



BDR (Bi-Directional Replication)

リリース 3.7.0

EnterpriseDB

2021 年 10 月 18 日

目次

第 1 章 Architectural Overview	3
1.1 基本アーキテクチャ	4
1.1.1 複数のグループ	4
1.1.2 複数のマスター	4
1.1.3 非同期、デフォルト	4
1.1.4 メッシュトポロジ	4
1.1.5 論理複製	5
1.1.6 高可用性	5
1.1.7 制限	5
1.2 アーキテクチャのオプションとパフォーマンス	6
1.2.1 BDR パフォーマンスの特性評価	6
1.2.2 1 つのグループ内の BDR MultiMaster	6
1.2.3 BDR AlwaysOn	7
1.2.4 BDR ワールドワイド	7
1.2.5 BDR オートスケール	8
1.3 展開	9
1.4 クロックとタイムゾーン	9
第 2 章 Application Usage	11
2.1 アプリケーションの動作	11
2.2 複製されていないステートメント	13
2.3 DML および DDL レプリケーション	13
2.4 異なるリリースレベル間の複製	14
2.5 相違点のあるノード間の複製	14
2.6 違いのあるノード間の比較	15
2.7 アプリケーションの一般規則	16
2.8 タイミングの考慮事項と同期複製	16
2.9 アプリケーションのテスト	16
2.9.1 TPAexec	17
2.9.2 CAMO / フェイルオーバーオプションを備えた pgbench	17
2.9.3 マルチノードアクセスの分離テスター	18
2.10 パフォーマンスのテストとチューニング	20
2.11 適合性の評価	21
2.11.1 主キー/レプリカ ID の更新の評価	21
2.11.2 テーブルまたは SELECT クエリでの LOCK の使用の評価	22
第 3 章 PostgreSQL Configuration for BDR	23
3.1 BDR の PostgreSQL 設定	23
3.2 BDR の 2ndQPostgres / EDB Postgres 拡張設定	24
3.3 BDR の pglogical 設定	24

3.4	BDR 固有の設定	25
3.4.1	競合処理	25
3.4.2	グローバルシークエンスパラメータ	25
3.4.3	DDL 処理	25
3.4.4	グローバルロック	26
3.4.5	ノード管理	27
3.4.6	汎用レプリケーション	27
3.4.7	CRDT	28
3.4.8	最大準備トランザクション	28
3.4.9	熱心な複製	28
3.4.10	最大 1 回コミットする	29
3.4.11	タイムスタンプベースのスナップショット	29
3.4.12	監視とログ	29
3.4.13	内部	30
第 4 章	Node Management	31
4.1	BDR グループの作成と参加	31
4.1.1	異種クラスターへの参加	33
4.1.2	接続 DSN と SSL (TLS)	33
4.2	監視ノード	34
4.3	ロジカルスタンバイノード	34
4.4	フィジカルスタンバイノード	36
4.5	サブグループ	38
4.5.1	サブスクライバー専用グループ	38
4.6	デコードワーカー	39
4.6.1	メモ	40
4.7	ノードの再起動とノードの復旧	40
4.7.1	BDR によって作成された複製スロット	41
4.7.2	ロング識別子のハッシュ	42
4.8	BDR グループからノードを削除する	43
4.8.1	BDR のアンインストール	43
4.9	BDR トポロジーのリスト	44
4.9.1	BDR グループのリスト	44
4.9.2	BDR グループ内のノードのリスト	44
4.9.3	ノード状態のリスト	45
4.10	シャードグループの管理	46
4.10.1	作成/ドロップ	46
4.10.2	展開	46
4.10.3	リバランス	46
4.11	ノード管理インターフェース	46
4.11.1	bdr.create_node	47
4.11.2	bdr.drop_node	47
4.11.3	bdr.create_node_group	48
4.11.4	bdr.alter_node_group_config	49
4.11.5	bdr.join_node_group	51
4.11.6	bdr.promote_node	52
4.11.7	bdr.wait_for_join_completion	52

