



BDR Bi-Directional Replication

リリース *4.1.0*

EDB

2022 年 06 月 05 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	Release Notes	3
2.1.1	BDR 4.1.0	4
2.1.2	BDR 4.0.2	10
2.1.3	BDR 4.0.1	12
2.1.4	BDR 4.0.0	15
2.2	Architectural Overview	19
2.2.1	基本アーキテクチャ	20
2.2.2	アーキテクチャーオプションとパフォーマンス	22
2.2.3	展開	23
2.2.4	時計とタイムゾーン	24
2.3	Application use	24
2.3.1	アプリケーションの動作	24
2.3.2	レプリケートされないステートメント	26
2.3.3	DML および DDL レプリケーション	27
2.3.4	異なるリリースレベル間の複製	27
2.3.5	違いのあるノード間の複製	27
2.3.6	違いのあるノード間の比較	29
2.3.7	アプリケーションの一般的なルール	29
2.3.8	タイミングの考慮事項と同期レプリケーション	30
2.3.9	アプリケーションテスト	30
2.3.10	パフォーマンスのテストとチューニング	34
2.3.11	適合性の評価	35
2.4	PostgreSQL configuration for BDR	36
2.4.1	BDR の PostgreSQL 設定	36
2.4.2	BDR 固有の設定	37
2.5	Node Management	46
2.5.1	BDR グループの作成と参加	47
2.5.2	ウィットネスノード	49
2.5.3	ロジカルスタンバイノード	50
2.5.4	フィジカルスタンバイノード	51
2.5.5	サブグループ	53
2.5.6	デコードワーカー	54
2.5.7	ノードの再起動とダウンしたノードの回復	55
2.5.8	BDR グループからのノードの削除	58
2.5.9	BDR トポロジのリスト	59
2.5.10	ノード管理インターフェース	60
2.5.11	ノード管理コマンド	71

2.6	DDL Replication	73
2.6.1	DDL レプリケーションオプション	74
2.6.2	BDR システムでの DDL の実行	75
2.6.3	DDL ロックの詳細	75
2.6.4	DDL の影響の最小化	77
2.6.5	ダウンロードでの DDL の処理	78
2.6.6	ステートメント固有の DDL レプリケーションに関する懸念	79
2.6.7	BDR DDL コマンド処理マトリックス	79
2.7	Security and Roles	95
2.7.1	カタログテーブル	96
2.7.2	BDRFunctions と演算子	96
2.7.3	カタログオブジェクトに対する特権の付与	96
2.7.4	ロール管理	96
2.7.5	ロールとレプリケーション	97
2.7.6	接続ロール	97
2.7.7	権限制限	98
2.7.8	BDR のデフォルト/事前定義されたロール	99
2.7.9	検証	102
2.8	Conflicts	104
2.8.1	競合はどのように発生するか	105
2.8.2	競合の種類	106
2.8.3	競合の回避または許容	117
2.8.4	競合検出	118
2.8.5	競合の解決	122
2.8.6	競合ロギング	126
2.8.7	LiveCompare によるデータ検証	128
2.9	Sequences	128
2.9.1	BDR グローバルシーケンス	128
2.9.2	UUID、KSUUID、およびその他のアプローチ	135
2.9.3	グローバルシーケンス管理インターフェイス	137
2.9.4	KSUUID v2Functions	142
2.9.5	KSUUID v1Functions	143
2.10	Column-level conflict detection	145
2.10.1	列レベルの競合解決の有効化と無効化	146
2.10.2	DDL ロック	147
2.10.3	現在とコミットのタイムスタンプ	148
2.10.4	列のタイムスタンプの検査	148
2.10.5	CRDT データ型を使用した列の競合の処理	150
2.10.6	注意事項	150
2.11	Conflict-free Replicated Data Types	151
2.11.1	状態ベースおよび操作ベースの CRDT	154
2.11.2	ディスク領域の要件	155
2.11.3	CRDT タイプと競合処理	155
2.11.4	CRDT タイプと競合レポート	156
2.11.5	CRDT 値のリセット	156
2.11.6	実装された CRDT データ型	157
2.12	Transaction streaming	163

2.12.1	BDR の機能強化	163
2.12.2	構成	164
2.12.3	モニタリング	166
2.13	Durability & Performance Options	166
2.13.1	概要	166
2.13.2	用語と定義	167
2.13.3	比較	168
2.13.4	操作の内部タイミング	169
2.13.5	構成	170
2.13.6	計画的なシャットダウンと再起動	170
2.13.7	BDR を使用したレガシー同期レプリケーション	170
2.14	Group Commit	171
2.14.1	要件	171
2.14.2	構成	171
2.14.3	リファレンス	173
2.15	Eager Replication	175
2.15.1	要件	175
2.15.2	使用法	175
2.15.3	エラーハンドリング	176
2.15.4	一般的な熱心な複製の効果	177
2.16	Commit At Most Once	178
2.16.1	要件	179
2.16.2	構成	179
2.16.3	障害シナリオ	180
2.16.4	アプリケーション使用	182
2.16.5	DDL とグローバルロックとの相互作用	187
2.16.6	CAMO の制限	188
2.16.7	パフォーマンスへの影響	189
2.16.8	クライアントアプリケーションのテスト	189
2.16.9	CAMO とグループコミット	189
2.17	Lag Control	189
2.17.1	要件	190
2.17.2	構成	190
2.17.3	概要	190
2.17.4	取引申請	191
2.17.5	制限	191
2.17.6	注意事項	192
2.17.7	組織目標の達成	192
2.18	AutoPartition	192
2.18.1	パーティションの自動作成	193
2.19	Timestamp-Based Snapshots	200
2.20	Replication Sets	201
2.20.1	レプリケーションセットの使用	201
2.20.2	パーティションテーブルの動作	202
2.20.3	外部キーの動作	202
2.20.4	レプリケーションセットの管理	203
2.20.5	レプリケーションセットのメンバーシップ	208

2.20.6	レプリケーションセットのリスト	210
2.20.7	DDL レプリケーションのフィルタリング	212
2.21	Stream Triggers	215
2.21.1	Apply 中に実行をトリガー	215
2.21.2	行型の用語	217
2.21.3	競合のトリガー	217
2.21.4	トリガーを変換する	218
2.21.5	ストリームトリガー変数	219
2.21.6	情報機能	220
2.21.7	行の内容	223
2.21.8	トリガーノート	223
2.21.9	ストリームトリガー操作インターフェイス	224
2.21.10	ストリームトリガーの例	226
2.22	Explicit Two-Phase Commit (2PC)	229
2.22.1	使用法	229
2.23	Catalogs and views	230
2.23.1	ユーザーに表示されるカタログとビュー	230
2.23.2	内部カタログとビュー	267
2.24	BDR System Functions	269
2.24.1	バージョン情報 Functions	269
2.24.2	システム情報 Functions	269
2.24.3	システムと進行状況の情報パラメーター	269
2.24.4	コンセンサスファンクション	270
2.24.5	ユーティリティ Functions	273
2.24.6	グローバルアドバイザーロック	281
2.24.7	モニタリング Functions	283
2.24.8	内部 Functions	286

第1章 概要

BDR は、マルチマスターレプリケーションとデータディストリビューションを高度な競合管理、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提供する PostgreSQL の拡張機能であり、最大ファイブナインの高可用性を備えた分散 PostgreSQL クラスターを有効にします。

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

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、EDB Postgres Advanced 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 拡張機能として展開できます。サポートされているバージョンの組み合わせの詳細については、

Comptibility Matrix を参照してください。

一部のキー BDR 機能は、ターゲットの PostgreSQL データベースサーバ内で使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 データベースサーバも採用することが不可欠です。例、BDR 機能「コミット アットワンス (CAMO)」が BDR のお客様のユースケースにとってミッションクリティカルである場合、CAMO のハンドルに必要なコア機能がないため、コミュニティの 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、

Choosing a Postgres distribution にあります。

BDR は、ネイティブに近い PostgreSQL との互換性を提供します。ただし、一部のアクセスパターンは、マルチノード設定 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 Release Notes

BDR は、マルチマスターレプリケーションとデータディストリビューションを高度な競合管理、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提供する PostgreSQL の拡張機能であり、非常に高可用性の分散 PostgreSQL クラスターを有効にします。

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

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

2.1.1 BDR 4.1.0

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

Type	Category	Description
Feature	Reliability and operability	BDR ノードでの Postgres のインプレースメジャーアップグレードをサポート この BDR リリースには、新しいコマンドラインユーティリティ <code>bdr_pg_upgrade</code> が含まれています。 <code>pg_upgrade</code> を使用して、BDR ノードで Postgres のメジャーバージョンアップグレードを実行します。 BDR クラスターで Postgres のメジャーバージョンアップグレードを行うために必要な時間とネットワーク帯域幅を削減します。
Feature	Performance and scalability	レプリケーションラグ制御 トランザクションのコミットが調整されるまでのレプリケーションラグしきい値の構成を追加します。これにより、同期レプリケーションに伴うすべてのトランザクションにレイテンシのインパクトを与えることなく、RPO を制限できます。
Feature	UX / Initial experience	デフォルトでの分散シーケンス <code>bdr.default_sequence_kind</code> のデフォルト値が <code>"distributed"</code> に変更されました。これは、 <code>"bigserial"</code> に SnowflakeId を使用し、 <code>"serial"</code> 列タイプに Galloc シーケンスを使用します。
Feature	UX	簡素化された同期レプリケーション構成 同期レプリケーションオプションを指定するための新しい構文 BDR グループと SQL ベースの管理（構成ファイルではなく）。 将来のバージョンでは、これは現在の Eager Replication と CAMO 構成オプションにも置き換わります。
Feature	High availability and disaster recovery	グループコミット 新しい構成構文で構成できる初期の種類の同期的コミット。
Feature	High availability and disaster recovery	ローカルモードへのフラグの切り替えに Raft リクエストが必要になることを許可します。に接続して、ローカルモードにスイッチします。</p>

次のページに続く

表 2 – 前のページからの続き

Type	Category	Description
Feature	High availability and disaster recovery	既にデタッチされたパーティションの `ALTER TABLE ... DETACH PARTITION CONCURRENTLY` でレプリケーションを続行できるようにする (RT78362) 同じインデックスが既に存在する場合に BDR 4 が `CREATE INDEX CONCURRENTLY` を処理する方法と同様に、`ALTER TABLE ... DETACH PARTITION CONCURRENTLY` が既にデタッチされたパーティションのレシーバである場合にレプリケーションを続行できるようになりました。
Feature	User Experience	DDL フィルターに追加のフィルタリングオプションを追加します。 DDL フィルターを使用すると、さまざまな DDL ステートメントをさまざまなレプリケーションセットにレプリケーションできます。レプリケーションセットのテーブルメンバーシップにより、さまざまなテーブルの DML をさまざまなレプリケーションセットを介してレプリケートできるのと同様に。 このリリースでは、DDL フィルタを使い make する新しいコントロールが追加されています。 - query_match -定義されたクエリーがこの正規表現とマッチする必要がある場合 - Exclusive - true の場合、他の一致したフィルタは考慮されません (つまり、排他フィルタのみが適用されます)。マルチプルの排他フィルタがマッチする場合、エラーをスローします
Feature	User Experience	`bdr.lock_table_locking` 構成変数を追加します。 有効にすると、これにより `LOCK TABLE` コマンドの動作が変更され、グローバル DML ロックが取得されます
Feature	Performance and scalability	LCR セグメントファイル のバッファ書き込みを実装します。これにより、I/O が削減され、Decoding Worker の CPU 使用率が向上します。
Feature	User Experience	競合検出のために部分的一意インデックス検索のサポートを追加しました (RT78368)。 ただし、式のインデックスは競合検出でまだサポートされていません。

次のページに続く

表 2 – 前のページからの続き

Type	Category	Description
Feature	User Experience	<code>`bdr.stat_subscription`</code>に追加の統計を追加します。
 - <code>nstream_insert</code> => ストリーミングされたトランザクションの INSERT の数
 - <code>nstream_update</code> => ストリーミングされたトランザクションの UPDATE の数
 - <code>nstream_delete</code> => ストリーミングされたトランザクションの数
 - <code>nstream_truncate</code> => ストリーミングされたトランザクションの TRUNCATE のカウント
 - <code>npre_commit_confirmations</code> => CAMO を使用する場合はプリコミット確認
 - <code>npre_commit</code> => プリコミットのカウント
 - <code>ncommit_prepared</code> => 2PC での準備されたコミットの数
 - <code>nabort_prepared</code> => 2PC での準備されたトランザクションのアボートの数
Feature	User Experience	<code>execute_locally</code> オプションを <code>bdr.replicate_ddl_command</code> に追加しました (RT73533)。<p>これにより、ローカルで実行せずに他のグループにレプリケーションするための <code>ddl</code> コマンドをオプションでキューイングできます。</p>
Feature	User Experience	<code>`bdr.alter_subscription_disable()`</code>に <code>`fast`</code> 引数を追加 (RT79798) <p>引数は <code>`immediate`</code> の動作にのみ影響します。 <code>`true`</code> (デフォルト) に設定すると、現在の作業を終了させずにワーカーを停止します。</p>
Feature	User Experience	すべてのタイプのワーカーの <code>`bdr.worker_error`</code> レコードを永続的に保持します。<p>BDR は、レシーバとライタのエラーが以前にエラーになった LSN を複製できたときに使用されました。ただし、これは他のワーカーの動作と一貫性がありませんでした。他のワーカーのエラーは永続的なものであり、過去の問題のトラブルシューティングも困難になりました。したがって、すべてのワーカータイプの最後のエラーレコードを永続的に保持します。</p>
Feature	User Experience	<code>`bdr.{add,remove}_camo_pair`</code> 関数を簡素化して、<code>void</code> を結果ます。
Feature	Initial Experience	グローバルロックを取得する前に接続性/ラグチェックを追加します。<p>明らかな接続性の問題がある場合に、アプリケーションまたはユーザがロックタイムアウトを取得するまで数分待つ必要があります。</p><p>、WARNING (デフォルト) または ERROR。</p>

次のページに続く

表 2 – 前のページからの続き

Type	Category	Description
Feature	Initial Experience	デフォルトでは、競合のみを競合ログログテーブルに記録しデフォルト。デフォルトではサーバーログファイルに記録されなくなりましたが、これはオーバーライドできます。
Feature	User Experience	ノード結合中のリモートエラーのレポートを改善します。
Feature	Reliability and operability	自動パーティションワーカーの最大昼寝時間を構成可能にします。
Feature	User Experience	autopartition で指定された上限までパーティションを要求する機能を追加します。
Feature	Initial Experience	サブスクリプション専用ノードでレプリケート DDL を実行しないでください。複製する場所がないため、複製しようとしても失敗します。これは、ロジカルスタンバイの動作と同じです。
Feature	User Experience	`bdr.accept_connections` 構成変数を追加します。 `false` の場合、BDR 出力プラグインを使用したレプリケーションスロットへの walsender 接続が失敗します。これは、主にバックアップからの単一ノードの復元中に役立ちます。
Bug fix	Reliability and operability	非 CAMO パートナー BDR ノードで `lock_timeout` を構成したままにします CAMO パートナーは、オリジンノードからトランザクションを適用するときに低い `lock_timeout` を使用します。これは、CAMO パートナーだけでなく、すべての BDR ノードに対して誤って行われたため、通常の BDR ノードの pglogical ライタプロセスで誤った `lock_timeout` エラーが発生した可能性があります。
Bug fix	User Experience	CAMO / Eager 確認待機の適切な待機イベントを表示する (RT75900) デフォルトの「不明な待機イベント」ではなく、`bdr.stat_activity` の正しい「BDR 準備フェーズ」/「BDR コミットフェーズ」を表示します。
Bug fix	User Experience	bdr.run_on_nodes のログを削減する (RT80973) 関数の「run_on_nodes」バリエーションで終了した場合、`bdr.ddl_replication` をオフに設定しログ。これにより、モニタリング機能のログの洪水がなくなります。
Bug fix	Reliability and operability	バイナリネットワークエンコーディングをサポートしていない複合型の配列と組み込み型の配列のレプリケーションを修正

次のページに続く

表 2 – 前のページからの続き

Type	Category	Description
Bug fix	Reliability and operability	ブートストラップ中に作成されたデータ型のレプリケーションを修正
Bug fix	Performance and scalability	WAL デコーダーによって処理された実行中のトランザクションレコードの終了 LSN を確認して、WAL デコーダースロットが最新の状態に保たれ、WAL 送信者がタイムリーに候補を取得します。
Bug fix	Reliability and operability	分割ノードで自動パーティションタスクが完了するのを待たない
Bug fix	Reliability and operability	フラッシュ位置を物理的スロットのみに報告するときに `bdr.standby_slot_names` チェックを制限します (RT77985、RT78290) <p>それ以外の場合、`bdr.standby_slot_names` の使用時に切断されたノードが存在する場合にフラッシュの進行状況が報告されません。
Bug fix	Reliability and operability	wal_receiver_timeout が近い場合は、walsender にフィードバックの返信を要求します
Bug fix	Reliability and operability	BDR 拡張機能で自動パーティション分割レプリケーションれたテーブルの依存関係を複数回レコードしないでください。 これにより、自動パーティションから「予期しない数の拡張機能依存関係レコード」エラーが発生し、これが発生すると競合が発生します。 `pg_depend` から二重の依存関係を削除して、既存の壊れたテーブルを手動で修正する必要があることに注意してください
Bug fix	Reliability and operability	レシーバでのキープアライブハンドリングを改善します。 ストリーミングトランザクションの途中でキープアライブに基づいて位置を更新しないでください。クラッシュ時にデータを失う可能性があります。 改善すべき新しいフラッシュとシグナリングロジックもあります低 TPS シナリオでのレイテンシ。
Bug fix	Reliability and operability	BDR ノードがデータベースに存在する場合にのみ、post `CREATE` コマンドを処理します。
Bug fix	Reliability and operability	競合履歴テーブルに ERROR 競合をログに記録しないでください。
Bug fix	Reliability and operability	マルチ挿入 (COPY) 中に conflict_slot が解放された後に使用されていた segfault を修正しました (RT76439)。

次のページに続く

表 2 – 前のページからの続き

Type	Category	Description
Bug fix	Reliability and operability	遅延スタンバイスロットに直面したときにワルセnderプロセスがスピンしないようにします (RT80295、RT78290)。 スタンバイスロットの1つが遅延している場合にワルセnderプロセスがCPUの100%を消費するように、ラッチをリセットするシグナリングを修正します。
Bug fix	Reliability and operability	`bdr.standby_slot_names`使用時の`wal_sender_timeout`のハンドリングを修正 (RT78290)
Bug fix	Reliability and operability	ALTER TABLE で基になるリレーションを1回だけロックするようにします (RT80204)。
Bug fix	User Experience	`bdr.monitor_local_replslots`の切断されたスロットのレポートを修正しました。これらは、以前に切断されたのではなく、欠落していると報告された可能性があります。
Bug fix	Reliability and operability	`bdr.get_subscription_progress()`ファンクションおよびそのファンクションを使用する`bdr.subscription_summary`でのダウンしたサブスクリプションのタイムスタンプレポートを修正しました。以前はゴミの値をレポートしていました。
Bug fix	Reliability and operability	BDR ワーカーのさまざまな場所での snapshot のハンドリングを修正しました。
Bug fix	User Experience	モニタリング機能で使用可能な値がない場合、タイムスタンプと LSN を NULL として報告することについて、より一貫性を保ちます。
Bug fix	Reliability and operability	ライタプロセスを切り替えるときのログ情報を削減します。
Bug fix	Reliability and operability	PG コマンドラインで設定パラメータが指定された場合、スーパーユーザチェックを行わないでください。そこではまだトランザクションを実行できません。その段階でスーパーユーザが変更されることが保証されています。
Bug fix	Reliability and operability	バイト単位のラグサイズを計算するには 64 ビットを使用します。ラグのラージオーバーフローのリスクをなくす。

アップグレード

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

- 4.0.0 以降
- 3.7.15
- 3.7.16

アップグレード処理前に、Upgrade Guide に記載されているプロセスと制限を読んで理解 make てください。

2.1.2 BDR 4.0.2

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

Type	Category	Description
Improvement	Reliability and operability	<code>bdr.max_worker_backoff_delay</code> を追加 (BDR-1767) これにより、バックオフ遅延のハンドリングが変更され、エラーが繰り返される場合に <code>bdr.min_worker_backoff_delay</code> から <code>bdr.max_worker_backoff_delay</code> に指数関数的に増加します。これにより、ログの spam が減少し、場合によっては不要な接続試行も防止されます。
Improvement	User Experience	<code>bdr.replicate_ddl_command()</code> に <code>execute_locally</code> オプションを追加 (RT73533) これにより、ローカルで実行せずに他のグループにレプリケーションするための ddl コマンドをオプションでキューイングできます。
Improvement	User Experience	JOIN 中のコンセンサス問題の ERROR を WARNING に変更する これらの一時的なエラーの報告は、 <code>bdr.worker_errors</code> にも表示されるため混乱を招きました。これらは WARNING に変更されました。
Bug fix	Reliability and operability	WAL デコーダーは、実行中のトランザクションレコードの終了 LSN を確認します (BDR-1264) WAL デコーダーによって処理された実行中のトランザクションレコードの終了 LSN を確認して、WAL デコーダースロットが最新の状態を維持し、WAL 送信者がタイムリーに候補を取得します。

次のページに続く

表 3 – 前のページからの続き

Type	Category	Description
Bug fix	Reliability and operability	分割ノードで自動パーティションタスクが完了するまで待機パートないでください（BDR -1867） .
Bug fix	User Experience	並列結合中のノード名前の再利用のハンドリングを改善しました（RT74789） ノードに世代番号があるため、ノードレコードが snapshot のパートとして受信された場合でも名前の再利用を簡単に識別できます。
Bug fix	Reliability and operability	BDR マネージャプロセスでのノード管理中のロックと snapshot の使用を make 更新 snapshotmake（マルチプル） がアカウントされます。
Bug fix	Reliability and operability	ローカルノードのドロップでのカタログのクリーンアップを改善します プライマリだけでなくすべてのグループを削除し、すべてのノード状態の履歴情報も削除します。
Bug fix	User Experience	ロジカル <p>の結合リクエストのエラーチェックを改善
Bug fix	Reliability and operability	コンセンサスでのさまざまなタイムアウトとスリープのハンドリングを改善します これにより、多くのコンセンサスリクエストや時間のかかるコンセンサスリクエストを処理するときに必要な新しいコンセンサス投票の量が削減されます。例、新しいノードの結合。
Bug fix	Reliability and operability	`wal_receiver_timeout` のハンドリングを修正しました（BDR -1848） BDR 3.7 および 4.0 でのリグレーションのため、`wal_receiver_timeout` は正しくトリガーされませんでした。
Bug fix	Reliability and operability	フラッシュ位置を物理的スロットのみに報告する場合の `bdr.standby_slot_names` チェックを制限します（RT77985、フラッシュ）<p>それ以外の場合、`bdr.standby_slot_names` の使用時に切断されたノードが存在する場合、
Bug fix	Reliability and operability	ブートストラップ中に作成されたデータ型のレプリケーションを修正しました（BDR-1784）
Bug fix	Reliability and operability	バイナリ転送をサポートしていないビルトイン型の配列のレプリケーションを修正しました（BDR-1042）
Bug fix	Reliability and operability	CAMO が使用されていない場合に CAMO 構成の警告が表示されないようにしました（BDR-1825）

アップグレード

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

- 4.0.0 以降

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

アップグレード処理前に、Upgrade Guide に記載されているプロセスと制限を読んで理解 make てください。

2.1.3 BDR 4.0.1

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

Type	Category	Description
Improvement	Reliability and operability	CAMO パートナーの接続試行の頻度を減らします。 構成を確認し、トランザクションのステータスを確認するために CAMO パートナーに接続できない場合、すぐにリトライしないでください (完全にビジーな pglogical マネージャプロセスにつながります)。1 分に 1 回。
Improvement	RPerformance and scalability	LCR セグメントファイルのバッファ読み取りを実装します (BDR-1422) LCR セグメントファイルのバッファリングを実装して、一度にマルチプルの LCR チャンクを読み取ることができます。これにより、Decoding Worker を使用する際の Wal Senders の I/O が削減され、CPU 使用率が向上します。
Improvement	Performance and scalability	不要な LCR セグメントの読み取りを回避 (BDR-1426) BDR は、少なくとも 1 つの使用可能な場合に新しい LCR セグメントのみを読み取ろうとします。これにより、Decoding Worker が有効になっているときの I/O ロードが削減されます。
Improvement	Performance and scalability	結合中の初期 COPY を含む COPY レプリケーションのパフォーマンスが、テーブルパーティションで大幅に改善されました (BDR-1479) ラージテーブルの場合、これによりロード時間が 1 オーダー以上改善されます。
Bug fix	Performance and scalability	パラレル適用ワーカーの選択を修正します (BDR-1761) これにより、パラレル適用が再度動作します。4.0.0 では、このバグのため並列適用は有効になりませんでした。

次のページに続く

表 4 – 前のページからの続き

Type	Category	Description
Bug fix	Reliability and operability	`bdr.camo_pairs` の Raftsnapshot ハンドリングを修正しました (BDR-1753) 以前のリリースでは、Raftsnapshot を介して受信したときに CAMO ペア構成に変更を正しく伝播しませんでした。
Bug fix	Reliability and operability	アップグレード後に BDR 3.7 からの Raft スナップショットを正しくハンドル (BDR-1754)
Bug fix	Reliability and operability	クラスターのインプレースアップグレードを実行する機能を除外しながら、snapshot の `bdr.camo_pairs` をアカウントに入れて CAMO 構成済みクラスターをアップグレードする (CAMO とは関係のないアップグレードの制限のため)。
Bug fix	Reliability and operability	タイムアウト後に CAMO からローカルモードにのみ切り替える (RT74892) CAMO 保護からローカルモードに切り替えるときにロード推定値を使用しないでください。推定値は、ローカルモードから CAMO 保護に戻す場合にのみ使用してください (CAMO パートナーの遅延によるトグル操作を防ぐため)。
Bug fix	Reliability and operability	パブリッシュされたレプリケーションセットリストが変更された場合のレプリケーションセットキャッシュの無効化を修正しました (BDR - 1715) 以前のバージョンでは、サブスクリプションが再接続されるまでにパブリッシュする必要があるレプリケーションセット (および結果としてどのテーブル) に関する古い情報を使用できました。
Bug fix	Reliability and operability	新しいチャンクが使用される場合、同時実行性の高い状況で galloc シーケンスによってローカルに生成される値のマルチプルを防止しシーケンス (RT76528) 新しい値を待っているセッション。これは修正されました。
Bug fix	Reliability and operability	ストリーミングトランザクションでのメモリーリークに対処します (BDR-1479) ラージトランザクションの場合、ストリーミングトランザクション機能を使用すると、メモリ使用量と I/O が大幅に削減されます。これにより、主に COPY レプリケーションのパフォーマンスが向上します。

次のページに続く

表 4 – 前のページからの続き

Type	Category	Description
Bug fix	Reliability and operability	ノードが分離しているときにキャッチアップソースが変更された場合、ノードの BDR フェーズの後にスロットを残さないでください（BDR ノード） すべてのノードのデータの一貫性を保つオーダーに、そのノードから欠落している変更を残りのノード間で転送します。これには、追加のレプリケーションスロットを一時的に作成する必要があります。通常、このレプリケーションスロットはキャッチアップフェーズの最後に削除されますが、変更のソースノードを変更する必要がある特定のシナリオでは、このスロットは以前に取り残されている可能性があります。このバージョンから、このスロットは常に正しく削除されます。
Bug fix	Reliability and operability	BDR グループにノードが 1 つしかない場合は、グループスロットを前方に移動します これにより、グループが単一の BDR ノードだけで実行されたままになった場合の WAL の蓄積によるディスクの枯渇を防ぎます。これは推奨される設定ではありませんが、WAL の蓄積は意図的なものではありません。
Bug fix	Reliability and operability	BDR グループにノードが 1 つしかない場合の Raft プロトコルバージョン それ以外の場合、シングルノードクラスターは、別のノードが追加されるまで常に最も古いサポートプロトコルを使用します。これにより、その単一のノードで使用可能な機能セットが制限される可能性があります。

アップグレード

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

- 3.7.14
- 4.0.0 以降

アップグレード処理前に、Upgrade Guide に記載されているプロセスと制限を読んで理解 make てください。

2.1.4 BDR 4.0.0

BDR 4.0 は BDR の新しいメジャーバージョンであり、このリリース番号で採用されたのはセマンティックバージョニングです（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。

Type	Category	Description
Feature	Compatibility	EDB Postgres Advanced 14 の BDR は、以前は EDB Postgres Extended でのみ使用できた次の機能をサポートするようになりました。
- Commit At Most Once - ノードに障害が発生した場合でも、アプリケーションが各トランザクションを 1 回のみコミットのに役立つ一貫性機能
- 積極的なスケラビリティ- 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します。 />- 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、特定の時刻に表示された、または表示されるデータを取得します。
- 行フリーズの自動化された動的構成 特定のコーナーケースでの UPDATE / DELETE 競合の解決の一貫性を向上させるため
- 評価チェック
- 欠落しているパーティションをエラーではなく競合としてハンドリングするためのサポート
- NOT VALID コンストレインと ALTER TABLE の高度な DDL 処理

次のページに続く

表 5 – 前のページからの続き

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

次のページに続く

表 5 – 前のページからの続き

Type	Category	Description
Feature	Performance	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。 <p>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` 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るため、並列性が向上します。さらに、ライタに直接ストリーミングする場合、トランザクションを同時に適用することもできます。これにより、並列性がさらに向上します。 </p><p>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、デコードされた変更はコミットされる前にダウンストリームに送信されます。変更は一連のファイルに書き込まれ、トランザクションが最終的にコミットしたときに適用されます。トランザクションが中断すると、変更は破棄されるため、アップストリームとダウンストリームの両方でリソースが無駄になります。 </p><p>サブトランザクションも自動的に処理されます。 </p><p>この機能は、PostgreSQL 14、EDB Postgres Extended 13 以降、および EDB Postgres EDB Postgres Advanced 14 で利用できます。Postgres のどのバージョンで利用できる機能の詳細については、:ref:`Choosing a Postgres distribution </pgd/latest/choosing_server/>` 付録を参照してください。 </p>
Feature	Compatibility	標準版とエンタープライズエディションの間の以前のバージョンの BDR に存在した違いは削除されました。BDR 4.0 では、サポートされている Postgres ディストリビューションとバージョン、つまり PostgreSQL v12-14、EDB Postgres Extended v12-14、および EDB Postgres Advanced 12-14 ごとに 1 つの拡張機能があります。 <p>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を使用して機能フラグを介して報告されます。詳細については、:ref:`Choosing a Postgres distribution </pgd/latest/choosing_server/>` を参照してください。 </p>

次のページに続く

表 5 – 前のページからの続き

Type	Category	Description
Feature	User Experience	BDR 4.0 拡張機能に対応する pglogical 4.0 拡張機能はありません。BDR には pglogical が必要なくなりました。 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。
Feature	Initial experience	一部の構成オプションでは、インストール後のエクスペリエンスを向上させるためにデフォルトが変更されています。 - 並列適用がデフォルトで有効になりました (2つのライター)。特にストリーミングが有効になっている場合、パフォーマンスが向上します。 - レプリケーションおよび `CREATE INDEX CONCURRENTLY` は、デフォルトで使用可能なすべてのノードに並列で (ストリーミングがサポートされている Postgres バージョンで) ライタにストリーミングされるようになりました。これらの操作の後。 - 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、サブスクリプションはエラー時に 1 秒に 1 回のみ接続をリトライします。
Feature	Reliability and operability	オリジンベースの競合検出を使用するテーブルのノード結合中の競合検出で誤検知の可能性が大幅に減少しました
Feature	Reliability and operability	CAMO ペアの構成を SQL カタログに移動する 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認識するには、CAMO 構成を個々のノードの postgresql.conf から Raft が管理する BDR システムカタログに移動します。これにより、例、別のアクティブノードの CAMO パートナーとして構成されているノードを誤って削除するのを防ぐことができます。 アップグレードプロセスの詳細については、:ref:`Upgrades chapter` </pgd/latest/upgrades/#upgrading-a-camo-enabled-cluster> を参照してください。 これは GUC `bdr.camo_partner_of` と `bdr.camo_origin_for` を 非推奨 にし、関数 `bdr.get_configured_camo_origin_for()` と `get_configured_camo_partner_of` を `bdr.get_configured_camo_partner` に置き換えます。

アップグレード

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

- 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読んで理解 make てください。

2.2 Architectural Overview

BDR は、メッシュトポロジを使用して、疎結合マルチマスターロジカルレプリケーションを提供します。これは、任意のサーバーに書き込むことがパート、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ることを意味します。



図 1: node diagram

デフォルトでは、BDR は非同期レプリケーションを使用し、ローカルコミット後にのみ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。

2.2.1 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 1 つのノードグループがあります。

マルチプルマスター

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ホットまたはウォームスタンバイとは異なります。

いつもすべてのマスターに書き込む必要はありません。ほとんどの場合、1 つのマスタのみに直接書き込む構成です。

デフォルトでは非同期

1 つの BDR ノードで行われた変更は、ローカルにコミットされるまで他のノードにレプリケートされません。その結果、データは常にすべてのノードで完全に同じではありません。一部のノードには、他のノードにまだ到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です。BDR では、マスターがマルチプルあり、その結果データストリームがマルチプルになるため、`synchronous_commit` と `synchronous_standby_names` を使用しても異なるノード上のデータが異なる場合があります。

メッシュトポロジ

BDR は、すべてのノードが他のすべてのノードに接続し、すべてのノードが相互に直接データを交換するメッシュネットワークを中心に構成されています。ノードの追加やノードの削除などの特別な状況を除き、BDR 内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送信されます。

論理レプリケーション

論理レプリケーションは、レプリケーション ID（通常はプライマリー）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レプリケーションを使用する物理的レプリケーションとは対照的に、**ロジカル**という用語を使用します。インデックスの変更は複製されないため、書き込みの増幅が回避され、帯域幅が削減されます。

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

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

レプリケートされたデータは、安全な場合にバイナリ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、ダウンしたノードをすばやく交換して続行できます。各スタンバイノードは、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は続行されます。ノードが回復すると、レプリケーションを中断したところからリスタートできます。

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

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

制限

BDR は、十分なハードウェアとネットワークで数百のノードを実行できますが、メッシュベースの展開では、通常、1つのクラスターで32を超えるノードを実行することはお勧めしません。各マスタノードは、マルチプルの物理的またはロジカルスタンバイノードで保護できます。スタンバイノードの数に特に制限はありませんが、一般的な使用法はマスタごとに2〜3個のスタンバイを使用することです。スタンバイノードはメッシュネットワークに接続を追加しないため、32ノードの推奨には含まれません。

これは現在許可されている最大の Raft 接続であるため、BDR には現在 1000 以下のアクティブノードの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、BDR は、PostgreSQL インスタンスごとに1つの BDR データベースのみを使用する場合に最適です。

BDR クラスターの推奨されるノードの最小数は3です。2つのノードでは、いずれかのノードが動作を停止するとコンセンサスが動作を停止するためです。

2.2.2 アーキテクチャーオプションとパフォーマンス

BDR パフォーマンスの特徴付け

BDR は、それぞれが異なるパフォーマンスとスケーラビリティ特性を持つ多数の異なるアーキテクチャで構成できます。

グループは、2つ以上のノード（サーバー）で構成される BDR グループの基本的なビルディングブロックです。グループ内では、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提供します。各グループには、専用のレプリケーションセットが定義されています。グループがノードを失った場合、グループから既存のノードをコピーすることで簡単に修復/交換できます。

BDR はすべてのノードですべての書き込みをリプレイする必要があるため、ほとんどのテーブルがレプリケートされる場合、BDR グループにマスタノードを追加しても大幅な書き込みスループットの向上はありません。BDR 書き込みは一般に、SQL を介した Postgres クライアントからの書き込みよりも効果的であるため、パフォーマンスをいくらか向上させることができます。読み取りスループットは通常、ノード数に比例して増加します。

次のアーキテクチャを使用できます。

- マルチマスタ/シングルグループ
- 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ことです。最初にそれを調べましょう。

1つのグループ内の BDR マルチマスタ

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

データのコピーはどこにでもあるため、SELECT はローカルノードにのみアクセスする必要があります。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、可能な合計 SELECT スループットが直線的に増加します。

INSERT、UPDATE、および DELETE（DML）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オリジナルの実行より小さいため、純粋な書き込みワークロードをマルチプルのノードで同時に実行すると、マルチノードクラスターはシングルノードよりも多くの TPS をハンドルできます。

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あります。スループットは、アプリケーションが実際に表示する競合の量に依存します。競合が非常に少ないアプリケーションは、単一のノードよりも優れたパフォーマンスを発揮します。競合の多いアプリケーションは、単一ノードよりもパフォーマンスが低下する可能性があります。これらの結果は、マルチマスタテクノロジーと一致しており、BDR の側面や特殊性ではありません。

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

レプリケーションの遅延が最小限に抑えられるように、変更はすべてのノードに同時に送信されます。ノードを追加することは、レプリケーションにより多くの CPU を使用することを意味するため、新しいノードが追加されるたびにピーク TPS がわずかに低下します。

ワークロードがすべての CPU リソースを使用しようとする、レプリケーションのリソースが制限され、レプリケーションの遅延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つの個別のリージョンにある2つのグループから構築されています。各グループは HA と IS を提供しますが、一緒に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非常に高可用性の AlwaysOn と呼びます。

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

1つのノードがメインアプリケーションのターゲットです。他のすべてのノードはシャドウノード（または「読み書きレプリカ」）と呼ばれ、必要に応じて引き継ぐのを待っています。ノードが接続を失った場合、すぐにシャドウノードにスイッチて処理を続行します。グループに障害が発生した場合、他のグループに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つのノードにのみ書き込むため、間の競合の可能性はほぼゼロになり、その結果、パフォーマンスへのインパクトが大幅に削減されます。

CAMO はローカルグループ内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惰です。

セカンダリアプリケーションはシャドウノードに対して実行しますが、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は、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合わせてレプリケーションの CPU オーバーヘッドが制限され、他のグループへの帯域幅が最小限に抑えられます。

2.2.3 展開

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

手動展開は推奨されておらず、サポートされていない場合があります。

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

ログメッセージと文書は現在英語でのみ提供されています。

2.2.4 時計とタイムゾーン

BDR は、マルチプルのタイムゾーンのノードで動作するように設計されているため、真にワールドワイドなデータベースクラスタが可能になります。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、保証 = UTC を使用して、人間が読めるサーバーログにアクセスして比較できるようにすることをお勧めします。

サーバーの時計は、NTP または他のソリューションを使用して同期する必要があります。

他のいくつかのソリューションの場合と同様、クロックの同期はパフォーマンスにとって重要ではありません。クロックスキューは、オリジンの競合検出にインパクトを与える可能性があります。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。

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

BDR は行バージョンの競合検出も提供します。

2.3 Application use

ユーザの観点からアプリケーションについて学びます。

2.3.1 アプリケーションの動作

BDR は、あるノードで行われた変更の他のノードへの複製をサポートしています。

BDR は、デフォルトで、ソースノードから他のノードへの INSERT、UPDATE、DELETE および TRUNCATE 操作によるすべての変更をレプリケートします。すべてのトリガーとルールが処理された後、最終的な変更のみが送信されます。例、INSERT ... ON CONFLICT UPDATE は、オリジンで何が発生したかに応じて、挿入または更新を送信します。更新または削除がゼロ行に影響する場合、変更は送信されません。

INSERT は前提条件なしで複製できます。

他のノードで複製更新と削除については、影響を受ける一意の行を特定する必要があります。BDR では、テーブルに PRIMARY KEY が定義されているか、UNIQUE 制約、または特定の列に明示的な REPLICATION IDENTITY が定義されている必要があります。これらのいずれかが定義されていない場合、ワーニングが生成され、後の更新または削除は明示的にブロックされます。テーブルに REPLICATION IDENTITY FULL が定義されている場合、一意インデックスは必要ありません。その場合、更新と削除が許可され、ライブ、有効、非遅延であり、式または WHERE 句を持たない最初の非一意インデックスを使用します。それ以外の場合は、シーケンシャルスキャンが使用されます。

定義されたレプリケーション ID がなくても TRUNCATE を使用できます。TRUNCATE コマンドのレプリケーションはサポートされていますが、外部キーで接続されたテーブルのグループを切り捨てる場合は注意してください。切り捨てアクションをレプリケートする場合、サブスクライバーは、レプリケーションセットが定義されている場合を除き、オリジン、明示的に指定された、または暗黙的に収集されたテーブルのグループを切り捨てます。詳細と例については、

Replication Sets を参照してください。これは、影響を受けるすべてのテーブルが同じサブスクリプションのパートである場合に正しく機能します。ただし、サブスクライバーで切り捨てる

テーブルに、同じ（または任意の）レプリケーションセットの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、サブスクライバーに切り捨てアクションを適用すると失敗します。

INSERT、UPDATE、および DELETE コマンドによって暗黙的に取得される行レベルのロックは、変更が加えられると複製されます。INSERT、UPDATE、DELETE、および TRUNCATE コマンドによって暗黙的に取得されたテーブルレベルのロックもレプリケートされます。ユーザセッションによる明示的な行レベルロック（SELECT ... FOR UPDATE/FOR SHARE）はレプリケートされず、アドバイザリー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、他のノードにレプリケートされません。SERIALIZABLE のトランザクション分離レベルはサポートされていますが、マルチプルのノードに同時トランザクションが存在する場合、トランザクションはノード間でシリアル化されません。

DML がマルチプルのノードで同時に実行され、非同期レプリケーションで実行すると競合が発生する可能性があります。

[CRDT データ型を使用した列の競合の処理](#) で説明されているさまざまな回避メカニズムが可能です。

`:ref:`bdr.sequences`<bdr.sequences>` で説明されている、シーケンスには特別なハンドリングが必要です。

BYTEA 列のバイナリデータは正常にレプリケートされるため、最大ギガバイトのデータの「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。

実表から実表へのみレプリケーションできます。つまり、サブスクリプション側のソースとターゲットのテーブルは、ビュー、マテリアライズドビュー、または外部テーブルではなく、テーブルである必要があります。実テーブル以外のテーブルをレプリケートしようとすると、エラーが発生します。更新可能ビューを介して行われた DML 変更は、オリジンの実表に解決されてから、ターゲットの同じベーステーブルに適用され名前。

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し、パーティションを含む変更をダウンストリームに複製します。

デフォルトでは、トリガーはオリジンノードでのみ実行されます。例、INSERT トリガーはオリジンノードで実行され、ターゲットノードに変更を適用しても無視されます。ALTER TABLE ... ENABLE ALWAYS TRIGGER を使用して、実行時にオリジンノードとターゲットで複製されたとき（「適用時」）の両方で実行トリガーを指定できます。または、REPLICA オプションを使用して適用時にのみ実行します：ALTER TABLE ... ENABLE REPLICA TRIGGER。

一部の種類のトリガーは、テーブルに存在し、現在有効になっている場合でも、適用時に実行されません。実行されないトリガーの種類は次のとおりです。

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

BDR レプリケーションの適用は、システムレベルのデフォルトの search_path を使用します。レプリカトリガー、ストリームトリガー、およびインデックス式関数は、他の search_path 設定を想定できますが、適用時に実行されます。これを防ぐには、デフォルトの search_path のみを使用してオブジェクト参照を明確に解決するか、常にオブジェクトへの完全修飾参照（スキーマ.objectname など）を使用するか、影響を受ける関数に ALTER FUNCTION ... SET search_path = ... を使用して関数の検索パスを設定します。

BDR は、テキストまたはその他の照合可能なデータ型に関連する問題がないことを前提としています。つまり、使用中のすべての照合はすべてのノードで使用でき、デフォルトの照合順序はすべてのノードで同じです。変更のレプリケーションは等値検索を使用してレプリカ ID 値を見つけるため、一致しない照合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た場合、行フィルターは照合の違いの影響を受ける可能性があります。

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

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

PostgreSQL は、揮発性関数を含む CHECK() 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、以前と同じ結果が結果れない後続の再実行によりデータが発散します。

BDR は外部キーの使用を制限しません。カスケード FK は許可されます。

2.3.2 レプリケートされないステートメント

次のユーザコマンドはいずれも BDR によって複製されないため、それらの効果はローカル/オリジンノードでのみ発生します。

- カーソル操作 (DECLARE、CLOSE、FETCH)
- 実行コマンド (DO、CALL、PREPARE、EXECUTE、EXPLAIN)
- セッション管理 (DEALLOCATE、DISCARD、LOAD)
- パラメータコマンド (SET、SHOW)
- 制約操作 (SET CONSTRAINTS)
- ロックコマンド (LOCK)
- テーブルメンテナンスコマンド (VACUUM、ANALYZE、CLUSTER、REINDEX)
- 非同期操作 (NOTIFY、LISTEN、UNLISTEN)

NOTIFY SQL コマンドと `pg_notify()` 関数は複製されないため、フェイルオーバーの場合の通知は信頼できません。これは、サーバーがクラッシュしたときにトランザクションがコミットされた場合、フェイルオーバー時に通知が簡単に失われることを意味します。LISTEN を実行しているアプリケーションは、フェイルオーバーの場合に通知を逃す場合があります。これは標準の PostgreSQL レプリケーションにも当てはまり、BDR はまだこれを改善していません。CAMO および Eager Replication オプションでは、NOTIFY SQL コマンドまたは `pg_notify()` ファンクションは許可されません。

2.3.3 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しません。DML ステートメントによる変更をレプリケートします。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、volatile ステートメントの実行結果が複製されることを意味し、ステートメントベースのレプリケーションで発生する可能性のあるノード間の相違がありません。

DDL レプリケーションは DML とは異なります。DDL の場合、BDR はステートメントを複製し、すべてのノードで実行します。したがって、DROP TABLE IF EXISTS はローカルノードで何も複製しない場合がありますが、DDL レプリケーションが有効になっている場合、ステートメントは実行のために他のノードに送信されます。詳細については、

[DML および DDL レプリケーション](#) を参照してください。

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

2.3.4 異なるリリースレベル間の複製

BDR は、PostgreSQL のメジャーバージョンが異なるノード間で複製するように設計されています。この機能は、ダウンタイムなしでメジャーバージョンをアップグレードできるように設計されています。

BDR は、BDR ソフトウェアのバージョンが異なるノード間で複製するようにも設計されています。この機能は、ダウンタイムなしでバージョンアップとメンテナンスを行えるように設計されています。

ただし、クラスター内のメジャーバージョンのノードに結合することはできますが、クラスターが新しいプロトコルバージョンを使用している場合、マイナーバージョンのノードを追加することはできません。これはエラーを返します。

これらの機能は両方とも、特定の制限の影響を受ける可能性があります。既知の非互換性については、

[Release Notes](#) を参照してください。

2.3.5 違いのあるノード間の複製

デフォルトでは、DDL はすべてのノードに自動的に送信されます。[DML および DDL レプリケーション](#) で説明されているように、これを手動で制御でき、それを使用してノード全体のデータベーススキーマ間の違いを作成できます。

BDR は、ノード間のマイナー違いでもレプリケーションを続行できるように設計されています。これらの機能は、ダウンタイムなしでアプリケーションスキーマを移行できるように、またはロジカルスタンバイノードでレポートまたはテストを行えるように設計されています。

現在、レプリケーションにはすべてのノードで同じテーブル名前が必要です。将来の機能では、異なるテーブル名間のマッピングが許可される可能性があります。

ターゲット上のパーティションを変更する更新のサポートを含め、パーティションテーブルに複製する通常のテーブルである

[Replication Sets](#) を参照してください。

デフォルトでは、すべての列が複製されます。BDR は、列名前に基づいてデータ列を複製します。列の名前は同じでデータ型が異なる場合、それを可能にするキャストが定義されている場合、ソース型からターゲット型にキャストしようとします。

BDR は、列の数が異なるテーブル間の複製をサポートしています。

ターゲットにソースからの列が欠落している場合、BDR は `target_column_missing conflict` を発生させます。デフォルトの競合リゾルバは `ignore_if_null` です。これにより、NULL 以外の値が到着するとエラーがスローされます。または、`ignore` の競合リゾルバでノードを構成することもできます。この設定はエラーをスローしませんが、追加の列を黙って無視します。

ターゲットにソースレコードにない追加の列がある場合、BDR は `source_column_missing conflict` を発生させます。デフォルトの競合リゾルバは `use_default_value` です。追加の列に NULL（NULL 可能な場合）またはデフォルトの式がある場合、レプリケーションは続行されデフォルトが、そうでない場合はエラーをスローし、レプリケーションを停止します。

変換トリガーをテーブルで使用して、デフォルト値を提供したり、適用前にさまざまな方法で着信データを変更したりできます。

ソースとターゲットに異なるコンストレインがある場合、レプリケーションが試行されますが、ソースの行をターゲットに適用できない場合は失敗します。ここで行フィルターがヘルプます。

あるスキーマからよりリラックスしたスキーマにデータを複製しても、エラーは発生しません。スキーマからより制限の厳しいスキーマへのデータの複製は、潜在的な障害のソースになる可能性があります。これを解決する正しい方法は、よりリラックスした側に制約を配置して、悪いデータを入力できないようにすることです。こうすれば、レプリケーションによって不良データが届くことはないため、より制限的なスキーマへの変換が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、別のスキーマで同じ列が XML として定義されている場合、TEXT 列に CHECK 制約を追加して、テキストが XML であることを強制します。

各ノードで異なるインデックスを持つテーブルを定義できます。デフォルトでは、インデックス定義が複製されます。ノード間のインデックスの不一致はエラーを発生させません。[DML および DDL レプリケーション](#) を参照してください。

`fillfactor` や `toast_tuple_target` などの記憶域パラメーターは、テーブルのノード間で問題なく異なります。その例外は、テーブルのストレージパラメータ `user_catalog_table` の値はすべてのノードで同一である必要があることです。

レプリケートされるテーブルは、各ノードで同じユーザ/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。

ロールは、ノードごとに接続用に異なるパスワードを持つことができますが、デフォルトでは、ロールへの変更は各ノードに同期されます。[DML および DDL レプリケーション](#) を参照してください。

2.3.6 違いのあるノード間の比較

LiveCompare は、データベース上のデータを BDR ノードおよび非 BDR ノードと比較するためのツール。比較して最終結果に到達するには、少なくとも 2 つの接続が必要です。

LiveCompare 1.3 以降、`all_bdr_nodes` set で設定できます。これにより、クラスター内の個別のノードに関連するすべての DSN を明確にする必要がなくなります。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のは初期接続と出力接続だけです。`logical_replication_mode` を設定すると、すべてのノードがどのように通信しているかが示されます。

すべての構成は、例、`bdrLC.ini` 名前付けの `.ini` ファイルで行われます。 `/etc/2ndq-livecompare/` でこの構成ファイルのテンプレートを見つけます。

LiveCompare の実行中、N+1 個の進行状況バーが表示されます。N はプロセスの数です。すべてのテーブルのソースが取得されると、1 秒あたりのトランザクション数 (tps) が測定されると時間が表示されます。これは時間をカウントし続け、最後に見積もりを提供し、合計実行時間を提供します。

このツールは、テーブル、スキーマ、`replication_sets` など、多くのカスタマイズとフィルターを提供します。LiveCompare はコンテキスト情報を失うことなく stop-start を使用できるため、都合の良いときに実行できます。比較後、概要と DML スクリプトが生成されるため、確認できます。DML を適用して、見つかった違いを修正します。

2.3.7 アプリケーションの一般的なルール

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。アプリケーションで同じ一意の識別子を挿入、[ABA problem](#) と呼ばれます。

BDR は、行が現在の行、最後の行、またはそれよりも古い行であるかを知ることができません。

同様に、BDR はテーブル名を使用して変更が再生されるテーブルを識別するため、同じオブジェクト名を作成、削除、および後で再利用するアプリケーションにも同様の ABA 問題が存在します。

これらの問題により、アプリケーションが従うべきいくつかのシンプルルールが発生します。

- 行に一意の識別子を使用します (INSERT)。
- 一意の識別子を変更しないでください (UPDATE)。
- 削除された一意の識別子を再利用しないでください。
- ドロップされたオブジェクト名の再利用を避けます。

一般的に、これらのルールを破ると、データの異常や発散につながる可能性があります。アプリケーションは、特定の条件が満たされている限り、これらのルールを破ることができますが、注意してください。異常が発生する可能性は低いですが、不可能ではありません。例、ダウンしたノードを含むすべてのノードで DELETE が再生された限り、行の値を再利用できます。これは通常 1 秒より小さいに発生しますが、1 つのノードで重大な問題が発生して正常に再起動できない場合は、数日かかる場合があります。

2.3.8 タイミングの考慮事項と同期レプリケーション

デフォルトでは非同期であるため、同等な遅れる場合があります、クライアントがマルチプルの BDR ノードに接続したり、それらを切り替えたりして古いデータを読み取ることができます。

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

Postgres の同期レプリケーション機能は BDR でも利用できます。さらに、BDR は、より同期レプリケーションのための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さまざまなモードの概要と比較については、

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

2.3.9 アプリケーションテスト

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

- [TPAexec](#)
- [CAMO/フェイルオーバーオプションを使用した `pgbench`](#)
- [マルチノードアクセスを備えた `isolationtester`](#)

TPAexec

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

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

```
tpaexec test mycluster mytest
```

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

CAMO/フェイルオーバーオプションを使用した `pgbench`

EDB Postgres Extended では、ユーザーが CAMO または通常の BDR デプロイメントの使用中にフェイルオーバーテストを実行できるように `pgbench` が拡張しました。次のオプションが追加されました。

```
- m, --mode=regular|camo|failover
mode in which pgbench should run (default: regular)
```

(次のページに続く)

(前のページからの続き)

```
- -retry
retry transactions on failover
```

これらのオプションに加えて、ノードにフェイルオーバーの同等な接続情報を指定する必要があります。

- `-m camo` または `-m failover` を使用して、`pgbench` のモードを指定します。 `-m failover` 仕様を使用して、通常の BDR 展開でフェイルオーバーをテストできます。
- `--retry` を使用して、`-m failover` モードでフェイルオーバーが発生したときにトランザクションをリトライするかどうかを指定します。このオプションは、`-m camo` モードではデフォルトで有効になっています。

CAMO 環境の例を次に示します。

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

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

フェイルオーバーモードで `--retry` が指定されている場合、進行中のトランザクションが再試行されます。このシナリオでは、進行中のトランザクションのステータスを見つける方法はありません。

マルチノードアクセスを備えた `isolationtester`

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

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

分離テストは、PostgreSQL の同時動作を調べるためのテストです。これらのテストでは、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ります。これには、マルチプルの同時接続を管理する必要がありますため、通常の `pg_regress` プログラムを使用してテストすることはできません。「分離」名前は、オリジナルの動機がシリアライズブル隔離レベルをテストすることであったという事実に由来します。他のソートの同時動作のテストも追加されました。

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

`pg_isolation_regress` は `pg_regress` に似たツールが、`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。マルチプルのホストをパラメーターとして受け入れるように変更されました。次に、ホスト `conninfo` をサーバー名とともに `isolationtester` バイナリに渡します。分離テスターは、これらのサーバー名を仕様ファイルの各セッションで指定された名前と比較し、それぞれのサーバーで指定されたテストを実行します。

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

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

仕様は5つの部分で構成され、次のオーダーでテストされます。

```
server "<name>"
```

これは、セッションを実行するサーバーの名前を定義します。0 個以上のサーバー"`<name>`" 指定があります。名前に対応する `conninfo` は、`isolationtester` を実行するコマンドによって提供されます。これは `quickstart_isolationtest.md` で説明されています。この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が許可されます。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 サブミッションとして実行され、`VACUUM` などの一部のステートメントをそのようなブロックで他のステートメントと組み合わせることができないためです。

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

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

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

通常、スペックファイルには複数の「セッション」パートがあります。各セッションは独自の接続で実行されます。セッションパートは、セットアップ、ティアダウン、および1つ以上の「ステップ」の3つのパートで構成されています。セッションごとのセットアップとティアダウンの部分は、テストごとのセットアップとティアダウンと同じ構文ですが、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。

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

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

各ステップには次の構文があります。

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

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

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

置換行は、そのオーダーで実行されるステップのリストを指定します。置換行はいくつでも表示できます。置換行が指定されていない場合、テストプログラムは各セッションのステップの可能な順序を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実行します）。手動で指定された「置換」行のステップのリストは、実際には使用可能なステップの置換である必要はありません。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です。

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

1. メインセットアップパート

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

1. セッションごとのティアダウン

1. メインの分解スクリプト

選択された各ステップは、そのセッションに関連付けられた接続に送信されます。

前提条件となるすべての make コマンドを実行した BDR 環境で分離テストを実行するには、次の手順を実行します。

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、分離リグレーションテストを実行できます。

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

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

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

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

`Isolationcheck-makecheck` は現在、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、単一のインスタンスでの分離テストの実行をサポートしています。

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

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

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

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

2.3.10 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ます。これらの書き込みを各ノードに戻すと、パフォーマンスが低下しコスト。

