



# **EDB Postgres Distributed**

リリース **4.2.0**

**EnterpriseDB Corporation**

2022 年 09 月 02 日



# 目次

|              |                                                                    |           |
|--------------|--------------------------------------------------------------------|-----------|
| <b>第 1 章</b> | <b>EDB Postgres Distributed Release notes</b>                      | <b>3</b>  |
| 1.1          | Release notes for EDB Postgres Distributed version 4.2.0 . . . . . | 4         |
| 1.2          | Release notes for EDB Postgres Distributed version 4.1.1 . . . . . | 5         |
| 1.3          | Release notes for EDB Postgres Distributed version 4.1.0 . . . . . | 7         |
| 1.4          | Release notes for EDB Postgres Distributed version 4.0.3 . . . . . | 17        |
| 1.5          | Release notes for EDB Postgres Distributed version 4.0.2 . . . . . | 17        |
| 1.6          | Release notes for EDB Postgres Distributed version 4.0.1 . . . . . | 20        |
| 1.7          | Release notes for EDB Postgres Distributed version 4.0.0 . . . . . | 24        |
| <b>第 2 章</b> | <b>‘Known issues’</b>                                              | <b>33</b> |
| 2.1          | データの一貫性 . . . . .                                                  | 33        |
| 2.2          | 問題のリスト . . . . .                                                   | 33        |
| 2.3          | 制限事項のリスト . . . . .                                                 | 34        |
| <b>第 3 章</b> | <b>Terminology</b>                                                 | <b>37</b> |
| 3.1          | 非同期レプリケーション . . . . .                                              | 37        |
| 3.2          | 可用性 . . . . .                                                      | 37        |
| 3.3          | CAMO または commit-at-most-once . . . . .                             | 37        |
| 3.4          | クラスタリング . . . . .                                                  | 37        |
| 3.5          | データシャーディング . . . . .                                               | 38        |
| 3.6          | BDR の Eager Replication . . . . .                                  | 38        |
| 3.7          | 結果的な一貫性 . . . . .                                                  | 38        |
| 3.8          | フェールオーバー . . . . .                                                 | 38        |
| 3.9          | 水平スケーリング または スケールアウト . . . . .                                     | 38        |
| 3.10         | 論理レプリケーション . . . . .                                               | 38        |
| 3.11         | ナイン . . . . .                                                      | 39        |
| 3.12         | ノード . . . . .                                                      | 39        |
| 3.13         | 物理レプリケーション . . . . .                                               | 39        |
| 3.14         | 読み取りのスケラビリティ . . . . .                                             | 39        |
| 3.15         | 目標復旧時点 (RPO) . . . . .                                             | 39        |
| 3.16         | 目標復旧時間 (RTO) . . . . .                                             | 39        |
| 3.17         | 単一障害点 (SPOF) . . . . .                                             | 40        |
| 3.18         | スイッチオーバー . . . . .                                                 | 40        |
| 3.19         | 同期レプリケーション . . . . .                                               | 40        |
| 3.20         | 2 フェーズコミット (2PC) . . . . .                                         | 40        |
| 3.21         | 垂直スケーリング または スケールアップ . . . . .                                     | 40        |
| 3.22         | 書き込みスケラビリティ . . . . .                                              | 40        |
| <b>第 4 章</b> | <b>Key components</b>                                              | <b>41</b> |
| 4.1          | Postgres サーバー . . . . .                                            | 41        |

|               |                                               |            |
|---------------|-----------------------------------------------|------------|
| 4.1.1         | High Availability Routing for Postgres (HARP) | 42         |
| 4.1.2         | BDR (Bi-Directional Replication)              | 79         |
| <b>第 5 章</b>  | <b>Choosing your architecture</b>             | <b>395</b> |
| 5.1           | 標準の EDB Always On アーキテクチャ                     | 395        |
| 5.2           | アーキテクチャの詳細                                    | 396        |
| 5.3           | アーキテクチャの選択                                    | 396        |
| 5.3.1         | Always On Bronze                              | 396        |
| 5.3.2         | Always On Silver                              | 398        |
| 5.3.3         | Always On Gold                                | 399        |
| 5.3.4         | Always On Platinum                            | 399        |
| <b>第 6 章</b>  | <b>Choosing a Postgres distribution</b>       | <b>401</b> |
| <b>第 7 章</b>  | <b>Choosing durability</b>                    | <b>403</b> |
| <b>第 8 章</b>  | <b>Other considerations</b>                   | <b>405</b> |
| 8.1           | 展開とサイジングの考慮事項                                 | 405        |
| 8.2           | 時計とタイムゾーン                                     | 405        |
| <b>第 9 章</b>  | <b>Deployment options</b>                     | <b>407</b> |
| 9.1           | TPAexec                                       | 407        |
| 9.1.1         | インクリメンタルな変更                                   | 408        |
| 9.1.2         | Ansible で拡張可能                                 | 408        |
| 9.1.3         | クラスター管理                                       | 408        |
| 9.1.4         | ただの Postgres です                               | 408        |
| 9.1.5         | はじめに                                          | 408        |
| <b>第 10 章</b> | <b>Upgrading</b>                              | <b>417</b> |
| 10.1          | アップグレード計画                                     | 417        |
| 10.1.1        | ローリングアップグレードの考慮事項                             | 418        |
| 10.1.2        | ローリングサーバーソフトウェアのアップグレード                       | 418        |
| 10.1.3        | ノード結合を使用したローリングアップグレード                        | 419        |
| 10.1.4        | CAMO 対応クラスターのアップグレード                          | 419        |
| 10.2          | アップグレードの準備                                    | 419        |
| 10.3          | サーバーソフトウェアのアップグレード                            | 419        |
| 10.3.1        | BDR 拡張機能のアップグレード                              | 420        |
| 10.3.2        | Postgres を停止する                                | 420        |
| 10.3.3        | アップグレードパッケージ                                  | 420        |
| 10.3.4        | Postgres を起動                                  | 420        |
| 10.3.5        | Postgres のアップグレード                             | 420        |
| 10.3.6        | マイナーバージョンの Postgres のアップグレード                  | 420        |
| 10.3.7        | メジャーバージョン Postgres のアップグレード                   | 421        |
| 10.4          | アップグレードのチェックと検証                               | 421        |
| 10.5          | アプリケーションスキーマのアップグレード                          | 421        |
| 10.5.1        | アプリケーションスキーマのローリングアップグレード                     | 421        |
| <b>第 11 章</b> | <b>Compatibility matrix</b>                   | <b>429</b> |

|                                                               |            |
|---------------------------------------------------------------|------------|
| <b>第 12 章 Backup and recovery</b>                             | <b>431</b> |
| 12.1 バックアップ                                                   | 431        |
| 12.1.1 pg_dump                                                | 431        |
| 12.1.2 物理バックアップ                                               | 432        |
| 12.1.3 最終的な一貫性                                                | 432        |
| 12.1.4 ポイントインタイムリカバリー (PITR)                                  | 433        |
| 12.2 復元する                                                     | 434        |
| 12.2.1 EDB Postgres 分散クラスターの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        | 434        |
| 12.2.2 BDR メタデータのクリーンアップ                                      | 435        |
| 12.2.3 レプリケーションオリジンのクリーンアップ                                   | 435        |
| 12.2.4 レプリケーションスロットのクリーンアップ                                   | 435        |
| <b>第 13 章 Monitoring</b>                                      | <b>437</b> |
| 13.1 モニタリングの概要                                                | 437        |
| 13.2 モニタリングノードの参加と削除                                          | 438        |
| 13.3 レプリケーションピアの監視                                            | 439        |
| 13.3.1 送信レプリケーションのモニタリング                                      | 439        |
| 13.3.2 着信レプリケーションの監視                                          | 442        |
| 13.3.3 LCR を使用した WAL 送信者の監視                                   | 442        |
| 13.4 BDR レプリケーションワーカーの監視                                      | 443        |
| 13.5 BDR ライターの監視                                              | 443        |
| 13.6 グローバルロックの監視                                              | 444        |
| 13.7 競合の監視                                                    | 444        |
| 13.8 外部モニタリング                                                 | 445        |
| 13.8.1 bdr.set_node_location                                  | 445        |
| 13.8.2 概要                                                     | 445        |
| 13.8.3 パラメーター                                                 | 445        |
| 13.8.4 bdr.set_network_path_info                              | 446        |
| 13.8.5 概要                                                     | 446        |
| 13.8.6 パラメーター                                                 | 446        |
| 13.8.7 bdr.network_monitoring ビュー                             | 446        |
| 13.9 統計を適用                                                    | 447        |
| 13.10 標準の PostgreSQL 統計ビュー                                    | 448        |
| 13.11 BDR バージョンの監視                                            | 448        |
| 13.12 ラフトコンセンサスのモニタリング                                        | 449        |
| 13.13 レプリケーションスロットのモニタリング                                     | 451        |
| 13.14 トランザクションの COMMIT の監視                                    | 452        |
| <b>第 14 章 EDB Postgres Distributed Command Line Interface</b> | <b>453</b> |
| 14.1 要件                                                       | 453        |
| 14.2 PGD CLI の使用                                              | 453        |
| 14.3 構成ファイルの指定                                                | 453        |
| 14.4 データベース接続文字列を渡す                                           | 454        |
| 14.4.1 Installing PGD CLI                                     | 455        |
| 14.4.2 Command reference                                      | 456        |



EDB Postgres Distributed は、高度な競合管理、データ損失保護、およびネイティブ論理レプリケーションよりも最大 5 倍速いスループット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、最大ファイブナインの高可用性を備えた分散 PostgreSQL クラスターを有効にします。



図 1: node diagram

デフォルトでは、EDB Postgres Distributed は非同期レプリケーションを使用し、ローカルコミット後にのみピアノードに変更を適用します。

*show-camo*、*CAMO* とグループコミット、または eager all node replication

機能を構成することにより、異なるノード、ノードのグループ、またはすべてのノード間で追加のレベルの同期を構成できます。





# 第1章 EDB Postgres Distributed Release notes

EDB Postgres Distributed ドキュメントには、マイナーリリースとパッチを含む EDB Postgres Distributed 4 の最新バージョンが記載されています。リリースノートには、各リリースの新機能に関する情報が記載されています。マイナーリリースまたはパッチリリースで導入された新機能の場合、コンテンツにはその機能が導入されたリリースも示されます。

| Release Date | EDB Postgres Distributed                                        | BDR   | HARP  | CLI   | TPAexec |
|--------------|-----------------------------------------------------------------|-------|-------|-------|---------|
| 2022 Aug 22  | <i>Release notes for EDB Postgres Distributed version 4.2.0</i> | 4.2.0 | 2.2.0 | 1.1.0 | 23.5    |
| 2022 June 21 | <i>BDR 4.1.1</i>                                                | 4.1.1 | 2.1.1 | 1.0.0 | 23.2    |
| 2022 May 17  | <i>BDR 4.1.0</i>                                                | 4.1.0 | 2.1.0 | 1.0.0 | 23.1    |
| 2022 Mar 31  | <i>Release notes for EDB Postgres Distributed version 4.0.3</i> | •     | 2.0.3 | •     | 22.10   |
| 2022 Feb 24  | <i>BDR 4.0.2</i>                                                | 4.0.2 | 2.0.2 | •     | 22.9    |
| 2022 Jan 31  | <i>BDR 4.0.1</i>                                                | 4.0.1 | 2.0.1 | •     | 22.6    |
| 2021 Dec 01  | <i>BDR 4.0.0</i>                                                | 4.0.0 | 2.0.0 | •     | 22.9    |

## 1.1 Release notes for EDB Postgres Distributed version 4.2.0

EDB Postgres Distributed バージョン 4.2.0 は EDB Postgres Distributed 4 のマイナーリリースであり、以前のバージョンで特定された問題の修正と新機能が含まれています。

| Component | Version | Type        | Description                                                                                                                                                                                                         |
|-----------|---------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.2.0   | Enhancement | 予約された galloc シーケンススロットの消費を許可します (BDR-2367、RT83437、RT68255) <p>バックグラウンドアロケーターによる将来の使用のために予約された galloc シーケンススロットは、コンセンサス障害の存在下で消費できます。 </p>                                                                           |
| BDR       | 4.2.0   | Bug fix     | アイドル接続での誤った pglogical レシーバータイムアウトを修正しました (RT82315) <p>レプリケートして適用するものが何もない場合、pglogical レシーバーは状態をタイムアウトと誤って解釈し、「ERROR: 08006: termination pglogical receiver due to timeout」で終了する場合があります。この誤解を修正します。 </p>             |
| BDR       | 4.2.0   | Bug fix     | スーパーユーザー以外の信頼できる拡張機能の「不十分な特権」エラーを修正しました (BDR-2350) <p>PostgreSQL 13 以降で非スーパーユーザーによって実行される CREATE EXTENSION コマンドは、実行された BDR ノードでは成功する場合がありますが、複製に失敗しますレプリケーションを壊す他のノードに。このようなコマンドを非オリジンノードに適用できるようにして、これを修正します。 </p> |

次のページに続く

表 2 – 前のページからの続き

| Component | Version | Type        | Description                                                                                                                                                                                                            |
|-----------|---------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.2.0   | Bug fix     | パラレル適用が有効になっている場合、再起動時に <code>global_locks</code> を再構築するときの競合状態を回避する (RT83435、BDR-2383) <p>これが発生する BDR ノードは回復不能になります。単一のライターにすべてのグローバルロックを回復させることにより、これを修正します。 </p>                                                    |
| BDR       | 4.2.0   | Bug fix     | 競合が <code>bdr.conflict_history</code> に記録されるときのスプリアスセグメンテーションフォールトを修正しました (BDR-2403、ライタープロセス、RT83928) <p>使用されているポインター。この使用法を修正します。 </p>                                                                                |
| BDR       | 4.2.0   | Bug fix     | <code>bdr_init_physical</code> が失敗したときにレプリケーションスロットをクリーンアップする (BDR-2364、RT74789) <p> <code>bdr_init_physical</code> がノードに参加できないまま中止すると、非アクティブなレプリケーションスロットが残ります。不規則な終了の前に非アクティブになっているときに、このようなレプリケーションスロットを削除します。 </p> |
| HARP      | 2.2.0   | Enhancement | 処理フックを含む、接続文字列に <code>sslpassword</code> のサポートを追加する (HNG-626)                                                                                                                                                          |

## 1.2 Release notes for EDB Postgres Distributed version 4.1.1

EDB Postgres Distributed バージョン 4.1.1 には、BDR 4.1.1 および HARP 2.1.1 のバグ修正が含まれています。TPAexec の変更とバグ修正については、

[TPAexec Release Notes for version 23.1](#) を参照してください。

| Component | Version | Type        | Description                                                                                                                                                                                                                                                                                                                 |
|-----------|---------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.1   | Feature     | ジェネリック関数を追加します<br>bdr.is_node_connected は、特定のピアの walsender がアクティブな場合に true を返します。                                                                                                                                                                                                                                           |
| BDR       | 4.1.1   | Feature     | ジェネリック関数を追加します<br>bdr.is_node_ready は、ラグが特定のスパン未満にある場合にブール値を返します。                                                                                                                                                                                                                                                           |
| BDR       | 4.1.1   | Bug fix     | ハードリンクを使用するための bdr.pg_upgrade に `--link` 引数のサポートを追加します。                                                                                                                                                                                                                                                                     |
| BDR       | 4.1.1   | Bug fix     | デフォルトのコミットスコープとして `bdr.node_group` によってまだ参照されている場合、`bdr.remove_commit_scope` を削除しないようにします。                                                                                                                                                                                                                                  |
| BDR       | 4.1.1   | Bug fix     | ノードの CAMO ペアのローカルモードへの Raft ベースの正しい切り替え (RT78928)                                                                                                                                                                                                                                                                           |
| BDR       | 4.1.1   | Bug fix     | コーナーケースの bdr.drop_node の潜在的なセグメンテーションを防止 (RT81900)                                                                                                                                                                                                                                                                          |
| BDR       | 4.1.1   | Bug fix     | トランザクションストリーミングと組み合わせて CAMO または Eager All Node トランザクションを使用しないようにします。トランザクションストリーミングは、CAMO および Eager All Node トランザクションと組み合わせて問題があることがわかりました。これが解決されるまで、BDR はその併用を禁止します。これには、CAMO 展開で構成を調整してトランザクションストリーミングを無効にする必要がある場合があります。<br>:ref:`Transaction Streaming Configuration` <transaction-streaming#configuration> を参照してください。 |
| HARP      | 2.1.1   | Enhancement | DCS 接続の損失に関する警告をログに記録する                                                                                                                                                                                                                                                                                                     |

次のページに続く

表 3 – 前のページからの続き

| Component | Version | Type        | Description                                          |
|-----------|---------|-------------|------------------------------------------------------|
| HARP      | 2.1.1   | Enhancement | メタデータの更新に時間がかかりすぎる場合に警告をログに記録します - 通常は高遅延ネットワークが原因です |
| HARP      | 2.1.1   | Bug fix     | 障害時に harp_proxy.service をリスタート                       |
| HARP      | 2.1.1   | Bug fix     | harpctl の接続管理の同時実行の問題を修正                             |
| HARP      | 2.1.1   | Bug fix     | アンマネージドクラスターで以前のリーダーへのプロキシ接続を試行しないでください              |
| HARP      | 2.1.1   | Bug fix     | 場所が空の場合に harpctl でパニックにならない                          |

## 1.3 Release notes for EDB Postgres Distributed version 4.1.0

EDB Postgres Distributed バージョン 4.1.0 には以下が含まれます。

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                           |
|-----------|---------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CLI       | 4.1.0   | Feature | 新しい <code>:ref:`command-line interface (CLI) &lt;/pgd/latest/cli&gt;`</code> を使用して、EDB Postgres 分散クラスターのレプリケーション、コンセンサス、ノードなどの情報を収集する機能。                                                                                                                                              |
| BDR       | 4.1.0   | Feature | 新しいコマンドラインユーティリティ <code>bdr_pg_upgrade</code> を使用して、データノードで Postgres の <code>:ref:`in-place major upgrades &lt;/pgd/latest/upgrades/bdr_pg_upgrade/&gt;`</code> をサポートします。このユーティリティは、標準の <code>pg_upgrade</code> コマンドを使用し、EDB Postgres 分散クラスターのメジャーバージョンアップグレードに必要な時間とネットワーク帯域幅を削減します。 |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                               |
|-----------|---------|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Feature | <a href="#">:ref: replication lag threshold</a><br></pgd/latest/overview/bdr/lag-control/>`を構成する機能を有効にします。しきい値に達すると、トランザクションのコミットが調整されます。このしきい値により、同期レプリケーションに伴うすべてのトランザクションに待機時間の影響を与えることなく、RPO を制限できます。 |
| BDR       | 4.1.0   | Feature | <a href="#">:ref: BDR グローバルシーケンス</a><br><BDR グローバルシーケンス>はデータ型に基づいて自動的に構成され、すべてのノードでカスタムのシーケンス処理構成を設定する必要はありません。新しい SnowflakeID アルゴリズムは、制限があった Timeshard を置き換えます。                                          |
| BDR       | 4.1.0   | Feature | ノードごとではなくグループごとに同期レプリケーションの持続性と可視性のオプションを構成するための新しい SQL レベルのインターフェイスを追加します。このアプローチにより、構成ファイルを使用する代わりに、すべてのノードを 1 か所から一貫して構成できます。                                                                          |
| BDR       | 4.1.0   | Feature | EDB Postgres 分散グループでトランザクションをコミットする前にクォーラムを要求できる新しい同期レプリケーションオプション<br><a href="#">:ref: CAMO とグループコミット</a><br><CAMO とグループコミット>を追加します。                                                                     |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                              |
|-----------|---------|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Feature | ローカルモードへの CAMO の切り替えに Raft リクエストが必要になるようにします。`require_raft` フラグを CAMO 保護からローカルモードへの切り替えの動作を制御する CAMO ペアリング構成に追加し、ローカルモードへの切り替えを許可するためにノードの大部分を接続することを要求するオプションを導入します。(RT78928)                                                                                           |
| BDR       | 4.1.0   | Feature | 既にデタッチされたパーティションの `ALTER TABLE ... DETACH PARTITION CONCURRENTLY` でレプリケーションを続行できるようにします。同じインデックスが既に存在する場合に BDR 4 が `CREATE INDEX CONCURRENTLY` を処理する方法と同様に、`ALTER TABLE ... DETACH PARTITION CONCURRENTLY` が既にデタッチされたパーティションのレシーバーである場合にレプリケーションを続行できるようになりました。(RT78362) |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                           |
|-----------|---------|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Feature | DDL フィルターに追加のフィルタリングオプションを追加します。DDL フィルターを使用すると、さまざまな DDL ステートメントをさまざまなレプリケーションセットに複製できます。レプリケーションセットのテーブルメンバーシップにより、さまざまなテーブルの DML をさまざまなレプリケーションセットを介してレプリケートできるのと同様に。このリリースでは、DDL フィルターを使いやすくする新しいコントロールが追加されています。<br />- query_match - 定義されたクエリがこの正規表現と一致する必要がある場合<br />- Exclusive - true の場合、他の一致したフィルターは考慮されません（つまり排他フィルターが適用されます）、複数の排他フィルターが一致するとエラーがスローされます |
| BDR       | 4.1.0   | Feature | `bdr.lock_table_locking` 構成変数を追加します。有効にすると、これは `LOCK TABLE` コマンドの動作を変更してグローバル DML ロックを取得します                                                                                                                                                                                                                                                                           |
| BDR       | 4.1.0   | Feature | LCR セグメントファイルのバッファリングされた書き込みを実装します。これにより、I/O が削減され、Decoding Worker の CPU 使用率が向上します。                                                                                                                                                                                                                                                                                   |
| BDR       | 4.1.0   | Feature | 競合検出のための部分的な一意のインデックス検索のサポートを追加します。ただし、式のインデックスはまだ競合検出でサポートされていません。 (RT78368)                                                                                                                                                                                                                                                                                         |

次のページに続く



表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|-----------|---------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Feature | <code>`bdr.stat_subscription`</code> に追加の統計を追加します。<br>- <code>nstream_insert</code> => ストリーミングされたトランザクションの INSERT の数<br>- <code>nstream_update</code> => ストリーミングされたトランザクションの UPDATE の数<br>- <code>nstream_delete</code> => ストリーミングされたトランザクションの数<br>- <code>transaction</code><br>- <code>nstream_truncate</code> => ストリーミングされたトランザクションの TRUNCATE の数<br>- <code>npre_commit_confirmations</code> => CAMO を使用する場合のプリコミット確認の数<br>- <code>npre_commit</code> => プリコミットの数<br>- <code>ncommit_prepared</code> => 2PC での準備されたコミットの数<br>- <code>nabort_prepared</code> => 2PC での準備されたトランザクションのアボート数 |
| BDR       | 4.1.0   | Feature | <code>execute_locally</code> オプションを <code>bdr.replicate_ddl_command</code> に追加します。これにより、ローカルで実行せずに他のグループにレプリケーションするための ddl コマンドをオプションでキューイングできます。(RT73533)                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| BDR       | 4.1.0   | Feature | <code>bdr.alter_subscription_disable()</code> に <code>fast</code> 引数を追加します。引数は <code>immediate</code> の動作にのみ影響します。 <code>true`</code> (デフォルト) に設定すると、現在の作業を終了させずにワーカーを停止します。(RT79798)                                                                                                                                                                                                                                                                                                                                                                                                       |
| BDR       | 4.1.0   | Feature | <code>void</code> を返すように <code>`bdr.{add,remove}_camo_pair`</code> 関数を簡素化します。                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                  |
|-----------|---------|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Feature | グローバルロックを取得する前に接続/ラグチェックを追加して、明らかな接続の問題がある場合にアプリケーションまたはユーザーがロックタイムアウトを取得するまで数分待つ必要がないようにします。DEBUG、LOG、WARNING（デフォルト）または ERROR に設定できます。                      |
| BDR       | 4.1.0   | Feature | デフォルトでは、競合のみを競合ログテーブルに記録します。デフォルトではサーバーログファイルに記録されなくなりましたが、これはオーバーライドできます。                                                                                   |
| BDR       | 4.1.0   | Feature | ノード参加中のリモートエラーのレポートを改善します。                                                                                                                                   |
| BDR       | 4.1.0   | Feature | 自動パーティションワーカーの最大昼寝時間を設定可能にします。                                                                                                                               |
| BDR       | 4.1.0   | Feature | autopartition で指定された上限までパーティションを要求する機能を追加します。                                                                                                                |
| BDR       | 4.1.0   | Feature | サブスクリプト専用ノードでレプリケート DDL を実行しないでください。複製する場所がないため、複製しようとしても失敗します。これは、ロジカルスタンバイの動作と同じです。                                                                        |
| BDR       | 4.1.0   | Feature | <code>`bdr.accept_connections`</code> 構成変数を追加します。 <code>`false`</code> の場合、BDR 出力プラグインを使用したレプリケーションスロットへの walsender 接続が失敗します。これは、主にバックアップからの単一ノードの復元中に役立ちます。 |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                           |
|-----------|---------|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Bug fix | <code>`lock_timeout`</code> を CAMO 以外の BDR ノードで構成したままにします。CAMO パートナーは、オリジンノードからトランザクションを適用するときに低い <code>`lock_timeout`</code> を使用します。これは CAMO パートナーだけでなく、すべての BDR ノードに対して誤って実行されたため、通常の BDR ノードの pglogical ライタブプロセスで誤った <code>`lock_timeout`</code> エラーが発生した可能性があります。 |
| BDR       | 4.1.0   | Bug fix | CAMO / Eager 確認待機の適切な待機イベントを表示します。デフォルトの「不明な待機イベント」の代わりに、 <code>`bdr.stat_activity`</code> の正しい「BDR 準備フェーズ」/「BDR コミットフェーズ」を表示します。(RT75900)                                                                                                                            |
| BDR       | 4.1.0   | Bug fix | <code>bdr.run_on_nodes</code> のログを減らします。関数の「 <code>run_on_nodes</code> 」バリエーションで完了した場合、 <code>`bdr.ddl_replication`</code> をオフに設定してもログに記録しません。これにより、監視機能のログの氾濫が解消されます。(RT80973)                                                                                       |
| BDR       | 4.1.0   | Bug fix | バイナリネットワークエンコーディングをサポートしていない複合型の配列と組み込み型の配列のレプリケーションを修正                                                                                                                                                                                                               |
| BDR       | 4.1.0   | Bug fix | ブートストラップ中に作成されたデータ型のレプリケーションを修正                                                                                                                                                                                                                                       |
| BDR       | 4.1.0   | Bug fix | WAL デコーダーによって処理された実行中のトランザクションレコードの終了 LSN を確認して、WAL デコーダースロットが最新の状態を維持し、WAL 送信者がタイムリーに候補を取得します。                                                                                                                                                                       |
| BDR       | 4.1.0   | Bug fix | 分割ノードで自動パーティションタスクが完了するのを待たない                                                                                                                                                                                                                                         |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                          |
|-----------|---------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Bug fix | フラッシュ位置を物理スロットのみに報告する場合、 <code>`bdr.standby_slot_names`</code> チェックを制限します。そうしないと、 <code>`bdr.standby_slot_names`</code> の使用時に切断されたノードが存在する場合にフラッシュの進行状況が報告されません。(RT77985、RT78290)                                    |
| BDR       | 4.1.0   | Bug fix | <code>wal_receiver_timeout</code> が近い場合は、 <code>walsender</code> にフィードバックの返信を要求します                                                                                                                                   |
| BDR       | 4.1.0   | Bug fix | BDR 拡張に対する自動パーティションテーブルの依存関係を複数回記録しないでください。これにより、これが発生すると、競合の自動パーティションと破損したレプリケーションから「ERROR: 予期しない数の拡張機能の依存関係レコード」エラーが発生しました。<br><p> <code>`pg_depend`</code> から二重の依存関係を削除して、既存の壊れたテーブルを手動で修正する必要があることに注意してください。 p>。 |
| BDR       | 4.1.0   | Bug fix | 受信機のキープアライブ処理を改善します。ストリーミングトランザクションの途中でキープアライブに基づいて位置を更新しないでください。クラッシュ時にデータを失う可能性があります。また、低 TPS シナリオでの待機時間を改善する新しいフラッシュ ロジックとシグナリング ロジックがあります。                                                                       |
| BDR       | 4.1.0   | Bug fix | BDR ノードがデータベースに存在する場合にのみ、 <code>post `CREATE`</code> コマンドを処理します。                                                                                                                                                     |
| BDR       | 4.1.0   | Bug fix | 競合履歴テーブルに ERROR 競合をログに記録しないでください。                                                                                                                                                                                    |
| BDR       | 4.1.0   | Bug fix | マルチ挿入 (COPY) 中に <code>conflict_slot</code> が解放された後に使用されていた <code>segfault</code> を修正しました (RT76439)。                                                                                                                  |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                               |
|-----------|---------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Bug fix | 遅延スタンバイスロットに直面したときに walsender プロセスがスピンするのを防ぎます。スタンバイスロットの 1 つが遅れている場合にワルセnderプロセスが CPU の 100 %を消費するように、ラッチをリセットするシグナリングを修正しました。(RT80295、RT78290)                          |
| BDR       | 4.1.0   | Bug fix | <code>`bdr.standby_slot_names`</code> 使用時の <code>`wal_sender_timeout`</code> のハンドリングを修正 (RT78290)                                                                         |
| BDR       | 4.1.0   | Bug fix | <code>`bdr.monitor_local_replslots`</code> の切断されたスロットのレポートを修正しました。これらは、以前に切断されたのではなく、欠落していると報告された可能性があります。                                                                |
| BDR       | 4.1.0   | Bug fix | <code>`bdr.get_subscription_progress()`</code> ファンクションおよびそのファンクションを使用する <code>`bdr.subscription_summary`</code> で のダウンロードしたサブスクリプションのタイムスタンプレポートを修正しました。以前はゴミの値を報告していました。 |
| BDR       | 4.1.0   | Bug fix | BDR ワーカーのさまざまな場所でのスナップショットの処理を修正しました。                                                                                                                                     |
| BDR       | 4.1.0   | Bug fix | モニタリング機能で使用可能な値がない場合、タイムスタンプと LSN を NULL として報告することについて、より一貫性を保ちます。                                                                                                        |
| BDR       | 4.1.0   | Bug fix | ライタプロセスを切り替えるときのログ情報を削減します。                                                                                                                                               |
| BDR       | 4.1.0   | Bug fix | PG コマンドラインで構成パラメーターが指定された場合、スーパーユーザーチェックを行いません。そこではまだトランザクションを実行できません。その段階でスーパーユーザーが変更されることが保証されています。                                                                     |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                          |
|-----------|---------|---------|----------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.1.0   | Bug fix | バイト単位のラグサイズを計算するには 64 ビットを使用します。ラグの大きいオーバーフローのリスクをなくす。                                                               |
| HARP      | 2.1.0   | Feature | BDR DCS は、ポーリングノードではなく、コンセンサスからのプッシュ通知を使用するようになりました。この変更により、新しいリーダーの選択にかかる時間が短縮され、HARP が BDR DCS で実行する負荷が削減されます。     |
| HARP      | 2.1.0   | Feature | TPA は各 HARP プロキシを 1 つずつ再起動し、戻ってくるまで待機して、ソフトウェアのアップグレード中にアプリケーションで発生したダウンタイムを削減します。                                   |
| HARP      | 2.1.0   | Feature | PGBouncer を HARP プロキシに直接埋め込むサポートは非推奨になり、HARP の次のメジャーリリースで削除されます。PG-Bouncer を HARPHARP をポイントするように TPA を構成できるようになりました。 |
| HARP      | 2.1.0   | Bug fix | <code>\harpctl promote &lt;node_name&gt;</code> は、指定されたノードとは異なるノードを昇格させることがありました。これは修正されました。(RT75406)                |
| HARP      | 2.1.0   | Bug fix | 分散コンセンサスサービスとして BDR を使用すると、フェンシングが失敗することがありました。これは修正されました。                                                           |
| HARP      | 2.1.0   | Bug fix | <code>\harpctl apply</code> は、クラスターが確立された後にリーダーのルーティングをオフにしなくなりました。(RT80790)                                         |
| HARP      | 2.1.0   | Bug fix | 障害が発生したデータベースを起動できない場合に、Harp-manager が終了しなくなりました。Harp-manager は、ランダムに増加する期間で再試行し続けます。(RT78516)                       |

次のページに続く

表 4 – 前のページからの続き

| Component | Version | Type    | Description                                   |
|-----------|---------|---------|-----------------------------------------------|
| HARP      | 2.1.0   | Bug fix | 内部 pgbouncer プロキシ実装でメモリリークが発生しました。これは修正されました。 |

## 1.4 Release notes for EDB Postgres Distributed version 4.0.3

これは、以前のバージョンで特定された問題の修正を含む HARP 2 のパッチリリースです。

| Component | Version | Type        | Description                                           |
|-----------|---------|-------------|-------------------------------------------------------|
| HARP      | 2.0.3   | Enhancement | HARP プロキシは読み取り専用のユーザー専用 TLS 証明書をサポート (RT78516)        |
| HARP      | 2.0.3   | Bug fix     | HARP プロキシは、50 秒後に終了する代わりに DCS への接続を試行し続けます。 (RT75406) |

## 1.5 Release notes for EDB Postgres Distributed version 4.0.2

これは BDR 4.0 および HARP 2.0 のメンテナンスリリースであり、以前のバージョンで特定された問題の修正とマイナーな改善が含まれています。

| Component | Version | Type        | Description                                                                                                                                                                                                                   |
|-----------|---------|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.2   | Enhancement | <i>bdr.max_worker_backoff_delay</i> を追加 (BDR-1767) <p>これにより、バックオフ遅延の処理が変更され、エラーが繰り返される場合に <i>bdr.min_worker_backoff_delay</i> から <i>bdr.max_worker_backoff_delay</i> に指数関数的に増加します。これにより、ログのスパムが減少し、場合によっては不要な接続試行も防止されます。</p> |

次のページに続く

表 6 – 前のページからの続き

| Component | Version | Type        | Description                                                                                                                                                                                    |
|-----------|---------|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.2   | Enhancement | <code>bdr.replicate_ddl_command()</code> に<br><code>`execute_locally`</code> オプションを追加 (RT73533) <p>これにより、ローカルで実行せずに他のグループにレプリケーションするための ddl コマンドをオプションでキューイングできます。 </p>                        |
| BDR       | 4.0.2   | Enhancement | JOIN 中のコンセンサス問題の<br>ERROR を WARNING<p>に変更<br>する これらの一時的なエラーの<br>報告は、bdr.worker_errors にも表<br>示されるため、混乱を招きました。<br>これらは WARNING に変更され<br>ました。 </p>                                               |
| BDR       | 4.0.2   | Bug fix     | WAL デコーダーは、実行中のトラ<br>ンザクションレコードの終了 LSN<br>を確認します (BDR-1264) <p><br>WAL デコーダーによって処理さ<br>れた実行中のトランザクションレ<br>コードの終了 LSN を確認して、<br>WAL デコーダースロットが最新<br>の状態を維持し、WAL 送信者が<br>タイムリーに候補を取得します。<br></p> |
| BDR       | 4.0.2   | Bug fix     | 分割ノードで自動パーティション<br>タスクが完了するまで待機しない<br>でください (BDR-1867) <p>.</p>                                                                                                                                |
| BDR       | 4.0.2   | Bug fix     | 並列結合中のノード名の再利用の<br>処理を改善しました (RT74789)<br><p>ノードに世代番号があるため、<br>ノードレコードがスナップショッ<br>トの一部として受信された場合で<br>も名前の再利用を簡単に識別でき<br>ます。 </p>                                                               |
| BDR       | 4.0.2   | Bug fix     | BDR マネージャープロセスでの<br>ノード管理中のロックとスナッ<br>プショットの使用を修正しました<br>(RT74789) <p>が考慮されます。<br></p>                                                                                                         |

次のページに続く



表 6 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                   |
|-----------|---------|---------|-------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.2   | Bug fix | ローカルノードのドロップでのカタログのクリーンアップを改善します<p>プライマリだけでなく、すべてのグループを削除し、すべてのノード状態の履歴情報も削除します。</p>                                          |
| BDR       | 4.0.2   | Bug fix | bdr_init_physical<p>の参加リクエストのエラーチェックを改善                                                                                       |
| BDR       | 4.0.2   | Bug fix | コンセンサスでのさまざまなタイムアウトとスリープの処理を改善します<p>これにより、多くのコンセンサスリクエストや時間のかかるコンセンサスリクエストを処理するときに必要な新しいコンセンサス投票の量が削減されます。たとえば、新しいノードの参加。</p> |
| BDR       | 4.0.2   | Bug fix | `wal_receiver_timeout`の処理を修正しました (BDR-1848) <p> BDR 3.7および4.0でのリグレッションのため、`wal_receiver_timeout`は正しくトリガーされませんでした。</p>         |
| BDR       | 4.0.2   | Bug fix | フラッシュ位置を物理スロットのみに報告するときに`bdr.standby_slot_names`チェックを制限します。                                                                   |
| BDR       | 4.0.2   | Bug fix | ブートストラップ中に作成されたデータ型のレプリケーションを修正しました (BDR-1784)                                                                                |
| BDR       | 4.0.2   | Bug fix | バイナリ転送をサポートしていないビルトイン型の配列のレプリケーションを修正しました (BDR-1042)                                                                          |
| BDR       | 4.0.2   | Bug fix | CAMO が使用されていない場合に CAMO 構成の警告が表示されないようにしました (BDR-1825)                                                                         |

次のページに続く

表 6 – 前のページからの続き

| Component | Version | Type        | Description                                                                                                                                                                                                                                                                                         |
|-----------|---------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HARP      | 2.0.2   | Enhancement | BDR コンセンサスが一般提供になりました。<p> HARP は、分散コンセンサスサービス (DCS) ソースに複数のオプションを提供します： BDR。 BDR コンセンサスオプションは、etcd が存在しないデプロイメントで使用できます。 BDR コンセンサスオプションの使用はベータとは見なされなくなり、実稼働環境での使用がサポートされるようになりました。 </p>                                                                                                           |
| HARP      | 2.0.2   | Enhancement | トランスポートレイヤープロキシの一般提供が開始されました。<p> HARP は、クライアントアプリケーションとデータベース間の接続をルーティングするための複数のプロキシオプションを提供します。アプリケーションレイヤー (L7) とトランスポートレイヤー (L4)。ネットワークレイヤー 4 またはトランスポートレイヤープロキシはネットワークパケットを転送するだけで、レイヤー 7 はネットワークトラフィックを終了します。以前はシンプルプロキシと呼ばれていたトランスポートレイヤープロキシは、ベータ版とは見なされなくなり、実稼働環境での使用がサポートされるようになりました。 </p> |

## 1.6 Release notes for EDB Postgres Distributed version 4.0.1

これは BDR 4.0 および HARP 2.0 のメンテナンスリリースであり、以前のバージョンで特定された問題の修正と、マイナーな機能強化が含まれています。

| Component | Version | Type        | Description                                                                                                                                                               |
|-----------|---------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.1   | Enhancement | CAMO パートナーの接続試行の頻度を減らします。 <p>構成を確認し、トランザクションのステータスを確認するために CAMO パートナーに接続できない場合、すぐに再試行しないでください（完全にビジーな pglogical マネージャープロセスにつながります）。 1 分に 1 回。 </p>                        |
| BDR       | 4.0.1   | Enhancement | LCR セグメントファイルのバッファ読み取りを実装します (BDR-1422) <p>一度に複数の LCR チャンクを読み取れるように、LCR セグメントファイルのバッファリングを実装します。これにより、Decoding Worker を使用する際の Wal Senders の I/O が削減され、CPU 使用率が向上します。 </p> |
| BDR       | 4.0.1   | Enhancement | 不要な LCR セグメントの読み取りを回避 (BDR-1426) <p>BDR は、少なくとも 1 つの使用可能な場合に新しい LCR セグメントのみを読み取ろうとします。これにより、Decoding Worker が有効になっているときの I/O 負荷が削減されます。 </p>                              |
| BDR       | 4.0.1   | Enhancement | 結合中の初期 COPY を含む COPY レプリケーションのパフォーマンスが、パーティションテーブルで大幅に改善されました (BDR-1479) <p>大きなテーブルの場合、これによりロード時間が 1 桁以上改善されます。 </p>                                                      |
| BDR       | 4.0.1   | Bug fix     | パラレル適用ワーカーの選択を修正します (BDR-1761) <p>これにより、パラレル適用が再度動作します。 4.0.0 では、このバグのため、並列適用は有効になりませんでした。 </p>                                                                           |
| BDR       | 4.0.1   | Bug fix     | `bdr.camo_pairs` の Raft スナップショットの処理を修正しました (BDR-1753) <p>以前のリリースでは、Raft スナップショットを介して受信したときに CAMO ペア構成に変更を正しく伝播しませんでした。 </p>                                               |
| BDR       | 4.0.1   | Bug fix     | アップグレード後に BDR 3.7 から Raft スナップショットを正しく処理する (BDR-1754)                                                                                                                     |
| BDR       | 4.0.1   | Bug fix     | クラスターのインプレースアップグレードを実行する機能を除外しながら、スナップショットの <i>bdr.camo_pairs</i> を考慮して CAMO 構成済みクラスターをアップグレードする（CAMO とは関係のないアップグレードの制限のため）。                                              |

次のページに続く

表 7 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                      |
|-----------|---------|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.1   | Bug fix | タイムアウト後にのみ CAMO からローカルモードに切り替える (RT74892) <p> CAMO 保護からローカルモードに切り替えるときに `catchup_interval` 推定値を使用しないでください。推定値は、ローカルモードから CAMO 保護に戻す場合にのみ使用してください (CAMO パートナーでの遅延が原因でトグルするのを防ぐため)。</p>                                                                                                                            |
| BDR       | 4.0.1   | Bug fix | パブリッシュされたレプリケーションセットリストが変更されたときのレプリケーションセットキャッシュの無効化を修正しました (BDR-1715) <p>以前のバージョンでは、サブスクリプションが再接続されるまでにパブリッシュする必要があるレプリケーションセット (および結果としてのどのテーブル) に関する古い情報を使用できました。</p>                                                                                                                                         |
| BDR       | 4.0.1   | Bug fix | 新しいチャンクが使用される場合、同時実行性の高い状況で galloc シーケンスによってローカルに生成される値の重複を防止します (RT76528) <p>新しい値を待っているセッション。これは修正されました。</p>                                                                                                                                                                                                    |
| BDR       | 4.0.1   | Bug fix | ストリーミングトランザクションでのメモリーリークに対処します (BDR-1479) <p>大規模なトランザクションの場合、ストリーミングトランザクション機能を使用すると、メモリ使用量と I/O が大幅に削減されます。これにより、主に COPY レプリケーションのパフォーマンスが向上します。</p>                                                                                                                                                            |
| BDR       | 4.0.1   | Bug fix | ノードが分離しているときにキャッチアップソースが変更された場合、ノードの PART_CATCHUP フェーズの後にスロットを残さないでください (BDR BDR) <p>すべてのノードのデータの一貫性を保つために、そのノードから欠落している変更を残りのノード間で転送します。これには、追加のレプリケーションスロットを一時的に作成する必要があります。通常、このレプリケーションスロットはキャッチアップフェーズの最後に削除されますが、変更のソースノードを変更する必要がある特定のシナリオでは、このスロットは以前に取り残されている可能性があります。このバージョンから、このスロットは常に正しく削除されます。</p> |

次のページに続く

表 7 – 前のページからの続き

| Component | Version | Type        | Description                                                                                                                                             |
|-----------|---------|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.1   | Bug fix     | BDR グループにノードが1つしかない場合は、グループスロットを前方に移動します<p>これにより、グループが単一の BDR ノードだけで実行されたままになった場合の WAL の蓄積によるディスクの枯渇を防ぎます。これは推奨される設定ではありませんが、WAL の蓄積は意図的なものではありません。</p> |
| BDR       | 4.0.1   | Bug fix     | BDR グループにノードが1つしかない場合の Advanced Raft プロトコルバージョン<p>それ以外の場合、シングルノードクラスターは、別のノードが追加されるまで常に最も古いサポートプロトコルを使用します。これにより、その単一のノードで使用可能な機能セットが制限される可能性があります。</p> |
| HARP      | 2.0.1   | Enhancement | etcd のような DCS に依存して異なるロケーションに個別にセットアップするのではなく、ロケーションごとにリーダーを選択するためのサポート。それでも、ロケーションの損失を生き残るために大部分のノードが必要になるため、ロケーションとデータベースノードの両方を奇数にすることをお勧めします。       |
| HARP      | 2.0.1   | Enhancement | BDR DCS は、ポーリングノードではなく、コンセンサスからのプッシュ通知を使用するようになりました。この変更により、新しいリーダーの選択にかかる時間が短縮され、HARP が BDR DCS で実行する負荷が削減されます。                                        |
| HARP      | 2.0.1   | Enhancement | TPA は各 HARP プロキシを 1 つずつ再起動し、戻ってくるまで待機して、ソフトウェアのアップグレード中にアプリケーションで発生したダウンタイムを削減します。                                                                      |
| HARP      | 2.0.1   | Enhancement | PGBouncer を HARP プロキシに直接埋め込むサポートは非推奨になり、HARP の次のメジャーリリースで削除されます。PGBouncer を HARP HARP をポイントするように TPA を構成できるようになりました。                                    |
| HARP      | 2.0.1   | Bug fix     | <code>harpctl promote &lt;node_name&gt;</code> は、指定されたノードとは異なるノードを昇格させることがありました。これは修正されました。[サポートチケット # 75406]                                           |
| HARP      | 2.0.1   | Bug fix     | 分散コンセンサスサービスとして BDR を使用すると、フェンシングが失敗することがありました。これは修正されました。                                                                                              |

次のページに続く

表 7 – 前のページからの続き

| Component | Version | Type    | Description                                                                                             |  |
|-----------|---------|---------|---------------------------------------------------------------------------------------------------------|--|
| HARP      | 2.0.1   | Bug fix | `harpctl apply`は、クラスターが確立された後にリーダーのルーティングをオフにしなくなりました。[サポートチケット # 80790]                                |  |
| HARP      | 2.0.1   | Bug fix | 障害が発生したデータベースを起動できない場合に、Harp-manager が終了しなくなりました。Harp-manager は、ランダムに増加する期間で再試行し続けます。[サポートチケット # 78516] |  |
| HARP      | 2.0.1   | Bug fix | 内部 pgbouncer プロキシ実装でメモリリークが発生しました。これは修正されました。                                                           |  |

## 1.7 Release notes for EDB Postgres Distributed version 4.0.0

EDB Postgres Distributed バージョン 4.0.0 には、BDR 34.0 および HARP 2.0 が含まれています。BDR 4.0 は BDR の新しいメジャーバージョンであり、このリリース番号で採用されたのはセマンティックバージョンングです（詳細については、

[semver.org](https://semver.org) を参照）。以前の 2 つのメジャーバージョンは 3.7 と 3.6 です。

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|-----------|---------|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.0   | Feature | <p>EDB Postgres Advanced 14 の BDR は、以前は EDB Postgres Extended でのみ使用できた次の機能をサポートするようになりました。</p> <ul style="list-style-type: none"> <li>- Commit At Most Once - ノードに障害が発生した場合でも、アプリケーションが各トランザクションを 1 回のみコミットするのに役立つ一貫性機能</li> <li>- 積極的な Eager Replication- トランザクションをコミットする前にクラスターのノード間を同期して、競合のないレプリケーションを提供します</li> <li>- ワーカーのデコード - ウォルセnderから別のワーカーへのデコードの分離により、多くのノードでのスケーラビリティが向上</li> <li>- レプリケーションのキャッチアップ時間の見積もり</li> <li>- タイムスタンプベースのスナップショット-複数のノードにわたって一貫した読み取りを提供し、特定の時刻に表示されるまたは表示されるデータを取得します</li> <li>- 行フリーズの自動化された動的構成特定のコーナーケースでの UPDATE / DELETE の競合解決の一貫性を向上させるため</li> <li>- 評価チェック</li> <li>- 欠落しているパーティションをエラーではなく競合として処理するためのサポート</li> <li>- NOT VALID 制約と ALTER TABLE の高度な DDL 処理</li> </ul> |

次のページに続く

表 8 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|-----------|---------|---------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.0   | Feature | <p>PostgreSQL 12-14 のコミュニティバージョンの BDR は、以前は EDB Postgres Advanced または EDB Postgres Extended でのみ使用できた次の機能をサポートするようになりました。</p> <ul style="list-style-type: none"> <li>競合のないレプリケートされたデータ型- 非同期マルチプルで数学的に証明された一貫性を提供する追加のデータ型-マスター更新シナリオ</li> <li>列レベルの競合解決- 列ごとに使用する機能の最終更新が解決を優先するため、異なるフィールドの UPDATE をいずれも失うことなく「マージ」できます</li> <li>変換トリガー-そのトリガーデータの着信ストリームで実行され、変更したり、高度なプログラムによるフィルタリングを実行したりできます</li> <li>競合トリガー-競合が検出されたときに呼び出され、カスタムの競合解決手法を使用する方法を提供します</li> <li>CREATE TABLE AS レプリケーション</li> <li>パラレル適用-複数のライターが着信変更を適用できるようにします</li> </ul> |

次のページに続く



表 8 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-----------|---------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.0   | Feature | <p>大規模なトランザクションのストリーミングをサポートします。</p> <p>&lt;p&gt;これにより、BDR は大きなトランザクション（サイズが `logical_decoding_work_mem` を超える）をダウンストリームのファイルまたはライタプロセスにストリーミングできます。これにより、トランザクションはコミットされる前でもデコードされるため、並列性が向上します。さらに、ライタに直接ストリーミングする場合、トランザクションを同時に適用することもできます。これにより、並列性がさらに向上します。 &lt;/p&gt;&lt;p&gt;大規模なトランザクションがファイルにストリーミングされると、それらはデコードされ、デコードされた変更はコミットされる前にダウンストリームに送信されます。変更は一連のファイルに書き込まれ、トランザクションが最終的にコミットしたときに適用されます。トランザクションが打ち切られると、変更は破棄されるため、アップストリームとダウンストリームの両方でリソースが無駄になります。 &lt;/p&gt;&lt;p&gt;サブトランザクションも自動的に処理されます。 &lt;/p&gt;&lt;p&gt;この機能は、PostgreSQL 14、EDB Postgres Extended 13 以降、および EDB Postgres Advanced 14 で利用できます。Postgres のどのバージョンで利用できる機能の詳細については、 :ref: `Choosing a Postgres distribution`&lt;Choosing a Postgres distribution&gt; 付録を参照してください。 &lt;/p&gt;</p> |

次のページに続く

表 8 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|-----------|---------|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.0   | Feature | <p>標準版とエンタープライズ版の BDR の以前のバージョンに存在した違いは削除されました。BDR 4.0 では、サポートされている Postgres ディストリビューションとバージョン、つまり PostgreSQL v12-14、EDB Postgres Extended v12-14、および EDB Postgres Advanced 12-14 ごとに 1 つの拡張機能があります。 &lt;p&gt;すべての機能が PostgreSQL のすべてのバージョンで利用できるわけではありません。利用可能な機能は、`bdr_config` コマンドラインユーティリティまたは `bdr.bdr_features()` データベース関数を使用して機能フラグを介して報告されます。詳細については、:ref:`Choosing a Postgres distribution&lt;Choosing a Postgres distribution&gt;` を参照してください。 &lt;/p&gt;</p> |
| BDR       | 4.0.0   | Feature | <p>BDR 4.0 拡張機能に対応する pglogical 4.0 拡張機能はありません。BDR には pglogical がなくなりました。 &lt;p&gt;これは、BDR 拡張機能とスキーマのみが存在し、構成パラメーターの名前が `pglogical.` から `bdr.` に変更されたことも意味します。 &lt;/p&gt;</p>                                                                                                                                                                                                                                                                                           |

次のページに続く

表 8 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------|---------|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.0   | Feature | 一部の構成オプションでは、インストール後のエクスペリエンスを向上させるためにデフォルトが変更されています。<br />-並列適用がデフォルトで有効になりました（2 つのライター）。特にストリーミングが有効になっている場合、パフォーマンスが向上します。 <br />-`COPY`および`CREATE INDEX CONCURRENTLY`は、デフォルトで使用可能なすべてのノードに並列で（ストリーミングがサポートされている Postgres バージョンで）ライターにストリーミングされ、レプリケーションのラグスパイクを排除または少なくとも削減します。これらの操作の後。<br />-グローバルロックのタイムアウトが 10 分に延長されました<br />-`bdr.min_worker_backoff_delay`はデフォルトで 1 秒になり、サブスクリプションはエラー時に 1 秒に 1 回のみ接続を再試行します。 |
| BDR       | 4.0.0   | Feature | オリジンベースの競合検出を使用するテーブルのノード結合中の競合検出で誤検知の可能性が大幅に減少しました                                                                                                                                                                                                                                                                                                                                                                      |

次のページに続く

表 8 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------|---------|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BDR       | 4.0.0   | Feature | <p>CAMO ペアの構成を SQL カタログに移動する&lt;p&gt; 構成ミスの可能性を減らし、 BDR クラスター内の CAMO ペアをグローバルに認識するには、CAMO 構成を個々のノードの postgresql.conf から Raft が管理する BDR システムカタログに移動します。これにより、たとえば、別のアクティブノードの CAMO パートナーとしてまだ構成されているノードを誤って削除するのを防ぐことができます。 &lt;/p&gt;&lt;p&gt;アップグレードプロセスの詳細については、 :ref:`Upgrades chapter &lt;/pgd/latest/upgrades/#upgrading-a-camo-enabled-cluster&gt;` を参照してください。 &lt;/p&gt;&lt;p&gt;これは GUC `bdr.camo_partner_of` および `bdr.camo_origin_for` を非推奨にし、関数 `bdr.get_configured_camo_origin_for()` および `get_configured_camo_partner_of` を `bdr.get_configured_camo_partner` に置き換えます。 &lt;/p&gt;</p> |
| HARP      | 2.0.0   | Change  | すべての操作を最適化するための golang のシステムの完全な書き換え                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| HARP      | 2.0.0   | Change  | クラスターの状態を YAML 経由でブートストラップまたは修正できるようになりました                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| HARP      | 2.0.0   | Feature | 構成が YAML になり、構成ファイルが `harp.ini` から `config.yml` に変更                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

次のページに続く

表 8 – 前のページからの続き

| Component | Version | Type    | Description                                                                                                                                                                                                                        |
|-----------|---------|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| HARP      | 2.0.0   | Feature | HARP プロキシは、サポートされているアーキテクチャでの HAProxy の必要性を非推奨にします。<p>HARP ルーターを使用して DCS コンテンツを HTTP ベースの URI 要求の適切なオンラインまたはオフライン状態に変換することは、リードマスターを決定するためにロードバランサーまたは HAProxy が必要であることを意味しました。HARP プロキシは、定期的な反復ステータスチェックなしでこれを自動的に行うようになりました。</p> |
| HARP      | 2.0.0   | Feature | DCS キーサブスクリプションを利用して、状態の変更に直接応答します。<p>関連するクラスターの状態の変更により、クラスターはすぐに応答するため、フェールオーバーとスイッチオーバーの時間が向上します。</p>                                                                                                                           |
| HARP      | 2.0.0   | Feature | etcd SSL 設定との互換性。<p>SSL 暗号化を介して etcd と通信できるようになりました。</p>                                                                                                                                                                           |
| HARP      | 2.0.0   | Feature | スイッチオーバー時のトランザクションラグはゼロです。<p>レプリケートされたすべてのトランザクションが再生されるまで、トランザクションは新しいリードノードにルーティングされないため、競合の可能性が減少します。</p>                                                                                                                       |
| HARP      | 2.0.0   | Feature | 実験的な BDR コンセンサスレイヤー。<p>分散コンセンサスサービス（DCS）として BDR コンセンサスを使用すると、実装に必要な変更の量が削減されます。</p>                                                                                                                                                |
| HARP      | 2.0.0   | Feature | 実験的なビルトインプロキシ。<p>セッション制御を強化するためのプロキシ実装。</p>                                                                                                                                                                                        |



## 第2章 ‘Known issues’

このセクションでは、EDB Postgres Distributed 4 の現在の既知の問題について説明します。

### 2.1 データの一貫性

**競合の監視** について読んで、データの一貫性の観点からの非同期操作モードの影響を理解してください。

### 2.2 問題のリスト

これらの既知の問題は BDR のチケットシステムで追跡されており、将来のリリースで解決される予定です。

- フェールオーバーとスイッチオーバー時間の HARP のパフォーマンスは、DCS ノード間のレイテンシーに非線形に依存します。これが、複数のリージョン（通常はゴールドとプラチナのレイアウト）にわたる EDB Postgres 分散デプロイの場合、HARP にリージョンごとに etcd クラスターを使用することをお勧めします。config.yml で harp\_consensus\_protocol オプションが etcd に設定されている場合、TPAexec はこれらのリージョンクラスターごとに実行する etcd を既に設定しています。

レイテンシーの高いネットワーク全体での DCS 展開では、leader\_lease\_duration HARP オプション (TPAexec の harp\_leader\_lease\_duration) を増やすことをお勧めします。

- 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、BDR ライタでのデッドロックが原因で更新ステートメントがハングする場合があります。

**競合の監視** の章で述べたように、skip

は update\_origin\_change の競合のデフォルトのリゾルバーではなく、この組み合わせは実稼働での使用を想定していません。そのノードへの到着順に基づいて、2 つの競合する更新の一方を破棄します。これにより、クラスターが発散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。

- デコーディングワーカー機能は、CAMO/EAGER/Group Commit では動作しません。CAMO/Eager/Group Commit を使用したインストールでは、enable\_wal\_decoder を無効にしておく必要があります。
- Decoding Worker は、デフォルトのレプリケーションセットでのみ機能します。
- ラグ制御は、完全に分離されたノード、つまり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、コミット遅延を調整しません。少なくとも 1 つのノードが接続されるとすぐに、レプリケーションラグ制御が作業を取得し、BDR 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、BDR は現在、履歴適用率に基づいた推定キャッチアップタイムではなく、ラグタイム（コミットタイムスタンプで測定）を使用します。
- 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。ペアの追加・削除のみ可能です。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は必要ありません。
- Group Commit は `show-camo` または `Eager All Node replication` と組み合わせることはできません。Eager Replication は現在、「グローバル」な BDR コミットスコープを使用する場合にのみ機能します。
- Eager Replication も Group Commit も `synchronous_replication_availability = 'async'` をサポートしていません。
- Group Commit は、`bdr.global_commit_timeout` 後のコミットのタイムアウトをサポートしていません。
- 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。また、明示的な 2 フェーズコミットをサポートしていません。TRUNCATE コマンドは許可されています。
- CAMO または Group Commit を使用すると、すべての DDL を実行できるわけではありません。
- Group Commit との組み合わせでの並列適用は現在サポートされていません。Group Commit を使用するときは、ノードグループに対して `num_writers` を 1 に設定するか（`bdr.alter_node_group_config` を使用）、GUC `bdr.writers_per_subscription` を介して無効にしてください（

Configuration of Generic Replication を参照）。

- 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エラーが発生したため、トランザクションが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。特定のコミットスコープを使用するトランザクションは、変更または削除する前に完了していることを確認してください。

## 2.3 制限事項のリスト

これは、設計による予想される制限の（包括的ではない）リストです。将来的に解決される予定はありません。

- ノードをフィジカルスタンバイに置き換えても、CAMO/Eager/Group Commit を使用するノードでは機能しません。フィジカルスタンバイと BDR を組み合わせることは、一般的にはお勧めできません。
- `galloc` シーケンスがロールバックされたトランザクションで作成され、同じ名前でも再度作成された場合、いくつかのチャンクをスキップする場合があります。これは、DDL レプリケーションがアクティブでないときに作成および削除され、DDL レプリケーションがアクティブなときに再度作成された場合にも発生します。シーケンスの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()` 関数への引数として指定できます。
- レガシー BDR 同期レプリケーションは、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。2 つは互換性がないため、一緒に使用しないでください。したがって、`synchronous_standby_names` に表示されるノードは、CAMO、



Eager、または Group Commit 構成の一部であってはなりません。ロジカルスタンバイとフィジカルスタンバイの両方を含む、他のノードへの同期レプリケーションを使用できます。



## 第3章 Terminology

以下の用語は、EDB Postgres 分散機能と、高可用性、レプリケーション、クラスタリングの領域で対処する要件を理解するために重要です。

### 3.1 非同期レプリケーション

オリジンノードでトランザクションが完了した後、クラスターメンバーにデータをコピーします。非同期レプリケーションは、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、複数の変更が同時に発生するため、競合が発生する可能性があります。発生した競合を管理する必要があります。

### 3.2 可用性

システムが指定された環境で使用されたときに、指定された時間に十分に動作する確率。多くの人にとって、これはアプリケーションのアップタイムとダウンタイムの全体的な量です。（**ナイン** も参照）

### 3.3 CAMO または commit-at-most-once

アプリケーションレベルでの追加のトランザクション管理で Eager Replication をラップして、トランザクションが複数回実行されないようにします。これは、ペイメントソリューションで見られる高額のトランザクションにとって重要です。これは、Oracle 機能の Transaction Guard とほぼ同じです。

### 3.4 クラスタリング

単一障害点を回避するために複数の冗長システムを管理する高可用性のためのアプローチ。エンドユーザーには1つのシステムに見えます。

## 3.5 データシャーディング

データを**シャード**と呼ばれるチャンクに分割し、それらを個別のノードに分散することにより、データベースのスケールアウトを有効にします。

## 3.6 BDR の Eager Replication

すべてのクラスターメンバーとの競合のないレプリケーション。技術的には、これは2フェーズコミット(2PC)を使用した同期論理レプリケーションです。

## 3.7 結果的な一貫性

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、同じ値に収束します。BDR では、これは、競合解決と競合のないレプリケートされたデータ型を備えた非同期論理レプリケーションによって実現されます。

## 3.8 フェールオーバー

高可用性データベースクラスターの障害を認識し、アプリケーションを別のアクティブなデータベースに接続するためのアクションを実行する自動化されたプロセス。目標は、ダウンタイムとデータ損失を最小限に抑えることです。

## 3.9 水平スケーリング または スケールアウト

トラフィックの増加を処理するための Web サーバーのスケールアウトなど、複数のノードにわたるワークロードを管理する最新の分散コンピューティングアプローチ。

## 3.10 論理レプリケーション

クラスター内のデータベース間でレプリケートされるデータの選択に関して、物理レプリケーションよりも柔軟性があります。また、クラスターメンバーはデータベースソフトウェアの異なるバージョン上にあることができます。

## 3.11 ナイン

特定の年の稼働時間の割合として表される可用性の尺度。スリーナイン（99.9%）では、1 か月あたり 43.83 分のダウンタイムが許容されます。フォーナイン（99.99 %）では、1 か月あたり 4.38 分のダウンタイムが許可されます。ファイブナイン（99.999%）では、1 か月あたり 26.3 秒のダウンタイムが許容されます。

## 3.12 ノード

クラスター内の 1 つのデータベースサーバー。クラスター内に複数のノードがあるため、用語「ノード」は用語「データベースサーバー」とは異なります。ノードには、データベースサーバー、OS、および物理ハードウェアが含まれます。これらは、高可用性のコンテキストで常に他のノードから分離されます。

## 3.13 物理レプリケーション

データベースディスクブロックの正確なコピーをコピーすることにより、データベースから 1 つ以上のスタンバイクラスターメンバーにすべての変更をコピーします。この方法には高速ですが、欠点があります。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、この方法は、他のいくつかのより複雑な制限に加えて、すべてのクラスターメンバーがデータベースソフトウェアの同じメジャーバージョン上にある場合にのみ使用できます。

## 3.14 読み取りのスケラビリティ

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への直接書き込みとレプリカノードへの読み取りを行わせることで実現できます。読み取りワークロードが増加するにつれて、パフォーマンスを維持するためにリードレプリカノードの数を増やすことができます。

## 3.15 目標復旧時点（RPO）

アプリケーションの配信が中断されたためにデータが失われる可能性のある最大ターゲット期間。非常に低いまたは最小限の RPO は、非常に高い可用性の原動力です。

## 3.16 目標復旧時間（RTO）

中断されたアプリケーションを復元するための目標時間。非常に低いまたは最小限の RTO は、非常に高い可用性の原動力です。

### 3.17 単一障害点（SPOF）

冗長性がないため、より高いレベルの可用性を実現できない、展開されたアーキテクチャ内のコンポーネントの識別。

### 3.18 スイッチオーバー

アプリケーション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。

### 3.19 同期レプリケーション

参加しているすべてのノードで変更が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。このアプローチは即時の一貫性を提供し、競合を回避しますが、ノード間で必要な調整が原因でレイテンシーのパフォーマンスコストが発生します。

### 3.20 2 フェーズコミット（2PC）

複数のデータベースノードにわたって一貫性を実現するためのマルチステッププロセス。

### 3.21 垂直スケーリング または スケールアップ

リソース（CPU、メモリ、ストレージ、ネットワーク）を増やして、そのアーキテクチャの物理的な制限に達するまでサポートする従来のコンピューティングアプローチ（Oracle Exadata など）。

### 3.22 書き込みスケーラビリティ

元のノードから他のクラスターメンバーへの書き込みをレプリケートすると、コストが削減されます。垂直スケールのアーキテクチャでは、共有リソースにより書き込みスケーラビリティが可能です。ただし、水平スケーリング（または非共有）アーキテクチャでは、これは非常に限られたシナリオでのみ可能です。

## 第4章 Key components

EDB Postgres Distributed は、メッシュトポロジを使用して疎結合マルチマスター論理レプリケーションを提供します。これは、任意のサーバーに書き込むことができ、変更は同じメッシュの一部である他のすべてのサーバーに行ごとに直接送信されることを意味します。

EDB Postgres Distributed は、クラスター全体を機能させるいくつかのコンポーネントで構成されています。

### 4.1 Postgres サーバー

3つの異なる Postgres ディストリビューションを使用できます。

- 標準の *PostgreSQL* 統計ビュー - オープンソース
- EDB Postgres Extended Server -

PostgreSQL と互換性があり、レプリケーション用に最適化されています

- EDB Postgres Advanced Server - Oracle 互換、レプリケーション用に最適化、および追加のエンタープライズ機能

どの Postgres ディストリビューションとバージョンが適切かは、必要な機能によって異なります。詳細な比較については、

*Choosing a Postgres distribution* の機能マトリックスを参照してください。

*BDR メタデータのクリーンアップ*

---

*BDR メタデータのクリーンアップ* 拡張機能がインストールされた Postgres サーバーは、

BDR ノードと呼ばれます。BDR ノードは、データノードまたは監視ノードのいずれかです。

ウィットネスノードはデータレプリケーションに参加せず、コンセンサスのタイブレーカーとしてのみ使用されます。

*High Availability Routing for Postgres (HARP)*

---

HARP は、EDB Postgres 分散クラスター用の接続管理ツールです。

コンセンサス駆動のクォーラムを活用して、半排他的な方法で正しい接続エンドポイントを決定し、アプリケーションからの意図しないマルチノード書き込みを防ぎます。これにより、データの競合の可能性が減少します。

### 4.1.1 High Availability Routing for Postgres (HARP)

Postgres の高可用性ルーティング (HARP) は、Bi-directional Replication (BDR) クラスター用のクラスター管理ツールです。このツールのコアデザインは、単一のデータセンターまたはリージョン内のすべてのアプリケーショントラフィックを一度に 1 つのノードにのみルーティングすることです。リードマスターと指定されたこのノードは、データ競合の可能性を減らすための主な書き込みターゲットとして機能します。

HARP は分散コンセンサスモデルを活用して、クラスター内の BDR ノードの可用性を判断します。リードマスターに障害が発生したか使用できない場合、HARP は新しいリードマスターを選択し、アプリケーショントラフィックをリダイレクトします。

BDR のコア機能と合わせて、アプリケーショントラフィックをリードマスターノードにルーティングするこのメカニズムにより、データ損失のリスクなしで高速なフェイルオーバーとスイッチオーバーが可能になります。

HARP には BDR バージョン 3.6 以降が必要です。

#### Release Notes

Postgres の高可用性ルーティング (HARP) は、Bi-directional Replication (BDR) クラスター用のクラスター管理ツールです。このツールのコアデザインは、単一のデータセンターまたはリージョン内のすべてのアプリケーショントラフィックを一度に 1 つのノードにのみルーティングすることです。リードマスターと指定されたこのノードは、データ競合の可能性を減らすための主な書き込みターゲットとして機能します。

このセクションのリリースノートには、各リリースの新機能に関する情報が記載されています。

| Version              | Release Date |
|----------------------|--------------|
| <i>Version 2.1.1</i> | 2022 June 21 |
| <i>Version 2.1.0</i> | 2022 May 17  |
| <i>Version 2.0.3</i> | 2022 Mar 31  |
| <i>Version 2.0.2</i> | 2022 Feb 24  |
| <i>Version 2.0.1</i> | 2021 Jan 31  |
| <i>Version 2.0.0</i> | 2021 Dec 01  |

#### Version 2.1.1

これは、以前のバージョンで特定された問題の修正を含む HARP 2 のパッチリリースです。

| Type        | Description                                          |
|-------------|------------------------------------------------------|
| Enhancement | DCS 接続の損失に関する警告をログに記録する                              |
| Enhancement | メタデータの更新に時間がかかりすぎる場合に警告をログに記録します - 通常は高遅延ネットワークが原因です |
| Bug Fix     | 障害時に harp_proxy.service をリスタート                       |
| Bug Fix     | harpctl の接続管理の同時実行の問題を修正                             |

次のページに続く



表 2 – 前のページからの続き

| Type    | Description                             |
|---------|-----------------------------------------|
| Bug Fix | アンマネージドクラスターで以前のリーダーへのプロキシ接続を試行しないでください |
| Bug Fix | 場所が空の場合に haprctl でパニックにならない             |

### Version 2.1.0

これは HARP 2 のマイナーリリースで、新機能と、以前のバージョンで特定された問題の修正が含まれています。

| Type    | Description                                                                                                                                         |
|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | BDR DCS は、ポーリングノードではなく、コンセンサスからのプッシュ通知を使用するようになりました。 <p>この変更により、短い間隔でポーリングする必要がなくなったため、新しいリーダーの選択と HARP が BDR DCS で行う負荷が削減されます。 </p>                |
| Feature | TPA は各 HARP プロキシを 1 つずつ再起動し、戻ってくるまで待機して、ソフトウェアのアップグレード中にアプリケーションで発生したダウンタイムを削減します。                                                                  |
| Feature | PGBouncer を HARP プロキシに直接埋め込むサポートは非推奨になり、HARP の次のメジャーリリースで削除されます。 <p>PGBouncer を HARP プロキシと同じノードに配置し、その HARP プロキシをポイントするように TPA を構成できるようになりました。 </p> |
| Bug Fix | <code>haprctl promote &lt;node_name&gt;</code> は、指定されたノードとは異なるノードを昇格させることがありました。これは修正されました。 (RT75406)                                               |
| Bug Fix | 分散コンセンサスサービスとして BDR を使用すると、フェンシングが失敗することがありました。これは修正されました。                                                                                          |
| Bug Fix | <code>haprctl apply</code> は、クラスターが確立された後にリーダーのルーティングをオフにになりました。 (RT80790)                                                                          |
| Bug Fix | 障害が発生したデータベースを起動できない場合に、Harp-manager が終了しなくなりました。 Harp-manager は、ランダムに増加する期間で再試行し続けます。 (RT78516)                                                    |
| Bug Fix | 内部 pgbouncer プロキシ実装でメモリリークが発生しました。これは修正されました。                                                                                                       |

### Version 2.0.3

これは、以前のバージョンで特定された問題の修正を含む HARP 2 のパッチリリースです。

| Type        | Description                                    |
|-------------|------------------------------------------------|
| Enhancement | HARP プロキシは読み取り専用のユーザー専用 TLS 証明書をサポート (RT78516) |

次のページに続く

表 4 – 前のページからの続き

| Type    | Description                                           |
|---------|-------------------------------------------------------|
| Bug Fix | HARP プロキシは、50 秒後に終了する代わりに DCS への接続を試行し続けます。 (RT75406) |

## Version 2.0.2

これは、以前のバージョンで特定された問題の修正を含む HARP 2 のパッチリリースです。

| Type        | Description                                                                                                                                                                                                                                                                                             |
|-------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Enhancement | BDR コンセンサスが一般提供になりました。 <p> HARP は、分散コンセンサスサービス (DCS) ソースに複数のオプションを提供します: BDR。 BDR コンセンサスオプションは、etcd が存在しないデプロイメントで使用できます。 BDR コンセンサスオプションの使用はベータとは見なされなくなり、実稼働環境での使用がサポートされるようになりました。 </p>                                                                                                              |
| Enhancement | トランスポートレイヤープロキシの一般提供が開始されました。 <p> HARP は、クライアントアプリケーションとデータベース間の接続をルーティングするための複数のプロキシオプションを提供します。 アプリケーションレイヤー (L7) とトランスポートレイヤー (L4)。 ネットワークレイヤー 4 またはトランスポートレイヤープロキシはネットワークパケットを転送するだけで、レイヤー 7 はネットワークトラフィックを終了します。 以前はシンプルプロキシと呼ばれていたトランスポートレイヤープロキシは、ベータ版とは見なされなくなり、実稼働環境での使用がサポートされるようになりました。 </p> |

## Version 2.0.1

これは、以前のバージョンで特定された問題の修正を含む HARP 2 のパッチリリースです。

| Type        | Description                                                                                                                                        |
|-------------|----------------------------------------------------------------------------------------------------------------------------------------------------|
| Enhancement | etcd のような DCS に依存して異なるロケーションに個別にセットアップするのではなく、ロケーションごとにリーダーを選択するためのサポート。 これでも、ロケーションの損失を生き残るために大部分のノードが必要になるため、ロケーションとデータベースノードの両方を奇数にすることをお勧めします。 |
| Enhancement | BDR DCS は、ポーリングノードではなく、コンセンサスからのプッシュ通知を使用するようになりました。 この変更により、新しいリーダーの選択にかかる時間が短縮され、HARP が BDR DCS で実行する負荷が削減されます。                                  |
| Enhancement | TPA は各 HARP プロキシを 1 つずつ再起動し、戻ってくるまで待機して、ソフトウェアのアップグレード中にアプリケーションで発生したダウンタイムを削減します。                                                                 |

次のページに続く

表 6 – 前のページからの続き

| Type        | Description                                                                                                         |
|-------------|---------------------------------------------------------------------------------------------------------------------|
| Enhancement | PGBouncer を HARP プロキシに直接埋め込むサポートは非推奨になり、HARP の次のメジャーリリースで削除されます。PGBouncer を HARPHARP をポイントするように TPA を構成できるようになりました。 |
| Bug Fix     | <code>harpctl promote &lt;node_name&gt;</code> は、指定されたノードとは異なるノードを昇格させることができました。これは修正されました。[サポートチケット # 75406]       |
| Bug Fix     | 分散コンセンサスサービスとして BDR を使用すると、フェンシングが失敗することがありました。これは修正されました。                                                          |
| Bug Fix     | <code>harpctl apply</code> は、クラスターが確立された後にリーダーのルーティングをオフにになりました。[サポートチケット # 80790]                                  |
| Bug Fix     | 障害が発生したデータベースを起動できない場合に、Harp-manager が終了してなくなりました。Harp-manager は、ランダムに増加する期間で再試行し続けます。[サポートチケット # 78516]            |
| Bug Fix     | 内部 pgbouncer プロキシ実装でメモリリークが発生しました。これは修正されました。                                                                       |

## Version 2.0.0

これは、製品の完全な書き換えで構成される HARP の新しいメジャーリリースです。

| Type    | Description                                                                                                                                                                                                                         |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Engine  | すべての操作を最適化するための golang のシステムの完全な書き換え                                                                                                                                                                                                |
| Engine  | クラスターの状態を YAML 経由でブートストラップまたは修正できるようになりました                                                                                                                                                                                          |
| Feature | 構成が YAML になり、構成ファイルが <code>harp.ini</code> から <code>config.yml</code> に変更                                                                                                                                                           |
| Feature | HARP プロキシは、サポートされているアーキテクチャでの HAProxy の必要性を非推奨にします。<p> HARP ルーターを使用して DCS コンテンツを HTTP ベースの URI 要求の適切なオンラインまたはオフライン状態に変換することは、リードマスターを決定するためにロードバランサーまたは HAProxy が必要であることを意味しました。HARP プロキシは、定期的な反復ステータスチェックなしでこれを自動的に行うようになりました。</p> |
| Feature | DCS キーサブスクリプションを利用して、状態の変更に直接応答します。<p> 関連するクラスターの状態の変更により、クラスターはすぐに応答するため、フェールオーバーとスイッチオーバーの時間が向上します。</p>                                                                                                                           |
| Feature | etcd SSL 設定との互換性。<p>SSL 暗号化を介して etcd と通信できるようになりました。</p>                                                                                                                                                                            |
| Feature | スイッチオーバー時のトランザクションラグはゼロです。<p>レプリケートされたすべてのトランザクションが再生されるまで、トランザクションは新しいリードノードにルーティングされないため、競合の可能性が減少します。</p>                                                                                                                        |
| Feature | 実験的な BDR コンセンサスレイヤー。<p>分散コンセンサスサービス (DCS) として BDR コンセンサスを使用すると、実装に必要な変更の量が削減されます。</p>                                                                                                                                               |

次のページに続く

表 7 – 前のページからの続き

| Type    | Description                                 |
|---------|---------------------------------------------|
| Feature | 実験的なビルトインプロキシ。<p>セッション制御を強化するためのプロキシ実装。</p> |

## HARP functionality overview

HARP は、BDR クラスターの高可用性に対する新しいアプローチです。コンセンサス駆動のクォーラムを活用して、半排他的な方法で正しい接続エンドポイントを決定し、アプリケーションからの意図しないマルチノード書き込みを防ぎます。

### クォーラムの重要性

HARP の主な目的は、管理する Postgres クラスターにフルクォーラムを適用することです。クォーラムとは、決定を行うために特定の最小限の出席者を義務付ける投票機関に適用される用語です。より簡単に：多数決ルール。

同点以外の結果に終わるには、奇数のノードが完全なクラスターメンバーシップを構成する必要があります。ただし、クォーラムはこの制限を厳密に要求しません。単純過半数で十分です。これは、N ノードのクラスターでは、クォーラムが意味のある投票を保持するために少なくとも  $N/2+1$  ノードが必要であることを意味します。

これらすべてにより、クラスターが「担当」するノードに関して常に合意することが保証されます。複数のノードで構成される EDB Postgres 分散クラスターの場合、これはプライマリ書き込みターゲットであるノードを決定します。HARP はこのノードをリードマスターとして指定します。

### 書き込みターゲットの削減

クォーラムの概念を無視するか、それを十分に適用しないと、「正しい」書き込みターゲットがあいまいまたは不明な「スプリットブレイン」シナリオが発生する可能性があります。標準の Postgres クラスターでは、1つのノードのみが書き込み可能で、残りのノードにレプリケーショントラフィックを送信することが重要です。

BDR などのマルチマスター対応のアプローチでも、クラスター全体で同一のデータを取得するために必要な競合管理の量を削減すると役立ちます。物理的な場所またはリージョンごとに複数の BDR ノードで構成されるクラスターでは、これは通常、単一の BDR ノードが「リーダー」として機能し、残りのノードが「シャドウ」であることを意味します。これらのシャドウノードは引き続き書き込み可能ですが、どうしても必要な場合を除き、書き込みはお勧めしません。

クォーラムを活用することで、すべてのノードがクラスター全体またはローカル BDR リージョンを表す正確な Postgres ノードに同意できます。定義上、クォーラムの残りの部分との接続を失ったノード、またはクォーラムによって無効にされたノードはクラスターリーダーになることはできません。

この制限により、書き込みが意図せずに2つの Postgres ノードに到達するスプリットブレイン状況が防止されます。VPN、プロキシ、ロードバランサー、DNS などのテクノロジーとは異なり、構成ミスやネットワークパーティションによってクォーラムから派生したコンセンサスを回避することはできません。コン

センサスレイヤーに接続して、HARP によって維持されるクォラムの状態を判断できる限り、有効なターゲットは 1 つだけです。

### 基本的なアーキテクチャ

HARP のデザインは、基本的にマネージャーとプロキシの 2 つの部分で構成されています。次の図は、これらが単一の Postgres インスタンスとどのように相互作用するかを説明しています。

