



EDB

Postgres Distributed

リリース 4.0.0

EDB

2022年06月14日

目次

第 1 章	Release notes	3
1.1	Release notes for EDB Postgres Distributed version 4	3
第 2 章	‘Known issues’	5
2.1	データの一貫性	5
2.2	問題のリスト	5
2.3	制限事項	6
第 3 章	Key components	9
3.1	Postgres サーバー	9
3.2	Postgres の拡張機能	9
3.2.1	BDR	9
第 4 章	Terminology	11
4.1	非同期レプリケーション	11
4.2	可用性	11
4.3	CAMO または commit-at-most-once	11
4.4	クラスタリング	11
4.5	データシャーディング	12
4.6	BDR の Eager Replication	12
4.7	結果的な一貫性	12
4.8	フェールオーバー	12
4.9	水平スケーリング または スケールアウト	12
4.10	論理レプリケーション	12
4.11	ナイン	13
4.12	ノード	13
4.13	物理レプリケーション	13
4.14	読み取りのスケラビリティ	13
4.15	目標復旧時点 (RPO)	13
4.16	目標復旧時間 (RTO)	13
4.17	単一障害点 (SPOF)	14
4.18	スイッチオーバー	14
4.19	同期レプリケーション	14
4.20	2 フェーズコミット (2PC)	14
4.21	垂直スケーリング または スケールアップ	14
4.22	書き込みスケラビリティ	14
第 5 章	Choosing your architecture	15
5.1	標準の EDB Always On アーキテクチャ	15
5.2	アーキテクチャの詳細	16
5.3	アーキテクチャの選択	16

5.3.1	Always On Bronze (single active location - cloud region or on prem data center)	17
5.3.2	Always On Silver (single active location - cloud region or on prem data center)	17
5.3.3	Always On Gold (two active locations - cloud regions or on prem data centers)	20
5.3.4	Always On Platinum (two locations; fast HA restoration)	21
第 6 章	Choosing a Postgres distribution	23
第 7 章	Choosing durability	25
第 8 章	Other considerations	27
8.1	展開とサイジングの考慮事項	27
8.2	時計とタイムゾーン	27
第 9 章	Deployment options	29
9.1	TPAexec	29
9.1.1	インクリメンタルな変更	30
9.1.2	Ansible で拡張可能	30
9.1.3	クラスター管理	30
9.1.4	ただの Postgres です	30
9.1.5	はじめに	30
9.1.5.1	Installing TPAexec	31
9.1.5.2	Using TPAexec	32
9.1.5.3	Deploying EDB Postgres Distributed	36
第 10 章	Upgrading	39
10.1	アップグレード計画	39
10.1.1	ローリングアップグレードの考慮事項	40
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	40
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	41
10.1.4	CAMO 対応クラスターのアップグレード	41
10.2	アップグレードの準備	41
10.3	サーバーソフトウェアのアップグレード	41
10.3.1	BDR 拡張機能のアップグレード	42
10.3.2	Postgres を停止する	42
10.3.3	アップグレードパッケージ	42
10.3.4	Postgres を起動	42
10.3.5	Postgres のアップグレード	42
10.3.6	マイナーバージョンの Postgres のアップグレード	42
10.3.7	メジャーバージョン Postgres のアップグレード	43
10.4	アップグレードのチェックと検証	43
10.5	アプリケーションスキーマのアップグレード	43
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	43
10.5.1.1	In-place Postgres Major Version Upgrades	45
第 11 章	Compatibility matrix	49
第 12 章	Backup and recovery	51
12.1	バックアップ	51
12.1.1	pg_dump	51

12.1.2	物理バックアップ	52
12.1.3	最終的な一貫性	52
12.1.4	ポイントインタイムリカバリ (PITR)	53
12.2	復元する	54
12.2.1	BDR 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	54
12.2.2	BDR メタデータのクリーンアップ	55
12.2.3	レプリケーションオリジンのクリーンアップ	55
12.2.4	レプリケーションスロットのクリーンアップ	55
第 13 章	Monitoring	57
13.1	モニタリングの概要	57
13.2	モニタリングノードの参加と削除	58
13.3	レプリケーションピアの監視	59
13.3.1	送信レプリケーションのモニタリング	59
13.3.2	着信レプリケーションの監視	61
13.3.3	LCR を使用した WAL 送信者の監視	62
13.4	BDR レプリケーションワーカーの監視	63
13.5	BDR ライターの監視	63
13.6	グローバルロックの監視	63
13.7	競合の監視	64
13.8	外部モニタリング	65
13.8.1	bdr.set_node_location	65
13.8.2	概要	65
13.8.3	パラメーター	65
13.8.4	bdr.set_network_path_info	65
13.8.5	概要	66
13.8.6	パラメーター	66
13.8.7	bdr.network_monitoring ビュー	66
13.9	統計を適用	66
13.10	標準の PostgreSQL 統計ビュー	67
13.11	BDR バージョンの監視	68
13.12	ラフトコンセンサスのモニタリング	69
13.13	レプリケーションスロットのモニタリング	71
13.14	トランザクションの COMMIT の監視	72
第 14 章	PGD CLI	73