まず、別のノードからの変更を再生するには CPU コスト、I/O コスト、および WAL レコードを生成します。SQL を再実行する必要があるため CPU オーバーヘッドが低いため、リソースの使用は通常オリジナルのトランザクションより小さいです。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、リプレイ時に I/O コストが発生する場合があります。

次に、変更を再生すると、ローカルのワークロードに対して競合が発生する可能性のあるテーブルレベルおよび行レベルロックが保持されます。競合のないレプリケートデータ型（CRDT）と列レベルの競合検出（CLCD）機能保証、同時更新でも正しい答えが得られますが、通常のロックオーバーヘッドは削除されません。ロックの競合が発生した場合は、更新の競合を回避するか、トランザクションをできるだけ短くしてください。大規模なトランザクションで頻繁に更新される行は、そのトランザクションのパフォーマンスにボトルネックを引き起こします。複雑なアプリケーションでは、スケーラビリティを維持するための配慮が必要です。

パフォーマンスに問題があると思われる場合は、ベンチマークツールを使用してパフォーマンステストを開発します。`pgbench` を使用すると、ユースケースに固有のカスタムテストスクリプトを作成できるため、SQL のオーバーヘッドを理解し、同時実行のインパクトを測定できます。

BDR の実行が遅い場合は、次のことをお勧めします。

1. 稼動システムの問題ケースにできる `make` 近づけて、`pgbench` のカスタムテストスクリプトを作成します。
2. 1 つのノードでスクリプトを実行して、ベースラインの数値を取得します。
3. 1 つのノードで実行したのと同じ合計数のセッションを使用して、稼動で発生するのと同じ数のノードでスクリプトを実行します。これは、マルチプルのノードに移動した結果を示しています。
4. これら 2 つのテストのセッション数を増やして、アプリケーションでの競合の増加の影響をプロットできるようにします。
5. テストがレプリケーションの遅延をアカウントに入れるのに十分な長さであることを確認します。
6. テスト中にレプリケーションの遅延が増大していないことを確認します。

通常の Postgres チューニング機能をすべて使用して、アプリケーションの重要な部分の速度を向上させます。

2.3.11 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、既に、または簡単に変更して BDR に準拠しています。アプリケーションを BDR 対応のセットアップにポイントできる評価アクティビティを行うことができます。BDR は、評価期間中に設定できるいくつかのノブを提供します。これらは、BDR 対応環境でのアプリケーションの適合性を判断するプロセスに役立ちます。

プライマリキー/レプリカアイデンティティの更新の評価

BDR は現在、UPDATE オペレーションによって PRIMARY KEY が変更された場合の競合解決を実行できません。プライマリキーは更新できますが、既存の値との競合が発生しないことを保証必要があります。

BDR は、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するための次の設定パラメータを提供します。

これらの構成パラメーターは評価にのみ使用します。単一ノードの BDR インスタンスで使用できますが、2つ以上のノードが相互にレプリケートする稼働 BDR クラスターでは使用しないでください。実際、評価パラメーターのいずれかが IGNORE 以外に設定されている場合、ノードがスタートしないか、新しいノードがクラスターに結合しない場合があります。

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

評価期間中にこのパラメータを有効にすることで、行のキー/レプリカアイデンティティ値の更新をログに記録できます。必要に応じて、そのような更新を潜在的にブロックすることもできます。例：

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

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

適用ワーカプロセスは、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です。これにより、BDR ライタプロセスがユーザトランザクションまたは相互に保持されているロックで待機する場合があります。

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

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

追跡できる 2 つのタイプのロックは次のとおりです。

- ユーザセッションによる明示的なテーブルレベルロック (`LOCK TABLE ...`)
- ユーザ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ことにより、このよう明示的ロックアクティビティを追跡（またはブロック）できます。例：

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

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
```

2.4 PostgreSQL configuration for BDR

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

2.4.1 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しているため、`logical` に設定する必要があります。
- `shared_preload_libraries` — `bdr` を含める必要がありますが、必要に応じてその前または後に他のエントリを含めることができます。ただし、`pglogical` は含めないでください。
- `track_commit_timestamp` — 競合する行のタイムスタンプを取得する競合を解決するには、`on` に設定する必要があります。

BDR では、これらの PostgreSQL 設定を適切な値に設定する必要があります。値はクラスターのサイズとスケールによって異なります。

- `logical_decoding_work_mem` — 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し、ローカルディスクに一時的に保存されます。デフォルトは 64 メガバイトですが、もっと大きく設定できます。
- `max_worker_processes` — BDR はレプリケーションとメンテナンスのタスクにバックグラウンドワーカーを使用するため、正常に動作するには十分なワーカーロットが必要です。データベース

プラスに、PostgreSQL のインスタンスプラスプラスのノードはデータベースのデータベース同等な。BDR グループ内の同等なごとに有効になっているライターごとに BDR グループからノードを削除するときに、一時的により多くのワーカプロセスが必要になる場合がありますノード。

- `max_wal_senders` —すべての同等なノードに 2 つ必要。
- `max_replication_slots` — `max_wal_senders` と同じ。
- `wal_sender_timeout` および `wal_receiver_timeout` —ノードが CAMO パートナーを切断または再接続と見なす速度を決定します。詳細は CAMO failure scenarios を参照してください。

N 個の同等なを持つグループの通常の実行では、BDR には N 個のスロットと WAL 送信者が必要です。同期中に、BDR は別の N-1 スロットと WAL 送信者を一時的に使用するため、この時折のピーク需要に合わせてパラメーターを十分に高く設定するようにしてください。

パラレル適用をオンにすると、スロット数を式*ライターから N スロットに増やす必要があります。これは、`max_replication_slots` もレプリケーション起点の最大数を設定し、パラレル適用の一部の機能はライタごとに追加のオリジンを使用するためです。

デコードワーカー が有効になっている場合、このプロセスには BDR グループごとに 1 つの追加のレプリケーションスロットが必要です。

単一のデータベースを持つ 4 ノードの BDR グループでの一般的な安全な推奨値は、`max_replication_slots` と `max_worker_processes` を 50 のように設定し、`max_wal_senders` を少なくとも 10 に設定することです。

これらのパラメーターを変更するには、ローカルノードを再起動する必要があります：`max_worker_processes`、`max_wal_senders`、`max_replication_slots`。

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

- `synchronous_commit` —BDR レプリケーションの耐久性とパフォーマンスに影響します。
`physical replication` と同様に。
- `synchronous_standby_names` —上記と同じ。

2.4.2 BDR 固有の設定

BDR 固有の構成設定を設定することもできます。特に断りのない限り、値はいつでも設定できます。

競合ハンドリング

- `bdr.default_conflict_detection` —新しく作成されたテーブルのデフォルトの競合検出メソッドを設定します。

`bdr.alter_table_conflict_detection()` と同じ値を受け入れます。

グローバルシーケンスパラメータ

- `bdr.default_sequence_kind` —デフォルトの *sequence kind* を設定します。デフォルト値は `distributed` です。これは、`snowflakeid` が `int8` シーケンス（つまり `bigserial`）に使用され、`galloc` シーケンスが `int4`（つまり `serial`）および `int2` シーケンスであることを意味します。

DDL ハンドリング

- `bdr.default_replica_identity` —新しく作成されたテーブルの `REPLICA IDENTITY` のデフォルト値を設定します。`REPLICA IDENTITY` は、更新または削除された行を識別するためにログ先行書き込みに書き込まれる情報を定義します。

受け入れられる値は次のとおりです。

- `DEFAULT` —プライマリキーの列の古い値を記録します（これは PostgreSQL のデフォルトの動作です）。- `FULL` —行のすべての列の古い値を記録します。- `NOTHING` —古い行に関する情報は記録しません。

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

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

`bdr.default_replica_identity` が `DEFAULT` であり、テーブルに `UNIQUE` 制約がある場合、自動的に `REPLICA IDENTITY` として選択されません。上記のように、テーブルの作成時または作成後に明示的に設定する必要があります。

テーブルのレプリカ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ネットワークで複製されるデータの量が増加します。

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

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

`bdr.ddl_replication = off` で DDL を実行するか、BDR 管理機能呼び出すと、管理者が介入するまでレプリケーションが停止する状況が発生する可能性があります。詳細は [DML および DDL レプリケーション](#) を参照してください。

`bdr.ddl_replication` が `off` に設定されている場合は常に、`LOG` レベルのログメッセージが PostgreSQL サーバログに出力されます。さらに、この設定が原因でキャプチャされた DDL コマンドまたは BDR レプリケーションファクションのレプリケーションがスキップされたときに、`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 `on`）。このパラメータを設定できるのはスーパーユーザだけです。この設定は、`bdr.ddl_replication` がオンになっている場合にのみ機能します。

外部の方法を使用せずにこれをオフにして、すべてのノードでロールが同期されていることを保証と、管理者が介入するまでレプリケーションされた DDL がレプリケーションを中断する場合があります。

詳細は [ロール操作ステートメント](#) を参照してください。

- `bdr.ddl_locking` —DDL のグローバルロックの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` またはスーパーユーザロールのみが設定できます。

可能なオプションは次のとおりです。

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

`bdr.ddl_replication` が `off` に設定されている場合は常に、LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定が原因でグローバルロック手順がスキップされたされるたびに、WARNING メッセージが書き込まれます。一部のステートメントで 2 つの WARNING メッセージが表示されるのは正常です。1 つは DML ロックをスキップするため、もう 1 つは DDL ロックをスキップするためです。

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

グローバルロック

- `bdr.ddl_locking` —上記で説明しました。
- `bdr.global_lock_max_locks` —ノードで保持できるグローバルロックの最大数(デフォルトは 1000)。Postgres サーバーのスタートにのみ設定できます。
- `bdr.global_lock_timeout` —グローバルロックの最大許容待機時間を設定します (デフォルト 10 分)。ゼロの値はこのタイムアウトを無効にします。
- `bdr.global_lock_statement_timeout` —グローバルロックを保持するステートメントの最大許容期間を設定します (デフォルト 60 分)。ゼロの値はこのタイムアウトを無効にします。
- `bdr.global_lock_idle_timeout` —グローバルロックを保持しているトランザクションの最大許容アイドル時間を設定します (デフォルト 10 分)。ゼロの値はこのタイムアウトを無効にします。
- `bdr.predictive_checks` —予測チェックのログレベル (現在、グローバルロックでのみ使用されます)。DEBUG、LOG、WARNING (デフォルト)、または ERROR です。予測チェックは、特定の操作を実行するときに予想されるクラスターの状態の初期の検証です。タイムアウトを待つのではなく、早期に失敗するための操作に使用できます。グローバルロックの用語では、BDR は、十分なノードが接続されており、グローバルロックに必要なクォーラムを取得するための合理的な遅延制限があることを確認します。

ノード管理

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

汎用レプリケーション

- `bdr.writers_per_subscription` — サブスクリプションごとのライターのデフォルト数（BDR では、グループの `bdr.alter_node_group_config` でこれを変更することもできます）。
- `bdr.max_writers_per_subscription` — サブスクリプションあたりのライターの最大数（上記の設定の上限を設定します）。
- `bdr.xact_replication` — 現在のトランザクションを複製します（デフォルトは `on`）。

これを BDR にトランザクションと、トランザクションロジカルデコーディングがローカルのみでロジカルデコーディング可視。データは、ロジカルスタンバイノードを含む他のノードに転送されません。

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

このパラメータは、`SET LOCAL` コマンドを使用して現在のトランザクション内でのみ設定できます。

注釈: トランザクションレプリケーションを無効にしても WAL は生成されますが、これらの変更はオリジンでフィルタリングされます。

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

- `bdr.permit_unsafe_commands` — 一般的な使用には安全でないと考えられるコマンドの安全性チェックをオーバーライドするオプション。

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

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

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

このオプションにより、ラージデータのレプリケーションは、一連の挿入ではなく、内部的に COPY としてロードされます。ノード結合時の初期データも同様にコピーされます。

- `bdr.maximum_clock_skew`

このオプションは、`bdr.maximum_clock_skew_action` をトリガーする前の受信トランザクションコミットタイムスタンプとサブスクライバーの現在の時刻の最大差を指定します。

現在再生されているトランザクションのタイムスタンプが、サブスクライバーの現在時刻と比較して将来かどうかを確認します。そうであり、差が `bdr.maximum_clock_skew` より大きい場合、`bdr.maximum_clock_skew_action` 設定で指定されたアクションを実行します。

デフォルトは-1で、クロックスキューを無視することを意味します（チェックはオフになっています）。全サーバーの時計を同期させる場合は0が有効です。トランザクションを再生しているという事実は、過去にコミットされたことを意味します。

- `bdr.maximum_clock_skew_action`

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

このオプションには2つの値があります。

- **WARN** —この事実に関するワーニングをログに記録します。サーバーログのフラッディングを防ぐために、警告は最大で1分に1回（デフォルト）ログに記録されます。- **WAIT** —現在のローカルタイムスタンプが、リモートコミットタイムスタンプから `bdr.maximum_clock_skew` を引いたものより古くなくなるまで待ちます。
- `bdr.accept_connections` —BDR への接続を有効または無効にするオプション。デフォルトは `on` です。

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

`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 ノードのフェールオーバー候補のストリーミング物理的レプリカが保証の変更を受信して、サブスクライバーに可視されるようにします。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時にコミットが消えないことが保証されます。

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

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

`bdr.standby_slot_names` には、任意のレプリケーションスロットをリストできます。ロジカル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それを受け取っていないために、フェイルオーバー時にプロバイダーから消えるという2つの異常が発生する可能性があります。

- 1人以上のサブスクライバーの場合、サブスクライバーは変更を適用した可能性がありますが、新しいプロバイダーは受け取った変更と競合する新しいトランザクションを実行可能性があります。

- 2人以上のサブスクリバースの場合、フェイルオーバー時に、すべてのサブスクリバースが変更を適用していません。コミットを受け取っていないサブスクリバースにはそれを取得する方法がないため、サブスクリバースの状態は一貫していません。

デザインで `bdr.standby_slot_names` を設定すると、プロバイダーのフェイルオーバー候補レプリカが追いついていない場合、サブスクリバースがプロバイダーよりも遅れます。したがって、モニタリングは不可欠です。

保証が役立つ別のユースケースは、サブスクリバース専用を使用して、サブスクリバース専用ノードが他の BDR ノードよりも前に移動しないようにすることです。

`bdr.standby_slots_min_confirmed`

BDR サブスクリバース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します。ライタプロセスが停止しているか、進行が遅い場合、キューがいっぱいになり、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。デフォルトはメガバイトで、許可される最大サイズはギガバイトです。GUC の設定には任意のストレージサイズ指定子を使用できますが、デフォルトは KB です。

`bdr.writer_output_queue_size`

このオプションは、レシーバがライタプロセスからデータをレシーバするために使用する共有メモリキューのサイズを指定します。ライタは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 32 KB で、最大許容サイズは 1 メガバイトです。GUC の設定には任意のストレージサイズ指定子を使用できますが、デフォルトは KB です。

`bdr.min_worker_backoff_delay`

レート制限 BDR バックグラウンドワーカーの起動は、特定のワーカーが `bdr.min_worker_backoff_delay` ミリ秒ごとよりも頻繁に再起動されないようにします。エラーが繰り返されると、最大 `bdr.max_worker_backoff_delay` までジッターが追加されると、バックオフが指数関数的に増加します。

時間単位のサフィックスがサポートされています。

注釈: 現在、この設定はレシーバ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行する速度に影響します。

`bdr.min_worker_backoff_delay` のデフォルトは 1 秒です。`bdr.max_worker_backoff_delay` の場合は 1 分です。

バックオフ遅延設定が変更され、PostgreSQL 構成がリロードされると、現在のすべてのバックオフがリセットを待機します。さらに、管理者がすべてのバックオフ間隔を強制的に終了できるように、エキスパイア関数が提供されています。

各タイプのワーカーの最終起動時刻を記憶するために、共有メモリ内の追跡テーブルが維持されます。この追跡テーブルは永続的ではありません。これは、クリーンでないバックエンドが終了した後のクラッシュリカバリ中のソフトリスタートを含む、PostgreSQL の再起動によってクリアされます。

ビュー `:ref:`bdr.worker_tasks`<bdr.worker_tasks>` を使用してこの状態を検査できるため、管理者は現在有効なバックオフレート制限を確認できます。

レート制限の目的で、ワーカーはタスク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ー ID、拡張機能の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、および同期ライターのリモートリレーション ID で構成されます。NULL は、特定の分類子が適用されない場合に使用されます。例、マネージャのワーカーにはサブスクリプション ID がなく、レシーバーにはライター ID がありません。

CRDT

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

の場合）は、ベースの CRDT タイプ（例、`crdt_pncounter` の `bigint`）のみを結果ます。`on` に設定すると、戻り値は CRDT 値の完全な表現形式を表します。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`** — 明示的な 2 フェーズコミット、CAMO、または Eager トランザクションによるクラスター全体の同時準備トランザクションの最大数に対処するには、十分に高く設定する必要があります。制限を超えると、ノードでローカル二相コミットまたは CAMO トランザクションを実行できず、クラスター上のすべての Eager トランザクションが防止されます。これは、PostgreSQL サーバーのスタートにのみ設定できます。

Eager Replication

- **`bdr.commit_scope`** — コミットスコープを `global` に設定すると、`eager all node replication`（デフォルトは `local`）が有効になります。
- **`bdr.global_commit_timeout`** — グローバル二相コミットの両方のステージ（デフォルトは 60 秒）と、CAMO パートナーを待機する時間の制限としての、コミットフェーズでの CAMO 保護されたトランザクションのタイムアウト。

Commit At Most Once

- `bdr.enable_camo` —CAMO 機能を有効にして制御するために使用されます。デフォルトは `off` です。これを `remote_write`、`remote_commit_async`、または `remote_commit_flush` に設定することで、CAMO をトランザクションごとにオンにできます。下位互換性のために、値 `on`、`true`、および `1` は最も安全な `remote_commit_flush` モードを設定しますが、`false` または `0` も CAMO を無効にします。
- `bdr.standby_dsn` —ローカルノードのクラッシュ後に変更された場合、接続文字列 (DSN) のマニュアルオーバーライドが CAMO パートナーに到達することを許可します。通常は設定されていません。Postgres サーバーのスタートにのみ設定できます。
- `bdr.camo_local_mode_delay` —トランザクションを確認する必要がある CAMO パートナーで通常発生するオーバーヘッドをエミュレートに CAMO のローカルモードで適用されるコミット遅延。デフォルトは 5 ミリ秒です。この機能を無効にするには `0` に設定します。
- `bdr.camo_enable_client_warnings` —CAMO プロパティを保証できないデータベースでアクティビティが実行された場合に警告を発します。これはデフォルトで有効になっています。十分な情報を得たユーザーは、これを無効にして、ログに記録される警告の量を減らすことができます。
- `synchronous_replication_availability` —CAMO パートナーが切断された後にノードが続行してコミットできるようにすることにより、可用性を高めるためにオプションで `async` にすることができます。デフォルト値の `wait` では、ノードは無期限に待機し、CAMO パートナーが再接続して確認を送信した後にのみコミットに進みます。

トランザクションストリーミング

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

[トランザクションストリーミング](#) を参照してください。

ラグ制御

- `bdr.lag_control_max_commit_delay` —許容できる最大許容ポストコミット遅延 (ミリ秒単位)。
- `bdr.lag_control_max_lag_size` —許容できる最大ラグサイズ (キロバイト)。
- `bdr.lag_control_max_lag_time` —許容できる最大遅延時間 (ミリ秒)。
- `bdr.lag_control_min_conforming_nodes` —許容可能なラグメジャーを下回るために必要なノードの最小数。

- `bdr.lag_control_commit_delay_adjust` —コミット遅延のマイクロ調整は、最大コミット遅延時間の割合として測定されます。デフォルト値の 0.01%では、最大コミット遅延に到達するには正味 100 回のインクリメントが必要です。
- `bdr.lag_control_sample_interval` —遅延サンプルとコミット遅延のマイクロ調整間の最小時間 (ミリ秒)。
- `bdr.lag_control_commit_delay_start` —許容可能なラグメジャーの割合として表される、コミット遅延の増分の適用がスタートされるラグのしきい値。デフォルト値の 1.0 %では、許容可能なラグ測定値を超えるまでコミット遅延のインクリメントは開始されません。

より小さい割合を設定することで、より早く「ラグ曲線を曲げ」て、許容可能なラグメジャーで漸近することにより、違反を防止できる場合があります。

タイムスタンプベースのスナップショット

- `snapshot_timestamp` — **タイムスタンプベースのスナップショット** の使用をオンにし、使用するタイムスタンプを設定します。
- `bdr.timestamp_snapshot_keep` —タイムスタンプベースの `snapshot` で使用する有効なスナップショットを保持する時間 (デフォルトは 0、過去のスナップショットを保持しないことを意味します)。

モニタリングとロギング

- `bdr.debug_level` —BDR がデバッグメッセージを書き込むために使用するログレベルを定義します。デフォルト値は `debug2` です。詳細な BDR デバッグ出力を見たい場合は `bdr.debug_level = 'log'` を設定してください。
- `bdr.trace_level` —上記と同様に、これは BDR トレースメッセージに使用するログレベルを定義します。BDR クラスターのすべてのノードでトレースを有効にすると、EDB サポートが問題を診断するのにヘルプ場合があります。これは、Postgres サーバーのスタートにのみ設定できます。

稼動 ディスクスペースまたはローテーションを値 $\geq$ フィルタリングに設定すると、非常にラージのログ出力が生成される可能性があります。

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

内部

- `bdr.raft_keep_min_entries` — ログコンパクションを行うときに Raft ログに保持するエントリーの最小数 (デフォルト 100)。値 0 はログの圧縮を無効にします。これは、Postgres サーバーのスタートにのみ設定できます。.. Warning :: ログの圧縮が無効になっている場合、ログのサイズは永久に大きくなります。
- `bdr.raft_response_timeout` — ネットワーク障害に対応するために、実装された Raft コンセンサスプロトコルは一定の時間が経過するとリクエストをタイムアウトしアカウント。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` — ステートマシンを前進させるために、Raft はエントリをその内部ログに追加します。通常のオペレーション中、追加には数ミリ秒しかかかりません。これにより、その追加アクションの期間に上限しきい値が設定され、それを超えると INFO メッセージがログに記録されます。これは問題を示している可能性があります。このパラメータのデフォルト値は 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に記録する場合。bdr コンセンサスリクエストのラウンドトリップ時間を測定し、時間がこのパラメータを超える場合は INFO メッセージをログに記録します。このパラメータのデフォルト値は 5000 ミリ秒です。
- `bdr.raft_group_max_connections` — Postgres サーバーのすべての BDR グループにわたる接続の最大数。これらの接続は、グループのノード間で bdr コンセンサス要求を運びます。このパラメータのデフォルト値は 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` — `bdr.bdr_version_num` で使用されるのと同じ数値フォーマット、たとえば、30618 で、下位互換性のあるバージョンを指定します。一般的に望ましくない効果がある場合でも、以前の BDR バージョンの正確な動作を有効にします。デフォルトは現在の BDR バージョンです。これはリリースごとに異なるため、値が現在のリリースと異なる場合を除き、構成ファイルで明示的に使用しないことをお勧めしバージョン。
- `bdr.track_replication_estimates` — 同等な適用率とキャッチアップ間隔に関するレプリケーションの見積もりを追跡します。CAMO のようなプロトコルは、この情報を使用して、同等な準備ができているかを推定できノード。このパラメータはデフォルトで有効になっています。
- `bdr.lag_tracker_apply_rate_weight` — ローカルノードからの WAL の適用に関して同等ながどの程度遅れているかをモニタし、ラグトラッキングの適用率の移動平均を計算します。このパラメータは、新しい計算値がこの移動平均計算にどの程度寄与するかを指定します。デフォルト値は 0.1 です。

2.5 Node Management

BDR グループのメンバである各データベースは、独自のノードで表す必要があります。ノードは、BDR グループ内のこ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、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ができ、さまざまなアーキテクチャを作成できます。

2.5.1 BDR グループの作成と参加

BDR の場合、すべてのノードは他のすべてのノードに接続する必要があります。make 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接続するように自動的に構成されます。このため、作成された最初の BDR ノードを含むすべてのノードは、他のノードが接続に使用できる `PostgreSQL connection string`（「データソース名前」の DSN と呼ばれることがあります）を認識する必要があります。両方の形式の接続文字列がサポートされています。したがって、`host=myhost port=5432 dbname=mydb` のようなキーと値のフォーマット、または URI フォーマット：`postgresql://myhost:5432/mydb` のいずれかを使用できます。

ローカルノードから BDR グループを作成するには、SQL ファンクション `bdr.create_node_group()` を使用します。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、他のノードがこのノードに接続するために使用する接続文字列を指定する必要があります。

ノードグループが作成されると、以降のすべてのノードは `bdr.join_node_group()` ファンクションを使用して BDR グループに結合できます。

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異なり、オプションでターゲットデータベースのみのベースバックアップを指定できます。これにより、このアクティビティがより速く完了 `make` とともに、不要なデータベースが除外されるため、使用され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、除外されたデータベースは新しいノードでクリーンアップされて削除されます。

新しい BDR ノードが既存の BDR グループに参加するか、ノードがアップストリーム同等なサブスクリバされる場合、レプリケーションを開始する前に、システムは既存のデータを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じで始まるように注意深く調整する必要があります。自分で `pg_dump` を使用するだけでは不十分です。BDR 拡張機能は、この初期コピーを作成するための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、指定されたソースノードを基礎として使用して既存のデータを同期し、BDR グループのメッシュトポロジで自分自身を確立するために必要なすべてのメタデータ情報を作成します。この初期コピー中にソースと新しいノード間の接続が切断された場合、結合プロセスを最初からやり直す必要があります。

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

`bdr.join_node_group()` ファンクションの `synchronize_structure` パラメータを使用して、スキーマの同期をオプションでスキップされたできます。その場合、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。

接続が最適な（最も近い）ソースノードを選択することをお勧めしノード。これにより、結合が終了するまでに必要な時間が短縮されます。

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

ロジカル結合プロシージャ (`bdr.join_node_group()` ファンクションを使用) は、COPY 操作を実行してデータ同期を実行し、有効になっている場合はマルチプルのライター (並列適用) を使用します。

ノード結合は、結合にかかる時間の大部分を他のノードジョインと同時に実行できます。ただし、一度に1つの通常ノードのみが **PROMOTE** または結合の状態になります。他のすべてのノードが稼働している場合、通常はかなり短くなります。サブスクライバー専用ノードはこのルールの例外であり、同時に **PROMOTE** 状態と **PROMOTING** 状態になる可能性があるため、結合プロセスは完全に同時です。

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、ノードがダウンしていても、大部分のノードが利用可能な場合に実行できます。これにより、ロジカル結合の実行が複雑になる場合があります。ロジカル結合中に、ソースノードのタイムスタンプのノードコミットタイムスタンプに設定されます。これより前のコミットタイムスタンプを持つノードでコミットされた変更 (ノードがダウンしているか、大幅な遅延があるため) は、他のノードからの変更と競合する可能性があります。この場合、新しく参加したノードは他のノードとは異なる方法で解決され、発散が発生する可能性があります。その結果、ノード間に重大なレプリケーションラグがある場合はノード結合を実行しないことをお勧めします。ただし、これが必要な場合は、新しく参加したノードで **LiveCompare** を実行して、すべてのノードが利用可能になって追いついたらデータの相違を修正します。

キャッシュ検索の失敗が原因で、ソースノードで同時 DDL アクティビティがある場合、`pg_dump` が失敗する場合があります。`bdr.join_node_group()` は内部で `pg_dump` を使用するため、ソースノードに同時の DDL アクティビティがある場合に失敗する場合があります。このような場合、結合を再試行すると機能します。

異種混合クラスターへの参加

BDR 4.0 ノードは、特定の最小メンテナンスリリース (3.7.6 など) または 3.7 と 4.0 ノードの混合で 3.7.x を実行している BDR クラスターに結合できます。このプロシージャは、ユーザが BDR メジャーバージョンだけでなく、基になる PostgreSQL メジャーバージョンもアップグレードする場合に役立ちます。これは、PostgreSQL 12 または 13 で実行されている 3.7 ノードを PostgreSQL 11 で PostgreSQL を実行している BDR クラスターに参加することで実現できメジャーリリース。もちろん、新しいノードは、既存のクラスター。

BDR は、一部のノードが1つの PostgreSQL メジャーリリースで 3.6 を実行していて、他のノードが別の PostgreSQL メジャーリリースで 3.7 を実行している場合でも、レプリケーションがすべての方向で正しく動作することを保証します。ただし、ユーザは、十分な数の新しいノードがクラスターに参加したら、古いノードを分割して、クラスターをすばやく同種の状態にすることをお勧めします。古いバージョンで使えない可能性のある DDL を実行しないように注意する必要があります。

異なる PostgreSQL のメジャー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的バックアップを使用できません。ノードはロジカル結合メソッドを使用して結合必要があります。これが必要なのは、PostgreSQL のメジャーリリースが互いにディスク上で互換性がないためです。

3.7 ノードが 3.6 ノードをソースとして使用してクラスターに参加する場合、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 は単なる `libpq` 接続文字列です。そのため、SSL 用のものを含む、許可された `libpq` 接続パラメータ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するこのようなパラメーターのセットの例を次に示します。

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

この設定では、ファイル `bdr_client.crt`、`bdr_client.key`、および `root.crt` は、適切な権限を持つ各ノードのデータディレクトリに存在する必要があります。`verify-full` モードの場合、サーバーの SSL 証明書が保証され、`root.crt` 認証局で直接または間接的に署名されていること、および接続で使用されるホスト名前またはアドレスが証明書の内容と一致することが確認されます。名前の場合、これはサブジェクトの別名とマッチします。証明書にそのような名前がない場合は、サブジェクトの共通名 (CN) フィールドと一致します。Postgres は現在 IP アドレスのサブジェクト代替名をサポートしていないため、接続が名前ではなく `address` で行われる場合、CN フィールドとマッチ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マッチング行が必要です。例：

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

別の設定では、クライアント証明書の代わりに `SCRAM-SHA-256` パスワードを使用し、証明書が適切に署名されている限りサーバー ID の検証を気にしません。ここで、DSN パラメーターは次のとおりです。

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

…そして、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` ファイルが必要です。

2.5.2 ウィットネスノード

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

追加のフルサイズのノードを作成するのではなく、マイクロノード（Witness ノードとも呼ばれます）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。

2.5.3 ロジカルスタンバイノード

BDR を使用すると、「ロジカルスタンバイノード」、「オフロードノード」、「読み取り専用ノード」、「受信専用ノード」または「ロジカルリードレプリカ」を作成できます。マスタノードは、ゼロ、1 つ以上のロジカルスタンバイノードを持つことができます。

物理的スタンバイノードでは、ノードが完全に起動せず、継続的リカバリモードのままになります。BDR 同様のことができます。bdr.join_node_group には、ノードをロジカルスタンバイノードとして途中で結合したままに make の pause_in_standby オプションがあります。ロジカルスタンバイノードは変更を受信しますが、ローカルに加えられた変更を他のノードに送信しません。

後で、必要に応じて bdr.promote_node() を使用して、ロジカルスタンバイを完全な通常の送信/受信ノードに移動します。

ロジカルスタンバイは、bdr.join_node_group の DSN で定義された 1 つのソースノードによってデータが送信されます。他のすべてのノードからの変更はこの 1 つのソースノードから受信され、マルチプルのサイト間の帯域幅が最小限に抑えられます。

高可用性には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、新しいマスタは引き続き任意の/すべてのロジカルスタンバイノードに電力を供給できます。
- ソースノードが停止した場合、1 つのロジカルスタンバイをノードに昇格させ、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ソースを置き換えることができます。マルチプルのロジカルスタンバイノードがある場合、他のノードは新しいマスタをフォローできないため、この手法の有効性は効果的に 1 つのロジカルスタンバイに制限されることに注意してください。

既存の BDR ノードから新しいスタンバイが作成された場合、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は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、ストリーミングレプリカでスロットが同期/作成される前に、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その BDR ノードを置き換えるように昇格させることはできません。

スタンバイのスロット同期アッププロセスは、アップストリームのファンクションを呼び出すことによりこれを解決します。このファンクションは、WAL スイッチを実行し、すべての BDR 同等な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 クラスター全体のグループスロットを移動します。上記により、グループスロットが短時間で前進します。これにより、スタンバイが最初のスロットの同期に必要な時間が短縮され、必要に応じてより迅速にフェイルオーバーできます。

PostgreSQL では、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ことが重要です。次のクエリー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。このクエリーが true を返すと、プロモーションを続行できます。

```
SELECT true FROM pg_catalog.pg_replication_slots WHERE
    slot_type = logical AND confirmed_flush_lsn IS NOT NULL;
```

WAL スイッチを手動で実行し、すべての BDR 同等な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 クラスター全体のスロット同期プロセスを微調整することもできます。このアクティビ

ティは、グループスロットを短時間で前進させ、スタンバイでのスロット同期アクティビティを早めます。次のクエリは、このためにターゲットデータベース内の BDR 同等なで実行できノード。

```
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スロットの同期にヘルプことを確認します。

ロジカルスタンバイノードは、必要に応じて物理的スタンバイノードを使用して保護できるため、Master->LogicalStandby->PhysicalStandby。LogicalStandby から LogicalStandby には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が追加のインデックス、データ、中間作業表、LISTEN/NOTIFY、一時テーブル、マテリアライズドビューおよびその他の違いの保持期間を長くできるため、非常に役立ちます。

昇格前にコミットされたロジカルスタンバイにローカルで加えられた変更は、他のノードに送信されません。昇格後にコミットされるすべてのトランザクションは以降に送信されます。ロジカルスタンバイへの書き込みを実行する場合は、昇格前にデータベースを静止することをお勧めします。

ロジカルスタンバイノードに DDL 変更を make ことができますが、それらはレプリケートされず、グローバル DDL ロックを取得しようとしません。DDL と同様に機能する BDR ファンクションもレプリケートされません。

DML および DDL レプリケーション を参照してください。ロジカル・スタンバイに互換性のない DDL 変更を加えた場合、データベースはダイバージェント・ノードと呼ばれます。ダイバージェントノードを昇格すると、現在レプリケーションが失敗します。その結果、ロジカルスタンバイノードをスタンバイとして使用する場合は、そのノードに異なる変更が加えられフリーようにするか、または異なるノードが昇格しないことを保証保証必要があります。

2.5.4 フィジカルスタンバイノード

BDR では、従来の物理的スタンバイフェイルオーバーノードを作成することもできます。これらは、一般的に、短い昇格プロシージャの後にクラスター内の BDR ノードを直接置き換えることを目的としています。標準の Postgres クラスターと同様に、ノードはこれらの物理的レプリカを持つことができます。

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

- BDR プライマリとスタンバイ間の接続は、物理的レプリケーションスロットを介したストリーミングレプリケーションを使用します。
- スタンバイには以下があります。- レプリケーションスロット (PostgreSQL <12 の場合、これらの設定は物理的にある必要があり PostgreSQL) : `postgresql.conf` : `- shared_preload_libraries = 'bdr'`、リストには他のプラグインも存在する可能性があります、`pglogical` を含めないでください- `hot_standby = on` - `hot_standby_feedback = on`

- プライマリには以下があります。 `-postgresql.conf` : `-bdr.standby_slot_names` は、スタンバイの `primary_slot_name` に使用される物理的レプリケーションスロットを指定する必要があります。

BDR ノードの動作中の物理的スタンバイを生成するにはこれで十分ですが、対処が必要な追加の懸念事項がいくつかあります。

確立されると、スタンバイには、BDR グループスロットを含むプライマリの他の BDR 関連のレプリケーションスロットの初期コピーをトリガーするのに十分な時間と WAL トラフィックが必要です。少なくとも、スタンバイのスロットは「ライブ」のみであり、`pg_replication_slots` によって報告されるようにゼロ以外の `confirmed_flush_lsn` をレポートする場合、フェイルオーバー後も存続します。

その結果、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 クラスター内の物理的スタンバイノードは、フェイルオーバーが正常に動作すると想定する前にチェックする必要があります。この予防措置を講じないと、スタンバイが BDR ノードとして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になり、フェイルオーバーが中止される可能性があります。

物理的スタンバイノードが最新の状態であり、昇格できるようにする保護メカニズム（構成された `bdr.standby_slot_names`）は、グループレプリケーションが物理的スタンバイノードが最新の場合にのみ発生するため、BDR グループの全体的なレプリケーションのレイテンシに影響します。

これらの理由から、一般的には、物理的スタンバイノードではなく、ロジカルスタンバイノードまたはサブスクリプト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。

すべてのノードでグループスロットが（可能な限り）拡張されていることを手動で保証できます。これは、次の SQL 構文を使用して、物理的スタンバイでの BDR 関連のレプリケーションスロットの作成を促進します。

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

フェイルオーバー時に、スタンバイはプライマリを置き換える 2 つのアクションのいずれかを実行する必要があります。

1. プライマリと同じ IP アドレスまたはホスト名の制御を引き受けます。
2. 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にのみ同期されるため、この新しいプライマリは既存の BDR ノードで予想されるよりも低い LSN を反映する場合があります。この場合、BDR は各遅延スロットを各 BDR ノードが使用した最後の場所に早送りします。

`bdr.standby_slot_names` パラメータにもノートしてください。これは、プライマリ->フィジカルスタンバイのリレーション BDR クラスターで、またはサブスクリプター専用グループを使用する場合に重要です。

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

スタンバイは他の BDR ノードと直接通信しないため、`standby_slot_names` パラメータは、名前付けスロットをグループスロットでも必要なコンストレインとして考慮するように BDR に通知します。設定すると、通常はグループスロットが進められていた場合でも、スタンバイに遅延が表示された場合にグループスロットが保持されます。

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

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

`synchronous_standby_names` に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 と Eager All Node Replication は異なる同期メカニズムを使用し、同期レプリケーションでは機能しません。`synchronous_standby_names` に CAMO パートナーが含まれているか（CAMO が使用されている場合）、BDR ノードがまったく含まれていない（Eager All Node Replication が使用されている場合）`make` を確認してください。フィジカルスタンバイなどの非 BDR ノードのみが含まれます。

2.5.5 サブグループ

Group には、0 個以上のサブグループを含めることもできます。各サブグループは、トップレベルの親グループ内の特定の目的に割り当てることができます。`node_group_type` は、サブグループを作成するときにタイプを指定します。

加入者専用グループ

名前が示すように、このタイプのノードはクラスター内の他のノードからのレプリケーションの変更のみをサブスクライブしますが、他のノードは `subscriber-only` ノードからレプリケーションの変更を受け取りません。これはロジカルスタンバイノードに多少似ていますが、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信できるため、クラスター内の 1 つのノードが使用不可になったり、分離したりしても影響を受けません。

`subscriber-only` ノードは完全に参加した BDR ノードであるため、レプリケートされたすべての DDL を受信して動作します。また、Raft を使用して、クラスター内のすべてのノードにステータスを一貫してレポートします。`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出に参加することもできません。また、レプリケートされた DDL を受信している間は、DDL または DML ロックの取得には参加しません。言い換えれば、現在ダウンしている `subscriber-only` ノードは、取得中の DML ロックを停止しません。

`subscriber-only` ノードは、BDR ツリートポロジ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すべての方向に変更を複製し、変更を受信するのみでクラスター内の他のノードに変更を送信しないラージの `subscriber-only` ノードがあります。このトポロジーは、多数のノードが原因で発生する接続の爆発を回避しますが、データの使用に使用できるラージにラージの leaf ノードを提供します。

オーダーノードをユーザするには、最初にタイプ「サブスクライバー専用」の BDR グループを作成する必要があります。これは、メンバノードがレプリケーションの変更を受け取るグループのサブグループである必要があります。サブグループが作成されたら、`subscriber-only` ノードになる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。「サブスクライバー専用」タイプのサブグループを複数作成でき、異なる親グループを持つことができます。

ノードが「サブスクライバー専用」サブグループに正常に参加すると、それは **subscriber-only** ノードになり、親グループのレプリケーション変更の受信をスタートします。**subscriber-only** ノードで直接行われた変更は複製されません。

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

注意事項

subscriber-only ノードはクラスター内のノードに変更をレプリケートしないため、ノードがクラスターから分離されたときにレプリケーションの変更を同期するソースとして機能できません。ただし、**subscriber-only** ノードが、クラスター内の他のノードが現在持っていないレプリケーションの変更を **parted** ノードから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。

今のところ、これは `bdr.standby_slot_names` と `bdr.standby_slots_min_confirmed` を適切に設定して、**subscriber-only** ノードよりも前に完全にアクティブな BDR ノードが常に存在するようにすることで解決できます。

これは、将来のリリースで改善される可能性があります。**subscriber-only** ノードがレプリケーションの先頭になることを許可してからそれらを同期のレプリケーションソースとして使用するか、別の完全に結合されたノードが分離されたときに一貫性のない **subscriber-only** ノードをオプションで削除する方法を提供できます。

2.5.6 デコードワーカー

BDR4 は、送信されるノードの数に関係なく、1 回のデコードを実行するワーカープロセスのデコードを有効にするオプションを提供します。これにより、各 BDR ノードに新しいプロセスである **wal** デコーダーが導入されます。接続ごとに 1 つの **WAL Sender** プロセスが引き続き存在しますが、これらのプロセスはデータの送受信のみを実行しタスク。これらの変更をまとめると、**WAL** 送信者プロセスが通信により多くの時間を費やすようになったため、より大きな BDR グループの CPU オーバーヘッドが削減され、レプリケーションスループットが向上します。

`enable_wal_decoder` は BDR グループごとのオプションであり、現在デフォルトで無効になっています。`bdr.alter_node_group_config()` を使用して、BDR グループの **Decoding Worker** を有効または無効にできます。

Decoding Worker が有効になっている場合、BDR は **Logical Change Record**（略して **LCR**）ファイルを保存して、デコードとすべてのサブスクライブノードがデータを受信した時の変更をバッファリングできます。**LCR** ファイルは、各ローカルノードのデータディレクトリ内の `pg_logical` ディレクトリに保存されます。レプリケーションの遅延が増加すると、**LCR** ファイルの数とサイズが変化するため、これもモニタリングが必要です。BDR ノードで必要ない **LCR** は定期的にクリーニングされます。2 つの連続するクリーンアップのインターバルは、デフォルトの 3 分である `bdr.lcr_cleanup_interval` によって制御されます。`bdr.lcr_cleanup_interval` がゼロの場合、クリーンアップは無効です。

無効にすると、各ノードにサブスクライブする各ノードの **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。

BDR グループに対してデコードワーカーが有効になっている場合でも、次の GUC はノードごとの LCR の稼動と使用を制御します。デフォルトでは、これらは `false` です。LCR を稼動して使用するには、BDRBDR 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、LCR を使用するすべての WAL センダーが再起動され、WAL を直接使用します。BDR グループ構成とともに `true` の場合、デコードワーカープロセスが開始されて LCR が生成され、WAL 送信者は LCR を使用します。
- `bdr.receive_lcr` - サブスクリバードで `true` を使用する場合、パブリッシャーノードで LCR を使用するよう要求します。

ノート

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

サブスクリバード専用ノードは各ノードから直接変更を受信するため、サブスクリバード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 Decoding Worker 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 に対応する WAL が必要ない場合は、削除できる場合があります。

リファレンスまでに、LCR ファイル名の最初の 24 文字は WAL ファイル名と似ています。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` を表すことが予想されます。次の 16 文字の名前は、WAL ストリームに対して LCR の変更を追跡するために使用される WAL セグメント番号に似ていシーケンス。ただし、ロジカル変更は、以下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ることにノートしてください。したがって、LCR セグメント内のそれらの配置は、WAL セグメント内の対応する WAL の配置とマッチしません。最後の 16 文字は、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。

2.5.7 ノードの再起動とダウンしたノードの回復

BDR は、ノードの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、そのノードから欠落しているデータを複製することにより、自動的にグループに再参加しノード。

ノードが起動すると、各接続は `bdr.node_slots.state = catchup` を表示し始め、欠損データの複製を開始します。キャッチアップは、各同等なからの欠落ノードの量に応じて一定期間継続され、サーバーのワークロードに応じて、時間の経過とともに増加する可能性があります。

各ノードの書き込みアクティビティの量が均一でない場合、より多くのデータを持つノードからのキャッチアップ期間は他のノードよりも大幅に長くなる可能性があります。最終的に、スロットの状態は `bdr.node_slots.state = streaming` に変更されます。

ノードが長時間（数時間や数日など）オフラインになると、さまざまな理由でリソースの問題が発生し始める可能性があります。ユーザーは、次の問題を理解することなく、拡張の停止を計画しないでください。

各ノードは変更情報を保持する（同等なごとに1つの *BDR* によって作成されたレプリケーションスロットを使用）ため、一時的に到達できないノードへの変更を後でリプレイできノード。同等なが無期限にオフラインのままであるノードでノード PostgreSQL トランザクション（スペースの WAL）が不足し、データベースサーバが次のようなエラーでシャットダウンする可能性がありますこれ：

```
PANIC: could not write to file "pg_wal/xlogtemp.559": No space left on device
```

…または、その他のディスク不足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 を保持し、カタログテーブルのパキューム処理を防ぎます。

EDB Postgres Extended Server および EDB Postgres Advanced Server では、オフラインノードもデータのフリーズを保持して、競合解決データの損失を防ぎます（

[オリジンの競合検出](#) を参照）。

管理者はノードの停止をモニタし（[モニタリングとロギング](#)を参照）、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、経時的に使用されるスペースを計算できる場合があります、重大な問題が発生する前にノードがダウンする可能性がある時間を予測できます。

BDR によって作成されたレプリケーションスロットは手動で削除しないでください。その場合、クラスターが破損しているため、以下で説明するように、スロットを使用していたノードをクラスターから分離する必要があります。

ノードがオフラインの間、他のノードはオフラインノードから同じデータセットをまだ受信していない可能性があるため、これはノード間でわずかな相違として表示される場合があります。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。後のバージョンでは、これが以前に行われる場合があります。

BDR によって作成されたレプリケーションスロット

BDR マスタノードでは、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 `bdr_<database name>_<group name>_<node name>`。N は、ダイレクショナルスタンバイを含むクラスター内の BDR ノードの合計数です。

ユーザはこれらのスロットをドロップしてはなりません^{**}。これらは BDRBDR 管理され、必要に応じてドロップします。

一方、Barman や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、ソフトウェアに適したコマンドを使用して、BDR に影響を与えずに作成または削除できます。他のソフトウェアで使用されているスロット名がプレフィックス `bdr_` で始まっていないことを確認してください。

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

- ノード `alpha` には3つのスロットがあります。- `bdr_mydb_mygroup` 名前付けの1つのグループスロット- `bdr_mydb_mygroup_beta` および `bdr_mydb_mygroup_gamma` 名前付けの2つのノードスロット

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

グループレプリケーションスロット

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

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

グループスロットでは次のことができます。

- 結合中にダウンしたノードが再び複製を開始した場合にデータロスすることなく、既存のノードをすべて稼働させずに（ノードの大部分は稼働している必要があります）、新しいノードを BDR グループに結合させます
- パートのノードが分割されたノードに完全に追いついていない場合でも、一貫してクラスターからノードを分割します
- いくつかの競合を見逃さないようにフリーズポイントを保持します
- タイムスタンプベースのスナップショットの履歴 `snapshot` を保持します

グループスロットは通常非アクティブであり、他のノードからの Raft 進行状況メッセージに応答して定期的のみ早送りされます。

警告：グループスロットを落とさないでください。通常は非アクティブですが、BDR クラスターが適切にオペレーションするためには不可欠です。削除された場合、上記の機能の一部またはすべてが動作しなくなったり、誤った結果が発生する可能性があります。

長い識別子のハッシュ

他の PostgreSQL 識別子と同様に、レプリケーションスロットの名前は 63 バイトを超え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、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によりこれを処理します。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、`group20xxxxxxxxxxxxxx`（長さ 20 バイト）名前付け BDR グループを使用して `db20xxxxxxxxxxxxxx`（長さ 20 バイト）名前付け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 `a30xxxxxxxxxxxxxxxxxxxxxxxxxx`（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_a30xxxxxxxxxxxxxx7f304a2
```


…since 3597186、be9cbd0、7f304a2 は、それぞれ db20xxxxxxxxxxxxxxxx、group20xxxxxxxxxxxxxxxx、a30xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx のハッシュです。

2.5.8 BDR グループからのノードの削除

BDR は拡張ノードの停止から回復するように設計されているため、ノードを完全に削除するかどうかをシステムに明示的に伝える必要があります。ノードを完全にシャットダウンし、他のノードに通知しないと、パフォーマンスが低下し、最終的にシステム全体が動作を停止します。

parting と呼ばれるノードの削除は、`bdr.part_node()` ファンクションを使用して実行されます。ノードを削除するには、(ノードの作成中に渡された) ノード名前を指定する必要があります。`bdr.part_node()` ファンクションは、削除するノードを含む BDR グループ内の任意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、別れは Raft コンセンサスを使用して行われ、動作するには大部分のノードがオンラインである必要があります。

分離プロセスはすべてのノードに影響します。ラフトリーダーは、ノード間の投票を管理して、どのノードが分離ノードからの最新のデータを持っているかを確認します。次に、残りのすべてのノードは最新のノードへの二次的な一時的な接続を *make* して、不足しているデータをキャッチできるようにします。

別れたノードはまだ BDR に認識されていますが、リソースを消費しません。ノードは、別れた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、分割されたノードのすべてのメタデータをクリアすることをお勧めします。

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、BDR 拡張機能を削除できません。例は次のとおりです。

- SnowflakeId や gallocc 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、インスタンス、依存オブジェクトを削除するか、列型を BDR 以外の同等のものに変更するか、シーケンス型を `local` に変更する必要があります。

警告: BDR 拡張機能の削除は、ノードが BDR ノードグループから正常に分離された場合、またはそれがグループの最後のノードである場合に実行する必要があります BDR メタデータを削除すると、他のノードとの間のレプリケーションが中断されます。

警告: ローカル BDR ノード、またはローカルデータベース内の BDR 拡張機能を削除すると、既存のセッションが BDR 固有のワークフローを実行しようとして失敗する場合があります。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、別れに問題があります。

2.5.9 BDR トポロジのリスト

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であるすべての BDR ノードグループをリストします（現在 1 行のみを結果ます）。

```
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（例： `mygroup` ）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
, 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ように、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。

ノードの状態のリスト

- **NONE** : ワーカーの起動時にノードの状態は設定解除され、すぐに現在の既知の状態に設定されると予想されます。
- **CREATED** : `bdr.create_node()` は実行されましたが、ノードはまだ BDR クラスタのメンバではありません。
- **JOIN_START** : `bdr.join_node_group()` はローカルノードを既存の BDR クラスタに結合させ始めます。
- **JOINING** : ノード結合が開始され、現在初期同期フェーズであり、ノードにスキーマとデータを作成しています。
- **CATCHUP** : 初期同期フェーズが完了しました。これで、結合は、結合が開始されてから上流の同等なでノードされたトランザクションを取得して適用する最後のステップになりました。
- **STANDBY** : ノードの結合は完了しましたが、変更のブロードキャストはまだ開始されていません。すべての参加はこの状態でしばらくかかりますが、ロジカルスタンバイとして定義されている場合、ノードはこの状態で続行されます。
- **PROMOTE** : ノードはロジカルスタンバイであり、`bdr.promote_node` を呼び出してノードの状態を **ACTIVE** に移動しました。これら 2 つの **PROMOTE** 状態は、1 つのノードのみが **STANDBY** よりも高く、**ACTIVE** よりも低い状態にあるという事実と一貫している必要があります。
- **PROMOTING** : ロジカルスタンバイからフル BDR ノードへの昇格が進行中です。
- **ACTIVE** : ノードは完全な BDR ノードであり、現在は **ACTIVE** です。これは、最も一般的なノードのステータスです。
- **PART_START** : ノードは **ACTIVE** または **STANDBY** で、`bdr.part_node` を呼び出して BDR クラスターからノードを削除しました。
- **PARTING** : ノードは他のノードから切断し、コンセンサスまたはレプリケーションでそれ以上のパートを果たしません。
- **PART_CATCHUP** : 非分割ノードは、最近分割されたノードから欠落しているデータを同期します。
- **PARTED** : ノードの分割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、状態 **PROMOTE** または **PROMOTING** のいずれかになります。

2.5.10 ノード管理インターフェース

ノードは、SQL インターフェイ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。

概要

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

パラメーター

- **node_name** -新しいノードの名前。データベースごとに許可されるノードは1つだけです。有効なノード名は、小文字、数字、ハイフン、およびアンダースコアで構成されます。
- **local_dsn** -ノードへの接続文字列

注意事項

このファンクションは、関連付けられたパブリック接続文字列を使用してローカルノードのレコードを作成するだけです。ローカルレコードは1つしか存在できないため、作成すると、ファンクションは再度実行されます。

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、それによって加えられた変更は現在のトランザクションで可視されます。

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 ノードをロックします。

bdr.drop_node

ノードを削除します。このファンクションは**通常の使用を目的と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、指定されたノードのメタデータをローカルデータベースから削除します。ノードは次のいずれかです。

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

概要

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

パラメーター

- **node_name** - 既存のノードの名前。
- **cascade** - 非推奨、将来的に削除されます
- **force** - すべての健全性チェックを回避し、矛盾を引き起こす危険性があるにもかかわらず、指定された BDR ノードのすべてのメタデータを強制的に削除します。強制的なノードドロップは、別に関連する緊急の場合にのみテクニカルサポートが使用します。

注意事項

これを実行する前に、`bdr.part_node()` を使用してノードを分割しておく必要があります。

この関数は、指定されたノードのメタデータをローカルデータベースから削除します。ノードはローカルノードのいずれかです。その場合、リモートノードに関する情報を含むすべてのノードメタデータが削除されます。または、リモートノードにすることができます。その場合、その特定のノードのメタデータのみが削除されます。

注釈: BDR4 は、一度に最大 1024 個のノードレコード (ACTIVE と PARTED の両方) を持つことができます。これは、snowflakeid および timehard シーケンスで使用するために、各ノードに一意的な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、この関数を使用してこれらのレコードを削除できます。

bdr.create_node_group

この関数は、ローカルノードをグループの唯一のメンバとする BDR グループを作成します。

概要

```
bdr.create_node_group(node_group_name text,  
                      parent_group_name text DEFAULT NULL,  
                      join_node_group boolean DEFAULT true,  
                      node_group_type text DEFAULT NULL)
```

パラメーター

- **node_group_name** - 新しい BDR グループの名前。ノード名前と同様に、有効なグループ名は小文字、数字、アンダースコアのみで構成する必要があります。
- **parent_group_name** - サブグループの親グループの名前。
- **join_node_group** - これは、ノードが作成中のグループに結合するかどうかを決定するのに役立ちます。デフォルト値は `true` です。これは、ノードが結合したくないシャードグループを作成しているときに使用されます。これは、**parent_group_name** が指定されている場合にのみ `false` になります。
- **node_group_type** - 有効な値は、`NULL`、「サブスクライバー専用」、「データノード」、「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」です。「サブスクライバー専用」タイプは、クラスター内の完全に参加したノードからの変更のみを受信し、レプリケーションの変更を送信しないノードのグループを作成するために使用されます。詳細については、[Subscriber-Only Nodes](#) 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、他の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。「subscriber-only」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。

注意事項

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

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。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。`bdr.wait_for_join_completion` を呼び出して、それらが完了するまで待機できます。

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（すべてのデフォルト）を持つオプションは変更されません。

概要

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

(次のページに続く)

(前のページからの続き)

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

パラメーター

- **node_group_name** - 既存の BDR グループの名前。ローカルノードはグループのパートである必要があります。
- **insert_to_update** - 下位互換性の理由で予約されています。
- **update_to_insert** - 下位互換性の理由で予約されています。BDR のバージョン。代わりに `bdr.alter_node_set_conflict_resolver` を使用してください。*
- **ignore_redundant_updates** - 下位互換性の理由で予約されています。
- **check_full_tuple** - 下位互換性の理由で予約されています。
- **apply_delay** - 下位互換性の理由で予約されています。
- **check_constraints** - レプリケートされたデータを書き込むときに適用プロセスがコンストレインをチェックするかどうか。このオプションは非推奨であり、BDR の将来のバージョンでは無効になるか削除される可能性があります。
- **num_writers** - このノードグループをバッキングするサブスクリプションの並列ライターの数。-1 はデフォルト (GUC `bdr.writers_per_subscription` で指定) が使用されることを意味します。有効な値は、-1 または正の整数です。
- **enable_wal_decoder** - Decoding Worker プロセスを有効/無効にします。`streaming_mode` が既に有効になっている場合、Decoding Worker プロセスを有効にできないことに注意してください。
- **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進行中にデコードされ、変更がダウンストリームに送信されます。値が `file` に設定されている場合、ストリーミングトランザクションの着信変更はファイルに保存され、トランザクションがアップストリームでコミットされた後にのみ適用されます。値が `writer` に設定されている場合、着信変更はライターの 1 つに直接送信されます (使用可能な場合)。並列適用が無効になっているか、ストリーミングトランザクションをフリーにハンドルできるライターがない場合、変更はファイルに書き込まれ、トランザクションがコミットされた後に適用されます。値が `auto` に設定されている場合、BDR はトランザクションプロパティと使用可能なリソースに応じて `file` と `writer` をインテリジェントに選択しようとします。WAL デコーダーが既に有効になっている場合、`streaming_mode` を有効にできないことに注意してください。