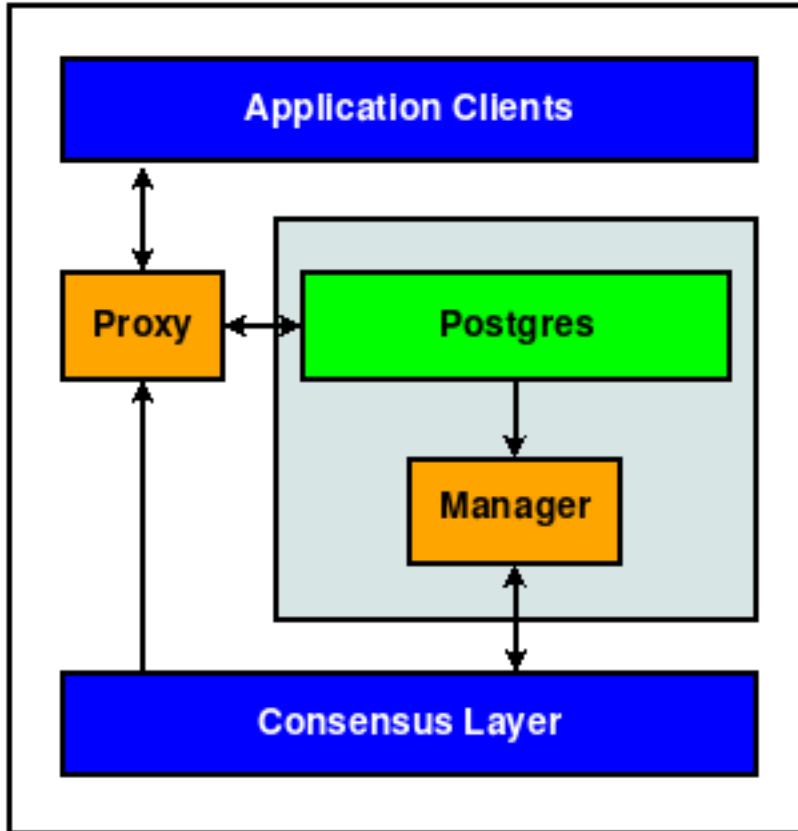


図 1: HARP Unit

コンセンサスレイヤーは、Harp Manager が割り当てられた Postgres ノードについて学習した情報を維持する外部エンティティであり、HARP プロキシはこの情報を有効な Postgres ノードターゲットに変換します。プロキシはコンセンサスレイヤーからノードターゲットを取得するため、このようなインスタンスが複数存在する可能性があります。

コンセンサスレイヤーとして BDR を使用している間、各サーバーノードは代わりにこのバリエーションに似ています。

いずれの場合も、各ユニットは次の要素で構成されます。

- Postgres または EDB インスタンス
- Postgres インスタンスのさまざまな属性を追跡するためのコンセンサスレイヤーリソース
- Postgres ノードの状態をコンセンサスレイヤーに伝える HARP Manager プロセス

\*コンセンサスレイヤーから派生した、適切なリードマスターノードにトラフィックを転送する HARP プロキシサービス

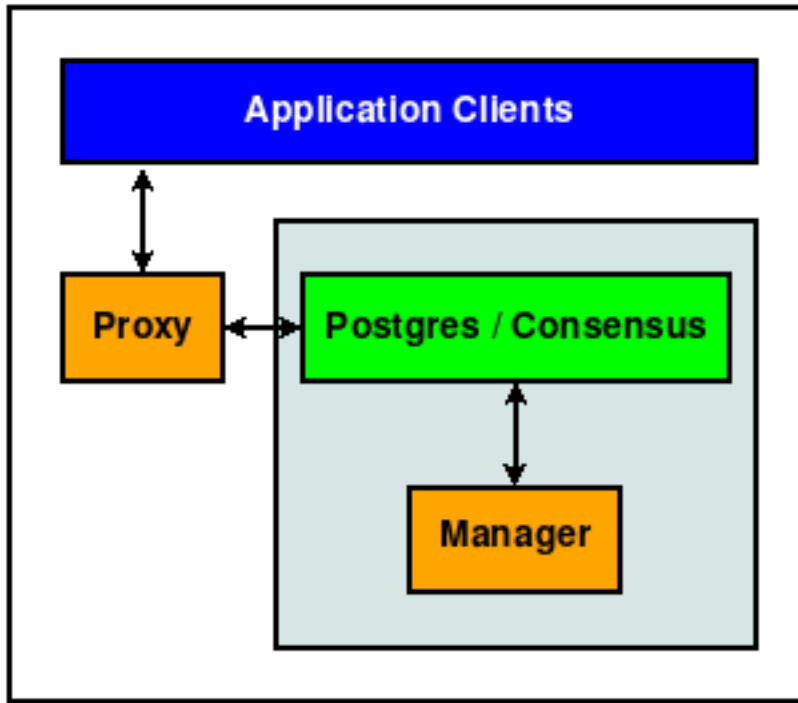


図 2: HARP Unit w/BDR Consensus

すべてのアプリケーションスタックがプロキシコンポーネント専用の追加ノードリソースにアクセスできるわけではないため、アプリケーションサーバーと組み合わせてスタックを簡素化できます。

これは、リードマスター/シャドウマスター構成で編成された単一のデータセンターで2つの BDR ノードを使用する一般的なデザインです。

BDR を HARP コンセンサスレイヤーとして使用する場合、クォーラムマジョリティを保証には、少なくとも3つの完全修飾 BDR ノードが存在する必要があります。(図には、BDR ノード間の接続は示されていません。)

### 仕組み

EDB Postgres 分散クラスターを管理する場合、HARP は定義された場所ごとに最大1つのリーダーノードを維持します。これはリードマスターと呼ばれます。この位置を取得できる他の BDR ノードは、リーダーロールを取得するまでシャドウマスター状態です。

アプリケーションは、プロキシサービスを介してのみ現在のリーダーに連絡できます。コンセンサスレイヤーはリーダーの状態を伝える前にクォーラム契約を必要とするため、プロキシサービスはトラフィックをそのノードに転送します。

高レベルでは、このメカニズムはマルチプルのノードとの同時のアプリケーションの相互作用を防ぎます。

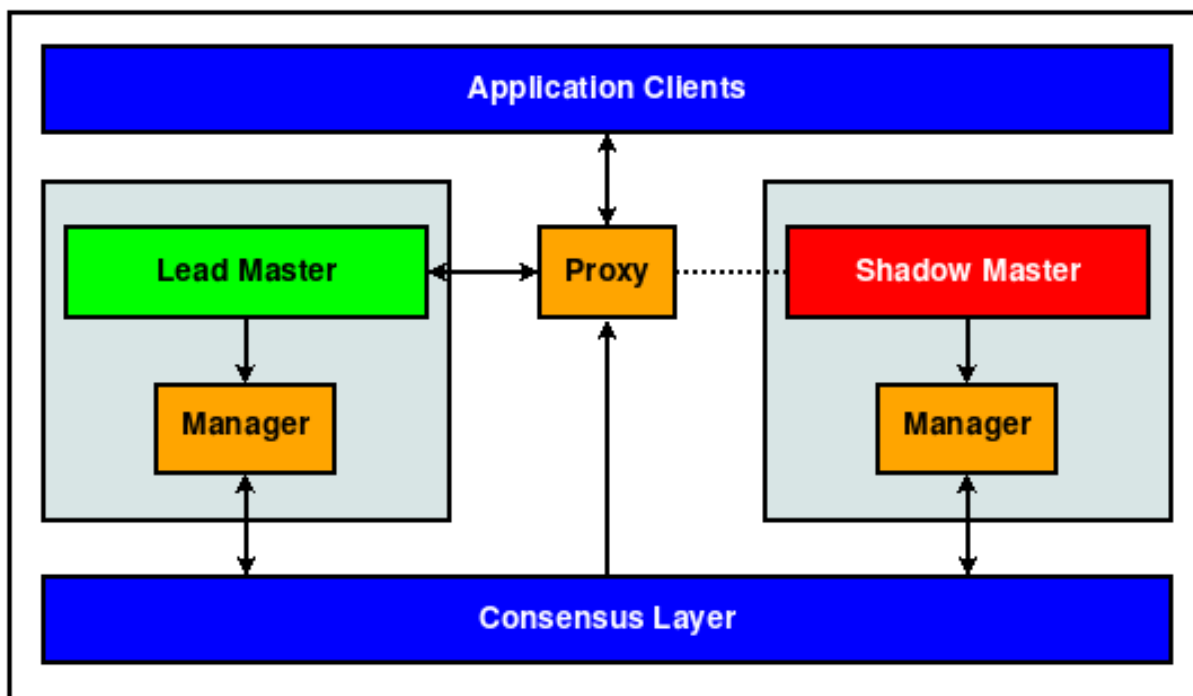


図 3: HARP Cluster

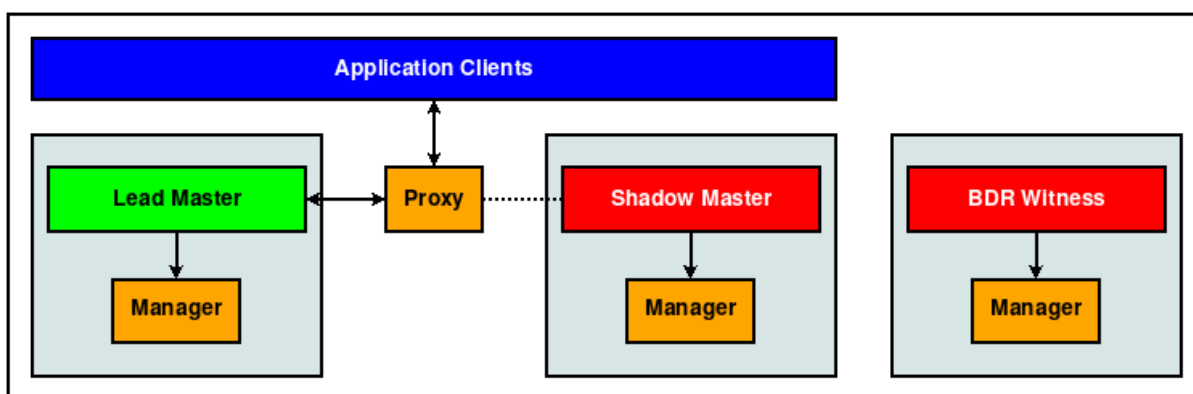


図 4: HARP Cluster w/BDR Consensus



## リーダーの決定

例として、単一のデータセンターに存在する可能性がある、ローカルに細分化された BDR Always-On グループでのリードマスターのロールを検討します。Postgres または Manager リソースが起動され、構成可能な更新間隔の後、次が発生する必要があります。

1. Manager は、割り当てられた Postgres リソースのステータスを確認します。- Postgres が実行されていない場合は、構成可能なタイムアウト後に再試行します。- Postgres が実行されている場合は続行します。
2. Manager は、コンセンサスレイヤーのリーダーリースのステータスを確認します。- リースが要求されていない場合は、リースを取得し、このマネージャーに割り当てられた Postgres インスタンスの ID を割り当てます。このリース期間は構成可能ですが、設定が低すぎると、予期しないリーダーシップの移行が発生する可能性があります。- リースが既に当社によって要求されている場合は、リースの TTL を更新します。- それ以外の場合は何もしません。

もっと多くのことが起こりますが、この単純化されたバージョンは何が起こっているのかを説明しています。リーダーリースは1つのノードでのみ保持でき、他の場所で保持されている場合、HARP マネージャーはあきらめて後で再試行します。

---

**注釈:** 選択されたコンセンサスレイヤーに応じて、リーダーリースのステータスを確認するために繰り返しループするのではなく、HARP は通知をサブスクライブします。この場合、リースの状態が変化するたびに、ポーリングではなくすぐに応答できます。現在、この機能は etcd コンセンサスレイヤーに制限されています。

---

これは、HARP 自分自身が選挙を行ったり、コンセンサスレイヤーに委任されたりするクォーラムを管理しないことを意味します。コンセンサスレイヤーのクォーラムはリースを取得する行為を確認する必要があるため、要求が成功した場合、そのノードはその場所のクラスターをリードします。

## 接続ルーティング

リードマスターのロールが確立されると、接続は HARP プロキシに反映されるのと同様の決定論的な結果で処理されます。HARP プロキシが特定のバックエンドリソースの接続ターゲットを決定する必要がある場合を考えます。

1. HARP プロキシは、設定された場所にある現在のリードマスターのコンセンサスレイヤーを照会します。
2. これが設定されていないか移行中の場合：- Postgres への新しいクライアント接続は禁止されていますが、クライアントは蓄積され、リードマスターが表示されるまで一時停止状態になります。- 既存のクライアント接続は現在のトランザクションを完了することが許可され、新しい接続と同様の保留状態に戻ります。
3. クライアント接続はリードマスターに転送されます。

この場合に示される相互作用は、HARP Manager または Postgres との相互作用を必要としません。コンセンサスレイヤーは、プロキシの観点からのすべての真実のソースです。



## コロケーション

ワークユニットの配置は、組織が次の原則に従う必要があります。

1. manager ユニットと Postgres ユニットが同一ノードに存在すること。
2. コンセンサスレイヤーの内容は、すべての運用作業単位の規範的な役割を指示します。

この取り決めは、クラスターフォーラムの責任をコンセンサスレイヤーに委任しますが、HARP は重要なロールの割り当てとキー/値のストレージにそれを活用します。コンセンサスレイヤーが動作不能または到達不能の場合、ストレージも取得も成功しないため、不正な Postgres ノードが接続を受け入れることを防ぎます。

その結果、コンセンサスレイヤーは一般に、安全性を最大限に高めるために HARP または HARP 管理対象ノードの外部に存在します。リファレンス図はこの分離を示していますが、必須ではありません。

---

**注釈:** クラスター状態を操作および管理するために、BDR には Raft コンセンサスモデルの独自の実装が含まれています。この同じレイヤーを利用して、外部依存関係への依存を減らし、サーバーリソースを保持するように HARP を構成できます。ただし、このアプローチの特定の欠点については、[コンセンサスレイヤー](#) で説明されています。

---

## 推奨されるアーキテクチャと使用

HARP は、主に2つ以上のデータセンターにあり、少なくとも5つの BDR ノードで構成される BDR Always-On アーキテクチャを表すように設計されました。この構成では、ロジカルスタンバイノードはカウントされません。

次の図は、現在および標準の表現を示しています。

この図では、HARP Manager は BDR ノード 1~4 に存在します。クラスターの初期状態は、BDR ノード 1 が DC A のリードマスターであり、BDR ノード 3 が DC B のリードマスターです。

この構成により、DC A の HARP プロキシリソースが BDR ノード 1 に接続され、DC B の HARP プロキシリソースが BDR ノード 3 に接続されます。

---

**注釈:** この図は DC ごとに1つの HARP プロキシのみを示していますが、これは例にすぎず、単一障害点とは見なされません。HARP プロキシノードはいくつでも存在でき、それらはすべてアプリケーショントラフィックを同じノードに転送します。

---

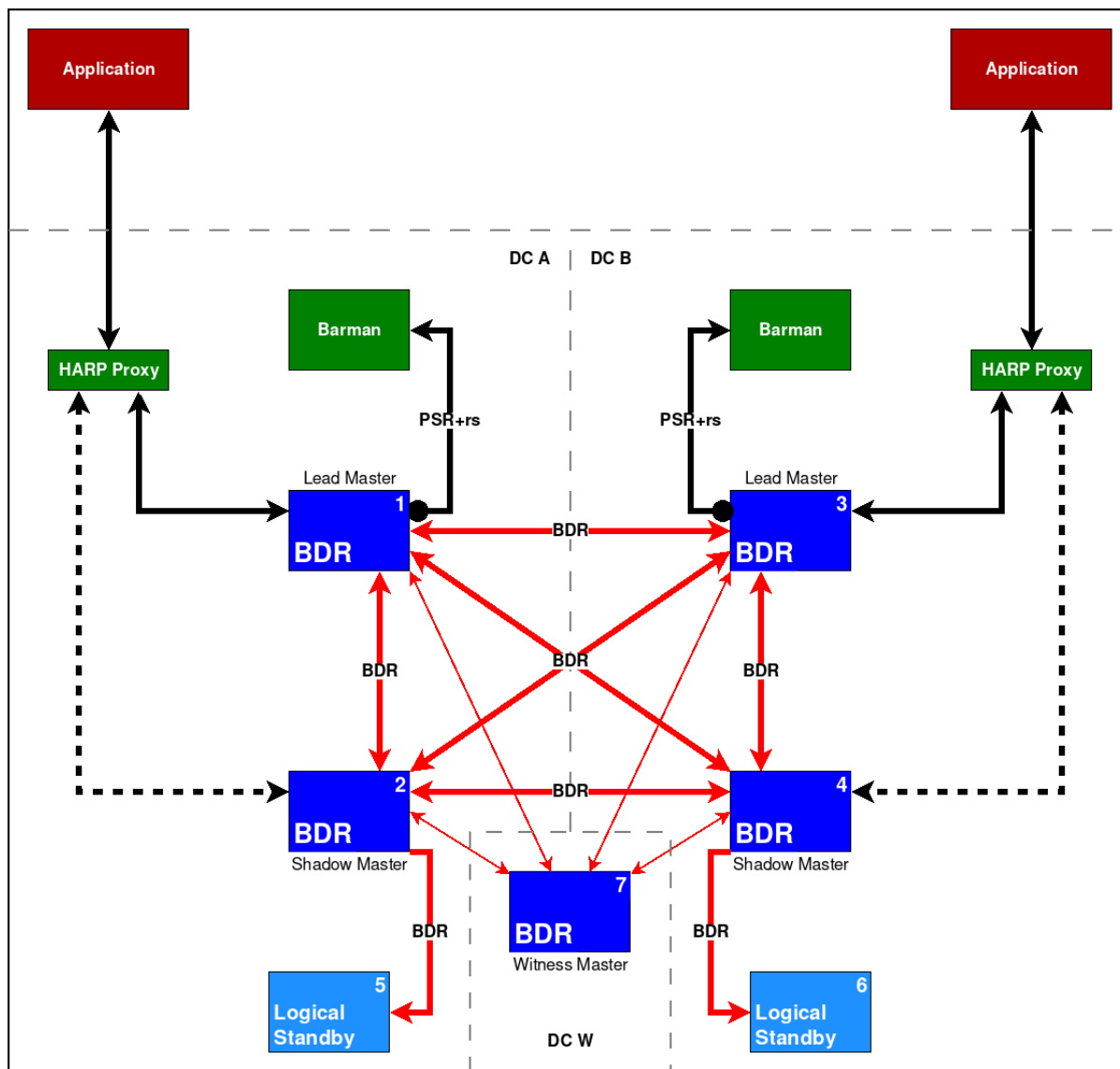


図 5: BDR Always-On Reference Architecture

## ロケーション構成

複数の BDR ノードがロケーションでリードマスターロックを取得できるようにするには、`config.yml` 構成ファイルでロケーションを定義する必要があります。

図に示されている BDR Always-On リファレンスアーキテクチャを再現するには、BDR ノード 1 および 2 の `config.yml` 構成に次の行を含めます。

```
location: dca
```

BDR ノード 3 および 4 の場合、次を追加します。

```
location: dcb
```

これは、これらのそれぞれのデータセンターで指定された HARP プロキシノードにも適用されます。

## BDR 3.7 との互換性

BDR 3.7 以降では、BDR ノードにロケーションを割り当てることにより、より直接的なロケーション定義を提供します。これを行うには、BDR ノードに接続しているときに次の SQL API ファンクションを呼び出します。したがって、BDR ノード 1 および 2 の場合、次のようにします。

```
SELECT bdr.set_node_location(dca);
```

BDR ノード 3 および 4 の場合：

```
SELECT bdr.set_node_location(dcb);
```

## Installation

HARP の標準インストールには、2 つのシステムサービスが含まれます。

- 管理対象ノードの HARP Manager (`harp-manager`)

\*他の場所での HARP プロキシ (`harp-proxy`)

これらのサービスをインストールして構成し、適切なクォーラムベースの接続ルーティングのために Postgres を管理するには 2 つの方法があります。

## ソフトウェアバージョン

HARP は外部ソフトウェアに依存しています。これらは、ここにリストされている最小バージョンに適合する必要があります。

| Software  | Min version |
|-----------|-------------|
| etcd      | 3.4         |
| PgBouncer | 1.14        |

## TPAExec

HARP をインストールして構成する最も簡単な方法は、クラスターの展開と管理に EDB TPAexec ユーティリティを使用することです。本ソフトウェアの詳細は、

[TPAexec product page](#) を参照してください。

**注釈:** TPAExec は現在、特に EDB Postgres 分散クラスター展開専用の EULA を介してのみ入手できます。TPAExec URL にアクセスできない場合は、営業またはアカウント担当者にお問い合わせください。

TPA config.yml ファイルに次の属性が含まれていることを確認して、クラスタールーティングが HARP を介して管理されていることを認識するように TPAexec を構成します。

```
cluster_vars:
  failover_manager: harp
```

**注釈:** 21.1 より前の TPAexec のバージョンでは、少し異なるアプローチが必要です。

```
cluster_vars:
  enable_harp: true
```

この後、クラスターを変更するための tpaexec コマンドを呼び出して、HARP をインストールします。

```
tpaexec provision ${CLUSTER_DIR}
tpaexec deploy ${CLUSTER_DIR}
```

クラスター固有の考慮事項を除き、他の変更は必要ありません。

## パッケージのインストール

現在、CentOS / RHEL パッケージは EDB パッケージインフラストラクチャによって提供されています。詳細は、[HARP product page](#) を参照してください。

### etcd パッケージ

現在、多くの一般的な Linux ディストリビューションの etcd パッケージは、標準のパブリックリポジトリでは利用できません。したがって、EDB は RHEL および CentOS バージョン 7 および 8、Debian、および Ubuntu LTS などのバリエーション用に etcd をパッケージしました。これらのライブラリを使用するには、HARP パッケージリポジトリにアクセスする必要があります。

### コンセンサスレイヤー

HARP の動作には分散コンセンサスレイヤーが必要です。現在、これは bdr または etcd のいずれかである必要があります。3 つ未満の BDR ノードを使用する場合、etcd に依存する必要がある場合があります。それ以外の場合、BDR サービスが停止すると、コンセンサスレイヤーが単一のノードに削減されるため、ノードのコンセンサスが妨げられ、Postgres ルーティングが無効になります。

### etcd

コンセンサスレイヤーとして etcd を使用している場合、etcd は Postgres ノードに直接インストールするか、アクセスできる別の場所にインストールする必要があります。

etcd をコンセンサスレイヤーとして設定するには、このコードを HARP config.yml 構成ファイルに含めます。

```
dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379
```

TPAExec を使用する場合、設定済みの etcd エンドポイントはすべてここに自動的に入力されます。

### BDR

bdr ネイティブコンセンサスレイヤーは、BDR 3.6.21 および 3.7.3 から利用できます。このコンセンサスレイヤーモデルでは、EDB Postgres 分散クラスターのルーティングを管理する際に補助ソフトウェアは必要ありません。

クラスターでクォーラムを使用できるようにするには、ノードのメンテナンスまたは停止中に BDR のコンセンサスレイヤーが応答し続けるように、常に 3 つ以上のノードを使用します。

BDR をコンセンサスレイヤーとして設定するには、これを config.yml 構成ファイルに含めます。

```
dcg:
  driver: bdr
  endpoints:
    - host=host1 dbname=bdrdb user=harp_user
    - host=host2 dbname=bdrdb user=harp_user
    - host=host3 dbname=bdrdb user=harp_user
```

BDR コンセンサスレイヤーのエンドポイントは、標準の Postgres DSN 接続フォーマットに従います。

## Configuring HARP for cluster management

HARP 構成ファイルは、読みやすいように簡素化された標準の YAML スタイルのフォーマットに従います。このファイルはデフォルトで/etc/harp ディレクトリにあり、`config.yml` という名前です

`-f/--config` 引数を使用して、すべての HARP 実行可能ファイルに構成ファイルの場所を明示的に提供できます。

### 標準構成

HARP は基本的に 3 つのコンポーネントとして動作します。

- HARP マネージャー
- HARP プロキシ
- harpctl

これらはそれぞれ、同じ標準の `config.yml` 構成フォーマットを使用します。これには、常に次のセクションが含まれます。

- `cluster.name` —すべてのオペレーションの対象となるクラスターの名前。
- `dcg` —すべてのエンドポイントの DCS ドライバーと接続構成。

これは、次のような HARP 操作には常に標準のプリアンブルが含まれることを意味します。

```
cluster:
  name: mycluster

dcg:
  ...
```

他のセクションは、オプションまたは名前付けた HARP コンポーネントに固有です。

## クラスター名

`cluster` 見出しの下に `name` エントリは、HARP とのすべての相互作用に必要です。各 HARP クラスターには、特定のクラスターの DCS 内の明確化とデータのラベル付けの両方の名前があります。

HARP マネージャーは、HARP プロキシと `harpcctl` が使用するクラスターに関する情報をここに書き込みます。HARP プロキシサービスは、このクラスター内のノードにトラフィックを誘導します。`harpcctl` 管理ツールはこのクラスターと対話します。

## DCS 設定

コンセンサスレイヤーの構成は、HARP 機能のキーとなります。DCS がなければ、HARP にはクラスターメタデータを保存する場所がなく、リーダーシップ選挙を行うこともできません。したがって、構成のこの部分は必須であり、特定の要素はオプションです。

次の複数の補助エントリを使用して、`dc`s という名前のセクションの下にすべての要素を指定します。

- `driver` : 使用するコンセンサスレイヤーの必須タイプ。現在は `etcd` または `bdr` です。コンセンサスレイヤーとしての `bdr` のサポートは実験的です。コンセンサスレイヤーとして `bdr` を使用すると、コンセンサスストレージ用の追加ソフトウェアが削減されますが、データベースのメンテナンス中にクォーラムを維持するために少なくとも 3 つの完全な BDR メンバーノードが必要です。
- `endpoints` : DCS に接続するために必要な接続文字列のリスト。可能であれば、ここに DCS のすべてのノードをリストします。これにより、DCS の大部分が引き続き動作し、ネットワークが到達できる限り、HARP が機能し続けます。

`etcd` をコンセンサスレイヤーとして使用する場合は次のとおりです。edb\_notranlate\_1 実験的な `bdr` コンセンサスレイヤーを使用する場合は次のとおりです。edb\_notranlate\_2 現在、`bdr` コンセンサスレイヤーは最初のエンドポイントがローカルの `postgres` インスタンスを指す必要があります。

- `request_timeout` : 要求が失敗したと見なされる時間（ミリ秒）。HARP は DCS に要求を行い、この時間内に応答を受信しない場合、操作は失敗したと見なされます。これにより、要求の性質に応じて、問題がエラーとしてログに記録されるか、再試行されます。デフォルト：250。

次の DCS SSL 設定は、edb\_notranlate\_3 yml dcs:driver:etcd endpoints: - host1:2379 - host2:2379 - host3:2379 edb\_notranlate\_4 yml クラスター: 名前:

dc: ドライバー: etcd エンドポイント: - host1:2379 - host2:2379 - host3:2379

manager: 名前: node1 log\_level: INFO edb\_notranlate\_5 yml cluster: 名前: mycluster

dc: ドライバー: etcd エンドポイント: - host1:2379 - host2:2379 - host3:2379

proxy: 名前: proxy1 場所: dc1 pgbouncer\_bin\_dir: /usr/sbin edb\_notranlate\_6 bash harpcctl set node node1 lease\_refresh\_interval=100 edb\_notranlate\_7 bash harpcctl set proxy global max\_client\_conn=1000 ``

## Cluster bootstrapping

HARP を使用するには、最小限のメタデータが DCS に存在する必要があります。クラスターを「ブートストラップ」するプロセスとは、本質的に、ノード、場所、およびその他のランタイム構成を一度にまたはリソースごとに初期化することを意味します。

このプロセスは、`harpctl apply` コマンドによって制御されます。詳細については、[harpctl command-line tool](#) を参照してください。

DCS を設定し、ブートストラップの前に機能することを確認します。

---

**重要:** これらの例の一部またはすべてを単一の YAML ドキュメントに組み合わせて、一度に適用できます。

---

### クラスター全体のブートストラップ

一部の設定はクラスター全体に適用され、ブートストラップ中に指定できます。現在、これは `event_sync_interval` ランタイム指示子にのみ適用されますが、他の人は後で追加される可能性があります。

フォーマットは次のとおりです。

```
cluster:
  name: mycluster
  event_sync_interval: 100
```

そのファイルの名前付けが `cluster.yml` であると仮定すると、次のように適用します。

```
harpctl apply cluster.yml
```

クラスター名が DCS でまだ定義されていない場合、これもその値を初期化します。

---

**重要:** ここで指定されたクラスター名パラメーターは、常に `config.yml` で指定されたクラスター名をオーバーライドします。ブートストラップファイルは、クラスターまたはその大規模な構成の一部をブートストラップするために必要なすべての要素を提供することが前提です。 `config.yml` ファイルは、主に HARP マネージャー、HARP プロキシ、または `harpctl` の実行を制御することを目的としています。

---

### ロケーションブートストラップ

すべての HARP ノードは、最大 1 つの場所に関連付けられます。この場所は、単一のデータセンター、複数の基になるサーバーで構成されるグループ化されたリージョン、Amazon アベイラビリティゾーンなどです。これは、HARP がノードをグループ化して、その場所のノードを 1 つだけがリードマスターとして表すことができるようにする論理構造です。

したがって、1 つ以上の場所を初期化する必要があります。このフォーマットは次のとおりです。



```
cluster:
  name: mycluster

locations:
  - location: dc1
  - location: dc2
```

そのファイルの名前付けが `locations.yml` であると仮定すると、次のように適用します。

```
harpctl apply locations.yml
```

クラスターの操作を実行するときは、名前をプリアンブルとして含めて、変更が適切な場所に送られるようにします。

ロケーションがブートストラップされると、簡単な検査で表示されます。

```
> harpctl get locations

Cluster   Location Leader Previous Leader Target Leader Lease Renewals
-----
mycluster dc1                                <nil>
mycluster dc2                                <nil>
```

どちらの場所も HARP によって認識され、ノードとプロキシの割り当てに使用できます。

## ノードのブートストラップ

HARP ノードは名前付きクラスターに存在し、名前が指定されている必要があります。これを超えて、他のすべての設定は動的であり、HARP との相互作用に影響を与える可能性があるため、DCS に保持されます。このため、

### 構成ファイルの指定

で説明されている 1 つ以上のランタイム指示子を使用して各ノードをブートストラップします。

ノードのブートストラップ中に、いくつかの必須フィールドがあります。

- `name`
- `location`
- `dsn`
- `pg_data_dir`

その他はすべてオプションであり、クラスターに依存する場合があります。一度に複数のノードをブートストラップできるため、一般的にこのフォーマットは次の構造に適合します。

```
cluster:
  name: mycluster
```

(次のページに続く)

(前のページからの続き)

```
nodes:
- name: node1
  location: dc1
  dsn: host=node1 dbname=bdrdb user=postgres
  pg_data_dir: /db/pgdata
  leader_lease_duration: 10
  priority: 500
```

そのファイルの名前付けが `node1.yml` であると仮定すると、次のように適用します。

```
harpctl apply node1.yml
```

ノードがブートストラップされると、簡単な検査で表示されます。

```
> harpctl get nodes
```

| Cluster         | Name | Location | Ready | Fenced | Allow | Routing | Routing | Status | Role        | Type |
|-----------------|------|----------|-------|--------|-------|---------|---------|--------|-------------|------|
| ↪ Lock Duration |      |          |       |        |       |         |         |        |             |      |
| -----           |      |          |       |        |       |         |         |        |             |      |
| ↪-----          |      |          |       |        |       |         |         |        |             |      |
| mycluster       | bdr1 | dc1      | true  | false  | true  |         | ok      |        | primary bdr | 30   |

## プロキシブートストラップ

ロケーションやノードとは異なり、プロキシは、ロケーション内のすべてのプロキシに適用される構成テンプレートを提供することもできます。これらは、`global` 指定の下で DCS に保存されます。各プロキシには名前がインスタンスとして存在する必要がありますが、一部の設定で特定のオーバーライドが必要な場合を除き、これ以上のカスタマイズは必要ありません。

これは、クラスターのデフォルトの構成設定が同じであるプロキシが複数存在する可能性があり、すべてのプロキシに対してこれらの値を繰り返す必要がないためです。

さらに、プロキシテンプレートをブートストラップするときは、接続の割り当て用に少なくとも 1 つのデータベースを定義します。これらの注意事項を念頭に置いて、このフォーマットは次のとおりです。

```
cluster:
  name: mycluster

proxies:
  monitor_interval: 5
  default_pool_size: 20
  max_client_conn: 1000
  database_name: bdrdb
  instances:
```

(次のページに続く)

(前のページからの続き)

```
- name: proxy1
- name: proxy2
  default_pool_size: 50
```

これにより、2つのプロキシ（proxy1 と proxy2 ）の HARP が構成されます。proxy2 のみにカスタム default\_pool\_size がありますが、それ以外の場合はグローバル設定を使用します。

そのファイルの名前付けが proxy.yml であると仮定すると、次のように適用します。

```
harpctl apply proxy.yml
```

プロキシテンプレートがブートストラップされると、簡単な検査で表示されます。

```
> harpctl get proxies
```

| Cluster   | Name   | Pool Mode | Auth Type | Max Client Conn | Default Pool Size |
|-----------|--------|-----------|-----------|-----------------|-------------------|
| mycluster | global | session   | md5       | 1000            | 20                |
| mycluster | proxy1 | session   | md5       | 1000            | 20                |
| mycluster | proxy2 | session   | md5       | 1000            | 50                |

## HARP Manager

HARP Manager は、ローカルの PostgreSQL / BDR ノードと対話し、その状態に関する情報をコンセンサスレイヤーに保存するデーモンです。Manager は、それぞれのロケーションのリーダーステータスを現在保持しているノードを特定し、構成（ラグ、CAMO ラグなど）の制約を適用して、資格のないノードがリーダーの検討対象とならないようにします。

クラスター内のすべての Postgres ノードには、HARP Manager が関連付けられている必要があります。他のノードは存在できますが、リードまたはシャドウマスターの役割、または HARP マネージャーを必要とするその他の機能として参加することはできません。

**重要:** HARP Manager は、データベースの起動および停止に使用されることを想定しています。HARP マネージャーを停止すると、データベースが停止します。HARP Manager を起動すると、データベースがまだ起動されていない場合は起動します。別の方法を使用してデータベースを停止すると、HARP マネージャーはデータベースを再起動します。

## 仕組み

起動時に、HARP Manager は `pg_ctl` を使用して Postgres をまだ実行していない場合は起動します。この後、`node.lease_refresh_interval` 設定で定義されたサーバーを定期的にチェックします。HARP Manager は、次を含む Postgres に関するさまざまなデータを収集します。

\*ノードの現在の LSN。

- Postgres が実行されており、接続を受け入れている場合。この特定のデータポイントは、定期的に更新する必要があるリースと見なされます。有効期限が切れると、HARP プロキシは既存のルーティングからノードを削除します。

\*すべてのアップストリーム BDR ピアノードの現在の適用 LSN 位置。

- CAMO が有効になっている場合： - CAMO パートナーの名前 - ピア CAMO の状態 (`is_ready`) - CAMO キューが受信して適用した LSN 位置

\*ノードが BDR か通常の Postgres かなどのノードタイプ。

- 読み取り/書き込み、物理ストリーミングレプリカ、ロジカルスタンバイなど、ノードの現在のロール。
- BDR ノードの状態。限定された場合を除き `ACTIVE`。
- その他のメタデータ収集の BDR ノード ID。

※その他のトラッキング値です。

---

**重要:** HARP で BDR ノードに名前を付ける場合、BDR ノード名は `node.name` 構成属性で表されるノード名と一致する必要があります。これはブートストラッププロセスで発生します。

---

ここで収集されたデータは、他の HARP Manager プロセスで完全に利用でき、遅延、パートナーの準備状況、およびスイッチオーバーとフェイルオーバーの動作を指示するその他の基準を評価するために使用されます。

ノードのメタデータを更新した後、HARP マネージャーは、リードマスターリースが既にローカルノードに保持されている場合は更新するか、リースが期限切れになっている場合はリースを取得しようとします。すべてのノードの現在の状態は他のすべてのノードに知られているため、以前のリードマスターであったノードが存在する場合、自動的に優先順位が付けられます。そうでない場合、他のすべてのノードは LSN ラグ、ノードの優先度、およびその他の基準で自分自身をリストし、最も適格なノードがリードマスターリースを押収します。

この手順は、ノードが存在する定義されたすべての場所で発生します。したがって、ロケーション DC1 と DC2 には、それぞれにリードマスターノードがあり、両方に個別のリースと選択プロセスがあります。

HARP Manager は、これらの Postgres ステータスチェック、リース更新、および選択を繰り返して、クラスターが HARP プロキシからの接続に対して常にリードマスターターゲットを持つようにします。

## 構成

HARP マネージャーは、`dc`s、`cluster`、および `manager` 構成スタanzasを想定しています。以下は機能的な例です。

```
cluster:
  name: mycluster

dc:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

manager:
  name: node1
  postgres_bin_dir: /usr/lib/postgresql/13/bin
```

実行中のインスタンスに `SIGHUP` を発行するか、サービスレベルのリロードを呼び出すことにより、構成ファイル（デフォルト：`/etc/harp/config.yml`）に変更を適用できます。

詳細については、[構成ファイルの指定](#) を参照してください。

## 使用法

これは、HARP Manager の基本的な使用法です。

```
Usage of ./harp-manager:
  -f string
      Optional path to config file (shorthand)
  --config string
      Optional path to config file
```

`harp-manager` をフォークされたデーモンとして起動するための引数はありません。このソフトウェアは、`systemd` またはコンテナ内でトップレベルプロセスとして起動されるように設計されています。これは、`journald` または接続されたコンテナターミナルを介してキャプチャおよびアクセスするために、出力が `STDOUT` および `STDERR` に送られることも意味します。

## Postgres の HARP Manager 制御の無効化と再有効化

Postgres の HARP Manager 制御を一時的に一時停止できます。これにより、デーモンの実行は継続されますが、クラスターの既存の動作に影響を与える可能性のある操作は実行されません。管理を再度有効にすると、操作が再開されます。

ノード管理を一時的に無効にする例は次のとおりです。

```
harpctl unmanage node node1
```

詳細については、[harpctl command-line tool](#) を参照してください。

HARP Manager によるノード管理はデフォルトで有効になっています。

## HARP Proxy

HARP プロキシは、クライアントアプリケーションと Postgres 間の抽象化レイヤーとして機能するデーモンです。コンセンサスレイヤーとインターフェイスして、現在のリードマスターノードの ID を取得し、トラフィックをその場所に転送します。計画的なスイッチオーバーまたは計画外のフェイルオーバー中に、DCS の指示に従って新しいリードマスターノードにリダイレクトします。

HARP プロキシは `pgbouncer` または `builtin` から選択できます。プロキシタイプを指定しない場合、デフォルトは `builtin` です。 `pgbouncer` を使用する場合、HARP プロキシは DCS と PgBouncer 間のインターフェイスレイヤーです。そのため、PgBouncer は前提条件であり、HARP プロキシがそのアクティビティを完全に管理するにはインストールする必要があります。

組み込みプロキシには、追加のソフトウェアは必要ありません。ビルトインを使用する場合、HARP プロキシはレベル 4 パススループロキシとして機能します。

### ビルトインプロキシ：仕組み

起動時に、HARP プロキシは、プロキシインスタンスごとにブートストラップファイルで指定されたリスニングアドレスとリスニングポートで着信接続を監視します。すべてのアプリケーションクライアントトラフィックは、組み込みプロキシを介して、このプロキシが動作している場所の現在のリードマスターノードに渡されます。

リードマスターリースが設定されていない場合、HARP プロキシは、新しいリードマスターが確立されるまですべての接続トラフィックを切断します。これは、`harpctl promote` を使用して新しいリードマスターへの計画的な移行を呼び出す場合にも適用されます。切断は即時です。

## 構成

プロキシタイプを `builtin` に設定して、ビルトインプロキシを選択します。組み込みプロキシに適用される他の唯一のオプションは、許可されるクライアント接続の最大数を指定します。`max_client_conn` がシステムが処理できるものよりも高い場合、プロキシがオンになっているシステムの能力内の設定に下げられます。

## PgBouncer : 仕組み

---

**注釈:** Harp Proxy が提供するものよりも多くの構成可能性が必要な場合は、ビルトインプロキシを使用して `pgbouncer` がそれをポイントするように設定することをお勧めします。

---

HARP プロキシは、起動時に PgBouncer がまだ実行されていない場合に起動し、クライアント接続を一時停止したままにします。その後、DCS に連絡してリードマスターの ID を判断し、これをデータベース接続のターゲットとして使用するように PgBouncer を構成し、接続アクティビティを再開します。すべてのアプリケーションクライアントトラフィックは、PgBouncer を介して、このプロキシが動作している場所の現在のリードマスターノードに渡されます。

PgBouncer の実行中、HARP プロキシは DCS の `monitor_interval` 構成設定に基づいてステータスを確認し、監視のために DCS に保存します。この構成により、`harpctl` を使用した問い合わせで、構成されているすべてのプロキシまたは1つのプロキシのステータスを取得できます。

リードマスターリースが設定されていない場合、HARP プロキシは新しいリードマスターが確立されるまですべての接続トラフィックを一時停止します。これは、`harpctl promote` を使用して新しいリードマスターへの計画的な移行を呼び出す場合にも適用されます。これには PgBouncer PAUSE コマンドを使用するため、既存のセッションは保留中のトランザクションが停止する前に完了することができます。

## PgBouncer 構成ファイル

HARP プロキシが接続管理とリダイレクトに PgBouncer を使用する場合、`pgbouncer.ini` ファイルが存在する必要があります。HARP マネージャーは、Proxy directives で定義されているさまざまなランタイム指示子に基づいてこのファイルをビルドします。

このファイルは、HARP プロキシで使用される `config.yml` と同じフォルダーにあります。HARP プロキシによって起動される PgBouncer プロセスはこの構成ファイルを使用し、デバッグまたは情報の目的で使用できます。この自動生成された `pgbouncer.ini` ファイルへの変更は、HARP プロキシが再起動するたびに失われるため、代わりに `harpctl set proxy` を使用してください。`harpctl set proxy` を呼び出しても、プロキシが再起動するまで `pgbouncer.ini` ファイルは更新されません。

## HARP プロキシノード管理の無効化と再有効化

PgBouncer の HARP プロキシ制御を一時的に一時停止できます。これにより、デーモンの実行は継続されますが、クラスターの既存の動作に影響を与える可能性のある操作は実行されません。管理を再度有効にすると、操作が再開されます。

特定のプロキシの管理を一時的に無効にする例は次のとおりです。

```
harpctl unmanage proxy proxy1
```

詳細については、[harpctl command-line tool](#) を参照してください。

プロキシノード管理はデフォルトで有効になっています。

## パススルーユーザー認証

pgbouncer では、`auth_user` および `auth_query` ランタイム指示子を使用するように HARP プロキシを構成することを強くお勧めします。これらが設定されていない場合、PgBouncer `userlist.txt` ファイルには、PgBouncer が Postgres に代わって認証する必要があるすべてのユーザーのユーザー名とパスワードのハッシュの組み合わせを含める必要があります。

pgbouncer ユーザーは使用しないでください。これは、基になる PgBouncer サービスを操作するための管理レベルのユーザーとして HARP プロキシによって使用されます。

## 構成

HARP プロキシは、`dc`s、`cluster`、および `proxy` 構成スタンプを想定しています。以下は機能的な例です。

```
cluster:
  name: mycluster

dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

proxy:
  name: proxy1
```

`` 各プロキシは DCS に接続して、接続をリッスンするホストとポートを取得します。

### 使用法

これは、 HARP プロキシの基本的な使用法です。

(次のページに続く)



(前のページからの続き)

```

```bash
Usage of ./harp-proxy:
  -f string
      Optional path to config file (shorthand)
  --config string
      Optional path to config file

```

`harp-proxy` をフォークされたデーモンとして起動するための引数はありません。このソフトウェアは、`systemd` またはコンテナ内でトップレベルプロセスとして起動されるように設計されています。これは、`journald` または接続されたコンテナターミナルを介してキャプチャおよびアクセスするために、出力が `STDOUT` および `STDERR` に送られることも意味します。

### harpctl command-line tool

`harpctl` は、コンセンサスレイヤーのコンテンツを直接操作して、目的のクラスタージオメトリに合わせるためのコマンドラインツールです。これを使用して、ノードのステータスを調べたり、ノードをリードマスターに「昇格」させたり、クラスター管理を無効/有効にしたり、クラスター設定をブートストラップしたりすることができます。

#### 概要

```

$ harpctl --help

Usage:
  harpctl [command]

Available Commands:
  apply      Command to set up a cluster
  completion generate the autocompletion script for the specified shell
  fence      Fence specified node
  get        Command to get resources
  help       Help about any command
  manage     Manage Cluster
  promote    Promotes the sepcified node to be primary,
  set        Command to set resources.
  unfence    unfence specified node
  unmanage   Unmanage Cluster
  version    Command to get version information

Flags:
  -c, --cluster string      name of cluster.
  -f, --config-file string  config file (default is /etc/harp/config.yml)

```

(次のページに続く)

(前のページからの続き)

```

--dcs stringArray      Address of dcs endpoints. ie: 127.0.0.1:2379
-h, --help             help for harpctl
-o, --output string     json, yaml.
-t, --toggle           Help message for toggle

```

Use `"harpctl [command] --help"` for more information about a command.

この基本的な概要に加えて、使用可能な各コマンドには、独自のシリーズの許可されたサブコマンドとフラグがあります。

## 構成

harpctl は、動作するためにコンセンサスレイヤーと対話する必要があります。これは、正常に実行するには、`config.yml` で特定の最小量の設定を定義する必要があることを意味します。これには以下が含まれます。

- `dcs.driver`
- `dcs.endpoints`
- `cluster.name`

etcd を使用した例：

```

cluster:
  name: mycluster
dcs:
  driver: etcd
  endpoints:
    - host1:2379
    - host2:2379
    - host3:2379

```

詳細は [構成ファイルの指定](#) を参照してください。

## 実行

次のように harpctl を実行します。

```
harpctl command [flags]
```

各コマンドには、独自のシリーズのサブコマンドとフラグがあります。次のコマンドを実行すると、これらのヘルプを利用できます。

```
harpctl <command> --help
```

## harpctl apply

**apply** コマンドを使用して、目的のクラスターのさまざまな属性を定義するファイルを使用して HARP クラスターを「ブートストラップ」する必要があります。

次のような **apply** コマンドを実行します。

```
harpctl apply <filename>
```

これにより、基本的に、初期クラスターメタデータ、デフォルトまたはカスタムの管理設定などがすべて作成されます。これは、DCS がクラスターを管理するための最終的なソースとして使用され、これらの設定を動的に変更できるためです。

これは、クラスター全体をブートストラップするために 1 回、オブジェクトのタイプごとに 1 回、または簡単にするためにノードごとに実行できます。

これは、単一ノードのブートストラップファイルの例です。

```
cluster:
  name: mycluster
nodes:
  - name: firstnode
    dsn: host=host1 dbname=bdrdb user=harp_user
    location: dc1
    priority: 100
    db_data_dir: /db/pgdata
```

ここで見られるように、複数の環境が同じ環境で動作している場合、すべての変更が正しい HARP クラスターをターゲットと保証に、常にクラスター名前のプレアンプルを含めることをお勧めします。

**apply** がエラーなしで完了すると、ノードはクラスターの残りの部分と統合されます。

**注釈:** 定義されたすべてのセクションが同時に適用されるため、このコマンドを使用してクラスター全体を一度にブートストラップすることもできます。ただし、初期定義中にクラスターの各部分を検証するのが難しくなるため、テスト以外でのこの使用はお勧めしません。

## harpctl fence

ローカルまたは指定されたノードをフェンス済みとしてマークします。このステータスのノードは、基本的にクラスターから完全に除外されます。HARP プロキシはトラフィックを送信せず、その代表的な HARP マネージャーはリードマスターリースを要求せず、さらなる手順も実行されます。実行されている場合、HARP Manager はノードで Postgres も停止します。

次のような **fence** コマンドを実行します。

```
harpctl fence (<node-name>)
```

node-name はオプションです。省略された場合、`harpctl` はローカルに構成されたノードの名前を使用します。

### `harpctl get`

クラスターのさまざまな要素のコンセンサスレイヤーに保存されている情報を取得します。これには、ノード、場所、クラスターなどが含まれます。完全なリストには次のものが含まれます。

- `cluster` — クラスターの状態を返します。
- `leader` — 現在または指定されたロケーションリーダーを返します。
- `location` — 現在または指定された場所情報を返します。
- `locations` — すべてのロケーションのリストを返します。
- `node` — 指定された Postgres ノードを返します。
- `nodes` — すべての Postgres ノードのリストを返します。
- `proxy` — 現在のプロキシ情報または指定されたプロキシ情報を返します。
- `proxies` — すべてのプロキシノードのリストを返します。

### `harpctl get cluster`

現在のクラスターのコンセンサスレイヤーに保存されている情報を取得します。

```
> harpctl get cluster
```

```
Name      Enabled
-  ---
mycluster true
```

### `harpctl get leader`

指定された場所の DCS に保存されている現在のリードマスターのノード情報を取得します。ロケーションが渡されない場合、`harpctl` は、実行された現在のノードのロケーションに基づいてそれを導き出そうとします。

例：

```
> harpctl get leader dc1
```

```
Cluster  Name  Ready Role    Type Location Fenced Lock Duration
-  -
mycluster mynode true  primary bdr  dc1      false  30
```

**harpctl get location**

指定された場所の場所情報を取得します。ロケーションが渡されない場合、harpctl は、実行された現在のノードのロケーションに基づいてそれを導き出そうとします。

例：

```
> harpctl get location dc1
```

| Cluster   | Location | Leader | Previous Leader | Target Leader | Lease | Renewals |
|-----------|----------|--------|-----------------|---------------|-------|----------|
| mycluster | dc1      | mynode | mynode          |               |       | <nil>    |

**harpctl get locations**

DCS に現在存在するすべてのロケーションの情報を取得します。

例：

```
> harpctl get locations
```

| Cluster   | Location | Leader   | Previous Leader | Target Leader | Lease | Renewals |
|-----------|----------|----------|-----------------|---------------|-------|----------|
| mycluster | dc1      | mynode   | mynode          |               |       | <nil>    |
| mycluster | dc2      | thatnode | thatnode        |               |       | <nil>    |

**harpctl get node**

指定されたノードの DCS に格納されているノード情報を取得します。

例：

```
> harpctl get node mynode
```

| Cluster   | Name   | Location | Ready | Fenced | Allow | Routing | Routing | Status | Role    | Type |
|-----------|--------|----------|-------|--------|-------|---------|---------|--------|---------|------|
| mycluster | mynode | dc1      | true  | false  | true  |         | ok      |        | primary | bdr  |

**harpctl get nodes**

クラスター内のすべてのノードの DCS に格納されているノード情報を取得します。

例：

```
> harpctl get nodes
```

| Cluster    | Name  | Location | Ready | Fenced | Allow | Routing | Routing | Status | Role    | Type | Lock | Duration |
|------------|-------|----------|-------|--------|-------|---------|---------|--------|---------|------|------|----------|
| myclusters | bdra1 | dc1      | true  | false  | true  |         | ok      |        | primary | bdr  | 30   |          |
| myclusters | bdra2 | dc1      | true  | false  | false |         | N/A     |        | primary | bdr  | 30   |          |
| myclusters | bdra3 | dc1      | true  | false  | false |         | N/A     |        | primary | bdr  | 30   |          |

**harpctl get proxy**

指定されたプロキシの DCS に格納されているプロキシ情報を取得します。このクラスターのプロキシのデフォルトを表示するには、`global` を指定します。

例：

```
> harpctl get proxy proxy1
```

| Cluster   | Name   | Pool    | Mode | Auth | Type | Max  | Client | Conn | Default | Pool | Size |
|-----------|--------|---------|------|------|------|------|--------|------|---------|------|------|
| mycluster | proxy1 | session |      | md5  |      | 1000 |        | 20   |         |      |      |

**harpctl get proxies**

クラスター内のすべてのプロキシの DCS に保存されているプロキシ情報を取得します。さらに、デフォルトのプロキシ設定の `global` 擬似プロキシをリストします。

例：

```
> harpctl get proxies
```

| Cluster   | Name   | Pool    | Mode | Auth | Type | Max  | Client | Conn | Default | Pool | Size |
|-----------|--------|---------|------|------|------|------|--------|------|---------|------|------|
| mycluster | global | session |      | md5  |      | 500  |        | 25   |         |      |      |
| mycluster | proxy1 | session |      | md5  |      | 1000 |        | 20   |         |      |      |
| mycluster | proxy2 | session |      | md5  |      | 1500 |        | 30   |         |      |      |

## harpctl manage

クラスターが管理状態にない場合、Postgres のモニタリングとコンセンサスレイヤーの更新を再開するようにすべての HARP Manager サービスに指示します。これは、HARP ソフトウェアの更新またはクラスター全体に影響を与える可能性のあるその他の重要な変更が続いて、メンテナンスが完了した後に実行します。

次のような `manage` コマンドを実行します。

```
harpctl manage cluster
```

**注釈:** 現在、クラスター管理は `cluster` レベルでのみ有効または無効にできます。後のバージョンでは、個々のノードまたはプロキシに対してこれを行うこともできます。

## harpctl promote

リーダーシップ要件を満たす次の利用可能なノードを現在の場所のリードマスターに昇格させます。これは要求されたイベントであるため、次の場合にスムーズなハンドオーバーを呼び出します。

1. 既存のリード CAMO がリード CAMO リースを CAMO する。-古いリードマスターと昇格したノード間のレプリケーションラグは `node.maximum_lag` 未満です。
2. 昇格したノードは、リードマスターリースを取得する唯一の有効な候補であり、現在の所有者によって解放されるとすぐに取得します。他のすべてのノードは、設定されていないリードマスターリースを無視します。CAMO が有効になっている場合、昇格したノードは、リードマスターリースを保持していても、CAMO キューが完全に適用されるまでクライアントトラフィックを一時的に無効にします。
3. HARP プロキシは、`pgbouncer` を使用している場合、進行中のトランザクションを完了できるように `PAUSE` 接続します。昇格したノードがリードマスターリースを要求すると、新しい接続ターゲットの `PgBouncer` を再構成し、データベーストラフィックを再開します。HARP プロキシがビルトインプロキシを使用している場合、クライアントから新しい接続が要求されると、既存の接続を終了し、リードマスターへの新しい接続を作成します。

次のような `promote` コマンドを実行します。

```
harpctl promote (<node-name>)
```

リードマスターになるための基準を満たさない場合でも、ノードを強制的にリードマスターに設定する `--force` オプションを提供します。これにより、CAMO ステータスまたはレプリケーションラグの検証が回避され、昇格したノードに即時に移行されます。これは、昇格するノードを正確に指定する唯一の方法です。

これを成功させるには、ノードがオンラインで動作している必要があります。このオプションは注意して使用してください。

## harpctl set

クラスター内の特定の属性を指定された値に設定します。これは、`apply` を使用するのではなく、特定のノード、プロキシ、場所、またはクラスターの構成設定を調整するために使用されます。これは、次のオブジェクトタイプに使用できます。

- `cluster` — クラスター関連の属性を設定します。
- `location` — 特定のロケーション属性を設定します。
- `node` — 特定のノード属性を設定します。
- `proxy` — 特定のプロキシ属性を設定します。

## harpctl set cluster

クラスター関連の属性のみを設定します。

例：

```
harpctl set cluster event_sync_interval=200
```

## harpctl set node

指定されたノードのノード関連の属性を設定します。ここでは、Node directives に記載されているオプションが有効です。

例：

```
harpctl set node mynode priority=500
```

## harpctl set proxy

名前付けたプロキシのプロキシ関連の属性を設定します。

Proxy directives に記載されているオプションはここで有効です。この方法で設定されたプロパティでは、新しい値を有効にする前にプロキシを再起動する必要があります。

例：

```
harpctl set proxy proxy1 max_client_conn=750
```

クラスター全体のプロキシのデフォルトには `global` を使用します。

```
harpctl set proxy global default_pool_size=10
```



## harpctl unfence

ローカルまたは指定されたノードから **fenced** 属性を削除します。これにより、以前に適用されたすべてのクラスター除外がノードから削除され、再びトラフィックを受信したり、リードマスターリースを保持したりできます。Postgres も実行されていない場合は起動されます。

次のような **unfence** コマンドを実行します。

```
harpctl unfence (<node-name>)
```

node-name はオプションです。省略すると、**harpctl** はローカルに構成されたノードの名前を使用します。

## harpctl unmanage

クラスター内のすべての HARP Manager サービスに、実行を継続するように指示しますが、Postgres をアクティブに監視したり、コンセンサスレイヤーのコンテンツを変更したりします。これは、ノードの停止などの通常のフェイルオーバーイベントが原因でリーダーシップの移行が行われないことを意味します。これは、HARP ソフトウェアに変更を加える前またはクラスターにその他の重要な変更を加える前のシステムまたは HARP のメンテナンスを対象としています。

次のような **unmanage** コマンドを実行します。

```
harpctl unmanage cluster
```

---

**注釈:** 現在、クラスター管理は `cluster` レベルでのみ有効または無効にできます。後のバージョンでは、個々のノードまたはプロキシに対してこれを行うこともできます。

---

## Consensus layer considerations

HARP は、分散制御システム (DCS) としても知られるコンセンサスレイヤーのさまざまな実装で動作できるように設計されています。

現在、次の DCS 実装がサポートされています。

- etcd - BDR

この情報は、サポートされている DCS 実装と HARP の相互作用に固有です。

## BDR ドライバーの互換性

bdr ネイティブコンセンサスレイヤーは、BDR バージョン 3.6.21 および 3.7.3 から利用できます。

投票クォーラムを維持するために、BDR 論理スタンバイノードは EDB Postgres 分散クラスターでのコンセンサス通信に参加しません。DCS クォーラム要件を満たすために、これらを合計ノードリストでカウントしないでください。

## クォーラムの維持

どのアーキテクチャのクラスターでも、投票クォーラムを介してコンセンサスを維持するには、少なくとも  $n/2 + 1$  個のノードが必要です。したがって、3 ノードのクラスターは単一ノードの停止を許容でき、5 ノードのクラスターは 2 ノードの停止を許容できます。コンセンサスが失われると、DCS が特定の場所のリードマスターであるノードを決定的に識別できないため、HARP は動作不能になります。

その結果、どちらの DCS を選択しても、ノードの半分以上が常に\_クラスター全体で\_使用可能である必要があります。これは、DCS ノードを 2 つ以上のデータセンターに分散する場合、重要な要素になる可能性があります。ネットワークパーティションは、投票過半数を維持できない場所でクォーラムを防止するため、HARP が動作を停止します。

したがって、コンセンサスレイヤーを構築するときは、奇数のノード（最低 3 つ）が重要です。理想的なケースは、ノードを少なくとも 3 つの独立した場所に分散して、単一のネットワークパーティションがコンセンサスを混乱させないようにすることです。

1 つの構成例は、2 つのデータセンターにプライマリ BDR ノードと一致する 2 つの DCS ノードを指定し、他の場所に 5 番目の DCS ノード（BDR など）を指定することです。このような設計を使用すると、2 つの BDR データセンター間のネットワークパーティションは、独立して配置されたノードのおかげでコンセンサスを混乱させません。

## マルチコンセンサスバリエーション

HARP は、設定されたロケーションごとに 1 つのリードマスターを想定しています。通常、各場所は `location` 構成設定を使用して HARP で指定されます。ロケーションごとに個別の DCS クラスターを作成することにより、HARP とは無関係にこの動作をエミュレートできます。

これを行うには、目的の場所ごとに異なる DCS 接続ターゲットを使用するように `config.yml` で HARP を構成します。

DC-A の HARP ノードは次のようなものを使用します。

```
location: dca
dcs:
  driver: etcd
  endpoints:
    - dcs-a1:2379
    - dcs-a2:2379
    - dcs-a3:2379
```

DC-B は、正規の場所にあるノードに対応する異なるホスト名を使用しますが：

```
location: dcb
dcs:
  driver: etcd
  endpoints:
    - dcs-a1:2379
    - dcs-a2:2379
    - dcs-a3:2379
```

このデザインには異なるデータセンター間の DCS 通信がないため、それらの間のネットワークパーティションは HARP オペレーションに影響しません。この結果、HARP は他の場所のノードを完全に認識せず、各場所は本質的に別個の HARP クラスターとして動作します。

BDR を DCS として使用する場合、BDR はすべての参加者ノードにわたってコンセンサスレイヤーを維持するため、これは不可能です。

このアプローチの潜在的な欠点は、`harpctl` が現在の場所の外部のノードと対話できないことです。ノード情報の取得、リードマスターの取得または設定、または他の場所をターゲットとするその他の操作を実行することはできません。基本的に、この組織は `harpctl` への `--location` パラメーターを使用不可にします。

### TPAexec とコンセンサス

これらの考慮事項も TPAexec に統合されています。etcd を使用してクラスターをデプロイすると、ロケーションごとに個別の DCS クラスターが構築され、厳密な一貫性を維持しながら高可用性が促進されます。

したがって、この構成例では、`first` の場所に割り当てられた DCS ノードをグループ化し、`second` の場所は別のクラスターです。

```
cluster_vars:
  failover_manager: harp
  harp_consensus_protocol: etcd

locations:
  - Name: first
  - Name: second
```

この動作をオーバーライドするには、特定のグループ化を強制するように `harp_location` を暗黙的に構成します。

したがって、この例では、すべての etcd ノードを単一の凝集 DCS レイヤーに返します。

```
cluster_vars:
  failover_manager: harp
  harp_consensus_protocol: etcd

locations:
```

(次のページに続く)

- **Name:** first
- **Name:** second
- **Name:** all\_dcs

**instance\_defaults:**

**vars:**

**harp\_location:** all\_dcs

**harp\_location** オーバーライドは、従来のデータセンターよりもリージョンのアベイラビリティゾーンを優先する Amazon などのクラウドプロバイダーを使用する場合、特定のノードグループを優先するためにも必要になる場合があります。

## Security and roles

基本的なパッケージのインストールと構成を超えて、HARP を動作させるには Postgres の権限が必要です。これらにより、コンセンサスレイヤーでノードステータスを維持するために必要に応じて Postgres または BDR に関する情報を収集できます。

## Postgres のアクセス許可

次のいずれかの方法で、`node.dsn` パラメーターで指定されたロールを作成します。

- `CREATE USER ...`
- `CREATE ROLE ... WITH LOGIN`

この構文により、ロールはデータベースにログインして診断情報を収集できます。

同様に、このロールの `pg_hba.conf` にエントリが存在する必要があります。これにはさまざまな方法があります。例として、すべてのデータベースホストが `10.10.*` のどこかにある VPN サブネットを考えます。このような場合、最も簡単なアプローチは特定の行を追加することです。

| #       | TYPE | DATABASE | USER      | ADDRESS      | METHOD        |
|---------|------|----------|-----------|--------------|---------------|
| hostssl | all  |          | harp_user | 10.10.1.1/16 | scram-sha-256 |

**注釈:** この場合、現在非推奨の ``md5`` ではなく、より新しい ``scram-sha-256`` 認証を使用しています。 `hostssl` を指定して、SSL 認証を要求することも選択しました。

## BDR のアクセス許可

BDR ノードには、特定のロールが HARP 対応ユーザに付与されている場合にのみ表示されるメタデータとビューがあります。この場合、HARP ユーザーには以下が必要です。

```
GRANT bdr_monitor TO harp_user;
```

bdr\_monitor BDR ロールは、クラスターオペレーションに関する進行中の情報を維持するステータスマonitoring ツールを対象としているため、HARP に最適です。

## BDR コンセンサス権限

dcs.driver 構成パラメーターが bdr に設定されている場合、HARP は BDR をコンセンサスレイヤーとして使用します。そのため、現在 bdr\_superuser ロールでのみ使用可能な API メソッドへのアクセスが必要です。これは、HARP 対応のユーザーが以下を必要とすることを意味します。