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。

CAMO または *commit-at-most-once*、Group Commit、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 Release notes

EDB Postgres Distributed のドキュメントでは、EDB Postgres Distributed 4 でサポートされている AlwaysOn アーキテクチャについて説明しています。個々の製品機能の詳細については、以下を参照してください。

- BDR release notes
- HARP release notes

Version	Release Date
Release notes for EDB Postgres Distributed version 4	2022 May 17

1.1 Release notes for EDB Postgres Distributed version 4

EDB Postgres Distributed バージョン 4 には以下が含まれます。

Feature	Description
Faster Postgres major version upgrade	Postgres Distributed は、完全にオンラインでのメジャーバージョンアップグレードを可能にする唯一の Postgres 高可用性テクノロジーです。この機能が改善され、ノード間でデータを移行せずにインプレースアップグレードが可能になりました。最大のクラスターでも、数分で最新のメジャーバージョンにアップグレードできるようになりました。
Replication lag control	非同期レプリケーションを使用する場合、トランザクションが調整されるまでのレプリケーションラグのしきい値を構成できます。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で待機時間の影響を受けることなく、RPO を低く保つことができます。
Distributed sequences by default	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代わりに、使用されるデータ型に最適な種類のシーケンスで自動的に構成された分散シーケンスを取得します。新しい SnowflakeID アルゴリズムは、デフォルトとして使用できない制約があった Timeshard を置き換えます。
Postgres Distributed Command Line Interface (CLI)	レプリケーション、コンセンサス、ノードの現在の状態など、Postgres 分散クラスターに関する情報を CLI から取得できます。クラスターのノード全体でクエリを実行することもできます。クエリに依存する代わりに、CLI で簡素化されたコマンドを使用できます。詳細については、 :ref:`EDB Postgres Distributed Command Line Interface </pgd_cli/latest>` を参照してください。

次のページに続く

表 2 – 前のページからの続き

Feature	Description
Simplified synchronous replication configuration	ノードごとに複数の異なる Grand Unified Configurations (GUC) を調整する代わりに、SQL レベルのインターフェイスを使用して、トランザクションの複製方法と、使用する耐久性と可視性の基準を構成できるため、すべてのノードを 1 か所から一貫して構成できます。これにより、将来的には（グループコミットとともに）同期レプリケーションに関する機能と柔軟性が向上します。
Group Commit	トランザクションをコミットする前に Postgres 分散グループ内でクォーラムを要求する同期レプリケーションを持つことができるため、EDB は Oracle DataGuard または Postgres 同期レプリケーションのより魅力的な代替手段になります。これにより、Eager Majority など、将来的に同期レプリケーションに関するより多くの機能が使用できるようになります。

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

このセクションでは、BDR のチケットシステムで追跡されている多くの既知の問題をリストします。これらは、将来のリリースのいずれかで再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の 1 つ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ありません。発散クラスターを引き起こす可能性があります。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では、つまり他のすべてのノードが到達不能または操作できない場合、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コントロールはその作業を取得し、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。ペアの追加または削除のみ可能です。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 **CAMO または `commit-at-most-once`** 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 レプリケーションも 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のサポー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。明示的な 2 フェーズコミットもサポートしていません。TRUNCATE コマンドが許可されていることに注意してください。
- 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場合があります。特定のコミットスコープを使用するトランザクションが終了したことを確認してから、変更または削除してください。

2.3 制限事項

このセクションには、設計によるものであり、将来的に解決される予定の設計上の制限の（非包括的な）リストが含まれています。

- CAMO/Eager/Group Commit を使用するノードでは、ノードをフィジカルスタンバイに置き換えても機能しません。フィジカルスタンバイと BDR を組み合わせることは、可能な場合でも一般的にはお勧めしません。
- `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てから同じ名前再度作成された場合、または 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た場合、DDL レプリケーションが再び作成される場合、アクティブ。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。シーケンスはいくつか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、ユーザーは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 構成の一部ではありません。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 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。

3.1 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。

3.2 Postgres の拡張機能

3.2.1 BDR

BDR 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。

!!!ノート「pglogical3」

BDR の古いバージョン (3.6/3.7) は、BDR がビルドするレプリケーションチャンネルを提供するために pglogical3 に依存していました。現在は使用されていません。

