法を提供します。ただし、使用方法には制限があります。たとえば、1 つのマスターノードのみが書き込みトランザクションを実行できます。また、この方法では、すべてのクラスターメンバーが、同じオペレーティングシステムと CPU アーキテクチャを備えたデータベースソフトウェアの同じメジャーバージョンを使用している必要があります。

## 7.20 Postgres クラスター

通常、PostgreSQL では、単一のサーバーで実行されている多数のデータベースをデータベースのクラスターと呼びます。この種類の Postgres クラスターは、可用性が高くありません。高可用性と冗長性を取得するには、*PGD クラスターへのアクセス* が必要です。

## 7.21 クォーラム

クォーラムとは、分散投票に参加するために必要な投票ノードの最小数です。これにより、行われた決定が有効であることが保証されます。たとえば、PGD クラスターで Raft *Monitoring Raft* コンセンサス が必要な場合、投票に参加する最小数の投票ノードが必要です。5 ノードクラスターの場合、クォーラムはクラスター投票の 3 ノードです。コンセンサスは 5/2+1 ノード、3 つのノードが同じように投票します。投票ノードが 2 つだけの場合、コンセンサスは決して確立されません。PGD では、 *DDL locking details* および Raft の決定にクォーラムが必要です。

## 7.22 レプリケートされた利用可能なフォールトトレランス Raft

分散クラスター内のマシンのクォーラムからの投票を使用してコンセンサスを確立するコンセンサスアルゴリズム。PGD は、グループトップレベルまたはローカル内で Raft を使用して、書き込みリーダーであるノードを確立します。

## 7.23 読み取りスケーラビリティ

増加する読み取りワークロードを処理するシステムの機能。たとえば、PGD は 1 つ以上のリードレプリカノードをクラスターに導入し、アプリケーションにプライマリノードに書き込みを指示し、レプリカノードに読み取りを行わせることができます。読み取りワークロードの増加に応じて、リードレプリカノードの数を増加してパフォーマンスを維持できます。

## 7.24 サブスクリプション

PGD ノードは、データに加えられた変更を関心のあるノードに公開します。他の PGD ノードは、これらの変更をサブスクライブするように求めます。この動作により、サブスクリプションが作成され、各ノードが更新されるメカニズムです。PGD ノードは、他の PGD ノードの変更を双方向にサブスクライブします。

## 7.25 スイッチオーバー

アプリケーションまたはプロキシとクラスター内のアクティブなデータベースノード間の接続の計画的な変更。通常はメンテナンスのために行われます。

## 7.26 同期レプリケーション

変更がすべての参加ノードで同時に更新される場合、通常は 2 フェーズコミットを活用します。このアプローチでは、コミットする前に変更を複製し、競合を解決しますが、ノード全体で必要な調整のため、レイテンシーのパフォーマンスコストが発生します。

## 7.27 サブスクライバー専用ノード

PGD クラスターは、双方向レプリケーションをベースとしています。ただし、読み取り専用サーバーが必要な場合など、一部のユースケースでは、双方向レプリケーションは必要ありません。この場合、加入者専用ノードが使用されます。データベース内の変更のみをサブスクライブして自分自身を最新の状態に保ち、ノードで直接実行に対して正しい結果を提供します。この機能を使用して、PGD クラスターでのホリゾンタル読み取りのスケーラビリティを有効にできます。

## 7.28 2 フェーズコミット 2PC

複数のデータベースノード間の一貫性を実現するためのマルチステップのプロセス。最初のフェーズでは、元のノードで準備され、すべての参加ノードに送信されるトランザクションが表示されます。各参加ノードは、トランザクションを適用できることを検証し、その準備ができていることを元のノードに通知します。これは準備フェーズです。第 2 フェーズでは、すべての参加ノードが準備ができていることを通知した場合、元のノードはトランザクションのコミットを続行し、参加ノードにもコミットするように通知します。これはコミットフェーズです。準備フェーズで、いずれかのノードが準備ができていないことを通知した場合、トランザクション全体が中止されます。このプロセスにより、すべてのノードが同じ変更を取得します。

## 7.29 垂直スケーリングまたはスケールアップ

Oracle Exadata など、そのアーキテクチャの物理的制限に到達するまで、特定のワークロードをサポートするリソース CPU、メモリ、ストレージ、ネットワークを増加する従来のコンピューティングアプローチ。

## 7.30 Witness ノード

Witness ノードは、主にクラスターがコンセンサスを確立するのを支援するために機能します。コンセンサスを確立するには、奇数のデータノードが必要です。リソースが制限されている場合、監視ノードを使用してクラスターの決定に参加できますが、データを複製することはできません。データを保持しないことは、スタンバイサーバーとして動作できない、または同期コミットのマジョリティを提供できないことを意味します。

## 7.31 ライトリーダー

Always-on アーキテクチャでは、ノードがアプリケーションの正しい接続エンドポイントとして選択されます。このノードは書き込みリーダーと呼ばれます。選択すると、PGD Connection Manager はクエリと更新をルーティングします。1 つのノードのみが書き込みを受信するため、意図しないマルチノードの書き込みを回避できます。書き込みリーダーは、データノードのクォーラムのコンセンサスによって選択されます。書き込みリーダーが使用できなくなると、データノードは別のノードを選択して書き込みリーダーになります。書き込みリーダーではないノードは、**シャドウノード** と呼ばれます。

## 7.32 ライター

*EDB_SUBSCRIPTION_TOKEN* がデータ変更を PGD ノードに配信すると、データベースサーバーはライタと呼ばれるワーカープロセスに、これらの変更を適用するタスクを実行します。

# 第8章　PGD compatibility

## 8.1 PGD の PostgreSQL バージョンとの互換性

次の表は、EDB Postgres DistributedPGD と互換性のある PostgreSQL のメジャーバージョンを示しています。

| PGD 6 | Postgres Version |
|---|---|
| *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 では、サポートされている PGD で Postgres メジャーバージョンの最新のマイナーバージョンを使用することをお勧めします。

## 8.2 オペレーティングシステムおよびアーキテクチャとの PGD の互換性

次の表は、EDB Postgres Distributed のバージョンと、さまざまなオペレーティングシステムおよびアーキテクチャとの互換性を示しています。

### 8.2.1 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 |

**注釈**

PGD の以前のバージョンについては、 *PGD compatibility* を参照してください。

# 第9章　EDB Postgres Distributed 6 release notes

EDB Postgres Distributed ドキュメントでは、マイナーリリースとパッチを含む EDB Postgres Distributed 6 の最新バージョンについて説明しています。リリースノートは、各リリースの新機能に関する情報を提供します。マイナーまたはパッチリリースで導入された新機能の場合、コンテンツには、機能を導入したリリースも示されます。

| Release Date | EDB Postgres Distributed |
|---|---|
| 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* |

## 9.1 EDB Postgres Distributed 6.1.0 release notes

リリース日 2025 年 8 月 19 日

これは、 PGD 6 のマイナーリリースであり、 PGD 4.4.0+および PGD 5.9.0+からの直接アップグレードサポートを導入します。これにより、古いバージョンからのシームレスなインプレースアップグレードが可能になる PGD 6 の最初のバージョンになります。この重要な機能に加えて、PGD 6.1.0 には、さまざまな新機能、機能強化、および重要なバグ修正が含まれています。

### 9.1.1 ハイライト

- **PGD 4 および PGD 5 から PGD 6 へのシームレスなアップグレード** PGD 4.4.0+および PGD 5.9.0+ユーザーは、PGD 6.1.0 で始まる PGD 6 への直接アップグレードを実行できるようになり、スムーズな移行を保証します。最新の機能まで。
- **Apache Iceberg へのレプリケーション** PGD は、Analytics Accelerator を使用する際に Apache Iceberg へのレプリケーションをサポートするようになりました。この新機能により、PGD クラスターから Iceberg データレイクにデータを直接ストリーミングし、PGD 内から Iceberg データを直接照会できます。これにより、大規模なデータの分析と統合のオプションが大幅に拡大します。
- **PGD 6 がマテリアライズドビューをサポートするようになりました** CREATE、ALTER、REFRESH、および DROP コマンドはクラスター全体でレプリケートされ、すべてのノードでマテリアライズドビューの一貫性が保証されます。マテリアライズドビューが作成されるか、同時に更新されると、上流ノードにデータが設定され、他のノードにレプリケートされ、この機能の完全なサポートを提供します。
- **PGD 6 が IBM LinuxONE** に登場しました。 PGD のネイティブ機能が IBM LinuxONE の新しいサポートで拡張され、最適化されたアップタイム、スループット、エネルギー効率を提供します。

### 9.1.2 機能

| Description | Addresses |
|---|---|
| **Postgres 設定 GUC を取得および更新するための CLI コマンドを追加しました** - `pgd node get-config`および`pgd node set-config`コマンドを追加して、ユーザーが CLI を介して Postgres 設定 GUC を取得および変更できるようにします。 | |
| **新しい `drop_after_retention_period`引数が `bdr.autopartition`ファンクションに追加されました** - `bdr.autopartition()`は、追加のブール引数`drop_after_retention_period`を取ります。ユーザーがパーティションをドロップせず、デタッチのみを希望する場合に false に設定できます。 | |
| **PGD CLI を使用したノード構成の簡素化** - pgd コマンドラインツールには、個々のノード構成の管理を簡単にする新しいコマンドが含まれています。`pgd node <node-name> get-config`および`pgd node <node-name> set-config`を使用して、クラスター内のノードの特定の Postgres 設定 GUC を CLI から直接取得および更新できるようになりました。 | |
| **PGD CLI ビューコミットスコープ** - PGD CLI には、クラスターのコミットスコープ構成を表示できる新しいコマンド`pgd commit-scopes list`が含まれています。このコマンドは、ルールや適用されるグループを含むすべてのコミットスコープの詳細を提供します。これにより、データの一貫性設定の可視性と管理が向上します。 | |
| **PGD CLI-クラスターからノードを分離する** - PGD CLI には、新しいコマンド`pgd node part`が含まれ、アクティブなクラスターからノードを削除する合理的な方法を提供します。このコマンドは、 *node_part* SQL ファンクションを呼び出すプロセスを自動化し、その進行状況をレポートし、コマンドラインから直接ノード管理を簡素化します。 | |
| **リーダー DDL ロック** - 新しいロックタイプのリーダー DDL ロックは、DDL ステートメントをロックするためにデフォルトで使用されます。このロックは書き込みリーダーのみをロックし、マジョリティノードがロック操作に参加する必要はありません。`bdr.ddl_locking`構成パラメーターを調整することにより、古い動作を復元できます。 | |
| **監視ノードへのロールのレプリケーション** - ロールは、ノードグループに参加するとき、およびデータノードで作成または変更されたときに監視ノードに複製されるようになりました。これは、主に GUC 権限を処理する場合に役立ちます。 | |
| **PGD クラスターのコミットスコープをリストするコマンドを追加しました** - PGD クラスターのコミットスコープ情報をリストするコマンド *pgd commit-scopes list* を追加します。 | |
| **クラスターから PGD ノードを分離するコマンドを追加しました** - コマンド`pgd node part`を追加して、クラスターから PGD ノードを分離します。 | |

### 9.1.3 機能強化

| Description | Addresses |
|---|---|
| **デフォルト値を表示するように `pgd group get-option`および `pgd group show`コマンドを更新します** - `pgd group get-option`および`pgd group show`コマンドは、すべてのオプションのデフォルト値を表示するようになりました。`pgd group show`コマンドは、オプション値のソースも表示します。 | |
| **CLI コマンドの PGD Essential コンストレインを適用する** - CLI は、必要に応じてコマンドの PGD Essential コンストレインをチェックおよび適用します。詳細については、 *pg_upgrade* を参照してください。 | |
| **Raft スナップショットのエクスポート/インポートを改善して Raft を完全にリセット** - Raft コンセンサスメカニズムがスナップショットを処理する方法を改善しました。これにより、まれにデータの一貫性の問題が発生した場合、スナップショットを復元するとシステムの状態が完全にリセットされ、一貫した開始ポイントから操作をスムーズに再開できます。 | |
| **PGD の EPAS インターバルパーティションのサポート** - 新しい機能強化により、EDB Postgres Advanced Server AutoPartition によって作成されたパーティションは、マルチノード PGD 環境内で正しく動作できます。この修正により、パーティションが各ノードで一貫して作成され、すべてのノードで可視されるようになります。 | |
| **max_connections エラーレポートの向上** - 接続マネージャーは、max_connections 制限に到達したときに、より役立つ具体的なエラーメッセージを提供するようになりました。ユーザーには「申し訳ありませんが、既にクライアントが多すぎます」のような明確なメッセージが表示されるため、接続の問題の診断が非常に簡単になります。 | |
| **明示的なトランザクションブロックでテーブルリライトを生成するコマンドのサポート** - PGD は、トランザクションブロック内でテーブルリライトを生成する ALTER TABLE コマンドの実行をサポートするようになりました。単一のトランザクションブロックで複数のコマンドがサポートされています。これにより、すべての変更を単一のアトミック単位として適用することにより複雑な DDL 操作が簡素化され、複数のコマンドが同じテーブルをターゲットとしている場合にパフォーマンスの最適化を提供します。 | |
| **PGD との Postgres 互換性-フェーズ 5** - これは、 Postgres の互換性に関する将来の更新のためのプレースホルダーです。 | |