```
GRANT bdr_superuser TO foobar;
```

現在、BDR コンセンサスモデルへのアクセスには、スーパーユーザーと同等の権限が必要です。

---

**重要:** BDR スーパーユーザーは Postgres スーパーユーザーではありません。bdr\_superuser ロールには、制限された機能、テーブル、ビュー、その他のオブジェクトへのアクセスなど、BDR で昇格した権限が付与されるだけです。

---

## 4.1.2 BDR (Bi-Directional Replication)

### 概要

BDR は、マルチマスターレプリケーションとデータ分散を高度な競合管理、データ損失保護、およびネイティブ論理レプリケーションよりも最大 5 倍高速なスループットで提供する PostgreSQL の拡張機能であり、最大ファイブナインの高可用性を備えた分散 PostgreSQL クラスタを有効にします。

BDR の動作に関する詳細な概要は、[Architectural overview](#) の章で説明されています。

### サポートされている PostgreSQL データベースサーバー

BDR は Postgres、EDB Postgres Extended Server、および EDB Postgres Advanced Server ディストリビューションと互換性があり、標準の Postgres 拡張機能として展開できます。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 matrix](#) を参照してください。

一部の主要な BDR 機能は、ターゲットの 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ます。したがって、BDR 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、BDR 機能 Commit At Most Once (CAMO) 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、CAMO を処理するために必要なコア機能がないため、コミュニティの PostgreSQL ディストリビューションを採用しないでください。

*Choosing a Postgres distribution* の完全な機能マトリックスの互換性を参照してください。

**BDR は、ネイティブに近い Postgres 互換性を提供します。ただし、一部のアクセスパターンは、マルチノードセットアップで単一のインスタンスで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も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 BDR がどのように動作するかについて詳しく説明します。

## Release Notes

BDR は、マルチマスターレプリケーションとデータ分散、および高度な競合管理、データ損失保護、およびネイティブ論理レプリケーションよりも最大 5 倍の高速なスループットを提供する PostgreSQL の拡張機能であり、非常に高可用性の分散 PostgreSQL クラスタを有効にします。

このセクションのリリースノートには、各リリースの新機能に関する情報が記載されています。

| Version          | Release Date |
|------------------|--------------|
| <i>BDR 4.1.1</i> | 2022 June 21 |
| <i>BDR 4.1.0</i> | 2022 May 17  |
| <i>BDR 4.0.2</i> | 2022 Feb 15  |
| <i>BDR 4.0.1</i> | 2022 Jan 18  |
| <i>BDR 4.0.0</i> | 2021 Dec 01  |

### BDR 4.1.1

これは BDR 4 のメンテナンスリリースで、新機能と、以前のバージョンで特定された問題の修正が含まれています。

| Type    | Category        | Description                                                                    |
|---------|-----------------|--------------------------------------------------------------------------------|
| Feature | User Experience | ジェネリック関数を追加します bdr.is_node_connected は、特定のピアの walsender がアクティブな場合に true を返します。 |
| Feature | User Experience | ジェネリック関数を追加します bdr.is_node_ready は、ラグが特定のスパン未満にある場合にブール値を返します。                 |

次のページに続く

表 10 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                                                                                                                                                    |
|---------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | User Experience             | ハードリンクを使用するための <code>bdr.pg_upgrade</code> に <code>--link</code> 引数のサポートを追加します。                                                                                                                                                                                                                                                |
| Bug fix | User Experience             | デフォルトのコミットスコープとして <code>bdr.node_group</code> に よ っ てまだ参照されている場合、 <code>bdr.remove_commit_scope</code> を削除しないようにします。                                                                                                                                                                                                           |
| Bug fix | Reliability and operability | ノードの CAMO ペアのローカルモードへの Raft ベースの正しい切り替え (RT78928)                                                                                                                                                                                                                                                                              |
| Bug fix | Reliability and operability | コーナケースの <code>bdr.drop_node</code> の潜在的なセグメンテーションを防止 (RT81900)                                                                                                                                                                                                                                                                 |
| Bug fix | User Experience             | トランザクションストリーミングと組み合わせた CAMO または Eager All Node トランザクションの使用を防止する <br/> CAMO および Eager All Node トランザクションと組み合わせたトランザクションストリーミングに問題があることが判明しました。これが解決されるまで、BDR はその併用を禁止します。これには、CAMO 展開で構成を調整してトランザクションストリーミングを無効にする必要がある場合があります。<br>:ref:`Transaction Streaming Configuration<../transaction-streaming#configuration>` を参照してください。 |

アップグレード

このリリースは、次のバージョンの BDR からのアップグレードをサポートしています。

- 4.0.0 以降
- 3.7.15
- 3.7.16

アップグレードする前に、

Upgrade Guide に記載されているプロセスと制限を読んで理解してください。

BDR 4.1.0

これは BDR 4 のマイナーリリースであり、新機能と、以前のバージョンで特定された問題の修正が含まれています。

| Type    | Category                    | Description                                                                                                                                                                                                                                                 |
|---------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Reliability and operability | BDR ノードでの Postgres のインプレースメジャーアップグレードをサポート<p>この BDR リリースには、新しいコマンドラインユーティリティ `bdr_pg_upgrade` が含まれています。`pg_upgrade` を使用して、BDR ノードで Postgres のメジャーバージョンアップグレードを実行します。</p><p>EDB Postgres 分散クラスターで Postgres のメジャーバージョンアップグレードを行うために必要な時間とネットワーク帯域幅を削減します。</p> |

次のページに続く



表 11 – 前のページからの続き

| Type    | Category                                | Description                                                                                                                                                       |
|---------|-----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Performance and scalability             | レプリケーションラグ制御<br><p>トランザクションのコミットが調整されるまでのレプリケーションラグしきい値の構成を追加します。これにより、同期レプリケーションに伴うすべてのトランザクションにレイテンシーの影響を与えることなく、RPO を制限できます。</p>                               |
| Feature | UX / Initial experience                 | デフォルトでの分散シーケンス<p> <i>bdr.default_sequence_kind</i> のデフォルト値が'distributed'に変更されました。これは、'bigserial'に SnowflakeId を使用し、'serial'列タイプに Galloc シーケンスを使用します。</p>          |
| Feature | UX                                      | 簡素化された同期レプリケーション構成<p>BDR グループと SQL ベースの管理（構成ファイルではない）に重点を置いた、同期レプリケーションオプションを指定するための新しい構文。</p><p>将来のバージョンでは、これは現在の Eager Replication と CAMO 構成オプションにも置き換わります。</p> |
| Feature | High availability and disaster recovery | グループコミット<p>新しい構成構文で構成できる初期の種類の同期コミット。</p>                                                                                                                         |
| Feature | High availability and disaster recovery | ローカルモードへの CAMO の切り替えに Raft リクエストが必要になることを CAMO します。に接続して、ローカルモードに切り替えます。</p>                                                                                      |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category                                | Description                                                                                                                                                                                                                                                                     |
|---------|-----------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | High availability and disaster recovery | 既にデタッチされたパーティションの `ALTER TABLE ... DETACH PARTITION CONCURRENTLY` でレプリケーションを続行できるようにする (RT78362) <p>同じインデックスが既に存在する場合に BDR 4 が `CREATE INDEX CONCURRENTLY` を処理する方法と同様に、`ALTER TABLE ... DETACH PARTITION CONCURRENTLY` が既にデタッチされたパーティションのレシーバーである場合にレプリケーションを続行できるようになりました。</p> |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                                                                                                                                                                                                         |
|---------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | User Experience             | DDL フィルターに追加のフィルタリングオプションを追加します。<p>DDL フィルターを使用すると、さまざまな DDL ステートメントをさまざまなレプリケーションセットに複製できます。レプリケーションセットのテーブルメンバーシップにより、さまざまなテーブルの DML をさまざまなレプリケーションセットを介してレプリケートできるのと同様に。</p><p>このリリースでは、DDL フィルタを使いやすくする新しいコントロールが追加されています。<br />- query_match - 定義されたクエリがこの正規表現と一致する必要がある場合<br />- 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、排他フィルタのみが適用されます）。複数の排他フィルタが一致すると、エラーがスローされます</p> |
| Feature | User Experience             | `bdr.lock_table_locking` 構成変数を追加します。<p>有効にすると、グローバル DML ロックを取得するように `LOCK TABLE` コマンドの動作が変更されます</p>                                                                                                                                                                                                                                                                                 |
| Feature | Performance and scalability | LCR セグメントファイル<p>のバッファリングされた書き込みを実装します。これにより、I/O が削減され、Decoding Worker の CPU 使用率が向上します。</p>                                                                                                                                                                                                                                                                                          |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | User Experience | 競合検出のために部分的な一意のインデックス検索のサポートを追加しました (RT78368)。<p>ただし、式のインデックスは競合検出でまだサポートされていません。</p>                                                                                                                                                                                                                                                                                                                                                                              |
| Feature | User Experience | `bdr.stat_subscription`に追加の統計を追加します。<br /> - nstream_insert =>ストリーミングされたトランザクションの INSERT の数<br /> - nstream_update =>ストリーミングされたトランザクションの UPDATE の数<br /> - nstream_delete =>ストリーミングされたトランザクションの数 transaction<br /> - nstream_truncate =>ストリーミングされたトランザクションの TRUNCATE の数<br /> - npre_commit_confirmations => CAMO を使用する場合のプリコミット確認の数<br /> - npre_commit =>プリコミットの数<br /> - ncommit_prepared =>2PCでの準備されたコミットの数<br /> - nabort_prepared =>2PCでの準備されたトランザクションのアボート数 |
| Feature | User Experience | execute_locally オプションを bdr.replicate_ddl_command に追加しました (RT73533)。<p>これにより、ローカルで実行せずに他のグループにレプリケーションするための ddl コマンドをオプションでキューイングできます。</p>                                                                                                                                                                                                                                                                                                                          |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category           | Description                                                                                                                                                                                                                                                                                                                                           |
|---------|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | User Experience    | <code>`bdr.alter_subscription_disable()</code> に<br><code>`fast`</code> 引数を追加 (RT79798)<br><p>&lt;p&gt;引 数 は <code>`immediate`</code> の<br/> 動作にのみ影響します。<br/> <code>`true`</code> (デフォルト) に設定<br/> すると、現在の作業を終了<br/> させずにワーカーを停止し<br/> ます。&lt;/p&gt;</p>                                                                                              |
| Feature | User Experience    | すべてのタイプのワーカー<br>の <code>`bdr.worker_error`</code> レコ<br>ードを永続的に保持します。<br><p>&lt;p&gt; BDR は、レシーバーと<br/> ライタのエラーが以前にエ<br/> ラーになった LSN を複製で<br/> きたときに使用されました。<br/> ただし、これは他のワーカー<br/> の動作と一貫性がありません<br/> でした。他のワーカーのエラ<br/> ーは永続的なものであり、過<br/> 去の問題のトラブルシューテ<br/> ィングも困難になりました。し<br/> たがって、すべてのワーカー<br/> タイプの最後のエラーレコー<br/> ドを永続的に保持します。&lt;/p&gt;</p> |
| Feature | User Experience    | <code>void</code> を返すように<br><code>`bdr.{add,remove}_camo_pair`</code> 関<br>数を簡素化します。                                                                                                                                                                                                                                                                  |
| Feature | Initial Experience | グローバルロックを取得す<br>る前に接続/ラグチェック<br>を追加します。<p>これに<br>より、アプリケーションまた<br>はユーザーは、明らかな<br>接続の問題がある場合に<br>ロックタイムアウトを取<br>得する必要がありません。<br></p><p>DEBUG、LOG に設<br>定できます。、WARNING<br>(デフォルト) または ER<br>ROR。</p>                                                                                                                                                    |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                  |
|---------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Initial Experience          | デフォルトでは、競合のみを競合ログテーブルに記録します。デフォルトではサーバーログファイルに記録されなくなりましたが、これはオーバーライドできます。                                                                                   |
| Feature | User Experience             | ノード参加中のリモートエラーのレポートを改善します。                                                                                                                                   |
| Feature | Reliability and operability | 自動パーティションワーカーの最大昼寝時間を設定可能にします。                                                                                                                               |
| Feature | User Experience             | autopartition で指定された上限までパーティションを要求する機能を追加します。                                                                                                                |
| Feature | Initial Experience          | サブスクリプト専用ノードでレプリケート DDL を実行しないでください。複製する場所がないため、複製しようとしても失敗します。これは、ロジカルスタンバイの動作と同じです。                                                                        |
| Feature | User Experience             | <code>`bdr.accept_connections`</code> 構成変数を追加します。 <code>`false`</code> の場合、BDR 出力プラグインを使用したレプリケーションスロットへの walsender 接続が失敗します。これは、主にバックアップからの単一ノードの復元中に役立ちます。 |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                                                            |
|---------|-----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Reliability and operability | 非 CAMO パートナー BDR ノードで `lock_timeout` を構成したままにします<p>CAMO パートナーは、オリジンノードからトランザクションを適用するときに低い `lock_timeout` を使用します。これは、CAMO パートナーだけでなく、すべての BDR ノードに対して誤って行われたため、通常の BDR ノードの pglogical ライタプロセスで誤った `lock_timeout` エラーが発生した可能性があります。</p> |
| Bug fix | User Experience             | CAMO / Eager 確認待機の適切な待機イベントを表示する (RT75900) <p>デフォルトの「不明な待機イベント」ではなく、`bdr.stat_activity` の正しい「BDR 準備フェーズ」/「BDR コミットフェーズ」を表示します。</p>                                                                                                     |
| Bug fix | User Experience             | bdr.run_on_nodes のログを削減する (RT80973) <p>`bdr.ddl_replication` をオフに設定すると、関数の「run_on_nodes」バリエーションが使用されます。これにより、モニタリング機能のログの洪水がなくなります。</p>                                                                                                |
| Bug fix | Reliability and operability | バイナリネットワークエンコーディングをサポートしていない複合型の配列と組み込み型の配列のレプリケーションを修正                                                                                                                                                                                |
| Bug fix | Reliability and operability | ブートストラップ中に作成されたデータ型のレプリケーションを修正                                                                                                                                                                                                        |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                 |
|---------|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Performance and scalability | WAL デコーダーによって処理された実行中のトランザクションレコードの終了 LSN を確認して、WAL デコーダースロットが最新の状態を維持し、WAL 送信者がタイムリーに候補を取得します。                                                                                             |
| Bug fix | Reliability and operability | 分割ノードで自動パーティションタスクが完了するのを待たない                                                                                                                                                               |
| Bug fix | Reliability and operability | フラッシュ位置を物理スロットのみに報告するときに `bdr.standby_slot_names` チェックを制限します。                                                                                                                               |
| Bug fix | Reliability and operability | wal_receiver_timeout が近い場合は、walsender にフィードバックの返信を要求します                                                                                                                                     |
| Bug fix | Reliability and operability | BDR 拡張機能で自動パーティション分割されたテーブルの依存関係を複数回記録しないでください。 <p>これにより、自動パーティションから「予期しない数の拡張機能依存関係レコード」エラーが発生し、これが発生すると競合が発生します。 </p><p>`pg_depend` から二重の依存関係を削除して、既存の壊れたテーブルを手動で修正する必要があることに注意してください</p> |

次のページに続く



表 11 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                            |
|---------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Reliability and operability | レシーバーでのキープアライブ処理を改善します。<p>ストリーミングトランザクションの最中は、クラッシュ時にデータを失う可能性があるため、キープアライブに基づいて位置を更新しないでください。</p>改善すべき新しいフラッシュとシグナリングロジックもあります低 TPS シナリオでの遅延。         |
| Bug fix | Reliability and operability | BDR ノードがデータベースに存在する場合にのみ、 <code>post `CREATE`</code> コマンドを処理します。                                                                                       |
| Bug fix | Reliability and operability | 競合履歴テーブルに ERROR 競合をログに記録しないでください。                                                                                                                      |
| Bug fix | Reliability and operability | マルチ挿入 (COPY) 中に <code>conflict_slot</code> が解放された後に使用されていた <code>segfault</code> を修正しました (RT76439)。                                                    |
| Bug fix | Reliability and operability | 遅延スタンバイスロットに直面したときにワルセNDERプロセスがスピンしないようにします (RT80295、RT78290)。<p>スタンバイスロットの 1 つが遅延している場合にワルセNDERプロセスが CPU の 100 % を消費するように、ラッチをリセットするシグナリングを修正します。</p> |
| Bug fix | Reliability and operability | <code>`bdr.standby_slot_names`</code> 使用時の <code>`wal_sender_timeout`</code> のハンドリングを修正 (RT78290)                                                      |
| Bug fix | Reliability and operability | ALTER TABLE で基になるリレーションを 1 回だけロックするようにします (RT80204)。                                                                                                   |

次のページに続く

表 11 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                               |
|---------|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | User Experience             | <code>`bdr.monitor_local_replslots`</code> の切断されたスロットのレポートを修正しました。これらは、以前に切断されたのではなく、欠落していると報告された可能性があります。                                                                |
| Bug fix | Reliability and operability | <code>`bdr.get_subscription_progress()`</code> フังก์ションおよびそのフังก์ションを使用する <code>`bdr.subscription_summary`</code> でダウンロードしたサブスクリプションのタイムスタンプレポートを修正しました。以前はゴミの値を報告していました。 |
| Bug fix | Reliability and operability | BDR ワーカーのさまざまな場所でのスナップショットの処理を修正しました。                                                                                                                                     |
| Bug fix | User Experience             | モニタリング機能で使用可能な値がない場合、タイムスタンプと LSN を NULL として報告することについて、より一貫性を保ちます。                                                                                                        |
| Bug fix | Reliability and operability | ライタプロセスを切り替えるときのログ情報を削減します。                                                                                                                                               |
| Bug fix | Reliability and operability | PG コマンドラインで構成パラメーターが指定された場合、スーパーユーザーチェックを行いません。そこではまだトランザクションを実行できません。その段階でスーパーユーザーが変更されることが保証されています。                                                                     |
| Bug fix | Reliability and operability | バイト単位のラグサイズを計算するには 64 ビットを使用します。ラグの大きいオーバーフローのリスクをなくす。                                                                                                                    |

## アップグレード

このリリースは、次のバージョンの BDR からのアップグレードをサポートしています。

- 4.0.0 以降
- 3.7.15
- 3.7.16

アップグレードする前に、

Upgrade Guide に記載されているプロセスと制限を読んで理解してください。

## BDR 4.0.2

これは BDR 4.0 のメンテナンスリリースで、以前のバージョンで特定された問題の修正とマイナーな改善が含まれています。

| Type        | Category                    | Description                                                                                                                                                                                                                                     |
|-------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Improvement | Reliability and operability | <code>bdr.max_worker_backoff_delay</code> を追加 (BDR-1767) <p>これにより、バックオフ遅延の処理が変更され、エラーが繰り返される場合に <code>bdr.min_worker_backoff_delay</code> から <code>bdr.max_worker_backoff_delay</code> に指数関数的に増加します。これにより、ログのスパムが減少し、場合によっては不要な接続試行も防止されます。</p> |
| Improvement | User Experience             | <code>`bdr.replicate_ddl_command()`</code> に <code>`execute_locally`</code> オプションを追加 (RT73533) <p>これにより、ローカルで実行せずに他のグループにレプリケーションするための <code>ddl</code> コマンドをオプションでキューイングできます。</p>                                                              |

次のページに続く

表 12 – 前のページからの続き

| Type        | Category                    | Description                                                                                                                                                 |
|-------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Improvement | User Experience             | JOIN 中のコンセンサス問題の ERROR を WARNING<p>に変更する これらの一時的なエラーの報告は、bdr.worker_errors にも表示されるため、混乱を招きました。これらは WARNING に変更されました。</p>                                    |
| Bug fix     | Reliability and operability | WAL デコーダーは、実行中のトランザクションレコードの終了 LSN を確認します (BDR-1264) <p> WAL デコーダーによって処理された実行中のトランザクションレコードの終了 LSN を確認して、WAL デコーダースロットが最新の状態を維持し、WAL 送信者がタイマーに候補を取得します。</p> |
| Bug fix     | Reliability and operability | 分割ノードで自動パーティションタスクが完了するまで待機しないでください (BDR-1867) <p>.</p>                                                                                                     |
| Bug fix     | User Experience             | 並列結合中のノード名の再利用の処理を改善しました (RT74789) <p>ノードに世代番号があるため、ノードレコードがスナップショットの一部として受信された場合でも名前の再利用を簡単に識別できます。</p>                                                    |
| Bug fix     | Reliability and operability | BDR マネージャープロセスでのノード管理中のロックとスナップショットの使用を修正しました (RT74789) <p>が考慮されます。</p>                                                                                     |

次のページに続く

表 12 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                   |
|---------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Reliability and operability | ローカルノードのドロップでのカタログのクリーンアップを改善します<p>プライマリだけでなく、すべてのグループを削除し、すべてのノード状態の履歴情報も削除します。</p>                                          |
| Bug fix | User Experience             | bdr_init_physical<p>の参加リクエストのエラーチェックを改善                                                                                       |
| Bug fix | Reliability and operability | コンセンサスでのさまざまなタイムアウトとスリープの処理を改善します<p>これにより、多くのコンセンサスリクエストや時間のかかるコンセンサスリクエストを処理するときに必要な新しいコンセンサス投票の量が削減されます。たとえば、新しいノードの参加。</p> |
| Bug fix | Reliability and operability | `wal_receiver_timeout` の処理を修正しました (BDR-1848) <p> BDR 3.7 および 4.0 でのリグレッションのため、`wal_receiver_timeout` は正しくトリガーされませんでした。</p>    |
| Bug fix | Reliability and operability | フラッシュ位置を物理スロットのみに報告するときに `bdr.standby_slot_names` チェックを制限します。                                                                 |
| Bug fix | Reliability and operability | ブートストラップ中に作成されたデータ型のレプリケーションを修正しました (BDR-1784)                                                                                |
| Bug fix | Reliability and operability | バイナリ転送をサポートしていないビルトイン型の配列のレプリケーションを修正しました (BDR-1042)                                                                          |
| Bug fix | Reliability and operability | CAMO が使用されていない場合に CAMO 構成の警告が表示されないようにしました (BDR-1825)                                                                         |

## アップグレード

このリリースは、次のバージョンの BDR からのアップグレードをサポートしています。

- 4.0.0 以降

BDR 3.7 からのアップグレードパスは現在安定していないため、ベータ版と見なす必要があります。テストは少なくとも BDR 3.7.15 で実行する必要があります。

### アップグレードする前に、

Upgrade Guide に記載されているプロセスと制限を読んで理解してください。

## BDR 4.0.1

これは BDR 4.0 のメンテナンスリリースで、以前のバージョンで特定された問題の修正とマイナーな改善が含まれています。

| Type        | Category                     | Description                                                                                                                                                            |
|-------------|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Improvement | Reliability and operability  | CAMO パートナーの接続試行の頻度を減らします。<br><p>構成を確認し、トランザクションのステータスを確認するために CAMO パートナーに接続できない場合、すぐに再試行しないでください（完全にビジーな pglogical マネージャープロセスにつながります）。1 分に 1 回。<br></p>                |
| Improvement | RPerformance and scalability | LCR セグメントファイルのバッファ読み取りを実装します（BDR-1422）<p>一度に複数の LCR チャンクを読み取れるように、LCR セグメントファイルのバッファリングを実装します。これにより、Decoding Worker を使用する際の Wal Senders の I/O が削減され、CPU 使用率が向上します。</p> |

次のページに続く

表 13 – 前のページからの続き

| Type        | Category                    | Description                                                                                                                                     |
|-------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| Improvement | Performance and scalability | 不要な LCR セグメントの読み取りを回避 (BDR-1426)<br><p> BDR は、少なくとも 1 つの使用可能な場合に新しい LCR セグメントのみを読み取ろうとします。これにより、Decoding Worker が有効になっているときの I/O 負荷が削減されます。</p> |
| Improvement | Performance and scalability | 結合中の初期 COPY を含む COPY レプリケーションのパフォーマンスが、パーティションテーブルで大幅に改善されました (BDR-1479)<br><p>大きなテーブルの場合、これによりロード時間が 1 桁以上改善されます。</p>                          |
| Bug fix     | Performance and scalability | パラレル適用ワーカーの選択を修正します (BDR-1761)<br><p>これにより、パラレル適用が再度動作します。4.0.0 では、このバグのため、並列適用は有効になりませんでした。</p>                                                |
| Bug fix     | Reliability and operability | `bdr.camo_pairs` の Raft スナップショットの処理を修正しました (BDR-1753) <p>以前のリリースでは、Raft スナップショットを介して受信したときに CAMO ペア構成に変更を正しく伝播しませんでした。</p>                      |
| Bug fix     | Reliability and operability | アップグレード後に BDR 3.7 から Raft スナップショットを正しく処理する (BDR-1754)                                                                                           |

次のページに続く

表 13 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                              |
|---------|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Reliability and operability | クラスターのインプレースアップグレードを実行する機能を除外しながら、スナップショットの <i>bdr.camo_pairs</i> を考慮して CAMO 構成済みクラスターをアップグレードする (CAMO とは関係のないアップグレードの制限のため)。                                                            |
| Bug fix | Reliability and operability | タイムアウト後にのみ CAMO からローカルモードに切り替える (RT74892)<br><p> CAMO 保護からローカルモードに切り替えるときに `catchup_interval` 推定値を使用しないでください。推定値は、ローカルモードから CAMO 保護に戻す場合にのみ使用してください (CAMO パートナーでの遅延が原因でトグルするのを防ぐため)。</p> |
| Bug fix | Reliability and operability | パブリッシュされたレプリケーションセットリストが変更されたときのレプリケーションセットキャッシュの無効化を修正しました (BDR-1715) <p>以前のバージョンでは、サブスクリプションが再接続されるまでにパブリッシュする必要があるレプリケーションセット (および結果としてどのテーブル) に関する古い情報を使用できました。</p>                  |

次のページに続く



表 13 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                                                                                                                                      |
|---------|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Reliability and operability | 新しいチャンクが使用される場合、同時実行性の高い状況で galloc シーケンスによってローカルに生成される値の重複を防止します (RT76528) <p>新しい値を待っているセッション。これは修正されました。</p>                                                                                                                                                                                                    |
| Bug fix | Reliability and operability | ストリーミングトランザクションでのメモリリークに対処します (BDR-1479) <p>大規模なトランザクションの場合、ストリーミングトランザクション機能を使用すると、メモリ使用量と I/O が大幅に削減されます。これにより、主に COPY レプリケーションのパフォーマンスが向上します。</p>                                                                                                                                                             |
| Bug fix | Reliability and operability | ノードが分離しているときにキャッチアップソースが変更された場合、ノードの PART_CATCHUP フェーズの後にスロットを残さないでください (BDR BDR) <p>すべてのノードのデータの一貫性を保つために、そのノードから欠落している変更を残りのノード間で転送します。これには、追加のレプリケーションスロットを一時的に作成する必要があります。通常、このレプリケーションスロットはキャッチアップフェーズの最後に削除されますが、変更のソースノードを変更する必要がある特定のシナリオでは、このスロットは以前に取り残されている可能性があります。このバージョンから、このスロットは常に正しく削除されます。</p> |

次のページに続く

表 13 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                |
|---------|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Bug fix | Reliability and operability | BDR グループにノードが 1 つしかない場合は、グループ スロットを前方に移動します<p>これにより、グループが単一の BDR ノードだけで実行されたままになった場合の WAL の蓄積によるディスクの枯渇を防ぎます。これは推奨される設定ではありませんが、WAL の蓄積は意図的なものではありません。</p> |
| Bug fix | Reliability and operability | BDR グループにノードが 1 つしかない場合の Advanced Raft プロトコルバージョン<p>それ以外の場合、シングルノードクラスターは、別のノードが追加されるまで常に最も古いサポートプロトコルを使用します。これにより、その単一のノードで使用可能な機能セットが制限される可能性があります。</p>  |

## アップグレード

このリリースは、次のバージョンの BDR からのアップグレードをサポートしています。

- 3.7.14
- 4.0.0 以降

### アップグレードする前に、

Upgrade Guide に記載されているプロセスと制限を読んで理解してください。

## BDR 4.0.0

BDR 4.0 は BDR の新しいメジャーバージョンであり、このリリース番号で採用されたのはセマンティックバージョンングです（詳細については、[semver.org](https://semver.org) を参照）。以前の 2 つのメジャーバージョンは 3.7 と 3.6 です。

| Type    | Category      | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
|---------|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Compatibility | <p>EDB Postgres Advanced 14 の BDR は、以前は EDB Postgres Extended でのみ使用できた次の機能をサポートするようになりました。</p> <p>&lt;br /&gt;- Commit At Most Once - ノードに障害が発生した場合でも、アプリケーションが各トランザクションを 1 回のみコミットするのに役立つ一貫性機能&lt;br /&gt;- 積極的な Eager Replication- トランザクションをコミットする前にクラスターのノード間を同期して、競合のないレプリケーションを提供します&lt;br /&gt;- ワーカーのデコード - ウォルセンダーから別のワーカーへのデコードの分離により、多くのノードでのスケーラビリティが向上&lt;br /&gt;- レプリケーションのキャッチアップ時間の見積もり&lt;br /&gt;- タイムスタンプベースのスナップショット-複数のノードにわたって一貫した読み取りを提供し、特定の時刻に表示されるまたは表示されるデータを取得します&lt;br /&gt;- 行フリーズの自動化された動的構成 特定のコーナーケースでの UPDATE / DELETE の競合解決の一貫性を向上させるため&lt;br /&gt;-評価チェック&lt;br /&gt;-欠落しているパーティションをエラーではなく競合として処理するためのサポート&lt;br /&gt;- NOT VALID 制約と ALTER TABLE の高度な DDL 処理</p> |

次のページに続く

表 14 – 前のページからの続き

| Type    | Category      | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
|---------|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Compatibility | PostgreSQL 12-14 のコミュニティバージョンの BDR は、以前は EDB Postgres Advanced または EDB Postgres Extended でのみ使用できた次の機能をサポートするようになりました。<br />- 競合のないレプリケートされたデータ型- 非同期マルチプルで数学的に証明された一貫性を提供する追加のデータ型- マスター更新シナリオ<br />- 列レベルの競合解決- 列ごとに使用する機能の最終更新が解決を優先するため、異なるフィールドの UPDATE をいずれも失うことなく「マージ」できます<br />- 変換トリガー- そのトリガーデータの着信ストリームで実行され、変更したり、高度なプログラムによるフィルタリングを実行したりできます<br />- 競合トリガー- 競合が検出されたときに呼び出され、カスタムの競合解決手法を使用する方法を提供します<br />- CREATE TABLE AS レプリケーション<br />- パラレル適用- 複数のライターが着信変更を適用できるようにします |

次のページに続く

表 14 – 前のページからの続き

| Type               | Category    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|--------------------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature            | Performance | <p>大規模なトランザクションのストリーミングをサポートします。 &lt;p&gt;これにより、 BDR は大きなトランザクション（サイズが`logical_decoding_work_mem`を超える）をダウンストリームのファイルまたはライタプロセスにストリーミングできます。これにより、トランザクションはコミットされる前でもデコードされるため、並列性が向上します。さらに、ライタに直接ストリーミングする場合、トランザクションを同時に適用することもできます。これにより、並列性がさらに向上します。 &lt;/p&gt;&lt;p&gt;大規模なトランザクションがファイルにストリーミングされると、それらはデコードされ、デコードされた変更はコミットされる前にダウンストリームに送信されます。変更は一連のファイルに書き込まれ、トランザクションが最終的にコミットしたときに適用されます。トランザクションが打ち切られると、変更は破棄されるため、アップストリームとダウンストリームの両方でリソースが無駄になります。 &lt;/p&gt;&lt;p&gt;サブトランザクションも自動的に処理されます。 &lt;/p&gt;&lt;p&gt;この機能は、PostgreSQL 14、 EDB Postgres Extended 13 以降、および EDB Postgres Advanced 14 で利用できます。Postgres のどのバージョンで利用できる機能の詳細については、<a href="#">:ref:Choosing a Postgres distribution&lt;Choosing a Postgres distribution&gt;</a>を参照してください。 &lt;/p&gt;</p> |
| 4.1. Postgres サーバー |             | <p>Postgres distribution&gt;`付 103</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |

表 14 – 前のページからの続き

| Type    | Category        | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
|---------|-----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Compatibility   | <p>標準版とエンタープライズ版の BDR の以前のバージョンに存在した違いは削除されました。 BDR 4.0 では、サポートされている Postgres ディストリビューションとバージョン、つまり PostgreSQL v12-14、EDB Postgres Extended v12-14、および EDB Postgres Advanced 12-14 ごとに 1 つの拡張機能があります。 &lt;p&gt;すべての機能が PostgreSQL のすべてのバージョンで利用できるわけではありません。利用可能な機能は、`bdr_config` コマンドラインユーティリティまたは `bdr.bdr_features()` データベース関数を使用して機能フラグを介して報告されます。詳細については、:ref:`Choosing a Postgres distribution&lt;Choosing a Postgres distribution&gt;` を参照してください。 &lt;/p&gt;</p> |
| Feature | User Experience | <p>BDR 4.0 拡張機能に対応する pglogical 4.0 拡張機能はありません。 BDR には pglogical がなくなりました。 &lt;p&gt;これは、BDR 拡張機能とスキーマのみが存在し、構成パラメーターの名前が `pglogical.` から `bdr.` に変更されたことも意味します。 &lt;/p&gt;</p>                                                                                                                                                                                                                                                                                           |

次のページに続く

表 14 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Initial experience          | <p>一部の構成オプションでは、インストール後のエクスペリエンスを向上させるためにデフォルトが変更されています。&lt;br /&gt;-並列適用がデフォルトで有効になりました（2 つのライター）。特にストリーミングが有効になっている場合、パフォーマンスが向上します。&lt;br /&gt;- `COPY` および `CREATE INDEX CONCURRENTLY` は、デフォルトで使用可能なすべてのノードに並列で（ストリーミングがサポートされている Postgres バージョンで）ライターにストリーミングされ、レプリケーションのラグスパイクを排除または少なくとも削減します。これらの操作の後。&lt;br /&gt;-グローバルロックのタイムアウトが 10 分に延長されました&lt;br /&gt;- `bdr.min_worker_backoff_delay` はデフォルトで 1 秒になり、サブスクリプションはエラー時に 1 秒に 1 回のみ接続を再試行します。</p> |
| Feature | Reliability and operability | <p>オリジンベースの競合検出を使用するテーブルのノード結合中の競合検出で誤検知の可能性が大幅に減少しました</p>                                                                                                                                                                                                                                                                                                                                                                                                   |

次のページに続く

表 14 – 前のページからの続き

| Type    | Category                    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------|-----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Feature | Reliability and operability | <p>CAMO ペアの構成を SQL カタログに移動します&lt;p&gt;構成ミスの可能性を減らし、 EDB Postgres 分散クラスター内の CAMO ペアをグローバルに認識するには、 CAMO 構成を個々のノードの postgresql.conf から Raft が管理する BDR システムカタログに移動します。これにより、たとえば、別のアクティブノードの CAMO パートナーとしてまだ構成されているノードを誤って削除するのを防ぐことができます。 &lt;/p&gt;&lt;p&gt;アップグレードプロセスの詳細については、 :ref:`Upgrades chapter &lt;/pgd/latest/upgrades/#upgrading-a-camo-enabled-cluster&gt;`を参照してください。 &lt;/p&gt;&lt;p&gt;これは GUC <code>`bdr.camo_partner_of`</code>および <code>`bdr.camo_origin_for`</code>を非推奨にし、関数 <code>`bdr.get_configured_camo_origin_for()`</code>および <code>`get_configured_camo_partner_of`</code>を <code>`bdr.get_configured_camo_partner`</code>に置き換えます。 &lt;/p&gt;</p> |

アップグレード

このリリースは、次のバージョンの BDR からのアップグレードをサポートしています。

- 3.7.13.1

アップグレードする前に、

Upgrade Guide に記載されているプロセスと制限を読んで理解してください。



## Architectural overview

BDR は、メッシュトポロジを使用して、疎結合のマルチマスター論理レプリケーションを提供します。これは、任意のサーバーに書き込むことができ、変更は同じ BDR グループに属する他のすべてのサーバーに行ごとに直接送信されることを意味します。



図 6: node diagram

デフォルトでは、BDR は非同期レプリケーションを使用し、ローカルコミット後にのみピアノードに変更を適用します。オプションの *Eager All-Node Replication* を使用した CAMO 機能により、コンセンサスを使用してすべてのノードでコミットできます。

## 基本的なアーキテクチャ

### 複数のグループ

BDR ノードは少なくとも 1 つの **ノードグループ** のメンバーであり、最も基本的なアーキテクチャでは、BDR クラスター全体に 1 つのノードグループがあります。

### 複数のマスター

BDR グループに参加する各ノード（データベース）は、他のメンバーから変更を受け取り、ユーザが直接書き込むことができます。

これは、1 つのマスターサーバーのみが書き込みを受け入れ、他のすべてのノードがマスターまたは別のスタンバイから複製するホットまたはウォームスタンバイとは異なります。

常にすべてのマスターに書き込む必要はありません。頻繁な構成では、ほとんどの場合、1 つのマスターのみに書き込みを指示します。

### デフォルトでは非同期

1 つの BDR ノードで行われた変更は、ローカルにコミットされるまで他のノードにレプリケートされません。その結果、データは常にすべてのノードで完全に同じではありません。一部のノードには、他のノードにまだ到達していないデータがあります。PostgreSQL のブロックベースのレプリケーションソリューションもデフォルトで非同期レプリケーションです。BDR では、マスターが複数あり、その結果、データストリームが複数になるため、`synchronous_commit` および `synchronous_standby_names` を使用した場合でも、異なるノード上のデータが異なる場合があります。

### メッシュトポロジー

BDR は、すべてのノードが他のすべてのノードに接続し、すべてのノードが相互に直接データを交換するメッシュネットワークを中心に構成されています。ノードの追加や削除などの特別な状況を除き、BDR でデータの転送はありません。データは EDB Postgres 分散クラスターの外部から到着するか、ネイティブの PostgreSQL 論理レプリケーションを使用して送信できます。

### 論理レプリケーション

論理レプリケーションは、レプリケーション ID（通常は主キー）に基づいてデータ行とその変更をレプリケートする方法です。正確なブロックアドレスとバイトごとのレプリケーションを使用する **物理**レプリケーションとは対照的に、**論理**という用語を使用します。インデックスの変更は複製されないため、書き込みの増幅が回避され、帯域幅が削減されます。

論理レプリケーションは、ソースノードからデータのスナップショットをコピーすることから始まります。それが完了すると、後のコミットはリアルタイムで発生すると他のノードに送信されます。変更は SQL を再実行せずに複製されるため、書き込まれた正確なデータが迅速かつ正確に複製されます。

ノードはソースノードでコミットされた順序でデータを適用するため、単一ノードからの変更に対してトランザクションの一貫性が保証されます。異なるノードからの変更は他のノードとは無関係に適用され、変更の迅速なレプリケーションが保証されます。

レプリケートされたデータは、安全な場合にバイナリ形式で送信されます。

## 高可用性

各マスターノードは1つ以上のスタンバイノードで保護できるため、ダウンしたノードをすばやく交換して続行できます。各スタンバイノードは、論理スタンバイノードまたはフィジカルスタンバイノードのいずれかです。

1つ以上のノードが現在利用できない場合でも、レプリケーションは現在接続されているノード間で続行されます。ノードが回復すると、レプリケーションを中断したところから再開できます。

ノードはさまざまなリリースレベルを実行し、通信に必要なプロトコルをネゴシエートできます。その結果、EDB Postgres 分散クラスターは、データベースソフトウェアのメジャーバージョンでも、ローリングアップグレードを使用できます。

DDL はデフォルトでノード間で複製されます。DDL の実行は、必要に応じてユーザーが制御して、アプリケーションのローリングアップグレードを許可できます。

## 制限

BDR は、十分なハードウェアとネットワークで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では、通常、1つのクラスターで32を超えるノードを実行することはお勧めしません。各マスターノードは、複数のフィジカルまたはロジカルスタンバイノードで保護できます。スタンバイノードの数に特に制限はありませんが、一般的な使用法はマスターごとに2~3個のスタンバイを持つことです。スタンバイノードはメッシュネットワークに接続を追加しないため、32ノードの推奨には含まれません。

BDR には、現在許可されている現在の最大 Raft 接続であるため、1000 アクティブノードのハード制限があります。

BDR は、1つの PostgreSQL インスタンスで最大10個のデータベースが異なる BDR ノードグループの BDR ノードにできるという制限を設けています。ただし、BDR は、PostgreSQL インスタンスごとに1つの BDR データベースのみを使用する場合に最適に機能します。

BDR 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の、EDB Postgres 分散クラスターの推奨されるノードの最小数は3です。2つのノードだけでは、いずれかのノードが応答しないと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 BDR 操作にはコンセンサスが必要です。EDB Postgres Distributed で使用されるコンセンサスメカニズムの詳細については、

Architectural details を参照してください。

## アーキテクチャのオプションとパフォーマンス

### BDR パフォーマンスの特徴付け

BDR は、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を持つ多数の異なるアーキテクチャで構成できます。

グループは、2 つ以上のノード（サーバー）で構成される BDR グループの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使用して異なるアベイラビリティゾーンにあり、即時のスイッチオーバーと高可用性が提供されます。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。

BDR はすべてのノードですべての書き込みを再生する必要があるため、ほとんどのテーブルがレプリケートされる場合、BDR 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向上はありません。BDR 書き込みは一般に、SQL を介した 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、パフォーマンスをいくらか向上させることができます。読み取りスループットは通常、ノード数に比例して増加します。

次のアーキテクチャを使用できます。

- マルチマスター/シングルグループ
- BDR AlwaysOn

最も単純なアーキテクチャは、グループを 1 つだけにすることです。

### 1 つのグループでの BDR マルチマスター

デフォルトでは、BDR 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だけで済みます。読み取り専用クラスターでは、1 つのノードのパフォーマンスはノードの数に影響されません。したがって、ノードを追加すると、可能な合計 SELECT スループットが直線的に増加します。

INSERT、UPDATE および DELETE（DML）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 適用のオーバーヘッドは元の実行よりも少ないため、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単一ノードよりも多くの TPS を処理できます。

競合処理には、スループットを低下させるコストがあります。スループットは、アプリケーションが実際に表示する競合の量によって異なります。競合が非常に少ないアプリケーションは、単一のノードよりも優れたパフォーマンスを発揮します。競合の多いアプリケーションは、単一の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致しています。彼らは BDR に固有のものではありません。

Eager Replication は競合を回避できますが、本質的にコストが高くなります。

レプリケーションの遅延が最小限に抑えられるように、変更はすべてのノードに同時に送信されます。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、ノード

が追加されるたびにピーク TPS がわずかに低下します。

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場合、このリソースはレプリケーションを制約し、レプリケーションの遅延に影響を与える可能性があります。

## BDR AlwaysOn

AlwaysOn アーキテクチャは、2つの異なるリージョンの2つのグループから構築されています。各グループは HA と IS を提供しますが、一緒に災害復旧 (DR) も提供します。そのため、このアーキテクチャを非常に高可用性の AlwaysOn と呼びます。

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

1つのノードがメインアプリケーションのターゲットです。他のすべてのノードはシャドウノード（または「読み書きレプリカ」）と呼ばれ、必要に応じて引き継ぐのを待っています。ノードが接続を失った場合、すぐにシャドウノードに切り替えて処理を続行します。グループに障害が発生した場合、他のグループに切り替えることができます。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。

主に1つのノードのみに書き込むため、間の競合の可能性はほぼゼロに削減されます。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。

CAMO はローカルグループでは積極的な Eager Replication であり、他のグループに関しては怠惰です。

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合わせてレプリケーションの CPU オーバーヘッドが制限され、他のグループへの帯域幅が最小限に抑えられます。

## 展開

BDR は、TPAexec またはテクニカルサポートによって承認された構成管理アプローチと展開アーキテクチャを使用して、少数の正常な構成の1つで展開されることを目的としています。

手動展開は推奨されておらず、サポートされていない場合があります。

アーキテクチャの TPAexec Architecture User Manual を参照してください。

ログメッセージとドキュメントは現在英語でのみ提供されています。

## 時計とタイムゾーン

BDR 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いるため、真にワールドワイドなデータベースクラスターが可能です。個々のサーバーを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て比較できるように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の時計を同期します。

クロックの同期は、他のいくつかのソリューションと同様に、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。BDR は存在するスキューを報告および管理するためのコントロールを提供します。

*Column-level conflict detection* で説明されているように、

BDR は行バージョンの競合検出も提供します。

## Application use

ユーザーの観点からアプリケーションについて学びます。

## アプリケーションの動作

BDR 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。

BDR は、デフォルトで、ソースノードから他のノードへの INSERT、UPDATE、DELETE および TRUNCATE 操作からのすべての変更を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、オリジンで何が発生したかに応じて、挿入または更新を送信します。更新または削除がゼロ行に影響する場合、変更は送信されません。

INSERT は前提条件なしで複製できます。

他のノードで複製する更新と削除については、影響を受ける一意の行を特定する必要があります。BDR では、テーブルに PRIMARY KEY、UNIQUE 制約、または特定の列に定義された明示的な REPLICATION IDENTITY 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、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 REPLICATION 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、遅延ではなく、式または WHERE 句を持たない最初の非一意のインデックスを使用します。それ以外の場合は、シーケンシャルスキャンが使用されます。

定義されたレプリケーション ID がなくても TRUNCATE を使用できます。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捨てる場合は注意してください。切り捨て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、オリジンで切り捨てられたのと同じグループのテーブルを切り捨てます。詳細と例については、

*Replication sets* を参照してください。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、これは正しく機能します。ただし、サブスクライバーで切り捨てるテーブルに、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切り捨てアクションを適用すると失敗します。



INSERT、UPDATE、および DELETE コマンドによって暗黙的に取得される行レベルのロックは、変更が加えられると複製されます。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 ロック (SELECT ... FOR UPDATE/FOR SHARE) は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保存された情報は、他のノードにレプリケートされません。SERIALIZABLE のトランザクション分離レベルは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間でシリアル化されません。

**DML が複数のノードで同時に実行され、非同期レプリケーションで実行すると競合が発生する可能性があります。これらは処理するか、回避する必要があります。**

**競合の監視** で説明されているさまざまな回避メカニズムが可能です。

:ref:`bdr.sequences` <bdr.sequences>` で説明されている、シーケンスには特別な処理が必要です。

BYTEA 列のバイナリデータは正常にレプリケートされるため、最大 1GB のデータの「ブロブ」が許可されます。PostgreSQL の「ラージオブジェクト」機能の使用は、BDR でサポートされていません。

ルールは起点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、適用中には実行されません。

実表から実表へのみレプリケーションできます。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、または外部テーブルではなく、テーブルである必要があります。実テーブル以外のテーブルをレプリケートしようとすると、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、オリジンの実表に解決されてから、ターゲットの同じ実表に適用されます。

BDR はパーティションテーブルを透過的にサポートします。つまり、パーティションテーブルをレプリケーションセットに追加し、パーティションを含む変更をダウンストリームに複製します。

デフォルトでは、トリガーはオリジンノードでのみ実行されます。たとえば、INSERT トリガーはオリジンノードで実行され、ターゲットノードに変更を適用しても無視されます。ALTER TABLE ... ENABLE ALWAYS TRIGGER を使用して、実行時に起点ノードとターゲットで複製されるとき（「適用時」）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のみに実行します：ALTER TABLE ... ENABLE REPLICA TRIGGER。

一部の種類のトリガーは、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)
- カラムごとの UPDATE トリガー (UPDATE OF column\_name [, ...])

BDR レプリケーションの適用は、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関数は、他の search\_path 設定を想定できますが、適用時に実行されます。これを防ぐには、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、常にオブジェクトへの完全修飾参照 (schema.objectname など) を使用するか、影響を受ける関数に ALTER FUNCTION ... SET search\_path = ... を使用して関数の検索パスを設定します。

BDR 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はすべてのノードで使用でき、デフォルトの照合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値検索を使用してレプリカ ID 値を見つけるため、一致しない照合修飾子で一意的なインデックスが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た場合、行フィルターは照合の違いの影響を受ける可能性があります。

PostgreSQL での非常に長い「トースト」データの BDR 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でも異なる可能性があります、問題は発生しません。

レプリカ ID 列が外部としてマークされている場合、BDR は正しく動作しません。

PostgreSQL は、揮発性関数を含む CHECK() 制約を許可します。BDR は適用時に CHECK() 制約を再実行するため、以前と同じ結果が返されない後続の再実行によりデータが発散します。

BDR は外部キーの使用を制限しません。FK のカスケードが許可されます。

### レプリケートされないステートメント

次のユーザコマンドはいずれも BDR によって複製されないため、それらの効果はローカル/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)
- パラメータコマンド (SET、SHOW)
- 制約操作 (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()` 関数は複製されないため、フェールオーバーが発生した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、フェールオーバーの場合に通知を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も当てはまり、BDR はまだこれを改善していません。CAMO および Eager Replication オプションでは、NOTIFY SQL コマンドまたは `pg_notify()` ファンクションは許可されません。

### DML および DDL レプリケーション

BDR 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る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複製されることを意味し、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ありません。

DDL レプリケーションの動作は DML とは異なります。DDL の場合、BDR はステートメントを複製し、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複製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、ステートメントは実行のために他のノードに送信されます。詳細については、

[DML および DDL レプリケーション](#) を参照してください。



BDR は、混合した DML ステートメントと DDL ステートメントが同じトランザクションでも正しく機能するように機能します。

### 異なるリリースレベル間の複製

BDR は、PostgreSQL のメジャーバージョンが異なる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メジャーバージョンをアップグレードできるように設計されています。

BDR は、BDR ソフトウェアのバージョンが異なるノード間で複製するようにも設計されています。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行え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に参加することはできますが、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。これはエラーを返します。

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、

[Release Notes](#) を参照してください。

### 違いのあるノード間の複製

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。

[DML および DDL レプリケーション](#) で説明されているように、これを手動で制御でき、それを使用してノード全体のデータベーススキーマ間の違いを作成できます。

BDR は、ノード間のわずかな違いで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行え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許可され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め、パーティションテーブルに複製する通常のテーブルであるソースなど、異なるパーティション定義を持つテーブル間で複製できます。動的パーティションルーティングは適用時に実行する必要がないため、ソースとターゲットでパーティション化定義が同じである場合、より高速になります。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、すべての列が複製されます。BDR は、列名に基づいてデータ列を複製します。列の名前は同じでデータ型が異なる場合、それを可能にするキャストが定義されている場合、ソース型からターゲット型にキャストしようとします。

BDR は、列の数が異なるテーブル間の複製をサポートしています。

ターゲットにソースからの列が欠落している場合、BDR は `target_column_missing conflict` 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。これにより、NULL 以外の値が到着するとエラーがスローされます。または、`ignore` の競合リゾルバーで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、追加の列を黙って無視します。

ターゲットにソースレコードにない追加の列がある場合、BDR 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。追加の列にデフォルト、NULL (NULL

可能な場合) またはデフォルトの式がある場合、レプリケーションは続行されますが、そうでない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。

変換トリガーを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、ソースの行をターゲットに適用できない場合は失敗します。ここで行フィルターが役立ちます。

あるスキーマからよりリラックスしたスキーマにデータを複製しても、エラーは発生しません。スキーマからより制限の厳しいスキーマへのデータの複製は、潜在的な障害の原因になる可能性があります。これを解決する正しい方法は、よりリラックスした側に制約を配置して、悪いデータを入力できないようにすることです。そうすれば、レプリケーションによって不正なデータが届くことはないため、より制限の厳しいスキーマへの変換が失敗することはありません。たとえば、あるスキーマに TEXT 型の列があり、別のスキーマが同じ列を XML として定義している場合、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。

各ノードで異なるインデックスを持つ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複製されます。ノードのサブセットまたはローカルにのみインデックスを作成する方法を指定するには、[DML および DDL レプリケーション](#) を参照してください。

fillfactor や toast\_tuple\_target などの記憶域パラメーターは、テーブルのノード間で問題なく異なります。その例外は、テーブルの記憶域パラメーター user\_catalog\_table の値は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/ロールが所有する必要があります。詳細については、[Security and roles](#) を参照してください。

デフォルトでは、ロールへの変更は各ノードに複製されますが、ロールは各ノードで接続用に異なるパスワードを持つことができます。ノードのサブセットまたはローカルでのみロールのパスワードを変更する方法を指定するには、[DML および DDL レプリケーション](#) を参照してください。

## 違いのあるノード間の比較

LiveCompare は、BDRBDR に対して、データベース上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 set で設定できます。これにより、クラスター内の個別のノードに関連するすべての DSN を明確にする必要がなくなります。EDB Postgres 分散クラスターには、接続情報を持つ 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のは初期接続と出力接続のみです。logical\_replication\_mode を設定すると、すべてのノードがどのように通信しているかが示されます。

たとえば、すべての構成はbdrLC.ini という名前の.ini ファイルで行われます。/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 個の進行状況バーが表示されます。N はプロセスの数です。すべてのテーブルのソースが取得されると、1 秒あたりのトランザクション数 (tps) が測定されると時間が表示されます。これは時間をカウントし続け、最後に見積もりを提供し、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。比較後、概要と DML スクリプトが生成されるため、確認できます。DML を適用して、見つかった違いを修正します。

## アプリケーションの一般的なルール

**BDR はレプリカ ID 値を使用して、変更する行を識別します。アプリケーションで同じ一意の識別子を挿入、削除し、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 [ABA problem](#) と呼ばれます。**

BDR は、行が現在の行、最後の行、またはそれよりも古い行であるかを判断できません。

同様に、BDR 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、削除、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。

これらの問題により、アプリケーションが従うべきいくつかの簡単なルールが発生します。

- 行に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を再利用しないでください。
- ドロップされたオブジェクト名の再利用を避けます。

一般的に、これらのルールを破ると、データの異常や発散につなが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ている限り、これらのルールを破ることができますが、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、行の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て正常に再起動できない場合は、数日かかる場合があります。

## タイミングの考慮事項と同期レプリケーション

デフォルトでは非同期であるため、ピアノードは遅れる場合があるため、クライアントは複数の BDR ノードに接続したり、それらを切り替えたりして古いデータを読み取ることができます。

このような古い読み取りを防ぐために、クライアントまたはプロキシに `queue wait function` が提供されます。

Postgres の同期レプリケーション機能は BDR でも利用できます。さらに、BDR は、より同期的なレプリケーションのために複数のバリエーションを提供します。利用可能なすべてのバリエーションとそのさまざまなモードの概要と比較については、

[Durability and performance options](#) を参照してください。

## アプリケーションテスト

他の手法に加えて、次のプログラムを使用して BDR アプリケーションをテストできます。

- *TPAexec*
- *CAMO/Failover* オプションを使用した *pgbench*
- マルチノードアクセスを備えた *isolationtester*

### TPAexec

TPAexec は、EDB Postgres Distributed ベースのものを含みリファレンス TPA アーキテクチャを展開するために EDB が使用するシステムです。

TPAexec には、各リファレンスアーキテクチャのテストスイートが含まれています。また、次のような構文を使用して、TPA クラスターに対して実行するテストのローカルコレクションの作成と管理が簡単になります。

```
tpaexec test mycluster mytest
```

開発者は、アプリケーションの主な予想されるプロパティを検証する独自のマルチノードスイートの TPAexec テストを作成することを強くお勧めします。

### CAMO/Failover オプションを使用した pgbench

EDB Postgres Extended では、ユーザーが CAMO または通常の BDR デプロイメントの使用中にフェイルオーバーテストを実行できるように pgbench が拡張されました。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)

- --retry
retry transactions on failover
```

DSN form には、これらのオプションに加えて、フェイルオーバーのピアノードの接続情報を指定する必要があります。

- `-m camo` または `-m failover` を使用して、pgbench のモードを指定します。 `-m failover` 仕様を使用して、通常の BDR 展開でフェールオーバーをテストできます。
- `--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。

CAMO 環境の例を次に示します。

```
pgbench -m camo -p $node1_port -h $node1_host bdrdemo \  
"host=$node2_host user=postgres port=$node2_port dbname=bdrdemo"
```

このコマンドはカモモードで実行されます。node1 に接続してテストを実行します。node1 への接続が失われた場合、pgbench は node2 に接続します。node2 に照会して、進行中のトランザクションのステータスを取得します。中断された飛行中のトランザクションは、迷彩モードで再試行されます。

フェイルオーバーモードで `--retry` が指定されている場合、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。

## マルチノードアクセスを備えた `isolationtester`

`isolationtester` は、ユーザーがマルチプルのセッションおよびマルチプルのノードでテストを実行できるように拡張されました。これは内部 BDR テストに使用されますが、ユーザアプリケーションのテストで使用することもできます。

```
$ isolationtester \
  --outputdir=./iso_output \
  --create-role=logical \
  --dbname=postgres \
  --server d1=dbname=node1 \
  --server d2=dbname=node2 \
  --server d3=dbname=node3
```

分離テストは、PostgreSQL の同時動作を調べるためのテストです。これらのテストでは、複数の相互作用するトランザクションを実行する必要があります。これには、複数の同時接続を管理する必要があるため、通常の `pg_regress` プログラムを使用してテストすることはできません。「分離」という名前は、元の動機がシリアライズ可能な分離レベルをテストすることであったという事実由来します。他の種類の同時動作のテストも追加されました。

PGXS を外部モジュールとして使用して構築されています。インストール時に、`isolationtester` バイナリファイルを作成します。これは `pg_isolation_regress` によって実行され、同時リグレーションテストを実行し、結果を観察します。

`pg_isolation_regress` は `pg_regress` に似たツールですが、`psql` を使用してテストを実行する代わりに、`isolationtester` を使用します。`pg_regress` と同じコマンドライン引数をすべて受け入れます。複数のホストをパラメーターとして受け入れるように変更されました。次に、ホスト `conninfo` をサーバー名とともに `isolationtester` バイナリに渡します。分離テスターは、これらのサーバー名を仕様ファイルの各セッションで指定された名前と比較し、それぞれのサーバーで指定されたテストを実行します。

トランザクションが重複するテストを定義するには、カスタム構文のテスト仕様ファイルを使用します。新しいテストを追加するには、仕様ファイルを `specs/` サブディレクトリに配置し、予想される出力を `expected/` サブディレクトリに追加し、テストの名前を `makefile` に追加します。

`Isolationtester` は、`libpq` を使用してマルチプルの接続を開き、スペックファイルで指定されたテストを実行するプログラムです。`libpq` 接続文字列は、接続するサーバーとデータベースを指定します。それ以外の場合は、環境変数から派生したデフォルトが使用されます。

仕様は 5 つの部分で構成され、次の順序でテストされます。

`server "<name>"`

これは、セッションを実行するサーバーの名前を定義します。0 個以上のサーバー "`<name>`" 指定があ

ります。名前に対応する conninfo は、isolationtester を実行するコマンドによって提供されます。これは quickstart\_isolationtest.md で説明されています。この部分はオプションです。

```
setup { <SQL> }
```

指定された SQL ブロックは、テストを実行する前に、1つのセッションでのみ実行されます。ここでテストテーブルまたはその他の必要なオブジェクトを作成します。この部分はオプションです。必要に応じて、複数のセットアップブロックが許可されます。それぞれは、指定された順序で個別に実行されます。複数のセットアップブロックを許可する理由は、各ブロックが単一の PQexec サブミッションとして実行され、VACUUM などの一部のステートメントをそのようなブロックで他のステートメントと組み合わせることができないためです。

```
teardown { <SQL> }
```

テストが終了した後、ティアダウン SQL ブロックが 1 回実行されます。これを使用して、セットアップで作成されたテストテーブルを削除するなど、次の置換に備えてクリーンアップします。この部分はオプションです。

```
session "<name>"
```

通常、スペックファイルには複数の「セッション」部分があります。各セッションは、独自の接続で実行されます。セッションパートは、セットアップ、ティアダウン、および 1 つ以上の「ステップ」の 3 つの部分で構成されています。セッションごとのセットアップとティアダウンの部分は、テストごとのセットアップとティアダウンと同じ構文ですが、セッションごとに実行されます。セットアップ部分には、通常、トランザクションを開始するための BEGIN コマンドが含まれています。

セッションパートも connect\_to 指定で構成されます。これは、このセッションが実行されるサーバーを示す先頭に指定されたサーバー名を指します。

```
connect_to "<name>"
```

各ステップには次の構文があります。

```
step "<name>" { <SQL> }
```

<name> はこのステップを識別する名前であり、SQL はステップで実行される SQL ステートメント（またはセミコロンで区切られたステートメント）です。ステップ名は、スペックファイル全体で一意である必要があります。

```
permutation "<step name>"
```

置換行は、その順序で実行されるステップのリストを指定します。置換行はいくつでも表示できます。置換行が指定されていない場合、テストプログラムは各セッションのステップの可能な順序をすべて自動的に生成します（1 つのセッションのステップを順番に実行します）。手動で指定された「置換」行のステップのリストは、実際には使用可能なステップの置換である必要はありません。たとえば、一部の手順を複数回繰り返したり、他の手順を省略したりできます。



で始まる行はコメントです。

セッションステップの順列（これらがスペックファイルで手動で指定されたか、自動生成されたかにかかわらず）に対して、分離テスターが実行されます。

#### 1. メインセットアップ部

1. セッションごとのセットアップパーツ
2. 選択されたセッションのステップ

#### 1. セッションごとのティアダウン

#### 1. メインの分解スクリプト

選択された各ステップは、そのセッションに関連付けられた接続に送信されます。

前提条件となるすべての make コマンドを実行した BDR 環境で分離テストを実行するには：

1. `make isolationcheck-install` を実行して `isolationtester` サブモジュールをインストールします。
2. `bdr-private` リポジトリから次のコマンドのいずれかを使用して、分離リグレーションテストを実行できます。

```
make isolationcheck-installcheck make isolationcheck-makecheck
```

`isolationcheck-installcheck` を実行するには、2 つ以上の postgresql サーバーを実行している必要があります。各サーバーの `conninfo` を BDR makefile の `pg_isolation_regress` に渡します。例：`pg_isolation_regress --server 'd1=host=myhost dbname=mydb port=5434' --server 'd2=host=myhost1 dbname=mydb port=5432'`

次に、`bdr-private/` リポジトリの `specs/isolation` ディレクトリにテストを含む `.spec` ファイルを追加します。`nbdr-private/` リポジトリの `expected/isolation` ディレクトリに `.out` ファイルを追加します。

次に、`make isolationcheck-installcheck` を実行します

`Isolationcheck-makecheck` は現在、複数のデータベース間で BDR を設定することにより、単一のインスタンスでの分離テストの実行をサポートしています。

次のように、適切なデータベース名と `bdr` インスタンスの `conninfo` を BDR makefile の `pg_isolation_regress` に渡す必要があります。

次に、`make isolationcheck-makecheck` を実行します

各ステップには、さらにアクションが実行されるまでブロックするコマンドを含めることができます（ほとんどの場合、他のセッションがブロックを解除するか、デッドロックを発生させます）。この機能を使用するテストでは、有効な順列、つまり、ブロックされたセッションがコマンドを実行しないことを手動で指定する必要があります。テストがそのルールに従わない場合、`isolationtester` は 300 秒後にキャンセルします。キャンセルが機能しない場合、`isolationtester` は 375 秒の待機時間後にクリーンに終了します。無効な順列のテストは避けてください。分離テストの実行に非常に時間がかかり、有用なテスト目的を果たさないためです。

`isolationtester` は、`pg_locks` ビューで待機中と表示されているかどうかを確認することにより、コマンドがブロックされたことを認識します。したがって、重量ロックのブロックのみが検出されます。

## パフォーマンスのテストとチューニング

BDR を使用すると、複数のマスター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、パフォーマンスが低下します。

まず、別のノードからの変更を再生するには CPU コストと I/O コストがあり、WAL レコードを生成します。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、リソースの使用は通常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、変更を再生すると、ローカルのワークロードに対して競合が発生する可能性のあるテーブルレベルおよび行レベルのロックが保持されます。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) 機能により、同時更新でも正しい答えが得られますが、通常のロックオーバーヘッドは削除されません。ロックの競合が発生した場合は、更新の競合を回避するか、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る行は、そのトランザクションのパフォーマンスにボトルネックを引き起こします。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の配慮が必要です。

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。pgbench 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。

BDR の実行が遅い場合は、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、pgbench 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、ベースラインの数値を取得します。
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、実稼働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。これは、マルチプルのノードに移動した結果を示しています。
4. これら 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の影響をプロットできるようにします。
5. テストがレプリケーションの遅延を考慮に入れるのに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の遅延が増大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。

## 適合性の評価

BDR は PostgreSQL と互換性がありますが、すべての PostgreSQL アプリケーションが分散データベースでの使用に適しているわけではありません。ほとんどのアプリケーションは既に、または簡単に変更して BDR に準拠しています。アプリケーションを BDR 対応のセットアップにポイントできる評価アクティビティを行うことができます。BDR は、評価期間中に設定できるいくつかのノブを提供します。これらは、BDR 対応環境でのアプリケーションの適合性を判断するプロセスに役立ちます。



## プライマリキー/レプリカ ID の更新の評価

BDR は現在、UPDATE 操作によって PRIMARY KEY が変更された競合解決を実行できません。主キーは更新できますが、既存の値との競合が発生しないことを確認する必要があります。

BDR は、テーブルの主キー/レプリカ ID が更新操作の対象となる頻度を評価するための次の構成パラメーターを提供します。

これらの構成パラメーターは評価にのみ使用してください。単一ノードの BDR インスタンスで使用できますが、2 つ以上のノードが相互に複製する実稼働 EDB Postgres 分散クラスターでは使用しないでください。実際、評価パラメーターのいずれかが IGNORE 以外に設定されている場合、ノードが起動しないか、新しいノードがクラスターに参加しない場合があります。

```
bdr.assess_update_replica_identity = IGNORE (default) | LOG | WARNING | ERROR
```

評価期間中にこのパラメーターを有効にすることで、行のキー/レプリカの ID 値への更新をログに記録できます。必要に応じて、このような更新を潜在的にブロックすることもできます。例：

```
CREATE TABLE public.test(g int primary key, h int);
INSERT INTO test VALUES (1, 1);

SET bdr.assess_update_replica_identity TO error;
UPDATE test SET g = 4 WHERE g = 1;
ERROR: bdr_assess: update of key/replica identity of table public.test
```

適用ワーカープロセスは、このパラメーターの設定を常に見捨てます。

## テーブルまたは SELECT クエリでの LOCK の使用の評価

BDR ライタプロセスは通常のユーザーセッションとほとんど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に従います。これにより、BDR ライタプロセスがユーザトランザクションまたは相互に保持されているロックで待機する場合があります。

BDR は、アプリケーションが明示的なロックを取得しているかどうかを評価するための次の構成パラメーターを提供します。

```
bdr.assess_lock_statement = IGNORE (default) | LOG | WARNING | ERROR
```

追跡できる 2 つのタイプのロックは次のとおりです。

- ユーザセッションによる明示的なテーブルレベルロック (LOCK TABLE ...)
- ユーザーセッションによる明示的な行レベルロック (SELECT ... FOR UPDATE/FOR SHARE)

評価期間中にこのパラメーターを有効にすることで、このような明示的なロックアクティビティを追跡（またはブロック）できます。例：

```
CREATE TABLE public.test(g int primary key, h int);
INSERT INTO test VALUES (1, 1);
```

(次のページに続く)

```
SET bdr.assess_lock_statement TO error;
SELECT * FROM test FOR UPDATE;
ERROR:  bdr_assess: "SELECT FOR UPDATE" invoked on a BDR node

SELECT * FROM test FOR SHARE;
ERROR:  bdr_assess: "SELECT FOR SHARE" invoked on a BDR node

SET bdr.assess_lock_statement TO warning;
LOCK TABLE test IN ACCESS SHARE MODE;
WARNING:  bdr_assess: "LOCK STATEMENT" invoked on a BDR node
```

## PostgreSQL configuration for BDR

いくつかの PostgreSQL 構成パラメーターは BDR ノードに影響します。これらのパラメーターは各ノードで個別に設定できますが、一般的にはお勧めしません。

## BDR の PostgreSQL 設定

BDR を正しく実行するには、次の PostgreSQL 設定が必要です。

- `wal_level` — BDR 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` — `bdr` を含める必要がありますが、必要に応じてその前後に他のエントリを含めることができます。ただし、`pglogical` は含めないでください。
- `track_commit_timestamp` — 競合を解決して、競合する各行のタイムスタンプを取得するには、`on` に設定する必要があります。

BDR 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。値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` — 論理デコードで使用するメモリバッファサイズ。これより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、もっと大きく設定できます。
- `max_worker_processes` — BDR はレプリケーションとメンテナンスの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ロットが必要です。各データベースの BDR 最小 BDR 数の式は次のとおりです。BDR グループ内のピアノードごとに有効になっているライターごとに BDR グループからノードを削除するときに、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` — すべてのピアノードごとに 2 つ必要です。
- `max_replication_slots` — `max_wal_senders` と同じ。
- `wal_sender_timeout` および `wal_receiver_timeout` — ノードが CAMO パートナーを切断または再接続と見なす速度を決定します。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、BDR には N 個のスロットと WAL 送信者が必要です。同期中、BDR は別の N - 1 個のスロットと WAL 送信者を一時的に使用するため、この時折のピーク需要に合わせてパラメーターを十分に高く設定してください。

パラレル適用をオンにすると、スロット数を式\*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起点の最大数を設定し、パラレル適用の一部の機能はライターごとに追加のオリジンを使用するためです。

**デコードワーカー** が有効になっている場合、このプロセスには BDR 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。

これらの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。`max_worker_processes`、`max_wal_senders`、`max_replication_slots`。

アプリケーションでこれらのパラメーターを設定することもできます。詳細は *Durability and performance options* を参照してください。

- `synchronous_commit` —BDR 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。

**物理レプリケーション** と同様に。

- `synchronous_standby_names` —上記と同じ。

## BDR 固有の設定

BDR 固有の構成設定を設定することもできます。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` —新しく作成された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 *sequence kind* を設定します。デフォルト値は `distributed` で、これは、`snowflakeid` が `int8` シーケンス（つまり `bigserial`）および `int4`（つまり `serial`）および `int2` 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` —新しく作成されたテーブルの `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。

- `DEFAULT` —主キーの列の古い値を記録します（これは PostgreSQL のデフォルトの動作です）。 - `FULL` —行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を記録しません。

詳細については、 [PostgreSQL documentation](#) を参照してください。

BDR は、 `PRIMARY KEY` または `UNIQUE` 制約のないテーブルで `UPDATE` および `DELETE` 操作を複製できません。例外は、テーブル固有の構成または `bdr.default_replica_identity` によって、テーブルのレプリカ ID が `FULL` である場合です。

`bdr.default_replica_identity` が `DEFAULT` であり、テーブルに `UNIQUE` 制約がある場合、自動的に `REPLICA IDENTITY` として選択されません。上記のように、テーブルの作成時または作成後に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、書き込まれる WAL の量とテーブルのネットワークで複製されるデータの量が増加します。

- `bdr.ddl_replication` —ノード間で DDL を自動的に複製します（デフォルトは `on`）。

このパラメーターは、`bdr_superuser` またはスーパーユーザーロールのみが設定できます。

`bdr.ddl_replication = off` を使用して DDL を実行するか、BDR 管理機能呼び出すと、管理者が介入するまでレプリケーションが停止する状況が発生する可能性があります。詳細は [DML および DDL レプリケーション](#) 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、この設定が原因でキャプチャされた DDL コマンドまたは BDR レプリケーションファクションのレプリケーションが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` —ノード間で `ROLE` コマンドを自動的に複製します（デフォルトは `on`）。このパラメーターを設定できるのはスーパーユーザーのみです。この設定は、`bdr.ddl_replication` がオンになっている場合にのみ機能します。

外部メソッドを使用せずにこれをオフにして、すべてのノードでロールが同期され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された DDL がレプリケーションを中断する場合があります。

詳細は [ロール操作ステートメント](#) を参照してください。

- `bdr.ddl_locking` —DDL のグローバルロックの動作モードを構成します。

このパラメーターは、`bdr_superuser` またはスーパーユーザーロールのみが設定できます。

可能なオプションは次のとおりです。

- オフ DDL 操作にグローバルロックを使用しないでください。 - `on` -すべての DDL 操作にグローバルロックを使用します。 - `dml` グローバルロックは、リレーションのグローバル DML ロックを取得して書き込みを防止する必要がある DDL 操作にのみ使用します。

`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、この設定が原因でグローバルロック手順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 2 つの WARNING メッセージが表示されるのは正常です。1 つは DML ロックをスキップするため、もう 1 つは DDL ロックをスキップするためです。

- `bdr.truncate_locking` — デフォルトでは `False` です。この構成オプションは TRUNCATE コマンドのロック動作を設定します。TRUNCATE が (`true` の場合) `bdr.ddl_locking` 設定に従うかどうかを判断します。

## グローバルロック

- `bdr.ddl_locking` — 上記で説明しました。
- `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 (デフォルトは 1000)。Postgres サーバーの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容時間を設定します (デフォルトは 10 分)。ゼロの値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容期間を設定します (デフォルトは 60 分)。ゼロの値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しているトランザクションの最大許容アイドル時間を設定します (デフォルトは 10 分)。ゼロの値はこのタイムアウトを無効にします。
- `bdr.predictive_checks` — 予測チェックのログレベル (現在、グローバルロックでのみ使用されます)。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の状態の早期の検証です。タイムアウトを待つのではなく、早期に失敗するための操作に使用できます。グローバルロックの用語では、BDR は、十分なノードが接続されており、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合理的な遅延制限があることを確認します。

## ノード管理

- `bdr.replay_progress_frequency` — クラスターの残りの部分にレプリケーション位置情報を送信する間隔 (デフォルトは 1 分)。
- `bdr.standby_slot_names` — これらのスロットは、他のスロットよりも前にレプリケーションの変更を受信して確認する必要があります。この設定は、主にフェイルオーバーにフィジカルスタンバイを使用する場合、またはサブスクリाइブ専用ノードを使用する場合に役立ちます。

## 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（BDR では、グループの `bdr.alter_node_group_config` で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あたりのライターの最大数（上記の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複製します（デフォルトは `on`）。

これをオフにすると、BDR 全体がローカルのみになります。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` またはスーパーユーザーロールのみが設定できます。

このパラメーターは、`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ー処理されます。

**警告:** `bdr.xact_replication` をオフにすると、ノード間のデータの一貫性が失われます。ノード間のデータの相違から回復する場合、またはレプリケーションを続行するために単一ノードでの変更が必要なレプリケーションの状況でのみ使用します。ご自身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するオプション。

`bdr_superuser` または PostgreSQL スーパーユーザーが必要です。

**警告:** 通常は安全と見なされないコマンドにより、一貫性のない結果が生成されるか、レプリケーションが完全に中断される可能性があります。ご自身の責任で使用してください。

- `bdr.batch_inserts` —単一のトランザクションで 1 つのテーブルへの連続した挿入の数は、そのテーブルの挿入のバッチ処理をオンにします。

このオプションにより、大規模なデータのレプリケーションは、一連の挿入ではなく、内部的に `COPY` としてロードされます。ノードジョイン時の初期データをコピーする方法でもあります。

- `bdr.maximum_clock_skew`

このオプションは、`bdr.maximum_clock_skew_action` をトリガーする前の受信トランザクションのコミットタイムスタンプとサブスクライバーの現在の時刻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かどうかを確認します。そうであり、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、クロックスキューを無視することを意味します（チェックはオフになっています）。全サーバーの時計を同期する場合は `0` が有効です。トランザクションを再生し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_clock_skew_action`

これは、`bdr.maximum_clock_skew` よりも高いクロックスキューが検出された場合のアクションを指定します。

このオプションには2つの値があります。

- **WARN** —この事実に関する警告をログに記録します。サーバーログのフラッディングを防ぐために、警告は最大で1分に1回（デフォルト）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るまで待ちます。
- `bdr.accept_connections` —BDR への接続を有効または無効にするオプション。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.standby_slot_names`

このオプションは通常、フェールオーバー構成で使用され、この BDR ノードのフェールオーバー候補のストリーミング物理レプリカがサブスライバーに表示される前にすべての変更を受信してフラッシュします。これにより、プロバイダーのスタンバイへのフェールオーバー時にコミットが消えないことが保証されます。

コンマ区切りの `bdr.standby_slot_names` リストに名前がリストされているレプリケーションスロットは、BDR ノードの `walsender` によって特別に扱われます。

BDR の論理レプリケーションウォルセnderは、ノードが他の BDR レプリケーションクライアントに送信する前に、すべてのローカル変更が `bdr.standby_slot_names` のレプリケーションスロットに送信されてフラッシュされるようにします。効果的に、スロットの名前付けリストと他のすべてのレプリケーションクライアントの間に同期レプリケーションバリアを提供します。

`bdr.standby_slot_names` には任意のレプリケーションスロットをリストできます。論理スロットと物理スロットの両方が機能しますが、通常は物理スロットに使用されます。

このセーフガードがないと、サブスライバーがコミットを受信し、フェールオーバー候補がまだそれを受け取っていないために、フェールオーバー時にプロバイダーから消えるという2つの異常が発生する可能性があります。

- 1人以上のサブスライバーの場合、サブスライバーは変更を適用した可能性があります、新しいプロバイダーは受け取った変更と競合する新しいトランザクションを実行する可能性があります。
- 2人以上のサブスライバーの場合、フェールオーバー時に、すべてのサブスライバーが変更を適用していません。コミットを受け取っていないサブスライバーにはそれを取得する方法がないため、サブスライバーの状態は一貫性がなく調整不能です。

`bdr.standby_slot_names` をデザインで設定すると、リストされているノードの必要な数が追いついていない場合、そこにリストされていない他のサブスライバーがプロバイダーよりも遅れます。したがって、モニタリングは不可欠です。

`bdr.standby_slot_names` が役立つ別のユースケースは、サブスクライバー専用ノードを使用して、通常の BDR ノードよりも前に移動しないようにする場合です。これを行うには、`bdr.standby_slots_min_confirmed` を少なくとも 1 に設定して、すべての通常の BDR ピアノードの論理スロットをリストします。

#### **`bdr.standby_slots_min_confirmed`**

BDR サブスクライバーにデータを送信する前に確認する必要がある `bdr.standby_slot_names` の数を制御します。

#### **`bdr.writer_input_queue_size`**

このオプションは、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ー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、許可される最大サイズは 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`**

このオプションは、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ー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、比較的小さなサイズのキューで十分です。デフォルトは 32 KB で、許可される最大サイズは 1 MB 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_delay`**

レート制限 BDR バックグラウンドワーカーの起動は、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します。エラーが繰り返されると、最大 `bdr.max_worker_backoff_delay` までジッターが追加されると、バックオフが指数関数的に増加します。

時間単位のスフィックスがサポートされています。

---

**注釈:** 現在、この設定はレシーバーワーカーにのみ影響します。つまり、主にサブスクリプションがエラーまたは接続エラー時に再接続を試行する速度に影響します。

---

`bdr.min_worker_backoff_delay` のデフォルトは 1 秒です。`bdr.max_worker_backoff_delay` の場合は 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、管理者がすべてのバックオフ間隔を強制的に終了できるように、`bdr.worker_task_reset_backoff_all()` ファンクションが提供されています。



各タイプのワーカーの最終起動時刻を記憶するために、共有メモリ内の追跡テーブルが維持されます。この追跡テーブルは永続的ではありません。これは、クリーンでないバックエンドが終了した後のクラッシュ復旧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 `:ref:`bdr.worker_tasks`<bdr.worker_tasks>` を使用してこの状態を検査できるため、管理者は現在有効な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、ワーカーはタスクごとに分類されます。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ー ID、拡張機能ライブラリの名前と関数名、拡張機能が提供するワーカー名、および同期ライターのリモートリレーションシップ ID で構成されます。NULL は、特定の分類子が適用されない場合に使用されます。たとえば、マネージャーのワーカーにはサブスクリプション ID がなく、レシーバーにはライター ID がありません。

## CRDT

- **bdr.crdt\_raw\_value** —

[CRDT データ型を使用した列の競合の処理](#) の出力フォーマットを設定します。デフォルトの出力（この設定が `off`

の場合）は、ベースの CRDT タイプ（たとえば、`crdt_pncounter` の `bigint`）のみを返します。on に設定すると、戻り値は CRDT 値の完全な表現を表します。たとえば、複数のノードからの状態を含めることができます。

## 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 トランザクションによるクラスター全体の同時準備トランザクションの最大数に対処するには、十分に高く設定する必要があります。制限を超えると、ノードでローカル 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 トランザクションが防止されます。これは、Postgres サーバーの起動時にのみ設定できます。

## Eager Replication

- **bdr.commit\_scope** —コミットスコープを `global` に設定すると、`eager all node replication`（デフォルトは `local`）が有効になります。
- **bdr.global\_commit\_timeout** —グローバル 2 フェーズコミットの両方のステージ（デフォルトは 60 秒）と、CAMO パートナーを待機する時間の制限としての、コミットフェーズでの CAMO で保護されたトランザクションのタイムアウト。

## Commit At Most Once

- `bdr.enable_camo` —CAMO 機能を有効にして制御するために使用されます。デフォルトは `off` です。これを `remote_write`、`remote_commit_async`、または `remote_commit_flush` に設定することで、CAMO をトランザクションごとにオンにできます。下位互換性のために、値 `on`、`true`、および `1` は最も安全な `remote_commit_flush` モードを設定しますが、`false` または `0` も CAMO を無効にします。
- `bdr.standby_dsn` —ローカルノードのクラッシュ後に変更された場合、接続文字列 (DSN) の手動オーバーライドが CAMO パートナーに到達することを許可します。通常は設定されていません。Postgres サーバーの起動時にのみ設定できます。
- `bdr.camo_local_mode_delay` —トランザクションを確認する必要がある CAMO パートナーで通常発生するオーバーヘッドをエミュレートするために CAMO のローカルモードで適用されるコミット遅延。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に警告を発します。これはデフォルトで有効になっています。十分な情報を得たユーザーは、これを無効にして、ログに記録される警告の量を減らすことができます。
- `synchronous_replication_availability` —CAMO パートナーが切断された後にノードが続行してコミットできるようにすることにより、可用性を高めるためにオプションで `async` にすることができます。デフォルト値の `wait` では、ノードは無期限に待機し、CAMO パートナーが再接続して確認を送信した後にのみコミットします。

## トランザクションストリーミング

- `bdr.default_streaming_mode` —サブスクライバーノードによるトランザクションストリーミングを制御するために使用されます。許可される値は次のとおりです。`off`、`writer`、`file`、および `auto`。デフォルトは `auto` です。`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。他のいずれか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らをサポートしており、グループレベルで構成されている場合に提供します。詳細については、

[トランザクションストリーミング](#) を参照してください。

## ラグ制御

- `bdr.lag_control_max_commit_delay` —許容できる最大許容ポストコミット遅延 (ミリ秒単位)。
- `bdr.lag_control_max_lag_size` —許容できる最大ラグサイズ (キロバイト)。
- `bdr.lag_control_max_lag_time` —許容できる最大遅延時間 (ミリ秒)。
- `bdr.lag_control_min_conforming_nodes` —許容可能なラグメジャーを下回るために必要なノードの最小数。
- `bdr.lag_control_commit_delay_adjust` —コミット遅延のマイクロ調整は、最大コミット遅延時間の割合として測定されます。デフォルト値の 0.01% では、最大コミット遅延に到達するには正味 100 回のインクリメントが必要です。

- `bdr.lag_control_sample_interval` —遅延サンプルとコミット遅延のマイクロ調整間の最小時間（ミリ秒）。
- `bdr.lag_control_commit_delay_start` —許容可能なラグメジャーの割合として表される、コミット遅延の増分の適用が開始されるラグのしきい値。デフォルト値の 1.0%では、許容可能なラグ測定値を超え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。

より小さい割合を設定することで、より早く「ラグ曲線を曲げ」て、許容可能なラグメジャーで漸近することにより、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

- `snapshot_timestamp` —  
タイムスタンプベースのスナップショット の使用をオンにし、使用するタイムスタンプを設定します。
- `bdr.timestamp_snapshot_keep` —タイムスタンプベースのスナップショットで使用する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、つまり過去のスナップショットを保持しません）。

## モニタリングとロギング

- `bdr.debug_level` —BDR 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 BDR デバッグ出力を見たい場合は `bdr.debug_level = 'log'` を設定してください。
- `bdr.trace_level` —上記と同様に、これは BDR 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。これは、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可能性があるため、ディスク領域の枯渇を防ぐためのログフィルタリング、アーカイブ、およびローテーションの計画がない限り、本番環境で長期間有効にしないでください。

- `bdr.track_subscription_apply` —各サブスクリプションの適用統計を追跡します。
- `bdr.track_relation_apply` —各リレーションの適用統計を追跡します。
- `bdr.track_apply_lock_timing` —リレーションの統計を追跡するときにロックのタイミングを追跡します。

## 内部

- **`bdr.raft_keep_min_entries`** &mdash; ログコンパクションを行うときに Raft ログに保持するエントリの最小数 (デフォルト 100)。値 0 はログの圧縮を無効にします。これは、Postgres サーバーの起動時にのみ設定できます。.. **Warning ::**  
ログの圧縮が無効になっている場合、ログのサイズは永久に大きくなります。
- **`bdr.raft_response_timeout`** &mdash; ネットワーク障害に対応するために、実装された Raft コンセンサスプロトコルは一定の時間が経過するとリクエストをタイムアウトします。このタイムアウトのデフォルトは 30 秒です。
- **`bdr.raft_log_min_apply_duration`** &mdash; ステートマシンを前進させるために、Raft はエントリをその内部ログに追加します。通常の操作中、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。このパラメーターのデフォルト値は 3000 ミリ秒です。
- **`bdr.raft_log_min_message_duration`** &mdash; コンセンサスリクエストをログに記録する場合。bdr コンセンサスリクエストのラウンドトリップ時間を測定し、時間がこのパラメーターを超える場合は INFO メッセージをログに記録します。このパラメーターのデフォルト値は 5000 ミリ秒です。
- **`bdr.raft_group_max_connections`** &mdash; Postgres サーバーのすべての BDR グループにわたる接続の最大数。これらの接続は、グループのノード間で bdr コンセンサス要求を運びます。このパラメーターのデフォルト値は 100 接続です。Postgres サーバーの起動時にのみ設定できます。
- **`bdr.backwards_compatibility`** &mdash; `bdr.bdr_version_num`、たとえば、30618 で使用されるのと同じ数値形式で、下位互換性があるバージョンを指定します。一般的に望ましくない効果がある場合でも、以前の BDR バージョンの正確な動作を有効にします。デフォルトは現在の BDR バージョンです。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`** &mdash; ピアノードの適用率とキャッチアップ間隔に関するレプリケーションの見積もりを追跡します。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`** &mdash; ローカルノードからの WAL の適用に関してピアノードがどの程度遅れているかを監視し、ラグトラッキングの適用率の移動平均を計算します。このパラメーターは、新しい計算値がこの移動平均計算にどの程度寄与するかを指定します。デフォルト値は 0.1 です。

## Node management

BDR グループのメンバーである各データベースは、独自のノードで表す必要があります。ノードは、BDR グループ内のデータベースの一意的識別子です。

現在、各ノードは 1 つのノードグループのメンバーになれます。(これは後のリリースで拡張される可能性があります。) 各ノードは 1 つ以上のレプリケーションセットをサブスクライブして、レプリケーションをきめ細かく制御できます。

BDR グループには 0 個以上のサブグループが含まれる場合があります、さまざまなアーキテクチャを作成できます。

## BDR グループの作成と参加

**BDR 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、それに接続するように既存のすべてのノードが構成されます。このため、作成された最初の BDR ノードを含むすべてのノードは、他のノードが接続に使用できる `PostgreSQL connection string`

(データソース名 (DSN) と呼ばれます) を認識する必要があります。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` などのキー値形式、または `postgres://myhost:5432/mydb` などの URI 形式を使用できます。

SQL ファンクション `bdr.create_node_group()` は、ローカルノードから BDR グループを作成します。これにより、そのノードで BDR 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 BDR グループに参加できるようになります。作成時に、他のノードがこのノードに接続するために使用する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、以降のすべてのノードは `bdr.join_node_group()` ファンクションを使用して BDR グループに参加できます。

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、既存のノードの `pg_basebackup` (またはフィジカルスタンバイ) を使用して新しいノードを作成します。`pg_basebackup` を使用している場合、`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、不要なデータベースが除外されるため、アクティビティがより速く完了し、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されて削除されます。

新しい BDR ノードが既存の BDR グループに参加する場合、またはノードがアップストリームピアに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最初は同じになるように注意深く調整する必要があります。自分で `pg_dump` を使用するだけでは不十分です。BDR 拡張機能は、この初期コピーを作成するための組み込み機能を提供します。

参加プロセス中に、BDR 拡張機能は、指定されたソースノードを基礎として使用して既存のデータを同期し、BDR 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やり直します。

クラスターに参加するノードには、BDR 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を含めないでください。BDR 拡張子を除き、新しく参加するデータベースは空にすることをお勧めします。ただし、必要なすべてのデータベースユーザーとロールが作成されていることが重要です。

オプションで、`bdr.join_node_group()` ファンクションの `synchronize_structure` パラメータを使用してスキーマの同期をスキップできます。この場合、スキーマは新しく参加するノードに既に存在している必要があります。

参加するソースノードとして、接続が最適な (最も近い) ソースノードを選択することをお勧めします。これにより、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、ほとんどの既存のノード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()` ファンクションを使用する) は、COPY 操作を実行してデータ同期を実行し、有効になっている場合はマルチプルのライター (並列適用) を使用します。

ノード参加は、参加にかかる時間の大部分を他のノード参加と同時に実行できます。ただし、一度に 1 つの通常ノードのみが **PROMOTE** または **PROMOTING** の状態になります。これは、他のすべてのノードが稼働している場合、通常はかなり短くなります。それ以外の場合、結合はこの段階でシリアル化されます。サブスクリバ専用ノードはこのルールの例外であり、同時に **PROMOTE** 状態と **PROMOTING** 状態になる可能性があるため、参加プロセスは完全に同時です。

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、ノードの過半数が利用可能な場合に実行できます。これにより、論理結合の実行が複雑にな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 (ノードがダウンしているか、大幅な遅延があるため) は、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る可能性があり、発散が発生します。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。これが必要な場合は、新しく参加したノードで **LiveCompare** を実行して、すべてのノードが利用可能になって追いついたらデータの相違を修正します。

キャッシュルックアップの失敗が原因で、ソースノードで同時 DDL アクティビティがある場合、`pg_dump` が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に失敗する場合があります。その場合、結合を再試行すると機能します。

## 異種クラスターへの参加

BDR 4.0 ノードは、特定の最小メンテナンスリリース (3.7.6 など) または 3.7 と 4.0 ノードの混合で 3.7.x を実行している EDB Postgres 分散クラスターに参加できます。この手順は、BDR メジャーバージョンだけでなく、基になる PostgreSQL メジャーバージョンもアップグレードする場合に役立ちます。これを実現するには、PostgreSQL 12 または 13 で実行されている 3.7 ノードを PostgreSQL 11 で 3.6.x を実行している EDB Postgres 分散クラスターに参加します。新しいノードは、既存のクラスター内のすべてのノードと同じ PostgreSQL メジャーリリースで実行することもできます。

BDR は、一部のノードが 1 つの PostgreSQL メジャーリリースで 3.6 を実行していて、他のノードが別の PostgreSQL メジャーリリースで 3.7 を実行している場合でも、レプリケーションがすべての方向で正しく動作することを保証します。ただし、十分な数の新しいノードがクラスターに参加したら、古いノードを分割することにより、クラスターをすばやく同種の状態にすることをお勧めします。古いバージョンで使用できない可能性のある DDL を実行しないでください。

別の PostgreSQL メジャーリリースに参加するノードは `bdr_init_physical` で取得した物理バックアップを使用できず、ノードは論理結合方法を使用して参加する必要があります。これが必要なのは、PostgreSQL のメジャーリリースが互いにディスク上で互換性がないためです。

3.7 ノードが 3.6 ノードをソースとして使用してクラスターに参加すると、競合解決などの特定の構成がソースノードからコピーされません。ノードは、クラスターに参加した後に構成する必要があります。

## 接続 DSN と SSL (TLS)

ノードは `libpq` を使用して接続するため、ノードの DSN は単なる `libpq` 接続文字列です。そのため、SSL 用のものを含む、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の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
```

このセットアップでは、ファイル `bdr_client.crt`、`bdr_client.key`、および `root.crt` は、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、および接続で使用す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が確認されます。名前の場合、これはサブジェクトの別名と一致します。証明書にそのような名前がない場合は、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の別名をサポートしていないため、名前ではなくアドレスで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、BDR 接続を行うユーザーの名前である必要があります。これは通常、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
```

別の設定では、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 を検証しません。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。

## Witness ノード

クラスターに偶数個のノードがある場合、ネットワーク分割（またはネットワークパーティションとも呼ばれます）が発生した場合に関係を断ち切るのに役立つ追加のノードを作成すると役立つ場合があります。

追加のフルサイズのノードを作成する代わりに、監視ノードと呼ばれるマイクロノードを作成できます。これは、テーブルやデータを複製しないように意図的に設定された通常の BDR ノードです。

## ロジカルスタンバイノード

BDR を使用すると、オフロードノード、読み取り専用ノード、受信専用ノード、または論理読み取りレプリカとも呼ばれます。マスターノードは、ゼロ、1 つ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。

フィジカルスタンバイノードでは、ノードが完全に起動しないため、継続的リカバリモードが強制されます。BDR 同様のことができます。bdr.join\_node\_group には、ノードをロジカルスタンバイノードとして途中で結合したままにするための pause\_in\_standby オプションがあります。ロジカル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。

後で、必要に応じて bdr.promote\_node() を使用して、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 つのソースノードによって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るため、複数のサイト間の帯域幅が最小限に抑えられます。

高可用性には複数のオプションがあります。

- ソースノードが停止した場合、1 つのフィジカルスタンバイをマスターに昇格できます。この場合、新しいマスターは、任意またはすべてのロジカルスタンバイノードに電力を供給し続けることができます。
- 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イ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ある場合、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 BDR ノードから新しいスタンバイが作成された場合、操作に必要なレプリケーションスロットは、グループスロットが最後に進められてから少なくとも 16 MB が経過するまで新しいスタンバイに同期されません。極端な場合、ストリーミングレプリカでスロットが同期または作成される前に、16 MB 全体が必要になる場合があります。この間隔中に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、グループスロットと他の依存スロットがまだ存在しないため、ストリーミングスタンバイをその BDR ノードに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アッププロセスは、アップストリーム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 BDR 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。これにより、グループスロットが短時間で前進します。これにより、スタンバイが最初のスロットの同期に必要な時間が短縮され、必要に応じてより迅速にフェールオーバーできます。

PostgreSQL では、プロモートする前に、スロットの同期がスタンバイで完了してい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 BDR 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、BDR クラスター全体でスロット同期プロセスを微調整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進み、スタンバイでのスロット同期アクティビティが早められます。このために、ターゲットデータベース内の BDR ピアノードで次のクエリを実行できます。

```
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に役立つことを確認します。

ロジカルスタンバイノードは、必要に応じてフィジカルスタンバイノードを使用して保護できるため、Master->LogicalStandby->PhysicalStandby。LogicalStandby から LogicalStandby にカスケードすることはできません。

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るため、フィジカルスタンバイの制限は適用されません。これを使用すると、ロジカルスタンバイに追加のインデックス、データ、中間作業表、LISTEN/NOTIFY、一時テーブル、マテリアライズドビューなどの違いを設定できるため、大きなメリットがあります。

昇格前にコミットされたロジカルスタンバイにローカルで加えられた変更は、他のノードに送信されません。昇格後にコミットされ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は、昇格の前にデータベースを静止してください。

ロジカルスタンバイノードに DDL 変更を加える場合がありますが、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 BDR ファンクションもレプリケートされません。

**DML および DDL レプリケーション** 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 変更を加えた場合、データベースは**ダイバージェントノード**です。ダイバージェントノードを昇格すると、現在レプリケーションが失敗します。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、そのノードに変更が加えられないようにするか、または分散ノードが昇格しないようにしてください。

## フィジカルスタンバイノード

BDR では、従来のフィジカルスタンバイフェールオーバーノードを作成することもできます。これらは、一般的に、短い昇格手順の後にクラスター内の BDR ノードを直接置き換えることを目的としています。標準の Postgres クラスターと同様に、ノードはこれらの物理レプリカをいくつでも持つことができます。

ただし、BDR でレプリケーションスロットおよびその他の機能要件を使用するため、これが適切に動作するための最小前提条件がいくつかあります。

- BDR プライマリとスタンバイ間の接続は、物理レプリケーションスロットを介したストリーミングレプリケーションを使用します。
- スタンバイには以下があります。
  - **recovery.conf** (PostgreSQL <12 の場合、これらの設定は **postgres.conf** にあります):
  - プライマリを指す **primary\_conninfo** :- **shared\_preload\_libraries = 'bdr'**、リストには他のプラグインもありますが、**pglogical** を含めないでください
  - **hot\_standby = on** - **hot\_standby\_feedback = on**

- プライマリには次のものがあります。- `postgresql.conf` : - `bdr.standby_slot_names` は、スタンバイの `primary_slot_name` に使用される物理レプリケーションスロットを指定します。

BDR ノードの動作中のフィジカルスタンバイを生成するにはこれで十分ですが、いくつかの追加の問題に対処する必要があります。

確立されると、スタンバイには、BDR グループスロットを含むプライマリの他の BDR 関連のレプリケーションスロットの初期コピーをトリガーするのに十分な時間と WAL トラフィックが必要です。少なくとも、スタンバイのスロットはライブであり、`pg_replication_slots` によって報告されるようにゼロ以外の `confirmed_flush_lsn` を報告する場合にのみ、フェイルオーバーを生き残ることができます。

そのため、フェールオーバーが正常に動作すると想定する前に、書き込みアクティビティの量が少ない新しく初期化された BDR クラスターのフィジカルスタンバイノードを確認します。この予防措置を講じないと、スタンバイが BDR ノードとして機能するために必要なレプリケーションスロットの不完全なサブセットになり、フェールオーバーが中止される可能性があります。

フィジカルスタンバイノードが最新の状態であり、昇格できるようにする保護メカニズム (`bdr.standby_slot_names` の構成) は、BDR グループの全体的なレプリケーション遅延に影響します。これは、フィジカルスタンバイノードが最新の場合にのみグループレプリケーションが発生するためです。

これらの理由から、通常、フィジカルスタンバイノードではなく、ロジカルスタンバイノードまたはサブスクライブ専用グループを使用することをお勧めします。比較すると、どちらも操作特性が優れています。

すべてのノードでグループスロットが（可能な限り）拡張されていることを手動で確認できます。これにより、次の SQL 構文を使用して、フィジカルスタンバイでの BDR 関連のレプリケーションスロットの作成が促進されます。

