



BDR (Bi-Directional Replication)

リリース *4.0.0*

EnterpriseDB

2021 年 12 月 19 日

目次

第 1 章	概要	1
第 2 章	サポートされている PostgreSQL データベースサーバー	3
2.1	Architectural Overview	3
2.1.1	基本アーキテクチャ	5
2.1.2	アーキテクチャのオプションとパフォーマンス	7
2.1.3	展開	8
2.1.4	クロックとタイムゾーン	8
2.2	Application Usage	9
2.2.1	アプリケーションの動作	9
2.2.2	複製されていないステートメント	11
2.2.3	DML および DDL レプリケーション	11
2.2.4	異なるリリースレベル間の複製	12
2.2.5	相違点のあるノード間の複製	12
2.2.6	違いのあるノード間の比較	13
2.2.7	アプリケーションの一般規則	14
2.2.8	タイミングの考慮事項と同期複製	14
2.2.9	アプリケーションのテスト	14
2.2.10	パフォーマンスのテストとチューニング	18
2.2.11	適合性の評価	19
2.3	PostgreSQL Configuration for BDR	21
2.3.1	BDR の PostgreSQL 設定	21
2.3.2	EDB PG 拡張および EDB PG BDR の詳細設定	22
2.3.3	BDR 固有の設定	22
2.4	Node Management	30
2.4.1	BDR グループの作成と参加	31
2.4.2	監視ノード	33
2.4.3	ロジカルスタンバイノード	34
2.4.4	フィジカルスタンバイノード	35
2.4.5	サブグループ	37
2.4.6	デコードワーカー	38
2.4.7	ノードの再起動とノードの復旧	40
2.4.8	BDR グループからノードを削除する	42
2.4.9	BDR トポロジーのリスト	43
2.4.10	ノード管理インターフェース	45
2.4.11	ノード管理コマンド	55
2.5	DDL Replication	57
2.5.1	DDL レプリケーションオプション	57
2.5.2	BDR システムでの DDL の実行	58
2.5.3	DDL ロックの詳細	59

2.5.4	DDL の影響を最小限に抑える	61
2.5.5	ダウンロードでの DDL の処理	62
2.5.6	ステートメント固有の DDL レプリケーションの懸念	62
2.5.7	BDR DDL コマンド処理マトリックス	63
2.6	Security and Roles	80
2.6.1	カタログ表	81
2.6.2	BDRFunctions と演算子	81
2.6.3	カタログオブジェクトに対する権限の付与	81
2.6.4	ロール管理	81
2.6.5	ロールとレプリケーション	82
2.6.6	接続ロール	82
2.6.7	特権の制限	83
2.6.8	BDR デフォルト/事前定義ロール	84
2.6.9	検証	87
2.7	Conflicts	89
2.7.1	競合の発生方法	90
2.7.2	競合の種類	91
2.7.3	競合の回避または許容	102
2.7.4	競合検出	103
2.7.5	競合の解決	107
2.7.6	競合ログ	110
2.7.7	LiveCompare によるデータ検証	112
2.8	Sequences	113
2.8.1	BDR グローバルシーケンス	113
2.8.2	UUID、KSUUID、およびその他のアプローチ	117
2.8.3	グローバルシーケンス管理インターフェイス	119
2.8.4	KSUUID v2Functions	123
2.8.5	KSUUID v1Functions	124
2.9	Column-Level Conflict Detection	126
2.9.1	列レベルの競合解決の有効化と無効化	127
2.9.2	DDL ロック	128
2.9.3	現在とコミットのタイムスタンプ	129
2.9.4	列のタイムスタンプの検査	129
2.9.5	CRDT データ型を使用した列の競合の処理	131
2.9.6	制限	131
2.10	Conflict-free Replicated Data Types	132
2.10.1	状態ベースおよび操作ベースの CRDT	135
2.10.2	ディスクスペースの要件	136
2.10.3	CRDT タイプと競合処理	137
2.10.4	CRDT タイプと競合レポート	137
2.10.5	CRDT 値のリセット	137
2.10.6	実装された CRDT データ型	138
2.11	Transaction streaming	144
2.11.1	BDR の機能強化	145
2.11.2	設定	145
2.11.3	監視	147
2.12	Durability & Performance Options	148

2.12.1	概要	148
2.12.2	比較	149
2.12.3	操作の内部タイミング	150
2.12.4	設定	151
2.12.5	計画的なシャットダウンと再起動	151
2.12.6	BDR を使用した同期レプリケーション	151
2.13	Eager Replication	152
2.14	CAMO	152
2.15	AutoPartition	152
2.15.1	パーティションの自動作成	152
2.16	Timestamp-Based Snapshots	159
2.17	Replication Sets	160
2.17.1	レプリケーションセットの使用	161
2.17.2	パーティションテーブルの動作	161
2.17.3	外部キーの動作	162
2.17.4	レプリケーションセットの管理	163
2.17.5	レプリケーションセットメンバーシップ	167
2.17.6	複製セットのリスト	169
2.17.7	DDL レプリケーションフィルタリング	170
2.18	Stream Triggers	173
2.18.1	適用中のトリガー実行	174
2.18.2	行タイプの用語	175
2.18.3	競合のトリガー	176
2.18.4	変換トリガー	177
2.18.5	ストリームトリガー変数	178
2.18.6	情報関数	179
2.18.7	行の内容	182
2.18.8	トリガーノート	182
2.18.9	ストリームトリガー操作インターフェイス	182
2.18.10	ストリームトリガーの例	185
2.19	Backup and Recovery	188
2.19.1	バックアップ	188
2.19.2	復元	191
2.20	Application Schema Upgrades	193
2.20.1	概要	193
2.20.2	アップグレード計画	193
2.20.3	アップグレードの準備	196
2.20.4	サーバーソフトウェアのアップグレード	197
2.20.5	データベースのエンコード	198
2.20.6	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	198
2.21	Explicit Two-Phase Commit (2PC)	199
2.21.1	使用法	200
2.22	Catalogs and Views	200
2.22.1	ユーザーに表示されるカタログと Views	200
2.22.2	内部カタログと Views	238
2.23	Monitoring	240
2.23.1	監視の概要	240

2.23.2	ノードの参加と削除の監視	240
2.23.3	レプリケーションピアの監視	242
2.23.4	BDR レプリケーションワーカーの監視	245
2.23.5	BDR ライターの監視	246
2.23.6	グローバルロックの監視	246
2.23.7	競合の監視	247
2.23.8	外部モニタリング	247
2.23.9	統計を適用する	249
2.23.10	標準の PostgreSQL 統計 Views	250
2.23.11	BDR バージョンの監視	251
2.23.12	ラフトコンセンサスの監視	251
2.23.13	レプリケーションスロットの監視	253
2.23.14	トランザクションのコミットの監視	254
2.24	BDR System Functions	254
2.24.1	バージョン情報 Functions	255
2.24.2	システムおよび進捗情報のパラメーター	255
2.24.3	ユーティリティ Functions	256
2.24.4	グローバルアドバイザーロック	263
2.25	Credits and Licence	265
2.26	' Appendix A	266
2.26.1	BDR 4.0.0	266
2.27	' Appendix B	268
2.28	' Appendix C	268
2.29	' Appendix D	269
2.29.1	データの一貫性	269
2.29.2	問題のリスト	269
2.30	' Appendix E	270
2.30.1	LLVM	270
2.30.2	OpenSSL	270
2.30.3	元の SSLeay ライセンス	271
2.30.4	PostgreSQL ライセンス	272
2.31	CAMO Clients	272

第1章 概要

BDR は、高度な競合管理、データ損失保護、およびネイティブロジカルレプリケーションよりも最大 5 倍高速な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配信を提供する PostgreSQL 拡張機能であり、最大 59 秒の高可用性を備えた分散 PostgreSQL クラスターを可能にします。

BDR の仕組みに関する詳細な概要は、[Architectural Overview](#) の章で説明されています。

第2章 サポートされている PostgreSQL データベースサーバー

BDR は PostgreSQL、 EDB Postgres Extended、 および EDB PostgresAdvanced フレーバーの PostgreSQL データベースサーバーと互換性があり、標準の PG 拡張として展開できます。完全な互換性マトリクスは次のとおりです。

BDR Version	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
4.0.0	14.1	14.1	14.1
4.0.0	13.5	13.5r1.1.7	13.5
4.0.0	12.9	12.9r1.1.9	12.9
3.7.13.1	13.5	13.5r1.1.7	13.5
3.7.13.1	12.9	12.9r1.1.9	12.9
3.7.13.1	11.14	11.14r2.1.9	11.14

一部のキー BDR 機能は、対象となる PostgreSQL データベースサーバ内で利用可能な特定のコア機能に依存することにノートすることが重要です。したがって、BDR のお客様は、ビジネスニーズに最適な PostgreSQL database サーバーフレーバーも採用することが不可欠です。たとえば、BDR 機能「Commit At Most Once (CAMO)」を使用することが BDR 顧客のユースケースにとってミッションクリティカルな場合、CAMO の処理に必要なコア機能がないため、コミュニティ PostgreSQL フレーバーを採用しないでください。機能マトリクスの完全な互換性は、付録 `eded_tran1` にあります。

BDR は、ほぼネイティブの PostgreSQL 互換性を提供します。ただし、一部のアクセスパターンは、単一ノードのインスタンスと同じように、マルチノードセットアップで必ずしもうまく機能しません。マルチノード設定で安全に複製できるものには、いくつかの制限もあります。[Application Usage](#) の章では、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。

2.1 Architectural Overview

BDR は、メッシュトポロジを使用した疎結合マルチマスターロジカル複製を提供します。つまり、任意のサーバーに書き込むことがパート、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ます。

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同等なに変更を適用します。オプションの `eager all node replication` 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。



図 1: node diagram

2.1.1 基本アーキテクチャ

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 クラスター全体に対して単一のノードグループがあります。

複数のマスター

BDR グループに参加している各ノード（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または別のスタンバイから複製するスタンバイであるホットスタンバイ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手紙を書く必要はありません。ほとんどの場合、1 つのマスタのみに書き込みを指示する構成です。

非同期、デフォルト

1 つの BDR ノードで行われた変更は、ローカルでコミットされるまで他のノードに複製されません。その結果、データは常にすべてのノードで正確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着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になります。BDR では、マルチプルのマスターがあり、結果としてマルチプルのデータストリームがあるため、`synchronous_commit` と `synchronous_standby_names` が使用されている場合でも、異なるノード上のデータは異なる場合があります。

メッシュトポロジ

BDR は、すべてのノードが他のすべてのノードに接続し、すべてのノードが互い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 内でのデータの転送はありません。データは BDR クラスターの外部から到着するか、ネイティブの PostgreSQL ロジカルレプリケーションを使用して先に送信される場合があります。

論理複製

論理レプリケーションは、レプリケーション ID（通常はプライマリキー）に基づいてデータ行とその変更をレプリケートするメソッドです。正確なブロックアドレスとバイト単位の **物理的** レプリケーションとは対照的に、用語 **ロジカル** を使用します。レプリケーション。インデックスの変更は複製されないため、書き込み増幅が回避され、帯域幅が削減されます。

ソースノードからデータの snapshot をコピーすることにより、論理レプリケーションが開始されます。それが完了すると、後のコミットがリアルタイムで発生するときに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が行われたオーダーでデータを適用し、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を保証します。異なるノードからの変更は、他のノードとは独立して適用され、変更の迅速なレプリケーションを保証します。

複製されたデータは、安全な場合はバイナリフォームで送信されます。

高可用性

各マスタノードは1つ以上のスタンバイノードで保護できるため、ダウンしたノードはすぐに交換して続行できます。各スタンバイノードは、ロジカルスタンバイノードまたは物理的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、現在接続されているノード間でレプリケーションが続行されます。ノードが回復すると、変更を失うことなく、レプリケーションを中断したところからリスタートできます。

ノードは異なるリリースレベルを実行して、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、データベースソフトウェアのメジャーバージョンでも、BDR クラスターは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自動的に複製されます。必要に応じて、DDL の実行をユーザ制御して、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、十分マルチプルロジカルハードウェアとネットワーク上で数百のノードを実行できますが、マスタベースの展開では、1つのクラスターで32をスタンバイノードを実行することは一般物理的に推奨されません。スタンバイノードの数ですが、一般的な使用法は、マスタごとに2~3個のスタンバイにすることです。スタンバイノードはメッシュネットワークに追加の接続を追加しないため、32ノードの推奨事項には含まれません。

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、アクティブノードが1000個以下というハードリミットがあります。

BDR は、1つの PostgreSQL インスタンスで最大10個のデータベースを異なる 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、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、BDR が最適に機能します。

BDR クラスターの最小推奨ノード数は3です。これは、2つのノードでは、ノードの1つが機能しなくなるとコンセンサスが機能しなくなるためです。

2.1.2 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性評価

BDR は、さまざまなアーキテクチャで構成できます。各アーキテクチャには、異なるパフォーマンスとスケラビリティの特性があります。

グループは、2つ以上のノード（サーバー）で構成される BDR グループの基本的な構成要素です。グループ内では、各ノードは専用のルーターとバックアップを備えた異なる AZ にあり、即時のスイッチオーバーと高可用性を実現します。各グループには、専用のレプリケーションセットが定義されています。グループがノードを失った場合、既存のノードをグループからコピーすることで簡単に修理/交換できます。

BDR グループにマスタノードを追加しても、ほとんどのテーブルがレプリケートされる場合、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、書き込みスループットが大幅に増加することはありません。一般に、BDR 書き込みは、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、パフォーマンスをある程度向上させることができます。通常、読み取りスループットはノード数に比例して増加します。

次のアーキテクチャが利用可能です。

- マルチマスタ/シングलगループ BDR AlwaysOn

最も単純なアーキテクチャは、1つのグループを持つだけなので、最初に調べてみましょう。

1つのグループ内の BDR MultiMaster

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されます。

データのコピーはどこにでもあるので、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の影響を受けません。したがって、ノードを追加すると、可能な SELECT スループットの合計が直線的に増加します。

INSERT、UPDATE、および DELETE（DML）はローカルで実行され、変更はグループ内のすべてのノードに伝播されます。DML 適用のオーバーヘッドは元の実行より小さいため、マルチプル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と、マルチノードクラスターは単一ノードよりも多くの TPS をハンドルできます。

競合ハンドリングには、スループットを低下させるコストがかかります。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依存します。競合が非常に低いアプリケーションは、単一ノードノードもマルチマスタが低下する可能性があります。それらは BDR のファセット特性ではありません。

積極的なレプリケーションは競合を回避できますが、本質的にはより高価です。

変更はすべてのノードに同時に送信されるため、レプリケーションラグが最小限に抑えられます。ノードを追加すると、レプリケーションに CPU が使用されるため、新しいノードが追加されるたびにピーク TPS がわずかに減少します。

ワークロードがすべての CPU リソースを使用しようとすると、リソース制約制約が発生し、レプリケーションラグに影響する可能性があります。

BDR AlwaysOn

AlwaysOn アーキテクチャは、2つの別々の地域にある2つのグループから構築されます。各グループはHAとISを提供しますが、一緒にディザスタリカバリー（DR）も提供するため、このアーキテクチャをAlwaysOn with Very High Availability と呼びます。

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

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」）として記述され、必要なときに引き継ぐのを待っています。nodelos のコンタクトが発生した場合、処理を続行するためにすぐにシャドウノードにスイッチします。aGroup に障害が発生した場合、他のグループにスイッチことができます。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、競合が発生する可能性はほとんどゼロになり、結果としてパフォーマンスへのインパクトは大幅に減少します。

CAMO は、ローカルグループ内での熱心なレプリケーションであり、他のグループに関しては怠りです。

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した場合、これらを減らすか中断する必要があります。

将来の機能：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伴う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化されます。

2.1.3 展開

BDR は、TPAexec または構成管理アプローチのいずれか、およびテクニカルサポートによって承認された展開アーキテクチャを使用して、少数の既知の良好な構成のいずれかに展開す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。

アーキテクチャについては、TPAexec Architecture User Manual を参照してください。

現在、ログメッセージと文書は英語でのみ利用可能です。

2.1.4 クロックとタイムゾーン

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おり、ほぼ全世界のデータベースクラスタを可能にします。個々のサーバーをマッチングタイムゾーンで構成する必要はありませんが、人間が読めるサーバーログがよりアクセスしやすく比較しやすいように `log_timezone = UTC` を使用することをお勧めします。

サーバークロックは、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。

クロック同期は、他のソリューションの場合のように、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 Origin Conflict Detection にインパクトを与える可能性がありますが、BDR は存在するスキューをレポートおよび管理するためのコントロールを提供します。BDR は、[Conflict Detection](#) で説明されているように、行バージョンの競合検出も提供します。

2.2 Application Usage

この章では、アプリケーションまたはユーザの観点から BDR を検討します。

ノードのセットアップについては、後の章で説明します DDL のレプリケーションや、レプリケーションを制御するためのさまざまなオプションについても同様です。

2.2.1 アプリケーションの動作

BDR は、1 つのノードで行われた変更を他のノードに複製することをサポートします。

デフォルトでは、BDR は INSERT、UPDATE、DELETE、TRUNCATE からのすべての変更をソースノードから他のノードに複製します。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。例、INSERT ... ON CONFLICT UPDATE は、オリジンで発生した内容に応じて INSERT または UPDATE を送信します。UPDATE または DELETE 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。

UPDATE および DELETE を他のノードに複製するには、影響を受ける一意の行を識別する必要があります。BDR では、テーブルに PRIMARY KEY が定義されているか、UNIQUE 制約があるか、特定の列で明示的な REPLICATION ID が定義されている必要があります。そ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、後で UPDATE または DELETE が明示的にブロックされます。テーブルに REPLICATION ID FULL が定義されている場合、一意インデックスは不要です。その場合、UPDATE および DELETE は許可されます。そして、ライブで、有効で、遅延ず、式または WHERE 句を持たない最初の非ユニークインデックスを使用します。そうでない場合は、シーケンシャルスキャンが使用されます。

TRUNCATE は、定義されたレプリケーション ID がなくても使用できます。TRUNCATE コマンドのレプリケーションはサポートされていますが、外部キーで接続されたテーブルのグループを切り捨てるときは注意が必要です。切り捨てアクションを複製する場合、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、明示的に指定または CASCADE を介して暗黙的に収集された、オリジンで切り捨てられた同じテーブルグループを切り捨てます。詳細と例については、章を参照してください。影響を受けるすべてのテーブルは、同じサブスクリプションのパートです。ただし、サブスクライバーで切り捨てられるテーブルに、同じ（またはいずれかの）レプリケーションセットのパートではないテーブルへの外部キーリンクがある場合、サブスクライバーでの切り捨てアクションの適用は失敗します。

INSERT、UPDATE、および DELETE コマンドによって暗黙的に取得された行レベルのロックは、変更が行わ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複製されます明示的行レベルロック (SELECT ... FOR UPDATE / FOR SHARE) はユーザセッションによって複製されず、アドバイザリロックも複製されません。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、他のノードに複製されません。SERIALIZABLE のトランザクション隔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で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 がマルチプルのノードで同時に実行される場合、非同期レプリケーションで実行する場合、潜在的な競合が発生する可能性があります、これら処理または回避する必要があります。さまざまな回避メカニズムが可能です。Conflicts の章でも説明されています。

シーケンスには、Sequenceschapter で説明されている特別なハンドリングが必要です。

BYTEA 列のバイナリデータは通常どおり複製され、最大 1GB のデータの「プロブ」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 ではサポートされていません。

ルールはオリジンノードでのみ実行されるため、レプリカに対して有効になっている場合でも、適用中には実行されません。

レプリケーションは、ベーステーブルからベーステーブルにのみ可能です。つまり、サブスクリプション側のソースおよび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、テーブルでなければなりません。ベース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。更新可能ビューを介して行われた DML 変更は、オリジンの tobase テーブルを介して解決され、ターゲットの同じベーステーブル名前に適用されます。

BDR はテーブルパーティション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、パーティションに関係する変更がダウンストリームにレプリケートされます。

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。例、INSERTtrigger はオリジンノードで実行され、ターゲットノードで変更を適用すると無視されます。ALTER TABLE ... ENABLE ALWAYS TRIGGER を使用して、トリガーを実行時の起点ノードとターゲット（レプリケート時）の両方で実行するように指定するか、REPLICA オプションを使用して、適用時のみ ALTER TABLE ... ENABLE REPLICA TRIGGER を実行できます。

一部のタイプのトリガーは、atable 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

- ステートメントレベルのトリガー（FOR EACH STATEMENT）
- 列ごとの UPDATE トリガー（UPDATE OF column_name [, ...]）

BDR レプリケーションの適用では、システムレベルのデフォルトの search_path が使用されます。Replica-triggers、ストリームトリガー、およびインデックス式関数は、適用時に実行と失敗する他のスキーマデフォルトを前提と保証場合があります。、または影響を受ける関数の ALTER FUNCTION ... SET search_path = ... を使用して関クションの検索パスを設定します。

BDR は、テキストまたはその他のデフォルト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ることに注意して照合順序。変更のレプリケーションでは、等値検索を使用してレプリカ ID 値を特定します。したがって、一致しない照合順序修飾子で一意性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ている場合、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 内の非常に長い「トースト」データの BDR ハンドリングは、ユーザ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異なる可能性が高いことに注意してください。ただし、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 は正しく機能できません。

PostgreSQL は、volatile 関数を含む CHECK () コンストレインを許可します。BDR は適用時に CHECK () コンストレインを再実行するため、以前と同じ結果を結果ない後続の再実行はデータの相違を引き起こします。

BDR は外部キーの使用を制限しません。カスケード FK は許可されます。

2.2.2 複製されていないステートメント

以下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、それらの影響はローカル/オリジンノードでのみ発生します。

- カーソル操作 (DECLARE、CLOSE、FETCH)
- 実行コマンド (DO、CALL、PREPARE、EXECUTE、EXPLAIN)
- セッション管理 (DEALLOCATE、DISCARD、LOAD)
- パラメーターコマンド (SET、SHOW)
- 制約の操作 (SET CONSTRAINTS)
- ロックコマンド (LOCK)
- テーブル保守コマンド (VACUUM、ANALYZE、CLUSTER、REINDEX)
- 非同期操作 (NOTIFY、LISTEN、UNLISTEN)

NOTIFY SQL コマンドと `pg_notify()` 関数はレプリケートされないため、フェイルオーバーの場合、通知は信頼できないことになります。これは、サーバーがクラッシュした時点でトランザクションがコミットされた場合、フェイルオーバーオーバーで通知が簡単に失われる可能性があることを意味します。フェイルオーバーの場合に通知を見逃す可能性があります。これは標準の PostgreSQL レプリケーションでは残念ながら真実であり、BDR はまだこれを改善していません。CAMO および Eager レプリケーションオプションは、NOTIFY SQL コマンドまたは `pg_notify()` ファンクションを許可しません。

2.2.3 DML および DDL レプリケーション

BDR は DML ステートメントを複製せず、DML ステートメントによって引き起こされた変更を複製することに注意してください。したがって、例、2つの行を変更した UPDATE は2つの変更を複製しますが、行を削除しなかった DELETE は何も複製しません。これは、揮発性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ないことを保証します。

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、それがすべてのノードで実行されます。そのため、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、ステートメントはまだ他のノードに送信されて実行 DDL レプリケーションします。

BDR は、同じトランザクション内であっても、混合された DML および DDL ステートメントが正しく機能することを保証に非常に長くなります。

2.2.4 異なるリリースレベル間の複製

BDR は、異なるメジャーバージョンの PostgreSQL を持つノード間で複製するように設計されています。これは、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た機能です。

BDR は、異なるバージョンの BDR ソフトウェアを持つノード間で複製するようにも設計されています。これは、ダウンタイムなしでバージョンをアップグレードおよびメンテナンスできるように設計された機能です。

ただし、クラスター内のノードをメジャーバージョンに結合させることは可能ですが、クラスターが新しいプロトコルバージョンを使用している場合、マイナーバージョンのノードを追加することはできません。これはエラーを結果めます。

上記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、リリースノートで説明します。

2.2.5 相違点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。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。これは、ノード間でデータベーススキーマの違いを作成するために使用できます。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したり、レポートまたはテスト用に論理スタンバイノードを許可したりするように設計されています。

現在、レプリケーションでは、すべてのノードで同じテーブル名前が必要です。futurefeature は、異なるテーブル名間のマッピングを許可するかもしれません。

ターゲットのパーティションを変更する更新のサポートを含む、分割された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することが可能です。ソースとターゲットのパーティショニング定義が同じ場合、適用時に動的パーティションルーティングを実行する必要がないため、高速になります。詳細については、レプリケーションセットの章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、列名前に基づいてデータ列を複製します。列の名前が同じでデータ型が異なる場合、それを許可するキャストが定義されていれば、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` の競合を発生させます。この競合については、デフォルトの競合が `ignore_if_null` を解決します。NULL 以外の値が到着した場合、これは **ERROR** をスローします。または、ノードは競合リゾルバ `ignore` を使用して構成することもできます。この設定は **ERROR** をスローせず、単に追加列を無視します。

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` の競合を発生させます。この競合のデフォルトの競合は `use_default_value` です。追加の列にデフォルト、NULL (null 可能の場合) またはデフォルト式がある場合、レプリケーションは続行しますが、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。

また、変換トリガーをテーブルで使用して、デフォルトを提供したり、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のコンストレインが異なる場合、レプリケーションが試行されますが、ソースからの行をターゲットに適用できない場合は失敗する可能性があります。ここで行フィルターがヘルプ場合があります。

あるスキーマからより緩和されたスキーマにデータを複製してもエラーは発生しません。スキーマからより制約的なスキーマにデータを複製すると、潜在的な障害のソースになります。したがって、不正なデータが入力されることはありません。これにより、レプリケーションを介して不正なデータが届くことはなないため、より制限的なスキーマへの変換に失敗することはありません。例、あるスキーマに TEXT 型の列があり、別のスキーマに XML と同じ列が定義されている場合、テキストが XML であることを強制する CHECK 制約を TEXT 列に追加します。

テーブルは、各ノードで異なるインデックスを使用して定義できます。デフォルトでは、インデックス定義が複製されます。DDL Replication を参照して、ノードのサブセットのみで、またはローカルでのみインデックスを作成する方法を指定します。

fillfactor や toast_tuple_target などのストレージパラメータは、テーブルのノード間で問題なく異なる場合があります。例外は、テーブルのストレージパラメータ user_catalog_table の値がすべてのノードで同一でなければならないことです。

複製されるテーブルは、各ノードの同じユーザ/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、ロールの変更は各ノードに複製されますが、ロールには各ノードの接続用に異なるパスワードが設定されている場合があります。DDL Replication を参照して、ノードのサブセットのみで、またはローカルでのみロールパスワードを変更する方法を指定します。

2.2.6 違いのあるノード間の比較

Livecompare は、BDR および非 BDR ノードに対するデータベースのデータ比較に使用されるツール。比較し、最終結果を得るには、最低 2 つの接続が必要です。

Livecompare 1.3 から、all_bdr_nodes を設定して構成できました。これにより、クラスター内の各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。BDR 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+ がジョブを完了するためにニーズ初期接続と出力接続のみです。logical_replication_mode を設定すると、すべてのノードの通信方法が示されます。

すべての構成は、例 bdrLC.ini 名前付けの .ini ファイル内で行われます。この構成ファイルのテンプレートは、パッケージのインストール後に配置された /etc/2ndq-livecompare/ の場所に表示されます。

LiveCompare の実行中に、N + 1 個のプログレスバーが表示されます。N はプロセスの数です。すべてのテーブルが取得されると、1 秒あたりのトランザクション (tps) が測定されるため、時間が表示されます。これは時間をカウントし続け、推定値を提供し、最後に合計実行時間を提供します。

このツールには、多くのカスタマイズとフィルターがあります。テーブル、スキーマ、replication_sets など。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、ユーザが確認できるように、要約と DML スクリプトが生成されます。見つかった違いがある場合は、修正するために DML を適用してください。

2.2.7 アプリケーションの一般規則

上記で説明したように、BDR はレプリカアイデンティティ値を使用して、変更する行を識別します。アプリケーションは、同じ一意の識別子を挿入、削除し、後で再利用すると問題を引き起こす可能性があります。これは ABA 問題と呼ばれます。BDR は、行が現在の行、最後の行、またははるかに古い行であるかどうかを知ることができません。< https://en.wikipedia.org/wiki/ABA_problem を参照してください。

同様に、BDR は変更をリプレイするテーブルを識別するためにテーブル名を使用するため、同じオブジェクト名を作成してからドロップし、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シンプルルールが生じます。

1. 行に一意の識別子を使用します (INSERT) 2. 一意の識別子の変更を避ける (更新) 3. 削除された一意の識別子の再利用を避ける 4. ドロップされたオブジェクト名の再利用を避ける

一般的な場合、これらの規則を破ると、データの異常と相違が生じる可能性があります。特定の条件が満たされている限り、アプリケーションはこれらの規則に違反する可能性があります。注意が必要です。異常が発生することはほとんどありませんが、不可能ではありません。例、DELETE がダウロードを含むすべてのノードで再生されている限り、行の値を再利用できます。これは通常 1 秒未満で発生する可能性があります。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、潜在的に数日かかる可能性があります。

2.2.8 タイミングの考慮事項と同期複製

デフォルトでは非同期であるため、同等な、マルチプルの BDR ノードに接続されているクライアントや、それらの間で切り替えて古いデータを読み取ることができるようにするために遅れることがあります。

`queue wait function` は、そ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提供されます。

Postgres の同期レプリケーション機能は、BDRas でも利用できます。さらに、BDR は、より多くの同期複製のためにマルチプル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異なるモードの概要と比較については、[Durability & Performance Optionschapter](#) を参照してください。

2.2.9 アプリケーションのテスト

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

- [TPAexec](#)
- [CAMO / Failover オプション付きの pgbench]
- [マルチノードアクセスのアイソレーションテスター]

TPAexec

TPAexec は、Postgres- BDR に基づくものを含むリファレンス TPAarchitectures を展開するために EDB によって使用されるシステムです。

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、次の例のような構文を使用して、TPA クラスターに対して実行されるテストのローカルコレクションの作成と管理を簡素化します。

```
tpaexec test mycluster mytest
```

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

CAMO /フェイルオーバーオプションを備えた pgbench

pgbench が拡張れ、CAMO または通常の BDR 展開を使用しながら、ユーザーがフェイルオーバーテストを実行できるようになりました。次の新しい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
```

上記のオプションに加えて、フェイルオーバーのピアノードに関するノード情報を `DSNform` で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、フェイルオーバーをテストできます。通常の BDR 展開。
- `--retry` を使用して、次の場合にトランザクションを再試行するかどうかを指定します。フェイルオーバーは `-m failover` モードで発生します。これはデフォルトで有効になっています `-m camo` モードの場合。

CAMO 環境での呼び出し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"
```

上記のコマンドは `camo` モードで実行されます。 `node1` に接続し、テストを実行します。 `node1` 接続への接続が失われた場合、pgbench は `node2` に接続します。 `node2` をクエリーして、実行中のトランザクションのステータスを取得します。中止および実行中のトランザクションは、 `camo` モードで再試行されます。

`failover` モードでは、 `--retry` が指定されている場合、実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、処理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の分離テスター

isolationtester は、ユーザーが複数のセッションとマルチプルのノードでテストを実行できるように拡張しました。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に実行される一連のテストです。これらのテストでは、マルチプルの相互作用するトランザクションを実行する必要があるため、マルチプルの同時接続を管理する必要があります。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」名前は、オリジナルの動機がシリアライズ隔離レベルをテストすることであったという事実由来しています。しかし、他のソートの並行動作のテストも追加されています。

外部モジュールとして PGXS を使用して構築されます。インストール時に、`bypg_isolation_regress` によって実行される `isolationtester` バイナリファイルを作成し、並行リグレーションテストを実行して結果を観察します。

`pg_isolation_regress` は `pg_regress` に似たツールが、`psql` を使用してテストを実行代わりに、`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、これらのホスト `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 `spec` ファイルの各セッションで指定された名前と比較し、指定されたテストを各サーバーで実行します。

重複するトランザクションでテストを定義するには、次のセクションで説明する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、`specs` /サブディレクトリに `spec` ファイルを配置し、`expected` /サブディレクトリに `expectedoutput` を追加し、メークファイルにテストの名前を追加します。

`Isolationtester` は、`libpq` を使用してマルチプルの接続を開き、`spec` 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、次のオーダーでテストされた 5 つの部分で構成されています。

`server "<name>"`

- これは、セッションが実行されるサーバーの名前を定義します。サーバーの指定は 0 個以上にすることができます。名前に対応する `conninfo` は、`isolationtester` を実行するコマンドを介して提供されます。これは `quickstart_isolationtest.md` で説明されています。このパートはオプション。

`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 1 回実行されます。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。このパートはオプション。必要に応じて、複数のセットアップブロックを使用できます。それぞれは、指定されたオーダーで

個別に実行されます。(マルチプル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 送信として実行され、VACUUM などの一部のステートメントがそのようなブロック内の他のステートメントと結合できないためです。)

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

- 分解 SQL ブロックは、テストの終了後に 1 回実行されます。これを使用して、セットアップによって作成されたテストテーブルを削除するなど、次の順列に備えてクリーンアップします。このパートはオプション。

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

- 通常、spec ファイルにはいくつかの「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、分解、および 1 つ以上の「ステップ」の 3 つの部分で構成されます。セッションごとのセットアップとティアダウンの部分の構文は、上記のテストごとのセットアップとティアダウンと同じですが、各セッションで実行されます。通常、セットアップパートには、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。
- さらに、セッションパートも `connect_to` 仕様で構成されます。これは、このセッションが実行されるサーバーを示す最初に指定されたサーバー名前を指します。
- `connect_to "<name>"`
- 各ステップには構文があります
- `step "<name>" { <SQL> }`
- ここで、<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、spec ファイル全体で一貫である必要があります。

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

- 順列行は、そのオーダーで実行されるステップのリストを指定します。順列行はいくつでも表示できます。順列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からステップの可能なすべての順序付けを自動的に生成します（1 つのセッションのステップをオーダーに実行します）。手動で指定された「順列」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順列である必要はないことに注意してください。インスタンス、いくつか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と見なされます。

セッションステップの順列（仕様ファイルで手動で指定されるか、自動生成されるか）ごとに、isolationtester はメインセットアップパート、セッションごとのセットアップパート、選択されセッションステップ、セッションごとの分解を実行します。maintardown 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。

すべての前提条件 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、以下の手順に従います。

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、アイソレーションリグレーションテストを実行できます。

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

A. `isolationcheck-installcheck` を実行するには、2 つ以上の `postgreslservers` を実行する必要があります。サーバーの `conninfo` を BDR メークファイルの `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'`

ここで、テストを含む `.spec` ファイルを `bdr-private / repo` の `specs / isolation` ディレクトリに追加します。
`bdr-private / repo` 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

B. `Isolationcheck-makecheck` は現在、マルチプルのデータベース間で BDR を設定することにより、単一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。

適切なデータベース名、`bdr` インスタンスの `conninfos` を次のように BDR メークファイルの `pg_isolation_regress` に渡す必要があります。`pg_isolation_regress --dbname=db1,db2 --server 'd1=dbname=db1' --server 'd2=dbname=db2'`

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が含まれる場合があります（おそらく、他のセッションはブロックを解除するかデッドロックを引き起こすステップを実行します）。この機能を使用するテストでは、有効な置換、つまりブロックされたセッションがコマンドを実行ことを期待しないもの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、合計 375 秒の待機時間の後、`isolationtester` は異常終了します。無効な置換のテストは、分離テストの実行に非常に長い時間がかかる可能性があり、有用なテスト目的に役立たない `make`、避ける必要があります。

`isolationtester` は、`pg_locks` ビューでコマンドが待機中として表示されているかどうかを確認することで、コマンドがブロックされたことを認識することに注意してください。したがって、重いロックのブロックのみが検出されます。

2.2.10 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、マルチプルのマスタノードに書き込みトランザクションを発行できコスト。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することに注意してください。

最初に、別のノードから変更をリプレイすると、CPU コスト、I/O コストが発生し、WAL レコードが生成されます。SQL を再実行する必要がないため、CPU オーバーヘッドが低くなるため、通常、リソースの使用量はオリジナルのトランザクションより少なくなります。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、リプレイに I/O コストがかかる場合があります。

次に、変更を再生すると、ローカルワークロードに対する競合を引き起こす可能性のあるテーブルレベルおよび行レベルロックが保持されます。CRDT（競合のないレプリケートされたデータ型）および CLCD（列レベルの競合検出）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する場合は、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。大きなトランザクション内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にいくつかの考えが必要です。

パフォーマンスに問題があると思われる場合は、上記の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することをお勧めします。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記述できます 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。

したがって、「BDR の実行速度が遅い」場合は、次のことをお勧めします。

1. pgbench のカスタムテストスクリプトを、稼動システムの問題 case にできる限り近づけて make します。1つのノードでスクリプトを実行して、ベースラインの図を取得します。1つのノードで行ったのと同じ数のセッションを使用して、稼動で発生する数のノードでスクリプトを実行します。これにより、マルチプルのノードに移動した場合の効果が表示されます。上記2つのテストのセッション数を増やして、競合の増加がアプリケーションに与える影響をプロットできるようにします。テストがレプリケーションの遅延をアカウントするのに十分な長さであることを確認します。テスト中にレプリケーションの遅延が増大しないようにしてください。

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。

2.2.11 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すでに、または簡単に変更して BDR 準拠にすることができます。ユーザーは、自分のアプリケーションを BDR 対応セットアップに向けることができる評価アクティビティを実行できます。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を支援します。

主キー/レプリカ ID の更新の評価

現在、BDR は、PRIMARY KEY が UPDATE オペレーションによって変更される場合、競合解決を実行できません。プライマリキーを更新することは許可されていますが、既存の値と競合しないようにする必要があります。

EDB Postgres Extended で実行する場合、BDR は次の構成パラメーターを提供して、テーブルのプライマリキー/レプリカアイデンティティが更新操作の対象となる頻度を評価します。

これらの構成パラメーターは、評価のみに使用する必要があることに注意してください。これらは単一ノード 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されている productionBDR 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、ノードのスタートに失敗するか、新しいノードがクラスターに結合できません。

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

評価期間中にこのパラメータを有効にすると、行のキー/レプリカアイデンティティ値の更新を記録できログ。必要に応じて、このような更新をブロックすることもできます。例えば

```
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
```

適用ワーカプロセスは、このパラメータの設定を常に無視します。

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サブジェクトます。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によって、または相互に保持されているロックを待機する場合があります。

EDB Postgres Extended で実行する場合、BDR は次の構成パラメータを提供して、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します。

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

追跡できるロックには次の 2 種類があります。

- 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を有効にすると、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。例えば