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

- **default_commit_scope** - デフォルトで使用するコミットスコープ、最初は「ローカル」コミットスコープ。これは、トップレベルのノードグループにのみ適用されます。個々のルールは、同じコミットスコープの異なるオリジングループに使用できます。詳細については、グループコミットの [オリジングループ](#) に関するセクションを参照してください。

注意事項

このファンクションは、デフォルトを変更する要求をグループコンセンサスメカニズムに渡します。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。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にすぐに可視されない場合があります。

このファンクションはロックを保持しません。

警告: このファンクションを使用して `apply\_delay` 値を変更する場合、変更は既にグループのメンバーであるノードには適用されません。この値は通常テスト以外では使用されないため、この制約は稼働で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、ローカルノードを既存の BDR グループに参加させます。

概要

```
bdr.join_node_group (
    join_target_dsn text,
    node_group_name text DEFAULT NULL,
    pause_in_standby boolean DEFAULT false,
    wait_for_completion boolean DEFAULT true,
    synchronize_structure text DEFAULT all
)
```

パラメーター

- **join_target_dsn** - ローカルノードを追加する BDR グループ内の既存の（ソース）ノードへの接続文字列を指定します。
- **node_group_name** - BDR グループのオプションの名前。デフォルトは NULL で、ソースノードに存在する情報からグループ名前を自動検出しようとします。
- **pause_in_standby** - オプションで、ロジカルスタンバイノードとしてのみ結合ように結合プロセスに指示します。これは後でフルメンバに昇格できます。
- **wait_for_completion** - 戻る前に結合プロセスが完了するのを待ちます。デフォルトは true です。
- **synchronize_structure** - 結合中にどのような構造（スキーマ）の同期を行うかを設定します。有効なオプションは、完全なデータベース構造を同期する「all」と、構造を同期しない「none」です（ただし、データは同期します）。

`wait_for_completion` が `false` に指定されている場合。これは、参加プロシージャが開始されるとすぐに戻る非同期呼び出しです。結合の進行状況は、ログと `bdr.state_journal_details` 情報ビューで確認できます。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します。

注意事項

このファンクションは、`join_target_dsn` 接続文字列が指すノードを介してグループコンセンサスメカニズムに要求を渡します。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実行されるため、ロールバックできません。`wait_for_completion` が `true` に設定されている場合、または後で `bdr.wait_for_join_completion` を呼び出すことによって、変更はローカルトランザクションにのみ可視されます。

ノードは単一のグループにのみ属することができるため、このファンクションは各ノードで1回のみ呼び出すことができます。

ノード結合は BDR グループで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 グループのフルメンバに昇格させます。

概要

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

注意事項

このファンクションは、デフォルトを変更する要求をグループコンセンサスメカニズムに渡します。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。昇格プロセスはバックグラウンドで行われるため、ロールバックできません。`wait_for_completion` が `true` に設定されている場合、または後で `bdr.wait_for_join_completion` を呼び出すことによって、変更はローカルトランザクションにのみ可視されます。

プロモーションプロセスは、他のプロモーションに対してロックを保持します。このロックは他の `bdr.promote_node` 呼び出しをブロックしませんが、プロモーションのバックグラウンドプロセスが一度に複数のノードで進むのを防ぎます。

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のを待ちます。

概要

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

パラメーター

- `verbose_progress` - オプションで、結合プロシージャ中に実行された個々のステップに関する情報を出力します。

注意事項

このファンクションは、ローカルノードのチェック状態が `bdr.create_node_group`、`bdr.join_node_group` または `bdr.promote_node` で設定されたターゲット状態に到達するまで待機します。

bdr.part_node

BDR グループからノードノードデータは削除しません。

このファンクションは、削除するノードを含む BDR グループ内のアクティブなノードから呼び出すことができます。ただし、明確にし `make` おくと、ノードが `PARTED` になると、クラスター内の他のノードをパートできません。

注釈: ローカルノードを*分割*する場合は、`wait_for_completion` を `false` に設定する必要があります。そうしないとエラーになります。

警告: このアクションは永続的です。ノードへのレプリケーションを一時的に停止したい場合は、`bdr.alter_subscription_disable()` を参照してください。

概要

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

パラメーター

- **node_name** - パートする既存のノードの名前。
- **wait_for_completion** - `true` の場合、ノードがクラスターから完全に分離されるまで、ファンクションは結果ません。それ以外の場合、ファンクションは分離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** - ローカルノード上のノードを強制的に削除します。コンセンサスに到達できなかった場合、またはノード分割プロセスがスタックした場合、これによりノードの状態がローカルに設定されます。

警告: `force = true` を使用すると、BDR グループが一貫性のない状態になる可能性があるため、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ためにのみ使用してください。

注意事項

このファンクションは、指定されたノードをパートするためのリクエストをグループコンセンサスメカニズムに渡しノード。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。別れのプロセスはバックグラウンドで行われるため、ロールバックできません。`wait_for_completion` が `true` に設定されている場合、分割プロセスによる変更はローカルトランザクションでのみ可視されます。

`force` を `true` に設定すると、コンセンサスが失敗すると、この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ます。このような場合、ファンクションはトランザクションであり（ファンクション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。`force` がマークに設定された状態で既に分割中のノードでファンクションが呼び出された場合、指定されたノードをローカルに分割されて終了します。これは、クラスターでコンセンサスに到達できない場合（つまり、ノードの大部分がダウンしている場合）、または何らかの理由で分割プロセスがスタックする場合にのみ役立ちます。ただし、他のノードは指定されたノードから欠落しているデータを再同期する必要があるため、書き込みを受信していた分割ノードがスタックすることなく分割プロセスに時間がかかる場合があることをアカウントすることが重要です。強制的に分離すると、この再同期が完全にスキップされるため、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。

概要

```
bdr.alter_node_interface(node_name text, interface_dsn text)
```

パラメーター

- `node_name` - 変更する既存のノードの名前
- `interface_dsn` - ノードの新しい接続文字列

注意事項

このファンクションはローカルノードでノード行われます。これは、通常、変更されるノードを含む BDR グループ内のすべてのノードで実行されることを意味します。

このファンクションはトランザクションです-ロールバックでき、変更は現在のトランザクションで可視されます。

このファンクションは、ローカルノードでロックを保持します。

bdr.alter_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。レジュームサブスクリプションとも呼ばれます。サブスクリプションが既に有効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

概要

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

パラメーター

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

注意事項

この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によって開始されます。

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。オプションで、無効になっているサブスクリプションに関連付けられているすべてのワーカーをすぐに停止することもできます。サブスクリプション購入の一時停止とも呼ばれます。サブスクリプションが既に無効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

概要

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

パラメーター

- **subscription_name** - 無効にするサブスクリプションの名前。NULL（デフォルト）の場合、ローカルノード上のすべてのサブスクリプションが無効になります。
- **immediate** - Immediate を使用してアクションをすぐに強制し、無効になっているサブスクリプションに関連付けられているすべてのワーカーを停止します。このオプションが true の場合、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。
- **fast** - この引数は、**immediate** の動作にのみ影響します。true（デフォルト）に設定されている場合、現在の作業が完了するのを待たずに、無効になっているサブスクリプションに関連付けられたすべてのワーカーを停止します。

注意事項

このファンクションはレプリケートされず、ローカルノードのサブスクリプション（特定のサブスクリプションまたはすべてのサブスクリプション）にのみ影響します。

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、サブスクリプションワーカーが停止するタイミングは `immediate` の値に依存します。`true` に設定されている場合、ワーカーは `COMMIT` を待機する前に停止を受け取り、`fast` 引数が `true` に設定されている場合、ワーカーの割り込みは現在の作業が完了するまで待機しません。

2.5.11 ノード管理コマンド

BDR は、既存のノードの物理的コピー（`pg_basebackup`）を介して BDR グループにノードを追加し、既存のノードの物理的スタンバイを BDR グループ内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ノード。

`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、高速なブロックレベルの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。

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物理的スタンバイノードとして既にアクティブになっている場合は、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ってから、マスタノードにプロモートしてから BDR グループに参加します。`--standby` オプションを使用すると、既存の物理的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。指定されたデータディレクトリの開始状態ではなく、新しい BDR ノードの終了状態を参照します。

このコマンドは、データベースからすべての PostgreSQL ネイティブロジカルレプリケーションサブスクリプションと、レプリケーション起点とスロットを削除します（または、`-S` オプションが使用されている場合は無効にします）。

概要

```
bdr_init_physical [OPTION] ...
```


オプション

一般オプション

- `-D, --pgdata=DIRECTORY` -新しいノードに使用されるデータディレクトリ。空/存在しないディレクトリ、または `pg_basebackup -X stream` コマンド（必須）を使用して設定されたディレクトリのいずれかです。
- `-l, --log-file=FILE` -ロギングに `FILE` を使用します。デフォルトは `bdr_init_physical_postgres` です。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` -使用するレプリケーションセット名前のコンマ区切りリストの名前。指定しない場合、すべてのレプリケーションセットが使用されます。
- `--standby` -完全な送信/受信ノードではなく、ロジカルスタンバイ（受信専用ノード）を作成します。
- `--node-group-name` -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。
- `-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` - 空/存在しないデータディレクトリ（`-D` オプション）と組み合わせて使用すると、選択的に `pg_basebackup` を実行します。これは EDB Postgres Extended Server のみの機能です。
- `-S` -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、無効にします。

接続オプション

- `-d, --remote-dsn=CONNSTR` -リモートノードの接続文字列（必須）
- `--local-dsn=CONNSTR` -ローカルノードの接続文字列（必須）

構成ファイルのオーバーライド

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

注意事項

コマンドで指定されたレプリケーションセット名は、ノードが BDR グループに参加する前にデータディレクトリに存在するデータには影響しません。これは、`bdr_init_physical` が独自のベースバックアップを作成するか、既存のベースバックアップを新しい BDR ノードに昇格させるかに当てはまります。したがって、`--replication-sets` オプションは、ノードが BDR ノードグループに参加した後にパブリッシュおよびサブスクライブしたデータにのみ影響します。この動作は、ロジカル結合でレプリケーションセットが使用される方法、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、結合が完了した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判断し、サブスクライブしたレプリケーションセットのメンバーではないテーブルを特定します。次の理由から、テーブルは削除せずに切り捨てることを強くお勧めします。

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、他のノードで誤ってテーブルを削除する可能性があります。
2. 後でテーブルをレプリケーションセットに追加し、ノードのサブセットで削除した場合は、追加する前に DDL 競合を作成せずにそれらのノードでのみ再作成する必要があります。任意のレプリケーションセット。

レプリケートされないテーブルを切り捨てて、それらは存在しますが空のままにする方がはるかに簡単で安全です。

BDR の将来のバージョンでは、物理的結合用に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が自動的に省略または削除され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ないでください。

2.6 DDL Replication

DDL は「Data Definition Language」の略です。データベースオブジェクトを作成、変更、および削除する SQL 言語のサブセット。

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

- 一時的またはログされていないリレーション
- 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)
- テーブルメンテナンスコマンド (VACUUM、ANALYZE、CLUSTER、REINDEX)
- オートバキュームのアクション
- 操作コマンド (CHECKPOINT、ALTER SYSTEM)
- データベースまたはテーブルスペースに関するアクション

自動 DDL レプリケーションにより、DDL 変更をすべてのノードに手動で配布して一貫性を保証ことなく、特定の DDL 変更を簡単に make ことができます。

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。DDL をレプリケートするには、DDL レプリケーションフィルターをレプリケーションセットに追加する必要があります。

[DDL レプリケーションのフィルタリング](#) を参照してください。

BDR は、DDL レプリケーションに関してはスタンドアロンの PostgreSQL とは大きく異なり、同じように扱うことが BDR の最も一般的な操作上の問題です。

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、ステートメント自身自身をレプリケートすることです。これはほとんどの場合非常にうまく機能しますが、DDL はすべてのノードで同様に実行必要があります。より微妙な点は、DDL は、拡張機能（つまり、ビルトイン以外）によって導入されたデータ型を含む、すべてのデータ型固有のパラメータ設定に関して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デフォルトのエンコーディングで正しく実行される必要があります。

2.6.1 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` はデフォルトであり、デフォルトではすべてのノードを意味するデフォルトのレプリケーションセットに DDL を複製します。デフォルト以外のレプリケーションセットは、

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

ファンクション `bdr.replicate_ddl_command()` を使用して、特定のレプリケーションセットに DDL をレプリケートすることもできます。これは、ノードがダウンしているときに DDL コマンドを実行する場合、またはノードまたは担当者セットのサブセット（site1 のすべてのノードなど）にインデックスまたはパーティションを作成する場合に役立ちます。

```
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、`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もできます。

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

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

警告: グローバルロックを使用せずに各ノードで DDL を手動で実行すると、競合する DDL または DML が同時に実行されている場合に BDR グループ全体がレプリケーションを停止する場合があります。

`bdr.ddl_replication` パラメータは、`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。

2.6.2 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 サーバと同じではありません。これは、集中ロックやトランザクションコーディネーターのない非同期マルチマスターレプリケーションに基づいています。これは、DDL の実行時に重要な意味を持ちます。

並列実行される DDL は、BDR で引き続き実行されます。DDL の実行は、実行時に各ノードでの平行オペレーションに影響を与えるパラメーターを尊重するため、ノード間の設定の違いが目立つ場合があります。

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

BDR は、これらの問題に対して 3 つのレベルの保護を提供します。

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

`ddl_locking = on` は最も厳密なオプションであり、DDL が任意のノードから同時に実行される可能性があり、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が回避されるため、新しい空のデータベーススキーマを作成するときに役立ちます。

これらのオプションは、`bdr_superuser`、スーパーユーザ、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、そのファンクションに渡された DDL コマンドに対してのみ指定された `bdr.ddl_locking` 設定を使用して、3 番目の引数を介してこのパラメータを直接設定できます。

2.6.3 DDL ロックの詳細

BDR でレプリケートされた DDL の正確性を強制するために使用される 2 種類のロックがあります。

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 が実行されるのを防ぎます。これにより、一般的なケースでは完全な正確さが保証されますが、多くのシンプルケースには明らかに厳しすぎます。BDR は、スキーマが変更されるトランザクションで初めて DDL 操作のグローバルロックを取得します。これにより、クラスター内の DDL 実行トランザクションが効果的にシリアル化されます。言い換えれば、DDL の実行中は、別のテーブルに影響を与えたとしても、ノード上の他の接続は別の DDL コマンドを実行できません。

DDL 操作のロックを取得するには、DDL を実行する BDR ノードが BDR グループ内の他のノードに接続し、DDL を実行排他的権限を付与するように要求します。ロック要求は通常のレプリケーションストリー

ムを介して送信され、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。したがって、ノード（または少なくとも大部分のノード）がレプリケーションの遅延をあまり発生せずに実行されていることが重要です。そうしないと、ノードがDDL ロックを取得するまでに非常に時間がかかる場合があります。過半数のノードが同意すると、DDL の実行が実行されます。

DDL ロックの順序付けは、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で実行された以前のすべての DDL の効果を保証オーダーに、以前の DDL を実行したノードに追いつくまで待機します。現在の DDL を実行しているノードが以前の DDL を実行したノードに対してレプリケーションが遅れている場合、ロックの取得に非常に時間がかかります。したがって、単一のノード、または他のノードで発生したレプリケーションの変更にはほぼ追いついたノードから DDL を実行することをお勧めします。

2 番目の種類は、リレーション DML ロックと呼ばれます。この種のロックは、制約または一意性制約のいずれか検査制約で使用され、DDL ステートメントにより、実行中の DML ステートメントが失敗する可能性があります。リレーション DML ロックは、一度に 1 つのリレーションにのみ影響します。リレーション DML ロックにより、レプリケーションがエラーで停止保証可能性のある変更がキューにある間は DDL が実行されません。

テーブルでグローバル DML ロックを取得するには、DDL を実行している BDR ノードが BDR グループ内の他のすべてのノードに接続し、書き込みに対してテーブルをロックするように要求し、そのテーブルへの保留中の変更がすべて排出されるまで待機します。すべてのノードが完全に追いつくと、DML ロックの発信者はフリーにテーブルのスキーマ変更を実行し、他のノードにレプリケートできます。

グローバル DML ロックは各ノードのテーブルに EXCLUSIVE LOCK を保持するため、実行中にそのテーブルに対する DML、他の DDL、VACUUM、およびインデックスコマンドをブロックすることに注意してください。これは、通常は EXCLUSIVE LOCK 以上をとらないコマンドに対してグローバル DML ロックが保持されている場合でも当てはまります。

保留中の DML 操作が排出されるのを待機すると、時間がかかる場合があります。レプリケーションが現在遅れている場合。これは、データの変更とは異なり、行の表現形式とコンストレインに影響を与えるスキーマの変更は、構成されたすべてのノードに到達可能で、現在の書き込みレートに十分対応している場合にのみ実行できることを意味します。このような DDL コマンドをノードのダウン中に実行する必要がある場合は、最初にダウンしたノードを構成から削除する必要があります。

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

BDR システムでの DDL の実行 で説明されているように、ロックの動作は `bdr.ddl_locking`

パラメータで指定されます。

- `ddl_locking = on` はグローバル DDL ロックを取得し、必要に応じてリレーション DML ロックを取得します。
- `ddl_locking = dml` はグローバル DDL ロックをスキップし、必要に応じてリレーション DML ロックを取得します。
- `ddl_locking = off` は、グローバル DDL ロックとリレーション DML ロックの両方をスキップします。

また、一部の BDR ファンクションは DDL を変更するため、これらのファンクションには DDL ロックの動作が適用されることに注意してください。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が保証できる場合にのみ安全です。この設定では、グローバル DDL ロックのみを必要とするステートメントはグローバルロックをまったく使用しません。

`ddl_locking = off` は、DDL を実行データベースオブジェクトに競合する DDL がないこと、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問題が発生した場合、データへの進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、同時に実行される DDL を適切にシリアル化できます。これらのシリアル化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、昇格するまでロジカルスタンバイノードでアクティブになりません。

一部の BDR 管理レプリケーションは DDL のように機能することに注意して DDL。レプリケートされた関数の完全なリスト [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 には、グローバル DML ロックは必要ありません。

グローバル DDL ロックとグローバル DML ロックのモニタリングは、[モニタリングとロギング](#)の章に示されています。

2.6.4 DDL の影響の最小化

任意のデータベースに対する優れた運用上のアドバイス、これらの点は BDR ではさらに重要になります。

- DDL のインパクトを最小限に抑えるには、DDL を実行するトランザクションを短くし、多くの行の変更と組み合わせないでください。また、長時間実行される外部キーまたはその他の制約の再チェックを避ける必要があります。
- ALTER TABLE の場合、単に ADD CONSTRAINT を使用するのではなく、ADD CONSTRAINT NOT VALID、続いて VALIDATE CONSTRAINT を使用した別のトランザクションを使用してください。VALIDATE CONSTRAINT はすべてのノードでリプレイされるまで待機するため、確認を受け取るまでにかかなりの遅延が発生することに注意してください。
- インデックスを作成するときは、可能な限り CONCURRENTLY オプションを使用します。

実行時間の長い DDL を実行する別の方法は、DDL レプリケーションを無効にしてから、各ノードで DDL ステートメントを個別に実行です。これは、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。グローバルロックのルールは引き続き適用されるため、この種の使用法で自分自身をロックアウトしないように注意してください。これは、通常の使用法よりも回避策と見なす必要があります。

```
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 `CREATE INDEX CONCURRENTLY` で使用することをお勧めします。CREATE INDEX CONCURRENTLY はマルチトランザクションコマンドであるため、セッション

全体で DDL レプリケーションを無効にする必要があることに注意してください。CREATE INDEX は、実行中の書き込みを防止するため、稼働システムでは回避する必要があります。REINDEX は 3.6 までのバージョンで複製されますが、BDR 3.7 以降では複製されません。REINDEX が保持する AccessExclusiveLocks のため、REINDEX の使用は避ける必要があります。

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発生するよりも単一のトランザクションにまとめる方が役立つ場合があるため、DDL ロックは 1 回だけ取得する必要があります。テーブルレベルのロックが通常の操作を妨げる場合、これは望ましくない場合があります。

DDL がシステムを長時間停止している場合、他のステートメントをキャンセルするのと同じように、元のノードで DDL をキャンセルできます。たとえば、`psql` または `pg_cancel_backend()` を使用します。他のノードから DDL ロックをキャンセルすることはできません。

（オプション）グローバルロックのタイムアウト設定で、グローバルロックにかかる時間を制御できます。`bdr.global_lock_timeout` は、グローバルロックを取得するための待機がキャンセルされるまでにかかる時間を制限します。`bdr.global_lock_statement_timeout` は、グローバルロックを保持するトランザクション内のステートメントのランタイム長さを制限し、`bdr.global_lock_idle_timeout` は、グローバルロックを保持するトランザクションの最大許容アイドル時間（ステートメント間の時間）を設定します。これらのタイムアウトはすべて、値をゼロに設定することで無効にできます。

送信元ノードで 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、リプレイが確認され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である理由です。

2.6.5 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）を取得した後にダウンした場合、ロックはアクティブなままです。タイムアウトが設定されている場合でも、グローバルロックはタイムアウトしません。ノードが復旧すると、保持しているすべてのグローバルロックを自動的にリリースします。

長期間（または永久に）ダウンしたままの場合は、グローバルロックをリリースするオーダーにノードを BDR グループから削除します。これが、SET コマンドを `bdr_superuser` として使用して緊急 DDL を実行し、`bdr.ddl_locking` の値を更新する理由の 1 つかもしれません。

グローバルロックを確認した後、それを取得ロック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、

前のセクションで述べたように、グローバル DDL ロックはノードの過半数のみが応答する必要があるため、クラスターのパートがダウンしても、マジョリティが実行されていて到達可能である限りロックします。クラスター全体が使用可能な場合を除きます。

グローバル DDL またはグローバル DML ロックがあるときに別のノードがダウンした場合、コマンドは正常に続行され、ロックが解放されます。

2.6.6 ステートメント固有の DDL レプリケーションに関する懸念

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。このようなコマンドは、`bdr.ddl_replication` をオフにして DDL レプリケーションをオフにしない限り、通常は許可されません。

BDR は、データベースでアクティブなときに一部の DDL ステートメントの実行を防ぎます。これにより、正しくレプリケートされた DDL Statements With Restrictions で説明されています。

BDR で完全に許可されないコマンドは、禁止されている DDL ステートメントで説明されています。

BDR でステートメントが許可されていない場合、多くの場合、同じことを行う別の方法を見つけることができます。例、`volatile` のデフォルト値で列を追加する `ALTER TABLE` を実行することはできませんが、一般的には、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依存する DDL は BDR でレプリケートできず、DDL レプリケーションを有効にして実行するとエラーがスローされることに注意してください。

2.6.7 BDR DDL コマンド処理マトリックス

次の表に、許可されるユーティリティまたは DDL コマンド、レプリケートされるコマンド、およびレプリケート時に取得されるグローバルロックの種類を示します。

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

Command	Allowed	Repl- icated	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

次のページに続く

表 6 – 前のページからの続き

Command	Allowed	Repl- cated	Lock
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
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	bdr.global_consensus_journal_detail	Y	DDL
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	:ref: bdr.global_consensus_journal_detail	Y	DDL
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

次のページに続く

表 6 – 前のページからの続き

Command	Allowed	Repli- cated	Lock
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	:ref:`bdr.global_ddl_consensus_journal_details` <bdr.global_con	DDL
COMMIT	Y	N	N
COMMIT PREPARED	Y	N	N
COPY	Y	N	N
COPY FROM	Y	N	N
CREATE ACCESS METHOD	Y	Y	DDL
CREATE AGGREGATE	Y	Y	DDL
CREATE CAST	Y	Y	DDL
CREATE COLLATION	Y	Y	DDL
CREATE CONSTRAINT	Y	Y	DDL
CREATE CONVERSION	Y	Y	DDL
CREATE DATABASE	Y	N	N
CREATE DATABASE LINK	Y	Y	DDL
CREATE DIRECTORY	Y	Y	DDL
CREATE DOMAIN	Y	Y	DDL
CREATE EVENT TRIGGER	Y	Y	DDL
CREATE EXTENSION	Y	Y	DDL
CREATE FOREIGN DATA WRAPPER	Y	Y	DDL
CREATE FOREIGN TABLE	Y	Y	DDL
CREATE FUNCTION	Y	Y	DDL
CREATE INDEX	Y	Y	DML
CREATE LANGUAGE	Y	Y	DDL
CREATE MATERIALIZED VIEW	Y	N	N
CREATE OPERATOR	Y	Y	DDL
CREATE OPERATOR CLASS	Y	Y	DDL
CREATE OPERATOR FAMILY	Y	Y	DDL
CREATE PACKAGE	Y	Y	DDL
CREATE PACKAGE BODY	Y	Y	DDL
CREATE POLICY	Y	Y	DML
CREATE PROCEDURE	Y	Y	DDL
CREATE PROFILE	Y	Y	DDL
CREATE PUBLICATION	Y	Y	DDL
CREATE QUEUE	Y	Y	DDL
CREATE QUEUE TABLE	Y	Y	DDL
CREATE REDACTION POLICY	Y	Y	DDL
CREATE RESOURCE GROUP	Y	N	N

次のページに続く

次のページに続く

Command	Allowed	Repl- icated	Lock
CREATE ROLE	Y	Y	DDL
CREATE ROUTINE	Y	Y	DDL
CREATE RULE	Y	Y	DDL
CREATE SCHEMA	Y	Y	DDL
CREATE SEQUENCE	:ref:`bdr.global_m_consensus` DDL	Y	DDL DDL_details`<bdr.global_m_consensus` DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	:ref:`bdr.global_m_consensus` DDL	Y	DDL DDL_details`<bdr.global_m_consensus` DDL
CREATE TABLE AS	:ref:`bdr.global_m_consensus` DDL	Y	DDL DDL_details`<bdr.global_m_consensus` 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
DISCARD ALL	Y	N	N
DISCARD PLANS	Y	N	N
DISCARD SEQUENCES	Y	N	N
DISCARD TEMP	Y	N	N
DO	Y	N	N
DROP ACCESS METHOD	Y	Y	DDL
DROP AGGREGATE	Y	Y	DDL
DROP CAST	Y	Y	DDL
DROP COLLATION	Y	Y	DDL
DROP CONSTRAINT	Y	Y	DDL
DROP CONVERSION	Y	Y	DDL
DROP DATABASE	Y	N	N
DROP DATABASE LINK	Y	Y	DDL
DROP DIRECTORY	Y	Y	DDL
DROP DOMAIN	Y	Y	DDL

表 6 – 前のページからの続き

Command	Allowed	Repl- cated	Lock
DROP EVENT TRIGGER	Y	Y	DDL
DROP EXTENSION	Y	Y	DDL
DROP FOREIGN DATA WRAPPER	Y	Y	DDL
DROP FOREIGN TABLE	Y	Y	DDL
DROP FUNCTION	Y	Y	DDL
DROP INDEX	Y	Y	DDL
DROP LANGUAGE	Y	Y	DDL
DROP MATERIALIZED VIEW	Y	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
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

次のページに続く

表 6 – 前のページからの続き

Command	Allowed	Repl- cated	Lock
DROP USER MAPPING	Y	Y	DDL
DROP VIEW	Y	Y	DDL
EXECUTE	Y	N	N
EXPLAIN	Y	:ref:`bdr.global_consensus_journal_details`<bdr.global_con	
FETCH	Y	N	N
GRANT	Y	:ref:`bdr.global_consensus_journal_details`<bdr.global_con	
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	:ref:`bdr.global_consensus_journal_details`<b
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	:ref:`bdr.global_consensus_journal_details`<bdr.global_con	
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	:ref:`bdr.global_consensus_journal_details`<bdr.global_con	
SELECT INTO	:ref:`bdr.global_consensus_journal_details`<bdr.global_con	DDL	
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	:ref:`bdr.global_consensus_journal_details`<bdr.global_con	
UNLISTEN	Y	N	N
VACUUM	Y	N	N

ALTER シーケンス

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

ALTER SEQUENCE ... RENAME は、galloc シーケンスではサポートされていません (のみ)。ALTER SEQUENCE ... SET SCHEMA は、galloc シーケンスではサポートされていません (のみ)。

ALTER テーブル

通常、ALTER TABLE コマンドが許可されます。ただし、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。

ALTER TABLE の許可されないコマンド

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

- ADD COLUMN ... DEFAULT (non-immutable expression) - 現在、異なるノードで異なるデータが得られるため、これは許可されません。推奨される回避策については、

[列の追加](#) を参照してください。

- ADD CONSTRAINT ... EXCLUDE - 除外コンストレインは現在サポートされていません。除外コンストレインは非同期システムではあまり意味がなく、再生できない変更につながります。
- ALTER TABLE ... SET WITH[OUT] OIDS - CREATE TABLE と同じ理由でサポートされていません。
- ALTER COLUMN ... SET STORAGE external - 列がテーブルのレプリカアイデンティティの列の場合、拒否されます。
- 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 - コマンドによってテーブル全体が書き換えられる場合、列の型の変更はサポートされていません。これは、変更がバイナリ強制互換ではない場合に発生します。バイナリ強制互換変更は一方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ないため、通常は許可されません。レプリケートされない ALTER COLUMN ... TYPE の場合、列が新しい型に自動的にキャスト可能な場合は許可されます (USING 句が含まれていません)。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、大規模なテーブルで拡張 AccessExclusiveLock を保持するため、いずれの場合も、このようなコマンドは高可用性データベースでは実行できない可能性があります。推奨される回避策については、

[列のタイプを変更する](#) を参照してください。

- ALTER TABLE ... ADD FOREIGN KEY - 現在のユーザに被参照のテーブルを読み取るパーミッションがない場合、または被参照のテーブルで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る場合、サポートされません。

次の例は、`timestamp` 型の定数数値を `timestampz` 型の列に追加しようとするため失敗します。`timestamp` と `timestampz` 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 timestampz DEFAULT timestamp 2100-01-01 00:00:00 NOT NULL;
```

BDR 3.7.4 以降、CHECK および FOREIGN KEY コンストレインなどの特定のタイプのコンストレインは、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、`ALTER TABLE ... VALIDATE CONSTRAINT` コマンドを使用して別のトランザクションで制約を検証するという 2 段階のプロセスが必要です。詳細については、[CONSTRAINT の追加](#) を参照してください。

ALTER TABLE のロック

ALTER TABLE の次のバリエーションは DDL ロックのみを取得し、DML ロックを取得しません。

- ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT
- ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression
- ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT
- 書き換えが必要ない場合は ALTER TABLE ... ALTER COLUMN ... TYPE
- ALTER TABLE ... ALTER COLUMN ... SET STATISTICS
- ALTER TABLE ... VALIDATE CONSTRAINT
- ALTER TABLE ... ATTACH PARTITION
- ALTER TABLE ... DETACH PARTITION
- ALTER TABLE ... ENABLE TRIGGER (ENABLE REPLICA TRIGGER は引き続き DML ロックを取得します)
- ALTER TABLE ... CLUSTER ON
- ALTER TABLE ... SET WITHOUT CLUSTER
- ALTER TABLE ... SET (storage_parameter = value [, ...])
- ALTER TABLE ... RESET (storage_parameter = [, ...])
- ALTER TABLE ... OWNER TO

ALTER TABLE の他のすべてのバリエーションは、変更されるテーブルで DML ロックを取得します。ALTER TABLE の一部のバリエーションには、以下に示す制限があります。

ALTER TABLE の例

この次の例は、型の変更がバイナリ強制互換であり、テーブルの書き換えが発生しないため、機能するため、カタログのみの変更として実行されます。

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

ただし、VARCHAR(128) から VARCHAR(20) への変更はバイナリ強制互換ではないため、この変更を行って上記のコマンドを元に戻すことはできません。

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

推奨される回避策については、後述を参照してください。

一般および非レプリケート環境で可能なさまざまなタイプの ALTER TABLE ...ALTER COLUMN TYPE (ATCT) 操作にコンテキストを提供すると役立ちます。

一部の ATCT 操作は、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、既存の列の型とターゲットの型がバイナリ強制互換です。例：

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

バイナリ強制性のため、列の型を VARCHAR または TEXT データ型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メタデータの更新です。

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

ただし、col2 のサイズを減らしたい場合、それは基になるテーブルデータの書き換えにつながります。通常、テーブルの書き換えは制限されています。

```
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ERROR: ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect
↳ replicated tables on a BDR node
```

非テキスト型の例を示すには、INTEGER 型の上記の col3 を検討してください。SMALLINT または BIGINT に変換しようとする ATCT 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
```

上記の両方の失敗したケースでは、現在の型からターゲット型への自動代入キャストが存在します。ただし、バイナリ強制はなく、基になるテーブルデータの書き換えが発生します。

このような場合、制御された DBA 環境では、すべてのノードの非レプリケート環境でこの列の型のローリングアップグレードを 1 つずつ採用することにより、列の型を自動的にキャスト可能なものに変更できます。DDL がレプリケートされず、列型が上記のように自動的にキャスト可能なものに変更される場合、

この同じテーブルの他のノードでの同時アクティビティとともに、変更を実行するノードでローカルに書き換えを許可できます。このレプリケートされない ATCT オペレーションをすべてのノードで 1 つずつ繰り返すことで、BDR クラスター全体で列タイプを必要に応じて変更できます。これには書き換えが含まれるため、アクティビティは短期間でも 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ることに注意してください。上記の詳細を設定したら、レプリケートされない変更アクティビティのローリングアップグレードを以下のように実行できます。

```
- - foreach node in BDR cluster do:
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(64);
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
RESET bdr.ddl_replication;
- - done
```

多くのデータ型で自動割り当てキャストを使用できるため、このローカルのレプリケートされない ATCT 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、USING 句を使用する ATCT 操作は、自動割り当てキャストがないために失敗する可能性がある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いいくつかの一般的な変換について以下に説明します。

```
- - 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、レプリケートされない環境で許可される可能性のある ATCT 操作の完全なリストではありません。明らかに、すべての ATCT 操作が機能するわけではありません。DDL レプリケーションを無効にしても、自動割り当てができない場合は失敗します。したがって、数値型からテキスト型への変換はレプリケートされていない環境では機能しますが、テキスト型から数値型への変換は失敗します。

```
SET bdr.ddl_replication TO FALSE;
- - conversion from BIGINT to TEXT works
ALTER TABLE sample ALTER COLUMN col3 TYPE TEXT;
- - conversion from TEXT back to BIGINT fails
ALTER TABLE sample ALTER COLUMN col3 TYPE BIGINT;
ERROR: ALTER TABLE ... ALTER COLUMN TYPE which cannot be automatically cast to new
↳type may not affect replicated tables on a BDR node
RESET bdr.ddl_replication;
```

レプリケートされない環境での ATCT 操作はさまざまな型変換をサポートしていますが、基になるテーブルデータに新しいデータタイプに割り当てることができない値が含まれている場合、書き換えが失敗する可能性があることにノートすることが重要です。例、列の現在のタイプは VARCHAR(256) である可能性があり、非複製の ATCT オペレーションを試みて VARCHAR(128) に変換しました。テーブルに 128 バイトを超える既存のデータがある場合、書き換えオペレーション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している場合、書き換えは成功します。ただし、他のノード（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をまだ実行していない）がこのローカル ATCT オペレーションに同時に 128 バイトを超える新しいデータを導入した場合、レプリケーションが失敗する可能性があります。これにより、クラスターでのレプリケーションが停止します。したがって、これらのレプリケートされないローリングデータタイプアップグレード操作を実行するときは、データベースおよびアプリケーションレベルでのデータタイプの制限と特性に注意することが重要です。このような ATCT 操作は、制御された完全に認識された DBA 環境で実行およびテストすることを強くお勧めします。これらの ATCT 操作は非対称であり、失敗した特定の変更をバックアウトすると、テーブルの書き換えが長時間続く可能性があ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は実行できないことにノートしてください。

ALTER タイプ

ノートは複製されロックが、PostgreSQL はこれらの依存関係をレコードしないため、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。

コメント

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

シーケンスを作成する

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

テーブルを作成する

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

CREATE TABLE AS と SELECT INTO

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

説明する

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

EXPLAIN ANALYZE レプリケーション

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

EXPLAIN ANALYZE ロック

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

付与と取り消し

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

ロックテーブル

LOCK TABLE は複製されませんが、bdr.lock_table_locking が on に設定されている場合にグローバル DML ロックを取得する場合があります。

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

セキュリティラベル

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

TRUNCATE レプリケーション

TRUNCATE コマンドは DDL ステートメントではなく 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よって異なります。

TRUNCATE ロック

TRUNCATE は他の DDL と同じように複製されませんが、`bdr.truncate_locking` が `on` に設定されている場合、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

ユーザは PostgreSQL インスタンスのグローバルオブジェクト。つまり、BDR は個々のデータベースレベルで動作しマルチプル。これは、ロール操作ステートメントのハンドリングには追加の考慮がニーズであることを意味します。

BDR では、レプリケートされた DDL によって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 が有効になった操作ロール場合（デフォルト）、*BDR 対応データベースでロール操作ステートメントが実行されます。*

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。

- ロールを作成する
- ALTER ROLE
- ドロップロール
- グラントロール
- ユーザーを作成
- ALTER USER
- DROP USER
- グループを作成
- ALTER グループ
- ドロップグループ

一般に、次のいずれかです。

- システムは `bdr.role_replication = off` で構成する必要があり、すべてのロール（ユーザとグループ）の変更は、Ansible、Puppet、Chef などの外部オーケストレーションツールで展開するか、`bdr.replicate_ddl_command(...)` を介して明示的に複製する必要があります。または
- PostgreSQL インスタンス上の 1 つの BDR 対応データベースに `bdr.role_replication = on` があり、すべてのロール管理 DDL がそのデータベースで実行されるように、システムを構成する必要があります。

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

ロールのレプリケーションがオフになっている場合、管理者は、1つのノードで DDL が使用するロールが他のノードにも存在することを保証必要があります。そうでない場合、他のノードでロールが作成されるまで BDR apply は ERROR で停止します。

注：BDR は、BDR 対応の PostgreSQL インスタンス内の BDR 非対応の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複製しません。例、DB ‘bdrdb’ (bdr group メンバ) と ‘postgres’ (bare db)、および `bdr.role_replication = on` がある場合、`bdrdb` で実行される `CREATE USER` は複製されますが、`postgres` で実行される `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一部は回避できます。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結果が生成される場合があります。

CONSTRAINT の追加

CHECK および FOREIGN KEY 制約は、DML ロックを必要とせずに追加できます。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`

これらの手順は、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。これらの手順はどちらもテーブルの 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。ただし、制約を検証するオーダーには、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他のすべてのノードからのレプリケーション変更を適用したことを保証する必要があります。それらのノードに制約を作成します。したがって、新しいメカニズムでは制約の検証中にすべてのノードを稼働させる必要はありませんが、すべてのノードが `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID` コマンドを適用して十分に進行している必要があります。BDR は、制約を検証する前に一貫した状態に到達するのを待ちます。

新しい機能では、クラスターが Raft プロトコルバージョン 24 以降で実行される必要があることに注意してください。Raft プロトコルがまだアップグレードされていない場合、古いメカニズムが使用され、DML ロック要求が発生します。

列の追加

volatile デフォルトを使用して列を追加するには、別のトランザクションで次のコマンドを実行します。

```
ALTER TABLE mytable ADD COLUMN newcolumn coltype; -- Note the lack of DEFAULT or NOT NULL
--> NULL

ALTER TABLE mytable ALTER COLUMN newcolumn DEFAULT volatile-expression;

BEGIN;
SELECT bdr.global_lock_table(mytable);
UPDATE mytable SET newcolumn = default-expression;
COMMIT;
```

これにより、スキーマの変更と行の変更が BDR で実行できる個別のトランザクションに分割され、BDR グループ内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、更新をバッチにまとめて、一度に数万または数十万を超える行を更新しないようにします。これは、トランザクションが埋め込みた PROCEDURE を使用して実行できます。

変更の最後のバッチは、テーブルでグローバル DML ロックを取得するトランザクションで実行されることが重要です。そうしないと、他のノードのテーブルに同時に挿入される行を見落とす可能性があります。

必要に応じて、UPDATE の終了後に ALTER TABLE mytable ALTER COLUMN newcolumn NOT NULL; を実行できます。

列のタイプを変更する

PostgreSQL は、次の例のように、回避できる場合にテーブルの書き換えを発生させます。

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

このステートメントは、制限をデータ型の変更ではなくテーブル制約に制約することにより、テーブルのリライトを回避するようにリライトできます。必要に応じて、後続のコマンドで検証して長いロックを回避できます。

```
CREATE TABLE foo (id BIGINT PRIMARY KEY, description VARCHAR(128));
ALTER TABLE foo
  ALTER COLUMN description TYPE varchar,
  ADD CONSTRAINT description_length_limit CHECK (length(description) <= 20) NOT VALID;
ALTER TABLE foo VALIDATE CONSTRAINT description_length_limit;
```

バリデーションが失敗した場合、失敗した行のみを UPDATE できます。この手法は、length() を使用して TEXT および VARCHAR、または scale() を使用して NUMERIC データ型に使用できます。

列の種類を変更する一般的な場合、最初に目的の種類の列を追加します。

```
ALTER TABLE mytable ADD COLUMN newcolumn newtype;
```

BEFORE INSERT OR UPDATE ON mytable FOR EACH ROW ..として定義されたトリガーを作成します。これは、テーブルへの新しい書き込みで新しい列が自動的に更新されるように、NEW.newcolumn を NEW.oldcolumn に割り当てます。

テーブルをバッチで UPDATE、埋め込みトランザクションで PROCEDURE を使用して oldcolumn の値を newcolumn にコピーします。作業をバッチ化すると、大きなテーブルの場合、レプリケーションの遅延を削減できヘルプ。ID のレンジまたは好みのメソッドで更新するか、小さなテーブルの場合はテーブル全体を一度に更新します。

CREATE INDEX ... 新しい列に必要なインデックス。ロック時間を短縮するには、各ノードで DDL レプリケーションなしで個別に実行しても安全です。

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

BEGIN トランザクション、DROP 追加したトリガー、ALTER TABLE は列に必要な DEFAULT を追加します。

列を削除しているため、テーブルに依存するビュー、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。列を ``CASCADE`` ドロップする場合は、それを参照していたものを保証再作成する必要があるの
ので注意してください。

他のタイプの変更

ALTER TYPE ステートメントは複製されますが、影響を受けるテーブルはロックされません。

この DDL を使用する場合、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たことを保証します。

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、DDL レプリケーションがアクティブで DDL フィルター設定で許可されている場合、グローバルロックを取得しようとし、アクションがレプリケートされることを意味します。詳細については、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`

コンフリクトマネジメント

- `bdr.alter_table_conflict_detection`
- `bdr.column_timestamps_enable`
- `bdr.column_timestamps_disable`

シーケンス管理

- `bdr.alter_sequence_set_kind`

ストリーミングトリガー

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

2.7 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` 拡張機能を設定し、非スーパーユーザーが BDR を作成できるように構成できます。

BDR の構成と管理にはスーパーユーザーアクセスは必要なく、お勧めしません。BDR に必要な権限は、PostgreSQL のデフォルトの/事前定義されたロールと同様に名前付けた、次のデフォルトの/事前定義されたロールに分割されます。

- `bdr_superuser` - 最高の特権ロール、すべての BDR テーブルとファンクションへのアクセス。
- `bdr_read_all_stats` - BDR の状態を理解するのに十分な、テーブル、ビュー、およびファンクションへの読み取り専用アクセスを持つロール。
- `bdr_monitor` - 現時点では `bdr_read_all_stats` と同じですが、後で拡張します。
- `bdr_application` - BDR を実行しているアプリケーションに必要な最小限の権限。
- `bdr_read_all_conflicts` - `bdr.conflict_history` ですべての競合をビューできます。

これらの BDR ロールは、BDR 拡張機能のインストール時に作成されます。詳細については、BDR Default Roles を参照してください。

BDR を管理するために、管理者がユーザーデータにアクセスする必要はありません。

競合を保護するための手配については、ここで説明されています [テーブルへの競合のログ](#)。

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

2.7.1 カタログテーブル

システムカタログと情報スキーマテーブルは、常に BDR によるレプリケーションから除外されます。

さらに、拡張機能が所有するテーブルはレプリケーションから除外されます。

2.7.2 BDRFunctions と演算子

すべての BDR 関数は `bdr` スキーマで公開されています。これらの関数への呼び出しは、`search_path` に `bdr` を置くのではなく、スキーマで修飾する必要があります。

すべての BDR 演算子は `pg_catalog` スキーマを介して使用できるため、ユーザーは `search_path` から `public` スキーマを問題なく除外できます。

2.7.3 カタログオブジェクトに対する特権の付与

管理者は、テーブル、ビュー、ファンクションなどのカタログオブジェクトに対する明示的な特権を付与しないでください。*BDR Default Roles* に記載されているロールのいずれかを付与することにより、これらのオブジェクトへのアクセスを管理します。

この要件は、結合のいずれかの側のノードの BDR（したがって BDR カタログ）のバージョンが完全に同じでなくても、ノードグループに参加できる柔軟性の結果です。

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

2.7.4 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。`CREATE USER` および `CREATE ROLE` コマンドは、BDR が実行されており、`bdr.role_replication` がオンになっているデータベースで実行されると、自動的に複製されます。ただし、これらのコマンドが同じ PostgreSQL インスタンス内の他のデータベースで実行される場合、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、これらは複製されません。

新しい BDR ノードが BDR グループに参加しても、`bdr_init_physical` を使用してノードを追加しない限り、既存のユーザーは自動的にコピーされません。これは意図的なものであり、重要なセキュリティ機能です。PostgreSQL では、ユーザーはマルチプルのデータベースにアクセスできます。デフォルトでは任意のデータベースにアクセスできます。BDR は、どのユーザーがどのデータベースにアクセスするかを知らないため、新しいノードにコピーするユーザーを安全に決定できません。

PostgreSQL では、次のコマンドですべてのユーザーをダンプできます。

```
pg_dumpall --roles-only > roles.sql
```

次に、ファイル `roles.sql` を編集して、不要なユーザーを削除してから、新しく作成したノードで再実行します。ID とアクセス管理ソリューション (IAM) によっては他のメカニズムも可能ですが、現時点では自動化されていません。

2.7.5 ロールとレプリケーション

ユーザが実行した DDL 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 変更は、ターゲットノードでテーブル所有ユーザとして複製されます。テーブルは各ノードで同じユーザが所有することをお勧めしますが、強制ではありません。

テーブル A がノード 1 でユーザ X が所有し、ノード 2 でユーザ Y が所有している場合、ユーザ Y がユーザ X よりも高い権限を持っている場合、これは権限エスカレーションと見なすことができます。一部のノードにはさまざまなユースケースがあるため、これを許可しますが、セキュリティ管理者がこのシチュエーションを計画および監査できるように警告します。

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レプリケートされます。これは、`FORCE ROW LEVEL SECURITY` が指定されている場合でも、`NO FORCE ROW LEVEL SECURITY` として適用される変更と同等です。これが望ましくない場合は、すべての行の複製を回避する `row_filter` を指定します。すべてのノードの行セキュリティポリシーを同一にするか、少なくとも互換性があるようにすることをお勧めしますが、強制はしません。

`bdr_superuser` は BDR のレプリケーションを制御し、任意のレプリケーションセットから任意のテーブルを追加/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、持つこともお勧めしません。レプリケーションセットファンクションへのアクセスを制限する必要がある場合、これらのファンクションの制限バージョンを適切なユーザーに `SECURITY DEFINER` ファンクションおよび `GRANT ed` として実装できます。

2.7.6 接続ロール

新しい BDR ノードを割り当てるとき、`bdr.create_node` の `local_dsn` 引数および `bdr.join_node_group` の `join_target_dsn` に DSN で指定されたユーザは、データベースオブジェクトを参照、作成、および管理するために頻繁に使用されます。

BDR は、これらの DSN で `SUPERUSER` 権限を持つロールを使用する場合でも権限昇格攻撃を防ぐように注意深く作成されています。

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

- ユーザは `REPLICATION` 属性を持っています
- データベースに対する `CREATE` パーミッションが付与されています
- `bdr_superuser` ロールを継承します
- 直接または所有者ロールからの権限を介して、レプリケートするすべてのデータベースオブジェクトを所有しています。

すべてのノードが参加したら、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、権限を次のように減らすことができます。

- ユーザは REPLICATION 属性を持っています。
- bdr_superuser ロールを継承します。

2.7.7 権限制限

BDR は追加の制限を適用し、TRIGGER または REFERENCES 権限のみに依存する DDL の使用を効果的に防止します。以下のサブセクションでは、これらについて説明します。

GRANT ALL は引き続き TRIGGER と REFERENCES の両方の特権を付与するため、特権を明示的に記述することをお勧めします。ALL の代わりに GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE。

外部キー特権

ALTER TABLE ... ADD FOREIGN KEY は、ユーザが参照先のテーブルに対する SELECT 権限を持っている場合、または被参照のテーブルで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る場合にのみサポートされます。

したがって、BDR で外部キーを作成するには REFERENCES 権限では不十分です。REFERENCES 権限のみに依存することは、通常、有用ではありません。バリデーションチェックをテーブルスキャンではなくトリガーを使用して実行ため、通常、コストが高すぎます。

トリガー

PostgreSQL では、テーブルの所有者と TRIGGER 権限を付与されたユーザーの両方がトリガーを作成できます。非テーブル所有者によって付与されたトリガーは、BDR でテーブル所有者として実行され、セキュリティの問題が発生する可能性があります。TRIGGER 権限はめったに使用されず、PostgreSQL コアチームは「個別の TRIGGER パーミッションは廃止されたものと考えています」と述べています。

BDR は、テーブルにトリガーを作成できるユーザーに関するより厳密なルールを使用することにより、この問題を軽減します。

- スーパーユーザ
- bdr_superuser
- テーブルの所有者は、PostgreSQL と同じルールに従ってトリガーを作成できます（トリガーで使用する関数の EXECUTE 権限が必要です）。
- テーブルの TRIGGER 権限を持つユーザーは、テーブルと同じ所有者が所有する関数を使用してトリガーを作成し、標準の PostgreSQL ルールを満たす場合にのみトリガーを作成できます（再度、関数の EXECUTE 権限が必要です）。したがって、テーブルと関数の両方が同じ所有者を持ち、所有者がユーザにテーブルの TRIGGER 権限と関数の EXECUTE 権限の両方を付与することにした場合、そのユーザはこの関数を使用してそのテーブルにトリガーを作成してもよいと想定され関数。

- 表に対する TRIGGER 権限を持つユーザは、EXECUTE 権限を持っている場合、SECURITY DEFINER 句で定義されたファンクションを使用してトリガーを作成できます。この句は、標準の PostgreSQL と BDR の両方で、ファンクションを常にファンクション自分自身の所有者のコンテキストで実行します。

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、そのトリガーによって使用され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、テーブルの所有者が所有せず、SECURITY DEFINER ファンクションを使用しないトリガーが既存のテーブルにある場合、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、`bdr.backwards_compatibility` を 30618（またはそれ以下）に設定して、以前のバージョンのように動作させることができます。

BDR レプリケーションの適用は、システムレベルのデフォルトの `search_path` のみを使用します。レプリカトリガー、ストリームトリガー、およびインデックス式関数は、他の `search_path` 設定を想定する場合がありますが、適用時に実行されます。これが発生し保証にするには、デフォルトの `search_path` のみを使用してオブジェクト参照を明確に解決するか（常にオブジェクトへの完全修飾参照、たとえばスキーマ.objectname を使用します）、`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。

2.7.8 BDR のデフォルト/事前定義されたロール

BDR の事前定義されたロールは、BDR 拡張機能のインストール時に作成されます。BDR 拡張機能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ることに注意してください。これにより、BDR を同じ PostgreSQL インスタンスのマルチプルデータベースで問題なく使用できます。

`GRANT ROLE DDL` ステートメントは BDR レプリケーションに参加しないため、クラスターの各ノードで実行が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権限
- スキーマ BDR のすべてのルーチンのすべての権限

`bdr_read_all_stats`

に対する SELECT 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`

に対する 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` のすべての特権にプラス
に対する EXECUTE 権限

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

bdr_application

に対する 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`

上記のファンクションの多くには、使用する前に追加の権限が必要であることに注意してください。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ます。これらの追加ルールは、特定の各ファンクションとともに文書化されています。

bdr_read_all_conflicts

BDR は、競合を `bdr.conflict_history` テーブルに記録します。競合はテーブルの所有者（のみ）に可視されるため、競合の履歴を読み取るために追加の権限は必要ありません。**すべての**テーブルの競合を確認できるユーザが便利な場合は、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。

2.7.9 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ました。

- ミスラ C
- ISO 26262
- ISO / IEC TS 17961
- OWASP トップ 10
- CERT C

- CWE トップ 25
- AUTOSAR

CIS ベンチマーク

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

	Result	Description
1.4	PASS	systemd サービスファイルが有効になっていることを確認する
1.5	PASS	データクラスターが正常に初期化されたことを確認する
2.1	PASS	ファイルのアクセス許可マスクが正しいことを確認します
2.2	PASS	PostgreSQL pg_wheel 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する
3.1.4	PASS	ログファイルの宛先ディレクトリが正しく設定されていることを確認します
3.1.5	PASS	ログファイルのファイル名パターンが正しく設定されていることを確認します
3.1.6	PASS	ログファイルのアクセス許可が正しく設定されていることを確認します
3.1.7	PASS	「log_truncate_on_rotation」が有効になっていることを確認します
3.1.8	PASS	ログファイルの最大存続時間が正しく設定されていることを確認します
3.1.9	PASS	最大ログファイルサイズが正しく設定されていることを確認します
3.1.10	PASS	正しい syslog 機能が選択されていることを確認します
3.1.11	PASS	PostgreSQL syslog メッセージのプログラム名前が正しいことを確認します
3.1.14	PASS	「debug_print_parse」が無効になっていることを確認します
3.1.15	PASS	「debug_print_rewriting」が無効になっていることを確認します
3.1.16	PASS	「debug_print_plan」が無効になっていることを確認します
3.1.17	PASS	「debug_pretty_print」が有効になっていることを確認します
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 7 – 前のページからの続き

	Result	Description
3.1.19	PASS	「log_disconnections」が有効になっていることを確認します
3.1.21	PASS	「log_hostname」が正しく設定されていることを確認します
3.1.23	PASS	「log_statement」が正しく設定されていることを確認します
3.1.24	PASS	「log_timezone」が正しく設定されていることを確認します
3.2	PASS	PostgreSQL Audit Extension (pgAudit) が有効になっていることを確認します
4.1	PASS	sudo が正しく構成されていることを確認します
4.2	PASS	過剰な管理者権限が取り消されていることを確認する
4.3	PASS	過剰なファンクション権限が取り消されていることを確認する
4.4	PASS	テスト済み 過剰な DML 権限が取り消されていることを確認する
5.2	Not Tested	「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ーが正しく構成されていることを確認します
6.7	Not Tested	FIPS 140-2 OpenSSL 暗号化が使用されていることを確認する
6.8	PASS	SSL が有効になっており、正しく構成されていることを確認します
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する

テスト 5.2 は手動で監査された場合に合格できますが、自動化可能なテストがないことに注意してください。

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 バリエーションには追加のパッケージソフトが必要です。

2.8 Conflicts

BDR は、アクティブ/アクティブまたはマルチマスターの DBMS です。非同期で使用すると、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しているときにデータの競合が発生する可能性があります。

競合はエラーではありません。ほとんどの場合、これらは BDR が発生時に検出および解決できるイベントです。解決策はアプリケーションの性質とデータの意味によって異なるため、BDR がアプリケーションに競合を解決する方法に関するレンジな選択肢を提供することが重要です。

デフォルトでは、競合行レベルで解決されます。2つのノードからの変更が競合する場合、ローカルまたはリモートのタプルが選択され、もう一方は破棄されます。例、2つの競合する変更のコミットのタイム

スタンプを比較し、新しい方を保持します。このアプローチにより、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスが確立されます。

列レベルの競合解決の有効化と無効化 で説明されているように、競合ハンドリングは構成可能です。

Stream Triggers で説明されている競合トリガーを使用して、テーブルごとに競合を検出して処理できます。

CLCD で説明されているように、列レベルの競合検出と解決は BDR で使用できます。

競合を回避したい場合は、BDR でこれらの機能を使用できます。

- *bdr.crdt_handlers* で説明されている競合のないデータ型 (CRDT)。
- *Eager Replication* で説明されている Eager Replication。

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、テーブルの所有者は競合をモニタして回避する方法を分析するか、アプリケーションタスクとして定期的にハンドルする計画を make 必要があります。

LiveCompare によるデータ検証 ツールを使用して、発散を定期的にスキャンすることもできます。

一部のクラスタリングシステムは、分散ロックメカニズムを使用してデータへの同時アクセスを防ぎます。これらは、サーバーが互いに非常に近い場合は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。

分散ロックは本質的に悲観的なアプローチです。BDR は、可能な場合は競合を回避しますが、特定の種類の競合の発生を許可し、発生時に解決するという楽観的なアプローチを提唱しています。

2.8.1 競合はどのように発生するか

ノード間の競合は、関連するすべてのトランザクションが同じノードで同時に発生する場合に発生しないイベントの結果として発生します。ノードはトランザクションがコミットした後にのみ変更を交換するため、各トランザクションはコミットされたノードで個別に有効です。他の競合する作業を同時に行った別のノードに適用された場合、無効です。

BDR レプリケーションは本質的に他のノードでトランザクションをリプレイするため、適用されるトランザクションと受信ノードでコミットされたトランザクションの間に競合がある場合、リプレイオペレーションが失敗する可能性があります。

Postgres には、UNIQUE インデックス、SEQUENCE 操作、行とリレーションのロック、SERIALIZABLE 依存関係の追跡など、それを防ぐためのトランザクション間通信メカニズムがあるため、すべてのトランザクションが単一のノードで実行されている場合、ほとんどの競合は発生しません。これらのメカニズムはすべて、進行中のトランザクション間で通信して、望ましくない同時実行の問題を防ぐ方法です。

BDR には分散トランザクションマネージャ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優れたパフォーマンスを発揮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と、異なるノード上のトランザクションが互いに完全に独立して実行されます。ノード間の独立性が低いと競合を完全に回避できます。そのため、BDR はこれが重要な場合に Eager Replication も提供します。

2.8.2 競合の種類

PRIMARY KEY または UNIQUE の競合

最も一般的な競合行の競合です。2つの操作が、単一のノードではできない方法で同じキーの行に影響します。BDR はそれらのほとんどを検出でき、`update_if_newer` 競合リゾルバを適用します。

行の競合には次のものが含まれます。

- INSERT と INSERT
- UPDATE と UPDATE
- UPDATE と DELETE
- INSERT と UPDATE
- INSERT と DELETE
- DELETE と DELETE

ビュー `bdr.node_conflict_resolvers` は、既知のすべての競合タイプの競合解決が現在どのように構成されているかに関する情報を提供します。

INSERT / INSERT の競合

最も一般的な競合 INSERT/INSERT は、2つの異なるノードでの INSERT 操作が同じ PRIMARY KEY 値（または、PRIMARY KEY が存在しない場合は単一の UNIQUE 制約）を持つタプルを作成する場合に発生します。

BDR は、この動作がユーザ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、発信元ノードのタイムスタンプに従って、2つのうちの最後に挿入されたタプルを保持することにより、このシチュエーションを処理します。

この競合は `insert_exists` 競合タイプを生成します。これは、デフォルトでは、より新しい（コミット時間に基づいて）行を選択し、その行のみを保持することで解決されます（`update_if_newer` リゾルバ）。他のリゾルバーを構成できます。詳細は [列レベルの競合解決の有効化と無効化](#) を参照してください。

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

[BDR グローバルシーケンス](#) を使用すると、この種の競合を効果的に解消できます。

マルチプルの UNIQUE コンストレインに違反する INSERT 操作

INSERT/INSERT の競合は、複数の UNIQUE 制約に違反する場合があります（そのうちの 1 つが PRIMARY KEY である場合があります）。新しい行が複数の UNIQUE 制約に違反し、他の複数の行に対して競合が発生する場合、レプリケーション変更を適用すると `multiple_unique_conflicts` 競合が発生します。

このような競合が発生した場合、レプリケーションを続行するにはいくつかの行を削除する必要があります。 `multiple_unique_conflicts` のリゾルバ設定に応じて、適用プロセスはエラーで終了するか、入力

行をスキップするか、一部の行を削除します。削除では、正しい PRIMARY KEY で行を保持し、他の行を削除しようとしています。

警告: この方法でマルチプルの行が競合している場合、競合解決の結果が挿入オペレーションを続行する場合、一部のデータは常に削除されます。

競合トリガーを使用して別の動作を定義することもできます。

UPDATE/UPDATE の競合

異なるノードでの 2 つの同時の UPDATE 操作が同じタプル（ただし、その PRIMARY KEY ではない）を変更する場合、UPDATE / リプレイの競合が発生する可能性があります。

これらは、構成とシチュエーションに基づいてさまざまな種類の競合を生成できます。テーブルが **行バージョンの競合検出** で構成されている場合、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 競合が生成されます。**オリジンの競合検出** を使用する場合、行のオリジンがチェックされます（オリジンは現在のローカル行の元のノードです）。それが変更された場合、update_origin_change 競合が生成されます。他のすべての場合、通常は競合を生成せずに UPDATE が適用されます。

これらの競合は両方とも、**INSERT/INSERT の競合** で説明されている insert_exists と同じ方法で解決されます。

PRIMARY KEY での UPDATE の競合

BDR は現在、PRIMARY KEY が UPDATE オペレーションによって変更された競合解決を実行できません。プライマリキーは更新できますが、既存の値との競合が発生しないことを保証必要があります。

プライマリキーの更新の競合は **発散する競合** であり、マニュアル介入が必要です。

Postgres でプライマリキーを更新できますが、Postgres と BDR の両方に問題があります。

シンプルスキーマは、次のことを説明する例を提供します。

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

主キー列を更新できるため、次の SQL は成功します。

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

ただし、テーブルにマルチプルの行があるとしています。

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

一部の UPDATE は成功します。

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