HARP は、BDR クラスター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。

第4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。

4.1 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。

4.2 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）

4.3 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。

4.4 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。

4.5 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、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。

4.6 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。

4.7 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。

4.8 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。

4.9 水平スケーリング または スケールアウト

増加したトラフィックを処理するための 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。

4.10 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。

4.11 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可されます。

4.12 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。

4.13 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。

4.14 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。

4.15 目標復旧時点（RPO）

アプリケーションの配信の中断が原因で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。

4.16 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。

4.17 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。

4.18 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。

4.19 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た場合、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。

4.20 2 フェーズコミット（2PC）

複数のデータベースノード間で一貫性を実現するためのマルチステッププロセス。

4.21 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）。

4.22 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5章 Choosing your architecture

Always On アーキテクチャは EDB の推奨されるプラクティスを反映しており、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ちます。これらの構成は、単一ロケーションのアーキテクチャから、ハードウェア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、AlwaysOn アーキテクチャ用の EDB の標準展開ツールである 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 (single active location - cloud region or on prem data center)*
：単一のアクティブな場所（データセンターまたはアベイラビリティゾーン [AZ]）
- *Always On Silver (single active location - cloud region or on prem data center)*
：フェールオーバー機能を迅速に復元するための冗長ハードウェアを備えた単一のアクティブな場所とディザスタリカバリー（DR）場所
- *Always On Gold (two active locations - cloud regions or on prem data centers)*：2 つのアクティブな場所
- *Always On Platinum (two locations; fast HA restoration)*：

ホットスタンバイモードの追加の冗長ハードウェアを備えた 2 つのアクティブロケーション

すべての Always On アーキテクチャは、段階的に堅牢な障害状況を保護します。たとえば、Always On Bronze はローカルハードウェア障害から保護しますが、場所（データセンターまたは AZ）の障害からの保護は提供しません。Always On Silver は、バックアップが別の場所に保持されるため、場所の壊滅的な損失が発生した場合にある程度の保護を提供します。ただし、データベースを最初にバックアップから復元する必要があるため、目標復旧時間（RTO）要件に違反する可能性があります。Always On Gold は、マルチマスターメッシュネットワークで接続された 2 つのアクティブなロケーションを提供するため、ロケーションがオフラインになった場合でもサービスを利用できます。最後に、Always On Platinum は両方の場所に冗長なホットスタンバイハードウェアを追加して、ハードウェアに障害が発生した場合にローカルの高可用性を維持します。

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、ゼロ目標復旧ポイント（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ゼロが保証されます。

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の追加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、[ラフトコンセンサスのモニタリング](#)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必要ありませんが、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 BDR 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、常に 3 つの BDR ノードがあります。2 つのアクティブな場所を使用する Always On Gold と Platinum では、RAFT 要件をサポートする監視ノードとして 5 番目の BDR ノードを導入します。

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 AlwaysOn アーキテクチャに接続します。ここで、各 HA プロキシサーバーはマルチホスト接続文字列の個別のエントリです。他の接続メカニズムは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をサポート

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	Silver	Gold	Platinum
Location failure protection	No (unless Barman is moved off-site)	Yes - Recovery from backup	Yes - instant failover to fully functional site	Yes - instant failover to fully functional site
Failover to DR or full DC	DR (if Barman is located offsite); NA otherwise	DR (if Barman is located offsite)	Full DC	Full DC

次のページに続く

表 1 – 前のページからの続き

	Bronze	Silver	Gold	Platinum
Fast local restoration of high availability after device failure	No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB	Yes; three local data nodes allow to maintain HA after device failure	No; time to restore HA: (1) VM prov + (2) approx 60 min/500GB	Yes; logical standbys can quickly be promoted to master data nodes
Cross data center network traffic	Backup traffic only (if Barman is located offsite); none otherwise	Backup traffic only (if Barman is located offsite); none otherwise	Full replication traffic	Full replication traffic
License cost	2 data nodes	3 data nodes	4 data nodes	4 data nodes 2 logical standbys