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

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
```

2.3 PostgreSQL Configuration for BDR

BDRnode に影響を与えるいくつかの PostgreSQL 構成パラメーターがあります。これらのパラメータは各ノードで異なるように設定できますが、一般的にはお勧めできません。

2.3.1 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 はロジカルデコーディングに依存するため、`logical` に設定する必要があります。
- `shared_preload_libraries`-これには `bdr` が含まれている必要がありますが、必要に応じて、前後のその他のエントリ、ただし「`pglogical`」は含まれる
- `track_commit_timestamp`-競合を解決するには、「`on`」に設定する必要があります競合する行のタイムスタンプを取得します。

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が、これはクラスターのサイズとスケールによって異なります。

- `logical_decoding_work_mem`-ロジカルデコーディング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させて保存されますローカルディスクに一時的に。デフォルトは 64MB ですが、もっと高く設定することもできます。
- `max_worker_processes`- BDR はレプリケーションにバックグラウンドワーカーを使用しますメンテナンスタスクが必要なため、十分なワーカーロットが必要です。正しく動作します。ワーカーの正しい最小数の式は次のとおりです。PostgreSQL インスタンスごとに 1 つ+そのインスタンスのデータベースごとに 1 つ+ 4 BDR 対応データベース BDR ノードの同等なごとに 1 つ+それぞれに 1 つデータベースごとに、BDR グループの同等なごとにライターが有効になりノード。ノードの実行中に、追加のワーカープロセスが一時的に必要な場合があります BDR グループから削除されました。ノードすべての同等なごとに 2 つ必要です。
- `max_replication_slots`-`max_wal_senders` と同じ。
- `wal_sender_timeout` および `wal_receiver_timeout`-方法の決定ノードはすぐにその CAMO パートナーを切断されたと見なします 再接続; [CAMO Failure Scenarios](#) を参照 詳細。

N 個の同等な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/ウォールセnderが必要であることに注意してください。同期中、BDR は一時的に `anotherN-1` スロット/`walsender` を使用するため、この時々のピーク需要に対応するために上記のパラメーターを十分に高く設定するよう注意してください。

並列適用をオンにすると、スロットの数を上記の式 *ライターから N スロットに増やすニーズがあります。これは、`max_replication_slots` がレプリケーション起点の最大数も設定し、パラレルアプライの機能の一部がライターごとに余分なオリジンを使用するためです。

WAL デコーダーが有効な場合、WAL デコーダープロセスでは、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になります。

単一のデータベースを使用する 4 ノード BDR グループの一般に安全な推奨値は、`max_replication_slots` および `max_worker_processes` を 50 および `max_wal_senders` に少なくとも 10 に設定することです。

また、これらのパラメーターを変更するには、ローカルノード: `max_worker_processes`、`max_wal_senders`、`max_replication_slots` を再起動する必要があることに注意してください。

アプリケーションでは、これらのパラメーターを設定することもできます。詳細については、[耐久性とパフォーマンスオプション] の章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 `physical replication` と同様の方法で。
- `synchronous_standby_names`- 上記と同じ

2.3.2 EDB PG 拡張および EDB PG BDR の詳細設定

次の Postgres 設定は、最大 1 回のコミット (CAMO) で考慮する必要があります。これは、PG 拡張との BDR の組み合わせでのみ使用可能な機能です。これらの一部は PG Extended でのみ利用可能です。他のコミュニティバージョンには既に存在しますが、CAMO と組み合わせで BDR にのみ関連するようになります。

- `synchronous_replication_availability`- オプションで `async` にできますノードの継続を許可することにより可用性を高め、CAMO パートナーが切断された後にコミットします。デフォルトでは `wait` の値、ノードは無期限に待機し、CAMO パートナーが再接続して送信した後にのみコミットする 確認。
- `snapshot_timestamp`- の使用をオンにします `timestamp-based snapshots` を使用して、使用するタイムスタンプを設定します。

2.3.3 BDR 固有の設定

また、設定可能な BDR 固有の構成設定もあります。特に明記しない限り、値は任意のユーザがいつでも設定できます。

競合処理

- `bdr.default_conflict_detection`- デフォルトの競合検出メソッドを設定します新しく作成されたテーブルの場合。と同じ値を受け入れます `bdr.alter_table_conflict_detection()`

グローバルシーケンスパラメータ

- `bdr.default_sequence_kind`- デフォルトの `sequence kind` を設定します。

DDL 処理

- `bdr.default_replica_identity`-REPLICA IDENTITY のデフォルト値を設定します新しく作成されたテーブル。REPLICA IDENTITY は、どの情報が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ます。
 - 受け入れられる値は次のとおりです。
 - `-DEFAULT`-プライマリキーの列の古い値があれば、それを記録します（これはデフォルトの PostgreSQL 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、テーブル固有の構成または `bdr.default_replica_identity` を介して、テーブルのレプリカアイデンティティが `FULL` でない限り、PRIMARY KEY または UNIQUE 制約のないテーブルで `UPDATEs` および `DELETEs` を複製できません。
- `bdr.default_replica_identity` が `DEFAULT` であり、テーブルに `'UNIQUE'` 制約がある場合、`'REPLICA IDENTITY'` として自動的に選択されません。テーブルの作成時に、または上記の文書で説明されているように、後で明示的に設定ニーズがあります。
- テーブルのレプリカアイデンティティを `FULL` に設定すると、書き込まれる WAL のボリュームと、テーブルのワイヤ上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「オン」）。
 - このパラメータは、`bdr_superuser` ロールまたはスーパーユーザロールによってのみ設定できます。
 - `bdr.ddl_replication = off` を使用して DDL を実行するか、BDR 管理関数を呼び出すと、管理者が介入できるまでレプリケーションが停止する状況が発生する可能性があります。詳細については、[the DDL replication chapter](#) を参照してください。
 - `bdr.ddl_replication` が `off` に設定されると、LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定により、キャプチャされた DDL コマンドまたは BDR レプリケーション機能のレプリケーションがスキップされたときに、`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「オン」）。このパラメータは、スーパーユーザのみが設定可能です。この設定 `bdr.ddl_replication` も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、保証のノード間でロールを同期させると、管理者が介入するまで複製 DDL がレプリケーションを中断する可能性があります。
 - 詳細については、[Role manipulation statements in the DDL replication chapter](#) を参照してください。
- `bdr.ddl_locking`-DDL のグローバルロックのオペレーションモードを構成します。
 - このパラメータは、`bdr_superuser` ロールまたはスーパーユーザロールによってのみ設定できます。
 - 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-オン-すべての DDL 操作にグローバルロックを使用する DDL 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 操作にのみグローバルロックを使用する
- `bdr.ddl_replication` が `off` に設定されると、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たときに、**WARNING** メッセージが書き込まれます。一部のステートメントでは、2 つの **WARNING** が発生するのが正常です。1 つは DML ロックをスキップし、もう 1 つは DDL ロックをスキップします。
- `bdr.truncate_locking`-デフォルトでは `False`、この構成オプションは `TRUNCATE` コマンドのロック動作。(真の場合) `TRUNCATE` するかどうかを決定します `bdr.ddl_locking` 設定に従います。

グローバルロック

- `bdr.ddl_locking`-上記で説明。
- `bdr.global_lock_max_locks`-可能なグローバルロックの最大数ノードで保持されます (デフォルトは 1000)。Postgres サーバーのスタートにのみ設定できます。
- `bdr.global_lock_timeout`-待機の最大許容期間を設定しますグローバルロックの場合 (デフォルトは 10 分)。値がゼロの場合、このタイムアウトは無効になります。
- `bdr.global_lock_statement_timeout`-の最大許容期間を設定しますグローバルロックを保持するステートメント (デフォルトは 60 分)。値がゼロの場合、このタイムアウトは無効になります。
- `bdr.global_lock_idle_timeout`-最大許容持続時間を設定しますグローバルロックを保持しているトランザクションのアイドル時間 (デフォルトは 10 分)。値がゼロの場合、このタイムアウトは無効になります。

ノード管理

- `bdr.replay_progress_frequency`-レプリケーション位置を送信する間隔残りのクラスターへの情報 (デフォルトは 1 分)。
- `bdr.standby_slot_names`-これらのスロットが受信と確認を要求するレプリケーションは他のものよりも先に変更されます。これは主に次の場合に便利ですフェイルオーバーのための物理的スタンバイの使用、またはサブスクライブ専用ノードの使用時。

汎用レプリケーション

- `bdr.writers_per_subscription`-あたりのデフォルトのライター数サブスクリプション (BDR では、これも変更できますグループの場合は `bdr.alter_node_group_config`)。
- `bdr.max_writers_per_subscription`-ライターの最大数サブスクリプションごと (上記の設定の上限を設定)。
- `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「オン」)。

- これをオフに `make` と、トランザクション全体がローカルのみになります。つまり、トランザクションは BDR およびその他すべての論理デコードのダウンストリームターゲットによるロジカルデコーディングロジカルデコーディングから可視なくなります。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は転送されません。
- このパラメータは、`bdr_superuser` ロールまたはスーパーユーザロールによってのみ設定できます。
- このパラメータは、`bdr.permit_unsafe_commands = on` を除き、`SET LOCAL` コマンドを使用して現在のトランザクション内でのみ設定できます。

!!! Note * トランザクションレプリケーションが無効になっている場合でも、WAL は生成されますが、これらの変更はオリジンでフィルタリングされます。

!!! Warning * `bdr.xact_replication` をオフにすると、ノード間でデータの不整合が発生します。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には単一ノードの変更が必要なレプリケーション状況でのみ使用してください。ご自身のリスクで使用してください。

- `bdr.permit_unsafe_commands`-コマンドの安全性チェックを無効にするオプション

- 一般的な使用には安全でないと見なされます。
- `bdr_superuser` または PostgreSQL スーパーユーザが必要です。

!!! Warning * 通常安全と見なされないコマンドは、一貫性のない結果を生成するか、レプリケーションを完全に中断する可能性があります。ご自身のリスクで使用してください。

- `bdr.batch_inserts`-内の 1 つのテーブルへの連続挿入数単一のトランザクションは、そのテーブルの挿入のバッチ処理をオンにします。

- このオプションを使用すると、挿入のセットではなく、ラージのデータを内部的に `COPY` としてレプリケーションできます。また、ノード結合中の初期データのコピー方法も示します。

- `bdr.maximum_clock_skew`

- これにより、`bdr.maximum_clock_skew_action` をトリガーする前に、着信トランザクションコミットタイムスタンプとサブスクライバーの現在の時刻との最大の差と見なされる値が指定されます。
- これにより、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の時刻と比較して将来にあるかどうかを確認されます。そして、その差が `bdr.maximum_clock_skew` よりも大きい場合、`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 です。つまり、クロックスキューを無視します（チェックはオフになります）。すべてのサーバーのクロックが同期されているときに 0 を設定することは有効です。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に実行するアクションを指定します。
- このオプションには 2 つの可能な値があります。
- `-WARN`-この事実に関するワーニングを記録します。サーバーログのフラッシュバックを防ぐため、警告は最大で毎分 1 回（デフォルト）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` を引いたものよりも古くなるまで待機します。

bdr.standby_slot_names

このオプションは通常、フェイルオーバー構成で使用され、この BDR ノードのフェイルオーバー候補のストリーミング物理的レプリカが、サブスクライバーに可視になる前に保証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イルオーバーでコミットが消滅しないことが保証されます。

名前がカンマ区切りの `b_tran_1` リストに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 `walsenderon` によって特別に処理されます。

BDR のロジカルレプリケーション `walsender` は、ノードが他の BDR 複製クライアントにそれらの変更を送信する前に、すべてのローカル変更が送信され、`bdr.standby_slot_names` のレプリケーションスロットにフラッシュされることを保証します。事実上、名前付きスロットのリストと他のすべてのレプリケーションクライアントとの間に同期レプリケーションバリアを提供します。

レプリケーションスロットは `bdr.standby_slot_names` にリストされます。論理スロットと物理的スロットの両方が機能しますが、通常は物理的スロットに使用されます。

このセーフガードがないと、サブスクライバーがコミットを受信し、フェイルオーバー候補がまだ受信していないためにフェイルオーバー時にプロバイダーから消失するという 2 つの異常が発生する可能性があります。

- 1 人以上のサブスクライバの場合、サブスクライバは変更を適用したが、新しいプロバイダーは、受信した変更と競合する新しいトランザクションを実行場合があります。プロバイダーに関する限り、それは決して起こりませんでした。

および/または

- 2 人以上のサブスクライバの場合、フェイルオーバーの時点で、すべてのサブスクライバが適用されているわけではありませんサブスクライバーは現在、一貫性のない、和解できない状態になっていますコミットを受信しなかったサブスクライバーには、コミットを取得する方法がないためです。今。

`bdr.standby_slot_names` を設定すると、プロバイダーのフェールオーバー候補のレプリカが維持されない場合、(デザインにより) サブスクライバーがプロバイダーに遅れることになります。したがって、監視は不可欠です。

`bdr.standby_slot_names` が役立つ別のユースケースは、サブスクライバーのみを使用する保証です。これにより、サブスクライバーのみのノードが他の BDR ノードのいずれよりも先に移動しないようにします。

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、レシーバがライタプロセスにデータを送信するために使用する共有メモリキューのサイズを指定するために使用されます。ライタプロセスが停止した場合や進行が遅い場合、キューがいっぱいになり、レシーバプロセスも停止する可能性があります。したがって、このキューに十分な共有メモリを提供することが重要です。デフォルトは 1MB で、最大許容サイズは 1GB です。任意のストレージサイズ指定子を使用して GUC を設定できますが、デフォルトはキロバイトです。

bdr.writer_output_queue_size

このオプションは、ライタプロセスからデータを受信するためにレシーバが使用する共有メモリキューのサイズを指定するために使用されます。ライタがラージ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 32kB、最大許容サイズは 1MB です。任意のストレージサイズ指定子を使用して GUC を設定できますが、デフォルトは kB です。

bdr.min_worker_backoff_delay

指定されたワーカーが `everybdr.min_worker_backoff_delay` ミリ秒よりも頻繁に再起動されるのを防ぐことにより、BDR バックグラウンドワーカーの起動を制限します。時間単位のサフィックスがサポートされています。

!!! Note * 現在、この設定はレシーバ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障害時に再接続を試みる速度に影響します。

デフォルトは 1 秒です。遅延は起動から起動までに適用される時間制限であるため、ワーカーの 1s のデフォルト値は、1 秒あたり最大 1 回の（再）起動に制限されます。

バックオフ遅延設定が変更され、PostgreSQL 構成が再ロードされると、現在のすべてのバックオフ待機がリセットされます。さらに、`bdr.worker_task_reset_backoff_all()` ファンクションが提供され、管理者はすべてのバックオフ間隔をすぐにエキスパイアにすることができます。

共有メモリ内の追跡テーブルは、各タイプのワーカーの最終起動時間を記憶するために維持されます。この追跡テーブルは永続的ではありません。これは、クリーンでないバックエンド終了後のクラッシュリカバリ中のソフト再起動など、PostgreSQL の再起動によってクリアされます。

ビュー `bdr.worker_tasks <monitoring#bdr.worker_tasks>`__` を使用してこの状態を検査し、管理者が現在有効なバックオフレート制限を確認できるようにします。

レート制限の目的で、労働者は「タスク」によって分類されます。このキーは、ワーカーロール、データベース oid、サブスクリプション ID、サブスクリプションライタ ID、拡張ライブラリ名前とファンクション名前、拡張機能が提供するワーカー名前、同期ライターのリモートリレーション 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、マネージャワーク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 Data Types の出力フォーマットを設定します。デフォルトの出力（この設定が結果の場合）は、現在の基本 CRDT タイプの値（例、`crdt_pncounter` の `bigint`）。`on` に設定すると、戻り値は CRDT 値。例、マルチプルのノードからの状態を含めることができます。

最大準備トランザクション

- `max_prepared_transactions`-対処するために十分に高く設定する必要があります同時準備トランザクションの最大数明示的な 2 フェーズコミット、CAMO または Eager によるクラスタートランザクション。制限を超えると、ノードがローカルの二相コミットまたは CAMO トランザクションクラスター上の熱心なトランザクション。Postgres サーバーのスタートにのみ設定できます。（EDB Postgres Extended）

熱心な複製

- `bdr.commit_scope`-コミットスコープを `global` に設定すると有効になります（デフォルト）。
- `bdr.global_commit_timeout`-グローバルの両方のステージのタイムアウト二相コミット（デフォルトは 60 秒）および CAMO で保護されたトランザクション CAMO を待機する時間の制限として、コミットフェーズでパートナー。

!!! Note * これは、EDB Postgres Extended でのみ利用可能です。

最大 1 回コミットする

- `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。CAMO はトランザクションごとにオンに切り替えることができますこれを `remote_write`、`remote_commit_async` に設定する、または `remote_commit_flush`。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。 `false` または `0` も CAMO を無効にします。
- `bdr.standby_dsn`-接続のマニュアルオーバーライドを許可します以来変更された場合に CAMO パートナーに到達するための文字列（DSN）ローカルノードのクラッシュ。通常は設定しないでください。Postgres サーバーのスタートにのみ設定できます。
- `bdr.camo_local_mode_delay`-で適用されるコミット遅延通常発生するオーバーヘッドをエミュレート CAMO のローカルモード CAMO パートナーはトランザクションを確認する必要があります。デフォルトは 5 ミリ秒です。0 に設定すると、この機能が無効になります。
- `bdr.camo_enable_client_warnings`-アクティビティが CAMO プロパティができないデータベースで実行保証されています。これはデフォルトで有効になっています。十分な情報に基づいたユーザーが選択できますこれを無効にして、ログに記録される警告の量を減らします。

!!! Note * これは、EDB Postgres Extended でのみ利用可能です。

トランザクションストリーミング

- `bdr.default_streaming_mode`-トランザクションストリーミングの制御に使用加入者ノード。許可される値は、`off`、`writer`、`file`、`auto` です。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要求しませんトランザクションストリーミング。他の許容値のいずれかに設定されている場合、サブスクライバーがトランザクションストリーミングを要求し、パブリッシャーが提供しますサポートしている場合はこれ（PostgreSQL を使用した BDR 4.0 からサポートを利用できます）14、EDB Postgres Extended 13 および EPAS 14）およびグループレベルで構成されます。のために詳細については、[Transaction Streaming](#) を参照してください。

タイムスタンプベースのスナップショット

- `bdr.timestamp_snapshot_keep`-の有効なスナップショットを保持する期間タイムスタンプベースの `snapshot` の使用（デフォルト 0、過去のスナップショットを保持しないことを意味します）。上記の `snapshot_timestamp` も参照してください。（EDB Postgres Extended）

監視とログ

- `bdr.debug_level`- BDR が書き込みに使用するログレベルを定義しますデバッグメッセージ。デフォルト値は `debug2` です。見たいなら詳細な BDR デバッグ出力、`bdr.debug_level = 'log'` を設定します。
- `bdr.trace_level`-上記と同様に、これはログレベルを定義します BDR トレースメッセージに使用します。のすべてのノードでトレースを有効にする BDR クラスターは、EDB サポートが問題を診断するのにヘルプます。Postgres サーバーのスタートにのみ設定できます。

!!! Warning * `bdr.debug_level` または `bdr.trace_level` を `>= log_min_messages` の値に設定すると、非常にラージのログ出力が生成される可能性があるため、ディスクスペースの枯渇を防ぐためにログフィルタリング、アーカイブ、ローテーションの計画が整っていない限り、稼動環境で長期間有効にしないでください。

- `bdr.track_subscription_apply`-の適用統計を追跡各サブスクリプション。
- `bdr.track_relation_apply`-それぞれの適用統計を追跡 リレーションの
- `bdr.track_apply_lock_timing`-追跡時の追跡ロックタイミングリレーションの統計。

内部

- `bdr.raft_keep_min_entries`-保持するエントリの最小数ログ圧縮を行うときにログをラフティン
グする（デフォルトは 100）。値 0 は無効になりますログ圧縮。 **警告：ログの圧縮が無効になってい
る場合、ログは永遠にサイズが大きくなります。** Postgres サーバーのスタートにのみ設定できます。
- `bdr.raft_response_timeout`-ネットワーク障害をアカウントして、実装されたラフトコンセンサ
スプロトコルは、一定の時間。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`-ステートマシンを移動するには転送すると、Raft は内部ログ
にエントリを追加します。通常時オペレーション、追加には数ミリ秒しかかかりません。これはその
追加アクションの期間の上限しきい値。INFO メッセージが記録されます。これは、実際の問題。こ
のパラメータのデフォルト値は 3000 ミリ秒です。
- `bdr.raft_log_min_message_duration`-コンセンサスリクエストをログするタイミング。ログコン
センサスリクエストの往復時間を測定し、時間がこのパラメータを超える場合の INFO メッセージ。
デフォルト値このパラメータの最大値は 5000 ミリ秒です。
- `bdr.raft_group_max_connections`-接続の最大数 Postgres サーバーのすべての BDR グループ全体。
これらの接続は運ぶグループのノード間の bdr コンセンサスリクエスト。これのデフォルト値パラ
メータは 100 接続です。Postgres サーバーのスタートにのみ設定できます。
- `bdr.backwards_compatibility`-バージョンを指定します下位互換性があり、同じ数値フォーマッ
トで `bdr.bdr_version_num`、たとえば 30618。の正確な動作を可能にします以前の BDR バージョ
ン、これが一般に望ましくない効果をもたらす場合でも。デフォルトは現在の BDR バージョンです。
これはリリースから変更されるためリリースするには、構成内で明示的に使用しないことをお勧めし
ます値が現在のバージョンと異なる場合を除き、ファイル。
- `bdr.track_replication_estimates`-レプリケーションの見積もりを用語で追跡する同等な適用
率とキャッチアップ間隔。この情報は CAMO などのプロトコルで使用して、同等なノード。このパ
ラメータはデフォルトで有効になっています。（EDB Postgres Extended）
- `bdr.lag_tracker_apply_rate_weight`-同等なからの遅れをモニタしますローカルノードから WAL
を適用するという点で、移動を計算します遅延追跡の適用率の平均。このパラメータは、この移動平
均における新しい計算値の寄与度計算。デフォルト値は 0.1 です。（EDB Postgres Extended）

2.4 Node Management

BDR グループのメンバである各データベースは、独自のノードで表される必要があります。ノードは、BDR グループ内のそのようなデータベースの一意の識別子です。

現在、各ノードは 1 つのノードグループのメンバにしかありません。これは今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を含めることもでき、さまざまな異なるアーキテクチャを作
成できます。

2.4.1 BDR グループの作成と参加

BDR の場合、すべてのノードに他のすべてのノードへの接続が必要です。構成を簡単にするために、新しいノードが参加すると、既存のすべてのノードが自動的に接続するように構成されます。このため、最初に作成された 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する（「データソース名前」の DSN と呼ばれることもある）ことを知っている必要があります。接続文字列の両方の形式がサポートされています。そのため、`host=myhost port=5432 dbname=mydb` などの key-value フォーマット、または URI フォーマット：`postgres://myhost:5432/mydb` のいずれかを使用できます。

SQL ファンクション `bdr.create_node_group()` を使用して、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 BDR グループ（その時点で 1 つのノードのみで構成される）に結合できるようになります。作成時には、他のノードがこのノードへの接続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、それ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 BDRgroup に結合できます。

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`（または物理的スタンバイ）を使用して新しいノードを作成します。`pg_basebackup` を使用する場合、`bdr_init_physical` ユーティリティは、データベースクラスタ全体のバックアップの以前の動作とは対照的に、オプションでターゲットデータベースのベースバックアップのみを指定できます。これ `make`、このアクティビティがより速く完了し、不要なデータベースが除外されるため、使用するスペースが少なくなります。ターゲットデータベースのみが指定されている場合、`excludeddatabases` はクリーンアップされ、新しいノードで削除されます。

!!! Warning * 一度に 1 つのノードのみが BDR ノードグループに結合か、BDR ノードグループから分離されます。進行中の別の結合またはパートオペレーションがある間に新しいノードが結合されている場合、結合が完了した後、新しいノードに一貫性のあるデータがない場合があります。

新しい BDR ノードが既存の BDR グループに参加するか、ノードがアップストリーム同等なサブスクライブされると、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同等なからローカルノードにコピーする必要があります。このコピーは、ローカルデータとリモートデータが同一で始まるように慎重に調整する必要があります。`pg_dump` を自分で使用するだけでは不十分です。BDRextension は、この初期コピーを作成するためのビルトインファシリティを提供します。

結合プロセス中に、BDR 拡張機能は、提供されたソースノードをベースとして使用して既存のデータを同期し、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、結合プロセスを最初から再開する必要があります。

クラスターに参加しているノードには、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ることはできません。BDR 拡張機能を除き、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ることが重要です。

スキーマの同期は、オプションで `bdr.join_node_group()` ファンクションの `synchronize_structureparameter` を使用してスキップされたできます。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。

最適な接続（つまり、最も近い）を持つソースノードを選択することをお勧めしノード。これにより、結合の完了に必要な時間が短縮されます。

結合プロシーダは、Raft コンセンサスアルゴリズムを使用して調整されます。これは、ほとんどの既存ノードがオンラインで到達可能である必要があります。

ロジカル結合プロシーダ (`bdr.join_node_group()` ファンクションを使用) は、COPY 操作を行うデータ同期を実行し、それらが有効な場合はマルチプルのライターを使用します (並列適用)。

ノード結合は、結合にかかる時間の大半で、他のノード結合と同時に実行できます。通常はかなり短い状態である **PROMOTE** または **PROMOTING** のいずれかの状態にできるのは、一度に 1 つの通常のノードのみです。サブスクライバーのみのノードは、このルールの例外であり、**PROMOTE** および **PROMOTING** 状態にも同時に存在できます。

結合プロセスはソースとして 1 つのノードのみを使用するため、ノードの大部分が使用可能な場合、ノードが停止しているときに実行できることに注意してください。これにより、ロジカル結合のコミット時にタイムスタンプが生じる可能性があります。ソースノードは、ソースノードの最新のコミットタイムスタンプに設定されます。これより前のコミットタイムスタンプを持つノードでコミットされた変更は (ノードがダウンしているか、大幅な遅れがあるため) 他のノードからの変更と競合する可能性があります。この場合、新しく結合されたノードは他のノードとは異なる方法で解決され、分岐が発生する可能性があります。そのため、ノード間に大きなレプリケーションラグが存在する場合はノード結合を実行しないことをお勧めしますが、これが必要な場合は、新しく結合したノードで **LiveCompare** を実行して、すべてのノードが利用可能になり追いついたらデータの相違を修正します。

キャッシュ検索の失敗により、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 **pg_dump** を内部的に使用するため、ソースノードで同時 DDL アクティビティがある場合、失敗する可能性があります。このような場合、結合の再試行が機能するはずですが。

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) で 3.7.x を実行している BDR クラスター、または 3.7 ノードと 4.0 ノードの混合に結合できます。このプロシーダは、ユーザが BDRmajor バージョンだけでなく基礎となるバージョンもアップグレードする場合に役立ちます。PostgreSQL のメジャーバージョン。これは、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 BDR クラスターに参加させることで実現できます。もちろん、新しいノードは、既存のクラスター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。

一部のノードが PostgreSQL メジャーリリースで 3.6 を実行し、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、BDR はレプリケーションがすべての方向で正しく機能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、ユーザは古いノードを切り離すことで、クラスターを迅速に不均質な状態にすることをお勧めします。古いバージョンでは使用できない DDL を実行しないように注意する必要があります。逆の場合も同様です。

異なるメジャー PostgreSQL リリースに参加するノードは、`bdr_init_physical` を介して取得された物理バックアップを使用できず、ノードはロジカル結合メソッドを使用して参加する必要があります。PostgreSQL のメジャーリリースは相互にディスク上で互換性がないため、これが重要です。

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場合、競合解決構成などの特定の構成はソースノードからコピーされないことに注意してください。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 は単純に `libpq` 接続文字列です。そのため、許可された `libpq` 接続パラメーター（SSL の接続パラメーターを含む）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ることに注意してください。クライアント証明書を使用したこのような一連のパラメーターの例を次に示します。

```
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`、`root.crt` が各ノードのデータディレクトリに適切な権限で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` Certificate Authority で直接または間接的に署名されます。接続で使用されるホスト名前またはアドレスが証明書の内容と一致すること。名前の場合、これは Subject Alternative Name にマッチするか、証明書にそのような名前がない場合、Subject の Common Name (CN) フィールドに一致します。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、接続が名前ではなくアドレスによって行われる場合、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 接続を行うユーザの名前である必要があります。これは通常、ユーザ `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ラインが必要です。例：

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

別の設定としては、`clientcertificates` 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の身元を確認する必要はありません。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` 行は次のようになります。

```
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` ユーザは正しいパスワードを含む `pgpass` ファイルを必要とします。

2.4.2 監視ノード

クラスターのノード数が偶数のイベント、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）が発生した場合に追加のノードを作成して結び付けを解除すると便利な場合がありますヘルプ。

追加のフルサイズノードを作成する代わりに、ウィットネスノードとも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。

2.4.3 ロジカルスタンバイノード

BDR では、「offloadnode」、「read-only ノード」、「receive-only ノード」または「ロジカル read レプリカ」とも呼ばれる「ロジカルスタンバイノード」を作成できます。マスタノードには、0、1、またはより多くのロジカルスタンバイノード。

物理的スタンバイノードでは、ノードが完全に起動することではなく、継続的なリカバリモードのままになります。BDR 同様のことが可能です。bdr.join_node_group には make オプションがあり、ノードをロジカルスタンバイノードとして中途半端にとどめます。論理スタンバイノードは変更を受信しますが、ローカルで行われた変更を他のノードに送信しません。

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、DSN inbdr.join_node_group によって定義された 1 つのソースノードによってデータを送信されます。他のすべてのノードからの変更は、この onsource ノードから受信され、マルチプルのサイト間の帯域幅を最小化します。

高可用性にはマルチプル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 つの物理的スタンバイをマスタに昇格できます。この場合、新しいマスタは、すべてまたはすべてのロジカルスタンバイノードにフィードを継続できます。
- ソースノードの場合死に、1 つのロジカルスタンバイをノードに昇格させ、ソースを置き換えることができますシングルマスタオペレーションと同様のフェイルオーバーオペレーションで。あることに注意してくださいマルチプルのロジカルスタンバイノードである場合、他のノードは新しいマスタをフォローできません。そのため、この手法の有効性は事実上 1 つのロジカルスタンバイ。

既存の BDR ノードの新しいスタンバイが作成された場合、グループスロットが最後に進められてから少なくとも 16 メガバイトの LSN が経過するまで、オペレーションに必要なレプリケーションスロットが新しいスタンバイに同期されないことに注意してください。極端な場合、これにはスロットがストリーミングレプリカで同期/作成される前にフル 16 メガバイトが必要になる場合があります。このインターバル中にフェイルオーバーまたはスイッチオーバーが発生した場合、グループスロットおよび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昇格させて BDR ノードを置き換えることはできません。

EDB Postgres Extended および EDB Postgres Advanced では、これは自動的に解決されます。スタンバイのスロット同期プロセスは、アップストリームでファンクションを呼び出すことでこれを解決します。このファンクションは、WAL 切り替えを実行し、すべての BDRpeer ノードに進行状況の更新をリプレイするように要求することにより、BDR クラスタ全体のグループスロットを移動します。上記により、グループスロットは短期間で先に進みます。これにより、スタンバイが初期スロットの同期アップに必要な時間を短縮し、必要に応じてより高速なフェイルオーバーを可能にします。

PostgreSQL では、スロットを昇格する前にスタンバイでスロットの同期が完了していることを保証することが重要です。次のクエリをターゲットデータベースのスタンバイで実行して、スロットがアップストリームと同期していることをモニタおよび保証できます。このクエリが true を返すときに、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;
```

また、WAL 切り替えを手動で実行し、すべての BDRpeer ノードに進行状況の更新をリプレイするように要求することにより、BDRcluster 全体でスロット同期プロセスをナッジすることもできます。このアクティビティにより、グループスロットが短いタイムスパンで先に移動し、スタンバイのスロット同期アップアクティビティも早まります。このために、ターゲットデータベースの BDR 同等なで次のクエリを実行できノード。

```
SELECT bdr.run_on_all_nodes('SELECT pg_catalog.pg_switch_wal()');
SELECT bdr.run_on_all_nodes('SELECT bdr.request_replay_progress_update()');
```

スタンバイで上記のモニタリングクエリーを使用して、これらのクエリが実際にそのスタンバイでのより高速なスロット同期にヘルプを確認します。

必要に応じて、物理的スタンバイノードを使用して論理スタンバイノード自体を保護することができるため、Master-> LogicalStandby-> PhysicalStandby のようになります。LogicalStandby から LogicalStandby にカスケードできないことに注意してください。

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、物理的スタンバイの制限は適用されないことに注意してください。これは、ロジカルスタンバイに追加のインデックス、データの保持期間の延長、中間作業テーブル、LISTEN / NOTIFY、一時テーブル、マテリアライズドビュー、およびその他の違いを持たせることができるため、大きなメリットがあります。

プロモーションの前にコミットするロジカルスタンバイに対してローカルに行われた変更は、他のノードに送信されません。プロモーション後にコミットすべてのトランザクションは、先に送信されます。ロジカルスタンバイへの書き込みを実行する場合、昇格する前にデータベースを静止するように注意するようにアドバイスしました。

ロジカルスタンバイノードに DDL の変更を make ことはできますが、それらは複製されず、グローバル DDL ロックの取得も試みません。DDL と同様に機能する BDR 関数も複製されません。[DDL Replication] を参照してください。ロジカルスタンバイに対して互換性のない DDL 変更を行った場合、データベースは分岐ノードと呼ばれます。現在、ダイバージェントノードの昇格はレプリケーションの失敗につながります。そのため、スタンバイとして使用する保証は、ロジカルスタンバイノードにダイバージェントな変更がフリーようにするか、ダイバージェントノードが昇格しないようにする必要があります。

2.4.4 フィジカルスタンバイノード

BDR は、従来の物理的スタンバイフェイルオーバーノードの作成も可能にします。これらは通常、短い昇格プロシージャの後にクラスタ内の BDR ノードを直接置き換えることを意図しています。標準の PostgreSQL クラスタと同様に、ノードにはこれらの物理レプリカがいくつあってもかまいません。

ただし、レプリケーションスロットの使用および BDR の他の機能要件により、これが適切に機能するための最小限の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ストリーミングを使用します物理的レプリケーションレプリケーションスロットを介した複製。 - スタンバイには次のものがあります。 - `recovery.conf` (PostgreSQL < 12 の場合、PostgreSQL 12+ の場合、これらの設定は `postgres.conf` にある必要があります)： - プライマリを指す `primary_conninfo - primary_slot_name` は、このスタンバイでのみ使用されるプライマリの物理的レプリケーションスロットの名前を指定します- `postgresql.conf`： - `shared_preload_libraries = 'bdr'`、リストには他のプラグインも存在できますが、`pglogical` を含めることはできません- `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 は物理的を指定する必要がありますスタンバイの primary_slot_name に使用されるレプリケーションスロット。

これは、BDR ノードの動作する物理的スタンバイを生成するには十分ですが、対処すべき追加の懸念事項がいくつかあります。

スタンバイが確立されると、BDR グループスロットを含むプライマリの他の BDR 関連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になります。少なくとも、スタンバイのスロットは「ライブ」のみであり、pg_replication_slots によって報告されるゼロ以外の confirmed_flush_lsn をレポートする場合、フェイルオーバー後も存続します。

その結果、フェイルオーバーが正常に機能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 クラスターの物理的スタンバイノードをチェックする必要があります。この予防措置を怠ると、BDR ノードとしてファンクション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がスタンバイに存在する可能性があり、フェイルオーバーが中止されます。

物理的スタンバイノードが最新であり、(bdr.standby_slot_names の構成に従って) 昇格できることを保証する保護メカニズムは、物理的スタンバイノードが最新の状態になった後にのみグループレプリケーションが発生するため、BDR グループの全体的なレプリケーションレイテンシに影響します。

これらの理由により、通常、物理的スタンバイノードの代わりにロジカルスタンバイノードまたはサブスクライプ専用グループのいずれかを使用することをお勧めします。両方とも比較してより優れた運用特性を備えているからです。

次の BDR スタンバイを使用して、グループスロットがすべてのノードで（可能物理的限り）進んで構文ように手動で保証でき SQL。

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

フェイルオーバー時に、スタンバイは 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。

1. Primary.2 と同じ IP アドレスまたはホスト名の制御を想定します。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 BDR 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 ノードが使用する最後の場所に早送りします。

bdr.standby_slot_names パラメータにもノートしてください。Tt は、プライマリ->フィジカルスタンバイのリレーション BDR クラスター、またはサブスクライバーのみのグループを使用する場合に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の最大のラグを示すクラスターノードの状態を常に反映するメンバスロットを維持しノード。グループスロットの状態。

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 パラメータは BDR に名前付けスロットをグループスロットの必要なコンストレインとして考慮するよう通知します。設定すると、ス

スタンバイが遅れを示す場合、通常はグループスロットが進んでいたとしても、グループスロットが保持されます。

物理的レプリカと同様に、このタイプのスタンバイも同期的レプリカとして構成できます。念のため、これには以下が必要です。

- スタンバイ時：
 - `-primary_conninfo` で一意の `application_name` を指定する
- プライマリで：
 - `-synchronous_commit` の有効化
 - `-synchronous_standby_names` にスタンバイ `application_name` を含める

`insynchronous_standby_names` では、物理的スタンバイと他の BDR ノードを混在させることができます。CAMO および 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 が使用されている場合）または BDR ノードがまったく含まれていない（BDR All Node Replication が使用されている）make を確認してください。

2.4.5 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、最上位の親グループ内の特定の目的に割り当てることができます。`node_group_type` は、サブグループが作成されときのタイプを指定します。

サブスライバー専用グループ

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を受け取りません。これはロジカルスタンバイノードと多少似ていますが、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の使用不可または分離の影響を受けません。

subscriber-only ノードは完全に結合された BDR ノードであるため、複製されたすべての DDL を受信し、それらに作用します。また、Raft を使用して、クラスター内のすべてのノードにそのステータスを一貫してレポートします。**subscriber-only** ノードには Raft の投票権がないため、Raft リーダーになることもリーダー選挙に参加することもできません。また、複製された DDL を受け取りますが、DDL または DML ロックの取得には参加しません。つまり、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 Treetopology のビルディングブロックを形成します。このトポロジでは、少数の完全にアクティブなノードがあり、すべての方向に変更を複製し、変更を受信するだけでクラスター内の他のノードに変更を送信しない **subscriber-only** ノードが多数あります。このトポロジは、ラージのノードに起因する接続の爆発を防ぎますが、データを消費するために使用できる非常にラージの leaf ノードを提供します。

オーダーノードを使用するには、ユーザは最初に「サブスクライバーのみ」タイプの BDR グループを作成する必要があります。メンバノードがレプリケーションの変更を受信するグループのサブグループである必要があります。サブグループが作成されると、**subscriber-only** ノードになる予定のすべてのノードがサブグループに結合する必要があります。「サブ親専用」タイプの複数のサブグループを作成でき、異なるサブグループを持つことができます。

ノードが「サブスクライバのみ」のサブグループに正常に参加すると、**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スタートします。**subscriber-onlynode** 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

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ると、レプリケーションの変更を同期するためのソースとして機能できません。ただし、**subscriber-onlynode** が、クラスター内の他のノードが現在持っていない分割されたノードからレプリケーションの変更を既に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。

現時点では、**bdr.standby_slot_names** および **bdr.standby_slots_min_confirmed** を適切に設定して、**subscriber-onlynodes** の前に常に完全にアクティブな BDR ノードが存在するようにすることで、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。レプリケーションで先に **b_tran_2** ノードを許可してから、同期のためにマストレプリケーションソースを使用するか、別の完全に結合されたノードが分離されたときにクラスターから矛盾する **subscriber-only** ノードをオプションで削除する方法を提供できます。

2.4.6 デコードワーカー

BDR4 には、データが送信されているノードの数に関係なく、デコードを 1 回実行するデコードワーカープロセスを有効にするオプションがあります。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 Sender プロセスがまだ存在しますが、これらのプロセスはデータの送信と受信のタスクを実行するだけです。これらの変更を総合すると、より大きな BDR グループの CPU オーバーヘッドが削減され、WAL Sender プロセスが通信により多くの時間を費やすようになるため、レプリケーションのスループットも向上します。

enable_wal_decoder は各 BDR グループのオプションであり、現在デフォルトで無効になっています。**bdr.alter_node_group_config()** を使用して、BDR グループの WAL デコーダーを有効または無効にすることができます。

WAL デコーダーが有効な場合、BDR は Logical Change Request (LCR、略称) ファイルを保存して、デコードとサブスクライブするすべてのノードがデータを受信したときの変更のバッファリングを可能にします。LCR ファイルは、各ローカルノードのデータディレクトリ内の **pg_logical** ディレクトリの下に保存されます。LCR ファイルの数とサイズは、レプリケーションの遅延が増加するにつれて変化するため、モニタリングも必要になります。BDR ノードのいずれにも不要な LCR は定期的にクリーニングされます。

2つの連続したクリーンアップのインターバルは、`bdr.lcr_cleanup_interval` によって制御されます。デフォルトは3分です。`bdr.lcr_cleanup_interval` がゼロの場合、クリーンアップは無効になります。

無効にすると、各ノードにサブスクライブしている各ノードに対して、WAL Sender プロセスによってロジカルデコーディングが実行されます。この場合、LCR ファイルは書き込まれません。

WDR デコーダーはBDR グループに対して有効になっていますが、次の GUC はノードごとの LCR の生産と使用を制御します。デフォルトでは、これらは `false` です。LCR の生産と使用のために、BDR グループで WAL デコーダーを有効にし、これらの GUC を BDR グループの各ノードで `true` に設定する必要があります。

- `bdr.enable_wal_decoder=false` にすると、WAL デコーダーを停止します。LCR を使用している WAL センダは WAL を使用するために再起動されます。`true` の場合 BDR グループ構成とともに、WAL デコーダーが実行され、LCR と WAL が生成されます送信者は LCR を使用します。
- `bdr.receive_lcr`-サブスクライブしているノードの `true` が WAL を要求する場合利用可能な場合、LCR を使用するためにパブリッシャーノードで送信します。

メモ

現在、WAL デコーダーは、実行中のノードに対応する変更をデコードします。単一のソースを介して、BDR グループのすべてのノードから変更がロジカルスタンバイに送信されます。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。

購読者専用ノードはそれぞれのノードから直接変更を受信するため、購読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、LCR が必要でない場合は、LCR に対応する WAL を削除できる可能性があります。

リファレンスまでに、LCR ファイル名の最初の 24 文字は WAL ファイル名のものと似ています。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 文字と同様に、TimeLineId を表すと予想されます。名前の 16 文字の次のシーケンスは、WAL ストリームに対して LCRchanges を追跡するのに使用される WAL セグメント番号と同様です。ノート、ロジカル変更は、下位のトランザクションのコミットオーダーに従って並べ替えられます。したがって、LCR セグメントでの配置は、WAL セグメントでの対応する WAL の配置とマッチしません。最後の 16 文字のセットは、LCR セグメント内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サイズで変数サイズです。LCR ファイルの最大サイズは、`bdr.max_lcr_segment_file_size` で制御できます。デフォルトは 1GB です。

この機能を動作させるには、EDB Postgres Extended 13 以降が必要です。

2.4.7 ノードの再起動とノードの復旧

BDR は、ノードのリスタート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、各同等な再接続し、そのノードから不足しているデータを複製することにより、グループに自動的に再参加しノード。

ノードが起動すると、各接続は同等な `= catchup` の表示を開始し、欠落したデータの複製を開始しノード。サーバーのワークロード。

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、データの多いノードからのキャッチアップ期間は他のノードよりもかなり長くかか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す可能性があります。ユーザーは、次の問題を理解せずに拡張の停止を計画しないでください。

ノードノードはオフラインノードを保持するため（同等なごとに 1 つを使用）、後で一時的にストレージできないノードへの変更をリプレイスペースノード。で* WAL *）、データベースサーバー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

…またはその他のディスク外関連の症状をレポートします。

さらに、オフラインノードのスロットもカタログ xmin を抑制し、カタログテーブルのバキューム処理を防ぎます。

EDB Postgres Extended では、オフラインノードもデータの凍結を抑制して、競合解決データの損失を防ぎます（参照：）。

管理者はノードの停止をモニタし（参照：）、ノードに十分なフリーディスクスペースがある `make` 必要があります。ワークロードが予測可能な場合、時間の経過とともに使用されるスペースの量を計算できるため、重大な問題が発生する前にノードがダウンできる最大時間を予測できます。

BDR によって作成された複製スロットは、手動で削除しないでください。その場合は、クラスターが破損し、スロットを使用していたノードをクラスターから切り離す必要があります（以下を参照）。

ノードがオフラインになっている間、他のノードはまだオフラインノードから同じデータセットを受信していない可能性があるため、ノード間でわずかな相違として表示されることに注意してください。ノード間のこの不均衡は、分割プロセス中に自動的に修正されます。これより後のバージョンでは、これが以前に行われる可能性があります。

BDR によって作成された複製スロット

BDR マスタノードでは、次のレプリケーションスロットが自動的に作成されます。

- `bdr_<database name>_<group name>` 名前付けの 1 つのグループスロット。
- N-1 ノードスロット、名前付けは `bdr_ <データベース名前> _ <グループ名前> _ <ノード 名前>`、N はクラスター内の BDR ノードの総数、ダイレクトロジカルスタンバイ（ある場合）を含む。

ユーザはこれらのスロットを削除してはいけません。これらは BDR によって自動的に作成されており、BDR によって管理されます。

一方、Barmanor ロジカル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は、BDR に影響を与えることなく、ソフトウェアの適切なコマンドを使用して作成またはドロップできます。bdr_。

例、3つのノード alpha、beta、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 名前付けの 1つのグループスロット
- bdr_mydb_mygroup_beta 名前付けの 2つのノードスロットと bdr_mydb_mygroup_gamma
- ノード beta には 3つのスロットがあります。
- bdr_mydb_mygroup 名前付けの 1つのグループスロット
- bdr_mydb_mygroup_alpha 名前付けの 2つのノードスロットと bdr_mydb_mygroup_gamma
- ノード gamma には 3つのスロットがあります。
- bdr_mydb_mygroup 名前付けの 1つのグループスロット
- bdr_mydb_mygroup_alpha 名前付けの 2つのノードスロットと bdr_mydb_mygroup_beta

グループ複製スロット

グループスロットは、主に BDR グループ内のノード（すべての論理スタンバイを含む）がこのノードからのアウトバウンドレプリケーションに対してキャッチし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前は、bdr.local_group_slot_name() ファンクションによって指定されます。

グループスロットは次のことができます。

- 既存のノードをすべて持たずに、新しいノードを BDR グループに結合させる稼働中（ただし、ノードの大部分は稼働している必要があります）、なし結合中にダウンしていたノードが起動した場合にデータロスが発生する再び複製する
- パートのノードが一貫していない場合でも、クラスターからのノードを一貫して別れたノードに完全に追いついた
- 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抑えます（EDB Postgres Extended）
- タイムスタンプベースのスナップショットの履歴 snapshot を保持します（EDB Postgres Extended）

通常、グループスロットは非アクティブであり、他のノードからの Raft 進行メッセージに応じて定期的のみ早送りされます。

警告：グループスロットをドロップしないでください。通常はアクティブではありませんが、BDR クラスターの適切なオペレーションには依然として重要です。削除された場合、上記の機能の一部またはすべてが機能しなくなり、および/または誤った結果になる可能性があります。

ロング識別子のハッシュ

レプリケーションスロットの名前は、他の PostgreSQLIdentifier と同様に、63 バイトより長くすることはできません。BDR は、結果のスロット名前がその制限に対して長すぎる場合に備えて、データベース名前、BDR グループ名前、およびノードの名前を短縮することでこれを処理します。識別子の短縮は、文字列の最終セクションを文字列自分自身のハッシュに置き換えることによって実行されます。

この例として、group20xxxxxxxxxxxxxxx (20 バイト長) という名前の BDR グループを使用して、db20xxxxxxxxxxxxxxx (20 バイト長) 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。ノード a30xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx (30 バイト長) に関連付けられたロジカルレプリケーションスロットが呼び出されます。

```
bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxxxxx7f304a2
```

…3597186、be9cbd0、7f304a2 はそれぞれ db20xxxxxxxxxxxxxxx、group20xxxxxxxxxxxxxxx、a30xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx のハッシュであるため。

2.4.8 BDR グループからノードを削除する

BDR は拡張ノード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通知する必要があります。ノードを完全にシャットダウンし、他のノードに通知しないと、パフォーマンスが低下し、最終的にシステム全体が機能しなくなります。

- parting *とも呼ばれるノードの削除は、bdr.part_node() 関数を使用して行われます。ノードを削除するには、ノード名前（ノード作成時に渡される）を指定する必要があります。bdr.part_node() ファクションは、削除されるノードを含む、BDR グループ内のアクティブノードから呼び出すことができます。

結合プロシージャと同様に、分割は Raft コンセンサスを使用して行われ、多数のノードが動作するようにオンラインにする必要があります。

分離プロセスはすべてのノードに影響します。Raft リーダーは、ノード間の投票を管理して、別のノードからの最新のデータを持つノードを確認します。その後、残りのすべてのノードは、最新のノードに二次的な一時 make な接続を確立し、欠落データをキャッチします。

分離されたノードはまだ BDR に認識されていますが、リソースを消費しません。Anode は、parted ノードとまったく同じ名前で再追加できます。まれに、ファンクション bdr.drop_node() を使用して partednode のすべてのメタデータをクリアすることをお勧めします。

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。これは、次のコマンドで実行できます。

```
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、BDRextension は削除できません。例は次のとおりです。

- timeshard や galloc などの BDR 固有のシーケンスを使用するテーブル
- 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 Views

これらの依存関係は、BDR 拡張機能を削除する前に削除する必要がありインスタンス、依存オブジェクトを削除する、列タイプを非 BDR に変更する、シーケンスタイプを tolocal に戻すなどです。

BDR BDR メタデータ機能のドロップは、ノードがその BDR ノードグループから正常に分離された場合、またはグループ内の最後のノードである場合にのみ実行レプリケーション必要があります。

!!! Warning * ローカル BDR ノード、またはローカルデータベースの BDR 拡張機能をドロップすると、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可能性があります。この問題は、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で解決できます。

bdr.drop_node() ファンクションもありますが、これは緊急時にのみ使用され、別れに問題がある場合に使用します。

2.4.9 BDR トポロジーのリスト

BDR グループのリスト

次の（シンプル）クエリーは、現在のノードがメンバになっているすべての BDR ノードグループをリストします（現在は 1 行のみを結果ます）。

```
SELECT node_group_name
FROM bdr.local_node_summary;
```

各ノードグループの構成は、より複雑なクエリーを使用して表示できます。

```
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

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は、次の例に示すように、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
```

(次のページに続く)

```
, 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 クエリー列に示されているノードの読み取り専用状態は、STANDBY として示されていることに注意してください。

ノード状態のリスト

- **NONE**：ワーカーの起動時にノードの状態が設定されず、すぐに設定されることが期待されます現在の既知の状態に。
- **CREATED**：bdr.create_node() は実行されましたが、ノードは BDR クラスターのメンバ。
- **JOIN_START**：結合は、ローカルノードを既存の BDR クラスター。
- **JOINING**：ノードの結合が開始され、現在は初期同期フェーズにあります。ノードでスキーマとデータを作成します。
- **CATCHUP**：初期同期フェーズが完了しました。結合は最後のステップになりましたアップストリームで実行されたトランザクションの取得と適用の結合が開始されノードからの同等な。
- **STANDBY**：ノードの結合は終了しましたが、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、bdr.promote_node を呼び出してノードの状態を **ACTIVE** に移動します。これらの 2 つの **PROMOTE**states は一貫している必要があります事実、**STANDBY** よりも高い状態にできるノードは 1 つだけですが、**ACTIVE** よりも低い。
- **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。
- **PART_START**：ノードは **ACTIVE** または **STANDBY** で、bdr.part_node を呼び出しました BDR クラスターからノードを削除します。
- **PARTING**：ノードは他のノードから切断され、それ以上のパートを果たしませんコンセンサスまたはレプリケーション。
- **PART_CATCHUP**：分割されていないノードは、最近別れたノード。
- **PARTED**：ノード分割オペレーション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 **PROMOTE** または **PROMOTING** の状態になります。

2.4.10 ノード管理インターフェース

ノードは、SQL インターフェースを使用して動的に追加および削除できます。

bdr.create_node

このファンクションはノードを作成します。

あらすじ

```
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。1つのノードのみが許可されますデータベース有効なノード名は、小文字、数字、ハイフンとアンダースコア。
- **local_dsn**-ノードへの接続文字列

注

このファンクションは、関連付けられた **public** 接続文字列でローカルノードのレコードを作成するだけです。ローカルレコードは1つしか存在できないため、一度作成すると、ファンクションを再度実行するとエラーになります。

このファンクションはトランザクションファンクションです。ロールバックでき、それによって行われた変更は現在のトランザクションに可視されます。

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

bdr.drop_node

ノードをドロップします。このファンクションは、**通常の使用を意図したものではありません**。テクニカルサポートの指示の下でのみ実行してください。

このファンクションは、特定のノードのメタデータをローカルデータベースから削除します。ノードは次のいずれかです。

- **** ローカル**** ノード。この場合、すべてのノードメタデータが削除されます。リモートノードに関する情報を含む。
- **リモートノード**。この場合、その特定のメタデータのみノードが削除されます。

あらすじ

```
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、危険の可能性があるにもかかわらず、特定の BDR ノードのすべてのメタデータ矛盾を引き起こします。強制ノードドロップは、関連する緊急事態の場合のみのテクニカルサポート 別れ。

注

これを実行する前に、`bdr.part_node()` を使用してノードを既に分離している必要があります。

このファンクションは、特定のノードのメタデータをローカルデータベースから削除します。ノードはローカルノードにすることができます。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。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。

!!! Note * BDR4 は、一度に最大 1024 のノードレコード (ACTIVE と PARTED の両方) を持つことができます。これは、各ノードにタイムハードシーケンスで使用するための一意のシーケンス番号が割り当てられているためです。PARTED ノードは現時点で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、このファンクションを使用してそれらのレコードを削除できます。

bdr.create_node_group

このファンクションは、ローカルノードをグループの唯一のメンバとして BDR グループを作成します。

あらすじ

```
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。ノードと同様に 名前、有効なグループ名は小文字、数字で構成する必要がありますおよびアンダースコアのみ。
- **parent_group_name**-サブグループの親グループの名前。
- **join_node_group**-これは、ノードが結合するかどうかを決定するのに役立ちますそれによって作成されているグループ。デフォルト値は true です。これが使用されますノードが結合したくないシャードグループを作成しているとき。parent_group_name が指定されている場合のみ、これは false になります。
- **node_group_type**-有効な値は、NULL、'subscriber-only'、'datanode'、「読み取りコーディネーター」および「書き込みコーディネーター」。「加入者専用」タイプはからの変更のみを受け取るノードのグループを作成するために使用されますクラスター内の完全に参加したノードが、レプリケーションを送信しない他のノードへの変更。詳細については、「サブスクライバー専用ノード」を参照してください。データノードは、グループがシャードを表すことを意味しますが、もう一方は値は、グループがそれぞれのコーディネーターを表すことを意味します。「サブスクライバーのみ」を除き、残りの 3 つの値は将来の使用のために予約されています。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

このファンクションは、ローカルノードで実行されているローカルコンセンサスワーカーに要求を渡します。

ファンクションはトランザクションではありません。グループの作成はバックグラウンドプロセスであるため、ファンクションが終了すると、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可視されない場合があります。

グループの作成はロックを保持しません。

bdr.alter_node_group_config

このファンクションは、既存の BDR グループの設定パラメータを変更します。NULL 値を持つオプション（それらすべてのデフォルト）は変更されません。

あらすじ

```
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,
*                               check_constraints boolean DEFAULT NULL,
*                               num_writers int DEFAULT NULL,
```

(次のページに続く)

*	<code>enable_wal_decoder</code> boolean DEFAULT NULL ,
*	<code>streaming_mode</code> text DEFAULT NULL)

パラメーター

- `node_group_name`-既存の BDR グループの名前。ローカルノードはパートでなければなりませんグループの。
- `insert_to_update`-後方互換性のために予約されています。
- `update_to_insert`-後方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`-後方互換性のために予約されています。
- `check_full_tuple`-後方互換性のために予約されています。
- `apply_delay`-後方互換性のために予約されています。
- `check_constraints`-適用プロセスがコンストレインをチェックするかどうか複製されたデータを書き込むとき。このオプションは非推奨であり、今後無効または削除される可能性があります BDR のバージョン
- `num_writers`-サブスクリプションバッキング用の並列ライターの数このノードグループ、-1 はデフォルトを意味します（GUC `bdr.writers_per_subscription`）が使用されます。有効な値
- 1 または正の整数です。
- `enable_wal_decoder`-WAL デコーダープロセスを有効/無効にします。に注意してください `streaming_mode` が既にある場合、WAL デコーダープロセスを有効にできません。有効。
- `streaming_mode`-ラージ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、ラージトランザクションはまだ進行中にデコードされ、変更は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ます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。もし値は `writer` に設定され、着信変更は直接に送信されますライターのいずれか（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 `no` ライタはストリーミングトランザクションをフリーにハンドル、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。もし値は `auto` に設定され、BDR はインテリジェントに選択を試みます `file` および `writer` は、トランザクションプロパティに応じて使用可能リソース WAL の場合、`streaming_mode` は有効にできないことに注意してください。デコーダーは既に有効になっています。
 - 詳細については、[Transaction Streaming](#) を参照してください。

注

このファンクションは、グループコンセンサスメカニズムにデフォルトを変更する要求を渡します。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。

このファンクションはトランザクションではありません。要求はバックグラウンドで処理されるため、ファンクション呼び出しは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からすぐには可視ない場合があります。

このファンクションはロックを保持しません。

!!! Warning * このファンクションを使用して `apply_delay` 値を変更する場合、その変更はすで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。通常、この値はテスト以外では使用されないため、この制約は稼働環境の使用にはほとんど影響しないことに注意してください。

bdr.join_node_group

このファンクションは、既存の BDR グループにローカルノードを結合します。

あらすじ

```
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`-既存の（ソース）ノードへの接続文字列を指定します BDR グループで、ローカルノードを追加します。
- `node_group_name`- BDR グループのオプション名前。デフォルトは NULL ソースに存在する情報からグループ名前を自動検出しようとします ノード
- `pause_in_standby`-オプションとして、結合プロセスに、ロジカルスタンバイノード。後でフルメンバに昇格できます。
- `wait_for_completion`-結合プロセスが完了するのを待つ戻る; デフォルトは true です。
- `synchronize_structure`-構造（スキーマ）の同期の種類を設定します結合中に行われるべきです。有効なオプションは、同期する「すべて」です完全なデータベース構造、および同期しない「なし」構造（ただし、データは引き続き同期されます）。

`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返される非同期呼び出しです。結合の進行状況は、ログおよび `thebdr.state_journal_details` 情報ビューで確認できます。または、`bdr.join_node_group()` が戻ったら、`bdr.wait_for_join_completion()` ファンクションを呼び出すことで確認できます。

注

このファンクションは、`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。加えられた変更は、コンセンサスメカニズムを介してグローバルに複製されます。

このファンクション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` に設定された場合、または後で `bdr.wait_for_join_completion` を呼び出して、変更がローカルトランザクションにのみ表示されます。

ノードは単一グループのパートにしかねないため、このファンクションは各ノードで 1 回だけ呼び出すことができます。

ノード結合は、BDR グループのロックを保持しません。

bdr.promote_node

このファンクションは、ローカルロジカルスタンバイノードを BDR グループの完全なメンバに昇格させます。

あらすじ

<pre>bdr.promote_node(wait_for_completion boolean DEFAULT true)</pre>
--

注

このファンクションは、グループコンセンサスメカニズムにデフォルトを変更する要求を渡します。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。

このファンクションはトランザクションではありません。プロモーションプロセスはバックグラウンドで行われるため、ロールバックできません。`wait_for_completion` が `true` に設定された場合、または後で `bdr.wait_for_join_completion` を呼び出して、変更が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前進するのを防ぎます。

bdr.wait_for_join_completion

このファンクションは、ローカルノードの結合プロシージャが完了するまで待機します。

あらすじ

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

パラメーター

- **verbose_progress**-オプションで個々のステップに関する情報を出力します結合プロシージャの間に取られます。

注

このファンクションは、ローカルノードのチェック状態が**bdr.create_node_group**、**bdr.join_node_group** または **bdr.promote_node** によって設定された **targetstate** 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（「パーツ」）しますが、ノードからデータを削除しません。

このファンクションは、削除されるノードを含む BDR グループ内のアクティブなノードから呼び出すことができます。ただし、明確に **make** に、ノードが **PARTED** されると、クラスタ内の他のノードをパートすることはできません。

!!! Note * あなたは **wait_for_completion** を **false** に設定しなければならないローカルノードを「分割」しています。そうでなければエラーになります。

!!! Warning * のアクションは永続的です。ノード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。

あらすじ

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

パラメーター

- **node_name**-パートする既存のノードの名前。結果-true の場合、ファンクションはノードはクラスターから完全に切り離されます。そうでない場合、ファンクションは単に別れプロシージャをスタートし、待機せずにすぐに結果ます。ローカルノードで実行する場合、または強制を使用する場合は、常に false に設定します。
- **force**-ローカルノードのノードを強制的に削除します。これによりコンセンサスに到達できなかった場合、またはノードが別れた場合、ノードの状態はローカルにプロセスが停止しました。

!!! Warning * **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、他の方法でノードを削除できないビザンチン障害から回復する場合にのみ使用する必要があります。 *