```
SELECT * FROM pktest;
```

```
pk | val
- ---+-----
  2 |   1
  4 |   3
(2 rows)
```

他の UPDATE は制約エラーで失敗します。

```
UPDATE pktest SET pk=4 WHERE pk=2;
```

```
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
```

```
DETAIL:  Key (pk)=(4) already exists
```

したがって、プライマリキーを更新する Postgres アプリケーションの場合、BDR がなくてもランタイムエラーを回避するように注意してください。

BDR では、同時にマルチプルの場所からの 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 2 つの変更を同時に実行すると機能します。

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

```
node2: UPDATE pktest SET pk=pk+1 WHERE pk = 4;
```

```
SELECT * FROM pktest;
```

```
pk | val
- ---+-----
  3 |   1
  5 |   3
(2 rows)
```

これらの次の 2 つの変更を同時に実行すると、両方の変更が受け入れられるため、発散エラーが発生します。ただし、他のノードに変更を適用すると、update_missing 競合が発生します。

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

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

このシナリオでは、各ノードでデータが異なります。

```
node1:
```

```
SELECT * FROM pktest;
```

```
pk | val
- ---+-----
  1 |   1
  5 |   3
```

(次のページに続く)

(前のページからの続き)

(2 rows)

node2:

SELECT * FROM pktest;

pk | val

- ---+-----

2 | 1

5 | 3

(2 rows)

:ref: `LiveCompare` によるデータ検証<LiveCompareによるデータ検証>` を使用して、このシチュエーションを特定して解決できます。

競合が同時に発生すると問題が発生します。これら 2 つの変更を同時に実行するのは簡単ではありません。

```
node1: UPDATE pktest SET pk=6, val=8 WHERE pk = 5;
```

```
node2: UPDATE pktest SET pk=6, val=9 WHERE pk = 5;
```

両方の変更がローカルに適用されるため、ノード間で相違が発生します。ただし、ターゲットでの適用は、重複キー値違反エラー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、マニュアル解決が必要になります。

この重複キー違反エラーを回避できるようになり、`conflict_type update_pkey_exists` を `skip`、`update`、または `update_if_newer` に設定してもレプリケーションは中断されません。これは、更新の性質によっては依然として発散につながる可能性があります。

`update_pkey_exists` を `update_if_newer` に設定することで、同じ古いキーが同じ新しいキーによって同時に更新されている場合の発散を回避できます。ただし、特定の状況では、`update_if_newer` でも発散が発生します。つまり、2 つの異なる行が両方とも同じ新しいプライマリキーに更新されます。

その結果、アプリケーション、特に BDR でプライマリキーの UPDATE 操作を許可しないことを強くお勧めします。アプリケーションの一部がプライマリキーを変更する場合、同時変更を回避するには、Eager Replication を使用し make ます。

警告: `update_pkey_exists conflict` の競合解決が更新の場合、常に行の 1 つが削除されます。

マルチプルの UNIQUE コンストレインに違反する UPDATE 操作

マルチプルの *UNIQUE* コンストレインに違反する *INSERT* 操作と同様に、着信 UPDATE が複数の *UNIQUE*

インデックス（または PRIMARY KEY）に違反すると、BDR は `multiple_unique_conflicts` 競合が発生させます。

BDR は、遅延の一意の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、競合が見つからない限り、ターゲットに正常に適用されます。ただし、遅延プライマリキーは

REPLICA IDENTITY として使用できないため、ユースケースはそれとマルチプルの一意のコンストレインの使用に関するワーニングによって既に制限されています。

UPDATE/DELETE の競合

あるノードが行を更新し、別のノードが同時に削除する可能性があります。この場合、再生時に UPDATE / リプレイの競合が発生する可能性があります。

削除され行がまだ検出可能な場合（削除された行は VACUUM によって削除されませんでした）、update_recently_deleted の競合が生成されます。デフォルトでは UPDATE はスキップされたされますが、この解像度を構成できます。詳細は [列レベルの競合解決の有効化と無効化](#) を参照してください。

ローカルノードのレプリケーションが遅れている場合、削除され行は UPDATE を受け取るまでにデータベースからクリーンアップできます。この場合、BDR は UPDATE /DELETE の競合と [INSERT / UPDATE の競合](#) を区別できず、update_missing の競合を生成します。

競合する DELETE と UPDATE の別のタイプは、行がローカルで更新された後の DELETE です。このシチュエーション、結果は使用される競合検出の種類によって異なります。デフォルトの

[オリジンの競合検出](#) を使用すると、競合はまったく検出されないため、DELETE

が適用され、行が削除されます。[行バージョンの競合検出](#) を有効にすると、delete_recently_updated の競合が生成されます。この競合タイプのデフォルトの解決策は、DELETE を適用して行を削除することですが、これを構成するか、競合トリガーで処理できます。

INSERT / UPDATE の競合

デフォルトの非同期オペレーションモードを使用する場合、ノードはオリジナルの INSERT を受け取る前に行の UPDATE を受け取る場合があります。これは、3 つ以上のノードがアクティブな場合にのみ発生します（

[3 つ以上のノードとの競合](#) を参照）。

これが発生すると、update_missing 競合が生成されます。デフォルトの競合リゾルバは insert_or_skip ですが、代わりに insert_or_error または skip を使用できます。挿入またはアクションを行うリゾルバーは、可能な場合（行全体を受け取ったとき）、最初に UPDATE からのデータに基づいて新しい行を INSERT しようとします。行の再構築を可能にするには、テーブルにニーズが含まれているか、行にトーストされたデータが含まれていない必要があります。

トーストされたデータの詳細については、[TOAST サポート詳細](#)を参照してください。

INSERT/DELETE の競合

INSERT /UPDATE の競合と同様に、ノードは、まだ INSERT を受け取っていない行で DELETE オペレーションを受け取る場合があります。これも 3 つ以上のノードが設定されている場合にのみ可能です（

[3 つ以上のノードとの競合](#) を参照）。

BDR は現在、この競合タイプを検出できません。INSERT オペレーションは競合タイプを生成せず、INSERT が適用されます。

DELETE オペレーションは常に `delete_missing` の競合を生成します。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。

DELETE/DELETE 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。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 操作は同じ効果があるため、この競合は無害です。そのうちの 1 つは無視しても問題ありません。

3 つ以上のノードとの競合

1 つのノードが 2 番目のノードに再生されてそこで更新される行を挿入する場合、3 番目のノードは最初のノードから INSERT を受信する前に 2 番目のノードから UPDATE を受信できます。このシナリオは INSERT /UPDATE の競合です。

これらの競合は、UPDATE . これにより、異なるノードに異なるデータが表示される可能性があります。これらは

[発散する競合](#) です。

この競合タイプは、3 つ以上のマスターでのみ発生する可能性があります、そのうち少なくとも 2 つがアクティブに書き込んでいる必要があります。

また、ノード 1 からノード 3 へのレプリケーションラグは、次の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 1 から INSERT を受け取ります
2. ノード 2 が UPDATE を実行する
3. ノード 3 はノード 2 から UPDATE を受信します
4. ノード 3 はノード 1 から INSERT を受け取ります

`insert_or_error`（または場合によっては `update_missing` 競合タイプの `insert_or_skip` 競合リゾルバ）を使用することが、これらの競合の実行可能な軽減戦略です。ただし、このオプションを有効にすると、INSERT /DELETE の競合の扉が開きます。

1. ノード 1 が UPDATE を実行する
2. ノード 2 は DELETE を実行します
3. ノード 3 はノード 2 から DELETE を受信します
4. ノード 3 はノード 1 から UPDATE を受け取り、それを INSERT

これらが問題になる場合は、テーブルまたはデータベースのフリーズ設定をチューニングして、`update_recently_deleted` として正しく検出されるようにすることをお勧めします。

別の方法は、*Eager Replication* を使用してこれらの競合を防ぐことです。

INSERT /DELETE の競合は、3 つ以上のノードでも発生する可能性があります。このような競合は、UPDATE が DELETE に置き換えられていることを除き、INSERT /UPDATE と同じです。これにより、`delete_missing` の競合が発生する可能性があります。

BDR は、`update_missing` の競合で発生する make に、各 INSERT を最近削除したかどうかを確認することを選択できます。ただし、これを行うと大部分のユーザーがペナルティを受けるため、現時点では単にコストをログに記録します。

後のリリースでは、`delete_missing` の競合が発生した場合、*LiveCompare* によるデータ検証 を使用した再チェックを介して INSERT /DELETE

の異常を自動的に解決します。これらは、`bdr.conflict_history_summary` ビューを確認することにより、アプリケーションで手動で実行できます。

これらの競合は、2 つの主要な問題のユースケースで発生する可能性があります。

- キューイングアプリケーションで使用できるように、INSERT の後にすぐに DELETE が続く
- テーブルのプライマリー識別子を再利用する場合

これらのケースはいずれも一般的ではありません。これらの問題のユースケースが発生した場合は、影響を受けるテーブルを複製しないことをお勧めします。

BDR は識別子の一意性に依存してレプリケーションを機能させるため、後者の場合に問題が BDR ます。

同じ一意の識別子を挿入、削除、および後で再利用するアプリケーションは、問題を引き起こす可能性があります。これは *ABA problem* と呼ばれます。

BDR には、行が現在の行、最後の行、またはそれよりも古い行であるかを知る方法がありません。

一意の識別子の再利用もビジネスの問題です。アプリケーションは一意の識別子を再利用する必要はありません。

変更がシステムを通過するまでにかかる時間内に識別子が再利用されると、問題が発生しインターバル。通常のオペレーションではその時間は短い場合がありますが、ノードがダウンすると、そのインターバルが数時間または数日に延長される場合があります。

アプリケーションで一意の識別子を再利用しないことをお勧めしますが、使用する場合は、1 年より小さいに再利用しないようにするための措置を講じます。

この問題は、シーケンスまたは UUID を使用するアプリケーションでは発生しません。

外部キー制約の競合

適用されるリモートトランザクションと既存のローカルデータ間の競合は、FOREIGN KEY (FK) コンストレインでも発生する可能性があります。

BDR は `session_replication_role = 'replica'` で変更を適用するため、変更を適用するときに外部キーが再チェックされません。アクティブ/アクティブ環境では、参照元のテーブルへの挿入と同時に被参照のテーブルが削除されると、FK 違反が発生する可能性があります。これは、INSERT /DELETE の競合に似ています。

シングルマスターの Postgres では、被参照する INSERT /UPDATE は、行レベルのロックを取得する前に DELETE 操作が終了する必要があります。DELETE が被参照した値を削除する場合、INSERT /UPDATE は FK チェックに失敗します。

マルチマスター BDR。ノード間の行レベルロックはありません。参照元のテーブルの INSERT は、参照のテーブルの DELETE の背後で待機しないため、両方のアクションを同時に実行できます。したがって、参照元のテーブルのあるノードの INSERT /UPDATE は、別のノードの被参照のテーブルの DELETE と同時に値を使用できます。これにより、参照元テーブルに値がなくなり被参照。

実際には、これは、参照するテーブルでの DELETE 操作とは別のトランザクションで、被参照されるテーブルで DELETE 操作が発生する場合に発生します。これは一般的なオペレーションではありません。

Orders -> OrderItems などの親子リレーションでは、これを行うことは一般的ではありません。OrderItem を完全に削除するよりもキャンセル済みとしてマークする可能性が高くなります。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ことはまれです。

FK がダンダリングされる可能性はありますが、一般的にこのリスクは非常に低いため、BDR はこの場合をカバーする汎用ソリューションを課しません。これが発生するシチュエーションを理解したら、2つのソリューションが考えられます。

最初の解決策は、FK の使用を、一般に一度に1つのノードからのみ変更される、頻繁に変更される、または変更の同時実行がアプリケーションによって行われる密接に関連するエンティティに制限することです。これにより、アプリケーションレベルでの FK 違反が回避されます。

2 番目の解決策は、BDR が提供する関数 `bdr.ri_fkey_trigger()` および `bdr.ri_fkey_on_del_trigger()` を使用して、この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 に設定することにより、FK の異常を回避します。これにより、1つのトリガー内のすべての FK が再チェックされるため、FK 違反を防ぐためにテーブルごとに1つのトリガーのみを追加する必要があります。

例として、Fact と RefData の2つのテーブルがあるとします。ファクトには、RefData を参照する FK があります。Fact は参照元のテーブルであり、RefData は被参照のテーブルです。各テーブルに1つのトリガーを追加する必要があります。

RefData で被参照された行が既に削除されている場合、Fact に列を NULL に設定するトリガーを追加します。

```
CREATE TRIGGER bdr_replica_fk_iu_trg
  BEFORE INSERT OR UPDATE ON fact
  FOR EACH ROW
```

(次のページに続く)


```
EXECUTE PROCEDURE bdr.ri_fkey_trigger();

ALTER TABLE fact
    ENABLE REPLICA TRIGGER bdr_replica_fk_iu_trg;
```

RefData テーブルで DELETE が発生したときに Fact に列を NULL に設定するトリガーを追加します。

```
CREATE TRIGGER bdr_replica_fk_d_trg
    BEFORE DELETE ON refdata
    FOR EACH ROW
    EXECUTE PROCEDURE bdr.ri_fkey_on_del_trigger();

ALTER TABLE refdata
    ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

両方のトリガーを追加すると、外部キーのダングリングが回避されます。

TRUNCATE の競合

TRUNCATE は、すべての行の DELETE と同様に動作しますが、行ごとの削除ではなく、テーブルデータを物理的に削除することによってこのアクションを実行します。その結果、行レベルの競合ハンドリングを利用できないため、明確な競合がある場合でも、TRUNCATE コマンドは他の DML アクションとの競合を生成しません。

その結果、TRUNCATE に対して他のノードで別の DML が同時に実行された場合、リプレイの順序付けによって異なる変更が発生する可能性があります。

次のいずれかのアクションを実行できます。

- TRUNCATE が他の同時 DML と一緒に実行されないことを確認します。 [LiveCompare](#) によるデータ検証 を利用して、このような矛盾を強調してください。
- TRUNCATE を、WHERE 句のない DELETE ステートメントに置き換えます。このアプローチは、大きなテーブルではパフォーマンスが非常に低下する可能性があります。
- `bdr.truncate_locking = 'on'` を設定して、TRUNCATE コマンドのロック動作を設定します。この設定は、TRUNCATE が `bdr.ddl_locking` 設定に従うかどうかを決定します。すべてのノードが起動している必要があるため、これは TRUNCATE のデフォルトの動作ではありません。この構成は、すべての場合に可能であるとは限りません。

除外制約の競合

BDR は除外コンストレインをサポートしていないため、作成できません。

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します。

分散非同期システムでは、異なるノード上のすべてのトランザクションが完全に分離されるため、制約に違反する行のセットが存在しないことを保証できません。除外コンストレインは、排他制約違反のためにリプレイを任意のノードから他のノードに進めることができないリプレイデッドロックにつながります。

BDR に排他制約を作成させる場合、またはスタンドアロンデータベースを BDR に変換するときに既存の制約を削除しない場合、レプリケーションが中断されることを想定してください。再び進行させるには、着信リモートタプルが競合するローカルタプルを削除または変更して、リモートトランザクションを適用できるようにします。

ロールとテーブルスペースの違いによるデータの競合

競合は、ノードがロールなどの異なるグローバル（Postgres システム全体の）データを持つ場合にも発生します。これにより、主に DDL の操作が正常に実行されて 1 つのノードでコミットされますが、他のノードに適用できません。

例、node1 には fred 名前付けのユーザがあり、そのユーザは node2 で作成されなかったとします。node1 の fred がテーブルを作成する場合、テーブルは所有者が fred に設定されて複製されます。DDL コマンドを node2 に適用すると、fred 名前付けのユーザがないため、DDL は失敗します。この失敗により、Postgres ログにエラーが出力されます。

BDR が実行されているデータベースにユーザ fred を作成して、この競合を解決するには、管理者の介入が必要です。bdr.role_replication = on を設定して、将来これを解決できます。

ロックの競合とデッドロックの中止

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

関連するロックには次のものが含まれます。

- ユーザセッションによる明示的なテーブルレベルロック（LOCK TABLE ...）
- ユーザセッションによる明示的な行レベルロック（SELECT ... FOR UPDATE/FOR SHARE）
- ローカルアクティビティまたは他のノードからのレプリケーションからの行 UPDATE、INSERT、または DELETE 操作による暗黙的なロック

BDR ライタプロセスは、ユーザトランザクションでデッドロックする可能性があります。その場合、ユーザトランザクションはライタプロセスが保持するロックを待機しています。2 つのライタプロセスが互いにデッドロックする場合もあります。Postgres のデッドロック検出器が介入して、問題のあるトランザクションの 1 つを終了します。BDR ライタプロセスが終了した場合、再試行しますが、通常は成功します。

これらの問題はすべて一時的なものであり、通常は管理者のアクションは必要ありません。ライタプロセスがアイドルユーザセッションのロックで長時間スタックしている場合、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実行できますが、これは、別のユーザセッションに影響を与える長いロックを保持しているユーザと違いはありません。

Postgres の `log_lock_waits` 機能を使用すると、ロック関連のリプレーストールを特定できヘルプ。

発散する競合

異なるノードで同じであるはずのデータが予期せず異なる場合、発散競合が発生します。発散する競合は発生しないはずですが、執筆時点でこのような競合のすべてを確実に防止できるわけではありません。

行の **PRIMARY KEY** を変更すると、すべてのノードが変更を再生する前に別のノードが同じ行のキーを変更した場合、発散の競合が発生する可能性があります。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ます。

行データが関係する発散コンフリクトでは、通常、一方のノードのデータを他のノードと一貫性を保つように手動で調整する管理者のアクションが必要です。ドキュメントのとおり **BDR** を使用し、安全でないマークされた設定または機能を回避する限り、このような競合は発生しません。

管理者は、このような競合を手動で解決する必要があります。競合の性質によっては、`bdr.ddl_replication` や `bdr.ddl_locking` などの高度なオプションの使用が必要になる場合があります。ただし、これらのオプションを不注意に使用 `make` と、状況がさらに悪化し、一般的な指示では対処できない競合が発生する可能性があります。

TOAST サポート詳細

Postgres は、**TOAST サポート詳細**と呼ばれる大きな列にアウトオブラインストレージを使用します。

ロジカルデコーディング (BDR がその上に構築されています) およびロジカルレプリケーションでハンドリングする **TOAST** 値は、テーブルのメイン行のパートとして保存されるインラインデータとは異なります。

TOAST 値は、値が変更された場合にのみトランザクションログ (WAL) にログオン。これは、特に **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。トーストされた列の値を変更しなかった **UPDATE** ステートメントはその列のない行を生成するためです。

INSERT / UPDATE の競合 で述べたように、`update_missing` の競合が

`insert_or_error` を使用して解決され、**TOAST** 列が欠落している場合、**BDR** はエラーを報告します。

ただし、非同期レプリケーションを使用した同時ワークロードの場合、これよりも微妙な問題があります (Eager トランザクションは影響を受けません)。例、A、B、および C という 3 つのノードを持つ **BDR** クラスタでの次のワークロードを想像してください。

1. ノード A で: `txn A1` は `UPDATE SET col1 = ' トースト data...'` を実行し、最初にコミットします。
2. ノード B で: `txn B1` は `UPDATE SET other_column = 'anything else' ; A1 の後にコミットします。`
3. ノード C: ノード A への接続が遅れています。
4. ノード C: `txn B1` が最初に適用され、`col1` の **TOAST** された列を見逃しますが、競合せずに適用されます。

5. ノード C：txn A1 が (update_origin_change で) 競合するため、スキップされたされます。
6. ノード C は A1 からのトーストされたデータを永遠に逃します。

BDR を使用する場合、このシナリオは通常問題になりません。(ビルトイン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ロギングを使用する場合です。) BDR は、最近 TOAST 列の変更を TOAST した行へのローカル UPDATE を検出し、ローカル UPDATE ' t TOAST を変更する。したがって、BDR は、異なるノード間でトーストされたデータの不一致を防ぎます。このシチュエーションにより、マルチプルのノードで更新が発生した場合 (つまり、タプルのオリジンが変更された場合)、WAL ロギングが増加します。BDR AlwaysOn アーキテクチャの場合、通常のように、すべての更新が単一のノードから行われた場合、追加の WAL オーバーヘッドはゼロです。

注釈: メインテーブルで同じことをせずに TOAST テーブルだけで `VACUUM FULL` または `CLUSTER` を実行すると、追加のロギングが機能するために必要なメタデータが削除されます。これは、そのようなステートメントの後の短期間、これらの同時実行の問題に対する保護が存在しないことを意味します。

警告: TOAST の追加の WAL ロギングは、標準の Postgres で *BEFORE UPDATE* トリガーを使用して実行されます。このトリガーは、テーブル上のすべての `BEFORE UPDATE` トリガーの中で (トリガー名前に基づいて) アルファベット順にソートする必要があります。これを簡単に `make` に `zzzz_bdr_` プレフィックスが付いていますが、その後にソートされる名前で作成しないでください。そうしないと、同時実行の問題に対する保護が得られません。

ただし、`insert_or_error` の競合解決には、`REPLICA IDENTITY FULL` の使用が必要です。

トーストされた列に関連するこれらの問題は、`REPLICA IDENTITY FULL` を持つテーブルには影響しません。この設定では、行全体がキーのパートと見なされるため、トーストされた値が常にキーのパートとしてログに記録されキー。BDR は、キー行から欠落しているデータを埋めて、新しい行を再構築できます。その結果、`REPLICA IDENTITY FULL` を使用すると、WAL サイズが大幅に増加する可能性があります。

2.8.3 競合の回避または許容

ほとんどの場合、競合を回避または許容するようにアプリケーションをデザインできます。

競合は、マルチプルのノードで同時に発生している場合にのみ発生します。競合を回避する最も簡単な方法は、1つのノードにのみ書き込むか、一度に1つの特定の行から特定の方法でのみ書き込むノードです。

これは、多くのアプリケーションで自然に発生します。例、多くのコンシューマアプリケーションでは、アカウントのデフォルトの請求先住所の変更など、所有ユーザのみがデータを変更できます。このようなデータ変更で更新の競合が発生することはほとんどありません。

ノードがダウンする直前に変更を `make` 可能性があるため、変更が失われたようです。次に、同じ変更を再度 `make` と、異なるノードで2つの更新が発生する可能性があります。ダウンしたノードが復旧すると、古い変更を他のノードに送信しようとしませんが、データの最後の更新が保持されるため拒否されます。

INSERT /INSERT の競合の場合、 [BDR グローバルシーケンス](#) を使用してこのタイプの競合を防ぎます。

部屋予約アプリケーションなど、オブジェクト間の関係を割り当てるアプリケーションの場合、`update_if_newer` を適用しても許容できるビジネス結果が得られない場合があります。つまり、2 人に同じ部屋を予約したことを個別に確認することは役に立ちません。最も簡単な解決策は、Eager Replication を使用して、1 つの予約のみが成功するように保証ことです。アプリケーションによっては、より複雑な方法も可能です。例、各ノードに 100 のシートを割り当て、そのノードのライタがそれらを予約できるようにします。ただし、ローカルで利用できるものがない場合は、ほとんどのシートが予約たら、分散ロックスキームまたは Eager Replication を使用します。

特定の種類の更新が 1 つの特定のノードからのみ行われるようにする別の手法は、さまざまな種類のトランザクションをさまざまなノードにルーティングすることです。例：

- 1 つのノードで小包を受け取りますが、別のノードを使用して小包を配達します
- 1 つのノードで注文が入力され、2 番目のノードで顧客に提供されるサービスアプリケーション

多くの場合、最善の方法は、競合の発生を許可し、BDR の競合解決メカニズムと連携して競合に対処するようにアプリケーションをデザインすることです。

2.8.4 競合検出

BDR は、競合検出に次のメカニズムを提供します。

- **オリジンの競合検出**（デフォルト）
- **行バージョンの競合検出**
- *Column-level conflict detection*

オリジンの競合検出

オリジンの競合検出は、トランザクションの発生元のノードに記録されたコミットタイムスタンプを使用および依存します。これには、クロックが正しく動作するか、2 つのノード間の最速メッセージの許容範囲内にある必要があります。そうでない場合、競合の解決はさらに先のノードを優先する傾向があります。パラメーター `bdr.maximum_clock_skew` および `bdr.maximum_clock_skew_action` を使用して、ノード間のクロックスキューを管理できます。

行の起点は、`track_commit_timestamp = on` の場合にのみ使用できます。

競合は、レプリケーションオリジンが変更されたかどうかに基づいて最初に検出されるため、競合ではないことが判明した状況で競合トリガーが呼び出されます。したがって、このメカニズムは誤検出の競合を生成する可能性があるため、正確ではありません。

起点情報は、行がフリーズするまでのみ利用できます。フリーズされた後に行に到着した更新は競合を発生させないため、すべての場合に適用されます。これは、`bdr_init_physical` で新しいノードを追加する場合の通常の場合であるため、その場合に競合が発生すると、多くの誤検出の結果が発生します。

オフラインだったノードが再接続してデータ変更の送信を開始すると、新しく到着した更新が更新するフリーズされた行よりも古い場合、これにより分岐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。

ノードの再起動とダウンしたノードの回復

で説明されているように、拡張停止のためにノードをダウンしたままにしないことをお勧めします。

EDB Postgres Extended Server および EDB Postgres Advanced Server では、ノードがダウンしている間の行のフリーズを BDR します。このメカニズムはこのシチュエーションを適切に処理するため、パラメータ設定を変更する必要はありません。

Postgres の他のバリエーションでは、このシチュエーションを注意して管理する必要がある場合があります。

フリーズは、通常、バキュームされる行が現在の xid の `vacuum_freeze_min_age` xid より古い場合に発生します。これは、これらのパラメーターに適切に高い値を構成する必要があることを意味します。

- `vacuum_freeze_min_age`
- `vacuum_freeze_table_age`
- `autovacuum_freeze_max_age`

トランザクションレートに基づいて値を選択し、データベースノードから競合データを削除するまでのダウンタイムを許可します。例、`vacuum_freeze_min_age` が 5 億に設定されている場合、1000 TPS を実行するノードは、競合データが削除されるまでに 5.5 日以上ダウンする可能性があります。CommitTS データ構造は、その設定で 5 ギガバイトのディスク上スペースを使用するため、トランザクションレートが低いシステムでは、設定を低くすることでメリットを得られます。

最初に推奨される設定は次のとおりです。

```
# 1 billion = 10GB
autovacuum_freeze_max_age = 1000000000

vacuum_freeze_min_age = 500000000

# 90% of autovacuum_freeze_max_age
vacuum_freeze_table_age = 900000000
```

次のことに注意してください。

- `autovacuum_freeze_max_age` はノードのスタートにのみ設定できます。
- `vacuum_freeze_min_age` を設定できるため、低い値を使用すると行が早期にフリーズされ、競合が無視される場合があります。個々のテーブルに `autovacuum_freeze_min_age` と `toast.autovacuum_freeze_min_age` を設定することもできます。
- `CLUSTER` または `VACUUM FREEZE` コマンドを実行すると、行が早期にフリーズされるため、競合が無視される場合があります。

行バージョンの競合検出

または、BDR は、行のバージョン管理を使用し、ノードのシステムクロックとは無関係に競合検出を make オプションを提供します。

行バージョンの競合検出では、3 つのことを有効にする必要があります。これらの手順のいずれかが正しく実行されない場合 **オリジンの競合検出** が使用されます。

1. BDR ノードグループで `check_full_tuple` を有効にする必要があります。
2. 行バージョンの競合検出を使用するすべてのテーブルで `REPLICA IDENTITY FULL` を有効にする必要があります。
3. `bdr.alter_table_conflict_detection` を使用して、テーブルで行バージョンの追跡を有効にする必要があります。この関数は、列（指定した名前）と、新しい列値を管理する `UPDATE` トリガーを追加します。列は `INTEGER` タイプとして作成されます。

カウンタは `UPDATE` でのみインクリメントされますが、この手法により `UPDATE` と `DELETE` の両方で競合を検出できます。

このアプローチは Lamport タイムスタンプに似ており、競合検出の ABA 問題を完全に防ぎます。

注釈: 行レベルの競合解決は、行のバージョン管理を使用しても **列レベルの競合解決の有効化と無効化** 構成に基づいて処理されます。行バージョンが生成される方法は、競合を検出する場合にのみ役立ちます。行のバージョンについての信頼できる情報として信頼しないでください。

特定のテーブルに使用されている現在の競合解決戦略を判断するには、ビュー `bdr.tables` の列 `conflict_detection` を参照してください。

`bdr.alter_table_conflict_detection`

テーブルの所有者は、特定のテーブルの競合検出の動作を変更できます。

概要

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


パラメーター

- **relation** —新しい競合検出メソッドを設定するリレーションの名前。
- **method** —使用する競合検出メソッド。
- **column_name** —列検出データの格納に使用する列。これはスキップされたできます。その場合、列名前は競合検出メソッドに基づいて選択されます。**row_origin** メソッドは、メタデータストレージ用の追加の列を必要としません。

認識されている競合検出方法は次のとおりです。

- **row_origin** —テーブルに対して行われた以前の変更の起点（[オリジンの競合検出](#)を参照）。これは、テーブルに追加の列を必要とせずにサポートされている唯一のメソッドです。
- **row_version** —行バージョン列（[行バージョンの競合検出](#)を参照）。
- **column_commit_timestamp** —列ごとのコミットタイムスタンプ（CLCD で説明）。
- **column_modify_timestamp** —列ごとの変更タイムスタンプ（CLCD で説明）。

注意事項

column_commit_timestamp と **column_modify_timestamp** の競合検出方法の違いの詳細については、[現在とコミットのタイムスタンプ](#)を参照してください。

このファンクションは、DDL ステートメントと同じ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ョンで DML グローバルロックを取得します。

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

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

警告: 競合検出メソッドを追加の列を使用してメタデータを保存する方法から変更すると、その列が削除されます。

警告: このファンクションは CAMO を無効にします（ワーニングとともに、これらが ``bdr.camo_enable_client_warnings`` で無効にされていない限り）。

競合タイプのリスト

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

- `insert_exists` —着信挿入が、プライマリキーまたは一意のキー/インデックスを介して既存の行と競合します。
- `update_differing` —着信更新のキー行がローカル行と異なります。これは、[行バージョンの競合検出](#)を使用している場合にのみ発生します。
- `update_origin_change` —着信更新は、別のノードによって最後に変更された行を変更しています。
- `update_missing` —着信更新が存在しない行を変更しようとしています。
- `update_recently_deleted` —受信した更新が、最近削除された行を変更しようとしています。
- `update_pkey_exists` —着信更新により、`PRIMARY KEY` が変更を適用しているノードに既に存在する値に変更されました。
- `multiple_unique_conflicts` —着信行は、ターゲットテーブル内のマルチプルの `UNIQUE` コンストレイン/インデックスと競合しています。
- `delete_recently_updated` —現在のノードでの行の最新の更新よりも古いコミットタイムスタンプを持つ着信削除、または[行バージョンの競合検出](#)を使用している場合。
- `delete_missing` —着信削除が存在しない行を削除しようとしています。
- `target_column_missing` —ターゲットテーブルには、受信行にある 1 つ以上の列がありません。
- `source_column_missing` —受信行には、ターゲットテーブルに存在する 1 つ以上の列がありません。
- `target_table_missing` —ターゲットテーブルがありません。
- `apply_error_ddl` —レプリケートされた DDL コマンドを適用するときに Postgres によってエラーがスローされました。

2.8.5 競合の解決

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

BDR では、次の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、指定されたノードでの競合解決の動作を設定します。

概要

```
bdr.alter_node_set_conflict_resolver(node_name text,
                                     conflict_type text,
                                     conflict_resolver text)
```

パラメーター

- **node_name** —変更されるノードの名前。
- **conflict_type** —設定を適用する競合の種類（[競合タイプのリスト](#)を参照）。
- **conflict_resolver** —指定された競合タイプに使用するリゾルバー（[競合リゾルバーのリスト](#)を参照）。

注意事項

現在、ローカルノードのみを変更できます。ファンクション呼び出しは複製されません。マルチプルノードの設定を変更する場合は、それぞれでファンクションを実行する必要があります。

このファンクションによる構成の変更は、`bdr.create_node_group` または `bdr.alter_node_group` で指定された競合解決のデフォルトの動作をオーバーライドします。

このファンクションはトランザクションです。変更をロールバックでき、現在のトランザクションで可視されます。

競合リゾルバーのリスト

BDR では、ハンドルできる競合タイプのカバレッジが異なるいくつかの競合リゾルバーを使用できます。

- **error** —エラーをスローし、レプリケーションを停止します。任意の競合タイプに使用できます。
- **skip** —リモート変更の処理をスキップし、次の変更でレプリケーションを続行します。 `insert_exists`、`update_differing`、`update_origin_change`、`update_missing`、`update_recently_deleted`、`update_pkey_exists`、`delete_recently_updated`、`delete_missing`、`target_table_missing`、`target_column_missing`、および `conflict` タイプに使用できます。
- **skip_if_recently_dropped** —ダウストリームで最近（1 日以内）に削除されたためにダウストリームに存在しないテーブルの場合、リモート変更をスキップします。それ以外の場合はエラーをスローします。 `target_table_missing` 競合タイプに使用できます。 `skip_if_recently_dropped` の競合リゾルバは、同じ名前のテーブルが削除された直後に再作成されると、問題が発生する可能性

があります。その場合、ノードの1つが、テーブルを再作成するためのDDLを認識する前に、再作成されたテーブルのDMLを認識する可能性があります。次に、テーブルが最近削除されたと仮定して、リモートデータを誤ってスキップし、データロスします。したがって、この競合リゾルバとともに削除された直後にオブジェクト `namesq` を再利用しないことをお勧めします。

- `skip_transaction` —競合を生成したトランザクション全体をスキップします。 `apply_error_ddl` の競合に使用できます。
- `update_if_newer` —リモート行が競合するローカル行よりも後にコミットされた場合（発信元ノードの壁時計によって決定）を更新します。タイムスタンプが同じ場合、ノードIDがタイブレーカーとして使用され、保証のノードで同じ行が選択されます（ノードIDが高い方が勝ちます）。 `insert_exists` 、 `update_differing` 、 `update_origin_change` 、および `update_pkey_exists` の競合タイプに使用できます。
- `update` —常に複製されたアクションを実行します。 `insert_exists` （INSERT を UPDATE に変換）、 `update_differing` 、 `update_origin_change` 、 `update_pkey_exists` 、および `delete_recently_updated` （削除を実行）に使用できます。
- `insert_or_skip` —オリジンから送信された利用可能な情報から新しい行を作成し、挿入してみてください。完全な行を作成するのに十分な情報がない場合は、変更をスキップします。 `update_missing` および `update_recently_deleted` の競合タイプに使用できます。
- `insert_or_error` —オリジンから送信された利用可能な情報から新しい行を作成して挿入しようとします。完全な行を構築するのに十分な情報がない場合は、エラーをスローしてレプリケーションを停止します。 `update_missing` および `update_recently_deleted` の競合タイプに使用できます。
- `ignore` —欠落しているターゲット列を無視して、処理を続行します。 `target_column_missing` 競合タイプに使用できます。
- `ignore_if_null` —リモート行の余分な列に NULL 値が含まれる場合、欠落しているターゲット列を無視します。それ以外の場合は、エラーをスローしてレプリケーションを停止します。 `target_column_missing` 競合タイプに使用できます。
- `use_default_value` —欠落している列値をデフォルト（それが列デフォルトの場合は NULL を含む）で埋めて、処理を続行します。デフォルトの処理中のエラーまたはコンストレインの違反（つまり、NOT NULL 列の NULL デフォルト）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` , and `delete_missing` の競合タイプは、

競合のトリガー

を使用してユーザー定義のロジックで解決することもできます。

このマトリクスは、競合リゾルバーがハンドルできる競合の種類を区別するのに役立ちます。

	in- sert_exists	up- dates_differing	up- dates_differing	up- dates_differing	up- dates_differing	up- dates_differing	deleted_key_exists	deleted_key_exists	deleted_key_exists	deleted_key_exists	deleted_key_exists	deleted_key_exists	deleted_key_exists
error	X	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	X
skip_if_recently_dropped												X	
update_if_newer	X	X	X			X							
update	X	X	X			X	X						X
insert_or_skip				X	X								
insert_or_error				X	X								
ignore									X				
ignore_if_null									X				
use_default_value										X			
conflict_trigger	X	X	X	X	X	X	X	X					X

デフォルトの競合リゾルバー

Conflict type	Resolver
insert_exists	update_if_newer
update_differing	update_if_newer
update_origin_change	update_if_newer
update_missing	insert_or_skip
update_recently_deleted	skip
update_pkey_exists	update_if_newer
multiple_unique_conflicts	error
delete_recently_updated	skip
delete_missing	skip
target_column_missing	ignore_if_null
source_column_missing	use_default_value
target_table_missing	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実行された特定のアクションに対応します。

`conflict_resolution` パラメータでは、現在次の競合解決がサポートされています。

- `apply_remote` — リモート（着信）行が適用されました。
- `skip` — 一行の処理がスキップされたされました（ローカルで変更はありません）。
- `merge` — リモート行とローカル行からの情報をマージして、新しい行が作成されました。

- `user` —ユーザーコード（競合トリガー）は、ターゲット表に書き込まれた行を生成しました。

2.8.6 競合ロギング

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 `bdr.conflict_history` テーブルに記録します。次の関数を使用して、この動作をより詳細に変更できます。

`bdr.alter_node_set_log_config`

ノードの競合ロギング構成を設定します。

概要

```
bdr.alter_node_set_log_config(node_name text,  
                              log_to_file bool DEFAULT true,  
                              log_to_table bool DEFAULT true,  
                              conflict_type text[] DEFAULT NULL,  
                              conflict_resolution text[] DEFAULT NULL)
```

パラメーター

- `node_name` —変更されるノードの名前。
- `log_to_file` —ノードログログファイルに記録するかどうか。
- `log_to_table` — `bdr.conflict_history` テーブルにログを記録するかどうか。
- `conflict_type` —ログに記録する競合の種類。NULL（デフォルト）はすべてを意味します。
- `conflict_resolution` —ログに記録する競合の解決。NULL（デフォルト）はすべてを意味します。

注意事項

ローカルノードのみ変更できます。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、それぞれでファンクションを実行する必要があります。

このファンクションはトランザクションです。変更をロールバックでき、現在のトランザクションで可視されます。

競合ロギング構成のリスト

ビュー `bdr.node_log_config` は、すべてのロギング構成を表示します。ロギング構成の名前、ログに記録する場所、および競合の種類と解決がリストされます。

テーブルへの競合のログ

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

この表は、列 `local_time` でレンジパーティション化されています。テーブルは Autopartition によって管理されています。デフォルトでは、毎日新しいパーティションが作成され、過去 1 か月の競合が維持されます。その後、古いパーティションは自動的に削除されます。自動パーティションは、7~14 個のパーティションを事前に作成します。`bdr_superuser` はこれらのデフォルトを変更できます。

BDR によって管理されるすべてのテーブルで生成された競合はこのテーブルに記録されるため、正当なユーザーのみが競合するデータを読み取れるように保証ことが重要です。BDR は、`bdr.conflict_history` テーブルに ROW LEVEL SECURITY ポリシーを定義することによりこれを行います。テーブルの所有者のみが、それぞれのテーブルの競合を読み取ることができます。基になるテーブルで 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はできません。FORCE オプションで作成された RLS ポリシーは、テーブルの所有者にも適用されます。その場合、基になるテーブルの一部またはすべての行が所有者でさえも読み取れない場合があります。そのため、BDR は競合ログテーブルにより厳しいポリシーも適用します。

デフォルトのロール `bdr_read_all_conflicts` は、`bdr_superuser` ロールを付与せずに、`bdr.conflict_history` テーブルに記録されたすべての競合の詳細を表示する必要があるユーザーに付与できます。

デフォルトのロール `bdr_read_all_stats` は、`bdr.conflict_history_summary` というカタログビューにアクセスできます。これにはユーザデータが含まれないため、ログに記録された競合をモニタリングできます。

競合レポート

テーブルに記録された競合は、レポートに要約できます。レポートを使用すると、アプリケーションの所有者は競合を特定、理解、解決し、アプリケーションの変更を導入して競合を防ぐことができます。

```
SELECT nspname, relname
, date_trunc(day, local_time) :: date AS date
, count(*)
FROM bdr.conflict_history
WHERE local_time > date_trunc(day, current_timestamp)
GROUP BY 1,2,3
ORDER BY 1,2;
```

nspname	relname	date	count
---------	---------	------	-------

(次のページに続く)


```

-----+-----+-----+-----
my_app | test   | 2019-04-05 |      1
(1 row)

```

2.8.7 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を対象としています。デフォルトでは、すべてのレプリケートされたテーブルを比較し、違いを報告します。LiveCompare は、Postgres やオラクルなどの非 BDR データソースでも動作します。

LiveCompare を使用して、着信行を継続的にモニタすることもできます。コンテキスト情報を失うことなく停止およびスタートできるため、都合の良いときに実行できます。

LiveCompare を使用すると、マルチプルのテーブルを同時にチェックできます。いくつかのテーブルまたはテーブル内の行のセクションのみをチェックできるように構成できます。チェックは、最初に行ハッシュ全体を比較することによって実行されます。異なる場合、LiveCompare は行全体を比較します。LiveCompare は、有用なサイズのバッチで行を比較することにより、オーバーヘッドを回避します。

違いが見つかった場合、期間をかけて再確認できるため、最終的な一貫性が遅れます。

詳細については、[LiveCompare によるデータ検証](#) の文書を参照してください。

2.9 Sequences

多くのアプリケーションでは、一意のサロゲート ID をデータベースエントリに割り当てる必要があります。多くの場合、データベースの SEQUENCE オブジェクトはこれらを生成するために使用されます。PostgreSQL、これらはシーケンスコマンドを使用して手動で作成し、`nextval()` ファンクションを呼び出すことで取得するか、`serial` および `bigserial` 列、または `GENERATED BY DEFAULT AS IDENTITY` 列のいずれかです。

ただし、PostgreSQL の標準シーケンスはマルチノードを認識せず、ローカルノードで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（破棄された INSERTs による）を引き起こすため、これは重要です。

2.9.1 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 がノード間で挿入を同期する場合、競合することはありません。

BDR グローバルシーケンスは PostgreSQL シーケンスを拡張するため、クラッシュセーフです。これらを使用するには、`bdr_application` ロールが付与されている必要があります。

グローバルシーケンスにはさまざまなアルゴリズムがあります。

- `SnowflakeId` シーケンス
- グローバルに割り当てられたレンジシーケンス

`SnowflakeId` シーケンスは、任意の時点でノード間通信を必要としないアルゴリズムを使用して値を生成するため、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という便利なプロパティがあります。

`SnowflakeId` シーケンスには、64 ビット `BIGINT` データ型でのみ機能し、19 桁の長さの値を生成するという制約があります。これは、JavaScript Integer 型などの一部のホスト言語データ型で使用するには長すぎる場合があります。グローバルに割り当てられたシーケンスは、ノード間のコンセンサスによって必要に応じて補充できる値のローカルレンジを割り当て、`BIGINT` または `INTEGER` シーケンスに適しています。

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

BDR は、`CREATE SEQUENCE` コマンドが実行されたときに、または `serial`、`bigserial`、または `GENERATED BY DEFAULT AS IDENTITY` 列が作成されたときに、どのような種類のシーケンスが作成されるかを決定する構成変数 `bdr.default_sequence_kind` も提供します。有効な設定は次のとおりです。

- `local`（デフォルト）は、新しく作成されたシーケンスが標準の PostgreSQL（ローカル）シーケンスであることを意味します。
- 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`。
- 時間、ノード ID、およびカウンタコンポーネントで構成される `BIGINT` シーケンスのグローバルシーケンスを作成する `snowflakeid` は、`INTEGER` シーケンスでは使用できません（したがって、`bigserial` には使用できますが、`serial` には使用できません）。
- 下位互換性のためにのみ提供されている `SnowflakeId` シーケンスの `timeshard` 古いバージョン、`SnowflakeId` が推奨されます
- `distributed` `bdr.default_sequence_kind` にのみ使用でき、`int8` シーケンス（つまり `bigserial`）には `snowflakeid` を、`int4`（つまり `serial`）および `int2` シーケンスには `galloc` を選択する特別な値。

`bdr.sequences` ビューは、個々のシーケンスの種類に関する情報を表示します。

`currval()` と `lastval()` は、すべてのタイプのグローバルシーケンスで正しく機能します。

SnowflakeId シーケンス

SnowflakeId シーケンスによって生成された ID は大まかに時間順に並べられているため、標準の PostgreSQL シーケンスのように、データ挿入のおおよそのオーダーを取得するために使用できます。同じミリ秒内に生成された値は、1 つのノードでさえも乱れる場合があります。緩い時間順序付けのプロパティは、レンジパーティションキーとしての使用に適していることを意味します。

SnowflakeId シーケンスは 1 つ以上のノードで動作し、ノード結合プロセスが完了した後のノード間通信は必要ありません。そのため、ネットワークパーティションが拡張されるリスクがあり、レプリケーションラグやノード間のレイテンシの影響を受けない場合でも、引き続き使用される可能性があります。

SnowflakeId シーケンスは、標準のシーケンスとは異なる方法で一意の ID を生成します。このアルゴリズムでは、シーケンス番号に 3 つのコンポーネントを使用します。シーケンスの最初のコンポーネントは、シーケンス番号生成時のタイムスタンプです。シーケンス番号の 2 番目のコンポーネントは、各 BDR ノードに割り当てられた一意の ID です。これにより、異なるノードの ID は常に異なります。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 を追加保証だけで競合が発生しませんが、シーケンスの別の有用なプロパティ、つまり、シーケンス番号の順序付けがデータが挿入されたオーダーにほぼ一致することも維持します。タイムスタンプを最初に置くと、これが保証されます。

SnowflakeId シーケンスには、いくつかの制限と警告が適用されます。

SnowflakeId シーケンスは 64 ビット幅で、`bigint` または `bigserial` が必要です。生成される値は少なくとも 19 桁の長さになります。実用的な 32 ビット `integer` バージョンはないため、`serial` シーケンスでは使用できません - 代わりにグローバルに割り当てられたレンジシーケンスを使用します。

SnowflakeId の場合、任意のノードでミリ秒ごとに生成されるシーケンス値の制限があります（これは、1 秒あたり約 400 万のシーケンス値を意味します）。シーケンス値の生成が指定されたミリ秒内にラップする場合、SnowflakeId シーケンスは次のミリ秒まで待機し、そのミリ秒の新しい値を取得します。

SnowflakeId シーケンスはタイムスタンプをシーケンス値にエンコードため、新しいシーケンス値は指定されたタイムフレーム内にのみ生成できます（システムクロックによって異なります）。SnowflakeId のエポックタイムである 2016-10-07 を使用できる最も古いタイムスタンプ。値は 2086 年に負の値にラップし、2156 までに数が完全になくなります。

タイムスタンプは SnowflakeId シーケンスの重要なパートであるため、postgres プロセスの有効期間内に使用された最新のタイムスタンプよりも古いタイムスタンプでシーケンスを生成しないようにする追加の保護があります。

SnowflakeId シーケンスの入力として使用されるシーケンスの `INCREMENT` オプションは効果的に無視されます。これは、多くのオブジェクトリレーショナルマッパー（ORM）ツール、特に Hibernate のように、シーケンス ID キャッシングを行うアプリケーションに関連する可能性があります。シーケンスは時間ベースであるため、アプリケーションがキャッシュされた値を使用して何かを実行できるようになるまでに、シーケンスは新しい非衝突値に進むため、実際的な効果はほとんどありません。

同様に、`START`、`MINVALUE`、`MAXVALUE` および `CACHE` 設定は、基になるシーケンスで変更できますが、そうするメリットはありません。シーケンスの下位 14 ビットが使用され、残りは破棄されるため、値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 SnowflakeId シーケンスには役立ちません。

タイムシャードシーケンス

タイムシャードシーケンスは、既存のインストールとの下位互換性のために提供されていますが、新しいアプリケーションでの使用はお勧めしません。代わりに SnowflakeId シーケンスを使用することをお勧めします。

Timeshard は SnowflakeId に非常に似ていますが、制限が異なり、保護が少なく、パフォーマンスが低下します。

timeshard と SnowflakeId の違いは次のとおりです。

- Timeshard は SnowflakeId を超える最大 16384 ミリ秒（1 秒あたり約 1600 万）を生成できますが、指定されたミリ秒内の周回に対する保護がないため、timeshard シーケンスを使用するスキーマは指定された列に timeshard 値を使用するときに UNIQUE 制約の使用を保護する必要があります。-タイムハードシーケンスのタイムスタンプコンポーネントは 2050 年に値を使い果たし、bigint と組み合わせて使用すると、値は 2033 年に負の数にラップします。これは、2033 年以降に生成されたシーケンスは負の値を持つことを意味します。これは SnowflakeId よりも大幅に短い時間間隔であり、SnowflakeId が好ましい主な理由です。-Timeshard シーケンスは時折ディスクメモリを必要とします（標準のローカルシーケンスと同様）。

グローバルに割り当てられたレンジシーケンス

グローバルに割り当てられたレンジ（または galloc）シーケンスは、値の範囲（チャンク）を各ノードに割り当てます。ローカルレンジが使用されると、他のノード間のコンセンサスによって新しい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ときにシーケンスジェネレータを進行させるには、ローカルノードをクラスター内の大部分のノードに接続する必要があります。

SnowflakeId シーケンスとは異なり、galloc シーケンスは、PostgreSQL が提供するすべてのシーケンスデータ型- smallint、integer、および bigint をサポートします。これは、53 ビット値のみをサポートするため、javascript で整数を使用するなど、64 ビットシーケンスに問題がある環境で、またはシーケンスが限られたスペースで出力に表示される場合に使用できることを意味します。

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ます。

- smallint - 1 000 の数字
- 整数- 1 000 000 の数字
- bigint - 1 000 000 000 の数字

各ノードは seq_chunk_size の 2 つのチャンクを割り当てます。1 つは現在の使用にプラスで、1 つは将来の使用のために予約しているため、1 つのノードから生成される値は単調に増加します。ただし、グローバルに見ると、生成される値はまったく順序付けられません。これにより、B ツリーインデックスへの影響によりパフォーマンスが低下する可能性があり、通常は生成された値がレンジパーティションキーとして役に立たないことを意味します。

galloc シーケンスの主な欠点は、割り当てられたレンジを使い果たすと、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レンジについてコンセンサスを求める必要があることです。BDR グループの大部分にアクセスできません。これは、後のリリースでは回避される可能性があります。

CACHE、START、MINVALUE および MAXVALUE オプションは galloc シーケンスで正しく機能しますが、シーケンスを galloc 種類に変換する前に設定する必要があります。INCREMENT BY オプションも正常に動作しますが、シーケンスデータ型ごとに割り当てられた上記の範囲以上のインクリメント値を割り当てることはできません。setval() は galloc シーケンスのグローバル状態をリセットしないため、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 `:ref:`bdr.sequence_alloc`<bdr.sequence_alloc>` で galloc シーケンスを追跡します。このカタログは、galloc シーケンスに現在割り当てられているチャンクを追跡するために必要です。シーケンス名前と名前空間はこのカタログに保存されます。シーケンスチャンクのアロケーションは Raft を介して管理されるのに対し、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を介して管理されるため、BDR は現在、galloc シーケンスの名前変更、別の名前空間への移動、または galloc シーケンスを含む名前空間の名前変更をサポートしていません。ユーザは、アプリケーションスキーマを設計するときにこの制限に注意する必要があります。

ローカルシーケンスから galloc シーケンスへの変換

ローカルシーケンスを galloc に変換する前に、いくつかの前提条件を満たす必要があります。

1. シーケンスと列のデータタイプがマッチすることを確認します

シーケンスのデータ型が使用する列のデータタイプデータタイプと一致することを確認します。例、bigint シーケンスを作成し、そのシーケンスによって返される nextval() に integer 列のデフォルトを割り当てることができます。bigint に 1 000 000 000 のブロックで割り当てられる galloc シーケンスを使用すると、3 つ以上のノードが使用されている場合、nextval() によって返される値がすぐに int4 のレンジを超えます。