5.3.1 Always On Bronze (single active location - cloud region or on prem data center)

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。

5.3.2 Always On Silver (single active location - cloud region or on prem data center)

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）

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 Bronze

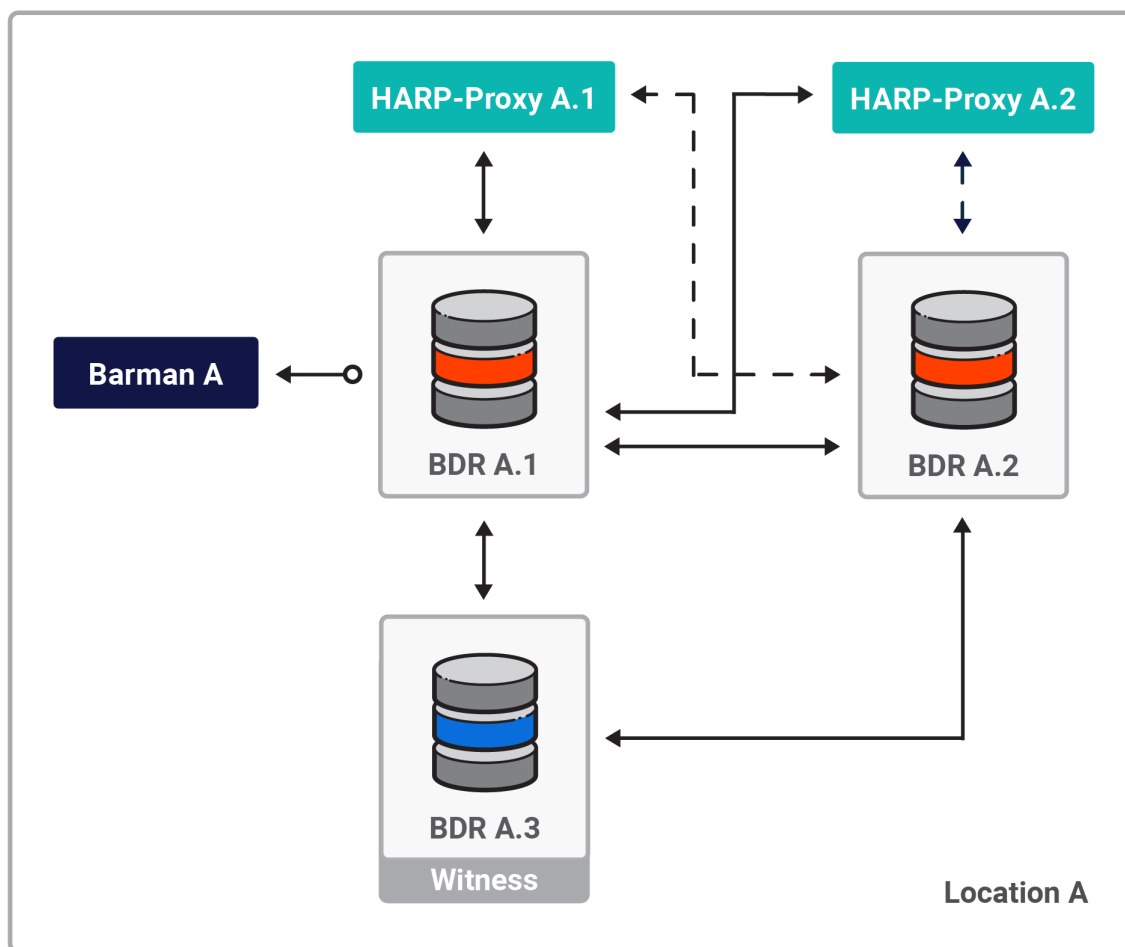