4.11.8	bdr.part_node	53
4.11.9	bdr.alter_node_interface	54
4.11.10	bdr.alter_subscription_enable	55
4.11.11	bdr.alter_subscription_disable	55
4.12	ノード管理コマンド	56
4.12.1	bdr_init_physical	56
第 5 章	DDL Replication	59
5.1	DDL レプリケーションオプション	59
5.2	BDR システムでの DDL の実行	60
5.3	DDL ロックの詳細	61
5.4	DDL の影響を最小限に抑える	63
5.5	ダウンノードでの DDL の処理	64
5.6	ステートメント固有の DDL レプリケーションの懸念	64
5.7	BDR DDL コマンド処理マトリックス	65
5.7.1	ALTER SEQUENCE	70
5.7.2	ALTER TABLE	70
5.7.3	ALTER TYPE	75
5.7.4	コメントオン	75
5.7.5	シーケンスの作成	75
5.7.6	テーブルの作成	75
5.7.7	CREATE TABLE AS および SELECT INTO	75
5.7.8	説明	76
5.7.9	付与および取り消し	76
5.7.10	ロックテーブル	76
5.7.11	セキュリティラベル	76
5.7.12	切り捨てレプリケーション	77
5.7.13	TRUNCATE ロック	77
5.7.14	ロール操作ステートメント	77
5.7.15	制限され DDL の回避策	78
5.7.16	DDL のように動作する BDRFunctions	80
第 6 章	Security and Roles	83
6.1	カタログ表	83
6.2	BDRFunctions と演算子	83
6.3	カタログオブジェクトに対する権限の付与	84
6.4	ロール管理	84
6.5	ロールとレプリケーション	84
6.6	接続ロール	85
6.7	特権の制限	85
6.7.1	外部キー特権	86
6.7.2	トリガー	86
6.8	BDR デフォルト/事前定義ロール	87
6.8.1	bdr_superuser	87
6.8.2	bdr_read_all_stats	87
6.8.3	bdr_monitor	89
6.8.4	bdr_application	89

6.8.5	bdr_read_all_conflicts	90
6.9	検証	90
6.9.1	コベリティ	90
6.9.2	CIS ベンチマーク	90
第 7 章	Conflicts	93
7.1	競合の発生方法	94
7.2	競合の種類	94
7.2.1	主キーまたは一意の競合	94
7.2.2	外部キー制約の競合	101
7.2.3	競合を打ち切る	102
7.2.4	除外制約の競合	103
7.2.5	ロールとテーブルスペースの違いによるデータの競合	103
7.2.6	ロックの競合とデッドロックの中止	103
7.2.7	発散する対立	104
7.2.8	TOAST サポートの詳細	104
7.3	競合の回避または許容	105
7.4	競合検出	106
7.4.1	オリジン競合検出	106
7.4.2	行バージョンの競合検出	107
7.4.3	bdr.alter_table_conflict_detection	108
7.4.4	競合タイプのリスト	109
7.5	競合の解決	110
7.5.1	bdr.alter_node_set_conflict_resolver	110
7.5.2	コンフリクトリゾルバのリスト	111
7.5.3	デフォルトの競合解決	113
7.5.4	競合解決のリスト	113
7.6	競合ログ	113
7.6.1	bdr.alter_node_set_log_config	114
7.6.2	競合の報告	115
7.7	LiveCompare によるデータ検証	116
第 8 章	Sequences	117
8.1	BDR グローバルシーケンス	117
8.1.1	タイムシャードシーケンス	118
8.1.2	グローバルに割り当てられたレンジのシーケンス	119
8.2	UUID、KSUUID、およびその他のアプローチ	122
8.2.1	UUID と KSUUID	122
8.2.2	ステップとオフセットのシーケンス	123
8.3	グローバルシーケンス管理インターフェイス	124
8.3.1	bdr.alter_sequence_set_kind	124
8.3.2	bdr.extract_timestamp_from_timesthard	125
8.3.3	bdr.extract_nodeid_from_timesthard	125
8.3.4	bdr.extract_localseqid_from_timesthard	126
8.3.5	bdr.timestamp_to_timesthard	126
8.4	KSUUID v2Functions	127
8.4.1	bdr.gen_ksuuid_v2	127