次の例は、何が起こるかを示しています。

```
CREATE SEQUENCE int8_seq;

SELECT sequencename, data_type FROM pg_sequences;
sequencename | data_type
-----+-----
int8_seq      | bigint
(1 row)

CREATE TABLE seqtest (id INT NOT NULL PRIMARY KEY);

ALTER SEQUENCE int8_seq OWNED BY seqtest.id;

SELECT bdr.alter_sequence_set_kind(public.int8_seq::regclass, galloc, 1);
alter_sequence_set_kind
-----
```

(次のページに続く)

(前のページからの続き)

(1 row)

```
ALTER TABLE seqtest ALTER COLUMN id SET DEFAULT nextval(int8_seq::regclass);
```

2つのノードで `INSERT INTO seqtest VALUES(DEFAULT)` を実行すると、テーブルには次の値が含まれます。

```
SELECT * FROM seqtest;
```

```
   id
-----
      2
20000000002
(2 rows)
```

ただし、3番目のノードで同じオペレーションをしようとする、シーケンスが値 `40000000002` を生成するため、`integer out of range` エラーで失敗します。次のセクションには、シーケンスのチャンクが割り当てられる方法の詳細が含まれています。

ちなみに: シーケンスの現在のデータタイプは、PostgreSQL の `pg_sequences` ビューから取得できます。シーケンスのデータタイプは、現在の値が新しいデータタイプの最大値を超えない限り、`ALTER SEQUENCE ... AS ...`、例: `ALTER SEQUENCE public.sequence AS integer` を変更できます。

2. シーケンスの新しいスタート値を設定する

シーケンス種類を `galloc` に変更すると、ローカルシーケンスの定義されたスタート値から書き換えられてリスタートします。これが稼働データベース内の既存のシーケンスで発生した場合、現在の値をクエリーしてから、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 ではユーザーがファンクション `bdr.alter_sequence_set_kind()` で開始値を渡すことができます。既にオフセットを使用していて、マルチプルのノードからの書き込みがある場合は、使用された最大の値を確認し、少なくとも次の値までシーケンスをリスタートする必要があります。

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

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

ユーザーはシーケンスをロックできないため、`max()` 値が照会されている間も操作を続行できるように、`$MARGIN` 値を残す必要があります。

bdr.sequence_alloc テーブルは、チャンクサイズとクラスター全体に割り当てられている範囲に関する情報を提供します。この例では、333, からシーケンスを開始し、クラスター内に2つのノードがあります。アロケーション数は4、つまりノードごとに2であり、チャンクサイズは1000000であり、整数シーケンス。

```
SELECT * FROM bdr.sequence_alloc
    WHERE seqid = public.categories_category_seq::regclass;
    seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |
seq_last_alloc
-----+-----+-----+-----+-----
categories_category_seq |      1000000 |      4000333 |      4 | 2020-
05-21 20:02:15.957835+00
(1 row)
```

各ノードの特定のシーケンスに現在割り当てられている範囲を表示するには、次のクエリを使用します。

- ノード Node1 は 333 から 2000333 の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
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,3); -- second range
range_start | range_end
-----+-----
    1000334 |    2000333
(1 row)
```

- ノード Node2 は 2000004 から 4000003 の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)') など）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ます。

ノードはチャンクを終了すると、新しいチャンクのコンセンサスを求め、最初に利用可能なチャンクを取得します。この場合、4000334 から 5000333 になります。これは新しい予約チャンクであり、古い予約チャンクの消費をスタートします。

2.9.2 UUID、KSUUID、およびその他のアプローチ

BDR で使用できるグローバルシーケンスを使用せずにグローバルに一意的 ID を生成する方法は他にもあります。例：

- UUID、およびその BDR バリエーション、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、）
- 外部調整されたナチュラルキー

BDR シーケンスは他の方法を行に使用できないことにノートして BDR。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のギャップレスである必要があるシーケンスの生成を調整するか、BDR グループ内の 1 つのノードでのみ生成する必要があります。

UUID と KSUUID

UUID キーは、代わりにシーケンスを完全に回避し、128 ビットのユニバーサル一意識別子を使用します。これらはランダムまたは擬似ランダムラージ値であり、同じ値を 2 回生成することはほとんど不可能です。UUID キーを使用する場合、ノードが継続的な通信を行う必要はありません。

非常にまれな衝突が発生したイベント、競合検出は、挿入された 2 つのレコードのうちの新しい方を選択して保持します。衝突ロギングは、有効にすると、そのようなイベントを記録しますが、衝突は約 2^{64} キーが生成された後にのみ発生するため、**例外的に**発生しません。

UUID キーの主な欠点は、ややスペースと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所やネットワークで送信される場合にも多くのスペースを消費することです。さらに、すべてのアプリケーションが UUID キーにうまく対応しているわけではありません。

BDR は、KSUUID として知られる UUID データタイプの K-Sortable バリエーションを操作するための関数を提供します。KSUUID 値は、UUID 標準に従って、タイムスタンプとランダムデータの両方を格納するという点で UUIDv1 に似ています。違いは、KSUUID は K-Sortable、つまりタイムスタンプで弱くソートできることです。これにより、よりコンパクトな **btree** インデックスが生成され、検索の有効性が向上し、結果データの自然な時間ソートが可能になるため、データベースキーとしてより便利になります。UUIDv1 とは異なり、KSUUID の値には、それらが生成されたコンピュータの MAC が含まれないため、KSUUID s を使用してもセキュリティ上の問題はありません。

KSUUID v2 はすべての場合に推奨されます。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、v1 と v2 の 2 つのバージョンがあります。レガシーの 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いるため、新規インストールには使用しないでくだ

さい。v1 と v2 の内部コンテンツは互換性がなく、それらを実行する機能も互換性はありません。KSUUID の v2 には、UUID バージョン番号も保存されなくなりました。

ステップとオフセットのシーケンス

オフセットステップシーケンスでは、各ノードで通常の PostgreSQL シーケンスが使用されます。各シーケンスは同じ量でインクリメントし、異なるオフセットで開始します。例、ステップ 1000 では、ノード 1 のシーケンスは 1001、2001、3001 などを生成し、ノード 2 は 1002、2002、3002 などを生成します。このスキームは、ノードが拡張通信できない場合でも適切に機能しますが、設計者は最大値を指定する必要がありますスキーマを確立する際のノードの数、およびノードごとの構成が必要です。ただし、間違いによりシーケンスが簡単にオーバーラップする可能性があります。

次のように、1つのノードで目的のシーケンスを作成することにより、BDR でこのアプローチを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指定します。

```
-- On node 1  
SELECT setval(some_seq, 1);  
  
-- On node 2  
SELECT setval(some_seq, 2);  
  
-- ... etc
```

将来的に変更することは困難で混乱を招くため、追加する可能性のあるすべてのノード用に十分なラージさの `INCREMENT` を許可する必要があります。

`bigint` 値を使用すると、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質的な懸念はありません。枯渇に近づくチャンスを得るには、何百ものマシンで毎秒何百万もの挿入を行う必要があります。

BDR は現在、そのようなステップ/オフセットシーケンスでのノードごとのオフセットの構成の自動化を提供していません。

複合キー

ステップ/オフセットシーケンスのバリエーションは、PRIMARY KEY (node_number, generated_value) で構成される複合キーを使用することです。ここで、ノード番号は通常、各ノードで異なる番号を返す関数から取得されます。このような関数は、DDL レプリケーションを一時的に無効にして定数 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 行テーブルを使用して作成できノード。

2.9.3 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の間で変換するためのインタフェースを提供します。

以下の関数は DDL とみなされるため、DDL レプリケーションとグローバルロックが適用され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がシーケンスの種類を設定できるようにしシーケンス。設定すると、seqkind は bdr.sequences ビューを介してのみ可視されます。他のすべての方法では、シーケンスは通常のシーケンスとして表示されます。

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現在アクティブな場合に適用されます。

DML および DDL レプリケーション を参照してください。

概要 `sql bdr.alter_sequence_set_kind(seqoid regclass, seqkind text)`

パラメーター

- seqoid - 変更されるシーケンスの名前または OID
- seqkind - 標準の PostgreSQL シーケンスの場合は local、グローバルに一意の BDR シーケンスの場合は snowflakeid または gallocc、またはレガシーのグローバルに一意のシーケンスの場合は timeshard

注意事項

シーケンスの種類を `galloc` に変更すると、そのシーケンスに最初に割り当てられた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。`galloc` に変更される前にシーケンスで使用されている既存の値がある場合は、次のコマンドを使用して、新しく生成された値が既存の値と競合しないように開始点を移動することをお勧めします。

ALTER SEQUENCE `seq_name` **START** `starting_value` **RESTART**

このファンクションは、DDL ステートメントと同じ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、グローバルな DDL ロックを取得します。また、シーケンスをローカルでロックします。

このファンクションはトランザクションです。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で可視です。

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

bdr.extract_timestamp_from_snowflakeid

このファンクションは、`snowflakeid` シーケンスのタイムスタンプコンポーネントを抽出します。結果値は「`timestampz`」型です。

概要 `sql bdr.extract_timestamp_from_snowflakeid(snowflakeid bigint)`

パラメーター- `snowflakeid` -スノーフレイクイドシーケンスの値

注意事項

このファンクションはローカルノードでのみ実行されます。

bdr.extract_nodeid_from_snowflakeid

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

概要 `sql bdr.extract_nodeid_from_snowflakeid(snowflakeid bigint)`

パラメーター- `snowflakeid` -スノーフレイク ID シーケンスの値

注意事項

このファンクションはローカルノードでのみ実行されます。

`bdr.extract_localseqid_from_snowflakeid`

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

概要 `sql bdr.extract_localseqid_from_snowflakeid(snowflakeid bigint)`

パラメーター- `snowflakeid` -スノーフレイク ID シーケンスの値

注意事項

このファンクションはローカルノードでのみ実行されます。

`bdr.timestamp_to_snowflakeid`

このファンクションは、タイムスタンプ値をダミーのスノーフレイク ID シーケンス値に変換します。

これは、`snowflakeid` 列の値および特定のタイムスタンプのインデックス付き検索または比較を行う場合に役立ちます。

例、`snowflakeid` シーケンスを使用している列 `id` を持つテーブル `foo` があれば、次のように昨日の午前 0 時からの変更数を取得できます。

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

このように定式化されたクエリーは、列 `id` でインデックススキャンを使用します。

概要 `sql bdr.timestamp_to_snowflakeid(ts timestamptz)`

パラメーター- `ts` -スノーフレイク ID シーケンス生成に使用されるタイムスタンプ

注意事項

このファンクションはローカルノードでのみ実行されます。

bdr.extract_timestamp_from_timeshard

このファンクションは、timeshard シーケンスのタイムスタンプコンポーネントを抽出します。結果値は「timestampz」型です。

概要

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

パラメーター

- timeshard_seq - タイムハードシーケンスの値

注意事項

このファンクションはローカルノードでのみ実行されます。

bdr.extract_nodeid_from_timeshard

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

概要

```
bdr.extract_nodeid_from_timeshard(timeshard_seq bigint)
```

パラメーター

- timeshard_seq - タイムハードシーケンスの値

注意事項

このファンクションはローカルノードでのみ実行されます。

bdr.extract_localseqid_from_timeshard

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

概要

```
bdr.extract_localseqid_from_timeshard(timeshard_seq bigint)
```

パラメーター

- `timeshard_seq` - タイムハードシーケンスの値

注意事項

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シャードシーケンス値に変換します。

これは、`timeshard` 列の値と特定のタイムスタンプのインデックス付き検索または比較を行う場合に役立ちます。

例、`timeshard` シーケンスを使用している列 `id` を持つテーブル `foo` があれば、次のように昨日の午前 0 時からの変更数を取得できます。

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

このように定式化されたクエリーは、列 `id` でインデックススキャンを使用します。

概要

```
bdr.timestamp_to_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

注意事項

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。

2.9.4 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。

概要

<code>bdr.gen_ksuuid_v2(timestamptz)</code>

注意事項

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 の値を比較します。

最初の値の方が新しい場合は 1、2 番目の値の方が小さい場合は -1、等しい場合は 0 を返します。

概要

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

パラメーター

- UUID - 比較する KSUUID v2

注意事項

この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。結果値は「timestampz」型です。

概要

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

パラメーター

- UUID - タイムスタンプを抽出する KSUUID v2 値

注意事項

このファンクションはローカルノードでのみ実行されます。

2.9.5 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。結果値のタイプは「UUID」です。

概要

```
bdr.gen_ksuuid()
```

注意事項

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。

最初の値の方が新しい場合は 1、2 番目の値の方が小さい場合は-1、等しい場合は 0 を返します。

概要

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

注意事項

この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。結果値は「timestampz」型です。

概要

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

パラメーター

- UUID -タイムスタンプを抽出する KSUUID v1 値

注意事項

このファンクションはローカルノードでのみ実行されます。

2.10 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、新しい方を保持する場合があります。これにより、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスが確立されます。

ただし、行レベルではなく列レベルで競合を解決する方が適切な場合があります。

シンプル例を考えます。2つの整数列 *a* と *b*、および単一の行 (1,1) を持つテーブル *t* があります。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（先行の UPDATE 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_newer の競合解決では、コミットのタイムスタンプを比較し、新しい行バージョンを保持します。2番目のノードが最後にコミットされたと仮定すると、最終的に (1,100) になり、列 *a* への変更を効果的に破棄します。

多くのユースケースでは、これは望ましい動作ですが、問題になる場合があります。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、共有テーブル内の列の専用サブセットを更新するマルチノードクラスターを考えます。その場合、さまざまなコンポーネントがお互いのつま先を踏んで、変更を上書きする可能性があります。

このようなユースケースでは、特定のテーブルの競合を列レベルで解決する方が適切な場合があります。これを実現するために、BDR は各列の最終変更のタイムスタンプを個別に追跡し、それを使用して最新の値（基本的に update_if_newer）を選択します。

前の例に適用すると、どちらのノードもそのような行を見たことがないにもかかわらず、両方のノードで (100,100) になります。

列レベルの競合解決を考えると、テーブルを垂直にパーティション分割されていると見ると便利です。これにより、各更新は1つのスライスのデータにのみ影響します。このアプローチにより、列の異なるサブセットに対する変更間の競合が排除されます。実際、垂直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になることさえあります。

列レベルの競合解決では、テーブルに `REPLICA IDENTITY FULL` が必要です。 `bdr.alter_table_conflict_detection` ファンクションはそれをチェックし、それ以外の場合はエラーで失敗します。

2.10.1 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()` ファンクションによって管理されます。

例

`bdr.alter_table_conflict_detection()` の使用方法を確認するには、簡単なテーブル `test_table` を作成し、それで列レベルの競合解決を有効にする次の例を考えます。

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

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

db=# SELECT bdr.alter_table_conflict_detection(
db(# my_app.test_table::regclass, column_modify_timestamp, cts);
  alter_table_conflict_detection
-----
t

db=# \d my_app.test_table
```

このファンクションは、(ファンクション呼び出しで指定されたように) 新しい `cts` 列を追加しますが、各変更の前に新しい列のタイムスタンプを維持する責任がある2つのトリガー (`BEFORE INSERT` および `BEFORE UPDATE`) も作成しました。

また、新しい列はデフォルト値で `NOT NULL` を指定しています。これは、`ALTER TABLE ... ADD COLUMN` はテーブルの書き換えを実行しないことを意味します。

注釈: `bdr.column_timestamps` データタイプの列は、悪影響を与える可能性があるため、他の目的で使わないことをお勧めします。例、テーブルを列レベルの競合解決に切り替えますが、これはトリガーがないと正しく機能しません。

列レベルの競合解決を備えたリストテーブル

次のクエリで列レベルの競合解決が有効になっているテーブルをリストできます。このクエリーは、`bdr.column_timestamp` 型の列の存在を検出します。

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

`bdr.column_timestamps_create`

このファンクションは、列レベルの競合解消を作成します。`column_timestamp_enable` 内で呼び出されます。

概要

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestamptz)
```

パラメーター

- `p_source` — 2つのオプションは、`current` または `commit` です。
- `p_timestamp` — タイムスタンプは、選択したソースによって異なります。`commit` の場合、`TIMESTAMP_SOURCE_COMMIT`。`current` の場合、`TIMESTAMP_SOURCE_CURRENT`。

2.10.2 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場合、コードは DDL ロックを使用して、スイッチ前からの保留中の変更がないことを保証します。このアプローチにより、両方のタプルまたはどちらでもタイムスタンプの競合のみが表示されます。そうしないと、コードはローカルタプルにタイムスタンプを、リモートのタプルに NULL を予期せずに表示する可能性があります。また、変更がすべてのノードで同じ方法（列レベルまたは行レベル）で解決されるようにします。

2.10.3 現在とコミットのタイムスタンプ

重要な決定は、変更された列に割り当てるタイムスタンプです。

デフォルトでは、変更された列に割り当てられるタイムスタンプは、`clock_timestamp` から取得したかのように現在のタイムスタンプです。これはシンプルで、多くの場合、完全に正しいです（例、競合する行が列の重複しないサブセットを変更する場合）。

ただし、さまざまな予期しない効果があります。

- タイムスタンプはステートメントの実行中に変更されるため、`UPDATE` がマルチプルの行に影響を与える場合、それぞれがわずかに異なるタイムスタンプを取得します。これは、同時変更の効果がさまざまな方法で「混在」する可能性があることを意味します（さまざまなノードで実行された変更がどのように正確にインターリーブするかに応じて）。
- タイムスタンプはコミットタイムスタンプとは無関係であり、競合を解決するために使用すると、結果がコミットオーダーと同等になりません。つまり、シリアル化できない可能性があります。

注釈: 将来的にステートメントとトランザクションのタイムスタンプを追加する可能性があります。これにより、同時実行のステートメントまたはトランザクションの影響が混在する問題に対処します。それでも、これらのオプションはいずれもコミット オーダーと同等の結果を生成できません。

実際のコミットタイムスタンプを使用することもできますが、この機能は現在実験的と見なされています。コミット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);
```

`bdr.column_timestamps_disable` を使用して無効にできます。

コミットタイムスタンプには現在、[注意事項](#)で説明されている制限があります。

2.10.4 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格納する列は、トリガーによって自動的に維持されます。直接変更しないでください。競合がどのように解決された例を調査しながら、現在のタイムスタンプ値を検査すると役立ちます。

このためには、次の3つの関数が役立ちます。

- `bdr.column_timestamps_to_text(bdr.column_timestamps)`

このファンクションは、タイムスタンプマッピングの人間が読める表現形式を返し、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 表現形式を変換し、値を `jsonb` にキャストするときに使用されます。

```
db=# select jsonb_pretty(cts::jsonb) from test_table;
           jsonb_pretty
-----
{
    "map": {
        "2": "2018-09-23T19:24:52.118583+02:00"
    },
    "source": "current",
    "default": "2018-09-23T19:24:52.118583+02:00"
}
(1 row)
```

- `bdr.column_timestamps_resolve(bdr.column_timestamps, xid)`

このファンクションは、最新のトランザクション（すで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で 2 番目の属性が更新されました（`attnum = 2` を使用）。

```
test=# select cts::jsonb from test_table;
           cts
-----
→{"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
→"2018-09-23T19:29:55.581823+02:00", "modified": [2]}
(1 row)

db=# select bdr.column_timestamps_resolve(cts, xmin)::jsonb from test_table;
           column_timestamps_resolve
-----
→{"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
→"2018-09-23T19:29:55.581823+02:00"}
(1 row)
```

2.10.5 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より高いタイムスタンプを持つ値を選択し、もう一方を破棄します。ただし、別のより複雑な方法で競合を調整できます。例、情報を破棄せずに競合する値をマージできる CRDT タイプを使用できます。

2.10.6 注意事項

- **UPDATE** によって変更される属性は、トリガーの古い行と新しい行を比較することによって決定されます。これは、属性が値を変更しない場合、明示的に設定されても変更されたものとして検出されないことを意味します。例、**UPDATE t SET a = a** は、どの行でも **a** を変更済みとしてマークしません。同様に、**UPDATE t SET a = 1** は、既に **1** に設定されている行に対して **a** を変更済みとしてマークしません。
- **INSERT** ステートメントの場合、新しい行と比較する古い行がないため、すべての属性が変更されると見なし、新しいタイムスタンプを割り当てます。これは、**INSERT** ステートメントに含まれず、デフォルト値を受け取った列にも適用されます。デフォルト値を持つ属性は検出できますが、自動的に含まれたか、明示的に指定されたかはわかりません。

これは、**INSERT** ステートメントが列の異なるサブセットを指定している場合でも、列レベルの競合解決が **INSERT-INSERT** の競合では機能しないことを効果的に意味します。新しい行には、古い行よりもすべて新しいタイムスタンプがあります。

- 列を独立して扱うことにより、すべての変更が同じノードで発生する場合に不可能な方法でコンストレーンに違反するのは簡単です。例、次のようなテーブルを考えます。

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

1つのノードが次のことを行うと仮定します。

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

別のノードが同時に行います。

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

これらの各更新は、最初の行で実行された場合に有効であるため、各ノードに渡されます。ただし、他のノードに複製すると、結果の行は **CHECK (A > b)** 制約に違反し、問題が手動で解決されるまで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カラムは自動管理されます。予測できない影響を与える可能性があるため、クエリで値を指定またはオーバーライドしないでください。（とにかく、可能な場合は値を無視します。）

- ・タイムスタンプのマッピングはトリガーによって維持されますが、トリガーが実行されるオーダーが重要です。したがって、タプルを変更し、**pgl_cldc_** トリガーの後に実行されるカスタムトリガーがある場合、変更された列は正しく検出されません。

- 通常のタイムスタンプを使用して変更/コミットをオーダーする場合、競合する変更のタイムスタンプがまったく同じである可能性があります（2つ以上のノードがたまたま同じタイムスタンプを生成したため）。このリスクは、通常の行レベルの競合解決でも発生する可能性があるため、列レベルの競合解決に固有のものではありません。このシチュエーションでは、ノード ID をタイブレーカーとして使用します（高いノード ID が勝ちます）。これにより、すべてのノードに同じ変更が適用されます。
- 異なるノード間にクロックスキューが存在する可能性があります。やや予期しない動作を引き起こす可能性があります（タイムスタンプが逆になっているため、一見新しい変更を破棄します）、パラメーター `bdr.maximum_clock_skew` および `bdr.maximum_clock_skew_action` を使用してノード間のクロックスキューを管理できます。

```
SELECT bdr.alter_node_group_config(group, ignore_redundant_updates := false);
```

2.11 Conflict-free Replicated Data Types

競合のないレプリケートデータ型（CRDT）は、行の 1 つを破棄する代わりに、同時に変更される行の値のマージをサポートしています。

各 CRDT 型は個別の PostgreSQL データタイプとして実装され、`bdr.crdt_handlers` カタログに追加のコールバックがあります。マージプロセスは、適用側の BDR ライタ内で発生します。追加のユーザ操作は必要ありません。

CLCD の章に記載されているように、CRDT では、テーブルで列レベルの競合解決を有効にする必要があります。

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データ型ではなく、`CREATE/ALTER TABLE` で特定のデータタイプ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単一の行を持つ次のテーブルを考えます。

```
CREATE TABLE non_crdt_example (
    id      integer      PRIMARY KEY,
    counter integer      NOT NULL DEFAULT 0
);

INSERT INTO non_crdt_example (id) VALUES (1);
```

2 つのノードで次の SQL を同時に発行したとします。

```
UPDATE non_crdt_example
SET counter = counter + 1 -- "reflexive" update
WHERE id = 1;
```

…両方の更新が適用された後、このクエリーを使用して結果の値を確認できます。

```
SELECT * FROM non_crdt_example WHERE id = 1;
id | counter
```

(次のページに続く)

(前のページからの続き)

```

-----+-----
 1 |          1
(1 row)

```

… update_if_newer conflict リゾルバにより、インクリメントの 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と、次のようになります。

```

CREATE TABLE crdt_example (
    id          integer          PRIMARY KEY,
    counter bdr.crdt_gcounter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_example,
    column_modify_timestamp, cts);

INSERT INTO crdt_example (id) VALUES (1);

```

再度、2 つのノードで次の SQL を発行し、変更が適用されるのを待ちます。

```

UPDATE crdt_example
    SET counter = counter + 1 -- "reflexive" update
WHERE id = 1;

SELECT id, counter FROM crdt_example WHERE id = 1;
 id | counter
-----+-----
 1 |          2
(1 row)

```

これは、CRDT により、競合する非同期同時更新に直面した場合でも、アキュムレーター列が正しく機能することを示しています。

crdt_gcounter タイプは、状態ベースの CRDT タイプの例であり、上記のように $x = x + 1$ などの再帰 UPDATE SQL でのみ機能します。

bdr.crdt_raw_value 構成オプションは、クエリが CRDT 型の現在の値を結果か、完全な内部状態を返すかを決定します。デフォルトでは、現在の数値のみが返されます。true に設定すると、クエリは完全な状態の表現形式を結果ます。特別なハッシュ演算子 (#) を使用すると、特別な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た場合、値は bdr.crdt_raw_value = on でのみ再設定できます。

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（たとえば、WHERE 句または選択リスト内の式）での列参照は、引き続き # 演算子の使用が必要な場合があります。

CRDT データ型の別のクラスが存在します。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、値の更新が同じノード上の以前の値と自動的に比較され、変更が他のすべてのノードにデルタとして適用されます。

```
CREATE TABLE crdt_delta_example (
    id          integer          PRIMARY KEY,
    counter     bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_delta_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_delta_example,
    column_modify_timestamp, cts);

INSERT INTO crdt_delta_example (id) VALUES (1);
```

2つのノードで次の SQL を同時に発行したとします。

```
UPDATE crdt_delta_example
    SET counter = 2          -- notice NOT counter = counter + 2
WHERE id = 1;
```

両方の更新を適用した後、このクエリーを使用して結果の値を確認できます。

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
 id | counter
----+-----
  1 |      4
(1 row)
```

通常の integer 列の場合、結果はもちろん 2 になります。しかし、デルタ CRDT カウンタで行を UPDATE するときは、古い行バージョンからスタートで新しい行バージョンを make し、両方をリモートノードに送信します。そこで見つけたバージョンと比較します（ローカル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、デルタをローカル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（+、#、!）と多くの一般的な集計関数（min、max、sum、avg）が pg_catalog で make され、bdr を調整せずに使用できます。

重要な問題は、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、他に何もすることなく、推定とクエリー計画が正常に機能します。

2.11.1 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。

オペレーションベースの CRDT タイプ (CmCRDT)

操作は明示的に転送されるのではなく、リモートノードから受け取った古い行と新しい行から計算されるため、オペレーションベースの型の実装は非常に簡単です。

現在、次のオペレーションベースの CRDT を実装しています。

- `crdt_delta_counter - bigint` カウンタ (インクリメント/デクリメント)
- `crdt_delta_sum - numeric` 総和 (インクリメント/デクリメント)

これらの型は、既存のデータ型を活用し (たとえば、`crdt_delta_counter` は `bigint` のドメインです)、デルタを計算するためのビットのコード。

このアプローチは、デルタの計算方法を知っている型でのみ可能ですが、結果は非常にシンプルで安価で (スペースと CPU の両方の観点から)、いくつかの追加の利点があり構文 (例: 基になるデータタイプ)。

主な欠点は、非同期および並行環境でこの値を確実にリセットできないことです。

注: 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、より複雑なオペレーション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、マルチプルのオペレーションは必要ありません)。しかし、その時点で、スペース要件 (ノードごとの状態が必要です)。

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 (インクリメントのみ)
- `crdt_gsum - numeric` 総和/カウンタ (インクリメントのみ)
- `crdt_pncounter - bigint` カウンタ (インクリメント/デクリメント)
- `crdt_pnsum - numeric` 総和/カウンタ (インクリメント/デクリメント)

通常、内部状態にはノードごとの情報が含まれるため、ディスク上のサイズは大きくなりますが、追加の利点があります。カスタムデータ型を実装する必要性は、より多くのコード (イン/アウト関数と演算子) を意味します。

利点は、値を確実にリセットできること、変更が失われた場合の自己修復性 (適切に操作されたクラスターでは発生しない)、およびソースノード以外から変更を受信できることです。

例、ノード A で値が変更され、変更は B にレプリケートされますが、C にはレプリケートされません (A と C 間のネットワーク問題のため)。B が値を変更し、この変更が C に複製されると、A からのオリジナ

ルの変更も含まれます。オペレーションベースの CRDT では、ノード C は AC ネットワーク接続が再び機能し始めるまで変更を受け取りません。

CvCRDT の主な欠点は、ディスクスペースの両方のコストが高いことです。既にクラスターから削除されたノードを含む、ノードごとにビットの情報が必要です。状態の複雑な性質（varlena タイプにシリアル化）は、CPU 使用率の増加を意味します。

2.11.2 ディスク領域の要件

重要な考慮事項は、CRDT タイプ、特にディスク上のサイズに関連するオーバーヘッドです。

オペレーションベースのタイプの場合、これはかなり簡単です。タイプは他のタイプの上の単なるドメインであり、ディスクスペース要件は同じです（ノードの数に関係なく）。

- `crdt_delta_counter - bigint` と同じ（8 バイト）
- `crdt_delta_sum - numeric` と同じ（変数、精度と位取りに応じて）

操作ベースの CRDT タイプはノードごとの情報を格納しないため、ノード数に依存しません。

状態ベースの型の場合、シチュエーションはより複雑です。すべての型は可変長（基本的に `bytea` 列として保存）であり、ヘッダーと、値を**変更**した各ノードの一定量のノードごとの情報で構成されます。

`bigint` バリエントの場合、おおよそのサイズを計算する式は次のとおりです（`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` バリエントの場合、ヘッダー部分とノードごとの部分の両方に `numeric` 可変長値が含まれるため、正確な式はありません。このような値をいくつ保持する必要があるかを示すには、次のようにします。

- `crdt_gsum` - 固定：20B (header) + $N * 4B$ (per-node) - 変数： $(2 + N)$ numeric の値
- `crdt_pnsum` - 固定：20B (header) + $N * 4B$ (per-node) - 変数： $(4 + 2 * N)$ numeric の値

注：値がマルチプルのノードで更新されない場合、クラスター内のノードの数は関係ありません。また、更新が同時に行われた（競合が発生した）かどうか関係ありません。

注：これらのノードのうちのいくつかがクラスターから既に削除されていても構いません。状態を圧縮する方法はまだありません。

2.11.3 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場合があるため（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 マージの両方を実行する必要があります。

競合の解決が最初に行われ、どのタプル（`applytuple`）を保持し、どのタプルを破棄するかを決定します。次にマージフェーズが発生し、破棄されたタプルの CRDT 列のデータが `apply` タプルにマージされます。

注：これにより、競合解決で高速パス（現在のトランザクションで変更されたもの）を使用できる場合でも、マージを毎回行うニーズがあるため、CRDT タイプは単純な競合解決と比べてやや高価になります。

2.11.4 CRDT タイプと競合レポート

デフォルトでは、検出された競合は個別に報告されます。CRDT 型がなければ、これは完全に理にかなっています。競合解決は基本的に使用可能な情報の半分（構成に応じてローカルまたはリモートの行）を破棄するためです。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結合できるため、データロスの問題が解消されます。これにより、競合の報告が不要になります。

このため、CRDT マージで競合を完全に解決できる場合、つまり各列が次の 2 つの条件の少なくとも 1 つを満たす場合、競合のレポートをスキップします。

- 1) ローカルタプルとリモータプルの値が同じ (NULL または等しい)
- 2) CRDT データタイプを使用する（したがって、マージできます）

注：これは、CRDT 列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が、ローカル/リモートのタプルの値はすべて同じです。

2.11.5 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的な性質は、異なるノードが変更ストリームのさまざまな場所でリセットオペレーションを認識する可能性があることを意味します（実装方法に関係なく）。異なるノードが同時にリセットを開始することもできます。つまり、他のノードからのリセットを監視する前。

言い換えれば、リセットオペレーションを正しく動作させるには、通常の操作に対して可換であるニーズがあります。値をリセットする多くの素朴な方法（単一ノードでは完全にうまくいくかもしれませんが）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アプローチは次のとおりです。

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

状態ベースの CRDT では、これは機能しません-他のノードの状態は破棄しますが、ローカルにのみです。リモートノードのマージ機能によって追加され、値が発散し、最終的に他のノードでの変更により戻されます。

操作ベースの CRDT では、更新が `-cnt` の減算として解釈されるため、これは機能しているように見える場合があります。ただし、同時リセットがない場合にのみ機能します。2 つのノードが同時にリセットを試行すると、最終的にデルタを 2 回適用し、負の値を取得します（これはリセットから予想したものではありません）。

`DELETE + INSERT` はリセットとして使用できるように見えるかもしれませんが、これにもいくつかの弱点があります。行が同じキーで再挿入された場合、すべてのノードが操作のストリーム内の同じ位置でそれを見ることは保証されません（他のノードからの変更に関して）。BDR は、同時発生 of データ異常につながる可能性があるため、同じ主キー値を再利用することをお勧めしません。

状態ベースの CRDT 型は、次のような特別な! 演算子を使用して、リセットを確実にハンドルできます。

```
UPDATE tab SET counter = !counter WHERE ...;
```

「確実に」とは、値に上記の2つの問題、つまり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ます。Eager Replication を使用して、いずれかの種類の CRDT を特定の値に設定することもできます。

2.11.6 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-grow-only カウンタ and 総和、正負のカウンタ and 総和、および delta カウンタ and 総和。カウンターとサムはほとんど同じように動作しますが、「カウンタ」型は整数ベース (bigint) であり、「総和」型は 10 進ベース (numeric) です。

[1] で説明されている追加の CRDT タイプは、後で実装される可能性があります。

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)

- 非負の値のインクリメントのみをサポート (value + int および counter + bigint 演算子)
- カウンタの現在の値は、# 演算子を使用するか、bigint にキャストして取得できます
- counter = value のようシンプル割り当てとは互換性がありません (これは、アプリケーションのどこかで新しい値が計算される場合の一般的なパターンです)
- ! 演算子 (counter = !counter) を使用して、カウンタをシンプルにリセットできます
- crdt_gcounter_to_text を使用して内部状態を検査できます

```
CREATE TABLE crdt_test (
    id          INT PRIMARY KEY,
    cnt         bdr.crdt_gcounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824);     -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);     -- error: negative value

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
```

(次のページに続く)

(前のページからの続き)

```

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)

- 非負値のインクリメントのみをサポート (sum + numeric)
- 総和の現在の値は、# 演算子を使用するか、numeric にキャストして取得できます
- sum = value のようシンプル割り当てとは互換性がありません (これは、アプリケーションのどこかで新しい値が計算される場合の一般的なパターンです)
- ! 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;

```

(次のページに続く)

(前のページからの続き)

```

SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment sum
UPDATE crdt_test SET gsum = gsum + 11.5 WHERE id = 1;
UPDATE crdt_test SET gsum = gsum + 120.33 WHERE id = 2;

- - error: minus operator not defined
UPDATE crdt_test SET gsum = gsum - 15.2 WHERE id = 1;

- - error: increment has to be non-negative
UPDATE crdt_test SET gsum = gsum + (-1.56) WHERE id = 1;

- - reset sum
UPDATE crdt_test SET gsum = !gsum WHERE id = 1;

- - get current sum value
SELECT id, gsum::numeric, gsum FROM crdt_test;

- - show internal structure of sums
SELECT id, bdr.crdt_gsum_to_text(gsum) FROM crdt_test;

```

ポジティブネガティブカウンタ (crdt_pncounter)

- 正と負の両方の値のインクリメントをサポート (counter + int および counter + bigint 演算子を介して)
- カウンタの現在の値は、# 演算子を使用するか、bigint にキャストして取得できます
- counter = value のようシンプル割り当てとは互換性がありません (これは、アプリケーションのどこかで新しい値が計算される場合の一般的なパターンです)
- ! 演算子 (counter = !counter) を使用して、カウンタをシンプルにリセットできます
- crdt_pncounter_to_text を使用して内部状態を検査できます

```

CREATE TABLE crdt_test (
    id          INT PRIMARY KEY,
    cnt         bdr.crdt_pncounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

- - enable CLCD on the table

```

(次のページに続く)

(前のページからの続き)

```

ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET cnt = cnt - 73     WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12)  WHERE id = 3;

-- get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

-- show internal structure of counters
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;

-- reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

-- get current counter value after the reset
SELECT id, cnt::bigint, cnt FROM crdt_test;

```

正負の総和 (crdt_pnsum)

- 正と負の両方の値のインクリメントをサポート (sum + numeric を介して)
- 総和の現在の値は、# 演算子を使用するか、numeric にキャストして取得できます
- sum = value のようシンプル割り当てとは互換性がありません (これは、アプリケーションのどこかで新しい値が計算される場合の一般的なパターンです)
- ! 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

```

(次のページに続く)

(前のページからの続き)

```

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 pnsun = pnsun + 1.44      WHERE id = 1;
UPDATE crdt_test SET pnsun = pnsun + 12.20     WHERE id = 2;
UPDATE crdt_test SET pnsun = pnsun + (-24.34) WHERE id = 3;

-- decrement sums
UPDATE crdt_test SET pnsun = pnsun - 7.3       WHERE id = 1;
UPDATE crdt_test SET pnsun = pnsun - 192.83    WHERE id = 2;
UPDATE crdt_test SET pnsun = pnsun - (-12.22) WHERE id = 3;

-- get current sum value
SELECT id, pnsun::numeric, pnsun FROM crdt_test;

-- show internal structure of sum
SELECT id, bdr.crdt_pnsun_to_text(pnsun) FROM crdt_test;

-- reset sum
UPDATE crdt_test SET pnsun = !pnsun WHERE id = 1;

-- get current sum value after the reset
SELECT id, pnsun::numeric, pnsun 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
- 正と負の両方の値のインクリメントをサポート
- counter = value のようシンプル割り当てと互換性があります (アプリケーションのどこかで新しい値が計算される場合に一般的)
- 値をリセットするシンプル方法がない (確実に)

```

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

```

(次のページに続く)

(前のページからの続き)

```

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
- 正と負の両方の値のインクリメントをサポート
- sum = value のようシンプル割り当てと互換性があります（アプリケーションのどこかで新しい値が計算される場合に一般的）
- 値をリセットするシンプル方法がない（確実に）

```

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

```

(次のページに続く)

(前のページからの続き)

```

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-wiki> >

2.12 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ライバーには送信されません。変更が `logical_decoding_work_mem` (PostgreSQL 13 以降) を超える場合、それらはディスクにスピルされます。これは、特にラージトランザクションでは、サブスライバーに到達するまでに遅延が発生し、パブリッシャーで追加の I/O が必要になることを意味します。

PostgreSQL 14 以降、トランザクションはオプションでデコードして、パブリッシャーでコミットされる前にサブスライバーに送信できます。サブスライバーは、受信した変更をステージングファイル（またはファイルのセット）に保存し、トランザクションのコミット時に適用します（またはトランザクションが中断した場合は破棄します）。これにより、トランザクションがコミットされるとすぐにサブスライバーにトランザクションを適用できます。

2.12.1 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。

- パブリッシャーで変更をディスクに書き込む必要はなくなりましたが、変更は各サブスライバーでディスクに書き込む必要があります
- トランザクションが打ち切られた場合、作業（各サブスライバーが受け取った変更と関連するストレージ I/O）は無駄になります

ただし、バージョン 3.7 から、BDR はパラレル適用をサポートし、各サブスライバでマルチプルのライタプロセスを有効にします。これを活用して、次の機能強化を提供します。

- デコードされたトランザクションは、サブスライバーのライタに直接ストリーミングできます
- デコードされたトランザクションは、サブスライバーのディスク上に保存する必要はありません
- サブスライバーでトランザクションの適用を開始する前に、トランザクションがコミットのを待つ必要はありません

警告

- 並列適用を有効にする必要があります
- ワークロードが多く的小さくて競合するトランザクションで構成されている場合、これはライター間で頻繁にデッドロックを引き起こす可能性があります

注釈: ライタへの直接ストリーミングはまだ実験的な機能であり、使用には注意が必要です。具体的には、ストリーミングのコミットタイムスタンプを利用できない場合があるため（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場合があります。

2.12.2 構成

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` を使用した**ノード構成** を介して
- グループレベルで、ファンクション `bdr.alter_node_group_config()` を使用した**グループ構成** を介して

`bdr.default_streaming_mode` を使用したノード構成

許可される値は次のとおりです。

- `off`
- `writer`
- `file`
- `auto`

デフォルト値は `auto` です。

この設定を有効にするには、サブスクリプションごとに `pglogical` レシーバプロセスをリスタートする必要があることに注意してください。これは、サーバーリスタートで実現できます。

`bdr.default_streaming_mode` が `off` 以外の値に設定されている場合、サブスクライバーはパブリッシャーにトランザクションストリーミングを要求します。これがどのように提供されるかは、グループ構成設定によっても異なります。詳細については、以下の `XXX` を参照してください。

bdr.alter_node_group_config() を使用したグループ構成

関数 `bdr.alter_node_group_config()` を使用したグループ構成のパラメータ `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 `:ref:bdr.node_group`<bdr.node_group`>` の列 `node_group_streaming_mode` に含まれています。返される値は単一の `char` 型で、取り得る値は `D` (`default`)、`W` (`writer`)、`F` (`file`)、`A` (`auto`) および `O` (`off`) 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` (トランザクションストリーミングを無効にする) に設定されていない限り、サブスクライバーはトランザクションストリーミングを要求します。

パブリッシャーがトランザクションストリーミングを提供できる場合、トランザクションサイズが `logical_decoding_work_mem` で設定されたしきい値を超えるたびにトランザクションをストリームします。通常、パブリッシャーは、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。一部の状況 (`COPY` など) を除き、サブヒントにトランザクションをライタにストリームするように指示する場合があります (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またはファイルにストリームできます。この決定は、いくつかの要因に基づいています。

- 並列適用がオフの場合 (`num_writers = 1`)、ファイルにストリーミングされます (ライタ 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されたトランザクションのハンドリングで既にビジー状態である場合、新しいトランザクションはファイルにストリーミングされます。[`bdr.writers`](`モニタリング#monitoring-bdr-writers`) を参照して、BDR ライタのステータスを確認してください。

ライタへのストリーミングが可能な場合 (つまり、フリーのライタが利用できる場合)、トランザクションをライタにストリームするかファイルにストリーミングするかは、次の表のようにグループとノードの設定の組み合わせに基づいています。

Group	Node	Streamed to
off	(any)	(none)
(any)	off	(none)
writer	file	file
file	writer	file
default	writer	writer
default	file	file
default	auto	writer
auto	(any)	writer

グループ構成が `auto` に設定されている場合、またはグループ構成が `default` であり、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必要があると示唆した場合にのみ、トランザクションがライタ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のライタにストリームするようにサブヒントにライタしフリー。

- COPY
- CREATE INDEX CONCURRENTLY

2.12.3 モニタリング

トランザクションストリーミングの使用は、サブスクライバーノードの `:ref:`bdr.stat_subscription`<bdr.stat_subscription>` ファンクションを介して監視できます。

- `nstream_writer` -ライタにストリーミングされたトランザクションの数
- `nstream_file` -ファイルにストリーミングされたトランザクションの数
- `nstream_commit` -コミットされたストリーミングトランザクションの数
- `nstream_abort` -中断されたストリーミングトランザクションの数
- `nstream_start` -開始されたストリーミングトランザクションの数
- `nstream_stop` -完全に受信されたストリーミングトランザクションの数

2.13 Durability & Performance Options

2.13.1 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、すべて個別に実装できます。

- **耐久性**：マルチプルのノードに書き込むと、クラッシュ回復力が向上し、クラッシュしてリスタートした後にデータを回復できます。

- **可視性**：クライアントへのコミット確認により、データベースはいくつかのノードセットでのコミットされたトランザクションの即時の可視性を保証します。
- **コミット後の競合なし**：クライアントは、トランザクションが最終的にすべてのノードに適用されることを信頼できます。または、アボートにエラーを直接通知します。

BDR は、同期レプリケーションのバリエーションを提供することにより、耐久性と可視性を保証する **CAMO とグループコミット** 機能を提供します。これは、物理的スタンバイ用の Postgres の `synchronous_commit` 機能に非常に似ていますが、ラージ規模な分散システムにより多くの柔軟性を提供します。

Group Commit に加えて、BDR には 2 つの追加モードもあります（現在、Group Commit と組み合わせることはできません）。

- **Commit At Most Once (CAMO)**。この機能は、ノードまたはネットワークで COMMIT に障害が発生した場合に、トランザクションが COMMIT（およびレプリケート）されたかどうかを知ることに問題を解決します。通常、COMMIT が処理されたかどうかを判断するのはハード場合があります。この機能を使用すると、新しいデータベースコネクションが以前の接続とは異なるノードでも、アプリケーションは何が起こったのかを知ることができます。この機能の詳細については、

Commit At Most Once の章を参照してください。

- **熱心な複製**。これは、コミットの前に競合をチェックするオプションの機能です。すべてのトランザクションはすべて *Eager All-Node Replication* を使用した CAMO の章を参照してください。

Postgres 自分自身は、単方向ですが、`synchronous variant` を提供します。下位互換性のために、BDR は引き続き `synchronous_commit` および `synchronous_standby_names` を介した同期レプリケーションの構成をサポートしています。

BDR を使用したレガシー同期レプリケーション を参照してくださいが、すべての場合に代わりに **CAMO とグループコミット**

を使用することをお勧めします。

この章では、さまざまな形式の同期的または一括レプリケーションとそのタイミングの側面について説明します。

2.13.2 用語と定義

この章以降の章では、さまざまなロールを担う BDR ノードについて説明します。これらはトランザクションごとに暗黙的に割り当てられ、同時トランザクションでも関係ありません。

オリジンはクライアントまたはアプリケーションからトランザクションを受信するノードです。最初にトランザクションを処理し、他の BDR ノードへのレプリケーションを開始し、確認またはエラーでクライアントに応答するのはノードです。

- **パートナーノード**は、グループコミットまたは CAMO 要件に従ってトランザクションを確認することが期待される BDR ノードです。
- **コミットグループ**は、コミットに関係するすべての BDR ノード、つまりオリジンとそのすべてのパートナーノードのグループであり、ほんの一部またはすべての同等なです。

2.13.3 比較

BDR で使用可能な同期レプリケーションのほとんどのオプションは、さまざまなレベルの同期を可能にし、パフォーマンスと、ノードまたはネットワークの停止に対する保護との間でさまざまなトレードオフを提供します。

次の表は、トランザクションが発行されたオリジンノードから COMMIT 確認を受け取った後、クライアントがレプリケートされた同等な期待できるノードをまとめたものです。「モード」列は、バリエーションに応じて異なる意味をとります。BDR を使用した PSR およびレガシー同期レプリケーションの場合、`synchronous_commit` 設定を参照します。CAMO の場合、`bdr.enable_camo` 設定を指します。最後に、Group Commit の場合、

`commit scope configuration` の確認要件を指します。

Variant	Mode	Received	Visible	Durable
async BDR	off (default)	no	no	no
PSR	remote_write (2)	yes	no	no (1)
PSR	on (2)	yes	no	yes
PSR	remote_apply (2)	yes	yes	yes
Group Commit	'ON received' nodes	yes	no	no
Group Commit	'ON replicated' nodes	yes	no	no
Group Commit	'ON durable' nodes	yes	no	yes
Group Commit	'ON visible' nodes	yes	yes	yes
CAMO	remote_write (2)	yes	no	no
CAMO	remote_commit_async (2)	yes	yes	no
CAMO	remote_commit_flush (2)	yes	yes	yes
Eager	n/a	yes	yes	yes
legacy (3)	remote_write (2)	yes	no	no
legacy (3)	on (2)	yes	yes	yes
legacy (3)	remote_apply (2)	yes	yes	yes

- (1) OS に書き込まれ、OS が実行されたままで Postgres のみがクラッシュする場合。
- (2) `synchronous_replication_availability` を `async` に設定してローカルモード（許可されている場合）に切り替えない限り、それ以外の場合は非同期 BDR のデフォルトの値が適用されます。
- (3) 非推奨。代わりに Group Commit の使用を検討してください。

レプレションは、正常に動作している同等なが、それ以上の通信を必要とせず、つまり、完全または部分的なネットワーク停止に直面した場合でも、最終的にトランザクションを適用できることを保証します。この確認には永続的なストレージが関係しないため、同等ながクラッシュした場合でもトランザクションの再ノードが必要になる場合があります。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、他のすべてのクライアントは、コミットがオリジンノードによって確認された直後にこの保証を提供するすべてのノードでトランザクションの結果を表示します。可視性がないと、接続されている他のクライアントがトランザクションの結果を表示できず、古い読み取りが発生する場合があります。

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。これは、データの受信（物理ストリーミングレプリケーションなど）または可視性（グループコミット、CAMO、Eager など）に関連します。前者はクラッシュ後の再送信の必要性を排除し、後者はリスタート後も可視性が維持されることを保証します。

2.13.4 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る方法で適用することを理解すると役立ちます。

物理的ストリーミングレプリケーションの場合、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ローカルに可視にします
- 同等な変更を受け取り、書き込みを発行しノード
- 同等な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にします

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ローカルに可視にします
- 同等なメモリの適用キューへの変更を受け取りノード
- 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にします
- 同等なディスクにフラッシュしてトランザクションを持続します

Group Commit、CAMO、および Eager の場合、オリジンノードはトランザクションをローカルで可視化する前に特定の数の確認を待機する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プリコミットレコードを WAL にフラッシュします
- 同等なメモリの適用キューへの変更を受け取りノード
- 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にします
- 同等なディスクにフラッシュしてトランザクションを持続します
- オリジンはトランザクションをローカルでコミットして可視にします

次の表は、違いをまとめたものです。

Variant	Order of apply vs persist	replication before or after commit
PSR	persist first	after WAL flush of commit record
BDR	apply first	after WAL flush of commit record
Group Commit	apply first	before COMMIT on origin
CAMO	apply first	before COMMIT on origin
Eager	apply first	before COMMIT on origin

2.13.5 構成

次の表は、非デフォルト値（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を与える構成設定の概要を提供します。

setting (GUC)	Group Commit	CAMO	Eager	PSR (1)
synchronous_standby_names	n/a	n/a	n/a	req
synchronous_commit	n/a	n/a	n/a	opt
synchronous_replication_availability	n/a	opt	n/a	opt
bdr.enable_camo	n/a	req	n/a	n/a
bdr.commit_scope	req	n/a	opt	n/a
bdr.global_commit_timeout	opt	opt	opt	n/a

- (1) この列の値は、BDR と組み合わせて使用される `synchronous_commit` および `synchronous_standby_names` にも適用されます。

2.13.6 計画的なシャットダウンと再起動

`remote_write` と組み合わせて受信確認または CAMO を使用する場合、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する前に消費されます。ただし、`immediate` シャットダウンモードでは、キューはシャットダウンで破棄され、停止したノードがキュー内のトランザクションを「忘れ」ます。オリジンノードに同時に障害が発生すると、両方のノードに障害が発生したかのように、データが失われる可能性があります。

適用キューがディスクにフラッシュされるように保証には、メンテナンスタスクに `smart` または `fast` 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。

2.13.7 BDR を使用したレガシー同期レプリケーション

注釈: このアプローチはお勧めしません。代わりに *CAMO とグループコミット* の使用を検討してください。

使用法

BDR を使用した同期レプリケーションを有効にするには、関連する BDR 同等なアプリケーション名前を `synchronous_standby_names` に追加する必要があります。FIRST x または ANY x を使用すると、非 BDR スタンバイノードの要件と競合しない場合、ある程度の柔軟性が提供されます。

追加すると、同期のレベルは `synchronous_commit` を介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` に追加すると同期レプリケーションが有効になります。`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（ロジカルレプリケーションの場合）。

グループコミットへの移行

BDR の Group Commit 機能は、`synchronous_commit` および `synchronous_standby_names` から独立して構成されています。代わりに、`bdr.commit_scope` GUC では、トランザクションごとにスコープを選択できます。また、各ノードで個別に構成された `synchronous_standby_names` の代わりに、グループコミットはグローバルに同期されたコミットスコープを使用します。

注釈: `synchronous_standby_names` とコミットスコープの文法は非常に似ていますが、前者はオリジンノードをアカウントしませんが、後者は説明することにノートすることが重要です。したがって、例 `synchronous_standby_names = 'ANY 1 (..)'` は `'ANY 2 (...)'` のコミットスコープと同等です。この選択により、多数決についての推論が容易になり、オリジンノードもトランザクションの耐久性と可視性に貢献することを反映しています。

2.14 Group Commit

Group Commit の目的は、単一ノードの障害または一時的な停止の場合にデータロスから保護することです。これは、COMMIT 時にトランザクションを正常に受信して確認するために複数の BDR ノードを必要とすることによって実現されます。

2.14.1 要件

通常の実行中、Group Commit はアプリケーションに対して完全に透過的です。フェイルオーバー時に、調整フェーズは、アプリケーションまたは中間のプロキシによって明示的にトリガーまたは統合されるニーズがあります。HARP はグループコミットのネイティブサポートを提供し、調整フェーズをトリガーするため、これをクライアントに対して同様に透過的にします。

オリジンノードでは、グループコミットでコミットされたトランザクションは、下で二相コミットを使用します。したがって、`max_prepared_transactions` は、ノードごとに発生するそのようなすべてのトランザクションをハンドルするのに十分な高さに構成する必要があります。

2.14.2 構成

Group Commit を使用するには、最初に Commit Scope を定義する必要があります。これにより、トランザクションのコミットに関係する BDR ノードが決まります。スコープが確立されると、次のようにグループコミットを使用するようにトランザクションを構成できます。

```
BEGIN;
SET LOCAL bdr.commit_scope = example_scope;
...
COMMIT;
```

この例を完了するには、問題のコミットスコープが以前に次のように定義されている場合があります。

```
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := example_bdr_group,
    rule := ANY 2 (example_bdr_group)
);
```

これは、`example_bdr_group` 名前付けのノードグループが存在し、少なくとも2つのBDRノードをメンバーとして（直接またはサブグループに）含むことを前提としています。`example_scope` 内でコミットされたトランザクションには、グループ内のBDRノードからの追加の確認が1つ必要です。オリジンノードとともに、これはグループ内の「ANY 2」ノードを説明し、コミット後にトランザクションが可視で耐久性があることが保証されます。

オリジングループ

コミットスコープのルールは、トランザクションがコミットされるノード、つまり、どのノードがトランザクションのオリジンとして機能するかによって異なります。これをアプリケーションに対して透過的にするために、BDRでは、トランザクションの発生元に応じてコミットスコープがさまざまなルールを定義できます。

例、ノードが左側と右側の2つのデータセンターにまたがるBDRクラスターを考えます。最上位のBDRノードグループの名前が `top_group` であるとしします。次のコマンドを使用して、サブグループを設定し、ローカルデータセンター内のすべてのノードがトランザクションを確認することを必要とするコミットスコープを作成できますが、リモートからは1つのノードのみです。

```
- - create sub-groups
SELECT bdr.create_node_group(node_group_name := left_dc,
                             parent_group_name := top_group,
                             join_node_group := false);
SELECT bdr.create_node_group(node_group_name := right_dc,
                             parent_group_name := top_group,
                             join_node_group := false);

- - create a commit scope with individual rules for each sub-group
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) AND ANY 1 (right_dc)
);
```

(次のページに続く)

(前のページからの続き)

```

SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) AND ALL (right_dc)
);

```

確認レベル

BDR ノードは、*Commit At Most Once* と同様に、さまざまな時点でトランザクションの確認を送信できます。保護レベルを上げると、確認ノードの観点から、これらは次のとおりです。

- **received** - リモート BDR ノードは、ローカルアプリケーションを開始する前に、トランザクションを受信した直後に確認します
- **replicated** - トランザクションの変更を適用した後、ディスクにフラッシュする前に確認します
- **durable** - すべての変更がディスクにフラッシュされた後にトランザクションを確認する
- **visible** (デフォルト) - すべての変更がディスクにフラッシュされ、同時実行トランザクションから可視になった後、トランザクションを確認します。

コミットスコープのルールでは、次のように **ON** を使用してこれらの確認レベルをノードグループ定義（括弧内）に追加できます。

- ANY 1 (right_dc) ON remote_write
- ALL (left_dc) ON remote_commit_flush (デフォルトで省略可)
- ALL (left_dc) ON remote_commit_async AND ANY 1 (right_dc) ON remote_write

2.14.3 リファレンス

コミットスコープの文法

リファレンスまでに、コミットスコープの文法は次のとおりです。

```

commit_scope: confirmation
              | confirmation AND commit_scope

confirmation: node_def (ON [received|replicated|durable|visible])

node_def:    ANY num (node_group [, ...])
              | MAJORITY (node_group [, ...])
              | ALL (node_group [, ...])