図 1: Always On Bronze architecture

Always On Silver

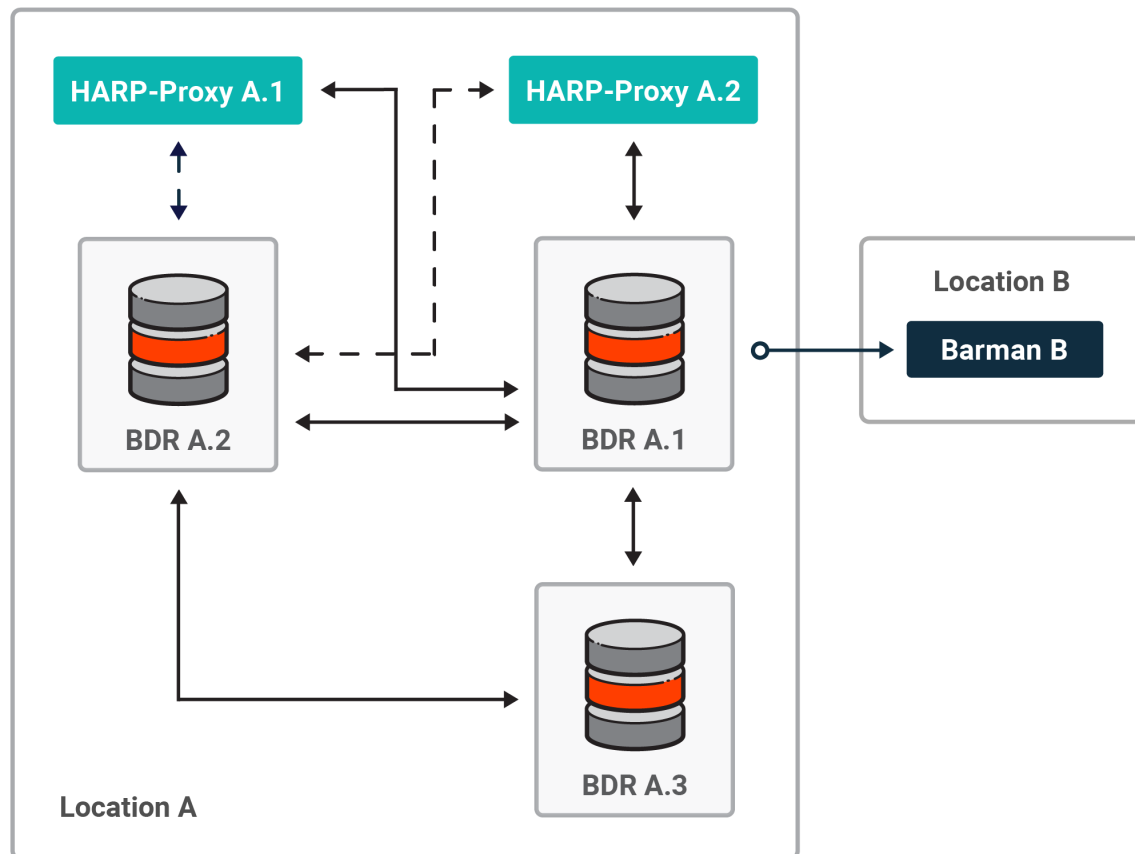


図 2: Always On Silver architecture

5.3.3 Always On Gold (two active locations - cloud regions or on prem data centers)

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の場合にのみリモートの場所を優先します。

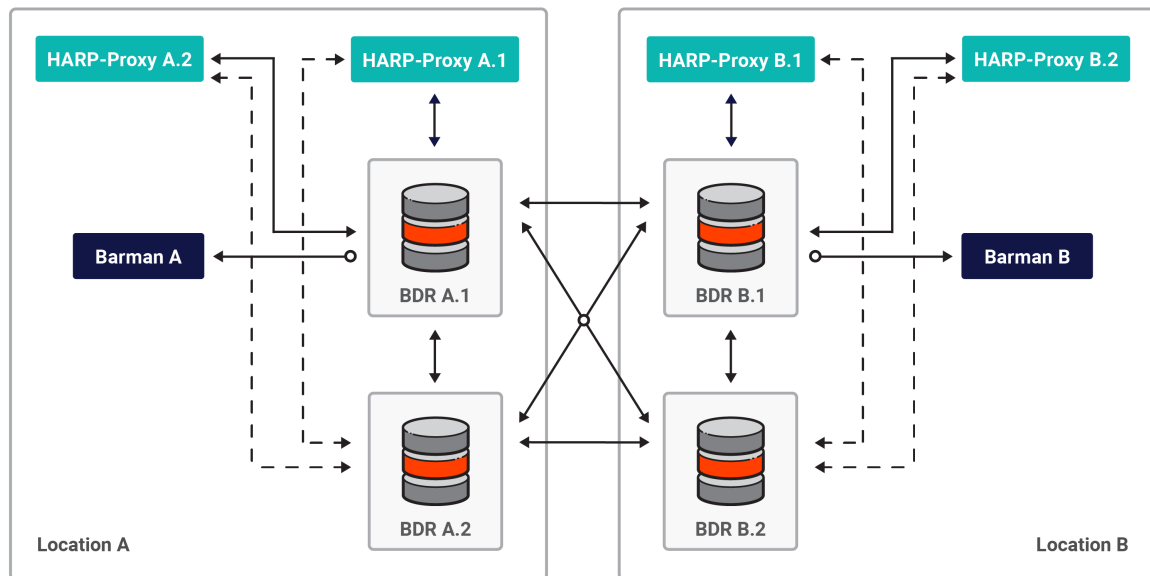
このアーキテクチャにより、競合が発生しない/低い場所での地理分散書き込みが可能になります