8.4.2	bdr.ksuuid_v2_cmp	127
8.4.3	bdr.extract_timestamp_from_ksuuid_v2	128
8.5	KSUUID v1Functions	128
8.5.1	bdr.gen_ksuuid	129
8.5.2	bdr.uuid_v1_cmp	129
8.5.3	bdr.extract_timestamp_from_ksuuid	129
第 9 章	Column-Level Conflict Detection	131
9.1	列レベルの競合解決の有効化と無効化	132
9.1.1	例	132
9.1.2	列レベルの競合解決を使用したテーブルの一覧表示	132
9.1.3	bdr.column_timestamps_create	133
9.2	DDL ロック	133
9.3	現在とコミットのタイムスタンプ	133
9.4	列のタイムスタンプの検査	134
9.5	CRDT データ型を使用した列の競合の処理	135
9.6	制限	136
第 10 章	Conflict-free Replicated Data Types	139
10.1	状態ベースおよび操作ベースの CRDT	141
10.1.1	操作ベースの CRDT タイプ (CmCRDT)	142
10.1.2	状態ベースの CRDT タイプ (CvCRDT)	142
10.2	ディスクスペースの要件	143
10.3	CRDT タイプと競合処理	144
10.4	CRDT タイプと競合レポート	144
10.5	CRDT 値のリセット	144
10.6	実装された CRDT データ型	145
10.6.1	成長専用カウンタ (crdt_gcounter)	145
10.6.2	成長のみの総和 (crdt_gsum)	146
10.6.3	正負カウンタ (crdt_pncounter)	147
10.6.4	正負の総和 (crdt_pnsum)	148
10.6.5	デルタカウンタ (crdt_delta_counter)	150
10.6.6	総和 (crdt_delta_sum)	150
第 11 章	Durability & Performance Options	153
第 12 章	Eager Replication	155
第 13 章	Commit At Most Once	157
第 14 章	Database Scaling	159
第 15 章	Timestamp-based Snapshots	161
第 16 章	Replication Sets	163
第 17 章	Stream Triggers	165
第 18 章	Backup and Recovery	167

第 19 章 Upgrades	169
第 20 章 Explicit Two-Phase Commit (2PC)	171
第 21 章 Catalogs and Views	173
第 22 章 Monitoring	175
22.1 監視の概要	175
22.2 ノードの参加と削除の監視	176
22.3 レプリケーションピアの監視	177
22.3.1 送信レプリケーションの監視	177
22.3.2 着信レプリケーションの監視	179
22.3.3 LCR を使用して WAL 送信者を監視する	180
22.4 BDR レプリケーションワーカーの監視	180
22.5 グローバルロックの監視	181
22.6 競合の監視	181
22.7 外部モニタリング	182
22.7.1 bdr.set_node_location	182
22.7.2 bdr.set_network_path_info	183
22.7.3 bdr.network_monitoring ビュー	183
22.8 統計を適用する	184
22.9 標準の PostgreSQL 統計 Views	185
22.10 BDR バージョンの監視	185
22.11 ラフトコンセンサスの監視	186
22.12 レプリケーションスロットの監視	188
22.13 トランザクションのコミットの監視	189
第 23 章 BDR System Functions	191
23.1 バージョン情報 Functions	191
23.1.1 bdr.bdr_edition	191
23.1.2 bdr.bdr_version	191
23.1.3 bdr.bdr_version_num	191
23.1.4 bdr.wal_sender_stats	192
23.2 システムおよび進捗情報のパラメーター	192
23.2.1 bdr.local_node_id	192
23.2.2 bdr.last_committed_lsn	192
23.2.3 transaction_id	193
23.3 ユーティリティ Functions	193
23.3.1 bdr.wait_slot_confirm_lsn	193
23.3.2 bdr.wait_for_apply_queue	193
23.3.3 bdr.get_node_sub_receive_lsn	194
23.3.4 bdr.get_node_sub_apply_lsn	195
23.3.5 bdr.run_on_all_nodes	195
23.3.6 bdr.global_lock_table	197
23.3.7 bdr.wait_for_xid_progress	197
23.3.8 bdr.local_group_slot_name	198
23.3.9 bdr.node_group_type	198
23.4 グローバルアドバイザリーロック	199