```
SELECT bdr.move_group_slot_all_nodes();
```

フェールオーバー時に、スタンバイは次の 2 つのアクションのいずれかを実行してプライマリを置き換える必要があります。

- プライマリと同じ IP アドレスまたはホスト名の制御を引き受けます。
- 他のすべての BDR ノードで `bdr.alter_node_interface` ファンクションを実行して、アドレスの変更を EDB Postgres 分散クラスターに通知します。

これが完了すると、他の BDR ノードは新しく昇格したスタンバイ → プライマリノードとの通信を再確立します。レプリケーションスロットは定期的にのみ同期されるため、この新しいプライマリは既存の BDR ノードで予想されるよりも低い LSN を反映する場合があります。この場合、BDR は各遅延スロットを各 BDR ノードが使用した最後の場所に早送りします。

`bdr.standby_slot_names` パラメーターにも注意してください。プライマリ→フィジカルスタンバイの関係がある EDB Postgres 分散クラスターで、またはサブスクライバー専用グループを使用する場合は、それを設定することが重要です。

BDR は、アウトバウンドレプリケーションで最も遅れているクラスターノードの状態を常に反映するグループスロットを維持します。物理レプリカを追加すると、グループスロットの状態に影響を与える非参加ノードメンバーがあることを BDR に通知する必要があります。

スタンバイは他の BDR ノードと直接通信しないため、`standby_slot_names` パラメーターは、グループスロットの必要な制約として名前付きスロットを考慮するように BDR に通知します。設定すると、グループスロットが通常に進んでいても、スタンバイが遅れる場合はグループスロットが保持されます。

他の物理レプリカと同様に、このタイプのスタンバイは同期レプリカとして構成することもできます。なお、これには以下が必要です。

- スタンバイ時： - `primary_conninfo` にユニークな `application_name` を指定
- プライマリで： - `synchronous_commit` を有効にする - `synchronous_standby_names` にスタンバイ `application_name` を含める

`synchronous_standby_names` にフィジカルスタンバイと他の BDR ノードを混在させることができます。CAMO と Eager All-Node Replication は異なる同期メカニズムを使用し、同期レプリケーションでは機能しません。`synchronous_standby_names` に CAMO パートナー（CAMO を使用する場合）または BDR ノードをまったく含めないでください（Eager All-Node Replication を使用する場合）。代わりに、フィジカルスタンバイなどの非 BDR ノードのみを使用します。

## 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。各サブグループは、トップレベルの親グループで特定の目的に割り当てることができます。`node_group_type` は、サブグループを作成するときにタイプを指定します。

## 購読者専用グループ

名前が示すように、このタイプのノードは、クラスター内の他のノードからのレプリケーションの変更のみをサブスクライブします。ただし、他のノードは `subscriber-only` ノードからレプリケーションの変更を受け取りません。これは、ロジカルスタンバイノードに多少似ています。ただし、ロジカルスタンバイとは対照的に、`subscriber-only` ノードはクラスターに完全に参加しています。クラスター内の他のすべてのノードからレプリケーションの変更を受け取ることができるため、クラスター内のいずれかのノードが使用不可または分離されても影響を受けません。

`subscriber-only` ノードは完全に参加した BDR ノードであるため、レプリケートされたすべての DDL を受け取り、それらに動作します。また、Raft を使用して、クラスター内のすべてのノードにステータスを一貫して報告します。`subscriber-only` ノードは Raft の投票権がないため、Raft リーダーになったり、リーダー選挙に参加したりできません。また、レプリケートされた DDL を受信しますが、DDL または DML ロックの取得には参加しません。言い換えれば、現在ダウンしている `subscriber-only` ノードは、DML ロックの取得を停止しません。

`subscriber-only` ノードは、BDR ツリートポロジの構成要素を形成します。このトポロジでは、少数の完全にアクティブなノードがすべての方向に変更を複製しています。多数の `subscriber-only` ノードは変更のみを受け取りますが、クラスター内の他のノードに変更を送信することはありません。このトポロジは、多数のノードによる接続の爆発を回避しますが、データの使用に使用できる非常に多数の leaf ノードを提供します。

`subscriber-only` ノードを利用するには、最初にタイプ `subscriber-only` の BDR グループを作成します。メンバーノードがレプリケーションの変更を受け取るグループのサブグループにします。サブグループを作成したら、`subscriber-only` ノードになるすべてのノードはサブグループに参加する必要があります。`subscriber-only` タイプのサブグループを複数作成でき、異なる親グループを持つことができます。

ノードが `subscriber-only` サブグループに正常に参加すると、それは `subscriber-only` ノードになり、親グループのレプリケーション変更の受信を開始します。`subscriber-only` ノードで直接行われた変更は

複製されません。

特定のタイプのサブグループを作成し、特定の親グループに属する方法については、`bdr.create_node_group()` を参照してください。

## 注意事項

`subscriber-only` ノードはクラスター内のノードに変更をレプリケートしないため、ノードがクラスターから分離された場合、レプリケーションの変更を同期するソースとして機能できません。ただし、`subscriber-only` ノードが既にクラスター内の他のノードにない `parted` ノードからレプリケーションの変更を受信して適用している場合、ノード間で不整合が発生します。

今のところ、`subscriber-only` ノードよりも前に完全にアクティブな BDR ノードが常に存在するように、`bdr.standby_slot_names` と `bdr.standby_slots_min_confirmed` を設定することでこれを解決できます。これは将来のリリースで改善される可能性があります。`subscriber-only` ノードがレプリケーションの先頭になることを許可してからそれらを同期のレプリケーションソースとして使用するか、別の完全に結合されたノードが分離されたときに一貫性のない `subscriber-only` ノードをオプションで削除する方法を提供します。

## デコードワーカー

BDR4 は、データが送信されるノードの数に関係なく、1 回のデコードを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。これにより、各 BDR ノードに新しいプロセス、WAL デコーダーが導入されます。接続ごとに 1 つの WAL 送信プロセスが存在しますが、これらのプロセスはデータの送受信のみを実行します。まとめると、これらの変更はより大きな BDR グループの CPU オーバーヘッドを削減し、WAL 送信者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、レプリケーションスループットも向上します。

`enable_wal_decoder` は BDR グループごと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_config()` を使用して、BDR 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と、BDR は論理変更レコード (LCR) ファイルを保存して、デコードとすべてのサブスクリバードがデータを受信したときの変更をバッファリングできます。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` ディレクトリに保存されます。レプリケーションの遅延が増加すると、LCR ファイルの数とサイズが変化するため、これも監視する必要があります。BDR ノードで必要とされない 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の間隔は、デフォルトの 3 分である `bdr.lcr_cleanup_interval` によって制御されます。`bdr.lcr_cleanup_interval` がゼロの場合、クリーンアップは無効です。

無効にすると、各ノードにサブスクリバードする各ノードの WAL 送信者プロセスによって論理デコードが実行されます。この場合、LCR ファイルは書き込まれません。

BDR グループのデコードワーカーが有効になっている場合でも、次の GUC はノードごとの 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 を作成して使用するには、BDR グループのデコードワーカーを有効にし、これらの GUC を BDR グループの各ノードで `true` に設定します。

- `bdr.enable_wal_decoder` — `false` を有効にすると、LCR を使用するすべての WAL 送信者が再起動して WAL を直接使用します。BDR グループ構成とともに `true` を使用すると、デコードワーカー

プロセスが開始されて LCR が生成され、WAL 送信者は LCR を使用します。

- `bdr.receive_lcr` —サブスクリバースタンプで `true` を使用する場合、パブリッシャーノードの WAL センダーに LCR を使用するよう要求します。

## ノート

現時点では、デコードワーカーは、実行されているノードに対応する変更をデコードします。ロジカルスタンバイには、BDR グループ内のすべてのノードから単一のソースを介して変更が送信されます。したがって、ロジカルスタンバイを提供する WAL 送信者は現在 LCR を使用できません。

サブスクリバースタンプ専用ノードは、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、特に必要がない場合、LCR に対応する WAL を削除でき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 ファイル名と似ています。現在名前の最初の 8 文字はすべて「0」です。将来的には、WAL セグメントファイル名の最初の 8 文字に似た `TimeLineId` を表すことが期待されています。次の 16 文字の名前は、WAL ストリームに対して LCR の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、それらが属す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ています。LCR ファイルはバイナリで可変サイズです。LCR ファイルの最大サイズは、デフォルトの 1GB である `bdr.max_lcr_segment_file_size` で制御できます。

## ノードの再起動とダウンしたノードの回復

BDR は、ノードの再起動またはノードの切断から回復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` を表示し始め、欠落したデータの複製を開始します。各ピアノードからの欠落データの量に応じて、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なる可能性があります。最終的に、スロットの状態は `bdr.node_slots.state = streaming` に変わります。

数時間や数日など、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引き起こし始める可能性があります。次の問題を理解せずに、長期にわたる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するため（ピアノードごとに 1 つの **レプリケーションスロットのクリーンアップ** を使用）、一時的に到達できないノードへの変更を後で再生できます。ピアノードが無期限にオフラインのままである場合、この蓄積された変更情報により、最終的にノードで PostgreSQL トランザクションロ



グ (pg\_wal の WAL) が不足し、データベースサーバが次のようなエラーで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 を抑え、カタログテーブルのバキュームを防ぎます。

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 (

[オリジンの競合検出](#) を参照)。

管理者はノードの停止を監視し ([Monitoring](#) を参照)、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、経時的に使用される領域を計算できる場合があります。重大な問題が発生するまでにノードがダウンする可能性がある時間を予測できます。

**BDR 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。これを行うと、クラスターが損傷し、**

**[BDR によって作成されたレプリケーションスロット](#)**

で説明されているように、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の間、他のノードはオフラインノードから同じデータセットをまだ受信していない可能性があるため、これはノード間でわずかな相違として表示される場合があります。分割プロセスは、ノード間のこの不均衡を修正します。(後のバージョンではこれを早く行うかもしれません。)

## BDR によって作成されたレプリケーションスロット

BDR マスターノードでは、次のレプリケーションスロットが BDR によって作成されます。

- bdr\_<database name>\_<group name> という名前の 1 つのグループスロット
- N-1 ノードスロット、名前付け bdr\_<database name>\_<group name>\_<node name>、ここで N は、ダイレクトロジカルスタンバイを含むクラスター内の BDR ノードの合計数です

**警告:** これらのスロットを削除しないでください。BDR はそれらを作成および管理し、必要に応じて削除します。

一方、ソフトウェア用の適切なコマンドを使用して、Barman や論理レプリケーションなどのソフトウェアに必要なレプリケーションスロットを作成または削除できます BDR。他のソフトウェアが使用するスロット名を接頭辞 bdr\_ で始めないでください。

たとえば、3 つのノード alpha、beta、および gamma で構成されるクラスターでは、BDR を使用して mydb データベースを複製し、BDR グループは mygroup と呼ばれます。

- ノード alpha には 3 つのスロットがあります。- bdr\_mydb\_mygroup という名前の 1 つのグループスロット - bdr\_mydb\_mygroup\_beta および bdr\_mydb\_mygroup\_gamma という名前の 2 つのノードスロット

- ノード **beta** には 3 つのスロットがあります。- **bdr\_mydb\_mygroup** という名前の 1 つのグループスロット - **bdr\_mydb\_mygroup\_alpha** および **bdr\_mydb\_mygroup\_gamma** という名前の 2 つのノードスロット
- ノード **gamma** には 3 つのスロットがあります。- **bdr\_mydb\_mygroup** という名前の 1 つのグループスロット - **bdr\_mydb\_mygroup\_alpha** および **bdr\_mydb\_mygroup\_beta** という名前の 2 つのノードスロット

## グループレプリケーションスロット

グループスロットは、このノードからの送信レプリケーションについて、主に BDR グループ内のノード（すべてのロジカルスタンバイを含む）が追いついた最も古い安全な位置を追跡するために BDR によって使用される内部スロットです。

グループスロット名は関数 `bdr.local_group_slot_name()` で与えられます。

グループスロットでは次のことができます。

- 既存のすべてのノードを起動して実行せずに（ノードの大部分は稼働している必要がありますが）、新しいノードを BDR グループに参加させます。参加中にダウンしたノードが再度複製を開始した場合。
- 一部のノードが分割されたノードに完全に追いついていない場合でも、一貫してクラスターからノードを分割します。
- いくつかの競合を見逃さないように、フリーズポイントを保持します。
- 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あり、他のノードからの Raft 進行状況メッセージにตอบสนองして定期的にのみ高速転送されます。

**警告:** グループスロットを削除しないでください。通常は非アクティブですが、EDB Postgres 分散クラスターの適切なオペレーションには依然として不可欠です。ドロップすると、一部またはすべての機能が動作しなくなったり、誤った結果が発生したりする可能性があります。

## 長い識別子のハッシュ

他の PostgreSQL 識別子と同様に、レプリケーションスロットの名前は 63 バイト以下にする必要があります。BDR は、結果のスロット名がその制限に対して長すぎる場合に、データベース名、BDR グループ名、およびノードの名前を短縮することによりこれを処理します。識別子を短くするには、文字列の最後のセクションを文字列自身自身のハッシュに置き換えます。

たとえば、`group20xxxxxxxxxxxxxx`（長さ 20 バイト）という BDR グループを使用して `db20xxxxxxxxxxxxxx`（長さ 20 バイト）という名前のデータベースを複製するクラスターを考えます。3597186、be9cbd0、7f304a2 はそれぞれ `db20xxxxxxxxxxxxxx`、`group20xxxxxxxxxxxxxx`、`a30xxxxxxxxxxxxxxxxxxxxxxxxxx` のハッシュであるため、ノード `a30xxxxxxxxxxxxxxxxxxxxxxxxxx`（長さ 30 バイト）に関連付けられた論理レプリケーションスロットが呼び出されます。

```
bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxxx7f304a2
```

## BDR グループからのノードの削除

BDR は拡張されたノードの停止から回復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伝える必要があります。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と、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**パーティング**とも呼ばれ、`bdr.part_node()` 関数を使用して実行されます。ノードを削除するには、(ノードの作成中に渡された) ノード名を指定する必要があります。削除するノードを含む、BDR グループ内のアクティブなノードから `bdr.part_node()` ファンクションを呼び出すことができます。

参加手順と同様に、別れは Raft コンセンサスを使用して行われ、動作するには大部分のノード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します。ラフト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データを持っているかを確認します。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確立して、不足しているデータをキャッチできるようにします。

別れたノードはまだ BDR に認識されていますが、リソースを消費しません。別れたノードと同じ名前でノードが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、`parted` ノードのすべてのメタデータをクリアすることができます。

## BDR のアンインストール

BDR 拡張機能を削除すると、メタデータテーブルを含むノード内のすべての BDR オブジェクトが削除されます。次のコマンドでこれを行うことができます。

```
DROP EXTENSION bdr;
```

データベースが BDR 固有のオブジェクトに依存している場合、BDR 拡張機能を削除できません。例は次のとおりです。

- `SnowflakeId` や `galloc` などの BDR 固有のシーケンスを使用したテーブル
- CRDT データ型を使用する列
- 一部の BDR カタログテーブルに依存するビュー

BDR 拡張機能を削除する前に、これらの依存関係を削除してください。たとえば、依存オブジェクトを削除するか、列タイプを BDR 以外の同等のものに変更するか、シーケンスタイプを `local` に戻します。

**警告:** ノードが BDR ノードグループから正常に分離された場合、またはグループの最後のノードである場合にのみ、BDR 拡張機能を削除できます。BDR メタデータを削除すると、他のノードとの間のレプリケーションが中断されます。



**警告:** ローカルデータベース内のローカル BDR ノードまたは BDR 拡張機能を削除すると、既存のセッションが BDR 固有のワークフローを実行しようとして失敗する場合があります。セッションを切断してからクライアントを再接続するか、インスタンスを再起動することにより、問題を解決できます。

`bdr.drop_node()` 関数もあります。この機能は、お別れに困ったときなど、緊急時にご利用ください。

## BDR トポロジのリスト

### BDR グループのリスト

次の単純なクエリは、現在のノードがメンバーであるすべての BDR ノードグループをリストします。現在、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);
```

### BDR グループ内のノードのリスト

次の例に示すように、特定のノードグループ（`mygroup` など）内のすべてのノードのリストを `bdr.node_summary` ビューから抽出できます。

```
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;
```

`current_state` または `target_state` クエリ列に示されるように、ノードの読み取り専用状態は、`STANDBY` として示されます。

## ノードの状態のリスト

- **NONE** : ワーカーの起動時にノードの状態は設定解除され、すぐに現在の既知の状態に設定されると予想されます。
- **CREATED** : `bdr.create_node()` は実行されましたが、ノードはまだ EDB Postgres 分散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 分散クラスターに参加させ始めます。
- **JOINING** : ノード参加が開始され、現在初期同期フェーズであり、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。これで、参加は、参加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して適用する最後のステップになりました。
- **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。すべての参加はこの状態でしばらく時間がかかりますが、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、ノードの状態を **ACTIVE** に移動するために `bdr.promote_node` を呼び出しました。これら 2 つの **PROMOTE** 状態は、1 つのノードのみが **STANDBY** よりも高く、**ACTIVE** よりも低い状態にあ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、現在は **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、`bdr.part_node` を呼び出して EDB Postgres 分散クラスターからノードを削除しました。
- **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欠落しているデータを同期します。
- **PARTED** : ノードの分割操作がすべてのノードで完了しました。

一度に 1 つのノードのみが、状態 **PROMOTE** または **PROMOTING** のいずれかになります。

## ノード管理インターフェース

SQL インターフェースを使用して、ノードを動的に追加および削除できます。

## bdr.create\_node

この関数はノードを作成します。

### 概要

```
bdr.create_node(node_name text, local_dsn text)
```

### パラメーター

- **node\_name** —新しいノードの名前。データベースごとに許可されるノードは 1 つだけです。有効なノード名は、小文字、数字、ハイフン、および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。

### 注意事項

この関数は、関連付けられた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 つしか存在できないため、作成すると、関数は再度実行されるとエラーを報告します。

この関数はトランザクション関数です。ロールバックでき、それによって加えられた変更は現在のトランザクションで表示されます。

この関数は、トランザクションが終了するまで、新しく作成された bdr ノードのロックを保持します。

## bdr.drop\_node

ノードを削除します。

**警告:** この関数は、通常の使用を対象としていません。テクニカルサポートから指示された場合にのみ実行してください。

この関数は、指定された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メタデータが削除されます。
- 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が削除されます。

## 概要

```
bdr.drop_node(node_name text, cascade boolean DEFAULT false, force boolean DEFAULT   
→false)
```

## パラメーター

- **node\_name** —既存のノードの名前。
- **cascade** —非推奨、将来的に削除されます。
- **force** —すべての健全性チェックを回避し、矛盾を引き起こす可能性があるにもかかわらず、指定された BDR ノードのすべてのメタデータを強制的に削除します。テクニカルサポートのみが、別に関連する緊急事態に備えて強制ノードドロップを使用します。

## 注意事項

これを実行する前に、`bdr.part_node()` を使用してノードを分割します。

この関数は、指定された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することができます。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メタデータが削除されます。または、リモートノードにすることもできます。その場合、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:** BDR4 は、各ノードに snowflakeid および timehard 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 個のノードレコード (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。これが問題になった場合、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.create\_node\_group**

この機能は、ローカルノードをグループの唯一のメンバーとする BDR グループを作成します。

## 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

- `node_group_name` —新しい BDR 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、およびアンダースコアのみで構成する必要があります。
- `parent_group_name` —サブグループの親グループの名前。
- `join_node_group` —このパラメーターは、ノードが作成するグループに参加するかどうかを判断するのに役立ちます。デフォルト値は `true` です。これは、ノードが参加したくないシャードグループを作成しているときに使用されます。`parent_group_name` を指定した場合のみ、これは `false` になります。
- `node_group_type` —有効な値は、`NULL`、`subscriber-only`、`datanode`、`read coordinator`、および `write coordinator` です。`subscriber-only` is used to create a group of nodes that receive changes only from the fully joined nodes in the cluster, but they never send replication changes to other nodes. See `:ref:`Subscriber-only nodes <#subscriber-only-nodes>`` for more details. `Datanode` implies that the group represents a shard, whereas the other values imply that the group represents respective coordinators. Except `subscriber-only`, the other values are reserved for future use. `NULL` は、通常の汎用ノードグループが作成されることを意味します。

## 注意事項

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リクエストを渡します。

この関数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、機能が終了すると、変更をロールバックできません。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、それらが完了するまで待つことができます。

グループの作成にはロックはありません。

## `bdr.alter_node_group_config`

この機能は、既存の BDR グループの設定パラメーターを変更します。`NULL` 値（すべてのデフォルト）を持つオプションは変更されません。

## 概要

```
bdr.alter_node_group_config(node_group_name text,
                             insert_to_update boolean DEFAULT NULL,
                             update_to_insert boolean DEFAULT NULL,
                             ignore_redundant_updates boolean DEFAULT NULL,
                             check_full_tuple boolean DEFAULT NULL,
                             apply_delay interval DEFAULT NULL,
```

(次のページに続く)

```

check_constraints boolean DEFAULT NULL,
num_writers int DEFAULT NULL,
                enable_wal_decoder boolean DEFAULT NULL,
                streaming_mode text DEFAULT NULL,
default_commit_scope text DEFAULT NULL)

```

## パラメーター

- `node_group_name` —既存の BDR グループの名前。ローカルノードはグループの一部である必要があります。
- `insert_to_update` —下位互換性のために予約されています。
- `update_to_insert` —下位互換性のために予約されています。BDR のバージョン。代わりに `bdr.alter_node_set_conflict_resolver` を使用してください。
- `ignore_redundant_updates` —下位互換性のために予約されています。
- `check_full_tuple` —下位互換性のために予約されています。
- `apply_delay` —下位互換性のために予約されています。
- `check_constraints` —レプリケートされたデータを書き込むときに適用プロセスが制約をチェックするかどうか。このオプションは非推奨であり、BDR の将来のバージョンでは無効化または削除されます。
- `num_writers` —このノードグループをバックアップするサブスクリプションの並列ライターの数。-1 は、デフォルト (GUC `bdr.writers_per_subscription` で指定) が使用されることを意味します。有効な値は、-1 または正の整数です。
- `enable_wal_decoder` —デコードワーカプロセスを有効/無効にします。`streaming_mode` が既に有効になっている場合、デコードワーカプロセスを有効にすることはできません。
- `streaming_mode` —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、ストリーミングは無効になります。他の値に設定すると、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、トランザクションがアップストリームで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 1 つに直接送信されます (使用可能な場合)。並列適用が無効になっている場合、またはライターがストリーミングトランザクションを自由に処理できない場合、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、BDR 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` を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。

詳細については、[トランザクションストリーミング](#) を参照してください。

- `default_commit_scope` —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。これは、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、[オリジングループ](#)を参照してください。

## 注意事項

このファンクションは、デフォルトを変更するリクエスト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。

この関数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、関数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。

この関数はロックを保持しません。

**警告:** この関数を使用して `apply\_delay` 値を変更しても、その変更は既にグループのメンバーであるノードには適用されません。この値は通常テスト以外では使用されないため、この制限は実稼働での使用にはほとんど影響しません。

## bdr.join\_node\_group

この機能は、ローカルノードを既存の BDR 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,
  pause_in_standby boolean DEFAULT false,
  wait_for_completion boolean DEFAULT true,
  synchronize_structure text DEFAULT all
)
```

## パラメーター

- `join_target_dsn` — ローカルノードを追加する BDR 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` — BDR 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。
- `pause_in_standby` — オプションで、ロジカルスタンバイノードとしてのみ参加するように参加プロセスに指示します。これは後でフルメンバーに昇格できます。
- `wait_for_completion` — 参加プロセスが完了するまで待ってから、戻ります。デフォルトは `true` です。

- `synchronize_structure` —結合中に実行する構造（スキーマ）同期の種類を設定します。有効なオプションは、完全なデータベース構造を同期する `all` と、構造を同期しない `none` です。ただし、データは同期されます。

`wait_for_completion` が `false` として指定されている場合、これは参加プロシージャが開始されるとすぐに戻る非同期呼び出しです。結合の進行状況は、ログと `bdr.state_journal_details` 情報ビューで確認するか、`bdr.join_node_group()` が戻った後に `bdr.wait_for_join_completion()` 関数を呼び出すことで確認できます。

### 注意事項

このファンクションは、`join_target_dsn` 接続文字列が指す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。

この関数はトランザクションではありません。参加プロセスはバックグラウンドで行われるため、ロールバックできません。`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` を呼び出すことにより、変更はローカルトランザクションにのみ表示されます。

ノードは単一のグループのみに属することができるため、この関数は各ノードで 1 回のみ呼び出すことができます。

ノード参加は BDR グループにロックを保持しません。

### `bdr.promote_node`

この機能は、ローカルロジカルスタンバイノードを BDR グループのフルメンバーに昇格させます。

### 概要

```
bdr.promote_node(wait_for_completion boolean DEFAULT true)
```

### 注意事項

このファンクションは、デフォルトを変更するリクエスト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。

この関数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行われるため、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` を呼び出すことにより、変更はローカルトランザクションにのみ表示されます。

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むのを防ぎます。



## bdr.wait\_for\_join\_completion

この関数は、ローカルノードの参加手続きが完了するのを待ちます。

### 概要

```
bdr.wait_for_join_completion(verbose_progress boolean DEFAULT false)
```

### パラメーター

- **verbose\_progress** — オプションで、結合手順中に実行された個々の手順に関する情報を出力します。

### 注意事項

この関数は、ローカルノードのチェックの状態が `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の状態に到達するまで待機します。

## bdr.part\_node

BDR グループからノードを削除（分割）しますが、ノードからデータは削除しません。

削除するノードを含む、BDR グループ内のアクティブなノードから関数を呼び出すことができます。ただし、ノードが分離されると、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。

**注釈:** ローカルノードを分割する場合は、`wait_for_completion` を `false` に設定する必要があります。それ以外の場合は、エラーを報告します。

**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()` を参照してください。

### 概要

```
bdr.part_node (
    node_name text,
    wait_for_completion boolean DEFAULT true,
    force boolean DEFAULT false
)
```

## パラメーター

- **node\_name** — 分割する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、関数は戻りません。それ以外の場合、関数は分離手順を開始し、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** を使用する場合は常に **false** に設定します。
- **force** — ローカルノード上のノードを強制的に削除します。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、これによりノードの状態がローカルに設定されます。

**警告:** *force = true* を使用すると、BDR グループが一貫性のない状態のままになる可能性があります。他の方法でノードを削除できない障害から回復する場合にのみ使用してください。

## 注意事項

この関数は、指定されたノードを分割するための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。別れのプロセスはバックグラウンドで行われ、元に戻すことはできません。**wait\_for\_completion** が **true** に設定されている場合、分割プロセスによる変更はローカルトランザクションでのみ表示されます。

**force** を **true** に設定すると、コンセンサスが失敗すると、この関数はローカルノードでのみ指定された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（ファンクションは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割中のノードで関数が呼び出された場合、指定されたノードをローカルで分割されて終了します。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、受信していた分離ノードが書き込みを行うと、分離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、指定されたノードから欠落しているデータを再同期する必要があります。強制的に分離すると、この再同期が完全にスキップされ、他のノードが一貫性のない状態のままになる可能性があります。

分割プロセスにはロックがありません。

## bdr.alter\_node\_interface

この関数は、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

- `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。

### 注意事項

この関数を実行し、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 BDR 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、変更は現在のトランザクションで表示されます。

関数はローカルノードでロックを保持します。

## bdr.alter\_subscription\_enable

この機能は、指定されたサブスクリプションまたはローカル BDR 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、レジューム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

- **subscription\_name** —有効にするサブスクリプションの名前。NULL（デフォルト）の場合、ローカルノード上のすべてのサブスクリプションが有効になります。
- **immediate** —これは現在効果がありません。

## 注意事項

この関数はレプリケートされず、ローカルノードの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、現在のトランザクションで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開始されます。

## **bdr.alter\_subscription\_disable**

この機能は、指定されたサブスクリプションまたはローカル BDR 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、無効になっている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

- **subscription\_name** —無効にするサブスクリプションの名前。NULL（デフォルト）の場合、ローカルノード上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、無効になっている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関数を実行することはできません。
- **fast** —この引数は、**immediate** の動作に影響します。 **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、無効になっているサブスクリプションに関連付けられたすべてのワーカーを停止します。

## 注意事項

この関数はレプリケートされず、ローカルノードの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、現在のトランザクションで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` に設定すると、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。

## ノード管理コマンド

BDR は、既存のノードの物理コピー (`pg_basebackup`) を使用して BDR グループにノードを追加したり、既存のノードのフィジカルスタンバイを BDR グループ内の新しいノードに変換したり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、これが新しいノードのベースとして使用されます。データディレクトリが既にフィジカルスタンバイノードとしてアクティブになっている場合は、Postgres を管理する `bdr_init_physical` を実行する前にスタンバイを停止する必要があります。最初は追いつきを待ってから、マスターノードに昇格して BDR グループに参加します。 `--standby` オプションを使用すると、既存のフィジカルスタンバイがロジカルスタンバイノードに変わります。これは、指定されたデータディレクトリの開始状態ではなく、新しい BDR ノードの終了状態を参照します。

このコマンドは、データベースからすべての 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削除します (または、 `-S` オプションが使用されている場合は無効にします)。

## 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` コマンドを使用して移入されたディレクトリのいずれかです。

- `-l, --log-file=FILE` — ログには `FILE` を使用します。デフォルトは `bdr_init_physical_postgres.log` です。
- `-n, --node-name=NAME` — 新しく作成されたノードの名前 (必須)。
- `--replication-sets=SETS` — 使用する複製セット名のコンマ区切りリストの名前。指定しない場合、すべての複製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` — ログの詳細度を上げます。
- `-L` — 空/存在しないデータディレクトリ (`-D` オプション) で使用する場合、選択的に `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削除する代わりに、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。

### 注意事項

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 BDR 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は影響しません。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 BDR ノードにプロモートされるかに当てはまります。したがって、`--replication-sets` オプションは、ノードが BDR 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、論理結合で複製セットを使用する方法とは異なります。

オペレーターは、結合の完了後に不要なテーブルを切り捨てることができます。`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、サブスクライブし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、テーブルは削除せずに切り捨て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必ずしも行 (DML) レプリケーションセットと同じではないため、誤って他のノードのテーブルを削除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、ノードのサブセットで削除した場合は、レプリケーションセットに追加する前に、DDL 競合を作成せずにそれらのノードでのみ再作成する必要があります。

レプリケートされないテーブルを切り捨てて、それらは存在しますが空のままにする方が簡単で安全です。BDR の将来のバージョンでは、物理的結合用に選択されたレプリケーションセットの一部ではないテーブルが自動的に省略または削除される可能性があります。動作はここに文書化されています。

## 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。

操作の利便性と正確さのために、BDR はほとんどの DDL アクションをレプリケートしますが、次の例外があります。

- 一時的またはログされていない関係
- **特定の DDL ステートメント（ほとんどが長時間実行）；**  
DDL statements with restrictions を参照）
- 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）
- autovacuum 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て一貫性を保証することなく、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。DDL をレプリケートするには、DDL レプリケーションフィルタをレプリケーションセットに追加する必要があります。

**DDL レプリケーションのフィルタリング** を参照してください。

BDR は、DDL レプリケーションに関しては、スタンドアロンの PostgreSQL とは大きく異なります。それを同じように扱うことは、BDR 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 レプリケーションは DDL の結果ではなくステートメント自身自身をレプリケートすることです。これはほとんどの場合非常にうまく機能しますが、DDL はすべてのノードで同様に実行する必要があります。より微妙な点は、DDL は拡張機能（ビルトインではない）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、各ノードで使用す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。

## DDL レプリケーションオプション

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

`bdr.ddl_replication = on` はデフォルトであり、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）に複製します。デフォルト以外のレプリケーションセットは、

DDL filter が定義されていない限り DDL をレプリケートしません。

ファンクション `bdr.replicate_ddl_command()` を使用して、特定のレプリケーションセットに DDL をレプリケートすることもできます。これは、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、またはサイト 1 のすべてのノードなど、ノードまたは担当者セットのサブセット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役立ちます。

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

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

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

設定すると、BDR 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 が同時に実行されている場合に BDR グループ全体がレプリケーションを停止する場合があります。

`bdr.ddl_replication` パラメーターは、`bdr_superuser`、スーパーユーザー、または `config` ファイルでのみ設定できます。

## BDR システムでの DDL の実行

BDR 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、中央ロックとトランザクションコーディネーターの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。

並列実行される DDL は、BDR で引き続き実行されます。DDL の実行では、実行時に各ノードでの並列操作に影響を与えるパラメーターが尊重されるため、ノード間の設定の違いに気付く場合があります。

競合する DDL の実行を防止します。そうでない場合、DDL レプリケーションによりエラーが発生し、レプリケーションが停止します。

BDR は、これらの問題に対して 3 つのレベルの保護を提供します。



`ddl_locking = 'dml'` は、一度に1つのノードからのみ DDL を実行する場合に使用可能な操作に最適なオプションです。これはデフォルトではありませんが、DDL の実行元を制御できる場合は、この設定を使用することをお勧めします。これにより、ノード間の競合がなくなります。ノード内の競合は既に PostgreSQL によって処理されています。

`ddl_locking = on` 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、正確性を保証したい場合に最適です。

`ddl_locking = off` は最も厳密でないオプションであり、一般的な使用では危険です。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ちます。

これらのオプションは、`bdr_superuser`、スーパーユーザー、または `config file` .

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

## DDL ロックの詳細

2種類のロックは、BDR でレプリケートされた DDL の正確性を強制します。

最初の種類はグローバル DDL ロックと呼ばれ、`ddl_locking = on` の場合にのみ使用されます。グローバル DDL ロックは、各 DDL ステートメントの実行中に他の DDL がクラスターで実行されるのを防ぎます。これにより、一般的な場合には完全な正確性が保証されますが、多くの単純な場合には厳しすぎます。BDR は、スキーマが変更さ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。言い換えれば、DDL の実行中は、異なるテーブルに影響を与える場合でも、ノード上の他の接続は別の 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には、DDL を実行する BDR ノードが BDR 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要求します。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、ノードも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（または少なくとも大部分のノード）がレプリケーションの遅延をあまり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。過半数のノードが同意すると、DDL の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る DDL ステートメントは、他のすべてのノードで同じ順序で実行されます。

DDL を実行しているノードがクラスター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 を実行したノードに対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は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。

2番目の種類は、リレーション DML ロックと呼ばれます。この種類のロックは、`ddl_locking = on` または `ddl_locking = dml` のいずれかで使用され、DDL ステートメントによってインフライト DML ステートメントが失敗する可能性があります。これらのエラーは、一意性制約、チェック制約、NOT NULL 制約などの制約を追加または変更した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。リレーション DML ロックにより、レプリケーションがエラーで停止する可能性のある変更がキューにある間は DDL が実行されません。

テーブルでグローバル DML ロックを取得するには、DDL を実行する BDR ノードが BDR 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し、そのテーブルへの保留中の変更がすべて排出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に 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、およびインデックスコマンドをブロックします。これは、通常は EXCLUSIVE LOCK 以上をとらないコマンドに対してグローバル DML ロックが保持されている場合でも当てはまります。

保留中の DML 操作が排出されるのを待機すると、時間がかかり、現在レプリケーションが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、行の表現と制約に影響を与えるスキーマの変更は、構成されたすべてのノードに到達でき、現在の書き込み速度に十分対応している場合にのみ実行できることを意味します。ノードのダウン中にこのような 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。

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。

**BDR システムでの DDL の実行** で説明されているように、ロックの動作は `bdr.ddl_locking`

パラメーターで指定されます。

- `ddl_locking = on` 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 = 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 BDR ファンクションは DDL を変更します。これらのファンクションには、DDL ロックの動作が適用されます。これは、各機能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が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、競合する DDL がなく、DDL が実行されるデータベースオブジェクトに競合する DML 操作がない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、データへ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ます。これらのシリアル化の失敗が発生すると、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 BDR 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、そのアクションがレプリケートされます。レプリケートされた関数の完全なリストは、

*DDL のように動作する BDR 関数* 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* に示されています。

## DDL の影響の最小化

任意のデータベースに対する優れた運用上のアドバイス、これらの点は BDR では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、多くの行の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ます。
- ALTER TABLE の場合、ADD CONSTRAINT を単独で使用するのではなく、ADD CONSTRAINT NOT VALID を使用した後に VALIDATE CONSTRAINT を使用して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します。これにより、確認を受け取るまでにかかなりの遅延が発生します。
- 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。

実行時間の長い 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から、各ノードで DDL ステートメントを個別に実行することです。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、この種の使用方法で自身をロックアウトしないように注意してください。

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

bdr.run\_on\_all\_nodes() テクニックを CREATE INDEX CONCURRENTLY で使用することをお勧めします。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ください。実行中の書き込みを防ぐため、実稼働システムでは CREATE INDEX を避けてください。REINDEX は 3.6 までのバージョンで複製されますが、BDR 3.7 以降では複製されません。REINDEX が保持する AccessExclusiveLocks のため、使用しないでください。

代わりに、PG12+または 2QPG11+で使用可能な REINDEX CONCURRENTLY （または reindexdb --concurrently）を使用してください。

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まとめる方が役立つ場合があるため、DDL ロックは 1 回だけ取得します。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停止している場合は、`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の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、グローバルロックを取得するための待機がキャンセルされるまでの時間を制限します。`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時長を制限し、`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容アイドル時間（ステートメント間の時間）を設定します。これらのタイムアウトは、値をゼロに設定することですべて無効にできます。

送信元ノードで DDL 操作がコミットされると、キャンセルまたは中止できません。BDR 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、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高速に保ちます。

### ダウンノードでの DDL の処理

グローバル DDL ロックを開始した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。ノードが復旧すると、保持しているすべてのグローバルロックを解放します。

長期間（または無期限に）ダウンしたままの場合は、ノードを BDR グループから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 1 つがダウンした場合、ロックを要求するそのコマンドの実行はノードが稼働しているかのように続行されます。

前述のように、グローバル DDL ロックはノードのマジョリティのみが応答する必要があるため、クラスターの一部がダウンしても、マジョリティが実行されていて到達可能である限り機能します。ただし、クラスター全体が使用可能でない限り、DML ロックを取得することはできません。

グローバル DDL またはグローバル DML ロックを使用すると、別のノードがダウンした場合、コマンドは正常に続行され、ロックが解放されます。

### ステートメント固有の DDL レプリケーションに関する問題

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。このようなコマンドは、`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、通常は許可されません。

**BDR 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ぎます。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。いくつかの制限付きでサポートされているステートメントは、**

`DDL statements with restrictions` で説明されています。

BDR で完全に許可されないコマンドは、禁止されている DDL ステートメントで説明されています。

ステートメントが BDR で許可されていない場合、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持つ列を追加する `ALTER TABLE` を実行することはできません。

ん。ただし、一般的には、動作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 0A0000) 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 BDR 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。

## BDR DDL コマンド処理マトリックス

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コマンド、およびレプリケート時に取得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。

| 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       | N          | N    |
| ALTER OPERATOR             | Y       | Y          | DDL  |
| ALTER OPERATOR CLASS       | Y       | Y          | DDL  |

次のページに続く

表 15 – 前のページからの続き

| Command                         | Allowed                    | Replicated | Lock                       |
|---------------------------------|----------------------------|------------|----------------------------|
| ALTER OPERATOR FAMILY           | Y                          | Y          | DDL                        |
| ALTER PACKAGE                   | Y                          | Y          | DDL                        |
| ALTER POLICY                    | Y                          | Y          | DDL                        |
| ALTER PROCEDURE                 | Y                          | Y          | DDL                        |
| ALTER PROFILE                   | Y                          | Y          | DDL                        |
| ALTER PUBLICATION               | Y                          | Y          | DDL                        |
| ALTER QUEUE                     | Y                          | Y          | DDL                        |
| ALTER QUEUE TABLE               | Y                          | Y          | DDL                        |
| ALTER REDACTION POLICY          | Y                          | Y          | DDL                        |
| ALTER RESOURCE GROUP            | Y                          | N          | N                          |
| ALTER ROLE                      | Y                          | Y          | DDL                        |
| ALTER ROUTINE                   | Y                          | Y          | DDL                        |
| ALTER RULE                      | Y                          | Y          | DDL                        |
| ALTER SCHEMA                    | Y                          | Y          | DDL                        |
| ALTER SEQUENCE                  | <a href="#">アーキテクチャの詳細</a> | 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                     | <a href="#">アーキテクチャの詳細</a> | Y          | <a href="#">アーキテクチャの詳細</a> |
| ALTER TABLESPACE                | Y                          | N          | N                          |
| 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                          |

次のページに続く



表 15 – 前のページからの続き

| Command                     | Allowed | Replicated                 | Lock |
|-----------------------------|---------|----------------------------|------|
| 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    |
| COMMENT                     | Y       | <a href="#">アーキテクチャの詳細</a> | 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       | N                          | N    |
| CREATE OPERATOR             | Y       | Y                          | DDL  |
| CREATE OPERATOR CLASS       | Y       | Y                          | DDL  |
| CREATE OPERATOR FAMILY      | Y       | Y                          | DDL  |

次のページに続く

表 15 – 前のページからの続き

| Command                          | Allowed                    | Replicated | Lock |
|----------------------------------|----------------------------|------------|------|
| CREATE PACKAGE                   | Y                          | Y          | DDL  |
| CREATE PACKAGE BODY              | Y                          | Y          | DDL  |
| CREATE POLICY                    | Y                          | Y          | DML  |
| CREATE PROCEDURE                 | Y                          | Y          | DDL  |
| CREATE PROFILE                   | Y                          | Y          | DDL  |
| 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                  | <a href="#">アーキテクチャの詳細</a> | Y          | DDL  |
| CREATE SERVER                    | Y                          | Y          | DDL  |
| CREATE STATISTICS                | Y                          | Y          | DDL  |
| CREATE SUBSCRIPTION              | Y                          | Y          | DDL  |
| CREATE SYNONYM                   | Y                          | Y          | DDL  |
| CREATE TABLE                     | <a href="#">アーキテクチャの詳細</a> | Y          | DDL  |
| CREATE TABLE AS                  | <a href="#">アーキテクチャの詳細</a> | Y          | DDL  |
| CREATE TABLESPACE                | Y                          | N          | N    |
| 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  |

次のページに続く



表 15 – 前のページからの続き

| Command                      | Allowed | Replicated | Lock |
|------------------------------|---------|------------|------|
| CREATE TYPE BODY             | Y       | Y          | DDL  |
| CREATE USER MAP-<br>PING     | Y       | Y          | DDL  |
| CREATE VIEW                  | Y       | Y          | DDL  |
| DEALLOCATE                   | Y       | N          | N    |
| DEALLOCATE ALL               | Y       | N          | N    |
| DECLARE CURSOR               | Y       | N          | N    |
| DISCARD                      | Y       | N          | N    |
| DISCARD ALL                  | Y       | N          | N    |
| DISCARD PLANS                | Y       | N          | N    |
| DISCARD SE-<br>QUENCES       | Y       | N          | N    |
| DISCARD TEMP                 | Y       | N          | N    |
| DO                           | Y       | N          | N    |
| DROP ACCESS<br>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<br>LINK        | Y       | Y          | DDL  |
| DROP DIRECTORY               | Y       | Y          | DDL  |
| DROP DOMAIN                  | Y       | Y          | DDL  |
| DROP EVENT TRIG-<br>GER      | Y       | Y          | DDL  |
| DROP EXTENSION               | Y       | Y          | DDL  |
| DROP FOREIGN DATA<br>WRAPPER | Y       | Y          | DDL  |
| DROP FOREIGN TA-<br>BLE      | Y       | Y          | DDL  |
| DROP FUNCTION                | Y       | Y          | DDL  |
| DROP INDEX                   | Y       | Y          | DDL  |
| DROP LANGUAGE                | Y       | Y          | DDL  |
| DROP MATERIALIZED<br>VIEW    | Y       | N          | N    |
| DROP OPERATOR                | Y       | Y          | DDL  |
| DROP OPERATOR<br>CLASS       | Y       | Y          | DDL  |
| DROP OPERATOR<br>FAMILY      | Y       | Y          | DDL  |

次のページに続く

表 15 – 前のページからの続き

| Command                        | Allowed | Replicated | Lock       |
|--------------------------------|---------|------------|------------|
| DROP OWNED                     | Y       | Y          | DDL        |
| DROP PACKAGE                   | Y       | Y          | DDL        |
| DROP PACKAGE BODY              | Y       | Y          | DDL        |
| DROP POLICY                    | Y       | Y          | DDL        |
| DROP PROCEDURE                 | Y       | Y          | DDL        |
| DROP PROFILE                   | Y       | Y          | DDL        |
| DROP PUBLICATION               | Y       | Y          | DDL        |
| DROP QUEUE                     | Y       | Y          | DDL        |
| DROP QUEUE TABLE               | Y       | Y          | DDL        |
| 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       | N          | N          |
| 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          |

次のページに続く

表 15 – 前のページからの続き

| Command                   | Allowed                    | Replicated                 | Lock                       |
|---------------------------|----------------------------|----------------------------|----------------------------|
| GRANT                     | Y                          | <a href="#">アーキテクチャの詳細</a> | 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                          | <a href="#">アーキテクチャの詳細</a> |
| 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                          | N                          | N                          |
| REINDEX                   | Y                          | N                          | N                          |
| RELEASE                   | Y                          | N                          | N                          |
| RESET                     | Y                          | N                          | N                          |
| REVOKE                    | Y                          | <a href="#">アーキテクチャの詳細</a> | DDL                        |
| REVOKE ROLE               | Y                          | Y                          | DDL                        |
| ROLLBACK                  | Y                          | N                          | N                          |
| ROLLBACK PREPARED         | Y                          | N                          | N                          |
| SAVEPOINT                 | Y                          | N                          | N                          |
| SECURITY LABEL            | Y                          | <a href="#">アーキテクチャの詳細</a> | DDL                        |
| SELECT INTO               | <a href="#">アーキテクチャの詳細</a> | Y                          | DDL                        |
| SET                       | Y                          | N                          | N                          |
| SET CONSTRAINTS           | Y                          | N                          | N                          |
| SHOW                      | Y                          | N                          | N                          |
| START TRANSACTION         | Y                          | N                          | N                          |
| TRUNCATE TABLE            | Y                          | <a href="#">アーキテクチャの詳細</a> | <a href="#">アーキテクチャの詳細</a> |
| UNLISTEN                  | Y                          | N                          | N                          |
| VACUUM                    | Y                          | N                          | N                          |

## ALTER シーケンス

通常、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 シーケンスではサポートされていません (のみ)。ALTER SEQUENCE ... SET SCHEMA は、galloc シーケンスではサポートされていません (のみ)。

## ALTER テーブル

通常、ALTER TABLE コマンドが許可されます。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 で許可されていないコマンド

ALTER TABLE の一部のバリエーションは現在 BDR ノードで許可されていません。

- ADD COLUMN ... DEFAULT (non-immutable expression) —現在、異なるノードで異なるデータが得られるため、これは許可されません。推奨される回避策については、

[列の追加](#) を参照してください。

- ADD CONSTRAINT ... EXCLUDE —除外制約は現在サポートされていません。除外制約は非同期システムではあまり意味がなく、再生できない変更につながります。
- ALTER TABLE ... SET WITH[OUT] OIDS — CREATE TABLE と同じ理由でサポートされていません。
- ALTER COLUMN ... SET STORAGE external —列がテーブルのレプリカ ID の列の場合、拒否されます。
- RENAME —自動パーティション分割されたテーブルの名前を変更できません。
- SET SCHEMA —自動パーティション分割テーブルのスキーマを設定できません。
- ALTER COLUMN ... TYPE —コマンドによってテーブル全体が書き換えられる場合、列の型の変更はサポートされていません。これは、変更がバイナリ強制ではない場合に発生します。バイナリの強制的な変更は、一方向にのみ許可される場合があります。たとえば、VARCHAR(128) から VARCHAR(256) への変更はバイナリ強制可能であるため許可されますが、VARCHAR(256) から VARCHAR(128) への変更はバイナリ強制ではないため、通常は許可されません。レプリケートされない ALTER COLUMN ... TYPE は、列が新しい型に自動的にキャスト可能な場合に許可されます (USING 句が含まれていません)。次に例を示します。テーブルの書き換えは、大規模なテーブルで長期間 AccessExclusiveLock を保持するため、いずれの場合も、このようなコマンドは高可用性データベースでは実行できない可能性があります。推奨される回避策については、

[列のタイプを変更する](#) を参照してください。

- ALTER TABLE ... ADD FOREIGN KEY —現在のユーザーに参照先テーブルを読み取る権限がない場合、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、は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;
```

BDR 3.7.4 以降、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。詳細については、[CONSTRAINT の追加](#) を参照してください。

## ALTER TABLE ロック

ALTER TABLE の次のバリエーションは、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
- 書き換えが必要ない場合は ALTER TABLE ... ALTER COLUMN ... TYPE
- 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エーションは、変更されるテーブルで DML ロックを取得します。ALTER TABLE の一部のバリエーションには、以下に示す制限があります。

## ALTER TABLE の例

この次の例は、型の変更がバイナリ強制的であるため、テーブルの書き換えが発生しないためです。カタログのみの変更として実行されます。

```
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) への変更はバイナリ強制ではないため、この変更を加えてコマンドを元に戻すことはできません。

```
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(20);
```

回避策については、[制限され DDL の回避策](#) を参照してください。

一般および非レプリケート環境で可能なさまざまな種類の ALTER TABLE ... ALTER COLUMN TYPE (ATCT) 操作にコンテキストを提供すると役立ちます。

一部の ATCT 操作では、基になる列タイプのメタデータのみが更新され、基になるテーブルデータを書き換える必要はありません。これは通常、既存の列の型とターゲットの型がバイナリ強制型である場合に当てはまります。例：

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

バイナリ強制性のため、列の型を 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;
```

ただし、col2 のサイズを減らしたい場合、それは基になるテーブルデータの書き換えにつながります。通常、テーブルの書き換えは制限されています。

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ERROR: ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect
↳ replicated tables on a BDR node
```

非テキスト型の例を示すには、INTEGER 型の上記の col3 を検討してください。SMALLINT または BIGINT に変換しようとする ATCT 操作は、上記と同様に失敗します。

```
ALTER TABLE sample ALTER COLUMN col3 TYPE bigint;
ERROR: ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect
↳ replicated tables on a BDR node
```

これらの両方の失敗したケースでは、現在の型からターゲット型への自動割り当てキャストがあります。ただし、バイナリ強制はなく、基になるテーブルデータの書き換えが発生します。

このような場合、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 1 つずつ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。DDL がレプリケートされず、列の型が自動的にキャスト可能なものに変更されたとします。次に、変更を実行するノードでローカルに書き換えを許可し、この同じテーブルの他のノードでの同時アクティビティを許可できます。次に、すべてのノードでこのレプリケートされない ATCT 操作を 1 つずつ繰り返して、EDB Postgres 分散クラスター全体で列タイプを必要に応じて変更します。これには書き換えが含まれるため、アクティビティはまだ短期間 DML ロックを取得するため、クラスター全体が使用可能である必要があります。これらの詳細を設定したら、次のようにレプリケートされない変更アクティビティのローリングアップグレードを実行できます。

```
- - foreach node in EDB Postgres Distributed cluster do:
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
RESET bdr.ddl_replication;
- - done
```

多くのデータ型で自動割り当てキャストを使用できるため、このローカルのレプリケートされない ATCT 操作はさまざまな変換をサポートします。また、USING 句を使用する ATCT 操作は、自動割り当てキャストがないために失敗する可能性があります。この例は、自動割り当てキャストを使用したいいくつかの一般的な変換を示しています。

```
- - foreach node in EDB Postgres Distributed cluster do:
SET bdr.ddl_replication TO FALSE;
ATCT operations to-from {INTEGER, SMALLINT, BIGINT}
ATCT operations to-from {CHAR(n), VARCHAR(n), VARCHAR, TEXT}
ATCT operations from numeric types to text types
RESET bdr.ddl_replication;
- - done
```

この例は、レプリケートされない環境で許可される可能性のある ATCT 操作の完全なリストではありません。すべての ATCT 操作が機能するわけではありません。DDL レプリケーションを無効にしても、自動割り当てをできない場合は失敗します。したがって、数値型からテキスト型への変換はレプリケートされない環境では機能しますが、テキスト型から数値型への変換は失敗します。

```
SET bdr.ddl_replication TO FALSE;
- - conversion from BIGINT to TEXT works
ALTER TABLE sample ALTER COLUMN col3 TYPE TEXT;
- - conversion from TEXT back to BIGINT fails
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
ERROR: ALTER TABLE ... ALTER COLUMN TYPE which cannot be automatically cast to new_
→type may not affect replicated tables on a BDR node
RESET bdr.ddl_replication;
```

レプリケートされない環境での ATCT 操作はさまざまな型変換をサポートしていますが、基になるテーブルデータに新しいデータ型に割り当てることができない値が含まれている場合、書き換えが失敗する場合があります。たとえば、列の現在の型が VARCHAR(256) であり、レプリケートされない ATCT 操作を使用して VARCHAR(128) に変換するとします。テーブルに 128 バイトを超える既存のデータがある場合、書き換え操作はローカルで失敗します。

```
INSERT INTO sample VALUES (1, repeat(a, 200), 10);
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(128);
INFO:  in rewrite
ERROR:  value too long for type character varying(128)
```

基になるテーブルのデータが新しい型の特性を満たしている場合、書き換えは成功します。ただし、レプリケートされないローリングデータ型のアップグレードをまだ実行していない他のノードがこのローカル ATCT 操作に同時に 128 バイトを超える新しいデータを導入すると、レプリケーションが失敗する可能性があります。これにより、クラスター内のレプリケーションが停止します。したがって、これらのレプリケートされないローリングデータ型アップグレード操作を実行するときは、データベースおよびアプリケーションレベルでのデータ型の制限と特性に注意してください。このような ATCT 操作は、制御された完全に認識した DBA 環境で実行およびテストすることを強くお勧めします。これらの ATCT 操作は非対称であり、失敗した特定の変更をバックアウトすると、テーブルの書き換えに時間がかかる場合があります。

また、トランザクションブロックで暗黙的なキャスト可能な ALTER アクティビティを実行することはできません。

## ALTER タイプ

ALTER TYPE は複製されますが、PostgreSQL はこれらの依存関係を記録しないため、そのデータ型を使用するすべてのテーブルにグローバル DML ロックが適用されるわけではありません。

[制限され DDL の回避策](#) を参照してください。

## コメント

COMMENT ON のすべてのバリエーションが許可されていますが、COMMENT ON TABLESPACE/DATABASE/LARGE OBJECT は複製されません。

## シーケンスを作成する

通常、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。

## テーブルを作成する

通常、CREATE TABLE はサポートされていますが、BDR ノードでは CREATE TABLE WITH OIDS は許可されていません。



## 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。

### 説明する

通常、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複製ルールに従います。

### EXPLAIN ANALYZE ロック

EXPLAIN ANALYZE は、分析されたステートメントのロック規則に従います。

### 付与と取り消し

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 は複製されません。

### ロックテーブル

LOCK TABLE は複製されませんが、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 のすべてのバリエーションが許可されていますが、SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT は複製されません。

## TRUNCATE レプリケーション

TRUNCATE コマンドは、DDL ステートメントとしてではなく、DML として複製されます。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 と同じように複製されませんが、`bdr.truncate_locking` が `on` に設定されていると、グローバル DML ロックを取得できます。

## ロール操作ステートメント

ユーザは PostgreSQL インスタンスのグローバルオブジェクトです。つまり、BDR は個々のデータベースレベルで動作します。これは、ロール操作ステートメントの処理には追加の考慮が必要であることを意味します。

BDR 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。ロールは、同じ付与、パスワードなどを持つ必要はありませんが、存在している必要があります。

`bdr.role_replication` が有効になっており（デフォルト）、ロール操作ステートメントが BDR 対応のデータベースで実行されている場合、BDR はロール操作ステートメントを複製します。

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で、または `bdr.replicate_ddl_command(...)` によって明示的に複製されたすべてのロールの変更（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 BDR 対応データベースが `bdr.role_replication = on` を持つ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。

ロールのレプリケーションがオフになっている場合、管理者は、1つのノードで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、BDR 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。

---

**注釈:** BDR は、BDR 対応の PostgreSQL インスタンスの BDR 非対応のデータベースで実行される場合、ロール管理ステートメントをキャプチャおよび複製しません。たとえば、DB 'bdrdb' (bdr グループメンバー) と 'postgres' (ベア DB)、および 'bdr.role\_replication = on' がある場合、*bdrdb* で実行される 'CREATE USER' は複製されますが、*postgres* で実行されるものは複製されません。

---

## 制限され DDL の回避策

BDR DDL 操作処理の制限の一部を回避できます。多くの場合、操作を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、過度のロックが必要な結果が得られます。

## CONSTRAINT 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

これらの手順は、2つの異なるトランザクションで実行します。これらの手順はどちらもテーブルでのみ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、制約を検証するには、BDR は次のことを保証する必要があります。

- クラスター内のすべてのノードが ADD CONSTRAINT コマンドを参照します。
- 制約を適用するノードは、それらのノードに NOT VALID 制約を作成する前に、他のすべてのノードからレプリケーションを変更します。

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を起動する必要はありませんが、すべてのノードが ALTER TABLE .. ADD CONSTRAINT ... NOT VALID コマンドを適用して十分に進行したことは依然として必要です。BDR は、制約を検証する前に一貫した状態に到達するのを待ちます。

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。

## 列の追加

揮発性のデフォルトを持つ列を追加するには、別のトランザクションで次のコマンドを実行します。

```
ALTER TABLE mytable ADD COLUMN newcolumn coltype; -- Note the lack of DEFAULT or NOT NULL
--> NULL

ALTER TABLE mytable ALTER COLUMN newcolumn DEFAULT volatile-expression;

BEGIN;
SELECT bdr.global_lock_table(mytable);
UPDATE mytable SET newcolumn = default-expression;
COMMIT;
```

このアプローチは、スキーマの変更と行の変更を BDR が実行できる個別のトランザクションに分割し、BDR グループ内のすべてのノードで一貫したデータを生成します。

最良の結果を得るには、更新をチャンクにまとめて、一度に数万または数十万を超える行を更新しないようにします。これは、トランザクションが埋め込まれた PROCEDURE を使用して実行できます。

変更の最後のバッチは、テーブルでグローバル DML ロックを取得するトランザクションで実行する必要があります。そうしないと、他のノードのテーブルに同時に挿入される行を見落とす可能性があります。

必要に応じて、UPDATE の終了後に ALTER TABLE mytable ALTER COLUMN newcolumn NOT NULL; を実行できます。

## 列のタイプを変更する

PostgreSQL は、次の例のように、回避できる場合にテーブルの書き換えを発生させます。

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

制限をデータ型の変更ではなくテーブル制約にすることにより、このステートメントを書き換えて、テーブルの書き換えを回避できます。必要に応じて、後続のコマンドで制約を検証して、長いロックを回避できます。

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(128));
ALTER TABLE foo
  ALTER COLUMN description TYPE varchar,
  ADD CONSTRAINT description_length_limit CHECK (length(description) <= 20) NOT VALID;
ALTER TABLE foo VALIDATE CONSTRAINT description_length_limit;
```

検証が失敗した場合、失敗した行のみを UPDATE できます。length() を使用して TEXT および VARCHAR、または scale() を使用して NUMERIC データ型でこの手法を使用できます。

列の種類を変更する一般的な場合、最初に目的の種類の列を追加します。

```
ALTER TABLE mytable ADD COLUMN newcolumn newtype;
```

BEFORE INSERT OR UPDATE ON mytable FOR EACH ROW ..として定義されたトリガーを作成します。これは、テーブルへの新しい書き込みで新しい列が自動的に更新されるように、NEW.newcolumn を NEW.oldcolumn に割り当てます。

UPDATE テーブルをバッチで実行し、トランザクションが埋め込まれた PROCEDURE を使用して oldcolumn の値を newcolumn にコピーします。作業をバッチ処理すると、大きなテーブルの場合、レプリケーションの遅延を削減できます。ID の範囲など、お好みの方法で更新できます。または、小さなテーブルの場合、テーブル全体を 1 回のパスで更新できます。

CREATE INDEX ... 新しい列に必要なインデックス。ロック期間を削減するには、各ノードで DDL レプリケーションなしで個別に実行しても安全です。

ALTER 必要に応じて、NOT NULL および CHECK 制約を追加する列。

1. BEGIN トランザクション。
1. 追加したトリガーを DROP します。
  1. ALTER TABLE は、列に必要な DEFAULT を追加します。
1. 古い列を DROP 。
  1. ALTER TABLE mytable RENAME COLUMN newcolumn TO oldcolumn 。
  2. COMMIT 。

**注釈:** 列を削除するため、テーブルに依存するビュー、プロシージャなどの再作成が必要になる場合があります。列を CASCADE 削除する場合は、それを参照しているものをすべて再作成する必要があるので注意してください。

## 他のタイプの変更

ALTER TYPE ステートメントは複製されますが、影響を受けるテーブルはロックされません。

この DDL を使用する場合は、新しいタイプを使用する前に、すべてのノードでステートメントが正常に実行されたことを確認してください。bdr.wait\_slot\_confirm\_lsn() 関数を使用してこれを実現できます。

この例では、DML ステートメントで新しい値を使用する前に、DDL がすべてのノードに書き込まれるようにします。

```
ALTER TYPE contact_method ADD VALUE email;
SELECT bdr.wait_slot_confirm_lsn(NULL, NULL);
```

## DDL のように動作する BDR 関数

次の BDR 管理機能は DDL のように機能します。これは、DDL レプリケーションがアクティブであり、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、個々の機能の文書を参照してください。

### レプリケーションセット管理

- `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`

### コンフリクトマネジメント

- `bdr.alter_table_conflict_detection`
- `bdr.column_timestamps_enable`
- `bdr.column_timestamps_disable`

### 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`

## Security and roles

BDR 拡張機能を作成できるのはスーパーユーザーのみです。ただし、必要に応じて、`pgextwlist` 拡張機能を設定し、非スーパーユーザーが BDR 拡張機能を作成できるように構成できます。

BDR 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。BDR に必要な権限は、PostgreSQL のデフォルト/事前定義ロールと同様に名前付けされた次のデフォルト/事前定義ロールに分割されます。

- `bdr_superuser` —すべての BDR テーブルとファンクションへのアクセスを持つ最も高い特権のロール。
- `bdr_read_all_stats` —BDR の状態を理解するのに十分な、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` —現時点では、`bdr_read_all_stats` と同じ。後日延長。

- `bdr_application` - BDR を実行するアプリケーションに必要な最小限の権限。
- `bdr_read_all_conflicts` — `bdr.conflict_history` のすべての競合を表示できます。

これらの 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。詳細については、BDR default roles を参照してください。

BDR を管理するために、管理者がユーザデータにアクセスする必要はありません。

競合を保護するための手配については、[テーブルへの競合のログ](#) で説明されています。

`BDR.conflict_history_summary` ビューを使用して競合を監視できます。

## 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、常に BDR によるレプリケーションから除外されます。

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。

## BDR の関数と演算子

すべての BDR 関数は `bdr` スキーマで公開されています。これらの関数への呼び出しは、`search_path` に `bdr` を配置するのではなく、スキーマで修飾する必要があります。

すべての BDR 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。

BDR default 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、参加のいずれかの側のノードの BDR（したがって BDR カタログ）のバージョンが完全に同じでなくても、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に対する特権が明示的に付与された場合、結合されるノードから抽出された対応する GRANT ステートメントが適用されないため、`bdr.join_node_group()` プロシージャが失敗する可能性があります。

## ロール管理

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 USER および CREATE ROLE コマンドは、BDR が実行されており、bdr.role\_replication がオンになっているデータベースで実行された場合、自動的に複製されます。ただし、これらのコマンドが同じ PostgreSQL インスタンス内の他のデータベースで実行された場合、それらのユーザーが BDR データベースに対する権限を持っている場合でも、レプリケートされません。

新しい BDR ノードが BDR グループに参加しても、bdr\_init\_physical を使用してノードを追加しない限り、既存のユーザーは自動的にコピーされません。これは意図的なものであり、重要なセキュリティ機能です。PostgreSQL では、ユーザーは複数のデータベースにアクセスできます。デフォルトでは任意のデータベースにアクセスできます。BDR は、どのユーザーがどのデータベースにアクセスするかを知らないため、新しいノードにコピーするユーザーを安全に決定できません。

PostgreSQL では、次のコマンドですべてのユーザーをダンプできます。

```
pg_dumpall --roles-only > roles.sql
```

次に、ファイル roles.sql を編集して、新しく作成したノードで再実行する前に不要なユーザーを削除できます。ID とアクセス管理ソリューション (IAM) によっては他のメカニズムも可能ですが、現時点では自動化されていません。

## ロールとレプリケーション

ユーザーが実行した DDL 変更は、各ノードで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、ターゲットノードでテーブル所有ユーザーとして複製されます。各ノードでテーブルを同じユーザーが所有することをお勧めしますが、強制はしません。

テーブル A が node1 でユーザー X が所有し、node2 でユーザー Y が所有しているとします。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。一部のノードにはさまざまなユースケースがあるため、これを許可しますが、セキュリティ管理者がこの状況を計画および監査できるように警告します。

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再適用せずに変更がレプリケートされます。これは、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。これが望ましくない場合は、すべての行の複製を回避する 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を同一にするか、少なくとも互換性があるようにすることはお勧めしますが、強制はしません。

ユーザ bdr\_superuser は、BDR のレプリケーションを制御し、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。bdr\_superuser には個々のテーブルに対する特権は必要なく、これもお勧めしません。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、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。



## 接続ロール

新しい BDR ノードを割り当てるとき、`bdr.create_node` の `local_dsn` 引数および `bdr.join_node_group` の `join_target_dsn` に DSN で指定されたユーザーは、データベースオブジェクトを参照、作成、管理するために頻繁に使用されます。

BDR 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権限昇格攻撃を防ぐように注意深く作成されています。

攻撃対象領域をさらに削減するには、上記の DSN でより制限されたユーザーを指定します。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。

- `REPLICATION` 属性を持っていること。
- 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。
- 直接または所有者ロールの権限から、レプリケートするすべてのデータベースオブジェクトを所有しています。

すべてのノードが参加したら、権限をさらに次のように減らして、DML および DDL レプリケーションを許可できます。

- `REPLICATION` 属性を持っていること。
- `bdr_superuser` ロールを継承します。

## 特権制限

BDR は追加の制限を適用し、`TRIGGER` または `REFERENCES` 権限のみに依存する `DDL` の使用を効果的に防止します。

`GRANT ALL` は引き続き `TRIGGER` 特権と `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、ユーザーが参照先のテーブルで `SELECT` 権限を持っている場合、または参照先のテーブルで現在のユーザーがバイパスできない `RLS` 制限を有効にしている場合にのみサポートされます。

したがって、BDR で外部キーを作成するには `REFERENCES` 特権では不十分です。`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、通常は役に立ちません。通常、正常に使用するには高すぎます。

## トリガー

PostgreSQL では、テーブルの所有者と TRIGGER 権限を付与された人の両方が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、BDR でテーブル所有者として実行され、セキュリティ問題が発生する可能性があります。TRIGGER 権限はめったに使用されず、PostgreSQL コアチームは「個別の TRIGGER 権限は廃止されたと考えています」と述べています。

BDR は、テーブルでトリガーを作成できるユーザーに関するより厳密なルールを使用することにより、この問題を軽減します。

- スーパーユーザー
- bdr\_superuser
- テーブルの所有者は、PostgreSQL と同じルールに従ってトリガーを作成できます（トリガーで使用する関数の EXECUTE 権限が必要です）。
- テーブルに対する TRIGGER 権限を持つユーザーは、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てトリガーを作成し、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（ここでもファンクションの EXECUTE 権限が必要です）。したがって、テーブルとファンクションの両方が同じ所有者であり、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、そのユーザーはこれ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ます機能。
- 表に対する TRIGGER 権限を持っているユーザーは、EXECUTE 権限を持っている場合、SECURITY DEFINER 句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できます。この句は、標準の PostgreSQL と BDR の両方で、ファンクションを常にファンクションの所有者のコンテキストで実行します。

このロジックは、PostgreSQL では、トリガーの所有者は作成したユーザーではなく、そのトリガーによって使用される関数の所有者であるという事実に基づいています。

同じルールが既存のテーブルに適用され、テーブルの所有者が所有していないトリガーが既存のテーブルにあり、SECURITY DEFINER ファンクションを使用しない場合、レプリケーションセットに追加できません。

これらのチェックは BDR 3.6.19 で追加されました。以前のバージョンの動作に依存するアプリケーションは、`bdr.backwards_compatibility` を 30618（またはそれ以下）に設定して、以前のバージョンのように動作できます。

BDR レプリケーションの適用は、システムレベルのデフォルトの `search_path` のみを使用します。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関数は、他の `search_path` 設定を想定しており、適用時に実行される場合があります。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（常にオブジェクトへの完全修飾参照、たとえば、`schema.objectname` を使用します）、影響を受ける関数に `ALTER FUNCTION ... SET search_path = ...` を使用して関数の検索パスを設定します。

## BDR のデフォルト/事前定義されたロール

BDR 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 拡張機能をデータベースから削除した後も、ロールは引き続き存在するため、必要に応じて手動で削除する必要があります。これにより、BDR を同じ PostgreSQL インスタンスの複数のデータベースで問題なく使用できます。

GRANT ROLE DDL ステートメントは BDR レプリケーションに参加しません。したがって、これをクラスターの各ノードで実行します。

### **bdr\_superuser**

- スキーマ BDR のすべてのテーブルのすべての権限
- スキーマ BDR のすべてのルーチンのすべての権限

### **bdr\_read\_all\_stats**

に対する SELECT 特権

- `bdr.conflict_history_summary`
- `bdr.ddl_epoch`
- `bdr.ddl_replication`
- `bdr.global_consensus_journal_details`
- `bdr.global_lock`
- `bdr.global_locks`
- `bdr.local_consensus_state`
- `bdr.local_node_summary`
- `bdr.node`
- `bdr.node_catchup_info`
- `bdr.node_conflict_resolvers`
- `bdr.node_group`
- `bdr.node_local_info`
- `bdr.node_peer_progress`
- `bdr.node_slots`
- `bdr.node_summary`
- `bdr.replication_sets`
- `bdr.sequences`
- `bdr.state_journal_details`

- `bdr.stat_relation`
- `bdr.stat_subscription`
- `bdr.subscription`
- `bdr.subscription_summary`
- `bdr.tables`
- `bdr.worker_errors`

に対する EXECUTE 特権

- `bdr.bdr_version`
- `bdr.bdr_version_num`
- `bdr.conflict_resolution_to_string`
- `bdr.conflict_type_to_string`
- `bdr.decode_message_payload`
- `bdr.get_global_locks`
- `bdr.get_raft_status`
- `bdr.get_relation_stats`
- `bdr.get_slot_flush_timestamp`
- `bdr.get_sub_progress_timestamp`
- `bdr.get_subscription_stats`
- `bdr.peer_state_name`
- `bdr.show_subscription_status`

### **bdr\_monitor**

`bdr_read_all_stats` のすべての特権に加えて

に対する EXECUTE 特権

- `bdr.monitor_group_versions`
- `bdr.monitor_group_raft`
- `bdr.monitor_local_replslots`

## bdr\_application

に対する EXECUTE 特権

- column\_timestamps データ型のすべての関数
- CRDT データ型のすべての関数
- 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

これらの機能の多くには、使用する前に必要な追加の権限があります。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については、特定の機能ごとに説明します。

## bdr\_read\_all\_conflicts

BDR は、競合を `bdr.conflict_history` テーブルに記録します。競合はテーブルの所有者のみに表示されるため、競合の履歴を読み取るために追加の権限は必要ありません。すべてのテーブルの競合を確認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。

## 検証

BDR は、次のツールとアプローチを使用して検証されました。

## コベリティ

Coverity Scan は、次のルールとコーディング標準を使用して、脆弱性に対するカバレッジを提供する BDR スタックを検証するために使用されました。

- ミスラ C
- ISO 26262
- ISO / IEC TS 17961
- OWASP トップ 10
- CERT C
- CWE トップ 25
- AUTOSAR

## CIS ベンチマーク

BDR スタックの検証には、2019 年 12 月 19 日 CIS PostgreSQL Benchmark v1 が使用されました。TPAexec のオプションとして使用可能な `cis_policy.yml` 構成を使用すると、Scored テストで次の結果が得られます。

|     | Result | Description                      |
|-----|--------|----------------------------------|
| 1.4 | PASS   | systemd サービスファイルが有効になっていることを確認する |
| 1.5 | PASS   | データクラスターが正常に初期化されたことを確認する        |
| 2.1 | PASS   | ファイルのアクセス許可マスクが正しいことを確認します       |

[次のページに続く](#)

表 16 – 前のページからの続き

|        | Result | Description                                 |
|--------|--------|---------------------------------------------|
| 2.2    | PASS   | PostgreSQL pg_wheel グループメンバーシップが正しいことを確認します |
| 3.1.2  | PASS   | ログの宛先が正しく設定されていることを確認します                    |
| 3.1.3  | PASS   | ログコレクターが有効になっていることを確認する                     |
| 3.1.4  | PASS   | ログファイルの宛先ディレクトリが正しく設定されていることを確認します          |
| 3.1.5  | PASS   | ログファイルのファイル名パターンが正しく設定されていることを確認します         |
| 3.1.6  | PASS   | ログファイルのアクセス許可が正しく設定されていることを確認します            |
| 3.1.7  | PASS   | 「log_truncate_on_rotation」が有効になっていることを確認します |
| 3.1.8  | PASS   | ログファイルの最大存続時間が正しく設定されていることを確認します            |
| 3.1.9  | PASS   | 最大ログファイルサイズが正しく設定されていることを確認します              |
| 3.1.10 | PASS   | 正しい syslog 機能が選択されていることを確認します               |
| 3.1.11 | PASS   | PostgreSQL syslog メッセージのプログラム名が正しいことを確認します  |
| 3.1.14 | PASS   | 「debug_print_parse」が無効になっていることを確認します        |
| 3.1.15 | PASS   | 「debug_print_rewriting」が無効になっていることを確認します    |
| 3.1.16 | PASS   | 「debug_print_plan」が無効になっていることを確認します         |
| 3.1.17 | PASS   | 「debug_pretty_print」が有効になっていることを確認します       |

次のページに続く

表 16 – 前のページからの続き

|        | Result     | Description                                            |
|--------|------------|--------------------------------------------------------|
| 3.1.18 | PASS       | 「log_connections」が有効になっていることを確認します                     |
| 3.1.19 | PASS       | 「log_disconnections」が有効になっていることを確認します                  |
| 3.1.21 | PASS       | 「log_hostname」が正しく設定されていることを確認します                      |
| 3.1.23 | PASS       | 「log_statement」が正しく設定されていることを確認します                     |
| 3.1.24 | PASS       | 「log_timezone」が正しく設定されていることを確認します                      |
| 3.2    | PASS       | PostgreSQL Audit Extension (pgAudit) が有効になっていることを確認します |
| 4.1    | PASS       | sudo が正しく構成されていることを確認します                               |
| 4.2    | PASS       | 過剰な管理者権限が取り消されていることを確認する                               |
| 4.3    | PASS       | 過剰な機能権限が取り消されていることを確認する                                |
| 4.4    | PASS       | テスト済み 過剰な DML 権限が取り消されていることを確認する                       |
| 5.2    | Not Tested | 「host」 TCP/IP ソケットを介したログインが正しく構成されていることを確認します          |
| 6.2    | PASS       | 「バックエンド」ランタイムパラメーターが正しく構成されていることを確認します                 |
| 6.7    | Not Tested | FIPS 140-2 OpenSSL 暗号化が使用されていることを確認する                  |
| 6.8    | PASS       | SSL が有効になっており、正しく構成されていることを確認します                       |

次のページに続く



表 16 – 前のページからの続き

|     | Result | Description                 |
|-----|--------|-----------------------------|
| 7.3 | PASS   | WAL アーカイブが構成され機能していることを確認する |

テスト 5.2 は手動で監査すると PASS になりますが、自動テストはありません。

テスト 6.7 は、CentOS を使用したデフォルトの展開で成功しますが、Debian パリアントには追加のパッケージが必要です。

## Conflicts

BDR は、アクティブ/アクティブまたはマルチマスターの DBMS です。非同期で使用すると、複数の異なるノードから同じ行または関連する行に書き込むと、標準データ型を使用するときに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 BDR が発生時に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、BDR がアプリケーションに競合を解決する方法に関するさまざまな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が選択され、もう一方は破棄されます。たとえば、2つの競合する変更についてコミットのタイムスタンプを比較し、新しい方を保持する場合があります。このアプローチにより、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が確立されます。

**列レベルの競合解決の有効化と無効化** で説明されているように、競合処理は構成可能です。

*Stream triggers* で説明されている競合トリガーを使用して、テーブルごとに競合を検出して処理できます。

CLCD で説明されているように、列レベルの競合検出と解決は BDR で使用できます。

競合を回避したい場合は、BDR でこれらの機能を使用できます。

- `:ref:`bdr.crdt_handlers`<bdr.crdt_handlers>` で説明されている競合のないデータ型 (CRDT)。
- Eager Replication で説明されている積極的な複製。

デフォルトでは、すべての競合は `bdr.conflict_history` に記録されます。競合が発生する可能性がある場合、表の所有者は競合を監視して回避する方法を分析するか、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 によるデータ検証** ツールを使用して、発散を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ぎます。これらは、サーバーが互いに非常に近い場合は適切に実行できますが、許容可能なパフォーマンスのために非常に低いレイテンシー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。BDR は、可能な場合は競合を回避しますが、特定の種類の競合の発生を許可し、発生したときに解決するという楽観的なアプローチを提唱しています。

## 競合はどのように発生するか

ノード間の競合は、関連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する場合に発生しない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行った別のノードに適用された場合、無効です。

BDR 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リプレイするため、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の間に競合がある場合、リプレイ操作が失敗する可能性があります。

Postgres には UNIQUE インデックス、SEQUENCE 操作、行と関係のロック、SERIALIZABLE 依存関係の追跡など、それを防ぐための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。

BDR には、分散トランザクションマネージャーまたはロックマネージャーがありません。これが、レイテンシーとネットワークパーティションで優れたパフォーマンスを発揮する理由の一部です。その結果、デフォルトの遅延レプリケーションを使用すると、異なるノード上のトランザクションが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と競合を完全に回避できます。そのため、BDR はこれが重要な場合に Eager Replication も提供します。

## 紛争の種類

### PRIMARY KEY または UNIQUE の競合

最も一般的な競合は行の競合です。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の行に影響します。BDR はそ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 と INSERT
- UPDATE と UPDATE
- UPDATE と DELETE
- INSERT と UPDATE
- INSERT と DELETE
- DELETE と DELETE

ビュー `bdr.node_conflict_resolvers` は、既知のすべての競合タイプの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 が存在しない場合は単一の UNIQUE 制約）を持つタプルを作成する場合に発生します。

BDR は、この動作がユーザ定義の競合ハンドラーによってオーバーライドされない限り、発信元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、この状況进行处理します。