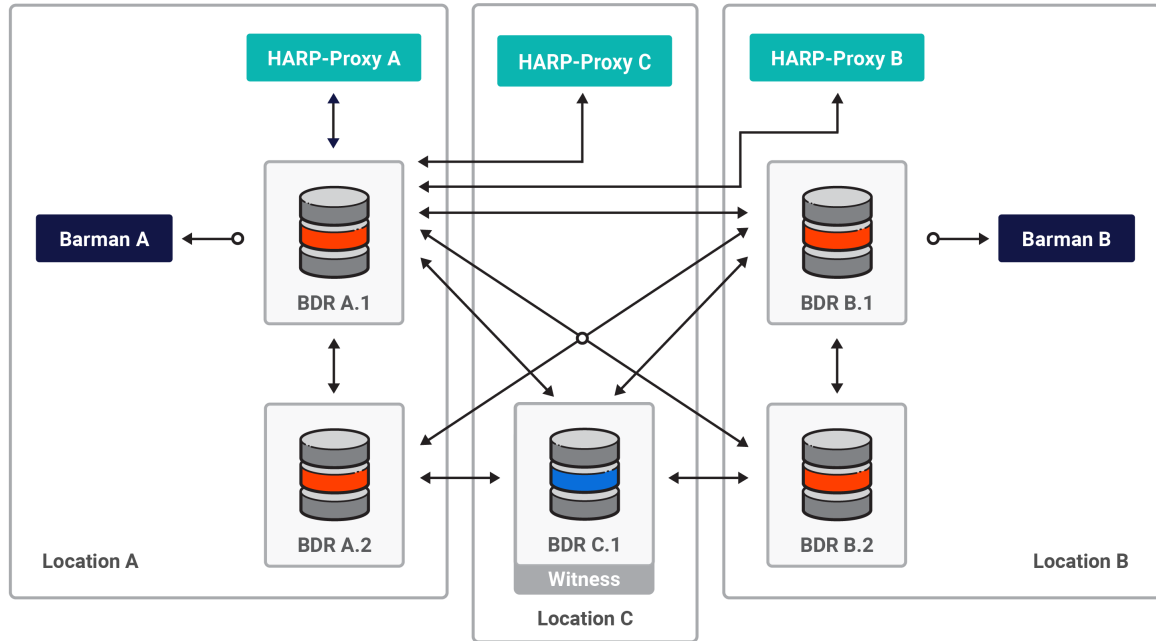
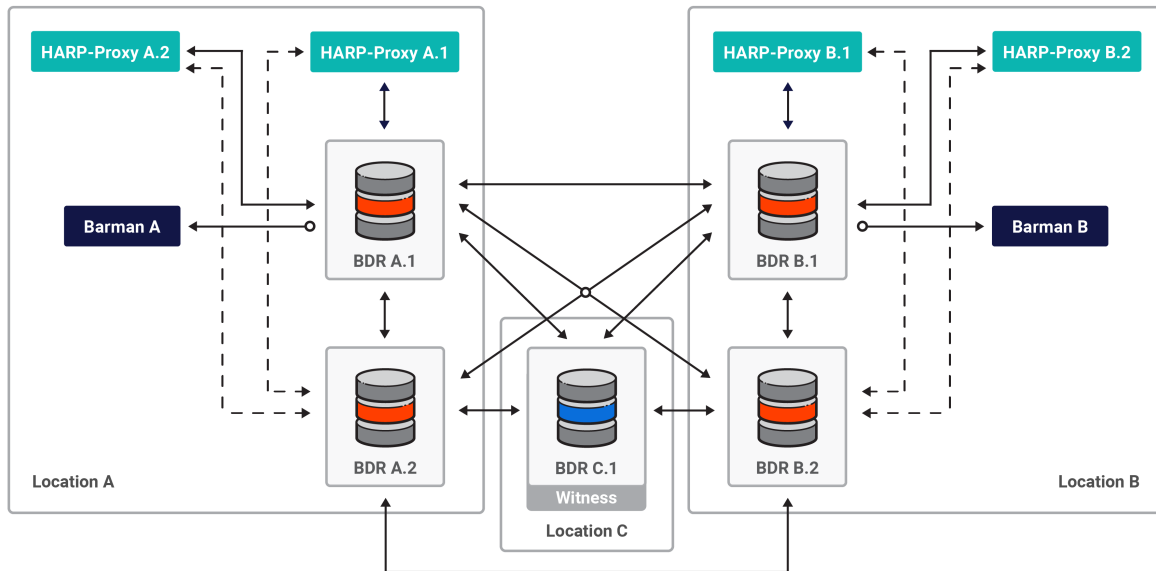
Always On Gold アーキテクチャでは、場所間の移動が必要ですが、すべてのデータはレプリケートされ、フェールオーバーがフルキャパシティで開始されます。

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 2DC, 2 Lead Masters



Always On Gold 3DC, 1 Lead Master**Always On Gold 3DC, 2 Lead Masters****5.3.4 Always On Platinum (two locations; fast HA restoration)**