```

コミットスコープルールの追加

ファンクション `bdr.add_commit_scope` は、指定されたコミットスコープ名前とオリジンノードグループのルールを作成します。BDR クラスタ内のすべてのノードでルールが同じ場合、最上位のノードグループに対してこのファンクションを 1 回呼び出すだけで、コミット範囲を完全に定義できます。

または、同じ `commit_scope_name` でマルチプル呼び出される場合がありますが、オリジンノードグループと、トランザクションのオリジンによって異なるコミットスコープのルールがあります。

概要

```
bdr.add_commit_scope(commit_scope_name NAME, origin_node_group NAME,  
                    rule TEXT)
```

コミット範囲ルールの変更

コミットスコープ内の単一のオリジンノードグループの特定のルールを変更するには、ファンクション `bdr.alter_commit_scope` を使用できます。

概要

```
bdr.alter_commit_scope(commit_scope_name NAME, origin_node_group NAME,  
                      rule TEXT)
```

コミットスコープルールの削除

`bdr.remove_commit_scope` を使用して、コミット範囲内の単一のルールを削除できます。コミットスコープにマルチプルのルールが定義されている場合、このファンクションをルールごとに 1 回呼び出して、コミットスコープ全体を完全に削除する必要があります。

概要

```
bdr.remove_commit_scope(commit_scope_name NAME, origin_node_group NAME)
```

2.15 Eager Replication

コミット後の競合を防ぐには、`bdr.commit_scope` パラメータを `global` に設定します。 `local` のデフォルト設定は `eager` レプリケーションを無効にするため、[Conflicts chapter](#) で説明されているように、BDR はコミット後に変更を適用し、潜在的な競合を解決します。

このモードでは、BDR は内部で二相コミット (2PC) を使用して、ローカルコミットの前に競合を検出および解決します。これは、アプリケーションの通常の `COMMIT` を暗黙的な二相コミットに変え、オリジンノードがコミットを続ける前に、すべての同等なトランザクションを準備することを要求します。準備フェーズ中に少なくとも 1 つのノードがダウンしているか到達できない場合、コミットは `bdr.global_commit_timeout` の後にタイムアウトし、トランザクションがアボートされます。PostgreSQL の明示的な 2PC に存在するように、一時テーブルの使用に制約はないことに注意してください。

準備ができると、Eager All-Node Replication は Raft を使用してコミットの決定に到達します。障害が発生した場合、これにより、残りの大部分のノードが一致コミットまたはアボートの決定に到達して、トランザクションを終了できます。これにより、トランザクションによってロックされたオブジェクトとリソースのブロックが解除され、クラスターが続行できます。

すべてのノードが動作可能な場合、オリジンはすべてのノードがコミットした後にのみクライアントにコミットを保証し、コミット後にすべてのノードでトランザクションがすぐに可視されるようにします。

2.15.1 要件

Eager All-Node Replication は準備されたトランザクションを内部で使します。したがって、すべてのレプリカノードは、すべての着信トランザクションをハンドルできるように十分な高さに `max_prepared_transactions` を構成する必要があります（おそらくローカル二相コミットおよび CAMO トランザクションに加えて）。

Configuration: Max Prepared Transactions

を参照)。すべてのノードで同じに、CAMO または Eager 全ノードレプリケーションが使用されるクラスター全体の同時トランザクションの最大数をカバーするのに十分な高さに構成することをお勧めします。それ以外は、特別な構成は必要なく、すべての BDR クラスターで Eager All-Node トランザクションを実行できます。

2.15.2 使用法

Eager All-Node Replication を有効にするには、クライアントはセッションレベルで、またはここに示すように個々のトランザクションに対してグローバルコミットスコープにスイッチする必要があります。

```
BEGIN;

... other commands possible...

SET LOCAL bdr.commit_scope = global;

... other commands possible...
```

クライアントは、トランザクションの最後に COMMIT を発行し続け、BDR に 2 つのフェーズを管理させることができます。

```
COMMIT;
```

2.15.3 エラーハンドリング

BDR がトランザクションを管理することを考えると、クライアントはニーズの結果を確認するだけです（シングルノードの Postgres を含むいずれの場合もお勧めします）。

オリジンノードに障害が発生した場合、残りのノードは最終的に（少なくとも `bdr.global_commit_timeout` の後）、グローバルにプリペアドトランザクションをロールバックすることを決定します。Raft は、一貫性のないコミットとロールバックの決定を防ぎます。ただし、これには接続されたノードの大部分が必要です。切断されたノードは、必要に応じて最終的にコミット（またはロールバック）できるようにトランザクションを準備し、決定してさらに進んだ可能性のあるノードの大部分と調整します。

CAMO を使用した Eager All-Node Replication

Eager All-Node Replication は CAMO を超えており、それを暗示しています。`bdr.commit_scope` が `global` に設定されている場合、`bdr.enable_camo` を追加で有効にする必要はありません。また、CAMO ペアを `bdr.add_camo_pair()` で構成する必要もありません。

CAMO パートナーのロールで他のアクティブな BDR ノードを使用して、トランザクションのステータスをクエリーできます。ただし、この CAMO 以外の使用法は、`require_camo_partner = false` の 3 番目の引数を持つ `bdr.logical_transaction_status` ファンクションに指示するニーズがあります。それ以外の場合、CAMO 構成の欠落について不平を言う場合があります（Eager トランザクションには必要ありません）。

そのファンクションの構成と呼び出しの違いを除き、クライアントは [CAMO/フェイルオーバーオプションを使用した `pgbench`](#) で説明されているプロトコルに従うニーズがあります。reference client implementations を参照してください。

制限事項

Eager Replication を使用するトランザクションはまだ DDL を実行できず、明示的な二相コミットもサポートしていません。これらは後のリリースで許可される可能性があります。TRUNCATE コマンドが許可されていることに注意してください。

クラッシュした回復不能な BDR ノードを物理的スタンバイに置き換えることは、現在、Eager All Node トランザクションと組み合わせてサポートされていません。

BDR は現在、グローバルコミットスコープのみを提供しています。以降のリリースでは、可用性を高めるために少ないノードで Eager Replication をサポートします。

Eager All Node レプリケーションを `synchronous_replication_availability = 'async'` と組み合わせることはできません。両方を構成しようとするエラーが発生します。

Decoding Worker 機能は、現在 Eager All Node トランザクションと組み合わせてサポートされていません。Eager を使用したインストールでは、Eager All Node トランザクションを使用して BDR ノードグループに対して `enable_wal_decoder` を無効にしておく必要があります。

同期レプリケーションは、Eager とは異なるトランザクション確認のメカニズムを使用します。2 つは互換性がないため、一緒に使用しないでください。したがって、Eager All Node トランザクションを使用するときは常に、`synchronous_standby_names` で BDR ノードが構成されていない `make` を確認してください。物理的スタンバイとして機能する非 BDR ノードへの同期レプリケーションを使用することは十分に可能です。

2.15.4 一般的な熱心な複製の効果

コミットレイテンシーの増加

同期ステップを追加することは、ノード間の追加の通信を意味し、コミット時に追加のレイテンシが発生します。Eager All Node Replication は、約 2 つのネットワークラウンドトリップを追加します（最悪のノード、最も遠い同等なへ）。ロジカルスタンバイノードと、まだ参加またはキャッチアップ中のノードは含まれませんが、最終的に変更を受け取ります。

同等ながトランザクションのローカル準備を確認できる前に、ローカルにも適用するニーズがありノード。これにより、トランザクションのサイズによっては、コミットのレイテンシがさらに長くなります。これは `synchronous_commit` 設定とは無関係であり、`bdr.commit_scope` が `global` に設定されている場合は常に適用されることに注意してください。

アバート率の増加

注釈: 現在、Eager Replication のパフォーマンスは予想外に遅いことが知られています（約 10 TPS のみ）。これは、次のリリースで改善される予定です。

シングルノード Postgres、またはデフォルトの非同期レプリケーションモードの BDR を使用しても、COMMIT 時のエラーはまれです。追加の同期ステップによりエラーのソースが追加されるため、アプリケーションはこのようなエラーを適切にハンドルするように準備する必要があります（通常はリトライループを適用します）。

アバートのレートは、ワークロードのみに依存します。多くの行を変更する大規模なトランザクションは、他の同時実行トランザクションと競合する可能性ははるかに高くなります。

2.16 Commit At Most Once

Commit At Most Once (CAMO) 機能の目的は、アプリケーションが複数回コミットするのを防ぐことです。

CAMO がないと、COMMIT が送信された後にクライアントが接続を失うと、アプリケーションはサーバーから応答を受信しないため、トランザクションがコミットされたかがわかりません。

アプリケーションは、次の 2 つのオプションのいずれかを簡単に決定できません。

- データが 2 回入力される場合があるため、同じデータでトランザクションを再試行します
- トランザクションを再試行せず、データがまったく処理されないリスク

これらはいずれも、価値の高いデータに関する重大なエラーです。

このシチュエーションを回避 `make1` つの方法は、トランザクションが一意インデックスを持つテーブルに少なくとも 1 つの `INSERT` を含めることですが、これはアプリケーションのデザインに依存し、アプリケーション固有のエラー処理ロジックが必要であるため、すべての場合。

BDR の CAMO 機能は、より一般的な解決策を提供し、`INSERT` を必要としません。 `bdr.enable_camo` または `bdr.commit_scope` によってアクティブ化されると、アプリケーションは、既に割り当てられている場合、トランザクション識別子を含むメッセージを受け取ります。それ以外の場合、トランザクションの最初の `write` ステートメントはその情報をクライアントに送信します。アプリケーションが明示的な `COMMIT` を送信する場合、プロトコルは、`COMMIT` が送信される前にアプリケーションがトランザクション識別子の通知を受信することを保証します。サーバーが `COMMIT` に応答しない場合、アプリケーションはトランザクション識別子を使用して別の BDR ノードにトランザクションの最終ステータスを要求することにより、このエラーをハンドルできます。以前のトランザクションのステータスがわかっている場合、アプリケーションはトランザクションをリトライするかどうかを安全に決定できます。

CAMO は 2 つのモードのいずれかで動作します。

- ペアモード
- Eager All Node Replication あり

ペアモードでは、CAMO は同じトップレベルの BDR グループからの 2 つの BDR マスタノードであるパートナーノードのペアを作成することによって機能します。このオペレーションモードでは、ペアの各ノードは他の同等なで実行された最近のトランザクションの結果を知っており、特に（必要に応じて）`COMMIT` 中に切断されたトランザクションの結果を知っています。アプリケーションからトランザクションを受信するノードは「オリジン」と呼ばれ、これらのトランザクションを確認するノードは「パートナー」と呼ばれます。ただし、CAMO ペアのノードの CAMO 構成に違いはありません。ペアは対称です。

Eager All-Node Replication を使用した CAMO と組み合わせると、CAMO はすべての同等な（つまり、完全な BDR マスタノード）が CAMO パートナーとして機能できるようにします。このモードでは、指定された CAMO パートナーを構成しないでください。

警告: CAMO で高度なエラーハンドリングを利用するには、ユーザーのアプリケーションを変更する必要があります。保護を得るには、パラメータを有効にするだけでは不十分です。リファレンスクライアントの実装は、要求に応じて顧客に提供されます。

2.16.1 要件

CAMO を使用するには、アプリケーションは明示的な COMMIT メッセージを別の要求として発行する必要があります（マルチステートメント要求のパートとしてではありません）。CAMO は、プロシージャまたは暗黙的なコミットを使用する単一ステートメントのトランザクションから発行されたトランザクションのステータスを提供できません。

2.16.2 構成

既存の BDR クラスターがノード `node1` および `node2` で構成されていると仮定します。両方のノードは `bdrdemo` という BDR 対応データベースのパートであり、両方とも同じノードグループ `mygroup` のパートです。ノードを相互に CAMO パートナーとして構成できます。

1. ノード `node1` および `node2` が `mygroup` ノードグループのパートである BDR クラスターを作成します。

1. 1つのノードでファンクション `bdr.add_camo_pair()` を実行します。

```
SELECT bdr.add_camo_pair(mygroup, node1, node2);
```

1. CAMO が提案する COMMIT エラーハンドリングを使用するようにアプリケーションを調整します。

サーバーレベルで CAMO を有効にすることはお勧めしません。必要ない場合でも、すべてのトランザクションのレイテンシが高くなります。代わりに、セッションまたはトランザクションレベルで CAMO をオンにして、個々のトランザクションに対して選択的に有効にします。

セッションレベルで CAMO を有効にするには：

```
SET bdr.enable_camo = remote_commit_flush;
```

トランザクションを開始してからコミットする前に、個々のトランザクションで CAMO を有効にするには：

```
SET LOCAL bdr.enable_camo = remote_commit_flush;
```

CAMO を有効にする `bdr.enable_camo` の有効な値は次のとおりです。

- `off`（デフォルト）
- `remote_write`
- `remote_commit_async`
- `remote_commit_flush` または `on`

各モードの動作の詳細については、同期レプリケーションモードの [違いのあるノード間の比較](#) を参照してください。`bdr.enable_camo = off` を設定すると、この機能が無効になりデフォルト。

Eager All-Node Replication を使用した CAMO

Eager All-Node Replication で CAMO を使用するために、どちらのノードも変更する必要はありません。トランザクションのスタート後にグローバルコミットスコープを有効にすれば十分です。`bdr.enable_camo` を設定する必要はありません。

```
BEGIN;  
SET LOCAL bdr.commit_scope = global;  
...  
COMMIT;
```

指定されたとおりに COMMIT エラーハンドリングを使用するようにアプリケーションを調整するニーズがありますが、使用可能な BDR ノードにフリーに接続して、トランザクションのステータスをクエリーできます。

2.16.3 障害シナリオ

さまざまな構成でさまざまな障害シナリオが発生します。

レシーバ側でのデータの永続性

デフォルトでは、PGL ライタはリモートノードからトランザクションを適用するときに `bdr.synchronous_commit = off` モードで動作します。これは CAMO にも当てはまります。つまり、おそらく CAMO パートナーのディスクに到達する前に、オリジンノードに対してトランザクションが確認されます。クラッシュまたはハードウェア障害が発生した場合、確認されたトランザクションが CAMO パートナーで自分自身で回復できない可能性があります。CAMO パートナーノードが回復するとトランザクションを再分散するため、これは CAMO オリジンノードが動作している限り問題ではありません。

これは、CAMO が単一ノード障害から保護できることを意味します。これは、ローカルモードだけでなく、リモート書き込みとの組み合わせにも適しています。

CAMO ペアの両方のノードの停止をカバーするには、`bdr.synchronous_commit = local` を使用して、コミット前の確認の前にフラッシュを強制します。これはリモート書き込みモードでもローカルモードでも機能せず、レイテンシの影響を受けやすいコミットパスにある CAMO パートナーの I/O 要件により、パフォーマンスにインパクトします。

ローカルモード

`synchronous_replication_availability = 'async'` の場合、ノード（つまり、マスタ）はその CAMO パートナーの準備ができているかどうかを検出します。そうでない場合は、一時的にローカルモードに切り替わります。ローカルモードの場合、ノードは CAMO モードに戻るまでトランザクションをローカルにコミットします。

これにより、COMMIT ステータスを取得することはできませんが、一貫性よりも可用性を選択できます。このモードは、単一ノードの障害を許容できます。CAMO ペアの両方のノードに障害が発生した場合、可用性を維持するために一致しないコミット決定を選択し、データの一貫性が失われる可能性があります。

CAMO パートナーが準備完了にスイッチには、接続するニーズがあり、推定キャッチアップインターバルが `bdr.global_commit_timeout` を下回るニーズがあります。CAMO パートナーの現在の準備ステータスは `bdr.is_camo_partner_ready` で確認できますが、`bdr.node_replication_rates` はキャッチアップ時間の現在の見積もりを提供します。

CAMO 保護からローカルモードへのスイッチは、コミットがトランザクションを超えるか、CAMO パートナーが既にわかっている場合はコミット時に切断された場合にのみトリガーされます。このスイッチは、推定キャッチアップインターバルとは無関係です。Raft にローカルモードへのスイッチを必要とするように CAMO ペアが構成されている場合、このスイッチには大部分のノードが動作している必要があります（

`bdr.add_camo_pair` の `require_raft`

フラグを参照）。これにより、分離されたノードがローカルモードに切り替わることによるスプリットブレインのシチュエーションを防ぐことができます。CAMO ペアに `require_raft` が設定されていない場合、オリジンノードはすぐにローカルモードに切り替わります。

CAMO パートナーへの TCP 接続のキープアライブとタイムアウトを制御する PostgreSQL 設定を使用して、送信ノードでの検出を構成できます。`wal_sender_timeout` は、ノードがローカルモードに切り替えるまで CAMO パートナーを待機する時間です。さらに、`bdr.global_commit_timeout` 設定は、CAMO パートナーに到達できないために COMMIT が発生する可能性のある最大遅延にトランザクションごとの制限を設定します。`wal_sender_timeout` よりも低い場合がありますが、これは同期的スタンバイにも影響し、応答性と安定性の良い妥協点を見つける必要があります。

ローカルモードから CAMO モードへのスイッチは、接続を開始する CAMO パートナーノードによって異なります。CAMO パートナーは、少なくとも 30 秒ごとに再接続を試行します。したがって、接続が再確立された後、CAMO パートナーがオリジンノードに接続するまでに最大 30 秒かかる場合があります。CAMO パートナーで蓄積された遅延は、CAMO 保護モードへのスイッチをさらに遅延させます。

通常の CAMO オペレーションとは異なり、ローカルモードでは追加のコミットオーバーヘッドはありません。これは、CAMO ペアが通常処理できるよりも多くのトランザクションをノードが継続的に処理できるようにするため、問題になる可能性があります。CAMO パートナーが最終的に再接続してトランザクションを適用したとしても、そのようなシチュエーションでは遅延が増加するだけであり、CAMO 保護を再確立できません。トランザクションスループットを人為的に調整するために、BDR は `bdr.camo_local_mode_delay` 設定を提供します。これにより、ローカルモードで COMMIT を任意の時間遅延できます。予想されるワークロード中に通常の CAMO モードでコミット時間を測定し、それに応じてこの遅延を構成することをお勧めします。デフォルトは 5 ミリ秒で、ローカルネットワークと比較的迅速な CAMO パートナーの応答を反映しています。

アーキテクチャと可用性の要件をビューして、ローカルモードを許可するかどうかの選択を検討します。次の例は、いくつかの詳細を提供します。

例：対称ノード

この例では、相互に CAMO パートナーである 2 つの BDR ノードを使用したセットアップを検討します。これは、BDR4 以降で可能な唯一の構成です。

この構成により、両方のノードで CAMO 動作が有効になります。したがって、競合が発生しない可能性が高い場合など、複数のノードで同時に書き込むことが許容されるワークロードパターンに適しています。

ローカルモードの場合

ローカルモードが許可されている場合、単一障害点はありません。1 つのノードに障害が発生した場合：

- 他のノードは、障害が発生したノードでの COMMIT 中に切断されたすべてのトランザクションのステータスを判断できます。
- 新しい書き込みトランザクションが許可されます。- 2 番目のノードにも障害が発生した場合、その時点でコミットされていたトランザクションの結果は不明です。

ローカルモードなし

ローカルモードが許可されていない場合、各ノードはトランザクションをコミットするために他のノードを必要とします。つまり、各ノードは単一障害点です。1 つのノードに障害が発生した場合：

- 他のノードは、障害が発生したノードでの COMMIT 中に切断されたすべてのトランザクションのステータスを判断できます。
- ノードが回復するまで、新しい書き込みトランザクションは防止されます。

2.16.4 アプリケーション使用

概要と要件

CAMO は、クライアントサイドでのリトライループと特定のエラーハンドリングに依存しています。それには 3 つの側面があります。

*トランザクションの COMMIT の結果を確認ニーズがあり、一時的なエラーの場合、クライアントはトランザクションをリトライする必要があります。

- COMMIT の前に、クライアントはノード ID とトランザクション ID（両方とも 32 ビット整数）で構成されるトランザクションのグローバル識別子を取得する必要があります。

*トランザクションの COMMIT を試行中に現在のサーバーに障害が発生した場合、アプリケーションは CAMO パートナーに接続し、そのトランザクションのステータスを取得し、応答に応じてリトライする必要があります。

アプリケーションは、COMMIT 中に切断された場合のトランザクションステータスを確認する目的のみ、グローバルトランザクション識別子を保存する必要があります。特に、アプリケーションに追加の永

続化レイヤーは必要ありません。アプリケーションが失敗した場合、リスタートするのはデータベースのニーズのみです。

CAMO ペアの追加

ファンクション `bdr.add_camo_pair()` は、BDR ノードの既存のペアを対称 CAMO ペアとして構成します。

`require_raft` オプションは、`synchronous_replication_availability` が `async` に設定されている場合にローカルモードにスイッチ方法と時期を制御し、一般的にそのようなスイッチを許可します。

概要

```
bdr.add_camo_pair(node_group text, left_node text, right_node text,
                  require_raft bool)
```

注釈: `left` と `right` の名前には特別な意味はありません。

注釈: BDR バージョン 4.0 以降、対称 CAMO 構成のみがサポートされています。つまり、ペアの両方のノードが互いの CAMO パートナーとして機能します。

CAMO ペアの構成の変更

ファンクション `bdr.alter_camo_pair()` を使用すると、`require_raft` を切り替えることができます。現在、ペアリングのノードを変更することはできません。代わりに、`bdr.remove_camo_pair` の後に `bdr.add_camo_pair` を使用する必要があります。

概要

```
bdr.alter_camo_pair(node_group text, left_node text, right_node text,
                    require_raft bool)
```


CAMO ペアの削除

ファンクション `bdr.remove_camo_pair()` は、2つのノードの CAMO ペアを削除し、これらの2つのノードで `bdr.enable_camo` による CAMO トランザクションの使用を禁止します。

概要

```
bdr.remove_camo_pair(node_group text, left_node text, right_node text)
```

注釈: `left` と `right` の名前には特別な意味はありません。

CAMO パートナー接続ステータス

ファンクション `bdr.is_camo_partner_connected` を使用すると、ペアモードで構成された CAMO パートナーノードの接続ステータスを確認できます。現在、Eager Replication で使用される CAMO に相当するものはありません。

概要

```
bdr.is_camo_partner_connected()
```

戻り値

CAMO パートナーが現在ローカルノード上の WAL 送信者プロセスに接続しているため、トランザクションデータを受信して確認を送信できるかどうかを示すブール値。

CAMO パートナーの準備

ファンクション `bdr.is_camo_partner_ready` を使用すると、ペアモードで構成された CAMO パートナーノードの準備ステータスを確認できます。その下で、これはローカルモードへの/からのスイッチをトリガーします。

概要

```
bdr.is_camo_partner_ready()
```

戻り値

CAMO パートナーがタイムリーに（`bdr.global_commit_timeout` が期限切れになる前に）ローカルノードから発信されたトランザクションを確認すると合理的に期待できるかどうかを示すブール値。

注釈: このファンクションは、過去または現在の状態を照会します。正の結果値は、CAMO パートナーが将来のトランザクションを確認できるかどうかを示しません。

CAMO パートナーを取得する

このファンクションは、ローカルノードの CAMO パートナー（ペアモードで構成）を表示します。

```
bdr.get_configured_camo_partner()
```

CAMO パートナーからの適用キューが消費されるのを待ちます

ファンクション `bdr.wait_for_camo_partner_queue` は、デフォルトで CAMO パートナーノードをクエリーする `bdr.wait_for_apply_queue` のラッパーです。ローカルノードが CAMO ペアのパートでない場合、エラーが発生します。

概要

```
bdr.wait_for_camo_partner_queue()
```

CAMO ノード間のトランザクションステータス

このファンクションにより、CAMO トランザクションが完全に解決されるのを待機できます。

```
bdr.camo_transactions_resolved()
```

取引状況クエリーファンクション

ノードに障害が発生したときにコミットされていたトランザクションのステータスを確認するには、アプリケーションでこのファンクションを使用する必要があります。

```
bdr.logical_transaction_status(node_id, xid, require_camo_partner)
```

ペアモードで使用する CAMO では、CAMO ペアのパートであるノードでのみこのファンクションを使用します。Eager レプリケーションとともに、すべてのノードで使用できます。

どちらの場合も、コミットが発行されてから 15 分以内にファンクションを呼び出す必要があります。CAMO パートナーは、このようなメタ情報を定期的に削除する必要があるため、古いトランザクションに正しい回答を提供できません。

トランザクションのステータスを照会する前に、このファンクションは受信キューが消費されて完全に適用されるのを待ちます。これにより、受け取ったがまだ適用されていないトランザクションに対する早期の否定的な回答が防止されます。

その名前にもかかわらず、これは常に読み取り専用のオペレーションではありません。ステータスが不明の場合、CAMO パートナーはトランザクションをコミットかアボートするかを決定し、その決定をローカルに保存して今後の一貫性を保証します。

クライアントは、オリジンでコミット前にこのファンクションを呼び出さないでください。そうしないと、トランザクションが強制的にロールバックされる場合があります。

概要

```
bdr.logical_transaction_status(node_id OID,
                               xid      OID,
                               require_camo_partner BOOL DEFAULT true)
```

パラメーター

- **node_id** — トランザクションの発生元の BDR ノードのノード ID。通常、PQ パラメータ `bdr.local_node_id` から COMMIT する前にクライアントによって取得されます。
- **xid** — オリジンノードのトランザクション ID。通常、PQ パラメータ `transaction_id` から COMMIT する前にクライアントによって取得されます（`enable_camo` を on、`remote_write`、`remote_commit_async`、または `remote_commit_flush` に設定する必要があります）。

Commit At Most Once settings を参照)

- **require_camo_partner** — デフォルトは true で、構成チェックを有効にします。これらのチェックを無効にし、Eager All-Node Replication によって保護されたトランザクションのステータスをクエリーするには、false に設定します。

戻り値

このファンクションは、次のいずれかの結果を返します。

- **'committed'::TEXT** — トランザクションはコミットされ、CAMO ペアの両方のノードで可視であり、最終的に他のすべての BDR ノードに複製されます。クライアントがリトライする必要はありません。
- **'aborted'::TEXT** — トランザクションは中止されました。他の BDR ノードに複製されることはありません。クライアントは、トランザクションをリトライするか、失敗をコミットするニーズがあります。
- **'in progress'::TEXT** — トランザクションはこのローカルノードでまだ進行中であり、まだコミットまたは中止されていません。トランザクションは COMMIT フェーズにあり、CAMO パートナーがコミットを確認または拒否するのを待っている可能性があります。推奨されるクライアントの反応は、オリジンノードから切断して CAMO パートナーに再接続して、代わりにクエリーすることです。間にロードまたはプロキシがあり、クライアントが照会するノードを制御できない場合、クライアントはステータスが **'committed'** または **'aborted'** に切り替わるまでのみ繰り返しポーリングできます。

Eager All-Node Replication の場合、同等なはまだコミットまたは中止されていないトランザクションに対してこの結果を生成します。これは、この場合、まだレプリケートされていない（またはオリジンノードで開始されていない）トランザクションでも、同等な BDR ノードで **in progress** の結果が得られる可能性があることを意味します。ただし、クライアントは、オリジンでコミット前にトランザクションステータスをクエリーしてはなりません。

- **'unknown'::TEXT** — 指定されたトランザクションは、将来のものであるか、その特定のノードにまだレプリケートされていないか、過去すぎるためです。そのようなトランザクションのステータスはまだないか、不明です。この結果値は、クライアントによる不適切な使用の兆候です。

クライアントは、エラー時にファンクション呼び出しをリトライする準備ができている必要があります。

2.16.5 DDL とグローバルロックとの相互作用

CAMO によって保護されたトランザクションには、DDL 操作を含めることができます。ただし、DDL はグローバルロックを使用します。これにより、ノード間の同期が既に提供されます。詳細については、[DDL ロックの詳細](#) を参照してください。

CAMO と DDL を組み合わせると、レイテンシが長くなり、グローバルデッドロックの可能性も高くなります。したがって、比較的低い `bdr.global_lock_timeout` を使用することをお勧めします。これにより、DDL が中止されるため、妥当な時間でデッドロックが解決されます。

非トランザクション DDL

次の DDL 操作はトランザクションブロックで許可されていないため、CAMO 保護のメリットを得られません。これらの場合、CAMO は内部で自動的に無効になります。

- すべての同時インデックス操作（CREATE、DROP、および REINDEX）
- REINDEX DATABASE、REINDEX SCHEMA、および REINDEX SYSTEM
- VACUUM

*パラメータなしの CLUSTER

- ALTER TABLE DETACH PARTITION CONCURRENTLY
- ALTER TYPE [enum] ADD VALUE
- ALTER SYSTEM
- CREATE および DROP DATABASE
- CREATE および DROP TABLESPACE
- ALTER DATABASE [db] TABLESPACE

2.16.6 CAMO の制限

- CAMO は、オリジンノードで最近失敗した COMMIT の結果をクエリーするように設計されているため、切断された場合は、CAMO パートナーからトランザクションステータスをすぐに要求するようにアプリケーションをコードします。障害が発生してからステータスを要求するまでの遅延をできるだけ短くします。アプリケーションは、15 分以上保存される CAMO の決定に依存してはなりません。
- アプリケーションが、例リスタートの結果として、割り当てられたグローバル識別子を忘れた場合、それを回復する簡単な方法はありません。したがって、アプリケーションは、未処理のトランザクションが終了するのを待ってからシャットダウンすることをお勧めします。
- クライアントが適切なチェックを適用するには、CAMO によって保護されたトランザクションを暗黙的なトランザクション制御を持つ単一のステートメントにすることはできません。トランザクション制御プロシージャまたはトランザクションをスタートまたは終了しようとする DO ブロックで CAMO を使用することもできません。
- CAMO はコミットステータスを解決しますが、コミットで保留中の通知をまだ解決していません。CAMO および Eager レプリケーションオプションでは、NOTIFY SQL コマンドまたは pg_notify() ファンクションは許可されません。LISTEN または UNLISTEN も許可しません。
- 変更を再生するとき、CAMO トランザクションは他のトランザクションと同じように競合を検出する場合があります。タイムスタンプの競合検出を使用する場合、CAMO トランザクションはオリジンノードの準備のタイムスタンプを使用します。これは、トランザクションがオリジンノード自身自身で可視になる前です。

2.16.7 パフォーマンスへの影響

CAMO は、コミット時にメッセージのラウンドトリップを追加することにより、Postgres レプリケーションプロトコルを拡張します。アプリケーションは、非同期レプリケーションよりもコミットのレイテンシ時間が長くなります。これは、主に関連するノード間のラウンドトリップ時間によって決まります。同時セッションの数を増やすと、並列処理が増えて、適切なトランザクションスループットがヘルプされます。

トランザクションを確認する CAMO パートナーは、トランザクション状態を保存する必要があります。非 CAMO オペレーションと比較して、これには、オリジンから適用されるトランザクションごとに追加のシークが必要になる場合があります。

2.16.8 クライアントアプリケーションのテスト

クライアントサイドで CAMO を適切に使用することは簡単ではありません。ノードのクラッシュやネットワークの停止などの障害シナリオに対して、BDR クラスターを使用してアプリケーションの動作をテストすることを強くお勧めします。

2.16.9 CAMO とグループコミット

CAMO は現在 *CAMO* とグループコミットでは動作しません。

2.17 Lag Control

BDR オリジンノード上のデータベースアプリケーションのデータスループットは、コミットされたデータをダウンストリームの同等な安全にレプリケートできるレートを超える場合があります。この不一致が高可用性アプリケーションで一定期間を超えて、または慢性的に続く場合、ディザリカバリまたはビジネス継続性計画に関連する組織の目標が満たされない可能性があります。

Replication Lag Control (RLC) 機能は、動的レート制限デバイスを使用してこの不均衡を調整し、BDR グループノード間のデータフローがこれらの組織の目標に準拠するように設計されています。これは、BDR ノード間のレプリケーションラグのエクステンツを制御することによって行います。

これらの目的の一部は次のとおりです。

- 1) 目標復旧時点 (RPC) は、通常時間として表される計画外のイベントが原因で失われる可能性のあるデータの最大量を指定します。レプリケートされないシステムでは、RPC を使用してバックアップ間隔を設定し、コミットされたデータが障害によって失われるリスクを制限します。レプリケートされたシステムの場合、RPC は、1 つ以上の同等な安全に適用されていないコミットされたデータの許容量を判断します。
- 2) リソース制約目標 (RCO) は、有限のストレージコンストレインがあることを認識しています。このストレージには、データベースファイル、WAL、およびオペレーションを続行するために必要な一時ファイルまたは中間ファイルが含まれます。レプリケートされたシステムの場合、遅延が増加すると、これらのストレージリソースに対する要求も増加します。
- 3) Group Elasticity Objective (GEO) は、ノードが同等な許容できるように保存できないクリップで新しいデータを発信していないことを保証します。その場合、その条件の検出は、データベースノードの数

を拡張する決定の1つのメトリックとして使用できます。同様に、その状態が緩和すると、データベースノードの数を縮小するという決定に影響を与える可能性があります。

ラグ制御は、クライアント接続が READ WRITE トランザクションをコミットレートで制御することにより、レプリケーションの遅延を管理します。レプリケーションラグは、達成する目的に応じて、ラグタイムまたはラグサイズとして測定されます。トランザクションのコミットレートは、構成された BDR コミット遅延時間を使用して調整されます。

2.17.1 要件

ラグ制御の使用を開始するには、次の手順を実行します。

- 1) すべてのデータベースアプリケーションで許容できる最大のコミット遅延時間 `bdr.lag_control_max_commit_delay` を決定します。
- 2) 使用するラグメジャーを決定します。ラグサイズ `bdr.lag_control_max_lag_size` またはラグタイム `bdr.lag_control_max_lag_time` のいずれかを選択する必要があります。
- 3) ノードの最小許容数として、ステップ 2) で選択したラグメジャーを満たす必要があるグループ `bdr.lag_control_min_conforming_nodes` 内の BDR ノードの数を決定します。

2.17.2 構成

ラグ制御は、`postgresql.conf` 構成パラメーターを使用してグループ内の各 BDR ノードで構成されます。ラグ制御を有効にするには、`bdr.lag_control_max_commit_delay` と、`bdr.lag_control_max_lag_size` または `bdr.lag_control_max_lag_time` のいずれかをゼロ以外の正の値に設定する必要があります。

`bdr.lag_control_max_commit_delay` は、必要に応じてサブミリ秒設定を含む小数パートを使用したミリ秒の指定を許可し、推奨します。

デフォルトでは、`bdr.lag_control_min_conforming_nodes` は 1 に設定されています。完全なリストについては、

[ラグ制御](#) を参照してください

2.17.3 概要

ラグ制御は、クライアント接続レベルで動作する動的な TPS レート制限メカニズムです。構成されたラグ制御のコンストレインを満たしながら、できるだけ目立たないように設計されています。これは、十分な BDR ノードが構成されたラグメジャーしきい値を下回るのに十分な速度で変更をレプリケートできる場合、BDR ランタイムのコミット遅延は 0 ミリ秒に固定されることを意味します。

これに当てはまらない場合、BDR ランタイムのコミット遅延は必要に応じて最小限に調整されますが、準拠ノードの数が最小しきい値に戻るまでそれ以上にはなりません。

最小ノードしきい値に達した後でも、ラグ制御は BDR ランタイムのコミット遅延をゼロに戻すドライブとし続けます。BDR コミット遅延は、ほとんどの場合、平衡レベルで上下しますが、データスループットまたはラグ適用率が向上すると、コミット遅延は減少します。

BDR コミット遅延は、コミット後の遅延です。これは、トランザクションがコミットされた後、トランザクションによってロックまたは取得されたすべての Postgres リソースが解放された後に発生します。したがって、遅延は、同時アクティブなトランザクションがその値を監視または変更したり、そのリソースを取得したりすることを妨げません。Postgres 拡張機能で管理される外部リソースについては、同じ保証はできません。拡張機能の依存関係に関係なく、BDR 拡張機能が `postgresql.conf` の拡張機能ベースのリソースマネージャーの前にリストされている場合、同じことが保証されます。

厳密に言えば、BDR コミット遅延はトランザクションごとの遅延ではありません。これは、特定のクライアント接続のトランザクションのストリームでのコミット遅延の平均値です。この手法により、コミットの遅延と値のきめの細かい調整が、OS スケジューラの粗い粒度、クロック割り込み、システムロードによるバリエーションをエスケープします。また、BDR ランタイムのコミット遅延を可能な限り最小の期間のマイクロ秒以内に解決して、ラグメジャーのしきい値を維持できます。

最後に、BDR のコミット遅延は Postgres のコミット遅延と混同しないでください。それらは無関係であり、異なる機能を実行するため、互いに置き換えないでください。

2.17.4 取引申請

BDR コミット遅延は、ユーザアプリケーションのデータを変更するすべての READ WRITE トランザクションに適用されます。これは、宣言された READ WRITE トランザクションを含む、データを変更しないトランザクションはコミット遅延から免除されることを意味します。

非同期トランザクションコミットは、BDR コミット遅延も実行します。これは直観に反するよう見えるかもしれませんが、非同期コミットは、そのパフォーマンスのおかげで、レプリケーションラグの最大の原因の 1 つになる可能性があります。

Postgres と BDR の補助プロセスは、トランザクションコミットで遅延しません。特に、BDR ライターは、ローカルノードでリモートトランザクションを適用するときにコミット遅延を実行しません。BDR ライターは送信レプリケーションのラグに何も寄与せず、トランザクションのコミットが遅延によって調整されないことにより、受信レプリケーションのラグを最も削減できるため、これはデザインです。

2.17.5 制限

最大コミット遅延 `bdr.lag_control_max_commit_delay` は、ハードリミットを表す上限値です。これは、いかなる状況においても、コミット遅延が設定された値を超えないことを意味します。逆に、最大ラグメジャー `bdr.lag_control_max_lag_size` および `bdr.lag_control_max_lag_time` は、超えることができるソフト制限です。最大コミット遅延に達した場合、遅延の継続的な増加を防ぐための遅延メジャーに対する追加のバックプレッシャーはありません。

BDR レプリケーションセットを変更しないオリジントランザクションをコミット遅延から免除する方法はありません。これらのトランザクションでは、最大トランザクション遅延をゼロ (0) に設定すると役立つ場合があります。

2.17.6 注意事項

アプリケーション TPS は、レプリケーションの遅延に影響を与える可能性のある多くの要因の 1 つです。その他の要因には、BDR コミット遅延の効果が低下する可能性のあるトランザクションの平均サイズが含まれます。特に、バルクロード操作ではレプリケーションの遅延が増加する可能性があり、最大許容遅延未満ではあるが、通常の実アプリケーションで合理的に予想されるレベルを超えて BDR ランタイムのコミット遅延が増加する可能性があります。

同様に、OLTP 要件が非常に高く、データ変更が少ないアプリケーションは、許容可能な BDR コミット遅延設定によって過度に制限される可能性があります。

これらの場合、SET [SESSION|LOCAL] コマンドを使用して、これらのアプリケーションのラグ制御設定をカスタム構成したり、それらのアプリケーションを変更したりすると便利な場合があります。例、バルクロード操作は、トランザクションスナップショット期間と WAL 保持サイズを制限したり、バルクロードが失敗した場合のリスタートポイントを確立したりするために、マルチプル of の小さなトランザクションに分割される場合があります。ラグ制御に従うと、これらのトランザクションのコミットは、非常に長い BDR コミット遅延をスケジュールして、以前の部分的なバルクロードによって発生した遅延を消化できます。

2.17.7 組織目標の達成

前述の目標リストの例：

- RPO は、適切な最大ラグタイムを設定することで満たすことができます。
- RCO は、適切な最大ラグサイズを設定することで満たすことができます。
- GEO は、BDR ランタイムのコミット遅延と BDR ランタイムの測定値をモニタリングすることで満たすことができます。

前述のように、BDR ランタイムの最大コミット遅延が BDR で構成されたコミット遅延制限に固定され、ラグメジャーの一貫性が BDR で構成された最大レベルを超える場合、それは BDR グループ拡張のマーカーになる可能性があります。

2.18 AutoPartition

AutoPartition を使用すると、自動パーティショニング管理によってテーブルを簡単にラージサイズに増やすことができます。これは、パーティションの作成と削除の競合の少ないロックなどの 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したときに削除できます。

BDR 管理は、主に SQL 呼び出し可能な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ています。 `search_path` に入れない限り、各関数の名前をスキーマ修飾する必要があります。

2.18.1 パーティションの自動作成

`bdr.autopartition()` は、表の自動レンジ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作成され、それ以外の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のメンテナンスアクションがいつどのように行われるかの定義を変更するだけです。

`bdr.autopartition()` は、テーブルの残りの部分をロックせずにパーティションをアタッチまたはデタッチ/ドロップできる機能を活用します（アンダーレイの Postgres バージョンがサポートしている場合）。

テーブルが RANGE パーティション化されていない場合、またはマルチカラムパーティションキーが使用されている場合、ERROR が発生します。

`partition_increment` の値のレンジごとに新しいパーティションが追加され、下限と上限が `partition_increment` 離れています。 `timestamp` または `date` 型のパーティションキーを持つテーブルの場合、`partition_increment` は `interval` 型の有効な定数でなければなりません。例、1 Day を指定すると、1 日おきのパーティション境界で新しいパーティションが毎日追加されます。

パーティション列が `snowflakeid`、`timeshard` または `ksuuid` シーケンスに接続されている場合、`partition_increment` 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 万個の値ごとに新しいパーティションが追加されます。

テーブルに既存のパーティションがない場合、指定された `partition_initial_lowerbound` が最初のパーティションの下限として使用されます。`partition_initial_lowerbound` 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` から値を導き出そうとします。例、`partition_increment` が 1 Day として指定された場合、`partition_initial_lowerbound` は自動的に `CURRENT DATE` に設定されます。`partition_increment` が 1 Hour として指定された場合、`partition_initial_lowerbound` は現在の日付の現在の時間に設定されます。後続のパーティションの境界は、`partition_increment` 値を使用して設定されます。

システムは常に、特定の最小数のアドバンスパーティションを作成しようとします。新しいパーティションを作成するかどうかを判断するオーダーに、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 の評価可能な式にすることができます。例、列型 `date` のパーティションテーブルの場合、`partition_autocreate_expression` が `DATE_TRUNC('day', CURRENT_DATE)` として指定され、`partition_increment` が 1 Day として指定され、`minimum_advance_partitions` が 2 として指定された場合、最後のパーティションの上限が `DATE_TRUNC('day', CURRENT_DATE) + '2 Days'::interval` より小さいになるまで新しいパーティションが作成されます。

この式は、システムが新しいパーティションをチェックするたびに評価されます。

列型 `integer` の分割テーブルの場合、`partition_autocreate_expression` は `SELECT max(partcol) FROM schema.partitioned_table` として指定できます。システムは、パーティション分割された列の最大値が最後のパーティションの上限の `minimum_advance_partitions * partition_increment` の距離内にあるかどうかを定期的にチェックします。クエリーが効率的に実行されるように、ユーザは `partcol` にインデックスを作成する必要があります。列型 `integer`、`smallint` または `bigint` のパーティションテーブルに `partition_autocreate_expression` が指定されていない場合、システムは自動的に `max(partcol)` に設定します。

`data_retention_period` が設定されている場合、この期間が経過するとパーティションは自動的に削除されます。ロックを最小限に抑えるために、新しいパーティションが追加されると同時にパーティションが削除されます。設定されていない場合、パーティションを手動で削除する必要があります。

`data_retention_period` パラメータは、タイムスタンプ（および関連する）ベースのパーティションでのみサポートされています。期間はパーティションの上限を考慮して計算され、パーティションは二次的なテーブルスペースに移行されるか、上限に対して指定された期間が経過すると削除されます。

デフォルトでは、AutoPartition はパーティションをグローバルに管理します。言い換えれば、1つのノードでパーティションが作成されると、クラスター内の他のすべてのノードにも同じパーティションが作成されます。したがって、すべてのパーティションは一貫しており、使用できることが保証されています。このために、AutoPartition は Raft を利用します。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、パーティションテーブルがレプリケートテーブルではないため、すべてのノードにすべてのパーティションを持つ必要がない場合、または望ましくない場合に役立ちます。例、ビルトインの `bdr.conflict_history` テーブルはレプリケートテーブルではなく、ローカルの AutoPartition によって管理されます。各ノードはこのテーブルのパーティションをローカルに作成し、十分に古くなると削除します。

`managed_locally` としてマークされたテーブルは、後でグローバルに管理するように変更することはできません。その逆も同様です。

アクティビティは、エントリーが `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必要はありません。これを `make` と、AutoPartition メタデータに一貫性がなくなり、失敗する可能性があります。

自動パーティションを構成する

`bdr.autopartition` ファンクションは、テーブルの自動パーティションを構成します。

概要

```
bdr.autopartition(relation regclass,  
    partition_increment text,  
    partition_initial_lowerbound text DEFAULT NULL,  
    partition_autocreate_expression text DEFAULT NULL,  
    minimum_advance_partitions integer DEFAULT 2,  
    maximum_advance_partitions integer DEFAULT 5,  
    data_retention_period interval DEFAULT NULL,  
    managed_locally boolean DEFAULT false,  
    enabled boolean DEFAULT on);
```

パラメーター

- `relation` - テーブルの名前または OID。
- `partition_increment` - 次のパーティション作成までのインターバルまたはインクリメント。
- `partition_initial_lowerbound` - テーブルにパーティションがない場合、この下限と `partition_increment` 離れた上限を持つ最初のパーティションが作成されます。
- `partition_autocreate_expression` - 新しいパーティションを作成するかどうかを検出するために使用されます。
- `minimum_advance_partitions` - システムは常に少なくとも `minimum_advance_partitions` パーティションを持とうとします。
- `maximum_advance_partitions` - 事前パーティションの数が `minimum_advance_partitions` を下回ると、一度に作成されるパーティションの数。
- `data_retention_period` - 古いパーティションが削除されるまでのインターバル（定義されている場合）。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/一時停止し、後で再開/再度有効にします。

例

日単位のパーティション、1 か月間データを保持：

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

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

残りのパーティションが2つしかない場合は、5つのアドバンスパーティションを作成します（各パーティションは10億のオーダーを保持できます）。

```
bdr.autopartition(Orders, 1000000000,
    partition_initial_lowerbound := 0,
    minimum_advance_partitions := 2,
    maximum_advance_partitions := 5
);
```

1 つの AutoPartition を作成する

`bdr.autopartition_create_partition()` を使用して、親テーブルにスタンドアロンの AutoPartition を作成します。

概要

```
bdr.autopartition_create_partition(relname regclass,  
                                  partname name,  
                                  lowerb text,  
                                  upperb text,  
                                  nodes oid[]);
```

パラメーター

- `relname` - アタッチする親テーブルの名前または OID
- `partname` - 新しい AutoPartition の名前
- `lowerb` - パーティションの下限
- `upperb` - パーティションの上限
- `nodes` - 新しいパーティションが存在するノードのリスト

パーティションの自動作成の停止

`bdr.drop_autopartition()` を使用して、指定されたリレーションの自動パーティションルールを削除します。リレーションの保留中の作業項目はすべて削除され、新しい作業項目は作成されません。

```
bdr.drop_autopartition(relation regclass);
```

パラメーター

- `relation` - テーブルの名前または OID

AutoPartition を 1 つ削除する

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。パーティションテーブルが正常に削除された場合、このファンクションは `true` を結果ます。

概要

```
bdr.autopartition_drop_partition(relname regclass)
```

パラメーター

- `relname` - 削除するパーティションテーブルの名前

ノート

これにより、選択したパーティションテーブルで `DROP TABLE` を使用する前に、親テーブルに `DDL` ロックが設定されます。

パーティションの作成を待つ

`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。このファンクションは、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受け取り、その値を保持するパーティションが作成されるまで待機します。

このファンクションは、パーティションがローカルに作成されるのを待機するだけです。パーティションがリモートノードにも存在することは保証しません。

すべての BDR ノードにパーティションが作成されるのを待つには、オーダーファンクションを使用します。このファンクションは、ローカルとすべてのリモートノードを内部的にチェックし、パーティションがどこにでも作成されるまで待機します。

概要

```
bdr.autopartition_wait_for_partitions(relation regclass, text bound);
```


パラメーター

- `relation` - テーブルの名前または OID
- `bound` - パーティションキー列の値。

概要

```
bdr.autopartition_wait_for_partitions_on_all_nodes(relation regclass, text bound);
```

パラメーター

- `relation` - テーブルの名前または OID。
- `bound` - パーティションキー列の値。

パーティションを見つける

`bdr.autopartition_find_partition()` ファンクションを使用して、指定されたパーティションキー値のパーティションを見つけます。その値を保持するパーティションが存在しない場合、ファンクションは `NULL` を返します。それ以外の場合は、パーティションの `OID` が返されます。

概要

```
bdr.autopartition_find_partition(relname regclass, searchkey text);
```

パラメーター

- `relname` - パーティションテーブルの名前。
- `searchkey` - 検索するパーティションキー値。

自動パーティショニングを有効/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。自動パーティショニングが既に有効になっている場合、何もしません。同様に、`bdr.autopartition_disable()` を使用して、指定されたテーブルで `AutoPartitioning` を無効にします。

概要

```
bdr.autopartition_enable(relname regclass);
```

パラメーター

- `relname` - AutoPartitioning を有効にするリレーシヨンの名前。

概要

```
bdr.autopartition_disable(relname regclass);
```

パラメーター

- `relname` - AutoPartitioning を無効にするリレーシヨンの名前。

概要

```
bdr.autopartition_get_last_completed_workitem();
```

クラスター内のすべてのノードで正常に完了した最後の作業項目の `id` を返します。

AutoPartition ワーカーを確認する

`bdr.autopartition_work_queue_check_status` 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しているバックグラウンドワーカーのステータスを確認できます。

ワーカーは次のビューで見ることができます。`autopartition_work_queue_local_status`
`autopartition_work_queue_global_status`

概要

```
bdr.autopartition_work_queue_check_status(workid bigint
                                           local boolean DEFAULT false);
```

パラメーター

- `workid` - AutoPartition ワーカーのキー
- `local` - ローカルのステータスのみを確認する

注意事項

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も、常にバックグラウンドで実行されています。無効なワーカー ID が使用された場合、ファンクションは「unknown」を結果返します。「進行中」は一般的なステータスです。

2.19 Timestamp-Based Snapshots

タイムスタンプベースのスナップショットを使用すると、通常の 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。これは、共通のポイントインタイムで異なる BDR ノード上のデータにアクセスするために使用できます。例、データ品質チェックのためにマルチプルノードの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動作しません。

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` パラメータを介して有効になります。これは、タイムスタンプ値または現在のタイムスタンプ（現在）を表す特別な値「current」のいずれかを受け入れます。`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、次のクエリは 2018-12-08 02:28:30 GMT における `customers` テーブルの状態を結果返します。

```
SET snapshot_timestamp = 2018-12-08 02:28:30 GMT;  
SELECT count(*) FROM customers;
```

BDR がなければ、これは将来のタイムスタンプまたは上記の特別な「現在」値でのみ機能するため、履歴クエリには使用できません。

BDR は、マルチノード環境でその機能を使用して改善します。まず、BDR は、他のノードへのすべての接続が指定されたタイムスタンプより前にデータベースに追加された未処理のデータをレプリケートするため、タイムスタンプベースの `snapshot` がマルチマスターグループ全体で一貫しています。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、そのノードで最近の履歴に対してクエリを実行できる時間枠を指定します。

任意の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）は指定されたインターバルの最大 2 倍より新しいデッド行をクリーニング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、ユーザテーブルにより多くの膨張が残る可能性があります。最初は、一般的な設定として 10 秒をお勧めしますが、必要に応じて変更できます。

クエリは通常と同様に `bdr.timestamp_snapshot_keep` よりも長く実行されることに注意してください。

また、タイムスタンプ `snapshot` がどの程度保持されているかに関する情報はサーバーのリスタートリスタート後に PostgreSQL でないインスタンスにノートしてください。

`bdr.timestamp_snapshot_keep` と `postgres_fdw` 拡張機能を組み合わせて、BDR グループ内のマルチブルのノードで一貫した読み取りを取得できます。これを外部テーブルと組み合わせて使用すると、ノード全体で並列クエリを実行できます。

この機能を使用する場合、マルチノードクエリ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。

2.20 Replication Sets

レプリケーションセットは、BDR ノードがサブスクライブできるテーブルのグループです。レプリケーションセットを使用して、各ノードが他のノードの正確なコピーである通常の対称マルチマスターよりも複雑なレプリケーショントポロジを作成できます。

すべての BDR グループは、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します。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複製されます。

2.20.1 レプリケーションセットの使用

`create_replication_set()` を使用して、挿入、更新、削除、または切り捨てのアクションを含めるかどうかを指定して、追加のレプリケーションセットを作成できます。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テーブルの作成時に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とき、およびその後に維持されます。

ノードが参加したら、レプリケーションセットからテーブルを削除できますが、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 BDR 管理ファンクションコールをレプリケートしません。`replication_set_add_ddl_filter` を使用して、複製するコマンドを定義します。

BDR は、すべてのノードにレプリケーションセット定義を作成します。次に、`alter_node_replication_sets` を使用して、各レプリケーションセットをパブリッシュおよび/またはサブスクライブするように各ノードを定義できます。

これらの定義を後で変更したり、レプリケーションセットを削除したりするための Functions が存在します。

注釈: 選択的レプリケーションにデフォルトのレプリケーションセットを使用しません。デフォルトでは DDL レプリケーションおよび管理ファンクション呼び出しにも使用されるため、クラスター内の BDR ノードでデフォルトのレプリケーションセットを削除または変更しないでください。

2.20.2 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ています。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すると、パーティションを含む変更がダウストリームに複製されます。

注釈: パーティションがパーティションテーブルを介して複製される場合、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複製されます。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。

個々のパーティションをレプリケーションセットに追加することができます。その場合、それらは通常のテーブルのように（ダウストリームのパーティションと同じ名前のテーブルに）複製されます。プロバイダーとサブスクライバーの両方でパーティショニング定義が同じである場合、パーティショニングロジックを実行する必要があるため、これにはパフォーマンス上の利点があります。

注釈: ルートパーティションテーブルがレプリケーションセットのパートである場合、個々のパーティションのメンバーシップは無視され、上記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。

2.20.3 外部キーの動作

外部キー制約は、参照元のテーブルの行が参照被参照のテーブルの行と一致するようにします。したがって、参照するテーブルがレプリケーションセットのメンバである場合、被参照されるテーブルも同じレプリケーションセットのメンバである必要があります。

BDR の現在のバージョンは、この条件を自動的にチェックまたは適用しません。したがって、レプリケーションセットにテーブルを追加するときに、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ように make 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、つまり参照元のテーブルがレプリケーションセットのメンバであり、被参照先のテーブルがそうで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。

```
SELECT t1.relname,
       t1.nspname,
       fk.conname,
       t1.set_name
FROM bdr.tables AS t1
```

(次のページに続く)

(前のページからの続き)

```

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)

```

これは、テーブル `t2` はレプリケーションセット `s2` のメンバですが、外部キー `t2_x_fkey` によって被参照されるテーブルは異なることを意味します。

注釈: `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パートであるすべてのテーブルでカスケードのない `TRUNCATE` になります。

```

TRUNCATE table1, referencing_table1, referencing_table2 ...