注

このファンクションは、指定されたノードを分割するために、グループコンセンサスメカニズムに要求を渡しノード。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。分離プロセスはバックグラウンドで行われるため、ロールバックできません。分割プロセスによって行われた変更は、**wait_for_completion** が true に設定されている場合にのみローカルトランザクションに可視されます。

force が true に設定されている場合、コンセンサスエラーが発生すると、このファンクション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ノード。このような場合、ファンクションはトランザクション関数であり（ファンクション自分自身がノードの状態を変更するため）、ロールバックできます。ファンクションが、**force** が true に設定されて既に分割中のノードで呼び出された場合、指定されたノードをローカルに分割してマークし、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）、または何らかの理由で分割プロセスが停止した場合にのみ役立ちます。ただし、他のノードが特定のノードから欠落しているデータを再同期する必要があるため、書き込みを受信していた分離ノードが書き込みを行うときに、スタックすることなく分離プロセスに時間がかかる可能性があることに注意することが重要です。強制分割はこの再同期を完全にスキップするため、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。

分離プロセスはロックを保持しません。

bdr.alter_node_interface

このファンクションは、指定されたノードの接続文字列（DSN）を変更します。

あらずじ

```
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

このファンクションはローカルノードでのみ実行され、変更はローカルノードでのみ行われノード。これは、通常、変更中のノードを含む BDR グループ内のすべてのノードで実行する必要があ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ナルです。ロールバックでき、変更は現在のトランザクションに表示されます。

このファンクションは、ローカルノードをロックします。

bdr.alter_subscription_enable

このファンクションは、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。サブスクリプションの再開とも呼ばれます。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。

あらすじ

```
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 の場合（デフォルト）、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

このファンクションは複製されず、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）にのみ影響します。

このファンクションはトランザクションです。ロールバックでき、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開始されます。

bdr.alter_subscription_disable

このファンクションは、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。必要に応じて、無効なサブスクリプションに関連付けられているすべてのワーカーをすぐに停止することもできます。pausesubscription と呼ばれます。サブスクリプションがすで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。

あらすじ

```
bdr.alter_subscription_disable(  
    *   subscription_name name DEFAULT NULL,  
    *   immediate boolean DEFAULT false  
)
```

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 の場合（デフォルト）、ローカルノード上のすべてのサブスクリプションは無効になります。
- **immediate**-即時を使用してアクションをすぐに強制し、停止します無効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。このオプションで true、このファンクションはトランザクションブロック内で実行できません。

注

このファンクションは複製されず、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）にのみ影響します。

このファンクションはトランザクションです。ロールバックでき、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、subscriptionworker が停止するタイミングは immediate の値に依存します。true に設定されている場合、ワーカーはすぐに停止します。false に設定すると、COMMIT の時点で停止します。

2.4.11 ノード管理コマンド

BDR は、既存ノードの物理コピー (`pg_basebackup`) を介してノードを BDR グループに追加し、既存ノードの物理スタンバイを BDR グループの新しいノードに変換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。

ユーザは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` コマンドを使用して、高速ブロックレベルコピーオペレーションを使用してディレクトリを埋めます。

空のデータディレクトリから開始する場合、選択的バックアップオプションを選択すると、そのデータベースのみがソースノードからコピーされます。除外されたデータベースは削除され、新しいノード (EDB Postgres Extended) でクリーンアップされます。

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すでに物理的なスタンバイノードとしてアクティブになっている場合、Postgres 自分自身を管理する `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。最初はキャッチアップを待機し、その後 BDR グループに参加する前にマスタノードにプロモートします。--standby オプションを使用すると、既存の物理スタンバイがロジカルスタンバイノードに変わることにご注意ください。それは新しい BDR ノードの終わり `stateof` を示します、指定されたデータディレクトリの開始状態ではありません。

このコマンドは、すべての PostgreSQL ネイティブロジカル複製サブスクリプションをデータベースから削除します (または、-S オプションが使用されている場合は単に無効にします)。また、レプリケーション起点元とスロットも削除します。

あらすじ

```
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されるデータディレクトリ。それ空のディレクトリまたは存在しないディレクトリのいずれか、または `pg_basebackup -X stream` コマンド (必須)。
- `-l`, `--log-file=FILE`-ロギングに `FILE` を使用します。デフォルトは `bdr_init_physical_postgres`。ログ。
- `-n`, `--node-name=NAME`-新しく作成されたノードの名前 (必須)。

- `--replication-sets=SETS`-レプリケーションのコンマ区切りリストの名前使用する名前を設定します。指定しない場合、すべてのレプリケーションセットが使用されます。
- `--standby`-完全ではなく、ロジカルスタンバイ（ノードのみを受信）を作成します送受信ノード。
- `--node-group-name`-結合グループ。デフォルトはソースノードと同じグループです。
- `-s, --stop`-初期化が完了したらサーバーを停止します。
- `-v`-ロギングの詳細度を上げます。
- `-L`-と組み合わせて使用すると、選択的な `pg_basebackup` を実行します空/存在しないデータディレクトリ (`-D` オプション)。(EDB Postgres Extended)
- `-S`-ロジカル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、単に無効にしますそれら。

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）

設定ファイルの上書き

- 新しい `pg_hba.conf` への `--hba-conf -path`
- `--postgresql-conf`-新しい `postgresql.conf` へのパス
- `--postgresql-auto-conf`-新しい `postgresql.auto.conf` へのパス

注

コマンドで指定されたレプリケーションセット名は、ノードが 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影響しません。これは、`bdr_init_physical` が独自の `basebackup` を作成するか、既存の `basebackupis` が新しい BDR ノードに昇格されるかに関係なく当てはまります。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のみ、公開およびサブスクライブされるデータに影響します。この動作は、レプリケーションセットが非論理結合で使用される方法、つまり `bdr.join_node_group()` を使用する場合は異なります。

不要なテーブルは、結合の完了後に演算子によって切り捨てられる場合があります。`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確認し、サブスクライブ先の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識別します。次の理由により、テーブルを削除するのではなく、切り捨てることを強くお勧めします。

1. 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、他のノードにテーブルを誤ってドロップする可能性があります。後でテーブルをレプリケーションセットに追加し、ノードの一部のサブセットにドロップした場合、レプリケーションに追加する前に、DDL の競合を作成せずにそれらのノードでのみテーブルを再作成するように注意する必要がありますセット。

レプリケートされていないテーブルを切り捨て、存在するが空のままにする方がはるかに簡単で安全です。

BDR の将来のバージョンでは、物理的結合の選択されたレプリケーションセットに含まれないテーブルを自動的に削除または削除する可能性があるため、アプリケーションはここに記載されている動作の詳細に依存しません。

2.5 DDL Replication

DDL は、「データ定義言語」の略です。データベースオブジェクトを作成、変更、削除する SQL language のサブセットです。

操作上の利便性と正確さのために、BDR は次の例外を除き、ほとんどの DDL actions を複製します。

- 一時的またはログに記録されていないリレーション
- 特定の、主に長時間実行さ DDL ステートメント（以下のリストを参照）
- ロックコマンド（LOCK）
- テーブル保守コマンド（VACUUM、ANALYZE、CLUSTER、REINDEX）オートバキュームのアクション
- 操作コマンド（CHECKPOINT、ALTER SYSTEM）
- データベースまたは表領域に関連するアクション

自動 DDL レプリケーションにより、DDL の変更をすべてのノードに手動で配布し、一貫性を保証ことなく、DDL の変更を簡単に確認できます。

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複製されます DDLDDL レプリケーションを追加する必要があります。[DDL Replication Filtering] 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 PostgreSQL とは大きく異なり、同じものとして扱うことは、BDR の最も一般的な運用上の問題です。

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、ステートメント自分自身を複製することです。ほとんどの場合、これは非常にうまく機能しますが、すべてのノードで DDL を同様に実行必要があるという要件があります。より微妙な点は、拡張機能によって導入されたデータ型（つまり、ビルトインではない）を含む、すべてのデータ型固有のパラメータ設定に関して DDL が不変でなければならないことです。例、DDL ステートメントは各ノードで使用する defaultencoding で正しく実行必要があります。

2.5.1 DDL レプリケーションオプション

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

`bdr.ddl_replication = on` がデフォルトであり、DDL をデフォルトのレプリケーションセットに複製します。これはデフォルトですべてのノードを意味します。デフォルト以外の複製セットは、`aDDL filter` が定義されていない限り、DDL を複製しません。

関数 `bdr.replicate_ddl_command()` を使用して、DDL を特定のレプリケーションセットに複製することでもできます。これは、ノードがダウンしているときに DDL コマンドを実行したい場合、またはノードまたは rep セット（たとえば、site1 のすべてのノード）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
```

自動 DDL レプリケーションをスキップして、`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することは可能ですが、推奨されません。

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

設定 `make` と、BDR がグローバルロックと実行された DDL コマンドの複製の両方をスキップするため、すべてのノードで DDL を手動で実行する必要があります。

!!! Warning * グローバルロックなしで各ノードで DDL を手動で実行すると、競合する DDL または DML が同時に実行されると、BDR グループ全体の複製が停止する可能性があります。

`bdr.ddl_replication` パラメータは、`bdr_superuser`、スーパーユーザ、または構成ファイルでのみ設定できます。

2.5.2 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 サーバーとは異なります。これは、集中ロックやトランザクションコーディネーターを使用しない非同期マルチマスターレプリケーションに基づいています。これは DDL の実行時に重要な意味を持ちます。

並行して実行される DDL は、BDR で引き続き実行されます。DDL の実行は、実行時に各ノードの並列オペレーションに影響するパラメーターを尊重するため、ノード間の設定の違いが顕著になる場合があります。

競合する DDL の実行を防止ニーズがあります。そうしないと、DDL レプリケーションがエラーを引き起こし、レプリケーションが停止します。

BDR は、これらの問題に対する 3 つのレベルの保護を提供します。

`ddl_locking = 'dml'` は操作に最適なオプションで、一度に 1 つのノードからのみ DDL を実行するときに使用できます。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、この設定を使用して、ノード間の競合が発生しないようにすることをお勧めします。ノード内の競合は、PostgreSQL によってすでに処理されています。

`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る可能性があり、正確性を保証したい場合に最適です。

`ddl_locking = off` は最も厳密では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するため、パフォーマンスのオーバーヘッドを回避し、新しい空のデータベーススキーマを作成する際に便利なオプションになります。

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

`bdr.replicate_ddl_command` を使用する場合、その関数に渡される 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、3 番目の引数を介してこのパラメーターを直接設定できます。

2.5.3 DDL ロックの詳細

BDR を使用して複製された DDL の正確さを強制するために使用されるロックには、2 種類があります。

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クラスターで他の DDL が実行されないようにします。これにより、一般的な場合は完全に正確になりますが、多くのシンプル場合には明らかに厳しすぎます。BDR は、スキーマの変更が行われるトランザクションで初めて DDL 操作のグローバルロックを取得します。これにより、クラスター内の DDL 実行トランザクションを効果的にシリアル化します。言い換えると、DDL の実行中は、異なるテーブルに影響を与える場合でも、どのノード上の他の接続も他の DDL コマンドを実行できません。

DDL 操作のロックを取得するために、DDL を実行している BDR ノードは、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利を付与するように依頼します。ロック要求は通常のレプリケーションストリームを介して送信され、ノードもレプリケーションストリームを介して応答します。そのため、ノード（または少なくとも過半数のノード）が多くのレプリケーション遅延なしで実行されていることが重要です。そうしないと、ノードが DDLlock を取得するのに非常に長い時間がかかる場合があります。ノードの大部分が一致すると、DDL の実行が実行されます。

DDL ロックの順序付けは、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シーケンスで実行されます。

DDL を実行しているノードがクラスター内で実行されているすべての PriorDDL のオーダーを保証に、以前の DDL を実行していたノードに追いつくまで待機しノード。現在の DDL を実行しているノードが、前の DDL を実行したノードとの関係で遅れている場合、ロックを取得するのに非常に長い時間がかかります。したがって、単一のノードまたは他のノードで発生する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` のいずれかであり、一意性制約、検査制約、NOT NULL 制約などの制約を追加または変更する場合など、DDL ステートメントによって飛行中の DML ステートメントが失敗する場合に使用されます。一度に 1 つのリレーション。リレーション保証は、キューに変更があり、エラーでレプリケーションが停止する可能性がある間、DDL が実行されないようにします。

テーブルのグローバル DML ロックを取得するために、DDL を実行している BDR ノードは、BDR グループ内の他のすべてのノードに連絡し、書き込みに対してテーブルをロックするように要求し、そのテーブルに対するすべての保留中の変更が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いており、DML ロックの発信者はテーブルのスキーマ変更を自由に実行し、それらを他のノードに複製します。

グローバル DML ロックは各ノードのテーブルで排他ロックを保持しているため、DML、他の DDL、VACUUM、およびそのテーブルに対するインデックスコマンドが実行中にブロックされることに注意してください。これは、通常は排他ロック以上を使用し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にも当てはまります。

保留中の DML 操作が排出されるのを待つには時間がかかるか、レプリケーションが現在遅れている場合は長くなります。つまり、行の表現形式とコンストレインに影響するスキーマ変更は、データの変更とは異なり、構成されたすべてのノードが到達可能であり、合理的に維持されている場合にのみ実行できます。ノードがダウンしている間にそのような DDL コマンドを絶対に実行する必要がある場合、ダウンノードを最初に構成から削除する必要があります。

DDL ステートメントが複製されない場合、グローバルロックは取得されません。

ロック動作は、*Executing DDL on BDR systems* で説明されているように、`bdr.ddl_locking` パラメータによって指定されます。

- `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。

また、一部の BDR 関数は DDL を変更するため、これらの関数には DDL ロック動作が適用されます。これは、各ファンクションのドキュメントに記載されています。

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

`ddl_locking = off` は、DDL を実行データベースオブジェクトで競合する DDL および競合する DML 操作がないことをユーザが保証できる場合にのみ安全です。ロックをオフにしてから問題が発生した場合、データの飛行中の変更が失われる可能性があります。発生した問題は、ユーザアプリケーションチームが解決する必要があります。

場合によっては、並行して実行する DDL を適切にシリアル化できます。これらのシリアル化エラーが発生した場合、DDL が再実行される場合があります。

DDL レプリケーションは、昇格され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 のように機能することに注意してください。つまり、DDLreplication がアクティブな場合、グローバルロックの取得を試み、そのアクションが複製されます。複製された関数の完全なリストは、*DDL のように動作する BDRFunctions* 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 は、グローバル DML ロックを必要としません。

グローバル DDL ロックおよびグローバル DML ロックの監視については、[Monitoring](#) の章で説明しています。

2.5.4 DDL の影響を最小限に抑える

すべてのデータベースに対する適切な運用上のアドバイス、これらのポイントは BDR でさらに重要になります。

DDL のインパクトを最小限に抑えるため、DDL を実行するトランザクションは短くする必要があります。行の多くの変更と組み合わせるのではなく、長い間避けるべきです外部キーまたは他の制約の再チェックを実行します。

- ALTER TABLE の場合は、ADD CONSTRAINT NOT VALID を使用してから、別の ADD CONSTRAINT を使用するだけでなく、VALIDATE CONSTRAINT を使用したトランザクション。VALIDATE CONSTRAINT はすべてのノードで再生されるまで待機することに注意してください。確認の受信に顕著な遅延を与えます。
- インデックスを作成するときは、可能な限り CONCURRENTLY オプションを使用します。

長時間実行される 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、各ノードで個別に DDL ステートメントを実行です。以下の例に示すように、単一の SQL ステートメントを使用して実行できますが、グローバルロックルールが適用されること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);
$ddl$);
```

上記の bdr.run_on_all_nodes () テクニックを CREATEINDEX CONCURRENTLY で使用することをお勧めし稼働。CREATEINDEX CONCURRENTLY はマルチトランザクションコマンドであるため、DDL レプリケーションをセッション全体で無効にする必要があることに注意してください。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。AccessExclusiveLocks が保持されているため、REINDEX の使用は避けてください。

代わりに、PG12+または2QPG11+で使用可能な REINDEX CONCURRENTLY を使用する（またはreindexdb-concurrently）必要があります。

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

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

複数の DDL ステートメントは、個々のステートメントとして起動されるのではなく、単一のトランザクションにまとめられることでメリットが得られる可能性があるため、DDL ロックは 1 回だけベータ化する必要があります。これは、テーブルレベルのロックが通常の操作に干渉する場合は望ましくない場合があります。

DDL がシステムを長時間保持しすぎると、psql の Control-C または pg_cancel_backend() を使用して他のステートメントをキャンセルするため、元のノードで DDL をキャンセルすることが可能かつ安全です。他のノードから DDL ロックをキャンセルすることはできません。

（オプションの）グローバルロックタイムアウト設定で、グローバルロックをロックする wait-for がキャンセル長さトランザクションまでの時間を制限します。グローバルロックを保持し、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すべて、値をゼロに設定することで無効にできます。

発信元ノードで DDL オペレーション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、リプレイを確認するまで待機する必要があります。これが、DDL トランザクションを短く高速に保つことが重要な理由です。

2.5.5 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 のいずれか) を取得した後にダウンした場合、ロックはアクティブのままです。タイムアウトが設定されていても、グローバルロックはタイムアウトしません。ノードが復旧した場合、保持しているすべてのグローバルロックを自動的にリリースします。

長時間 (または永久に) ダウンしたままになっている場合は、グローバルロックをリリースオーダーに BDR グループからノードを削除します。これが、SET コマンドを `bdr_superuser` として使用して `bdr.ddl_locking` 値を更新する緊急 DDL を実行する理由の 1 つである可能性があります。

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、ロックを要求するコマンドの実行は、ノードがアップしたかのように続きます。

前のセクションで述べたように、グローバル DDL ロックは応答するノードの過半数のみを必要とするため、大部分が実行中で到達可能である限り、クラスターのパートがダウンしていても機能しますが、DML ロックはクラスタ全体が利用可能です。

グローバル DDL またはグローバル DML ロックがあり、別のノードがダウンした場合、コマンドは正常に続行し、ロックが解除されます。

2.5.6 ステートメント固有の DDL レプリケーションの懸念

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

BDR は、データベース上でアクティブなときに一部の DDL ステートメントが実行されないようにします。これにより、正しく複製できないステートメントや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。一部の制限付きでサポートされているステートメントは、[制限付きの DDL ステートメント] で説明されています。一方、BDR で完全に禁止されているコマンドは、禁止されている DDL ステートメントでカバーされています。

ステートメントが BDR で許可されていない場合、同じことを行う別の方法を見つけることがしばしば可能です。例、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が、通常は、動作するシリーズの独立した `ALTER TABLE` および `UPDATE` ステートメントとして言い換えることができます。

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

一時オブジェクトを参照または一時オブジェクトに依存する DDL は、BDR によって複製できず、DDL 複製を有効にして実行するとエラーがスローされることに注意してください。

2.5.7 BDR DDL コマンド処理マトリックス

次の表は、許可されるユーティリティまたは DDL コマンド、複製されるコマンド、および複製時に使用されるグローバルロックの種類を示しています。

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。このようなコマンドにはすべて、次の表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
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

次のページに続く

表 2 – 前のページからの続き

Command	Allowed	Replicated	Lock
ALTER QUEUE 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	[De-tails](#bdr_ddl_allowed_AlterSeqStmt)	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	[De-tails](#bdr_ddl_allowed_AlterTableStmt)	Y	[De-tails](#bdr_ddl_lock_relation_AlterTableStmt)
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
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	[De-tails](#bdr_ddl_can_replicate_comment)	DDL

次のページに続く

表 2 – 前のページからの続き

Command	Allowed	Replicated	Lock
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
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

次のページに続く

表 2 – 前のページからの続き

Command	Allowed	Replicated	Lock
CREATE 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	[De-tails](#bdr_ddl_allowed_CreateSeqStmt)	Y	DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	[De-tails](#bdr_ddl_allowed_CreateStmt)	Y	DDL
CREATE TABLE AS	[De-tails](#bdr_ddl_allowed_CreateTableAsStmt)	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
CREATE TYPE BODY	Y	Y	DDL
CREATE USER MAPPING	Y	Y	DDL
CREATE VIEW	Y	Y	DDL
DEALLOCATE	Y	N	N
DEALLOCATE ALL	Y	N	N
DECLARE CURSOR	Y	N	N
DISCARD	Y	N	N

次のページに続く

表 2 – 前のページからの続き

Command	Allowed	Replicated	Lock
DISCARD ALL	Y	N	N
DISCARD PLANS	Y	N	N
DISCARD SE- QUENCES	Y	N	N
DISCARD TEMP	Y	N	N
DO	Y	N	N
DROP ACCESS METHOD	Y	Y	DDL
DROP AGGREGATE	Y	Y	DDL
DROP CAST	Y	Y	DDL
DROP COLLATION	Y	Y	DDL
DROP CONSTRAINT	Y	Y	DDL
DROP CONVERSION	Y	Y	DDL
DROP DATABASE	Y	N	N
DROP DATABASE LINK	Y	Y	DDL
DROP DIRECTORY	Y	Y	DDL
DROP DOMAIN	Y	Y	DDL
DROP EVENT TRIG- GER	Y	Y	DDL
DROP EXTENSION	Y	Y	DDL
DROP FOREIGN DATA WRAPPER	Y	Y	DDL
DROP FOREIGN TA- BLE	Y	Y	DDL
DROP FUNCTION	Y	Y	DDL
DROP INDEX	Y	Y	DDL
DROP LANGUAGE	Y	Y	DDL
DROP MATERIALIZED VIEW	Y	N	N
DROP OPERATOR	Y	Y	DDL
DROP OPERATOR CLASS	Y	Y	DDL
DROP OPERATOR FAMILY	Y	Y	DDL
DROP OWNED	Y	Y	DDL
DROP PACKAGE	Y	Y	DDL
DROP PACKAGE BODY	Y	Y	DDL
DROP POLICY	Y	Y	DDL
DROP PROCEDURE	Y	Y	DDL
DROP PROFILE	Y	Y	DDL
DROP PUBLICATION	Y	Y	DDL
DROP QUEUE	Y	Y	DDL

次のページに続く

表 2 – 前のページからの続き

Command	Allowed	Replicated	Lock
DROP QUEUE TABLE	Y	Y	DDL
DROP REDACTION POLICY	Y	Y	DDL
DROP RESOURCE GROUP	Y	N	N
DROP ROLE	Y	Y	DDL
DROP ROUTINE	Y	Y	DDL
DROP RULE	Y	Y	DDL
DROP SCHEMA	Y	Y	DDL
DROP SEQUENCE	Y	Y	DDL
DROP SERVER	Y	Y	DDL
DROP STATISTICS	Y	Y	DDL
DROP SUBSCRIPTION	Y	Y	DDL
DROP SYNONYM	Y	Y	DDL
DROP TABLE	Y	Y	DML
DROP TABLESPACE	Y	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	[De-tails](#bdr_ddl_can_replicate_explain_stmt)	[De-tails](#bdr_ddl_lock_explain_stmt)
FETCH	Y	N	N
GRANT	Y	[De-tails](#bdr_ddl_can_replicate_grant)	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	N

次のページに続く

表 2 – 前のページからの続き

Command	Allowed	Replicated	Lock
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	[De-tails](#bdr_ddl_can_replicate_grant)	DDL
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	[De-tails](#bdr_ddl_can_replicate_seclabel)	DDL
SELECT INTO	[De-tails](#bdr_ddl_allowed_CreateTableAsStmt)	Y	DDL
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	[De-tails](#bdr_ddl_can_replicate_truncate_stmt)	[De-tails](#bdr_ddl_lock_truncate_stmt)
UNLISTEN	Y	N	N
VACUUM	Y	N	N

</div>

ALTER SEQUENCE

通常、ALTER SEQUENCE はサポートされますが、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 TABLE

通常、ALTER TABLE コマンドは許可されます。ただし、サポートされていないサブコマンドがいくつかあります。

</ div>

ALTER TABLE 禁止コマンド

現在、ALTER TABLE の一部のバリエーションは BDR ノードでは許可されていません。

- **ADD COLUMN ... DEFAULT (non-immutable expression)**-これは許可されていません現在、異なるノードで異なるデータが生成されます。見る推奨される回避策については、*Adding a Column*。
- **ADD CONSTRAINT ... EXCLUDE**-現時点では、除外コンストレインはサポートされていません。非同期システムでは、除外コンストレインはあまり意味がありませ *make*。リプレイできない変更につながります。
- **ALTER TABLE ... SET WITH[OUT] OIDS**-同じ理由でサポートされていません CREATE TABLE のように。
- **ALTER COLUMN ... SET STORAGE external**-列が次の場合は拒否されますテーブルのレプリカアイデンティティの列の 1 つ。
- **RENAME**-自動パーティション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テーブルのスキーマを設定できません。
- **ALTER COLUMN ... TYPE**-列のタイプの変更は、コマンドにより、テーブル全体が書き換えられます。バイナリ強制互換ではありません。バイナリ強制互換変更は、一方向にのみ許可されることに注意してください。例、VARCHAR (128) から VARCHAR (256) への変更はバイナリ強制互換であるため、許可されますが、VARCHAR (256) から VARCHAR (128) への変更はバイナリ強制互換ではありませんしたがって、通常は許可されていません。複製されていない ALTER COLUMN ... TYPE の場合列が新しい型に自動的にキャストできる場合は許可されます (USING 句は含まれません)。例については、以下を参照してください。テーブルの書き換えは、大きなテーブルで拡張 AccessExclusiveLock を使用するため、このようなコマンドいずれにしても、高可用性データベースでは実行不可能です。推奨される回避策については、*Changing a Column's Type* を参照してください。
- **ALTER TABLE ... ADD FOREIGN KEY**-現在のユーザが持っていない場合はサポートされません被参照テーブルを読み取るパーミッション、または被参照テーブルの場合現在のユーザがバイパスできない RLS 制限が有効になっています。

次の例は、timestamp 型の定数数値を timestamptz 型の列に追加しようとするため失敗します。timestamp と timestamptz の間のキャストは、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 以降では、CHECK や FOREIGN KEY コンストレインなどの特定の種類のコンストレインを、DML ロックを取得せずに追加できます。ただし、これには最初に NOT VALID 制約を作成し、次に ALTER

TABLE ... VALIDATE CONSTRAINT コマンドを介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、*Adding a CONSTRAINT* を参照してください。

</div>

ALTER TABLE ロック

ALTER TABLE の次のバリエントは、DDL ロックのみを取得し、aDML ロックは「**」ではありません：

- ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT
- ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression
- ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT
- 書き換えが不要な場合は ALTER TABLE ... ALTER COLUMN ... TYPE (現在、EDB Postgres Extended および EDB Postgres Advanced でのみ利用可能)
- 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ントは、変更対象のテーブルで DML ロックを取得します。ALTER TABLE の一部のバリエントには、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互換であり、したがってテーブルの書き換えを行わないため、カタログのみの変更として実行されるため、機能します。

```
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互換ではないため、上記のコマンドを逆にするためにこの変更を行うことはできません。

```
ALTER TABLE foo ALTER COLUMN description TYPE VARCHAR(20);
```

推奨される回避策については、後で参照してください。

一般的な環境およびレプリケートされない環境で可能なさまざまなタイプの ALTER TABLE ... ALTER COLUMN TYPE (ATCT) 操作のコンテキストを提供すると便利です。

一部の ATCT 操作は、基になる列タイプのメタデータのみを更新し、基になるテーブルデータの書き換えを必要としません。これは、通常、既存の列型とターゲット型がバイナリ強制変換可能な場合です。例：

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

バイナリ型の保磁力のため、列タイプを VARCHAR または TEXT datatypes に変更してもかまいません。繰り返しますが、これは基になる列タイプの単なるメタデータの更新です。

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

ただし、col2 のサイズを縮小する場合は、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与えるには、typeINTEGER を使用した上記の 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 オペレーションは、上記と同様の方法で失敗します。

```
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
```

上記の両方の失敗例では、現在のタイプからターゲットタイプへの自動割り当てキャストが存在します。ただし、最終的な基本データの書き換えを引き起こす最終的な強制力はありません。

このような場合、制御された DBA 環境では、すべてのノードで非複製環境でこの列のタイプに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することができます。DDL がレプリケートされず、列タイプの変更が上記のように自動的にキャスト可能なものである場合、同じテーブルの他のノードでの同時アクティビティとともに、変更を実行するノードでローカルに書き換えを許可することができます。その後、この複製されない ATCT オペレーションをすべてのノードで1つずつ繰り返して、BDR クラスター全体にわたって列タイプの望ましい変更をもたらすことができます。これは書き換えを伴うため、アクティビティはまだ短い期間 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。上記の仕様が整ったら、次のように非複製者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 BDR cluster do:
SET bdr.ddl_replication TO FALSE;
```

(次のページに続く)

(前のページからの続き)

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
RESET bdr.ddl_replication;
-- done
```

多くのデータ型で利用可能な自動割り当てキャストにより、このローカルの複製されない ATCT オペレーションは、さまざまな変換をサポートします。また、自動割り当てキャストがないため、USING 句を使用する ATCT 操作は失敗する可能性が高いことにノートしてください。自動割り当てキャストを使用した一般的な変換のいくつかを以下に示します。

```
-- foreach node in BDR 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
```

上記は、レプリケートされていない環境で許容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。明らかに、すべての ATCT 操作が機能するとは限りません。DDL レプリケーションを無効にしても、自動割り当てが不可能な場合は失敗します。そのため、数値型からテキスト型への変換は非複製環境で機能しますが、テキスト型から数値型への逆変換は失敗します。

```
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ますが、基になるテーブルデータに新しいデータタイプに割り当てることができない値が含まれる場合、書き換えが失敗する可能性があることにノートすることが重要です。例、列の現在の型は VARCHAR(256) である可能性があり、複製されない ATCToperation を試して VARCHAR(128) に変換しました。テーブルに 128 バイトよりも広い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)
```

基になるテーブルデータが新しい型の特性を満たす場合、書き換えは成功します。ただし、他のノード（まだ複製されていないローリングデータタイプアップグレードを実行していない）がこのローカル ATCT オ

ペレーションに 128 バイトを超える新しいデータを同時に導入すると、複製が失敗する可能性があります。これにより、クラスター内で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タイプのアップグレード操作を実行する際には、データベースレベルおよびアプリケーションレベルでのデータタイプの制限と特性を認識することが重要です。制御された完全に認識された DBA 環境で、このような ATCT 操作を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、失敗するとテーブルの書き換えが長時間続く可能性がある特定の変更をバックアウトすることに注意する必要があります。

また、上記の暗黙的なキャスト可能な ALTER アクティビティは、トランザクションブロックで実行できないことにノートしてください。

!!! Note * これは現在、 EDB Postgres Extended および EDB Postgres Advanced でのみ機能します。

ALTER TYPE

ユーザーはノートは複製されますが、PostgreSQL はこれらの依存関係を記録しないため、そのデータタイプを使用するすべてのテーブルにグローバル DML ロックが適用されないことに注意してください。以下の回避策を参照してください。

</ div>

コメントオン

COMMENT ON のすべてのバリエーションが許可されますが、edCOMMENT ON TABLESPACE/DATABASE/LARGE OBJECT は複製されません。

</ div>

シーケンスの作成

通常、CREATE SEQUENCE はサポートされますが、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。

</ div>

テーブルの作成

通常、CREATE TABLE はサポートされますが、BDR ノードでは CREATE TABLE WITH OIDS は許可されません。

</ div>

CREATE TABLE AS および SELECT INTO

CREATE TABLE AS および SELECT INTO は、EDB Postgres Extended および EDB Postgres Advanced でのみ許可され、サブコマンドも許可されている場合にのみ許可されます。

theCREATE TABLE AS が Postgres のバリエーションでサポートされていない場合は、代わりにを使用して同じ効果を実現できます。

```
CREATE TABLE mytable;
INSERT INTO mytable SELECT ... ;
```

説明

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

</div>

EXPLAIN ANALYZE レプリケーション

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

</div>

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。

</div>

付与および取り消し

通常、GRANT および REVOKE ステートメントがサポートされますが、GRANT/REVOKE ON TABLESPACE/LARGE OBJECT は複製されません。

ロックテーブル

LOCK TABLE はローカルでのみ実行され、複製されません。トランザクションコミット後に通常のレプリケーションが発生するため、LOCK TABLE は他のノードに影響を与えません。

テーブルをグローバルにロックする場合、ユーザーは `bdr.global_lock_table()` を呼び出してグローバル DML ロックを明示的に要求できます。

</div>

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ますが、SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT は複製されません。

</div>

切り捨てレプリケーション

TRUNCATE コマンドは DDL ステートメントとしてではなく DML として複製されるため、テーブルの TRUNCATE が複製されるかどうかは、影響を受ける各テーブルのレプリケーションセット設定に依存します。

</div>

TRUNCATE ロック

TRUNCATE は他の DDL と同じ方法で複製されませんが、`bdr.truncate_locking` が on に設定されている場合、グローバル DML ロックを取得する場合があります。

ロール操作ステートメント

ユーザーは PostgreSQL インスタンスのグローバルオブジェクト。つまり、BDR が個々のデータベースレベルで動作している間、複数のデータベースにまたがっています。これは、ロール操作ステートメントのハンドリングに特別な配慮がニーズであることを意味します。

BDR では、複製された DDL によって被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、存在する必要があります。

`bdr.role_replication` が有効（デフォルト）の場合、BDR はロール操作ステートメントを複製し、ロール操作ステートメントは BDR 対応データベースで実行されます*。

ロール操作ステートメントには、次のステートメントが含まれます。

- ロールの作成
- ALTER ROLE
- ドロップロール
- 権限付与
- ユーザーの作成
- ALTER USER
- ユーザーをドロップ
- グループの作成
- ALTER GROUP

- ドロップグループ

一般的に、次のいずれかです。

- システムは `bdr.role_replication = off` およびすべてのロール（ユーザおよびグループ）の変更は、外部オーケストレーションによって展開する必要があります Ansible、Puppet、Chef などのツール、または経由で明示的に複製されたツール `bdr.replicate_ddl_command(...)`; または
- システムは、BDR 対応データベースが 1 つだけになるように構成する必要があります PostgreSQL インスタンスでは `bdr.role_replication = on` とすべてを持っていますロール管理 DDL はそのデータベースで実行する必要があります。

1 つのデータベース内ですべてのロール管理コマンドを実行することを強くお勧めします。

ロールレプリケーションがオフになっている場合、管理者は、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保証必要があります。そうしないと、BDR が他のノードでロールが作成されるまで **ERROR** で停止します。

注：BDR は、BDR 対応の PostgreSQL インスタンス内の非 BDR 対応データベースで実行された場合、ロール管理ステートメントをキャプチャおよびレプリケートしません。例、DB ‘`bdrdb`’ (bdr group メンバ) および ‘`postgres`’ がある場合 (bare db)、および `bdr.role_replication = on` の場合、`bdrdb` で実行される `CREATE USER` は複製されますが、`postgres` で実行される `CREATE USER` は複製されません。

制限され DDL の回避策

BDR DDL オペレーションハンドリングの制限のいくつかは回避できます。多くの場合、オペレーション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望ましい結果が生成されます。

制約の追加

BDR 3.7.4 以降、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 段階のプロセスが必要です。

- `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID`
- `ALTER TABLE ... VALIDATE CONSTRAINT`

これらのステップは、2 つの異なるトランザクションで実行する必要があります。これらのステップは両方ともテーブルの DDL ロックのみを取得するため、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。しかし、制約を検証するオーダーに、BDR はクラスター内のすべてのノードが `ADD CONSTRAINT` コマンドを認識し、制約を検証するノードがそれらのノードに `NOT VALID` 制約を作成する前に他のすべてのノードから `replicationchanges` を適用することを保証する必要があります。そのため、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、十分な進歩を遂げている必要があります。BDR は、制約を検証する前に、`consistentstate` に到達するのを待ちます。

新しい機能を使用するには、クラスターを Raft protocolversion 24 以降で実行する必要があります。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。

!!! Note * これは現在、 EDB Postgres Extended および EDB Postgres Advanced でのみ機能します。

列を追加する

volatile デフォルトの列を追加するには、これらのコマンドを個別のトランザクションで実行します。

```
* ALTER TABLE mytable ADD COLUMN newcolumn coltype; -- Note the lack of DEFAULT or
↳ NOT NULL
* ALTER TABLE mytable ALTER COLUMN newcolumn DEFAULT volatile-expression;
* BEGIN;
  SELECT bdr.global_lock_table('mytable');
* UPDATE mytable SET newcolumn = default-expression;
COMMIT;
```

これにより、スキーマの変更と行の変更が BDR で実行可能な個別のトランザクションに分割され、aBDR グループのすべてのノードで一貫したデータが得られます。

最良の結果を得るには、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、更新をチャンクにバッチします。これは、埋め込みトランザクションで 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ことが重要です。そうしないと、他のノードのテーブルに同時に挿入された行を見逃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、次の例に、回避できる場合にテーブルの書き換えを引き起こします。

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

このステートメントは、データ型の変更ではなくテーブルの制約を制限することにより、テーブルの書き換えを回避するように書き換えることができます。

```
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;
```

バリデーションに失敗した場合は、失敗した行のみを更新できます。この手法は、length() を使用する TEXT および VARCHAR、または scale() を使用する NUMERIC データ型で使用できます。

列タイプを変更する一般的なケースでは、最初に目的のタイプの列を追加します。

```
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、NEW.newcolumn を NEW.oldcolumn に割り当てて、テーブルへの新しい書き込みが新しい列を自動的に更新するようにします。

UPDATE を一括してテーブルに組み込み、PROCEDURE を埋め込みトランザクションで使用して oldcolumn の値を newcolumn にコピーします。ワークをバッチ処理すると、大きなテーブルである場合にレプリケーションの遅延を軽減できヘルプ。ID のレンジまたは任意のメソッドで更新することは問題ありません。また、テーブル全体を小さなテーブルにまとめて更新することもできます。

CREATE INDEX ... 新しい列に必要なインデックス。ロック期間を短縮するために、各ノードで DDL レプリケーションなしで個別に実行される CREATE INDEX ... CONCURRENTLY を使用するのが安全です。

ALTER 必要に応じて、NOT NULL および CHECK コンストレインを追加する列。

BEGIN トランザクション、DROP は追加したトリガー、ALTER TABLE は列に必要な anyDEFAULT を追加し、DROP は古い列、andb_tran_6、COMMIT の順に追加します。

列をドロップするため、ビュー、プロシージャなどを再作成する必要がある場合があります。テーブルに依存します。`CASCADE` 列を削除する場合は、それを参照するすべてのものを再作成する必要がある保証注意してください。

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、ユーザ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保証必要があります。これは、bdr.wait_slot_confirm_lsn() ファンクションを使用して実現できます。

例、

```
ALTER TYPE contact_method ADD VALUE 'email';
SELECT bdr.wait_slot_confirm_lsn(NULL, NULL);
```

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ていることを保証します。

DDL のように動作する BDRFunctions

次の BDR 管理機能は DDL のように機能します。これは、DDLreplication がアクティブで、DDL フィルター設定で許可されている場合、グローバルロックの取得を試み、そのアクションが複製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。

複製セット管理

- 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`

紛争管理

- `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`

2.6 Security and Roles

BDR 拡張機能はスーパーユーザーのみが作成できますが、必要に応じて、`pgextwlist` 拡張機能をセットアップし、非スーパーユーザーが BDR を作成できるように構成することもできます。

BDR のデフォルトと PostgreSQL には、スーパーユーザアクセスは必要ありません。また、推奨もされてデフォルト BDR ん。

- `bdr_superuser` *-すべての BDR テーブルと機能にアクセスできる最高特権ロール。
- `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、および関数への読み取り専用アクセス権を持つロール。
- `bdr_monitor` *-現時点では `bdr_read_all_stats` と同じで、後で拡張ます。
- `bdr_application` * BDR を実行するアプリケーションに必要な最小限の特権。
- `bdr_read_all_conflicts` - `'bdr.conflict_history'` のすべて*の競合をビューできます。

これらの BDR ロールは、BDR 拡張機能がインストールされるときに作成されます。詳細については、以下の [BDR デフォルトロール] を参照してください。

BDR を管理するには、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準備については、ここで説明されています `b_tran_1`。

競合は、`BDR.conflict_history_summary` ビューを使用して監視できます。

2.6.1 カタログ表

システムカタログと情報スキーマテーブルは、BDR によって常に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。

2.6.2 BDRFunctions と演算子

すべての BDR 関数は `bdr` スキーマで公開されます。これらの関数の呼び出しは、`bdr` を `search_path` に入れるのではなく、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` スキーマを介して利用できるため、ユーザーは問題なく `search_path` から `public` スキーマを除外できます。

2.6.3 カタログオブジェクトに対する権限の付与

管理者は、テーブル、ビュー、関数などのカタログオブジェクトに対する明示的な権限を付与しないでください。[BDRDefault Roles] で文書化されたロールの 1 つを付与することにより、それらのオブジェクトへのアクセスを管理します。

この要件は、結合のいずれかの側のノードがまったく同じバージョンの BDR（したがって BDRcatalog）を持たない場合でも、ノードグループに参加できる柔軟性の結果です。

より正確には、個々のカタログオブジェクトに対する特権が明示的に付与されている場合、結合されているノードから抽出された対応する GRANT ステートメントが結合しているノードに適用されない可能性があるため、`bdr.join_node_group()` プロシージャは失敗する可能性があります

2.6.4 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 BDR が実行され、`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、`.CREATE USER` および `CREATE ROLE` コマンドは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 インスタンスの他のデータベースで実行される場合、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、それらは複製されません。

新しい BDR ノードが BDR グループに参加すると、`bdr_init_physical` を使用してノードが追加されない限り、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が、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、どのノードを新しいノードにコピーするか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
```

その後、ファイル `roles.sql` を編集して、不要なユーザーを削除してから、新しく作成されたノードで再実行できます。ID および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。

2.6.5 ロールとレプリケーション

ユーザが実行した DDL の変更は、各ノードで同じユーザとして適用されます。

テーブルへの DML の変更は、ターゲットノードのテーブル所有ユーザとして複製されます。ただし、強制ではありませんが、各ノードの同じユーザがテーブルを所有することをお勧めします。

テーブル A がノード 1 のユーザ X に所有され、ノード 2 のユーザ Y に所有されている場合、ユーザ Y がユーザ X より高い特権を持っている場合、これは権限エスカレーションと見なされる可能性があります。セキュリティ管理者がこのシチュエーションを計画および監査できるようにする

行単位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複製されます。これは、`FORCE ROW LEVEL SECURITY` が指定されている場合でも、`asNO FORCE ROW LEVEL SECURITY` に適用される変更と同等です。すべてのノードの行セキュリティポリシーが同一であるか、少なくとも互換性があることが推奨されますが、強制されません。

`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からテーブルを追加または削除できることに注意してください。 `bdr_superuser` は、個々のテーブルに対する特権を必要とせず、持つことも推奨されません。レプリケーションセット関数へのアクセスを制限する必要がある場合、これらの関数の制限バージョンを `SECURITY DEFINER` 関数として実装し、適切なユーザーに `GRANTed` することができます。

2.6.6 接続ロール

新しい BDR ノードを割り当てる場合、`bdr.create_node` の `local_dsn` 引数と `join_target_dsn` of `bdr.join_node_group` の DSN で指定されたユーザは、データベースオブジェクトの参照、作成、および管理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、権限エスカレーション攻撃を防ぐために慎重に記述されています。

攻撃対象をさらに減らすために、上記の DSN でより制限されたユーザを指定できます。少なくとも、このようなユーザは、次の規定が満たされるように、すべてのノードで権限を付与する必要があります。

- ユーザが `REPLICATION` 属性を持っている
- データベースに対する `CREATE` パーミッションが付与されます
- `bdr_superuser` ロールを継承します
- 直接または経由して、複製するすべてのデータベースオブジェクトを所有している所有者のロールからの許可。

すべてのノードが結合されると、DML および DDL レプリケーションを引き続き許可するために、アクセス許可がさらに次のように削減される場合があります。

- ユーザには REPLICATION 属性があります。
- bdr_superuser ロールを継承します。

2.6.7 特権の制限

BDR は追加の制限を実施し、TRIGGER または REFERENCES 特権のみに依存する DDL の使用を効果的に防止します。次のサブセクションでこれらについて説明します。

GRANT ALL は引き続き TRIGGER と REFERENCES の両方の権限を付与するため、ALL ではなく、GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE などの権限を明示的に指定することをお勧めします。

外部キー特権

ALTER TABLE ... ADD FOREIGN KEY がサポートされるのは、ユーザが被参照先テーブルに対する SELECT 権限を持っている場合、または被参照テーブルで RLS 制限が有効になっていて、現在のユーザがバイパスできない場合のみです。

したがって、REFERENCES 権限は、BDR を使用した外部キーの作成を許可するには不十分です。REFERENCES 権限のみに依存することは、テーブルスキャンではなくトリガーを使用してバリデーションチェックを実行するため、通常は有用ではありません。

トリガー

PostgreSQL では、トリガーはテーブルの所有者と TRIGGER 権限を付与された人の両方が作成できます。テーブル以外の所有者から付与されたトリガーは、セキュリティの問題を引き起こす可能性のある BDR のパーミッション所有者として実行され PostgreSQL。

BDR は、誰がテーブルにトリガーを作成できるかについてより厳しい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
- 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。(トリガーによって使用されるファンクションに対する 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、次の場合にのみトリガーを作成できます。それらは、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ますテーブルで、標準の PostgreSQL ルールを満たします(ここでも EXECUTE が必要です) ファンクションに対する権限)。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を持ち、所有者は、テーブルに対する TRIGGER 権限と EXECUTE の両方をユーザに与えることにしましたファンクションに対する権限、そのユーザが作成することは問題ないと想定されます。このファンクションを使用したそのテーブルのトリガー。

- テーブルに対する TRIGGER 権限を持つユーザーは、次を使用してトリガーを作成できます。EXECUTE がある場合、SECURITY DEFINER 句で定義されている関数それらの権限。この句により、ファンクションは常にコンテキストで実行されます標準の PostgreSQL と BDR の両方におけるファンクション自分自身の所有者の。

上記のロジックは、PostgreSQL では、トリガーの所有者はそれを作成したユーザではなく、そのトリガーが使用するファンクションの所有者であるという事実に基づいています。

同じルールが既存のテーブルに適用され、既存のテーブルにテーブルの所有者が所有していないトリガーがあり、SECURITY DEFINER 関数を使用しない場合、それをレプリケーションセットに追加することはできません。

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、`edbdr.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させることができます。

BDR レプリケーションの適用では、システムレベルのデフォルトの `search_path` のみが使用されます。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、適用時に実行すると失敗する他の `search_path` 設定を想定する場合があります。これが発生し保証にするには、`defaultsearch_path` のみ（常にスキーマ.objectname などのオブジェクトへの完全修飾参照を使用）を使用してオブジェクト参照を明確に解決するか、`ALTER FUNCTION ... SET search_path = ...` を使用してファンクションの検索パスを設定します影響を受ける機能用。

2.6.8 BDR デフォルト/事前定義ロール

BDR 拡張機能のインストール時に、BDR 定義済みロールが作成されます BDR 拡張機能がデータベースから削除された後も、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、同じ PostgreSQL インスタンス上の複数のデータベースで BDR を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ないため、クラスターの各ノードでこれを実行必要があることに注意してください。

bdr_superuser

- スキーマ BDR のすべてのテーブルのすべての特権
- スキーマ BDR のすべてのルーチンのすべての特権

bdr_read_all_stats

SELECT 権限

- `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`

EXECUTE 権限

- `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` からのすべての特権、プラス

EXECUTE 権限

- `bdr.monitor_group_versions`
- `bdr.monitor_group_raft`
- `bdr.monitor_local_replslots`

bdr_application

EXECUTE 権限

- `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`

上記の関数の多くには、使用する前に追加の特権が必要であることを注意してください。例、`bdr.alter_sequence_set_kind` を正常に実行には、テーブルの所有者である必要がありファンクション。

bdr_read_all_conflicts

BDR は、競合を `bdr.conflict_history` テーブルに記録します。競合はテーブル所有者のみに表示されるため（のみ）、競合履歴を読み取るために追加の特権は必要ありません。*** all テーブルの競合を見ることが出来るユーザがいると便利な場合は、オプションでそのユーザにロール `bdr_read_all_conflicts` *を付与できます。**

2.6.9 検証

BDR は、次のツールとアプローチを使用して検証されています。

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ています。

- MISRA 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

CIS PostgreSQL Benchmark v1、2019 年 12 月 19 日は、BDR スタックの検証に使用されています。TPAexec でオプションとして使用可能な `cis_policy.yml` 構成を使用すると、スコア付きテストで次の結果が得られます。

表 3: :header:, 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	ファイル権限マスクが正しいことを確認します
2.2	PASS	PostgreSQL グループメンバ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	ログ収集機構が有効になっていることを確認します
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	ログファイルの権限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有効期間が正しく設定されていることを確認します
3.1.9	PASS	ログファイルの最大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 メッセージのプログラム名前が正しいことを確認してください
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_rewritten' 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
3.1.18	PASS	'log_connections' 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 3 – 前のページからの続き

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	過度の管理者権限が取り消されていることを確認します
4.3	PASS	過剰なファンクション権限が取り消されていることを確認します
4.4	PASS	テスト済み過剰な DML 特権が取り消されていることを確認します
5.2	Not Tested	'ホスト' 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 140-2OpenSSL 暗号化が使用されていることを確認します
6.8	PASS	SSL が有効になっていて正しく構成されていることを確認します
7.3	PASS	WAL アーカイビングが構成され機能していることを確認します
		NA

手動で監査した場合、テスト 5.2 は合格できますが、自動化可能な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 バリエーションの追加パッケージソフトが必要です。

2.7 Conflicts

BDR は、アクティブ/アクティブまたはマルチマスター DBMS です。非同期で使用する場合、マルチプル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、BDR によって発生したときに自動的に検出および解決できるイベントです。解決はアプリケーションの性質とデータの意味に依存するため、競合を解決する方法に関して、BDR がアプリケーションにレンジ選択肢を提供することが重要です。

デフォルトでは、競合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

カルタプルまたはリモートタプルを選択し、他のタプルを破棄します。例、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、新しいもの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順のようなセマンティクスが確立されます。

この章では、標準データ型と行レベルの競合について詳しく説明します。

この章で後述するように、競合のハンドリングは構成可能です。競合は、[Stream Triggers](#) の章で説明されている競合トリガーを使用して、テーブルごとに異なる方法で検出および処理できます。

列レベルの競合検出および解決は、[CLCD](#) の章で説明されている BDR で利用できます。

競合を回避したい場合は、これらの機能を BDR で使用できます。

- 競合のないデータ型 (CRDT) -[CRDT](#) の章で説明されています。
- イーガーレプリケーション-[Eager Replication](#) の章で説明されています。

デフォルトでは、すべての競合は `bdr.conflict_history` に記録されます。競合が発生する可能性がある場合、テーブル所有者はそれらをモニタし、分析して回避方法を確認するか、アプリケーションタスクとして定期的にそれらをハンドルする計画をスキャン必要がありツール。