23.4.1	bdr.global_advisory_lock	199
23.4.2	bdr.global_advisory_unlock	200
第 24 章 Credits and Licence		203
第 25 章 ' Appendix A		205
25.1	BDR 3.7.12	205
25.1.1	改善	205
25.1.2	解決された問題	205
25.1.3	アップグレード	206
25.2	BDR 3.7.11	206
25.2.1	改善	206
25.2.2	解決された問題	207
25.2.3	アップグレード	207
25.3	BDR 3.7.10	207
25.3.1	改善	207
25.3.2	解決された問題	208
25.4	BDR 3.7.9	209
25.4.1	改善	209
25.4.2	解決された問題	209
25.5	BDR 3.7.8	211
25.5.1	重要な注意	211
25.5.2	BDR 3.7 のハイライト	211
25.5.3	改善	212
25.5.4	解決された問題	213
25.5.5	その他の変更	214
25.6	BDR 3.7.7	214
25.6.1	重要な注意	214
25.6.2	改善	215
25.6.3	解決された問題	216
25.7	BDR 3.7.6	217
25.7.1	重要な注意	217
25.7.2	改善	217
25.7.3	解決された問題	218
25.7.4	その他の変更	218
25.8	BDR 3.7.5	219
25.8.1	重要な注意	219
25.8.2	改善	219
25.8.3	解決された問題	220
25.9	BDR 3.7.4	221
25.9.1	重要な注意	221
25.9.2	改善	221
25.9.3	解決された問題	222
25.10	BDR 3.7.3	223
25.10.1	重要な注意	223
25.10.2	改善	223
25.10.3	解決された問題	224

25.10.4	その他の改善	225
25.11	BDR 3.7.2	225
25.11.1	重要な注意	225
25.11.2	BDR 3.7 のハイライト	226
25.11.3	その他の改善	226
25.11.4	解決された問題	227
25.12	BDR 3.6.19	227
25.12.1	解決された問題	227
25.12.2	改善	228
25.13	BDR 3.6.18	229
25.13.1	改善	229
25.13.2	解決された問題	229
25.14	BDR 3.6.17	230
25.14.1	改善	230
25.14.2	解決された問題	231
25.15	BDR 3.6.16	232
25.15.1	改善	232
25.15.2	解決された問題	232
25.16	BDR 3.6.15	233
25.16.1	改善	233
25.16.2	解決された問題	233
25.17	BDR 3.6.14	234
25.17.1	改善	234
25.17.2	解決された問題	234
25.18	BDR 3.6.12	236
25.18.1	改善	236
25.18.2	解決された問題	236
25.19	BDR 3.6.11	237
25.19.1	改善	237
25.19.2	解決された問題	238
25.20	BDR 3.6.10	239
25.20.1	改善	239
25.20.2	解決された問題	240
25.21	BDR 3.6.9	240
25.21.1	改善	240
25.21.2	解決された問題	240
25.22	BDR 3.6.8	241
25.22.1	改善	241
25.22.2	解決された問題	241
25.23	BDR 3.6.7.1	242
25.23.1	解決された問題	242
25.24	BDR 3.6.7	242
25.24.1	改善	242
25.24.2	解決された問題	243
25.25	BDR 3.6.6	244
25.25.1	改善	244
25.25.2	解決された問題	244