### 9.1.4 の変更

| Description | Addresses |
|---|---|
| **listen_address グループオプションの削除** - 構成を簡素化するために、 *listen_address* `オプションが *`pgd group set-option* `から削除されました。接続マネージャーは、 *`listen_addresses* GUC からの値を直接使用するようになり、以前はサポートされていなかったホスト名または複数のアドレスの設定に関する問題を解決します。 | |

### 9.1.5 バグ修正

| Description | Addresses |
|---|---|
| **`witness-only`グループの `pgd cluster verify`コマンド警告を修正しました** - `pgd cluster verify`コマンドは、`witness-only`グループの `Data nodes per group`チェックの警告を表示しました。このチェックは`witness-only`グループには適用されないため、ステータス `Ok`が表示されます。 | |
| **setup コマンドのデータディレクトリチェックを修正します** - `pgd node setup`コマンドは、指定されたデータ dir オプションに `PG_VERSION`ファイルの存在をチェックします。ただし、コードは誤って `pgdata`ではなく `bindir`オプションを使用しています。 | |
| **混合バージョンクラスターの修正** - 混合バージョンの PGD クラスターたとえば、Postgres 12 と Postgres 16 の組み合わせ、postgres のローリングアップグレード中に一時的に使用されるアーキテクチャでレプリケーションが失敗する可能性のあるバグを解決しました。以前は、新しいノードの DDL ステートメントは、古いノードで「認識されない構成パラメーター」エラーを発生してレプリケーションスロットの切断を引き起こす場合がありました。この修正により、ローリングアップグレード中または一時的な混合バージョン状態でレプリケーションが正しく機能するようになりました。 | |
| **ルーティングの変更に関する接続マネージャーの動作** - 接続マネージャーは、クラスタールーティング構成の変更たとえば、グローバルとルーティングなしの切り替えなど、次の接続を正しく処理するようになりました。これにより、予期しない動作が解決され、構成の更新後にクライアント接続が意図したとおりにルーティングされます。 | |
| **分割されたノードをリビルドする際の node_pkey エラーの修正** - 分割されたノードがリビルドされているときに一意のコンストレイン「node_pkey」エラーに違反したコンセンサスワーカーがクラッシュする可能性があるバグに修正が追加されました。この修正により、この問題が発生せずに`bdr_init_physical`を使用して分割されたノードを正常にリビルドできるようになりました。 | |
| **Witness-Only グループの PGD CLI cluster verify コマンドの修正** - pgd cluster verify コマンドは、データノードを含まない監視専用グループの警告を表示しなくなりました。これにより、コマンドがこれらのグループに誤ってフラグを立てた問題が解決され、検証出力がすべてのクラスタータイプで正確になるように保証されます。 | |

次のページに続く

表 5 – 前のページからの続き

| Description | Addresses |
|---|---|
| **遅延ノードを分離するための Raft 修正** - 以前はリーダーであった非常に遅延したノードがクラスターから分離された後、Raft は復旧に失敗しなくなりました。これは、Raft リーダーが「BDR node XXX not found」エラーでクラッシュする可能性があるバグを修正します。これには、以前は Raft 状態の手動で強制的なリセットが必要でした。現在、Raft はこのシナリオを正しく処理し、自動的に復旧し、クラスターの安定性を保証します。 | |
| **PGD CLI マルチグループセットアップの修正** - マルチグループクラスターの作成を防止した PGD CLI setup コマンドのバグが修正されました。以前は、ノードを 2 番目のグループに追加しようとすると、コマンドは役に立たないエラーで失敗していました。この修正により、コマンドが新しいグループのノードの作成を正しく処理するようになり、マルチグループクラスターのセットアップを成功させることができます。 | |
| **CLI の setup コマンド GUC Update の修正** - PGD CLI の setup コマンドは、リモートノードの Postgres 設定を更新しようとしても失敗しなくなりました。この修正は、クラスターのセットアップ中に発生した「不正な配列リテラル」エラーを解決し、初期化プロセスの信頼性を向上させます。 | |
| **Snowflake シーケンスの「Clock Has Moved Backwards」エラーの修正** - Snowflake シーケンスを使用すると、システムクロックが正常に機能していた場合でも、「クロックは後退しました」エラーを発生させる可能性のある競合状態が修正されました。この修正により、シーケンスが同時アクセスを正しく処理できるようになり、このエラーを防止し、Snowflake シーケンス生成の信頼性が向上します。 | |
| **pg_failover_slots によるクラッシュを修正する** - 復旧が進行中である場合、またはバックエンドがデータベースに接続されていない場合、CDC フックが no-op であることを確認します。これは、`pg_failover_slots`がアクティブな場合に必ず発生する可能性があります。 | |
| **接続マネージャー stat の stat ビューのクラッシュを修正します** - ビュー`bdr.stat_connection_manager_hba_file_rules`には、`pg_hba.conf`ファイル、特に"all"値の`Address/Mask`列でいくつかの解析問題がありました。 | |
| **ノード種類チェック** - PGD CLI には、ノード種類が適切に設定されていないノードの検証チェックがあります。このチェックは、ノード種類の設定を逃した以前のメジャーバージョンのアップグレードからのノードで警告を発します。 | |
| **`pgd node setup`コマンドでノード GUC の更新をスキップする** - `pgd node setup`は、PGD ノードのセットアップ中に Postgres 設定を検証します。 CLI が使用する`bdr_superuser`ロールは ALTER SYSTEM クエリーを実行できないため、コマンドはノード GUC の更新をスキップします。コマンドは Postgres 設定の検証を続行し、設定の変更が必要な場合は警告メッセージを表示します。 | |

## 9.2 EDB Postgres Distributed 6.0.2 release notes

リリース日 2025 年 6 月 25 日

これは、 EDB Postgres Distributed 6.0 の最初のパッチリリースであり、バグ修正と PGD 6.0 で導入された新機能の機能強化が含まれています。

### 9.2.1 バグ修正

| Description | Addresses |
|---|---|
| **`node setup`コマンドによって生成されたスロット名に関する問題を修正しました** - コマンドは、PGD スロット名と互換性のないアンダースコア `_`を含む UUID 値としてスロット名を生成しました。 | |
| **bdr.stat_connection_manager_get_stats() のセグメンテーションフォールトを修正** - ファンクションによって返されたメモリは、不正なメモリコンテキストに割り当てられました。 | |
| **BDR オブジェクトの GRANT チェックが EPAS オブジェクトタイプで動作することを確認します** - ユーザーに BDR 拡張オブジェクトの権限を付与できるかどうかを決定するメカニズムは、EPAS 固有のオブジェクトで失敗していました。これは PGD リリース 5.7.0 からの回帰であり、現在修正されています。 | 49649 |
| **`pgd cluster show`コマンドのチェックの名前が `Clock Skew`から `Clock Drift`に変更されました** - チェックの名前は、PGD の他の場所で使用される用語と一貫性を持って変更されます。 | |
| **CLI が `pgd cluster verify`コマンドの GUC の値の取得に失敗した場合の警告メッセージを追加しました** - CLI が何らかの理由で GUC の値を取得できない場合、コマンドはクラッシュする可能性があります。 | |

## 9.3 EDB Postgres Distributed 6.0.1 release notes

リリース日 2025 年 6 月 9 日

PGD 6 は、Postgres のよりシンプルで復元力の高い高可用性を提供します。従来のストリーミングレプリケーションでは、アップグレードと日常的なメンテナンスのためのダウンタイムが必要になることが多く、複雑なツールに依存しています。PGD は、ビルトインの論理レプリケーションベースのアーキテクチャでこれらの課題を解決し、アプリケーションを中断せずにオンラインアップグレードとメンテナンスを可能にし、チームが運用の変更中でもサービスをスムーズに実行し続けるのを支援します。また、シームレスなフェイルオーバーを提供し、外部プロキシ、ロードバランサー、またはコンセンサスシステムの必要性を排除します。

### 9.3.1 ハイライト

- **新しいビルトイン接続マネージャー**クライアント接続を正しいノードに自動的にルーティングし、アプリケーションアーキテクチャを簡素化し、動的なトポロジの変更をサポートし、ビルトインセッションプーラーと専用の読み取り/書き込みおよび読み取り専用ポートを含みます。外部ソフトウェ

アや複雑な構成は必要ありません。この新しいコンポーネントは、PGD 6 以降では使用できなくなった PGD プロキシを置き換えます。

- **事前定義されたコミットスコープ**ビルトインのトランザクション持続性プロファイルを使用して一貫性の選択を簡素化します。複雑なセットアップは必要ありません。システムカタログで定義され、すぐに使用できるスコープを使用して、パフォーマンスと保護の適切なバランスを選択します。
- **クラスターセットアップのための新しい CLI コマンド** pgd node setup コマンドにより、コマンドラインから直接初期クラスター作成とノードの追加が可能になりました。これにより、ユーザーは PGD を展開する方法の柔軟性が向上し、展開ツールを一貫した方法で標準化できます。

### 9.3.2 機能

| Description | Addresses |
|---|---|
| **ビルトイン接続マネージャー** - 接続のルーティングを自動的に処理し、非リーダーへの読み取り専用接続を強制できる新しいビルトイン接続マネージャー。 | |
| **CLI クラスターセットアップ** - PGD CLI では、クラスターの初期セットアップと、`pgd node setup`コマンドを使用してコマンドラインからノードを追加できるようになりました。 | |
| **グループ作成/参加時にシーケンス種類を設定します** - 古いバージョンで行っていたノードを作成するときではなく、bdr グループを作成/参加するときに、*bdr.default_sequence_kind* GUC に基づいて分散内のシーケンスを変換します。 | |
| **分散シーケンスの開始値を自動的に設定します** - galloc シーケンスの start 値を、ローカルシーケンスで最後に使用された後の有効な数値に設定します。この変更により、分散シーケンス、特に galloc を作成するときに、既に使用されているものに基づいて start 値を調整する必要がありません。 | |
| **自動同期と調整の有効化** - ノード状態の自動同期と調整がデフォルトで有効になりました。これは、ノードが自分の状態をリーダーノードと自動的に同期し、手動介入を必要とせずに差異を調整することを意味します。詳細は *pg_upgrade* をご覧ください。 | |
| **node_uuid 列を bdr.node および bdr.local_node に追加します** - node_uuid は、指定された名前のノードのインスタンスを一意に識別します。ランダムな node_uuid は、ノードの作成時に生成され、ノードの存続期間を通して一定のままです。 node_id 列は、ノード名ではなく node_uuid から取得されるようになりました。 今のところ、同じ名前のノードを再参加できる前に、ノードを完全に分離する必要があります。これは将来のリリースで緩和され、古いインスタンスの part_node プロセスが開始され、完了する前に再参加が許可される可能性があります。 当分の間、古い PGD バージョンからのアップグレードと、古い PGD ノードを使用するクラスターでの混合バージョン操作はサポートされていません。この制限は、将来のリリースで解決される予定です。 | |

次のページに続く

表 7 – 前のページからの続き

| Description | Addresses |
|---|---|
| **レプリケーションオリジンとスロットの命名スキームを変更する** - レプリケーションオリジンとスロット名はノード uuid を使用するようになったため、指定された名前のノードの特定の化身に対応します。同様に、グループ名の代わりにノードグループの uuid が使用されます。データベース名の代わりにデータベース名のハッシュが使用されます。 オリジン名とノード名は、ユーザーの観点から不透明な識別子として扱う必要があることに注意してください。これらの名前の構造に依存したり、これらが人間のオペレーターにとって特に意味があると期待すべきではありません。新しい命名スキームは次のとおりです。 **スロット命名規則** - ノードへの通常スロット => *bdr_node_<targetuuid>_<dbhash>* - ノードの参加スロット => *bdr_node_<targetuuid>_<dbhash>_tmp* - トップグループのグループスロット => *bdr_group_<topgroupuuid>_<dbhash>* - 任意の転送用スロット + リードからリード=> *bdr_node_<targetuuid>_<originidhex>_<dbhash>*-分析スロット => *bdr_analytics_<groupuuid>_<dbhash>*-デコードスロット => *bdr_decoder_<topgroupuuid>_<dbhash>* **オリジン命名規則** - 通常のオリジンからノードへの => *bdr_<originuuid>_<dbhash>* - ソースノードへの fwd オリジン => *bdr_<originuuid>_<sourceoidhex>_<dbhash>* | |
| **PGD Essential のシステムで許可されるノードグループの数の制限** - 任意の時点で、3 つ以下のノードグループ 1 つのトップグループと 2 つのサブグループが存在できることを確認します。制限を超えると、エラーが発生します。 | |
| **適用された PGD Essential 制限-データノード数** - PGD Essential クラスターが 4 つを超えるデータノードに参加することを許可しないでください。 | |
| **指定が指定された LSN に到達するまで待機する `bdr.wait_node_confirm_lsn()`ファンクションを追加しました** - `bdr.wait_node_confirm_lsn()`は、指定されたノードの confirmed_flush_lsn を参照します、それ以外の場合は、そのノードの`pg_replication_origin_progress()`を照会し、指定された LSN に到達するのを待機します。ノード。 | |
| **サブスクライバ専用ノードをデータノードグループに追加できるようになりました** - 以前のバージョンでは、サブスクライバ専用ノードはタイプ「サブスクライバ専用」のノードグループにのみ追加できました。 PGD 6 では、create_node を使用するときに node_kind='subscriber_only' を指定することにより、サブスクライバー専用ノードをデータノードグループに追加することもできます。 join_node_group は、データノードグループを使用して実行できます。 | |
| **`bdr.local_analytics_slot_name()` SQL ファンクションの追加** - 分析スロットの名前を返します。これは、分析機能が使用されているかどうかに関係なく、正しい名前を生成するだけです。 | |
| **Add node_uuid 列を `bdr.node_summary`ビューに** - bdr.node および bdr.local_node への node_uuid 列の追加を補完するために追加されました | |