一部のクラスタリングシステムは、データへの同時アクセスを防ぐために分散ロックメカニズムを使用します。これらは、サーバーが非常に近い場合に合理的に実行できますが、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。

分散ロックは基本的に悲観的なアプローチであり、BDR は楽観的なアプローチを支持します。可能な場合は競合を回避しますが、競合が発生した場合はそれらを発生させて解決します。

!!! Warning “Upgrade Notes” * すべての SQL 可視インターフェイスは `bdr` スキーマにあります。`bdr_conflicts` または `bdr_crdt` スキーマの以前に非推奨になったすべてのインターフェースは削除され、3.7+ノードまたは少なくとも1つの3.7+ノードを含むグループでは動作しません**。すべての BDR パージョンにすでに存在する `bdr` スキーマのものを使用してください。

2.7.1 競合の発生方法

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した場合に起こり得ない一連のイベントの結果として発生します。ノードはトランザクションのコミット後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が、同時に他の競合する作業を行った別のノードに適用した場合は無効になります。

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再生するため、適用されているトランザクションと受信ノードでコミットされたトランザクションとの間に競合がある場合、リプレイオペレーションは失敗する可能性があります。

すべてのトランザクションが単一ノードで実行される場合、ほとんどの競合が発生しない理由は、PostgreSQL がそれを防止するためのトランザクション間通信メカニズム-`UNIQUE` インデックス、`SEQUENCES`、行およびリレーションのロック、`SERIALIZABLE` 依存関係の追跡などであるためです。これらのメカニズムはすべて通信する方法です望ましくない同時実行性の問題を防ぐための進行中のトランザクション間。

BDR には分散トランザクションマネージャまたはロックマネージャがありません。これが、レイテンシとネットワークパーティションで適切に機能する理由のパートです。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場合、異なるノード上のトランザクションは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。これが、BDR がこれが重要な場合のレプリケーションを積極的に提供する理由です。

2.7.2 競合の種類

主キーまたは一意の競合

最も一般的な競合行の競合であり、2つの操作が単一のノードでは実行できない方法で同じキーを使用して arow に影響します。BDR はそれらのほとんどを検出でき、`update_if_newer` 競合リゾルバを適用します。

行の競合は次のとおりです。

- INSERT vs INSERT
- UPDATE vs UPDATE
- UPDATE vs DELETE
- INSERT vs UPDATE
- INSERT vs DELETE
- DELETE vs DELETE

ビュー `bdr.node_conflict_resolvers` は、既知のすべての競合タイプに対して現在競合解決が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つの異なるノード上の INSERTs が同じ PRIMARY KEY 値（または noPRIMARY KEY が存在する場合、単一の UNIQUE 制約に同じ値）を持つタプルを作成する場合に発生します。

BDR は、ユーザー定義の競合ハンドラによってオーバーライドされない限り、発信元ホストのタイムスタンプに従って、最近挿入された2つのタプルを保持することでこれを処理します。

この競合により `insert_exists` 競合タイプが生成されます。これは、デフォルトで新しい（コミット時間に基づいて）行を選択し、その行のみ（`update_if_newer` リゾルバ）を保持することで解決されます。他のリゾルバーを構成できます-詳細については、[競合の解決](#)を参照してください。

この競合タイプを解決するには、列レベルの競合解決とユーザー定義の競合トリガーを使用することもできます。

このタイプの競合は、[Global Sequences](#) を使用することで効果的に解消できます。

複数の一意の **Constraints** に違反する挿入

INSERT / INSERT の競合は、複数の UNIQUE 制約（そのうちの 1 つは PRIMARY KEY かもしれません）に違反する可能性があります。新しい行が複数の UNIQUE 制約に違反しており、その結果、複数の別の行と競合する場合、レプリケーションの変更を適用すると `amultiple_unique_conflicts` の競合が発生します。

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オーダーには一部の行を削除する必要があります。`multiple_unique_conflicts` のリゾルバ設定に応じて、適用プロセスはエラーで終了するか、着信行をスキップするか、行の一部を自動的に削除します。自動削除では、常に正しい PRIMARY KEY で行を保持し、他の行を削除しようとします。

!!! Warning * マルチプルの行がこのように競合する場合、競合解決の結果が挿入オペレーションを続行することである場合、データの一部は常に削除されます！

競合トリガーを使用して異なる動作を定義することもできます。

更新/更新の競合

異なるノード上の 2 つの同時 UPDATE が同じタプルを変更する場合（ただし、PRIMARY KEY は変更しない）、リプレイ時に 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異なる競合の種類を生成できます。テーブルが [行バージョン競合検出] で構成されている場合、オリジナルの（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` 競合が生成されオリジン。行がチェックされます（オリジンは、現在の行が由来するノードです）。変更された場合、`update_origin_change` の競合が生成されます。他のすべての場合、UPDATE は通常、競合が発生することなく適用されます。

これらの競合は両方とも、前述のように `insert_exists` と同じ方法で解決されます。

プライマリキーの更新の競合

現在、BDR は PRIMARY KEY 操作が UPDATE オペレーションによって変更されている場合、競合解決を実行できません。primarykey を更新することはできますが、既存の値と競合しない保証にする必要があります。

プライマリキーの更新の競合は [Divergent Conflicts] であり、マニュアルの演算子介入が必要です。

PK の更新は PostgreSQL で可能ですが、PostgreSQL と BDR の両方に問題があります。

説明するために、非常にシンプルスキーマの例を作成しましょう。

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

主キー列の更新が可能のため、この SQL は成功します。

```
UPDATE pktest SET pk=2 WHERE pk=1;
```

…しかし、テーブルにマルチプルの行がある場合、例えば：

```
INSERT INTO pktest VALUES (3,3);
```

…その後、いくつかの更新が成功します：

```
UPDATE pktest SET pk=4 WHERE pk=3;
```

```
SELECT * FROM pktest;
```

```
pk | val
```

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

```
* 2 | 1
```

```
* 4 | 3
```

```
(2 rows)
```

…しかし、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
```

そのため、PK を更新する PostgreSQL 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非常に注意してください。

BDR では、マルチプルの場所から同時に UPDATE が許可されている場合、シチュエーションはより複雑になります。

これらの 2 つの変更を同時に実行すると機能します。

```
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)
```

…しかし、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、変更が他のノードに適用されると、update_missing の競合が発生します。

```
node1: UPDATE pktest SET pk=1 WHERE pk = 3;
```

```
node2: UPDATE pktest SET pk=2 WHERE pk = 3;
```

…各ノードで異なるデータを残す：

```
node1:
```

```
SELECT * FROM pktest;
```

```
pk | val
```

(次のページに続く)

(前のページからの続き)

```

-----+-----
* 1 | 1
* 5 | 3
(2 rows)

node2:
SELECT * FROM pktest;
pk | val
-----+-----
* 2 | 1
* 5 | 3
(2 rows)

```

このシチュエーションは、LiveCompare を使用して特定および解決できます。

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ことは簡単には解決できません。

```

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、ノード間に相違が生じます。ただし、ターゲットへの適用は、重複キー値違反 ERROR で両方のノードで失敗します。これにより、レプリケーションが停止し、現在はマニュアルで解決する必要があります。

`conflict_typeupdate_pkey_exists` を `skip`、`update`、または `update_if_newer` に設定すると、この重複キー違反エラーを回避できるようになり、レプリケーションが中断しなくなりました。これは、更新の性質によっては依然として相違を引き起こす可能性があります。

`update_pkey_exists` を `update_if_newer` に設定することにより、`sameold` キーが同じ新しいキーによって同時に更新されている上記のようなケースでの相違を回避できます。ただし、特定の状況では、`update_if_newer` でも、つまり、2 つの異なる行が両方とも同じ新しいプライマリキーに同時に更新される場合でも、相違が発生します。

その結果、特に BDR を使用して、アプリケーションで PK UPDATE を許可しないことを強くお勧めします。主キーを変更するアプリケーションの部分がある場合、同時変更を回避するには、Eager レプリケーションを使用し make それらの変更を行います。

!!! Warning * `update_pkey_exists` の競合の解決により更新が発生した場合、行の 1 つが常に削除されます！

複数の一意の Constraints に違反する更新

[複数の UNIQUE Constraints に違反する INSERT] のように、incomingUPDATE が複数の UNIQUE インデックス（および/または PRIMARY KEY）に違反する場合、BDR は `multiple_unique_conflicts` の競合を発生させます。

BDR は、遅延一意コンストレインをサポートしています。トランザクションがソースでコミットできる場合、競合が発生しない限り、ターゲットにクリーンに適用されますが、遅延プライマリキーはコンストレインワーニングとして使用できません。

更新/削除の競合

1 つのノードが、別のノードが同時に `b_tran_4` した行を UPDATE することができます。この場合、再生時に UPDATE / リプレイの競合が発生する可能性があります。

DELETED 行がまだ検出可能な場合（削除され行が VACUUM によって削除されていない場合）、`update_recently_deleted` の競合が生成されます。デフォルトでは、UPDATE は単にスキップされたされますが、これに対する解決策は構成できます。詳細については [競合解決] を参照してください。

削除され行は、ローカルノードがレプリケーションで遅れている場合に UPDATE を受信するまでに、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 conflicts と [INSERT / UPDATE Conflicts] を区別できず、単に、`update_missing` の競合を生成します。

競合する別のタイプの DELETE と UPDATE は、行がローカルで UPDATED になった後に実行される DELETE 操作です。このシチュエーションでは、結果は使用される競合検出のタイプに依存します。デフォルトの [Origin Conflict Detection] を使用すると、競合はまったく検出されず、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、これは競合トリガーを介して構成または処理できます。

INSERT / UPDATE の競合

オペレーションのデフォルトの非同期モードを使用する場合、ノードはオリジナルの INSERT が受信される前に行の anUPDATE を受信する場合があります。これは、3 つ以上のノードがアクティブな場合にのみ発生します（以下の 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` の競合が生成されます。デフォルトの競合リゾルバは `insert_or_skip` ですが、代わりに `insert_or_error` または `skip` を使用できます。insert-or-action を実行するリゾルバーは、可能であれば（行全体が受信されたとき）UPDATE からのデータに基づいて新しい行を INSERT に最初に試行します。行の再構築を可能にするには、テーブルに `haveREPLICA IDENTITY FULL` がニーズであるか、行に TOAST されたデータが含まれていない必要があります。

TOAST されたデータの詳細については、*TOAST サポートの詳細* を参照してください。

挿入/削除の競合

INSERT / UPDATE の競合と同様に、ノードは、INSERT をまだ受け取っていない行で aDELETE オペレーションを受け取ることもあります。これも3つ以上のノードがセットアップされている場合にのみ可能です（以下の3つ以上のノードとの競合を参照）。

現在、BDR は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 オペレーションは常に delete_missing の競合を生成しますが、これはデフォルトでオペレーションをスキップすることで解決されます。

削除/削除の競合

2つの異なるノードが同じタプ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。これは、デフォルトではオペレーションをスキップすることで解決されます。

両方の DELETE が同じ効果を持つため、この競合は無害です。したがって、どちらか一方を安全に無視できます。

3つ以上のノードとの競合

1つのノード INSERTs が行を2番目のノードと UPDATEdthere で再生される場合、3番目のノードは2番目のノードから UPDATE を受信してから、1番目のノードから INSERT を受信できます。これは INSERT / UPDATE の競合です。

これらの競合は、UPDATE を破棄することで処理されます。これにより、異なるノードで異なるデータが発生する可能性があります。つまり、これらは [Divergent Conflicts] です。

この競合タイプは3人以上のマスターでのみ発生する可能性があり、そのうち少なくとも2人がアクティブに書き込みを行う必要があることに注意してください。

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、次のシーケンス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12から INSERT を受信します。ノード2は UPDATE3 を実行します。ノード3はノード24から UPDATE を受け取ります。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 競合タイプの場合は insert_or_skip 競合リゾルバー）を使用することは、競合に対する実行可能な緩和戦略です。ただし、このオプションを有効にすると、INSERT / DELETE の競合が発生することに注意してください。以下を参照してください。

1. ノード1が UPDATE2 を実行します。ノード2は DELETE3 を実行します。ノード3はノード24から DELETE を受け取ります。ノード3はノード1から UPDATE を受け取り、INSERT に変換します

これらが問題である場合、update_recently_deleted として正しく検出されるように、tableor データベースの凍結設定を調整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_missingconflict 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 make に、各 INSERT を最近削除されたチェックに入れることを選択できます。ただし、これを行うと、大部分のユーザーにペナルティが科せられるため、現時点では delete_missing をログに記録するだけです。

以降のリリースでは、ログの競合が発生した場合、LiveCompare を使用した再チェックにより、INSERT / DELETE の異常が自動的に解決されます。後で参照してください。

これらの競合は、主に2つの問題のユースケースで発生する可能性があります。

- INSERT、それに続く DELETE-キューイングアプリケーションで使用できるように
- テーブルの PK 識別子が再利用される場合

これらのケースはいずれも一般的ではないため、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、レプリケーションが正しく機能する make に識別子の BDR 性に依存しているため、後者の場合に問題があります。

同じ一意の識別子を挿入、削除し、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 問題として知られています。BDR には、行が現在の行、最後の行、またははるかに古い行であるかどうかを知る方法がありません。< https://en.wikipedia.org/wiki/ABA_problem >

一意の識別子の再利用もまた、ビジネス上の問題です。これは、時間の経過とともに一意の識別ができないため、監査、トレーサビリティ、および適切なデータ品質が妨げられるためです。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のにかかる時間インターバル内に発生する識別子の再利用は、困難を引き起こします。通常のオペレーションではその時間は短いかもしれませんが、ダウンノードはその間隔を数時間または数日に延長する場合があります。

アプリケーションは一意の識別子を再利用しないことをお勧めしますが、その場合は、1 年より小さいの期間内に再利用を避けるための措置を講じます。

シーケンスまたは UUID を使用するアプリケーションは、この問題の影響を受けません。

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との競合は、FOREIGN KEY コンストレイン (FK) でも発生する可能性があります。

BDR は session_replication_role = 'replica' を使用して変更を適用するため、変更を適用するときに外部キーは再チェックされません。アクティブ/アクティブ環境では、被参照テーブルへの挿入と同時に参照テーブルに削除が発生すると、FK 違反になります。これは、INSERT / DELETE の競合に似ています。

最初に問題を説明してから、解決ソリューションを提供します。

シングルマスター PostgreSQL では、参照される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 が完了するまで待機する必要があります。DELETE が被参照した値を削除すると、INSERT / UPDATE は FK チェックに失敗します。

マルチマスター BDR には、ノード間行レベルロックはありません。そのため、参照元テーブルでの INSERT は、被参照テーブルでの DELETE の後ろで待機しないため、両方のアクションが同時に発生する可能性があります。したがって、参照元テーブルの 1 つのノードの INSERT / UPDATE は、別のノードの被参照テーブルの DELETE と同時に値を利用できます。これにより、参照テーブルの値が参照テーブルに存在しなくなります。

実際には、これは、参照被参照で DELETE が発生した場合にのみ発生します。これは一般的なオペレーションではありません。

親子リレーション（Orders-> OrderItems など）では、これを行うのは一般的ではありません。OrderItem を完全に削除するよりも、OrderItem を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。リファレンス/ルックアップデータの場合、新しいファクトデータに同じ値を使用すると同時にエントリを完全に削除するのは奇妙です。

FK がぶら下がる可能性はありシチュエーションが、このリスクは一般的に非常に低いため、BDR はこのケースをカバーする一般的なソリューションを課しません。

最初の解決策は、FK の使用を、通常一度に 1 つのノードのみから変更されるか、まれにしか変更されない、または変更の並行性がアプリケーションを介す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、このケース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 に設定することで FK の異常を回避します。テーブルごとに FK 違反を防ぎます。

例として、Fact と RefData の 2 つのテーブルがあります。ファクトには RefData を参照する FK があります。ファクトは参照テーブルであり、RefData は被参照テーブルです。各テーブルに 1 つのトリガーを追加するニーズがあります。

RefData の被参照された row がすでに削除されている場合、Fact で列を 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 で列を 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
```

(次のページに続く)

(前のページからの続き)

```

*   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、外部キーがぶら下がるのを防ぎます。

競合を打ち切る

TRUNCATE は、すべての行の DELETE と同様に動作しますが、行ごとの削除ではなく、テーブルデータの物理的削除によってこのアクションを実行します。結果として、行レベルの競合ハンドリングは利用できないため、TRUNCATE コマンドは、明確な競合がある場合でも、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、リプレイの順序付けによって発散的な変更が発生する可能性があります。

ユーザーは、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ようにします
 - LiveCompare を使用して、このような矛盾を強調します。
- TRUNCATE を WHERE 句のない DELETE ステートメントに置き換え、これはパフォーマンスが非常に低い可能性が高いことに注意してくださいより大きなテーブル。
- `bdr.truncate_locking = 'on'` を設定して TRUNCATE コマンドを設定しますロック動作。TRUNCATE が `bdr.ddl_locking` に従うかどうかを決定します設定。これすべてのノードが必要なため、TRUNCATE のデフォルトの動作ではありません起きているので、すべての場合において可能または望ましいとは限りません。

除外制約の競合

BDR は除外コンストレインをサポートせず、その作成を妨げます。

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、すべての除外コンストレインを手動で削除する必要があります。

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ているため、制約に違反する行が存在しないことを保証ことはできません。除外コンストレインは、排他制約違反のためにノードから他のノードにリプレイが進まないリプレイデッドロックを引き起こします。

BDR で強制的に排他制約を作成する場合、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は、レプリケーションが中断することを予告する必要があります。再度進行させるには、リモートトランザクションを適用できるように、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

また、ノードのロールなどのグローバル (PostgreSQL システム全体) データが異なる場合にも競合が発生する可能性があります。これにより、操作 (主に DDL) が発生し、1 つのノードで正常に実行およびコミットできますが、他のノードに適用できません。

例、**node1** には **fred** 名前付けのユーザがありますが、そのユーザは **node2** で作成されていません。**node1** の **fred** がテーブルを作成する場合、所有者が **fred** に設定されて複製されます。DDL コマンドが **node2** に適用されると、**fred** 名前付けのユーザがいないため、DDL は失敗します。この失敗により、PostgreSQL ログに **ERROR** 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 **fred** を作成することにより、この競合を解決するには管理者の介入が必要です。今後これを解決するには、**bdr.role_replication = on** を設定することをお勧めします。

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によって、または相互に保持されているロックを待機する場合があります。

関連するロックには以下が含まれます。

- 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 ...**)
- ユーザセッションによる明示的な行レベルロック (**SELECT ... FOR UPDATE/FOR SHARE**)
- 行 **UPDATEs**、**INSERTs**、または **DELETEs** による暗黙的なロックローカルアクティビティから、または他のノードからのレプリケーションから

BDR ライタプロセスは、ユーザートランザクションでデッドロックすることもできます。この場合、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も互いにデッドロックする可能性があります。PostgreSQL のデッドロックディテクタは、問題のあるトランザクションの 1 つに介入して終了します。BDR ライタプロセスが終了した場合、単にリトライし、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、通常は管理者の操作は必要ありません。ライタプロセスがアイドル状態のユーザセッションのロックの背後で長時間停滞している場合、管理者はユーザセッションを終了してレプリケーションを再度フローさせることができますが、これは別のユーザセッションに影響する長いロックを保持しているユーザーと同じです。

PostgreSQL で **log_lock_waits** facility を使用すると、ロック関連のリプレイストールを識別できヘルプ。

発散する対立

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、分岐の競合が発生します。さまざまな競合が発生することはありますが、このような競合のすべてが執筆時点で確実に防止できるわけではありません。

すべてのノードが変更をリプレイする前に別のノードが同じ行のキーを変更すると、行の **PRIMARY KEY** を変更すると、競合が生じる可能性があります。プライマリ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てください。

行データに関連するさまざまな競合では、通常、管理者がノードの 1 つのデータを手動で調整して、他のノードと整合性を保つ必要があります。文書化されているとおりに BDR が使用され、安全でないとマークされた設定または機能が回避される限り、このような競合は発生しないはずです。

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 `make` と事態がさらに悪化する可能性があります、あらゆる種類の競合を解決するための一般的な指示を与えることはできません。

TOAST サポートの詳細

PostgreSQL は、**TOAST** と呼ばれる大きな列にライン外ストレージを使用します。

ロジカルデコーディング（BDR はその上に構築されます）およびロジカルレプリケーションでの TOAST 値のハンドリングは、テーブルのメイン行のパートとして格納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）にログオン。これは、特に、更新された列の値を変更しなかった `UPDATE` ステートメントがその列のない行を生成するため、`UPDATE` の競合をハンドリングするときに問題を引き起こす可能性があります。[`INSERT / UPDATE Conflicts`] で述べたように、`update_missingconflict` 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 はエラーを生成します。

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、上記よりも微妙な問題があります（積極的なトランザクションは影響を受けません）。例、A、B、C という 3 つのノードを持つ BDR クラスターで次のワークロードを想像してください。

1. ノード A：txn A1 は `UPDATE SET col1 = ' トースト data ... '` を実行し、first2 をコミットします。ノード B：txn B1 は `UPDATE SET other_column = ' anything else' ;` そして A13 以降にコミットします。ノード C：ノード A への接続が遅れる 4。ノード C：txn B1 が最初に適用され、col1 の TOAST された列が欠落しますが、競合せずに適用されます 5。ノード C：txn A1 は（`update_origin_change` で）競合し、スキップされます 6。ノード C は A1 からのトーストされたデータを永遠に失います

BDR が最近複製された行にローカルの `UPDATE` を検出すると、BDR が TOAST 列の独自のロギングを追加するため、BDR を使用する場合（ビルトインロジカルレプリケーションまたはマルチマスターのプレーン `pglogical` を使用する場合）、通常は上記の問題はありません TOAST 列の変更。localUPDATE は TOAST を変更していません。したがって、BDR は、マルチプルのノードで更新が発生した場合（つまり、タプルのオリジンが変更された場合）に WAL ロギングが増加する代わりに、TOAST されたデータの不整合を防ぎます。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になります BDR AlwaysOn アーキテクチャでは通常そうです。

!!! Note * メインテーブルでも同じことを行わずに、TOAST テーブルでのみ `VACUUM FULL` または `CLUSTER` を実行すると、余分な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。存在しません。

!!! Warning * TOAST の追加の WAL ロギングは、`BEFORE UPDATE` トリガーを使用して行われます。このトリガーは、テーブルのすべての `BEFORE UPDATE` トリガーの中で、トリガー名前に基づいてアルファベット順に最後にソートする必要があります。これを簡単に `makemake` に `zzzz_bdr_` の接頭辞が付いていますが、その後にソートされる名前のトリガーを作成しないでください。そうしないと、並行性の問題に対する保護が存在しません。EDB Postgres Extended で BDR を使用する場合、このトリガーは作成または使用されません。

ただし、`insert_or_error` の競合解決には、`REPLICA IDENTITY FULL` の使用が引き続き必要です。

TOASTed 列に関連するこれらの問題は、行全体がキーのパートであると見なされるため、この設定は常にキーの一部として TOAST された値をログするため、`edREPLICA IDENTITY FULL` のあるテーブルに影響しません。BDR は、新しい行を再構築するのに十分スマートであり、キー行の欠落データを埋めます。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ることに注意してください。

2.7.3 競合の回避または許容

ほとんどの場合、競合を回避するか、競合を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する可能性があるため、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みを行うか、特定の行から一度に特定のノードから特定の行にのみ書き込みを行うことです。

これは多くのアプリケーションで自然に起こります。例、多くの消費者アプリケーションでは、所有するユーザのみがデータを変更できます。たとえば、アカウントのデフォルトの請求先住所を変更すると、データの変更はめったに発生しません。

ノードがダウンする直前に変更を `make` と、変更が失われたように見える場合があります。その後、同じ変更を再度行って、異なるノードを介して2つの更新を `make` ことができます。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、[Global Sequences](#) を使用すると、このタイプの競合を完全に防ぐことができます。

ルーム予約アプリケーションなど、オブジェクト間に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。最も簡単な解決策は、保証を使用して、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応じて、より複雑な方法が可能になる場合があります。たとえば、各ノードに 100 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。

特定のタイプの更新が1つの特定のノードからのみ発生するようにする別の手法は、異なるノードを介して異なるタイプのトランザクションをルーティングすることです。例：

- 1つのノードで小包を受け取りますが、別のノードで小包を配達します。
- 注文が1つのノードに入力されるサービスアプリケーション、作業は2番目のノードで準備してから、別のノードで顧客に提供しました。

最善のアクションコースは、頻繁に競合を発生させ、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。

2.7.4 競合検出

BDR は、競合検出のために次のメカニズムを提供します。

- [Origin Conflict Detection] (デフォルト)
- 行バージョンの競合検出
- Column-Level Conflict Detection

オリジン競合検出

(以前は Timestamp Conflict Detection として知られていましたが、これは混乱を招きました。)

オリジン競合検出は、トランザクションの発信元のホストで記録されたコミットタイムスタンプを使用し、依存します。これには、正しく動作するためにクロックが同期していること、または2つのノード間の最速のメッセージの許容範囲内にあることが必要です。そうでない場合、競合解決は、さらに先のノードを優先する傾向があります。ノード間のクロックスキューは、パラメーター `bdr.maximum_clock_skew` および `bdr.maximum_clock_skew_action` を使用して管理できます。

行の起点は、`track_commit_timestamps = on` の場合のみ利用可能です。

競合は、複製元が変更されたかどうかに基づいて最初に検出されるため、競合トリガーは、実際の競合ではないことが判明する可能性のある状況と呼ばれます。したがって、このメカニズムは、偽陽性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。

オリジン情報は、行が凍結されるまでしか使用できません。凍結された行に到着する更新は競合を引き起こさないため、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通常のケースです。そのため、競合を発生させると、多くの誤検知ケースが発生します。

しばらくオフラインになってい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した凍結行よりも実際に古い場合、これにより発散エラーが発生する可能性があります。挿入と削除は、このシチュエーションの影響を受けません。

ユーザーは、[Node Restart and Down Node Recovery](#) で説明されているように、拡張の停止のためにノードを放置しないことをお勧めします。

EDB Postgres Extended では、ノードがダウンしている間、パラメータ設定を変更することなくこのシチュエーションを適切にハンドルするために、BDR は行の凍結を自動的に抑制します。

Postgres の他の亜種では、ユーザーはこのシチュエーションを慎重に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` よりも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な高い値を設定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する必要があります。競合データがデータベースサーバから削除されるまでの猶予期間はダウンタイムです。例、ディスク上が5億に設定されている場合、競合データが削除される前に、1000 TPS を実行しているノードがわずか5.5日ギガバイトする可能性があります。fromlower 設定。

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
```

以下に注意してください。

- `autovacuum_freeze_max_age` はサーバーのスタートにのみ設定できます。
- `vacuum_freeze_min_age` はユーザー設定可能ですので、値が低いと行が早くフリーズされ、競合が発生する可能性があります無視されます。`autovacuum_freeze_min_age` およびトースト`.autovacuum_freeze_min_age` 個々のテーブルに設定することもできます。
- `CLUSTER` または `VACUUM FREEZE` コマンドも実行します行を早期にフリーズすると、競合が無視される可能性があります。

行バージョンの競合検出

または、BDR には、ノードのシステムクロックに関係なく、行のバージョン管理と競合検出を使用するオプションがあります。

行バージョンの競合検出では、3つの項目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、[Origin Conflict Detection] が使用されます。

1. BDR ノードグループに対して `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追跡を有効にする必要があります。この関数は、新しい列（ユーザー定義名前）と新しい列の値を管理する `UPDATE` トリガーを追加します。列は `INTEGER` タイプとして作成されます。

カウンタは `UPDATE` でのみ増加しますが、この手法により、`UPDATE` と `DELETE` の両方の競合検出が可能になります。

このアプローチは、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。

!!! Note * 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、[競合解決] 構成に基づいて処理されます。行バージョンの生成方法は、競合の検出にのみ有用であり、どのバージョンの行がより新しいかに関する信頼できる情報として信頼すべきではありません。

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決定するには、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。

あらすじ

```
bdr.alter_table_conflict_detection(relation regclass,
*                               method text,
*                               column_name name DEFAULT NULL)
```

パラメーター

- **relation**-新しい競合検出メソッドを設定するリレーシヨンの名前。
- **method**-使用する競合検出メソッド。
- **column_name**-列検出データの保存に使用する列。これはスキップされたできます。この場合、列名前は自動的に選択されます競合検出メソッド。 **row_origin** メソッドは必要ありませんメタデータストレージ用の追加の列。

競合検出のために認識されている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オリジン（参照 [元の競合検出] 上記）。これは、サポートされている唯一のメソッドですテーブルに追加の列は必要ありません。
- **row_version**-行バージョン列（**行バージョンの競合検出**を参照）上記）。
- **column_commit_timestamp**-列ごとのコミットタイムスタンプ（章）。
- **column_modify_timestamp**-列ごとの変更タイムスタンプ（章）。

注

`column_commit_timestamp` と `column_modify_timestamp` の競合検出方法の違いの詳細については、CLCD の章の [Current vs Commit Timestamp](#) セクションを参照してください。

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが `ddl filters` 構成の影響を受けることを意味します。

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んで DML グローバルロックを取得します。

このファンクションはトランザクションです-トランザクションの ROLLBACK で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に可視されます。

`bdr.backwards_compatibility` が 30618 以下に設定されていない限り、`bdr.alter_table_conflict_detection` ファンクションは `relation` の所有者のみが実行できます。

!!! Warning * メタデータを保存するために余分な列を使用するものから競合検出メソッドを変更すると、その列は削除されることにノートしてください。

!!! Warning * このファンクションは、CAMO を自動的に無効にします（これらが `bdr.camo_enable_client_warnings` で無効にされない限り、ワーニングとともに）。

競合タイプのリスト

BDR は、`conflict_type` パラメータとして使用できる以下の競合タイプを認識します。

- `insert_exists`-を介して着信挿入が既存の行と競合するプライマリキーまたは一意のキー/インデックス。
- `update_differing`-着信更新のキー行がローカルと異なります行。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`-入ってくる更新があった行を変更しています最後に別のノードによって変更されました。
- `update_missing`-着信更新が行を変更しようとしています 存在します。
- `update_recently_deleted`-着信更新が行を変更しようとしています最近削除されました。
- `update_pkey_exists`-着信更新により PRIMARY KEY が変更されました変更を適用しているノードに既に存在する値。
- `multiple_unique_conflicts`-着信行がマルチプルと競合しますターゲット表の一意のコンストレーン/インデックス。
- `delete_recently_updated`-コミットのタイムスタンプが古い着信削除現在のノードの行の最新の更新より、または**行バージョンの競合検出**を使用します。
- `delete_missing`-着信削除は、そうでない行を削除しようとしています存在します。
- `target_column_missing`-ターゲット表に 1 つ以上の列がありません入ってくる行に存在する。
- `source_column_missing`-着信行に 1 つ以上の列がありませんターゲット表に存在するもの。
- `target_table_missing`-ターゲットテーブルがありません。
- `apply_error_ddl`-適用時に PostgreSQL によりエラーがスローされました複製された DDL コマンド。

2.7.5 競合の解決

ほとんどの競合は自動的に解決できます。BDR のデフォルトは `alast-update-wins` メカニズム、より正確には `update_if_newerconflict` リゾルバです。このメカニズムは、競合検出に使用される同じコミットタイムスタンプに基づいて、2つの競合する行の最後に挿入または変更された行を保持します。特定のコーナーケースシナリオでの動作は、`[bdr.create_node_group]` および `[bdr.alter_node_group]` で使用される設定によって異なります。

BDR を使用すると、ユーザは次のファンクションを使用して競合解決のデフォルトの動作をオーバーライドできます。

`bdr.alter_node_set_conflict_resolver`

このファンクションは、特定のノードでの競合解決の動作を設定します。

あらすじ

```
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`-設定が適用される競合タイプ（[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（[競合解消者のリスト](#)を参照）

注

現在、変更できるのはローカルノードのみです。ファンクション呼び出しは複製されません。マルチプルのノードの設定を変更する場合は、各ノードで機能を実行する必要があります。

このファンクションによる構成の変更は、`[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することに注意してください。

このファンクションはトランザクションです-行われた変更はロールバックされ、現在のトランザクションから可視になります。

競合解消者のリスト

BDR には、ハンドル可能な競合タイプのカバレッジが異なる複数の競合リゾルバーがあります。

- **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、レプリケーションを続行します次の変更で。insert_exists、update_differing、update_origin_change、update_missing、update_recently_deleted、update_pkey_exists、delete_recently_updated、delete_missing、target_table_missing、target_column_missing および source_column_missing の競合タイプ。
- **skip_if_recently_dropped**-テーブルの場合、リモート変更をスキップしますダウストリームに存在しない（最近（現在は 1 日）ダウストリームでドロップ; それ以外の場合はエラーをスローします。に使用できます target_table_missing 競合タイプ。skip_if_recently_dropped の競合リゾルバは、同じ名前のテーブルがすぐに再作成される場合に課題を引き起こす可能性があります落とした後。その場合、ノードの 1 つで DML が表示される場合があります DDL がテーブルを再作成する前に、テーブルを再作成しました。それからテーブルが最近削除されたと仮定して、リモートデータを誤ってスキップしますそして、データロスを引き起こします。したがって、オブジェクト名を再利用しないことをお勧めしますこれらの衝突リゾルバと共に落とされた直後。
- **skip_transaction**-を生成したトランザクション全体をスキップします対立。 apply_error_ddl の競合に使用できます。
- **update_if_newer**-リモート行が後でコミットされた場合に更新する（as 発信元サーバーの壁時計によって決定されます）ローカル行。タイムスタンプが同じ場合、ノード ID はタイブレーカーとして使用されます保証のノードで同じ行が選択されるようにします（より高い nodeid が優先されます）。insert_exists、update_differing、update_origin_change に使用できますおよび update_pkey_exists の競合タイプ。
- **update**-常に複製されたアクションを実行します。insert_exists に使用できます（INSERT を UPDATE に変換します）。update_differing、update_origin_change、update_pkey_exists、および delete_recently_updated（削除を実行）。
- **insert_or_skip**-によって送信された利用可能な情報から新しい行を構築しようオリジンと INSERT; 構築するのに十分な情報がない場合行全体、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**-から送信された利用可能な情報から新しい行を構築しようオリジンとそれを挿入; 構築するのに十分な情報がない場合行全体、エラーをスローし、レプリケーションを停止します。update_missing および update_recently_deleted 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**-余分な列がリモート行には NULL 値が含まれます。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**-欠落している列の値をデフォルトで埋めます（含むそれが列のデフォルトである場合は NULL）、処理を続行します。いずれかのエラーデフォルトまたはコンストレイン違反を処理します（つまり、NULL デフォルトで NOT NULL 列）はレプリケーションを停止します。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、および delete_missing の競合タイプは、[Conflict Triggers](#) を使用するユーザー定義ロジックでも解決できます。

これは、コンフリクトリゾルバがハンドルできるコンフリクトタイプを個別化するのにヘルプマトリクスです。

表 4: :header:, insert_exists, update_differing, update_origin_change, update_missing, update_recently_deleted, update_pkey_exists, delete_recently_updated, delete_missing, target_column_missing, source_column_missing, target_table_missing, multiple_unique_conflicts

error	X	X	X	X	X	X	X	X	X	X	X	X
skip	X	X	X	X	X	X	X	X	X	X	X	X
skip_if_recently_dropped											X	
update_if_newer	X	X	X			X						
update	X	X	X			X	X					X
insert_or_skip				X	X							
insert_or_error				X	X							
ignore									X			
ignore_if_null									X			
use_default_value										X		
conflict_trigger	X	X	X	X	X	X	X	X				X

デフォルトの競合解決

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

次のページに続く

表 5 – 前のページからの続き

Conflict Type	Resolver
multi- ple_unique_conflicts	error
delete_recently_updated	skip
delete_missing	skip
target_column_missing	ignore_if_null
source_column_missing	use_default_value
target_table_missing	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行われた特定のアクションに対応します。

現在、`conflict_resolution` パラメータでは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`-行の処理はスキップされたされました（変更は行われていません）ローカル
- `merge`-新しい行が作成され、リモート行とローカル行の情報がマージされました
- `user`-ユーザコード（競合トリガー）が書き込まれた行を生成しましたターゲット表へ

2.7.6 競合ログ

マルチマスターの競合の診断とハンドリングを容易にするために、BDR はデフォルトですべての競合を PostgreSQL ログログファイルに記録します。この動作は、次の機能を使用してより細かく変更できます。

`bdr.alter_node_set_log_config`

ノードの競合ロギング構成を設定します。

あらすじ

```
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`-サーバーログログに記録するかどうか
- `log_to_table`-`bdr.conflict_history` テーブルにログを記録するかどうか
- `conflict_type`-ログに記録する競合タイプ。NULL（デフォルト）はすべてを意味します
- `conflict_resolution`-ログの競合解決。NULL（デフォルト）はすべてを意味します

注

現在、変更できるのはローカルノードのみです。ファンクション呼び出しは複製されません。マルチプルのノードの設定を変更する場合は、各ノードで機能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に対応しています。変更はロールバックでき、現在のトランザクションに可視されます。

競合ログ設定の一覧表示

ビュー `bdr.node_log_config` には、すべてのロギング構成が表示されます。ロギング構成の名前、ログを記録する場所、および記録する競合タイプと解決策がリストされます。

テーブルへの競合のロギング

`log_to_table` が `true` に設定されている場合、競合はテーブルに記録されます。競合ロギングのターゲットテーブルは `bdr.conflict_history` です。

このテーブルは、`local_time` 列でパーティション分割されたレンジです。テーブルは自動パーティションによって管理されます。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 か月間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する場合があります。

BDR によって管理されるすべてのテーブルに対して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保証ことが重要です。これを行うには、`bdr.conflict_history` テーブルで ROW LEVEL SECURITY ポリシーを定義します。テーブルの所有者のみが、それぞれのテーブルの競合を読み取ることができます。基になるテーブル自体に 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、基礎となるテーブルの一部またはすべての行は、所有者でさえも読み取りできない場合があります。したがって、競合ログテーブルに対してより厳しいポリシーを適用します。

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。

デフォルトロール `bdr_read_all_stats` は、ユーザデータを含まない `bdr.conflict_history_summary` と呼ばれるカタログビューにアクセスでき、ログに記録された競合を監視できます。

競合の報告

テーブルに記録された競合は、レポートに要約できます。これにより、アプリケーションの所有者は、競合を特定、理解、解決したり、競合を防ぐためにアプリケーションの変更を導入したりできます。

```
SELECT nspname, relname
, date_trunc('day', local_time) :: date AS date
, count(*)
FROM bdr.conflict_history
WHERE local_time > date_trunc('day', current_timestamp)
GROUP BY 1,2,3
ORDER BY 1,2;
* nspname | relname |      date      | count
-----+-----+-----+-----
my_app   | test   | 2019-04-05 |      1
(1 row)
```

2.7.7 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、それらが同一であることを確認するように設計されたユーティリティプログラムです。

LiveCompare は BDR スタックのパートとして含まれており、BDR ノードの任意のペアで有効にでき、デフォルトでは、複製されたすべてのテーブルを比較し、差異をレポートします。LiveCompare は、Postgres and オラクルなどの非 BDR データソースでも機能します。

LiveCompare は、受信行を継続的にモニタするためにも使用できます。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、都合の良い時間に実行できます。

LiveCompare は、マルチプルのテーブルの同時チェックを許可し、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チェックを許可するように設定できます。チェックは、最初に全行ハッシュを比較してから実行されます。有用なサイズのバッチの行。

差異が見つかった場合は、一定期間にわたって再チェックすることができ、結果整合性の遅延を許容します。

詳細については、LiveCompare のドキュメントを参照してください。

2.8 Sequences

多くのアプリケーションでは、一意の代理 ID をデータベースデータベースオブジェクトを使用してこれらを生成します。PostgreSQL では、これらは、`CREATE SEQUENCE` コマンドを使用して手動で作成され、`nextval()` ファンクション、`serial` および `bigserial` シーケンスまたは `alternative GENERATED BY DEFAULT AS IDENTITY` 列を呼び出して取得することができます。

ただし、PostgreSQL の標準シーケンスはマルチノード対応ではなく、ローカルノードで一意の値のみを生成します。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ロス（INSERTs の破棄による）を引き起こすため、これは重要です。

2.8.1 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（必ずしもすべてではない）に機能する非同期分散システムで一意の合成キーを生成する簡単な方法を提供します。

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` 制約を定義した場合、ノードが他のノードと同じ値を取得することはできません。BDR がノード間の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。そ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。

- タイムシャードシーケンス
- グローバルに割り当てられたレンジシーケンス

タイム制約シーケンスは、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するため、より高速で堅牢であり、作成されたタイムスタンプを記録する便利なプロパティがあります。64 ビットの `BIGINT` データ型で、19 桁のレンジを生成しローカルが、これは Javascript 整数型などの一部のホスト言語データ型では使用するには長すぎる場合があります。 `BIGINT` または `INTEGER` シーケンスに適したものにします。

グローバルシーケンスは、`bdr.alter_sequence_set_kind()` 関数を使用して作成できます。このファンクションは標準の PostgreSQL シーケンスを取り、BDR グローバルシーケンスとしてマークします。また、シーケンスを変換して標準の PostgreSQL シーケンスに戻すこともできシーケンス（以下を参照）。

BDR は構成変数 `bdr.default_sequence_kind` も提供します。これは、`CREATE SEQUENCE` コマンドが実行されるとき、または `serial`、`bigserial` or `GENERATED BY DEFAULT AS IDENTITY` 列が作成されるときに作成されるシーケンスの種類を決定します。有効な設定は次のとおりです。

- `local`（デフォルト）は、新しく作成されたことを意味しますシーケンスは、標準の PostgreSQL（ローカル）シーケンスです。
- 常にグローバルに割り当てられたレンジシーケンスを作成する `galloc`。
- `timeshard` は、`BIGINT` シーケンスのタイムシャーディングされたグローバルシーケンスを作成します。 `INTEGER` シーケンスで使用すると、エラーがスローされます。

`bdr.sequences` ビューには、個々のシーケンスの種類に関する情報が表示されます。

`currval()` および `lastval()` は、すべてのタイプのグローバルシーケンスに対して正しく機能します。

タイムシャードシーケンス

タイムハードシーケンスによって生成された ID は、標準の PostgreSQL シーケンスのように、データ挿入の大まかなオーダーを取得するために使用できるように、緩やかに時間順に並べられます。同じミリ秒以内に生成された値は、1 つのノード上であってもオーダーが狂う可能性があります。緩やかな時間順序のプロパティは、それらが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。

タイムシャードシーケンスは 1 つ以上のノードで機能し、ノード結合プロセスの完了後にノード間通信を必要としません。そのため、拡張ネットワークパーティションのリスクがあり、レプリケーションラグやノード間レイテンシの影響を受けない場合でも、引き続き使用できます。

Timeshard シーケンスは、標準シーケンスとは異なる方法で一意の ID を生成します。アルゴリズムは、シーケンス番号に 3 つのコンポーネントを使用します。シーケンスの最初のコンポーネントは、シーケンス番号生成時のタイムスタンプです。シーケンス番号の 2 番目のコンポーネントは、各 BDR ノードに割り当てられた一意の ID です。これにより、異なるノードからの ID は常に異なることが保証されます。最後に、3 番目のコンポーネントは、ローカルシーケンス自分自身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 を追加オーダーことで、競合がないことを保証できますが、順序付けの別の有用なプロパティを保持する必要もあります。タイムスタンプを最初に置くことでこれが保証されます。

タイムハードシーケンスには、いくつかの制限と注意事項が適用されます。

タイムシャードシーケンスは 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、`serialsequences` では使用できません。代わりにグローバルに割り当てられたレンジシーケンスを使用してください。

任意の与えられたシーケンスのために `anygiven` ノードでミリ秒あたりに生成された 8192 シーケンス値の制限があります。1 つのノード上の 1 つのシーケンスからミリ秒あたり 8192 を超えるシーケンスが生成される場合、生成された値はラップして衝突する可能性があります。パフォーマンス上の理由でチェックは行われません。各 ms のスタートに値は 0 にリセットされません。通常、衝突は `INSERT` または `UPDATE` で `aUNIQUE` 制約違反になります。異なるノードで生成されたシーケンス値はノード ID を含むため、決して衝突することはないため、複製の競合を引き起こすことはできません。

実際には、これは無害です。他の作業が行われたり、行が挿入されたり、インデックスが更新されたりするため、値はこの制約を引き起こすほど速く生成されません。

おそらくもっと重要なのは、タイムスタンプコンポーネントが 2050 年に値を使い果たし、`bigint` と組み合わせ使用した場合、値は 2033 年に負の数にラップれることを意味します。この日付を超えてアプリケーションをデプロイする予定がある場合は、下記の `UUID`、`KSUUID`、およびその他のアプローチのいずれかを試すか、代わりにグローバルに割り当てられたレンジシーケンスを使用してください。

タイムハードシーケンスの入力として使用されるシーケンスの `INCREMENT` オプションは、事実上無視されます。これは、多くのオブジェクトリレーショナルマッパー (ORM) ツール、特に Hibernate のように、シーケンスキャッシングを行うアプリケーションに関連する可能性があります。アプリケーションがキャッシュされた値で何でもできるようになるまでに。

同様に、START、MINVALUE、MAXVALUE、および CACHE の設定は、基になるシーケンスで変更される場合がありますが、そうすることには利点がありません。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、値のレンジの制限は関数の結果に影響しません。同じ理由で、`setval()` はタイムハードシーケンスには役立ちません。

グローバルに割り当てられたレンジのシーケンス

グローバルに割り当てられたレンジ（または `galloc`）シーケンスは、各ノードに値の範囲（チャンク）を割り当てます。ローカルレンジが使い果たされると、他のノード間のコンセンサスによって新しいレンジがグローバルに割り当てられます。これは、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レンジが使い果たされた場合、シーケンスジェネレータが進行するには、クラスター内のノードの大部分にローカルノードを接続する必要があります。

タイムハード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`、整数、`bigint`）をサポートします。つまり、`gallocsequences` は、JavaScript で整数を使用するなど、64 ビットシーケンスが問題のある環境で使用できることを意味しスペース。

各投票によって割り当てられるレンジは、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ています。

- `smallint`-1000 個の数字
- 整数-1 000 000 の数字
- `bigint`-1000 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、将来の使用のために予約したチャンクです。したがって、1 つのノードから生成される値は単調に増加します。ただし、グローバルに表示すると、生成された値はまったく順序付けられません。これにより、b-tree 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、生成された値はレンジパーティションキーとして有用ではないことを意味します。

`galloc` シーケンスの主な欠点は、割り当てられたレンジが使い果たされると、シーケンスジェネレータは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要求する必要があることです。BDR グループにアクセスできません。これは今後のリリースで回避される可能性があります。

CACHE、START、MINVALUE、および MAXVALUE オプションは `galloc` シーケンスで正しく機能しますが、シーケンスを `galloc` 種類に変換する前にそれらを設定する必要があります。INCREMENT BY オプションも正しく機能しますが、各シーケンスデータ型に割り当てられた上記の範囲以上のインクリメント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ないため、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 `bdr.sequence_alloc` の `galloc` シーケンスを追跡します。このカタログは、`galloc` シーケンスに現在割り当てられているチャンクを追跡するために必要です。シーケンス名前と名前空間はこのカタログに保存されます。シーケンス名前/名前空間への変更はレプリケーションストリームで管理されるのに対し、シーケンスチャンクアロケーションは Raft で管理されるため、BDR は現在、`gallocsequences` 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。ユーザは、アプリケーションスキーマの設計中にこの制限に留意する必要があります。

使用法

ローカルシーケンスを galloc に変換する前に、次の前提条件に注意する必要があります。

シーケンスの種類が galloc に変更されると、書き換えられ、ローカルシーケンスの定義されたスタート値からリスタートされます。これが稼働データベースの既存のシーケンスで発生する場合は、現在の値をクエリーし、スタート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 はユーザーがファンクション `bdr.alter_sequence_set_kind()` を使用して開始値を渡すことを許可します。すでにオフセットを使用していて、マルチプルのノードから書き込みがある場合は、最大使用値を確認し、少なくとも次までシーケンスをリスタートする必要があります値。

```
-- determine highest sequence value across all nodes
SELECT max((x->'response'->0->'nextval')::bigint)
  * FROM json_array_elements(
  *   bdr.run_on_all_nodes(
  *     E'SELECT nextval(\'public.sequence\');'
  *   )::jsonb AS x;

-- turn into a galloc sequence
SELECT bdr.alter_sequence_set_kind('public.sequence'::regclass, 'galloc',
  ↳$MAX+MARGIN);
```

ユーザーはシーケンスをロックできないため、最大 () 値のクエリ中に操作を続行できるようにするには、\$ MARGIN 値を残す必要があります。

この例では、333, からシーケンスを開始し、クラスターに2つのノードがあるため、アロケーション4がいくつかあることがわかります。nodeand とチャンクサイズごとに2は1000000で、整数シーケンスに関連しています。

```
SELECT * FROM bdr.sequence_alloc
  * WHERE seqid = 'public.categories_category_seq'::regclass;
  *   seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |
  seq_last_alloc
-----+-----+-----+-----+-----
↳-----
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 までのレンジを使用しています。

```
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
```

(次のページに続く)

(前のページからの続き)

```
(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、20000004 から 40000003 までのレンジを使用しています。

```
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)