25.26 BDR 3.6.5	245
25.26.1 改善	245
25.26.2 解決された問題	246
25.27 BDR 3.6.4	246
25.27.1 BDR 3.6.4 のハイライト	247
25.27.2 解決された問題	247
25.27.3 その他の改善	248
25.28 BDR 3.6.3	248
25.28.1 BDR 3.6.3 のハイライト	248
25.28.2 解決された問題	248
25.28.3 その他の改善	248
25.29 BDR 3.6.2	249
25.29.1 BDR 3.6.2 のハイライト	249
25.29.2 解決された問題	249
25.30 BDR 3.6.1	250
25.30.1 3.6.1 のハイライト	250
25.30.2 解決された問題	250
25.31 BDR 3.6.0.2	251
25.31.1 解決された問題	251
25.32 BDR 3.6.0.1	251
25.32.1 解決された問題	251
25.33 BDR 3.6.0	252
25.33.1 BDR 3.6 のハイライト	252
25.33.2 解決された問題	252
25.33.3 その他の改善	252
 第 26 章 ’ Appendix B	 253
第 27 章 ’ Appendix C	255
第 28 章 ’ Appendix D	257
第 29 章 ’ Appendix E	259

title: BDR (Bi-Directional Replication)

BDR（Bi-Directional Replication の略）は、メッシュトポロジでマルチマスタークラスターを構築するためのソリューションを提供する PostgreSQL の拡張機能です。つまり、任意のサーバーに書き込むことができ、変更は他のすべてのサーバーに行ごとに送信されます。同じ BDR グループのパートです。

BDR バージョン 3（“BDR3”）は、[pglogical3extension](#) に基づいています。ただし、BDR3 について知っておく必要があるすべてのものがここに含まれており、[pglogical docs](#) を参照することは不要であり、混乱を招く可能性もあります。

この文書では、BDR1 および BDR2 と呼ばれる以前のアーキテクチャではなく、BDR3 のみを参照しています。BDR3 には重要かつ重要な違いがあり、以前のドキュメントを参照したり、BDR3 に記載されているものに依存したりしないでください。

以下に示すように、BDR3 は PostgreSQL のさまざまなバージョンとバリエーションをサポートしています。

**** BDR バージョンサポート互換性マトリクス：****

BDR	Server	Supported Versions
3.6	PostgreSQL	10, 11
3.6	EDB Postgres Extended	11
3.7	PostgreSQL	11, 12, 13
3.7	EDB Postgres Advanced	11, 12, 13
3.7	EDB Postgres Extended	11, 12, 13

EDB Postgres Extended は、以前は 2ndQ Postgres として知られていました。

一部の機能は、Postgres サーバーの特定のバージョンでのみ使用可能です。

- 競合のない複製されたデータ型-提供する追加のデータ型非同期マルチマスター更新で数学的に証明された一貫性 シナリオ（EDB Postgres Advanced、EDB Postgres Extended）
- 列レベルの競合解決-列ごとの最終更新を使用する機能別のフィールドの更新を「マージ」できるように、解像度が優先されますどちらかを失う（EDB Postgres Advanced、EDB Postgres Extended）
- 変換トリガー-の着信ストリームで実行されるトリガー変更または高度なプログラムフィルタリングを行う機能を提供するデータ（EDB Postgres Advanced、EDB Postgres Extended）
- 競合トリガー-競合が検出されたときに呼び出されるトリガー、カスタムの競合解決手法を使用する方法を提供する（EDB Postgres Advanced、EDB Postgres Extended）
- 追加の DDL サポート（CREATE TABLE AS）（EDB Postgres Advanced、EDB Postgres Extended）
- NOT VALID コンストレインおよび ALTER TABLE の高度な DDL 処理（EDB Postgres Advanced、EDB Postgres Extended）
- ロジカル/フィジカルスタンバイサーバーの追加の同期により高速化フェイルオーバー可能なスタンバイの構築（EDB Postgres Advanced、EDB Postgres Extended）