この競合は `insert_exists` 競合タイプを生成します。これは、デフォルトでは、新しい（コミット時間に基づいて）行を選択し、その 1 つだけを保持することで解決されます（`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 [列レベルの競合解決の有効化と無効化](#) を参照してください。

この競合の種類を解決するには、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。

[BDR グローバルシーケンス](#) を使用すると、この種の競合を効果的に解消できます。

## 複数の UNIQUE 制約に違反する INSERT 操作

INSERT / INSERT の競合は、複数の UNIQUE 制約に違反する場合があります（そのうちの 1 つが PRIMARY KEY である場合があります）。新しい行が複数の UNIQUE 制約に違反し、他の複数の行と競合が発生した場合、レプリケーション変更を適用すると `multiple_unique_conflicts` 競合が発生します。

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応じて、適用プロセスはエラーで終了するか、受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 で行を保持し、他の行を削除しようとします。

**警告:** この方法で複数の行が競合している場合、競合解決の結果が挿入操作を続行する場合、一部のデータは常に削除されます。

競合トリガー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 つの同時の UPDATE 操作が同じタプル（ただし、その PRIMARY KEY ではない）を変更する場合、再生時に UPDATE / UPDATE の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できます。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（起点は現在のローカル行の元のノードです）。それが変更された場合、`update_origin_change` 競合が生成されます。他のすべての場合、通常は競合を生成せずに UPDATE が適用されます。

これらの競合は両方とも、*INSERT/INSERT の競合* で説明されている `insert_exists` と同じ方法で解決されます。

## PRIMARY KEY での UPDATE の競合

BDR は現在、PRIMARY KEY が UPDATE 操作によって変更された競合解決を実行できません。主キーは更新できますが、既存の値との競合が発生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 *発散する競合* であり、手動による介入が必要です。

Postgres で主キーを更新することはできますが、Postgres と BDR 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);
```

主キー列を更新できるため、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てします。

```
INSERT INTO pktest VALUES (3,3);
```

一部の UPDATE 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 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、BDR がなくてもランタイムエラーを回避するように注意してください。

BDR では、同時に複数の場所からの UPDATE が許可されている場合、状況はより複雑になります。

これら 2 つの変更を同時に実行すると機能します。

```
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)
```

これらの次の2つの変更を同時に実行すると、両方の変更が受け入れられるため、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、各ノードでデータが異なります。

```
node1:
SELECT * FROM pktest;
pk | val
---+-----
 1 |   1
 5 |   3
(2 rows)
```

```
node2:
SELECT * FROM pktest;
pk | val
---+-----
 2 |   1
 5 |   3
(2 rows)
```

:ref:`LiveCompare` によるデータ検証<LiveCompare によるデータ検証>` を使用して、この状況を特定して解決できます。

競合が同時に発生すると問題が発生します。これら2つの変更を同時に実行するのは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;
```

両方の変更がローカルに適用されるため、ノード間で相違が発生します。ただし、ターゲットでの適用は両方のノードで重複キー値違反エラーで失敗します。これにより、レプリケーションが停止し、手動解決が必要になります。

この重複キー違反エラーを回避できるようになり、conflict\_type update\_pkey\_exists を skip、update、

または `update_if_newer` に設定してもレプリケーションは中断されません。これは、更新の性質によっては依然として発散につながる可能性があります。

`update_pkey_exists` を `update_if_newer` に設定することで、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の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更新されます。

その結果、アプリケーション、特に BDR 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 conflict` の競合解決が更新の場合、常に行の 1 つが削除されます。

## 複数の UNIQUE 制約に違反する UPDATE 操作

複数の **UNIQUE 制約に違反する INSERT 操作** と同様に、着信 UPDATE が複数の UNIQUE

インデックス（または PRIMARY KEY）に違反すると、BDR は `multiple_unique_conflicts` 競合が発生させます。

BDR は、据え置きの一意的制約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、競合が検出されない限り、ターゲットに正常に適用されます。ただし、据え置き主キーは REPLICA IDENTITY として使用できないため、ユースケースはそれと複数の一意性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

あるノードが行を更新し、別のノードが同時に削除する可能性があります。この場合、再生時に UPDATE /DELETE の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（削除された行は VACUUM によって削除されませんでした）、`update_recently_deleted` の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 [列レベルの競合解決の有効化と無効化](#) 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れている場合、削除された行は UPDATE を受け取るまでにデータベースからクリーンアップできます。この場合、BDR は UPDATE /DELETE の競合と **INSERT / UPDATE の競合** を区別できず、`update_missing` の競合を生成します。

競合する DELETE と UPDATE の別のタイプは、行がローカルで更新された後の DELETE です。この場合、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、競合はまったく検出されないため、

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` の競合が生成されます。この競合タイプのデフォルトの解決策は、DELETE を適用して行を削除することですが、これを構成するか、競合トリガーで処理できます。

## INSERT / UPDATE の競合

デフォルトの非同期モードを使用する場合、ノードは元の INSERT を受け取る前に行の UPDATE を受け取る場合があります。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します（

3 つ以上のノードとの競合 を参照）。

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。挿入またはアクションを行うリゾルバーは、可能な場合（行全体を受け取ったとき）、最初に UPDATE からのデータに基づいて新しい行を INSERT しようとします。行の再構築を可能にするには、テーブルに `REPLICA IDENTITY FULL` が含まれているか、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 サポート詳細](#)を参照してください。

## INSERT/DELETE の競合

INSERT /UPDATE の競合と同様に、ノードは、まだ INSERT を受け取っていない行で DELETE 操作を受け取る場合があります。これも 3 つ以上のノードが設定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）。

BDR は現在、この競合タイプを検出できません。INSERT 操作は競合タイプを生成せず、INSERT が適用されます。

DELETE 操作は常に `delete_missing` の競合を生成します。デフォルトでは、操作をスキップすることで解決されます。

## DELETE/DELETE の競合

2 つの異なるノードが同じテーブルを同時に削除すると、DELETE /DELETE の競合が発生します。

これにより、常に `delete_missing` の競合が生成されます。デフォルトでは、操作をスキップすることで解決されます。

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。そのうちの 1 つは無視しても問題ありません。

## 3 つ以上のノードとの競合

1 つのノードが 2 番目のノードに再生されてそこで更新される行を挿入する場合、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 の競合です。

これらの競合は、UPDATE . これにより、異なるノードに異なるデータが表示される可能性があります。これらは

発散する競合 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、そのうち少なくとも2つがアクティブに書き込んで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のに十分な大きさである必要があります。

1. ノード2はノード1からINSERTを受け取ります
2. ノード2がUPDATEを実行する
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を受け取ります

`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）を使用することが、これらの競合の実行可能な軽減戦略です。ただし、このオプションを有効にすると、INSERT/DELETEの競合の扉が開きます。

1. ノード1がUPDATEを実行する
2. ノード2は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を受け取り、それをINSERT

これらが問題になる場合は、テーブルまたはデータベースのフリーズ設定をチューニングして、`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。

別の方法は、[Eager Replication](#) を使用してこれらの競合を防ぐことです。

INSERT/DELETEの競合は、3つ以上のノードでも発生する可能性があります。このような競合は、UPDATEがDELETEに置き換えられていることを除き、INSERT/UPDATEと同じです。これにより、`delete_missing`の競合が発生する可能性があります。

BDRは、`update_missing`の競合で発生するように、各INSERTを最近削除したかどうかを確認することを選択できます。ただし、これを行うと大部分のユーザーがペナルティを受けるため、現時点では単に`delete_missing`をログに記録します。

後のリリースでは、`delete_missing`の競合が発生した場合、

[LiveCompareによるデータ検証](#)を使用した再チェックを介してINSERT/DELETE

の異常を自動的に解決します。これらは、`bdr.conflict_history_summary`ビューを確認することにより、アプリケーションで手動で実行できます。

これらの競合は、2つの主要な問題のユースケースで発生する可能性があります。

- キューイングアプリケーションで使用できるように、INSERTの後にすぐにDELETEが続きます
- テーブルの主キー識別子を再利用する場合

これらのケースはいずれも一般的ではありません。これらの問題のユースケースが発生した場合、影響を受けるテーブルを複製しないことをお勧めします。

BDRは識別子の一意性に依存してレプリケーションを機能させるため、後者の場合に問題がBDRです。



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、問題を引き起こす可能性があります。これは

[ABA problem](#) と呼ばれます。

BDR には、行が現在の行、最後の行、またはそれよりも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もビジネスの問題です。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。

変更がシステムを通過するまでにかかる時間内に識別子が再利用されると、問題が発生します。通常の実作ではその時間は短い場合がありますが、ノードがダウンすると、その間隔が数時間または数日に延長される場合があります。

アプリケーションで一意の識別子を再利用しないことをお勧めしますが、使用する場合は、1 年以内に再利用しないようにす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。

## 外部キー制約の競合

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間で競合が発生する場合は、FOREIGN KEY (FK) 制約。

BDR は `session_replication_role = 'replica'` で変更を適用するため、変更を適用するときに外部キーが再チェックされません。アクティブ/アクティブ環境では、参照元のテーブルへの挿入と同時に参照先のテーブルが削除されると、FK 違反が発生する可能性があります。これは、INSERT /DELETE の競合に似ています。

シングルマスター Postgres では、参照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 が参照された値を削除する場合、INSERT /UPDATE は FK チェックに失敗します。

マルチマスター BDR。ノード間の行レベルのロックはありません。参照元のテーブルの INSERT は、参照先のテーブルの DELETE の背後で待機しないため、両方のアクションを同時に実行できます。したがって、参照元のテーブルのあるノードの INSERT /UPDATE は、別のノードの参照先のテーブルの DELETE と同時に値を使用できます。これにより、参照元テーブルに値がなくなります。

実際には、これは、参照するテーブルでの DELETE 操作とは別のトランザクションで参照されるテーブルで DELETE 操作が発生する場合に発生します。これは一般的な操作ではありません。

Orders -> OrderItems などの親子関係では、これを行うこと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/検索データの場合、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。

FK がダングリングされる可能性はありますが、一般的にこのリスクは非常に低いため、BDR はこのケース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したら、2 つの解決策が考えられます。

最初の解決策は、FK の使用を、通常、一度に 1 つのノードからのみ変更される、頻繁に変更される、または変更の同時実行がアプリケーションによって行われる、密接に関連するエンティティに制限することです。これにより、アプリケーションレベルでの FK 違反が回避されます。

2 番目の解決策は、BDR が提供する関数 `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ると、これらの関数は 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 の異常を回避します。これにより、1 つのトリガーですべての FK が再チェックされるため、FK 違反を防ぐためにテーブルごとに 1 つのトリガーのみを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。ファクトには、RefData を参照する FK があります。Fact は参照元のテーブルであり、RefData は参照先のテーブルです。各テーブルに 1 つのトリガーを追加する必要があります。

RefData で参照された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 に列を 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();

ALTER TABLE refdata
    ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

両方のトリガーを追加すると、外部キーのダングリングが回避されます。

## 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って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を利用できないため、明確な競合がある場合でも、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 に対して他のノードで別の DML が同時に実行された場合、リプレイの順序が異なる変更を引き起こす可能性があります。

次のいずれかのアクションを実行できます。

- TRUNCATE が他の同時 DML と一緒に実行されないことを確認します。このような矛盾を強調するには、

*LiveCompare* によるデータ検証 を使用してください。

- 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、大きなテーブルではパフォーマンスが非常に低下する可能性があります。
- `bdr.truncate_locking = 'on'` を設定して、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 のデフォルトの動作ではありません。この構成は、すべての場合に可能であるとは限りません。

## 除外制約の競合

BDR は除外制約をサポートしていないため、作成を防ぎます。

既存のスタンドアロンデータベースを BDR データベースに変換する場合、すべての除外制約を手動で削除します。

分散非同期システムでは、異なるノード上のすべてのトランザクションが完全に分離されるため、制約に違反する行のセットが存在しないことを保証できません。除外制約は、除外制約違反のために再生を任意のノードから他のノードに進めることができない再生デッドロックにつながります。

BDR に除外制約を作成させる場合、またはスタンドアロンデータベースを BDR に変換するときに既存の制約を削除しない場合、レプリケーションが中断されることが予想されます。再び進行させるには、着信リモートタプルが競合するローカルタプルを削除または変更して、リモートトランザクションを適用できるようにします。

## ロールとテーブルスペースの違いによるデータの競合

競合は、ノードがロールなどの異なるグローバル（Postgres システム全体の）データを持つ場合にも発生します。これにより、主に DDL の操作が正常に実行されて 1 つのノードでコミットされますが、他のノードに適用できません。

たとえば、`node1` には `fred` という名前のユーザーがあり、そのユーザーは `node2` で作成されなかったとします。`node1` の `fred` がテーブルを作成する場合、テーブルは所有者が `fred` に設定されて複製されます。DDL コマンドを `node2` に適用すると、`fred` という名前のユーザーがないため、DDL は失敗します。この障害により、Postgres ログにエラーが出力されます。

BDR が実行されているデータベースにユーザー `fred` を作成して、この競合を解決するには、管理者の介入が必要です。`bdr.role_replication = on` を設定して、将来的にこれを解決できます。

## ロックの競合とデッドロックの中止

BDR ライタプロセスは通常のユーザーセッションとほとんど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に従います。これにより、BDR ライタプロセスがユーザトランザクションまたは相互に保持されてい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。

- ユーザセッションによる明示的なテーブルレベルロック (`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 操作による暗黙的なロック

BDR ライタプロセスは、ユーザトランザクションでデッドロックする可能性があります。その場合、ユーザトランザクションはライタプロセスが保持するロックを待機しています。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器が介入して、問題のあるトランザクションの1つを終了します。BDR ライタプロセスが終了した場合、再試行しますが、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あり、通常は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を再度実行できますが、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することと変わりません。

Postgres の `log_lock_waits` 機能を使用すると、ロック関連のリプレイストールを特定できます。

### 発散する競合

異なる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、発散競合が発生します。発散する競合は発生しないはずですが、執筆時点でこのような競合のすべてを確実に防止できるわけではありません。

行の PRIMARY KEY を変更すると、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、発散の競合が発生する可能性があります。主キーを変更しないか、指定された1つのノードでのみ変更します。

行データが関係する発散コンフリクトでは、一般に、一方のノードのデータを他方のノードと一致するように調整する管理者のアクションが必要です。ドキュメントのとおり BDR 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` などの高度な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、状況がさらに悪化し、一般的な指示では対処できない競合が発生する可能性があります。

### TOAST サポート詳細

Postgres は、[TOAST サポート詳細](#)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。

論理デコード（BDR がその上に構築されています）および論理レプリケーションで処理する TOAST 値は、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。これは、特に UPDATE の競合を処理するときに問題を引き起こす可能性があります。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です。

[INSERT / UPDATE の競合](#) で述べたように、`update_missing` の競合が

`insert_or_error` を使用して解決され、TOAST 列が欠落している場合、BDR はエラーを報告します。

ただし、非同期レプリケーションを使用した同時ワークロードの場合、これよりも微妙な問題があります（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 という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 SET col1 = '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'`; A1 の後にコミットします。
3. ノード C: ノード A への接続が遅れています。
4. ノード C: txn B1 が最初に適用され、col1 の TOAST された列を見逃しますが、競合せずに適用されます。
5. ノード C: txn A1 は (update\_origin\_change で) 競合するため、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遠に逃します。

BDR を使用する場合、このシナリオは通常問題になりません。(ビルトイン論理レプリケーションまたはマルチマスターのプレーンな TOAST を使用する場合があります。) BDR は、最近 TOAST 列の変更をレプリケートした行へのローカル UPDATE を検出し、ローカル UPDATE が `t` TOAST を変更する。したがって、BDR は、異なるノード間でトーストされたデータの不整合を防ぎます。この状況により、複数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 ロギングが増加します。BDR AlwaysOn アーキテクチャの場合、通常のように、すべての更新が単一のノードから行われた場合、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 テーブルだけで `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、そのようなステートメントの後の短期間、これらの同時実行の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 の追加の WAL ロギングは、標準の Postgres で `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並べ替える必要があります。これを簡単にするために `zzzz_bdr_` プレフィックスが付いていますが、その後にソートされる名前で作成しないでください。そうしないと、同時実行の問題に対する保護が得られません。

ただし、`insert_or_error` の競合解決には、`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` を持つ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見なされるため、トーストされた値が常にキーの一部としてログに記録されます。BDR は、キー行から欠落しているデータを埋めて、新しい行を再構築できます。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。

### 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設計できます。

競合は、複数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のみ特定の行に書き込むことです。

これは、多くのアプリケーションで自然に発生します。たとえば、多くの消費者アプリケーションでは、アカウントのデフォルトの請求先住所の変更など、所有者のみがデータを変更できます。このようなデータ変更で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行う可能性があるため、変更が失われたようです。次に、同じ変更を再度行くと、異なるノードで2つの更新が発生する可能性があります。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せんが、データの最後の更新が保持されるため拒否されます。

#### INSERT /INSERT の競合の場合、

**BDR グローバルシーケンス** を使用してこの種の競合を防ぎます。

部屋予約アプリケーションなど、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** を適用しても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、2人に同じ部屋を予約したことを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、より複雑な方法も可能です。たとえば、各ノードに100のシートを割り当て、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ら、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1つの特定のノードからのみ行われるようにする別の手法は、さまざまな種類のトランザクションをさまざまなノードにルーティングすることです。例：

- 1つのノードで小包を受け取りますが、別のノードを使用して小包を配達します
- 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、最善の方法は、競合の発生を許可し、BDR の競合解決メカニズムと連携して競合に対処するようにアプリケーションを設計することです。

### 競合検出

BDR は、競合検出に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

オリジンの競合検出は、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メッセージの許容範囲内にある必要があります。そうでない場合、競合の解決はさらに先の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。

行の起点は、**track\_commit\_timestamp = on** の場合にのみ使用できます。

競合は、レプリケーション起点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、競合ではないことが判明した状況で競合トリガーが呼び出されます。したがって、このメカニズムは誤検出の競合を生成する可能性があるため、正確ではありません。

起点情報は、行がフリーズするまでのみ利用できます。フリーズされた後に行に到着した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` で新しいノードを追加する場合の通常の場合であるため、その場合に競合が発生すると、多くの誤検出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フリーズされた行よりも古い場合、これにより異なる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。

### ノードの再起動とダウンしたノードの回復

で説明されているように、長期にわたる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の行のフリーズを BDR 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 の他のバリエーションでは、この状況を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 の `vacuum_freeze_min_age` xid より古い場合に発生します。つまり、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ります。

- `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設けます。たとえば、`vacuum_freeze_min_age` が 5 億に設定されている場合、1000 TPS を実行するノードは、競合データが削除されるまでに 5.5 日強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 のディスク領域を使用するため、トランザクションレートが低いシステムでは、設定を低くすることでメリットを得られます。

最初に推奨される設定は次のとおりです。

```
# 1 billion = 10GB

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` はノードの先頭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され、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` と `toast.autovacuum_freeze_min_age` を設定することもできます。

- CLUSTER または VACUUM FREEZE コマンドを実行すると、行が早期にフリーズされるため、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

または、BDR は、行の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では、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。

1. BDR ノードグループで `check_full_tuple` を有効にする必要があります。
2. 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` を有効にする必要があります。
3. `bdr.alter_table_conflict_detection` を使用して、テーブルで行バージョンの追跡を有効にする必要があります。このファンクションは、列（指定した名前）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。

カウンターは `UPDATE` でのみインクリメントされますが、この手法により `UPDATE` と `DELETE` の両方で競合を検出できます。

このアプローチは Lamport タイムスタンプに似ており、競合検出の ABA 問題を完全に防ぎます。

---

**注釈:** 行レベルの競合解決は、行のバージョン管理を使用しても **列レベルの競合解決の有効化と無効化** 構成に基づいて処理されます。行バージョンが生成される方法は、競合を検出する場合にのみ役立ちます。行のバージョンが新しいかどうかについての信頼できる情報として信頼しないでください。

---

特定のテーブルに使用されている現在の競合解決戦略を判断するには、ビュー `bdr.tables` の列 `conflict_detection` を参照してください。

### `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の動作を変更できます。

#### 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リレーシヨンの名前。
- **method** —使用する競合検出方法。
- **column\_name** —列検出データの格納に使用する列。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用の追加の列を必要としません。

認識されている競合検出方法は次のとおりです。

- **row\_origin** —タプルに対して行われた以前の変更の起点（[オリジンの競合検出](#)を参照）。これは、テーブルに追加の列を必要とせずに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（[行バージョンの競合検出](#)を参照）。
- **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）。

## 注意事項

**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、[現在とコミットのタイムスタンプ](#)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複製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。

このファンクションは、列レベルの競合解決が有効になっているリレーシヨンの **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** を使用して効果をロールバックでき、変更は現在のトランザクションで表示されます。

**bdr.alter\_table\_conflict\_detection** ファンクションは、**bdr.backwards\_compatibility** が 30618 以下に設定されていない限り、**relation** の所有者のみが実行できます。

**警告:** 余分な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、その列が削除されます。

**警告:** この関数は **CAMO** を無効にします（警告とともに、これらが ``bdr.camo_enable_client_warnings`` で無効にされていない限り）。

## 競合タイプのリスト

BDR は、`conflict_type` パラメーターとして使用できる次の競合タイプを認識します。

- `insert_exists` —着信挿入が、主キーまたは一意のキー/インデックスを介して既存の行と競合しています。
- `update_differing` —着信更新のキー行がローカル行と異なります。これは、[行バージョンの競合検出](#)を使用している場合にのみ発生します。
- `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` —着信更新が存在しない行を変更しようとしています。
- `update_recently_deleted` —受信した更新が、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` が変更を適用しているノードに既に存在する値に変更されました。
- `multiple_unique_conflicts` —着信行は、ターゲットテーブル内の複数の `UNIQUE` 制約/インデックスと競合しています。
- `delete_recently_updated` —現在のノードでの行の最新の更新よりも古いコミットタイムスタンプを持つ着信削除、または[行バージョンの競合検出](#)を使用している場合。
- `delete_missing` —着信削除が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、受信行にある 1 つ以上の列がありません。
- `source_column_missing` —受信行には、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。BDR はデフォルトで `last-update-wins` メカニズム、より正確には `update_if_newer` 競合リゾルバーです。このメカニズムは、競合検出に使用されるのと同じコミットタイムスタンプに基づいて、競合する 2 つの行のうち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。

BDR では、次の機能を使用して、競合解決のデフォルトの動作をオーバーライドできます。

## bdr.alter\_node\_set\_conflict\_resolver

この関数は、指定された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の種類（[競合タイプのリスト](#)を参照）。
- **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）。

### 注意事項

現在、ローカルノードのみを変更できます。関数呼び出しは複製されません。複数のノードの設定を変更する場合は、それぞれで関数を実行する必要があります。

この関数による構成の変更は、`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の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。変更をロールバックでき、現在のトランザクションで表示されます。

### 競合リゾルバーのリスト

BDR では、処理できる競合タイプのカバレッジが異なるいくつかの競合リゾルバーを使用できます。

- **error** —エラーをスローし、レプリケーションを停止します。任意の競合タイプに使用できます。
- **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 `conflict` タイプに使用できます。
- **skip\_if\_recently\_dropped** —ダウンストリームで最近（1 日以内）に削除されたためにダウンストリームに存在しないテーブルの場合、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。 `skip_if_recently_dropped` の競合リゾルバーは、同じ名前のテーブルが削除された直後に再作成されると、問題が発生する可能性があります。その場合、ノードの 1 つが、テーブルを再作成するための DDL を認識する前に、再作成されたテーブルの DML を認識する可能性があります。次に、テーブルが最近削除されたと想定

して、リモートデータを誤ってスキップし、データを損失します。したがって、この競合リゾルバーとともに削除された直後にオブジェクト `namesq` 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。 `apply_error_ddl` の競合に使用できます。
- `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（発信元ノードの壁時計によって決定）を更新します。タイムスタンプが同じ場合、ノード ID が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ます（ノード ID が高い方が勝ちます）。 `insert_exists` 、 `update_differing` 、 `update_origin_change` 、および `update_pkey_exists` の競合タイプに使用できます。
- `update` —常に複製されたアクションを実行します。 `insert_exists`（INSERT を UPDATE に変換）、 `update_differing` 、 `update_origin_change` 、 `update_pkey_exists` 、および `delete_recently_updated`（削除を実行）に使用できます。
- `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作成し、挿入してみてください。完全な行を作成するのに十分な情報がない場合は、変更をスキップします。 `update_missing` および `update_recently_deleted` の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作成して挿入しようとします。完全な行を構築するのに十分な情報がない場合は、エラーをスローしてレプリケーションを停止します。 `update_missing` および `update_recently_deleted` の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` —リモート行の余分な列に 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、エラーをスローして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（それが列のデフォルトの場合は NULL を含む）で入力し、処理を続行します。デフォルトまたは制約の違反（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します。 `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`, and `delete_missing` の競合タイプは、

**競合トリガー** を使用してユーザー定義ロジックで解決することもできます。

このマトリックスは、競合リゾルバーが処理できる競合の種類を区別するのに役立ちます。

## 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-<br>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           | skip_if_recently_dropped |
| apply_error_ddl                | error                    |

## 競合解決のリスト

競合解決は、競合リゾルバーによって選択され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。

`conflict_resolution` パラメーターでは、現在次の競合解決がサポートされています。

- `apply_remote` — リモート（着信）行が適用されました。
- `skip` — 一行の処理がスキップされました（ローカルで変更はありませんでした）。
- `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、ターゲット表に書き込まれた行を生成しました。

## 競合ログ

マルチマスターの競合の診断と処理を容易にするために、BDR はデフォルトですべての競合を `bdr.conflict_history` テーブルに記録します。次の関数を使用して、この動作をより詳細に変更できます。

### `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の種類。 `NULL` (デフォルト) はすべてを意味します。
- `conflict_resolution` — ログに記録する競合の解決。 `NULL` (デフォルト) はすべてを意味します。

## 注意事項

ローカルノードのみ変更できます。関数呼び出しは複製されません。複数のノードの設定を変更する場合は、それぞれで関数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。変更をロールバックでき、現在のトランザクションで表示されます。

## 競合ログ構成のリスト

ビュー `bdr.node_log_config` は、すべてのロギング構成を表示します。ログ構成の名前、ログに記録する場所、および競合の種類と解決がリストされます。

## テーブルへの競合のログ

`log_to_table` が `true` に設定されている場合、競合はテーブルに記録されます。競合ログの対象テーブルは `bdr.conflict_history` です。

この表は、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 Autopartition によって管理されています。デフォルトでは、毎日新しいパーティションが作成され、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自動的に削除されます。自動パーティションは、7~14 個のパーティションを事前に作成します。`bdr_superuser` はこれらのデフォルトを変更できます。

BDR 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するデータを読み取れるようにすることが重要です。BDR は、`bdr.conflict_history` テーブルに 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、それぞれのテーブルで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、所有者でさえ競合を読み取ることができません。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が所有者でさえも読み取れない場合があります。そのため、BDR は競合ログテーブルにより厳しいポリシーも適用します。

デフォルトのロール `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すべての競合の詳細を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、ユーザーデータを含まない `bdr.conflict_history_summary` というカタログビューにアクセスできるため、ログに記録された競合を監視できます。

## 競合レポート

テーブルに記録された競合は、レポートに要約できます。レポートを使用すると、アプリケーションの所有者は競合を特定、理解、解決し、アプリケーションの変更を導入して競合を防ぐことができます。

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

(次のページに続く)

|         |  |      |  |            |  |   |
|---------|--|------|--|------------|--|---|
| my_app  |  | test |  | 2019-04-05 |  | 1 |
| (1 row) |  |      |  |            |  |   |

## LiveCompare によるデータ検証

LiveCompare は、2 つのデータベースを比較して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 BDR スタックの一部として含まれており、BDR ノードの任意のペアを対象としています。デフォルトでは、すべてのレプリケートされたテーブルを比較し、違いを報告します。LiveCompare は、Postgres や Oracle などの非 BDR データソースでも機能します。

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 を使用すると、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をチェックできるように構成できます。チェックは、最初に行ハッシュ全体を比較することによって実行されます。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、期間をかけて再確認できるため、最終的な一貫性が遅れます。

詳細については、[LiveCompare によるデータ検証](#) のドキュメントを参照してください。

## 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、データベースの 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され、nextval() 関数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を認識せず、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競合とデータ損失（破棄された INSERT アクション）を引き起こすため、これは重要です。



## BDR グローバルシーケンス

このため、BDR は、**グローバルシーケンス**と呼ばれる BDR 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。

BDR 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が必ずしもすべてではない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。

BDR グローバルシーケンスを使用すると、挿入の競合の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。BDR がノード間で挿入を同期する場合、競合することはありません。

BDR 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` ロールが付与されている必要があります。

グローバルシーケンスにはさまざまなアルゴリズムがあります。

- `SnowflakeId` シーケンス
- グローバルに割り当てられた範囲シーケンス

`SnowflakeId` シーケンスは、任意の時点で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。より高速で堅牢で、値が作成されたタイムスタンプを記録するという便利なプロパティがあります。

`SnowflakeId` シーケンスには、64 ビットの `BIGINT` データ型でのみ機能し、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 Integer 型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、ノード間のコンセンサスによって必要に応じて補充できる値のローカル範囲を割り当て、`BIGINT` または `INTEGER` シーケンスに適したものにします。

`bdr.alter_sequence_set_kind()` ファンクションを使用して、グローバルシーケンスを作成できます。このファンクションは、標準の PostgreSQL シーケンスを取り、それを BDR 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 シーケンスに変換することもできます。

BDR は、`CREATE SEQUENCE` コマンドが実行されたときに、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する構成変数 `bdr.default_sequence_kind` も提供します。有効な設定は次のとおりです。

- `local` (デフォルト)、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。
- `galloc`、常にグローバルに割り当てられた範囲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 (したがって、`bigserial` には使用できますが、`serial` には使用できません)。
- `timeshard`、これは `SnowflakeId` シーケンスの古いバージョンであり、下位互換性のためにのみ提供されています。`SnowflakeId` が優先されます。
- `distributed`、これは `bdr.default_sequence_kind` でのみ使用できる特別な値です。`int8` シーケンス (つまり `bigserial`) には `snowflakeid` を選択し、`int4` (つまり `serial`) および `int2` シーケンスには `galloc` を選択します。

`bdr.sequences` ビューは、個々のシーケンスの種類に関する情報を表示します。

`currval()` および `lastval()` は、すべてのタイプのグローバルシーケンスで正しく機能します。

## SnowflakeId シーケンス

SnowflakeId シーケンスによって生成された ID は大まかな時間順序付けであるため、標準の PostgreSQL シーケンスのように、それらを使用して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、同じミリ秒内に生成された値は順不同である場合があります。緩い時間順序のプロパティは、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のノード間通信は必要ありません。そのため、ネットワークパーティションが拡張されるリスクがある場合でも、引き続き使用できます。レプリケーションのラグやノード間の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。このアルゴリズムでは、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 BDR ノードに割り当てられた一意の ID です。これにより、異なるノードの ID は常に異なります。3番目のコンポーネントは、ローカルシーケンスによって生成された番号です。

シーケンス番号に一意のノード ID を追加するだけで競合が発生しませんが、シーケンスの別の有用なプロパティも保持します。シーケンス番号の順序は、テーブルにデータが挿入された順序にほぼ対応しています。タイムスタンプを最初に置くと、これが保証されます。

SnowflakeId シーケンスには、いくつかの制限と警告が適用されます。

SnowflakeId シーケンスは 64 ビット幅で、`bigint` または `bigserial` が必要です。生成される値は少なくとも 19 桁です。実用的な 32 ビット `integer` バージョンはないため、`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範囲シーケンスを使用します。

SnowflakeId の場合、特定のノードで 1 ミリ秒に生成されるシーケンス値の制限は 4096 です（1 秒あたり約 400 万のシーケンス値）。シーケンス値の生成が指定されたミリ秒以内にラップする場合、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、指定されたタイム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 までに数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た最新のものよりも古いタイムスタンプを持つシーケンスを生成しないようにする追加の保護があります。

SnowflakeId シーケンスの入力として使用されるシーケンスの `INCREMENT` オプションは効果的に無視されます。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 のように、シーケンス ID キャッシングを行うアプリケーションに関連する場合があります。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を使用して何かを実行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、実際の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、変更しても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、値の範囲の制限は関数の結果に影響しません。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

Timeshard 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での使用は推奨されません。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、SnowflakeId よりも多い、ミリ秒あたり最大 16384 個（1 秒あたり約 1600 万個）を生成できます。ただし、指定された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。タイムハードシーケンスを使用するスキーマは、特定の列にタイムハード値を使用するときに **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 年に値を使い果たし、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 よりも大幅に短い時間であり、SnowflakeId が好ましい主な理由です。- タイムシャードシーケンスでは、定期的な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）。SnowflakeId はメモリで計算されるため、SnowflakeId シーケンスは一般的に timeshard シーケンスよりも少し高速です。

## グローバルに割り当てられた範囲シーケンス

グローバルに割り当てられた範囲（または `galloc`）シーケンスは、各ノードに値の範囲（チャンク）を割り当てます。ローカル範囲が使用されると、他のノード間のコンセンサスによって新しい範囲がグローバルに割り当てられます。これはキースペース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範囲が使い果たされたときにシーケンスジェネレーターを進行させるには、ローカルノードをクラスター内の大部分のノードに接続する必要があります。

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限られたスペースで出力に表示される場合の JavaScript での整数の使用が含まれます。

各投票によって割り当てられる範囲は、現在、シーケンスが使用しているデータ型に基づいて事前に決定されます。

- `smallint` - 1 000 の数字
- 整数 1 000 000 の数字
- `bigint` - 1 000 000 000 の数字

各ノードは `seq_chunk_size` の 2 つのチャンクを割り当てます。1 つは現在の使用に加えて、1 つは将来の使用のために予約されているため、1 つのノードから生成される値は単調に増加します。ただし、グローバルに見ると、生成される値はまったく順序付けられません。これは、B ツリーインデックスへの影響によ

りパフォーマンスが低下する可能性があり、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、割り当てられた範囲を使い果たすと、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範囲についてコンセンサスを求める必要があることです。BDR グループの大部分にアクセスできない場合、これにより遅延や操作上の問題が発生する可能性があります。これは、後のリリースでは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 シーケンスで正しく機能します。ただし、シーケンスを galloc 種類に変換する前に設定する必要があります。INCREMENT BY オプションも正しく動作します。ただし、シーケンスデータ型ごとに割り当てられた上記の範囲以上のインクリメント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。BDR は、特別な BDR カタログ `:ref:bdr.sequence_alloc`<bdr.sequence_alloc`>` で `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 によって管理されるのに対し、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るため、BDR は現在、galloc シーケンスの名前変更、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設計するときは、この制限に注意してください。

## ローカルシーケンスから 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を満たす必要があります。

### 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す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、そのシーケンスによって返される nextval() に integer 列のデフォルトを割り当てることができます。bigint に 1 000 000 000 のブロックで割り当てられる galloc シーケンスを使用すると、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 の範囲を超えます。

次の例は、何が起こ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;
```

(次のページに続く)

(前のページからの続き)

```

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)` を実行すると、テーブルには次の値が含まれます。

```

SELECT * FROM seqtest;
   id
-----
      2
2000000002
(2 rows)

```

ただし、3番目のノードで同じ操作を試みると、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 の `pg_sequences` ビューからシーケンスの現在のデータ型を取得できます。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する

シーケンス種類を `galloc` に変更すると書き換えられ、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内の既存のシーケンスで発生した場合、現在の値を照会してから、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、BDR ではユーザーが関数 `bdr.alter_sequence_set_kind()` で開始値を渡すことができます。既にオフセットを使用していて、複数のノードからの書き込みがある場合は、使用された最大の値を確認し、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->0->>nextval)::bigint)
FROM json_array_elements(
    bdr.run_on_all_nodes(
        ESELECT nextval(\public.sequence\);
    )::jsonb AS x;

-- turn into a galloc sequence

```

(次のページに続く)

(前のページからの続き)

```
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、`max()` 値が照会されている間も操作を続行できるように、`$MARGIN` 値を残す必要があります。

`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範囲に関する情報を提供します。この例では、333 からシーケンスを開始し、クラスター内に2つのノードがあります。ノードごとに2つの割り当て数があり、チャンクサイズは1000000であり、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
-----+-----+-----+-----+
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範囲を表示するには、次のクエリを使用します。

- ノード Node1 は 333 から 2000333 の範囲を使用しています。

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,2); -- first range
range_start | range_end
-----+-----
          334 |    1000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,3); -- second range
range_start | range_end
-----+-----
    1000334 |    2000333
(1 row)
```

- ノード Node2 は 2000004 から 4000003 の範囲を使用しています。

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,2); -- first range
range_start | range_end
-----+-----
    2000334 |    3000333
(1 row)
```

(次のページに続く)



(前のページからの続き)

```
SELECT last_value AS range_start, log_cnt AS range_end
      FROM categories_category_seq WHERE ctid = (0,3); -- second range
range_start | range_end
-----+-----
 3000334 |   4000333
```

**注釈:** まだ最初の範囲のみが表示されるため、単一のクエリ（`WHERE ctid IN ('(0,2)', '(0,3)')` など）に結合できません。

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、最初に利用可能なチャンクを取得します。この例では、4000334 から 5000333 です。これは新しい予約済みチャンクであり、古い予約済み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

BDR で使用できるグローバルシーケンスを使用せずにグローバルに一意の ID を生成する方法は他にもあります。例：

- UUID とその BDR バリエント、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

BDR アプリケーションは他の方法を安全に使用できません BDR はノード間の行ロックをとらないため、シーケンス生成に `SELECT ... FOR UPDATE`、`UPDATE ... RETURNING ...` などに依存するカウンターテーブルベースのアプローチは BDR で正しく機能しません。複数のノードで同じ値が生成されます。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 BDR 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、外部ソースからの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、BDR グループ内の 1 つのノードでのみ生成します。

## UUID と KSUUID

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ランダムな値であり、同じ値を 2 回生成することはほとんど不可能です。UUID キーを使用する場合、ノードが継続的な通信を行う必要はありません。

万が一、衝突が発生した場合、競合検出は、挿入された 2 つのレコードのうち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、そのようなイベントが記録されます。ただし、衝突は約  $2^{64}$  キーが生成された後にのみ発生する可能性があるため、例外的に発生する可能性はありません。

UUID キーの主な欠点は、スペースとネットワークの面でやや非効率なことです。主キーとしてだけでなく、外部キーで参照される場所やネットワーク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 キーに対応しているわけではありません。

BDR は、KSUUID と呼ばれる UUID データの K-Sortable バリエントを提供するための関数を提供します。これは、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格納するという点で UUIDv1 に似ています。違いは、KSUUID は K-Sortable、つまりタイムスタンプで弱くソートできることです。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、検索の有効性が向上し、結果データの自然な時間ソートが可能になるため、データベースキーとしてより便利になります。UUIDv1 とは異なり、KSUUID の値には、それらが生成されたコンピューターの MAC が含まれないため、使用してもセキュリティ上の懸念はありません。

KSUUID v2 はすべての場合に推奨されます。通常の比較演算子で生成された値を直接ソートできます。

BDR には KSUUID の 2 つのバージョンがあります：v1 と v2。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保持されています。新規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、それらを実行する機能も互換性がありません。KSUUID の v2 には、UUID バージョン番号も保存されなくなりました。

## ステップとオフセットのシーケンス

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でインクリメントし、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、`node1` のシーケンスは 1001、2001、3001 など生成します。`node2` のシーケンスは 1002、2002、3002 など生成します。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、設計者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。間違いは、シーケンスの重複につながりやすいです。

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、BDR 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の開始値を指定します。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混乱を招くため、追加する可能性のあるすべてのノード用に十分な大きさの `INCREMENT` を確保してください。



`bigint` の値を使用すると、10000 以上のオフセットを使用してもキーが枯渇する心配はありません。何百もの機械が毎秒何百万もの挿入を行うと、枯渇に近づく可能性があります。

BDR は現在、そのようなステップ/オフセットシーケンスでのノードごとのオフセットの構成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エーションは、`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関数から取得されます。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 行の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。

## グローバルシーケンス管理インターフェース

BDR は、標準の PostgreSQL シーケンスと BDR 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。

以下の関数は DDL とみなされるため、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。

### `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るようにします。設定すると、`seqkind` は `bdr.sequences` ビューを介してのみ表示されます。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。

BDR 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は 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。

*DML および DDL レプリケーションを参照してください。*

## 概要

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text)
```

## 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 BDR シーケンスの場合は `snowflakeid` または `gallocc`、またはレガシーのグローバルに一意のシーケンスの場合は `timeshard`。

## 注意事項

シーケンスの種類を `galloc` に変更すると、そのシーケンスに最初に割り当てられた範囲はシーケンスの開始値を開始点として使用します。`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ある場合は、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、開始点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複製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。

この関数は、グローバルな 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` で効果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。戻り値は `timestampz` 型です。

**概要** `sql bdr.extract_timestamp_from_snowflakeid(snowflakeid bigint)`

### パラメーター

- `snowflakeid` — スノーフレイクイド シーケンスの値。

## 注意事項

この関数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 `sql bdr.extract_nodeid_from_snowflakeid(snowflakeid bigint)`

#### パラメーター

- `snowflakeid` スノーflakeid シーケンスの値。

#### 注意事項

この関数はローカルノードでのみ実行されます。

#### `bdr.extract_localseqid_from_snowflakeid`

この関数は、`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 `sql bdr.extract_localseqid_from_snowflakeid(snowflakeid bigint)`

#### パラメーター

- `snowflakeid` スノーflakeid シーケンスの値。

#### 注意事項

この関数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` 列の値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` がある場合、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)
```

このように定式化されたクエリは、列 `id` でインデックススキャンを使用します。

概要 `sql 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。

### 注意事項

この関数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。戻り値は `timestamptz` 型です。

### 概要

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

### パラメーター

- `timeshard_seq` — タイムハードシーケンスの値。

### 注意事項

この関数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` — タイムハードシーケンスの値。

## 注意事項

この関数はローカルノードでのみ実行されます。

### **bdr.extract\_localseqid\_from\_timeshard**

この関数は、`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` — タイムハードシーケンスの値。

## 注意事項

この関数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タイムシャードシーケンス値に変換します。

これは、`timeshard` 列の値と特定のタイムスタンプのインデックス付き検索または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` がある場合、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)
```

このように定式化されたクエリは、列 `id` でインデックススキャンを使用します。

## 概要

```
bdr.timestamp_to_timesthard(ts timestamptz)
```

## パラメーター

- `ts` — タイムハードシーケンス生成に使用するタイムスタンプ。

## 注意事項

この関数はローカルノードでのみ実行されます。

## KSUUID v2 関数

KSUUID v2 データ、K-Sortable UUID データを操作するための関数。

### **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刻を使用して、新しい KSUUID v2 値を生成します。システム時刻を使用して KSUUID を自動的に生成する場合は、NULL 引数を渡します。

戻り値は UUID 型です。

## 概要

```
bdr.gen_ksuuid_v2(timestamptz)
```

## 注意事項

この関数はローカルノードでのみ実行されます。

### **bdr.ksuuid\_v2\_cmp**

この関数は、KSUUID v2 の値を比較します。

最初の値の方が新しい場合は 1、2 番目の値の方が小さい場合は -1、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 — 比較する KSUUID v2。

## 注意事項

この関数はローカルノードでのみ実行されます。

## **bdr.extract\_timestamp\_from\_ksuuid\_v2**

この関数は、KSUUID v2 のタイムスタンプコンポーネントを抽出します。戻り値は `timestampz` 型です。

## 概要

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

## パラメーター

- UUID — タイムスタンプを抽出する KSUUID v2 値。

## 注意事項

この関数はローカルノードでのみ実行されます。

## **KSUUID v1 の機能**

KSUUID v1 データ、K-Sortable UUID データ (v1) を操作するための関数。

## **bdr.gen\_ksuuid**

この関数は、現在のシステム時刻を使用して、新しい KSUUID v1 値を生成します。戻り値は UUID 型です。

### 概要

```
bdr.gen_ksuuid()
```

### 注意事項

この関数はローカルノードでのみ実行されます。

### **bdr.uuid\_v1\_cmp**

この関数は、KSUUID v1 の値を比較します。

最初の値の方が新しい場合は 1、2 番目の値の方が小さい場合は -1、等しい場合は 0 を返します。

### 概要

```
bdr.uuid_v1_cmp(uuid, uuid)
```

### 注意事項

この関数はローカルノードでのみ実行されます。

### パラメーター

- UUID — 比較する KSUUID v1。

### **bdr.extract\_timestamp\_from\_ksuuid**

この関数は、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。戻り値は `timestampz` 型です。

### 概要

```
bdr.extract_timestamp_from_ksuuid(uuid)
```



## パラメーター

- UUID — タイムスタンプを抽出する KSUUID v1 値。

## 注意事項

この関数はローカルノードで実行されます。

## 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。つまり、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を選択し、もう一方を破棄します。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、新しい方を保持する場合があります。これにより、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が確立されます。

ただし、行レベルではなく列レベルで競合を解決する方が適切な場合があります。

2つの整数列 `a` と `b`、および単一の行 `(1,1)` を持つテーブル `t` がある簡単な例を考えます。1つのノードで実行すると仮定します。

```
UPDATE t SET a = 100
```

別のノードで同時に（先行の UPDATE を受け取る前に）実行します。

```
UPDATE t SET b = 100
```

これにより、UPDATE-UPDATE の競合が発生します。update\_if\_newer 競合解決では、コミットのタイムスタンプを比較し、新しい行バージョンを保持します。2番目のノードが最後にコミットされたと仮定すると、最終的に `(1,100)` になり、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、これは望ましい動作ですが、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異なるノードに接続され、共有テーブル内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お互いのつま先を踏んで、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方が適切な場合があります。これを行うために、BDR は各列の最終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（基本的に update\_if\_newer）を選択します。

前の例に適用すると、どちらのノードもそのような行を表示していないにもかかわらず、両方のノードで `(100,100)` になります。

列レベルの競合解決を検討するときは、テーブルを垂直にパーティション分割されたものとして表示すると、各更新が1つのスライスのデータにのみ影響する場合があります。このアプローチにより、列の異なるサブセットに対する変更間の競合が排除されます。実際、垂直方向のパーティション化は、列レベルの競合解決の実用的な代替手段になることさえあります。

列レベルの競合解決では、テーブルに `REPLICA IDENTITY FULL` が必要です。 `bdr.alter_table_conflict_detection` 関数はそれをチェックし、それ以外の場合はエラーで失敗し

ます。

## 列レベルの競合解決の有効化と無効化

列レベルの競合解決は、`bdr.alter_table_conflict_detection()` ファンクションによって管理されます。

### 例

`bdr.alter_table_conflict_detection()` の使用方法を確認するには、簡単なテーブル `test_table` を作成し、それで列レベルの競合解決を有効にする次の例を考えます。

```
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, column_modify_timestamp, cts);
  alter_table_conflict_detection
- -----
t

db=# \d my_app.test_table
```

この関数は、(関数呼び出しで指定されたとおりに) 新しい `cts` 列を追加しますが、各変更の前に新しい列のタイムスタンプを維持する責任がある 2 つのトリガー (`BEFORE INSERT` および `BEFORE UPDATE`) も作成しました。

また、新しい列はデフォルト値で `NOT NULL` を指定しています。これは、`ALTER TABLE ... ADD COLUMN` はテーブルの書き換えを実行しないことを意味します。

---

**注釈:** `bdr.column_timestamps` データ型の列は、悪影響を与える可能性があるため、他の目的で使用しないことをお勧めします。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いと正しく機能しません。

---

## 列レベルの競合解決を備えたリストテーブル

次のクエリを使用して、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。このクエリは、`bdr.column_timestamp` 型の列の存在を検出します。

```
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);
```

## `bdr.column_timestamps_create`

このファンクションは、列レベルの競合解消を作成します。`column_timestamp_enable` 内で呼び出されます。

### 概要

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestamptz)
```

### 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、`TIMESTAMP_SOURCE_COMMIT`。`current` の場合、`TIMESTAMP_SOURCE_CURRENT`。

## DDL ロック

テーブルの列の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、切り替え前からの保留中の変更がないことを確認します。このアプローチにより、両方のタプルまたはどちらでもタイムスタンプの競合のみが表示されます。そうしないと、コードはローカルのタプルにタイムスタンプを、リモートのタプルに NULL を予期せず表示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）で解決されるようにします。

## 現在とコミットのタイムスタンプ

重要な決定は、変更された列に割り当てられるタイムスタンプです。

デフォルトでは、変更された列に割り当てられるタイムスタンプは、あたかも `clock_timestamp` から取得されたかのように、現在のタイムスタンプです。これは簡単で、多くの場合、完全に正しいです（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）。

ただし、さまざまな予期しない効果があります。

- タイムスタンプはステートメントの実行中に変更されるため、`UPDATE` が複数の行に影響を与える場合、それぞれがわずかに異なるタイムスタンプを取得します。これは、同時変更の影響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（さまざまなノードで実行された変更がどの程度正確にインターリーブするかに応じて）。
- タイムスタンプはコミットタイムスタンプとは関係なく、競合を解決するために使用すると、結果がコミット順序と同等ではなく、シリアル化できない可能性があります。

---

**注釈:** 将来的にステートメントとトランザクションのタイムスタンプを追加する可能性があります。これにより、同時実行のステートメントまたはトランザクションの効果が混在する問題に対処します。それでも、これらのオプションはいずれも、コミット順序と同等の結果を生成できません。

---

実際のコミットタイムスタンプを使用することもできますが、この機能は現在実験的と見なされています。コミットのタイムスタンプを使用するには、列レベルの競合解決を有効にするときに最後のパラメーターを `true` に設定します。

```
SELECT bdr.column_timestamps_enable(test_table::regclass, cts, true);
```

`bdr.column_timestamps_disable` を使用して無効にできます。

コミットタイムスタンプには現在、[注意事項](#)で説明されている制限があります。

## 列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、トリガーによって自動的に維持されます。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するときに、現在のタイムスタンプ値を検査すると役立ちます。

このためには、次の3つの関数が役立ちます。

- `bdr.column_timestamps_to_text(bdr.column_timestamps)`

この関数は、タイムスタンプマッピングの人間が判読できる表現を返し、値を `text` にキャストするときに使用されます。

```
db=# select cts::text from test_table;
```

```
cts
```

```
-----  
->-----
```

(次のページに続く)

(前のページからの続き)

```
{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)`

この関数は、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によって変更された属性のコミットタイムスタンプでマッピングを更新します (すでにコミットされている場合)。これは、コミットの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で2番目の属性が更新されました (`attnum = 2` を使用)。

```
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)
```

## CRDT データ型を使用した列の競合の処理

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、もう一方を破棄します。ただし、別のより複雑な方法で競合を調整できます。たとえば、情報を破棄せずに競合する値をマージできる CRDT タイプを使用できます。

### 注意事項

- **UPDATE** によって変更される属性は、トリガーの古い行と新しい行を比較することによって決定されます。これは、属性が値を変更しない場合、明示的に設定されても変更されたもの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、どの行でも **a** を変更済み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に対して **a** を変更済みとしてマークしません。
- **INSERT** ステートメントの場合、新しい行と比較する古い行がないため、すべての属性が変更されると見なし、新しいタイムスタンプを割り当てます。これは、**INSERT** ステートメントに含まれず、デフォルト値を受け取った列にも適用されます。デフォルト値を持つ属性は検出できますが、自動的に含まれたか、明示的に指定されたかはわかりません。

これは、**INSERT** ステートメントで列の異なるサブセットを指定していても、列レベルの競合解決が **INSERT-INSERT** の競合では機能しないことを効果的に意味します。新しい行には、古い行よりもすべて新しいタイムスタンプがあります。

- 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に不可能な方法で制約に違反するのは簡単です。たとえば、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くと仮定します。

```
UPDATE t SET a = 100;
```

別のノードが同時に行います。

```
UPDATE t SET b = 500;
```

これらの各更新は、最初の行で実行された場合に有効であるため、各ノードに渡されます。ただし、他のノードに複製すると、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複製は停止します。

- タイムスタンプマッピングを格納するカラムは自動管理されます。予測できない影響を与える可能性があるため、クエリで値を指定またはオーバーライドしないでください。（とにかく、可能な場合は値を無視します。）

- タイムスタンプのマッピングはトリガーによって維持されますが、トリガーが実行される順序が重要です。したがって、タプルを変更し、`pgl_cld_` トリガーの後に実行されるカスタムトリガーがある場合、変更された列は正しく検出されません。

- 通常のタイムスタンプを使用して変更/コミットを注文する場合、競合する変更のタイムスタンプがまったく同じである可能性があります（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）。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のものではありません。この状況では、ノード ID をタイブレーカーとして使用します（高いノード ID が勝ちます）。これにより、すべてのノードに同じ変更が適用されます。
- 異なるノード間にクロックスキューが存在する可能性があります。やや予期しない動作を引き起こす可能性があります（タイムスタンプが逆になっているため、一見新しい変更を破棄します）、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。

```
SELECT bdr.alter_node_group_config(group, ignore_redundant_updates := false);
```

### 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、同時に変更される行の値のマージをサポートします。

各 CRDT 型は、`bdr.crdt_handlers` カタログに追加された追加のコールバックを備えた個別の PostgreSQL データ型として実装されます。マージプロセスは、ユーザーの操作を必要とせずに、適用側の BDR ライタ内で発生します。

CLCD に記載されているように、CRDT では、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。

必要なアクションは、`CREATE/ALTER TABLE` で整数などの標準の組み込みデータ型ではなく、特定のデータ型を使用することです。たとえば、1 つの通常の整数カウンターと 1 つの行がある次のテーブルについて考えます。

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

(次のページに続く)

(前のページからの続き)

```

 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つを失ったことを示しています。代わりに CRDT カウンターデータ型を使用すると、結果は次のようになります。

```

CREATE TABLE crdt_example (
    id          integer          PRIMARY KEY,
    counter bdr.crdt_gcounter    NOT NULL DEFAULT 0
);

ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_example,
    column_modify_timestamp, cts);

INSERT INTO crdt_example (id) VALUES (1);

```

2つのノードで同時に次の SQL を発行し、変更が適用されるのを待ちます。

```

UPDATE crdt_example
    SET counter = counter + 1    -- "reflexive" update
WHERE id = 1;

SELECT id, counter FROM crdt_example WHERE id = 1;
 id | counter
-----+-----
  1 |      2
(1 row)

```

この例は、CRDT により、競合する非同期同時更新に直面した場合でも、アキュムレーター列が正しく機能することを示しています。

`crdt_gcounter` タイプは、例に示すように、 $x = x + 1$  などの再帰 UPDATE SQL でのみ機能する状態ベースの CRDT タイプの例です。

`bdr.crdt_raw_value` 構成オプションは、クエリが現在の値を返すか、CRDT 型の完全な内部状態を返すかを決定します。デフォルトでは、現在の数値のみが返されます。`true` に設定すると、クエリは完全な状態の表現を返します。特殊なハッシュ演算子 (`#`) を使用すると、特殊な演算子を使用せずに現在の数値のみを要求できます (デフォルトの動作)。`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` でのみロードできます。

**注釈:** `bdr.crdt_raw_value` は、クライアントに返されるデータ、つまり選択リスト内の単純な列参照のみにフォーマットを適用します。クエリの他の部分 (`WHERE` 句や選択リスト内の式など) での列参照は、引き続き `#` 演算子を使用する必要があります。



CRDT データ型の別のクラスは、**デルタ CRDT** 型と呼ばれます。これらは、操作ベースの CRDT 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、変更が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 です。ただし、デルタ CRDT カウンターで行を更新する場合、古い行バージョンから始めて、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（ローカルバージョンなど）と比較します。標準 CRDT は NEW バージョンと LOCAL バージョンをマージしますが、デルタ CRDT は OLD バージョンと NEW バージョンを比較し、デルタを LOCAL バージョンに適用します。

CRDT タイプは、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。これにより、search\_path を調整せずに利用できるようになります。

重要な問題は、クエリの計画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 型は透過的に処理されます。ANALYZE とオプティマイザの両方が機能するため、他に何もしなくても見積もりとクエリ計画が正常に機能します。

## 状態ベースおよび操作ベースの CRDT

[1] の表記に従って、操作ベースと状態ベースの両方の CRDT が実装されます。

### 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るのではなく、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、次の操作ベースの CRDT が実装されています。

- `crdt_delta_counter` — `bigint` カウンター (インクリメント/デクリメント)
- `crdt_delta_sum` — `numeric` sum (インクリメント/デクリメント)

これらの型は、既存のデータ型 (たとえば、`crdt_delta_counter` は `bigint` のドメイン) を利用して、デルタを計算します。

このアプローチは、デルタの計算方法がわかっているタイプでのみ可能ですが、結果はシンプルで安価 (スペースと CPU の両方) であり、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、非同期および並行環境でこの値を確実にリセットできないことです。

---

**注釈:** カスタムデータ型を作成し、状態と最後の操作を保存することにより、より複雑な操作ベースの型を実装できます。(すべての変更が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) は失われますが、領域要件を除き、状態ベースの型と比較して利点はありません (たとえば、まだリセットできません)。(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データ型を直接使用することはできません。

現在、4 つの状態ベースの CRDT が実装されています。

- `crdt_gcounter` — `bigint` カウンター (インクリメントのみ)
- `crdt_gsum` — `numeric` 合計/カウンター (インクリメントのみ)
- `crdt_pncounter` — `bigint` カウンター (インクリメント/デクリメント)
- `crdt_pnsum` — `numeric` sum/counter (インクリメント/デクリメント)

通常、内部状態にはノードごとの情報が含まれるため、オンディスクサイズは増加しますが、追加の利点があります。カスタムデータ型を実装する必要性は、より多くのコード (イン/アウト関数と演算子) を意味します。

利点は、値を確実にリセットできること、変更が失われた場合の自己修復性（適切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から変更を受信できることです。

たとえば、ノード A で値が変更され、A と C 間のネットワーク問題が原因で C ではなく B に変更がレプリケートされるとします。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、A からの元の変更。操作ベースの CRDT では、ノード C は A-C ネットワーク接続が再び機能し始めるまで変更を受け取りません。

CvCRDT の主な欠点は、ディスク領域と CPU 使用率の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、各ノードに関する少しの情報が必要です。状態の複雑な性質（varlena タイプにシリアル化）は、CPU 使用率の増加を意味します。

## ディスク領域の要件

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。

操作ベースのタイプの場合、これは簡単です。タイプは他のタイプの上の単なるドメインであり、ノードの数に関係なく同じディスク領域要件があるためです。

- `crdt_delta_counter` — `bigint` と同じ（8 バイト）
- `crdt_delta_sum` — `numeric` と同じ（精度と位取りに応じて可変）

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、ノード数に依存しません。

状態ベースの型の場合、状況はより複雑です。すべての型は可変長（基本的に `bytea` 列として保存）であり、ヘッダーと、値を変更した各ノードの一定量のノードごとの情報で構成されます。

`bigint` バリエーションの場合、おおよそのサイズを計算する式は次のとおりです（N は、この値を変更したノードの数を示します）。

- `crdt_gcounter` — 32B (header) + N \* 12B (per-node)
- `crdt_pncounter` — 48B (header) + N \* 20B (per-node)

`numeric` バリエーションの場合、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。このような値をいくつ保持する必要があるかを示すには：

- `crdt_gsum` - 固定：20B (header) + N \* 4B (per-node) - 変数：(2 + N) `numeric` の値
- `crdt_pnsum` - 固定：20B (header) + N \* 4B (per-node) - 変数：(4 + 2 \* N) `numeric` の値

**注釈：** 複数のノードで値が更新されない場合、クラスター内にノードがいくつあっても構いません。更新が同時に行われたかどうかに関係ありません（競合が発生します）。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ていても構いません。状態を圧縮する方法はまだありません。

## 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ます）、通常の競合解決と CRDT マージの両方を実行する必要があります。

競合の解決が最初に行われ、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、破棄されたタプルの CRDT 列のデータが `applytuple` にマージされます。

---

**注釈:** この処理により、マージは毎回発生する必要があるため、単純な競合解決と比較して CRDT タイプがやや高価になります。これは、競合解決でいずれかの高速パス（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 型がない場合、競合の解決は使用可能な情報の半分（構成に応じてローカルまたはリモートの行）を破棄するため、これは理にかなっています。これは、データの損失を示します。

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結合できるため、データ損失の問題が解消されます。これにより、競合の報告が不要になります。

このため、CRDT マージで競合を完全に解決できる場合、つまり各列が次の 2 つの条件の少なくとも 1 つを満たす場合、競合レポートはスキップされます。

- 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 データ型を使用しているため、マージできます。

---

**注釈:** これは、CRDT 列がないがローカル/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がスキップされることを意味します。

---

## CRDT 値のリセット

CRDT 値をリセットすることはできますが、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、異なるノードが変更ストリームの異なる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。異なるノードが同時に、つまり他のノードからのリセットを監視する前に、リセットを開始することもできます。

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いく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。

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。他のノードの状態は破棄しますが、ローカルのみです。リモートノードのマージ機能によって追加され、値が発散し、最終的に他のノードでの変更が原因で返されます。

操作ベースの CRDT では、更新が `-cnt` の減算として解釈されるため、これは機能しているように見える場合があります。ただし、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試行すると、デルタが2回適用され、負の値が取得されます（リセットでは想定されません）。

`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、すべてのノードが操作のストリーム内の同じ位置でそれを見ることは保証されません。BDR は、同時発生 of データ異常につながる可能性があるため、同じ主キー値の再利用を特にお勧めしません。

状態ベースの CRDT 型は、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 ...;
```

「信頼できる」とは、値に複数の同時リセットと発散の2つの問題がないことを意味します。

操作ベースの CRDT タイプは、*Eager Replication* を使用してのみ確実にリセットできます。これにより、複数の同時リセットが回避されます。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。

## 実装された CRDT データ型

現在、6つの CRDT データ型が実装されています。

- 成長専用のカウンターと合計
- 正負のカウンターと合計
- デルタカウンターと合計

カウンターと合計の動作はほとんど同じですが、カウンターの型は整数ベース (`bigint`) であり、合計の型は10進ベース (`numeric`) です。

[1] で説明されている追加の CRDT タイプは、後で実装される可能性があります。

次のクエリを使用して、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;
```

## 成長専用カウンター (crdt\_gcounter)

- 非負の値を持つインクリメントのみをサポートします (value + int および counter + bigint 演算子)。
- # 演算子を使用するか、bigint にキャストして、カウンターの現在の値を取得できます。
- counter = value (アプリケーションのどこかで新しい値が計算される場合の一般的なパターン) のような単純な割り当てと互換性はありません。
- ! 演算子 (counter = !counter) を使用したカウンターの単純なリセットを許可します。
- 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
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;
```

## 成長のみの合計 (crdt\_gsum)

- 非負の値のインクリメントのみをサポートします (sum + numeric)。
- # 演算子を使用するか、numeric にキャストして、合計の現在の値を取得できます。
- sum = value (アプリケーションのどこかで新しい値が計算される場合の一般的なパターン) のような単純な割り当てと互換性がありません。
- ! 演算子 (sum = !sum) を使用した合計の単純なリセットを許可します。
- 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

-- 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;
```

## ポジティブネガティブカウンター (crdt\_pncounter)

- 正と負の両方の値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` にキャストして、カウンターの現在の値を取得できます。
- `counter = value` (アプリケーションのどこかで新しい値が計算される場合の一般的なパターン) のような単純な割り当てと互換性はありません。
- ! 演算子 (`counter = !counter`) を使用したカウンターの単純なリセットを許可します。
- `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
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;
```



## 正負の和 (crdt\_pnsum)

- 正と負の両方の値のインクリメントをサポートします (sum + numeric を介して)。
- then # 演算子を使用するか、numeric にキャストして、合計の現在の値を取得できます。
- sum = value (アプリケーションのどこかで新しい値が計算される場合の一般的なパターン) のような単純な割り当てと互換性はありません。
- ! 演算子 (sum = !sum) を使用した合計の単純なリセットを許可します。
- 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

-- 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` 列とまったく同じように機能します。
- 正と負の両方の値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 (アプリケーションのどこかで新しい値が計算される場合に一般的)。
- 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  
  
-- 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` 列とまったく同じように機能します。
- 正と負の両方の値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 (アプリケーションのどこかで新しい値が計算される場合に一般的)。
- 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;

-- 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;

```

[1] [https://en.wikipedia.org/wiki/Conflict-free\\_replicated\\_data\\_type](https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type)

## 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は送信されません。変更が `logical_decoding_work_mem` (PostgreSQL 13 以降) を超える場合、ディスクに書き込まれます。これは、特に大規模なトランザクションでは、サブスクライバーに到達するまでに遅延があり、パブリッシャーで追加の I/O が必要になる可能性があることを意味します。

PostgreSQL 14 以降、トランザクションは、パブリッシャーでコミットされる前にオプションでデコードしてサブスクライバーに送信できます。サブスクライバーは、受信した変更をステージングファイル（またはファイルのセット）に保存し、トランザクションのコミット時に適用します（またはトランザクションが中断した場合は破棄します）。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ます。

## BDR の機能強化

PostgreSQL の組み込み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書き込む必要はなくなりましたが、各サブスクリバ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た場合、作業（各サブスクリバが受け取った変更と関連するストレージ I/O）は無駄になります。

ただし、バージョン 3.7 以降、BDR は並列適用をサポートし、各サブスクリバで複数のライター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、サブスクリバのライター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、サブスクリバのディスクに保存する必要はありません。
- サブスクリバで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するのを待つ必要はありません。

## 警告

- 並列適用を有効にする必要があります。
- 多くの小さく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、ライター間で頻繁にデッドロックを引き起こす可能性があります。

---

**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。使用には注意してください。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場合があります。（トランザクションが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` を使用した**ノード構成**を使用
- グループレベルで、関数 `bdr.alter_node_group_config()` を使用した**グループ構成**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。

この設定を変更するには、設定を有効にするには、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する必要があります。これは、サーバーの再起動で実現できます。

`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からのトランザクションストリーミングを要求します。これがどのように提供されるかは、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_config()` を使用したグループ構成

ファンクション `bdr.alter_node_group_config()` を使用したグループ構成のパラメーター `streaming_mode` を使用して、グループトランザクションストリーミング構成を設定できます。

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 `:ref:bdr.node_group`<`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、取り得る値は `D` (`default`)、`W` (`writer`)、`F` (`file`)、`A` (`auto`)、および `O` (`off`)

### 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。`off` (トランザクションストリーミングを無効にする) 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。一部の状況 (`COPY` など) 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。

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。この決定は、いくつかの要因に基づいています。

- 並列適用がオフ (`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。)

- 並列適用がオンになっているが、すべてのライターがストリーミングされたトランザクションの処理で既にビジー状態である場合、新しいトランザクションがファイルにストリーミングされます。BDR ライタのステータスは `:ref:`bdr.writers`<bdr.writers>` を参照してください。

ライターへのストリーミングが可能な場合（つまり、無料のライターを利用できる場合）、トランザクションをライター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、次の表のようにグループとノードの設定の組み合わせに基づいています。

| 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` の場合、パブリッシャーがこれを行うように指示した場合にのみ、トランザクションはライター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ーにストリーミングするようにヒントしています。これらは競合がないことが知られており、ライターが安全に処理できます。

- COPY
- CREATE INDEX CONCURRENTLY

## モニタリング

サブスクライバーノードで `:ref:`bdr.stat_subscription`<bdr.stat_subscription>` ファンクションを使用して、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。

- `nstream_writer` —ライター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` —中断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。

## Durability and performance options

### 概要

同期または \* Eager Replication\*は、トランザクションをコミットする前にクラスターの少なくとも 2 つのノード間で同期します。この同期は、関連するがすべて個別に実装できるアプリケーションに関連する 3 つのプロパティを提供します。

- **耐久性**：複数のノードに書き込むと、クラッシュ回復力が向上し、クラッシュして再起動した後にデータを回復できます。
- **可視性**：クライアントへのコミット確認により、データベースはいくつかのノードセットでの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。
- **コミット後に競合なし**：クライアントは、トランザクションが最終的にすべてのノードに適用されることを信頼したり、クライアントにエラーを直接通知したりできます。

BDR は、同期レプリケーションのバリエーションを提供することにより、耐久性と可視性を保証する **CAMO とグループコミット** 機能を提供します。この機能は、フィジカルスタンバイ用の Postgres `synchronous_commit` 機能に似ていますが、大規模な分散システムにより多くの柔軟性を提供します。

Group Commit に加えて、BDR は現在 Group Commit と組み合わせることができない他の 2 つのモードも提供しています。

- **Commit At Most Once (CAMO)**。この機能は、ノードまたはネットワークに障害が発生した場合にトランザクションがコミット（およびレプリケート）されたかどうかを知るに関する問題を解決します。通常、COMMIT が処理されたかどうかを知るのは難しい場合があります。この機能を使用すると、新しいデータベース接続が以前の接続とは異なる場合でも、アプリケーションは何が起こったのかを知ることができます。この機能の詳細については、

*Commit At Most Once* を参照してください。

- **Eager Replication**。これは、コミットの前に競合をチェックするオプションの機能です。すべてのトランザクションはすべてのノードで同時に適用および準備され、レプリケーションの競合が検出されない場合にのみコミットされます。この機能により、パフォーマンスは低下しますが、強い一貫性が保証されます。この機能の詳細については、

*Eager All-Node Replication* を使用した **CAMO** を参照してください。

Postgres は、単方向ですが `synchronous variant` を提供する **Physical Streaming Replication (PSR)** を提供します。下位互換性のために、BDR は引き続き `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サポートしています。

**BDR を使用したレガシー同期レプリケーション** を参照してください。すべての場合に代わりに **CAMO とグループコミット**

を使用することをお勧めします。

## 用語と定義

BDR ノードはさまざまなロールを取ることができます。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でも関係ありません。

- **オリジン**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し、他の BDR 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するのはノードです。
- **パートナーノード**は、グループコミットまたは CAMO 要件に従ってトランザクションを確認することが期待される BDR 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 BDR ノード、つまりオリジンとそのすべてのパートナーノードのグループであり、少数またはすべてのピアノードです。

## 比較

BDR で使用可能な同期レプリケーションのほとんどのオプションは、さまざまなレベルの同期を可能にし、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との間でさまざまなトレードオフを提供します。

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 確認を受け取った後、クライアントがレプリケートされたピアノードに期待できることをまとめたものです。Mode 列は、バリエーションに応じて異なる意味をとります。BDR を使用した PSR およびレガシー同期レプリケーションの場合、`synchronous_commit` 設定を参照します。CAMO の場合、`bdr.enable_camo` 設定を指します。また、Group Commit の場合、

`commit scope configuration` の確認要件を参照します。

| Variant      | Mode                    | Received | Visible | Durable |
|--------------|-------------------------|----------|---------|---------|
| async BDR    | off (default)           | no       | no      | no      |
| PSR          | remote_write (2)        | yes      | no      | no (1)  |
| PSR          | on (2)                  | yes      | no      | yes     |
| PSR          | remote_apply (2)        | yes      | yes     | yes     |
| Group Commit | 'ON received' nodes     | yes      | no      | no      |
| Group Commit | 'ON replicated' nodes   | yes      | no      | no      |
| Group Commit | 'ON durable' nodes      | yes      | no      | yes     |
| Group Commit | 'ON visible' nodes      | yes      | yes     | yes     |
| CAMO         | remote_write (2)        | yes      | no      | no      |
| CAMO         | remote_commit_async (2) | yes      | yes     | no      |
| CAMO         | remote_commit_flush (2) | yes      | yes     | yes     |
| Eager        | n/a                     | yes      | yes     | yes     |
| legacy (3)   | remote_write (2)        | yes      | no      | no      |
| legacy (3)   | on (2)                  | yes      | yes     | yes     |
| legacy (3)   | remote_apply (2)        | yes      | yes     | yes     |

- (1) OS に書き込まれ、OS が実行されたままで Postgres のみがクラッシュする場合。



- (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）、それ以外の場合は非同期 BDR のデフォルトの値が適用されます。
- (3) 非推奨。代わりに Group Commit の使用を検討してください。

受信により、ネットワーク全体または部分的な障害が発生した場合でも、正常に動作しているピアが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が保証されます。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た場合で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。同期と見なされるすべてのモードがこの保護を提供します。

可視性は、トランザクションがリモートで適用されたことを意味します。他のすべてのクライアントは、コミットがオリジンノードによって確認された直後にこの保証を提供すると、すべてのノードでトランザクションの結果を表示します。可視性がないと、接続されている他のクライアントがトランザクションの結果を表示できず、古い読み取りが発生する可能性があります。

耐久性はピアノードのストレージに関連し、ピアノードのクラッシュと復旧後のデータの損失に対する保護を提供します。これは、データの受信（物理ストリーミングレプリケーションなど）または可視性（Group Commit、CAMO、および Eager など）に関連します。前者はクラッシュ後の再送信の必要性を排除し、後者はリスタート後も可視性が維持されることを保証します。

## 操作の内部タイミング

さまざまなモードがどのように機能するかをよりよく理解するには、PSR と BDR がトランザクションを異なる方法で適用することを理解すると役立ちます。

物理ストリーミングレプリケーションの場合、操作の順序は次のとおりです。

- Origin 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で可視にします。
- ピアノードは変更を受け取り、書き込みを発行します。
- ピアは受信した変更をディスクにフラッシュします。
- ピアは変更を適用し、トランザクションをローカルに表示します。

BDR では、操作の順序が異なります。

- Origin 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で可視にします。
- ピアノードは、メモリ内の適用キューへの変更を受け取ります。
- ピアは変更を適用し、トランザクションをローカルに表示します。
- 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。

Group Commit、CAMO、および Eager の場合、オリジンノードはトランザクションをローカルで可視化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。

- Origin は、準備またはプリコミットレコードを WAL にフラッシュします。
- ピアノードは、メモリ内の適用キューへの変更を受け取ります。
- ピアは変更を適用し、トランザクションをローカルに表示します。

- 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。
- オリジンはトランザクションをコミットしてローカルに表示します。

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   |
| BDR          | apply first               | after WAL flush of commit record   |
| Group Commit | apply first               | before COMMIT on origin            |
| CAMO         | apply first               | before COMMIT on origin            |
| Eager        | apply first               | before COMMIT on origin            |

## 構成

次の表は、デフォルト以外の値 (req) またはオプション (opt) に設定する必要があるが、特定のバリエーションに影響を与える構成設定の概要を示しています。

| setting (GUC)                        | Group Commit | CAMO | Eager | PSR (1) |
|--------------------------------------|--------------|------|-------|---------|
| synchronous_standby_names            | n/a          | n/a  | n/a   | req     |
| synchronous_commit                   | n/a          | n/a  | n/a   | opt     |
| synchronous_replication_availability | n/a          | opt  | n/a   | opt     |
| bdr.enable_camo                      | n/a          | req  | n/a   | n/a     |
| bdr.commit_scope                     | req          | n/a  | opt   | n/a     |
| bdr.global_commit_timeout            | opt          | opt  | opt   | n/a     |

- (1) この列の値は、BDR と組み合わせて使用されている `synchronous_commit` および `synchronous_standby_names` にも適用されます。

## 計画的なシャットダウンと再起動

受信確認または CAMO を `remote_write` と組み合わせて Group Commit 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消費されます。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が破棄されるため、停止したノードがキュー内のトランザクションを「削除」します。起点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、データが失われ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` または `fast shutdown` を使用します。このアプローチにより、必要な同期レベルが維持され、データの損失が防止されます。

## BDR を使用したレガシー同期レプリケーション

---

**注釈:** このアプローチはお勧めしません。代わりに *CAMO* と *グループコミット* の使用を検討してください。

---

### 使用法

BDR を使用して同期レプリケーションを有効にするには、関連する BDR 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* の使用は、非 BDR スタンバイノードの要件と競合しない限り、ある程度の柔軟性を提供します。

追加したら、デフォルトは `on` である `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。この設定は、`synchronous_standby_names` に追加すると既に同期レプリケーションが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。

BDR はトランザクションを永続化する前に適用するため、値 `on` と `remote_apply` は同等です（論理レプリケーションの場合）。

### グループコミットへの移行

BDR のグループコミット機能は、`synchronous_commit` および `synchronous_standby_names` から独立して構成されています。代わりに、`bdr.commit_scope` GUC では、トランザクションごとにスコープを選択できます。また、各ノードで個別に構成された `synchronous_standby_names` の代わりに、グループコミットはグローバルに同期されたコミットスコープを使用します。

---

**注釈:** `synchronous_standby_names` と Commit Scopes の文法は似ていますが、前者は起点ノードを説明しませんが、後者は説明します。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、多数決についての推論が容易になり、オリジンノードがトランザクションの耐久性と可視性にも寄与することを反映しています。

---

### Group Commit

Group Commit の目的は、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損失から保護することです。これを行うには、COMMIT 時にトランザクションを正常に受信して確認するために複数の BDR ノードを要求します。

## 要件

通常の操作中、Group Commit はアプリケーションに対して完全に透過的です。フェールオーバー時に、調整フェーズは、アプリケーションまたは中間のプロキシによって明示的にトリガーまたは統合される必要があります。HARP はグループコミットのネイティブサポートを提供し、調整フェーズをトリガーするため、これをクライアントに等しく透過的にします。

オリジンノードでは、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、下で 2 フェーズコミットを使用します。したがって、ノードごとに発生するそのようなすべてのトランザクションを処理するのに十分な高さに `max_prepared_transactions` を構成します。

## 構成

Group Commit を使用するには、最初に `commit_scope` を定義します。これにより、トランザクションのコミットに関係する BDR ノードが決まります。スコープを確立したら、次のように Group Commit を使用するようにトランザクションを構成できます。

```
BEGIN;
SET LOCAL bdr.commit_scope = example_scope;
...
COMMIT;
```

この例では、以前にコミットスコープを次のように定義している可能性があります。

```
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := example_bdr_group,
    rule := ANY 2 (example_bdr_group)
);
```

これは、`example_bdr_group` という名前のノードグループが存在し、少なくとも 2 つの BDR ノードをメンバーとして直接またはサブグループに含むことを前提としています。`example_scope` でコミットされたトランザクションには、グループ内の BDR ノードからの追加の確認が 1 つ必要です。起点ノードとともに、これはグループ外の「ANY 2」ノードを説明し、トランザクションはコミット後の可視性と持続性が保証されます。

## オリジングループ

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よって異なります。これをアプリケーションに対して透過的にするために、BDR では、トランザクションの発生元に応じてコミットスコープがさまざまなルールを定義できます。

たとえば、ノードが左側と右側の 2 つのデータセンターにまたがる EDB Postgres 分散クラスターを考えます。最上位の BDR ノードグループの名前が `top_group` であるとし、次のコマンドを使用して、サブグループを設定し、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しますが、リモートからは 1 つのノードのみを確認するコミットスコープを作成できます。

```

- - 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
);

- - create a commit scope with individual rules
- - for each sub-group
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) AND ANY 1 (right_dc)
);
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) AND ALL (right_dc)
);

```

## 確認レベル

BDR ノードは、*Commit At Most Once* と同様に、さまざまな時点でトランザクションの確認を送信できます。保護レベルを高めると、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 BDR ノードは、ローカルアプリケーションを起動する前に、トランザクションを受信した直後に確認します。
- **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、同時実行トランザクションから見えるようになったら、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 を使用して、これらの確認レベルを括弧内のノードグループ定義に追加できます。

- ANY 2 (right\_dc) ON replicated
- ALL (left\_dc) ON visible (デフォルトで省略可)
- ALL (left\_dc) ON received AND ANY 1 (right\_dc) ON durable

## リファレンス

### コミットスコープの文法

参考までに、コミットスコープの文法は次のように構成されています。

```
commit_scope:
    confirmation [AND ...]

confirmation:
    node_def (ON [received|replicated|durable|visible])

node_def:
    ANY num (node_group [, ...])
    | MAJORITY (node_group [, ...])
    | ALL (node_group [, ...])
```

---

**注釈:** *synchronous\_standby\_names* とコミットスコープの文法は非常に似ていますが、前者は起点ノードを考慮しないことに注意することが重要です。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、多数決についての推論が容易になり、オリジンノードがトランザクションの耐久性と可視性にも寄与することを反映しています。

---

### コミットスコープルールの追加

ファンクション `bdr.add_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。EDB Postgres 分散クラスター内のすべてのノードでルールが同じ場合、最上位ノードグループに対してこの関数を 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` で複数回呼び出すことができますが、トランザクションの起点によって異なる起点ノードグループとコミットスコープのルールが異なります。

## 概要

```
bdr.add_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT)
```

## コミット範囲ルールの変更

コミットスコープ内の単一のオリジンノードグループの特定のルールを変更するには、ファンクション `bdr.alter_commit_scope` を使用できます。

### 概要

```
bdr.alter_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT)
```

## コミットスコープルールの削除

`bdr.remove_commit_scope` を使用して、コミット範囲内の単一のルールを削除できます。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.remove_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME)
```