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
```

注単一のクエリー（WHERE ctid IN ('(0,2)', '(0,3)' ）に結合することはできません。最初のレンジのみが表示されるためです。

ノードがチャンクを終了すると、コンセンサスに新しいものを要求し、最初に利用可能になります。この場合、4000334 から 5000333 になります。これが新しい予約チャンクになり、古い予約チャンクの消費をスタートします。

2.8.2 UUID、KSUUID、およびその他のアプローチ

BDR で使用できる globalsequences 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、およびその 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（マニュアル）
- 外部で調整された自然キー

BDR アプリケーションは他の方法を安全に使用できないことにノートしてください：BDR はノード間の行ロックを取得しないため、SELECT ... FOR UPDATE、UPDATE ... RETURNING ... などのシーケンス生成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 では正しく動作しません。同じ値が複数のノードで生成されます。同じ理由で、「ギャップレス」シーケンス生成の通常の戦略は 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、二相コミットを使用して外部ソースからギャップのないシーケンスの生成を調整するか、BDR グループの 1 つのノードでのみ生成する必要があります。

UUID と KSUUID

代わりに、UUID キーは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。これらは非常にランダム値または擬似ランダム値であり、同じ値を 2 回生成することはほぼ不可能です。UUID キーを使用する場合、ノードが継続的に通信する必要はありません。

信じられないほど衝突が発生したイベント、競合検出は、保持する 2 つの挿入されたレコードの新しい方を選択します。競合ロギングは、有効になっている場合、そのようなイベントをレコードしますが、約 2^{64} キーが生成された後にのみ衝突が実際に発生する可能性があるため、例外的に発生することはほとんどありません。

UUID キーの主なマイナス面は、それらがややスペースがあることです-ネットワークが非効率的であり、プライマリキーとしてだけでなく、外部キーで被参照される場合や、ワイヤで送信される場合により多くのスペースを消費します。

BDR は、KSUUID と呼ばれる UUID データの K-Sortable バリエーションを提供します。これは、PostgreSQL の標準の UUID データタイプを使用して保存できる値を生成します。KSUUID の値は、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 と似ています。違いは、KSUUID が K- タイムスタンプであるということです。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、検索の効率が向上し、結果データの自然な時間ソートが可能になるため、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の MAC が含まれないため、KSUUIDs を使用してもセキュリティ上の問題はありません。

KSUUID v2 がすべての場合に推奨されるようになりました。生成された値は、通常の比較演算子で直接ソートできます。

BDR には KSUUID の 2 つのバージョン、v1 および v2 があります。従来の KSUUID v1 は非推奨になりましたが、既存のインストールをサポートするオーダーに保持されており、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ないため、それらを操作する機能にも互換性がありません。KSUUID の v2 も UUID バージョン番号を格納しなくなりました。

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だけ増加し、異なるオフセットで開始されます。例、ステップ 1000 では、node1 のシーケンスは 1001、2001、3001 などを生成し、node2 は 1002 を生成します、2002、3002 など。このスキームは、ノードが拡張通信できない場合でもうまく機能しますが、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。ただし、間違いは簡単に重複するシーケンスにつながる可能性があります。

次のように、1 つのノードで目的のシーケンスを作成することにより、このアプローチを BDR で構成するのは比較的シンプルです。

```
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、各ノードで `setval()` を呼び出して、各ノードに異なるオフセット開始値を与えます。例：

```
-- On node 1
SELECT setval('some_seq', 1);

-- On node 2
SELECT setval('some_seq', 2);

* -- ... etc
```

将来的にノードを変更することは困難で混乱を招くため、追加したいすべてのノード用のスペースを確保するのに十分なラージさの `INCREMENT` を許可する必要があります。

`bigint` の値を使用する場合、10000 以上のオフセットを使用しても、キーの枯渇に関する実際的な懸念はありません。消耗に近づく可能性を得るには、数百台のマシンで毎秒数百万回の挿入を行う何百年も必要です。

BDR は現在、このようなステップ/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。通常、ノード番号は、各ノードで異なる番号を返すファンクションから取得されます。このようなファンクションは、一時的に DDL レプリケーションを無効にして定数 SQL ファンクションを作成するか、レプリケーションセットのパートではない 1 行のテーブルを使用して各ノードに異なる値を格納することで作成できます。

2.8.3 グローバルシーケンス管理インターフェイス

BDR は、標準の PostgreSQL シーケンスと BDR グローバルシーケンスを変換するためのインタフェースを提供します。

次の関数は 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が適用されることに注意してください。

bdr.alter_sequence_set_kind

シーケンスの所有者は、シーケンスの種類を設定できシーケンス。一度設定すると、`seqkind` は `bdr.sequences` ビューでのみ可視されます。他のすべての方法では、シーケンスは通常のシーケンスとして表示されます。

BDR はこのファンクションを DDL として扱うため、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（現在アクティブな場合）。[DDL Replication] を参照してください。

あらすじ

```
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`、BDR の場合は `timeshard` 「タイムアンドシャーディング」ベースのアルゴリズムを使用するグローバルシーケンス [BDR Global Sequences] セクション、またはグローバルに割り当てられたレンジの `galloc` ノード間のコンセンサスを使用して、一意の範囲を割り当てるシーケンス各ノードのシーケンス番号

注

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レンジは、シーケンスのスタート値を開始点として使用します。シーケンスが `galloc` に変更される前にすでに使用されている既存の値がある場合、次のコマンドを使用して、新しく生成された値が既存の値と競合しないように開始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レプリケーションメカニズムを使用します。つまり、レプリケーションは `ddl filters` 構成の影響を受けます。

このファンクションは、グローバル DDL ロックを取得します。また、シーケンスをローカルでロックします。

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、変更は現在のトランザクションに可視されます。

`bdr.backwards_compatibility isset` が 30618 以下に設定されていない限り、`bdr.alter_sequence_set_kind` ファンクションはシーケンスの所有者のみが実行できます。

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。結果値のタイプは「timestampz」です。

あらすじ

```
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

このファンクションは、ローカルノードでのみ実行されます。

bdr.extract_nodeid_from_timeshard

このファンクションは、timeshard シーケンスの nodeid コンポーネントを抽出します。

あらすじ

```
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

このファンクションは、ローカルノードでのみ実行されます。

bdr.extract_localseqid_from_timeshard

このファンクションは、timeshard シーケンスのローカルシーケンス値コンポーネントを抽出します。

あらすじ

```
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

このファンクションは、ローカルノードでのみ実行されます。

bdr.timestamp_to_timeshard

このファンクションは、タイムスタンプ値をダミーのタイムハードシーケンス値に変換します。

これは、インデックス検索または timeshard 列の値の比較や特定のタイムスタンプに役立ちます。

例、timeshardsequence を使用している列 id を持つテーブル foo を指定すると、次のように昨日の真夜中以降の変更の数を取得できます。

```
SELECT count(1) FROM foo WHERE id > bdr.timestamp_to_timeshard('yesterday')
```

この方法で定式化されたクエリーは、id 列のインデックススキャンを使用します。

あらすじ

```
bdr.timestamp_to_timeshard(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されるタイムスタンプ

注

このファンクションはローカルノードでのみ実行されます。

2.8.4 KSUUID v2Functions

KSUUID v2 データ、K-Sortable UUID データを操作するための Functions。

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。

結果値のタイプは「UUID」です。

あらすじ

```
bdr.gen_ksuuid_v2(timestamptz)
```

注

このファンクションは、ローカルノードでのみ実行されます。

bdr.ksuuid_v2_cmp

このファンクションは、KSUUID v2 値を比較します。

最初の値がより新しい場合は 1、2 番目の値がより低い場合は-1、等しい場合は 0 を返します。

あらすじ

```
bdr.ksuuid_v2_cmp(uuid, uuid)
```

パラメーター

- UUID-比較する KSUUID v2

注

このファンクションはローカルノードでのみ実行されます。

bdr.extract_timestamp_from_ksuuid_v2

このファンクションは、KSUUID v2 のタイムスタンプコンポーネントを抽出します。結果値のタイプは「timestampz」です。

あらすじ

```
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

このファンクションは、ローカルノードでのみ実行されます。

2.8.5 KSUUID v1 Functions

KSUUID v1 データ、K-Sortable UUID データ（v1）を操作するための Functions。

bdr.gen_ksuuid

このファンクションは、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。結果値の型は「UUID」です。

あらすじ

```
bdr.gen_ksuuid()
```

注

このファンクションは、ローカルノードでのみ実行されます。

bdr.uuid_v1_cmp

このファンクションは、KSUUID v1 の値を比較します。

最初の値がより新しい場合は 1、2 番目の値がより低い場合は-1、等しい場合は 0 を返します。

あらすじ

```
bdr.uuid_v1_cmp(uuid, uuid)
```

注

このファンクションは、ローカルノードでのみ実行されます。

パラメーター

- UUID-比較する KSUUID v1

bdr.extract_timestamp_from_ksuuid

このファンクションは、KSUUID v1 または UUIDv1 値のタイムスタンプコンポーネントを抽出します。結果値のタイプは「timestampz」です。

あらすじ

```
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

このファンクションは、ローカルノードでのみ実行されます。

2.9 Column-Level Conflict Detection

デフォルトでは、競合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のいずれかを選択し、もう一方を破棄します。例、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、新しいもの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することが保証され、クラスター全体でコミット順序のようなセマンティクスが確立されます。

ただし、場合によっては、行レベルレベルではなく列レベルで競合を解決することが適切な場合があります。

2つの整数列 “a” と “b” を持つテーブル “t” と単行 (1,1) があるシンプル例を考えてみましょう。1つのノードで実行と仮定します。

```
UPDATE t SET a = 100
```

…別のノードで同時に (precedUPDATE を受信する前に) 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_newerconflict 解決では、コミットタイムスタンプを比較し、newrow 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、(1,100) になり、列「a」への変更を事実上破棄します。

多くのユースケースでは、これは望ましい動作であり期待される動作ですが、場合によってはこれが問題になる可能性があります-例、アプリケーションの各パートが異なるノードに接続され、共有テーブルの列の専用サブセットを更新するマルチノードクラスターを検討してください。その場合、異なるコンポーネントが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、列レベルで特定のテーブルの競合を解決の方が適切な場合があります。これを実現するために、BDR は各列の最後の変更のタイムスタンプを個別に追跡し、それを使用して最新の値（本質的に update_if_newer）を選択します。

前の例に適用すると、両方のノードで (100,100) になりますが、どちらのノードにもこのような行は表示されません。

列レベルの競合解決を検討する場合、各更新が1つのスライス内のデータのみに影響するように、テーブルが垂直にパーティション化されているように見えると便利な場合があります。これにより、列の異なるサブセットに対する変更間の競合がなくなります。実際、垂直パーティショニングは列レベルの競合解決の実用的な代替手段でさえあります。

列レベルの競合解決には、テーブルに `REPLICAIDENTITY FULL` が必要です。`bdr.alter_table_conflict_detection` ファンクションはそれをチェックし、そうでなければエラーで失敗します。

!!! Note * 現在、この機能は EDB Postgres Extended および EDB Postgres Advanced でのみ利用可能です。

2.9.1 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()` 関数によって管理されます。

例

`bdr.alter_table_conflict_detection()` の使用方法を説明するために、簡単なテーブル `test_table` を作成し、列レベルの競合解決を有効にする次の例を検討します。

```
db=# CREATE TABLE my_app.test_table (id SERIAL PRIMARY KEY, val INT);
CREATE TABLE

db=# ALTER TABLE my_app.test_table REPLICA IDENTITY FULL;
ALTER TABLE

db=# SELECT bdr.alter_table_conflict_detection(
db(# 'my_app.test_table'::regclass, 'column_modify_timestamp', 'cts');
alter_table_conflict_detection
-----
t

db=# \d my_app.test_table
```

ファンクションが新しい `cts` 列を追加することがわかります（ファンクション呼び出しで指定されています）が、各変更の前に新しい列のタイムスタンプを維持する 2 つのトリガー（`BEFORE INSERT` と `BEFORE UPDATE`）も作成しました。

また、言及する価値があるのは、新しい列が `NOT NULL` をデフォルト値で指定していることです。つまり、`ALTER TABLE ... ADD COLUMN` はテーブルの書き換えを実行しません。

- 注*: `bdr.column_timestamps` データタイプの列の使用はお勧めしませんさまざまな悪影響を与える可能性があるため、他の目的のために（テーブルを列レベルの競合解決に切り替えます。これにより、トリガーなどがなければ正しく動作しません。）。

列レベルの競合解決を使用したテーブルの一覧表示

列レベルの競合解決が有効になっているテーブルは、`bdr.column_timestamp` 型の列の存在を検出する次のクエリーでリストできます。

```
SELECT nc.nspname, c.relname
FROM pg_attribute a
JOIN (pg_class c JOIN pg_namespace nc ON c.relnamespace = nc.oid)
    * ON a.attrelid = c.oid
JOIN (pg_type t JOIN pg_namespace nt ON t.typnamespace = nt.oid)
    * ON a.atttypid = t.oid
WHERE NOT pg_is_other_temp_schema(nc.oid)
    * AND nt.nspname = 'bdr'
    * AND t.typname = 'column_timestamps'
    * AND NOT a.attisdropped
    * AND c.relkind IN ('r', 'v', 'f', 'p');
```

`bdr.column_timestamps_create`

このファンクションは、列レベルの競合解決を作成します。これは `withincolumn_timestamp_enable` と呼ばれます。

あらすじ

```
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 ‘コミット’ です。
- `p_timestamp`-タイムスタンプは選択したソースに依存します: ‘コミット’ の場合、次に `TIMESTAMP_SOURCE_COMMIT`; ‘current’ の場合、`TIMESTAMP_SOURCE_CURRENT`。

2.9.2 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と、コードは DDL ロックを使用して、切り替え前から保留中の変更がないことを保証し、両方のタプルのタイムスタンプまたは両方のタイムスタンプとの競合のみを保証します。そうしないと、コードが予期せずローカルタプルにタイムスタンプを、リモートタプルに NULL を検出する可能性があります。また、すべてのノードで同じ方法（列レベルまたは行レベル）で変更が解決されるようにします。

2.9.3 現在とコミットのタイムスタンプ

重要な質問は、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たタイムスタンプは、`clock_timestamp` から取得されたかのように `currenttimestamp` です。これはシンプルで、多くの場合、完全に正しいです（たとえば、競合する行が重複しない列のサブセットを変更する場合）。

ただし、予期しないさまざまな影響がある場合があります。

- ステートメントの実行中にタイムスタンプが変更されるため、`UPDATE` マルチプルの行に影響し、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「混在」になる可能性があることを意味します方法（変更がどのように正確に実行されたかによって異なりますノードインターリーブ）。
- タイムスタンプはコミットタイムスタンプとは無関係であり、それを使用して競合の解決は、結果がコミットオーダーと等しくないことを意味します。これは、おそらくシリアライズできないことを意味します。

注：将来、ステートメントおよび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、同時ステートメントまたはトランザクションの混合効果に関する問題に対処できます。それでも、これらのオプションはいずれも、コミットオーダーと同等の結果を生成することはできません。

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を `true` に設定します。

```
SELECT bdr.column_timestamps_enable('test_table'::regclass, 'cts', true);
```

これは、`bdr.column_timestamps_disable` を使用して無効にすることもできます。

現在、コミットタイムスタンプには、「制限」セクションで説明されているいくつかの制限があります。

2.9.4 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保存する列は、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。例、特定の競合がどのように解決されたかを調査する際に、現在のタイムスタンプ値を調べると便利です。

この目的には3つの機能があります。

- `bdr.column_timestamps_to_text(bdr.column_timestamps)`
 - このファンクションは、タイムスタンプマッピングの人間が読み取れる表現形式を返し、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)`

- このファンクションはタイムスタンプマッピングの JSONB 表現形式を有効にし、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（既にコミットされている場合）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。これは、コミットタイムスタンプを使用する場合にのみ重要です。例、この場合、最後のトランザクションは2番目の属性を（`attnum = 2` で）更新しました：

```
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)
```

2.9.5 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、単にタイムスタンプの高い値を選択し、他の値を破棄します。ただし、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 タイプを使用するなど、さまざまな（より複雑な）方法で競合を調整することができます。

2.9.6 制限

- **UPDATE** によって変更された属性は、トリガーの古い行と新しい行。これは、属性が値を変更しない場合、変更されたとしても検出されません明示的に設定されます。例、`UPDATE t SET a = a` は `a` をマークしません任意の行に対して変更されます。同様に、`UPDATE t SET a = 1` はマークしません `1` に既に設定されている行に対して変更された `a`。
- **INSERT** ステートメントの場合、新しい行と比較する古い行はありません 1 つに、すべての属性を変更することを考慮して割り当てます新しいタイムスタンプ。これは、含まれていない列にも適用されます **INSERT** ステートメントでデフォルト値を受け取りました。検出できたどの属性にはデフォルト値がありますが、次のことを決定することはできませんそれは自動的に含まれるか、ユーザによって明示的に指定されました。
 - これは事実上、**INSERT-INSERT** の競合に対して列レベルの競合解決が機能しないことを意味します（**INSERT** ステートメントが列の異なるサブセットを指定している場合でも、新しい行は古い行よりも新しいタイムスタンプを持つため）。
- 列を個別に処理することにより、コンストレインに違反しやすくなりますすべての変更が同じ場所で発生した場合には不可能な方法でノード例、次のようなテーブルを考えてみましょう。

```
CREATE TABLE t (id INT PRIMARY KEY, a INT, b INT, CHECK (a > b));
INSERT INTO t VALUES (1, 1000, 1);
```

… 1 つのノードが以下を行うと仮定します。

```
UPDATE t SET a = 100;
```

…別のノードが同時に行う場合：

```
UPDATE t SET b = 500;
```

- これらの更新はそれぞれ、最初の行で実行されると有効になるため、各ノードに渡されます。ただし、他のノードに複製する場合、結果の行は **CHECK (A > b)** 制約に違反し、問題が手動で解決されるまでレプリケーションは停止します。
- タイムスタンプマッピングを格納する列は自動的に管理されます。しないでくださいクエリで値を指定またはオーバーライドします。予測不可能な効果（とにかく可能な場合は値を無視します）。
- タイムスタンプマッピングはトリガーによってオーダーされますが、トリガーの実行は重要です。したがって、変更するカスタムトリガーがある場合タプルおよび `pgl_clcd_` トリガーの後に実行され、変更された列は正しく検出されません。

- 変更/コミットをオーダーに通常のタイムスタンプを使用する場合、可能です競合する変更のタイムスタンプがまったく同じであること（なぜなら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ました）。このリスク発生する可能性があるため、列レベルの競合解決に固有ではありません通常の行レベルの競合解決でも、ノード ID をこのシチュエーションでのタイブレーカー（より高いノード ID が勝つ）。同じ変更がすべてのノードに適用されます。
- 異なるノード間にクロックスキューがある可能性があります。それながら
 - やや予期しない動作を引き起こす可能性があります
 - タイムスタンプが反転しているために変更されます)、ノード間のクロックスキューは
 - パラメーター `bdr.maximum_clock_skew` を使用して管理され、
 - `bdr.maximum_clock_skew_action`。

```
SELECT bdr.alter_node_group_config('group', ignore_redundant_updates := false);
```

2.10 Conflict-free Replicated Data Types

競合のない複製されたデータ型（CRDT）は、行の1つを破棄する代わりに、同時に変更された行からの値のマージをサポートします（これは従来の競合解決が行うことです）。

各 CRDT タイプは、`bdr.crdt_handlers` カタログに追加のコールバックが追加された個別の PostgreSQL データタイプとして実装されます。マージプロセスは、適用側の BDR ライタ内で発生します。追加のユーザー操作は必要ありません。

CRDT では、[CLCD](#) の章で説明されているように、テーブルで列レベルの競合解決を有効にする必要があります。

ユーザが行う唯一のアクションは、整数などの標準のビルトインではなく、`CREATE / ALTER TABLE` で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と1つの行がある次のテーブルを考えてください。

```
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 を同時に発行した場合：

```
UPDATE non_crdt_example
  * SET counter = counter + 1    -- "reflexive" update
WHERE id = 1;
```

…結果の値は、両方の更新が適用された後、このクエリーを使用して表示できます。

```
SELECT * FROM non_crdt_example WHERE id = 1;
* id | counter
-----+-----
* 1 | 1
(1 row)
```

…update_if_newerconflict リゾルバにより、増分の1つが失われたことを示しています。代わりに CRDT カウンタデータタイプを使用する場合は、次のようなことを確認する必要があります。

```
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);
```

ここでも、2つのノードで次の SQL を同時に発行し、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)
```

これは、CRDT がアキュムレーター列を正しく許可することを示しています。そうでなければ、競合する非同期の同時更新に直面してもです。

crdt_gcounter 型は、上記のように $x = x + 1$ などの再帰的な UPDATE SQL でのみ機能する状態ベースの CRDT 型の例です。

bdr.crdt_raw_value 構成オプションは、クエリが CRDT タイプの現在の値または完全な内部状態を返すかどうかを決定します。デフォルトでは、現在の数値のみが返されます。true に設定すると、クエリは完全な状態の表現形式を結果ます。特別なハッシュ演算子 (#) を使用して、特別な演算演算子を使用せずに現在の数値のみを要求できます（これはデフォルトの動作です）。bdr.crdt_raw_value = on を使用して完全な状態がダンプされる場合、値は bdr.crdt_raw_value = on でのみ再取得できます。

注：bdr.crdt_raw_value は、クライアントに返されるデータのフォーマットのみを適用します。つまり、選択リスト内のシンプル列参照。クエリーの他の部分の列参照（たとえば、WHERE 句または選択リストの偶数式）でも、#演算子の使用が必要になる場合があります。

別のクラスの CRDT データ型が存在します。これは「デルタ CRDT」型と呼ばれます（後で説明するように、操作ベースの CRDT の特別なサブクラスです）。

デルタ CRDT を使用すると、値の更新は同じノードの以前の値と自動的に比較され、その後、他のすべてのノードで変更が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 を同時に発行した場合：

```
UPDATE crdt_delta_example
  *   SET counter = 2          -- notice NOT counter = counter + 2
WHERE id = 1;
```

結果の値は、両方の更新が適用された後、このクエリーを使用して表示できます。

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
  *   id | counter
-----+-----
  *   1 |      4
(1 row)
```

通常の integer 列では、結果はもちろん 2 になります。しかし、デルタ CRDT カウンタで行を更新する場合、OLDrow バージョンからスタートし、新しい行バージョンを make して、両方をリモートノードに送信し、そこで見つかったバージョンと比較します（LOCAL バージョンと呼びましょう）。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て、デルタを LOCAL バージョンに適用します。

CRDT タイプは bdr のパートとして 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!）および多くの一般的な集計関数（min、max、sum、make、edcreate_tran_8 tweaksearch_path）を使用する必要はありません。

重要な質問は、クエリー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 タイプ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推定とクエリープランニングが正常に機能します。

!!! Note * 現在、この機能は EDB Postgres Extended および EDB Postgres Advanced でのみ利用可能です。

2.10.1 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 を実装します。

操作ベースの CRDT タイプ (CmCRDT)

オペレーション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信した古い行と新しい行から計算されるため、操作ベースのタイプの実装は非常に簡単です。

現在、これらの操作ベースの CRDT を実装しています。

- `crdt_delta_counter-bigint` カウンタ (増分/減分)
- `crdt_delta_sum-numeric` 総和 (増分/減分)

これらのタイプは、デルタを計算するためのビットのコードを使用して、既存のデータタイプ (`bigint` の `crdt_delta_counter` isa ドメインなど) を活用します。

このアプローチは、デルタの計算方法がわかっている型に対してのみ可能ですが、結果は非常にシンプルで安価であり (スペースと CPU の両方の点で)、いくつかの追加の利点があり構文 (たとえば、横たわっているデータタイプ)。

主な欠点は、非同期環境と同時環境ではこの値を確実にリセットできないことです。

注: カスタムデータ型を作成し、状態と最後のオペレーションを保存することにより、より複雑な操作ベースの型を実装することもできます (個々の変更をすべてデコードして転送するため、複数の操作は必要ありません)。ただし、その時点では、スペース要件を除いて、状態ベースのタイプ (リセットする機能はまだありません) に比べて利点を得ることなく、主な利点 (単純さ、既存のデータ型の再利用) を失いますノードごとの状態)。

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 を実装しています。

- `crdt_gcounter-bigint` カウンタ (増分のみ)
- `crdt_gsum-numeric` 総和/ カウンタ (増分のみ)
- `crdt_pncounter-bigint` カウンタ (増分/減分)
- `crdt_pnsum-numeric` 総和/ カウンタ (increments / decrements)

通常、内部状態にはノードごとの情報が含まれディスク上の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 (入力/出力関数および演算子) を意味します。

利点は、値を確実にリセットする機能、失われた変更 (適切に運用されたクラスターでは発生しないはず) が存在する場合の自己回復機能、およびソースノード以外からの変更を受信する機能です。

例、ノード A で値が変更され、変更ゲットが B に複製され、C ではなく（A と C 間のネットワークの問題のため）、B が値を変更し、この変更が C に複製される場合、A からのオリジナルの変更。操作ベースの CRDT では、AC ネットワーク接続が再び機能するまでノード C は変更を受信しませんでした。

CvCRDT の主な欠点は、ディスクスペースの点でコストが高いことです-クラスターから既に削除されたノードを含む各ノードの情報がビット必要です。状態の複雑な性質（varlena 型にシリアル化）は、CPU 使用率の増加を意味します。

2.10.2 ディスクスペースの要件

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、タイプは他のタイプの上にあるドメインにすぎず、同じディスクスペース要件を持っているため（ノードがいくつあっても）、これはかなり簡単です。

- `crdt_delta_counter-bigint` と同じ（8 バイト）
- `crdt_delta_sum-numeric` と同じ（変数、精度に応じておおよそスケール）

操作ベースの CRDT タイプはノードごとの情報を保存しないため、ノードの数に依存しません。

状態ベースのタイプの場合、シチュエーションはより複雑です。すべてのタイプは可変長であり（本質的に `bytea` 列として格納されます）、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お考えください。

- `crdt_gsum`
- 修正：20B (header) + N * 4B (per-node)
- 変数：(2 + N) numeric 値
- `crdt_pnsum`
- 修正：20B (header) + N * 4B (per-node)
- 変数：(4 + 2 * N) numeric 値
- 注*：クラスター内にいくつかのノードがあるかは関係ありません。値がマルチプルのノードで更新されることはありません。また、それは重要ではありません更新は同時（競合の原因）であったかどうか。
- 注*：また、それらのノードがすでにいくつあったかは関係ありません。クラスターから削除されました。状態を圧縮する方法はまだありません。

2.10.3 CRDT タイプと競合処理

テーブルには CRDT 列と非 CRDT 列の両方が含まれる可能性があるため（実際、ほとんどの列は非 CRDT であると予想されます）、通常の競合解決と CRDT マージの両方を行う必要があります。

競合解決が最初に発生し、保持するタプル（適用するタプル）と破棄するタプルを決定する責任があります。次にマージフェーズが発生し、CRDT 列のデータが破棄されたタプルから適用タプルにマージされます。

- 注*: これにより、CRDT タイプはプレーンに比べてやや高価になります競合の解決。マージは毎回発生ニーズがあるため、競合解決が高速パスのいずれかを使用できる場合（変更現在のトランザクションなどで）。

2.10.4 CRDT タイプと競合レポート

デフォルトでは、検出された競合はサーバーログに書き込まれます。CRDT タイプがなければ、これは完全に理にかなっていません。なぜなら、競合の解決は本質的に利用可能な情報の半分を破棄するからです（設定によってはローカルまたはリモート行）。これはデータロスを示します。

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組み合わせて、データロスの問題を解消できます。これにより、競合レポートが不要になります。

このため、CRDT マージによって競合を完全に解決できる場合、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、競合レポートをスキップします。

- 1) ローカルとリモートのタプルの値が同じ（NULL または等しい）
- 2) CRDT データタイプを使用します（したがって、マージできます）

- 注*: これは、競合がない場合にも競合レポートをスキップすることを意味します CRDT 列。ただし、ローカル/リモートタプルのすべての値は等しい。

2.10.5 CRDT 値のリセット

CRDT 値のリセットは可能ですが、特別なハンドリングが必要です。クラスターの非同期性は、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オペレーションを見ることができることを意味します（実装方法は関係ありません）。異なるノードも同時にリセットを開始できます。すなわち、他のノードからのリセットを観察する前に。

言い換えれば make、リセットオペレーションが正しく動作するためには、通常の操作に対して可換であるニーズがあります。値をリセットするための多くの単純な方法（単一ノードで完全に機能する場合があります）は、まさにこの理由で失敗します。

例、値をリセットする最も簡単な方法は次のとおりです。

```
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り追加され、値が発散し、最終的に他のノードの変更によりそれを受け取ります。

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実行しようすると、デルタを2回適用し、負の値を取得します（リセットから予想したものではありません）。

`DELETE + INSERT` をリセットとして使用することもできますが、これ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、すべてのノードが（他のノードからの変更に関して）操作のストリームの同じ位置に表示されることは保証されません。BDR は、同じプライマリキー値を再利用することを特に推奨します。これは、同時ケースではデータの異常を引き起こす可能性があるためです。

状態ベースの CRDT タイプは、次のような特別な!演算子を使用して、確実にリセットをハンドルできます。

```
UPDATE tab SET counter = !counter WHERE ...;
```

「信頼できる」とは、値に上記の2つの問題（マルチプルの同時リセットと発散）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、マルチプルの同時リセットが回避されるためです。

2.10.6 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-成長のみのカウンタと総和、正負のカウンタと総和、およびデルタカウンタと総和。「総和」タイプは10進数ベース（`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。

現在実装されている CRDT データ型は、次のクエリーでリストできます。

```
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 (`crdt_gcounter`)

- 負でない値の増分のみをサポート (`value + int`
 - および `counter + bigint` 演算子)
- カウンタの現在の値は、`#`演算子を使用して取得できます。または `bigint` にキャストして
- `counter = value` のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション）
- `!`演算子を使用して、カウンタをシンプルにリセットできます。（`counter = !counter`）
- `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;

```

成長のみの総和 (crdt_gsum)

- 負でない値の増分のみをサポートします (sum + numeric)
- 総和の現在の値は、#演算子を使用して取得できます。または numeric にキャストして
- sum = value のようシンプル割り当てと互換性がない (これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション)
- !演算子 (sum = !sum) を使用して、総和をシンプルにリセットできます。
- crdt_gsum_to_text を使用して内部状態を検査できます

```
CREATE TABLE crdt_test (  
  *   id          INT PRIMARY KEY,  
  *   gsum        bdr.crdt_gsum NOT NULL DEFAULT 0.0  
);  
  
INSERT INTO crdt_test VALUES (1, 0.0);      -- initialized to 0  
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24  
INSERT INTO crdt_test VALUES (3, -45.31);  -- error: negative value  
  
-- 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;
```

正負カウンタ (crdt_pncounter)

- 正と負の両方の値を持つ増分をサポート（を通じて
 - counter + int および counter + bigint 演算子）
- カウンタの現在の値は、#演算子を使用して取得できます。または bigint にキャストすることにより
- counter = value のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション）
- !演算子を使用して、カウンタをシンプルにリセットできます。（counter = !counter）
- 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;

```

正負の総和 (crdt_pnsum)

- 正と負の両方の値を持つ増分をサポート（を通じて
 - `sum + numeric`）
- 総和の現在の値は、`#`演算子を使用して取得できます。または `numeric` にキャストすることにより
- `sum = value` のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるときの一般的なパターンですアプリケーション）

- !演算子 (sum = !sum) を使用して、総和をシンプルにリセットできます。
- crdt_pnsum_to_text を使用して内部状態を検査できます

```
CREATE TABLE crdt_test (  
  *   id          INT PRIMARY KEY,  
  *   pnsum       bdr.crdt_pnsum NOT NULL DEFAULT 0  
);  
  
INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0  
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24  
INSERT INTO crdt_test VALUES (3, -45.31); -- initialized to -45.31  
  
-- 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 (`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
- 正と負の両方の値を持つ増分をサポート
- `counter = value` のようシンプル割り当てと互換性があります（新しい値がアプリケーションのどこかで計算される場合に一般的）
- 値をリセットするシンプル方法はありません（確実）

```
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;
```

総和 (`crdt_delta_sum`)

- `numeric` ドメインとして定義されているため、`numeric` 列とまったく同じように機能します
- 正と負の両方の値を持つ増分をサポート
- `sum = value` のようシンプル割り当てと互換性があります（新しい値がアプリケーションのどこかで計算される場合に一般的）
- 値をリセットするシンプル方法はありません（確実）

```

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 >

2.11 Transaction streaming

ロジカルレプリケーションでは、トランザクションはパブリッシャーで同時にデコードされますが、トランザクションがコミットされるまでサブスクライバーに送信されません。変更が `logical_decoding_work_mem` (PostgreSQL 13 以降) を超える場合、変更はディスクに書き込まれます。これは、特にラージトランザクションでは、サブスクライバーに到達する前に多少の遅延が発生し、パブリッシャーで追加の I/O が必要になる場合があることを意味します。

PostgreSQL 14 以降、オプションでトランザクションをデコードし、パブリッシャーでコミットする前に*サブスクライバーに送信できます。サブスクライバーは、着信した変更をステージングファイル（またはファイルのセット）に保存し、トランザクションのコミット時にそれらを適用します（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。

この機能は、PostgreSQL 14 (EPAS 14 および EDB Postgres Extended 14 を含む) から利用でき、EDB Postgres Extended 13 にバックポートされています。

2.11.1 BDR の機能強化

上記のように、PostgreSQL のビルトイントランザクションストリーミングには次の制限があります。

- パブリッシャーのディスクに変更をスピルする必要はもうありませんが、変更は各サブスクライバーのディスクに書き込まれる
- トランザクションが中断した場合、作業（各サブスクライバーが受信した変更関連付けられたストレージ I/O）は無駄になります

ただし、バージョン 3.7 以降、BDR は Parallel Apply をサポートし、各サブスクライバーでマルチプルのライタープロセスを有効にします。これにより、以下の機能強化が提供されます。

- デコードされたトランザクションは、サブスクライバー上のライタに直接ストリーミングできます。
- デコードされたトランザクションは、サブスクライバーのディスク上に保存する必要はありません
- 適用を開始する前にトランザクションがコミットのを待つ必要はありませんサブスクライバーでのトランザクション

警告

- 並列適用を有効にする必要があります
- ワークロードが多数の小規模で競合するトランザクションで構成されている場合、これは
 - 作家間の頻繁なデッドロック

ライターへのストリーミングはまだ実験的な機能であり、

注意して使用してください。具体的には、ストリーミングのコミットタイムスタンプが利用できない可能性があるため（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るため）、競合解決ではうまく機能しない可能性があります。

2.11.2 設定

トランザクションストリーミングは、次の 2 つの場所で構成されます。

- ノードレベルで、GUC `bdr.default_streaming_mode` 経由
- グループレベルで、ファンクション `bdr.alter_node_group_config()` を使用

bdr.default_streaming_mode を使用したノード設定

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。

有効にするには、この設定を変更するには、各サブスクリプションの pglogical レシーバプロセスをリスタートする必要があります。これは、serverrestart 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も依存します。詳細については、以下のセクション XXX を参照してください。

bdr.alter_node_group_config () を使用したグループ設定

ファンクション `bdr.alter_node_group_config()` のパラメータ `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、ビューの `node_group_streaming_mode` 列に含まれています。返される値は単一の char 型であり、可能な値は (、)、(、)、(、) および (、) です。

構成設定の効果

トランザクションストリーミングは、GUC `bdr.default_streaming_mode` によってサブスクライバーレベルで制御されます。off (transactionstreaming を無効にする) に設定しない限り、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 `inlogical_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ウィルスストリームします。パブリッシャーは通常、トランザクションをファイルにストリーミングするかライタにス

トリーミングするかを制御できません。状況 (COPY など) を除き、トランザクションをライターにストリームするようサブスクライバーにヒントを与える場合があります (可能な場合)。

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ーまたはファイルにストリームできます。決定はいくつかの要因に基づいています。

- 並列適用がオフの場合 (`num_writers = 1`)、ファイルにストリーミングされます (ライター 0 は常に非ストリーミングトランザクション用に予約しています)
- 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のハンドリングですでに忙しい場合トランザクションの場合、新しいトランザクションはファイルにストリーミングされます。見る `[bdr.writers]` (モニタリング.md # Monitoring BDR Writers) で BDR を確認します ライターのステータス。

ライターへのストリーミングが可能な場合 (つまり、フリーのライターが利用できる場合)、トランザクションをライターまたはファイルにストリームするかどうかの決定は、次の表のグループとノードの設定の組み合わせに基づきます。

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` であり、ノード構成が `auto` である場合、パブリッシャーがこれを行うことを示唆した場合にのみ、トランザクションがライターにストリーミングされることに注意してください。

現在、パブリッシャーは、次のトランザクションタイプについてライターにストリームするようにサブスクライバーにヒントを提供します。

- COPY
- CREATE INDEX CONCURRENTLY

2.11.3 監視

トランザクションストリーミングの使用は、サブスクライバノードの `bdr.stat_subscription` 関数を介して監視できます。

- `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`-中止されたストリーミングトランザクションの数
- `nstream_start`-開始されたストリーミングトランザクションの数
- `nstream_stop`-完全に受信されたストリーミングトランザクションの数

2.12 Durability & Performance Options

2.12.1 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。これにより、アプリケーションに関連する3つのプロパティが提供されますが、これらは関連していますが、すべて個別に実装できます。

- **耐久性**：マルチプルのノードへの書き込みにより、クラッシュ障害性が向上また、クラッシュおよびリスタート後にデータを回復できます。
- **可視性**：クライアント、データベースへのコミット確認付きいくつかのコミットされたトランザクションの即時可視性を保証しますノードのセット。
- **コミット後の競合なし**：クライアントはトランザクションに依存して、最終的に、さらに競合することなくすべてのノードに適用されるか、またはクライアントにエラーを直接通知するアボート。

BDR は Postgres 自分自身の `synchronous_commit` オプションと統合され、BDR ノード間で使用できる同期レプリケーションのバリエーションを提供します。BDR は、2つの追加のレプリケーションモードも提供します。

- **最大1回コミット (CAMO)**。この機能は、知っていることで問題を解決しますトランザクションがコミット（および複製）されているかどうか COMMIT 中の特定のエラー。通常、次のことを知るのハードかもしれません COMMIT が処理されたかどうか。この機能により、アプリケーションは新しいデータベースコネクションがノードへの場合でも、何が起こったかを調べる以前の接続よりも。この機能の詳細については、[Commit At Most Once](#) の章。
- **熱心なレプリケーション**。これはレプリケーションを避けるためのオプション機能です競合。すべてのトランザクションは**すべてのノード**に同時に適用され、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します。この機能はパフォーマンスを低下させますが、非常に強力な一貫性保証を提供します。この機能の詳細については、[Eager All-Node Replication](#) を参照してください。章。

Postgres 自分自身は (PSR) を提供します。これは単方向ですが、BDR と組み合わせて使用できるものを提供します。

この章では、同期的または `eagerreplication` のさまざまな形式とそのタイミングの側面について説明します。

2.12.2 比較

BDR で利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、異なる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異なる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 確認を受信した後、複製されたピアノードからクライアントが期待できるものをまとめたものです。

Variant	Mode	Received	Visible	Durable
PGL/BDR	off (default)	no	no	no
PGL/BDR	remote_write (2)	yes	no	no
PGL/BDR	on (2)	yes	yes	yes
PGL/BDR	remote_apply (2)	yes	yes	yes
PSR	remote_write (2)	yes	no	no (1)
PSR	on (2)	yes	no	yes
PSR	remote_apply (2)	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

- (1) OS に書き込まれ、OS の実行が継続し、Postgres がクラッシュします。*
- (2) 設定によりローカルモードに切り替えられない限り（許可されている場合）`synchronous_replication_availability` から `async`、それ以外の場合非同期 BDR のデフォルトの値が適用されます。*

受信により、同等な最終的なトランザクションのすべての変更を、それ以上の通信を必要とせずに、つまりネットワークの完全または部分的な停止に直面することなく最終的に適用できるようになります。同期的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、同時トランザクションとの可能な競合は解決されました。永続性がない場合、つまり、トランザクションを永続化する前に、ピアノードのクラッシュがこのノードを元に戻す可能性があります（そして、変更を再送信して再適用する必要があります）。

耐久性は、同等なストレージに関連し、同等なクラッシュとリカバリ後のデータの損失に対する保護を提供しノード。クラッシュの前にトランザクションがすでに可視されていた場合、再び可視されるように回復されます。それ以外の場合、トランザクションのペイロードは永続化され、同等なトランザクションを最終的に（データの再ノードを必要とせずに）適用できます。

2.12.3 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かなり異なって適用することを理解することが役立ちます。

物理的ストリーミングレプリケーションでは、操作のオーダーは次のとおりです。

- オリジンはコミットレコードを WAL にフラッシュし、トランザクションを作成しますローカルに可視
- 同等な変更を受信し、書き込みを発行しノード
- 同等な変更を受信した変更をディスクにフラッシュします
- 同等な変更を適用し、トランザクションをローカルに可視します

BDR では、操作のオーダーが異なります。

- オリジンはコミットレコードを WAL にフラッシュし、トランザクションを作成しますローカルに可視
- 同等な変更はメモリの適用キューへの変更を受信しノード
- 同等な変更を適用し、トランザクションをローカルに可視します
- 同等な変更はディスクにフラッシュすることでトランザクションを永続化します

CAMO および Eager All Node Replication の場合、トランザクションをローカルで表示する前に、オリジンノードが確認を待つことにノートしてください。操作のオーダーは次のとおりです。

- オリジンは、準備または事前コミットレコードを WAL にフラッシュします
- 同等な変更はメモリの適用キューへの変更を受信しノード
- 同等な変更を適用し、トランザクションをローカルに可視します
- 同等な変更はディスクにフラッシュすることでトランザクションを永続化します
- オリジンはコミットし、トランザクションをローカルに可視します

次の表は、違いをまとめたものです。

Variant	Order of apply vs persist on peer nodes	Replication before or after origin WAL commit record write
PSR	persist first	after
BDR	apply first	after
CAMO	apply first	before (triggered by pre-commit)
Eager	apply first	before (triggered by prepare)

2.12.4 設定

次の表は、デフォルト値（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する構成設定の概要を示しています。

setting (GUC)	PSR	PGL	CAMO	Eager
synchronous_standby_names	req	req	n/a	n/a
synchronous_commit	opt	opt	n/a	n/a
synchronous_replication_availability	opt	opt	opt	n/a
bdr.enable_camo	n/a	n/a	req	n/a
bdr.commit_scope	n/a	n/a	n/a	req
bdr.global_commit_timeout	n/a	n/a	opt	opt

2.12.5 計画的なシャットダウンと再起動

PGL または CAMO を `remote_write` と組み合わせて使用する場合は、計画的なシャットダウンまたはリスタートに注意する必要があります。デフォルトでは、適用キューはシャットダウン前に消費されます。ただし、`immediate` シャットダウンモードでは、キューはシャットダウンに破棄され、停止したノードはキュー内のトランザクションを「忘れる」ことになります。別のノードで同時に障害が発生すると、両方のノードで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保証には、メンテナンスタスクに `smart` または `fast` のいずれかのシャットダウンを使用してください。これにより、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。

2.12.6 BDR を使用した同期レプリケーション

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 同等名の `applicationname` を `tosynchronous_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、これが非 BDR スタンバイノードの要件と競合しない場合、多くの柔軟性を提供します。

追加された同期レベルは、`synchronous_commit` を使用してトランザクションごとに構成できます。デフォルトは `on` です。つまり、`synchronous_standby_names` に追加すると、すでに同期レプリケーションが有効になっています。`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（論理複製の場合）。

制限

BDR は、物理ストリーミングレプリケーションと同じ構成（および内部メカニズム）を使用するため、BDR ノード間の同期レプリケーションを構成する際に（物理的、非 BDR）スタンバイのニーズ性を考慮するニーズがあります。

2.13 Eager Replication

```
<AuthenticatedContentPlaceholder target = " https:// 文書.enterprisedb.com / bdr4 / リリース/ latest / eager /  
" トピック= "Eager Replication" />
```

2.14 CAMO

```
<AuthenticatedContentPlaceholder target = " https:// 文書.enterprisedb.com / bdr4 / リリース/ latest / camo /  
" トピック= "CAMO" />
```

2.15 AutoPartition

AutoPartition では、自動パーティション管理により、テーブルを簡単にラージサイズに拡大できます。これは、パーティションの作成と削除の低競合ロックなど、BDR の追加機能を利用します。

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すると削除されます。

BDR 管理は、主に SQL 呼び出し可能関数を介して実行されます。BDR のすべての関数は、BDR スキーマで公開されます。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で修飾する必要があります。

2.15.1 パーティションの自動作成

bdr.autopartition() は、テーブルの自動範囲パーティショニングの定義を作成または変更するために使用されます。定義が存在しない場合は定義が作成され、そうでない場合は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せず、新しいパーティションメンテナンスアクションが実行されるタイミングと方法の定義を変更するだけです。

bdr.autopartition() は、EDB Postgres 拡張機能を活用して、テーブルの残りの部分をロックせずにパーティションをアタッチまたはデタッチ/ドロップできるようにします。この機能は、SELECT クエリを許可しながら新しいテーブルスペースを設定します。

テーブルが RANGE パーティション化されていない場合、または複数列パーティションキーが使用されている場合は、エラーが発生します。

partition_increment の値のレンジごとに、下限と上限の partition_increment が離れた新しいパーティションが追加されます。タイプ timestamp または date の partitionkey を持つテーブルの場合、

`partition_increment` はタイプ `interval` の有効な定数でなければなりません。例、1 Day を指定すると、新しいパーティションが毎日追加され、パーティションの境界は1日離れます。

パーティション列が `timeshard` または `ksuuid` シーケンスに接続されている場合、`partition_increment` は `interval` タイプとして指定する必要があります。それ以外の場合、パーティションキーが整数または数値の場合、`partition_increment` は同じデータ型の有効な定数でなければなりません。例、「1000000」を指定すると、100 万の値ごとに新しいパーティションが追加されます。

テーブルに既存のパーティションがない場合、指定された `partition_initial_lowerbound` が `first-partition` の下限として使用されます。`partition_initial_lowerbound` が指定されていない場合、システムはパーティション列タイプと指定された `partition_increment` からその値を導き出します。例、`partition_increment` が 1Day として指定されている場合、`partition_initial_lowerbound` は自動的に `CURRENTDATE` に設定されます。`partition_increment` が 1 Hour として指定されている場合、`partition_initial_lowerbound` は `currentdate` 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オーダーに、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 評価可能な式にすることができます。例、列タイプ `date` のパーティションテーブルでは、`ifpartition_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、最後のパーティションの上限が `thandb_tran_8` 未満になるまで新しいパーティションが作成されます。

式は、システムが新しいパーティションをチェックするたびに評価されます。

列タイプ `integer` のパーティションテーブルの場合、`partition_autocreate_expression` は `SELECT max(partcol) FROM schema.partitioned_table` として指定できます。次に、システムは、パーティション化された列の最大値が最後のパーティションの上限の `minimum_advance_partitions * partition_increment` の距離内にあるかどうかを定期的にチェックします。クエリーが効率的に実行されるように、ユーザが `partcol` にインデックスを作成することが期待されます。`partition_autocreate_expression` が列タイプ `integer`、`smallint`、または `bigint` のパーティションテーブルに指定されていない場合、システムは自動的に `max(partcol)` に設定します。

`data_retention_period` が設定されている場合、パーティションはこの期間後に自動的にドロップされます。ロックを最小限に抑えるため、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。設定しない場合、パーティションを手動で削除する必要があります。

`data_retention_period` パラメータは、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ます。期間は、パーティションの上限を考慮して計算され、パーティションはセカンダリテーブルスペースに移行されるか、上限と比較して指定された期間のいずれかが期限切れになると削除されます。

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。そのため、すべてのパーティションは一貫しており、使用可能であることが保証されています。このために、`AutoPartition` は `Raft` を使用します。この動作は、`managed_locally` を `true` として渡すことで変更できます。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、パーティション化されたテーブルがレプリケートされた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配置する必要がない、または望ましくない場合に役立ちます。例、組み込みの `bdr.conflict_history`

テーブルはレプリケートされたテーブルではなく、AutoPartition によってローカルで管理されます。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、十分に古くなるとそれらを削除します。

managed_locally としてマークされたテーブルは、後でグローバルに管理するために変更することはできません。

アクティビティは、エントリーが enabled = on とマークされている場合にのみ実行されます。

ユーザは、AutoPartition によって管理されるテーブルのパーティションを手動で作成または削除することは想定されていません。これを make と、AutoPartition メタデータが不整合になり、失敗する可能性があります。

AutoPartition 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ニングを構成します。

あらすじ

```
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-次のパーティション作成までのインターバルまたはインクリメント。
- partition_initial_lowerbound-テーブルにパーティションがない場合、この下限と partition_increment 離れた上限を持つ最初のパーティションバインドが作成されます。
- partition_autocreate_expression-は、新しいパーティションを作成するときかどうかを検出するために使用されます。
- minimum_advance_partitions-システムは常に最小の minimum_advance_partitions パーティション。
- maximum_advance_partitions-単一で作成されるパーティションの数アドバンスパーティションの数が minimum_advance_partitions を下回ったら行きます。
- data_retention_period-次の場合、古いパーティションが削除されるまでのインターバル定義済み。これは migrate_after_period より大きい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化/一時停止し、後で再開/再有効化できます。

例

毎日のパーティション、1 か月間データを保持します。

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

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

残りのパーティションが2つしかない場合、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10億の注文を保持できます）。

```
bdr.autopartition('Orders', '10000000000',
* partition_initial_lowerbound := '0',
* minimum_advance_partitions := 2,
* maximum_advance_partitions := 5
* );
```

AutoPartition を1つ作成する

`bdr.autopartition_create_partition()` を使用して、親テーブルにスタンドアロン AutoPartition を作成します。

あらすじ

```
bdr.autopartition_create_partition(relname regclass,
*                               partname name,
*                               lowerb text,
*                               upperb text,
*                               nodes oid[]);
```

パラメーター

- **relname**-接続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を停止する

`bdr.drop_autopartition()` を使用して、指定されたリレーションの自動パーティション分割ルールを削除します。リレーションの保留中の作業項目はすべて削除され、新しい作業項目は作成されません。

```
bdr.drop_autopartition(relation regclass);
```

パラメーター

- **relation**-テーブルの名前または OID

AutoPartition を 1 つドロップする

このファンクション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。partitionable が正常に削除された場合、ファンクションは true を結果返します。

あらすじ

```
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

メモ

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つ

`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前とパーティションキー列の値を取得し、その値を保持するパーティションが作成されるまで待機します。

この関数は、パーティションがローカルで作成されるのを待つだけです。パーティションがリモートノードにも存在することを保証するものではありません。

すべての BDR ノードでパーティションが作成されるのを待つには、オーダー関クションを使用します。この機能は、ローカルノードとすべてのリモートノードを内部的にチェックし、パーティションが至る所に作成されるまで待機します。

あらすじ

```
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。

あらすじ

```
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、ファンクションは `NULL` を返します。それ以外の場合、パーティションの `OID` が返されます。

あらすじ

```
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値。

AutoPartitioning を有効/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。 `AutoPartitioning` が既に有効になっている場合は、何もしません。同様に、 `bdr.autopartition_disable()` を使用して、特定のテーブルで `AutoPartitioning` を無効にします。