- 並列適用-マルチプルのライターが着信変更を適用できるようにします (EDB Postgres Advanced、EDB Postgres Extended)
- Eager Replication-クラスターのノード間で同期する前競合のフリーレプリケーションを提供するためにトランザクションをコミットする (EDB Postgres Extended)
- 最大 1 回コミット-一貫性のある機能各トランザクションを一度だけコミットするアプリケーション。ノード障害の存在 (EDB Postgres Extended)
- タイムスタンプベースのスナップショット-マルチプルで一貫した読み取りを提供指定された時間に出現する、または出現するデータを取得するためのノード (EDB Postgres Extended)
- レプリケーションのキャッチアップ時間の見積もり (EDB Postgres Extended)
- 凍結を抑えて、更新/削除の競合の解決を支援 (EDB Postgres Extended)
- デコードワーカー (EDB Postgres Extended 13 以降)

現在 EDB Postgres Extended でのみ使用可能な機能は、EDB Postgres Advanced 14 で使用可能になる予定です。

第1章 Architectural Overview

BDR は、メッシュトポロジを使用した疎結合マルチマスターロジカル複製を提供します。つまり、任意のサーバーに書き込むことが可能で、変更は同じ BDR グループに属する他のすべてのサーバーに行ごとに直接送信されます。



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。

1.1 基本アーキテクチャ

1.1.1 複数のグループ

BDR ノードは少なくとも 1 つの **ノードグループ** のメンバであり、最も基本的なアーキテクチャでは、BDR クラスター全体に対して単一のノードグループがあります。

1.1.2 複数のマスター

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

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

常にすべてのマスターに手紙を書く必要はありません。ほとんどの場合、1 つのマスタにのみ書き込みを指示する構成が頻繁に行われます。ただし、一方向のレプリケーションが必要な場合は、`pglogical` の使用がより適切な場合があります。

1.1.3 非同期、デフォルト

1 つの BDR ノードで行われた変更は、ローカルでコミットされるまで他のノードに複製されません。その結果、データは常にすべてのノードで正確に同じではありません。一部のノードには、他のノードにまだ到着していないデータがあります。PostgreSQL のブロックベースのレプリケーションソリューションも、デフォルトで非同期レプリケーションになります。BDR では、マルチプルのマスターがあり、結果としてマルチプルのデータストリームがあるため、`synchronous_commit` と `synchronous_standby_names` が使用されている場合でも、異なるノード上のデータは異なる場合があります。

1.1.4 メッシュトポロジ

BDR は、すべてのノードが他のすべてのノードに接続し、すべてのノードが互いに直接データを交換するメッシュネットワークを中心に構成されています。ノードの追加や削除などの特別な状況を除き、BDR 内でのデータの転送はありません。データは、BDR クラスターの外部から到着するか、`pglogical` またはネイティブの PostgreSQL ロジカルレプリケーションを使用して先に送信されます。

1.1.5 論理複製

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

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

ノードは、ソースノードでコミットが行われたオーダーでデータを適用し、単一ノードからの変更に対してトランザクションの一貫性を保証します。異なるノードからの変更は、他のノードとは独立して適用され、変更の迅速なレプリケーションを保証します。

複製されたデータは、安全な場合はバイナリフォームで送信されます。

1.1.6 高可用性

各マスタノードは 1 つ以上のスタンバイノードで保護できるため、ダウンしたノードはすぐに交換して続行できます。各スタンバイノードは、ロジカルスタンバイノードまたは物理的スタンバイノードのいずれかです。

1 つ以上のノードが現在利用できない場合でも、現在接続されているノード間でレプリケーションが続行されます。ノードが回復すると、変更を失うことなく、レプリケーションを中断したところからリスタートできます。

ノードは異なるリリースレベルを実行して、通信に必要なプロトコルをネゴシエートできます。その結果、データベースソフトウェアのメジャーバージョンでも、BDR クラスタはローリングアップグレードを使用できます。

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

1.1.7 制限

BDR は 1 つのクラスターで最大 99 個のマスタノードでテストされていますが、現在は最大 32 個のマスタノードで使用するよう設計されています。各マスタノードは、マルチプル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定の制限はありませんが、一般的な使用法では、マスタごとに 2~3 個のスタンバイ、マスタごとに通常最大 32 個のスタンバイを使用します。