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

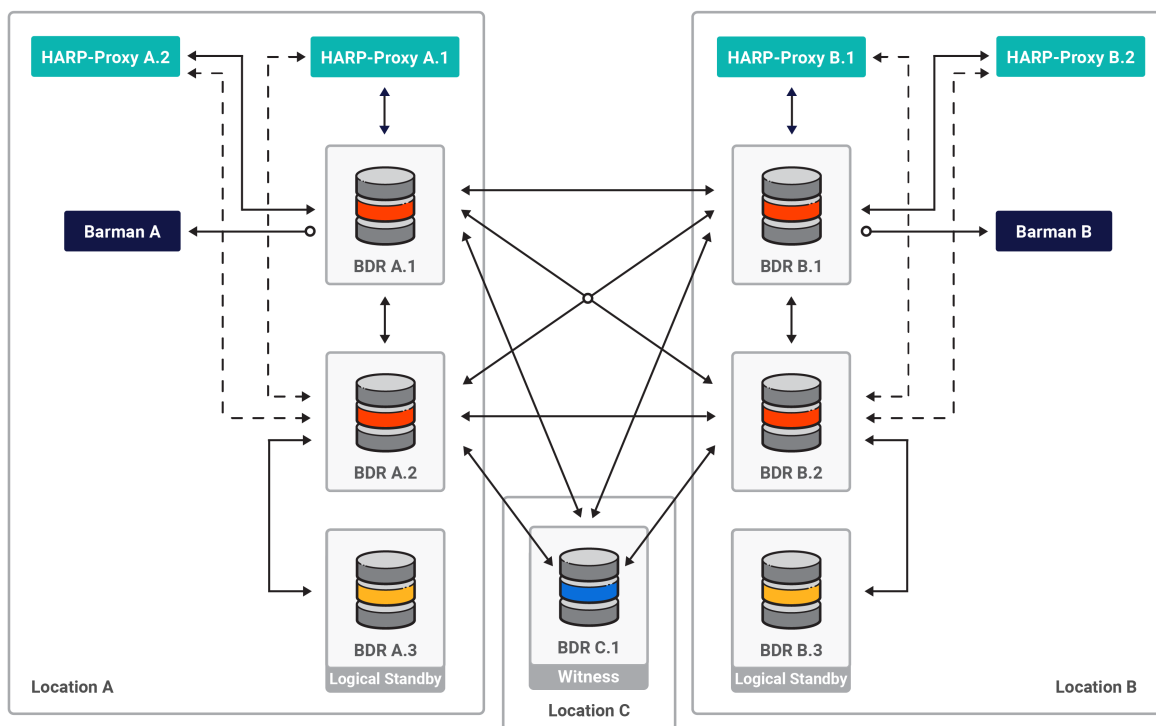


図 3: Always On Platinum architecture

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。使用可能な EDB Postgres Distributed 機能は、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 の処理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。

次の表に、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*
- *BDR の Eager Replication*
- Group Commit

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、CPU とメモリのリソースを適切に割り当てるには、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ためのコントロールを提供します。

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。

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、シングルリージョンの 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 との互換性などの追加の価値や、IBM Power や OpenShift などのサポートされているプラットフォームを提供します。

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に EDB Distributed Postgres 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

クラスターが起動して実行されると、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が立つことはありません。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。


```
yum install tpaexec
``` - Debian/Ubuntu の場合
```

```
apt-get install tpaexec
```

3. 追加の依存関係をインストールします。

```
/opt/EDB/TPA/bin/tpaexec setup ``
```

4. インストールを確認します。edb\_notranlate\_6

### 9.1.5.2 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 構成ファイルを生成します。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 (Bronze、Silver、Gold、Platinum) のいずれかを指定します。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、その後に`--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生成します。 `tpcaexec 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。この名前でも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なサブネット（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われ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` と同じように動作しますが、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、既定のリストで使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-]`) のみに一致し、十分な数のホスト名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使用される場所に `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は、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あり、TPAexec が 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、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完成します。

詳細については、[TPAexec](#) を参照してください。

### 9.1.5.3 Deploying EDB Postgres Distributed

次の手順では、Amazon EC2 を使用して AlwaysOn 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。

```
$ tpaexec configure myedbdpcluster --architecture BDR-Always-ON --layout Silver --
→platform aws --bdr-version 4 --2q --2Q-repositories products/2ndqpostgres/release
```

これにより、`myedbcluster` 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、TPAexec がクラスターの作成に使用する `config.yml` 構成ファイルが含まれます。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/myedbcluster

export TPA_2Q_SUBSCRIPTION_TOKEN=<token>
```

1. クラスターをプロビジョニングします。

```
tpaexec provision myedbcluster
```

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、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。

1. 必要なパッケージを展開し、実際の EDB Postgres 分散クラスターを設定します。

```
tpaexec deploy myedbcluster
```