---

**注釈:** ノードグループによってまだ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許可されていません

---

## Eager Replication

コミット後の競合を防ぐには、`bdr.commit_scope` パラメーターを `global` に設定します。`local` のデフォルト設定は Eager Replication を無効にするため、[競合の監視](#)で説明されているように、BDR はコミット後に変更を適用し、潜在的な競合を解決します。

このモードでは、BDR は内部で 2 フェーズコミット (2PC) を使用して、ローカルコミットの前に競合を検出および解決します。これは、アプリケーションの通常の COMMIT を暗黙的な 2 フェーズコミットに変換し、元のノードがコミットを続行する前に、すべてのピアノードでトランザクションを準備する必要があります。準備フェーズ中に少なくとも 1 つのノードがダウンしているか到達できない場合、コミットは `bdr.global_commit_timeout` の後にタイムアウトし、トランザクションが中止されます。PostgreSQL の明示的な 2PC に存在する一時テーブルの使用に制限はありません。

準備ができると、Eager All-Node Replication は Raft を使用してコミットの決定に到達します。障害が発生した場合、これにより、残りの大部分のノードが一致するコミットまたはアボートの決定に到達して、ト

ランザクションを終了できます。これにより、トランザクションによってロックされたオブジェクトとリソースのブロックが解除され、クラスターが続行できます。

すべてのノードが動作可能なままである場合、オリジンはすべてのノードがコミットした後にのみクライアントにコミットを確認し、コミット後にすべてのノードでトランザクションがすぐに表示されるようにします。

## 要件

Eager All-Node Replication は、準備されたトランザクションを内部で使います。したがって、すべてのレプリカノードは、場合によってはローカルで 2 フェーズコミットと CAMO トランザクションに加えて、すべての着信トランザクションを処理できるように `max_prepared_transactions` を構成する必要があります。（

Configuration: Max-prepared transactions を参照）。すべてのノードで同じに、CAMO または Eager

All-Node Replication が使用されるクラスター全体の同時トランザクションの最大数をカバーするのに十分な高さで構成することをお勧めします。それ以外は、特別な構成は必要なく、すべての EDB Postgres 分散クラスターは Eager All-Node トランザクションを実行できます。

## 使用法

Eager All-Node Replication を有効にするには、次に示すように、クライアントはセッションレベルまたは個々のトランザクションでグローバルコミットスコープに切り替える必要があります。

```
BEGIN;  
  
... other commands possible...  
  
SET LOCAL bdr.commit_scope = global;  
  
... other commands possible...
```

クライアントは、トランザクションの最後に COMMIT を発行し続け、BDR に 2 つのフェーズを管理させることができます。

```
COMMIT;
```



## エラー処理

BDR 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 を含むいずれの場合にもお勧めします。)

起点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 `bdr.global_commit_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、必要に応じてトランザクションを最終的にコミット (またはロールバック) して、決定してさらに進んだ可能性のあるノードの大部分と調整します。

## CAMO を使用した Eager All-Node Replication

Eager All-Node Replication は CAMO を超えて、それを意味します。`bdr.commit_scope` が `global` に設定されている場合、`bdr.enable_camo` も有効にする必要はありません。また、CAMO ペアを `bdr.add_camo_pair()` で構成する必要もありません。

CAMO パートナーのロールで他のアクティブな BDR ノードを使用して、トランザクションのステータスを照会できます。ただし、この非 CAMO 使用法を `require_camo_partner = false` の 3 番目の引数で `bdr.logical_transaction_status` 関数に示す必要があります。それ以外の場合、CAMO 構成がないと不平を言う場合があります (Eager トランザクションには必要ありません)。

その関数の構成と呼び出しの違いを除き、クライアントは `show-camo` で説明されているプロトコルに従う必要があります。リファレンスクライアントの実装は、要求に応じて顧客に提供されます。

## 制限事項

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。明示的な 2 フェーズコミットもサポートしていません。これらは後のリリースで許可される可能性があります。TRUNCATE コマンドは許可されています。

クラッシュした回復不能な BDR ノードをフィジカルスタンバイに置き換えることは、現在、Eager All-Node トランザクションと組み合わせてサポートされていません。

BDR は現在、グローバルコミットスコープのみを提供しています。後のリリースでは、可用性を高めるために少ないノードで Eager Replication をサポートします。

Eager All-Node Replication と `synchronous_replication_availability = 'async'` を組み合わせることはできません。両方を構成しようとすると、エラーが発生します。

Eager All-Node トランザクションは、現在、Decoding Worker 機能またはトランザクションストリーミングと組み合わせてサポートされていません。Eager を使用したインストールでは、BDR ノードグループで `enable_wal_decoder` および `streaming_mode` を無効にしておく必要があります。

同期レプリケーションは、Eager 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 つは互換性がありません。一緒に使用しないでください。したがって、Eager All-Node トランザクションを使用するときは常に、`synchronous_standby_names` で BDR ノードが構成されていないことを確認してください。フィジカルスタンバイとして動作する非 BDR ノードへの同期レプリケーションを使用できます。

## 一般的な Eager Replication の効果

### コミットのレイテンシーの増加

同期ステップを追加することは、ノード間の追加の通信を意味し、コミット時に追加のレイテンシーが発生します。Eager All-Node Replication は、約 2 つのネットワークラウンドトリップを追加します（最悪の場合、最も遠いピアノードへ）。ロジカル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。

ピアノードは、トランザクションのローカル準備を確認する前に、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、コミットのレイテンシーがさらに増加します。これは `synchronous_commit` 設定とは無関係で、`bdr.commit_scope` が `global` に設定されている場合は常に適用されます。

### アボート率の増加

---

**注釈:** Eager Replication のパフォーマンスは現在、予想外に遅いことが知られています（約 10 TPS のみ）。これは次のリリースで改善される予定です。

---

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 BDR を使用しても、COMMIT 時のエラーはまれです。追加の同期ステップによりエラーのソースが追加されるため、アプリケーションは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（通常は再試行ループを適用します）。

アボートの率は、ワークロードのみに依存します。多くの行を変更する大規模なトランザクションは、他の同時実行トランザクションと競合する可能性ははるかに高くなります。

### Commit At Most Once

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するのを防ぐことです。

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信しないため、トランザクションがコミットされたかどうか分かりません。

アプリケーションは、次の 2 つのオプションのいずれかを簡単に決定できません。

- データが 2 回入力される場合があるため、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、価値の高い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、トランザクションが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。これはアプリケーションの設計に依存し、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。

BDR の CAMO 機能は、より一般的な解決策を提供し、INSERT を必要としません。 `bdr.enable_camo` または `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、既に割り当てられている場合、トランザクション識別子を含むメッセージを受け取ります。それ以外の場合、トランザクション

の最初の書き込み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 BDR ノードに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の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 2 つのモードのいずれかで動作します。

- ペアモード
- Eager All Node Replication あり

ペアモードでは、CAMO は同じトップレベルの BDR グループの 2 つの BDR マスター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って機能します。この動作モードでは、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（必要に応じて）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を「オリジン」、これらのトランザクションを確認するノードを「パートナー」と呼ぶ場合があります。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。

*Eager All-Node Replication* を使用した CAMO と組み合わせると、

CAMO はすべてのピア（つまり、完全な BDR マスターノード）が CAMO パートナーとして機能できるようにします。このモードでは、指定された CAMO パートナーを構成する必要はありません。

**警告:** CAMO で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。保護を得るには、パラメーターを有効にするだけでは不十分です。リファレンスクライアントの実装は、要求に応じて顧客に提供されます。

## 要件

CAMO を使用するには、アプリケーションは明示的な COMMIT メッセージを別の要求として発行する必要があります（マルチステートメント要求の一部としてではありません）。CAMO は、プロシージャ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の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

既存の EDB Postgres 分散クラスターがノード **node1** および **node2** で構成されていると仮定します。両方のノードは **bdrdemo** という BDR 対応データベースの一部であり、両方とも同じノードグループ **mygroup** の一部です。ノードを相互に CAMO パートナーとして構成できます。

1. ノード **node1** および **node2** が **mygroup** ノードグループの一部である EDB Postgres 分散クラスターを作成します。

1. 一方のノードで関数 `bdr.add_camo_pair()` を実行します。

```
SELECT bdr.add_camo_pair(mygroup, node1, node2);
```

CAMO が提案する COMMIT エラー処理を使用するようにアプリケーションを調整します。

サーバーレベルで CAMO を有効にすることはお勧めしません。必要ない場合でも、すべてのトランザクションのレイテンシーが高くなります。代わりに、セッションまたはトランザクションレベルで CAMO をオンにして、個々のトランザクションに対して選択的に有効にします。

セッションレベルで CAMO を有効にするには：

```
SET bdr.enable_camo = remote_commit_flush;
```

トランザクションを開始してからコミットする前に、個々のトランザクションで CAMO を有効にするには：

```
SET LOCAL bdr.enable_camo = remote_commit_flush;
```

CAMO を有効にする `bdr.enable_camo` の有効な値は次のとおりです。

- `off` (デフォルト)
- `remote_write`
- `remote_commit_async`
- `remote_commit_flush` または `on`

各モードの動作の詳細については、同期レプリケーションモードの [違いのあるノード間の比較](#) を参照してください。`bdr.enable_camo = off` を設定すると、この機能が無効になります。

## Eager All-Node Replication を使用した CAMO

Eager All-Node Replication で CAMO を使用するために、どちらのノードも変更する必要はありません。トランザクションの開始後にグローバルコミットスコープを有効にすれば十分です。`bdr.enable_camo` を設定する必要はありません。

```
BEGIN;  
SET LOCAL bdr.commit_scope = global;  
...  
COMMIT;
```

指定されたとおりに COMMIT エラー処理を使用するようにアプリケーションを調整する必要がありますが、使用可能な BDR ノードに自由に接続して、トランザクションのステータスを照会できます。

## 障害シナリオ

さまざまな構成でさまざまな障害シナリオが発生します。

### 受信側でのデータの永続性

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、オリジンノードに対して確認されます。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が CAMO パートナーで自分自身で回復できない可能性があります。CAMO パートナーノードが回復するとトランザクションを再分散するため、これは CAMO オリジン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、ローカルモードだけでなく、リモート書き込みとの組み合わせにも適しています。

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します。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機能せず、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します。

### ローカルモード

`synchronous_replication_availability = 'async'` の場合、ノード（つまり、マスター）はその 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。そうでない場合は、一時的にローカルモードに切り替わります。ローカルモードの場合、ノードは CAMO モードに戻るまでトランザクションをローカルにコミットします。

これにより、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し、データの一貫性が失われる可能性があります。

CAMO パートナーが準備完了に切り替えるには、接続する必要があり、推定キャッチアップ間隔が `bdr.global_commit_timeout` を下回る必要があります。CAMO パートナーの現在の準備ステータスは `bdr.is_camo_partner_ready` で確認できますが、`bdr.node_replication_rates` はキャッチアップ時間の現在の見積もりを提供します。

CAMO 保護からローカルモードへの切り替えは、コミットが `bdr.global_commit_timeout` を超えるか、CAMO パートナーが既にわかっている場合はコミット時に切断された場合にのみ CAMO されます。このスイッチは、推定キャッチアップ間隔とは無関係です。Raft にローカルモードへの切り替えを必要とするように CAMO ペアが構成されている場合、このスイッチには大部分のノードが動作している必要があります（

`bdr.add_camo_pair` の `require_raft`

フラグを参照）。これにより、分離されたノードがローカルモードに切り替わることによるスプリットブレインの状況を防ぐことができます。CAMO ペアに `require_raft` が設定されていない場合、オリジンノードはすぐにローカルモードに切り替わります。

CAMO パートナーへの TCP 接続のキープアライブとタイムアウトを制御する PostgreSQL 設定を使用して、送信ノードでの検出を構成できます。wal\_sender\_timeout は、ノードがローカルモードに切り替えるまで CAMO パートナーを待機する時間です。さらに、bdr.global\_commit\_timeout 設定は、CAMO パートナーに到達できないために COMMIT が発生する可能性のある最大遅延にトランザクションごとの制限を設定します。wal\_sender\_timeout よりも低い場合がありますが、これは同期スタンバイにも影響し、応答性と安定性の良い妥協点を見つける必要があります。

ローカルモードから 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よって異なります。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 パートナーがオリジン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 パートナーで蓄積された遅延により、CAMO 保護モードへの切り替えがさらに遅れます。

通常の CAMO 操作とは異なり、ローカルモードでは追加のコミットオーバーヘッドはありません。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ようにするため、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としても、そのような状況では遅延が増加するだけであり、CAMO 保護を再確立できません。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、BDR 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、これはローカル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の要件を考慮して、ローカル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、いくつかの詳細を提供します。

#### 例：対称ノードペア

この例では、相互に CAMO パートナーである 2 つの BDR ノードを使用したセットアップを検討します。これは、BDR4 以降で可能な唯一の構成です。

この構成により、両方のノードで CAMO 動作が有効になります。したがって、競合が発生しない可能性が高い場合など、複数のノードで同時に書き込むことが許容されるワークロードパターンに適しています。

#### ローカルモードの場合

ローカル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の 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。



## ローカルモードなし

ローカルモードが許可されていない場合、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の 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。

- ノードが回復するまで、新しい書き込みトランザクションは防止されます。

## アプリケーション使用

### 概要と要件

CAMO は、クライアント側での再試行ループと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、一時的なエラーの場合、クライアントはトランザクションをリトライ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（両方とも 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のトランザクションステータスを確認する目的のみ、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、アプリケーションに追加の永続化レイヤーは必要ありません。アプリケーションに障害が発生した場合、再起動するのはデータベース内の情報のみです。

### CAMO ペアの追加

ファンクション `bdr.add_camo_pair()` は、BDR ノードの既存のペアを対称 CAMO ペアとして構成します。

`require_raft` オプションは、`synchronous_replication_availability` が `async` に設定されている場合にローカルモードに切り替える方法と時期を制御し、一般的にそのような切り替えを許可します。

### 概要

```
bdr.add_camo_pair(node_group text, left_node text, right_node text,
                  require_raft bool)
```

注釈: `left` と `right` の名前には特別な意味はありません。

**注釈:** BDR バージョン 4.0 以降、対称 CAMO 構成のみがサポートされています。つまり、ペアの両方のノードが互いの CAMO パートナーとして機能します。

---

### CAMO ペアの構成の変更

ファンクション `bdr.alter_camo_pair()` を使用すると、`require_raft` を切り替えることができます。現在、ペアリングのノードを変更することはできません。代わりに、`bdr.remove_camo_pair` の後に `bdr.add_camo_pair` を使用する必要があります。

#### 概要

```
bdr.alter_camo_pair(node_group text, left_node text, right_node text,  
                    require_raft bool)
```

### CAMO ペアの削除

関数 `bdr.remove_camo_pair()` は、2つのノードの CAMO ペアを削除し、これらの2つのノードで `bdr.enable_camo` による CAMO トランザクションの使用を禁止します。

#### 概要

```
bdr.remove_camo_pair(node_group text, left_node text, right_node text)
```

---

**注釈:** `left` と `right` の名前には特別な意味はありません。

---

### CAMO パートナー接続ステータス

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。



## 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 送信者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て確認を送信できるかどうかを示すブール値。

## CAMO パートナーの準備

ファンクション `bdr.is_camo_partner_ready` 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準備ステータスを確認できます。その下で、これはローカルモードへの/からの切り替えをトリガーします。

## 概要

```
bdr.is_camo_partner_ready()
```

## 戻り値

CAMO パートナーがタイムリーに（`bdr.global_commit_timeout` が期限切れになる前に）ローカルノードから発信されたトランザクションを確認すると合理的に期待できるかどうかを示すブール値。

---

**注釈:** この関数は、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。

---

## CAMO パートナーを取得する

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。

```
bdr.get_configured_camo_partner()
```

## CAMO パートナーからの適用キューが消費されるのを待ちます

関数 `bdr.wait_for_camo_partner_queue` は、デフォルトで CAMO パートナーノードを照会する `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、エラーが発生します。

### 概要

```
bdr.wait_for_camo_partner_queue()
```

## CAMO ノード間のトランザクションステータス

この機能により、CAMO トランザクションが完全に解決されるのを待機できます。

```
bdr.camo_transactions_resolved()
```

### 取引状況照会機能

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、アプリケーションで次のファンクションを使用する必要があります。

```
bdr.logical_transaction_status(node_id, xid, require_camo_partner)
```

ペアモードで使用する CAMO では、CAMO ペアの一部であるノードでのみこの機能を使用します。Eager レプリケーションとともに、すべてのノードで使用できます。

いずれの場合も、コミットが発行されてから 15 分以内に関数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削除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、この関数は受信キューが消費されて完全に適用されるのを待ちます。これにより、受信したがまだ適用されていないトランザクションに対する早期の否定的な回答が防止されます。

その名前にもかかわらず、これは常に読み取り専用の操作ではありません。ステータスが不明の場合、CAMO パートナーはトランザクションをコミットするかアボートするかを決定し、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関数を呼び出さないでください。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。

## 概要

```
bdr.logical_transaction_status(node_id OID,
                               xid      OID,
                               require_camo_partner BOOL DEFAULT true)
```

## パラメーター

- **node\_id** — トランザクションの発生元の BDR ノードのノード ID。通常、PQ パラメータ `bdr.local_node_id` から COMMIT する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、PQ パラメーター `transaction_id` から COMMIT する前にクライアントが取得します (`enable_camo` を on、`remote_write`、`remote_commit_async`、または `remote_commit_flush` に設定する必要があります。Commit At Most Once settings を参照)
- **require\_camo\_partner** — デフォルトは `true` で、構成チェックを有効にします。これらのチェックを無効にし、Eager All-Node Replication によって保護された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

この関数は、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表示され、最終的に他のすべての BDR ノードに複製されます。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ました。他の BDR ノードにレプリケートされません。クライアントは、トランザクションを再試行するか、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っ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、オリジンノードから切断し、CAMO パートナーに再接続して、代わりに照会することです。間にロードバランサーまたはプロキシがあり、クライアントが照会するノード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'` に切り替わるまでのみ繰り返しポーリングできます。

Eager All-Node Replication 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。これは、この場合、まだレプリケートされていない（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 BDR ノードで `in progress` の結果が得られる可能性があることを意味します。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、将来のものであるか、その特定のノードにまだレプリケートされていないか、過去すぎるためです。そのようなトランザクションのステータスはまだないか、わかっていません。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、エラー時に関数呼び出しを再試行する準備ができている必要があります。

## DDL とグローバルロックとの相互作用

CAMO によって保護されたトランザクションには、DDL 操作を含めることができます。ただし、DDL はグローバルロックを使用します。これにより、ノード間の同期が既に提供されます。詳細については、[DDL ロックの詳細](#)を参照してください。

CAMO と DDL を組み合わせると、待ち時間が長くなり、グローバルデッドロックが発生する可能性も高くなります。したがって、比較的低い `bdr.global_lock_timeout` を使用することをお勧めします。これにより、DDL が中止されるため、妥当な時間でデッドロックが解決されます。

## 非トランザクション DDL

次の DDL 操作はトランザクションブロックで許可されていないため、CAMO 保護のメリットを得られません。これらの場合、CAMO は内部で自動的に無効になります。

- すべての同時インデックス操作（CREATE、DROP、および REINDEX）
- REINDEX DATABASE、REINDEX SCHEMA、および REINDEX SYSTEM
- VACUUM
- パラメーターなしの CLUSTER
- ALTER TABLE DETACH PARTITION CONCURRENTLY
- ALTER TYPE [enum] ADD VALUE
- ALTER SYSTEM
- CREATE および DROP DATABASE
- CREATE および DROP TABLESPACE
- ALTER DATABASE [db] TABLESPACE

## CAMO の制限

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るため、切断された場合は、CAMO パートナーからトランザクションステータスをすぐに要求するようにアプリケーションをコーディングします。障害が発生してからステータスを要求するまでの遅延をできるだけ短くします。アプリケーションは、15 分以上保存される CAMO の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、未処理のトランザクションが終了するのを待ってからシャットダウン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持つ単一のステートメントにすることはできません。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 CAMO を使用することもできません。

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はまだ解決していません。CAMO および Eager レプリケーションオプションでは、NOTIFY SQL コマンドまたは `pg_notify()` ファンクションは許可されません。LISTEN または UNLISTEN も許可しません。

- 変更を再生するとき、CAMO トランザクションは他のトランザクションと同じように競合を検出する場合があります。タイムスタンプの競合検出を使用する場合、CAMO トランザクションはオリジンノードの準備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身で可視になる前です。
- CAMO は現在、トランザクションストリーミングに対応していません。CAMO の使用を計画している場合は、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。これは、グローバルにまたは BDR 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。

## 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。アプリケーションは、非同期レプリケーションよりもコミットの待機時間が長くなります。これは、主に関連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、適切なトランザクションスループットが得られます。

トランザクションを確認する CAMO パートナーは、トランザクション状態を保存する必要があります。非 CAMO 操作と比較して、これには、オリジンから適用されるトランザクションごとに追加の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、BDR クラスタ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。

## CAMO とグループコミット

CAMO は現在 **CAMO とグループコミット** では動作しません。

## Lag control

BDR オリジン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パイノードに安全にレプリケートできる速度を超える場合があります。この不均衡が高可用性アプリケーションで一定期間を超えてまたは慢性的に続く場合、ディザスタリカバリーまたはビジネス継続性計画に関連する組織の目標が満たされない可能性があります。

レプリケーションラグ制御 (RLC) 機能は、動的レート制限デバイスを使用してこの不均衡を調整し、BDR グループノード間のデータフローがこれらの組織の目標に準拠するように設計されています。これは、BDR ノード間のレプリケーション遅延の範囲を制御することによって行います。

これらの目的の一部は次のとおりです。

- 目標復旧ポイント（RPO）は、計画外のイベントが原因で失われる可能性のあるデータの最大許容量を指定し、通常は時間で表されます。レプリケートされないシステムでは、RPO を使用してバックアップ間隔を設定し、コミットされたデータが障害によって失われるリスクを制限します。レプリケートされたシステムの場合、RPO は、1 つ以上のピアノードに安全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、有限のストレージ制約があることを認めています。このストレージには、データベースファイル、WAL、および操作を続行するために必要な一時ファイルまたは中間ファイルが含まれます。レプリケートされたシステムの場合、遅延が増加すると、これらのストレージリソースに対する要求も増加します。
- グループ弾性目標（GEO）は、ノードがピアノードに受け入れられない形で保存できないクリップで新しいデータを発信していないことを保証します。その場合、その条件の検出をデータベースノードの数を拡張する決定の 1 つのメトリックとして使用できます。同様に、その状態が緩和すると、データベースノードの数を縮小するという決定に影響を与える可能性があります。

ラグ制御は、クライアント接続が READ WRITE トランザクションをコミットできる速度を制御することにより、レプリケーションの遅延を管理します。レプリケーションラグは、達成する目的に応じて、ラグタイムまたはラグサイズとして測定されます。トランザクションのコミット率は、構成された BDR のコミット遅延時間を使用して調整されます。

## 要件

ラグ制御の使用を開始するには：

1. すべてのデータベースアプリケーションで許容できる最大のコミット遅延時間 `bdr.lag_control_max_commit_delay` を決定します。
1. 使用するラグメジャーを決定します。ラグサイズ `bdr.lag_control_max_lag_size` またはラグタイム `bdr.lag_control_max_lag_time` のいずれかを選択します。
1. ステップ 2 で最小許容ノード数として選択したラグメジャーを満たす必要があるグループ `bdr.lag_control_min_conforming_nodes` 内の BDR ノードの数を決定します。

## 構成

ラグ制御は、`postgresql.conf` 構成パラメーターを使用してグループ内の各 BDR ノードで構成されます。ラグ制御を有効にするには、`bdr.lag_control_max_commit_delay` と、`bdr.lag_control_max_lag_size` または `bdr.lag_control_max_lag_time` のいずれかをゼロ以外の正の値に設定します。

`bdr.lag_control_max_commit_delay` は、必要に応じてサブミリ秒設定を含む、小数部によるミリ秒の指定を許可および推奨します。

デフォルトでは、`bdr.lag_control_min_conforming_nodes` は 1 に設定されています。完全なリストについては、

[ラグ制御](#) を参照してください



## 概要

ラグ制御は、クライアント接続レベルで動作する動的な TPS レート制限メカニズムです。構成されたラグコントロールの制約を満たしながら、できるだけ目立たないように設計されています。これは、十分な BDR ノードが構成されたラグメジャーしきい値を下回るのに十分な速度で変更をレプリケートできる場合、BDR ランタイムのコミット遅延は 0 ミリ秒に固定されることを意味します。

これに当てはまらない場合、BDR ランタイムのコミット遅延を必要に応じて最小限に調整しますが、準拠ノードの数が最小しきい値に戻るまでそれ以上には調整しません。

最小ノードしきい値に達した後でも、ラグ制御は BDR ランタイムのコミット遅延をゼロに戻そうとし続けます。BDR コミット遅延は、ほとんどの場合、平衡レベルで上下しますが、データスループットまたはラグ適用率が向上すると、コミット遅延は減少します。

BDR 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したり、リソースを取得したりすることを妨げるものではありません。Postgres 拡張機能で管理される外部リソースについては、同じ保証はありません。拡張機能の依存関係に関係なく、BDR 拡張機能が `postgresql.conf` の拡張機能ベースのリソースマネージャーの前にリストされている場合、同じことが保証されます。

厳密に言えば、BDR 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のストリームでのコミット遅延の平均値です。この手法により、コミットの遅延と値のきめ細かな調整で、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を回避できます。また、BDR ランタイムの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解決して、ラグメジャーのしきい値を維持できます。

---

**注釈:** BDR のコミット遅延と Postgres のコミット遅延を混同しないでください。それらは無関係であり、異なる機能を実行します。一方を他方で置き換えしないでください。

---

## 取引申請

BDR 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。これは、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は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、BDR コミット遅延も実行します。これは直感に反するのに見えるかもしれませんが、そのパフォーマンスのおかげで、非同期コミットはレプリケーションラグの最大の原因の 1 つになる可能性があります。

Postgres と BDR の補助プロセスは、トランザクションのコミットで遅延しません。最も注目すべきは、BDR 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。BDR ライターは送信レプリケーションのラグに何も寄与せず、トランザクションのコミットが遅延によって調整されないことにより、受信レプリケーションのラグを最も削減できるため、これは設計です。

## 制限

最大コミット遅延 `bdr.lag_control_max_commit_delay` は、ハード制限を表す上限値です。これは、コミット遅延が構成された値を超えることはないことを意味します。逆に、最大ラグメジャー `bdr.lag_control_max_lag_size` および `bdr.lag_control_max_lag_time` は、超えることができるソフト制限です。最大コミット遅延に達した場合、遅延の継続的な増加を防ぐための遅延対策に追加のバックプレッシャーはありません。

BDR 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では、最大トランザクション遅延を 0 に設定すると便利です。

## 注意事項

アプリケーション TPS は、レプリケーションの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。その他の要因には、BDR コミット遅延の効果が低下する可能性のあるトランザクションの平均サイズが含まれます。特に、バルクロード操作ではレプリケーションの遅延が増加する可能性があり、これにより、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 BDR ランタイムのコミット遅延が増加する可能性があります。それでも最大許容遅延は未満です。

同様に、OLTP 要件が非常に高く、データ変更が少ないアプリケーションは、許容可能な BDR コミット遅延設定によって過度に制限される可能性があります。

このような場合、`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したり、それらのアプリケーションを変更したりすると便利です。たとえば、バルクロード操作は、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、バルク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ために、複数の小さなトランザクションに分割される場合があります。ラグ制御に従って、これらのトランザクションのコミットは、非常に長い BDR コミットの遅延をスケジュールして、以前の部分的なバルクロードが原因で発生したラグを解消することもできます。

## 組織目標の達成

前述の目標リストの例：

- RPO は、適切な最大ラグタイムを設定することで満たすことができます。
- RCO は、適切な最大ラグサイズを設定することで満たすことができます。
- GEO は、BDR ランタイムのコミット遅延と BDR タイムラグの測定値を監視することにより満たすことができます。

前述のように、BDR 実行時の最大コミット遅延が BDR で構成されたコミット遅延制限に固定され、ラグメジャーが BDR で構成された最大レベルを一貫して超えている場合、このシナリオは BDR グループ拡張のマーカーになる可能性があります。



## AutoPartition

AutoPartition を使用すると、自動パーティション管理によってテーブルを簡単に大きなサイズに増やすことができます。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 BDR 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、データ保持期間が終了したときに削除できます。

BDR 管理は、主に SQL で呼び出すことができる関数によって実現されます。BDR のすべての機能は `bdr` スキーマで公開されています。search\_path に入れない限り、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

`bdr.autopartition()` は、テーブルの自動レンジ分割の定義を作成または変更します。定義が存在しない場合は作成されます。そうしないと、後の実行で定義が変更されます。

`bdr.autopartition()` は実際のテーブルをロックしません。新しいパーティションのメンテナンスアクションをいつどのように行うかの定義を変更します。

`bdr.autopartition()` は、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/ドロップできる機能を活用します（基になる Postgres バージョンがサポートしている場合）。

テーブルが RANGE パーティション分割されていない場合、またはマルチカラムパーティションキーが使用されている場合、ERROR が発生します。

`partition_increment` の値の範囲ごとに新しいパーティションが追加され、下限と上限が `partition_increment` 離れています。timestamp または date 型のパーティションキーを持つテーブルの場合、`partition_increment` は interval 型の有効な定数でなければなりません。たとえば、1 Day を指定すると、1 日おきのパーティション境界で新しいパーティションが毎日追加されます。

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 型として指定しなければなりません。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なければなりません。たとえば、10000000 を指定すると、100 万個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 として指定された場合、`partition_initial_lowerbound` は CURRENT DATE に設定されます。`partition_increment` が 1 Hour として指定された場合、`partition_initial_lowerbound` は現在の日付の現在の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、特定の最小数のアドバンスパーティションを作成しようとします。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 で評価できる式にすることができます。たとえば、列型 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 として指定されている場合、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。

この式は、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。

列型 `integer` の分割テーブルの場合、`partition_autocreate_expression` を `SELECT max(partcol) FROM schema.partitioned_table` として指定できます。システムは、パーティション分割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。列型 `integer`、`smallint`、または `bigint` のパーティションテーブルに `partition_autocreate_expression` が指定されていない場合、システムはそれを `max(partcol)` に設定します。

`data_retention_period` が設定されている場合、この期間の後にパーティションが削除されます。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。この値が設定されていない場合、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（および関連する）ベースの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。パーティションは、セカンダリ表領域に移行されるか、上限に対して指定された期間が経過すると削除されます。

デフォルトでは、AutoPartition はパーティションをグローバルに管理します。つまり、1つのノードでパーティションが作成されると、クラスター内の他のすべてのノードにも同じパーティションが作成されます。したがって、すべてのパーティションは一貫しており、使用できることが保証されています。このために、AutoPartition は Raft を利用します。`managed_locally` を `true` として渡すことで、この動作を変更できます。その場合、すべてのパーティションは各ノードでローカルに管理されます。これは、パーティションテーブルがレプリケートテーブルではないため、すべてのノードにすべてのパーティションを持つ必要がない場合、または望ましくない場合に役立ちます。たとえば、組み込みの `bdr.conflict_history` テーブルはレプリケートテーブルではなく、ローカルの AutoPartition によって管理されます。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。

`managed_locally` としてマークされたテーブルをグローバルに管理するように変更することはできません。その逆も同様です。

アクティビティは、エントリが `enabled = on` とマークされている場合にのみ実行されます。

AutoPartition で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する必要はありません。これを行うと、AutoPartition メタデータに一貫性がなくなり、失敗する可能性があります。

## 自動パーティションを構成する

`bdr.autopartition` ファンクションは、テーブルの自動パーティション化を構成します。

### 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,
    managed_locally boolean DEFAULT false,
    enabled boolean DEFAULT on);
```

### パラメーター

- `relation` — テーブルの名前または `Oid`。
- `partition_increment` — 一次のパーティション作成の間隔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かどうかを検出するために使用されます。
- `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つように試みます。
- `maximum_advance_partitions` — アドバンスパーティションの数が `minimum_advance_partitions` を下回ると、一度に作成され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` よりも大きくなければなりません。
- `managed_locally` — `true` の場合、パーティションはローカルで管理されます。
- `enabled` — アクティビティを無効または一時停止し、後で再開または再度有効にできます。

## 例

日単位のパーティション、1 か月間データを保持：

```
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);
```

残りのパーティションが2つだけになったら、5つのアドバンスパーティションを作成します（各パーティションは10億のオーダーを保持できます）。

```
bdr.autopartition(Orders, 10000000000,  
  partition_initial_lowerbound := 0,  
  minimum_advance_partitions := 2,  
  maximum_advance_partitions := 5  
);
```

## 1 つの AutoPartition を作成する

`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。

## パーティションの自動作成の停止

`bdr.drop_autopartition()` を使用して、指定されたリレーシヨンの自動パーティションルールを削除します。リレーシヨンの保留中の作業項目がすべて削除され、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`。

## AutoPartition を 1 つ削除する

この機能は削除する単一のパーティションを指定できるため、BDR AutoPartition テーブルが作成されたら、`bdr.autopartition_drop_partition` を使用します。パーティションテーブルが正常に削除された場合、ファンクションは `true` を返します。

### 概要

```
bdr.autopartition_drop_partition(relname regclass)
```

### パラメーター

- `relname` — 削除するパーティションテーブルの名前。

### ノート

これにより、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 ロックが設定されます。

## パーティションの作成を待ちます

`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が作成されるのを待ちます。この関数は、パーティションテーブル名とパーティションキー列の値を受け取り、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。

この関数は、パーティションがローカルに作成されるのを待機するだけです。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しません。

すべての BDR ノードでパーティションが作成されるまで待機するには、`bdr.autopartition_wait_for_partitions_on_all_nodes()` ファンクションを使用します。この関数は、ローカル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にでも作成されるまで待機します。

## 概要

```
bdr.autopartition_wait_for_partitions(relation regclass, text bound);
```

## パラメーター

- **relation** — テーブルの名前または Oid。
- **bound** — パーティションキー列の値。

## 概要

```
bdr.autopartition_wait_for_partitions_on_all_nodes(relation regclass, text bound);
```

## パラメーター

- **relation** — テーブルの名前または Oid。
- **bound** — パーティションキー列の値。

## パーティションを見つける

**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、関数は NULL を返します。それ以外の場合は、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** — パーティションテーブルの名前。
- **searchkey** — 検索するパーティションキーの値。

## 自動パーティショニングを有効または無効にする

`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。自動パーティショニング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()` を使用して、指定されたテーブルの AutoPartitioning を無効にします。

### 概要

```
bdr.autopartition_enable(relname regclass);
```

### パラメーター

- `relname` — 自動パーティショニングを有効にするリレーシヨンの名前。

### 概要

```
bdr.autopartition_disable(relname regclass);
```

### パラメーター

- `relname` — AutoPartitioning を無効にするリレーシヨンの名前。

### 概要

```
bdr.autopartition_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。

## AutoPartition ワーカーを確認する

`bdr.autopartition_work_queue_check_status` 関数を使用すると、AutoPartitions を維持するために仕事を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。

ワーカーは次のビューで見ることができます。`autopartition_work_queue_local_status`  
`autopartition_work_queue_global_status`

## 概要

```
bdr.autopartition_work_queue_check_status(workid bigint
                                           local boolean DEFAULT false);
```

## パラメーター

- `workid` — AutoPartition ワーカーのキー。
- `local` — ローカルステータスのみを確認します。

## 注意事項

AutoPartition ワーカーは、`bdr.autopartition` 関数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。無効なワーカー ID が使用された場合、関数は `unknown` を返します。In-progress が代表的なステータスです。

## Timestamp-based snapshots

タイムスタンプベースのスナップショットを使用すると、通常の MVCC スナップショットではなく、ユーザー指定のタイムスタンプを使用して一貫した方法でデータを読み取ることができます。これを使用して、共通の時点で異なる BDR ノードのデータにアクセスできます。たとえば、これを使用して、データ品質チェックのために複数のノードのデータを比較できます。

この機能は現在、書き込みトランザクションでは動作しません。

`snapshot_timestamp` パラメーターを使用して、タイムスタンプベースのスナップショットの使用を有効にします。このパラメーターは、タイムスタンプ値または現在のタイムスタンプ（現在）を表す特別な値 `'current'` を受け入れます。`snapshot_timestamp` が設定されている場合、クエリはそのタイムスタンプを使用して、通常の MVCC セマンティクスではなく行の可視性を判断します。

たとえば、次のクエリは 2018-12-08 02:28:30 GMT における `customers` テーブルの状態を返します。

```
SET snapshot_timestamp = 2018-12-08 02:28:30 GMT;
SELECT count(*) FROM customers;
```

BDR がなければ、これは将来のタイムスタンプまたは特別な「現在」値でのみ機能するため、履歴クエリには使用できません。

BDR は、マルチノード環境でその機能を使用して改善します。まず、BDR は、他のノードへのすべての接続が、指定されたタイムスタンプより前にデータベースに追加された未処理のデータを複製することを確認します。これにより、タイムスタンプベースのスナップショットがマルチマスターグループ全体で一貫していることが保証されます。次に、BDR は `bdr.timestamp_snapshot_keep` というパラメーターを追加します。これは、そのノードで最近の履歴に対してクエリを実行できる時間枠を指定します。

任意の間隔を指定できますが、VACUUM（自動バキュームを含む）は指定された間隔の最大 2 倍より新しいデッド行をクリーンアップしないことに注意してください。これは、トランザクション ID が同じ時間解



放されないことも意味します。その結果、これを使用すると、ユーザーテーブルにより多くの膨張が残る可能性があります。最初は、一般的な設定として 10 秒をお勧めしますが、必要に応じて変更できます。

クエリの実行が受け入れられると、クエリは通常と同じように `bdr.timestamp_snapshot_keep` よりも長く実行される場合があります。

また、スナップショットがどの程度保持されているかに関する情報は、サーバーの再起動後も残りません。タイムスタンプベースのスナップショットで使用可能な最も古いタイムスタンプは、PostgreSQL インスタンスの最終再起動の時刻です。

`bdr.timestamp_snapshot_keep` と `postgres_fdw` 拡張機能を組み合わせて、BDR グループ内の複数のノードで一貫した読み取りを取得できます。これを使用して、外部テーブルで使用すると、ノード間で並列クエリを実行できます。

この機能を使用する場合、マルチノードクエリのノード数に制限はありません。

タイムスタンプベースのスナップショットを使用しても、ノード間のトラフィックや帯域幅は増加しません。クエリデータに加えてタイムスタンプ値のみが渡されます。

## Replication sets

レプリケーションセットは、BDR 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 BDR 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デフォルトのレプリケーションセットであり、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ます。すべてのノードがサブスクライブしています。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複製されます。

### レプリケーションセットの使用

`create_replication_set()` を使用して、挿入、更新、削除、または切り捨てのアクションを含めるかどうかを指定して、複製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 つ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。

レプリケーションセットに追加または削除するテーブルを手動で定義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以降も維持されます。

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 BDR 管理ファンクション呼び出しをレプリケートしません。`replication_set_add_ddl_filter` を使用して、複製するコマンドを定義します。

BDR は、すべてのノードに複製セット定義を作成します。次に、`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。

関数を使用して、これらの定義を後で変更したり、複製セットを削除したりできます。

**注釈:** 選択的複製にデフォルトの複製セットを使用しないでください。デフォルトでは DDL レプリケーションと管理ファンクション呼び出しにも使用されるため、クラスター内の BDR ノードでデフォルトのレプリケーションセットを削除または変更しないでください。

## パーティション分割されたテーブルの動作

BDR はパーティションテーブルを透過的にサポートします。つまり、パーティションテーブルをレプリケーションセットに追加できます。いずれかのパーティションを含む変更は、ダウンストリームに複製されます。

**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にレプリケートされます。例外は *TRUNCATE* コマンドで、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で複製されます。

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。その場合、パーティションは通常のテーブルと同様に（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）複製されます。プロバイダーとサブスクライバーの両方でパーティショニング定義が同じである場合、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、これにはパフォーマンス上の利点があります。

**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。

## 外部キーの動作

外部キー制約は、参照元のテーブルの各行が参照先のテーブルの行と一致するようにします。したがって、参照する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。

BDR の現在のバージョンは、この条件を自動的に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場合、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。参照元のテーブルはレプリケーションセットのメンバーですが、参照先のテーブルは次のとおりです。

```
SELECT t1.relname,
       t1.nspname,
       fk.conname,
       t1.set_name
FROM bdr.tables AS t1
JOIN pg_catalog.pg_constraint AS fk
```

(次のページに続く)

(前のページからの続き)

```

    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)

```

これは、テーブル t2 はレプリケーションセット s2 のメンバーですが、外部キー t2\_x\_fkey によって参照されるテーブルはそうではないことを意味します。

TRUNCATE CASCADE コマンドは、コマンドを複製する前に複製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一部であるすべてのテーブルでカスケードのない TRUNCATE になります。

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

## レプリケーションセットの管理

複製セットの管理。

bdr.alter\_node\_replication\_sets を除き、以下の機能を DDL とみなします。DDL レプリケーションとグローバルロックが現在アクティブな場合、それらに適用されます。

*DML および DDL レプリケーション* を参照してください。

### bdr.create\_replication\_set

このファンクションは、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、DDLDDL 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** —新しい複製セットの名前。BDR 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切り捨てをレプリケートするかどうかを示します。
- **autoadd\_tables** —新しく作成された（将来）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** —この複製セットに既存のすべてのユーザーテーブルを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のみ効果があります。

## 注意事項

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 BDR 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。すべての DDL および管理ファンクション呼び出しを複製するデフォルトのグループレプリケーションセット用に、既存の DDL フィルターが設定されています。グループの作成時に作成されますが、

BDR グループのデフォルトのレプリケーションセットで DDL または BDR 管理ファンクションコールをレプリケートしたくない場合は削除できます。

このファンクションは、DDL ステートメントと同じ複製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。

この関数は DDL グローバルロックをとります。

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** で効果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** — 既存の複製セットの名前。
- **replicate\_insert** — このレプリケーションセット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切り捨てをレプリケートするかどうかを示します。
- **autoadd\_tables** — このレプリケーションセットに新しく作成された（将来）テーブルを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すべて以前と同じです。

### 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

この関数は DDL グローバルロックをとります。

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 で効果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

この関数は、既存の複製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` — 既存の複製セットの名前。

### 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

この関数は DDL グローバルロックをとります。

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` で効果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。そのノードでレプリケーションが停止します。その場合は、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。同じ理由から、参加す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削除しないでください。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

この関数は、ノードがパブリッシュ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、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複製する複製セットの配列。配列が空の場合、グループのデフォルトの複製セットが使用されます。

### 注意事項

この関数はローカルノードでのみ実行され、いかなる方法でも複製されません。

この関数はノードが参加する前に実行されるように設計されているため、リストされているレプリケーションセットは存在するかどうかはチェックされません。エラーを回避するために、レプリケーションセット名を正しく指定してください。

これにより、BDR グループに属しているノードだけでなく、まだグループに参加していないノードでも関数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が制限されます。ただし、スキーマは、レプリケーションセットの設定に関係なく、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、後のバージョンの BDR で自動的に削除される可能性があります。これは、

ddl filters

構成が DDL のレプリケーションを妨げる場合でも現在当てはまります。

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、他のノードによってパブリッシュされ、これらのノードからこの関数が実行されるノードに変更を複製します。

## 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したり、テーブルを削除したりできます。これは、これらのテーブルの変更のみのレプリケーション（DML）に影響します。スキーマの変更（DDL）は、DDL レプリケーションセットフィルタによって処理されます（[DDL レプリケーションのフィルタリング](#)を参照）。

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で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、どのノードにレプリケートするアクションを決定します。この決定は、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 と B の両方にサブスクライブしている場合、INSERT と UPDATE の両方のアクションがレプリケートされます。

### 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。

これにより、テーブルがレプリケーションセットに追加され、この瞬間から変更の複製が開始されます（トランザクションのコミット）。テーブルがノードに持つ可能性のあ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** — 複製セットの名前。 NULL（デフォルト）の場合、BDR グループのデフォルトの複製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（現在は何もしません。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される SQL 式。この式が定義されていない（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 を使用して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、行を複製します。False 値は行を複製しません。式にサブクエリを含めたり、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 オーバーヘッドが発生しますが、フィルタリングされた行のデータの送信は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 コマンドを削除しません。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複製できます。 [違いのあるノード間の複製](#) を参照してください。

## 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

`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 で効果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` — 複製セットの名前。 NULL (デフォルト) の場合、BDR グループのデフォルトの複製セットが使用されます。

## 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

この関数は DDL グローバルロックをとります。

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 で効果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。

## レプリケーションセットのリスト

次のクエリを使用して、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;
```

:ref: `外部キーの動作<外部キーの動作>`

では、参照先のテーブルが参照元の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を示します。

次の SQL を使用して、現在のノードがパブリッシュ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,
       COALESCE(
           pub_repsets, pub_repsets
       ) AS pub_repsets,
       COALESCE(
           sub_repsets, sub_repsets
       ) AS 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を実行するには、次のクエリを使用できます。このアプローチは、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($$
      SELECT
        node_id,
        node_name,
        COALESCE(
          pub_repsets, pub_repsets
        ) AS pub_repsets,
        COALESCE(
          sub_repsets, sub_repsets
        ) AS sub_repsets
      FROM bdr.local_node_summary;
    $$)::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、例を示しています。

| 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 レプリケーションのフィルタリング

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 BDR グループレプリケーションセットを介して発生します。これは、BDR グループと同じ名前の DDL フィルターで実現されます。このフィルターは、BDR グループの作成時にデフォルトの BDR グループレプリケーションセットに追加されます。

これを調整するには、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します。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無関係です。データの変更と同様に、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に一致した場合でも、1 回のみレプリケートされます。

次のクエリを使用して、既存の DDL フィルターをリストできます。フィルターごとに、コマンドタグとロール名に適用される正規表現が表示されます。

```
SELECT * FROM bdr.ddl_replication;
```

次の関数を使用して、DDL フィルターを操作できます。これらは DDL と見なされるため、DDL レプリケーションとグローバルロックの対象となります。

### 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、そのセットに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。これは、BDR 管理機能のレプリケーションにも影響します。

これは、ノードでの DDL の実行を妨げるものではありません。DDL が他のノードにレプリケートされるかどうかのみが変更されます。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ーがあるとします。インデックスコマンドは、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、引き続き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、通常の PostgreSQL コマンドの `EVENT TRIGGERs` で使用されるものと同じです。典型的な例は、ロジカルスタンバイで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するフィルターを作成することです。

修飾機能名に一致するタグ名を使用して、使用する BDR 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 BDR 機能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` はどちら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** —複製セットの名前。 NULL の場合、BDR 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 フィルターの名前。これは BDR グループ全体で一意である必要があります
- **command\_tag** —コマンドタグを一致させるための正規表現。 NULL はすべてに一致することを意味します
- **role\_name** —ロール名を一致させるための正規表現。 NULL はすべてのロールに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** —クエリを照合するための正規表現。 NULL はすべてのクエリに一致することを意味します
- **exclusive** —true の場合、他の一致フィルターは考慮されません（つまり、排他フィルターのみが適用されます）。複数の排他フィルターが一致すると、エラーがスローされます。これは、メインの複製セットでデフォルトの複製を維持しながら、特定のコマンドを特定の複製セットにルーティングする場合に役立ちます。

## 注意事項

このファンクションは、DDL ステートメントと同じ複製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 フィルター構成への変更のレプリケーションが既存の ddl フィルター構成の影響を受けることも意味します。

この関数は DDL グローバルロックをとります。

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 で効果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ーを表示するには、ビュー `bdr.ddl_replication` を使用します。

## 例

BDR 管理機能のみを含めるには、次のようなフィルタを定義します。

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, mygroup_admin, $$bdr\..*$$);
```

インデックス DDL 以外をすべて除外するには：

```
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オペレーションを含め、他のすべてのオペレーションを除外するには、2つのフィルターを追加します。1つはテーブル用、もう1つはインデックス用です。これは、複数のフィルターが許可されたすべての DDL コマンドの結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自分自身を含む 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** —複製セットの名前。 NULL の場合、BDR グループのデフォルトの複製セットが使用されます。
- **ddl\_filter\_name** —削除する DDL フィルタの名前。

#### 注意事項

このファンクションは、DDL ステートメントと同じ複製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 フィルター構成への変更のレプリケーションが既存の DDL フィルター構成の影響を受けることも意味します。

この関数は DDL グローバルロックをとります。

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 で効果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。

### Stream triggers

BDR には、ダウンストリーム/ターゲットノードでの追加のデータ処理に使用できる新しいタイプのトリガーが導入されています。

- コンフリクトトリガー
- 変身トリガー

これらの種類のトリガーは、**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ており、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 つのトリガー関数を使用できます。トリガー関数は、次の形式で定義されたプログラムです。トリガーを作成するために、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 BDR 関数 `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。

作成されると、トリガーはカタログ表 `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、名前のサフィックスは `'_bdrc'` または `'_bdrt'` です。ビュー `bdr.triggers` 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、その特定の名前のバリエーションと ALL のバリエーション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、トリガーが通常の SQL トリガーとして実行されなくなります。

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。オリジンノードで実行するオプションはありません。ただし、オリジンでの `row_filter` 式の使用を検討することをお勧めします。

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 BDR 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。これを使用して、たとえば、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、そのノードに固有のクラッシュセーフなテーブルに記録できます。動作例については、[ストリームトリガーの例](#) を参照してください。

### Apply 中に実行をトリガー

変換トリガーは、トリガー テーブル内の受信変更ごとに最初に 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されるため、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 の変更については、ターゲット行を見つけます。ターゲット行がない場合、そ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、以前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た通常の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。

## 行方不明の列の競合解決

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一致させようとします。

入力行に存在するがターゲットテーブルにない列は、タイプ `target_column_missing` の競合をトリガーします。逆に、ターゲットテーブルに存在するが受信行にない列は、`source_column_missing` 競合をトリガーします。これら2つの競合タイプのデフォルトの解決は、それぞれ `ignore_if_null` と `use_default_value` です。

これは、スキーマのローリングアップグレードのコンテキスト、たとえば、スキーマの新しいバージョンで新しい列が導入される場合に関連します。スキーマの古いバージョンから新しいバージョンに複製する場合、ソース列がありません。新しく導入された列にデフォルト値が設定されるため、`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のノードから古いスキーマバージョンのノードに複製すると、ターゲットテーブルに列がありません。`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 以外の値を持つ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` の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。

これを行うには、次のクエリを使用します。これは、ノードごとに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);
```

## データ損失と発散のリスク

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 ではない行が存在する可能性があります。その行はノード 2 に複製され、列 `c` の値は破棄されます（例、`(pk=1)`）。

次に、列 `c` がノード 2 のテーブルに追加されると、それは最初に既存のすべての行で NULL に設定され、上記の行が `(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いため、クラスターは発散しています。

ノード 2 にレプリケートされる行には `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スキーマのローリング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。このプラクティスは、相違を検出して修正するのに役立ちます。



## 行型の用語

これらの行タイプを使用します。

- `SOURCE_OLD` が更新前の行、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、つまり競合している行です。

## 競合トリガー

競合トリガーは、BDR によって競合が検出されたときに実行されます。彼らは、競合が発生したときに何が起るかを決定します。

- トリガー関数が行を返す場合、アクションはターゲットに適用されます。
- トリガー関数が `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、トリガーが `DELETE` に対して呼び出された場合、`DELETE` をスキップする場合、トリガーは `NULL` を返します。`DELETE` を続行したい場合は、行の値を返します。`SOURCE_OLD` または `TARGET` が機能します。競合する操作が `INSERT` または `UPDATE` であり、選択された解決が競合する行の削除である場合、トリガーは削除を明示的に実行して `NULL` を返す必要があります。トリガー関数は、必要に応じて他の 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した場合、2 つ以上の変更が同時に発生しています。これらの変更を適用すると、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、ノード間に大きな時間差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できません。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保証するのは、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。テクニカルサポートは、BDR に付属の `isolationtester` ツールを使用して、すべての競合トリガーを正式にテストすることをお勧めします。

### 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。一貫性のない動作が発生する可能性があり、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、BDR 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、実際の競合はないため、この競合検出メカニズムは誤検出の競合を生成する可能性があります。競合トリガーは、これらすべてを同じように処理する必要があります。

場合によっては、タイムスタンプの競合検出で競合がまったく検出されません。たとえば、`DELETE` が `UPDATE` の直後に発生する同時実行 `UPDATE/DELETE` では、最初に `UPDATE` を認識し、次に `DELETE` を認識しません。

競合が見られない場合、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、競合が発生し、競合トリガーで処理できます。

トリガー関数は、操作の種類に応じて、競合に関係するデータ行だけでなく、追加の状態情報にアクセスします。

- INSERT では、競合トリガーはソースから SOURCE\_NEW 行および TARGET 行にアクセスできます。
- UPDATE では、競合トリガーはソースおよび TARGET 行から SOURCE\_OLD および SOURCE\_NEW 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 行および TARGET 行にアクセスできます。

`bdr.trigger_get_row()` ファンクションを使用して、SOURCE\_OLD、SOURCE\_NEW、または TARGET 行を取得できます（その操作の値が存在する場合）。

競合トリガーへの変更はトランザクションで発生し、ALTER TABLE の一部のバリエーションが処理される方法と同様に、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。

コンフリクトトリガー内で主キーを更新すると、実行タイミングの違いによりユニーク制約違反エラーにつながる場合があります。したがって、競合トリガーで主キーを更新しないでください。

### トリガーを変換する

これらのトリガーは、特定のテーブルに対してデータストリームのすべての行に対して実行されることを除き、競合トリガーに似ています。戻り値と公開された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。

BDR の各テーブルに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処理し、必要に応じて追加の操作を実行できます。任意の列の値を変更したり、それらを NULL に設定したりできます。戻り値によって、実行されるアクションが決まります。

- トリガー関数が行を返す場合、ターゲットに適用されます。
- トリガー関数が NULL 行を返した場合、これ以上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、他のアクションを自由に実行できます。

トリガー関数は、競合に関係する行だけでなく、追加の状態情報にもアクセスできます。

- 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。

関数 `bdr.trigger_get_row()` を使用して、SOURCE\_OLD または SOURCE\_NEW の行を取得できます。このタイプのトリガーは、そのようなターゲット行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 行トリガーと非常によく似ていますが、次の重要な違いがあります。

- 変換トリガーは、着信変更ごとに呼び出されます。BEFORE トリガーは UPDATE に対してまったく呼び出されません。テーブル内に一致する行が見つからない場合、DELETE が変更されます。
- パーティションテーブルのルーティングが発生する前に、変換トリガーが呼び出されます。
- 変換トリガーは、通常の SQL トリガーでは使用できない SOURCE\_OLD を介してルックアップキーにアクセスします。

## ストリームトリガー変数

競合トリガーと変換トリガーはどちらも、トリガー API が提供する事前定義された変数と BDR が提供する追加情報関数を介して、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、以下の事前定義された変数を使用できます。

### TG\_NAME

データ型名。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」のサフィックス（トリガーのタイプによって異なります）が付いています。

### TG\_WHEN

データ型テキスト。この変数は、競合と変換の両方のトリガーに対して BEFORE を示します。 `bdr.trigger_get_type()` 情報関数を呼び出すことで、ストリームの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

データ型テキスト: トリガーが起動された操作を識別する INSERT、UPDATE、または DELETE の文字列。

### TG\_RELID

データ型 oid: トリガーの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の原因となったテーブルの名前。

## TG\_TABLE\_SCHEMA

データ型名: トリガーの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()` ステートメントでトリガー関数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()` ステートメントからの引数。インデックスは 0 からカウントされます。無効なインデックス (0 未満または TG\_NARGS 以上) は NULL 値になります。

## 情報機能

### `bdr.trigger_get_row`

この関数は、識別子で指定されたトリガー行の内容を `RECORD` として返します。この関数は、不適切に呼び出された場合、つまり、オペレーションタイプ (`TG_OP`) が `DELETE` の場合に `SOURCE_NEW` で呼び出された場合、`NULL` を返します。

## 概要

```
bdr.trigger_get_row(row_id text)
```

## パラメーター

- `row_id` 一行の識別子。トリガーの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)。

### 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・タイムスタンプを返します。行がフリーズしているか利用できないために利用できない場合、NULL を返します。行識別子 SOURCE\_OLD に対しては常に NULL を返します。

#### 概要

```
bdr.trigger_get_committs(row_id text)
```

#### パラメーター

- **row\_id**—行の識別子。トリガーのタイプと操作に応じて、SOURCE\_NEW、SOURCE\_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

### 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 を返します。行がフリーズしているか利用できないために利用できない場合、NULL を返します。SOURCE\_OLD および SOURCE\_NEW 行識別子に対して常に NULL を返します。

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。トリガーのタイプと操作に応じて、SOURCE\_NEW、SOURCE\_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

### bdr.trigger\_get\_type

この関数は、現在のトリガータイプを返します。これは、CONFLICT または TRANSFORM です。ストリームトリガーの外部で呼び出された場合は 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返します。それ以外の場合は、NULL を返します。

この関数の可能な戻り値については、[競合タイプのリスト](#) を参照してください。

## 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 のオリジンに対応するノード ID を返します。オリジンが有効でない場合（つまり、行がローカルで発生した場合）、トリガーの行引数に応じて、ソースまたはターゲットノードのノード ID を返します。行識別子 SOURCE\_OLD に対しては常に NULL を返します。この関数を使用して、常に信頼できるソースノードを優先するように競合トリガーを定義できます。

## 概要

```
bdr.trigger_get_origin_node_id(row_id text)
```

## パラメーター

- row\_id 一行の識別子。トリガーのタイプと操作に応じて、SOURCE\_NEW、SOURCE\_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

### **bdr.ri\_fkey\_on\_del\_trigger**

BEFORE トリガーとして呼び出されると、このファンクションは FOREIGN KEY 情報を使用して FK の異常を回避します。

## 概要

```
bdr.ri_fkey_on_del_trigger()
```

## 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 REPLICATION IDENTITY 設定、およびターゲットテーブルの内容に依存します。

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に UPDATE または DELETE を適用した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。

## トリガーノート

トリガーの実行順序：

- トリガーを変換する - ターゲットの受信行ごとに 1 回実行します。
- 通常のトリガー 1 行に 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操作インターフェース

REPLICATION IDENTITY FULL を持つテーブル、または TOAST が適用される列のないテーブルでのみストリームトリガーを作成できます。

## 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** —このトリガーを起動するリレーション。
- **function** —実行する関数。
- **args** —オプション。トリガー関数が実行時に受け取るパラメーターの配列を指定します（TG\_ARGV 変数の内容）。

## 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

このファンクションは、トリガーが作成され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 で効果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、ファンクションの **relation** および EXECUTE 権限が必要です。これは、30619 以上の `bdr.backwards_compatibility` に適用されます。追加のセキュリティルールは、競合トリガーを含むすべてのトリガーに BDR で適用されます。

*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,  
                             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** —このトリガーを起動するリレーション。
- **function** —実行する関数。
- **args** —オプション。トリガー関数が実行時に受け取るパラメーターの配列を指定します（TG\_ARGV 変数の内容）。

## 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

このファンクションは、トリガーが作成され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 で効果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、ファンクションの **relation** および **EXECUTE** 権限が必要です。追加のセキュリティルールは、BDR で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**

この関数は、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` に設定すると、この関数は欠落しているトリガーを無視します。

## 注意事項

このファンクションは、DDL ステートメントと同じ複製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。

このファンクションは、トリガーが作成され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` で効果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。

## 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;
```

(次のページに続く)

(前のページからの続き)

```

    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 timestampz;
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;
$$;

```

この例は、信頼できるソースの競合検出 (信頼できるサイト、優先ノード、または Always Wins 解決とも呼ばれます) を実装する競合トリガーを示しています。これは `bdr.trigger_get_origin_node_id()` 関数を使用して、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;
$$;

```

### Explicit two-phase commit (2PC)

アプリケーションは、BDR 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 Distributed Transaction Processing (DTP) モデルは、次の 3 つのソフトウェアコンポーネントを想定しています。

- トランザクションの境界を定義し、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リソースマネージャー (データベースやファイルアクセスシス

テムなどの 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回復に責任を持つトランザクションマネージャー (TM) と呼ばれる別個のコンポーネント

BDR は、`PREPARE TRANSACTION` および `COMMIT PREPARED / ROLLBACK PREPARED` コマンドを使用した明示的な外部 2PC をサポートしています。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単一のリソースマネージャーに見えます。

`bdr.commit_scope` が `local` の場合、ローカルノードのみでトランザクションを準備します。コミットされると、変更がレプリケートされ、BDR はコミット後の競合解決を適用します。

`bdr.commit_scope` を `local` に設定して使用することは、明示的な 2 フェーズコミットでは意味がないように見えるかもしれませんが、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するオプションが提供されています。

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を持つ 2 フェーズコミットは、標準の PostgreSQL とまったく同じように機能します。ローカルコミットスコープを使用し、CAMO を無効にします。

```
BEGIN;

SET LOCAL bdr.enable_camo = off;
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すると、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 2 番目のフェーズを呼び出す必要があります。