### 9.3.3 機能強化

| Description | Addresses |
|---|---|
| **複数の競合行の解決** - `pk_exists`および`multiple_unique_conflicts`競合タイプは両方とも、競合の一部である古い行を削除することにより、複数の競合行を解決できるようになりました。 `multiple_unique_conflicts`はデフォルトで`update_if_newer`リゾルバーになるため、デフォルトでエラーをスローしなくなりました。 | |
| **`bdr.stat_activity`ビューの改善** - `backend_type`は、追加のプロセス ID なしで PGD ワーカーの一貫したワーカータイプを表示するようになりました。`wait_event_type`および`wait_event`には、一部のイベントの「拡張機能」を表示する代わりに、より多くの待機イベントが含まれるようになりました。また、接続管理関連の列が追加され、実際のクライアントアドレス/ポート、およびセッションが読み取り専用かどうかを表示します。 | |
| **PARTED ノードはクラスター内のすべてのノードから自動的に削除されます** - PGD 6.0.0 から、 bdr.part_node 機能が強化され、parted ノードのメタデータをクラスター内のすべてのノードから自動的に削除します。 - ローカルノードの場合、リモートノードに関する情報を含むすべてのノードのメタデータが削除されます。 - リモートノードの場合、その特定のノードのメタデータのみを削除します。したがって、このリリースでは-すべてのノードのグループスロットが PARTED ノードから発生するすべてのトランザクションに追いつくまで、ノードは PART_CLEANUP 状態にあります。ノードはに移動するとすぐに削除されるため、ノードは PARTED 状態に残りません。 PARTED 状態。 | |
| **`pgd node show` CLI コマンドの `--summary`および `--options`フラグ** - *--summary* **`および`**--options`フラグを`pgd node show`コマンドに追加して、`pgd node show`コマンドの出力をフィルタリングします。これは、他の`show`コマンドとの対称性も維持します。 | |
| **`pgd cluster verify` CLI コマンドで検証された GUC の追加** - *bdr.lock_table_locking* **`および`** *`bdr.truncate_locking* GUC を `pgd cluster verify`コマンドで検証された GUC のリストに追加します。 | |
| **不変ではない `ALTER TABLE... ADD COLUMN`呼び出しの制限が削除されました** - 不変ではない `ALTER TABLE... ADD COLUMN`呼び出しの制限が削除されました。 | |
| **論理結合中にロールとテーブルスペースを同期する** - スキーマが結合ソースノードから復元される前に、ロールとテーブルスペースが同期されるようになりました。すでに既存のロールまたはテーブルスペースまたは EPAS プロファイルが存在する場合、それらは結合ソースノードからの設定と同じ設定、パスワードなどを持つように更新されます。システムロールつまり initdb によって作成されたものは同期されません。 | |
| **`bdr.node_group_config_summary`ビューの紹介** - 新しい`bdr.node_group_config_summary`ビューには、有効値、有効値のソース、デフォルト値、値を継承できるかどうかを含むグループオプションに関する詳細情報が含まれています。これは`pg_settings`と同様の精神です。 | |

次のページに続く

表 8 – 前のページからの続き

| Description | Addresses |
|---|---|
| **リーダー DML ロック** - 新しいロックタイプのリーダー DML ロックは、DML をブロックする必要がある DDL ステートメントをロックするためにデフォルトで使用されます。このロックは書き込みリーダーのみをロックし、すべてのノードがロック操作に参加する必要はありません。`bdr.ddl_locking`構成パラメーターを調整することにより、古い動作を復元できます。 | |
| **run_on_*ファンクションでの bdr.xact_レプリケーションの無効化** - ファンクション`run_on_nodes`、*run_on_all_nodes*、および`run_on_group`は、デフォルトで`bdr.xact_replication`を`off`に設定するようになりました。 | |
| **デフォルトでレプリカ ID がフル** - `bdr.default_replica_identity`の`auto`値が REPLICA IDENTITY FULL に変更されました。この設定は、ノードのクラッシュと復旧にわたる挿入、更新、および削除間の競合検出における一部のエッジケースを防止します。 PGD グループが作成され、最初の PGD ノードのデータベースが空でない場合、つまりデータのあるテーブルがいくつかある場合、すべてのテーブルの REPLICA IDENTITY は`bdr.default_replica_identity`に従って設定されます。 | |
| **DDL 操作としてのテーブルスペースレプリケーションがサポートされています** - テーブルスペース操作 *CREATE/ALTER/DROP TABLESPACE* **が *DDL* 操作としてレプリケートされるようになりました。ユーザーが同じマシンで複数のノードを使用した構成を実行している場合、開発者オプション** *:ref: `pg_upgrade <Monitoring through SQL>*** を有効にする必要があります。 | |
| **CLI デバッグメッセージの改善** - ログメッセージのフォーマットを改善して、読みやすく Postgres ログメッセージと対称的です。 | |
| **`pgd cluster verify --settings` CLI コマンド出力の新しい列** - `pgd cluster verify --settings`コマンドの結果に`recommended_value`列を追加します。列は表形式出力には表示されませんが、JSON 出力には表示されます。 | |
| **CLI のソートされた出力の表示** - 表形式の出力を使用するコマンドの出力がリソース名でソートされるようになりました。複数のリソースを表示するコマンドは、各リソース列で出力を順番にソートします。 | |
| **加入者専用ノードのレプリケーション** - 加入者専用ノードは、大部分のデータノードにレプリケートされた後にのみデータを受信するようになりました。これには、特別な構成は必要ありません。その後、 bdr.standby_slot_names および bdr.standby_slots_min_confirmed オプションは、同様のフィジカルスタンバイ機能が pg_failover_slots 拡張機能および PG17+で提供されているため、削除されます。 | |
| **自動ノードの同期とリコンシリエーションはデフォルトで有効になっています** - GUC *pg_upgrade* はデフォルトでオフでしたが、6.0 ではデフォルトでオンになりました。この GUC 設定により、ノードがしばらくダウンしているときに、他のすべてのノードがこのノードに関して等しく自動的にキャッチされます。また、ダウンするノードによって孤立した準備されたトランザクションがある場合、それらは解決され、それらを作成したコミットスコープのルールに従ってアボートまたはコミットされます。 | |
| **非推奨の古い CLI コマンドを削除します** - 非推奨ですが、下位互換性のためにサポートされている古い PGD 5 以下の CLI コマンドを削除します。 | |

次のページに続く

表 8 – 前のページからの続き

| Description | Addresses |
|---|---|
| **コミットスコープロジックはデータノードでのみ実行されるようになりました** - 以前は、非データノードがコミットスコープロジックを処理しようとしましたが、無害ではありますが、混乱を招く可能性がありましたログメッセージ。 | |
| **ダンプおよびリストア操作の開始と停止を明示的にログに記録します** - これにより、ノードのクローン作成プロセスの可視性が向上し、考えられる問題のデバッグが支援されます。 | |

### 9.3.4 の変更

| Description | Addresses |
|---|---|
| **ルーティングがサブグループでデフォルトで有効になるようになりました** - ルーティングおよび拡張子 raft がデータグループデータノードを含むサブグループでデフォルトで有効になるようになりました。 | |
| **ファンクション `bdr.join_node_group` はトランザクションで実行できなくなります** - グループ結合をロールバックすることはできないため、べき等トランザクションの一部を形成できません。 | |
| **非推奨の `pause_in_standby` パラメーターがファンクション `bdr.join_node_group()` から削除されました** - `pause_in_standby` は、PGD 5.0.0 以降非推奨になっています。 `bdr.create_node()` を実行するときに論理スタンバイノードを指定する必要があります | |
| **BDR グローバルシーケンスは `UNLOGGED` として作成または設定できません** - ログに記録されていない BDR シーケンスは、サーバーのクラッシュに続いて予期しない動作を表示する場合があります。既存のログのない BDR シーケンスは、ログに記録されたものに変換できます。 | |

### 9.3.5 バグ修正

| Description | Addresses |
|---|---|
| **低下したクラスターでの CLI `pgd cluster show` コマンドの問題を修正します** - N ノードクラスターで 1 つのノードのみがアップおよび実行されている場合、 `pgd cluster show` コマンドはクロックドリフトのエラーで失敗しました。コマンドは、他のコンポーネント `health` および `summary` の有効な出力を結果ことができますが、 `clock-drift` の適切なエラーを報告します。 | |
| **存在しないノードが指定された場合の CLI `pgd node show` コマンドの問題を修正します** - 存在しないノードがコマンドに指定されている場合、 `pgd node show` コマンドはクラッシュしました。コマンドは、適切なエラーメッセージを表示して正常に失敗するように修正されました。 | |

次のページに続く

表 10 – 前のページからの続き

| Description | Addresses |
|---|---|
| **`pgd node upgrade`コマンドを使用してローリング Postgres アップグレード後に壊れたレプリケーションスロットの問題を修正します** - ノードのアップグレード中にライタの原点位置を親原点にマージします。 PGD 5 以前のライタのオリジン名は、インプレースアップグレード中に変更される可能性がある親オリジン ID にマップされます。 | 46412,48747 |
| **`pgd replication show` CLI コマンドのタイムスタンプ解析の問題を修正しました** - `pgd replication show`コマンドは、EPAS タイムスタンプをフォーマットするときにクラッシュしていました。 | |
| **分割するノードが存在しないグループに属する場合がある問題を修正しました** - 特定のノードを分割すると、その同じノードには、オリジンが既に分割されグループがドロップされたサブスクリプションを持つ場合があります。以前は、これにより PGD が中断されていましたが、修正されました。 | |
| **num_writers は正または-1 である必要があります** - bdr.alter_node_group_option() および bdr.alter_node_group_config() で使用される num_writers オプションは、正または-1 である必要があります。 | |
| **非一意のインデックスへの更新によるレプリケーションの破損を修正します** - 非一意のインデックスによるテーブルの更新が ERROR `concurrent INSERT when looking for delete rows`となり、レプリケーションを中断する場合を修正します。 | 43523,43802,45244,47815 |
| **アップグレード後の Raft リーダー選択タイムアウト/失敗を修正します** - 非推奨の GUC `bdr.raft_election_timeout`で設定されたカスタム値が代替品`bdr.raft_global_election_timeout`に適用されることを確認します | |
| **サブスクライバー専用ノードで無効なサブスクリプションが再有効化されていないことを確認します** - サブスクリプションの再構成中に、サブスクリプションに変更が必要な場合は、ユーザーが明示的に無効にされている可能性があるため、有効にしないでください。リーダーシップの変更がない場合、サブスクリプションの再構成をスキップします。 | 46519 |
| **サブスクライバー専用ノードは、 DDL の実行時にロックを取得しません** - サブスクライバー専用ノードは、 DDL の実行時にクラスターでロックを取得しようとしなくなりました。 DDL はローカルで実行され、他のノードにレプリケートされません。 | 47233 |
| **データベースシステムのシャットダウンのハングを修正** - 高速データベースシステムのシャットダウン中に WAL 送信者が終了しない原因となる可能性のある、ダウンストリームによる非トランザクション WAL メッセージの承認を修正しました。 | 49022 |
| **bdr_init_physical のデッドロックの問題を修正しました** - bdr_init_physical の不要なノードデータのクリーニングと同時監視クエリー間のデッドロックを修正しました。 | 46952 |
| **新しいクラスターノードの一貫性の問題を修正しました** - クラスターに参加する新しいノードが、すべてのデータノードに対するレプリケーションの進行状況を取得する前に CATCHUP フェーズを終了する場合の問題を修正しました。これにより、新しいノードがクラスターと同期しない場合があります。 | |

次のページに続く

表 10 – 前のページからの続き

| Description | Addresses |
| --- | --- |
| **CREATE SEQUENCE 警告に正しいシーケンスタイプが表示されることを確認します** - 場合によっては、シーケンスが実際には`snowflakeid`であるときに、警告メッセージが`timeshard`を参照していました。 | |

# 第10章　Known issues and limitations

## 10.1 既知の問題

これらは EDB Postgres Distributed 6.1 の現在既知の問題です。これらの既知の問題は PGD のチケットシステムで追跡され、将来のリリースで解決される予定です。