`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、アップグレードをほとんどゼロで行うことができます。

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、ソフトウェアの混合バージョン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、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）のプロトコルと機能を使用するため、プロトコルの変更が必要な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、プロトコル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 を使用し続ける場合があります。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、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、潜在的に大規模なデータ転送が必要になります。

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使用できません。実際、現在 `bdr_init_physical` では、ソースと参加ノードの BDR バージョンでさえ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変更がいくつか含まれています。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 サーバーを最初に停止して、アップグレード中に予期せず再起動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、新しいバージョンの BDR パッケージをインストールすることです。これにより、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 BDR 拡張機能は、起動時に新しいバイナリが古いバージョンの拡張機能を検出すると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、DDL Replication

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機能させるには、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 [Replicating between nodes with differences](#) を参照してください。

1 つのノードが新しいテーブルを追加する DDL を実行すると、最新の DDL をまだ受け取っていないノードは余分な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、次のクエリで実行されます。

```
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、次のクエリで実行されます。

```
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 を実行する場合、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 TABLE](#) を参照してください。セクション。

### 10.5.1.1 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*` - 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
- 両方のクラスターにピア認証が構成されています。`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 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>
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>

次のページに続く

表 1 – 前のページからの続き

Steps	--check supplied
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>
Connecting to new cluster	<i>skip</i>
Creating replication origin Repeated for each origin	<i>skip</i>
Advancing replication origin Repeated for each origin	<i>skip</i>
Creating replication slot Repeated for each slot	<i>skip</i>
Stopping new cluster	<i>skip</i>

## 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		2	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、BDR クラスタ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善の方法は、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

BDR 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（明らかに多数のコピーなど）する必要があります。
- ただし、各ノードに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

BDR クラスター内のノードは最終的に一貫性があるですが、完全に一貫性があるわけではありません。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 BDR クラスター内の 2 つのノードが同じ一連の状態をたどらない場合がある（そして通常は経過しない）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 は次の状態を経ます。

(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ことは、バックアップ戦略の決定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 (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、オリジンごとに異なるポイントとして到達します。

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、一貫して 1 つのタイムスタンプが適用されます。

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 に復元される複製起点のリストは、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起点の場合、`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、回復はデフォルトで元の PostgreSQL PITR の動作になります。

---

**注釈:** この機能は EDB Postgres Extended でのみ使用でき、現在 Barman は `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 BDR 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、BDR クラスターの現在のコンテンツをクローンして、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 ノードに接続できません。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/書き込みに昇格します。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が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、赤で表示
- 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ます。

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 `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。

ノードが分割されると、クラスター内の一部のノードがその分割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 `bdr.node_slots`
- 着信レプリケーションを監視するための `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。Replication Slots created by BDR 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 `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
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-entepreise` 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送信レプリケーションについて `bdr.node_slots` を照会できます。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
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、クラスター内の遅延を完全に把握するには、参加しているすべてのノードでこのクエリを実行する必要があ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にかかる時間を測定するものではありません。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

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) は、`bdr.subscription_summary` ビューを照会することで監視できます。これは、BDR 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
```

(次のページに続く)

(前のページからの続き)

```

- [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

Decoding Worker 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 logical standby 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。

- 待機の原因が BDR に関連する場合、`wait_event` 列に拡張情報があります。
- 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エラー（ある場合）を表示します。このビューにはアクティブなエラーのみが表示されるため、ワーカーに一時的な問題が発生していたが回復した場合、ビューは空になります。

## 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 `</bdr/latest/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。グローバルな勧告ロックの場合、`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勧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

ユーザー指定のメタデータを保存して、監視ツールが BDR 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。

これらか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、通常、ダウストリームマスターによって更新されます。これは、`autovacuum`

の正しい機能に不可欠です。ダウ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ものと\*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機能する必要があります。Raft Consensus を動作させずに BDR 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、ノードの分割、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 Raft Consensus ステータスを同時に取得します。例：

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
 node_id | node_name | state | leader_id
-----+-----+-----+-----
 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。

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

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）では不十分な場合があります。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直接進めることを考えると、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になっている必要があります。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 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 PGD CLI

EDB Postgres 分散コマンドラインインターフェイス (PGD CLI) は、BDR クラスターを管理するためのツールです。BDR 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し、次に関する情報を取得できます。

- ノードの概要
- 各ノードの BDR/Postgres バージョン
- レプリケーションとサブスクリプションの詳細
- メンバーシップとワーカイベント
- raft コンセンサスの詳細
- CAMO (Commit at Most once) の詳細
- クロックキュー
- クラスター状態の概要

詳細については、EDB Postgres Distributed Command Line Interface を参照してください。