```
COMMIT PREPARED some-global-id;
```

**Catalogs and views** ここでは、カタログとビューをアルファベット順に示しています。

ユーザーに表示されるカタログとビュー

#### **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、自動パーティション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、行レベルのセキュリティで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るテーブルの所有者が可能です。

詳細については、[テーブルへの競合のログ](#) を参照してください。

#### **bdr.conflict\_history** 列

| Name                | Type                     | Description                                             |
|---------------------|--------------------------|---------------------------------------------------------|
| sub_id              | oid                      | この競合を引き起こしたサブスクリプション。<br>`bdr.subscription` テーブルに結合できます |
| 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_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 LSN をコミットします         |
| 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                     | 検出された競合の種類                                              |
| conflict_resolution | text                     | 選択された競合の解決                                              |
| 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）                      |

次のページに続く

表 22 – 前のページからの続き

| Name                    | Type                     | Description                                                             |
|-------------------------|--------------------------|-------------------------------------------------------------------------|
| relid                   | oid                      | 競合するリレーション（インデックスが削除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                     | 着信する競合する行の Json 表現                                                      |
| local_tuple             | json                     | ローカルの競合する行の Json 表現                                                     |
| apply_tuple             | json                     | 結果の（適用された）行の Json 表現                                                    |
| local_tuple_xmin        | xid                      | ローカルで競合する行を生成したトランザクション（ <i>local_tuple</i> が設定されており、行がフリーズされていない場合）    |
| local_tuple_node_id     | oid                      | ローカルの競合する行を生成したノード（ <i>local_tuple</i> が設定されており、行がフリーズされていない場合）         |
| local_tuple_commit_time | timestamp with time zone | ローカルで競合している行の最新の変更のタイムスタンプ（ <i>local_tuple</i> が設定されており、行がフリーズされていない場合） |

**bdr.conflict\_history\_summary**

行の競合に関するユーザーが読める詳細を含むビュー。

**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     |
| 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()` インターフェースによって使用される Raft ベースの内部 KV ストア用の永続ストレージ。

**bdr.consensus\_kv\_data** 列

| Name         | Type       | Description                                    |
|--------------|------------|------------------------------------------------|
| kv_key       | text       | ユニークキー                                         |
| kv_val       | json       | json 形式の任意値                                    |
| kv_create_ts | timestampz | 最終書き込みタイムスタンプ                                  |
| kv_ttl       | int        | ミリ秒単位の値の存続時間                                   |
| kv_expire_ts | timestampz | 有効期限のタイムスタンプ<br>( $kv\_create\_ts + kv\_ttl$ ) |

**bdr.camo\_decision\_journal**

`bdr.logical_transaction_status` が呼び出された場合に、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的なジャーナル。`bdr.node_pre_commit` とは異なり、これは通常の操作条件（つまり、CAMO ペアの両方のノードが実行されて接続されている）で処理される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」、中止の場合は「a」   |
| decision_ts    | timestampz | デシジョンタイム                |

**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」) 関数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機能 |

**bdr.ddl\_replication**

このビューには、現在の DDL filters によって設定された 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 フィルターの名前            |
| set_ddl_tag  | text | 適用されるコマンドタグ (正規表現)      |
| set_ddl_role | text | 適用されるロール (正規表現)         |
| set_name     | name | このフィルターが定義されている複製セットの名前 |

**bdr.depend**

このテーブルは、BDR カタログ内の内部オブジェクト依存関係を追跡します。

**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の管理中に送信されたすべての Raft メッセージをログに記録します。

**bdr.global\_consensus\_response\_journal** カタログの場合、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、`:ref:`bdr.global_consensus_journal_details`<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  | 仕訳の ID            |
| term          | int8  | ラフトターム            |
| origin        | oid   | リクエストが発生したノードの ID |
| req_id        | int8  | リクエストの ID         |
| req_payload   | bytea | リクエストのペイロード       |
| 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()** 関数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    |
|-------------|-------|----------------|
| log_index   | int8  | 仕訳の ID         |
| term        | int8  | ラフトターム         |
| 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 | リクエストのペイロード    |

次のページに続く

表 29 – 前のページからの続き

| Name                      | Type | Description             |
|---------------------------|------|-------------------------|
| origin_node_name          | name | 要求が発生したノードの名前           |
| message_type_no           | oid  | リクエストの BDR メッセージタイプの ID |
| message_type              | text | 要求の BDR メッセージタイプの名前     |
| message_payload           | text | リクエストの BDR メッセージペイロード   |
| re-sponse_message_type_no | oid  | 応答の BDR メッセージタイプの ID    |
| re-sponse_message_type    | text | 応答の BDR メッセージタイプの名前     |
| response_payload          | text | レスポンスの BDR メッセージペイロード   |
| response_errcode_no       | text | 応答の SQLSTATE            |
| response_errcode          | text | レスポンスのエラーコード            |
| response_message          | text | レスポンス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の管理中に受信した Raft メッセージへのすべての応答を収集します。

**bdr.global\_consensus\_journal** カタログの場合、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、`:ref:`bdr.global_consensus_journal_details`<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  | 仕訳の ID         |
| res_status    | oid   | レスポンスのステータスコード |
| res_payload   | bytea | レスポンスのペイロード    |
| 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、サーバーの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格納されます。

**bdr.global\_lock** の可視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するとは限らないため、使用状況を監視するには、

:ref:`bdr.global\_locks` <*bdr.global\_locks*>` ビューをお勧めします。

このテーブルの内容を変更しないでください。重要な BDR 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     | ロックの種類 (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 (勧告ロック用) |
| key2           | integer | 2 番目の 32 ビットキーまたは 64 ビットキーの 32 ビット (勧告ロック用) |
| key_is_bigint  | boolean | 64 ビット整数キーが使用されている場合は true (勧告ロック用)         |

**bdr.global\_locks**

このノードでアクティブなグローバル ロックを含むビュー。

:ref:`bdr.global\_locks` <bdr.global\_locks>` ビューは、BDR の共有メモリロック状態の追跡を公開するため、管理者は BDR のグローバルロックアクティビティと進行状況をより深く理解できます。

グローバルロックの詳細については、[グローバルロックの監視](#) を参照してください。

**bdr.global\_locks 列**

| Name                            | Type       | Description                              |
|---------------------------------|------------|------------------------------------------|
| <i>origin_node_id</i>           | oid        | グローバルロックが発生したノードの OID                    |
| <i>origin_node_name</i>         | name       | グローバルロックが発生したノードの名前                      |
| <i>lock_type</i>                | text       | ロックの種類（DDL または DML）                      |
| <i>relation</i>                 | 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（勧告ロックの場合） |
| <i>pid</i>                      | int4       | ロックを保持しているプロセスの PID                      |
| <i>acquire_stage</i>            | text       | ロック取得プロセスの内部状態                           |
| <i>waiters</i>                  | int4       | 同じグローバルロックを待っているバックエンドのリスト               |
| <i>global_lock_request_time</i> | timestampz | 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刻         |
| <i>local_lock_request_time</i>  | timestampz | ローカルノードがローカルロックの取得を開始した時刻                |
| <i>last_state_change_time</i>   | timestampz | `acquire_stage` の最終変更時刻                  |

列の詳細：

- **relation**：DML ロックの場合、**relation** は、DML ロックを取得した関係を示します。グローバル勧告ロックの場合、**relation** 列には、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 ビット整数が表示されます。

- **origin\_node\_id** および **origin\_node\_name**：これらがローカルノードの ID と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、取得する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

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 ロック要求に応じようとしています。

- **pid** : ローカルノードが要求側ノードの場合、グローバル DDL ロックを要求したプロセスのプロセス ID。他のノードではヌル。オリジンノードを照会して、ロッカーの pid を特定します。
- **global\_lock\_request\_time** : グローバルロック要求のイニシエーターがグローバルロックの取得の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 になります。この時間は DDL ロック要求の先頭にスタンプされ、DDL エポック管理と必要なレプリケーション保留キューのフラッシュにかかっ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の取得を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かかっ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った時間は含まれません。自ノード再起動後にロックを再取得すると、ノード再起動時刻になります。
- **last\_state\_change\_time** : このグローバルロックエントリの `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.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  | 仕訳の ID          |
| log_term  | int8  | ラフトターム          |
| snapshot  | bytea | いかなる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は、ローカルノード上の Raft の現在の状態を保存します。

#### **bdr.local\_consensus\_state** 列

| Name         | Type | Description   |
|--------------|------|---------------|
| node_id      | oid  | ノードの ID       |
| current_term | int8 | ラフトターム        |
| apply_index  | int8 | ラフトアプライインデックス |

次のページに続く

表 34 – 前のページからの続き

| Name              | Type | Description               |
|-------------------|------|---------------------------|
| voted_for         | oid  | 今期にこのノードからの投票             |
| last_known_leader | oid  | 最後にわかっているラフトリーダーの 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[] | パブリッシュされた複製セット  |
| sub_repsets | text[] | サブスクライブされた複製セット |

**bdr.local\_node\_summary**

:ref:`bdr.node\_summary` <*bdr.node\_summary*>` と同じ情報を含むがローカルノードのみのビュー。

**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 | 実行される同期の種類             |
| sync_subid   | oid  | 同期を行うサブスクリプションの ID     |
| sync_nspname | name | 同期されたテーブルのスキーマ名 (ある場合) |
| sync_relname | name | 同期されたテーブルの名前 (ある場合)    |
| sync_status  | char | 同期の現在の状態               |

次のページに続く

表 36 – 前のページからの続き

| Name              | Type   | Description                    |
|-------------------|--------|--------------------------------|
| sync_remote_relid | oid    | アップストリームに同期されたテーブル (ある場合) の ID |
| sync_end_lsn      | pg_lsn | 同期状態が最後に変化した位置                 |

**bdr.network\_path\_info**

ノードの場所間のネットワークコストに関するユーザー定義の情報を保存するカタログビュー。

**bdr.network\_path\_info** 列

| Name            | Type    | Description                         |
|-----------------|---------|-------------------------------------|
| node_group_name | name    | BDR グループの名前                         |
| node_region1    | text    | ノードリージョン名、<br>bdr.node_location から  |
| node_region2    | text    | ノードリージョン名、<br>bdr.node_location から  |
| node_location1  | text    | ノードロケーション名、<br>bdr.node_location から |
| node_location2  | text    | ノードロケーション名、<br>bdr.node_location から |
| network_cost    | numeric | ノードロケーション名、<br>bdr.node_location から |

**bdr.node**

このテーブルには、クラスタ内のすべての BDR ノードがリストされています。

**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" | 参加中に行われるスキーマの同期 |

次のページに続く



表 38 – 前のページからの続き

| Name                 | Type  | Description                         |
|----------------------|-------|-------------------------------------|
| 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）          |
| 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[] | ノードでサポートされているプロトコルバージョンの範囲          |

**bdr.node\_catchup\_info**

このカタログテーブルは、結合または部分的なプロシージャに関連する場合、各ノードに関連するキャッチアップ情報を記録します。

**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）が別れノード（origin\_node\_id）から欠落しているデータを必要とする場合、転送することにより、すでにそれを持っているノード（node\_source\_id）から取得できます。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\_conflict\_resolvers**

既知のすべての競合タイプに対して現在構成されている競合解決。

**bdr.node\_conflict\_resolvers** 列

| Name              | Type | Description        |
|-------------------|------|--------------------|
| conflict_type     | text | 紛争の種類              |
| 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 BDR ノードグループがリストされています。

**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のレプリケーションセット            |
| node_group_default_repset_name | text     | このノードグループのデフォルトの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サブスクライバーが待機する時間     |
| node_group_check_constraints   | bool     | データの適用時に適用プロセスが制約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クアップするサブスクリプションに使用するライターの数 |

[次のページに続く](#)

表 41 – 前のページからの続き

| Name                          | Type    | Description                                                |
|-------------------------------|---------|------------------------------------------------------------|
| node_group_enable_wal_decoder | boolean | グループに enable_wal_decoder が設定されているかどうか                      |
| node_group_streaming_min      | integer | 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 |

**bdr.node\_group\_replication\_sets**

BDR 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図。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   | BDR 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 によって複製されるアクション                             |
| def_repset_ext     | name   | デフォルトの「外部」repset の名前 (通常は def_repset と同じ)                |
| def_repset_ext_ops | text[] | デフォルトの「外部」repset によって複製されるアクション (通常は def_repset_ops と同じ) |

**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とは対照的に）。

**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) |

**bdr.node\_location**

ノードの場所に関するユーザー定義の情報を保存するカタログビュー。

**bdr.node\_location** 列

| Name            | Type | Description    |
|-----------------|------|----------------|
| node_group_name | name | BDR グループの名前    |
| node_id         | oid  | ノードの ID        |
| node_region     | text | ユーザー指定のリージョン名  |
| node_location   | text | ユーザー指定のロケーション名 |

**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か |
| log_conflict_type | ログに記録する競合の種類 (NULL がすべてを意味する場合)   |
| log_conflict_res  | NULL がすべてを意味する場合、ログに記録する競合解決      |

**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、**bdr.replay\_progress\_frequency** ミリ秒ごとに他のすべてのノードに進行状況を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)。このリレーションには  $N * (N-1)$  行が必要です。

モニタリングのために **bdr.node\_slots** ビューにもっと興味があるかもしれません。

*Monitoring* 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発信ノードの 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   | timestampz | 送信元ノードによってレポートが送信された時刻                             |
| last_update_recv_time   | timestampz | 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     | レポート時の送信元ノードの LSN                                  |
| peer_position           | pg_lsn     | 発信元ノードから見たノードのピアの最新の LSN                           |
| peer_replay_time        | timestampz | レポートノードによって見られるピアの最新のリプレイ時間                        |
| last_update_horizon_xid | oid        | 内部解像度ホライズン: より低い xid はすべてレポートノードで解決されていることがわかっています |

次のページに続く

表 46 – 前のページからの続き

| Name                    | Type   | Description              |
|-------------------------|--------|--------------------------|
| last_update_horizon_lsn | pg_lsn | 内部解決期間：報告ノードの LSN に関して同じ |

**bdr.node\_pre\_commit**

Commit At Most Once (CAMO) パートナーとして構成されたノードで内部的に使用されます。過去 15 分間に CAMO パートナーがトランザクションで行った決定を示しています。

**bdr.node\_pre\_commit** 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」、中止の場合は「a」   |
| local_xid      | xid        | ローカルノードのトランザクション ID     |
| commit_ts      | timestampz | トランザクションのコミットタイムスタンプ    |
| decision_ts    | timestampz | デシジョンタイム                |

**bdr.node\_replication\_rates**

このビューには、特定のノードからの発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 | リプレイとして報告された最新の位置（表示）          |

次のページに続く

表 48 – 前のページからの続き

| Column           | Type     | Description                              |
|------------------|----------|------------------------------------------|
| 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     | replay_lsn とオリジン上の現在の WAL 書き込み位置のバイト差    |
| replay_lag_size  | text     | replay_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い  |
| apply_rate       | bigint   | ピアノードで 1 秒あたりに適用されている LSN                |
| 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を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`` を使用してください。

### bdr.node\_slots

このビューには、BDR 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。

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、[送信レプリケーションのモニタリング](#) を参照してください。

### bdr.node\_slots 列

| Name            | Type | Description            |
|-----------------|------|------------------------|
| target_dbname   | name | ターゲットノードのデータベース名       |
| node_group_name | name | BDR グループの名前            |
| node_group_id   | oid  | BDR グループの OID          |
| origin_name     | name | 起点ノードの名前               |
| target_name     | name | ターゲットノードの名前            |
| origin_id       | oid  | オリジンノードの OID           |
| target_id       | oid  | ターゲットノードの OID          |
| local_slot_name | name | BDR によるレプリケーションスロットの名前 |

次のページに続く

表 49 – 前のページからの続き

| Name                | Type    | Description                                                                                               |
|---------------------|---------|-----------------------------------------------------------------------------------------------------------|
| slot_name           | name    | Postgres によるスロットの名前 (同上)                                                                                  |
| is_group_slot       | boolean | スロットがこのノードのノードグループ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) |
| is_decoder_slot     | boolean | このスロットは Decoding Worker によって使用されますか                                                                       |
| plugin              | name    | このスロットを使用した論理デコードプラグイン (pg-logical_output または bdr でなければなりません)                                             |
| 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                                                                                            |
| active              | bool    | スロットはアクティブですか (接続はありますか)                                                                                  |
| active_pid          | int4    | スロットに接続されているプロセスの 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再開できる LSN                                                                                       |
| confirmed_flush_lsn | pg_lsn  | 最新の確認された複製位置                                                                                              |
| usesysid            | oid     | レプリケーションセッションを実行しているユーザーの sysid                                                                           |
| username            | name    | レプリケーションセッションを実行し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`で 使用)                                                       |
| client_addr         | inet    | クライアント接続の IP アドレス                                                                                         |

次のページに続く



表 49 – 前のページからの続き

| Name             | Type       | Description                                      |
|------------------|------------|--------------------------------------------------|
| 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    | timestampz | 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     | 書かれたとおりに報告された最新の位置                               |
| flush_lsn        | pg_lsn     | ディスクにフラッシュされたと報告された最新の位置                         |
| replay_lsn       | pg_lsn     | リプレイとして報告された最新の位置 (表示)                           |
| 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       | sent_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       | flush_lsn と現在の WAL 書き込み位置のバイト差                   |
| replay_lag_bytes | int8       | replay_lsn と現在の WAL 書き込み位置のバイト差                  |
| sent_lag_size    | text       | sent_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い            |
| write_lag_size   | text       | write_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い           |
| flush_lag_size   | text       | flush_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い           |
| replay_lag_size  | text       | replay_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い          |

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、ローカルノードが認識しているすべての BDR 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                         |
|-------------------------------------|-------------------|-------------------------------------|
| <code>node_name</code>              | <code>name</code> | ノードの名前                              |
| <code>node_group_name</code>        | <code>name</code> | ノードが属する BDR グループの名前                 |
| <code>interface_connstr</code>      | <code>text</code> | ノードへの接続文字列                          |
| <code>peer_state_name</code>        | <code>text</code> | 人間が読める形式のノードの一貫した状態                 |
| <code>peer_target_state_name</code> | <code>text</code> | ノードが到達しようとしている状態（参加または昇格中）          |
| <code>node_seq_id</code>            | <code>int4</code> | 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 |
| <code>node_local_dbname</code>      | <code>name</code> | ノードのデータベース名                         |
| <code>set_repl_ops</code>           | <code>text</code> | デフォルトの複製セットはどの操作を複製しますか             |
| <code>node_id</code>                | <code>oid</code>  | ノードの OID                            |
| <code>node_group_id</code>          | <code>oid</code>  | BDR ノードグループの OID                    |

### `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              |
|-------------------------------|-------------------------|--------------------------|
| <code>queued_at</code>        | <code>timestampz</code> | ステートメントがキューに入れられたのはいつですか |
| <code>role</code>             | <code>name</code>       | ステートメントを実行したロール          |
| <code>replication_sets</code> | <code>text[]</code>     | ステートメントが発行されたレプリケーションセット |

次のページに続く

表 51 – 前のページからの続き

| Name         | Type | Description                                                                                                 |
|--------------|------|-------------------------------------------------------------------------------------------------------------|
| message_type | char | メッセージのタイプ。指定可能な値：<br />A - Table sync<br />D - DDL<br />S - Sequence<br />T - Truncate<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パayload                                                                           |

**bdr.replication\_set**

レプリケーションセットの構成を格納するテーブル。ユーザークエリの場合、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    | 複製セットの名前                                 |
| 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 | 複製セットが TRUNCATE を複製するかどうかを示します           |
| set_isinternal     | boolean | 予約済み                                     |
| set_autoadd_tables | boolean | このレプリケーションセットに新しいテーブルが自動的に追加されるかどうかを示します |

次のページに続く

表 52 – 前のページからの続き

| Name             | Type    | Description                         |
|------------------|---------|-------------------------------------|
| set_autoadd_seqs | boolean | この複製セットに新しいシーケンスが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メンバーシップを格納するテーブル。ユーザークエリの場合、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格納するテーブル。ユーザークエリの場合、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 フィルターの名前     |
| set_ddl_tag  | text | DDL フィルターのコマンドタグ |
| set_ddl_role | text | DDL を実行するロール     |

**bdr.replication\_sets**

BDR 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（現在どのノードでも使用されていません）。

**bdr.replication\_sets** 列

| Name               | Type    | Description                                          |
|--------------------|---------|------------------------------------------------------|
| set_id             | oid     | レプリケーションセットの<br>OID                                  |
| set_name           | name    | 複製セットの名前                                             |
| replicate_insert   | boolean | レプリケーションセットが<br>INSERT をレプリケートする<br>かどうかを示します        |
| replicate_update   | boolean | レプリケーションセットが<br>UPDATE をレプリケートする<br>かどうかを示します        |
| replicate_delete   | boolean | レプリケーションセットが<br>DELETE をレプリケートする<br>かどうかを示します        |
| replicate_truncate | boolean | 複製セットが TRUNCATE<br>を複製するかどうかを示し<br>ます                |
| set_autoadd_tables | boolean | このレプリケーションセッ<br>トに新しいテーブルが自動<br>的に追加されるかどうかを<br>示します |
| set_autoadd_seqs   | boolean | この複製セットに新しい<br>シーケンスが自動的に追加<br>されるかどうかを示します          |

**bdr.schema\_changes**

スキーマに対するすべての変更を表示する単純なビューが BDR に勝ちます。

**bdr.schema\_changes** 列

| Name                    | Type       | Description    |
|-------------------------|------------|----------------|
| schema_changes_ts       | timestampz | 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     | その値内のチャンクのシーケンス番号 |
| seq_allocated_up_to | bigint     | 該当なし              |
| seq_nallocs         | bigint     | 該当なし              |
| seq_last_alloc      | timestampz | 最後に割り当てられたシーケンス   |

**bdr.schema\_changes**

BDR のスキーマに対するすべての変更を表示する単純なビュー。

**bdr.schema\_changes** 列

| Name                    | Type        | Description    |
|-------------------------|-------------|----------------|
| schema_changes_ts       | timestamptz | 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**

割り当てられたシーケンスを表示するビュー。

**bdr.sequence\_alloc** 列

| Name                | Type        | Description       |
|---------------------|-------------|-------------------|
| seqid               | regclass    | シーケンスの ID         |
| seq_chunk_size      | bigint      | その値内のチャンクのシーケンス番号 |
| seq_allocated_up_to | bigint      | 該当なし              |
| seq_nallocs         | bigint      | 該当なし              |
| 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 BDR ブックキーピングのシーケンスを除く、すべてのシーケンスがその種類とともにリストされます。

**bdr.sequences** 列

| Name    | Type | Description                              |
|---------|------|------------------------------------------|
| nspname | name | シーケンスを含む名前空間                             |
| relname | name | 配列の名前                                    |
| 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ますが、待機が BDR に関連する場合に `wait_event` が正しく設定されます。

**bdr.stat\_relation**

各関係に統計を適用します。追跡が有効になっており、特定のリレーションに対して何かがレプリケートされた場合にのみデータが含まれます。

**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かかった合計時間 |
| ninsert             | bigint           | リレーションで複製された挿入の数            |
| nupdate             | bigint           | リレーションのために複製された更新の数         |
| ndelete             | bigint           | リレーションでレプリケートされた削除の数        |
| ntruncate           | bigint           | リレーションで複製された切り捨ての数          |
| 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           | リレーションのために読み取られた共有ブロックの合計数  |
| shared_blks_dirtied | bigint           | リレーションのためにダーティ化された共有ブロックの総数 |
| shared_blks_written | bigint           | リレーションに書き込まれた共有ブロックの総数      |

次のページに続く



表 61 – 前のページからの続き

| Column            | Type             | Description                                                                      |
|-------------------|------------------|----------------------------------------------------------------------------------|
| blk_read_time     | double precision | リレーシヨンのブロックの読み取りに費やされた合計時間（ミリ秒）（`track_io_timing`が有効になっている場合、それ以外の場合はゼロ）         |
| blk_write_time    | double precision | リレーシヨンのブロックの書き込みにかかった合計時間（ミリ秒）（`track_io_timing`が有効になっている場合、それ以外の場合はゼロ）          |
| lock_acquire_time | double precision | リレーシヨンのロックの取得にかかった合計時間（ミリ秒）（`bdr.track_apply_lock_timing`が有効になっている場合、それ以外の場合はゼロ） |

**bdr.stat\_subscription**

各サブスクリプションに統計を適用します。トラッキングが有効になっている場合にのみデータが含まれます。

**bdr.stat\_subscription** 列

| Column   | Type   | Description                  |
|----------|--------|------------------------------|
| sub_name | name   | サブスクリプションの名前                 |
| subid    | oid    | サブスクリプションの OID               |
| 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の数        |
| nabort   | bigint | このサブスクリプションに対してライタが行ったアボートの数 |
| 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   |

次のページに続く

表 62 – 前のページからの続き

| Column              | Type   | Description                                                                       |
|---------------------|--------|-----------------------------------------------------------------------------------|
| 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 ( <code>skip_transaction`競合リゾルバー</code> による) |
| 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の数                                                               |
| 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 | デッドロックが原因のエラーの数                                                                   |
| nretries            | bigint | ライタが行った再試行の数 (完全な再起動/再接続なし)                                                       |
| 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 | 中断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 <code>STREAM START</code> メッセージの数                                           |
| nstream_stop        | bigint | 処理された <code>STREAM STOP</code> メッセージの数                                            |
| 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の合計ヒット数                                                    |
| 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ー化された共有ブロックの合計数                                                  |

次のページに続く

表 62 – 前のページからの続き

| Column               | Type                     | Description                                                                               |
|----------------------|--------------------------|-------------------------------------------------------------------------------------------|
| 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                                                             |
| 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）（ <code>track_io_timing</code> が有効になっている場合、それ以外の場合はゼロ） |
| blk_write_time       | double precision         | ミリ秒単位のブロックの書き込みにサブスクリプションが費やした合計時間（ <code>track_io_timing</code> が有効になっている場合、それ以外の場合はゼロ）  |
| connect_time         | timestamp with time zone | 現在のアップストリーム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間                                                                   |
| start_lsn            | pg_lsn                   | このサブスクリプションがアップストリームからの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行ったコミットの数                                                        |

**bdr.subscription**

このカタログテーブルには、ローカル BDR 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      | 対象ノードの 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になっているかどうか（レプリケーションである必要があります）       |
| 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（通常は 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る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 BDR 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     | ノードが属する BDR 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     | 起点ノードの名前                                       |
| 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ですか                                |
| 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[]   | サブスクライブし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、オリジン以外の他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トランザクションを遅らせる                     |
| 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オリジン名                 |
| 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      | ノードが属する BDR グループの OID                          |
| sub_id                | oid      | サブスクリプションの OID                                 |
| origin_id             | oid      | オリジンノードの OID                                   |
| target_id             | oid      | ターゲットノードの OID                                  |
| receive_lsn           | pg_lsn   | 受信した変更またはメッセージの最新の LSN (これはリスタートの場合に戻ることができます) |

[次のページに続く](#)

表 64 – 前のページからの続き

| Name                       | Type       | Description                                         |
|----------------------------|------------|-----------------------------------------------------|
| receive_commit_lsn         | pg_lsn     | 受信した最後の COMMIT の最新の LSN (これはリスタートの場合に戻ることができます)     |
| 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        | timestampz | ディスクに永続的にフラッシュされた、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_replay_timestamp | timestampz | 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.replication\_status**

このビューには、ローカルノードと EDB Postgres 分散クラスター内の他のすべてのノード間の着信レプリケーションステータスが表示されます。サブスクリプションが同じ LSN から少なくとも 2 回リスタートし、現在のアップストリーム接続が確立された後にまだ単一のトランザクションが適用されていない場合、レプリケーションはブロックされていると見なします。再起動後の最初のトランザクションが非常に大きく、まだ適用されている場合、**replication\_blocked** の結果が間違っている可能性があります。

これがロジカル・スタンバイ・ノードの場合、その上流ノードのステータスのみが表示されます。同様に、サブスクライバー専用ノードはレプリケーションの変更を他のノードに送信しないため、レプリケーションステータスは表示されません。

**bdr.replication\_status** 列

| Column           | Type    | Description              |
|------------------|---------|--------------------------|
| node_id          | oid     | ローカルノードの OID             |
| node_name        | name    | ローカルノードの名前               |
| origin_node_id   | oid     | オリジンノードの OID             |
| origin_node_name | name    | 起点ノードの名前                 |
| sub_id           | oid     | このオリジンノードのサブスクリプションの OID |
| sub_name         | name    | このオリジンノードのサブスクリプションの名前   |
| connected        | boolean | このノードはオリジンノードに接続されていますか? |

次のページに続く

表 65 – 前のページからの続き

| Column              | Type                     | Description                    |
|---------------------|--------------------------|--------------------------------|
| replication_blocked | boolean                  | このオリジンのレプリケーションは現在ブロックされていますか? |
| connect_time        | timestamp with time zone | 現在の接続が確立された時刻                  |
| disconnect_time     | timestamp with time zone | 最後の接続が切断された時刻                  |
| uptime              | interval                 | このオリジンで現在の接続がアクティブになってからの時間    |

**bdr.tables**

このビューには、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がリストされます。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                                                |
| relname            | name   | リレーションの名前                                                    |
| set_name           | name   | 複製セットの名前                                                     |
| set_ops            | text[] | レプリケートされたオペレーションのリスト                                         |
| 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、現在は NULL である必要があります

**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 | 関連する機能の名前        |
| 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**

読みやすい列を持つ `bdr.trigger` の展開図。

| Name               | Type               | Description |
|--------------------|--------------------|-------------|
| trigger_name       | name               | トリガーの名前     |
| event_manipulation | text               | 操作          |
| trigger_type       | bdr.trigger_type   | トリガーの種類     |
| trigger_table      | bdr.trigger_reloid | それを読み出すテーブル |
| trigger_function   | name               | 使用した機能      |

**bdr.workers**

実行中の BDR 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、BDR ワーカーの状態をさらに詳しく知ることができます。

**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       |

[次のページに続く](#)



表 69 – 前のページからの続き

| Name                    | Type       | Description                         |
|-------------------------|------------|-------------------------------------|
| 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       |
| worker_commit_timestamp | timestampz | このワーカーによって処理された最後のコミットのタイムスタンプがある場合 |
| worker_local_timestamp  | timestampz | 上記のコミットが処理されたローカル時間                 |

**bdr.worker\_errors**

BDR バックグラウンドワーカースプロセスからのエラーの永続的なログ。

**bdr.worker\_errors** 列

| Name                  | Type       | Description                                        |
|-----------------------|------------|----------------------------------------------------|
| node_group_name       | name       | BDR グループの名前                                        |
| origin_name           | name       | 起点ノードの名前                                           |
| source_name           | name       | 該当なし                                               |
| 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_name              | name       | サブスクリプションの名前                                       |
| worker_role           | int4       | このワーカーのロールの内部識別子（1：マネージャー、2：受信、3：ライター、4：出力、5：拡張機能） |
| worker_role_name      | text       | ロール名                                               |
| worker_pid            | int4       | エラーが発生したワーカーのプロセス ID                               |
| error_time            | timestampz | エラーの日時                                             |
| error_age             | interval   | エラーからの経過時間                                         |
| error_message         | text       | エラーの説明                                             |
| error_context_message | text       | エラーが発生したコンテキスト                                     |
| remoterelid           | oid        | そのノード上のリモートリレーションの OID                             |

**bdr.writers**

BDR 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        | ワーカプロセスのプロセス ID                       |
| syncing_rel           | int        | 同期されるリレーションの OID (ある場合)               |
| 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      |
| 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 (ある場合)   |
| remote_commit_lsn     | pg_lsn     | 処理された最後のコミットの LSN                     |
| commit_queue_position | int        | 内部コミットキューでの位置                         |
| commit_timestamp      | timestampz | 処理された最後のコミットのタイムスタンプ                  |
| 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     | このライタによってコミットされ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ファイルへのストリーミング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 |

**bdr.worker\_tasks**

**bdr.worker\_tasks** ビューには、BDR 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起動と登録アクティビティに関する基本的な統計が表示されます。

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 BDR 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 BDR 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_name | text                     | ワーカーロール名              |
| task_key_dboid            | oid                      | データベース識別子（利用可能な場合）    |
| datname                   | name                     | データベースの名前（利用可能な場合）    |
| task_key_subid            | oid                      | サブスクリプション識別子（利用可能な場合） |
| sub_name                  | name                     | サブスクリプションの名前（利用可能な場合） |
| task_key_ext_libname      | name                     | ライブラリの名前（ほとんどの場合 bdr） |
| task_key_ext_funcname     | name                     | 関数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                      | 利用可能な場合、リモート同期関係の識別子  |
| task_pid                  | integer                  | ワーカーのプロセス ID          |
| task_registered           | timestamp with time zone | ワーカー登録のタイムスタンプ        |
| since_registered          | interval                 | ワーカーが登録してからの間隔        |
| task_attached             | timestamp with time zone | ワーカーアタッチタイムスタンプ       |
| since_attached            | interval                 | ワーカーがアタッチされてからの間隔     |

次のページに続く

表 72 – 前のページからの続き

| Column                           | Type                     | Description          |
|----------------------------------|--------------------------|----------------------|
| task_exited                      | timestamp with time zone | ワーカー出口のタイムスタンプ       |
| since_exited                     | interval                 | ワーカーが終了してからの間隔       |
| task_success                     | boolean                  | ワーカーはまだ実行されていますか？    |
| task_next_launch_not_before      | timestamp with time zone | ワーカーが再度再起動されるタイムスタンプ |
| until_launch_allowed             | interval                 | 次の打ち上げまでの残り時間        |
| task_last_launch_request_id      | integer                  | 起動を要求したプロセス ID       |
| task_last_launch_request_time    | timestamp with time zone | リクエストが行われたときのタイムスタンプ |
| since_last_request               | interval                 | 最後の要求からの間隔           |
| task_last_launch_request_applied | boolean                  | 最後の要求は成功しましたか？       |
| task_nrequests                   | integer                  | リクエスト数               |
| 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 | 前回登録したタスクのタイムスタンプ    |
| since_prev_registered            | interval                 | 前回登録からの間隔            |
| task_prev_launched               | timestamp with time zone | 前回の起動のタイムスタンプ        |
| since_prev_launched              | interval                 | 前回の打ち上げからの間隔         |
| task_prev_exited                 | timestamp with time zone | 前のタスクが終了したときのタイムスタンプ |
| 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 | 最初の登録が発生したときのタイムスタンプ |
| since_first_registered           | interval                 | 初回登録からの間隔            |

**bdr.autopartition\_work\_queue**

autopartition ワーカ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.autopartition\_work\_queue** 列

| Column             | Type   | Description                                                                    |
|--------------------|--------|--------------------------------------------------------------------------------|
| ap_wq_workid       | bigint | 作業項目の一意の ID                                                                    |
| ap_wq_ruleid       | int    | autopartition_rules にリストされているルール ID。ルールは bdr.autopartition コマンドを使用して指定します      |
| ap_wq_relname      | name   | 自動パーティション化され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表領域の名前                                                         |
| ap_wq_partname     | name   | ワークアイテムによって作成されたパーティションの名前                                                     |
| ap_wq_work_kind    | char   | 作業の種類は、「c」（パーティションの作成）、「m」（パーティションの移行）、「d」（パーティションの削除）、「a」（パーティションの変更）のいずれかです。 |
| ap_wq_work_sql     | text   | 作業項目の SQL クエリ                                                                  |
| ap_wq_work_depends | Oid[]  | 作業項目が依存するノードの OID                                                              |

**bdr.autopartition\_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.autopartition\_workitem\_status** 列

| Column           | Type       | Description                                    |
|------------------|------------|------------------------------------------------|
| ap_wi_workid     | bigint     | ワークアイテムの ID                                    |
| ap_wi_nodeid     | Oid        | 作業項目が処理されているノードの OID                           |
| ap_wi_status     | char       | ステータスは、「q」（キュー）、「c」（完了）、「f」（失敗）、「u」（不明）のいずれかです |
| ap_wi_started_at | timestampz | 作業項目の開始タイムスタンプ                                 |

次のページに続く

表 74 – 前のページからの続き

| Column            | Type       | Description    |
|-------------------|------------|----------------|
| ap_wi_finished_at | timestampz | 作業項目の終了タイムスタンプ |

**bdr.autopartition\_local\_work\_queue**

autopartition ワーカ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.autopartition\_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して処理します。

**bdr.autopartition\_local\_work\_queue** 列

| Column             | Type   | Description                                                                    |
|--------------------|--------|--------------------------------------------------------------------------------|
| ap_wq_workid       | bigint | 作業項目の一意の 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します     |
| ap_wq_relname      | name   | 自動パーティション化され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表領域の名前。                                                        |
| ap_wq_partname     | name   | ワークアイテムによって作成されたパーティションの名前                                                     |
| ap_wq_work_kind    | char   | 作業の種類は、「c」（パーティションの作成）、「m」（パーティションの移行）、「d」（パーティションの削除）、「a」（パーティションの変更）のいずれかです。 |
| ap_wq_work_sql     | text   | 作業項目の SQL クエリ                                                                  |
| ap_wq_work_depends | Oid[]  | 常に NULL                                                                        |

**bdr.autopartition\_local\_workitem\_status**

ローカル管理テーブルの作業項目のステータス。

**bdr.autopartition\_local\_workitem\_status** 列

| Column            | Type       | Description                                             |
|-------------------|------------|---------------------------------------------------------|
| ap_wi_workid      | bigint     | ワークアイテムの ID                                             |
| ap_wi_nodeid      | Oid        | 作業項目が処理されている<br>ノードの OID                                |
| ap_wi_status      | char       | ステータスは、「q」(キュー)、<br>「c」(完了)、「f」(失敗)、「u」<br>(不明) のいずれかです |
| ap_wi_started_at  | timestampz | 作業項目の開始タイムスタ<br>ンプ                                      |
| ap_wi_finished_at | timestampz | 作業項目の終了タイムスタ<br>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 | カモパートナーのノード名                                              |
| is_camo_partner_connected  | text | 接続状態                                                      |
| is_camo_partner_ready      | text | 準備状況                                                      |
| camo_transactions_resolved | text | 保留中および未解決の<br>CAMO トランザクションは<br>ありますか                     |
| apply_lsn                  | text | リプレイとして報告された<br>最新の位置 (表示)                                |
| receive_lsn                | text | 受信した変更またはメッ<br>セージの最新の LSN (リス<br>タートの場合は前に戻るこ<br>とができます) |
| apply_queue_size           | text | apply_lsn と receive_lsn のバ<br>イト差                         |

**bdr.camo\_pairs**

すべてのクラスターに構成されているすべての CAMO ペアに関する情報。

**bdr.camo\_pairs** 列

| Name          | Type | Description                 |
|---------------|------|-----------------------------|
| node_group_id | oid  | ノードグループ ID                  |
| left_node_id  | oid  | ノード ID                      |
| right_node_id | oid  | ノード ID                      |
| require_raft  | bool | ローカルモードへの切り替えが過半数を必要とするかどうか |

**注釈:** *left* と *right* の名前には特別な意味はありません。BDR4 は対称 CAMO 構成のみを構成できます。つまり、ペアの両方のノードは相互に CAMO パートナーです。

**bdr.commit\_scopes**

グループコミットを有効にするために **bdr.commit\_scope** 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_group | 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 | 範囲の定義                      |

**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              |
| state            | text | ノード上のラフトワーカーの状態     |
| leader_id        | oid  | RAFT_LEADER のノード ID |
| current_term     | int  | いかな選挙の内部 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数         |
| voting_nodes     | int  | 投票するノード数            |
| 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 BDR スロット情報を収集します。

**bdr.group\_replslots\_details** 列

| Name            | Type     | Description                                    |
|-----------------|----------|------------------------------------------------|
| node_group_name | text     | BDR グループの名前                                    |
| origin_name     | text     | 起点ノードの名前                                       |
| 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（接続はありますか）                        |
| 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                           |

次のページに続く

表 81 – 前のページからの続き

| Name            | Type | Description                     |
|-----------------|------|---------------------------------|
| sent_lag_bytes  | int8 | sent_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 | flush_lsn と現在の WAL 書き込み位置のバイト差  |
| replay_lag_byte | int8 | replay_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<br>名前                            |
| target_node_name           | text | サブスクリプションのター<br>ゲットの名前                        |
| last_xact_replay_timestamp | text | 最後に再生されたトランザ<br>クションのタイムスタンプ                  |
| sub_lag_seconds            | text | now から<br>last_xact_replay_timestamp<br>までのラグ |

**bdr.group\_versions\_details**

`bdr.run_on_all_nodes` を使用して、すべてのノードから BDR 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 | ノードの BDR バージョン         |

## 内部カタログとビュー

### `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                                         |
|------------------------------------|-------------------------|-----------------------------------------------------|
| <code>ddl_epoch</code>             | <code>int8</code>       | 単調増加エポック番号                                          |
| <code>origin_node_id</code>        | <code>oid</code>        | このエポックの作成を要求したノードの内部ノード ID                          |
| <code>epoch_consume_timeout</code> | <code>timestampz</code> | このエポックのタイムアウト                                       |
| <code>epoch_consumed</code>        | <code>boolean</code>    | ローカルノードがエポックを完全に処理するとすぐに <code>true</code> に切り替わります |
| <code>epoch_consumed_lsn</code>    | <code>boolean</code>    | ローカルノードがエポックを処理した LSN                               |

### `bdr.internal_node_pre_commit`

内部カタログ表。 `bdr.node_pre_commit` ビューを使用します。

### `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                                                     |
|----------------------|-------------------|-----------------------------------------------------------------|
| <code>seqid</code>   | <code>oid</code>  | シーケンスの内部 OID                                                    |
| <code>seqkind</code> | <code>char</code> | 内部シーケンスの種類 ('l'=local、't'=timeshard、's'=snowflakeid、'g'=galloc) |

## `bdr.state_journal`

内部ノード状態ジャーナル。代わりに、診断のために `bdr.state_journal_details` を使用してください。

## `bdr.state_journal_details`

各ノードのノード状態の変更はすべて、診断目的で `bdr.state_journal` に永続的にログに記録されます。このビューは、ノード名と人間が読める状態の名前を提供し、そのジャーナルのすべての情報を運びます。ノードが正常に参加すると、最後の状態エントリは `BDR_PEER_STATE_ACTIVE` です。これは、`bdr.node_slots.state` にリストされている各レプリケーション接続の状態とは異なります。

## `bdr.state_journal_details` 列

| Name                       | Type       | Description            |
|----------------------------|------------|------------------------|
| <code>state_counter</code> | oid        | ノードごとに単調に増加するイベントカウンター |
| <code>node_id</code>       | oid        | 内部ノード ID               |
| <code>node_name</code>     | name       | ノードの名前                 |
| <code>state</code>         | oid        | 内部状態 ID                |
| <code>state_name</code>    | text       | 人間が読める状態の名前            |
| <code>entered_time</code>  | timestampz | 現在のノードが状態の変更を監視した時点    |

## BDR system functions

主に SQL から呼び出すファンクションを使用して、BDR 管理を実行します。BDR のすべての機能は `bdr` スキーマで公開されています。スキーマは、`search_path` に `bdr` を置く代わりに、これらの関数への呼び出しを修飾します。

## バージョン情報機能

### `bdr.bdr_version`

この関数は、現在使用されている BDR バージョンのテキスト表現を取得します。

**bdr.bdr\_version\_num**

この関数は、現在使用されている BDR のバージョン番号を取得します。バージョン番号は単調に増加するため、この値を小なりと大なりの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 1 つの数値に返します。

```
MAJOR_VERSION * 10000 + MINOR_VERSION * 100 + PATCH_RELEASE
```

**システム情報機能****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返します。

**システムと進行情報パラメーター**

BDR は、psql の SHOW を使用するか、クライアントアプリケーションから PQparameterStatus（または同等の）を使用して照会できるいくつかのパラメーターを公開します。

**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されているノード ID に設定されます。これにより、アプリケーションは透過プロキシの背後でも、接続されているノードを把握できます。

CAMO でも使用されます。Connection pools and proxies を参照してください。

**bdr.last\_committed\_lsn**

非同期トランザクションのすべての COMMIT の後、このパラメーターは、起点ノードのコミットレコードの最後を指すように更新されます。それを bdr.wait\_for\_apply\_queue と組み合わせると、アプリケーションは複数のノードにわたって因果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になるまで待機できます。

## transaction\_id

Postgres がトランザクション ID を割り当てるとすぐに、CAMO が有効になっている場合、このパラメーターは更新されて、割り当てられたトランザクション ID が表示されます。

## bdr.is\_node\_connected

### 概要

```
bdr.is_node_connected(node_name name)
```

特定のピアの Walsender がこのノードでアクティブかどうかを確認して、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も低いか、それとも bdr.global\_commit\_timeout よりも低いかを確認して、ブール値を返します。

## コンセンサス機能

### bdr.consensus\_disable

サーバーが再起動するか、bdr.consensus\_enable を使用して再度有効になるまで（どちらか早い方）、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。

**警告:** コンセンサスを無効にすると BDR の一部の機能が無効になり、長期間無効のままにすると EDB Postgres 分散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てください。

### bdr.consensus\_enable

ローカルノードで無効になっているコンセンサスワーカーを再度有効にしました。

## bdr.consensus\_proto\_version

ローカルノードで現在使用されているコンセンサスプロトコルバージョンを返します。

BDR グループの再構成の内部メカニズムが必要です。

## 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して適用された状態から新しい BDR コンセンサススナップショットを生成し、bytea として返します。

デフォルトでは、サポートされている最も高い 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である必要もありません。ただし、`bdr.consensus_snapshot_import()` はそ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 ログ位置の後ろにある同じ BDR 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。

この機能を動作させるには、ローカル BDR コンセンサスワーカーを無効にする必要があります。典型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();
```

BDR 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 ロックの試行が失敗するか、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。
- Eager および CAMO トランザクションが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的に非常に短いです。

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て、代わりに使用する方が実用的な場合があります。そうすることで、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(IN snapshot bytea)
```

通常は同じ BDR ノードグループ内の別のノードから、`bdr.consensus_snapshot_export()` によって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。

別のノードにある `bdr.local_consensus_snapshot` テーブルの `snapshot` フィールドから直接抽出し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、破損またはユーザエラーが発生した場合に、BDR ノードのカatalogの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するノードの `apply_index` がスナップショットをエクスポートするノードの `commit_index` 以下である場合、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。 ) ログがすでに先にあるためにスナップショットを受け入れないノードは、エラーを発生させ、変更を行いません。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フェッチするため、インポートされたスナップショットは完全に最新である必要はありません。

この機能を動作させるには、インポートするノードで BDR 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。

次を実行して、この関数を使用して、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());
```

このアプローチは、現在適用されているログ位置まで Raft ログを切り捨てる場合もあります。

## bdr.consensus\_snapshot\_verify

### 概要

```
bdr.consensus_snapshot_verify(IN snapshot bytea)
```

`bdr.consensus_snapshot_export()` によってエクスポートされた特定の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返します。

## **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返します。bdr.get\_consensus\_status のエイリアス。

## **bdr.raft\_leadership\_transfer**

### 概要

```
bdr.raft_leadership_transfer(IN node_name text, IN wait_for_completion boolean)
```

node\_name で識別されるノードを Raft リーダーとして要求します。要求は任意の BDR ノードから開始でき、指定されたノードにリーダーシップを転送するために現在のリーダーに内部的に転送されます。指定されたノードは、完全な投票権を持つ ACTIVE BDR 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で処理されます。bdr.raft\_election\_timeout 期間中にノードがリーダーになれない場合、他の有能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 プロトコルごとに期間をかけて変更できます。true の結果は、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、関数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 リーダーにならない場合（たとえば、ネットワークの問題により）、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 wait\_for\_completion で statement\_timeout を設定することをお勧めします。

### ユーティリティ関数

## **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 を渡すまで待機します。位置が指定されていない場合、ローカルノードで現在の書き込み位置が使用されます。

スロット名が渡されない場合、すべての BDR スロットが LSN を渡すまで待機します。

この関数は、他のノードからの変更について 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 が早く完了するように設定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)
```

## パラメーター

- `slot_name` —レプリケーションスロットの名前、または NULL の場合、すべての BDR スロット (のみ)。
- `target_lsn` —待機する LSN。NULL の場合、ローカルノード上の現在の書き込み LSN を使用します。

**bdr.wait\_for\_apply\_queue**

ファンクション `bdr.wait_for_apply_queue` を使用すると、BDR ノードは、特定の BDR ノードから発生する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、古い読み取りを防ぐことができます。

便宜上、BDR は CAMO および CAMO パートナーノードにこの機能のバリエーションを提供します。

`bdr.wait_for_camo_partner_queue` を参照してください。

特定の LSN が指定されている場合、それがピアが待機する復旧ストリームのポイントです。これを、以前の接続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` で使用できます。

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確認し、指定されたピアノードから受信した最後のトランザクションの LSN を使用して、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わからない場合に特に役立ちます。この場合、送信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しません。

## 概要

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

## パラメーター

- `peer_node_name` —着信トランザクションがキューに入れられ、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消費されるのを待ちます。
- `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内の LSN。通常、ピアノードから `bdr.last_committed_lsn` を介して学習されます。

### bdr.get\_node\_sub\_receive\_lsn

サブスクライバーでこの関数を使用して、特定のオリジンから受信した最後の LSN を取得できます。適用す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に入れるように、フィルター処理を行うことも、フィルター処理することもできます。

この関数出力と `bdr.get_node_sub_apply_lsn()` の出力の差が、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

- `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、この関数は最後の 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、サブスクライバー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関数を使用して、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

- `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.run\_on\_all\_nodes

すべてのノードでクエリを実行する関数。

**警告:** この関数は、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を実行します。この関数に権限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

- `query` — 実行する任意のクエリ。

## 注意事項

この関数は他のノードに接続してクエリを実行し、各ノードからの結果を JSON 形式で返します。各ノードから複数の行が返され、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` またはその他の実行時パラメーターは設定されていないため、デフォルトが使用されます。

この機能は、通常のレプリケーションを行いません。既知のすべて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、コマンドは既にクラスター内のすべてのノードに送信されているため、接続は `bdr.ddl_replication = off` で作成されます。

レプリケーションが中断され、ノード間の一貫性が失われる危険があるため、この機能を使用するときは注意してください。透過的な 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

次のクエリのように、監視でこの関数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ものを返します。

```
[
  {
    "dsn": "host=node1 port=5432 dbname=bdrdb user=postgres ",
    "node_id": "2232128708",
    "response": {
      "command_status": "SELECT 1",
      "command_tuples": [
        {
          "origin_name": "node1",
          "target_name": "node2",
```

(次のページに続く)

(前のページからの続き)

```

        "local_slot_name": "bdr_bdrdb_bdrgroup_node2",
        "replay_lag_size": "0 bytes"
    }
]
},
"node_name": "node1"
},
{
    "dsn": "host=node2 port=5432 dbname=bdrdb user=postgres ",
    "node_id": "2058684375",
    "response": {
        "command_status": "SELECT 1",
        "command_tuples": [
            {
                "origin_name": "node2",
                "target_name": "node1",
                "local_slot_name": "bdr_bdrdb_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関数。

**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で、リモートノードで任意のクエリを実行します。このファンクションに権限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

- `node_names` —クエリが実行されるノード名の Text ARRAY。
- `query` —実行する任意のクエリ。

## 注意事項

この関数は他のノードに接続してクエリを実行し、各ノードからの結果を JSON 形式で返します。JSON 配列としてエンコードされた、各ノードから複数の行を返すことができます。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` またはその他の実行時パラメーターは設定されていないため、デフォルトが使用されます。

この機能は、通常のレプリケーションを行いません。既知のすべて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、コマンドは既にクラスター内のすべてのノードに送信されているため、接続は `bdr.ddl_replication = off` で作成されます。

レプリケーションが中断され、ノード間の一貫性が失われる危険があるため、この機能を使用するときは注意してください。透過的な 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。

## `bdr.run_on_group`

ノードのグループでクエリを実行する関数。

**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で、リモートノードで任意のクエリを実行します。このファンクションに権限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

- `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。

## 注意事項

この関数は他のノードに接続してクエリを実行し、各ノードからの結果を JSON 形式で返します。JSON 配列としてエンコードされた、各ノードから複数の行を返すことができます。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` またはその他の実行時パラメーターは設定されていないため、デフォルトが使用されます。

この機能は、通常のレプリケーションを行いません。既知のすべて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、コマンドは既にクラスター内のすべてのノードに送信されているため、接続は `bdr.ddl_replication = off` で作成されます。

レプリケーションが中断され、ノード間の一貫性が失われる危険があるため、この機能を使用するときは注意してください。透過的な 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。

## `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 ロックについては、[DDL ロックの詳細](#) を参照してください。

## 概要

```
bdr.global_lock_table(relation regclass)
```

## パラメーター

- `relation` —ロックするリレーションの名前または oid。

## 注意事項

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 権限が必要です。

## bdr.wait\_for\_xid\_progress

この関数を使用して、特定のノード（ノード ID で識別）で発生した特定のトランザクション（XID で識別）がクラスターで十分に進行するまで待機できます。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

- `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のを待ちます。それ以外の場合は、現在のノードのみを待ちます。

### 注意事項

現在これらのトランザクションのみが追跡されているため、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この関数を使用できます。間違った `origin_node_id` または `origin_topxid` が指定された場合、関数は永久に、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

```
bdrdb=# SELECT bdr.local_group_slot_name();
 local_group_slot_name
-----
 bdr_bdrdb_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()` に渡された値と同じですが、グループの作成時に `node_group_type` が NULL として渡された場合に `normal` が返されます。

### 例

```
bdrdb=# SELECT bdr.node_group_type(bdrgroup);
node_group_type
-----
normal
```

## グローバル勧告ロック

BDR は、グローバル勧告ロックをサポートしています。これらのロックは、BDR 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いて、PostgreSQL で利用可能な勧告ロックに似ています。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多数決コンセンサスによって取得され、すべてのノードの過半数が連携できる限り、1 つ以上のノードがダウンまたは遅れ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。そのため、同じノード上の別のノードまたは別のバックエンドがオブジェクトのアドバイザリ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。

アドバイザリロックは本質的にトランザクションです。したがって、トランザクションの終了前に明示的に解放されない限り、トランザクションが終了すると自動的にロックが解放されます。この場合、リリースされ次第利用可能になります。セッションレベルのアドバイザリ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、他のセッションでできるように解放するには、3回アンロックする必要があります。

### **bdr.global\_advisory\_lock**

このファンクションは、指定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

- **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

- **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

- **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**

クラスタ全体のバージョンチェックを提供するために、この関数はビュー `bdr.group_version_details` から返される BDR バージョン情報を使用します。

## 概要

```
bdr.monitor_group_versions()
```

## 注意事項

この関数は、[Monitoring](#) で説明されているように、フィールド `status` および `message` を返します。

この関数は `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、この関数はビュー `bdr.group_raft_details` から返された BDR Raft 情報を使用します。

## 概要

```
bdr.monitor_group_raft()
```

## 注意事項

この関数は、[Monitoring](#) で説明されているように、フィールド `status` および `message` を返します。

この関数は `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、BDR 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。

## 概要

```
bdr.monitor_local_replslots()
```

## 注意事項

この関数は、[レプリケーションスロットのモニタリング](#) で説明されているように、フィールド `status` および `message` を返します。

## bdr.wal\_sender\_stats

[デコードワーカー](#) が有効になっている場合、この機能は、各 WAL 送信者が読み取っているデコーダースロットと現在の LCR（論理変更レコード）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() &rarr; setof record (pid integer, is_using_lcr boolean, ↵  
↪ decoder_slot_name TEXT, lcr_file_name TEXT)
```

## 出力列

- `pid` — WAL 送信者の PID (`pg_stat_replication` の `pid` 列に対応)。
- `is_using_lcr` — WAL 送信者が LCR ファイルを送信しているかどうか。 `is_using_lcr` が `FALSE` の場合、次の列は `NULL` です。
- `decoder_slot_name` —デコーダーの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**

デコードワーカーが有効になっている場合、この関数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、

`pg_replication_slots` を介して利用できるよりも細かいワーカーの進行状況に関する情報を提供します。

## 概要

```
bdr.get_decoding_worker_stat() &rarr; setof record (pid integer, decoded_upto_lsn pg_
↳lsn, waiting BOOL, waiting_for_lsn pg_lsn)
```

## 出力列

- `pid` —デコードワーカーの PID (`pg_replication_slots` の `active_pid` 列に対応)。
- `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までの LSN。
- `waiting` —デコードワーカーが新しい WAL を待っているかどうか。
- `waiting_for_lsn` —次に予想される WAL の LSN。

## 注意事項

詳細については、[LCR を使用した WAL 送信者の監視](#) を参照してください。

**bdr.lag\_control**

ラグ制御が有効になっている場合、この機能は、ローカルノードと現在のデータベースに構成されているラグメジャーに準拠するノードの遅延と数に関する情報を表示します。

## 概要

`bdr.lag_control()`

## 出力列

- `commit_delay` —現在のランタイム コミット遅延（ミリ秒単位）。
- `commit_delay_maximum` —構成された最大コミット遅延（ミリ秒単位）。
- `commit_delay_adjustment` —サンプル間隔中に可能なランタイムコミット遅延への変更（ミリ秒単位）。
- `conforming_nodes` —ラグメジャーに準拠するノードの現在のランタイム数。
- `conforming_nodes_minimum` —ラグメジャーに準拠するために必要なノードの構成された最小数。それ未満では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ラグサイズ（キロバイト）。
- `lag_bytes_maximum` —構成された最大ラグサイズ（キロバイト）。
- `lag_time_threshold` —ミリ秒単位のコミット遅延が適用されるラグタイム。
- `lag_time_maximum` —構成された最大遅延時間（ミリ秒）。
- `sample_interval` —ミリ秒単位のラグサンプルと可能なコミット遅延調整間の構成された最小時間。

## 内部機能

### BDR メッセージペイロード関数

`bdr.decode_message_response_payload` および `bdr.decode_message_payload`

これらの関数は、コンセンサスペイロードをより人間が読みやすい出力にデコードします。

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.get_global_locks`

この機能は、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\_slot\_flush\_timestamp**

指定された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 内部機能レプリケーション機能**

`bdr.internal_alter_sequence_set_kind` 、 `internal_replication_set_add_table` 、  
`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するファンクション。

BDR 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にのみ存在します。

### **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。

BDR グループの再構成の内部メカニズムが必要です。

### **bdr.isolation\_test\_session\_is\_blocked**

グローバルロック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し）するヘルパー関数。

分離/並行性テストに使用されます。

### **bdr.local\_node\_info**

この機能は、BDR グループの再構成の内部メカニズムが必要とするローカルノードの情報を表示します。

ビュー `bdr.local_node_summary` は、ユーザーが使用するのに役立つ同様の情報を提供します。

### **bdr.msgb\_connect**

コンセンサスプロトコルが使用する、別のノードの接続プラーに接続するための機能。

**bdr.msgb\_deliver\_message**

コンセンサスプロトコルによって使用される、別のノードの接続プラーにメッセージを送信するための関数。

**bdr.peer\_state\_name**

この関数は、ノードの状態 (`node_state`) をテキスト表現に変換し、主に `bdr.node_summary` ビューを実装するために使用されます。

**bdr.request\_replay\_progress\_update**

「リプレイ進行状況更新」Raft メッセージの即時書き込みを要求します。これは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.seq\_nextval**

シーケンスインクリメントの内部実装。

*BDR グローバルシーケンス* と対話するクエリでは、標準の `nextval`

の代わりにこの関数を使用します。

**注意事項**

以下も内部 BDR シーケンス操作関数です。 `bdr.seq_currval` と `bdr.sql_lastval` は自動的に使用されます。

**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、主に `bdr.subscription_summary` ビューを実装するために使用されます。

**bdr.conflict\_resolution\_to\_string**

競合解決を oid からテキストに変換します。

ビュー `bdr.conflict_history_summary` はこれを使用して、競合解決のためのユーザーフレンドリーな情報を提供します。



**bdr.conflict\_type\_to\_string**

競合タイプを oid からテキストに変換します。

ビュー `bdr.conflict_history_summary` はこれを使用して、競合解決のためのユーザーフレンドリーな情報を提供します。

**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.reset\_subscription\_stats**

`bdr.stat_subscription` で表示されるように、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、ブール値の結果を返します。

**bdr.reset\_relation\_stats**

`bdr.stat_relation` から見た、リレーションの統計をリセットした後、ブール値の結果を返します。

**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**

- `xid` — オリジンが返されるトランザクション ID、

**bdr.difference\_fix\_origin\_create**

引数として渡された名前を `bdr_` プレフィックスを追加して、レプリケーションオリジンを作成します。オリジンの内部 ID を返します。これは、`postgres` のスーパーユーザー権限ではなく `bdr_superuser` が必要であることを除いて、`pg_replication_origin_create()` と同じ機能を実行します。

## 概要

### **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していることをマークします。このファンクションは、セッションに事前に作成された `bdr_local_only_origin` ローカルレプリケーション起点を暗黙的に使用します。リプレイ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、このファンクションが `postgres` のスーパーユーザー権限ではなく `bdr_superuser` を必要とすることを除き、`pg_replication_origin_session_setup()` と同じ機能を実行します。関数の以前の形式である `bdr.difference_fix_session_setup(text)` は非推奨であり、今後のリリースで削除されます。

## 概要

`bdr.difference_fix_session_setup()`

### **bdr.difference\_fix\_session\_reset**

現在のセッションをどのオリジンからも再生していないものとしてマークし、基本的に `bdr.difference_fix_session_setup()` の効果をリセットします。 `void` を返します。この関数には、`postgres` のスーパーユーザー権限ではなく `bdr_superuser` が必要であることを除いて、`pg_replication_origin_session_reset()` と同じ機能があります。

## 概要

`bdr.difference_fix_session_reset()`

### **bdr.difference\_fix\_xact\_set\_avoid\_conflict**

LSN ' 0/0' およびタイムスタンプ ' 2000-01-01' でコミットされたトランザクションを再生するものとして、現在のトランザクションをマークします。この関数には、`postgres` のスーパーユーザー権限ではなく `bdr_superuser` が必要であることを除いて、`pg_replication_origin_xact_setup('0/0', '2000-01-01')` と同じ機能があります。

## 概要

`bdr.difference_fix_xact_set_avoid_conflict()`

**bdr.resynchronous\_table\_from\_node** (node\_name name、リレーション regclass)

リモートノードからリレーションを再同期します。

**概要**

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

**パラメーター**

- **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コピーするリレーション。

**注意事項**

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、ローカルでリレーションを切り捨て、リモートノードからリレーションにデータをコピーします。

リレーションは、同じ名前と定義で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。

- 同じパーティション定義を持つパーティションテーブルの再同期
- パーティションテーブルから非パーティションテーブルへ、およびその逆の再同期
- 外部キー制約を一時的に削除して再作成することによる参照テーブルの再同期

参照されるテーブルで関数を実行した後、参照される列のデータが参照する列の値と一致しなくなった場合、エラーがスローされます。参照テーブルデータを再同期した後、関数を再実行します。

さらに、リモートノードからデータをコピーした後にローカルで生成された列の値を計算することにより、生成された列を含む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では無視されます。

**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.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。これはttlに依存します。ttlがNULLの場合、これはinfinityを返します。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

- **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** —設定すると、現在の値が **prev\_value** と等しい場合にのみ書き込み操作が実行されます。
- **ttl** —新しい値の存続時間（ミリ秒）。

### 注意事項

これは内部関数であり、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この関数を使用しないでください。

### **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

- **key** —取得する任意のキー。

## 注意事項

これは内部関数であり、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この関数を使用しないでください。

### `bdr.alter_subscription_skip_changes_upto`

論理レプリケーションはバージョン間でレプリケートできますが、ロールのようなグローバルな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の適用プロセスでエラーが発生し、変更の適用が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` はレプリケーションをブロックしている欠落しているテーブル、`CREATE` は必要なロール、`GRANT` は必要なパーミッションなどです。ただし、その方法で問題を解決できず、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。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、論理デコード関数を使用して変更ストリームをピークします。

トランザクションが 1 つの変更のみを加えた場合を除き、多くの場合、ターゲット側にトランザクションの効果を手動で適用する必要があるため、次の例に示すように、可能な限り問題の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できるため、これは便利な関数です。論理デコードでいくつかの種類のエラーをバイパスする場合がありますが、より高速にスキップします。

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機能します。

通常の手順は次のとおりです。

1. 問題のサブスクリプションと問題のコミットの LSN を特定します。
1. サブスクリプションを無効にします。
1. 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。
1. ターゲットノード上の `bdr.alter_subscription_skip_changes_upto`。
1. 必要に応じて、修復した変更または同等の変更をターゲットに適用します。
  1. サブスクリプションを再度有効にします。

**警告:** この関数を使用すると問題が悪化します。それが唯一の選択肢であると確信できる場合を除き、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がエラーで失敗し、ターゲット側での変更などの影響の少な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した。

次の例のように、エラーログで、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ません。

```
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。それはすべてです。そのため、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レコードをコピーして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、読みやすくするために複数行に分割されていますが、1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 に変更することで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より高速にスキップします。

```
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（またはそれらの修復バージョン）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);
```





## 第5章 Choosing your architecture

Always On アーキテクチャは、プラクティスをカプセル化し、マルチプルの構成で可能な限り最高のサービス可用性を達成するのに役立つ EDB の Trusted Postgres アーキテクチャを反映しています。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、ハードウェアの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 %の可用性を実現する機能を活用しています。

ここで説明されている例以外のアーキテクチャにも EDB Postgres Distributed を使用できます。ユースケース固有のバリエーションが本番環境に正常にデプロイされました。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。また、Always On アーキテクチャ用の EDB の標準デプロイツールである TPAExec を有効にして、バリエーションを実稼働環境でサポートする必要があります。

### 5.1 標準の EDB Always On アーキテクチャ

EDB は、4 つの標準 Always On アーキテクチャを特定しました。

- *Always On Bronze*  
：2 つのデータノード、監視ノード、およびローカルバックアップがある単一の場所
- *Always On Silver*  
：3 つのデータノードを持つ単一の場所とオフサイトバックアップを持つ 2 番目の場所
- *Always On Gold*  
：それぞれ 2 つのデータノードを持つ 2 つの場所と witness を持つ 3 番目の場所
- *Always On Platinum*  
：それぞれ 3 つのデータノードを持つ 2 つの場所（2 つのマスターとホットスタンバイモードの追加の冗長ハードウェア）

各アーキテクチャは、データを少なくとも 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、ゼロ復旧ポイント（RPO）を提供できるため、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がゼロになります。ただし、いずれかのデータノードに障害が発生すると RPO が延長されるため、Bronze アーキテクチャでは同期レプリケーションはお勧めしません。

可用性の保証を高めると、ハードウェアとライセンス、ネットワーク要件、および運用の複雑さに関する追加のコストが発生します。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の要件を注意深く検討してください。

## 5.2 アーキテクチャの詳細

EDB Postgres Distributed は、*show-raft* ベースのコンセンサスアーキテクチャを使用します。通常のデータベース操作（挿入、選択、削除）にはクラスター全体のコンセンサスはありませんが、EDB Postgres Distributed は、新しいグローバルシーケンスの生成や分散 DDL 操作など、コンセンサスを必要とする決定を行う際に奇数のノードの恩恵を受けます。より単純なアーキテクチャでも、すべてがデータを保存しているわけではない場合でも、1つの場所内に常に3つのノードがあります。2つのロケーションを使用する Always On Gold と Platinum では、RAFT 要件をサポートするための監視ノードとして5番目のノードが導入されています。

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 HARP プロキシサーバーは、マルチホスト接続文字列内の個別のエントリです。他の接続メカニズムは運用環境に正常にデプロイされていますが、標準の Always On アーキテクチャの一部ではありません。

## 5.3 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。

|                                                                  | Bronze                                                        |
|------------------------------------------------------------------|---------------------------------------------------------------|
| Minimum Locations Needed                                         | 1                                                             |
| Location failure protection                                      | No - unless offsite backup                                    |
| Failover to DR or full DC                                        | NA (DR only if offsite backup)                                |
| Fast local restoration of high availability after device failure | No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB |
| Cross location network traffic                                   | None (unless offsite backup then backup traffic only)         |
| License cost                                                     | 2 data nodes                                                  |

### 5.3.1 Always On Bronze

Always On Bronze アーキテクチャには以下が含まれます。

- 2つの BDR データノード
- データを保持しないがコンセンサスに使用される 1つの BDR ノード
- 2つの HARP-アプリケーショントラフィックを「リード」マスターにルーティングするためのプロキシノード
- バックアップとリカバリ用の 1つのバーマンノードはオンサイトまたはオフサイトにできます

BDR と HARP-プロキシノードはアベイラビリティゾーンに分散する必要があります。

## Always On Bronze

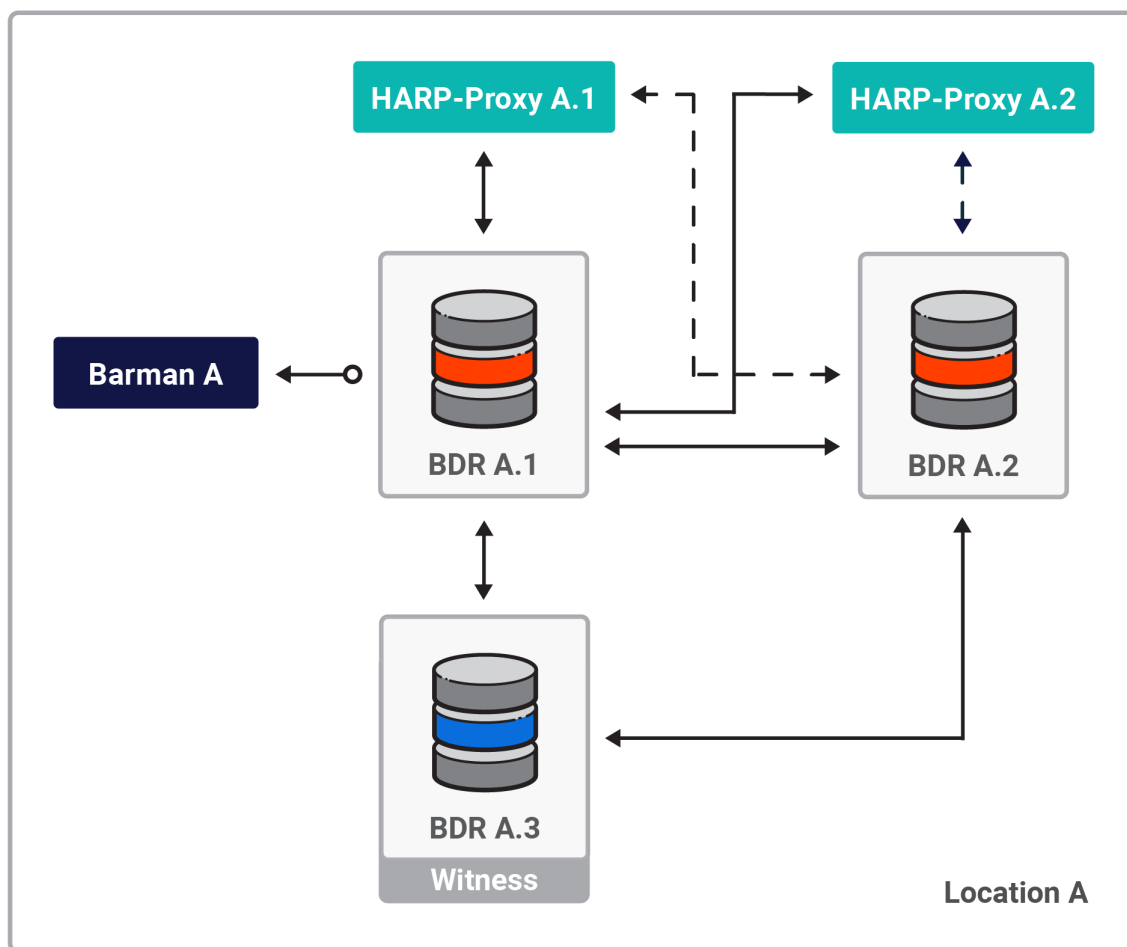


図 1: Always On Bronze architecture

### 5.3.2 Always On Silver

Always On Silver アーキテクチャには以下が含まれます。

- 3つの BDR データノード
- 2つの HARP-プロキシノード
- Barman オフサイト（オフサイトはオプションですが、推奨されます）

BDR ノードと HARP ノードは AZ 全体に分散する必要があります。Barman はロケーション A にいます。

#### Always On Silver

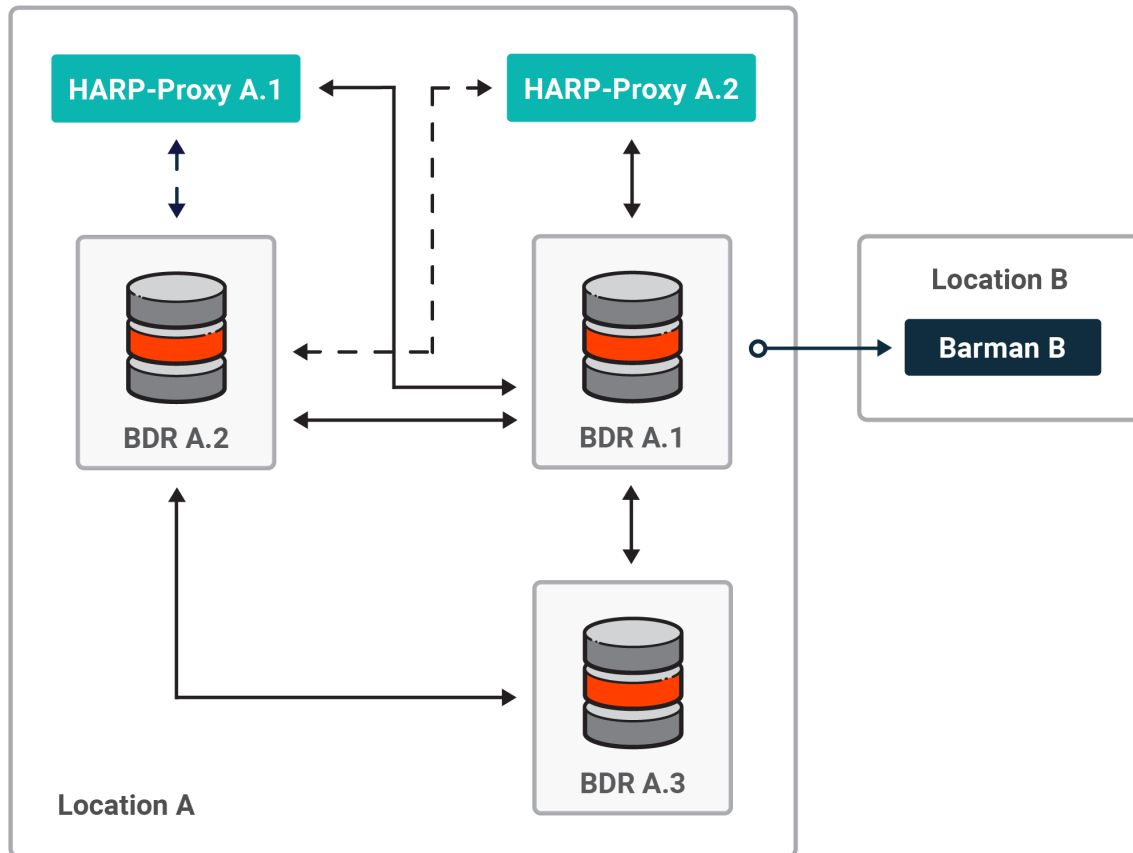


図 2: Always On Silver architecture

### 5.3.3 Always On Gold

このアーキテクチャは、ローカルの回復性/冗長性を最初に優先し、ロケーション全体がオフラインまたは障害が発生した場合にのみリモートの場所を優先します。

このアーキテクチャにより、競合が発生しない/低い場所で地理的に分散された書き込みが可能になります

Always On Gold アーキテクチャでは、場所間の移動が必要ですが、すべてのデータはレプリケートされ、フェールオーバーがフル容量で開始されるとアプリケーションで使用できます。

Always On Gold アーキテクチャには以下が含まれます。

- 4つの BDR データノード（ロケーション A に 2 つ、ロケーション B に 2 つ）
- ロケーション C の 1 つの BDR ノード（オプションですが推奨）
- 4つの HARP-プロキシノード（ロケーション A に 2 つ、ロケーション B に 2 つ）
- 2つの Barman ノード（ロケーション A に 1 つ、ロケーション B に 1 つ）

Always On Gold 3DC, 2 Lead Masters

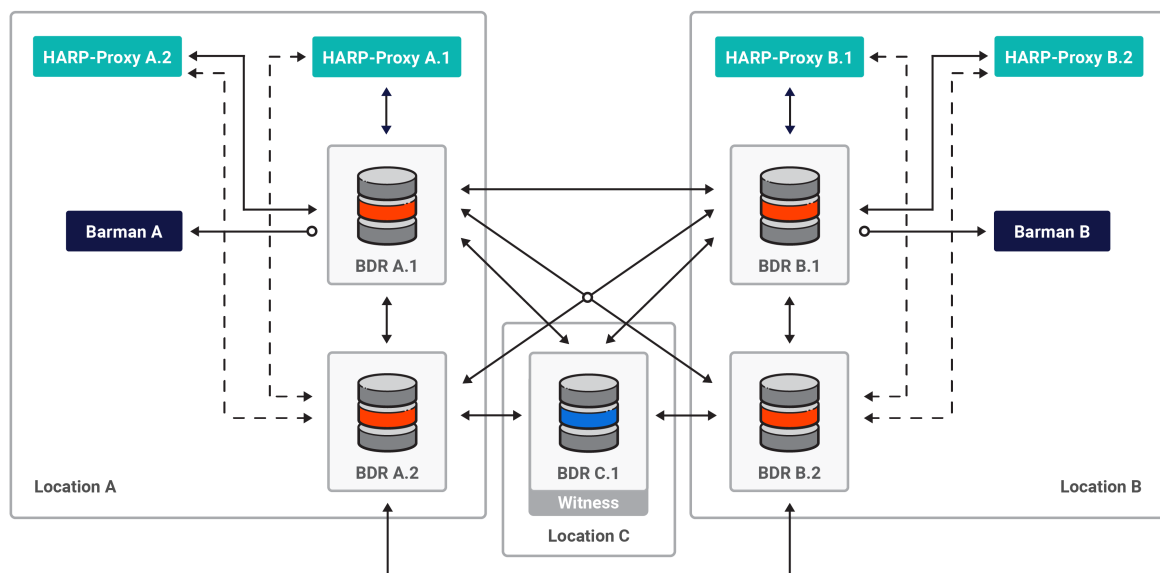


図 3: Always On Gold architecture

### 5.3.4 Always On Platinum

Always On Platinum アーキテクチャには次のものが含まれます。

- 4つの BDR データノード（ロケーション A に 2 つ、ロケーション B に 2 つ）
- 2つのロジカルスタンバイノード（1基はA拠点、1基はB拠点）
- ロケーション C の 1 つの監視ノード（オプションですが推奨）
- 4つの HARP-プロキシノード（ロケーション A に 2 つ、ロケーション B に 2 つ）

- 2つの Barman ノード（ロケーション A に 1 つ、ロケーション B に 1 つ）

どちらの場所にも冗長なホットスタンバイハードウェアがあり、ハードウェアに障害が発生した場合にバックアップからの BDR ノードの復元を待たずにローカルの高可用性を維持します。

#### Always On Platinum

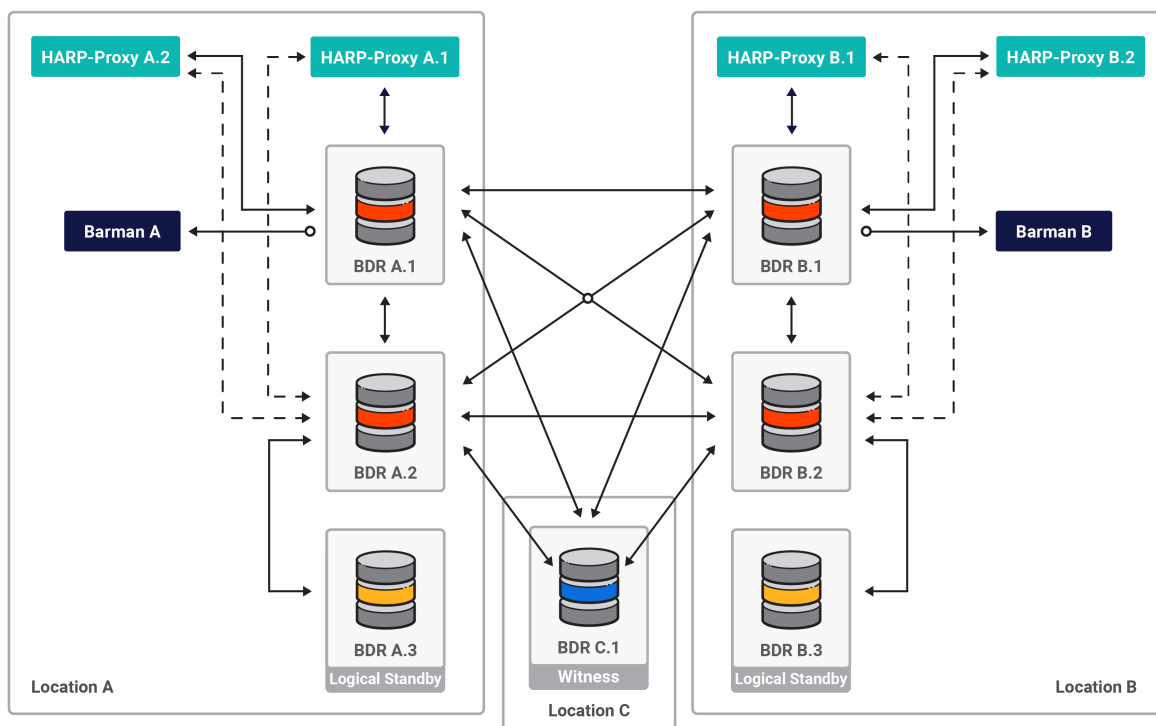


図 4: Always On Platinum architecture

## 第6章 Choosing a Postgres distribution

EDB Postgres Distributed は、PostgreSQL、EDB Postgres Extended Server、または EDB Postgres Advanced Server の 3 つの異なる Postgres ディストリビューションで展開できます。特定の EDB Postgres Distributed 機能を利用できるかどうかは、使用する Postgres ディストリビューションによって異なります。したがって、ビジネスニーズに最適な Postgres ディストリビューションを採用することが不可欠です。たとえば、「Commit At Most Once (CAMO)」機能があ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、オープンソースの PostgreSQL は CAMO を処理するために必要なコア機能を備えていないため、採用しないでください。

次の表は、Postgres ディストリビューションとバージョンに依存する EDB Postgres Distributed の機能をリストしています。

| Feature                                         | PostgreSQL | EDB Postgres Extended | EDB Postgres Advanced |
|-------------------------------------------------|------------|-----------------------|-----------------------|
| Commit At Most Once (CAMO)                      | N          | Y                     | 14+                   |
| Eager Replication                               | N          | Y                     | 14+                   |
| Decoding Worker                                 | N          | 13+                   | 14+                   |
| Assessment Tooling                              | N          | Y                     | 14+                   |
| Lag Tracker                                     | N          | Y                     | 14+                   |
| Lag Control                                     | N          | Y                     | 14+                   |
| Timestamp Snapshots                             | N          | Y                     | 14+                   |
| Transaction Streaming                           | 14+        | 13+                   | 14+                   |
| Missing Partition Conflict                      | N          | Y                     | 14+                   |
| No need for UPDATE Trigger on tables with TOAST | N          | Y                     | 14+                   |
| Automatically hold back FREEZE                  | N          | Y                     | 14+                   |





## 第7章 Choosing durability

EDB Postgres Distributed を使用すると、一貫性、可用性、および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。

\*非同期

- 同期 (`synchronous_standby_names` を使用)
- *Commit At Most Once*
- *Eager All-Node Replication* を使用した CAMO
- CAMO とグループコミット

詳細については、*Choosing durability* を参照してください。



## 第8章 Other considerations

展開を計画するときは、これらの他の考慮事項を確認してください。

### 8.1 展開とサイジングの考慮事項

実稼働デプロイメントの場合、EDB は、各 Postgres データノードと各ロジカルスタンバイに少なくとも 12 コアをお勧めします。監視ノードはデータレプリケーションオペレーションに参加しないため、この要件を満たす必要はありません。ロジカルスタンバイのサイズは常にデータノードとまったく同じにして、ノードの昇格によるパフォーマンスの低下を回避します。実稼働展開では、HARP プロキシノードにはそれぞれ少なくとも 4 つのコアが必要です。EDB は、EDB のプロフェッショナルサービスチームと協力して、パフォーマンス要件の詳細なベンチマークを行うことをお勧めします。

開発目的では、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、ロジカルスタンバイ、Barman ノード、および HARP プロキシノードを仮想マシンに、またはベアメタル展開モードで展開できます。ただし、データノードと HARP プロキシノード間のアンチアフィニティを維持します。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。また、同じ物理ハードウェア上の VM に複数の HARP プロキシノードを展開しないでください。これも復元性を低下させます。

単一のデータノードを持つ単一の HARP プロキシノードを同じ物理ハードウェアに展開できますが、適切な CPU とメモリのリソース割り当てを保証するために VM として展開する必要があります。

### 8.2 時計とタイムゾーン

EDB Postgres Distributed 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いるため、真にワールドワイドなデータベースクラスターが可能です。個々のサーバーを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、人間が読めるサーバーログにアクセスして比較できるようにすることをお勧めします。

サーバーの時計は、NTP または他のソリューションを使用して同期する必要があります。

他のソリューションの場合とは異なり、クロックの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コントロールを提供します。

*Column-level conflict detection* で説明されているように、EDB Postgres

Distributed は行バージョンの競合検出も提供します。



## 第9章 Deployment options

次の方法を使用して、EDB Postgres 分散製品を展開してインストールできます。

- TPAexec は、Ansible を使用して、TPA (Trusted Postgres Architecture) の指定に従って Postgres クラスターを構築するオーケストレーションツールです。さまざまなシナリオで Postgres を設定および操作する方法を文書化したリファレンスアーキテクチャのセット。TPA は、EDB が続くベストプラクティスを表しており、その推奨事項は、実稼働環境と同様に迅速なテストベッドのセットアップに適用されます。

近日公開:

- BigAnimal は、Oracle 互換性が組み込まれたフルマネージドの Database-as-a-Service であり、クラウドアカウントで実行され、Postgres の専門家によって操作されます。BigAnimal を使用すると、データベースのセットアップ、管理、スケーリングが簡単になります。EDB Postgres Distributed を介した極度の高可用性サポートを追加すると、単一リージョンの Always On Gold クラスター、つまり、3 番目のアベイラビリティゾーンに監視ノードを持つ単一のクラウドリージョンの異なるアベイラビリティゾーンの 2 つの BDR グループが可能になります。

EDB Postgres Distributed for Kubernetes は、BDR レプリケーションを使用して、マルチマスターアーキテクチャを備えた高可用性 Postgres データベースクラスターのライフサイクル全体をカバーする EDB によって設計、開発、およびサポートされます。オープンソースの CloudNativePG オペレーターに基づいており、EDB Postgres Advanced Server を使用した Oracle や、IBM Power や OpenShift などの追加のサポートされているプラットフォームとの互換性などの追加の価値を提供します。

### 9.1 TPAexec

セルフホストとクラウド内の両方の物理マシンと仮想マシン (EC2) を含む自己管理設定で EDB Postgres Distributed を展開する標準的な方法は、TPAexec と呼ばれる EDB の展開ツールを使用することです。

TPAexec は、Ansible を使用して、さまざまなシナリオで Postgres を設定および操作する方法を文書化したリファレンスアーキテクチャのセットである Trusted Postgres Architecture (TPA) の指定に従って Postgres クラスターを構築するオーケストレーションツールです。TPA は、EDB が続くベストプラクティスを表しており、その推奨事項は、実稼働環境と同様に迅速なテストベッドセットアップに適用されます。

TPAexec パッケージは、アクティブな BDR サブスクリプションを持つ EDB のお客様ののみが利用できます。アクセスをリクエストするには、アカウントマネージャーにお問い合わせください。

### 9.1.1 インクリメンタルな変更

TPAexec は、プロビジョニング、展開、およびテストがべき等になるように注意深く設計されています。それらを実行し、`config.yml` に変更を加え、プロセスを再度実行して変更をデプロイできます。構成またはインスタンスで何も変更されていない場合、プロセス全体を再実行しても何も変更されません。

### 9.1.2 Ansible で拡張可能

TPAexec はさまざまな構成オプションをサポートしているため、`config.yml` を編集して `provision/deploy/test` を再実行するだけで多くのことができます。既に実装されているものをを超える必要がある場合は、フックスクリプトを作成して展開プロセスを拡張できます。

このメカニズムにより、展開のすべての段階で Ansible のすべての機能を自由に使用できます。たとえば、`hooks/pre-deploy.yml` のタスクはメインのデプロイの前に実行されます。また、展開後や他の多くのフックもあります。

### 9.1.3 クラスター管理

クラスターが稼働すると、TPAexec は、構成の変更、スイッチオーバー、ゼロダウンタイムのマイナーバージョンのアップグレードなど、便利なクラスター管理機能を提供します。これらの機能により、手動で変更を行うよりもクラスターを簡単かつ安全に管理できます。

### 9.1.4 ただの Postgres です

TPAexec は、多くの機能が構成された複雑なクラスターを作成できますが、結果は Postgres のみです。インストールは、生活をよりシンプルにするように設計されたいくつかの規則に従いますが、あなたとデータベースの間に隠された魔法や何かはありません。他の Postgres インストールで実行できるすべてのことを、TPA クラスターで実行できます。

### 9.1.5 はじめに

TPAexec をインストールし、それを使用して最初の EDB Postgres 分散クラスターをデプロイするには、次のトピックを参照してください。

- *Installing TPAexec*
- *Using TPAexec*
- Example: Deploying EDB Postgres Distributed

詳細については、*TPAexec* を参照してください。



## Using TPAexec

TPAexec を使用して、EDB Postgres 分散クラスターを構成、プロビジョニング、デプロイします。

### 構成する

`tpaexec configure` コマンドは、選択したオプションに基づいて、クラスターを記述する単純な YAML 構成ファイルを作成します。構成はすぐに使用する準備ができており、ニーズに合わせて変更できます。構成ファイルの編集は、クラスターの作成前と作成後の両方に構成を変更する通常の方法です。

構文は次のとおりです。

```
tpaexec configure <cluster_dir> --architecture <architecture_name> --layout <layout>
→[options]
```

必要な構成オプションは次のとおりです。

| Flags                                  | Description                                                                                                                                                                    |
|----------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>--architecture</code>            | 必須。EDB Postgres 分散展開の場合は <i>BDR-Always-ON</i> に設定します。                                                                                                                          |
| <code>--layout</code>                  | 必須。サポートされている 4 つのアーキテクチャーのいずれかを指定します。Bronze、Silver、Gold、および Platinum 詳細については、 <a href="#">:ref: `Choosing your architecture` &lt;Choosing your architecture&gt;</a> を参照してください。 |
| <code>--harp-consensus-protocol</code> | 必須。レイアウトがブロンズまたはシルバーの場合は、 <i>bdr</i> を使用します。レイアウトがゴールドまたはプラチナの場合は、 <i>etcd</i> を使用します。                                                                                         |

例：

```
[tpa]$ tpaexec configure ~/clusters/speedy \
    --architecture BDR-Always-ON \
    --layout gold \
    --harp-consensus-protocol etcd \
```

最初の引数は、`speedy` または `~/clusters/speedy` などのクラスターディレクトリである必要があります（クラスターの名前はどちらの場合も `speedy` です）。すべてのクラスターを `~/clusters` などの共通ディレクトリに保持することをお勧めします。アーキテクチャを選択するための次の引数は `--architecture` であり、その後 `--layout` が続きます。

このコマンドは、`~/clusters/speedy` という名前のディレクトリを作成し、`BDR-Always-ON` アーキテクチャと `gold` レイアウトのレイアウトに従う `config.yml` という名前の構成ファイルを作成します。`tpaexec info` コマンドを使用して、`configure` コマンドの実行時に指定した内容に基づいて、構成オプションでサポートされている値を確認できます。



## 一般的な構成オプション

その他の構成オプションは次のとおりです。

### 所有者

すべてのクラスターは、プロビジョニングされたリソースの責任者を直接追跡する必要があります。

デフォルトでは、クラスターは、`tpaexec provision` を実行しているユーザーのログイン名によって所有されているものとしてタグ付けされます。この名前で個人を識別できない場合（たとえば、`postgres`、`ec2-user`）、`--owner SomeId` を指定して識別可能な所有者を設定する必要があります。

イニシャル、「姓」、またはその他のあなたを一意に識別するものを使用できます。

### プラットフォームオプション

`--platform` のデフォルト値は `aws` です。BDR-Always-ON アーキテクチャでサポートされているプラットフォームです。

`--region` を指定して、アクセスできる（そして必要な数のインスタンスの作成を許可する）既存の AWS リージョンを指定します。デフォルトのリージョンは `eu-west-1` です。

AWS の有効なインスタンスタイプで `--instance-type` を指定します。デフォルトは `t3.micro` です。

### サブネット選択

デフォルトでは、各クラスターには 10.33/16 の下のランダムな/28 サブネットが割り当てられますが、アーキテクチャによっては 1 つ以上のサブネットがあり、各サブネットは/24 から/29 のどこにある場合もあります。

特定のサブネットを使用するには `--subnet` を指定します。たとえば、`--subnet 192.0.2.128/27`。

または、`--subnet-pattern` を指定して、指定されたパターンに一致するランダムサブネット（アーキテクチャに必要な数）を生成します。たとえば、`--subnet-pattern 192.0.x.x`。

### ディスク領域

`--root-volume-size` を指定して、ルートボリュームのサイズを GB 単位で設定します。たとえば、`--root-volume-size 64`。デフォルトは 16GB です。（インスタンスの作成に使用されるイメージによっては、ルートボリュームの最小サイズがある場合があります。）

個別の `postgres` と `barman` のボリュームをサポートするアーキテクチャの場合：

`--postgres-volume-size` を指定して、Postgres ボリュームのサイズを GB 単位で設定します。デフォルトは 16GB です。

Barman ボリュームのサイズを GB 単位で設定するには、`--barman-volume-size` を指定します。デフォルトは 32GB です。

## 配布

`--os` または `--distribution` を指定して、クラスタのインスタンスで使用する OS を指定します。値は大文字と小文字が区別されます。

選択したプラットフォームによって、使用可能なディストリビューションとデフォルトで使用するディストリビューションが決まります。詳細については、`tpaexec info platforms/<platformname>` を参照してください。

通常、「Debian」、「RedHat」、および「Ubuntu」を使用して、Postgres およびその他のソフトウェアがプリインストールされている TPA イメージを選択できます（展開時間を短縮します）。代わりに在庫分布画像を使用するには、値に「-minimal」を追加します（例、`--distribution Debian-minimal`）。

## 2ndQuadrant リポジトリ

デフォルトでは、TPAexec は 2ndQuadrant パブリックリポジトリをインストールし、アーキテクチャに必要な製品リポジトリを追加します。

`--2Q-repositories source/name/release ...` を指定して、2ndQuadrant パブリックリポジトリに加えて各インスタンスにインストールする 2ndQuadrant リポジトリの完全なリストを指定します。

これを行う場合、`tpaexec` を実行する前に `TPA_2Q_SUBSCRIPTION_TOKEN=xxx` をエクスポートする必要があります。

[EnterpriseDB customer portal](#) からサブスクリプショントークンを取得できます（

サポート>ソフトウェアサブスクリプション>追加）。

## ソフトウェアバージョン

デフォルトでは、TPAexec は Postgres の最新のメジャーバージョンを使用します。`--postgres-version` を指定して、サポートされている以前のメジャーバージョンをインストールします。

デフォルトでは、TPAexec は常にすべてのパッケージの最新バージョンをインストールします。通常、これは望ましい動作ですが、一部のテストシナリオでは、特定のパッケージバージョンを選択する必要があります。たとえば、

```
- --postgres-package-version 10.4-2.pgdg90+1
- --repmgr-package-version 4.0.5-1.pgdg90+1
- --barman-package-version 2.4-1.pgdg90+1
- --pglogical-package-version 2.2.0*
- --bdr-package-version 3.0.2*
- --pgbouncer-package-version 1.8*
```

`--extra-packages` または `--extra-postgres-packages` を指定して、追加のパッケージをインストールします。前者はシステムパッケージとともにインストールするパッケージをリストし、後者は postgres パッケージとともに後でインストールするパッケージをリストします。（前のリストで Postgres に依存するパッケージに言及した場合、Postgres はまだインストールされていないため、インストールは失敗します。）引数は変更なしでインストールのためにパッケージマネージャーに渡されます。

`--extra-optional-packages` オプションは `--extra-packages` と同様に動作しますが、指定されたパッケージをインストールできなくてもエラーではありません。

## ホスト名

デフォルトでは、`tpaexec configure` は、数十の名前の事前承認リストから必要な数のホスト名をランダムに選択します。ほとんどのクラスターではこれで十分です。

別のリストから名前を選択するには、`--hostnames-from` を指定します（たとえば、既定のリストで利用可能な名前よりも多くの名前が必要な場合）。ファイルには、1 行に 1 つのホスト名を含める必要があります。

`--hostnames-pattern` を指定して、ホスト名を `egrep-syntax` パターンに一致するものに制限します。これを行う場合、パターンが有効なホスト名（`[a-zA-Z0-9-]`）のみと一致し、十分な数を見つける必要があります。

## ロケーション

デフォルトでは、`tpaexec configure` は、選択したアーキテクチャで利用される場所に `first`、`second` などの名前を使用します。

`--location-names` を指定して、各場所に意味のある名前を付けます。

## Commit At Most Once

`--enable-camo` を指定して、各リージョンの BDR プライマリインスタンスのペアを互いの Commit At Most Once (CAMO) パートナーに設定します。詳細については、Commit At Most Once (CAMO) を参照してください。

## 規定

`tpaexec provision` コマンドは、クラスターに必要なインスタンスとその他のリソースを作成します。プロセスの詳細は、クラスターの構成時に選択したアーキテクチャ（例、BDR-Always-ON）とプラットフォーム（例、AWS）によって異なります。

たとえば、必要な権限で AWS にアクセスすると、TPAexec は EC2 インスタンス、VPC、サブネット、ルーティングテーブル、インターネットゲートウェイ、セキュリティグループ、EBS ボリューム、Elastic IPなどをプロビジョニングします。

「ベア」プラットフォームを選択し、接続の詳細を指定することにより、既存のサーバーを「プロビジョニング」することもできます。ベアメタルサーバーでも、クラウドプラットフォームで個別にプロビジョニングされたサーバーでも、まるで TPAexec で作成されたかのように使用できます。

単一のプラットフォームに制限されません。必要に応じて、クラスターを一部の AWS インスタンス（マルチプルリージョン）と一部のオンプレミスサーバー、または他のデータセンターのサーバーに分散できます。

プロビジョニング段階の最後に、基本的なオペレーティングシステムがインストールされた必要な数のインスタンスが作成されます。TPAexec は SSH 経由でアクセスできます（ルートに `sudo` を使用）。

### 展開する

`tpaexec deploy` コマンドは、プロビジョニングされたサーバー（TPAexec によって作成された場合とそうでない場合があります。ただし、SSH および `sudo` アクセスが利用できる限り、誰が作成したかは関係ありません）にインストールして構成します。これには、レプリケーション、バックアップなどが含まれます。

展開段階の最後に、EDB Postgres Distributed が稼働しています。

### テスト

`tpaexec test` コマンドは、デプロイされたクラスターに対してさまざまなアーキテクチャおよびプラットフォーム固有のテストを実行して、期待どおりに動作することを確認します。

テスト段階が完了すると、完全に機能するクラスターができます。

詳細については、[TPAexec](#) を参照してください。

## Deploying EDB Postgres Distributed

次の手順では、Amazon EC2 を使用して Always On Silver アーキテクチャで EDB Postgres Distributed をセットアップします。

### 1. 構成ファイルを生成します。edb\_notranlate\_0

これにより、TPAexec がクラスターの作成に使用する `config.yml` 構成ファイルを含む `myedbdpcluster` という現在の作業ディレクトリにサブディレクトリ `directory` が作成されます。サーバーに使用される IP アドレス範囲を変更したり、ノードの場所を調整したりする場合など、必要に応じて `config.yml` を編集します。

BDR 4（デフォルトバージョン）と 2ndQuadrant リポジトリを使用して指定するオプションを含めて、EDB Postgres Extended Server をインストールしました。デフォルトでは、TPAexec は 2ndQuadrant パブリックリポジトリをインストールし、アーキテクチャに必要な製品リポジトリを追加します。この例では、`--2Q-repositories` を指定したため、2ndQuadrant パブリックリポジトリに加えて、各インスタンスに 2ndQuadrant リポジトリの完全なリストがインストールされます。これを行ったため、プロビジョニング手順の前に、最初に `TPA_2Q_SUBSCRIPTION_TOKEN=xxx` をエクスポートする必要があります。

[EnterpriseDB customer portal](#) からサブスクリプショントークンを取得できます（

サポート>ソフトウェアサブスクリプション>追加）。