BDR は、タイムハードシーケンスを使用する場合、グループから以前に削除（分割/削除）されたノードをカウントせず、1024 ノード（マスタノードとロジカルスタンバイの両方をカウント）を超えないことを想定しています。

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。1 つのインスタンス内のマルチプルノード/データベースを同じ BDR ノードグループのパートにすることはサポートされていません。

1.2 アーキテクチャのオプションとパフォーマンス

1.2.1 BDR パフォーマンスの特性評価

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

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

次のアーキテクチャが利用可能です。

- マルチマスター/シングルグループ BDR AlwaysOn BDR ワールドワイド BDR オートスケール

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

1.2.2 1 つのグループ内の BDR MultiMaster

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

データのコピーはどこにでもあるので、SELECT はローカルノードにアクセスするだけで済みます。読み取り専用クラスターでは、1 つのノードのパフォーマンスはノードの数の影響を受けません。したがって、ノードを追加すると、可能な SELECT スループットの合計が直線的に増加します。

INSERT、UPDATE、および DELETE（DML）はローカルで実行され、変更はグループ内のすべてのノードに伝播されます。DML 適用のオーバーヘッドは元の実行より小さいため、マルチプルノードで純粋な書き込みワークロードを同時に実行すると、マルチノードクラスターは単一ノードよりも多くの TPS をハンドルできます。

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

積極的なレプリケーションは競合を回避できますが、本質的にはより高価です。

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

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

1.2.3 BDR AlwaysOn

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

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

1つのノードがメインアプリケーションのターゲットです。他のすべてのノードはシャドウノード（または「読み書きレプリカ」）として記述され、必要なときに引き継ぐのを待っています。nodelos のコンタクトが発生した場合、処理を続行するためにすぐにシャドウノードにスイッチします。aGroup に障害が発生した場合、他のグループにスイッチことができます。スケーラビリティはこのアーキテクチャの目標ではありません。

主に1つのノードのみに書き込むため、競合が発生する可能性はほとんどゼロになり、結果としてパフォーマンスへのインパクトは大幅に減少します。

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

セカンダリアプリケーションはシャドウノードに対して実行される場合がありますが、メインアプリケーションがそのノードの使用を開始した場合、これらを減らすか中断する必要があります。

将来の機能：1つのノードが他のグループのメインレプリケーターとして選択され、クラスターの成長に伴うレプリケーションのCPUオーバーヘッドが制限され、他のグループへの帯域幅が最小化されます。

1.2.4 BDR ワールドワイド

このアーキテクチャでは、マルチプルのBDRグループが世界中のマルチプルのリージョンに存在しますが、グループ内のすべてのノードは常に1つのリージョン内にあります。

ボリュームのテーブルが独自のグループ内でのみ複製されるように構成されている場合、ピーク書き込みTPSはグループの数に応じて線形にスケーリングされます。これにより、ジオシャーディングとも呼ばれる入力場所に基づいたシャーディングデータが可能になります。

これにより、このアーキテクチャは、場所がアプリケーションを自然に分割する方法であるIoT、モニタリング、またはその他の大量データ収集アプリケーションでの使用に非常に適しています。ボリュームのデータはそのリージョンを離れないため、リージョン間のレプリケーションによる高い帯域幅コストを回避できるだけでなく、データのプライバシーと管轄に関する強力な法律に従うことができます。

BDR は、このアーキテクチャでマルチノード読み取りクエリを許可します。大規模なマルチノードクエリは、ノード数が増加するにつれて、応答時間の観点から線形にスケーリングします。各クエリはすべてのサブクラスターの1つのノードで実行されるため、スループットは制限されます。

1.2.5 BDR オートスケール

このアーキテクチャでは、グループの配列を使用して並列計算プラットフォームを作成します。

デフォルトでは、すべてのノードにテーブルが作成されます。レンジテーブルパーティションの場合、ユーザはパーティションをグループ全体にストライピングして自動シャーディングを許可するディストリビューションキーを指定できます。たとえば、TimeStamp、CustomerId、WarehouseId。範囲の代わりに個別のキー値も使用でき、リストベースのパーティショニングを効果的にサポートすることに注意してください。