- マテリアライズドビューのレプリケーションは、新しく作成されたマテリアライズドビューでのみ機能します。6.1.0 にアップグレードする前に作成されたマテリアライズドビューは、アップグレード後に複製されません。この問題を防ぐには、次のいずれかを行うことができます。
- PGD 6.1.0. にアップグレードする前に、各ノードでマテリアライズドビューを手動で作成します。- または - PGD 6.1.0. にアップグレードした後、マテリアライズドビューを削除して作成して、複製されるようにします。
- `update_origin_change` 競合のリゾルバーが `skip` に設定され、`synchronous_commit=remote_apply` が使用され、同じ行の同時更新が 2 つの異なるノードに繰り返し適用される場合、PGD ライタとのデッドロックが原因で更新ステートメントの 1 つがハングする場合があります。Conflicts で述べたように、 `skip` は `update_origin_change` 競合のデフォルトのリゾルバーではなく、この組み合わせは運用環境で使用することを目的としていません。そのノードへの到着順序に基づいて、2 つの競合する更新のうちの 1 つを破棄します。これにより、発散クラスターが発生する可能性があります。`skip` 競合リゾルバーの使用を選択するまれな状況では、`remote_apply` モードの使用に関する問題に注意してください。
- Decoding Worker 機能は、CAMO/Eager/Group Commit では動作しません。CAMO/Eager/Group コミットを使用したインストールでは、`enable_wal_decoder` を無効のままにする必要があります。
- Lag Control は、完全に分離されたノードではいかなる方法でもコミット遅延を調整しません。これは、他のすべてのノードが到達不能または動作していない場合です。少なくとも 1 つのノードが接続すると、レプリケーションラグコントロールはその作業を取得し、PGD コミット遅延を再度調整します。
- 時間ベースのラグ制御の場合、PGD は現在、履歴適用レートに基づいた推定キャッチアップ時間ではなく、コミットタイムスタンプによって測定されたラグタイムを使用しています。
- CAMO ペアの CAMO パートナーを変更することは現在不可能です。ペアを追加または削除することのみが可能です。ペアを追加または削除するには、Postgres の再起動や構成のリロードも必要ありません。
- グループコミットは CAMO と組み合わせることはできません。
- Eager Replication を使用するトランザクションは、まだ DDL を実行できません。TRUNCATE コマンドが使用できます。
- パラレル適用は、グループコミットとの組み合わせで現在サポートされていません。グループコミット

を使用するときは、(a) bdr.alter_node_group_option を使用してノードグループの num_writers を 1 に設定するか、(b) GUC bdr.writers_per_subscription を使用して、必ず無効にしてください。*Configuration of generic replication* を参照してください。

- 現在、コミットスコープの変更または削除に対する保護はありません。同時に変更または削除されるコミットスコープでトランザクションを実行すると、トランザクションを適用しようとしているダウンストリームノードでのエラーが原因で、トランザクションブロックまたはレプリケーションが完全に停止する可能性があります。変更または削除する前に、特定のコミットスコープを使用するトランザクションが終了していることを確認してください。
- PGD CLI は、クラスターから分離されたノードにまだ接続している場合、クラスターの状態に関する古いデータを返すことができます。*pgd-cli-config.yml* ファイルを編集するか、 *--dsn* 設定を変更して、クラスター内のアクティブなノードのみが接続用にリストされるようにします。

コミットスコープを安全に変更するには、 bdr.alter_commit_scope を使用します。

- シリアル化可能なトランザクションで実行される DDL は、エラー `ERROR: could not serialize access due to read/write dependencies among transactions` が発生する場合があります。回避策は、シリアル化可能なトランザクションの外部で DDL を実行することです。
- EDB Postgres Advanced Server 17 データ型 *BFILE* は、現在サポートされていません。これは、`BFILE` がデータベースに保存されるファイル参照であり、ファイル自分自身はデータベースの外部に保存され、レプリケートされないためです。

## 10.2 制限事項

展開を計画するときは、次の EDB Postgres 分散 PGD デザインの制限を考慮してください。

### 10.2.1 ノード

- PGD は、適切なハードウェアとネットワークを想定して、数百のノードを実行できます。ただし、メッシュベースの展開の場合、通常は 1 つのクラスターで 48 を超えるノードを実行することはお勧めしません。48 ノード制限を超えて追加の読み取りスケーラビリティが必要な場合は、メッシュネットワークへの接続を追加せずにサブスクライバ専用ノードを追加できます。
- PGD のコンセンサスメカニズムにフォールトトレランスを提供するために、グループ内のノードの最小推奨数は 3 です。2 つのノードのみの場合、ノードの 1 つが応答しない場合、コンセンサスは失敗します。分散シーケンス生成などの一部の PGD 操作では、コンセンサスが必要です。EDB Postgres Distributed が使用するコンセンサスメカニズムの詳細については、*Architecture overview* を参照してください。

### 10.2.2 単一のインスタンス上の複数のデータベース

同じ Postgres インスタンス上の複数のデータベースで PGD を使用することのサポートは、PGD 5 から **非推奨** であり、PGD 6 ではサポートされなくなります。製品の機能を拡張するにつれて、操作的および機能的に導入される追加の複雑さはありません。マルチデータベースデザインではより長く実行可能です。

これがベストプラクティスであり、PGD インスタンスごとに 1 つのデータベースのみを構成することをお勧めします。

現在、CLI や Connection Manager などのツールがその推奨事項を成文化しています。

単一のインスタンスで最大 10 のデータベースをホストすることは引き続き可能ですが、そうすると多くの差し迫ったリスクと現在の制限が発生します。

- PGD 構成の変更が必要な場合は、データベースごとに管理コマンドを実行する必要があります。これを行うと、潜在的な不一致とエラーのリスクが増加します。
- 各データベースを個別に監視する必要があり、オーバーヘッドを追加します。
- 接続マネージャーは、データベースレベルではなく Postgres インスタンスレベルで動作します。つまり、リーダーノードはすべてのデータベースで同じです。
- データベースを追加するごとに、サーバーのリソース要件が増加します。それぞれが、レプリケーションを維持する独自のワーカープロセス、たとえば、論理ワーカー、WAL センダー、WAL レシーバーなどを必要とします。それぞれは、レプリケーションクラスター内の他のインスタンスへの独自の接続セットも必要です。これらのニーズは、すべてのデータベースのパフォーマンスに重大な影響を与える可能性があります。
- CAMO やグループコミットなどの同期レプリケーション方法は、予想通りに動作しません。Postgres WAL はデータベース間で共有されるため、同期コミット確認は任意のデータベースから取得できますが、必ずしも正しいコミットの順序ではありません。
- CLI 統合は、1 つのデータベースを想定しています。

### 10.2.3 耐久性オプショングループコミット/ CAMO

PGD 耐久性オプションの動作にはさまざまな制限があります。これらの制限は、Group Commit と CAMO 間の相互作用、およびそれらが *WAL decoder* や transaction streaming などの PGD 機能とどのように相互作用するかの製品です。

また、古い同期レプリケーションとの相互運用性、明示的な 2 フェーズコミットとの相互運用性、およびコミットスコープルール内でサポートされていない組み合わせにも制限があります。

次の制限は、コミットスコープの使用と、それらが有効にするさまざまな耐久性オプションに適用されます。

### 10.2.4 一般的な耐久性の制限

- Legacy synchronous replication は、CAMO、Eager 、および Group Commit で使用されるものとは異なるトランザクション確認のメカニズムを使用します。この 2 つは互換性がないため、一緒に使用しないでください。Group Commit、CAMO、または Eager を使用するときは、`synchronous_standby_names` で PGD ノードが構成されていないことを確認します。
- Postgres の 2 フェーズコミット 2PC トランザクションつまり *PREPARE TRANSACTION* は、CAMO、Group Commit、または Eager では使用できません。これらの機能は下で 2 フェーズコミットを使用するためです。

### 10.2.5 グループコミット

Group Commit は、グループ内のノードを介した構成可能な同期コミットを有効にします。この機能を使用する場合、次の制限を考慮してください。

- グループコミットを使用する場合、すべての DDL を実行できるわけではありません。サポートされていない DDL を使用する場合、警告がログに記録され、トランザクションのコミットスコープがローカルに設定されます。サポートされている DDL 操作は次のとおりです。
  - 非コンカレント `CREATE INDEX`
  - 非コンカレント `DROP INDEX`
  - 個々のテーブルまたはインデックスの非コンカレント `REINDEX`
  - `CLUSTER` 単一のリレーションまたはインデックスのみ
  - `ANALYZE`
  -
  -
  -
  -
  -
- 明示的な 2 フェーズコミットは、既に 2 フェーズコミットを使用しているため、グループコミットではサポートされていません。
- 同じトランザクションでの異なるコミット決定オプションの組み合わせ、または同じトランザクションでの異なる競合解決オプションの組み合わせは、サポートされていません。
- 現在、Raft のコミット決定は非常に遅く、非常に低い TPS を生成します。`eager` 競合解決設定でのみこれらを使用して、PGD 4 以前の Eager All-Node Replication 動作を取得することをお勧めします。

### 10.2.6 熱心な

Eager は、グループコミットを介して利用できます。競合する可能性のあるトランザクションを積極的に中止することにより、競合を回避します。これには、グループコミットと同じ制限が適用されます。

Eager では、 `NOTIFY` SQL コマンドまたは `pg_notify()` ファンクションは許可されていません。 `LISTEN` または `UNLISTEN` も許可されません。

## 10.3 CAMO

Commit At Most Once CAMO は、アプリケーションが複数回コミットするのを防ぐことを目的とした機能です。この機能を使用する場合、計画時に次の制限を考慮してください。

- CAMO は、オリジンノードで最近失敗した COMMIT の結果を照会するように設計されています。切断が発生した場合、アプリケーションは CAMO パートナーからトランザクションステータスを要求する必要があります。障害後、ステータスを要求する前に、遅延ができるだけ少ないことを確認します。アプリケーションは、15 分を超えて保存される CAMO 決定に依存してはなりません。

- 再起動などの結果、アプリケーションが割り当てられたグローバル識別子を忘れた場合、それを回復する簡単な方法はありません。したがって、アプリケーションはシャットダウンする前に、未処理のトランザクションが終了するのを待機することをお勧めします。
- クライアントが適切なチェックを適用するには、CAMO によって保護されたトランザクションを、暗黙的なトランザクション制御を備えた単一のステートメントにすることはできません。また、CAMO は、トランザクション制御プロシージャ、またはトランザクションを開始または終了しようとする DO ブロックで使用することはできません。
- CAMO はコミットステータスを解決しますが、コミット時の保留中の通知を解決しません。CAMO、NOTIFY SQL コマンドまたは pg_notify() ファンクションは許可されていません。 LISTEN または UNLISTEN も許可されません。
- 変更を再生するときに、CAMO トランザクションは他のトランザクションと同様に競合を検出する場合があります。タイムスタンプ競合検出が使用される場合、CAMO トランザクションは、プリペアオンザオリジンノードのタイムスタンプを使用します。これは、トランザクションがオリジンノード自分自身で可視される前です。
- CAMO は現在トランザクションストリーミングと互換性がありません。CAMO を使用する予定がある場合は、必ずトランザクションストリーミングを無効にしてください。このオプションはグローバルにまたは PGD ノードグループで構成できます。*Transaction streaming configuration* を参照してください。
- CAMO は現在デコードワーカーと互換性がありません。CAMO を使用する予定がある場合は、デコードワーカーを有効にしないようにしてください。このオプションは PGD ノードグループで構成できます。*Decoding worker disabling* を参照してください。
- CAMO を使用する場合、すべての DDL を実行できるわけではありません。サポートされていない DDL を使用する場合、警告がログに記録され、トランザクションのコミットスコープはローカルのみに設定されます。サポートされている DDL 操作は次のとおりです。
  - 非コンカレント CREATE INDEX
  - 非コンカレント DROP INDEX
  - 個々のテーブルまたはインデックスの非コンカレント REINDEX
  - CLUSTER 単一のリレーションまたはインデックスのみ
  - ANALYZE
  -
  -
  -
  -
  -
- 明示的な 2 フェーズコミットは、既に 2 フェーズコミットを使用しているため、CAMO ではサポートされていません。
- CAMO トランザクションのみを DEGRADE TO 句と組み合わせて、可用性が低下した場合に非同期オペレーションに切り替えることができます。

### 10.3.1 混合 PGD バージョン

PGD は、アップグレードプロセス中に PGD の混合バージョンが動作できるようにすることにより、*enable rolling upgrades of PGD* に開発されました。私たちは、ユーザーがアップグレード中にのみ混合バージョンを実行し、アップグレードが開始されたら、そのアップグレードを完了することを想定しています。アップグレード時を除き、PGD の混合バージョンの実行はサポートされていません。

### 10.3.2 その他の制限

この非包括的なリストには、予想されるデザインによる他の制限が含まれています。将来的にそれらを解決する予定はありません。展開を計画するときは、次の制限を考慮してください。

- `galloc` シーケンスは、ロールバックしたトランザクションでシーケンスを作成し、同じ名前で再度作成する場合、一部のチャンクをスキップする場合があります。チャンクのスキップは、DDL レプリケーションがアクティブでないときにシーケンスを作成して削除し、DDL レプリケーションがアクティブなときに再度作成する場合にも発生する可能性があります。順序保証に違反しないため、問題の影響は軽度です。シーケンスは、一部の最初のチャンクのみをスキップします。また、回避策として、シーケンスの開始値を `bdr.alter_sequence_set_kind()` ファンクションの引数として指定できます。

# 第11章　Upgrading

PGD と Postgres は密接に関連していますが、これらは別のアップグレードパスを持つ別の製品です。このセクションでは、PGD と Postgres の両方をアップグレードする方法について説明します。