あらすじ

```
bdr.autopartition_enable(relname regclass);
```

パラメーター

- `relname`-`AutoPartitioning` を有効にするリレーションの名前。

あらすじ

```
bdr.autopartition_disable(relname regclass);
```

パラメーター

- `relname-AutoPartitioning` を無効にするリレーションの名前。

あらすじ

```
bdr.autopartition_get_last_completed_workitem();
```

クラスタ内のすべてのノードで正常に完了した最後のワークアイテムの id を返します。

AutoPartition ワーカーを確認する

`bdr.autopartition_work_queue_check_status` ファンクションを使用すると、AutoPartitions を維持するためにジョブを実行しているバックグラウンドワーカーのステータスを確認できます。

ワーカーは次のビューで確認できます:`autopartition_work_queue_local_status``autopartition_work_queue_global_`

あらすじ

```
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

AutoPartition ワーカーは、`bdr.autopartition` ファンクションが初めて呼び出される前であっても、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 が使用されている場合、ファンクションは「不明」を結果ます。「進行中」は一般的なステータスです。

2.16 Timestamp-Based Snapshots

PG Extended のタイムスタンプベースのスナップショット機能により、通常の MVCCsnapshot ではなく、ユーザー指定のタイムスタンプを介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、異なる BDR ノード上のデータに共通のポイントインタイムでアクセスできます。例、データ品質チェックのために複数のノードでデータを比較する方法として。現時点では、この機能は書き込みトランザクションでは機能しません。

!!! Note * 現在、この機能は EDB Postgres Extended および EDB Postgres Advanced でのみ利用可能です。

タイムスタンプベースのスナップショットの使用は、`snapshot_timestamp` パラメーターによって有効になります。これは、現在のタイムスタンプ（現在）を表すタイムスタンプ値または特別な値 'current' を受け入れます。`snapshot_timestamp` が設定されている場合、クエリは通常の MVCC セマンティクスではなく、そのタイムスタンプを使用して行の可視性を判断します。

例、次のクエリーは、2018-12-08 02:28:30 GMT の `customers` テーブルの状態を結果ます。

```
SET snapshot_timestamp = '2018-12-08 02:28:30 GMT';
SELECT count(*) FROM customers;
```

単純な PG 拡張では、これは将来のタイムスタンプまたは上記の特別な「現在の」値でのみ機能するため、履歴クエリには使用できません（長期ロードマップ上にあります）。

BDR は、マルチノード環境でその機能と連携して機能を改善します。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 snapshot の前にデータベースに追加された未処理のデータをレプリケートすることを確認します。次に、BDR は、`bdr.timestamp_snapshot_keep` という追加のパラメータを追加します。これはそのノードの最近の歴史に対して `whichqueries` を実行できる時間の窓を指定します。

任意のインターバルを指定できますが、`VACUUM`（オートバキュームを含む）は、指定された間隔の 2 倍までの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、ユーザテーブルの膨張が大きくなる可能性があります。最初は、必要に応じて変更することをお勧めしますが、一般的な設定として 10 秒をお勧めします。

クエリが実行のために受け入れられると、クエリーは通常どおり、問題なく `bdr.timestamp_snapshot_keep` より長くクエリーされる可能性があることに注意してください。

また、スナップショットがどこまで保持されたかについての情報はサーバーのリスタートにノットられないため、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 インスタンスの最後のリスタートの時刻です。

`bdr.timestamp_snapshot_keep` の使用と `postgres_fdw` 拡張を組み合わせると、aBDR グループのマルチプルのノードで一貫した読み取りを取得できます。これは、外部テーブルと組み合わせると使用される場合、ノード間で並列クエリを実行するために使用できます。

この機能を使用する場合、マルチノードクエリー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。クエリーデータに加えて、タイムスタンプ値のみが渡されます。

2.17 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すると、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、グループ自分自身と同じ名前のレプリケーションセットを自動的に作成します。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、すべてのユーザテーブ

ルと DDL レプリケーションに使用され、すべてのノードがサブスクライブされます。つまり、デフォルトでは、すべてのユーザテーブルがすべてのノード間でレプリケートされます。

2.17.1 レプリケーションセットの使用

追加のレプリケーションセットは、`create_replication_set()` を使用して作成できます。挿入、更新、削除、または切り捨てアクションを含めるかどうかを指定します。既存のテーブルをセットに自動的に追加するオプションがあり、2 番目のオプションは作成時にテーブルを自動的に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後も保持されます。

ノードが参加しても、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、新しいテーブルの追加は再同期オペレーションで行う必要があります。

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 `BDRadministration` ファンクション呼び出しを複製しません。`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、すべてのノードでレプリケーションセット定義を作成します。その後、各ノードを定義して、`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および/またはサブスクライブします。

これらの定義を後で変更したり、`replicationset` を削除したりする Functions があります。

!!! Note * 選択レプリケーションにデフォルトのレプリケーションセットを使用しないでください。デフォルトの DDL レプリケーションおよび管理ファンクション呼び出しにも使用されるため、クラスター内の BDR ノードのデフォルト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。

2.17.2 パーティションテーブルの動作

BDR はテーブルパーティション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、パーティションに関する変更がダウンストリームにレプリケートされます。

!!! Note * パーティションテーブルを通じてパーティションが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、影響を受けるテーブルまたはパーティションのリストを常に複製する `TRUNCATE` コマンドです。

レプリケーションセットに個々のパーティションを追加することができます。その場合、通常のテーブルのように（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。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、プロバイダとサブスクライバの両方でパーティショニング定義が同じ場合、これにはパフォーマンス上の利点があります。

!!! Note * 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パートである場合、個々のパーティションのメンバーシップは無視され、そのルートテーブルのメンバーシップのみがアカウントされます。

2.17.3 外部キーの動作

外部キー制約により、参照元テーブルの行が被参照テーブルの行と一致することが保証されます。したがって、参照テーブルがレプリケーションセットのメンバである場合、被参照テーブルも同じレプリケーションセットのメンバである必要があります。

現在のバージョンの BDR は、この条件を自動的に確認または強制しません。したがって、テーブルをレプリケーションセットに追加するときに、外部キーを介して被参照されるすべてのテーブルも追加される make を確認するのは、データベース管理者の責任です。

次のクエリーを使用して、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。つまり、参照テーブルはレプリケーションセットのメンバですが、参照テーブルはそうではありません。

```
SELECT t1.relname,
       *      t1.nspname,
       *      fk.conname,
       *      t1.set_name
       * FROM bdr.tables AS t1
       * JOIN pg_catalog.pg_constraint AS fk
       *   ON fk.conrelid = t1.relid
       *   AND fk.contype = 'f'
WHERE NOT EXISTS (
  * SELECT *
  *   FROM bdr.tables AS t2
  *  WHERE t2.relid = fk.confrelid
  *    AND t2.set_name = t1.set_name
);
```

このクエリー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)
```

つまり、テーブル t2 はレプリケーションセット s2 のメンバですが、外部キー t2_x_fkey によって被参照されるテーブルはそうではありません。

!!! Note * TRUNCATE CASCADE コマンドは、コマンドを複製する前にレプリケーションセットのメンバーシップをアカウントします。

```
TRUNCATE table1 CASCADE;
```

これは、レプリケーションセットのみの一部であるすべてのテーブルでカスケードなしの TRUNCATE になります。

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

2.17.4 レプリケーションセットの管理

レプリケーションセットの管理。

`bdr.alter_node_replication_sets` を例外で、次の関数は 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が現在アクティブになっている場合は適用されます。[DDL Replication] 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。

このコマンドの複製は、DDL フィルタリング設定を DDL レプリケーション構成の影響を受けます。

あらすじ

```
bdr.create_replication_set(set_name name,
    *               replicate_insert boolean DEFAULT true,
    *               replicate_update boolean DEFAULT true,
    *               replicate_delete boolean DEFAULT true,
    *               replicate_truncate boolean DEFAULT true,
    *               autoadd_tables boolean DEFAULT false,
    *               autoadd_existing boolean DEFAULT true)
```

パラメーター

- **set_name**-新しいレプリケーションセットの名前。BDR 全体で一意である必要があります グループ
- **replicate_insert**-このテーブルに挿入するかどうかを示しますレプリケーションセットを複製する必要があります
- **replicate_update**-このテーブルの更新かどうかを示しますレプリケーションセットを複製する必要があります
- **replicate_delete**-この中のテーブルから削除するかどうかを示しますレプリケーションセットを複製する必要があります
- **replicate_truncate**-この中のテーブルの切り捨てかどうかを示しますレプリケーションセットを複製する必要があります
- **autoadd_tables**-新しく作成された（将来の）テーブルを作成するかどうかを示しますこのレプリケーションセットに追加されました
- **autoadd_existing**-すべての既存のユーザテーブルをこのレプリケーションセットに追加されました。これは、`autoadd_tables` が `true` に設定

注

デフォルトでは、新しいレプリケーションセットは DDL または BDR 管理関数呼び出しを複製しません。レプリケーションセットの DDL レプリケーションのセットアップ方法については、以下の `ddl filters` を参照してください。すべての DDL および `adminfunction` 呼び出しをレプリケートするデフォルトグループレプリケーションセット用の既存の DDL フィルターセットがあります。これは、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レプリケーションセットが `replicateDDL` または BDR に望ましくない場合はドロップできます管理ファンクション呼び出し。

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは `ddl filters` 構成の影響を受けます。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです-効果はトランザクションの `ROLLBACK` でロールバックでき、変更は現在のトランザクションに可視されます。

`bdr.alter_replication_set`

このファンクションは、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。

このコマンドの複製は、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。

あらすじ

```
bdr.alter_replication_set(set_name name,  
    *                replicate_insert boolean DEFAULT NULL,  
    *                replicate_update boolean DEFAULT NULL,  
    *                replicate_delete boolean DEFAULT NULL,  
    *                replicate_truncate boolean DEFAULT NULL,  
    *                autoadd_tables boolean DEFAULT NULL)
```

パラメーター

- `set_name`-既存のレプリケーションセットの名前
- `replicate_insert`-このテーブルに挿入するかどうかを示しますレプリケーションセットを複製する必要があります
- `replicate_update`-このテーブルの更新かどうかを示しますレプリケーションセットを複製する必要があります
- `replicate_delete`-この中のテーブルから削除するかどうかを示しますレプリケーションセットを複製する必要があります
- `replicate_truncate`-この中のテーブルの切り捨てかどうかを示しますレプリケーションセットを複製する必要があります

- `autoadd_tables`-新しく作成された（将来の）テーブルが必要かどうかを示しますこのレプリケーションセットに追加されました

NULL（デフォルト）に設定されたオプションは、以前と同じままです。

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは `ddl filters` 構成の影響を受けます。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです-トランザクションの `ROLLBACK` で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に可視されます。

`bdr.drop_replication_set`

このファンクションは、既存のレプリケーションセットを削除します。

このコマンドの複製は、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。

あらすじ

```
bdr.drop_replication_set(set_name name)
```

パラメーター

- `set_name`-既存のレプリケーションセットの名前

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは `ddl filters` 構成の影響を受けます。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです-トランザクションの `ROLLBACK` で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に可視されます。

!!! Warning * 少なくとも別のノードで使用されているレプリケーションセットを削除しないでください。削除すると、そのノードでレプリケーションが停止します。これが発生した場合は、影響を受けるノードをそのレプリケーションセットからサブスクライブ解除してください。同じ理由で、進行中の結合オペレーションがあり、結合されているノードがそのレプリケーションセットのメンバである場合、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対してローカルであり、グループに参加する前にノードで構成できるためです。

以下で説明する `alter_node_replication_sets` を使用して、ノードのレプリケーションセットサブスクリプションを管理できます。

`bdr.alter_node_replication_sets`

このファンクションは、ノードが発行して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。

あらすじ

```
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`-指定された場所に複製するレプリケーションセットの配列ノード; 空の配列では、グループのデフォルトレプリケーションセットが使用されます

注

このファンクションはローカルノードでのみ実行され、いかなる方法でも複製されません。

このファンクションは、ノードが参加する前に実行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の存在は**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するよう注意してください。

これにより、BDR グループのパートであ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ファンクションを呼び出して、結合中に同期されるデータを制限できます。ただし、レプリケーションセットの設定に関係なく、スキーマは**常に完全に同期**されていることにノートしてください。つまり、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。不要なテーブルは、`bdr.tables` カタログテーブルを参照して削除できます。これらは、BDR の後のバージョンで自動的に削除される場合があります。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。

これらのノードからこのファンクションが実行されるノードへの変更を実際に複製するために、この呼び出し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公開される必要があります。

2.17.5 レプリケーションセットメンバーシップ

テーブルは、1つまたはマルチプルのレプリケーションセットに追加および削除できます。これは、これらのテーブルの変更のレプリケーション（DML）にのみ影響し、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（以下の *DDL レプリケーションフィルタリング* 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の構成と組み合わせてレプリケーションセットのテーブルメンバーシップを使用して、どのアクションをどのノードにレプリケートするかを決定します。決定は、すべてのメンバーシップとレプリケーションセットオプションのユニオンを使用して行われます。つまり、テーブルが INSERT のみをレプリケートレプリケーションセット A のメンバーであり、UPDATE のみをレプリケーションセット B の場合、ターゲットノードも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すると、INSERT と UPDATE の両方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、レプリケーションセットにテーブルを追加します。

これにより、レプリケーションセットにテーブルが追加され、この時点から（またはトランザクションコミットではなく）変更のレプリケーションがスタートされます。テーブルにノード上にある既存のデータは同期されません。

このコマンドの複製は、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。

あらすじ

```
bdr.replication_set_add_table(relation regclass,
*                               set_name name DEFAULT NULL,
*                               columns text[] DEFAULT NULL,
*                               row_filter text DEFAULT NULL)
```

パラメーター

- **relation**-テーブルの名前または OID
- **set_name**-レプリケーションセットの名前。NULL（デフォルト）の場合、BDR グループのデフォルトレプリケーションセットが使用されます
- **columns**-将来の使用のために予約（現在は何も行わず、NULL でなければなりません）
- **row_filter**-複製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場合（つまり、NULL-デフォルト）、すべての行が送信されます

row_filter は、ブール値を生成する式を指定します。NULL を使用します。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。式にはサブクエリを含めることはできず、複製される現在の行の列以外の変数を参照することもできません。systemcolumns は被参照できません。

row_filter は、ターゲットノードではなく、オリジンノードで実行されます。これにより、この特定のテーブルのレプリケーションに追加の CPU オーバーヘッドがかかりますが、フィルター処理された行の

データ送信が完全に回避されるため、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。 `TRUNCATE` コマンドは、レプリケーションセットレベルでフィルタリングできます。先に。

テーブルの一部の列だけを複製することは可能です。 [edReplicating between nodes with differences](#)。

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは `ddl filters` 構成の影響を受けます。

ファンクションは、`row_filter` が `NULL` でない場合、レプリケーションセットに追加されているリレーションで `DML` グローバルロックを取得します。それ以外の場合は、`DDL` グローバルロックのみを取得します。

このファンクションはトランザクションです-効果はトランザクションの `ROLLBACK` でロールバックでき、変更は現在のトランザクションに可視されます。

`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。

このコマンドの複製は、`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。

あらすじ

```
bdr.replication_set_remove_table(relation regclass,  
                                *                               set_name name DEFAULT NULL)
```

パラメーター

- `relation`-テーブルの名前または `OID`
- `set_name`-レプリケーションセットの名前。 `NULL` (デフォルト) の場合、`BDR` グループのデフォルトレプリケーションセットが使用されます

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは `ddl filters` 構成の影響を受けます。

ファンクションは `DDL` グローバルロックを取得します。

このファンクションはトランザクションです-効果はトランザクションの `ROLLBACK` でロールバックでき、変更は現在のトランザクションに可視されます。

2.17.6 複製セットのリスト

既存のレプリケーションセットは、次のクエリーでリストできます。

```
SELECT set_name
FROM bdr.replication_sets;
```

このクエリーを使用して、特定の複製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';
```

上記のセクション**外部キーの動作**で、参照テーブルが被参照テーブルと同じレプリケーションセットに含まれていないすべての外部キーをリストするクエリをレポートします。

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 s.node_id,
*       s.node_name,
*       COALESCE(
*         i.pub_repsets, s.pub_repsets
*       ) AS pub_repsets,
*       COALESCE(
*         i.sub_repsets, s.sub_repsets
*       ) AS sub_repsets
FROM bdr.local_node_summary s
INNER JOIN bdr.node_local_info i ON i.node_id = s.node_id;
```

これにより、次のような出力が生成されます。

* node_id	node_name	pub_repsets	sub_repsets
1834550102	s01db01	{bdrglobal,bdrs01}	{bdrglobal,bdrs01}

(1 row)

クラスター内のすべてのノードに対して同じクエリーを実行し、どのレプリケーションセットがすべてのノードに同時に関連付けられているかを取得するには、次のクエリーを使用できます。

```
WITH node_repsets AS (
* SELECT jsonb_array_elements(
*   bdr.run_on_all_nodes($$
*     SELECT s.node_id,
*     s.node_name,
*     COALESCE(
*       i.pub_repsets, s.pub_repsets
*     ) AS pub_repsets,
```

(次のページに続く)

```

*      COALESCE(
*          i.sub_repsets, s.sub_repsets
*      ) AS sub_repsets
* FROM bdr.local_node_summary s
* INNER JOIN bdr.node_local_info i
* ON i.node_id = s.node_id;
*   $$)::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、表示されます：

* 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)

2.17.7 DDL レプリケーションフィルタリング

デフォルトでは、サポートされているすべての DDL のレプリケーションは、デフォルトの BDRgroup レプリケーションセットを介して行われます。これは、BDR グループと同じ名前の DDL フィルターが存在することで実現されます。これは、BDR グループの作成時にデフォルトの BDRgroup レプリケーションセットに自動的に追加されます。

上記は、すべての既存のレプリケーションセットの DDL レプリケーションフィルターを変更することで調整できます。これらのフィルターは、レプリケーションセットのテーブルメンバーシップに依存しません。データの変更と同様、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マルチプルのフィルターと一致する場合でも、1 回だけレプリケートされます。

次のクエリーを使用して、既存の DDL フィルターをリストできます。これは、コマンドタグとロール名前に適用される正規表現を各フィルターに表示します。

```
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、DDLreplication およびグローバルロックのサブジェクトになることに注意してください。

bdr.replication_set_add_ddl_filter

このファンクションは、レプリケーションセットに DDL フィルターを追加します。

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されているノードに複製されます。これは、BDR 管理機能のレプリケーションにも影響します。

これはどのノードでも DDL の実行を妨げるものではなく、DDL が他のノードに複製されるかどうかのみを変更することに注意してください。したがって、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場合、各ノードに直接接続し、そのノードで目的の DDL を実行することにより、インデックスコマンドを自由に実行できます。

DDL フィルターは、`command_tag` および `role_name` を指定して、一部の DDL ステートメントのみの複製を許可できます。`command_tag` は、通常の PostgreSQL コマンドで `EVENT TRIGGERS` が使用するものと同じです。典型的な例は、ロジカルスタンバイ上の追加のインデックスコマンドが他のすべてのノードに複製されるのを防ぐフィルターを作成することです。

BDR 管理関数の使用は、修飾されたファンクション名前にマッチングするタグ名を使用してフィルター処理できます（例、`bdr.replication_set_add_table` は同じ名前のファンクションのコマンドタグになります）。例、これにより、`bdr.*` を使用してすべての BDR 関数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マッチングに使用されます。`command_tag` と `role_name` は両方とも、大文字と小文字を区別する正規表現として評価されます。

あらすじ

```
bdr.replication_set_add_ddl_filter(set_name name,
*                               ddl_filter_name text,
*                               command_tag text,
*                               role_name text DEFAULT NULL)
```

パラメーター

- `set_name`-レプリケーションセットの名前。NULL の場合、BDR グループのデフォルトレプリケーションセットが使用されます
- `ddl_filter_name`-DDL フィルターの名前。これは全体で一意でなければなりません BDR グループ全体
- `command_tag`-マッチングコマンドタグの正規表現。NULL はすべてにマッチ
- `role_name`-マッチングロール名前の正規表現。NULL はすべての役割にマッチ

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは **ddl filters** 構成の影響を受けます。ノート-これは、ddlfilter 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を意味します！

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです-トランザクションの ROLLBACK で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に可視されます。

定義されているレプリケーションフィルターをビューするには、`bdr.ddl_replication` ビューを使用します。

例

BDR 管理機能のみを含めるには、次のようなフィルターを定義します。

```
SELECT bdr.replication_set_add_ddl_filter('mygroup', 'mygroup_admin', $$bdr\..*$$);
```

インデックス DDL 以外のすべてを除外するには：

```
SELECT bdr.replication_set_add_ddl_filter('mygroup', 'index_filter',
*      '^(?! (CREATE INDEX|DROP INDEX|ALTER INDEX)).*');
```

テーブルとインデックスに対するすべての操作を含め、他のすべてを除外するには、テーブル用とインデックス用の2つのフィルターを追加します。これは、複数のフィルターが許可されているすべての DDL コマンドのユニオンを提供することを示しています。

```
SELECT bdr.replication_set_add_ddl_filter('bdrgroup', 'index_filter', '^(?!INDEX).*$
→');
SELECT bdr.replication_set_add_ddl_filter('bdrgroup', 'table_filter', '^(?!TABLE).*$
→');
```

bdr.replication_set_remove_ddl_filter

このファンクションは、レプリケーションセットから DDL フィルターを削除します。

このコマンドの複製は、DDL フィルタリング設定自体を含む DDL レプリケーション構成の影響を受けます！

あらすじ

```
bdr.replication_set_remove_ddl_filter(set_name name,
*                                ddl_filter_name text)
```

パラメーター

- `set_name`-レプリケーションセットの名前。NULL の場合、BDR グループのデフォルトレプリケーションセットが使用されます
- `ddl_filter_name`-削除する DDL フィルターの名前

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが `ddl filters` 構成の影響を受けることを意味します。これは、DDL フィルタ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を意味することに注意してください。

ファンクションは DDL グローバルロックを取得します。

このファンクションはトランザクションです-トランザクションの ROLLBACK で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に可視されます。

2.18 Stream Triggers

BDR 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を導入します。

- 競合のトリガー
- トリガーの変換

これらのタイプのトリガーを合わせて、ストリームトリガーと呼びます。

!!! Note * 現在、この機能は EDB Postgres Extended および EDB Postgres Advanced でのみ利用可能です。

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おり、PostgreSQL BEFORE トリガーアーキテクチャを活用し、PostgreSQL BEFORE トリガーと同様のパフォーマンス特性を持つ可能性があります。

通常の PostgreSQL トリガーと同様に、1 つのトリガーファンクションをマルチプルトリガー定義で使用できます。トリガーファンクションは、このフォームで定義されたプログラムです: `CREATE FUNCTION ... RETURNS TRIGGER`。実際のトリガーを作成するには、`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が作成されます。

作成されると、トリガーはカタログテーブル `pg_trigger` に可視されます。ストリームトリガーは `tgisinternal = true` と `tgenabled = 'D'` としてマークされ、名前のサフィックス「`_bdrc`」または

「`_bdr`」が付きます。`visibleb_tran_4` は、テーブルにリレーションするトリガーに関する情報、実行されているプロシージャの名前、トリガーをトリガーするイベント、およびトリガータイプを提供します。

そのため、通常の SQL 処理ではストリーム SQL が有効にならないことに注意して名前。

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ることに注意してください。オリジンノードで実行オプションはありませんが、オリジンで `row_filter` 式の使用を検討することもできます。

また、`streamtrigger` の実行中に適用される DML は他の BDR ノードに複製されず、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あり、インスタンス、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、そのノード固有のクラッシュセーフテーブルにログするために使用できます。この章の最後に実用例を示します。

2.18.1 適用中のトリガー実行

トランスフォームトリガーは、トリガーテーブルの着信変更ごとに 1 回、最初に実行されます。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し、非常にレンジの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、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するそれ以上の処理はありません。

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てい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、存在する場合は、そのテーブルに存在する競合トリガーを呼び出します。

列の競合解決がありません

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マッチしようとしています。

入力行に存在するがターゲットテーブルには存在しない列は、`target_column_missing` 型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列が受信行に存在しない場合、`source_column_missing` の競合が発生します。これら 2 つの競合タイプのデフォルトの解像度は、それぞれ `ignore_if_null` と `use_default_value` です。

これは、ローリングスキーマアップグレードのコンテキストに関連しています。forinstance、スキーマの新しいバージョンが新しい列を導入する場合。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、ソース列が欠落しており、`use_default_valuestrategy` が適切です。これは、新しく導入された列にデフォルト値を設定するためです。

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レプリケーションする場合、列はアップグレードにありません。ignore_if_null リゾルバは、ユーザーがアップグレードされたノードのいずれか、新しい列に NULL 以外の値を持つタプル。

この例をビューして、ローリングスキーマアップグレードの適切な設定は、target_column_missing の競合の場合に ignore リゾルバを適用するように各ノードを構成することです。

これは、node1 を実際のノード名前に置き換えた後、各ノードで別々に実行する必要がある次のクエリーで行われます：

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

データ損失と発散リスク

このセクションでは、競合リゾルバを ignore に設定すると、データロスとクラスターの発散がどのように発生するかを示します。

次の例を考えてみましょう。テーブル t はノード 1 と 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。

コンフリクトリゾルバが ignore に設定されている場合、ノード 1 に c が null ではない行 (pk=1, col=100) など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、c 列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、既存のすべての行で最初に NULL に設定され、上記で考慮された行は (pk=1, col=NULL) になります：pk=1 を持つ行はすべてのノードで同一ではなくなり、したがってクラスターは分岐します。

デフォルトの ignore_if_null リゾルバは、このリスクの影響を受けないことに注意してください。ノード 2 に複製される行には col=NULL があるためです。

この例に基づいて、ignore リゾルバが使用されたローリングスキーマアップグレードの最後に、クラスター全体に対して LiveCompare を実行して、相違を検出して修正 make ことをお勧めします。

2.18.2 行タイプの用語

このドキュメントでは、次の行タイプを使用しています。

- SOURCE_OLD は、更新前の行、つまりキーです。
- SOURCE_NEW は、別のノードからの新しい行です。
- TARGET は、ノード上にすでに存在する行、つまり競合する行です。

2.18.3 競合のトリガー

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、競合が発生したときに何が起こるかを決定するために使用されます。

- トリガーファンクションが行を返す場合、アクションはターゲットに適用されます。
- トリガーファンクションが NULL 行を返す場合、アクションはスキップされます。

明確にするために、DELETE に対してトリガーが呼び出された場合、DELETE をスキップしたい場合、トリガーは NULL を返す必要があります。DELETE を続行する場合は、行の値を結果ます-SOURCE_OLD または TARGET のいずれかが機能します。競合するオペレーションが INSERT または UPDATE のいずれかであり、選択された解決が競合する行の削除である場合、トリガーは明示的に削除を実行して結果なければなりません NULL。トリガーファンクションは、選択に応じて他の SQL アクションを実行できますが、これらのアクションはレプリケートではなくローカルにのみ適用されます。

2 つ以上のノード間で実際のデータの競合が発生すると、同時に 2 つ以上の変更が発生します。これらの変更を適用すると、競合解決は各ノードで独立して発生します。これは、conflictresolution が各ノードで 1 回発生し、その間に大幅な時間差で発生する可能性があることを意味します。その結果、conflicttrigger のマルチプルの実行間で通信する可能性はありません。トリガーに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ようにすることは、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-複数の競合トリガーを単一のテーブルに指定できますが、それらは個別のイベントにマッチ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にのみマッチする必要があります。同じテーブルの同じイベントにマッチングする複数のトリガーは推奨されません。一貫性のない動作が発生する可能性があります、将来のリリースでは禁止されます。

同じ競合トリガーが複数のイベントに一致する場合、トリガー内で TG_OP 変数を使用して、競合を引き起こしたオペレーションを識別できます。

デフォルトでは、BDR 行のレプリケーション元の変更を監視することで競合を検出します。そのため、変更が 1 つしか発生していない場合でも競合トリガーを呼び出すことができます。この場合、実際の競合は存在しないため、この競合検出メカニズムは偽陽性の競合を生成する可能性があると言います。前述のように、競合トリガーはそれらすべてを同一にハンドルの必要があります。

場合によっては、タイムスタンプの競合の検出では競合がまったく検出されないことに注意してください。例、UPDATE の直後に DELETE が発生する同時 UPDATE / DELETE では、最初に UPDATE を表示し、次に DELETE を表示するノードでは競合は発生しません。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。同じシチュエーションですが、行バージョンの競合検出を使用すると、競合が表示され、競合トリガーによって処理できます。

トリガーファンクションは、オペレーションの種類に応じて、競合に関係するデータ行だけでなく、追加の状態情報にもアクセスできます。

- INSERT では、競合トリガーは SOURCE_NEW 行にアクセスできますソースおよび TARGET 行
- UPDATE では、競合トリガーは SOURCE_OLD およびソースからの SOURCE_NEW 行および TARGET 行
- DELETE では、競合トリガーは SOURCE_OLD 行にアクセスできますソースおよび TARGET 行

ファンクション `bdr.trigger_get_row()` は、そのオペレーションの値が存在する場合、`SOURCE_OLD`、`SOURCE_NEW`、または `TARGET` 行を取得するために使用できます。

競合トリガーの変更はトランザクション的に発生し、`ALTER TABLE` の一部のバリエーションの処理方法と同様に、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。

プライマリキーが競合トリガー内で更新されると、実行のタイミングの違いにより一意性制約違反エラーが発生することがあります。したがって、ユーザーは競合トリガー内でプライマリキーを更新しないでください。

2.18.4 変換トリガー

これらのトリガーは、特定のテーブルに対するデータストリームのすべての行に対して実行されることを除いて、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特定される前に実行されるため、`TARGET` 行はありません。

必要に応じて、BDR の各テーブルでマルチプルの変換トリガーを指定します。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で除去し、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、`NULL` に設定したりできます。戻り値は、さらに実行されるアクションを決定します。

- トリガーファンクションが行を返す場合、ターゲットに適用されます。
- トリガーファンクションが `NULL` 行を返す場合、それ以上のアクションはありません実行され、まだ実行されていないトリガーは実行されません。
- トリガーファンクションは、選択に応じて他のアクションを実行できます。

トリガーファンクションは、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。

ファンクション `bdr.trigger_get_row()` を使用して、`SOURCE_OLD` または `SOURCE_NEW` rows を取得できます。`TARGET` 行は使用できません。このタイプのトリガーは、ターゲット行が存在する場合、その行が識別される前に実行されるためです。

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。

- すべての着信変更に対して変換トリガーが呼び出されます。`BEFORE` トリガーは、`UPDATE` および `DELETE` の変更に対してまったく呼び出されませんテーブルにマッチング行が見つからない場合。
- 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、`SOURCE_OLD` を介してロックアップキーにアクセスできます。通常の SQL トリガーでは使用できません。

2.18.5 ストリームトリガー変数

競合トリガーと変換トリガーの両方は、トリガー API が提供する定義済み変数と 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できます。

PL/pgSQL には、次の定義済み変数が存在します。

TG_NAME

データ型名前; 実際にトリガーされたトリガーの名前を含む変数。実際のトリガー名前には、トリガーの作成時に指定された名前と比較して、「`_bdr`」または「`_bdr`」サフィックス（トリガータイプに応じて）が付けられます。

TG_WHEN

データ型テキスト。これは、競合トリガーと変換トリガーの両方に対して **BEFORE** と表示されます。ストリームトリガータイプは、`bdr.trigger_get_type()` information ファンクションを呼び出して取得できます（以下を参照）。

TG_LEVEL

データ型テキスト。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オペレーションを示す **INSERT**、**UPDATE** または **DELETE** の文字列。

TG_RELID

データ型 oid; トリガー呼び出しを引き起こしたテーブルのオブジェクト ID。

TG_TABLE_NAME

データ型名前; トリガー呼び出しを引き起こしたテーブルの名前。

TG_TABLE_SCHEMA

データ型名前; トリガー呼び出しを引き起こしたテーブルのスキーマの名前。テーブルパーティションの場合、これはルート表の名前です。

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ファンクションに与えられた引数の数。

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 より小さい以上) は、NULL 値になります。

2.18.6 情報関数

`bdr.trigger_get_row`

このファンクションは、識別子として `RECORD` として指定されたトリガー行の内容を返します。このファンクションは、不適切に呼び出された場合、つまり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。

あらすじ

```
bdr.trigger_get_row(row_id text)
```

パラメーター

- `row_id`-行の識別子。 `SOURCE_NEW`、`SOURCE_OLD`、および `TARGET`、トリガーのタイプとオペレーションに応じて (の文書を参照個々のトリガータイプ)。

bdr.trigger_get_committs

このファンクションは、識別子によって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。行が凍結されて行が利用できないため利用できない場合、これは NULL を結果ます。行識別子 SOURCE_OLD に対して常に NULL を返します。

あらすじ

```
bdr.trigger_get_committs(row_id text)
```

パラメーター

- `row_id`-行の識別子。SOURCE_NEW、SOURCE_OLD、および TARGET、トリガーのタイプとオペレーションに応じて（の文書を参照個々のトリガータイプ）。

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。行が凍結されて行が利用できないため利用できない場合、これは NULL を結果ます。SOURCE_OLD および SOURCE_NEW 行識別子に対しては常に NULL を返します。

これは、競合トリガーでのみ使用可能です。

あらすじ

```
bdr.trigger_get_xid(row_id text)
```

パラメーター

- `row_id`-行の識別子。SOURCE_NEW、SOURCE_OLD、および TARGET、トリガーのタイプとオペレーションに応じて（の文書を参照個々のトリガータイプ）。

bdr.trigger_get_type

このファンクションは、CONFLICT または TRANSFORM のいずれかの現在のトリガータイプを返します。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。

あらすじ

```
bdr.trigger_get_type()
```

bdr.trigger_get_conflict_type

このファンクションは、`conflicttrigger` 内で呼び出された場合は現在の競合タイプを返し、そうでない場合は `NULL` を返します。

このファンクションの可能な結果値については、[競合タイプ] (`conflicts.md` #競合タイプのリスト) を参照してください。

あらすじ

```
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 `triggerrow_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、行がローカルで生成されていることを意味します）、トリガー行引数に応じて、ソースノードまたはターゲットノードのノード ID を結果ます。行識別子 `SOURCE_OLD` に対して常に `NULL` を返します。これを使用して、常に信頼されたソースノードを優先する競合トリガーを定義できます。以下の例を参照してください。

あらすじ

```
bdr.trigger_get_origin_node_id(row_id text)
```

パラメーター

- `row_id`-行の識別子。 `SOURCE_NEW`、`SOURCE_OLD`、および `TARGET`、トリガーのタイプとオペレーションに応じて（の文書を参照個々のトリガータイプ）。

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外部キー情報を使用して FK の異常を回避します。

あらすじ

`bdr.ri_fkey_on_del_trigger()`

2.18.7 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 REPLICAIENTITY 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、targettable で UPDATE または DELETE を適用するときに行が見つかった場合にのみデータが含まれます。行が見つからない場合、TARGET は NULL になります。

2.18.8 トリガーノート

トリガーの実行オーダー：

- トリガーの変換-ターゲットの入力行ごとに 1 回実行
- 通常のトリガー-行ごとに 1 回実行
- 競合トリガー-競合が存在する行ごとに 1 回実行

2.18.9 ストリームトリガー操作インターフェイス

ストリームトリガーは、bdr-enterprise 拡張のパートとして提供される SQL インターフェイスを使用して管理されます。

ストリームトリガーは、REPLICA IDENTITY FULLable テーブルまたは TOASTable 列のないテーブルでのみ作成できます。

bdr.create_conflict_trigger

このファンクションは、新しい競合トリガーを作成します。

あらすじ

```
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」
- **relation**-このトリガーを起動するリレーション
- **function**-実行ファンクション
- **args**-オプション。トリガーファンクションが使用するパラメーターの配列を指定します実行時に受け取る (TG_ARGV 変数の内容)

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが [ddl filters](#) 構成の影響を受けることを意味します。

このファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

このファンクションはトランザクションです-トランザクションの ROLLBACK で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に可視されます。

通常の PostgreSQL トリガーと同様に、`bdr.create_conflict_trigger` 関数には、ファンクションの `relation` および `EXECUTE` privilege に対する `TRIGGER` 権限が必要です。これは、30619 以上の `abdr.backwards_compatibility` に適用されます。BDR では、競合トリガーを含むすべてのトリガーに追加のセキュリティルールが適用されます。 [security chapter on triggers](#) を参照してください。

bdr.create_transform_trigger

このファンクションは、新しい変換トリガーを作成します。

あらすじ

```
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」
- **relation**-このトリガーを起動するリレーション
- **function**-実行ファンクション
- **args**-オプション、トリガーファンクションが行うパラメーターの配列を指定実行時に受け取る (TG_ARGV 変数の内容)

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが **ddl filters** 構成の影響を受けることを意味します。

このファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

このファンクションはトランザクションです-トランザクションの ROLLBACK で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に可視されます。

通常の PostgreSQL トリガーと同様に、**bdr.create_transform_trigger** 関数には、ファンクションの **relation** および EXECUTEprivilege に対する TRIGGER 権限が必要です。トランスフォームトリガーを含むすべてのトリガーには、BDR で追加のセキュリティルールが適用されます。 [these security chapter on triggers](#) を参照してください。

bdr.drop_trigger

このファンクションは、既存のストリームトリガー（競合と変換の両方）を削除します。

あらすじ

```
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**-どのリレーションが定義されたトリガーか
- **ifexists**-true true に設定すると、このコマンドは欠落を無視します トリガー

注

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが **ddl filters** 構成の影響を受けることを意味します。

このファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

このファンクションはトランザクションです-トランザクションの ROLLBACK で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に可視されます。

bdr.drop_trigger ファンクションは、**relation** の所有者のみが実行できます。

2.18.10 ストリームトリガーの例

update_if_newerconflict リゾルバ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;
```

(次のページに続く)

(前のページからの続き)

```
END;
$$;
```

カウンタ列にデルタ変更を適用し、他のすべての列に SOURCE_NEW を使用する競合トリガー：

```
CREATE OR REPLACE FUNCTION delta_count_trg_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    * DELTA bigint;
    * SOURCE_OLD record;
    * SOURCE_NEW record;
    * TARGET record;
BEGIN
    * SOURCE_OLD := bdr.trigger_get_row('SOURCE_OLD');
    * SOURCE_NEW := bdr.trigger_get_row('SOURCE_NEW');
    * TARGET := bdr.trigger_get_row('TARGET');
    * DELTA := SOURCE_NEW.counter - SOURCE_OLD.counter;
    * SOURCE_NEW.counter = TARGET.counter + DELTA;
    * RETURN SOURCE_NEW;
END;
$$;
```

すべての変更を適用する代わりにログテーブルに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;
```

(次のページに続く)

(前のページからの続き)

```

    *      RETURN NULL; -- do not apply the change
END;
$$;

```

以下の例は、信頼できるサイト、優先ノード、または常に Winsresolution と呼ばれる、信頼できるソースの競合検出を実装する競合トリガーを示しています。これは、`bdr.trigger_get_origin_node_id ()` ファンクションを使用して、3つ以上のノードで機能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 $$
DECLARE
    SOURCE record;
    TARGET record;
    * TRUSTED_NODE    bigint;
    SOURCE_NODE       bigint;
    TARGET_NODE       bigint;
BEGIN
    TARGET := bdr.trigger_get_row('TARGET');
    IF (TG_OP = 'DELETE')
    * SOURCE := bdr.trigger_get_row('SOURCE_OLD');
    ELSE
    * SOURCE := bdr.trigger_get_row('SOURCE_NEW');
    END IF;
    * TRUSTED_NODE := current_setting('customer.trusted_node_id');
    * SOURCE_NODE := bdr.trigger_get_origin_node_id('SOURCE_NEW');
    TARGET_NODE := bdr.trigger_get_origin_node_id('TARGET');
    * IF (TRUSTED_NODE = SOURCE_NODE) THEN
    * RETURN SOURCE;
    ELIF (TRUSTED_NODE = TARGET_NODE) THEN
    * RETURN TARGET;
    ELSE
    * RETURN NULL; -- do not apply the change
    END IF;
END;
$$;

```

2.19 Backup and Recovery

この章では、BDR 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、分散された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 つまたは複数のノードが失われた場合、残りのノードから新しいノードを直接複製するためにそれらを置き換える最良の方法。

BDR でのバックアップとリカバリのロールは、次のような状況で DisasterRecovery (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
- マルチプルノード間での重大な修正不可能なデータ破損データ破損、アプリケーションエラー、またはセキュリティ違反

2.19.1 バックアップ

pg_dump

「ロジカルバックアップ」とも呼ばれる `pg_dump` は、通常 BDR で使用できます。

`pg_dump` は、ローカルシーケンスであるかのようにローカルシーケンスとグローバルシーケンスの両方をダンプすることに注意してください。これは、シーケンスをダンプして他の PostgreSQL データベースに移植できるようにするためのものです。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します。

復元後のスクリプトを作成して、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る場合、ダンプは `bdr.crdt_raw_value = on` でのみ再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では、BDR のバックアップとリカバリに物理的バックアップ技術を使用することを推奨しています。

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 ノードの物理的バックアップは、任意の PostgreSQL ノードに適用される同じ手順で実行できノードは、BDR ノードを実行する単なる PostgreSQL ノードです。

PostgreSQL バックアップテクニックを 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、物理的バックアップには、インスタンス内のすべてのデータベースが含まれます。あなたが計画する必要がありますデータベースを簡単にバックアップおよび復元できるようにします。

- バックアップでは、1つのノードのみのコピーが `make` されます。最も単純な場合、すべてのノードにはすべてのデータのコピーがあるため、バックアップのみが必要です。すべてのデータをキャプチャする1つのノード。ただし、BDR の目標はその単一のコピーを含むサイトがダウンした場合に会ったので、最小は、サイトごとに少なくとも1つのノードバックアップである必要があります（明らかに多くのコピーなど）。
- 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があります。レプリケーションセットの定義は複雑な場合があるため、すべてのノードがすべてのレプリケーションセットをサブスクライブしないでください。これらの場合、バックアップ計画には、複製されていないものをバックアップする方法の計画も含める必要があります。ローカルデータと、それぞれにサブスクライブする少なくとも1つのノードのバックアップレプリケーションセット。

最終的な一貫性

BDR クラスター内のノードは、**最後には一貫性を持つがありますが、完全に一貫性があります**。特定のノードの物理的バックアップは、そのノードが実際に想定する状態に限定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つのノードが同じ状態シーケンスを通過しない（および通常は通過しない）方法を示しています。

N1 と N2 の2つのノードを持つクラスターを考えてみましょう。初期状態は `S` です。トランザクション `W1` がノード `N1` に適用され、同時に競合しないトランザクション `W2` がノード `N2` に適用される場合、ノード `N1` は次の状態になります。

(N1) <code>S --> S + W1 --> S + W1 + W2</code>
--

…一方、ノード `N2` は次の状態を通過します。

(N2) <code>S --> S + W2 --> S + W1 + W2</code>
--

つまり、ノード `N1` は状態 `S + W2` を**決して**想定せず、ノード `N2` likewise は決して状態 `S + W1` を想定しません。両方のノードは同じ状態 `S + W1 + W2` になります。このシチュエーションを考慮すると、バックアップ戦略の決定方法に影響する場合があります。

ポイントインタイムリカバリ (PITR)

上記の例では、W1 と W2 の両方が T1 の時点で発生しますが、W1 の変更は N2 まで N2 に適用されないため、変更も時間的に一貫性がありません。

PostgreSQL PITR は、シングルマスタから COMMIT オーダーで到着する変更を前提に設計されています。したがって、PITR は、特定の特定時点 (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 つのノードをその視点 (T1 など) から単一の特定時点に復元できますが、その状態はその時間の近くにコミットしたが、ノードにまだ到着していない他のノードからの他のデータを含めないでください。その結果、リカバリは部分的に一貫していない、または少なくとも 1 つの複製オリジンのみで一貫していると見なされる場合があります。

これを要求するには、標準構文を使用します：

```
recovery_target_time = T1
```

BDR はマルチプルのマスターからの変更を許可し、すべてが 1 つのノードの WAL ログに記録され、レプリケーション識別子を使用して個別に識別されます。

BDR を使用すると、特定の時点までのすべてまたは一部のレプリケーション起点元の PITR が可能になり、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が提供されます。

したがって、マルチオリジンの場合、WAL ストリームには、すべてが 1 つの大きなストリームに混在した複数のストリームが含まれてビューと見なされます。まだ 1 つの PIT しかありませんが、各オリジンに対して別々のポイントとして別々に到達します。

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。その時点までのすべての変更を適用します。ただし、そのオリジンの 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ません。

WAL で 1 つの LSN 「停止ポイント」になりますが、「単一オリジン PITR」と同様に、1 つのシングルタイムスタンプも一貫して適用されます。

定義された PIT に到達すると、必要に応じて、リカバリ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、リカバリされたサーバーが昇格する場合は、まずシャットダウンして、`pg_resetwal` を使用して LSN をこのサーバーのタイムラインで使用されるよりも大きい LSN 値に転送します。これにより、重複する LSN が生成されないようにします。論理デコードによって。

上記の特定の例では、N1 は T1 に復元されますが、N1 が後まで適用されていないなくても、T1 によってコミットされた他のノードからの変更も含まれます。

マルチオリジン PITR を要求するには、リカバリ.conf ファイルで標準構文を使用します。

```
recovery_target_time = T1
```

T1 に復元されるレプリケーション起点元のリストは、新しいパラメータ `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、`recovery_target_origins` のリストとしてオリジンサブセットを指定できます。

```
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` で指定されたリストのターゲットがいつ達成されるかに応じて、リカバリは任意の時点で停止する場合があります。

`multi_recovery.conf` ファイルがない場合、リカバリはデフォルトで、シングルマスタから COMMIT オーダーで到着する変更を前提として設計されたオリジナルの PostgreSQL PITR 動作になります。

!!! Note * これは EDB Postgres Extended でのみ利用可能な機能であり、Barman は現在、`multi_recovery.conf` ファイルを自動的に作成しません。

2.19.2 復元

標準の PostgreSQL ノードと同じプロシージャで物理的バックアップを作成できますが、もう少し複雑なのは、BDR ノードの物理的バックアップを**復元**することです。

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

物理的バックアップを復元するための最も一般的なユースケースには、たとえばデータセンターの障害がイベントしたインスタンス、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。

このプロシージャを実行して、BDR クラスターの現在のコンテンツを複製し、QA または開発インスタンスをシードすることもできます。

その場合、単一の BDR ノードの物理的バックアップに基づいて BDR 機能を復元できます。オプションで WAL アーカイブをプラスできます。

- まだいくつかの BDR ノードが稼働している場合は、ホストを隔離します BDR ノードを復元したため、残りの 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乱させないようにします。
- いずれかの物理的バックアップから単一の PostgreSQL ノードを復元する BDR ノード。
- 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` およびリカバリ時に PostgreSQL をスタートして最新までリプレイします状態。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、スタンバイ中の場合は読み取り/書き込みにプロモートしますリカバリ残っているノードからフェンスで囲ってください！
- 物理的バックアップに含まれていた残りの BDR メタデータをクリーンアップし、以下に説明します。PostgreSQL インスタンスを完全に停止してリスタートします。
- に基づいた標準プロシージャでさらに BDR ノードを追加します `bdr.join_node_group()` ファンクション呼び出し。

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、次のように達成されます。

1. を使用して BDR ノードをドロップします。PostgreSQL を完全に停止して再起動します（重要！）。

複製元のクリーンアップ

複製元は、システムカタログに永続的に記録されるため、別の手順で明示的に削除する必要があります。したがって、バックアップおよび復元されたインスタンスに含まれます。これらは、依存関係として明示的に記録されないため、BDR 拡張機能を削除するときに自動的に削除されません。

BDR は、リモートマスタノードごとに 1 つのレプリケーションオリジンを作成し、着信レプリケーションの進行状況をクラッシュセーフ方法で追跡します。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）クラスター内の各ノードに対して 1 回。複製元は次のようにリストできます。