```

2.20.4 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、次のファンクションは DDL とみなされるため、DDL レプリケーションとグローバルロックが現在アクティブな場合、それらに適用されることに注意してください。

DML および DDL レプリケーション を参照してください。

bdr.create_replication_set

このファンクションは、レプリケーションセットを作成します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.create_replication_set(set_name name,  
                           replicate_insert boolean DEFAULT true,  
                           replicate_update boolean DEFAULT true,  
                           replicate_delete boolean DEFAULT true,  
                           replicate_truncate boolean DEFAULT true,  
                           autoadd_tables boolean DEFAULT false,  
                           autoadd_existing boolean DEFAULT true)
```

パラメーター

- **set_name** - 新しいレプリケーションセットの名前。BDR グループ全体で一意である必要があります
- **replicate_insert** - このレプリケーションセット内のテーブルへの挿入をレプリケートするかどうかを示します
- **replicate_update** - このレプリケーションセット内のテーブルの更新をレプリケートするかどうかを示します
- **replicate_delete** - このレプリケーションセット内のテーブルからの削除をレプリケートするかどうかを示します
- **replicate_truncate** - このレプリケーションセット内のテーブルの切り捨てをレプリケートするかどうかを示します
- **autoadd_tables** - 新しく作成された（将来の）テーブルをこのレプリケーションセットに追加するかどうかを示します
- **autoadd_existing** - 既存のすべて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するかどうかを示します。これは **autoadd_tables** が **true** に設定されている場合にのみ効果があります

注意事項

デフォルトでは、新しいレプリケーションセットは DDL または BDR 管理ファンクションコールをレプリケートしません。レプリケーションセットの DDL レプリケーションを設定する方法については、以下の `ddl filters` を参照してください。デフォルトグループレプリケーションセットには、グループの作成時に作成されるすべての DDL および管理ファンクションコールを複製する既存の DDL フィルタが設定されていますが、BDR グループのデフォルトレプリケーションセットが DDL または BDR 管理ファンクションコール。

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 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で可視です。

`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。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.alter_replication_set(set_name name,
                           replicate_insert boolean DEFAULT NULL,
                           replicate_update boolean DEFAULT NULL,
                           replicate_delete boolean DEFAULT NULL,
                           replicate_truncate boolean DEFAULT NULL,
                           autoadd_tables boolean DEFAULT NULL)
```

パラメーター

- `set_name` - 既存のレプリケーションセットの名前
- `replicate_insert` - このレプリケーションセット内のテーブルへの挿入をレプリケートするかどうかを示します
- `replicate_update` - このレプリケーションセット内のテーブルの更新をレプリケートするかどうかを示します
- `replicate_delete` - このレプリケーションセット内のテーブルからの削除をレプリケートするかどうかを示します
- `replicate_truncate` - このレプリケーションセット内のテーブルの切り捨てをレプリケートするかどうかを示します

- `autoadd_tables` -新しく作成された（将来の）テーブルをこのレプリケーションセットに追加するかどうかを示します

NULL（デフォルト）に設定されているオプションは以前と同じままです。

注意事項

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 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で可視です。

`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。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.drop_replication_set(set_name name)
```

パラメーター

- `set_name` - 既存のレプリケーションセットの名前

注意事項

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 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で可視です。

警告: 少なくとも別のノードで使用されているレプリケーションセットを削除しないでください。そのノードでレプリケーションが停止します。これが発生した場合は、そのレプリケーションセットから影響を受けるノードのサブスクライブを解除してください。同じ理由から、進行中の結合オペレーションがあり、参加するノードがそのレプリケーションセットのメンバである場合は、レプリケーションセットを削除しないでください。レプリケーションセットのメンバーシップは、結合の開始時にのみチェックされます。これは、レプリケーションセットの使用に関する情報が各ノードにローカルであるため、グループに参加する前にノードで構成できるために発生します。

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の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、ノードがパブリッシュおよびサブスクライブするレプリケーションセットを変更します。

概要

```
bdr.alter_node_replication_sets(node_name name,
                                set_names text[])
```

パラメーター

- `node_name` - 変更するノード。現在はローカルノードである必要があります
- `set_names` - 指定されたノードに複製するレプリケーションセットの配列。空の配列はグループのデフォルトのレプリケーションセットを使用します

注意事項

このファンクションはローカルノードでのみ実行され、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は存在がチェックされません。エラーを回避するために、レプリケーションセット名を正しく指定してください。

これにより、BDR グループに含まれるノードだけでなく、まだグループにパート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、スキーマはレプリケーションセットの設定に関係なく*常に完全に同期され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。`bdr.tables` カタログテーブルを参照することで、不要なテーブルを削除できます。これらは、後のバージョンの BDR で自動的に削除される可能性があります。これは、

ddl filters

構成が DDL のレプリケーションを妨げる場合でも現在当てはまります。

この呼び出しの後にノードがサブスクライブするレプリケーションセットは、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に実際に変更をレプリケートするために、他のノード/によってパブリッシュされる必要があります。

2.20.5 レプリケーションセットのメンバーシップ

テーブルは、1 つ以上のレプリケーションセットに追加および削除できマルチプル。これは、これらのテーブルの変更のレプリケーションは、[DDL レプリケーションのフィルタリング](#) を参照)。

レプリケーションは、レプリケーションセットのテーブルメンバーシップをノードレプリケーションセット構成と組み合わせて使用して、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。この決定は、すべてのメンバーシップとユニオンセットオプションを使用して行われレプリケーションします。これは、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 と、UPDATE のみをレプリケートレプリケーションセット B のメンバである場合、ターゲットノードがレプリケーションセット A と B の両方にサブスクライブされている場合、INSERT と UPDATE の両方がレプリケートされることを意味します。

bdr.replication_set_add_table

このファンクションは、テーブルをレプリケーションセットに追加します。

これにより、レプリケーションセットにテーブルが追加され、この瞬間から変更のレプリケーションがスタートされます（つまり、トランザクションコミット）。テーブルがノード上に持つ可能性のある既存のデータは同期されません。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.replication_set_add_table(relation regclass,  
                             set_name name DEFAULT NULL,  
                             columns text[] DEFAULT NULL,  
                             row_filter text DEFAULT NULL)
```

パラメーター

- **relation** - テーブルの名前または OID
- **set_name** - 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のレプリケーションセットが使用されます
- **columns** - 将来の使用のために予約されています（現在は何もしません。NULL である必要があります）
- **row_filter** - レプリケートされた行のフィルタリングに使用される SQL 式。この式が定義されていない場合（つまり NULL - デフォルト）、すべての行が送信されます

row_filter は、NULL を使用してブール結果を生成する式を指定します。True または Unknown に評価される式は行を複製します。False 値は行を複製しません。式にサブクエリを含めることも、レプリケート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。システム列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 オーバーヘッドが発生しますが、フィルタリングされた行のデータの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、ターゲットノードに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除外できます。以前を参照してください。

テーブルの一部の列のみを複製することができます。 [違いのあるノード間の複製](#) を参照してください。

注意事項

このファンクションは、DDL ステートメントと同じ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、**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で可視です。

bdr.replication_set_remove_table

このファンクションは、レプリケーションセットからテーブルを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.replication_set_remove_table(relation regclass,  
                                set_name name DEFAULT NULL)
```

パラメーター

- **relation** - テーブルの名前または OID
- **set_name** - レプリケーションセットの名前。NULL（デフォルト）の場合、BDR グループのデフォルトのレプリケーションセットが使用されます

注意事項

このファンクションは、DDL ステートメントと同じ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 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で可視です。

2.20.6 レプリケーションセットのリスト

次のクエリーで既存のレプリケーションセットをリストできます。

```
SELECT set_name  
FROM bdr.replication_sets;
```

このクエリーを使用して、特定のレプリケーションセット内のすべてのテーブルをリストできます。

```
SELECT nspname, relname  
FROM bdr.tables  
WHERE set_name = myrepset;
```

セクション [外部キーの動作](#) では、被参照参照が同じレプリケーションセットに含まれていないすべての外部キーをリストするクエリーをレポートします。

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 node_id,  
       node_name,  
       COALESCE(  
           pub_repsets, pub_repsets  
       ) AS pub_repsets,
```

(次のページに続く)

(前のページからの続き)

```

COALESCE(
    sub_repsets, sub_repsets
) AS sub_repsets
FROM bdr.local_node_summary;

```

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、すべてのノードに関連付けられているレプリケーションセットを同時に取得するには、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($$
            SELECT
                node_id,
                node_name,
                COALESCE(
                    pub_repsets, pub_repsets
                ) AS pub_repsets,
                COALESCE(
                    sub_repsets, sub_repsets
                ) AS sub_repsets
            FROM bdr.local_node_summary;
        $$)::jsonb
    ) AS j
)
SELECT j->response->command_tuples->0->>node_id AS node_id,
       j->response->command_tuples->0->>node_name AS node_name,
       j->response->command_tuples->0->>pub_repsets AS pub_repsets,
       j->response->command_tuples->0->>sub_repsets AS sub_repsets
FROM node_repsets;

```

これにより、例次のように表示されます。

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)

2.20.7 DDL レプリケーションのフィルタリング

デフォルトでは、サポートされているすべての DDL のレプリケーションは、デフォルトの BDR グループ レプリケーションセットを介して発生します。これは、BDR グループと同じ名前の DDL フィルタが存在することによって実現されます。BDR グループは、BDR グループの作成時にデフォルトの BDR グループ レプリケーションセットに追加されます。

上記は、既存のすべてのレプリケーションセットの DDL レプリケーションフィルターを変更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とは無関係です。データの変更と同様に、各 DDL ステートメントは、マルチプルのレプリケーションセットのマルチプルのフィルターに一致するかどうかに関係なく、一度だけ複製されます。

次のクエリーを使用して、既存の DDL フィルタをリストできます。フィルタごとに、コマンドタグとロール名前に適用される正規表現を表示します。

```
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、DDL レプリケーションと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、DDL フィルタをレプリケーションセットに追加します。

指定されたフィルターに一致する DDL は、そのセットにサブスクライブ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。

これは任意のノードでの DDL の実行を妨げるものではないことに注意してください DDL が他のノードにレプリケートされるかどうかのみが変更されます。したがって、2つのノード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でも、各ノードに直接接続し、そのノードで目的の DDL を実行することにより、インデックスコマンドを自由に実行できます。

DDL フィルターは `command_tag` および `role_name` を指定して、一部の DDL ステートメントのみのレプリケーションを許可できます。`command_tag` は、通常の PostgreSQL コマンドの `EVENT TRIGGERs` で使用されるものと同じです。典型的な例は、ロジカルスタンバイでの追加のインデックスコマンドが他のすべてのノードにレプリケートされないようにするフィルターを作成することです。

使用する BDR 管理機能は、修飾ファンクション名前とマッチングするタグ名を使用してフィルタリングできます（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、すべての BDR 機能を `bdr.*` を使用して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` はどちら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。

概要

```
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のレプリケーションセットが使用されます
- **ddl_filter_name** -DDL フィルターの名前。これは BDR グループ全体で一意である必要があります
- **command_tag** -コマンドタグをマッチングさせるための正規表現。 NULL はすべてにマッチすることを意味します
- **role_name** -ロール名前をマッチングさせるための正規表現。 NULL はすべてのロールにマッチすることを意味します

注意事項

このファンクションは、DDL ステートメントと同じ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。ノート-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します。

このファンクションは 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で可視です。

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、ビュー `bdr.ddl_replication` を使用します。

例

BDR 管理機能のみを含めるには、次のようなフィルタを定義します。

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, mygroup_admin, $$bdr\..*$$);
```

インデックス DDL 以外をすべて除外するには：

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, index_filter,
                                         ^(?!(CREATE INDEX|DROP INDEX|ALTER INDEX)).*);
```

テーブルとインデックスに対するすべてのオペレーションを含め、他のすべてのオペレーションを除外するには、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、マルチプルのフィルターが許可されたすべての 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 フィルタを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定自分自身を含む、DDL レプリケーション構成の影響を受けます。

概要

```
bdr.replication_set_remove_ddl_filter(set_name name,
                                     ddl_filter_name text)
```

パラメーター

- **set_name** -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のレプリケーションセットが使用されます
- **ddl_filter_name** - 削除する DDL フィルターの名前

注意事項

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ノートしてください。

このファンクションは 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で可視です。

2.21 Stream Triggers

BDR には、ダウストリーム/ターゲットノードでの追加のデータ処理に使用できる新しいタイプのトリガーが導入されています。

- コンフリクトトリガー
- 変身トリガー

これらのタイプのトリガーは、ストリームトリガーと呼ばれます。

ストリームトリガーは、構文がトリガーのように設計されており、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、単に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ために、`CREATE TRIGGER` コマンドを使用する必要はありません。代わりに、ストリームトリガーは特別な BDR 関数 `bdr.create_conflict_trigger()` および `bdr.create_transform_trigger()` を使用して作成されます。

作成されると、トリガーはカタログ表 `pg_trigger` に可視されます。ストリームトリガーは `tgisinternal = true` および `tgenabled = 'D'` としてマークされ、名前サフィックスが `'_bdrc'` または `'_bdrt'` になります。ビュー `bdr.triggers` は、テーブルにリレーションするトリガー、実行されているプロシージャの名前、それをトリガーするイベント、およびトリガータイプに関する情報を提供します。

したがって、ストリームトリガーは通常の SQL 処理では有効になっていないことに注意してください。このため、`ALTER TABLE ... ENABLE TRIGGER` は、その特定の名称のバリエーションと ALL バリエーションの両方のストリームトリガーに対してブロックされ、トリガーが通常の SQL トリガーとして実行されません。

これらのトリガーはダウ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、オリジンでの `row_filter` 式の使用を検討することをお勧めします。

また、ストリームトリガーの実行中に適用される DML は他の BDR ノードに複製されず、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に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、例を示します。

2.21.1 Apply 中に実行をトリガー

変換トリガーは、トリガーテーブル内の着信変更ごとに 1 回、最初に実行されます。これらのトリガーは、マッチングターゲット行を見つけようとする前に起動されるため、非常にレンジ変換を効率的かつ一貫して適用できます。

次に、UPDATE および DELETE の変更については、ターゲット行を見つけます。ターゲット行がない場合、それらの変更タイプに対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。

行方不明の列の競合の解決

変換トリガーが実行される前に、PostgreSQL は着信タプルをターゲットテーブルの行タイプとマッチさせようとします。

入力行に存在するがターゲットテーブルにない列は、タイプ `target_column_missing` の競合をトリガーします。逆に、ターゲットテーブルに存在するが受信行にない列は、`source_column_missing` 競合をトリガーします。これら2つの競合タイプのデフォルトの解決は、それぞれ `ignore_if_null` と `use_default_value` です。

これは、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ます。インスタンス、スキーマの新しいバージョンで新しい列が導入された場合。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、ソース列がありません。新しく導入された列にデフォルト値が設定されるため、`use_default_value` 戦略が適切です。

ただし、新しいスキーマバージョンのノードから古いスキーマバージョンのノードにレプリケーションする場合、ターゲットテーブルに列がなく、`ignore_if_null` リゾルバはローリングアップグレードには適していませんユーザは、アップグレードされたノードのいずれかで、新しい列に NULL 以外の値を持つタプルを挿入します。

この例をビューすると、スキーマのローリングアップグレードの適切な設定は、`target_column_missing` の競合が発生した場合に `ignore` リゾルバ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、次のクエリーで実行されノード。

```
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。

次の例を考えます。テーブル `t` はノード 1 および 2 に存在しますが、その列 `col` はノード 1 にのみ存在します。

競合リゾルバが `ignore` に設定されている場合、`c` が null ではない行がノード 1 に存在する可能性があります。その行はノード 2 に複製され、列 `c` の値は破棄されます（例、`(pk=1)`）。

`c` 列がノード 2 のテーブルに追加されると、それは最初に既存のすべての行で NULL に設定され、上記の行は `(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、クラスターは発散する。

ノード 2 に複製される行には `col=NULL` があるため、デフォルトの `ignore_if_null` リゾルバ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。

この例に基づいて、`ignore` リゾルバが使用されたスキーマのローリングアップグレードの最後にクラスター全体に対して `LiveCompare` を実行して、相違が検出されて修正される `make` を確認することをお勧めします。

2.21.2 行型の用語

このドキュメントでは、次の行タイプを使用します。

- `SOURCE_OLD` は更新前の行、つまりキーです。
- `SOURCE_NEW` は、別のノードからの新しい行です。
- `TARGET` は、ノードに既に存在する行、つまり競合して行です。

2.21.3 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` 行を返した場合、アクションはスキップされます。

明確にするために、トリガーが `DELETE` に対して呼び出された場合、`DELETE` をスキップする場合、トリガーは `NULL` を結果必要があります。 `DELETE` を続行したい場合は、行の値を結果ます- `SOURCE_OLD` または `TARGET` が機能します。競合するオペレーションが `INSERT` または `UPDATE` であり、選択された解決が競合する行の削除である場合、トリガーは明示的に削除を実行して `NULL` を結果必要があります。トリガーファンクションは、必要に応じて他の `SQL` アクションを実行できますが、これらのアクションはローカルにのみ適用され、レプリケートされません。

2つ以上のノード間で実際のデータの競合が発生すると、2つ以上の変更が同時に発生します。これらの変更を適用すると、競合の解決は各ノードで個別に発生します。これは、競合の解決が各ノードで1回発生し、その間に大きな時間差があることを意味します。その結果、競合トリガーのマルチプルの実行間で通信が行われる可能性はありません。トリガーが関連するすべてのイベントに対してまったく同じ結果を与えることを保証するのは、競合トリガーの作成者の責任です。テクニカルサポートは、BDR に付属の `isolationtester` ツールを使用して、すべての競合トリガーを正式にテストすることをお勧めします。

警告:

- 単一のテーブルに複数の競合トリガーを指定できますが、個別のイベントと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のみと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お勧めしません。一貫性のない動作を引き起こす可能性があり、将来のリリースでは禁止されます。

同じ競合トリガーが複数のイベントと一致する場合、トリガー内で `TG_OP` 変数を使用して、競合を生成したオペレーションを特定できます。

デフォルトでは、BDR 行のレプリケーションオリジンの変更を監視することにより競合を検出するため、変更が 1 つだけ発生する場合でも競合トリガーが呼び出される可能性があります。この場合、実際の競合はないため、この競合検出メカニズムは誤検知の競合を生成する可能性があると言えます。前述のように、競合トリガーはこれらすべてを同じようにハンドルする必要があります。

場合によっては、タイムスタンプの競合検出で競合がまったく検出されないことに注意してください。例、DELETE が UPDATE の直後に発生する同時 UPDATE/DELETE では、最初に UPDATE を参照し、次に DELETE を参照するノードは競合を参照しません。競合が確認されない場合、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、競合が発生し、競合トリガーで処理できます。

トリガーファンクションは、オペレーションの種類に応じて、追加の状態情報と競合に関係するデータ行にアクセスします。

- INSERT では、競合トリガーはソースから SOURCE_NEW 行にアクセスできます
- UPDATE では、競合トリガーはソースから SOURCE_OLD および SOURCE_NEW 行および TARGET 行にアクセスできます
- DELETE では、競合トリガーはソースから SOURCE_OLD 行にアクセスできます

`bdr.trigger_get_row()` ファンクションを使用して、SOURCE_OLD、SOURCE_NEW、または TARGET 行を取得できます（そのオペレーションの値が存在する場合）。

競合トリガーへの変更はトランザクションで発生し、ALTER TABLE の一部のバリエーションが処理される方法と同様に、構成変更のレプリケーション中にグローバル DML ロックによって保護されます。

コンフリクトトリガー内でプライマリーキーを更新すると、実行タイミングの違いにより一意性制約違反エラーになる場合があります。したがって、ユーザーは競合トリガー内でプライマリーキーを更新しないでください。

2.21.4 トリガーを変換する

これらのトリガーは、特定のテーブルに対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き、競合トリガーに似ています。結果値と公開された変数の動作は似ていますが、変換トリガーはターゲット行が識別される前に実行されるため、TARGET 行はありません。

必要に応じて、BDR の各テーブルに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処理し、必要に応じて追加の操作を実行できます。任意の列の値を変更したり、それらを NULL に設定したりできます。結果値によって、さらに実行されるアクションが決まります。

- トリガーファンクションが行を返す場合、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、これ以上のアクションはなく、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、必要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 行を取得できます。このタイプのトリガーは、そのようなターゲット行が識別される前に実行されるため、TARGET 行は使用できません。

変換トリガーは、通常の BEFORE 行トリガーと非常によく似ていますが、次の重要な違いがあります。

- 変換トリガーは、着信変更ごとに呼び出されます。テーブルでマッチング行が見つからない場合、UPDATE および DELETE の変更に対して BEFORE トリガーは呼び出されません。
- 変換トリガーは、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呼び出されます。
- 変換トリガーは、通常の SQL トリガーでは使用できない SOURCE_OLD を介してロックアップキーにアクセスします。

2.21.5 ストリームトリガー変数

競合トリガーと変換トリガーは、トリガー API が提供する事前定義された変数と BDR が提供する追加情報関数を介して、行とメタデータに関する情報にアクセスします。

PL/pgSQL には、次の事前定義された変数が存在します。

TG_NAME

データ型名前; 実際に起動されたトリガーの名前を含む変数。トリガーの作成中に指定された名前と比較して、実際のトリガー名前には「_bdrt」または「_bdrc」サフィックス（トリガータイプによって異なります）が付いていることに注意してください。

TG_WHEN

データ型テキスト。これは、Conflict と Transform の両方のトリガーに対して BEFORE と表示します。ストリームトリガータイプは、`bdr.trigger_get_type()` 情報ファンクション（下記参照）を呼び出すことで取得できます。

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; トリガーを呼び出したテーブルのオブジェクト ID。

TG_TABLE_NAME

データ型名前; トリガーを呼び出したテーブルの名前。

TG_TABLE_SCHEMA

データ型名前; トリガーの呼び出しの原因となったテーブルのスキーマの名前。テーブルパーティションの場合、これはルートテーブルの名前です。

TG_NARGS

データ型整数; `bdr.create_conflict_trigger()` または `bdr.create_transform_trigger()` ステートメントでトリガー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 TG_NARGS 以上) は NULL 値になります。

2.21.6 情報機能**`bdr.trigger_get_row`**

このファンクションは、識別子で指定されたトリガー行の内容を RECORD として戻します。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。

概要

```
bdr.trigger_get_row(row_id text)
```

パラメーター

- **row_id**-行の識別子。トリガーのタイプとオペレーションに応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかです（個々のトリガータイプの文書を参照）。

bdr.trigger_get_committs

このファンクションは、識別子で指定されたトリガー行のコミット・タイムスタンプを返します。行がフリーズされているか、行が使用できないために使用できない場合、これは NULL を結果ます。行識別子 SOURCE_OLD には常に NULL を返します。

概要

```
bdr.trigger_get_committs(row_id text)
```

パラメーター

- **row_id**-行の識別子。トリガーのタイプとオペレーションに応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかです（個々のトリガータイプの文書を参照）。

bdr.trigger_get_xid

このファンクションは、識別子で指定された TARGET 行のローカルトランザクション ID を返します。行がフリーズされているか、行が使用できないために使用できない場合、これは NULL を結果ます。SOURCE_OLD および SOURCE_NEW の行識別子には常に NULL を返します。

これは、競合トリガーでのみ使用できます。

概要

```
bdr.trigger_get_xid(row_id text)
```

パラメーター

- `row_id` - 行の識別子。トリガーのタイプとオペレーションに応じて、`SOURCE_NEW`、`SOURCE_OLD`、および `TARGET` のいずれかです（個々のトリガータイプの文書を参照）。

`bdr.trigger_get_type`

このファンクションは、現在のトリガータイプを返します。これは、`CONFLICT` または `TRANSFORM` のいずれかです。ストリームトリガーの外部で呼び出された場合は `null` を返します。

概要

```
bdr.trigger_get_type()
```

`bdr.trigger_get_conflict_type`

このファンクションは、競合トリガー内で呼び出された場合は現在の競合タイプを返し、それ以外の場合は `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、[競合タイプのリスト](#) を参照してください。

概要

```
bdr.trigger_get_conflict_type()
```

`bdr.trigger_get_origin_node_id`

このファンクションは、引数として渡されたトリガー `row_id` のオリジンに対応するノード ID を返します。オリジンが有効でない場合（行がローカルで発生したことを意味します）、トリガーの行引数に応じて、ソースまたはターゲットノードのノード ID を結果ます。行識別子 `SOURCE_OLD` には常に `NULL` を返します。これは、常に信頼できるソースノードを優先するように競合トリガーを定義するために使用できます。以下の例を参照してください。

概要

```
bdr.trigger_get_origin_node_id(row_id text)
```

パラメーター

- `row_id`-行の識別子。トリガーのタイプとオペレーションに応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかです（個々のトリガータイプの文書を参照）。

bdr.ri_fkey_on_del_trigger

BEFORE トリガーとして呼び出された場合、このファンクションは FOREIGN KEY 情報を使用して FK の異常を回避します。

概要

```
bdr.ri_fkey_on_del_trigger()
```

2.21.7 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 REPLICA IDENTITY 設定、およびターゲットテーブルの内容によって異なります。

TARGET 行は、競合トリガーでのみ使用できます。TARGET 行には、ターゲットテーブルで UPDATE または DELETE を適用したときに行が見つかった場合にのみデータが含まれます。行が見つからない場合、TARGET は NULL になります。

2.21.8 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 - ターゲットの受信行ごとに 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

2.21.9 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、REPLICA IDENTITY FULL または TOAST able 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。

概要

```
bdr.create_conflict_trigger(trigger_name text,  
                            events text[],  
                            relation regclass,  
                            function regprocedure,  
                            args text[] DEFAULT {})
```

パラメーター

- **trigger_name** - 新しいトリガーの名前
- **events** - このトリガーを起動するイベントの配列。有効な値は「INSERT」、「UPDATE」および「DELETE」です
- **relation** - このトリガーを起動するリレーション
- **function** - どのファンクションを実行か
- **args** - オプショナル; トリガーファンクションが実行時に受け取るパラメーターの配列を指定します (TG_ARGV 変数の内容)

注意事項

このファンクションは、DDL ステートメントと同じ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。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で可視です。

通常の PostgreSQL トリガーと同様に、**bdr.create_conflict_trigger** ファンクションには、**relation** に対する TRIGGER 権限およびファンクションに対する EXECUTE 権限が必要です。これは、30619 以上の **bdr.backwards_compatibility** に適用されます。追加のセキュリティルールは、競合トリガーを含むすべてのトリガーに BDR で適用されます。

security 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。

概要

```
bdr.create_transform_trigger(trigger_name text,
                             events text[],
                             relation regclass,
                             function regprocedure,
                             args text[] DEFAULT {})
```

パラメーター

- **trigger_name** - 新しいトリガーの名前
- **events** - このトリガーを起動するイベントの配列、有効な値は「INSERT」、「UPDATE」および「DELETE」
- **relation** - このトリガーを起動するリレーション
- **function** - どのファンクションを実行か
- **args** - オプション、トリガーファンクションが実行時に受け取るパラメーターの配列を指定します (TG_ARGV 変数の内容)

注意事項

このファンクションは、DDL ステートメントと同じ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。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で可視です。

通常の PostgreSQL トリガーと同様に、権限ファンクションには、**relation** および **EXECUTE** 権限が必要ファンクション。追加のセキュリティルールは、BDR で変換トリガーを含むすべてのトリガーに適用されます。

security 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。

概要

```
bdr.drop_trigger(trigger_name text,  
                 relation regclass,  
                 ifexists boolean DEFAULT false)
```

パラメーター

- `trigger_name` - 既存のトリガーの名前
- `relation` - どのリレーションが定義されたトリガーです
- `ifexists` - `true` に設定すると、このコマンドは欠落しているトリガーを無視します

注意事項

このファンクションは、DDL ステートメントと同じ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。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で可視です。

`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。

2.21.10 ストリームトリガーの例

`update_if_newer` 競合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;
```

(次のページに続く)

(前のページからの続き)

```

    END IF;
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 timestamptz;
BEGIN
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    COMMITTS := bdr.trigger_get_committs(SOURCE_NEW);

    IF (TG_OP = INSERT) THEN
        INSERT INTO log SELECT I, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = UPDATE) THEN

```

(次のページに続く)

(前のページからの続き)

```

        INSERT INTO log SELECT U, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = DELETE) THEN
        INSERT INTO log SELECT D, COMMITTS, row_to_json(SOURCE_OLD);
    END IF;

    RETURN NULL; -- do not apply the change
END;
$$;

```

以下の例は、信頼できるソースの競合検出を実装する競合ノード。これは、`bdr.trigger_get_origin_node_id()` フังก์ションを使用して、3つ以上のノードで動作するソリューションを提供します。

```

CREATE OR REPLACE FUNCTION test_conflict_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE record;
    TARGET record;

    TRUSTED_NODE    bigint;
    SOURCE_NODE     bigint;
    TARGET_NODE     bigint;
BEGIN
    TARGET := bdr.trigger_get_row(TARGET);
    IF (TG_OP = DELETE)
        SOURCE := bdr.trigger_get_row(SOURCE_OLD);
    ELSE
        SOURCE := bdr.trigger_get_row(SOURCE_NEW);
    END IF;

    TRUSTED_NODE := current_setting(customer.trusted_node_id);

    SOURCE_NODE := bdr.trigger_get_origin_node_id(SOURCE_NEW);
    TARGET_NODE := bdr.trigger_get_origin_node_id(TARGET);

    IF (TRUSTED_NODE = SOURCE_NODE) THEN
        RETURN SOURCE;
    ELSIF (TRUSTED_NODE = TARGET_NODE) THEN
        RETURN TARGET;
    ELSE
        RETURN NULL; -- do not apply the change
    END IF;
END;
$$;

```

2.22 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。

[Distributed Transaction Processing: The XA Specification](#) を参照してください。

X/Open Distributed Transaction Processing (DTP) モデルは、3つのソフトウェアコンポーネントを想定しています。

- トランザクションの境界を定義し、トランザクションを構成するアクションを指定するアプリケーションプログラム (AP)。
- 共有リソースへのアクセスを提供するリソースマネージャー (データベースやファイルアクセスシステムなどの RM)。
- トランザクションに識別子を割り当て、その進行状況を監視し、トランザクションの完了と障害リカバリを担当するトランザクションマネージャ (TM) と呼ばれる別個のコンポーネント。

BDR は、PREPARE TRANSACTION および COMMIT PREPARED/ROLLBACK PREPARED コマンドを使用した明示的な外部 2PC をサポートしています。外部的には、BDR クラスタは、単一のセッションの Transaction Manager にとって単一の Resource Manager に見えます。

`bdr.commit_scope` が `local` の場合、トランザクションはローカルノードでのみ準備されます。コミットされると、変更がレプリケートされ、BDR はコミット後の競合解決を適用します。

`bdr.commit_scope` を `local` に設定することは、明示的な二相コミットでは無意味に見えるかもしれませんが、ユーザがトランザクションのレイテンシと堅牢性間のトレードオフを制御できるようにするオプションが提供されています。

明示的な二相コミットは、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。

2.22.1 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です。

```
PREPARE TRANSACTION some-global-id;
```

最初のフェーズが成功すると、すべてのノードが変更を適用し、トランザクションをコミットする準備が整います。次に、クライアントは同じノードから 2 番目のフェーズを呼び出す必要があります。

```
COMMIT PREPARED some-global-id;
```

2.23 Catalogs and views

ここでは、カタログとビューをアルファベット順に示しています。

2.23.1 ユーザーに表示されるカタログとビュー

`bdr.conflict_history`

このテーブルは、競合がログに記録されるデフォルトテーブル。テーブルは列 `local_time` で RANGE パーティション化されており、自動パーティションによって管理されます。デフォルトのデータ保持期間は 30 日です。

このテーブルへのアクセスは、所有するテーブルのすべての競合を表示できるテーブルの所有者が可能で、行単位セキュリティによって制限されます。

詳細については、[テーブルへの競合のログ](#) を参照してください。

`bdr.conflict_history` 列

Name	Type	Description
<code>sub_id</code>	<code>oid</code>	この競合を引き起こしたサブスクリプション。 `bdr.subscription` テーブルに結合できます
<code>local_xid</code>	<code>xid</code>	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
<code>local_lsn</code>	<code>pg_lsn</code>	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
<code>local_time</code>	timestamp with time zone	競合の現地時間
<code>remote_xid</code>	<code>xid</code>	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
<code>remote_commit_lsn</code>	<code>pg_lsn</code>	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
<code>remote_commit_time</code>	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
<code>conflict_type</code>	text	検出された競合の種類
<code>conflict_resolution</code>	text	選択された競合の解決

次のページに続く

表 14 – 前のページからの続き

Name	Type	Description
conflict_index	regclass	競合インデックス（インデックスが削除されていない場合にのみ有効）
reloid	oid	競合するリレーション（インデックスが削除されていない場合にのみ有効）
nspname	text	競合時に競合が発生したリレーションのスキーマの名前（名前変更に従いません）
relname	text	競合時に競合が発生したリレーションの名前（名前変更に従いません）
key_tuple	json	行のマッチングに使用されるキーの Json 表現形式
remote_tuple	json	着信する競合する行の 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	ローカルで競合する行を生成したトランザクション（ <i>local_tuple</i> が設定されており、行がフリーズされていない場合）
local_tuple_node_id	oid	ローカルの競合する行を生成したノード（ <i>local_tuple</i> が設定されており、行がフリーズされていない場合）
local_tuple_commit_time	timestamp with time zone	ローカルで競合している行の最新の変更のタイムスタンプ（ <i>local_tuple</i> が設定されており、行がフリーズされていない場合）

bdr.conflict_history_summary

行の競合に関するユーザーが判読可能な詳細を含むビュー。

bdr.conflict_history_summary 列

Name	Type	Description
nspname	text	スキーマの名前
relname	text	テーブルの名前
local_time	timestamp with time zone	競合の現地時間
local_tuple_commit_time	timestamp with time zone	ローカルコミットの時間
remote_commit_time	timestamp with time zone	リモートコミットの時間
conflict_type	text	競合の種類
conflict_resolution	text	採択された決議

`bdr.consensus_kv_data`

`bdr.consensus_kv_store()` および `bdr.consensus_kv_fetch()` インターフェースによって使用される Raft ベースの内部 KV ストア用の永続ストレージ。

`bdr.consensus_kv_data` 列

Name	Type	Description
<code>kv_key</code>	text	ユニークキー
<code>kv_val</code>	json	json フォーマットの任意値
<code>kv_create_ts</code>	timestampz	最終書き込みタイムスタンプ
<code>kv_ttl</code>	int	ミリ秒単位の値の存続時間
<code>kv_expire_ts</code>	timestampz	有効期限のタイムスタンプ (<code>kv_create_ts + kv_ttl</code>)

`bdr.camo_decision_journal`

`bdr.logical_transaction_status` が呼び出された場合、フェイルオーバー後に CAMO パートナーノードによって解決された決定の永続的なジャーナル。`bdr.node_pre_commit` とは異なり、これは通常の動作条件（つまり、CAMO ペアの両方のノードが実行されて接続されている）で処理されるトランザクションをカバーしません。このジャーナルのエントリは、自動的にクリーンアップされることはありません。これは、システムが依存しない診断ツール。

`bdr.camo_decision_journal` 列

Name	Type	Description
<code>origin_node_id</code>	oid	トランザクションが実行されたノードの OID
<code>origin_xid</code>	oid	リモートオリジンノードのトランザクション ID
<code>decision</code>	char	コミットの場合は「c」、アボートの場合は「a」
<code>decision_ts</code>	timestampz	デシジョンタイム

`bdr.crdt_handlers`

この表は、すべての CRDT データ型のマージ（「ハンドラー」）関数をリストしています。

bdr.crdt_handlers 列

Name	Type	Description
crdt_type_id	regtype	CRDT データタイプ ID
crdt_merge_id	regproc	このデータタイプのマージファンクション

bdr.ddl_replication

このビューには、現在の DDL filters によって設定された DDL レプリケーション構成がリストされます。

bdr.ddl_replication 列

Name	Type	Description
set_ddl_name	name	DDL フィルターの名前
set_ddl_tag	text	適用されるコマンドタグ（正規表現）
set_ddl_role	text	適用されるロール（正規表現）
set_name	name	このフィルターが定義されているレプリケーションセットの名前

bdr.depend

このテーブルは、BDR カタログ内の内部オブジェクト依存関係を追跡します。

bdr.global_consensus_journal

このカタログテーブルは、グローバルコンセンサスの管理中に送信されたすべての Raft メッセージをログに記録します。

bdr.global_consensus_response_journal カタログの場合、ペイロードはバイナリエンコードされた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、

:ref:`bdr.global\_consensus\_journal\_details` <*bdr.global_consensus_journal_details*>` ビューを参照してください。

bdr.global_consensus_journal 列

Name	Type	Description
log_index	int8	エントリーの ID
term	int8	ラフトターム
origin	oid	リクエストが発生したノードの ID
req_id	int8	リクエストの ID
req_payload	bytea	リクエストのペイロード
trace_context	bytea	要求のトレースコンテキスト

bdr.global_consensus_journal_details

このビューは、`bdr.decode_message_payload()` ファンクションを使用してペイロードをデコードし、送信された Raft メッセージと対応する応答を表示します。

bdr.global_consensus_journal_details 列

Name	Type	Description
log_index	int8	エントリーの ID
term	int8	ラフトターム
request_id	int8	リクエストの ID
origin_id	oid	リクエストが発生したノードの ID
req_payload	bytea	リクエストのペイロード
origin_node_name	name	要求が発生したノードの名前
message_type_no	oid	リクエストの BDR メッセージタイプの ID
message_type	text	要求の BDR メッセージタイプの名前
message_payload	text	リクエストの BDR メッセージペイロード
re-sponse_message_type_no	oid	応答の BDR メッセージタイプの ID
re-sponse_message_type	text	応答の BDR メッセージタイプの名前
response_payload	text	レスポンスの BDR メッセージペイロード
response_errcode_no	text	応答の SQLSTATE
response_errcode	text	レスポンスのエラーコード
response_message	text	レスポンスのエラーメッセージ

bdr.global_consensus_response_journal

このカタログテーブルは、グローバルコンセンサスの管理中に受信した Raft メッセージへのすべての応答を収集します。

bdr.global_consensus_journal カatalogの場合、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、

:ref:`bdr.global\_consensus\_journal\_details` <*bdr.global_consensus_journal_details*>` ビューを参照してください。

bdr.global_consensus_response_journal 列

Name	Type	Description
log_index	int8	エントリーの ID
res_status	oid	レスポンスのステータスコード
res_payload	bytea	レスポンスのペイロード
trace_context	bytea	応答のトレースコンテキスト

bdr.global_lock

このカタログテーブルには、サーバーの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。

bdr.global_lock の可視行は必ずしもすべてのグローバルロックアクティビティを反映するとは限らないため、使用状況をモニタリングには、

:ref:`bdr.global\_locks` <*bdr.global_locks*>` ビューをお勧めします。

このテーブルの内容を変更しないでください。重要な BDR カタログです。

bdr.global_lock 列

Name	Type	Description
ddl_epoch	int8	ロックの DDL エポック
origin_node_id	oid	グローバルロックが発生したノードの OID
lock_type	oid	ロックの種類 (DDL または DML)
nspname	name	ロックされたリレーションのスキーマ名前
relname	name	ロックされたリレーションのリレーション名前

次のページに続く

表 23 – 前のページからの続き

Name	Type	Description
groupid	oid	トップレベルグループの OID (アドバイザリロックの場合)
key1	integer	最初の 32 ビットキーまたは 64 ビットキーのオーダー 32 ビット (勧告ロック用)
key2	integer	2 番目の 32 ビットキーまたは 64 オーダービットキーの 32 ビット (勧告ロック用)
key_is_bigint	boolean	64 ビット整数キーが使用されている場合は true (勧告ロック用)

bdr.global_locks

このノードのアクティブなグローバルロックを含むビュー。 `:ref:`bdr.global_locks`<bdr.global_locks>` ビューは、BDR の共有メモリロック状態の追跡を公開するため、管理者は BDR のグローバルロックアクティビティと進行状況をより深く理解できます。

グローバルロックの詳細については、`Monitoring global locks` を参照してください。

bdr.global_locks 列

Name	Type	Description
<i>origin_node_id</i>	oid	グローバルロックが発生したノードの OID
<i>origin_node_name</i>	name	グローバルロックが発生したノードの名前
<i>lock_type</i>	text	ロックの種類 (DDL または DML)
<i>relation</i>	text	ロックされたリレーション名前 (DML ロックの場合) またはキー (勧告ロックの場合)
<i>pid</i>	int4	ロックを保持しているプロセスの PID
<i>acquire_stage</i>	text	ロック取得プロセスの内部状態
<i>waiters</i>	int4	同じグローバルロックを待っているバックエンドのリスト
<i>global_lock_request_time</i>	timestampz	このグローバルロックの取得がオリジンノードによって開始された時刻
<i>local_lock_request_time</i>	timestampz	ローカルノードがローカルロックの取得を開始した時刻
<i>last_state_change_time</i>	timestampz	<code>`acquire_stage`</code> の最終変更時刻

列の詳細：

- **relation** : DML ロックの場合、**relation** は DML ロックを取得したリレーションを示します。グローバル勧告ロックの場合、**relation** 列には、実際にロックが取得された 2 つの 32 ビット整数または 1 つの 64 ビット整数が表示されます。

- **origin_node_id** および **origin_node_name** : これらがローカルノードの ID および名前と同じである場合、ローカルノードはグローバル DDL ロックのイニシエータ、つまり、取得するトランザクションを実行しているノードです。これらのフィールドが別のノードを指定している場合、ローカルノードは代わりにローカル DDL ロックを取得して、リモートノードからのグローバル DDL ロック要求に応じようとしています。
- **pid** : ローカルノードが要求側ノードの場合、グローバル DDL ロックを要求したプロセスのプロセス ID。他のノードではヌル。オリジンノードを照会して、ロッカーの pid を特定します。
- **global_lock_request_time** : グローバルロック要求イニシエーターがグローバルロックの取得プロセスを開始したタイムスタンプ。現在のノードで不明な場合は null になります。この時間は DDL ロック要求の先頭にスタンプされ、DDL エポック管理と必要なレプリケーション保留キューのフラッシュにかかった時間が含まれます。現在、オリジンノードでのみ知られています。
- **local_lock_request_time** : ローカルノードがこのグローバルロックのローカルロックの取得を開始したタイムスタンプ。これには、重量セッションロックの取得にかかった時間が含まれますが、DDL エポックまたはキューのフラッシュにかかった時間は含まれません。ローカルノードリスタート後にロックを再取得するとノードリスタート時間になります。
- **last_state_change_time** : このグローバルロックエントリーの **bdr.global_locks.acquire_stage** フィールドが最後に変更されたタイムスタンプ。

bdr.local_consensus_snapshot

このカタログテーブルには、ローカルノードによって作成または受信されたコンセンサススナップショットが含まれています。

bdr.local_consensus_snapshot 列

Name	Type	Description
log_index	int8	エントリーの ID
log_term	int8	ラフトターム
snapshot	bytea	raftsnapshot データ

bdr.local_consensus_state

このカタログテーブルは、ローカルノード上の Raft の現在の状態を保存します。

bdr.local_consensus_state 列

Name	Type	Description
node_id	oid	ノードの ID
current_term	int8	ラフトターム
apply_index	int8	ラフトアプライインデックス
voted_for	oid	今期にこのノードからの投票
last_known_leader	oid	最後に認識されたラフトリーダーの node_id

bdr.local_node

このテーブルは、現在の Postgres インスタンスの現在のデータベース内のローカルノードを識別します。

bdr.local_node 列

Name	Type	Description
node_id	oid	ノードの ID
pub_repsets	text[]	パブリッシュされた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

bdr.local_node_summary

:ref:`bdr.node_summary``<bdr.node_summary>` と同じ情報を含むがローカルノードのみのビュー。

bdr.local_sync_status

サブスクリプションまたはテーブル同期プロセスのステータスに関する情報。

bdr.local_sync_status 列

Name	Type	Description
sync_kind	char	実行される同期の種類
sync_subid	oid	同期を行うサブスクリプションの ID
sync_nspname	name	同期されたテーブルのスキーマ名前（ある場合）
sync_relname	name	同期されたテーブルの名前（ある場合）
sync_status	char	同期の現在の状態

[次のページに続く](#)

表 28 – 前のページからの続き

Name	Type	Description
sync_remote_relid	oid	アップストリームに同期されたテーブル（ある場合）の ID
sync_end_lsn	pg_lsn	同期状態が最後に変化した位置

bdr.network_path_info

ノードの場所間のネットワークコストに関するユーザー定義の情報を保存するカタログビュー。

bdr.network_path_info 列

Name	Type	Description
node_group_name	name	BDR グループの名前
node_region1	text	ノードリージョン名前、bdr.node_location から
node_region2	text	ノードリージョン名前、bdr.node_location から
node_location1	text	ノードの場所の名前、bdr.node_location から
node_location2	text	ノードの場所の名前、bdr.node_location から
network_cost	numeric	ノードの場所の名前、bdr.node_location から

bdr.node

このテーブルには、クラスタ内のすべての BDR ノードがリストされています。

bdr.node 列

Name	Type	Description
node_id	oid	ノードの ID
node_name	name	ノードの名前
node_group_id	oid	ノードグループの ID
source_node_id	oid	ソースノードの ID
synchronize_structure	"char"	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状態（結合または昇格中）
seq_id	int4	一意のシーケンス番号を生成するために使用されるノードのシーケンス識別子
dbname	name	ノードのデータベース名前
node_dsn	char	ノードの接続文字列

次のページに続く

表 30 – 前のページからの続き

Name	Type	Description
proto_version_ranges	int[]	ノードでサポートされているプロトコルバージョンの範囲

bdr.node_catchup_info

このカタログテーブルは、結合またはパートプロシージャに関連する場合、各ノードに関する関連するキャッチアップ情報を記録します。

bdr.node_catchup_info 列

Name	Type	Description
node_id	oid	ノードの ID
node_source_id	oid	データのソースとして使用されるノードの ID
slot_name	name	このソースに使用されるスロット
min_node_lsn	pg_lsn	ノードが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

ノード (node_id) が別れのノード (origin_node_id) から欠落しているデータをニーズとする場合、転送することで既にそれを持っているノード (node_source_id) から取得できます。このテーブルのレコードは、ノード (node_id) が BDR クラスターのメンバになるまで持続します。

bdr.node_conflict_resolvers

既知のすべての競合タイプに対して現在構成されている競合解決。

bdr.node_conflict_resolvers 列

Name	Type	Description
conflict_type	text	競合の種類
conflict_resolver	text	この競合タイプに使用されるリゾルバー

bdr.node_group

このカタログテーブルには、すべての BDR ノードグループがリストされています。

bdr.node_group 列

Name	Type	Description
node_group_id	oid	ノードグループの ID
node_group_name	name	ノードグループの名前
node_group_default_repset	oid	このノードグループのデフォルトのレプリケーションセット
node_group_default_repset_text	text	このノードグループのデフォルトのレプリケーションセット
node_group_parent_id	oid	親グループの ID（これがルートグループの場合は 0）
node_group_flags	int	グループフラグ
node_group_uuid	uuid	グループの uuid
node_group_apply_delay	interval	プロバイダーからの変更を適用するまでにサブスクライバーが待機する時間
node_group_check_constraints	bool	データの適用時に適用プロセスがコンストレインをチェックするかどうか
node_group_num_writers	int	このノードグループをバックアップするサブスクリプションに使用するライターの数
node_group_enable_wal_decoder	boolean	グループに enable_wal_decoder が設定されているかどうか
node_group_streaming_mode	mode	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

bdr.node_group_replication_sets

BDR グループ用に作成されるデフォルトのレプリケーションセットを示すビュー。bdr.replication_sets も参照してください。

bdr.node_group_replication_sets 列

Name	Type	Description
node_group_name	name	BDR グループの名前
def_repset	name	デフォルトの repset の名前

次のページに続く

表 34 – 前のページからの続き

Name	Type	Description
def_repset_ops	text[]	デフォルトの repset によって複製されるアクション
def_repset_ext	name	デフォルトの「外部」repset の名前（通常は def_repset と同じ）
def_repset_ext_ops	text[]	デフォルトの「外部」repset によって複製されるアクション（通常は def_repset_ops と同じ）

bdr.node_local_info

ローカルノードに固有のノードごとの構成を保存するために使用されるカタログテーブル（ノードごとの構成のグローバルビューとは対照的に）。

bdr.node_local_info 列

Name	Type	Description
node_id	oid	ノードの OID（ローカルノードを含む）
applied_state	oid	ノード状態の内部 ID
ddl_epoch	int8	ノードによって処理された最後のエポック番号
slot_name	name	そのノードへの接続に使用されるスロットの名前（ローカルノードの場合は NULL）

bdr.node_location

ノードの場所に関するユーザー定義の情報を保存するカタログビュー。

bdr.node_location 列

Name	Type	Description
node_group_name	name	BDR グループの名前
node_id	oid	ノードの ID
node_region	text	ユーザー指定のリージョン名前
node_location	text	ユーザー指定のロケーション名前

bdr.node_log_config

競合ロギング構成に関する情報を保存するカタログビュー。

bdr.node_log_config 列

Name	Description
log_name	ロギング構成の名前
log_to_file	サーバーログファイルに記録するかどうか
log_to_table	テーブルにログを記録するかどうか、およびどのテーブルがターゲットか
log_conflict_type	ログに記録する競合の種類 (NULL の場合)
log_conflict_res	NULL がすべてを意味する場合、ログに記録する競合解決

bdr.node_peer_progress

レプリケーションストリーム内のすべてのノードの進行状況を追跡するために使用されるカタログ。クラスター内のすべてのノードは、**bdr.replay_progress_frequency** ミリ秒ごとに他のすべてのノードに進行状況を定期的にブロードキャストします (デフォルトは 60000 ミリ秒、つまり 1 分)。このリレーションには $N * (N-1)$ 行が必要です。

モニタリング目的で **bdr.node_slots** ビューにもっと関心があるかもしれません。

モニタリングとロギング も参照してください。

bdr.node_peer_progress 列

Name	Type	Description
node_id	oid	この位置情報を報告した発信ノードの OID
peer_node_id	oid	この位置情報が報告されたノードの同等な (リモートノード) の OID
last_update_sent_time	timestampz	送信元ノードによってレポートが送信された時間
last_update_rcv_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	レポート時の送信元ノードの LSN
peer_position	pg_lsn	発信元ノードから見たノードの同等な最新の LSN
peer_replay_time	timestampz	レポートノードによって見られる同等な最新のリプレイ時間
last_update_horizon_xid	oid	内部解像度ホライズン: より低い xid はすべてレポートノードで解決されていることがわかっています

次のページに続く

表 38 – 前のページからの続き

Name	Type	Description
last_update_horizon_lsn	pg_lsn	内部解決期間：報告ノードの LSN に関して同じ

bdr.node_pre_commit

Commit At Most Once (CAMO) パートナーとして構成されたノードで内部的に使用されます。過去 15 分間に CAMO パートナーがトランザクションで行った決定を示しています。

bdr.node_pre_commit 列

Name	Type	Description
origin_node_id	oid	トランザクションが実行されたノードの OID
origin_xid	oid	リモートオリジンノードのトランザクション ID
decision	char	コミットの場合は「c」、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
commit_ts	timestampz	トランザクションのコミットタイムスタンプ
decision_ts	timestampz	デシジョンタイム

bdr.node_replication_rates

このビューには、特定のノードからの発信レプリケーションアクティビティに関する情報が含まれています。

bdr.node_replication_rates 列

Column	Type	Description
peer_node_id	oid	この情報が報告されたノードの同等な（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	リプレイ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 とオリジン上の現在の WAL 書き込み位置のバイト差
replay_lag_size	text	replay_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
apply_rate	bigint	同等なで 1 秒あたりに適用されているノード
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までに必要なおおよその時間

注釈: `replay_lag` は再接続後すぐにゼロに設定されます。回避策として、`replay_lag_bytes`、`replay_lag_size`、または `catchup_interval` を使用してください。

bdr.node_slots

このビューには、BDR が現在のデータベースで使用するレプリケーションスロットに関する情報が含まれています。

このビューのフィールドの使用と解釈に関するガイダンスについては、[Monitoring outgoing replication](#) を参照してください。

bdr.node_slots 列

Name	Type	Description
target_dbname	name	ターゲットノードのデータベース名前
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 によるレプリケーションスロットの名前
slot_name	name	Postgres によるスロットの名前（同上）
is_group_slot	boolean	スロットがこのノードのノードグループクラッシュリカバリスロットである場合は true (「Group Replication Slot」](nodes#Group Replication Slot) を参照)
is_decoder_slot	boolean	このスロットは Decoding Worker によって使用されますか
plugin	name	このスロットを使用した論理デコードプラグイン (pglogical_output または bdr でなければなりません)
slot_type	text	スロットのタイプ（ロジカルにする必要があります）
datoid	oid	現在のデータベースの OID
database	name	現在のデータベースの名前
temporary	bool	スロットは一時的なものですか
active	bool	スロットはアクティブですか（接続はありますか）

次のページに続く

表 41 – 前のページからの続き

Name	Type	Description
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	最新の確認された複製位置
usesysid	oid	レプリケーションセッションを実行しているユーザの sysid
username	name	レプリケーションセッションを実行しているユーザのユーザ名
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）
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	接続が開始されたとき
state	text	レプリケーションの状態（キャッチアップ、ストリーミングなど）またはオフラインの場合は「切断」
sent_lsn	pg_lsn	最新の送信位置
write_lsn	pg_lsn	書かれたとおりに報告された最新の位置
flush_lsn	pg_lsn	ディスクにフラッシュされたと報告された最新の位置
replay_lsn	pg_lsn	リプレイとして報告された最新の位置（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	sent_lsn と現在の WAL 書き込み位置のバイト差
write_lag_bytes	int8	write_lsn と現在の WAL 書き込み位置のバイト差
flush_lag_bytes	int8	flush_lsn と現在の WAL 書き込み位置のバイト差
replay_lag_bytes	int8	replay_lsn と現在の WAL 書き込み位置のバイト差
sent_lag_size	text	sent_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
write_lag_size	text	write_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
flush_lag_size	text	flush_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
replay_lag_size	text	replay_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い

注釈: *replay_lag* は再接続後すぐにゼロに設定されます。回避策として、*replay_lag_bytes* または *re-*

`play_lag_size`` を使用します。

bdr.node_summary

このビューには、ローカルノードが認識しているすべての BDR ノードに関する概要情報が含まれています。

bdr.node_summary 列

Name	Type	Description
node_name	name	ノードの名前
node_group_name	name	ノードが属する 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	人間が読めるフォームのノードの一貫した状態
peer_target_state_name	text	ノードが到達しようとしている状態（結合または昇格中）
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複製しますか
node_id	oid	ノードの OID
node_group_id	oid	BDR ノードグループの OID

bdr.queue

このテーブルには、レプリケートされた DDL ステートメントの履歴レコードが保存されます。

bdr.queue 列

Name	Type	Description
queued_at	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[]	ステートメントが発行されたレプリケーションセット
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

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。ユーザクエリの場合、代わりに **bdr.replication_sets** ビューを確認することをお勧めします。

bdr.replication_set 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_nodeid	oid	ノードの OID（現在は常にローカルノード oid）
set_name	name	レプリケーションセットの名前
replicate_insert	boolean	レプリケーションセットが INSERT をレプリケートするかどうかを示します
replicate_update	boolean	レプリケーションセットが UPDATE をレプリケートするかどうかを示します
replicate_delete	boolean	レプリケーションセットが DELETE をレプリケートするかどうかを示します
replicate_truncate	boolean	レプリケーションセットが TRUNCATE を複製するかどうかを示します
set_isinternal	boolean	予約済み
set_autoadd_tables	boolean	このレプリケーションセットに新しいテーブルが自動的に追加されるかどうかを示します
set_autoadd_seqs	boolean	このレプリケーションセットに新しいシーケンスが自動的に追加されるかどうかを示します

bdr.replication_set_table

レプリケーションセットテーブルメンバーシップを格納するテーブル。ユーザクエリの場合、代わりに **bdr.tables** ビューを確認することをお勧めします。

bdr.replication_set_table 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_reloid	regclass	テーブルのローカル ID
set_att_list	text[]	予約済み
set_row_filter	pg_node_tree	コンパイルされ行フィルタリング式

bdr.replication_set_ddl

レプリケーションセット ddl レプリケーションフィルターを格納するテーブル。ユーザクエリの場合、代わりに bdr.ddl_replication ビューを確認することをお勧めします。

bdr.replication_set_ddl 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_ddl_name	name	DDL フィルターの名前
set_ddl_tag	text	DDL フィルターのコマンドタグ
set_ddl_role	text	DDL を実行するロール

bdr.replication_sets

BDR グループで定義されたレプリケーションセットを表示するビュー。現在どのノードでも使用されていません。

bdr.replication_sets 列

Name	Type	Description
set_id	oid	レプリケーションセットの 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 を複製するかどうかを示します
set_autoadd_tables	boolean	このレプリケーションセットに新しいテーブルが自動的に追加されるかどうかを示します
set_autoadd_seqs	boolean	このレプリケーションセットに新しいシーケンスが自動的に追加されるかどうかを示します

bdr.schema_changes

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が BDR に勝ちます。

bdr.schema_changes 列

Name	Type	Description
schema_changes_ts	timestampz	トリガーの ID
schema_changes_change	char	変更タイプのフラグ
schema_changes_classid	oid	クラス ID
schema_changes_objectid	oid	オブジェクト ID
schema_changes_subid	smallint	サブスクリプション
schema_changes_descr	text	オブジェクトが変更されました
schema_changes_addrname	text[]	スキーマ変更の場所

bdr.sequence_alloc

galloc シーケンスのアロケーション詳細を表示するビュー。

bdr.sequence_alloc 列

Name	Type	Description
seqid	regclass	シーケンスの ID
seq_chunk_size	bigint	その値内のチャンクのシーケンス番号
seq_allocated_up_to	bigint	該当なし
seq_nallocs	bigint	該当なし
seq_last_alloc	timestampz	最後に割り当てられたシーケンス

bdr.schema_changes

BDR のスキーマに対するすべての変更を表示するシンプルビュー。

bdr.schema_changes 列

Name	Type	Description
schema_changes_ts	timestampz	トリガーの ID
schema_changes_change	char	変更タイプのフラグ
schema_changes_classid	oid	クラス ID
schema_changes_objectid	oid	オブジェクト ID
schema_changes_subid	smallint	サブスクリプション
schema_changes_descr	text	オブジェクトが変更されました
schema_changes_addrname	text[]	スキーマ変更の場所

bdr.sequence_alloc

割り当てられたシーケンスを表示するビュー。

bdr.sequence_alloc 列

Name	Type	Description
seqid	regclass	シーケンスの ID
seq_chunk_size	bigint	その値内のチャンクのシーケンス番号
seq_allocated_up_to	bigint	該当なし
seq_nallocs	bigint	該当なし
seq_last_alloc	timestampz	最後に割り当てられたシーケンス

bdr.sequences

このビューには、内部 BDR ブックキーピングの順序を除く、すべての順序がその種類とともにリストされます。

bdr.sequences 列

Name	Type	Description
nspname	name	シーケンスを含む名前空間
relname	name	シーケンスの名前
seqkind	text	シーケンスのタイプ (「ローカル」、「timeshard」、「galloc」)

bdr.stat_activity

各バックエンドまたはワーカープロセスの動的アクティビティ。

これには、`pg_stat_activity` と同じ情報が含まれますが、待機が BDR に関連する場合に `wait_event` が正しく設定されます。

bdr.stat_relation

各リレーションに統計を適用します。追跡が有効になっていて、特定のリレーションが複製された場合にのみデータが含まれます。

bdr.stat_relation 列

Column	Type	Description
nspname	name	リレーションのスキーマの名前
relname	name	リレーションの名前
relid	oid	リレーションの OID
total_time	double precision	リレーションのレプリケーションの処理にかかった合計時間
ninsert	bigint	リレーションで複製された挿入の数
nupdate	bigint	リレーションのために複製された更新の数
ndelete	bigint	リレーションでレプリケートされた削除の数
ntruncate	bigint	リレーションで複製された切り捨ての数
shared_blks_hit	bigint	リレーションの共有ブロックキャッシュヒットの合計数
shared_blks_read	bigint	リレーションのために読み取られた共有ブロックの合計数
shared_blks_dirtied	bigint	リレーションのためにダーティ化された共有ブロックの総数
shared_blks_written	bigint	リレーションに書き込まれた共有ブロックの総数
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（ <code>'track_io_timing'</code> が有効になっている場合、それ以外の場合はゼロ）
blk_write_time	double precision	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（ <code>'track_io_timing'</code> が有効になっている場合、それ以外の場合はゼロ）
lock_acquire_time	double precision	リレーションのロックの取得にかかった合計時間、ミリ秒（ <code>'bdr.track_apply_lock_timing'</code> が有効になっている場合、それ以外の場合はゼロ）

bdr.stat_subscription

サブスクリプションごとに統計を適用します。トラッキングが有効になっている場合にのみデータが含まれます。

bdr.stat_subscription 列

Column	Type	Description
sub_name	name	サブスクリプションの名前
subid	oid	サブスクリプションの OID
nconnect	bigint	このサブスクリプションがアップストリームに接続した回数
ncommit	bigint	このサブスクリプションが行ったコミットの数
nabort	bigint	このサブスクリプションに対してライタが行ったアボートの数
nerror	bigint	このサブスクリプションでライタがヒットしたエラーの数
nskippedtx	bigint	このサブスクリプションのライタによってスキップされたされたトランザクションの数 (`skip_transaction` 競合リゾルバによる)
ninsert	bigint	このサブスクリプションが行った挿入の数
nupdate	bigint	このサブスクリプションが行った更新の数
ndelete	bigint	このサブスクリプションが行った削除の数
ntruncate	bigint	このサブスクリプションが行った切り捨ての数
nddl	bigint	このサブスクリプションが実行した DDL 操作の数
ndeadlocks	bigint	デッドロックが原因のエラーの数
nretries	bigint	ライタが行った再試行の数 (完全なリスタート/再接続なし)
nstream_writer	bigint	ライタにストリーミングされたトランザクションの数
nstream_file	bigint	ファイルにストリーミングされたトランザクションの数
nstream_commit	bigint	コミットされたストリーミングトランザクションの数
nstream_abort	bigint	中断されたストリーミングトランザクションの数
nstream_start	bigint	処理された STREAT START メッセージの数
nstream_stop	bigint	処理された STREAM STOP メッセージの数
shared_blks_hit	bigint	サブスクリプションによる共有ブロックキャッシュの合計ヒット数
shared_blks_read	bigint	サブスクリプションによって読み取られた共有ブロックの合計数

次のページに続く

表 54 – 前のページからの続き

Column	Type	Description
shared_blks_dirtied	bigint	サブスクリプションによってダーティー化された共有ブロックの合計数
shared_blks_written	bigint	サブスクリプションによって書き込まれた共有ブロックの合計数
blk_read_time	double precision	サブスクリプションがブロックの読み取りに費やした合計時間、ミリ秒（`track_io_timing` が有効になっている場合、それ以外の場合はゼロ）
blk_write_time	double precision	ミリ秒単位のブロックの書き込みにサブスクリプションが費やした合計時間（`track_io_timing` が有効になっている場合、それ以外の場合はゼロ）
connect_time	timestamp with time zone	現在のアップストリーム接続が確立された時間、接続されていない場合は NULL
last_disconnect_time	timestamp with time zone	最後のアップストリーム接続がドロップされた時間
start_lsn	pg_lsn	このサブスクリプションがアップストリームからのレプリケーションのスタートを要求した LSN
retries_at_same_lsn	bigint	サブスクリプションが同じ LSN 値からリスタートされた回数
curr_ncommit	bigint	現在の接続が確立された後にこのサブスクリプションが行ったコミットの数

bdr.subscription

このカタログテーブルには、ローカル BDR ノードが所有するすべてのサブスクリプションとそのモードがリストされています。

bdr.subscription 列

Name	Type	Description
sub_id	oid	サブスクリプションの ID
sub_name	name	サブスクリプションの名前
nodegroup_id	oid	ノードグループの ID
origin_node_id	oid	オリジンノードの ID
source_node_id	oid	ソースノードの ID
target_node_id	oid	対象ノードの ID
subscription_mode	char	サブスクリプションのモード
sub_enabled	bool	サブスクリプションが有効になっているかどうか（レプリケーションである必要があります）
apply_delay	interval	このサブスクリプションの変更の適用はどのくらい遅れますか（通常は 0）

次のページに続く

表 55 – 前のページからの続き

Name	Type	Description
slot_name	name	このサブスクリプションで使用す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するローカルオリジン
num_writers	int	このサブスクリプションが使用するライタプロセスの数
streaming_mode	char	サブスクリプションのストリーミング構成
replication_sets	text[]	このサブスクリプションによってレプリケートされるレプリケーションセット (NULL = all)
forward_origin	text[]	このサブスクリプションによって転送されたオリジン (NULL = all)

bdr.subscription_summary

このビューには、ローカルノードが他のノードに対して持っているすべての BDR サブスクリプションに関する概要情報が含まれています。

bdr.subscription_summary 列

Name	Type	Description
node_group_name	name	ノードが属する BDR グループの名前
sub_name	name	サブスクリプションの名前
origin_name	name	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ですか
sub_slot_name	name	このサブスクリプションで使用するオリジンノードのスロット名前
sub_replication_sets	text[]	サブスクライブしたレプリケーションセット
sub_forward_origins	text[]	サブスクリプションは、オリジン以外の他のノードから転送された変更を受け入れますか
sub_apply_delay	interval	オリジンと比較してこれだけトランザクションを遅らせる
sub_origin_name	name	このサブスクリプションで使用す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	ノードが属する BDR グループの OID
sub_id	oid	サブスクリプションの OID
origin_id	oid	オリジンノードの OID

次のページに続く

表 56 – 前のページからの続き

Name	Type	Description
target_id	oid	ターゲットノードの OID
receive_lsn	pg_lsn	受信した変更またはメッセージの最新の LSN（これはリスタートの場合に戻ることができます）
receive_commit_lsn	pg_lsn	受信した最後の COMMIT の最新の LSN（これはリスタートの場合に戻ることができます）
last_xact_replay_lsn	pg_lsn	このサブスクリプションで再生された最後のトランザクションの LSN
last_xact_flush_lsn	timestampz	ディスクに永続的にフラッシュされた、このサブスクリプションで再生された最後のトランザクションの LSN
last_xact_replay_timestamp	timestampz	このサブスクリプションで再生された最後のトランザクションのタイムスタンプ

bdr.replication_status

このビューには、ローカルノードと BDR クラスタ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 回リスタートされ、現在のアップストリーム接続が確立された後にまだ単一のトランザクションが適用されていない場合、レプリケーションはブロックされていると見なします。リスタート後の最初のトランザクションが非常に大きく、まだ適用されている場合、**replication_blocked** の結果が間違っている可能性があります。

これがロジカル・スタンバイ・ノードの場合、その上流ノードのステータスのみが表示されます。同様に、サブスクライバー専用ノードはレプリケーションの変更を他のノードに送信しないため、レプリケーションステータスは表示されません。

bdr.replication_status 列

Column	Type	Description
node_id	oid	ローカルノードの OID
node_name	name	ローカルノードの名前
origin_node_id	oid	オリジンノードの OID
origin_node_name	name	オリジンノードの名前
sub_id	oid	このオリジンノードのサブスクリプションの OID
sub_name	name	このオリジンノードのサブスクリプションの名前
connected	boolean	このノードはオリジンノードに接続されていますか？
replication_blocked	boolean	このオリジンのレプリケーションは現在ブロックされていますか？
connect_time	timestamp with time zone	現在の接続が確立された時刻

次のページに続く

表 57 – 前のページからの続き

Column	Type	Description
disconnect_time	timestamp with time zone	最後の接続が切断された時刻
uptime	interval	このオリジンで現在の接続がアクティブになってからの時間

bdr.tables

このビューには、レプリケーションセットのテーブルメンバーシップに関する情報がリストされます。テーブルが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回表示されます。

bdr.tables 列

Name	Type	Description
relid	oid	リレーションの OID
nspname	name	スキーマリレーションの名前
relname	name	リレーションの名前
set_name	name	レプリケーションセットの名前
set_ops	text[]	レプリケートされたオペレーションのリスト
rel_columns	text[]	レプリケートされた列のリスト (NULL = すべての列) (*)
row_filter	text	行フィルタリング式
conflict_detection	text	使用される競合検出メソッド: row_origin (デフォルト)、row_version または column_level

(*) これらの列は将来の使用のために予約れており、現在は NULL である必要があります

bdr.trigger

このビューでは、作成されたすべてのストリームトリガーを表示できます。多くの場合、ここでのトリガーは `bdr.create_conflict_trigger` から作成されます。

bdr.trigger 列

Name	Type	Description
trigger_id	oid	トリガーの ID
trigger_reloid	regclass	関連関クションの名前
trigger_pgtgid	oid	Postgres トリガー ID
trigger_type	char	トリガー呼び出しの種類

次のページに続く

表 59 – 前のページからの続き

Name	Type	Description
trigger_name	name	トリガーの名前

bdr.trigger

読みやすい列を持つ **bdr.trigger** の展開ビュー。

Name	Type	Description
trigger_name	name	トリガーの名前
event_manipulation	text	オペレーション
trigger_type	bdr.trigger_type	トリガーの種類
trigger_table	bdr.trigger_reloid	それを呼び出すテーブル
trigger_function	name	使用したファンクション

bdr.workers

実行中の BDR ワーカープロセスに関する情報。

これを pid を使用して **bdr.stat_activity** と結合すると、BDR ワーカーの状態をさらに詳しく知ることができます。

bdr.workers 列

Name	Type	Description
worker_pid	int	ワーカープロセスのプロセス ID
worker_role	int	ワーカーロールの数値表現形式
worker_role_name	text	ワーカーロールの名前
worker_subid	oid	ワーカーが関連付けられている場合のサブスクリプション ID
worker_commit_timestamp	timestampz	このワーカーによって処理された最後のコミットのタイムスタンプがある場合
worker_local_timestamp	timestampz	上記のコミットが処理されたローカル時間

bdr.worker_errors

BDR バックグラウンドワーカプロセスからのエラーの永続的なログ。

bdr.worker_errors 列

Name	Type	Description
node_group_name	name	BDR グループの名前
origin_name	name	オリジンノードの名前
source_name	name	該当なし
target_name	name	ターゲットノードの名前（通常はローカルノード）
sub_name	name	サブスクリプションの名前
worker_role	int4	このワーカーのロールの内部識別子（1: マネージャ、2: receive、3: ライタ、4: 出力、5: 拡張機能）
worker_role_name	text	ロール名前
worker_pid	int4	エラーが発生したワーカーのプロセス ID
error_time	timestampz	エラーの日時
error_age	interval	エラーからの経過時間
error_message	text	エラーの説明
error_context_message	text	エラーが発生したコンテキスト
remoterelid	oid	そのノード上のリモートリレーションの OID

bdr.writers

BDR ライタプロセスに関する特定の情報。

bdr.writers 列

Name	Type	Description
sub_name	name	サブスクリプションの名前
pid	int	ワーカープロセスのプロセス ID
syncing_rel	int	同期されるリレーションの OID（ある場合）
streaming_allowed	text	このライタは、ライタへの直接ストリーミングのターゲットになりますか

次のページに続く

表 63 – 前のページからの続き

Name	Type	Description
is_streaming	bool	このライタにストリーミングされているトランザクションはありますか
remote_xid	xid	処理中のトランザクションのリモートトランザクション ID (ある場合)
remote_commit_lsn	pg_lsn	処理された最後のコミットの LSN
commit_queue_position	int	内部コミットキューでの位置
commit_timestamp	timestampz	処理された最後のコミットのタイムスタンプ
nxacts	bigint	このライタによって処理されたトランザクションの数
ncommits	bigint	このライタによってコミットされたトランザクションの数
naborts	bigint	このライタによって中止されたトランザクションの数
nstream_file	bigint	このライタによって処理されたファイルへのストリーミングトランザクションの数
nstream_writer	bigint	このライタによって処理されたライターへのストリーミングトランザクションの数

bdr.worker_tasks

bdr.worker_tasks ビューには、BDR の現在のワーカー起動レート制限状態と、バックグラウンドワーカー起動と登録アクティビティに関する基本的な統計が表示されます。

ここにリストされている他のビューとは異なり、現在のデータベースと BDR ノードに固有ではありません。現在の PostgreSQL インスタンスのすべての BDR ノードの状態が表示されます。現在のデータベースに参加して、フィルタリングします。

bdr.worker_tasks は walsender と出力プラグインを追跡しません。

bdr.worker_tasks 列

Column	Type	Description
task_key_worker_role	integer	ワーカーロール識別子
task_key_worker_role_name	text	ワーカーロール名前
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 (可能な場合)
task_key_ext_libname	name	ライブラリの名前 (ほとんどの場合 bdr)
task_key_ext_funcname	name	ファンクションエントリーポイントの名前

次のページに続く

表 64 – 前のページからの続き

Column	Type	Description
task_key_ext_workername	name	ワーカーに割り当てられた名前
task_key_remoterelid	oid	利用可能な場合、リモート同期リレーシヨンの識別子
task_pid	integer	ワーカーのプロセス ID
task_registered	timestamp with time zone	ワーカー登録のタイムスタンプ
since_registered	interval	ワーカーが登録してからの間隔
task_attached	timestamp with time zone	ワーカーアタッチタイムスタンプ
since_attached	interval	ワーカーがアタッチされてからの間隔
task_exited	timestamp with time zone	ワーカー出口のタイムスタンプ
since_exited	interval	ワーカーが終了してからの間隔
task_success	boolean	ワーカーはまだ実行されていますか？
task_next_launch_not_before	timestamp with time zone	ワーカーが再度再起動されるタイムスタンプ
until_launch_allowed	interval	次の打ち上げまでの残り時間
task_last_launch_requestorpid	integer	起動を要求したプロセス ID
task_last_launch_request_time	timestamp with time zone	リクエストが行われたときのタイムスタンプ
since_last_request	interval	最後の要求からの間隔
task_last_launch_request_applied	boolean	最後の要求は成功しましたか？
task_nrequests	integer	リクエスト数
task_nregistrations	integer	登録数
task_prev_pid	integer	前世代のプロセス ID
task_prev_registered	timestamp with time zone	前回登録したタスクのタイムスタンプ
since_prev_registered	interval	前回登録からの間隔
task_prev_launched	timestamp with time zone	前回の起動のタイムスタンプ
since_prev_launched	interval	前回の打ち上げからの間隔
task_prev_exited	timestamp with time zone	前のタスクが終了したときのタイムスタンプ
since_prev_exited	interval	前のタスクが終了してからの間隔
task_first_registered	timestamp with time zone	最初の登録が発生したときのタイムスタンプ
since_first_registered	interval	初回登録からの間隔

`bdr.autopartition_work_queue`

autopartition ワーカーによって作成および処理された作業項目が含まれています。作業項目は 1 つのノードでのみ作成され、別のノードで処理されます。

bdr.autopartition_work_queue 列

Column	Type	Description
ap_wq_workid	bigint	作業アイテムの一意の ID
ap_wq_ruleid	int	autopartition_rules にリストされているルールの ID。ルールは bdr.autopartition コマンドを使用して指定します
ap_wq_relname	name	自動パーティション化されるリレーションの名前
ap_wq_relnamespace	name	この作業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）のいずれかです。
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

bdr.autopartition_workitem_status

各ノードでローカルに更新される作業項目のステータス。

bdr.autopartition_workitem_status 列

Column	Type	Description
ap_wi_workid	bigint	ワークアイテムの ID
ap_wi_nodeid	Oid	作業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）のいずれかです
ap_wi_started_at	timestampz	作業アイテムのスタートタイムスタンプ
ap_wi_finished_at	timestampz	作業アイテムの終了タイムスタンプ

bdr.autopartition_local_work_queue

autopartition ワーカーによって作成および処理された作業項目が含まれています。これは bdr.autopartition_work_queue に似ていますが、これらの作業項目はローカル管理テーブル用です。各ノードは、クラスター内の他のノードから独立して、独自のローカル作業項目を作成して処理します。

bdr.autopartition_local_work_queue 列

Column	Type	Description
ap_wq_workid	bigint	作業アイテムの一意の ID
ap_wq_ruleid	int	autopartition_rules にリストされているルールの ID。ルールは bdr.autopartition コマンドを使用して指定します
ap_wq_relname	name	自動パーティション化されるリレーシヨンの名前
ap_wq_relnamespace	name	この作業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）のいずれかです。
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

bdr.autopartition_local_workitem_status

ローカル管理テーブルの作業項目のステータス。

bdr.autopartition_local_workitem_status 列

Column	Type	Description
ap_wi_workid	bigint	ワークアイテムの ID
ap_wi_nodeid	Oid	作業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）のいずれかです

次のページに続く

表 68 – 前のページからの続き

Column	Type	Description
ap_wi_started_at	timestampz	作業アイテムのスタートタイムスタンプ
ap_wi_finished_at	timestampz	作業アイテムの終了タイムスタンプ

bdr.group_camo_details

`bdr.run_on_all_nodes` を使用して、すべてのノードから CAMO 関連の情報を収集します。

bdr.group_camo_details 列

Name	Type	Description
node_id	text	内部ノード ID
node_name	text	ノードの名前
camo_partner	text	カモ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	リプレイとして報告された最新の位置（可視）
receive_lsn	text	受信した変更またはメッセージの最新の LSN（リスタートの場合は前に戻ることができます）
apply_queue_size	text	apply_lsn と receive_lsn のバイト差

bdr.camo_pairs

すべてのクラスターで構成されているすべての CAMO ペアに関する情報。

bdr.camo_pairs 列

Name	Type	Description
node_group_id	oid	ノードグループ ID
left_node_id	oid	ノード ID
right_node_id	oid	ノード ID
require_raft	bool	ローカルモードへの切り替えが過半数を必要とするかどうか

注釈: `left` と `right` の名前には特別な意味はありません。BDR4 は対称 CAMO 構成のみを構成できます。つまり、ペアの両方のノードは相互に CAMO パートナーです。

bdr.commit_scopes

グループコミットを有効にするために `bdr.commit_scope` に使用できるすべてのコミットスコープを保存するカタログ。

bdr.commit_scopes 列

Name	Type	Description
commit_scope_id	oid	被参照するスコープの ID
commit_scope_name	name	被参照するスコープの名前
commit_scope_origin_node_group	oid	ID で被参照される、ルールが適用されるノードグループ
sync_scope_rule	text	範囲の定義

bdr.group_raft_details

`bdr.run_on_all_nodes` を使用して、すべてのノードから Raft コンセンサスステータスを収集します。

bdr.group_raft_details 列

Name	Type	Description
node_id	oid	内部ノード ID
node_name	name	ノードの名前
state	text	ノード上のラフトワーカーの状態
leader_id	oid	RAFT_LEADER のノード ID
current_term	int	raft 選出の内部 ID
commit_index	int	Raftsnapshot の内部 ID
nodes	int	アクセス可能なノード数
voting_nodes	int	投票するノード数
protocol_version	int	このノードのプロトコルバージョン

bdr.group_replslots_details

`bdr.run_on_all_nodes` を使用して、すべてのノードから BDR スロット情報を収集します。

bdr.group_replslots_details 列

Name	Type	Description
node_group_name	text	BDR グループの名前
origin_name	text	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す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（接続はありますか）
state	text	レプリケーションの状態（キャッチアップ、ストリーミングなど）またはオフラインの場合は「切断」
write_lag	interval	報告された書き込みのおおよその遅延時間
flush_lag	interval	報告されたフラッシュのおおよその遅延時間
replay_lag	interval	報告されたリプレイのおおよそのラグタイム
sent_lag_bytes	int8	sent_lsn と現在の WAL 書き込み位置のバイト差
write_lag_bytes	int8	write_lsn と現在の WAL 書き込み位置のバイト差
flush_lag_bytes	int8	flush_lsn と現在の WAL 書き込み位置のバイト差
replay_lag_byte	int8	replay_lsn と現在の WAL 書き込み位置のバイト差

bdr.group_subscription_summary

`bdr.run_on_all_nodes` を使用して、すべてのノードからサブスクリプションステータスを収集します。

bdr.group_subscription_summary 列

Name	Type	Description
origin_node_name	text	サブスクリプションのオリジンの名前
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
sub_lag_seconds	text	now から last_xact_replay_timestamp までのラグ

bdr.group_versions_details

`bdr.run_on_all_nodes` を使用して、すべてのノードから BDR 情報を収集します。

bdr.group_versions_details 列

Name	Type	Description
<code>node_id</code>	<code>oid</code>	内部ノード ID
<code>node_name</code>	<code>name</code>	ノードの名前
<code>postgres_version</code>	<code>text</code>	ノード上の PostgreSQL バージョン
<code>bdr_version</code>	<code>text</code>	ノードの BDR バージョン

2.23.2 内部カタログとビュー**bdr.ddl_epoch**

DDL エポックごとの状態を保持する内部カタログテーブル。

bdr.ddl_epoch 列

Name	Type	Description
<code>ddl_epoch</code>	<code>int8</code>	単調に増加するエポック番号
<code>origin_node_id</code>	<code>oid</code>	このエポックの作成を要求したノードの内部ノード ID
<code>epoch_consume_timeout</code>	<code>timestampz</code>	このエポックのタイムアウト
<code>epoch_consumed</code>	<code>boolean</code>	ローカルノードがエポックを完全に処理するとすぐに <code>true</code> に切り替わります

bdr.internal_node_pre_commit

内部カタログ表。 `bdr.node_pre_commit` ビューを使用します。

bdr.sequence_kind

各非ローカルシーケンスのタイプを格納する内部状態テーブル。診断にはビュー `bdr.sequences` をお勧めします。

bdr.sequence_kind 列

Name	Type	Description
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'=タイムハード、's'=snowflakeid、'g'=galloc)

bdr.state_journal

内部ノード状態ジャーナル。代わりに、診断のために `bdr.state_journal_details` を使用してください。

bdr.state_journal_details

各ノードのノード状態の変更はすべて、診断目的で `bdr.state_journal` に永続的にログに記録されます。このビューは、ノード名と人間が読める状態の名前を提供し、そのジャーナルのすべての情報を運びます。ノードが正常に参加すると、最後の状態エントリーは `BDR_PEER_STATE_ACTIVE` です。これは、`bdr.node_slots.state` にリストされている各レプリケーション接続の状態とは異なります。

bdr.state_journal_details 列

Name	Type	Description
state_counter	oid	単調増加イベントカウンタ、ノードごと
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	人間が読める状態の名前
entered_time	timestampz	現在のノードが状態の変化を監視した時点

2.24 BDR System Functions

BDR 管理は、主に SQL 呼び出し可能な関数を介して実行されます。BDR のすべての機能は `bdr` スキーマで公開されています。これらの関数への呼び出しは、`search_path` に `bdr` を置くのではなく、スキーマ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、現在使用されている 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するため、この値を小なりと大なりの比較に使用できます。

次の式は、メジャーバージョン、マイナーバージョン、およびパッチリリースで構成されるバージョン番号を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 システム情報 Functions

`bdr.get_relation_stats`

リレーション情報を返します。

`bdr.get_subscription_stats`

現在のサブスクリプション統計を返します。

2.24.3 システムと進行状況の情報パラメーター

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 がレポートするそのようなすべてのパラメータをリストします。

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。 Connection pools and proxies 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各 COMMIT の後、このパラメータは更新されて、オリジンノードのコミットレコードの最後をポイントします。bdr.wait_for_apply_queue と組み合わせて、これにより、アプリケーションはマルチプルのノードにわたって因果読み取りを実行できます。つまり、トランザクションがリモートで可視になるまで待機できます。

transaction_id

CAMO が有効になっている場合、Postgres がトランザクション ID を割り当てるとすぐにこのパラメータが更新され、割り当てられたトランザクション ID が表示されます。

2.24.4 コンセンサスファンクション

bdr.consensus_disable

サーバーをリスタートするまで、または bdr.consensus_enable を使用して再度有効にするまで（どちらか早い方）、ローカルノードのコンセンサスワーカーを無効にします。

警告: コンセンサスを無効にすると BDR の一部の機能が無効になり、長期間無効のままにすると最終的に BDR クラスターの可用性にインパクトを与えます。このファンクションは、テクニカルサポートと連携してのみ使用してください。

bdr.consensus_enable

ローカルノードで無効になっているコンセンサスワーカーを再度有効にします。

bdr.consensus_proto_version

ローカルノードで現在使用されているコンセンサスプロトコルバージョンを返します。

BDR グループの再構成の内部メカニズムで必要です。

bdr.consensus_snapshot_export

概要