## 11.1 PGD のアップグレード

EDB Postgres Distributed は、柔軟なプラットフォームです。これは、アップグレードパスが PGD のインストール方法に大きく依存することを意味します。

- ***Upgrading PGD clusters manually*** —PGD クラスターを手動でインストールして構成した場合、マイナーとメジャーの両方のバージョン間でクラスターを移動できます。
- ***Supported PGD upgrade paths*** —サポートされているいくつかのアップグレードパスを使用できます。
- ***PGD compatibility*** — PGD 3.x または 4.x から PGD 5.x 以降にアップグレードする場合、バージョン間の互換性の変更を理解する必要があります。

## 11.2 Postgres または Postgres と PGD メジャーバージョンのアップグレード

- ***In-place Postgres major version upgrades*** — `pgd node upgrade` を使用して、1 つ以上のノードで Postgres バージョンまたは Postgres と PGD メジャーバージョンを手動でアップグレードする方法。
- ***Performing a Postgres major version rolling upgrade on a PGD cluster*** —クラスターで Postgres のメジャーバージョンのアップグレードを実行する方法。

## 11.3 その他のアップグレード

- ***Application schema upgrades*** —PGD で複数の分散サーバーを実行している場合、アプリケーションのスキーマを安全にアップグレードするためのガイド。

### 11.3.1 Upgrading PGD clusters manually

EDB Postgres Distributed は複数のソフトウェアコンポーネントで構成されているため、アップグレード戦略はアップグレードされるコンポーネントに部分的に依存します。

通常、**ローリングアップグレード**と呼ばれるアプローチを使用して、ほぼゼロのダウンタイムでクラスターをアップグレードできます。このアプローチを使用して、ノードは 1 つずつアップグレードされ、アプリケーション接続は既にアップグレードされたノードにスイッチオーバーされます。

すべてのノードを停止し、すべてのノードでアップグレードを実行して、クラスター全体を再起動することもできます。このアプローチは、標準の PostgreSQL セットアップと同じです。すべてのノードを同時にアップグレードするこの戦略は、ソフトウェアの混合バージョンでの実行を回避するため、最も簡単です。ただし、ダウンタイムが発生するため、何らかの理由でローリングアップグレードを実行できない場合を除き、お勧めしません。

EDB Postgres Distributed クラスターをアップグレードするには

1. アップグレードを計画します。
2. アップグレードの準備をします。
3. サーバーソフトウェアをアップグレードします。
4. アップグレードを確認および検証します。

### アップグレード計画

各ノードをアップグレードするには、主に 2 つの方法があります。

- ノードを新しいソフトウェアバージョンにアップグレードします。**ローリングサーバーソフトウェアのアップグレード** を参照してください。
- ノードを、新しいバージョンがインストールされたものに置き換えます。**ノード参加を使用したローリングアップグレード** を参照してください。

これらのアプローチは両方ともローリング方法で使用できます。

### ローリングアップグレードの考慮事項

クラスターでローリングアップグレードが実行されている間、ソフトウェアの混合バージョンがクラスターで実行されています。たとえば、ノード A の PGD 4.3.6 があり、ノード B およびノード C が 5.6.1 であるとします。この状態では、レプリケーションとグループ管理は最も古いバージョンこの例では 4.3.6 のプロトコルと機能を使用するため、新しいバージョンが提供するプロトコルの変更を必要とする新機能は無効になっています。すべてのノードが同じバージョンにアップグレードされると、新機能が有効になります。

同様に、WAL デコーダー対応ノードを含むクラスターがローリングアップグレードを実行している場合、高いバージョンの PGD ノードの WAL デコーダーは、より高い pg 論理バージョンで *logical change records (LCRs)* を生成します。PGD ノードの下位バージョンの WAL デコーダは、下位の pg 論理バージョンの LCR を生成します。その結果、プロトコルバージョンの不一致のため、PGD ノードの上位バージョンの WAL 送信者は LCR を使用することは想定されていません。PGD ノードの下のバージョンでは、WAL 送信者は引き続き LCR を使用できます。すべての PGD ノードが同じ PGD バージョンになると、WAL 送信者は LCR を使用します。

ローリングアップグレードは、以前のリリースのすべてのノードを含むクラスターで開始されます。次に、すべてのノードが新しいリリースになるまで、一度に 1 つのノードを新しいリリースにアップグレードして続行します。同時に実行されるソフトウェアのバージョンは 3 つまでです。別のアップグレードを開始する前に、すべてのノードを完全にアップグレードした状態でアップグレードを完了する必要があります。

ビジネスリスクを削減するために追加の注意が必要な場合、アップグレードの実行により多くの時間が必要になる場合があります。最大限の注意を払い、実稼働システムのアップグレードに必要な時間を削減するために、最初に別のテスト環境でアップグレードを実行することをお勧めします。

アップグレードを完了するために絶対に必要な時間を超えて、ソフトウェアの混合バージョンで実行しないでください。*pgd commit-scopes list* コマンドを使用して、クラスター内のバージョンを確認できます。

混合バージョンでの実行が長くなると、問題が発生する可能性が高くなり、それらの診断と解決が難しくなります。ビジネスのオフピーク時間に、短期間にアップグレードすることをお勧めします。

ローリングアップグレードを使用してソフトウェアのメジャーバージョンをアップグレードすることはできますが、1 つのクラスターでの PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server の混合はサポートされていません。したがって、このアプローチを使用して Postgres バリアントを変更することはできません。

**警告**

EDB Postgres Distributed のダウングレードはサポートされていません。クラスターを手動で再構築する必要があります。

### ローリングサーバーソフトウェアのアップグレード

ローリングアップグレードとは、クラスターを停止せずにクラスター内の各ノードで **ローリングサーバーソフトウェアのアップグレード** がシークエンシャルにアップグレードされる場合です。各ノードは一時的にクラスターへの参加を停止され、サーバーソフトウェアがアップグレードされます。更新されると、クラスターに返され、不在中のクラスターのアクティビティに追いつきます。

実際の手順は、Postgres コンポーネントが新しいメジャーバージョンにアップグレードされるかどうかによって異なります。

アップグレードプロセス中に、現在アップグレードされていないノードにアプリケーションをスイッチオーバーして、アプリケーションのデータベースの継続的な可用性を提供できます。

### ノード参加を使用したローリングアップグレード

サーバーソフトウェアをアップグレードするもう 1 つの方法は、新しいノードをクラスターに参加させ、後で古いバージョンのソフトウェアを実行している既存のノードのいずれかをドロップすることです。

このアプローチの場合、手順は常に同じです。ただし、ノード参加が含まれるため、大規模なデータ転送が必要になる可能性があります。

すべてのノードが Postgres の新しい同じリリースにアップグレードされるまで、新しい Postgres バージョンでのみ使用可能な機能を使用しないように注意してください。これは、Postgres の新しいリリースに追加された新しい DDL 構文に特に当てはまります。

**注釈**

*bdr_init_physical* は、ソースノードのバイトごとのコピーを作成するため、メジャー Postgres バージョンから別のバージョンへのアップグレード中に使用できません。実際、現在 `bdr_init_physical` では、ソースの PGD バージョンと参加ノードでもまったく同じであることが必要です。新しいノードに参加する方法によるローリングアップグレードには使用できません。代わりに、論理結合を使用します。

### CAMO 対応クラスターのアップグレード

CAMO 対応クラスターをアップグレードするには、CAMO グループを 1 つずつアップグレードする必要がありますが、アップグレードするグループの CAMO 保護を無効にし、新しい commit scope ベースの設定を使用して再構成する必要があります。

CAMO ペアを構成する 2 つの BDR ノードを PGD 5.0 にアップグレードするには、次のアプローチをお勧めします。

1. 2 つのノードのいずれか上のトランザクションで `bdr.enable_camo` が `off` のままであることを確認するか、クライアントを 2 つのノードからリダイレクトします。CAMO を使用しようとするときに CAMO ペアリングを削除すると、エラーが発生し、それ以上のトランザクションが防止されます。
2. `bdr.camo_origin_for` および `bdr.camo_parter_of` をリセットすることにより BDR 3.7.x からアップグレードする場合、または `bdr.remove_camo_pair` BDR 4.x のいずれかにより CAMO の構成を解除することにより、ペアのカップリングを解除します。
3. 2 つのノードを PGD 5.0 にアップグレードします。
4. 2 つのノードの専用ノードグループを作成し、そのノードグループに移動します。
5. このノードグループ、および CAMO を使用するノードのペアの commit scope を作成します。
6. `default_commit_scope` を設定するか、セッションまたはトランザクションに `bdr.enable_camo` の代わりに `bdr.commit_scope` を明示的に設定するようにクライアントを変更することにより、CAMO 保護を再度アクティブにします。
7. 必要に応じて、クライアントが CAMO 保護ノードに再度接続できるようにします。

### アップグレードの準備

ソフトウェアの各メジャーリリースには、以前のリリースとの互換性に影響を与える可能性のあるいくつかの変更が含まれています。これらは、Postgres 構成、展開スクリプト、および PGD を使用するアプリケーションに影響を与える可能性があります。アップグレードの前に、これらの変更を検討し、必要な調整を行うことをお勧めします。

*EDB Postgres Distributed 6 release notes* およびバージョン固有のアップグレードノートで記載されている個々の変更を参照してください。

### サーバーソフトウェアのアップグレード

個々のノードに分散された EDB Postgres のアップグレードは、インプレースで発生します。BDR 拡張機能をアップグレードするときにバックアップと復元を行う必要はありません。

### BDR 拡張機能のアップグレード

BDR 拡張機能のアップグレードプロセスは、次の高レベルの手順で構成されています。

#### ノードをフェンスする

アップグレードが完了するまで、アップグレードされるノードが書き込みリーダーにならないように、アップグレードを開始する前にノードをフェンスする必要があります。

#### Postgres を停止します

バイナリパッケージのアップグレード中に、通常は最初に実行中の Postgres サーバーを停止するのが最善です。そうすることで、アップグレード中に予期せず再起動が発生した場合に、混合バージョンがロードされなくなります。

#### アップグレードパッケージ

アップグレードの最初の手順は、新しいバージョンの BDR パッケージをインストールすることです。このインストールでは、新しいバイナリと拡張 SQL スクリプトの両方をインストールします。この手順はオペレーティングシステムに固有です。

#### Postgres を開始する

パッケージがアップグレードされたら、Postgres インスタンスを起動できます。BDR 拡張機能は、新しいバイナリが拡張機能の古いバージョンを検出すると、起動時にアップグレードされます。

#### ノードのフェンスを解除します

ノードのアップグレードが完了した後、ノードのフェンスを解除できます。

PGD 6.1 にアップグレードする前に、 PGD 4 クラスターは PGD 4.4 を実行している必要があります。

PGD 4.4 から PGD 6.1 にアップグレードするには、Harp プロキシから Connection Manager に移動する追加の手順が必要です。詳細については、実際の例**実際の例 *PGD 4* から *PGD 6.1* へのアップグレード** を参照してください。

PGD 6.1 にアップグレードする前に、 PGD 5 クラスターは PGD 5.9 を実行している必要があります。

PGD 5.9 から PGD 6.1 にアップグレードするには、 PGD プロキシから Connection Manager に移動する追加の手順が必要です。 詳細については、実際の例**実際の例 *PGD 5* から *PGD 6.1* へのアップグレード** を参照してください。

Postgres のインプレースアップグレードのプロセスは、Postgres の新しいマイナーバージョンにアップグレードするか、Postgres の新しいメジャーバージョンにアップグレードするかによって異なります。

#### マイナーバージョン Postgres のアップグレード

Postgres の新しいマイナーバージョンへのアップグレードは、***BDR* 拡張機能のアップグレード** に似ています。通常、Postgres の停止、パッケージのアップグレード、および Postgres の再起動だけが必要なすべてです。

ただし、特定のマイナーバージョンのアップグレードには、インデックスの再作成などのより多くの手順が推奨される場合があります。アップグレード先の Postgres のバージョンのリリースノートを参照してください。

### メジャーバージョン Postgres のアップグレード

Postgres の新しいメジャーバージョンへのアップグレードは、マイナーバージョンへのアップグレードよりも複雑です。

EDB Postgres Distributed は、`pgd node upgrade` コマンドラインユーティリティを提供し、*in-place Postgres major version upgrades* を実行するために使用できます。

> **注釈**
>
> Postgres、 BDR 拡張機能、その他を含むソフトウェアの新しいメジャーバージョンにアップグレードする場合、アプリケーションがアップグレードするソフトウェアのターゲットバージョンと互換性があることを確認することが常に重要です。