```
SELECT * FROM pg_replication_origin;
```

…および BDR によって作成されたものは、上記の例のように、名前によって簡単に認識されます。

複製スロットのクリーンアップ

物理的バックアップが `pg_basebackup` で作成された場合、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。

他のいくつかのバックアップ方法では、レプリケーションスロットが保持される場合があります。これは、古い状態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削除します。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、これはほとんど役に立ちません。

!!! Warning * ライブ BDR ノードでこれを実行しないでください。

2.20 Application Schema Upgrades

この章では、BDR クラスター上のソフトウェアのアップグレード処理と、アップグレード中のアプリケーションのダウンタイムを最小限に抑える方法について説明します。

2.20.1 概要

BDR クラスターには、基礎となる PostgreSQL ソフトウェアまたはそのフレーバーと PGLocal / BDR ソフトウェアの2つのソフトウェアセットがあります。これらのソフトウェアバージョンのいずれかまたは両方を、サポートされているメジャーリリースにアップグレードすることについて説明します。

BDR クラスターをアップグレードするには、各ノードで次の手順を実行する必要があります。

- アップグレードを計画する
- アップグレードの準備
- サーバソフトウェアのアップグレード Postgres をリスタートします
- アップグレードの確認と検証

2.20.2 アップグレード計画

BDR 3.7 リリースは PostgreSQL 11-13 をサポートしていますが、BDR 4.0 は PostgreSQL バージョン 12-14 をサポートしています。完全なリスト互換ソフトウェアについては、(product-matrix.md) ページを参照してください。BDR 4.0 は新しい PostgreSQL リリースをサポートしているため、BDR 3.7 から BDR 4.0 へのアップグレード処理中に、アプリケーションのダウンタイムを最小限に抑えるか、まったく停止せずに新しい PostgreSQL リリースをアップグレードすることもできます。

BDR バージョンをアップグレードするには、大きく2つの方法があります。

- 一度に1つのノードを新しい BDR バージョンにアップグレードします。
- 新しいバージョンの BDR ソフトウェアを実行している新しいノードに参加し、その後、オプションで古いノードの1つをドロップします。

BDR ソフトウェアのアップグレード処理のみに関心がある場合は、2つの方法のいずれかを使用できます。ただし、PostgreSQL バージョンもアップグレードする場合は、2番目のメソッドを使用する必要があります。

!!! Warning * 現在、最初のメソッドを使用して BDR 3.7 から BDR 4.0 にアップグレードすることはできません。3.7 を 4.0 にアップグレードする唯一の方法は、*Rolling Upgrade Using Node Join* セクションで説明されているように、4.0 ノードを BDR 3.7 クラスターに結合させることです。この制約は、BDR 4 の将来のバージョンで解除される可能性があります。

サーバーソフトウェアのローリングアップグレード

ローリングアップグレードとは、レプリケーションを機能させたまま、BDR グループの各ノードで次の *ServerSoftware Upgrade* を次々に実行するプロセスです。

4.0 へのアップグレードは、3.7 からのみサポートされ、特定の最小メンテナンスリリース (3.7.13.1 など) を使用します。実際に必要な最小バージョンについては、リリースノートを参照してください。そのため、ノードが古い 3.7 リリースで実行されている場合、最初に最低限必要なバージョンにアップグレードする必要があり、その後でのみ 4.0 にアップグレードできます。

単一ノードのデータベースと同様に、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、クラスター全体をリスタートすることができます。すべてのノードを同時にアップグレード処理この戦略は、混在 BDR 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ます。

アップグレードプロセス中に、アプリケーションを現在アップグレードされていないノードに切り替えて、アプリケーションの BDR グループの継続的な可用性を提供できます。

クラスターがローリングアップグレードを行っている間、BDR の混在バージョン間でレプリケーションが行われます。例、nodeA には BDR 3.7.11 があり、nodeB および nodeC には 4.0.0 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 3.7.11) のプロトコルと機能を使用するため、新しいバージョンで提供されるプロトコルの変更を必要とする新しい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、新しい機能が自動的に有効になります。

BDR クラスターは、簡単にアップグレードできるように設計されています。ほとんどの BDR リリースはローリングアップグレードをサポートしています。つまり、1 つのリリースレベルでクラスターのパートを実行し、2 番目の互換性のあるリリースレベルでクラスターの残りのパートを実行します。

ローリングアップグレードは、以前のリリースのすべてのノードを含むクラスターで開始され、その後、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処理します。問題が発生した場合は、技術サポートに連絡してオプションについて話し合っ提供することなく、ダウングレードを試みないでください。

ユーザ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判断した場合、アップグレードプロセスには拡張かかる場合がありますが、2 つのリリースレベルの共存期間を延長するための TechnicalSupport からの議論と明示的な同意なしに 30 日以上かかることはありません。

アップグレード中に問題が発生した場合、新しい異なるリリースレベルへの 2 回目のアップグレードを開始しないでください。通常の使用では、2 つのアップグレードが同時に発生することはありません。テクニカルサポートからの具体的かつ明示的な指示がない限り、ノードを 3 番目のリリースにアップグレードしないでください。これが発生する可能性があるのは、何らかの理由でアップグレードが失敗し、現在のクラスターアップグレードプロセスを継続して成功するためにホットフィックスが必要な場合です。BDR は 2 つ以上のリリースレベルで設計およびテストされていますが、特定の場合を除き、稼動環境での使用に依存することはできません。

ノード結合を使用したローリングアップグレード

基礎となる PostgreSQL メジャーバージョンのアップグレードの有無にかかわらず、BDR ソフトウェアをアップグレード処理もう 1 つのメソッドは、新しいノードをクラスターに結合させ、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 つを後でドロップすることです。このメソッドでも、すべてのノードが最終的に新しいバージョンにアップグレードされるまで、ソフトウェアの新しいバージョンでのみ使用可能な一部の機能は使用できない場合があります。

このリリースの BDR 4.0 を実行している新しいノードは 3.7 クラスターに結合でき、クラスター内の各ノードは最新の 3.7.x バージョンの BDR を実行しています。参加ノードは、サポートされている PostgreSQL バージョン 12~14 のいずれかを実行できますが、PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced の混合は現在サポートされていません。

すべてのノードが新しい同じリリースの PostgreSQL にアップグレードされるまで、新しい PostgreSQL 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、PostgreSQL の新しいリリースに追加された新しい DDL 構文に特に当てはまります。

`bdr_init_physical` はソースノードをバイト単位で作成することに注意してください。そのため、1 つのメジャー PostgreSQL バージョンから別のバージョンにアップグレード処理際には使用できません。実際、現在 `bdr_init_physical` では、ソースと結合ノードの EvenBDR バージョンが完全に同じである必要があります。Soit は、新しいノードメソッドに参加することによるローリングアップグレードには使用できません。そのような場合はすべて、ロジカル結合を使用する必要があります。

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 つが動作可能である必要があります。アップグレードでは、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ていないことを保証か、ローリングアップグレード戦略を使用して、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との間でノードが調整するのに十分な時間を与えることをお勧めします

CAMO ペアの構成が BDR3.7 と比較して大幅に変更されました。postgresql.conf の GUC の代わりに、ペアリングは BDR システムカタログ `bdr.camo_pairs` に保存されるようになりました。CAMO ペアを含む BDR クラスターを 3.7 から 4.0 にアップグレードするには、次の手順を実行する必要があります。

- `bdr.camo_partner_of` と `bdr.camo_origin_for` を削除しますすべてのノードの構成。
- この変更の影響を受けるすべてのノードを再起動します（既存の BDR バージョン）、1 つずつ。これにより、CAMO が一時的に無効になります `bdr.enable_camo` を設定してトランザクションを実行しようすると警告になります。
- すべてのノードを BDR に、または ローリングアップグレード。
- `bdr.add_camo_pair` を使用して CAMO を再構成します。このファンクションは BDR クラスター内のすべてのノードがアップグレードされた後にのみ機能します。

2.20.3 アップグレードの準備

BDR 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す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 の構成、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する場合があります。アップグレードの前に考慮し、場合によっては調整することをお勧めします。

pglogical

pglogical4 は存在せず、`shared_preload_libraries` を介して pglogical のいずれかのバージョンが Postgres の同じインスタンスにロードされた場合、BDR4 は機能しません。

ノード管理

`bdr.create_node_group()` ファンクションには多くの変更があります。

- サブグループを作成できるようになり、グループツリーが作成されます。BDR クラスター全体の構造。監視ビューが更新されました それに応じて。
- 非推奨のパラメーター `insert_to_update`、`update_to_insert`、`ignore_redundant_updates`、`check_full_tuple`、および `apply_delay` は削除されました。`insert_to_update` の代わりに `bdr.alter_node_set_conflict_resolver()` を使用します。`update_to_insert`。`check_full_tuple` はそのままでは不要ですテーブル競合検出構成に基づいて自動的に処理されます。

競合

競合の解決とロギングの設定は、新しいノードでデフォルトを使用するのではなく、結合ソースノードから新しく結合するノードにコピーされるようになりました。

一部の競合タイプのデフォルトの競合解決が変更されました。新しいデフォルトについては、(`conflicts.md` # `default-conflict-resolvers`) を参照してください。

競合ロギングインターフェイスは、`bdr.alter_node_add_log_config` および `bdr.alter_node_remove_log_config` から `bdr.alter_node_set_log_config` に変更されました。

デフォルトの競合ロギングテーブルの名前付けは `bdr.conflict_history` になり、古い `bdr.apply_log` は存在なくなりました。新しいテーブルは、BDR の Autopartition 機能を使用してパーティション化されます。

すべての競合がデフォルトでログファイルと競合テーブルの両方に記録されるようになりました。

廃止された関数 `bdr.row_version_tracking_enable()` と `bdr.row_version_tracking_disable()` は削除されました。代わりに、`edbdr.alter_table_conflict_detection()` を使用します。

競合ハンドリングの構成の一部は、`edpglogical` スキーマに格納されなくなりました。`pglogicaltables` を直接使用していた診断クエリは、`bdr` スキーマの適切なテーブルにスイッチ必要があります。

削除または名前変更された設定 (GUC)

代わりに `bdr.prefix` を使用するよう、接頭辞付きの `pglogical.` 構成変数はすべて名前が変更されました。

2.20.4 サーバソフトウェアのアップグレード

個々のノードでの BDR ソフトウェア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処理する場合、バックアップと復元には必要ありません。

!!! Warning * 現在、このメソッドは BDR 3.7 から 4.0 へのアップグレード処理には使用できません。3.7 を 4.0 にアップグレードする唯一の方法は、*Rolling Upgrade Using Node Join* セクションで説明されているように、4.0 ノードを BDR 3.7 クラスタに結合させることです。この制約は、BDR 4 の将来のバージョンで解除される可能性があります。

アップグレードの最初の手順は、新しいバージョンの BDR パッケージソフト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 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、使用するオペレーティングシステムによって異なります

Postgres を再起動する

バイナリおよび拡張スクリプトを自分自身でアップグレードしても、実行中の PostgreSQL インスタンスの BDR はアップグレードされません。そのためには、PostgreSQL インスタンスを再起動して、新しい BDR バイナリをロードできるようにする必要があります (BDR バイナリは PostgreSQL サーバーのスタートにロードされます)。その後、ノードがアップグレードされます。拡張 SQL アップグレードスクリプトは、必要に応じて自動的に実行されます。

!!! Warning * PostgreSQL インスタンスを再起動する前に `ALTER EXTENSION ... UPDATE` コマンドを実行しないことが重要です。これは、SQL に見える拡張機能をアップグレードするだけで、古いバイナリを保持するため、予期しない動作やクラッシュを引き起こす可能性があります。`ALTER EXTENSION ... UPDATE` コマンドは必要ありません。BDR4 は、必要に応じて SQL に見える拡張機能を自動的に維持します。

アップグレードの確認と検証

このプロシージャの後、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に必ず `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機能していることを確認しノード。

2.20.5 データベースのエンコード

すべての複製データベースで UTF-8 エンコーディングを使用することをお勧めします。BDR は、異なるエンコーディングのデータベース間のレプリケーションをサポートしていません。現在、`/alter` エンコーディングをアップグレードするためのサポートされているパスはありません。

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のアップグレードには 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リスタート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、これはいくつかのダウンタイムを課します。

このダウンタイムをなくすために、BDR は、次のセクションで説明されているように、ローリングアプリケーションスキーマのアップグレードを実行する方法を提供します。

2.20.6 ローリングアプリケーションスキーマのアップグレード

デフォルトでは、DDL はすべてのノードに自動的に送信されます。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。これは、ノード間でデータベーススキーマの違いを作成するために使用できます。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したり、レポートまたはテスト用に論理スタンバイノードを許可したりするように設計されています。

!!! Warning * アプリケーションスキーマのアップグレードは、BDR ではなくユーザが管理します。これを稼動クラスターで正しく機能させるには、慎重なスクリプト作成が必要になります。広範なテストが推奨されます。

詳細については、[hereReplicating between nodes with differences](#) をご覧ください。

1 つのノードが新しいテーブルを追加する DDL を実行する場合、まだ最新の DDL を受け取っていないノードは、リゾルバビューテーブルに対処する必要がありますスキーマ。 `atarget_table_missing` の競合の場合。これは、任意の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、永続的な設定を目的としています。

これは、`node1` を実際のノード名前に置き換えた後、各ノードで別々に実行する必要がある次のクエリーで行われます：

```
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 を実行する場合、まだ最新の DDL を受け取っていないノードは追加の列に対処する必要がありますビュー。 `target_column_missing` 競合のリゾルバ。これは、1 つのノードに追加の列が追加される前に実行する必要がある、永続的な設定を目的としています。

これは、`node1` を実際のノード名前に置き換えた後、各ノードで別々に実行する必要がある次のクエリーで行われます：

```
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 を実行するシチュエーション、最新の DDL をまだ受け取っていないノードは、不足している列に対処する必要がありリゾルバ。したがって、NULL を受け入れず、DEFAULT 値も持たない列には、2 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デフォルト値を追加します。列を削除します。

Constraints はローリング方式で削除できます。現時点では、一度に 1 ノードずつローリング方式でテーブル制約を追加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 を実行すると、現在の型とターゲット型の間のバイナリ強制力の存在に応じて、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、テーブルのアップグレードは制限されていますが、制御された DBA 環境では、すべてのノード（1 つずつ。詳細については、[ALTER TABLE](#) セクションを参照してください。

2.21 Explicit Two-Phase Commit (2PC)

アプリケーションは、BDR で明示的に二相コミットを使用することを選択できます。See [Distributed Transaction Processing: The XA Specification](#)。

X/Open 分散トランザクション処理 (DTP) モデルは、3 つのソフトウェアコンポーネントを想定しています。

- トランザクション境界を定義し、指定するアプリケーションプログラム (AP) トランザクションを構成するアクション。
- 提供するリソースマネージャー (RM、データベースやファイルアクセスシステムなど) 共有リソースへのアクセス。
- 識別子を割り当てるトランザクションマネージャ (TM) と呼ばれる別のコンポーネントトランザクションの進行状況を監視し、トランザクション完了および障害リカバリ用。

BDR は、PREPARE TRANSACTION および COMMIT PREPARED / ROLLBACK PREPARED コマンドを使用して、明示的な外部 2PC をサポートします。外部では、BDR クラスタは、単一セッションのトランザクションマネージャーに対して単一のリソースマネージャーのように見えます。

`bdr.commit_scope` が `local` の場合、トランザクションはローカルノードでのみ準備されます。コミットされると、変更が複製され、BDR はコミット後の競合解決を適用します。

`local` に設定された `bdr.commit_scope` の使用は、明示二相コミットでは無意味に思えるかもしれませんが、ユーザーがトランザクションのレイテンシと堅牢性の間のトレードオフを制御できるオプションが提供されます。

明示的な二相コミットは、CAMO またはグローバルコミットスコープと組み合わせて機能しません。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。

2.21.1 使用法

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て、CAMO を無効にしてください。

```
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スタートするには、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にすることができます。

```
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';
```

2.22 Catalogs and Views

このセクションには、BDR のアルファベットオーダーで使用されるシステムカタログとビューのリストが含まれています。

2.22.1 ユーザーに表示されるカタログと Views

bdr.conflict_history

このテーブルは、競合が記録されるデフォルトテーブル。テーブルは列 `local_time` でパーティション分割され、Autopartition によって管理されます。デフォルトのデータ保持期間は30日です。

このテーブルへのアクセスは、行単位セキュリティによって制限されている所有者が所有するテーブルのすべての競合を見る可能性のあるテーブル所有者であれば誰でも可能です。

詳細については、[Logging Conflicts to a Table](#) を参照してください。

bdr.conflict_history の列

Name	Type	Description
sub_id	oid	どのサブスクリプションがこの競合を引き起こしたか。`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_time	timestamp with time zone	紛争のローカルタイム
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（[競合の種類のリスト] を参照）
conflict_resolution	text	選択された競合解決（[競合解決のリスト] を参照）
conflict_index	regclass	競合するインデックス（以降インデックスが削除されなかった場合にのみ有効）
relid	oid	競合するリレーション（以降、インデックスが削除されなかった場合にのみ有効）
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（名前変更に従わない）
relname	text	競合時に競合が発生したリレーションの名前（名前変更に従わない）
key_tuple	json	行のマッチングに使用されるキーの json 表現形式
remote_tuple	json	競合する着信行の json 表現形式
local_tuple	json	ローカルの競合する行の json 表現形式
apply_tuple	json	結果の（適用された）行の json 表現形式
local_tuple_xmin	xid	ローカルの競合する行を生成したトランザクション（`local_tuple` が設定され、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（`local_tuple` が設定され、行がフリーズされていない場合）
local_tuple_commit_time	timestamp with time zone	ローカルの競合する行の最後の既知の変更タイムスタンプ（`local_tuple` が設定され、行がフリーズされていない場合）
		NA

bdr.conflict_history_summary

行の競合に関するユーザーが読み取り可能な詳細を含むビュー。

bdr.conflict_history_summary の列

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	紛争のローカルタイム
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
		NA

bdr.consensus_kv_data

`bdr.consensus_kv_store()` および `bdr.consensus_kv_fetch()` インターフェイスによって使用される内部 Raft ベースの KV ストアの永続ストレージ。

bdr.consensus_kv_data の列

Name	Type	Description
kv_key	text	一意のキー
kv_val	json	json フォーマットの任意の値
kv_create_ts	timestampz	最終書き込みタイムスタンプ
kv_ttl	int	ミリ秒単位の値で存続する時間
kv_expire_ts	timestampz	有効期限のタイムスタンプ (<i>kv_create_ts</i> + <i>kv_ttl</i>)
		NA

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、これは通常の運用状態で処理されたトランザクション（つまり、CAMO ペアの両方のノードが実行され接続されている）をカバーしません。このジャーナルのエントリが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、システムがまったく依存しない純粋な診断ツールです。

bdr.camo_decision_journal の列

Name	Type	Description
origin_node_id	oid	トランザクションが実行されたノードの OID
origin_xid	oid	リモートオリジンノードのトランザクション ID
decision	char	コミットの場合は「c」、アボートの場合は「a」
decision_ts	timestampz	決定時間
		NA

!!! Note * 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。

bdr.crdt_handlers

この表には、すべての CRDT データ型のマージ（「ハンドラ」）関数がリストされています。

bdr.crdt_handlers の列

Name	Type	Description
crdt_type_id	regtype	CRDT データタイプ ID
crdt_merge_id	regproc	このデータタイプのマージファンクション
		NA

!!! Note * 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。

`bdr.ddl_replication`

このビューには、現在の `DDL filters` によってセットアップされた DDL レプリケーション構成がリストされます。

`bdr.ddl_replication` の列

Name	Type	Description
<code>set_ddl_name</code>	name	DDL フィルターの名前
<code>set_ddl_tag</code>	text	適用するコマンドタグ（正規表現）
<code>set_ddl_role</code>	text	適用する役割（正規表現）
<code>set_name</code>	name	このフィルターが定義されているレプリケーションセットの名前
		NA

`bdr.depend`

このテーブルは、BDR カタログ内の内部オブジェクトの依存関係を追跡します。

`bdr.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 メッセージを記録します。

`bdr.global_consensus_response_journal` カタログに関しては、ペイロードはバイナリエンコードフォーマットで保存されます。これは `bdr.decode_message_payload()` ファンクションでデコードできます。詳細については、``bdr.global_consensus_journal_details`` <#bdr.global_consensus_journal_details>`__ビューを参照してください。

`bdr.global_consensus_journal` の列

Name	Type	Description
<code>log_index</code>	int8	ジャーナルエントリーの ID
<code>term</code>	int8	いかなだ用語
<code>origin</code>	oid	リクエストが発信されたノードの ID
<code>req_id</code>	int8	リクエストの ID
<code>req_payload</code>	bytea	リクエストのペイロード
<code>trace_context</code>	bytea	リクエストのトレースコンテキスト
		NA

bdr.global_consensus_journal_details

このビューには、`bdr.decode_message_payload()` ファンクションを使用してペイロードをデコードする、送信された Raft メッセージと対応する応答が表示されます。

bdr.global_consensus_journal_details の列

Name	Type	Description
log_index	int8	ジャーナルエントリーの ID
term	int8	いかだ用語
request_id	int8	リクエストの ID
origin_id	oid	リクエストが発信されたノードの ID
req_payload	bytea	リクエストのペイロード
origin_node_name	name	リクエストが発信されたノードの名前
message_type_no	oid	リクエストの BDR メッセージタイプの ID
message_type	text	リクエストの 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
response_message_type_no	oid	応答の BDR メッセージタイプの ID
response_message_type	text	応答の BDR メッセージタイプの名前
response_payload	text	応答の BDR メッセージペイロード
response_errcode_no	text	応答の SQLSTATE
response_errcode	text	応答のエラーコード
response_message	text	応答のエラーメッセージ
		NA

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 メッセージに対するすべての応答を収集します。

`bdr.global_consensus_journal` カタログに関しては、ペイロードはバイナリエンコードフォーマットで保存されます。これは、`bdr.decode_message_payload()` ファンクションでデコードできます。詳細については、``bdr.global_consensus_journal_details`` <`#bdr.global_consensus_journal_details``> ビューを参照してください。

bdr.global_consensus_response_journal の列

Name	Type	Description
log_index	int8	ジャーナルエントリーの ID
res_status	oid	応答のステータスコード
res_payload	bytea	応答のペイロード

次のページに続く

表 18 – 前のページからの続き

Name	Type	Description
trace_context	bytea	応答のトレースコンテキスト
		NA

bdr.global_lock

このカタログテーブルには、サーバーのリスタート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。

使用状況をモニタリングには、**bdr.global_lock** に表示される行が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わけではないため、オペレーターは可視ビューを優先する必要があります。

このテーブルの内容は変更しないでください。これは重要な BDR カタログです。

bdr.global_lock の列

Name	Type	Description
ddl_epoch	int8	ロックの DDL エPOCH
origin_node_id	oid	グローバルロックが発生したノードの OID
lock_type	oid	ロックのタイプ (DDL または DML)
nspname	name	ロックされたリレーションのスキーマ名前
relname	name	ロックされたリレーションのリレーション名前
groupid	oid	トップレベルグループの OID (アドバイザーロック用)
key1	integer	最初の 32 ビットキーまたはオーダー 32 ビットの 64 ビットキー (アドバイザーロックの場合)
key2	integer	2 番目の 32 ビットキー以上の 64 ビットキーの 32 ビット (オーダーリロック用)
key_is_bigint	boolean	64 ビット整数キーが使用されている場合は True (アドバイザーロックの場合)
		NA

bdr.global_locks

このノードのアクティブなグローバルロックを含むビュー。**bdr.global_locks** ビューは、BDR の共有メモリロック状態の追跡を公開し、管理者が BDR のグローバルロックアクティビティと進捗状況をより深く理解できるようにします。

グローバルロックの詳細については、[Monitoring Global Locks](#) を参照してください。

bdr.global_locks の列

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	ロックのタイプ (DDL または DML)
<i>relation</i>	text	ロックされたリレーション名前 (DML ロックの場合) またはキー (アドバイザリロックの場合)
<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機し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取得がオリジンノードによって開始された時間
<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
		NA

列の詳細：

- **relation**：DML ロックの場合、**relation** は DML とのリレーションを示しますロックが取得されます。グローバルアドバイザリロックの場合、**relation** 列は実際に表示されますロックが取得される 2 つの 32 ビット整数または 1 つの 64 ビット整数。
- **origin_node_id** および **origin_node_name**：これらがローカルと同じ場合ノードの ID と名前、ローカルノードはグローバル DDL のイニシエーターですロック、つまり、取得トランザクションを実行しているノードです。これらのフィールド別のノードを指定すると、ローカルノードは代わりにそのノードを取得しようとしますリモートノードからのグローバル DDL ロック要求を満たすローカル DDL ロック。
- **pid**：グローバル DDL ロックを要求したプロセスのプロセス ID ローカルノードが要求ノードであるなら。他のノードではヌル。クエリーロッカー **pid** を決定するオリジンノード。
- **global_lock_request_time**：グローバルロックリクエストが発生したタイムスタンプイニシエータは、グローバルロックを取得するプロセスを開始しました。null の場合があります現在のノードでは不明です。この時間は最初に刻印されます DDL ロック要求の、および DDL エポック管理にかかった時間を含む必要な保留中のレプリケーションキューのフラッシュ。現在のみオリジンノードで既知。
- **local_lock_request_time**：ローカルノードが開始したタイムスタンプこのグローバルロックのローカルロックを取得しようとしています。これにはヘビーウェイトセッションロックの取得にかかった時間。ただし、含まれません。DDL エポックまたはキューフラッシュにかかった時間。ロックが再取得された場合ローカルノードのリスタート後、これはノードのリスタート時間になります。
- **last_state_change_time**：のタイムスタンプこのグローバルロックの **bdr.global_locks.acquire_stage** フィールドの最終変更エントリー

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ます。

bdr.local_consensus_snapshot の列

Name	Type	Description
log_index	int8	ジャーナルエントリーの ID
log_term	int8	いかだ用語
snapshot	bytea	いかだ snapshot データ
		NA

bdr.local_consensus_state

このカタログテーブルは、ローカルノードの Raft の現在の状態を格納します。

bdr.local_consensus_state の列

Name	Type	Description
node_id	oid	ノードの ID
current_term	int8	いかだ用語
apply_index	int8	いかだ適用インデックス
voted_for	oid	今期このノードによる投票
last_known_leader	oid	最後に認識された Raft リーダーの node_id
		NA

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のローカルノードを識別します。

bdr.local_node の列

Name	Type	Description
node_id	oid	ノードの ID
pub_repsets	text[]	公開されたレプリケーションセット
sub_repsets	text[]	サブスクライブされたレプリケーションセット
		NA

bdr.local_node_summary

`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 の列

Name	Type	Description
sync_kind	char	どのような同期が行われた/行われたか
sync_subid	oid	同期を行うサブスクリプションの ID
sync_nspname	name	同期テーブルのスキーマ名前（存在する場合）
sync_relname	name	同期されたテーブルの名前（存在する場合）
sync_status	char	同期の現在の状態
sync_remote_relid	oid	アップストリームの同期テーブル（存在する場合）の ID
sync_end_lsn	pg_lsn	同期状態が最後に変化した位置
		NA

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情報を格納するカタログビュー。

bdr.network_path_info の列

Name	Type	Description
node_group_name	name	BDR グループの名前
node_region1	text	bdr.node_location からのノード領域名前
node_region2	text	bdr.node_location からのノード領域名前
node_location1	text	bdr.node_location からのノードロケーション名前
node_location2	text	bdr.node_location からのノードロケーション名前
network_cost	numeric	bdr.node_location からのノードロケーション名前
		NA

bdr.node

この表には、クラスター内のすべての BDR ノードがリストされています。

bdr.node の列

Name	Type	Description
node_id	oid	ノードの ID
node_name	name	ノードの名前
node_group_id	oid	ノードグループの ID
source_node_id	oid	ソースノードの ID
synchronize_structure	char	結合中に行われるスキーマの同期
node_state	oid	ノードの一貫した状態
target_state	oid	ノードが到達しようとしていることを示します (結合中またはプロモーション中)
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前
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るプロトコルバージョン範囲
		NA

bdr.node_catchup_info

このカタログ表は、結合またはパートプロシージャに関連する場合、各ノードの関連するキャッチアップ情報を記録します。

bdr.node_catchup_info の列

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	ノードが同等なからの直接リプレイにスイッチことができるノード 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
		NA

ノード (node_id) が別のノード (origin_node_id) からの欠落データをニーズとする場合、転送を介して既にノード (node_source_id) を持っているノードから取得できます。このテーブルのレコードは、ノード (node_id) がメンバになるまで保持されます。BDR クラスターの。

bdr.node_conflict_resolvers

すべての既知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 の列

Name	Type	Description
conflict_type	text	競合の種類
conflict_resolver	text	この競合タイプに使用されるリゾルバー
		NA

bdr.node_group

このカタログ表には、すべての BDR ノードグループがリストされます。

bdr.node_group の列

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_ext	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_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	サブスクライバーがプロバイダーからの変更を適用するまでの待機時間
node_group_check_constraints	bool	適用プロセスがデータを適用するときにコンストレインをチェックする必要があるかどうか
node_group_num_writers	int	このノードグループを支援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 29 – 前のページからの続き

Name	Type	Description
node_group_enable_wal_decoder	bool	グループに enable_wal_decoder が設定されているかどうか
node_group_streaming_mode	char	トランザクションストリーミング設定: 'O'-オフ、'F'-ファイル、'W'-ライター、'A'-自動、'D'-デフォルト
		NA

bdr.node_group_replication_sets

BDR グループ用に作成されたデフォルトのレプリケーションセットを示すビュー。bdr.replication_sets も参照してください。

bdr.node_group_replication_sets の列

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 によって複製されたアクション
def_repset_ext	name	デフォルトの*外部*レプセットの名前（通常は def_repset と同じ）
def_repset_ext_ops	text[]	デフォルトの* external * repset（通常は def_repset_ops と同じ）によって複製されるアクション
		NA

bdr.node_local_info

（ノードごとの構成のグローバルビューとは対照的に）ローカルノードに固有のノードごとの構成を格納するために使用されるカタログテーブル。

bdr.node_local_info の列

Name	Type	Description
node_id	oid	ノード（ローカルノードを含む）の OID
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

次のページに続く

表 31 – 前のページからの続き

Name	Type	Description
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)
		NA

bdr.node_location

ノードの場所に関するユーザー定義情報を保存するカタログビュー。

bdr.node_location の列

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	ユーザー指定の地域名前
node_location	text	ユーザー指定のロケーション名前
		NA

bdr.node_log_config

競合ロギングの構成に関する情報を格納するカタログビュー。

bdr.node_log_config の列

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であるか
log_conflict_type	NULL がすべてを意味する場合、ログに記録する競合タイプ
log_conflict_res	NULL がすべてを意味する場合、ログに記録する競合解決
	NA

bdr.node_peer_progress

レプリケーションストリーム内のすべてのノードの進行状況を追跡するために使用されるカタログ。クラスター内のすべてのノードは、その進行状況を定期的に他のすべてのノードにブロードキャストします（デフォルトは 60000 ms-1 分）。このリレーションの N * (N-1) 行を期待します。

監視目的の `bdr.node_slots` ビューに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 の列

Name	Type	Description
node_id	oid	この位置情報を報告した発信元ノードの OID
peer_node_id	oid	この位置情報が報告されたノードの同等な（リモートノード）の OID
last_update_sent_time	timestampz	発信元ノードからレポートが送信された時刻
last_update_recv_time	timestampz	ローカルサーバがレポートを受信した時刻
last_update_node_lsn	pg_lsn	レポート時点での発信元ノードの LSN
peer_position	pg_lsn	発信元ノードから見たノードの同等な最新の LSN
peer_replay_time	timestampz	レポートノードから見た同等な最新のリプレイ時間
last_update_horizon_xid	oid	内部解決期間：すべての下位 xid は、レポートノードで解決されたことがわかっています
last_update_horizon_lsn	pg_lsn	内部解決期間：レポートノードの LSN に関して同じ
		NA

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。CAMO パートナーが過去 15 分間にトランザクションに対して行った決定を示します。

bdr.node_pre_commit の列

Name	Type	Description
origin_node_id	oid	トランザクションが実行されたノードの OID
origin_xid	oid	リモートオリジンノードのトランザクション ID
decision	char	コミットの場合は「c」、アボートの場合は「a」
local_xid	xid	ローカルノードのトランザクション ID

次のページに続く

表 35 – 前のページからの続き

Name	Type	Description
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	決定時間
		NA

!!! Note * 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。

bdr.node_replication_rates

このビューには、指定された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ます

bdr.node_replication_rates の列

Column	Type	Description
peer_node_id	oid	この情報が報告されたノードの同等な（リモートノード）の OID
target_name	name	ターゲット同等なノードの名前
sent_lsn	pg_lsn	最新の送信位置
replay_lsn	pg_lsn	再生済みとして報告された最新の位置（可視）
replay_lag	interval	報告されたリプレイのおおよそのラグタイム
replay_lag_bytes	int8	replay_lsn とオリジンの現在の WAL 書き込み位置のバイト差
replay_lag_size	text	replay_lsn と現在の WAL 書き込み位置の人間が読み取れるバイト数の違い
apply_rate	bigint	同等なで 1 秒あたりに適用されるノード
catchup_interval	interval	同等ながまだ適用されていないすべての変更ノードのに必要なおおよその時間
		NA

!!! Note * 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 を使用することをお勧めします。

!!! Note * 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。

bdr.node_slots

このビューには、BDR が現在のデータベースで使用してい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、[Monitoring Outgoing Replication](#) を参照してください。

bdr.node_slots の列

Name	Type	Description
target_dbname	name	ターゲットノードのデータベース名前
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	オリジンノードの名前
target_name	name	ターゲットノードの名前
origin_id	oid	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 に従ったレプリケーションスロットの名前
slot_name	name	Postgres に従ったスロットの名前（上記と同じである必要があります）
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の場合は True（[* Group Replication Slot *]（nodes.md # Group Replication Slot）を参照）
is_decoder_slot	boolean	このスロットは Decoding Worker によって使用されていますか？
plugin	name	このスロットを使用する論理デコードプラグイン（pglogical_output または bdr である必要があります）
slot_type	text	スロットのタイプ（ロジカル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（接続が接続されていますか）
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リスタートできる LSN
confirmed_flush_lsn	pg_lsn	最新の確認済み複製位置
usesysid	oid	レプリケーションセッションが実行されているユーザの sysid
username	name	レプリケーションセッションが実行されているユーザのユーザ名
application_name	text	クライアント接続のアプリケーション名前（`synchronous_standby_names`で使用する）
client_addr	inet	クライアント接続の IP アドレス
client_hostname	text	クライアント接続のホスト名
client_port	int4	クライアント接続のポート
backend_start	timestampz	接続開始時

次のページに続く

表 37 – 前のページからの続き

Name	Type	Description
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、...）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	再生済みとして報告された最新の位置（可視）
write_lag	interval	報告された書き込みのおおよその遅延時間
flush_lag	interval	報告されたフラッシュのおおよその遅延時間
replay_lag	interval	報告されたリプレイのおおよそのラグタイム
sent_lag_bytes	int8	send_lsn と現在の WAL 書き込み位置のバイト差
write_lag_bytes	int8	write_lsn と現在の WAL 書き込み位置のバイト差
flush_lag_bytes	int8	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	send_lsn と現在の WAL 書き込み位置の人間が読み取れるバイト数の違い
write_lag_size	text	write_lsn と現在の WAL 書き込み位置の人間が読み取れるバイト数の違い
flush_lag_size	text	flush_lsn と現在の WAL 書き込み位置の人間が読み取れるバイト数の違い
replay_lag_size	text	replay_lsn と現在の WAL 書き込み位置の人間が読み取れるバイト数の違い
		NA

!!! Note * `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 `replay_lag_size` を使用することをお勧めします。

`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` の列

Name	Type	Description
node_name	name	ノードの名前
node_group_name	name	ノードが含まれる BDR グループの名前
interface_connstr	text	ノードへの接続文字列
peer_state_name	text	人間が読み取れるフォームのノードの一貫した状態

次のページに続く

表 38 – 前のページからの続き

Name	Type	Description
peer_target_state_name	text	ノードが到達しようとしている状態（結合中またはプロモーション中）
node_seq_id	int4	一意のシーケンス番号を生成するために使用されるノードのシーケンス識別子
node_local_dbname	name	ノードのデータベース名前
set_repl_ops	text	デフォルトのレプリケーションセットはどの操作をレプリケートしますか
node_id	oid	ノードの OID
node_group_id	oid	BDR ノードグループの OID
		NA

bdr.queue

このテーブルには、複製された DDL ステートメントの履歴レコードが格納されます。

bdr.queue の列

Name	Type	Description
queued_at	timestampz	ステートメントがキューに入れられたのはいつですか
role	name	ステートメントを実行したロール
replication_sets	text[]	どのレプリケーションセットに公開されたステートメントでしたか
message_type	char	メッセージの種類
message	json	ステートメントのレプリケーションに必要なメッセージのペイロード
		NA

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。ユーザクエリの代わりに、**bdr.replication_sets** ビューを確認することをお勧めします。

bdr.replication_set の列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_nodeid	oid	ノードの Oid（現在は常にローカルノード oid）
set_name	name	レプリケーションセットの名前
replicate_insert	boolean	レプリケーションセットが INSERT をレプリケートするかどうかを示します
replicate_update	boolean	レプリケーションセットが UPDATE をレプリケートするかどうかを示します
replicate_delete	boolean	レプリケーションセットが DELETE をレプリケートするかどうかを示します
replicate_truncate	boolean	レプリケーションセットが TRUNCATE をレプリケートするかどうかを示します
set_isinternal	boolean	予約済み
set_autoadd_tables	boolean	新しいテーブルがこのレプリケーションセットに自動的に追加されるかどうかを示します
set_autoadd_seqs	boolean	新しいシーケンスがこのレプリケーションセットに自動的に追加されるかどうかを示します
		NA

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。代わりに、ユーザクエリの `bdr.tables` ビューを確認することをお勧めします。

bdr.replication_set_table の列

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	コンパイルされ行フィルタリング式
		NA

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。代わりに、ユーザクエリの `bdr.ddl_replication` ビューを確認することをお勧めします。

bdr.replication_set_ddl の列

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
		NA

bdr.replication_sets

ノードによって現在使用されていない場合でも、BDR グループで定義されたレプリケーションセットを示すビュー。

bdr.replication_sets の列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_name	name	レプリケーションセットの名前
replicate_insert	boolean	レプリケーションセットが INSERT をレプリケートするかどうかを示します
replicate_update	boolean	レプリケーションセットが UPDATE をレプリケートするかどうかを示します
replicate_delete	boolean	レプリケーションセットが DELETE をレプリケートするかどうかを示します
replicate_truncate	boolean	レプリケーションセットが TRUNCATE をレプリケートするかどうかを示します
set_autoadd_tables	boolean	新しいテーブルがこのレプリケーションセットに自動的に追加されるかどうかを示します
set_autoadd_seqs	boolean	新しいシーケンスがこのレプリケーションセットに自動的に追加されるかどうかを示します
		NA

bdr.schema_changes

BDR 内のスキーマに対するすべての変更を表示するシンプルビュー。

bdr.schema_changes の列

Name	Type	Description
schema_changes_ts	timestampstz	トリガーの ID
schema_changes_change	char	変更タイプのフラグ
schema_changes_classid	oid	クラス ID
schema_changes_objectid	oid	オブジェクト ID
schema_changes_subid	smallint	サブスクリプション
schema_changes_descr	text	オブジェクトが変更されました
schema_changes_addnames	text[]	スキーマ変更の場所
		NA

bdr.sequence_alloc

galloc シーケンスのアロケーションの詳細を表示するビュー。

bdr.sequence_alloc の列

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	NA
		NA
seq_nallocs	bigint	NA
		NA
seq_last_alloc	timestampstz	最後に割り当てられたシーケンス
		NA

bdr.schema_changes

BDR 内のスキーマに対するすべての変更を表示するシンプルビュー。

bdr.schema_changes の列

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_addnames	text[]	スキーマ変更の場所
		NA

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 の列

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	NA
		NA
seq_nallocs	bigint	NA
		NA
seq_last_alloc	timestampz	最後に割り当てられたシーケンス
		NA

bdr.sequences

このビューには、内部 BDR 簿記用のシーケンスを除く、すべてのシーケンスとその種類がリストされます。

bdr.sequences の列

Name	Type	Description
nspname	name	シーケンスを含む名前空間
relname	name	シーケンスの名前
seqkind	text	シーケンスのタイプ ('ローカル'、'timeshard'、'gal-loc')
		NA

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、待機が BDR に関連する場合に wait_event が正しく設定されることを除いて、pg_stat_activity と同じ情報が含まれます。

bdr.stat_relation

各リレーションに統計を適用します。追跡が有効で、特定のリレーションに対して何かが複製された場合にのみデータが含まれます。

bdr.stat_relation の列

Column	Type	Description
nspname	name	リレーションのスキーマの名前
relname	name	リレーションの名前
relid	oid	リレーションの Oid
total_time	double precision	リレーションのレプリケーションの処理に費やされた合計時間
ninsert	bigint	リレーションに対して複製された挿入の数
nupdate	bigint	リレーションに対して複製された更新の数
ndelete	bigint	リレーションに対して複製された削除の数
ntruncate	bigint	リレーションに対して複製された切り捨ての数

次のページに続く

表 49 – 前のページからの続き

Column	Type	Description
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総数
shared_blks_read	bigint	リレーションに対して読み取られた共有ブロックの総数
shared_blks_dirtied	bigint	リレーションでダーティになった共有ブロックの総数
shared_blks_written	bigint	リレーション用書き込まれた共有ブロックの総数
blk_read_time	double precision	リレーションのブロックの読み取りに費やされた合計時間（ミリ秒単位）（`track_io_timing` が有効な場合、それ以外の場合はゼロ）
blk_write_time	double precision	リレーションのブロックの書き込みに費やされた合計時間（ミリ秒単位）（`track_io_timing` が有効な場合、それ以外の場合はゼロ）
lock_acquire_time	double precision	リレーションのロックの取得に費やされた合計時間（ミリ秒単位）（`bdr.track_apply_lock_timing` が有効な場合、それ以外の場合はゼロ）
		NA

bdr.stat_subscription

各サブスクリプションに統計を適用します。追跡が有効な場合にのみデータが含まれます。

bdr.stat_subscription の列

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数
nabort	bigint	ライタがこのサブスクリプションに対して行った中止の数
nerror	bigint	このサブスクリプションでライタがヒットしたエラーの数
nskippedtx	bigint	このサブスクリプションでライタがスキップされたしたトランザクションの数（`skip_transaction` 競合リゾルバが原因）
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

次のページに続く

表 50 – 前のページからの続き

Column	Type	Description
ndeleter	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	デッドロックによって発生したエラーの数
nretries	bigint	ライタが行った再試行回数（完全なリスタート/再接続を行わない場合）
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	中止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 STREATSTART メッセージの数
nstream_stop	bigint	処理された STREAMSTOP 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ヒットの総数
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総数
shared_blks_dirtied	bigint	サブスクリプションによってダーティされた共有ブロックの総数
shared_blks_written	bigint	サブスクリプションによって書き込まれた共有ブロックの総数
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）（`track_io_timing` が有効な場合、それ以外の場合はゼロ）
blk_write_time	double precision	サブスクリプションがブロックの書き込みに費やした合計時間（ミリ秒単位）（`track_io_timing` が有効な場合、それ以外の場合はゼロ）
connect_time	timestamp with time zone	現在のアップストリーム接続が確立された時刻。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切断された時刻
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スタートを要求した LSN
retries_at_same_lsn	bigint	同じ LSN 値からサブスクリプションが再起動された試行回数
curr_ncommit	bigint	現在の接続が確立された後にこのサブスクリプションが行ったコミットの数
		NA

bdr.subscription

このカタログ表には、ローカル BDRnode が所有するすべてのサブスクリプションと、それらのモードがリストされます。

bdr.subscription の列

Name	Type	Description
sub_id	oid	サブスクリプションの ID
sub_name	name	サブスクリプションの名前
nodegroup_id	oid	ノードグループの ID
origin_node_id	oid	オリジンノードの ID
source_node_id	oid	ソースノードの ID
target_node_id	oid	ターゲットノードの ID
subscription_mode	char	サブスクリプションのモード
sub_enabled	bool	サブスクリプションが有効かどうか（レプリケーションである必要があります）
apply_delay	interval	このサブスクリプションへの変更の適用はどのくらい遅れる必要がありますか（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複製された複製セット（NULL = すべて）
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 = すべて）
		NA

bdr.subscription_summary

このビューには、ローカルノードから他のノードへのすべての 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 の列

Name	Type	Description
node_group_name	name	ノードが含まれる 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	オリジンノードの名前
target_name	name	ターゲットノードの名前（通常はローカルノード）
sub_enabled	bool	サブスクリプションは有効になっていますか
sub_slot_name	name	このサブスクリプションで使用されるオリジンノードのスロット名前
sub_replication_sets	text[]	サブスクライブされたレプリケーションセット
sub_forward_origins	text[]	サブスクリプションは、オリジン以外の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トランザクションをこれだけ遅らせる
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前
bdr_subscription_mode	char	サブスクリプションモード
subscription_status	text	サブスクリプションワーカーのステータス
node_group_id	oid	ノードが含まれる 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（再起動した場合は逆方向に進む可能性があります）
receive_commit_lsn	pg_lsn	最後に受信した COMMIT の最新の LSN（再起動の場合、これは逆方向に進む可能性があります）
last_xact_replay_lsn	pg_lsn	このサブスクリプションで再生された最後のトランザクションの LSN
last_xact_flush_lsn	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 LSN は、永続的にディスクにフラッシュされます
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
		NA

bdr.replication_status

このビューには、ローカルノードと BDR クラスター内の他のすべてのノード間の受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 回再開され、現在のアップストリーム接続が確立された後に単一のトランザクションがまだ適用されていない場合、レプリケーションがブロックされると考えます。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_blockedresult` は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、そのアップストリームノードのステータスのみが表示されます。同様に、サブスクライバのみの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、レプリケーションのステータスは表示されません。

bdr.replication_status の列

Column	Type	Description
<code>node_id</code>	oid	ローカルノードの OID
<code>node_name</code>	name	ローカルノードの名前
<code>origin_node_id</code>	oid	オリジンノードの OID
<code>origin_node_name</code>	name	オリジンノードの名前
<code>sub_id</code>	oid	このオリジンノードのサブスクリプションの OID
<code>sub_name</code>	name	このオリジンノードのサブスクリプションの名前
<code>connected</code>	boolean	このノードはオリジンノードに接続されていますか？
<code>replication_blocked</code>	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？
<code>connect_time</code>	timestamp with time zone	現在の接続が確立された時刻
<code>disconnect_time</code>	timestamp with time zone	最後の接続が切断された時刻
<code>uptime</code>	interval	現在の接続がこのオリジンでアクティブになってからの期間
		NA

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一覧表示されます。テーブルがマルチプルのレプリケーションセットに存在する場合、このテーブルにマルチプル表示されます。

bdr.tables の列

Name	Type	Description
relid	oid	リレーションの OID
nspname	name	スキーマリレーションの名前は
relname	name	リレーションの名前
set_name	name	レプリケーションセットの名前
set_ops	text[]	複製された操作のリスト
rel_columns	text[]	複製された列のリスト (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メソッド: row_origin (デフォルト)、row_version または column_level
		NA

(*) これらの列は将来の使用のために予約れており、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確認できます。多くの場合、トリガーは `bdr.create_conflict_trigger` から作成されます。

bdr.trigger の列

Name	Type	Description
trigger_id	oid	トリガーの ID
trigger_reloid	regclass	関連ファクションの名前
trigger_pgtgid	oid	Postgres トリガー ID
trigger_type	char	トリガーコールの種類
trigger_name	name	トリガーの名前
		NA

bdr.triggers

より読みやすい列を含む `bdr.trigger` の展開ビュー。

Name	Type	Description
trigger_name	name	トリガーの名前
event_manipulation	text	オペレーション (s)
trigger_type	bdr.trigger_type	トリガーの種類
trigger_table	bdr.trigger_reloid	それを呼び出すテーブル

[次のページに続く](#)

表 56 – 前のページからの続き

Name	Type	Description
trigger_function	name	使用したファンクション
		NA

bdr.workers

BDR ワーカープロセスの実行に関する情報。

これを pid を使用して `bdr.stat_activity` と結合すると、BDR ワーカーの状態をさらに洞察することができます。

bdr.workers の列

Name	Type	Description
worker_pid	int	ワーカープロセスのプロセス ID
worker_role	int	ワーカーのロールの数値表現形式
worker_role_name	text	ワーカーロールの名前
worker_subid	oid	ワーカーが 1 つに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タイムスタンプ（存在する場合）
worker_local_timestamp	timestampz	上記のコミットが処理された現地時間（ある場合）
		NA

bdr.worker_errors

BDR バックグラウンドワーカープロセスからのエラーの永続的なログ。

bdr.worker_errors の列

Name	Type	Description
node_group_name	name	BDR グループの名前
origin_name	name	オリジンノードの名前
source_name	name	NA
		NA
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、2:受信、3:ライタ、4:出力、5:拡張)

次のページに続く

表 58 – 前のページからの続き

Name	Type	Description
worker_role_name	text	ロール名前
worker_pid	int4	エラーの原因となっているワーカーのプロセス ID
error_time	timestampz	エラーの日時
error_age	interval	エラーからの期間
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
subwriter_id	oid	NA
		NA
subwriter_name	name	NA
		NA
		NA

bdr.writers

BDR ライタプロセスに関する特定の情報。

bdr.writers の列

Name	Type	Description
sub_name	name	サブスクリプションの名前
pid	int	ワーカープロセスのプロセス ID
syncing_rel	int	同期されているリレーションの Oid（存在する場合）
streaming_allowed	text	このライタを直接ライタストリーミングのターゲットにすることはできますか
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（存在する場合）
remote_commit_lsn	pg_lsn	最後に処理されたコミットの LSN
commit_queue_position	int	内部コミットキュー内の位置
commit_timestamp	timestampz	最後に処理されたコミットのタイムスタンプ
		NA

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、バックグラウンドワーカー起動および登録アクティビティに関するいくつかの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および BDR ノードに固有ではありません。現在の PostgreSQL インスタンス上のすべての 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 の列

Column	Type	Description
task_key_worker_role	integer	ワーカーロール識別子
task_key_worker_role_name	text	ワーカーロール名前
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（おそらく bdr）
task_key_ext_funcname	name	ファンクションエントリポイントの名前
task_key_ext_workername	name	ワーカーに割り当てられた名前
task_key_remoterelid	oid	可能な場合、リモート同期リレーションの識別子
task_pid	integer	ワーカーのプロセス ID
task_registered	timestamp with time zone	労働者登録タイムスタンプ
since_registered	interval	ワーカーが登録してからの間隔
task_attached	timestamp with time zone	ワーカー添付タイムスタンプ
since_attached	interval	ワーカーがアタッチしてからの間隔
task_exited	timestamp with time zone	ワーカー終了タイムスタンプ
since_exited	interval	ワーカーが終了してからの間隔
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_requestor_pid	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

[次のページに続く](#)

表 60 – 前のページからの続き

Column	Type	Description
since_last_request	interval	最後のリクエストからの間隔
task_last_launch_request_approved	boolean	最後のリクエストは成功しましたか？
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	以前に登録されたタスクのタイムスタンプ
since_prev_registered	interval	前回の登録からの間隔
task_prev_launched	timestamp with time zone	前回の発売のタイムスタンプ
since_prev_launched	interval	前回の起動からの間隔
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	最初の登録が行われたときのタイムスタンプ
since_first_registered	interval	最初の登録からの間隔
		NA

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ます。作業項目は 1 つのノードでのみ作成され、異なるノードで処理されます。

bdr.autopartition_work_queue の列

Column	Type	Description
ap_wq_workid	bigint	作業アイテムの一意の ID
ap_wq_ruleid	int	autopartition_rules にリストされているルール ID。ルールは bdr.autopartition コマンドを使用して指定されます
ap_wq_relname	name	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	このワークアイテムのルールで指定されたテーブルスペースの名前。
ap_wq_partname	name	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）のいずれかになります。

次のページに続く

表 61 – 前のページからの続き

Column	Type	Description
ap_wq_work_sql	text	作業アイテムの SQL クエリー
ap_wq_work_depends	Oid[]	作業アイテムが依存するノードの Oid
		NA

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 の列

Column	Type	Description
ap_wi_workid	bigint	作業アイテムの ID
ap_wi_nodeid	Oid	作業アイテムが処理されているノードの Oid
ap_wi_status	char	ステータスは、「q」（キュー）、「c」（完了）、「f」（失敗）、「u」（不明）のいずれかになります。
ap_wi_started_at	timestamptz	作業アイテムのスタート timestamptz
ap_wi_finished_at	timestamptz	作業アイテムの終了 timestamptz
		NA