```
bdr.consensus_snapshot_export(version integer DEFAULT NULL)
```

ローカルノードの現在コミットして適用された状態から新しい BDR コンセンサス snapshot を生成し、bytea として結果を返します。

デフォルトでは、サポートされている最も高い Raft バージョンの snapshot がエクスポートされます。ただし、明示的な version 番号を渡すことでオーバーライドできます。

エクスポートするノードは現在の Raft リーダーである必要はなく、リーダーの最新の状態で完全な最新である必要もありません。ただし、そのような snapshot は bdr.consensus_snapshot_import() によって受け入れられない場合があります（以下を参照）。

新しい snapshot は、ローカルノードの bdr.local_consensus_snapshot テーブルに自動的に保存されません。呼び出し元のみ返されます。

生成された snapshot は、エクスポートノードのログ位置の後ろにある同じ BDR ノードグループ内の他のノードの bdr.consensus_snapshot_import() に渡すことができます。

このファンクションを動作させるには、ローカル BDR コンセンサスワーカーを無効にする必要があります。典型的な使用法は次のとおりです。

```
SELECT bdr.bdr_consensus_disable();
\copy (SELECT * FROM bdr.consensus_snapshot_export()) TO my_node_consensus_snapshot.
→data
SELECT bdr.bdr_consensus_enable();
```

BDR コンセンサスワーカーが無効になっている間、ノードでの DDL ロックの試行は失敗するかタイムアウトになり、galloc シーケンスは新しい値を取得せず、Eager および CAMO トランザクションは一時停止または ERROR になり、分散コンセンサスシステムをニーズとする他の機能が中断されます。必要なダウンタイムは一般的に非常に短いです。

ユースケースによっては、bdr.local_consensus_snapshot テーブルの snapshot フィールドから既に存在する snapshot を抽出して、代わりに使用する方が実用的な場合があります。これを行うために、コンセンサスワーカーを停止する必要はありません。

bdr.consensus_snapshot_import

概要

```
bdr.consensus_snapshot_import(IN snapshot bytea)
```

通常は同じ BDR ノードグループ内の別のノードから、`bdr.consensus_snapshot_export()` によってエクスポートされたコンセンサス snapshot をインポートします。

別のノードにある `bdr.local_consensus_snapshot` テーブルの `snapshot` フィールドから直接抽出した snapshot を使用することもできます。

このファンクションは、破損またはユーザの間違いが発生した場合に、BDR ノードのカatalogの状態を既知の良好な状態にリセットするのに役立ちます。

snapshot が生成された時点でインポートノードの `apply_index` がスナップショットをエクスポートするノードの `commit_index` 以下である場合、スナップショットをインポートできます。`bdr.get_raft_status()` を参照してください。ログが既に先にあるために snapshot を受け入れることができないノードは、`ERROR` を発生させます。snapshot がインポートされると、ノードは現在のリーダーから残りの変更をフェッチするため、インポートされた snapshot ショットは完全に最新である必要はありません。

このファンクションを動作させるには、インポートするノードで BDR コンセンサスワーカーを無効にする必要があります。詳細については、`bdr.consensus_snapshot_export()` のノートを参照してください。

次を実行して、これを使用してローカルノードに新しい Raftsnapshot を生成させることができます。

```
SELECT bdr.consensus_snapshot_import(bdr.consensus_snapshot_export());
```

これにより、現在適用されているログ位置まで Raft ログが切り捨てられる場合があります。

bdr.consensus_snapshot_verify

概要

```
bdr.consensus_snapshot_verify(IN snapshot bytea)
```

`bdr.consensus_snapshot_export()` によってエクスポートされた指定されたコンセンサス snapshot を確認します。snapshot ヘッダーには、が生成されたバージョンが含まれており、ノードは同じバージョンに対して検証しようとします。

snapshot は、同じノードまたはクラスター内の他のノードでエクスポートされた可能性があります。snapshot を検証するノードがエクスポートされた snapshot のバージョンをサポートしていない場合、エラーが発生します。

bdr.get_consensus_status

現在のコンセンサス（Raft）ワーカーに関するステータス情報を返します。

bdr.get_raft_status

現在のコンセンサス（Raft）ワーカーに関するステータス情報を返します。`bdr.get_consensus_status` のエイリアス。

bdr.raft_leadership_transfer

概要

```
bdr.raft_leadership_transfer(IN node_name text, IN wait_for_completion boolean)
```

`node_name` で識別されるノードを Raft リーダーとして要求します。要求は任意の BDR ノードから開始でき、指定されたノードにリーダーシップを転送するために現在のリーダーに内部的に転送されます。指定されたノードは、アクティブな BDR ノードであり、完全な投票権を持つ必要があります。

`wait_for_completion` が `false` の場合、要求はベストエフォートで処理されます。`bdr.raft_election_timeout` 期間内にノードがリーダーになれない場合、他の有能なノードが再びリーダーになります。また、リーダーシップは Raft プロトコルごとに期間にわたって変わる可能性があります。`true` が結果れる結果は、要求が正常に送信されたことのみを示します。

`wait_for_completion` が `true` の場合、ファンクションは指定されたノードが新しいリーダーになるまで待機し、要求されたノードが Raft リーダーにならない場合（ネットワークの問題などにより）無限に待機します。したがって、無限ループを防ぐには、常に `wait_for_completion` と組み合わせて `statement_timeout` を設定することをお勧めします。

2.24.5 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で再生されるまで待機できます。

スロットが特定の LSN を渡すまで待機します。位置が指定されていない場合、ローカルノードで現在の書き込み位置が使用されます。

スロット名前が渡されない場合、すべての BDR スロットが LSN を渡すまで待機します。

このファンクションは、他のノードからの変更を 1000 ミリ秒ごとにポーリングします。

スロットが同時にドロップされた場合、そのスロットの待機は終了します。ノードが現在ダウンしており、そのスロットを更新していない場合、待機は続行されます。その場合、`statement_timeout` をより早く完了するように設定することをお勧めします。

概要

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

パラメーター

- `slot_name` - レプリケーションスロットの名前、または NULL の場合、すべての BDR スロット (のみ)
- `target_lsn` - 待機する LSN、または NULL の場合はローカルノードで現在の書き込み LSN を使用します

`bdr.wait_for_apply_queue`

ファンクション `bdr.wait_for_apply_queue` を使用すると、BDR ノードは、特定の BDR ノードから発生する特定のトランザクションのローカルアプリケーションを待機できます。その同等な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ノード。アプリケーションまたはプロキシは、このファンクションを使用して、古い読み取りを防ぐことができます。

便宜上、BDR は CAMO および CAMO パートナーノードにこのファンクションの特別なバリエーションを提供します。

`bdr.wait_for_camo_partner_queue` を参照してください。

特定の LSN が指定されている場合、それが同等なからのリカバリストリームのポイントです。これは、以前の接続または同時のノードでその同等なから取得した `bdr.last_committed_lsn` 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ローカルの受信バッファを確認し、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。既に受け取っ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、同等なに障害が発生し、どのトランザクションがノードされたかがわからない場合に特に役立ちます。この場合、送信側でまだ転送中またはバッファリングされているトランザクションは待機しないことに注意してください。

概要

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

パラメーター

- ノード着信トランザクションがキューに入れられ、待機する必要がある同等な名前。NULL の場合、すべての同等な適用キューが消費されるのを待ちます。
- ノード - 待機する同等なからのレプリケーションストリーム内の LSN。通常、同等なから `bdr.last_committed_lsn` を介して学習されノード。

`bdr.get_node_sub_receive_lsn`

このファンクションをサブスクライバーで使用して、指定されたオリジンから受信した最後の LSN を取得できます。適用するトランザクションに関連する LSN インクリメントのみをアカウントに入れるようにフィルター処理するか、フィルター処理しません。

このファンクションの出力と `bdr.get_node_sub_apply_lsn()` の出力の差が、対応する適用キューのサイズを測定します。

概要

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

パラメーター

- `node_name` - LSN を取得するレプリケーションストリームのソースであるノードの名前/
- `committed` - デフォルト (true) では、このファンクションは最後の LSN 全体ではなく、受信したトランザクションのコミットのみをアカウントします。サブスクライバーノードに影響を与えないアクションを含みます。

`bdr.get_node_sub_apply_lsn`

このファンクションをサブスクライバーで使用して、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。

概要

```
bdr.get_node_sub_apply_lsn(node_name name)
```

パラメーター

- `node_name` - LSN を取得するレプリケーションストリームのソースであるノードの名前。

`bdr.run_on_all_nodes`

すべてのノードでクエリーを実行するファンクション。

警告: このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザの権限を使用して、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権限を付与する場合は注意が必要です。

概要

```
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。

注意事項

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、各ノードからの結果を json フォーマットで返します。各ノードから複数の行が返され、json 配列としてエンコードされます。ノードがダウンしているため接続できないなどのエラーが応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメーター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実行しません。既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドは既にクラスター内のすべてのノードに送信されているため、`bdr.ddl_replication = off` で接続が作成されます。

レプリケーションが中断され、ノード間の一貫性が失われるリスクがあるため、このファンクションを使用するときは注意してください。透過的な DDL レプリケーションまたは `bdr.replicate_ddl_command()` を使用して DDL をレプリケートします。DDL は将来のリリースでブロックされる可能性があります。

例

例、次のクエリーで、このファンクションをモニタリングで使用する と便利 です。

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
    FROM bdr.node_slots
    WHERE origin_name IS NOT NULL
$$);
```

…2 ノードクラスター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でクエリーを実行するファンクション。

警告: このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザの権限を使用して、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権限を付与する場合は注意が必要です。

概要

```
bdr.run_on_nodes(node_names text[], query text)
```

パラメーター

- `node_names` - text クエリーが実行されるノード名の配列。
- `query` - 実行される任意のクエリー。

注意事項

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、各ノードからの結果を json フォーマットで返します。各ノードから複数の行が返され、json 配列としてエンコードされます。ノードがダウンしているため接続できないなどのエラーが応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメーター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実行しません。既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドは既にクラスター内のすべてのノードに送信されているため、`bdr.ddl_replication = off` で接続が作成されます。

レプリケーションが中断され、ノード間の一貫性が失われるリスクがあるため、このファンクションを使用するときは注意してください。透過的な DDL レプリケーションまたは `bdr.replicate_ddl_command()` を使用して DDL をレプリケートします。DDL は将来のリリースでブロックされる可能性があります。

bdr.run_on_group

ノードのグループでクエリーを実行するファンクション。

警告: このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザの特権で、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権限を付与する場合は注意が必要です。

概要

```
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。

注意事項

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、各ノードからの結果を json フォーマットで返します。各ノードから複数の行が返され、json 配列としてエンコードされます。ノードがダウンしているため接続できないなどのエラーが応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメーター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実行しません。既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドは既にクラスター内のすべてのノードに送信されているため、`bdr.ddl_replication = off` で接続が作成されます。

レプリケーションが中断され、ノード間の一貫性が失われるリスクがあるため、このファンクションを使用するときは注意してください。透過的な DDL レプリケーションまたは `bdr.replicate_ddl_command()` を使用して DDL をレプリケートします。DDL は将来のリリースでブロックされる可能性があります。

`bdr.global_lock_table`

このファンクションは、指定されたテーブルのグローバル DML ロックを取得します。グローバル DML ロックについては、[DDL ロックの詳細](#) を参照してください。

概要

```
bdr.global_lock_table(relation regclass)
```

パラメーター

- `relation` - ロックされるリレーションの名前または OID。

注意事項

このファンクションは、`ddl_locking` 設定とは無関係にグローバル DML ロックを取得します。

`bdr.global_lock_table` ファンクションには、`bdr.backwards_compatibility` が 30618 以下に設定されている場合を除き、ロックされた `relation` に対する `UPDATE`、`DELETE`、または `TRUNCATE` 権限が必要です。

`bdr.wait_for_xid_progress`

このファンクションを使用して、特定のノード（ノード ID で識別）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` まで待機できます。進行状況は、ノードに適用されているトランザクションとして定義され、このノードはトランザクションが適用される前に行われた他のすべてのレプリケーション変更を確認します。

概要

```
bdr.wait_for_xid_progress(origin_node_id oid, origin_topxid int4, allnodes boolean,  
→DEFAULT true)
```

パラメーター

- `origin_node_id` - トランザクションが発生したノードのノード ID。
- `origin_topxid` - トランザクションの XID。
- `allnodes` - `true` の場合、すべてのノードでトランザクションが進行するのを待ちます。それ以外の場合は、現在のノードのみを待ちます。

注意事項

現在、これらのトランザクションのみが追跡されているため、このファンクションは DDL コマンドをレプリケートしたトランザクションにのみ使用できます。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久に、または `statement_timeout` に到達するまで待機する場合があります。

bdr.local_group_slot_name

ローカルノードのグループスロットの名前を返します。

例

```
bdrdb=# SELECT bdr.local_group_slot_name();
local_group_slot_name
-----
bdr_bdrdb_bdrgroup
```

bdr.node_group_type

指定されたノードグループのタイプを返します。戻り値は、ノードグループの作成時に `bdr.create_node_group()` に渡された値と同じですが、グループの作成時に `node_group_type` が `NULL` として渡された場合に `normal` が返されます。

例

```
bdrdb=# SELECT bdr.node_group_type(bdrgroup);
node_group_type
-----
normal
```

2.24.6 グローバルアドバイザリロック

BDR は、グローバル勧告ロックをサポートしています。これらのロックは、BDR がサポートする勧告ロックが本質的にグローバルであることを除いて、PostgreSQL で利用可能な勧告ロックに非常に似ています。これらは、DDL ロックと同様のセマンティクスに従います。したがって、アドバイザリロックは多数決コンセンサスによって取得されるため、すべてのノードの過半数が連携できる限り、1 つ以上のノードがダウンまたは遅れている場合でも使用できます。

現在、EXCLUSIVE ロックのみ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ロックを既に取得している場合、他のノードまたはバックエンドはロックが解放されるまで待機する必要があります。

アドバイザリロックは本質的にトランザクションです。したがって、ロックは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解放されます。その場合、解放されるとすぐに利用できます。セッションレベルのアドバイザリロックは現在サポートされていません。

グローバル勧告ロックはリエントラントです。したがって、同じリソースが 3 回ロックされている場合、他のセッションでできるように解放するには 3 回アンロックする必要があります。

bdr.global_advisory_lock

このファンクションは、指定されたオブジェクトの EXCLUSIVE ロックを取得します。ロックが利用できない場合、ロックが利用可能になるか、`bdr.global_lock_timeout` に到達するまで待機しロック。

概要

```
bdr.global_advisory_lock(key bigint)
```

パラメーター

- `key` - アドバイザリロックが取得されるオブジェクト。

概要

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

パラメーター

- `key1` - 複合キーの最初のパート。
- `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、アプリケーション定義のソースで以前に取得したロックを解放します。ロックは、アプリケーションによって同じトランザクションで以前に取得されている必要があります。そうでない場合、`ERROR` が発生します。

概要

```
bdr.global_advisory_unlock(key bigint)
```

パラメーター

- `key` - アドバイザリロックが取得されるオブジェクト。

概要

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

パラメーター

- `key1` - 複合キーの最初のパート。
- `key2` - 複合キーの 2 番目のパート。

2.24.7 モニタリング Functions

bdr.monitor_group_versions

クラスタ全体のバージョンチェックを提供するために、このファンクションはビュー `bdr.group_version_details` から返される BDR バージョン情報を使用します。

概要

```
bdr.monitor_group_versions()
```

注意事項

モニタリングとロギング で説明されているように、このファンクションは、フィールド `status` および `message` を持つレコードを返します。

このファンクションは `bdr.run_on_all_nodes()` を呼び出します。

bdr.monitor_group_raft

クラスタ全体の Raft チェックを提供するために、このファンクションはビュー `bdr.group_raft_details` から返された BDR Raft 情報を使用します。

概要

```
bdr.monitor_group_raft()
```

注意事項

このファンクションは、[モニタリングとロギング](#)で説明されているように、フィールド `status` およびレコードを返します。

このファンクションは `bdr.run_on_all_nodes()` を呼び出します。

bdr.monitor_local_replslots

このファンクションは、ビュー `pg_replication_slots`（スロットアクティブまたは非アクティブ）から返されたレプリケーションスロットステータス情報を使用して、BDR グループスロットを除くすべてのレプリケーションスロットを考慮したローカルチェックを提供します。

概要

```
bdr.monitor_local_replslots()
```

注意事項

このファンクションは、[Monitoring Replication Slots](#)で説明されているように、フィールド `status` およびレコードを返します。

bdr.wal_sender_stats

[デコードワーカー](#) が有効になっている場合、このファンクションは、各 WAL 送信者が読み取っているデコーダースロットと現在の LCR（Logical Change Record）セグメントファイルに関する情報を表示します。

概要

```
bdr.wal_sender_stats() &rarr; setof record (pid integer, is_using_lcr boolean, ↵
↵decoder_slot_name TEXT, lcr_file_name TEXT)
```

出力列

- pid - WAL 送信者の PID (pg_stat_replication の pid 列に対応)
- is_using_lcr - WAL 送信者が LCR ファイルを送信しているかどうか。is_using_lcr が FALSE の場合、次の列は NULL になります。
- decoder_slot_name - デコーダーのレプリケーションスロットの名前。
- lcr_file_name - 現在の LCR ファイルの名前。

bdr.get_decoding_worker_stat

デコードワーカー が有効になっている場合、このファンクションは現在のデータベースに関連付けられた Decoding

Worker の状態に関する情報を表示します。これは、pg_replication_slots を介して利用できるよりも詳細な Decoding Worker の進行状況に関する情報も提供します。

概要

```
bdr.get_decoding_worker_stat() &rarr; setof record (pid integer, decoded_upto_lsn pg_
↵lsn, waiting BOOL, waiting_for_lsn pg_lsn)
```

出力列

- pid - Decoding Worker の PID (pg_replication_slots の active_pid 列に対応)
- decoded_upto_lsn - Decoding Worker がトランザクションログを読み取った LSN
- waiting - Decoding Worker が新しい WAL を待っているかどうか
- waiting_for_lsn - 次に予想される WAL の LSN

注意事項

詳細については、Monitoring WAL senders using LCR を参照してください。

bdr.lag_control

ラグ制御 が有効になっている場合、このファンクションは、コミット遅延と、ローカルノードと現在のデータベースに構成されているラグメジャーに準拠するノードの数に関する情報を表示します。

概要

`bdr.lag_control()`

出力列

- `commit_delay` - ミリ秒単位の現在のランタイムのコミット遅延
- `commit_delay_maximum` - ミリ秒単位で構成された最大コミット遅延
- `commit_delay_adjustment` - ミリ秒単位のサンプルインターバル中に可能なランタイムコミット遅延への変更
- `conforming_nodes` - ラグメジャーに準拠するノードの現在のランタイム数
- `conforming_nodes_minimum` - ラグメジャーに準拠するために必要なノードの構成された最小数、それ以下ではコミット遅延調整が適用されます
- `lag_bytes_threshold` - コミット遅延が適用されるラグサイズ（キロバイト）
- `lag_bytes_maximum` - 構成された最大ラグサイズ（キロバイト）
- `lag_time_threshold` - コミット遅延が適用される遅延時間（ミリ秒）
- `lag_time_maximum` - ミリ秒単位の構成された最大遅延時間
- `sample_interval` - 遅延サンプル間の構成された最小時間とミリ秒単位の可能なコミット遅延調整

2.24.8 内部 Functions

BDR メッセージペイロード Functions

`bdr.decode_message_response_payload` および `bdr.decode_message_payload`

これらの関数は、コンセンサスペイロードをより人間が読みやすい出力にデコードします。

主に `bdr.global_consensus_journal_details` デバッグビューによって使用されます。

bdr.get_global_locks

このファンクションは、ローカルノードで保持されているグローバルロックに関する情報を表示します。

`bdr.global_locks` ビューを実装して、ロックのより詳細な概要を提供するために使用されます。

bdr.get_slot_flush_timestamp

指定されたレプリケーションスロットの最後のフラッシュ位置確認のタイムスタンプを取得します。

`bdr.node_slots` ビューを実装するために内部で使用されます。

BDR 内部ファンクションレプリケーション Functions

`bdr.internal_alter_sequence_set_kind`、`internal_replication_set_add_table`、`internal_replication_set_remove_table`

さまざまなファンクション呼び出しのレプリケーションのために内部で使用する Functions。

BDR の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためにのみ存在します。

bdr.internal_submit_join_request

新しいノードに参加するためのコンセンサスリクエストを送信します。

BDR グループの再構成の内部メカニズムで必要です。

bdr.isolation_test_session_is_blocked

グローバルロックのブロックの追加チェックを使用してオリジナルの `pg_isolation_test_session_is_blocked` を拡張（および実際に呼び出し）するヘルパファンクション。

分離/並行性テストに使用されます。

bdr.local_node_info

このファンクションは、BDR グループの再構成の内部メカニズムが必要とするローカルノードの情報を表示します。

ビュー `bdr.local_node_summary` は、ユーザが使用するのに役立つ同様の情報を提供します。

bdr.msgb_connect

コンセンサスプロトコルによって使用される、別のノードのコネクションプーラに接続するためのファンクション。

bdr.msgb_deliver_message

コンセンサスプロトコルによって使用される、別のノードのコネクションプーラにメッセージを送信するためのファンクション。

bdr.peer_state_name

このファンクションは、ノードの状態 (`node_state`) をテキスト表現形式に変換し、主に `bdr.node_summary` ビューを実装するために使用されます。

bdr.request_replay_progress_update

「リプレイ進行状況の更新」Raft メッセージの即時書き込みを要求します。これは主にテスト目的で使用されますが、コンセンサスメカニズムが機能しているかどうかをテストするために使用することもできます。

bdr.seq_nextval

シーケンスインクリメントの内部実装。

このファンクションは、[BDR グローバルシーケンス](#) と対話するクエリで標準の `nextval` の代わりに使用されます。

注意事項

以下も内部 BDR シーケンス操作関数です。 `bdr.seq_currval` と `bdr.sql_lastval` は自動的に使用されます。

bdr.show_subscription_status

サブスクリプションのステータスに関する情報を取得し、主に `bdr.subscription_summary` ビューを実装するために使用されます。

bdr.conflict_resolution_to_string

競合解決を oid からテキストに変換します。

ビュー `bdr.conflict_history_summary` はこれを使用して、競合解決のためのユーザフレンドリ情報を提供します。

bdr.conflict_type_to_string

競合タイプを oid からテキストに変換します。

ビュー `bdr.conflict_history_summary` はこれを使用して、競合解決のためのユーザフレンドリ情報を提供します。

bdr.get_node_conflict_resolvers

ローカルノード上のすべての競合リゾルバーのテキスト文字列を表示します。

bdr.reset_subscription_stats

`bdr.stat_subscription` で表示されるように、サブスクリプションによって作成された統計をリセットした後、ブール値を返します。

bdr.reset_relation_stats

`bdr.stat_relation` で表示されるように、リレーションの統計をリセットした後、ブール値を返します。

bdr.pg_xact_origin

指定されたトランザクションのオリジン ID を返します。

概要

<code>bdr.pg_xact_origin(xmin xid)</code>

パラメーター

- `xid` - オリジンを返すトランザクション ID

bdr.difference_fix_origin_create

引数として渡された名前でレプリケーションオリジンを作成しますが、`bdr_` プレフィックスを追加します。オリジンの内部 ID を返します。これは、`postgres` のスーパーユーザ権限ではなく `bdr_superuser` が必要であることを除いて、`pg_replication_origin_create()` と同じ機能を実行します。

概要

bdr.difference_fix_session_setup

現在のセッションを現在のオリジンから再生していることを示します。このファンクションは、セッションに対して事前に作成された `bdr_local_only_origin` ローカルレプリケーションオリジンを暗黙的に使用します。リプレイの進行状況を報告できます。`void` を返します。これは `pg_replication_origin_session_setup()` と同じ機能を実行しますが、これには `postgres` のスーパーユーザ権限ではなく `bdr_superuser` が必要です。ファンクションの以前のフォーム: `bdr.difference_fix_session_setup(text)` は非推奨であり、今後のリリースで削除されることに注意してください。

概要

`bdr.difference_fix_session_setup()`

bdr.difference_fix_session_reset

現在のセッションをオリジンから再生していないものとしてマークし、基本的に `bdr.difference_fix_session_setup()` の効果をリセットします。 `void` を返します。これは `pg_replication_origin_session_reset()` と同じ機能を実行しますが、これには `postgres` のスーパーユーザ権限ではなく `bdr_superuser` が必要です。

概要

`bdr.difference_fix_session_reset()`

bdr.difference_fix_xact_set_avoid_conflict

LSN '0/0' およびタイムスタンプ '2000-01-01' でコミットされたトランザクションを再生するものとして、現在のトランザクションをマークします。これは、postgres のスーパーユーザ権限ではなく `bdr_superuser` が必要であることを除いて、`pg_replication_origin_xact_setup('0/0', '2000-01-01')` と同じ機能を実行します。

概要

```
bdr.difference_fix_xact_set_avoid_conflict()
```

bdr.resynchronous_table_from_node (node_name 名前、リレーション regclass)

リモートノードからリレーションを再同期します。

概要

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

パラメーター

- `node_name` - リレーションデータのコピー/再同期元のノード。
- `relation` - リモートノードからコピーされるリレーション。

注意事項

これにより、リレーションのグローバル DML ロックが取得され、ローカルでリレーションが切り捨てられ、リモートノードからリレーションにデータがコピーされます。

リレーションは、同じ名前と定義で両方のノードに存在する必要があります。

同じパーティション定義を持つテーブルパーティションの再同期、パーティションテーブルから非パーティションテーブルへの再同期、および外部キーコンストレインを一時的に削除して再作成することによる被参照テーブルの再同期はすべてサポートされています。

被参照されるテーブルでファンクションを実行した後、被参照される列のデータが参照する列の値と一致しない場合、エラーがスローされ、参照テーブルデータを再同期した後にファンクションを再実行する必要があります。

さらに、リモートノードからデータをコピーした後、生成された列の値をローカルで計算することにより、生成された列を含むテーブルの再同期をサポートします。

現在、`row_filters` はこのファンクションでは無視されます。

`bdr.resynchronize_table_from_node` ファンクションは、テーブルの所有者が `bdr_superuser` 権限を持っている場合にのみ実行できます。

`bdr.consensus_kv_store`

一貫した KV ストアに値を保存します。

値の有効期限のタイムスタンプを返します。これは `ttl` に依存します。`ttl` が `NULL` の場合、これは `infinity` を結果ます値が削除された場合は `-infinity` を結果ます。

概要

```
bdr.consensus_kv_store(key text, value jsonb,  
                        prev_value jsonb DEFAULT NULL, ttl int DEFAULT NULL)
```

パラメーター

- `key` - 挿入、更新、または削除するための任意の一意のキー。
- `value` - 保存する json 値、`NULL` の場合、既存のレコードは削除されます
- `prev_value` - 設定すると、現在の値が `prev_value` と等しい場合にのみ書き込みオペレーションが実行されます。
- `ttl` - 新しい値の存続時間（ミリ秒）。

注意事項

これは内部ファンクションであり、主に `HARP` によって使用されます。

警告: このファンクションは、ユーザアプリケーションでは使用しないでください。

`bdr.consensus_kv_fetch`

一貫した KV ストアから json フォーマットで値を取得します。

概要

```
bdr.consensus_kv_fetch(IN key text) RETURNS jsonb
```

パラメーター

- **key** - フェッチする任意のキー。

注意事項

これは内部ファンクションであり、主に HARP によって使用されます。

警告: このファンクションは、ユーザアプリケーションでは使用しないでください。

bdr.alter_subscription_skip_changes_upto

ロジカルレプリケーションはバージョン間でレプリケートできますが、ロールのようなグローバル変更をレプリケートせず、選択的にレプリケートできるため、ロジカルレプリケーションの適用プロセスでエラーが発生し、変更の適用が停止する場合があります。

このような問題は、可能な限りターゲット側に変更を加えて修正する必要があります。レプリケーションをブロックしている欠落しているテーブル、**CREATE** 必要なロール、**GRANT** 必要なパーミッションなどを削除します。ただし、問題をその方法で修正できず、トランザクションを完全にスキップ必要がある場合があります。変更は、トランザクション全体としてスキップされたされます。スキップする場所を決定するには、以下の例のようにログ出力を使用してコミット LSN を見つけるか、ロジカルデコーディング機能を使用して変更ストリームをピークします。

トランザクションが1つの変更のみを加えた場合を除き、多くの場合、ターゲット側にトランザクションの効果を手動で適用する必要があるため、可能な限り問題のあるトランザクションを保存することが重要です。以下の例を参照してください。

`pg_catalog.pg_logical_slot_get_binary_changes` を使用して目的の LSN にスキップすることにより、`bdr.alter_subscription_skip_changes_upto` なしで変更をスキップすることができるため、これは本当に便利ファンクションです。より高速なスキップを行います。ただし、ロジカルデコーディングでいくつかの種類のエラーをバイパスする場合があります。

このファンクションは、無効になっているサブスクリプションでのみ機能します。

通常のシーケンスは次のとおりです。

*問題のサブスクリプションと問題のコミットの LSN を特定する

*サブスクリプションを無効にする

- `pg_catalog.pg_logical_slot_peek_changes` を使用してトランザクションのコピーを保存するソースノードで（可能であれば）

*ターゲットノードの `bdr.alter_subscription_skip_changes_upto`

*必要に応じて、ターゲットに修復または同等の変更を手動で適用します

*サブスクリプションを再度有効にします

警告: このファンクションを使用すると、問題が簡単に悪化します。それが唯一の選択肢であると確信がない限り、何もしないでください。

概要

```
bdr.alter_subscription_skip_changes_upto(  
    subname text,  
    skip_upto_and_including pg_lsn  
);
```

例

トランザクションの適用が `ERROR` で失敗し、ターゲット側での変更などの影響の少ない修正ではこの問題を解決しないと判断しました。あなたは、トランザクションをスキップする必要があると判断しました。

この人工的な例のように、エラーログで、スキップするコミットレコード LSN を見つけます。

```
ERROR:  XX000: CONFLICT: target_table_missing; resolver skip_if_recently_dropped_  
→returned an error: table does not exist  
CONTEXT:  during apply of INSERT from remote relation public.break_me in xact with_  
→commit-end lsn 0/300AC18 xid 131315  
committs 2021-02-02 15:11:03.913792+01 (action #2) (effective sess origin id=2 lsn=0/  
→300AC18)  
while consuming I message from receiver for subscription bdr_regression_bdrgroup_  
→node1_node2 (id=2667578509)  
on node node2 (id=3367056606) from upstream node node1 (id=1148549230, reporiginid=2)
```

ログのこの部分には、必要な情報があります。 `the_target_lsn: 0/300AC18` `the_subscription: bdr_regression_bdrgroup_node1_node2`

次に、サブスクリプションを無効にして、適用ワーカーがレプリケーションスロットに接続しようとしないうにします。

```
SELECT bdr.alter_subscription_disable(the_subscription);
```

トランザクションの一部のみをスキップすることはできません。それはすべてか無かです。したがって、サブスクリプションのスロット名前を使用して、最初にプロバイダー側で `COPY ing` して、そのレコードを保存することを強くお勧めします。

```
\copy (SELECT * FROM pg_catalog.pg_logical_slot_peek_changes(the_slot_name,  
    the_target_lsn, NULL, min_proto_version, 1, max_proto_version, 1,  
    startup_params_format, 1, proto_format, json))  
TO transaction_to_drop.csv WITH (FORMAT csv);
```

この例は読みやすくするためにマルチプル行に分割されていますが、`\copy` は複数行のコマンドをサポートしていないため、1 行で発行する必要があることに注意してください。

上記の「peek」を「get」に変更することで変更をスキップできますが、`bdr....skip_changes_upto` はすべてのデータのデコードと出力を回避するより高速にスキップします。

```
SELECT bdr.alter_subscription_skip_changes_upto(subscription_name,  
    the_target_lsn);
```

必要または必要に応じて、ダンプされたトランザクションの内容をガイドとして使用して、同じ変更（またはそれらの修復バージョン）をターゲットノードに手動で適用します。

最後に、サブスクリプションを再度有効にします。

```
SELECT bdr.alter_subscription_enable(the_subscription);
```