### アップグレードのチェックと検証

PGD ノードをアップグレードした後、バイナリの現在のバージョンを確認できます。

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

ノードをアップグレードした後は常に monitoring をチェックして、アップグレードされたノードが予想通りに動作していることを確認します。

### PGD 5- PGD プロキシから接続マネージャーへの移動

次の手順を使用して、PGD プロキシから接続マネージャーに移動します。

1. PGD 5.9 ノードのいずれかから次のクエリーを実行して、すべてのユーザーパスワードの SCRAM ハッシュがすべてのノードで同じであることを確認します。

- ```
  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);
  ```

-
  ```

**これを実行した後、新しいユーザーは 5.9 に追加されません。追加された場合は、クエリを再度実行します。上記のブロックはパスワードを変更しません。すべてのノードのクラスターで SCRAM ハッシュが同じであることを保証するだけです。 .. ::**

2. 書き込みリーダーにならないように、*pgd node <node-name> set-option route_fence true* を使用してこのクラスター内のノードをフェンスします。 3. GUC *bdr.enable_builtin_connection_manager* を*true*に有効にします。 4. サーバーを再起動します。 5. サーバーで実行されている PGD プロキシを停止します。 6. サーバーを再起動します。デフォルトのポートで実行されている接続マネージャーで起動します。プロキシの読み取りポートと書き込みポートが異なる場合、Connection Manager ポートの読み取りポートと書き込みポートは、 *bdr.alter_node_group_option()* によってプロキシと同じになるように変更できます。 7. ノードのフェンスを解除します。 このノードは、ユーザーからの接続を受け入れ、接続マネージャーを介して書き込みリーダーにルーティングできるようになりました。 8. クラスター内のノードごとにこれを繰り返します。 これにより、すべてのノードが接続マネージャーを介してルーティングされるようになります。 9. PGD 5 から PGD 6.1 へのメジャーバージョンのアップグレードも実行する場合は、 *rolling upgrade steps* に進みます。

### 11.3.2 Supported PGD upgrade paths

#### バージョン 6.1.0 へのアップグレード

EDB Postgres Distributed は semantic versioning を使用します。同じメジャーバージョン内のすべての変更には下位互換性があるため、アップグレード時のリスクが低下し、アップグレードターゲットとして新しいマイナーまたはパッチリリースを選択できます。

次のアップグレードパスは PGD 6.1 でサポートされています。

| Source version | Target version |
|---|---|
| 4.4.0 | 6.1.0 |
| 5.9.0 | 6.1.0 |
| 6.0.2 | 6.1.0 |

### 11.3.3 Compatibility changes

私たちはこの情報をできるだけ効果的かつ正確に提供するよう努めており、すぐに更新される予定です。

### 11.3.4 Application schema upgrades

EDB Postgres Distributed のアップグレードと同様に、アプリケーションスキーマをアップグレードするには 2 つのアプローチがあります。より簡単なオプションは、影響を受けるすべてのアプリケーションを停止し、スキーマのアップグレードを実行し、新しいスキーマバリアントを使用するようにアップグレードされたアプリケーションを再起動することです。このアプローチでは、ある程度のダウンタイムが発生します。

このダウンタイムを排除するために、 EDB Postgres Distributed は、ローリングアプリケーションスキーマのアップグレードを実行するための便利なツールを提供します。

次の推奨事項とヒントは、クラスターに対するアプリケーションスキーマのアップグレードの影響を軽減します。

### ローリングアプリケーションスキーマのアップグレード

デフォルトでは、 DDL はすべてのノードに自動的に送信されます。DDL replication で説明しているように、この動作は手動で制御できます。このアプローチを使用して、ノード全体にデータベーススキーマ間の差異を作成できます。

PGD は、ノード間のわずかな違いがあってもレプリケーションを続行できるように設計されています。これらの機能は、ダウンタイムなしでアプリケーションスキーマを移行できるように、またはロジカルスタンバイノードでレポートまたはテストをできるように設計されています。

これを実稼働クラスターで正しく動作させるには、注意的なスクリプトが必要です。広範なテストをお勧めします。

詳細は、 Replicating between nodes with differences を参照してください。

1 つのノードが新しいテーブルを追加する DDL を実行すると、最新の DDL をまだ受け取っていないノードは追加のテーブルを処理する必要があります。これを考慮して、ローリングスキーマアップグレードの適切な設定は、`target_table_missing` 競合が発生した場合に `skip` リゾルバーを適用するようにすべてのノードを構成することです。ノードにテーブルを追加する前に、この構成を実行します。この設定は永続的なものです。

次のクエリーを **各ノードで個別に** 実行します。 `node1` を実際のノード名に置き換えます。

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

1 つのノードがテーブルに列を追加する DDL を実行すると、最新の DDL をまだ受け取っていないノードが追加の列を処理する必要があります。これを考慮して、ローリングスキーマアップグレードの適切な設定は、`target_column_missing` 競合が発生した場合に `ignore` リゾルバーを適用するようにすべてのノードを構成することです。1 つのノードに列を追加する前にこれを実行します。この設定は永続的なものです。

次のクエリーを **各ノードで個別に** 実行します。 `node1` を実際のノード名に置き換えます。

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

1 つのノードがテーブルから列を削除する DDL を実行すると、最新の DDL をまだ受け取っていないノードが不足している列を処理する必要があります。この状況により、`use_default_value` リゾルバーを使用する `source_column_missing` 競合が発生します。したがって、NULL を受け入れず、DEFAULT 値を持たない列には、2 段階のプロセスが必要です。

1. NOT NULL 制約を削除するか、すべてのノードの列の DEFAULT 値を追加します。
2. カラムを取り外します。

コンストレインは、ローリング方法で削除できます。現在、一度に 1 ノードずつ、ローリング方法でテーブルコンストレインの追加を処理する方法はありません。

1 つのノードが既存の列の型を変更する DDL を実行すると、現在の型とターゲット型間のバイナリ強制性の存在によっては、オペレーションで基になるテーブルデータが書き換えられない場合があります。その場合、それは基になる列タイプのメタデータの更新だけです。テーブルの書き換えは通常制限されています。ただし、制御された DBA 環境では、すべてのノードの非レプリケート環境でこの列のタイプのローリングアップグレードを採用することにより、列のタイプを自動的にキャスト可能なものに変更できます。詳細は、ALTER TABLE を参照してください。

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

コマンドラインユーティリティ *pgd node upgrade* を使用して、PGD ノードを Postgres の新しいメジャーバージョン、または Postgres と PGD のメジャーバージョンにアップグレードできます。

> **注釈**
>
> 5.7.0 より前のバージョンでは、インプレースメジャーバージョンのアップグレードに使用されるコマンドは `bdr_pg_upgrade` でした。ただし、このコマンドには、 `pgd node upgrade` のように、Postgres メジャーバージョンと PGD バージョンの両方を同時にアップグレードするオプションがありません。

`pgd node upgrade` は、標準の pg_upgrade のラッパーで、PGD 固有のロジックをプロセスに追加して、スムーズなアップグレードを保証します。

#### 用語

この用語は、アップグレードプロセスと関連するコンポーネントを説明するときに使用されます。

*Postgres* **クラスター**—データベースファイルは、実行可能ファイルとデータの両方で、実行時にシステム上の Postgres データベースインスタンスを構成します。

**古い** *Postgres* **クラスター**—アップグレードする既存の Postgres クラスター、データを移行するクラスター。

**新しい** *Postgres* **クラスター**—データの移行先の新しい Postgres クラスター。この Postgres クラスターは、古いクラスターの 1 つ前のメジャーバージョンである必要があります。

#### 注意事項

両方の Postgres クラスターが pg_upgrade のすべての要件を満たしている必要があるという事実を含む、標準の Postgres メジャーバージョンのアップグレードの注意事項が適用されます。

さらに、他のツールがレプリケーションスロットとレプリケーション起点を使用している場合、`pgd node upgrade` を使用しないでください。アップグレード後には、PGD スロットとオリジンのみが復元されます。

`pgd node upgrade` のいくつかの前提条件を満たす必要があります。

- 古い Postgres クラスターを使用してアプリケーションを切断します。たとえば、PGD クラスター内の別のノードにリダイレクトできます。
- 両方の Postgres クラスターのピア認証を構成します。 `bdr_pg_upgrade` はピア認証が必要です。
- 新しいバージョンの PGD と同じまたは新しいバージョンを新しいクラスターにインストールする必要があります。

- PGD バージョンは 4.1.0 以降である必要があります。バージョン 3.7.22 以降もサポートされています。
- 新しいクラスターはシャットダウン状態である必要があります。
- 新しいクラスターに PGD パッケージをインストールする必要があります。
- 新しいクラスターは、古いクラスター構成と一致するように、必要に応じて既に初期化および構成されている必要があります。
- データベース、テーブル、およびその他のオブジェクトは、新しいクラスターに存在してはなりません。

**注釈**

PGD 5.7.0+にアップグレードする場合、両方のクラスターで同じ PGD バージョンを実行する必要はありません。新しいクラスターは 5.7.0 以降で実行されている必要があります。その場合、*pgd node upgrade* は PGD バージョンを 5.7.x にアップグレードし、Postgres メジャーバージョンをアップグレードします。

`pgd node upgrade` を実行する前に、古い Postgres クラスターを起動することをお勧めします。CLI は、古い Postgres クラスターがシャットダウンしている場合、それを起動します。

### 使用法

Postgres または Postgres と PGD の新しいメジャーバージョンにアップグレードするには、最初に新しいバージョンパッケージをインストールする必要があります。

### `pgd node upgrade` コマンドライン

`pgd node upgrade` は、すべてのパラメーターを pg_upgrade に渡します。したがって、pg_upgrade でサポートされているパラメーターを指定できます。

### 概要

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

### オプション

pg_upgrade のオプションに加えて、次のパラメーターを `pgd node upgrade` に渡すことができます。

### 必須パラメーター

これらのパラメーターは、コマンドラインで指定するか、 `--database` パラメーターを除くすべての場合、同等の環境変数で指定します。これらは `pgd node upgrade` によって使用されます。

- `-b, --old-bindir` —古い Postgres クラスターの bin ディレクトリ。
- `-B, --new-bindir` —新しい Postgres クラスターの bin ディレクトリ。
- `-d, --old-datadir` —古い Postgres クラスターデータディレクトリ。
- `-D, --new-datadir` —新しい Postgres クラスターデータディレクトリ。

- `--database` — PGD データベース名。

#### オプショナルパラメーター

これらのパラメーターはオプショナルであり、`pgd node upgrade` によって使用されます。

- `-p, --old-port` —古いクラスターポート番号。
- `-s, --socketdir` —アップグレード中にポストマスターソケットに使用するディレクトリ。
- `--check` —チェックのみを実行し、クラスターを変更しないように指定します。

#### その他のパラメーター

上記のいずれでもない他のパラメーターは pg_upgrade に渡されます。pg_upgrade は次のパラメーターを受け入れます。

- `-j, --jobs` —同時に使用するプロセスまたはスレッドの数。
- `-k, --link` —新しいクラスターにファイルをコピーする代わりに、ハードリンクを使用します。
- `-o, --old-options` —古い postgres コマンドに渡すオプション。複数の呼び出しが追加されます。
- `-O, --new-options` —新しい postgres コマンドに渡すオプション。複数の呼び出しが追加されます。
- `-N, --no-sync` —アップグレードされたクラスター内のすべてのファイルがディスクに書き込まれるのを待たないでください。
- `-P, --new-port` —新しいクラスターポート番号。
- `-r, --retain` —正常に完了した後でも SQL ファイルとログファイルを保持します。
- `-U, --username` —クラスターのインストールユーザー名。
- `--clone` —効率的なファイルのクローン作成を使用します。

#### 環境変数

コマンドラインパラメーターの代わりにこれらの環境変数を使用できます。

- `PGBINOLD` —古い Postgres クラスターの bin ディレクトリ。
- `PGBINNEW` —新しい Postgres クラスターの bin ディレクトリ。
- `PGDATAOLD` —古い Postgres クラスターデータディレクトリ。
- `PGDATANEW` —新しい Postgres クラスターデータディレクトリ。
- `PGPORTOLD` —古い Postgres クラスターポート番号。
- `PGSOCKETDIR` —アップグレード中にポストマスターソケットに使用するディレクトリ。

### 例

次のようなシナリオが考えられます。

- アップグレードするクラスターのノード名は kaolin です。
- 古い Postgres クラスターの bin ディレクトリは/usr/lib/postgresql/16/bin です。
- 新しい Postgres クラスターの bin ディレクトリは/usr/lib/postgresql/17/bin です。
- 古い Postgres クラスターデータディレクトリは/var/lib/postgresql/16/main です。
- 新しい Postgres クラスターデータディレクトリは/var/lib/postgresql/17/main です。
- データベース名は bdrdb です。

次のコマンドを使用して、クラスターをアップグレードできます。