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ます。これは bdr.autopartition_work_queue と似ていますが、これらの作業項目はローカルで管理されるテーブル用です。各ノードは、クラスター内の他のノードとは無関係に、独自のローカル作業項目を作成および処理します。

bdr.autopartition_local_work_queue の列

Column	Type	Description
ap_wq_workid	bigint	作業アイテムの一意の ID
ap_wq_ruleid	int	autopartition_rules にリストされているルール ID。ルールは bdr.autopartition コマンドを使用して指定されます
ap_wq_relname	name	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	このワークアイテムのルールで指定されたテーブルスペースの名前。
ap_wq_partname	name	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

次のページに続く

表 63 – 前のページからの続き

Column	Type	Description
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）のいずれかになります。
ap_wq_work_sql	text	作業アイテムの SQL クエリー
ap_wq_work_depends	Oid[]	常に NULL
		NA

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 の列

Column	Type	Description
ap_wi_workid	bigint	作業アイテムの ID
ap_wi_nodeid	Oid	作業アイテムが処理されているノードの Oid
ap_wi_status	char	ステータスは、「q」（キュー）、「c」（完了）、「f」（失敗）、「u」（不明）のいずれかになります。
ap_wi_started_at	timestamptz	作業アイテムのスタート timestamptz
ap_wi_finished_at	timestamptz	作業アイテムの終了 timestamptz
		NA

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 の列

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前
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 CAMO トランザクションはありますか
apply_lsn	text	再生済みとして報告された最新の位置（可視）

次のページに続く

表 65 – 前のページからの続き

Name	Type	Description
receive_lsn	text	受信した変更またはメッセージの最新の LSN（再起動した場合は逆方向に移動できます）
apply_queue_size	text	apply_lsn と receive_lsn のバイト差
		NA

!!! Note * 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。

bdr.camo_pairs

すべてのクラスターで構成されたすべての CAMO ペアに関する情報。

bdr.camo_pairs の列

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
		NA

!!! Note * 名前 left および right 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互いに CAMO パートナーです。

bdr.group_raft_details

bdr.run_on_all_nodes を使用して、すべてのノードから Raft Consensus ステータスを収集します。

bdr.group_raft_details の列

Name	Type	Description
node_id	oid	内部ノード ID
node_name	name	ノードの名前
state	text	ノードのいかなるワーカーの状態
leader_id	oid	RAFT_LEADER のノード ID
current_term	int	いかなる選挙内部 ID
commit_index	int	いかなる snapshot 内部 ID
nodes	int	アクセス可能なノードの数
voting_nodes	int	投票するノードの数
protocol_version	int	このノードのプロトコルバージョン

次のページに続く

表 67 – 前のページからの続き

Name	Type	Description
		NA

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 の列

Name	Type	Description
node_group_name	text	BDR グループの名前
origin_name	text	オリジンノードの名前
target_name	text	ターゲットノードの名前
slot_name	text	このサブスクリプションで使用されるオリジンノードのスロット名前
active	text	スロットはアクティブですか（接続が接続されていますか）
state	text	レプリケーションの状態（キャッチアップ、ストリーミング、...）またはオフラインの場合は「切断」
write_lag	interval	報告された書き込みのおおよその遅延時間
flush_lag	interval	報告されたフラッシュのおおよその遅延時間
replay_lag	interval	報告されたリプレイのおおよそのラグタイム
sent_lag_bytes	int8	send_lsn と現在の WAL 書き込み位置のバイト差
write_lag_bytes	int8	write_lsn と現在の WAL 書き込み位置のバイト差
flush_lag_bytes	int8	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
		NA

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 の列

Name	Type	Description
origin_node_name	text	サブスクリプションのオリジンの名前
target_node_name	text	サブスクリプションのターゲットの名前
last_xact_replay_timestamp	text	最後に再生されたトランザクションのタイムスタンプ

次のページに続く

表 69 – 前のページからの続き

Name	Type	Description
sub_lag_seconds	text	now と last_xact_replay_timestamp の間の遅れ
		NA

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 の列

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
		NA

2.22.2 内部カタログと Views**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 の列

Name	Type	Description
ddl_epoch	int8	単調に増加するエポック数
origin_node_id	oid	このエポックの作成を要求したノードの内部ノード ID
epoch_consume_timeout	timestampz	このエポックのタイムアウト
epoch_consumed	boolean	ローカルノードがエポックを完全に処理するとすぐに true に切り替わります
		NA

bdr.internal_node_pre_commit

内部カタログ表。bdr.node_pre_commit ビューを使用してください。

!!! Note * 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。

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。診断目的には、viewbdr.sequences が推奨されます。

bdr.sequence_kind の列

Name	Type	Description
seqid	oid	シーケンスの内部 OID
seqkind	char	内部シーケンスの種類（'l'=ローカル、't'=タイムシャード、'g'=ガロック）
		NA

bdr.state_journal

内部ノード状態ジャーナル。代わりに、診断目的で 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すべての変更は、診断のためにエントリーに永続的に記録されます。このビューは、ノード名と人間が読める状態名を提供し、そのジャーナルのすべての情報を保持しノード。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 の列

Name	Type	Description
state_counter	oid	ノードごとに単調に増加するイベントカウンタ
node_id	oid	内部ノード ID
node_name	name	ノードの名前
state	oid	内部状態 ID
state_name	text	人間が読み取れる州名前
entered_time	timestampz	現在のノードが状態の変化を観察した時点

2.23 Monitoring

レプリケーションセットアップの監視は、システムが最適に実行され、ディスクスペースが不足したり、操作を停止する可能性のある他の障害が発生したりしないように保証に重要です。

例、レプリケーションスロットが大幅に遅れスタートした場合に管理者に警告が発せられ、事前に対策を講じることが保証に、自動モニタリングを実施することが重要です。

EDB は、バージョン 8.1 の BDR をサポートする Postgres Enterprise Manager (PEM) を提供します。また、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 make ことができます。

2.23.1 監視の概要

BDR グループは、多くの場合ノードと呼ばれるマルチプルのサーバーで構成されます。グループ全体の健全性を保証には、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 関数を実行して、3つのレベルのいずれかを使用して BDR の健全性の評価を提供します。

- OK-しばしば緑と表示される
- WARNING-しばしば黄色として表示される
- CRITICAL-しばしば赤と表示される
- UNKNOWN と同様に-認識されない状況の場合、多くの場合赤と表示されます

BDR は、さまざまな内部メトリックの瞬間的な状態を示す動的カタログビュー、およびユーザが要求した構成のデフォルトや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には bdr_monitor または bdr_read_all_stats でアクセスできるものもありますが、セキュリティ要件が高いユーザーまたは内部情報を含むものもあります。

BDR を使用すると、各ノードを個別にモニタしたり、単一のノードにアクセスしてグループ全体をモニタしたりできます。各ノードを個別にモニタ場合は、各ノードに接続してモニタリング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。

bddr_superuser によって 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、すべてのノードに対して独自の呼び出しを行うことが make ます。

2.23.2 ノードの参加と削除の監視

デフォルトでは、ノード管理機能は、結合またはパート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 ファンクションの引数を使用してオフにすることができます。待機がオフになっている場合、結合またはパートオペレーション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、ヘルプファンクション bdr.wait_for_join_completion() は、すべての未処理のノード結合操作が完了するまで、PostgreSQL セッションを一時停止させます。

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 クエリー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 <catalogs>`__` は、キャッチアップ状態に関する情報を提供します。これは、ノードの結合またはノードの分離に関連する可能性があります。

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、すでにそのデータを受信して転送できるノードから一時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
```

2.23.3 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モニタリングに使用される 2 つの主なビューがあります。

- 発信レプリケーションをモニタリングするための ``bdr.node_slots <catalogs>`__`
- 着信レプリケーションをモニタリングするための ``bdr.subscription_summary <catalogs>`__`

`bdr.node_slots` によって提供される情報のほとんどは、標準の PostgreSQL レプリケーションモニタリングビューへのクエリによっても取得できます。

各ノードには BDR グループスロットが 1 つありますが、これには接続しないでください。また、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、何かがダウンしたり切断されたりすることを意味するものではありません。``Replication Slots created by BDR <nodes>`__` を参照してください。

送信レプリケーションの監視

発信レプリケーションアクティビティのモニタリング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モニタリングするための ``bdr.node_replication_rates <catalogs>`__`

`bdr.node_replication_rates` ビューは、特に、発信レプリケーションアクティビティの全体像と、同等なキャッチアップ推定値を提供します。

```
# 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のレートを示します。それは同等ながローカルノードからデータを消費している `rateat` です。ノードがクラスターに再接続するときの `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、同等ながローカルノードデータに追いつくのに必要な時間を参照する `catchup_intervalcolumn` を使用することをお勧めしノード。以下に説明するように、他のフィールドも `bdr.node_slotsview` を介して利用できます。

!!! Note * 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` をクエリーできます。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に関する情報と、現在のノードで BDR によって作成された追加の複製スロットが表示されます。

```
# 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、クラスターの遅延を完全にビューするには、参加しているすべてのノードでこのクエリーを実行する必要があります。

`replay_lag_bytes` は、ローカルサーバーの現在の WAL 書き込み位置と、同等なが最後に確認した位置である `replay_lsn` との WAL 位置の違いを報告しノード。`replay_lag_size` は、人間が読める形式の単なるものです。WAL には通常、複製されないが、`VACUUM` アクティビティ、インデックスの変更、同じノード上の他のデータベースに関連付けられた書き込み、レプリケーションセットのパートではないテーブル

の書き込みなど、`replay_lag_bytes` でカウントされる書き込みが多数含まれることを理解することが重要です。ここでも報告されるバイトの遅れは、同等な最新のノードにするためにワイヤ上で複製する必要があるデータの量ではなく、処理する必要があるサーバサイドの WAL の量のみです。

同様に、ノードは、同等ながキャッチアップするのにかかる時間、または `bdr.node_slots` が照会されたときに現在の位置から書き込み位置にリプレイするのにかかる時間の測定値ではありません。同等なが最新のコミットと現在の壁時計時間を確認したときの遅延を測定します。`bdr.node_replication_rates` の `replay_lag_bytes` および `replay_lag_size` または `catchup_interval` をモニタすることをお勧めします。これは、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です。

ロジカルレプリケーションがトランザクションをストリーミングしている間、バイトと時間の両方の遅れは進みません。コミットがレプリケートされるときにのみ変更されます。そのため、遅延は「鋸歯状」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、同等ながコミットしてフラッシュし、確認をノードすると再び下降します。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 make 接続でのみ意味のある他のフィールドになります。`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、クライアントが最後に再生および保存されたことがわかっている最後の位置であるためです。`_lag_size` と `_lag_bytes` フィールドは、`confirmed_flush_lsn` とローカルサーバの現在の WAL 挿入位置との間の距離をレポートします。

注：`restart_lsn` が他のノード列の背後にあるのは正常です。これは、レプリケーションまたは同等なの遅延の問題を示すものではありません。`restart_lsn` は、PostgreSQL の内部ロジカルデコーディングが中断された場合に WAL を読み取る必要がある位置であり、通常、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 make は、切断後のレプリケーションのリスタートを遅くし、望ましい以上の WAL を保持することができますが、そうでなければ無害になります。

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` ビューをクエリすることで監視できます。これにより、BDR クラスタ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が表示されます。例：

```
# 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
```

(次のページに続く)

(前のページからの続き)

sub_slot_name	bdr_postgres_bdrgroup_node1
subscription_status	replicating

LCR を使用して WAL 送信者を監視する

が有効になっている場合、各 WAL 送信者の現在の LCR () ファイルに関する情報は、ファンクションを介して監視できます。例：

```
postgres=# SELECT * FROM bdr.wal_sender_stats();
   * pid   | is_using_lcr | decoder_slot_name | lcr_file_name
-----+-----+-----+-----
2059904 | f           |                   | 
2059909 | t           | bdr_postgres_bdrgroup_decoder | 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、デコードワーカーが有効になっていない場合、または WAL 送信者が [ロジカルスタンバイ] (nodes.md # Logical Standby Nodes) を提供している場合です。

さらに、デコードワーカーに関する情報は、function bdr.get_decoding_worker_stat を介して監視できます。例：

```
postgres=# SELECT * FROM bdr.get_decoding_worker_stat();
   * pid   | decoded_upto_lsn | waiting | waiting_for_lsn
-----+-----+-----+-----
1153091 | 0/1E5EEE8       | t       | 0/1E5EF00
(1 row)
```

2.23.4 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、bdr.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これらの洞察を提供します。

- 次の場合、wait_event 列には拡張情報があります。待機の理由は BDR に関連しています。
- query 列は、BDR ワーカーでは空白になります。ただし、ライタプロセスが DDL を実行しているとき

bdr.workers ビューには、BDR ワーカー固有の詳細が表示されますが、bdr.stat_activity からは利用できません。

ビュー `bdr.worker_errors` は、作業の継続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）を示します。このビューではアクティブなエラーのみが可視されるため、ワーカーに一時的な問題があったが回復した場合、ビューは空になります。

2.23.5 BDR ライターの監視

ライターのアクティビティをモニタ別のシステムビュー `bdr.writers` があります。このビューには、ライターワーカーのみの現在のステータスが表示されます。以下が含まれます。

- ライターが属するサブスクリプションを識別する `sub_name`
- ライタープロセスの `pid`
- ライターがアプリケーションのサポートをサポートしているかどうかを知るための `streaming_allowed` 進行中のストリーミングトランザクション
- ライターが現在適用しているかどうかを知るための `is_streaming` ストリーミングトランザクション
- `commit_queue_position` で、ライターの位置を確認しますコミットキュー。

BDR は、オリジンに追加された同じコミットオーダーに従うことにより、コミット順序付けを尊重します。並列ライターの場合、マルチプルのライターが異なるトランザクションを同時に適用する必要があります。`commit_queue_position` は、コミットオーダーを示します。値 `0` は、ライターが最初にコミットを意味します。値 `-1` は、コミット位置がまだ不明であることを意味します。これは、ストリーミングトランザクションまたはライターがその時点でトランザクションを適用していない場合に発生する可能性があります。

2.23.6 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、グローバルロックには2つのタイプがあります。

DDL ロック、永続的なすべての DDL 操作のシリアル化に使用データベース内の（一時的ではない）オブジェクト（テーブル）DDL 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る DML リレーションロックリレーション定義を変更する操作

DDL オペレーションのタイプと `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どちらかまたは両方のエン트리タイプを作成できます。

ローカルノードで保持されているグローバルロックは、に可視されます。このビューには、ロックのタイプが表示されます。関係ロックは、どのリレーションがロックされているか、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許可されているかどうかを示します。グローバルアドバイザリロックの場合、列はロックが取得されるアドバイザリキーを示しキー。

以下は、`bdr.ddl_locking = on` で `anALTER TABLE` ステートメントを実行しているときの `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、カタログの文書を参照してください。

2.23.7 競合の監視

レプリケーション `conflicts` は、マルチプルのノードが相互に対話できる方法で同じ行に影響を与える変更を行う場合に発生する可能性があります BDR システムを保証して、競合を特定し、可能な場合はアプリケーションの変更を行って、競合を排除または頻度を減らします。

デフォルトでは、すべての競合は `bdr.conflict_history` 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可視保証にします。所有者は競合を予想し、それらを分析して、もしあれば、解決すべき問題と見なされる可能性があるものを確認する必要があります。

モニタリングのために、ユーザデータを含まない `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、現在の日に発生した競合の数をカウントするクエリー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');
```

2.23.8 外部モニタリング

ユーザーが提供したメタデータを保存して、モニタリングツールが BDR クラスターを理解およびモニタできるようにすることができます。この情報を集中化することにより、`externaltools` は任意の単一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ワーニング/アラームしきい値など、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` にはこれらの関数とテーブルに対する特権があります。ビュー `bdr.network_monitoring` には、`bdr_read_all_stats` ロールからもアクセスできます。

bdr.set_node_location

このファンクションは、ノードメタデータを `bdr.node_location` に挿入します

あらすじ

```
bdr.set_node_location(  
    node_group_name text,  
    node_name text,  
    node_region text,  
    node_location text);
```

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`- サーバー名前、可用性ゾーンなど。

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経路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。

あらすじ

```
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)
```

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前
- `region2`-リモートサーバ名前
- `location1`-オリジンのデータセンター名前
- `location2`-リモートデータセンター名前
- `network_cost`-ネットワーク転送のコストを表すアブストラクト値
- `warning_threshold`-しきい値を超えると遅延が発生する
- `alarm_threshold`-アラームが発生するまでの遅延

`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` 関数によって定義されます。

2.23.9 統計を適用する

BDR は、各サブスクリプションと各テーブルの両方に適用されるレプリケーション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提供します：

- 複製された 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ている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ている進行中のトランザクションの数
- コミット/中止された進行中のストリーミングトランザクションの数

リレーションのみの場合、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
- リレーションのロック（ある場合）を取得するための合計ロック待機時間（のみ）

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対して複製された 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、BDR GUCs `bdr.track_subscription_apply` および `bdr.track_relation_apply` によってそれぞれ制御されます。

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。

2.23.10 標準の PostgreSQL 統計 Views

テーブルおよびインデックスの使用に関する統計は、通常、`downstreammaster` によって更新されます。これは、`autovacuum` の正しいファンクションに不可欠です。ダウストリームマスタにローカル書き込みがなく、統計がリセットされていない場合、これらの2つのビューはアップストリームとダウストリームの対応する結果を表示する必要があります。

- `pg_stat_user_tables`
- `pg_statio_user_tables`

!!! Note * アップストリームテーブルの統計がダウストリーム統計と同様であるとは必ずしも期待していません。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てください。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、アップストリームとダウストリームのテーブル統計は同じ量だけ変化します。一方の側のすべての変更が他方の側に複製されるためです。

インデックスは変更の適用に使用されるため、下流側の識別インデックスは、非識別インデックスよりも `b_tran_1s` および `DELETEs` を実行したワークロードでより頻繁に使用されるように見える場合があります。

ビルトインのインデックスモニタリングビューは次のとおりです。

- `pg_stat_user_indexes`
- `pg_statio_user_indexes`

これらのビューはすべて、[PostgreSQL documentation on the statistics views](#) で詳細に説明されています。

2.23.11 BDR バージョンの監視

BDR では、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 `functionbdr.run_on_all_nodes()` を使用して、すべてのノードから Postgres および BDR 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。クラスターが異なる BDR バージョンを長時間実行しないことをお勧めします。

モニタリングのために、次のアラートレベルをお勧めします。

- status = UNKNOWN、メッセージ=このノードは BDR グループのパートではありません
- status = OK、メッセージ=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、メッセージ=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、メッセージ=異なる BDR バージョンを実行しているノードがあります他のノードと比較した場合

説明された動作は、`bdr.group_version_details` ビューから返された 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する `functionbdr.monitor_group_versions()` に実装されています。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status | message
-----+-----
 OK     | All nodes are running same BDR versions
```

2.23.12 ラフトコンセンサスの監視

Raft Consensus は常にクラスタ全体で機能する必要があります。Raft Consensus が機能しない状態で 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機能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 は機能しません - クラスターメンテナンス操作（結合ノード、パートノード、スタンバイのプロモート）まだ許可されていますが、終了しない可能性があります（単にハングします）BDR ノード間でノードステータスが正しく同期されない場合があります 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 ファイルを保持しますディスク

ビュー `bdr.group_raft_details` は、関数 `bdr.run_on_all_nodes()` と `bdr.get_raft_status()` を使用して、すべてのノードから RaftConsensus 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
```

以下のすべての条件が満たされている場合、ラフトコンセンサスは正しく機能していると言えます。

- すべてに対して有効な状態（RAFT_LEADER または RAFT_FOLLOWER）が定義されていますノード
- ノードの 1 つだけが RAFT_LEADER です
- `leader_id` はすべての行で同じであり、`node_id` とマッチする必要があります `state = RAFT_LEADER` がある行の

Raft Consensus は、随時、新しい選挙をスタートして、新しい RAFT_LEADER を定義します。選挙中に、RAFT_LEADER が存在せず、一部のノードが自分自身を RAFT_CANDIDATE と見なす仲介者がいる場合があります。選挙全体が `bdr.raft_election_timeout` より長くかかることはありません（デフォルトでは 6 秒に設定されています）。上記のクエリで選挙中のシチュエーションが返された場合は、`edbdr.raft_election_timeout` を待ってから再度クエリを実行します。`afterbdr.raft_election_timeout` が過ぎても上記の条件のいくつかはまだ満たされていない場合、Raft Consensus は機能していません。

Raft Consensus は、単一のノードでのみ正常に動作しない場合があります。例、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、通常およびレプリケーション接続（ファイル `pg_hba.conf` を確認）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互いに離れていたり、ネットワークのレイテンシが長い場合は、`bdr.raft_election_timeout`（6 秒）のデフォルト値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 RaftConsensus がまだ機能しない場合は、すべてのノードで `bdr.raft_election_timeout` をたとえば 30 秒に増やすことを検討してください。BDR 3.6.11 以降、`settingbdr.raft_election_timeout` の設定に必要なのはサーバーのリロードのみです。

Raft Consensus がクラスターの運用タスクにどのように影響し、asRaft Consensus がグループスロットを進める責任があるかを考えると、モニタリングアラートレベルを次のように定義できます。

- `status = UNKNOWN`、メッセージ=このノードは BDR グループのパートではありません
- `status = OK`、メッセージ= Raft Consensus は正常に機能しています
- `status = WARNING`、メッセージ=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`、メッセージ= RAFT_CANDIDATE としてノードがあり、選挙が進行中の可能性があります
- `status = WARNING`、メッセージ= RAFT_LEADER はありません。選挙は進行中

- status = CRITICAL、メッセージ= Raft Consensus に単一のノードがあります
- status = CRITICAL、メッセージ= RAFT_CANDIDATE としてノードがありますが、RAFT_LEADER が定義されています
- status = CRITICAL、メッセージ=異なるリーダーに続くノードがあります RAFT_LEADER としてノードセットよりも

説明した動作は、bdr.group_raft_details ビューから返された Raft コンセンサスステータス情報を使用してクラスタ全体の Raft チェックを提供する functionbdr.monitor_group_raft() に実装されています。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status | message
-----+-----
 OK      | Raft Consensus is working correctly
```

2.23.13 レプリケーションスロットの監視

各 BDR ノードは以下を保持します。

- アクティブな BDR 同等なごとに 1 つのレプリケーションスロット
- 1 つのグループレプリケーションスロット

例：

```
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 bdr_<DATABASE>_<GROUP>_<PEER> の規則に従い、BDR グループスロット名前は functionbdr.local_group_slot_name() を使用してアクセスできる Conventionbdr_<DATABASE>_<GROUP> に従います。

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。同等なレプリケーションスロットがアクティブでない場合は、次のことを意味します。

- 対応する同等な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。ルート原因は、例、ノードの 1 つで DDL レプリケーションが無効になっている状態で、互換性のない DDL が実行された可能性があります。

ただし、ほとんどの場合、BDR グループレプリケーションスロットは非アクティブです。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使用している場合、LSNを進めます。したがって、グループスロットのステータスを監視する必要はありません。

ファンクション `bdr.monitor_local_replslots()` は、すべての 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。例：

```
bdrdb=# SELECT * FROM bdr.monitor_local_replslots();
 status | message
-----+-----
 OK      | All BDR replication slots are working correctly
```

次のステータス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

2.23.14 トランザクションのコミットの監視

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 CAMO と Eagerreplication の 2 つの新しいトランザクションコミットモードも提供します。これらの各モードは、COMMIT での追加のレイテンシを犠牲にして、追加の堅牢性機能を提供します。COMMIT での追加時間は、レポートカタログを使用して動的に監視できます。1 つ以上の同期的スタンバイからの確認を待機している 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 をレポートします。

2.24 BDR System Functions

BDR 管理は、主に SQL 呼び出し可能関数を介して実行されます。BDR のすべての関数は、BDR スキーマで公開されます。これらの関数の呼び出しは、`bdr` を `search_path` に入れるのではなく、スキーマで修飾する必要があります。

このページには、文書の他のセクションで説明されていない追加のシステム機能が含まれています。

2.24.1 バージョン情報 Functions

bdr.bdr_version

このファンクションは、現在使用中の BDR バージョンのテキスト表現形式を取得します。

bdr.bdr_version_num

このファンクションは、現在使用されている BDR バージョンの数値表現形式を取得します。バージョン番号は単調に増加しているため、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単一の数値に変換するために使用されます。

$\text{MAJOR_VERSION} * 10000 + \text{MINOR_VERSION} * 100 + \text{PATCH_RELEASE}$

2.24.2 システムおよび進捗情報のパラメーター

BDR は、`psql` の `SHOW` を介して、または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。このセクションでは、BDR レポートの対象となるこのようなパラメーターをすべてリストします。

bdr.local_node_id

セッションの初期化時に、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あっても接続されているノードを把握できます。

また、CAMO と組み合わせて使用されます。[CAMO.md # connection-pools-and-proxies]（接続プールとプロキシ）セクション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、このパラメータは更新され、オリジンノードのコミットレコードの終わりを指します。`bdr.wait_for_apply_queue` との組み合わせにより、アプリケーションはマルチプルのノード間で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され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、このパラメータは更新され、割り当てられたばかりのトランザクション ID が表示されます。

!!! Note * これは、EDB Postgres Extended でのみ利用可能です。

2.24.3 ユーティリティ Functions

bdr.wait_slot_confirm_lsn

ユーザは、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にリプレイされるまで待つことができます。

スロットが特定の LSN を通過するまで待機します。位置が指定されていない場合、現在の書き込み位置がローカルノードで使用されます。

スロット名前が渡されない場合、すべての BDR スロットが LSN を通過するまで待機します。

このファンクションは、他のノードからの変更を 1000 ミリ秒ごとにポーリングします。

スロットが同時にドロップされると、そのスロットの待機が終了します。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、待機は続行されます。

あらすじ

```
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、または NULL の場合、現在の書き込み LSN を使用しますローカルノード

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 ノードから発信され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つことができノード。その `peernode` 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結果れます。アプリケーションまたはプロセスは、この関数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 (`edbdr.wait_for_camo_partner_queue` を参照) に対してこのファンクションの特別なバリエーションを提供します。

特定の LSN が指定されている場合、それは同等なからのリカバリストリーム内の待機ポイントです。これは、以前の接続または同時接続でその同等なから取得したノードと組み合わせて使用できます。

指定された `target_lsn` が NULL の場合、このファンクションは `localreceive` バッファをチェックし、指定された同等なから受信した最後のトランザクションの LSN を使用しノード。すで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ています。これは、`peernode` に障害が発生し、送信されたトランザクションが不明な場合に特に役立ちます。この場合、送信側でまだ送信中またはバッファリングされているトランザクションは待機されません。

あらすじ

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

パラメーター

ノード着信元の同等な名前トランザクションはキューに入れられ、待機する必要があります のために。NULL の場合、すべての同等な適用キューが消費されるまで待機します。ノード同等なからのレプリケーションストリームの LSN 待つために、通常は `bdr.last_committed_lsn` から 同等なノード。

bdr.get_node_sub_receive_lsn

サブスクライバーでこのファンクションを使用して、指定されたオリジンから受け取った最後の LSN を取得できます。適用または非フィルタリングされるトランザクションに関連する LSN 増分のみを考慮に入れるためにフィルター処理されます。

このファンクションの出力と `bdr.get_node_sub_apply_lsn()` の出力の差により、対応する適用キューのサイズが測定されます。

あらすじ

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

パラメーター

- `node_name`-のソースであるノードの名前 LSN を取得するレプリケーションストリーム/
- `committed`-デフォルト (true) の場合、このファンクションはアカウントは、最後ではなく、受信したトランザクションのコミットのみ LSN 全体。サブスクライバーに影響を与えないアクションを含む ノード

bdr.get_node_sub_apply_lsn

サブスクライバーでこのファンクションを使用して、指定された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。

あらすじ

```
bdr.get_node_sub_apply_lsn(node_name name)
```

パラメーター

- `node_name`-のソースであるノードの名前 LSN を取得するレプリケーションストリーム。

bdr.run_on_all_nodes

すべてのノードでクエリーを実行するファンクション。

!!! Warning * 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の権限を使用して、リモートノードで任意のクエリーを実行します。このファンクションに特権を付与するときは注意が必要です。

あらすじ

```
bdr.run_on_all_nodes(query text)
```

パラメーター

- `query`-実行される任意のクエリー。

注

このファンクションは他のノードに接続してクエリーを実行し、各ノードから json フォーマットで結果を返します。json 配列としてエンコードされた各ノードから複数の行が返される場合があります。ノードがダウンしているために接続できないなどのエラーは、応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメータは設定されていないため、デフォルトが使用されます。

このファンクションは通常のレプリケーションを経由せず、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、コマンドはすで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、`bdr.ddl_replication = off` で接続が作成されます。

このファンクションを使用する場合は、レプリケーションを中断し、ノード間で不整合を引き起こすリスクがあるため注意してください。透過的な DDL replication または `bdr.bdr_replicate_ddl_command()` を使用して DDL を複製します。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

このファンクションは、例次のクエリのモニタリングで使用する则便利です。

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
    *   FROM bdr.node_slots
    *   WHERE origin_name IS NOT NULL
$$);
```

… 2 ノードクラスタでは、次のようなものが結果れます。

```
[
  *   {
  *       "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",
  *                   "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に対してクエリーを実行するファンクション。

!!! Warning * 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の権限を使用して、リモートノードで任意のクエリーを実行します。このファンクションに特権を付与するときは注意が必要です。

あらすじ

```
bdr.run_on_nodes(node_names text[], query text)
```

パラメーター

- **node_names**-クエリーが実行されるノード名のテキスト配列。
- **query**-実行される任意のクエリー。

注

このファンクションは他のノードに接続してクエリーを実行し、各ノードから json フォーマットで結果を返します。json 配列としてエンコードされた各ノードから複数の行が返される場合があります。ノードがダウンしているために接続できないなどのエラーは、応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメータは設定されていないため、デフォルトが使用されます。

このファンクションは通常のレプリケーションを経由せず、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、コマンドはすで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、`bdr.ddl_replication = off` で接続が作成されます。

このファンクションを使用する場合は、レプリケーションを中断し、ノード間で不整合を引き起こすリスクがあるため注意してください。透過的な DDL replication または `bdr.bdr_replicate_ddl_command()` を使用して DDL を複製します。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ーを実行するファンクション。

!!! Warning * 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の権限を使用して、リモートノードで任意のクエリーを実行します。このファンクションに特権を付与するときは注意が必要です。

あらすじ

```
bdr.run_on_group(node_group_name text, query text)
```

パラメーター

- `node_group_name`-クエリーが実行されるノードグループの名前。
- `query`-実行される任意のクエリー。

注

このファンクションは他のノードに接続してクエリーを実行し、各ノードから json フォーマットで結果を返します。json 配列としてエンコードされた各ノードから複数の行が返される場合があります。ノードがダウンしているために接続できないなどのエラーは、応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメータは設定されていないため、デフォルトが使用されます。

このファンクションは通常のレプリケーションを経由せず、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、コマンドはすで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、`bdr.ddl_replication = off` で接続が作成されます。

このファンクションを使用する場合は、レプリケーションを中断し、ノード間で不整合を引き起こすリスクがあるため注意してください。透過的な DDL replication または `bdr.bdr_replicate_ddl_command()` を使用して DDL を複製します。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 Locking Details](#) を参照してください。

あらすじ

```
bdr.global_lock_table(relation regclass)
```

パラメーター

- `relation`-ロックされるリレーションの名前または OID。

注

このファンクションは、`ddl_locking` 設定とは無関係にグローバル DML ロックを取得します。

`bdr.backwards_compatibility` isset が 30618 以下に設定されていない限り、`bdr.global_lock_table` ファンクションでは、ロックされた `relation` で `UPDATE`、`DELETE`、または `TRUNCATE` privilege が必要です。

bdr.wait_for_xid_progress

このファンクションは、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 `make` のを待つために使用できます。進行状況は、ノードに適用され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他のすべてのレプリケーションの変更が行われたことを確認します。

あらすじ

```
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あったノードのノード ID 発生しました。
- `origin_topxid`-トランザクションの XID。
- `allnodes`-true の場合、トランザクションが進行するのを待つすべてのノード。そうでなければ、現在のノードを待つだけです。

注

これらのトランザクションのみが現在追跡されているため、このファンクションは、DDL コマンドを複製したトランザクションに対してのみ使用できます。間違った `origin_node_id` または `origin_topxid` が指定された場合、ファンクションは永遠に、または `statement_timeout` がヒット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()` に渡されたものと同じですが、グループの作成時に `node_group_type` が NULL として渡された場合は `normal` が返されます。

例

```
bdrdb=# SELECT bdr.node_group_type('bdrgroup');
node_group_type
-----
normal
```

2.24.4 グローバルアドバイザリーロック

BDR はグローバルアドバイザリーロックをサポートしています。これらのロックは、BDR がサポートするアドバイザリーロックが本質的にグローバルであることを除いて、PostgreSQL で使用可能なアドバイザリーロックに非常に似ています。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、すべてのノードの過半数が連携できる限り、1 つまたは複数のノードがダウンまたは遅れている場合でも、多数決コンセンサスによってアドバイザリーロックを取得できます。

現在、排他ロックのみをサポートしています。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ー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除されるのを待つ必要があります。

アドバイザリーロックは本質的にトランザクションです。そのため、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除されレベル。その場合、リリースされるとすぐに使用可能になります。

グローバルアドバイザリー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、他のセッションで使用するために 3 回ロック解除する必要があります。

bdr.global_advisory_lock

このファンクションは、提供されたオブジェクトの排他ロックを取得します。ロックが使用可能になるか、bdr.global_lock_timeout に達するまで待機します。

あらすじ

```
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。

あらすじ

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

パラメーター

- **key1**-複合キーの最初のパート。
- **key2**-複合キーの 2 番目のパート。

bdr.global_advisory_unlock

このファンクションは、アプリケーションが定義したソースで以前に取得したロックを解放しました。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以前に取得されている必要があります。そうでない場合は、エラーが発生します。

あらすじ

```
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。

あらすじ

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

パラメーター

- **key1**-複合キーの最初のパート。
- **key2**-複合キーの 2 番目のパート。

2.25 Credits and Licence

BDR は、このチームによって設計、開発、テストされています。

- ペトル・イエリネク
- クレイグリナー
- マーカス・ワナー
- パヴァン・デオラシー
- トマス・ボンドラ
- サイモン・リッグス
- ニキル・ソントック
- パラヴィ・ソントック
- アムルタデオラシー
- ラヒラ・サイード
- Ashutosh Bapat
- アビジット・メノン・セン
- フローリンイリオン
- オリバー・リッグス

Copyright © 2018-2020 2ndQuadrant Ltd Copyright © 2021 EnterpriseDB UK Ltd

BDR は EDB 使用ライセンスの下で提供されます。

これらの文書の複製は禁止されています。

2.26 ' Appendix A

2.26.1 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ている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 を参照してください）。以前の 2 つのメジャーバージョンは 3.7 と 3.6 です。

改善

EDB Postgres Advanced 14 の BDR は、次の機能をサポートするようになりました以前は EDB Postgres Extended でのみ利用可能でした：

- -最大 1 回コミット-ノード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 回のみコミットのを支援する一貫性機能-イーガーレプリケーション-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で同期して、競合のフリーレプリケーションを提供-デコードワーカー-多数のノードでスケラビリティを向上させるために、デコードを wal の送信者から分離したワーカーに分離-レプリケーションのキャッチアップ時間の見積もり-タイムスタンプベースのスナップショット-マルチプルのノードで一貫した読み取りを提供し、特定の時間に出現または出現するデータを取得-特定のコーナーケースでの UPDATE / DELETE 競合解決の一貫性を改善するための行凍結の自動的構成-評価チェック-エラーではなく競合として欠落しているパーティションのハンドリングのサポート-NOT VALID コンストレインおよび ALTER TABLE の高度な DDL 処理

PostgreSQL 12-14 のコミュニティバージョンの BDR は、次の機能をサポートするようになりました以前は EDB Postgres Advanced または EDB Postgres Extended：

- -競合のないレプリケートされたデータ型-非同期マルチマスター更新シナリオ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-列レベルの競合解決-列ごとに最終更新を使用する機能により、異なるフィールドの更新を「マージ」できるどちらも失うことなく-変換トリガー-ストリームデータで実行されるトリガーで、それを変更したり、高度なプログラムフィルタリングを実行したりできます-競合トリガー-競合が検出されたときに呼び出され、カスタム競合を使用する方法を提供します解決手法-CREATE TABLE AS レプリケーション-並列適用-マルチプルのライターが着信変更を適用できるようにする
- ラージトランザクションのストリーミングをサポートします。
 - これにより、BDR はラージトランザクション（サイズが `logical_decoding_work_mem` をより大きい）をダウンストリームのファイルまたはライタプロセスのいずれかにストリーミングできます。これにより、トランザクションがコミットされる前にデコードされ、並列性が向上します。さらに、トランザクションはライターに直接ストリームされる場合、同時に適用することもできます。これにより、並列処理がさらに改善されます。

ラージ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、コミットされた前でもデコードされた変更が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されるときに適用されます。トランザクションが中断すると、変更は破棄され、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。

- サブトランザクションも自動的に処理されます。

- この機能は、PostgreSQL 14、EDB Postgres Extended 13+および EDB Postgres Advanced 14 で使用できます。どの機能を Postgres のどのバージョンで利用できるかについての詳細は、付録 [Feature Compatibility](#) を参照してください。

BDR の以前のバージョンに存在していた標準間の違いエンタープライズエディションは削除されました。BDR 4.0 では、1 つの拡張機能がありますサポートされている各 Postgres ディストリビューションとバージョン、つまり PostgreSQL v12-14 に対して、EDB Postgres Extended v12-14 および EDB Postgres Advanced 12-14。

PostgreSQL のすべてのバージョンですべての機能が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()` データベースファンクションのいずれかを使用する機能フラグによって報告されます。詳細については、付録 [Feature Compatibility](#) を参照してください。

- BDR 4.0 拡張に対応する BDR 4.0 拡張はありません。BDR には `pglogical` の要件はなくなりました。
 - 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。
- 一部の構成オプションには、インストール後の改善のためにデフォルトが変更されています経験：
 - `-並列適用`がデフォルトで有効になりました（2つのライターを使用）。特にストリーミングを有効にすると、パフォーマンスが向上します。`-COPY`と `CREATE INDEX CONCURRENTLY` は、デフォルトで利用可能なすべてのノードに並行して（ストリーミングがサポートされている Postgres バージョンで）ライターに直接ストリーミングされ、これらの操作後のレプリケーションラグスパイクを排除または少なくとも削減します。`-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`-bdr.min_worker_backoff_delay` はデフォルトで 1 秒になり、エラー時にサブスクリプションが 1 秒間に 1 回だけ接続をリトライするようになりました
- 競合検出中の誤検知の可能性を大幅に削減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
- CAMO ペアの構成を Raft (CAMO) に移動する
 - 構成ミスの `make` 性を減らし、BDR クラスター内の CAMO ペアをグローバルに知らせるには、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、例、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が誤ってドロップされるのを防ぐことができます。

アップグレードプロセスの詳細については、[Upgrades chapter](#) を参照してください。

- これにより、GUC の `bdr.camo_partner_of` および `bdr.camo_origin_for` が非推奨になり、関数 `bdr.get_configured_camo_origin_for()` および `get_configured_camo_partner_of` が `bdr.get_configured_camo_partner` に置き換えられます。

アップグレード

このリリースでは、次のバージョンの BDR からのアップグレード処理がサポートされています。

- 3.7.13.1

アップグレード処理前に、[Upgrade Guide](#) に記載されているプロセスと制限を make 読んで理解してください。

2.27 ' Appendix B

BDR の一部の機能は、それらの機能をサポートできる特定のバージョンの Postgres でのみ機能します。また、PostgreSQL、EDB Postgres Extended、および Postgres の EDB Postgres EDB Postgres Advanced バリエーションで機能する機能にも違いがあります。

次の表に、BDR の機能と、Postgres の指定されたバリエーションと、オプションでどのバージョンからサポートされているかを示します。

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+
Assesment Tooling	N	Y	14+
Lag Tracker	N	Y	14+
Timestamp Snapshots	N	Y	14+
Transaction Streaming	14+	13+	14+
Missing Partition Conflict	N	Y	14+
No UPDATE Trigger on tables with TOAST	N	Y	14+

2.28 ' Appendix C

<AuthenticatedContentPlaceholder target = " [https:// 文書.enterprisedb.com / bdr4 / リリース / latest / isolation / details /](https://文書.enterprisedb.com/bdr4/リリース/latest/isolation/details/) " トピック = "付録 C：競合の詳細" />

2.29 ' Appendix D

このセクションでは、BDR4 の現在既知の問題について説明します。

2.29.1 データの一貫性

`Conflicts` を読んで、データの一貫性に関する非同期オペレーションモードの意味を理解してください。

2.29.2 問題のリスト

このセクションの残りのパートでは、BDR のチケットシステムで追跡されている既知の問題の数をリストし識別子。

- `update_origin_change` のリゾルバが競合する場合 `skip` に設定され、`synchronous_commit=remote_apply` が使用されます。同じ行の同時更新が 2 つに繰り返し適用されます異なるノード、更新ステートメントの 1 つが原因でハングする可能性があります BDR ライタとのデッドロック。で述べたように `Conflicts` の章、`skip` はデフォルトではありません `update_origin_change` 競合のリゾルバ、およびこれ組み合わせは、稼動環境での使用を意図していません。破棄します到着オーダーに基づく 2 つの競合更新の 1 つそのノード上で、発散クラスターを引き起こす可能性があります。まれに、シチュエーションの使用を選択する場合コンフリクトリゾルバ、使用の問題にノートしてください `remote_apply` モード。

スキップシーケンスは、シーケンスはロールバックされたトランザクションで作成され、その後作成されます再び同じ名前で、または DDL 作成およびドロップされる場合レプリケーションはアクティブではなく、DDL のときに再度作成されますレプリケーションはアクティブです。問題のインパクトは軽度です。なぜならシーケンスは保証違反はありません。シーケンスは一部をスキップします初期チャック。また、回避策として、ユーザはへの引数としてのシーケンスの開始値 `bdr.alter_sequence_set_kind()` ファンクション。

BDR 3.7.7 からの PG Extended 13 でのアップグレードは、追加することによってのみサポートされます新しいノード、および同じデータディレクトリのインプレースアップグレードによる**ではない**。

- `bdr.monitor_local_replslots()` ファンクションは CRITICAL 結果を結果場合があります「欠落している BDR レプリケーションスロットが少なくとも 1 つある」と言ってもすべてのスロットがロジカルスタンバイまたはサブスクリプト専用ノードの存在下に存在するグループ。
- デコードワーカー機能が CAMO / EAGER で機能しない
- デコードワーカーはデフォルトのレプリケーションセットでのみ動作します

2.30 ' Appendix E

このセクションでは、BDR4 で使用されるライブラリと対応するライセンスをリストします。

Library	License
LLVM	BSD (3-clause)
OpenSSL	SSLey License AND OpenSSL License
Libpq	PostgreSQL License

2.30.1 LLVM

Copyright © 1994 The University of the University of the California. Allrights 予約。

ソースおよびバイナリ形式での再配布および使用は、変更の有無にかかわらず、次の条件が満たされている場合に許可されます。

1. ソースコードの再配布には、上記の著作権表示、この条件リスト、および以下の免責事項を保持する必要があります。バイナリフォームでの再配布では、上記の著作権表示、この条件リスト、および以下の免責事項をディストリビューション物とともに提供される文書および/またはその他の資料に再現する必要があります。書面による事前のパーミッションなしに、本ソフトウェアから派生した製品を支持またはプロモートに、大学の名前もその貢献者の名前も使用することはできません。

このソフトウェアは、管理者および提供者 \AS が提供するものであり、商品性および特定の適合性の黙示的保証を含むがこれに限定されない明示的または黙示的な保証は放棄されます。いかなる場合においても、再生成者または貢献者は、直接的、間接的、偶発的、特別、例示的、または間接的な損害（代替品またはサービスの調達、使用の損失、データ、または損失、ORBUSIN; 契約、厳格責任、または不法行為（その他の過失責任者を含む）のいずれにおいても、このソフトウェアの使用から生じた責任、および損害の可能性について通知された責任の理論

2.30.2 OpenSSL

Copyright © 1998-2004 OpenSSL Project。禁無断転載

ソースおよびバイナリ形式での再配布および使用は、変更の有無にかかわらず、次の条件が満たされている場合に許可されます。

1. ソースコードの再配布では、上記の著作権表示、この条件リスト、および以下の免責事項を保持する必要があります。

2. バイナリフォームでの再配布では、上記の著作権表示、この条件リスト、および以下の免責事項を、ディストリビューション物とともに提供される文書および/またはその他の資料に複製する必要があります。

3. このソフトウェアの機能または使用について言及するすべての広告資料には、「この製品には、OpenSSL Toolkit で使用するために OpenSSL Project によって開発されたソフトウェアが含まれます。」(<http://www.openssl.org/>)

4. 「OpenSSL Toolkit」および「OpenSSL Project」という名前は、書面による事前のパーミッションなしに、このソフトウェアから派生した製品を推奨またはプロモートに使用しないでください。書面パーミッションについては、openssl-coreopenssl.org にお問い合わせください。

5. このソフトウェアから派生した製品は、OpenSSL Project の事前の書面パーミッションなしに「OpenSSL」と呼ばれたり、名前に「OpenSSL」が表示されたりすることはありません。

6. いかなるフォームの再配布でも、以下の承認を保持する必要があります。「この製品には、OpenSSL Toolkit で使用するために OpenSSL Project によって開発されたソフトウェアが含まれます (< <http://www.openssl.org/>)」>

このソフトウェアは、OpenSSL プロジェクト「AS IS」によって提供され、商品性および特定の適合性の黙示的な保証を含むが、これに限定されない、明示的または黙示的な保証は否認されます。いかなる場合でも、OpenSSL プロジェクトまたはその貢献者は、直接的、間接的、偶発的、特別、例示的、または結果的損害（代替品またはサービスの調達、使用、データ、利益の損失を含むが、これらに限定されない）の責任を負わないものとします事業の中断）契約、厳格な責任、または不法行為（過失またはその他を含む）を問わず、責任の理論上、いかなる場合でも、このソフトウェアの使用に起因する可能性があります。

この製品には、Eric Young (eaycryptsoft.com) によって作成された暗号化ソフトウェアが含まれています。この製品には、TimHudson (tjhcryptsoft.com) によって作成されたソフトウェアが含まれています。

2.30.3 元の SSLeay ライセンス

Copyright © 1995-1998 Eric Young (eaycryptsoft.com) 無断複写・転載を禁無断転載

このパッケージは、Eric Young (eaycryptsoft.com) によって記述された SSL 実装です。この実装は、Netscapes SSL に準拠するように記述されています。

このライブラリは、次の条件が満たされている限り、商用および非商用での使用はフリーです。以下の条件は、RC4、RSA、lhash、DES などのコードであるこのディストリビューションで見つかったすべてのコードに適用されます。SSL コードだけではありません。このディストリビューションに含まれる SSL ドキュメントは、ホルダーが Tim Hudson (tjhcryptsoft.com) であることを除き、同じ著作権条件で保護されています。

Copyright は Eric Young のままであるため、コード内の Copyright 表示は削除されません。このパッケージを製品で使用する場合、Eric Young には、使用するライブラリの一部の作成者として帰属が与えられフォーム。これは、パッケージで提供されるプログラムスタートアップのテキストメッセージまたは文書（オンラインまたはテキスト）の。

ソースおよびバイナリ形式での再配布および使用は、変更の有無にかかわらず、次の条件が満たされている場合に許可されます。

1. ソースコードの再配布には、著作権表示、この条件リスト、および以下の免責事項を含める必要があります。バイナリフォームの再配布では、上記の著作権表示、この条件リスト、および以下の免責事項を、ディストリビューション物とともに提供される文書および/またはその他の資料に再現する必要があります。このソフトウェアの機能または使用について言及するすべての広告資料には、「この製品には Eric Young (ライブラリ) によって作成された暗号化ソフトウェアが含まれています」ワード使用許諾が表示されている必要があります。暗号関連ではありません:-)。4. apps ディレクトリ（アプリケーションコード）から Windows 固有のコード（またはその派生物）を含める場合、「この製品には Tim Hudson (tjhcryptsoft.com) によって作成されたソフトウェアが含まれます」という謝辞を含める必要があります。

このソフトウェアは ERIC YOUNG “\AS IS” によって提供され、商品性および特定の目的への適合性の黙示的な保証を含むがこれに限定されない明示的または黙示的な保証は否認されます。いかなる場合においても、著者または貢献者は、直接的、間接的、偶発的、特別、模範的、または結果的損害（代替財の調達、データの損失、使用権、損失、損害、または損失）に対する責任を負いません。契約、厳格責任、不法行為（過失またはその他を含む）を問わず、損害の可能性について通知された場合でも、このソフトウェアの使用のいかなる方法においても、責任および責任の理論に起因します。

このコードの一般公開されているバージョンのライセンスおよび配布条項は変更できません。つまり、このコードを単純にコピーして別のディストリビューションライセンス（GNU Public Licence を含む）に入れることはできません。

2.30.4 PostgreSQL ライセンス

PostgreSQL データベース管理システム（旧称 Postgres、その後 Postgres95）

部分 Copyright © 1996-2020、The PostgreSQL Global Development Group

部分 Copyright © 1994、The University of the University of the California

上記の著作権表示とこの段落と次の2つの段落がすべてのコピーに記載されている場合、このソフトウェアとそのドキュメントをあらゆる目的で、無料で、書面による同意なしに使用、コピー、修正、配布する許可がここに付与されます。

いかなる場合においても、カリフォルニア大学は、本ソフトウェアおよびその文書の使用に起因する利益の損失を含む、直接的、間接的、特別、偶発的、または結果的な損害に対して、カリフォルニア大学が支援する場合でも、いかなる責任も負わないものとします可能性のある損害。

カリフォルニア大学は、商品性および特定の目的への適合性の暗黙の保証を含むが、これらに限定されない保証を特に放棄します。ここで提供されるソフトウェアは「現状のまま」であり、カリフォルニア大学には、メンテナンス、サポート、更新、拡張、または変更を提供する義務はありません。

2.31 CAMO Clients

<AuthenticatedContentPlaceholder target = ” [https:// 文書.enterprisedb.com / bdr4 / リリース/ latest / camo_clients /](https://文書.enterprisedb.com/bdr4/リリース/latest/camo_clients/)” トピック= “CAMO クライアント” />