```
cd ~/clusters/myedbdpcluster

export TPA_2Q_SUBSCRIPTION_TOKEN=<token>
```

1. クラスターをプロビジョニングします。edb\_notranlate\_2 プラットフォーム（デフォルトのプラットフォーム）として AWS を指定したため、TCAexec は EC2 インスタンス、VPC、サブネット、ルーティングテーブル、インターネットゲートウェイ、セキュリティグループ、EBS ボリューム、Elastic IPなどをプロビジョニングします。

1. 必要なパッケージを展開し、実際の EDB Postgres 分散クラスターをセットアップします。

`deploy` コマンドが正常に実行されると、クラスターを使用できます。`psql` または他のデータベースクライアントを介して接続できます。

クラスターが期待どおりに実行されていることを確認するテストを実行することもできます。



## 第10章 Upgrading

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するコンポーネントによって部分的に異なります。

一般に、ノードを1つずつアップグレードし、アプリケーション接続を既にアップグレードされているノードに切り替える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することにより、ダウンタイムをほぼゼロでクラスターをアップグレードできます。

標準の PostgreSQL セットアップと同様に、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から、クラスター全体を再起動することもできます。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、ソフトウェアのバージョンが混在した状態での実行を回避するため、最も簡単ですが、明らかにダウンタイムが発生し、何らかの理由でローリングアップグレードが不可能な場合を除きお勧めしません。

EDB Postgres 分散クラスターをアップグレードするには、次の手順を実行します。

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。
1. サーバーソフトウェアをアップグレードします。
  1. アップグレードを確認して検証します。

### 10.1 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、大きく分けて2つの方法があります。

**\*新しいソフトウェアバージョンへのノードのインプレースアップグレードについては、[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ているものに置き換える。  
[ノード結合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式で実行できます。

### 10.1.1 ローリングアップグレードの考慮事項

クラスターのローリングアップグレード中に、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、nodeA には BDR 3.7.16 がありますが、nodeB と nodeC には 4.1.0 があります。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（この例の場合は 3.7.16）の Protokol と機能を使用するため、Protokol の変更を必要とする新しいバージョンが提供する新機能は無効になります。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、新機能が自動的に有効になります。

同様に、WAL デコーダーが有効なノードを備えたクラスターがローリングアップグレードを行っている場合、より高いバージョンの BDR ノードの WAL デコーダーはより高い pglogical バージョンの LCR を生成し、より低いバージョンの BDR ノードの WAL デコーダーはより低い pglogical バージョンの LCR を生成します。その結果、より高いバージョンの BDR ノードの WAL 送信者は、Protokol バージョンの不一致のために LCR を使用すると予想されませんが、より低いバージョンの BDR ノードでは、WAL 送信者は LCR を使用し続ける可能性があります。すべての BDR ノードが同じ BDR 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、すべてのノードが以前のリリースであるクラスターから始まり、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ます。コンポーネントの 3 つ以上のバージョンを同時に実行しないでください。つまり、すべてのノードで以前のアップグレードプロセスが完全に完了するまで、新しいアップグレードを開始しないでください。

ソフトウェアの混合バージョンを無期限に実行することはお勧めできませんが、ビジネスリスクを軽減するために注意が必要であるとユーザーが判断した場合、アップグレードプロセスには時間がかかる場合があります。

ローリングアップグレードはソフトウェアのメジャーバージョンのアップグレードに使用できますが、1 つのクラスターに PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server を混在させることはサポートされていません。

**警告:** EDB Postgres Distributed のダウングレードは\*サポートされていない\*ため、クラスターの手動リビルドが必要です。

### 10.1.2 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、クラスターの残りの部分の動作を維持しながら、クラスター内の各ノードで *Server Software Upgrade* プロセスが実行されるプロセスです。

実際の手順は、Postgres コンポーネントを新しいメジャーバージョンにアップグレードす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切り替えて、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。



### 10.1.3 ノード結合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 1 つを削除することです。

このアプローチの場合、手順は常に同じですが、ノードジョインが含まれるため、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 のより新しい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する必要があります。これは、Postgres の新しいリリースに追加された可能性のある新しい DDL 構文に特に当てはまります。

---

**注釈:** `bdr_init_physical` はソースノードをバイトごとに作成するため、Postgres のメジャー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は使用できません。実際、現在 `bdr_init_physical` では、ソースの BDR バージョンと参加ノードがまったく同じである必要があります。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、論理結合を使用する必要があります。

---

### 10.1.4 CAMO 対応クラスターのアップグレード

CAMO 保護では、CAMO ペアのノードの少なくとも 1 つが動作している必要があります。アップグレードの場合、CAMO で保護されたトランザクションがアップグレードと同時に実行されないようにするか、ローリングアップグレード戦略を使用して、ノードにアップグレードとアップグレードによる対応するノードのダウンタイムを調整するための十分な時間を与えることをお勧めします。

## 10.2 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 BDR 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検討し、場合によっては調整することをお勧めします。

*Release Notes* に記載されている個々の変更と、このトピックのバージョン固有のアップグレードノートを参照してください。

## 10.3 サーバーソフトウェアのアップグレード

個々のノードでの EDB Postgres Distributed のアップグレードはインプレースで行われます。BDR 拡張機能をアップグレードする際のバックアップと復元は必要ありません。

### 10.3.1 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、いくつかの簡単な手順で構成されています。

### 10.3.2 Postgres を停止する

バイナリパッケージのアップグレード中は、通常、実行中の Postgres サーバーを停止して、アップグレード中に予期しない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いようにすることをお勧めします。

### 10.3.3 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、BDR パッケージの新しいバージョンをインストールすることです。これにより、新しいバイナリと拡張機能 SQL スクリプトの両方がインストールされます。この手順はオペレーティングシステム固有です。

### 10.3.4 Postgres を起動

パッケージがアップグレードされると、Postgres インスタンスを起動できます。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、BDR 拡張機能は起動時に自動的にアップグレードされます。

### 10.3.5 Postgres のアップグレード

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どうか大きく依存します。

### 10.3.6 マイナーバージョンの 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 *upgrading the BDR extension* に似ています。通常、必要なのは Postgres を停止し、パッケージをアップグレードして、Postgres を再起動することだけです。

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、再インデックスなどの追加の手順が推奨される場合があります。アップグレードする Postgres の特定のバージョンのリリースノートを参照してください。

### 10.3.7 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、より複雑なプロセスです。

EDB Postgres Distributed は、`bdr_pg_upgrade` コマンドラインユーティリティを提供します。これを使用して *In-place Postgres Major Version Upgrades* を実行できます。

---

**注釈:** Postgres、BDR 拡張機能など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、アプリケーションと特定のソフトウェアのターゲットバージョンとの互換性を保証することが常に重要です。

---

## 10.4 アップグレードのチェックと検証

この手順の後、BDR ノードがアップグレードされます。次のように BDR4 バイナリの現在のバージョンを確認できます。

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

ノードのアップグレード後は常に *Monitoring* をチェックして、アップグレードされたノードが期待どおりに動作していることを確認します。

## 10.5 アプリケーションスキーマのアップグレード

BDR 自分自身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、アップグレードされたアプリケーションを再起動して新しいスキーマバリエーションを使用することです。繰り返しますが、これによりダウンタイムが発生します。

このダウンタイムを解消するために、BDR はアプリケーションスキーマのローリングアップグレードを実行する方法を提供します。

### 10.5.1 アプリケーションスキーマのローリングアップグレード

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。これは、

*DML および DDL レプリケーション*

で説明されているように、手動で制御できます。これを使用して、ノード間のデータベーススキーマ間の違いを作成できます。BDR は、ノード間にわずかな違いがある場合で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行えるように設計されています。

**警告:** 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は、BDR の外部で管理する必要があります。実稼働クラスターでこれを正しく動作させるには、慎重なスクリプトが必要です。大規模なテストをお勧めします。

詳細は [違いのあるノード間の複製](#) 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。この観点から、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` の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。これは、ノードに追加のテーブルを追加する前に実行する必要があります、永続的な設定を目的としています。

これは、`node1` を実際のノード名に置き換えた後、**各ノードで個別に実行** する必要があります。

```
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は追加の列を処理する必要があります。この観点から、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` の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。これは、1 つのノードに列を追加する前に実行する必要があります。永続的な設定を目的としています。

これは、`node1` を実際のノード名に置き換えた後、**各ノードで個別に実行** する必要があります。

```
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は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` の競合が発生します。したがって、NULL を受け入れず、DEFAULT 値を持たない列には、2 つのステップ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に DEFAULT 値を追加します。
2. カラムを取り外します。

制約はローリングで削除できます。現在、テーブル制約の追加を一度に 1 ノードずつ処理する方法はサポートされていません。

1 つのノードが既存の列の型を変更する DDL を実行する場合、現在の型とターゲットの型間のバイナリ強制性の存在によっては、操作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、基になる列タイプのメタデータの更新のみになります。通常、テーブルの書き換えは制限されています。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型のローリングアップグレードを 1 つずつ採用することにより、列の型を自動的にキャスト可能なものに変更できます。詳細については、[ALTER テーブル](#) を参照してください。セクション。

## In-place Postgres Major Version Upgrades

コマンドラインユーティリティ `bdr_pg_upgrade` を使用して、BDR ノードを Postgres の新しいメジャーバージョンにアップグレードできます。

`bdr_pg_upgrade` は、BDR 固有のロジックで標準の `bdr_pg_upgrade コマンドライン` を内部的に使用して、スムーズなアップグレードを保証します。

### 用語

このドキュメントでは、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ためにさまざまな用語が使用されています。

- `old cluster*` - データの移行元である、アップグレードする既存の Postgres クラスターノード。
- `new cluster*` - データの移行先の新しい Postgres クラスター。このクラスターノードは、古いクラスターよりも 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

`bdr_pg_upgrade コマンドライン` のすべての要件を両方のクラスターが満たす必要があるという事実を含め、標準の Postgres メジャーバージョンアップグレードの予防措置が適用されます。

さらに、レプリケーションスロットとレプリケーションオリジンを使用する他のツールがある場合は、`bdr_pg_upgrade` を使用しないでください。アップグレード後に BDR スロットとオリジンのみが復元されます。

`bdr_pg_upgrade` には、満たす必要があるいくつかの前提条件があります。

- 古いクラスターを使用しているアプリケーションは切断されました。たとえば、クラスター内の別のノードにリダイレクトできます
- ピア認証は両方のクラスターに構成されています。`bdr_pg_upgrade` はピア認証が必要です
- 両方のクラスターの BDR バージョンは完全に同じであり、バージョン 4.1.0 以上である必要があります
- 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
- BDR パッケージ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
- 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が新しいクラスターに存在してはなりません

また、シャットダウンすると CLI が古いクラスターを起動するため、`bdr_pg_upgrade` を実行する前に古いクラスターを起動することをお勧めします。

## 使用法

Postgres 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をインストールする必要があります。

## bdr\_pg\_upgrade コマンドライン

bdr\_pg\_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、*bdr\_pg\_upgrade コマンドライン*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

`bdr_pg_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 bdr\_pg\_upgrade に渡すことができます。

- -b, --old-bindir - 古いクラスター bin ディレクトリ (必須)
- -B, --new-bindir - 新しいクラスター bin ディレクトリ (必須)
- -d, --old-datadir - 古いクラスターデータディレクトリ (必須)
- -D, --new-datadir - REQUIRED 新しいクラスターデータディレクトリ (必須)
- --database - BDR データベース名 (必須)
- -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 postmaster 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でください

## 環境変数

コマンドラインパラメーターの代わりに環境変数を使用できます。

- PGBINOLD - 古いクラスター bin ディレクトリ
- PGBINNEW - 新しいクラスター bin ディレクトリ
- PGDATAOLD - 古いクラスターデータディレクトリ
- PGDATANEW - 新しいクラスターデータディレクトリ
- PGPORTOLD - 古いクラスターポート番号
- PGSOCKETDIR - アップグレード中に postmaster ソケットに使用するディレクトリ

## 例

次のシナリオがあるとします。

- 古いクラスタの bin ディレクトリは `/usr/lib/postgresql/13/bin`
- 新しいクラスター bin ディレクトリは `/usr/lib/postgresql/14/bin` です
- 旧クラスタデータディレクトリは `/var/lib/postgresql/13/main`
- 新しいクラスタデータディレクトリは `/var/lib/postgresql/14/main` です
- データベース名は `bdrdb`

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。

```
bdr_pg_upgrade \
- --old-bindir /usr/lib/postgresql/13/bin \
- --new-bindir /usr/lib/postgresql/14/bin \
- --old-datadir /var/lib/postgresql/13/main \
- --new-datadir /var/lib/postgresql/14/main \
- --database bdrdb
```

## 実行される手順

`bdr_pg_upgrade` の実行時に実行される手順。

**注釈:** `bdr_pg_upgrade` への引数として `--check` が指定されている場合、CLI はデータベースを変更するステップを 1 スキップします。

## BDR Postgres チェック

| Steps                                           | <code>--check</code> supplied |
|-------------------------------------------------|-------------------------------|
| Collecting pre-upgrade new cluster control data | <i>run</i>                    |
| Checking new cluster state is shutdown          | <i>run</i>                    |
| Checking BDR versions                           | <i>run</i>                    |
| Starting old cluster (if shutdown)              | <i>skip</i>                   |
| Connecting to old cluster                       | <i>skip</i>                   |
| Checking if bdr schema exists                   | <i>skip</i>                   |

次のページに続く

表 1 – 前のページからの続き

| Steps                                  | --check supplied |
|----------------------------------------|------------------|
| Turning DDL replication off            | <i>skip</i>      |
| Terminating connections to database.   | <i>skip</i>      |
| Disabling connections to database      | <i>skip</i>      |
| Waiting for all slots to be flushed    | <i>skip</i>      |
| Disconnecting from old cluster         | <i>skip</i>      |
| Stopping old cluster                   | <i>skip</i>      |
| Starting old cluster with BDR disabled | <i>skip</i>      |
| Connecting to old cluster              | <i>skip</i>      |
| Collecting replication origins         | <i>skip</i>      |
| Collecting replication slots           | <i>skip</i>      |
| Disconnecting from old cluster         | <i>skip</i>      |
| Stopping old cluster                   | <i>skip</i>      |

### pg\_upgrade の手順

標準の pg\_upgrade ステップが実行されます

**注釈:** --check が提供されている場合は pg\_upgrade に渡されます

### BDR のアップグレード後の手順

| Steps                                  | --check supplied |
|----------------------------------------|------------------|
| Collecting old cluster control data    | <i>skip</i>      |
| Collecting new cluster control data    | <i>skip</i>      |
| Advancing LSN of new cluster           | <i>skip</i>      |
| Starting new cluster with BDR disabled | <i>skip</i>      |

次のページに続く



表 2 – 前のページからの続き

| Steps                                                    | --check supplied |
|----------------------------------------------------------|------------------|
| Connecting to new cluster                                | <i>skip</i>      |
| Creating replication origin<br>Repeated for each origin  | <i>skip</i>      |
| Advancing replication origin<br>Repeated for each origin | <i>skip</i>      |
| Creating replication slot<br>Repeated for each slot      | <i>skip</i>      |
| Stopping new cluster                                     | <i>skip</i>      |

### Supported BDR upgrade paths

| 3.7.13.1 | 3.7.14 | 3.7.15 | 3.7.16 | 4.0.0 and later | Target BDR version |
|----------|--------|--------|--------|-----------------|--------------------|
|          |        | X      | X      | X               | 4.2.0              |
|          |        | X      | X      | X               | 4.1.1              |
|          |        | X      | X      | X               | 4.1.0              |
|          |        | X      |        | X               | 4.0.2              |
|          | X      |        |        | X               | 4.0.1              |
| X        |        |        |        |                 | 4.0.0              |



## 第11章 Compatibility matrix

| BDR | PGD CLI | HARP | PostgreSQL | EDB Postgres Extended | EDB Postgres Advanced |
|-----|---------|------|------------|-----------------------|-----------------------|
| 4   | 1       | 2    | 12-14      | 12-14                 | 12-14                 |
| 3.7 |         | 2    | 11-13      | 11-13                 | 11-13                 |
| 3.6 |         |      | 10-11      | 11                    |                       |

- PGD CLI 1 は BDR 4.1 以降でサポートされています。
- HARP 2 の BDR DCS は、BDR 3.7.15 以降および 4.0.1 以降でサポートされています。



## 第12章 Backup and recovery

この章では、EDB Postgres 分散クラスターのバックアップと復元について説明します。

BDR は、分散型の高可用性システムとして設計されています。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。

BDR のバックアップとリカバリの役割は、次のような状況で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
- データ破損、アプリケーションエラー、またはセキュリティ違反の結果としての、複数のノードにわたる重大な訂正不能なデータ破損

### 12.1 バックアップ

#### 12.1.1 pg\_dump

pg\_dump は、「論理バックアップ」とも呼ばれ、BDR で正常に使用できます。

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することに注意してください。これは意図的なもので、BDR 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の種類のメタデータが失われることを意味するため、復元すると、復元時にすべてのシーケンスの種類が効果的にリセットされます。

各シーケンスの正確なシーケンスの種類をリセットする復元後のスクリプトを作成するには、次のような SQL スクリプトを使用できます。

```
SELECT SELECT bdr.alter_sequence_set_kind(||
    nspname||.||relname||,||seqkind||);
FROM bdr.sequences
WHERE seqkind != local;
```

pg\_dump が bdr.crdt\_raw\_value = on を使用して実行されている場合、ダンプは bdr.crdt\_raw\_value = on でのみリロードできることに注意してください。

テクニカルサポートは、BDR のバックアップと復元に物理バックアップ技術を使用することをお勧めします。

### 12.1.2 物理バックアップ

EDB Postgres 分散クラスター内のノードの物理バックアップは、[Barman](#) などの標準の PostgreSQL ソフトウェアを使用して取得できます。

BDR ノードの物理バックアップは、PostgreSQL ノードに適用されるのと同じ手順で実行できます BDRBDR 機能を実行する PostgreSQL ノードです。

PostgreSQL のバックアップ技術を BDR に適用する際に考慮しなければならない特定のポイントがいくつかあります。

- BDR 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。簡単にバックアップと復元ができるようにデータベースを計画する必要があります。
- 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。最も単純な場合、すべてのノードにすべてのデータのコピーがあるため、1 つのノードのみをバックアップしてすべてのデータをキャプチャ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、BDR の目標は達成されないため、最小はサイトごとに少なくとも 1 つのノードのバックアップにする必要があります（明らかに多数のコピーなど）。
- ただし、各ノードには複製されていないローカルデータがある場合、および/または複製セットの定義が複雑なため、すべてのノードがすべての複製セットにサブスクライブしない場合があります。このような場合、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をバックアップする方法と、各レプリケーションセットにサブスクライブするノードのバックアップも含める必要があります。

### 12.1.3 最終的な一貫性

EDB Postgres 分散クラスター内のノードは**最終的に一貫性がありますが**、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（以下の [例] を参照）。

次の例は、同じ EDB Postgres 分散クラスター内の 2 つのノードが同じ一連の状態を経ない場合がある（そして通常は経ない場合がある）方法を示しています。

2 つのノード N1 と N2 を持つクラスターを考えます。最初は S 状態です。トランザクション W1 がノード N1 に適用され、同時に非競合のトランザクション W2 がノード N2 に適用された場合、ノード N1 は次の状態になります。

|      |   |     |        |     |             |
|------|---|-----|--------|-----|-------------|
| (N1) | S | --> | S + W1 | --> | S + W1 + W2 |
|------|---|-----|--------|-----|-------------|

…一方、ノード N2 は次の状態になります。

|      |   |     |        |     |             |
|------|---|-----|--------|-----|-------------|
| (N2) | S | --> | S + W2 | --> | S + W1 + W2 |
|------|---|-----|--------|-----|-------------|

つまり、ノード N1 は状態  $S + W2$  を**決して**想定し、ノード N2 も同様に状態  $S + W1$  を想定しませんが、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、バックアップ戦略を決定する方法に影響を与える可能性があります。

### 12.1.4 ポイントインタイムリカバリー (PITR)

上記の例では、W1 と W2 は両方とも T1 に発生しますが、W1 の変更は T2 まで適用されません。

PostgreSQL PITR は、単一のマスターから COMMIT オーダーで変更が届くことを前提に設計されています。したがって、PITR は、1 つの特定の時点 (PIT) に到達するまで変更をスキャンするだけで可能です。このスキームを使用すると、1 つのノードをその観点から単一のポイントインタイムに復元できます。T1 ですが、その状態には、その時間近くにコミットされたがまだノードに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、リカバリーは部分的に一貫性がない、または少なくとも 1 つの複製起点についてのみ一貫していると考えられる場合があります。

これを要求するには、標準の構文を使用します。

```
recovery_target_time = T1
```

BDR は、複数のマスターからの変更を許可します。すべてが 1 つのノードの WAL ログ内に記録され、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。

BDR は、すべてまたは一部のレプリケーション起点の PITR を特定の時点に許可し、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視点を提供します。

したがって、マルチオリジンの場合、WAL ストリームは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ていると見なします。PIT はまだ 1 つだけですが、それはオリジンごとに異なるポイントとして到達されます。

要求されたオリジンが PIT を見つけるまで WAL ストリームを読み取ります。オリジンの PIT に到達した後、オリジンのトランザクションレコードをコミット済みとしてマークしないことを除き、その時点までのすべての変更を適用します。

WAL では 1 つの LSN 「ストップポイント」になりますが、「単一オリジン 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 に達したら、必要に応じて、リカバリを続行できるように後の PIT を設定することもできます。

目的の停止ポイントに到達した後、回復したサーバーを昇格させる場合は、最初にシャットダウンし、`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ているよりも高い LSN 値に移動します。これにより、論理デコードによって生成された LSN が重複しないことが保証されます。

上記の特定の例では、N1 は T1 に復元されますが、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、`recovery.conf` ファイルで標準の構文を使用します。

```
recovery_target_time = T1
```

T1 に復元されるレプリケーション起点のリストは、新しいパラメーター `recovery_target_origins` を使用して、別の `multi_recovery.conf` ファイルで指定する必要があります。

```
recovery_target_origins = *
```

…または、元のサブセットを `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は常に暗黙的に実行されることに注意してください。`recovery_target_origins` で指定されていないオリジンの場合、`recovery_target_origins` に記載されているリストのターゲットが達成される時期に応じて、回復は任意の時点で停止する場合があります。

`multi_recovery.conf` ファイルが存在しない場合、リカバリはデフォルトで元の PostgreSQL PITR の動作になります。これは、変更が単一のマスターから COMMIT オーダーで到着することを想定して設計されています。

---

**注釈:** この機能は EDB Postgres Extended でのみ利用でき、Barman は現在 `multi_recovery.conf` ファイルを自動的に作成しません。

---

## 12.2 復元する

標準の PostgreSQL ノードと同じ手順で物理バックアップを作成できますが、少し複雑なのは BDR ノードの物理バックアップを\*\*復元することです。

### 12.2.1 EDB Postgres 分散クラスターの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 BDR ノードの障害または交換が含まれます。

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、BDR 機能は、単一の BDR ノード、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復元できます。

- まだいくつかの BDR ノードが稼働している場合は、BDR ノードを復元したホストをフェンスオフして、残っている BDR ノードに接続できないようにします。これにより、新しいノードが既存のクラスターを混同しないようにします。
- BDR 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た WAL アーカイブがある場合は、適切な `recovery.conf` を作成し、PostgreSQL をリカバリで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、スタンバイリカバリの場合は読み取り/書き込みにプロモートします。生き残ったノードからフェンスしたままにしてください！
- 以下に説明するように、物理バックアップに含まれる残りの BDR 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()` ファンクションコールに基づいた標準の手順で、さらに BDR ノードを追加します。

### 12.2.2 BDR メタデータのクリーンアップ

残りの BDR メタデータのクリーニングは、次のように実行されます。

1. `bdr.drop_node` を使用して BDR ノードを削除します
2. PostgreSQL を完全に停止してから再起動します（重要！）。

### 12.2.3 レプリケーションオリジンのクリーンアップ

レプリケーション元はシステムカタログに永続的に記録され、バックアップと復元されたインスタンスに含まれるため、別の手順で明示的に削除する必要があります。これらは依存関係として明示的に記録されないため、BDR 拡張機能を削除しても自動的に削除されません。

BDR 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て、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します。したがって、次を実行する必要があります。

```
SELECT pg_replication_origin_drop(bdr_dbname_grpname_nodename);
```

…（前の）クラスター内のノードごとに 1 回。レプリケーションオリジンは次のようにリストできます。

```
SELECT * FROM pg_replication_origin;
```

…そして、BDR によって作成されたものは、上の例のように名前ですべてに認識されます。

### 12.2.4 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。

他の一部のバックアップ方法では、レプリケーションスロットが古いまたは無効な状態である可能性があります。バックアップを復元したら、次のようにします。

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

…**すべての**レプリケーションスロットをドロップします。一部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、これが役立つことはほとんどありません。

**警告:** ライブ BDR ノードでこれを実行しないでください。



## 第13章 Monitoring

レプリケーション設定を監視することは、システムが最適に実行され、ディスク領域が不足したり、操作を停止する可能性のあるその他の障害が発生したりしないように重要です。

自動モニタリングを導入して、たとえば、レプリケーションスロットがひどく遅れ始めた場合に、管理者にアラートを送信し、プロアクティブなアクションを取れるようにすることが重要です。

EDB は、バージョン 8.1 から BDR をサポートする Postgres Enterprise Manager (PEM) を提供します。または、ツールまたはユーザーは、以下で説明する機能を使用して BDR に独自の呼び出しを行うことができます。

### 13.1 モニタリングの概要

BDR グループは、多くの場合ノードと呼ばれる複数のサーバーで構成されています。グループ全体の状態を確認するには、すべてのノードを監視する必要があります。

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 BDR の状態を評価できます。

- OK - 多くの場合、緑色で示されます
- WARNING - 黄色で表示されることが多い
- CRITICAL - 多くの場合、赤で示されます
- および UNKNOWN - 認識されない状況の場合、多くの場合赤で示されます

BDR は、さまざまな内部メトリックの瞬時の状態を表示する動的カタログビューと、ユーザーが要求した構成のデフォルトおよび/または構成の変更を保存する BDR 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、より高いセキュリティ要件を持つユーザーまたは内部情報が含まれているものもあります。

BDR を使用すると、各ノードを個別に監視したり、単一のノードにアクセスして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したい場合は、各ノードに接続して監視要求を発行するだけです。単一のノードからグループを監視する場合は、bdr.group で始まるビューを使用します。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組み立てるためです。

bdr\_superuser から bdr.run\_on\_all\_nodes() ファンクションへのアクセスが許可されている場合は、すべてのノードを独自に呼び出すことができます。

## 13.2 モニタリングノードの参加と削除

デフォルトでは、ノード管理機能は参加または部分操作が完了するまで待機します。これは、それぞれの `wait_for_completion` 関数引数を使用してオフにできます。待機がオフになっている場合、結合操作または部分操作がいつ終了するかを確認するには、`bdr.node_summary` および `bdr.state_journal_details` を介して間接的にノードの状態を確認します。

呼び出されると、ヘルパー関数 `bdr.wait_for_join_completion()` は、すべての未処理のノード参加操作が完了するまで PostgreSQL セッションを一時停止します。

次に、2つのノードがアクティブで、別のノードが参加していることを示す `bdr.node_summary` からの SELECT クエリの出力例を示します。

```
# 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                                  |
| node_seq_id         | 3                                        |
| node_local_dbname   | postgres                                 |

また、テーブル `:ref:`bdr.node_catchup_info`<bdr.node_catchup_info>` は、ノードの参加または分離に関連す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ノードからすべてのデータを受信しなかった可能性があります。そのため、既にそのデータを受信して転送できるノードから一時的なスロットを作成します。

`catchup_state` は次のいずれかです。

```

10 = setup
20 = start
30 = catchup
40 = done

```

## 13.3 レプリケーションピアの監視

レプリケーションアクティビティの監視には、主に2つのビューが使用されます。

- 発信レプリケーションを監視するための `:ref:`bdr.node_slots` <bdr.node_slots>`
- 着信レプリケーションを監視するための `:ref:`bdr.subscription_summary` <bdr.subscription_summary>`

`bdr.node_slots` によって提供される情報のほとんどは、標準の PostgreSQL レプリケーション監視ビュー

`pg_catalog.pg_stat_replication` および `pg_catalog.pg_replication_slots` を照会することでも取得できます。

各ノードには1つの BDR グループスロットがあります。これは接続されるべきではなく、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。

[BDR によって作成されたレプリケーションスロット](#) を参照してください。

### 13.3.1 送信レプリケーションのモニタリング

送信レプリケーションアクティビティの監視に使用される追加のビューがあります。

- 発信レプリケーションを監視するための `:ref:`bdr.node_replication_rates` <bdr.node_replication_rates>`

`bdr.node_replication_rates` ビューは、特にピアノードのキャッチアップ推定とともに、発信レプリケーションアクティビティの全体像を提供します。

```
# 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

```

(次のページに続く)

(前のページからの続き)

|                  |            |
|------------------|------------|
| 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   |

上記の `apply_rate` は、1 秒あたりのバイト数を指します。これは、ピアがローカルノードからのデータを消費しているレートです。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。この情報の修正に取り組んでいます。回避策として、ピアノードがローカルノードのデータに追いつくため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。以下で説明するように、他のフィールドも `bdr.node_slots` ビューを介して利用できます。

---

**注釈:** このカタログは、`bdr-enterprise` 拡張機能がインストールされている場合にのみ存在します。

---

管理者は、ローカルノードからの発信レプリケーションについて `bdr.node_slots` に照会できます。現在のノードで認識さ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 BDR 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
```

(次のページに続く)

(前のページからの続き)

|                  |                             |
|------------------|-----------------------------|
| 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                       |

BDR はメッシュネットワークであるため、クラスター内のラグを完全に把握するには、このクエリを参加しているすべてのノードで実行する必要があることに注意してください。

`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された最後の位置である `replay_lsn` との WAL 位置の差を報告します。`replay_lag_size` は、人間が判読できる形式です。WAL には通常、複製されないがまだ `replay_lag_bytes` でカウントされる多くの書き込みが含まれていることを理解することが重要です。セットなど。したがって、ここで報告されるバイト単位のラグは、ピアノードを最新の状態にするためにネットワークで複製する必要があるデータの量ではなく、処理が必要なサーバーサイド 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かかる時間、または `bdr.node_slots` が照会されたときの現在の位置から書き込み位置まで再生するのにかかる時間の尺度ではありません。ピアが最新のコミットを確認してから現在の実時間までの遅延を測定します。ノードが再接続した直後にこの列がゼロに設定されるため、`replay_lag_bytes` と `bdr.node_replication_rates` の `replay_lag_size` または `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ラグは進行しません。コミットがレプリケートされる場合にのみ変更されます。したがって、ラグは「のこぎり波」になる傾向があり、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、および確認を送信すると再び低下します。報告された LSN 位置は、同様の理由でスムーズに進むのではなく「階段状」になります。

レプリケーションが切断されると (`active = 'f'`)、`active_pid` 列は NULL になり、`client_addr` およびアクティブな接続でのみ意味を持つ他のフィールドも同様です。`state` フィールドは `'disconnected'` になります。`_lsn` フィールドは `confirmed_flush_lsn` と同じになります。これは、クライアントが再生および保存した最後の位置であるためです。`_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と現在時間の間の経過時間を表示し、`_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 挿入位置との間の距離を報告します。

注: `restart_lsn` が他の `lsn` 列の後ろにあるのは正常です。これは、レプリケーションまたはピアノードの遅延の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 を読み取っていないからではない位置であり、一般的にまだ複製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開を遅くし、必要以上の WAL を強制的に保持する可能性があります。それ以外は無害です。心配な場合は、実行時間が非常に長いトランザクションと、忘れられた準備されたトランザクションを探してください。

### 13.3.2 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) を監視できます。これは、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                 |

### 13.3.3 LCR を使用した WAL 送信者の監視

**デコードワーカー** が有効になっている場合、各 WAL 送信者の現在の LCR (Logical Change Record

) ファイルに関する情報は、関数 `bdr.wal_sender_stats` を介して監視できます。

```
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 | 000000000000000000000000000000001400000000000000000 |
| 2059916 | t            | bdr_postgres_bdrgroup_decoder | 000000000000000000000000000000001400000000000000000 |

(3 rows)

`is_using_lcr` が `FALSE` の場合、`decoder_slot_name` / `lcr_file_name` は `NULL` になります。これは、Decoding Worker が有効になっていない場合、または WAL 送信者が **ロジカルスタンバイノード**を提供している場合です。



さらに、Decoding Worker に関する情報は、ファンクション `bdr.get_decoding_worker_stat` を介して監視できます。

```
postgres=# SELECT * FROM bdr.get_decoding_worker_stat();
 pid | decoded_upto_lsn | waiting | waiting_for_lsn
-----+-----+-----+-----
 1153091 | 0/1E5EEE8      | t       | 0/1E5EF00
(1 row)
```

## 13.4 BDR レプリケーションワーカーの監視

すべての BDR ワーカーは、`pg_stat_activity` と同じ列と情報コンテンツを持つシステムビュー `bdr.stat_activity` に表示されます。したがって、このビューは、BDR 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 BDR に関連する場合、`wait_event` 列に拡張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を除き、BDR ワーカーでは `query` 列は空白になります

`bdr.workers` ビューには、`bdr.stat_activity` からは利用できない BDR ワーカー固有の詳細が表示されます。

ビュー `bdr.worker_errors` 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（ある場合）を表示します。これは永続的な情報であるため、エラーの存在だけでなく、エラーの時刻をメモすることが重要です。ほとんどのエラーはその性質上一時的なものであり、BDR ワーカーは失敗した操作を再試行します。

## 13.5 BDR ライターの監視

ライタのアクティビティを監視するための別のシステムビュー `bdr.writers` があります。このビューには、ライタワーカーのみの現在のステータスが表示されます。それには以下が含まれます：

- ライタが属するサブスクリプションを識別する `sub_name`
- ライタプロセスの `pid`
- ライタが進行中のストリーミングトランザクションのアプリケーションをサポートしているかどうかを知るための `streaming_allowed`
- 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の `is_streaming`
- `commit_queue_position` は、コミットキュー内のライタの位置を確認します。

BDR 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。並列ライタの場合、複数のライタが同時に異なるトランザクションを適用している可能性があります。`commit_queue_position` は、コミットする順序を示しています。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わかっていないことを意味します。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ときに発生する可能性があります。

## 13.6 グローバルロックの監視

現在 DDL レプリケーションにのみ使用されているグローバルロックは、BDR 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、データベース内の永続的な（一時ではない）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対して一方または両方のエントリタイプが作成されます。

ローカルノードで保持されているグローバルロックは、*the* `'bdr.global_locks'` view [pgd/latest/overview/bdr/catalogs#bdrglobal\\_locks](https://pgd/latest/overview/bdr/catalogs#bdrglobal_locks) で可視です。このビューは、ロックの種類を示しています。リレーションロックの場合、ロックされているリレーション、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ているかどうかを示します。グローバルな警告ロックの場合、`lock_type` 列には `GLOBAL_LOCK_ADVISORY` が表示され、`relation` 列にはロックが取得された警告キーが表示されます。

以下は、`bdr.ddl_locking = on` で `ALTER TABLE` ステートメントを実行している間の `bdr.global_locks` の出力例です。

```
# 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
```

ロックの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、カタログのドキュメントを参照してください。

## 13.7 競合の監視

レプリケーション **競合の監視** は、複数のノードが互いに相互作用できる方法で同じ行に影響を与える変更を行った場合に発生する可能性があります。BDR システムを監視して、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解消するか、頻度を下げる必要があります。

デフォルトでは、すべての競合は `bdr.conflict_history` に記録されます。これ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によって保護され、レプリケートされたテー

ブルの所有者のみが参照できます。所有者は競合を予想し、分析して、解決が必要な問題と見なされるものを確認する必要があります。

監視には、ユーザーデータを含まない `bdr.conflict_history_summary` を使用します。効率的なクエリプランを使用して、当日に発生した競合の数をカウントするクエリの例は次のとおりです。

```
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);
```

## 13.8 外部モニタリング

ユーザーが指定したメタデータを保存して、監視ツールが EDB Postgres 分散クラスターを理解して監視できるようにすることができます。この情報を一元化することにより、外部ツールは単一のノードにアクセスし、特定の接続のネットワークコストや警告/アラームのしきい値など、クラスター全体に関する詳細を読み取ることができます。

`bdr_superuser` は、これらのファンクションとテーブルに対する権限を持っています。ビュー `bdr.network_monitoring` は、`bdr_read_all_stats` ロールでもアクセスできます。

### 13.8.1 bdr.set\_node\_location

この関数は、ノードのメタデータを `bdr.node_location` に挿入します

### 13.8.2 概要

```
bdr.set_node_location(
    node_group_name text,
    node_name text,
    node_region text,
    node_location text);
```

### 13.8.3 パラメーター

- `node_group_name` - BDR グループの名前
- `node_name` - ノードの名前
- `node_region` - データセンターサイトまたはリージョン
- `node_location` - サーバー名、アベイラビリティゾーンなど

### 13.8.4 bdr.set\_network\_path\_info

このファンクションは、ノード間のネットワークパスのネットワークパスメタデータをテーブル `bdr.network_path_info` に挿入します。

### 13.8.5 概要

```
bdr.set_network_path_info(  
    node_group_name text,  
    region1 text,  
    region2 text,  
    location1 text,  
    location2 text,  
    network_cost numeric,  
    warning_threshold numeric,  
    alarm_threshold numeric)
```

### 13.8.6 パラメーター

- `node_group_name` - BDR グループの名前
- `region1` - オリジンサーバー名
- `region2` - リモートサーバー名
- `location1` - 発信元データセンター名
- `location2` - リモートデータセンター名
- `network_cost` - ネットワーク転送のコストを表す抽象値
- `warning_threshold` - しきい値を上げるべき遅延
- `alarm_threshold` - それを超えるとアラームが発生する遅延

### 13.8.7 bdr.network\_monitoring ビュー

このビューは、ノード間のネットワークパスに関する情報を収集します。

ロギングの構成は、`bdr.alter_node_set_log_config` 関数で定義されます。

## 13.9 統計を適用

BDR は、サブスクリプションとテーブルの両方について、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あります。サブスクリプション統計用の `bdr.stat_subscription` とリレーション統計用の `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た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

リレーションの場合のみ、これらの統計：

- リレーションのレプリケーションの処理にかかった合計時間
- リレーション（のみ）のロック（ある場合）を取得するための合計ロック待機時間

サブスクリプションの場合のみ、これらの統計：

- サブスクリプションに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 BDR GUC `bdr.track_subscription_apply` および `bdr.track_relation_apply` によって制御されます。

これらの出力例は次のようになります。

```
# 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アップストリームに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。

これらのビューの統計カウンターは、関数 `bdr.reset_subscription_stats` および `bdr.reset_relation_stats` を使用してゼロにリセットできます。

## 13.10 標準の PostgreSQL 統計ビュー

テーブルとインデックスの使用に関する統計は、ダウンストリームマスターによって通常更新されます。これは、

`autovacuum`

の正しい機能に不可欠です。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 つのビューにはアップストリームとダウンストリームの間で対応する結果が表示されます。

- `pg_stat_user_tables`
- `pg_statio_user_tables`

**注釈:** 上流のテーブル統計が下流の統計と\*似ているとは限りません。同じ量だけ\*変化\*することを期待しています。統計が 1M の挿入と 1M の更新を示すテーブルの例を考えます。新しいノードが BDR 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表示されます。ただし、その瞬間から、一方の変更がもう一方の側に複製されるため、アップストリームとダウンストリームのテーブル統計は同じ量で変更されます。

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 s および DELETE s を実行するワークロードで頻繁に使用されるように見える場合があります。

組み込みのインデックス監視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](#) で詳しく説明されています。

## 13.11 BDR バージョンの監視

BDR を使用すると、同じクラスター内のノード間で異なる Postgres バージョンと異なる BDR バージョンを実行できます。これ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 と BDR のバージョンを同時に取得します。例：

```
bdrdb=# SELECT node_name, postgres_version, bdr_version
        FROM bdr.group_versions_details;
 node_name | postgres_version | bdr_version
-----+-----+-----
 node1    | 14.1             | 4.0.0
 node2    | 14.1             | 4.0.0
```

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バージョンを実行することです。クラスターで異なる BDR バージョンを長時間実行しないことをお勧めします。

監視のために、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 BDR 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 BDR 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=他のノードと比較すると異なる BDR バージョンを実行しているノードがあります

説明されている動作は、ファンクション `bdr.monitor_group_versions()` で実装されています。ファンクションは、ビュー `bdr.group_version_details` から返された BDR 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：

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status | message
-----+-----
 OK     | All nodes are running same BDR versions
```

## 13.12 ラフトコンセンサスのモニタリング

Raft Consensus は常にクラスター全体で動作するはずです。Raft Consensus を動作させずに 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。

- BDR データ変更のレプリケーションはまだ正常に動作している可能性があります
- 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
- クラスターのメンテナンス操作（ノードへの参加、ノードの一部、スタンバイの昇格）は引き続き許可されますが、終了しない場合があります（単にハングします）
- BDR ノード間でノードの状態が正しく同期しない場合がある
  - BDR グループレプリケーションスロットは LSN を進めないため、WAL ファイルをディスクに保持します

ビュー `bdr.group_raft_details` は、関数 `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 Consensus ステータスを取得します。例：

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
 node_id | node_name | state | leader_id
-----+-----+-----+-----
```

(次のページに続く)

|            |       |               |            |
|------------|-------|---------------|------------|
| 1148549230 | node1 | RAFT_LEADER   | 1148549230 |
| 3367056606 | node2 | RAFT_FOLLOWER | 1148549230 |

以下の条件がすべて満たされている場合、Raft Consensus は正しく機能していると言えます。

- すべてのノードで有効な状態（RAFT\_LEADER または RAFT\_FOLLOWER）が定義されています
- ノードの 1 つだけが RAFT\_LEADER
- leader\_id はすべての行で同じであり、state = RAFT\_LEADER が含まれる行の node\_id と一致する必要があります

時々、Raft Consensus は新しい RAFT\_LEADER を定義するための新しい選挙を開始します。選挙中に、RAFT\_LEADER がなく、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。全体の選択には bdr.raft\_election\_timeout よりも時間がかかりません（デフォルトでは 6 秒に設定されています）。上記のクエリで選挙の状況が返された場合、単に bdr.raft\_election\_timeout を待ってからクエリを再度実行してください。bdr.raft\_election\_timeout が経過しても上記のいくつかの条件が満たされない場合、Raft コンセンサスは機能していません。

Raft Consensus は単一のノードでのみ正しく機能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 BDR ノードは、通常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（ファイル pg\_hba.conf を確認してください）
- BDR バージョンはすべてのノードで同じです
- bdr.raft\_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や、ネットワークの待機時間が長い場合、デフォルト値の bdr.raft\_election\_timeout（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ても Raft コンセンサスがまだ機能しない場合は、すべてのノードで bdr.raft\_election\_timeout を 30 秒に増やすことを検討してください。BDR 3.6.11 以降、bdr.raft\_election\_timeout を設定するにはサーバーのリロードのみが必要です。

Raft コンセンサスがクラスター操作タスクにどのように影響するか、および Raft コンセンサスがグループスロットを進める責任があるため、次のようにモニタリングアラートレベルを定義できます。

- status=UNKNOWN、message=このノードは BDR グループの一部ではありません
- status=OK、message=Raft Consensus 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 のノードがあります。選択が進行中の可能性があります
- status=WARNING、message=RAFT\_LEADER はありません。選択が進行中の可能性があります
- status=CRITICAL、message=Raft Consensus には単一のノードがあります
- status=CRITICAL、message=RAFT\_LEADER が定義されているときに RAFT\_CANDIDATE としてノードがあります



- 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

説明されている動作は関数 `bdr.monitor_group_raft()` で実装されています。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て、クラスター全体の Raft チェックを提供します。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status | message
-----+-----
 OK      | Raft Consensus is working correctly
```

## 13.13 レプリケーションスロットのモニタリング

各 BDR ノードは以下を保持します。

- アクティブな BDR ピアごとに 1 つのレプリケーションスロット
- 1 つのグループ複製スロット

例：

```
bdrdb=# SELECT slot_name, database, active, confirmed_flush_lsn
FROM pg_replication_slots ORDER BY slot_name;
 slot_name | database | active | confirmed_flush_lsn
-----+-----+-----+-----
 bdr_bdrdb_bdrgroup | bdrdb | f | 0/3110A08
 bdr_bdrdb_bdrgroup_node2 | bdrdb | t | 0/31F4670
 bdr_bdrdb_bdrgroup_node3 | bdrdb | t | 0/31F4670
 bdr_bdrdb_bdrgroup_node4 | bdrdb | t | 0/31F4670
```

ピアスロット名は規則 `bdr_<DATABASE>_<GROUP>_<PEER>` に従いますが、BDR グループスロット名は規則 `bdr_<DATABASE>_<GROUP>` に従います。これには、関数 `bdr.local_group_slot_name()` を使用してアクセスできます。

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、次のことを意味する可能性があります。

- 対応するピアがシャットダウンされているか、アクセスできません。または
- BDR の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.worker_errors` も確認します。根本的な原因は、たとえば、いずれかのノードで 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、BDR 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。BDR はこのスロットを維持し、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にその 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。

関数 `bdr.monitor_local_replslots()` は、すべての BDR ノードレプリケーションスロットが期待どおりに機能しているかどうかの概要を提供します。

```
bdrdb=# 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。

- 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

## 13.14 トランザクションの COMMIT の監視

デフォルトでは、BDR トランザクションはローカルノードでのみコミットします。その場合、トランザクション COMMIT は迅速に処理されます。

BDR は標準の PostgreSQL 同期レプリケーションで使用できますが、BDR は 2 つの新しいトランザクションコミットモード、CAMO と Eager レプリケーションも提供します。これらの各モードは、追加の堅牢性機能を提供しますが、COMMIT で追加のレイテンシーが犠牲になります。COMMIT での追加時間は、プロセスがさまざまな `wait_event` 状態を報告する `bdr.stat_activity` カタログを使用して動的に監視できます。1 つ以上の同期スタンバイからの確認を待っている COMMIT のトランザクションは SyncRep 待機イベントを報告しますが、2 つの新しいモードは EagerRep を報告します。

## 第14章 EDB Postgres Distributed Command Line Interface

EDB Postgres 分散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。

クラスターリソースに関する情報を検査、管理、および取得するために使用可能なコマンドについては、[Command reference](#) を参照してください。

TPAexec が PGD CLI を展開する方法、スタンドアロンサーバーに PGD CLI を手動でインストールする方法、および接続文字列を指定する方法については、

[Installing PGD CLI](#) を参照してください。

### 14.1 要件

PGD CLI を実行するには、postgres のスーパーユーザー権限が必要です。

### 14.2 PGD CLI の使用

pgd は、PGD コマンドラインインターフェイスのコマンド名です。コマンドオプションの説明については、コマンドリファレンスの [PGD CLI の使用](#) を参照してください。サンプルのユースケースについては、以下のセクションを参照してください。

### 14.3 構成ファイルの指定

ファイルの名前を変更するか、別の場所に移動する場合は、オプションの `-f` または `--config-file` フラグを使用して新しい名前と場所を指定します。例：

```
pgd show-nodes -f /opt/my-config.yml
```

## 14.4 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コマンドに直接渡します。この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、フラグが優先されます。例：

```
pgd show-nodes --dsn "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
``` ## 出力フォーマットの指定
```

PGD CLI は、次の出力形式をサポートしています。

```
.. csv-table::
:header: Format,Considerations
:widths: 10,30
:align: left
:class: longtable
```

tabular, デフォルトのフォーマット。データを表形式で表示します。

json, フォーマットなしで生データを表示します。一部のコマンドでは、json 出力に、追加のフィールドや詳細なメッセージなど、表形式の出力に表示されるデータが表示される場合があります。

yaml, フィールドオーダーがアルファベット順であることを除き、json と同じです。実験的であり、将来のバージョンでは完全にサポートされない可能性があります。

`-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 または yaml に変更します。例：

```
```sh
pgd show-nodes -o json
``` ## 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、次のように入力します。

```
```sh
pgd help
```

特定のコマンドとそのパラメーターのヘルプを表示するには、pgd help <command\_name> と入力します。例：

```
pgd help show-nodes
```

### 14.4.1 Installing PGD CLI

TPAexec は、デフォルトで、各 BDR ノードに PGD CLI をインストールして構成します。クラスター内の非 BDR インスタンスに PGD CLI をインストールする場合は、展開する前に TPAexec の構成ファイルでそのインスタンスに pgdcli ロールをアタッチするだけです。詳細については、[TPAexec](#) を参照してください。

#### 手動インストール

BDR repository から入手可能な .deb および .rpm

パッケージを使用して、Linux マシンに PGD CLI を手動でインストールできます。パッケージ名は edb-pgd-cli です。例：

```
# for Debian

sudo apt-get install edb-pgd-cli
```

PGD CLI が TPAexec によって構成されている場合、自動的に接続されますが、スタンドアロンの EDB Postgres 分散クラスターへの手動インストールでは、接続文字列を指定する必要があります。

#### データベース接続文字列の指定

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するか（次のセクションを参照）、接続文字列をコマンドに直接渡すことができます（[sample use case](#) を参照）。

#### 構成ファイルの使用

pgd-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 BDR 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ンであり、検証されません。

例：

```
cluster:
  name: cluster-name
  endpoints:
    - "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
```

pgd-config.yml は、デフォルトで/etc/edb ディレクトリにあります。PGD CLI は、次の場所で pgd-config.yml を検索します（優先順位 - 高いものから低いもの）。

1. /etc/edb （デフォルト）
2. \$HOME/.edb
3. . （作業ディレクトリ）

ファイルの名前を変更するか、別の場所に移動する場合は、コマンドを入力するときにオプションの-f または--config-file フラグを使用して、新しい名前と場所を指定します。

sample use case を参照してください。

### 14.4.2 Command reference

pgd は、PGD コマンドラインインターフェイスのコマンド名です。

#### 概要

EDB Postgres 分散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。

#### オプション

|                          |                                                                                        |
|--------------------------|----------------------------------------------------------------------------------------|
| -f, --config-file string | config file; ignored if<br>--dsn flag is present (default "/etc/edb/pgd-config.yml")   |
| --dsn string             | database connection string<br>e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres " |
| -h, --help               | help for pgd                                                                           |
| -L, --log-level string   | logging level: debug, info, warn, error (default "error")                              |
| -o, --output string      | output format: json, yaml                                                              |

#### こちらをご覧ください

- *check-health*
- *show-camo*
- *show-clockscrew*
- *show-events*
- *show-nodes*
- *show-raft*
- *show-replslots*
- *show-subscriptions*
- *show-version*

## check-health

EDB Postgres 分散クラスターのヘルスを確認します。

### 概要

すべてのノードにアクセスできるか、すべてのレプリケーションスロットが動作しているか、CAMO ペアが接続されているかなど、さまざまなチェックを実行します。

このコマンドの実行中にクラスターの負荷が高い場合、または多数のノードが含まれる場合、クロックスキューの現在の実装は不正確なスキュー値を返す場合があることに注意してください。

```
pgd check-health [flags]
```

### 例

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down; bdr-a1 and bdr-b1 are CAMO partners)

```
$ pgd check-health
```

| Check      | Status   | Message                                                     |
|------------|----------|-------------------------------------------------------------|
| -----      | -----    | -----                                                       |
| CAMO       | Critical | At least 1 CAMO partner is not connected                    |
| ClockSkew  | Critical | Clockskew cannot be determined for at least 1 BDR node pair |
| Connection | Critical | The node bdr-b1 is not accessible                           |
| Raft       | Warning  | There is no RAFT_LEADER, an election might be in progress   |
| Replslots  | Critical | There is at least 1 BDR replication slot which is inactive  |
| Version    | Warning  | There is at least 1 node that is not accessible             |

Example 2 (3 node cluster, all nodes are up but system clocks are not in sync)

```
$ pgd check-health
```

| Check      | Status  | Message                                                       |
|------------|---------|---------------------------------------------------------------|
| -----      | -----   | -----                                                         |
| CAMO       | Ok      | All CAMO pairs are connected                                  |
| ClockSkew  | Warning | At least 1 BDR node pair has clockskew greater than 2 seconds |
| Connection | Ok      | All BDR nodes are accessible                                  |
| Raft       | Ok      | Raft Consensus is working correctly                           |
| Replslots  | Ok      | All BDR replication slots are working correctly               |
| Version    | Ok      | All nodes are running same BDR versions                       |

Example 3 (3 node cluster, all nodes are up and all checks are Ok)

(次のページに続く)

(前のページからの続き)

```
$ pgd check-health
```

| Check      | Status | Message                                                    |
|------------|--------|------------------------------------------------------------|
| -----      | -----  | -----                                                      |
| CAMO       | Ok     | All CAMO pairs are connected                               |
| ClockSkew  | Ok     | All BDR node pairs have clockskew within permissible limit |
| Connection | Ok     | All BDR nodes are accessible                               |
| Raft       | Ok     | Raft Consensus is working correctly                        |
| Replslots  | Ok     | All BDR replication slots are working correctly            |
| Version    | Ok     | All nodes are running same BDR versions                    |

オプション

```
-h, --help    help for check-health
```

親コマンドから継承されたオプション

```
-f, --config-file string  config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-config.yml")
    --dsn string           database connection string
                           e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-camo

BDR CAMO（Commit at Most once）の詳細を表示します。

概要

CAMO パートナーの名前、接続と準備状況、保留中および未解決の CAMO トランザクション、apply\_lsn と receive\_lsn の違いなど、BDR CAMO（コミット アット ワンス）の詳細を表示します。このコマンドは、EDB Postgres Extended Server および EDB Postgres Advanced Server（v14 以降）でのみ使用できます。

```
pgd show-camo [flags]
```



## 例

Example 1 (3 node cluster, bdr-a1 and bdr-b1 are CAMO partner but bdr-b1 is down)

```
$ pgd show-camo
```

```
Node   CAMO Partner Connected Ready Transactions Resolved Apply LSN  Receive LSN
→Apply Queue Size
-----
→-----
bdr-a1 bdr-b1      false      false true                0/E42C99B0 0/E42C99B0  0
```

Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

```
$ pgd show-camo
```

```
Node   CAMO Partner Connected Ready Transactions Resolved Apply LSN  Receive LSN
→Apply Queue Size
-----
→-----
bdr-a1 bdr-b1      true       true  true                0/E533DAB8 0/E533DAB8  0
bdr-b1 bdr-a1      true       false true                3/7AE81A28 3/7AE81A28  0
```

Example 3 (3 node cluster, all nodes are up and in streaming state)

```
$ pgd show-camo
```

```
Node   CAMO Partner Connected Ready Transactions Resolved Apply LSN  Receive LSN
→Apply Queue Size
-----
→-----
bdr-a1 bdr-b1      true       true  true                0/E56AE520 0/E56AE520  0
bdr-b1 bdr-a1      true       true  true                3/7B180BA8 3/7B180BA8  0
```

## オプション

```
-h, --help  help for show-camo
```

## 親コマンドから継承されたオプション

```
-f, --config-file string    config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-config.yml")
    --dsn string            database connection string
                             e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string      logging level: debug, info, warn, error (default "error")
-o, --output string         output format: json, yaml
```

**show-clockskew**

各 BDR 間のクロックスキューのステータスを表示します。

## 概要

クラスタ内の各 BDR 間のクロックスキューのステータスを表示します。

このコマンドの実行中にクラスターの負荷が高い場合、または多数のノードが含まれる場合、クロックスキューの現在の実装は不正確なスキュー値を返す場合があることに注意してください。

記号 意味 --- - --- - \* ok ~ warning (skew > 2 秒) !クリティカル (スキュー > 5 秒) ×ダウン/到達不能?  
不明 - 該当なし

```
pgd show-clockskew [flags]
```

## 例

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

```
$ pgd show-clockskew
```

```
Node    bdr-a1 bdr-b1 bdr-c1 Current Time
----    -
bdr-a1  *      ?      *      2022-03-30 07:02:21.334472
bdr-b1  x      *      x      x
bdr-c1  *      ?      *      2022-03-30 07:02:21.186809
```

Example 2 (3 node cluster, all nodes are up)

```
$ pgd show-clockskew
```

```
Node    bdr-a1 bdr-b1 bdr-c1 Current Time
----    -
bdr-a1  *      *      *      2022-03-30 07:04:54.147017
```

(次のページに続く)

(前のページからの続き)

|        |   |   |   |                            |
|--------|---|---|---|----------------------------|
| bdr-b1 | * | * | * | 2022-03-30 07:04:54.340543 |
| bdr-c1 | * | * | * | 2022-03-30 07:04:53.90451  |

## オプション

```
-h, --help    help for show-clockskew
```

## 親コマンドから継承されたオプション

```
-f, --config-file string    config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-config.yml")
    --dsn string            database connection string
                             e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string      logging level: debug, info, warn, error (default "error")
-o, --output string         output format: json, yaml
```

## show-events

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イベントを表示します。

## 概要

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イベントを表示します。出力は Time 列で降順にソートされます。メッセージ列は数行後に切り捨てられます。完全なメッセージを表示するには、json 出力形式（「-o json」）を使用します。

各ノードの状態の詳細については、show-nodes コマンドのヘルプ（「pgd show-nodes -h」）を参照してください。

```
pgd show-events [flags]
```

## 例

Example 1 (3 node cluster)

```
$ pgd show-events --lines 10
```

| Time                          | Observer | Node  | Subject | Node  | Type            | Message |
|-------------------------------|----------|-------|---------|-------|-----------------|---------|
| ----                          | -----    | ----- | -----   | ----- | ----            | -----   |
| 2022-04-19 19:45:43.077712+00 | bdr-b1   |       | bdr-c1  |       | receiver worker | error_  |

→pglogical worker received fast finish request, exiting

(次のページに続く)

(前のページからの続き)

```

2022-04-19 19:45:43.066804+00 bdr-c1      bdr-a1      receiver worker error
↳pglogical worker received fast finish request, exiting
2022-04-19 19:45:43.057598+00 bdr-b1      bdr-a1      receiver worker error
↳pglogical worker received fast finish request, exiting
2022-04-19 19:45:43.046515+00 bdr-c1      bdr-b1      receiver worker error
↳pglogical worker received fast finish request, exiting
2022-04-19 19:45:43.033369+00 bdr-a1      bdr-c1      receiver worker error
↳pglogical worker received fast finish request, exiting
2022-04-19 19:45:43.013203+00 bdr-a1      bdr-b1      receiver worker error
↳pglogical worker received fast finish request, exiting
2022-04-19 19:45:40.024662+00 bdr-c1      bdr-c1      node state change      ACTIVE
2022-04-19 19:45:40.024575+00 bdr-b1      bdr-c1      node state change      ACTIVE
2022-04-19 19:45:40.022788+00 bdr-a1      bdr-c1      node state change      ACTIVE
2022-04-19 19:45:38.961424+00 bdr-c1      bdr-c1      node state change
↳PROMOTING

```

## オプション

```

-h, --help          help for show-events
-n, --lines int     show top n lines

```

## 親コマンドから継承されたオプション

```

-f, --config-file string  config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-config.yml")
    --dsn string          database connection string
                           e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string    logging level: debug, info, warn, error (default "error")
-o, --output string       output format: json, yaml

```

## show-nodes

EDB Postgres 分散クラスター内のすべてのノードとその概要を表示します。

### 概要

EDB Postgres 分散クラスター内のすべてのノードと、名前、ノード ID、グループ、現在/ターゲットの状態を含むそれらの概要を表示します。

#### ノードの状態

- **NONE**：ワーカーの起動時にノードの状態は設定解除され、すぐに現在の既知の状態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 分散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 分散クラスターに参加させ始めます。
- **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。これで、参加は、参加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して適用する最後のステップになりました。
- **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。すべての参加はこの状態でしばらくかかりますが、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` を呼び出してノードの状態を **ACTIVE** に移動しました。これらの 2 つの **PROMOTE** 状態は、1 つのノードのみが **STANDBY** よりも高く、**ACTIVE** よりも低い状態に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**：ノードはフル BDR 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**：ノードはアクティブまたはスタンバイでした。`bdr.part_node` を呼び出して、EDB Postgres 分散クラスターからノードを削除しました。
- **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欠落しているデータを同期します。
- **PARTED**：ノードの分割操作がすべてのノードで完了しました。

一度に 1 つのノードのみが、状態 **PROMOTE** または **PROMOTING** のいずれかになります。

[現在の状態] または [ターゲットの状態] 列に示されるように、ノードの読み取り専用状態は **STANDBY** と示されることに注意してください。

```
pgd show-nodes [flags]
```

例

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

```
$ pgd show-nodes
```

| Node   | Node ID    | Node Group | Current State | Target State | Status      | Seq ID |
|--------|------------|------------|---------------|--------------|-------------|--------|
| bdr-a1 | 3136956818 | bdrgroup   | ACTIVE        | ACTIVE       | Up          | 1      |
| bdr-b1 | 2380210996 | bdrgroup   | ACTIVE        | ACTIVE       | Unreachable | 2      |
| bdr-c1 | 1804769977 | bdrgroup   | ACTIVE        | ACTIVE       | Up          | 3      |

Example 2 (3 node cluster, all nodes are up)

```
$ pgd show-nodes
```

| Node   | Node ID    | Node Group | Current State | Target State | Status | Seq ID |
|--------|------------|------------|---------------|--------------|--------|--------|
| bdr-a1 | 3136956818 | bdrgroup   | ACTIVE        | ACTIVE       | Up     | 1      |
| bdr-b1 | 2380210996 | bdrgroup   | ACTIVE        | ACTIVE       | Up     | 2      |
| bdr-c1 | 1804769977 | bdrgroup   | ACTIVE        | ACTIVE       | Up     | 3      |

Example 3 (cluster with witness, logical standby and subscriber-only nodes)

Note: In contrast to logical standby, the subscriber-only nodes are fully joined node to the cluster

```
$ pgd show-nodes
```

| Node               | Node ID    | Node Group      | Current State | Target State | Status | Seq ID |
|--------------------|------------|-----------------|---------------|--------------|--------|--------|
| bdr-a1             | 3136956818 | bdrgroup        | ACTIVE        | ACTIVE       | Up     | 1      |
| bdr-b1             | 2380210996 | bdrgroup        | ACTIVE        | ACTIVE       | Up     | 2      |
| witness-c1         | 1450365472 | bdrgroup        | ACTIVE        | ACTIVE       | Up     | 3      |
| logical-standby-a1 | 1140256918 | bdrgroup        | STANDBY       | STANDBY      | Up     | 4      |
| logical-standby-b1 | 3541792022 | bdrgroup        | STANDBY       | STANDBY      | Up     | 5      |
| subscriber-only-c1 | 2448841809 | subscriber-only | ACTIVE        | ACTIVE       | Up     | 6      |

## オプション

```
-h, --help    help for show-nodes
```

## 親コマンドから継承されたオプション

```
-f, --config-file string    config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-config.yml")
    --dsn string            database connection string
                             e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string      logging level: debug, info, warn, error (default "error")
-o, --output string         output format: json, yaml
```

## show-raft

BDR Raft（コンセンサスプロトコル）の詳細を表示します。

## 概要

Raft 状態（リーダー、フォロワー）、Raft 選挙 ID、投票ノード数などの BDR Raft（コンセンサスプロトコル）の詳細を表示します。

注：ネットワークパーティションなどの場合、出力は CLI が接続されているノードによって異なる場合があります。

```
pgd show-raft [flags]
```

## 例

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

```
$ pgd show-raft
```

| Node   | Raft State    | Raft Term | Commit Index | Nodes | Voting Nodes | Protocol Version |
|--------|---------------|-----------|--------------|-------|--------------|------------------|
| bdr-c1 | RAFT_FOLLOWER | 29        | 6081272      | 3     | 3            | 4002             |
| bdr-a1 | RAFT_LEADER   | 29        | 6081272      | 3     | 3            | 4002             |
| bdr-b1 |               |           |              |       |              |                  |

Example 2 (3 node cluster, all nodes are up)

```
$ pgd show-raft
```

(次のページに続く)

(前のページからの続き)

| Node   | Raft State    | Raft Term | Commit Index | Nodes Voting | Nodes | Protocol Version |
|--------|---------------|-----------|--------------|--------------|-------|------------------|
| bdr-c1 | RAFT_FOLLOWER | 38        | 6132327      | 3            | 3     | 4002             |
| bdr-a1 | RAFT_LEADER   | 38        | 6132331      | 3            | 3     | 4002             |
| bdr-b1 | RAFT_FOLLOWER | 38        | 6132336      | 3            | 3     | 4002             |

Example 3 (3 node cluster, all nodes are up but bdr-a1 is not able to connect to other nodes; following is the output when cli is connected to bdr-a1)

```
$ pgd show-raft
```

| Node   | Raft State    | Raft Term | Commit Index | Nodes Voting | Nodes | Protocol Version |
|--------|---------------|-----------|--------------|--------------|-------|------------------|
| bdr-c1 |               |           |              |              |       |                  |
| bdr-a1 | RAFT_FOLLOWER | 40        | 6176769      | 3            | 3     | 4002             |
| bdr-b1 |               |           |              |              |       |                  |

Example 4 (cluster with witness, logical standby and subscriber-only nodes)

Note: Unlike full-bdr (or witness node), logical standby and subscriber-only nodes dont have raft voting rights.

```
$ pgd show-raft
```

| Node               | Raft State    | Raft Term | Commit Index | Nodes Voting | Nodes | Protocol Version |
|--------------------|---------------|-----------|--------------|--------------|-------|------------------|
| bdr-a1             | RAFT_LEADER   | 0         | 10268        | 6            | 3     | 4003             |
| bdr-b1             | RAFT_FOLLOWER | 0         | 10279        | 6            | 3     | 4003             |
| witness-c1         | RAFT_FOLLOWER | 0         | 10281        | 6            | 3     | 4003             |
| logical-standby-a1 | RAFT_FOLLOWER | 0         | 10281        | 6            | 3     | 4003             |
| logical-standby-b1 | RAFT_FOLLOWER | 0         | 10281        | 6            | 3     | 4003             |
| subscriber-only-c1 | RAFT_FOLLOWER | 0         | 10281        | 6            | 3     | 4003             |



## オプション

```
-h, --help    help for show-raft
```

## 親コマンドから継承されたオプション

```
-f, --config-file string    config file; ignored if
                             --dsn flag is present (default "/etc/edb/pgd-config.yml")
    --dsn string            database connection string
                             e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string      logging level: debug, info, warn, error (default "error")
-o, --output string         output format: json, yaml
```

## show-replslots

BDR レプリケーションスロットのステータスを表示します。

### 概要

BDR レプリケーションスロットのステータスを表示します。verbose フラグを指定した出力は、スロットがアクティブかどうか、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）、おおよそのラグなどの詳細を提供します。

記号 意味 — — — — \* ok ~ warning (lag > 10M) !クリティカル (ラグ> 100M OR スロットが「非アクティブ」または「切断」) x ダウン/到達不能-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。

```
pgd show-replslots [flags]
```

### 例

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

```
$ pgd show-replslots
```

```
Node    bdr-a1 bdr-b1 bdr-c1
-----
bdr-a1  *      !(6.6G) *
bdr-b1  x      *      x
bdr-c1  *      !(6.9G) *
```

(次のページに続く)

(前のページからの続き)

```
$ pgd show-replslots --verbose
```

| Origin Node | Target Node | Status (active/state) | Write Lag (bytes/duration)    | Flush Lag<br>→(bytes/duration) | Replay Lag (bytes/duration) | Sent Lag (bytes) |
|-------------|-------------|-----------------------|-------------------------------|--------------------------------|-----------------------------|------------------|
| bdr-a1      | bdr-b1      | f / disconnected      | 6.6G / 8 days 02:58:36.243723 | 6.6G / 8                       |                             |                  |
| bdr-a1      | bdr-c1      | t / streaming         | 0B / 00:00:00                 | 0B /                           |                             |                  |
| bdr-c1      | bdr-a1      | t / streaming         | 0B / 00:00:00.000812          | 0B /                           |                             |                  |
| bdr-c1      | bdr-b1      | f / disconnected      | 6.9G / 8 days 02:58:36.004415 | 6.9G / 8                       |                             |                  |

Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

```
$ pgd show-replslots
```

| Node   | bdr-a1 | bdr-b1  | bdr-c1 |
|--------|--------|---------|--------|
| bdr-a1 | *      | !(6.9G) | *      |
| bdr-b1 | *      | *       | *      |
| bdr-c1 | *      | !(5.8G) | *      |

```
$ pgd show-replslots --verbose
```

| Origin Node | Target Node | Status (active/state) | Write Lag (bytes/duration) | Flush Lag<br>→(bytes/duration) | Replay Lag (bytes/duration) | Sent Lag (bytes) |
|-------------|-------------|-----------------------|----------------------------|--------------------------------|-----------------------------|------------------|
| bdr-a1      | bdr-b1      | t / catchup           | 6.9G / 00:00:00.000778     | 6.9G /                         |                             |                  |
| bdr-a1      | bdr-c1      | t / streaming         | 0B / 00:00:00.104121       | 0B /                           |                             |                  |
| bdr-b1      | bdr-a1      | t / streaming         | 0B / 00:00:00              | 0B /                           |                             |                  |
| bdr-b1      | bdr-c1      | t / streaming         | 0B / 00:00:00              | 0B /                           |                             |                  |
| bdr-c1      | bdr-a1      | t / streaming         | 6.8K / 00:00:00            | 6.8K /                         |                             |                  |
| bdr-c1      | bdr-b1      | t / catchup           | 5.5G / 00:00:00.008257     | 5.5G /                         |                             |                  |

(次のページに続く)

(前のページからの続き)

Example 3 (3 node cluster, all nodes are up and in streaming state)

```
$ pgd show-replslots
```

```
Node    bdr-a1 bdr-b1 bdr-c1
```

```
-----
```

```
bdr-a1 *      *      *
```

```
bdr-b1 *      *      *
```

```
bdr-c1 *      *      *
```

```
$ pgd show-replslots --verbose
```

```
Origin Node Target Node Status (active/state) Write Lag (bytes/duration) Flush Lag
↳(bytes/duration) Replay Lag (bytes/duration) Sent Lag (bytes)
```

```
-----
↳-----
bdr-a1      bdr-b1      t / streaming      0B / 00:00:00      0B /
↳00:00:00      0B / 00:00:00      0B
bdr-a1      bdr-c1      t / streaming      0B / 00:00:00      0B /
↳00:00:00      0B / 00:00:00      0B
bdr-b1      bdr-a1      t / streaming      0B / 00:00:00      0B /
↳00:00:00      0B / 00:00:00      0B
bdr-b1      bdr-c1      t / streaming      0B / 00:00:00      0B /
↳00:00:00      0B / 00:00:00      0B
bdr-c1      bdr-a1      t / streaming      0B / 00:00:00      528B /
↳00:00:00      528B / 00:00:00      0B
bdr-c1      bdr-b1      t / streaming      528B / 00:00:00      528B /
↳00:00:00      528B / 00:00:00      0B
```

Example 4 (cluster with witness, logical standby and subscriber-only nodes;  
upstream for logical-standby-a1 is bdr-a1 and for logical-standby-b1 it is bdr-b1)

Note:

1. A logical standby is sent data only by one source node, but no other nodes receive replication changes from it
2. Subscriber-only node subscribes to replication changes from other nodes in the cluster, but no other nodes receive replication changes from it

```
$ pgd show-replslots
```

```
Node                bdr-a1 bdr-b1 logical-standby-a1 logical-standby-b1 subscriber-
↳only-c1 witness-c1
```

```
-----
```

```
↳-----
```

```
bdr-a1                *      *      *      -      *
```

```
*
```

(次のページに続く)

(前のページからの続き)

|                    |   |   |   |   |   |
|--------------------|---|---|---|---|---|
| bdr-b1             | * | * | - | * | * |
| *                  |   |   |   |   |   |
| logical-standby-a1 | - | - | * | - | - |
| -                  |   |   |   |   |   |
| logical-standby-b1 | - | - | - | * | - |
| -                  |   |   |   |   |   |
| subscriber-only-c1 | - | - | - | - | * |
| -                  |   |   |   |   |   |
| witness-c1         | * | * | - | - | * |
| *                  |   |   |   |   |   |

オプション

|               |                               |
|---------------|-------------------------------|
| -h, --help    | help for show-replslots       |
| -v, --verbose | verbose output (default true) |

親コマンドから継承されたオプション

|                          |                                                           |
|--------------------------|-----------------------------------------------------------|
| -f, --config-file string | config file; ignored if                                   |
|                          | --dsn flag is present (default "/etc/edb/pgd-config.yml") |
| --dsn string             | database connection string                                |
|                          | e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "  |
| -L, --log-level string   | logging level: debug, info, warn, error (default "error") |
| -o, --output string      | output format: json, yaml                                 |

show-subscriptions

BDR サブスクリプション（受信レプリケーション）の詳細を表示します。

概要

オリジン/ターゲットノード、最後にリプレイされたトランザクションのタイムスタンプ、現在から最後にリプレイされたトランザクションのタイムスタンプなどの BDR サブスクリプション（受信レプリケーション）の詳細を表示します。

|                                |
|--------------------------------|
| pgd show-subscriptions [flags] |
|--------------------------------|

例

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

```
$ pgd show-subscriptions
```

| Origin Node | Target Node | Last Transaction Replayed At  | Lag Duration (seconds) |
|-------------|-------------|-------------------------------|------------------------|
| bdr-a1      | bdr-c1      | 2022-04-23 13:13:40.854433+00 | 0.514275               |
| bdr-b1      | bdr-a1      |                               |                        |
| bdr-b1      | bdr-c1      |                               |                        |
| bdr-c1      | bdr-a1      | 2022-04-23 13:13:40.852233+00 | 0.335464               |

Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

```
$ pgd show-subscriptions
```

| Origin Node | Target Node | Last Transaction Replayed At  | Lag Duration (seconds) |
|-------------|-------------|-------------------------------|------------------------|
| bdr-a1      | bdr-b1      | 2022-04-23 13:14:45.669254+00 | 0.001686               |
| bdr-a1      | bdr-c1      | 2022-04-23 13:14:46.157913+00 | -0.002009              |
| bdr-b1      | bdr-a1      |                               |                        |
| bdr-b1      | bdr-c1      |                               |                        |
| bdr-c1      | bdr-a1      | 2022-04-23 13:14:45.698472+00 | 0.259521               |
| bdr-c1      | bdr-b1      | 2022-04-23 13:14:45.667979+00 | 0.002961               |

Example 3 (3 node cluster, all nodes are up and in streaming state)

```
$ pgd show-subscriptions
```

| Origin Node | Target Node | Last Transaction Replayed At  | Lag Duration (seconds) |
|-------------|-------------|-------------------------------|------------------------|
| bdr-a1      | bdr-b1      | 2022-04-23 13:15:39.732375+00 | 0.034462               |
| bdr-a1      | bdr-c1      | 2022-04-23 13:15:40.179618+00 | 0.002647               |
| bdr-b1      | bdr-a1      | 2022-04-23 13:15:39.719994+00 | 0.305814               |
| bdr-b1      | bdr-c1      | 2022-04-23 13:15:40.180886+00 | 0.001379               |
| bdr-c1      | bdr-a1      | 2022-04-23 13:15:39.714397+00 | 0.311411               |
| bdr-c1      | bdr-b1      | 2022-04-23 13:15:39.714397+00 | 0.052440               |

Example 4 (cluster with witness, logical standby and subscriber-only nodes;  
upstream for logical-standby-a1 is bdr-a1 and for logical-standby-b1 it is bdr-b1)

Note: Logical standby and subscriber-only nodes receive changes but do not  
send changes made locally to other nodes

```
$ pgd show-subscriptions
```

(次のページに続く)

(前のページからの続き)

| Origin Node | Target Node        | Last Transaction Replayed At  | Lag Duration (seconds) |
|-------------|--------------------|-------------------------------|------------------------|
| -----       | -----              | -----                         | -----                  |
| bdr-a1      | bdr-b1             | 2022-04-23 13:40:49.106411+00 | 0.853665               |
| bdr-a1      | logical-standby-a1 | 2022-04-23 13:40:50.72036+00  | 0.138430               |
| bdr-a1      | logical-standby-b1 |                               |                        |
| bdr-a1      | subscriber-only-c1 | 2022-04-23 13:40:50.72036+00  | 0.016226               |
| bdr-a1      | witness-c1         | 2022-04-23 13:40:50.470142+00 | 0.001514               |
| bdr-b1      | bdr-a1             | 2022-04-23 13:40:49.10174+00  | 1.095422               |
| bdr-b1      | logical-standby-a1 |                               |                        |
| bdr-b1      | logical-standby-b1 | 2022-04-23 13:40:50.713666+00 | 0.271213               |
| bdr-b1      | subscriber-only-c1 | 2022-04-23 13:40:50.713666+00 | 0.022920               |
| bdr-b1      | witness-c1         | 2022-04-23 13:40:50.471789+00 | -0.000133              |
| witness-c1  | bdr-a1             | 2022-04-23 13:40:49.107706+00 | 1.089456               |
| witness-c1  | bdr-b1             | 2022-04-23 13:40:49.107706+00 | 0.852370               |
| witness-c1  | logical-standby-a1 |                               |                        |
| witness-c1  | logical-standby-b1 |                               |                        |
| witness-c1  | subscriber-only-c1 | 2022-04-23 13:40:50.719844+00 | 0.016742               |

オプション

|            |                             |
|------------|-----------------------------|
| -h, --help | help for show-subscriptions |
|------------|-----------------------------|

親コマンドから継承されたオプション

|                          |                                                                                        |
|--------------------------|----------------------------------------------------------------------------------------|
| -f, --config-file string | config file; ignored if<br>--dsn flag is present (default "/etc/edb/pgd-config.yml")   |
| --dsn string             | database connection string<br>e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres " |
| -L, --log-level string   | logging level: debug, info, warn, error (default "error")                              |
| -o, --output string      | output format: json, yaml                                                              |

**show-version**

各ノードにインストールされている BDR と Postgres のバージョンを表示します。

**概要**

クラスター内の各ノードにインストールされている BDR と Postgres のバージョンを表示します。

```
pgd show-version [flags]
```

**例**

Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

```
$ pgd show-version
```

```
Node    BDR Version Postgres Version
```

```
----
```

```
bdr-c1 4.1.0      14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-  
→1.buster+1)
```

```
bdr-a1 4.1.0      14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-  
→1.buster+1)
```

```
bdr-b1
```

Example 2 (3 node cluster, all nodes are up)

```
$ pgd show-version
```

```
Node    BDR Version Postgres Version
```

```
----
```

```
bdr-c1 4.1.0      14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-  
→1.buster+1)
```

```
bdr-a1 4.1.0      14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-  
→1.buster+1)
```

```
bdr-b1 4.1.0      14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-  
→1.buster+1)
```

## オプション

|                                               |
|-----------------------------------------------|
| <code>-h, --help</code> help for show-version |
|-----------------------------------------------|

## 親コマンドから継承されたオプション

|                                       |                                                                                        |
|---------------------------------------|----------------------------------------------------------------------------------------|
| <code>-f, --config-file string</code> | config file; ignored if<br>--dsn flag is present (default "/etc/edb/pgd-config.yml")   |
| <code>--dsn string</code>             | database connection string<br>e.g. "host=bdr-a1 port=5432 dbname=bdrdb user=postgres " |
| <code>-L, --log-level string</code>   | logging level: debug, info, warn, error (default "error")                              |
| <code>-o, --output string</code>      | output format: json, yaml                                                              |