```
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
```

### 実行された手順

これらの手順は、pgd node upgrade の実行時に実行されます。

> **注釈**
>
> *--check* が `pgd node upgrade` への引数として指定される場合、 CLI はデータベースを変更する手順をスキップします。

### PGD Postgres チェック

| 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* |

次のページに続く

表 2 – 前のページからの続き

| Steps | *--check* supplied |
|---|---|
| Starting old cluster with PGD disabled | *skip* |
| Connecting to old cluster | *skip* |
| Collecting replication origins | *skip* |
| Collecting replication slots | *skip* |
| Disconnecting from old cluster | *skip* |
| Stopping old cluster | *skip* |

#### pg_upgrade 手順

標準の pg_upgrade 手順が実行されます。

**注釈**

指定された場合、 *--check* は pg_upgrade に渡されます。

#### PGD のアップグレード後の手順

| 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* |
| Creating replication slot, repeated for each slot | *skip* |
| Stopping new cluster | *skip* |

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

#### Postgres メジャーバージョンのアップグレード

Postgres データベースのメジャーバージョンをアップグレードして、改善された機能、パフォーマンス強化、およびセキュリティ更新にアクセスすることは、一般的な管理タスクです。EDB Postgres 分散 PGD クラスターに対して同じことを行うことは、基本的に同じプロセスですが、ローリングアップグレードとして実行されます。

ローリングアップグレードプロセスにより、クラスターの可用性と操作の継続性を維持しながら、個々のクラスターノードを新しいメジャー Postgres バージョンに更新できます。このアプローチでは、各ノードがシークエンシャルにアップグレードされるときに、クラスターの残りの部分を操作可能なままにすることにより、ダウンタイムを最小限に抑え、データの整合性を保証します。

次の一般的な手順と実際の例の概要は、スムーズで制御されたアップグレードプロセスを提供するのに役立ちます。

### アップグレードの準備

アップグレードの準備をするには、アップグレードしようとしているサブグループとノードを特定し、最初のアップグレード順序をメモします。

これを行うには、SSH を使用してノードのいずれかに接続し、`pgd nodes list` コマンドを実行します。

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

`pgd nodes list` コマンドは、PGD クラスター内のすべてのノードと、各ノードが属しているサブグループを表示します。次に、各サブグループの書き込みリーダーであるノードを見つけます。

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

このコマンドは、クラスターで実行されている`<group_name>` によってトークン化された pgd グループに関する情報を表示します。どのノードが書き込みリーダーであるかなど。操作の継続性を維持するには、書き込みリーダーをアップグレードする前に、サブグループ内の別のノードにスイッチする必要があります。計画的なスイッチオーバーの数を最小限に抑えるには、ノードのサブグループをアップグレードするときに、最後にライタリーダーをアップグレードします。

アップグレードが完了するまで、アップグレードされるノードが書き込みリーダーにならないように、アップグレードを開始する前にノードをフェンスし、ノードのアップグレードが完了した後にノードのフェンスを解除する必要があります。

計画のためにどのノードが現在の書き込みリーダーであるかを確認した場合でも、サブグループの書き込みリーダーは、そのノードをアップグレードする前に、操作上の理由でいつでも別のノードに変更される可能性があります。したがって、ノードをアップグレードする前に、ノードが書き込みリーダーでないことを確認する必要があります。

これで、各サブグループで最後に指定された書き込みリーダーノードをアップグレードすることを目的として、一度に 1 サブグループごとにアップグレード順序を決定するための十分な情報が得られました。

### 各ノードでアップグレードを実行する

**重要**

データの損失を防ぐため、アップグレードプロセスを開始する前に、データベースと構成ファイルがバックアップされていることを確認してください。

アップグレードの準備 を使用して、SSH 経由で接続しているときに各ノードで次の手順を実行します。

- **現在の Postgres バージョンを確認します** * PGD からバージョンを表示します。

`sudo -u postgres pgd nodes list --versions` . * 予想されるメジャーバージョンが実行されていることを確認します。

- **ターゲットノードが書き込みリーダーでないことを確認します***ターゲットノードがアップグレードするグループの書き込みリーダーであるかどうかを確認します。

`sudo -u postgres pgd group <group_name> show --summary` *ターゲットノードが、アップグレードしているグループ/サブグループの現在の書き込みリーダーである場合、別のノードへの計画的なスイッチオーバーを実行します。

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

- **ノードをフェンスする** * アップグレードされるノードが、アップグレードが完了するまで書き込みリーダーにならないように、アップグレードを開始する前にノードをフェンスする必要があります。

- **ターゲットノードで Postgres を停止します***現在のノードで Postgres サービスを停止します。

```
sudo systemctl stop postgres
```

ターゲットノードは、クラスター内のノードとしてアクティブに参加していません。

- **PGD とユーティリティのインストール** * アップグレード先の Postgres バージョンと互換性のある PGD とそのユーティリティをインストールします。

```
sudo apt install edb-bdr6-pg<new_postgres_version_number>
```

- **新しい Postgres インスタンスを初期化します***新しいバージョンの PostgreSQL のデータベースファイルを格納するディレクトリを作成します。

`sudo mkdir -p /opt/postgres/datanew` **ユーザー** *postgres* **が ``*chown*`` を使用してディレクトリへの所有権権限があることを確認します。**作成したディレクトリで新しい PostgreSQL データベースクラスターを初期化します。この手順には、PostgreSQL の新しくインストールされたバージョンが提供する `initdb` コマンドの使用が含まれます。 `--data-checksums` フラグを含めて、クラスターがデータチェックサムを使用するようにします。

```
sudo -u postgres <path_to_postgres_bin>/initdb -D /opt/postgres/datanew
--data-checksums
```

<path_to_postgres_bin> を、新しくインストールされた PostgreSQL バージョンの bin ディレクトリへのパスに置き換えます。

このコマンドは、postgres ユーザーまたは適切な権限を持つ別のユーザーとして実行する必要がある場合があります。

- **構成を新しい Postgres バージョンに移行する現在の** *PostgreSQL* **データディレクトリで次の構成ファイルを見つけます。**`postgresql.conf` —データベースシステムに関連する設定を含むメイン構成ファイル。 ``*postgresql.auto.conf*`` — ``*ALTER SYSTEM*`` **コマンドで変更した設定など、***PostgreSQL* **によって設定された設定が含まれています。**`pg_hba.conf` —クライアント認証を管理し、どのユーザーがどのホストからどのデータベースに接続できるかを指定します。 ``*conf.d*`` **ディレクトリ全体存在する場合** - *mdash;* **構成設定を別のファイルに整理して、管理しやすくします。**これらのファイルと `conf.d` ディレクトリを、PostgreSQL のアップグレードバージョン用に作成した新しいデータディレクトリにコピーします。

- **Postgres サービスが非アクティブであることを確認します***続行する前に、古いデータディレクトリと新しいデータディレクトリの両方でアクティブな PostgreSQL プロセスがないことを確認することが重要です。この検証手順により、アップグレードプロセス中のデータの破損または競合が防止されます。

sudo systemctl status postgres コマンドを使用して、Postgres が停止したことを確認します。停止していない場合は、systemctl stop postgres を実行して、停止されたことを再度確認します。

- **バージョンアップグレードの PGDATA ディレクトリをスワップします** * /opt/postgres/data を/opt/postgres/dataold に、/opt/postgres/datanew を/opt/postgres/data に名前変更します。

この手順により、システムは次の重要なフェーズ、pgd ノードのアップグレードを実行して PostgreSQL バージョンの移行を完了するための準備が整います。

- **アップグレードの実現可能性を確認する** * pgd node upgrade ツールは、現在のセットアップの予備スキャンを実行し、アップグレードプロセスを妨げる可能性のある潜在的な問題を特定するように設計された--check オプションを提供します。

pgd node upgrade によって作成されたアップグレードログファイルを保存できるように、/home/upgrade/ などのユーザー postgres に所有権が付与されたアップグレードディレクトリからこのチェックを実行する必要があります。安全性チェックを開始するには、 --check オプションを pgd node upgrade コマンドに追加します。

この操作は、変更を加えずにアップグレードプロセスをシミュレートし、互換性問題、非推奨機能、またはアップグレードの成功に必要な構成調整に関する洞察を提供します。*このチェックで示された警告またはエラーに対処して、新しいバージョンへのスムーズな移行を保証します。

- **Postgres メジャーバージョンのアップグレードを実行します ``*--check*``オプションを指定せずに ``*pgd node <node_name> upgrade*`` コマンドを実行して、アップグレードプロセスを実行します。**注意が必要なエラーまたは警告がないかコマンド出力を監視することが重要です。*アップグレードプロセスにかかる時間は、データベースのサイズとセットアップの複雑さによって異なります。

- **Postgres サービス構成の更新** * postgres.service ファイルのバージョン番号を更新することにより、新しい PostgreSQL バージョンを反映するようにサービス構成を更新します。

sudo sed -i -e 's/<old_version_number>/<new_version_number>/g' /etc/systemd/system/postgres.service *システムのサービスマネージャーを更新して、これらの変更を適用します。

sudo systemctl daemon-reload

- **Postgres を再起動します** * PostgreSQL サービスの再起動に進みます。

systemctl start postgres

- **新しい Postgres バージョンを検証する** * PostgreSQL インスタンスがアップグレードされたことを確認します。

sudo -u postgres pgd nodes list --versions

- **ノードのフェンスを解除する** * アップグレードを検証した後、ノードのフェンスを解除できます。

- **アップグレード後のクリーンアップアップグレードの直後、高い運用負荷を導入する前に、``*ANALYZE*``オプションを使用して ``*vacuumdb*`` を実行します。このコマンドを実行すると、即時のパフォーマンスへの影響が最小限に抑えられ、より正確なテストのためにデータベースが準備されます。**古いバージョンのデータディレクトリ/opt/postgres/dataold を削除します。

### 実際の例 PGD 4 から PGD 6.1 へのアップグレード

この作業した例では、PGD 4 から PGD 6.1 へのインプレースメジャーバージョンのローリングアップグレードについて説明します。

#### 概要

HARP プロキシベースのルーティングを使用する PGD 4 クラスターは、クラスター全体が 6.1.0 以降にアップグレードされるまで、すべてのノードにこのルーティング方法を継続します。HARP プロキシベースのルーティングは、混合バージョンクラスター内で同じように機能します。4.x クラスターには書き込みリーダーがないため、HARP は独自のメカニズムを使用してリーダーを選択します。クラスター全体がバージョン 6.1.0 以降にアップグレードされると、接続マネージャーは既に有効になって準備が整います。アプリケーションは接続マネージャーポート/ノードを指すことができ、ルーティングを開始します。動作していることが確認されたら、プロキシを停止できます。

> **注釈**
>
> この例では、 PGD アーキテクチャで推奨されているように、Harp プロキシが PGD ノードと同じ場所にないことを前提としています。PGD ノードと同じ場所に Harp プロキシがある場合、アップグレード手順については EDB サポートに連絡してください。

#### harp-プロキシリーダーの確認

harp-プロキシリーダーではないノードでアップグレードを開始します。どのノードが harp プロキシリーダーであるかを確認します。

```
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
```

#### ノードをフェンスする

HARP からアップグレードするノードをフェンスオフし、フェンスされたことを確認して、アップグレードの途中でリーダーにならないようにします。

```
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␣
```

(次のページに続く)

(前のページからの続き)

```
↪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
```

### Postgres サービスを停止します

フェンスされたノードで、Postgres サービスを停止します。

### HARP マネージャーを停止します

フェンスされたノードで、 HARP マネージャーを停止します。

### パッケージの削除とインストール

フェンスされたノードで、 PGD 4.4 および cli パッケージを削除し、 PGD 6.1 パッケージをインストールします。

### Postgres サービスを開始する

フェンスされたノードで、Postgres サービスを開始します。これにより、接続マネージャーが有効になっている状態で PGD ローカルノードの PGD 6.1 へのインプレースアップグレードが実行されます。

### HARP マネージャーをスタートする

フェンスされたノードで、 HARP マネージャーを起動します。

### ノードのフェンスを解除する

アップグレードされたノードのフェンシングを HARP から解除します。

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

### すべてのノードに対して手順を繰り返します。

他のすべてのノードで同じ手順を繰り返します。

### クラスターバージョンの確認

`bdr.group_raft_details` を実行して、更新されたクラスターバージョンがバージョン 6001 であることを確認します。

### SCRAM ハッシュの確認

アップグレードノードから次のクエリーを実行して、SCRAM ハッシュが各ユーザーのすべてのノードで同じであることを確認します。これは、アプリケーションが接続マネージャーに切り替える前に必要です。

```
DO $$
DECLARE
    rec RECORD;
    command TEXT;
BEGIN
    FOR rec IN SELECT rolname,rolpassword FROM pg_authid WHERE rolcanlogin = true AND␣
↪rolpassword like SCRAM-SHA%
    LOOP
        command := ALTER ROLE  || quote_ident(rec.rolname) ||  WITH ENCRYPTED␣
↪PASSWORD  ||  || rec.rolpassword || ;
        EXECUTE command;
    END LOOP;
END;
$$;
SELECT wait_slot_confirm_lsn(NULL,NULL);
```

### ルーティングを有効にする

グローバルまたはローカルルーティング要件に従って、ノードグループルーティングを有効にします。ローカルルーティングの場合はサブグループで有効にし、グローバルルーティングの場合はトップグループで有効にします。

```
bdrdb=# SELECT bdr.alter_node_group_option(node_group_name := bdrgroup,config_key :=␣
↪enable_routing, config_value := true::TEXT);
```

出力

```
 alter_node_group_option
- ------------------------
 (1 row)
```

### 接続マネージャーに切り替える

これで、アプリケーションを Connection Manager に安全に切り替えられるはずです。

### harp マネージャーとプロキシサービスを停止します。

これで、実行中の harp マネージャーとプロキシサービスを安全に停止できるはずです。

> **注釈**

これで、PGD 4 から PGD 6.1 へのインプレースメジャーバージョンのローリングアップグレードの例が完了しました。

### 実際の例 PGD 5 から PGD 6.1 へのアップグレード

この例では、 pgd node upgrade コマンドを使用して、3 ノード PGD 5、EPAS 13 クラスターから PGD 6、EPAS 17 クラスターのインプレースメジャーバージョンローリングアップグレード PGD & Postgres について説明します。

#### 前提条件

3 ノード PGD 5、EPAS 13 クラスターが稼働していることを確認します。これは、TPA 展開されたクラスターです。

```
pgd-a1:/home/rocky $ pgd nodes list
```

出力

```
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
```

#### 新しいサーバーと PGD のパッケージをインストールします

EPAS 17 のパッケージと、対応する PGD 6 パッケージがクラスター内のすべてのノードにインストールされていることを確認します。バイナリの競合を防ぐには、PGD 6 パッケージをインストールする前に、PGD 5 パッケージつまり edb-pgd5-cli および edb-bdr5-epas13 を削除する必要があります。以下のコマンドは、RHEL 8 プラットフォームで使用されました。特定のプラットフォームに適切なコマンドを使用します。

```
dnf remove edb-pgd5-cli
dnf install edb-as17-server edb-pgd6-essential-epas17 -y
```

#### アップグレード前の手順

#### バージョンチェック

クラスターの現在のバージョンを確認しますオプショナル。

```
pgd-a1:/home/rocky $ pgd nodes list --versions
```

出力

```
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
```

#### 接続マネージャーに移動

PGD 5 は、ルーティングに PGD プロキシを使用します。 PGD 6 では、PGD プロキシは Connection Manager に置き換えられました。 PGD 5 から PGD 6 にアップグレードする場合、次の手順を使用して Connection Manager に移動します。 *PGD 5- PGD* **プロキシから接続マネージャーへの移動** を参照してください。

#### 書き込みリーダーノードの検証

アップグレードするノードが書き込みリーダーノードではないことを確認します。

```
pgd-a1:/home/rocky $ pgd group group-a show --summary
```

出力

```
Group Property    Value
- ---------------- -------
Group Name        group-a
Parent Group Name dc-1
Group Type        data
Write Leader      pgd-a2
Commit Scope
```

現在の書き込みリーダーはノード pgd-a2 であるため、ノード pgd-a1 をアップグレードするのが適切です。

アップグレードするノードの場合、書き込みリーダーを別のノードに切り替えます。

必要に応じて、 `pgd group set-leader` コマンドを使用して書き込みリーダーを切り替えます。

```
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
```

出力

```
 Group Property    | Value
- ------------------+---------
 Group Name        | group-a
 Parent Group Name | dc-1
 Group Type        | data
 Write Leader      | pgd-a1
 Commit Scope      |
```

### ノードをフェンスする

書き込みリーダーにならないように、pgd node <node-name> set-option route_fence true を使用してこのクラスター内のノードをフェンスします。

### 新しい Postgres インスタンスを初期化する

initdb ユーティリティを実行して、新しいサーバーを初期化します。--data-checksums オプションが設定されていることを確認します。

```
/usr/edb/as17/bin/initdb -D /var/lib/edb/as17/data -E UTF8 --data-checksums
```

デフォルトのデータディレクトリを使用したくない場合は、新しいデータディレクトリを作成します。この例では、簡単にするためにデフォルトを使用します。

### 構成を新しい Postgres バージョンに移行する

次のファイルとディレクトリ存在する場合は、PostgreSQL のアップグレードバージョン用に作成した新しいデータディレクトリにコピーします。

- postgresql.conf
- postgresql.auto.conf
- pg_hba.conf
- conf.d ディレクトリ

```
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/
```

TPA 展開されたクラスターがある場合は、 conf.d ディレクトリもコピーします。

```
cp -r /opt/postgres/data/conf.d /var/lib/edb/as17/data/
```

たとえば、 operator_precedence_warning GUC が発生する場合がありますが、新しい構成では無視できます。

```
sudo systemctl stop postgres
sudo systemctl status postgres
```

> **注釈**
>
> PostgreSQL インスタンスがサービスとして構成されているため、この例では systemctl コマンドが使用されています。セットアップが異なる場合は、pg_ctl ユーティリティの使用が必要になる場合があります。

実際のアップグレードを実行する前に、ドライランを実行して、互換性とアップグレードの実現可能性を確認します。pgd node upgrade ツールには、アップグレードプロセスの一部のドライランを実行する--check

オプションがあります。このオプションを使用して、アップグレードをスムーズに行うことができます。--check オプションを指定して upgrade コマンドを実行します。

```
/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
```

チェックが成功すると、次のような出力が返されます。

```
Performing BDR Postgres Checks
- ------------------------------
Getting old PG instance shared directory                    ok
Getting new PG instance shared directory                    ok
Collecting pre-upgrade new PG instance control data         ok
Checking new cluster state is shutdown                      ok
Checking BDR extension versions                             ok
Checking Postgres versions                                  ok

Finished BDR pre-upgrade steps, calling pg_upgrade
- ------------------------------------------------

Performing Consistency Checks
- -----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
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*
```

アップグレードを実行します。

ドライランチェックに合格した場合、 --check オプションを指定せずにコマンドを実行して、アップグレードを実行できます。

```
/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
```

アップグレードが成功すると、次のような出力が返されます。

```
Performing BDR Postgres Checks
- ------------------------------
Getting old PG instance shared directory                    ok
Getting new PG instance shared directory                    ok
Collecting pre-upgrade new PG instance control data         ok
Checking new cluster state is shutdown                      ok
Checking BDR extension versions                             ok
Checking Postgres versions                                  ok

Collecting Pre-Upgrade BDR Information
- -------------------------------------
Collecting pre-upgrade old PG instance control data         ok
Connecting to the old PG instance                           ok
Checking for BDR extension                                  ok
Checking BDR node name                                      ok
Terminating connections to database                         ok
Waiting for all slots to be flushed                         ok
Disconnecting from old cluster PG instance                  ok
Stopping old PG instance                                    ok
Starting old PG instance with BDR disabled                  ok
Connecting to the old PG instance                           ok
Collecting replication origins                              ok
Collecting replication slots                                ok
Disconnecting from old cluster PG instance                  ok
Stopping old PG instance                                    ok

Finished BDR pre-upgrade steps, calling pg_upgrade
- ------------------------------------------------

Performing Consistency Checks
```

(次のページに続く)

(前のページからの続き)

```
- ----------------------------
Checking cluster versions                                   ok
Checking database user is the install user                  ok
Checking database connection settings                       ok
Checking for prepared transactions                          ok
Checking for contrib/isn with bigint-passing mismatch       ok
Checking data type usage                                    ok
Checking for user-defined encoding conversions              ok
Checking for user-defined postfix operators                 ok
Checking for incompatible polymorphic functions             ok
Checking for not-null constraint inconsistencies            ok
Creating dump of global objects                             ok
Creating dump of database schemas
                                                            ok
Checking for presence of required libraries                 ok
Checking database user is the install user                  ok
Checking for prepared transactions                          ok
Checking for new cluster tablespace directories             ok

If `pg_upgrade` fails after this point, you must re-initdb
the new cluster before continuing.

Performing Upgrade
- -----------------
Setting locale and encoding for new cluster                 ok
Analyzing all rows in the new cluster                       ok
Freezing all rows in the new cluster                        ok
Deleting files from new pg_xact                             ok
Copying old pg_xact to new server                           ok
Setting oldest XID for new cluster                          ok
Setting next transaction ID and epoch for new cluster       ok
Deleting files from new pg_multixact/offsets                ok
Copying old pg_multixact/offsets to new server              ok
Deleting files from new pg_multixact/members                ok
Copying old pg_multixact/members to new server              ok
Setting next multixact ID and offset for new cluster        ok
Resetting WAL archives                                      ok
Setting frozenxid and minmxid counters in new cluster       ok
Restoring global objects in the new cluster                 ok
Restoring database schemas in the new cluster
                                                            ok
Copying user relation files
                                                            ok
Setting next OID for new cluster                            ok
Sync data directory to disk                                 ok
```

(次のページに続く)

(前のページからの続き)

```
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
```

**注釈**

ハードリンクに `--link` オプションを使用できます。このオプションは、両方のデータディレクトリが同じファイルシステム上にある場合に機能します。 詳細については、PostgreSQL ドキュメントの pg_upgrade を参照してください。

### アップグレード後の手順

#### Postgres サービスファイルの更新

/etc/systemd/system/postgres.service にある PostgreSQL サービスファイルで、新しいサーバーのサーバーバージョン、データディレクトリ、およびバイナリディレクトリを更新します。

更新されたサービスファイルの例

```
[Unit]
Description=Postgres 17 (TPA)
After=syslog.target
After=network.target

[Service]
Type=simple
User=enterprisedb
Group=enterprisedb
OOMScoreAdjust=-1000
Environment=PG_OOM_ADJUST_VALUE=0
Environment=PGDATA=/var/lib/edb/as17/data
StandardOutput=syslog
ExecStart=/usr/edb/as17/bin/edb-postgres -D ${PGDATA} -c config_file=/var/lib/edb/
↪as17/data/postgresql.conf
ExecStartPost=+/bin/bash -c echo 0xff > /proc/$MAINPID/coredump_filter
ExecReload=/bin/kill -HUP $MAINPID
KillMode=mixed
KillSignal=SIGINT
Restart=no
LimitCORE=infinity

[Install]
WantedBy=multi-user.target
```

#### postgres サービスを開始する

daemon-reload を実行し、Postgres サービスを開始します。

```
systemctl daemon-reload
systemctl start postgres
```

> **注釈**
>
> サーバーがサービスとして実行されていない場合は、サービスファイルの更新をスキップし、 pg_ctl ユーティリティを使用してサーバーを起動できます。

### アップグレードされたクラスターバージョンの確認

次のコマンドを使用して、アップグレードされたクラスターバージョンを確認します。

```
pgd-a1:/home/rocky $ /usr/edb/as17/bin/pgd nodes list --versions
```

出力

```
 Node Name  | BDR Version         | Postgres Version
- -----------+---------------------+------------------
 pgd-a1     | PGD 6.1.0 Essential | 17.6.0
 pgd-a2     | 5.9.0               | 13.22.28
 witness-a1 | 5.9.0               | 13.22.28
```

ノード **pgd-a1** の BDR バージョンは 6.1.0 にアップグレードされ、Postgres バージョンは 17.6.0 にアップグレードされました。

### ノードのフェンスを解除する

`pgd node <node-name> set-option route_fence false` を使用してこのクラスター内のノードのフェンスを解除して、書き込みリーダーにならないようにします。

### 接続マネージャーが動作していることを確認する

アップグレードしたノードで接続マネージャーポートデフォルトでは 6444 を介してクエリーを実行します。

```
pgd-a1:/home/rocky $ psql "host=pgd-a1 port=6444 dbname=bdrdb user=enterprisedb " -c
↪"select node_name from bdr.local_node_summary;"
```

出力

```
 node_name
- ----------
 pgd-a2
(1 row)
```

### クリーンアップとバキューム分析

ベストプラクティスとして、この時点でデータベースにバキュームを実行します。アップグレードが実行されると、アップグレード後レポートに次のものが含まれていることに気づいたかもしれません。

```
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
```

これで、バキュームを実行できます。ターゲットノードで、次を実行します。

```
/usr/edb/as17/bin/vacuumdb -U enterprisedb --all --analyze-in-stages
```

このノードを元に戻す必要がない場合は、古いクラスターのデータファイルをクリーンアップすることもできます。

```
./delete_old_cluster.sh
```

### 残りのノードをアップグレードする

クラスター内のすべてのノードに対してこれらの手順を実行する必要があります。唯一の違いは、upgrade コマンドのノード名です。簡単なリファレンスのために、ノード **pgd-a2** および **witness-a1** のコマンドが提供されています。

ノード **pgd-a2**

```
/usr/edb/as17/bin/pgd node pgd-a2 upgrade \
- -database bdrdb -B /usr/edb/as17/bin \
- -socketdir /tmp \
- -old-bindir /usr/edb/as13/bin \
- -old-datadir /opt/postgres/data \
- -new-datadir /var/lib/edb/as17/data \
- -username enterprisedb \
- -old-port 5444 \
- -new-port 5444
```

ノード **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
```

### クラスターの最終的な状態を確認する

次のコマンドを使用して、ノードバージョンを確認します。

```
pgd-a2:/home/rocky $ /usr/edb/as17/bin/pgd nodes list --versions
```

出力

```
 Node Name  | BDR Version          | Postgres Version
- ----------+----------------------+------------------
 pgd-a1     | PGD 6.1.0 Essential  | 17.6.0
 pgd-a2     | PGD 6.1.0 Essential  | 17.6.0
 witness-a1 | PGD 6.1.0 Essential  | 17.6.0
```

クラスターのすべてのノードは、PGD 6.1.0 および EPAS 17.6.0 にアップグレードされました。

### 接続マネージャーの検証

すべてのデータノードについて、次のコマンドを使用して接続マネージャーを確認します。

```
pgd-a2:/home/rocky $ psql "host=pgd-a2 port=6444 dbname=bdrdb user=enterprisedb " -c
↪"select node_name from bdr.local_node_summary;"
```

出力

```
 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;"
```

出力

```
 node_name
- ----------
 pgd-a2
(1 row)
```

**注釈**

これで、3 ノード PGD 5、EPAS 13 クラスターから PGD 6、EPAS 17 クラスターのインプレースメジャーバージョンローリングアップグレード PGD & Postgres の作業例が完了しました。