すべてのノードで同一のテーブルは、複製されたテーブルと呼ばれます。グループ間で分割されたテーブルは、分散テーブルまたは分割テーブルと呼ばれます。複製されたテーブルは、他のテーブルに結合できます。分散テーブルは、複製されたテーブル、またはまったく同じデータディストリビューションを共有する他の分散テーブルにのみ結合できます。

分散テーブルは常に完全に均等に分散されているとは限りません。頻繁にアクセスされるシャードがある場合、不均衡なワークロードに対処するためにそのグループにノードを追加できます。

このフォームのシャーディングにより、アプリケーションが直接ルーティングとして知られている適切なグループにトランザクションを直接送信する場合、OLTP 書き込みパフォーマンスは線形に拡張できます。このモードでは、Apache Cassandra などの分散 NoSQL システムと多少似たアーキテクチャを使用できます。自動スケールは、クラスターが拡張または縮小ニーズがあるときに問題を引き起こし、必要に応じて柔軟にスケーリングする機能を制限するため、ハッシュパーティションをサポートしません。

ワークロードがトランザクションを正しいグループに直接送信できない場合、それらを適切なグループ内の任意のノードにルーティングするコーディネーターノードに対して実行します。その後、コーディネーターはプロキシとして機能し、インダイレクトプロキシルーティングを実行します。各グループには、優先書き込みノードと他の 1-2 個のシャドウノードがあります。読み取りは、デフォルトでシャドウノードに向けられ、残りが優先ノードである場合は preferredwrite ノードに向けられます。

コーディネーターノードはレイテンシを追加し、コーディネーターで費やされる時間の割合が増加するにつれてスケーラビリティを制限できます。小さな読み取り専用要求が多いワークロードではコーディネーター時間が占める割合が高くなりますが、ラージ意思決定支援クエリではコーディネーター時間が占める割合が小さいため、最適なスケーリングを行います。

シャードルーティングには、BDRcatalog テーブル内に保持されているシャード配布メタデータへのアクセスが必要です。コーディネーターノードはこの情報をカタログテーブルに保存するため、直接ローカルアクセスが可能です。シャード配布メタデータは、クライアント側プログラム内またはルーティングミドルウェア内にキャッシュして、これらのコンポーネントから直接ルーティングを実行できるようにすることもできます。

AutoScale は、このアーキテクチャでマルチノード読み取りクエリを許可します。大きなマルチノードクエリは、ノードの数が増えるにつれて、応答時間の観点から線形にスケーリングします。各クエリーはすべてのサブクラスターの 1 つのノードで実行されるため、スループットは制限されます。

AutoScale は、aCoordinator を使用する場合のマルチノード書き込みトランザクションをまだサポートしていませんが、これらは将来のリリースでサポートされる予定です。今のところ、これらの操作は受け入れられますが、アトミックではありません。マルチノード書き込みトランザクションは、シャードアーキテクチャのスケーラビリティを制限します。

1.3 展開

BDR3 は、TPAexec または構成管理アプローチのいずれかと、テクニカルサポートによって承認された展開アーキテクチャを使用して、少数の既知の適切な構成のいずれかに展開することを目的としています。

手動展開は推奨されておらず、サポートされていない場合があります。

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

現在、ログメッセージと文書は英語でのみ利用可能です。

1.4 クロックとタイムゾーン

BDR は、マルチプルのタイムゾーンのノードで動作するように設計されており、ほぼ全世界のデータベースクラスタを可能にします。個々のサーバーをマッピングタイムゾーンで構成する必要はありませんが、人間が読めるサーバーログがよりアクセスしやすく比較しやすいように `log_timezone = UTC` を使用することをお勧めします。

サーバークロックは、NTP または他のソリューションを使用して同期する必要があります。

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

第2章 Application Usage

この章では、アプリケーションまたはユーザの観点から BDR を検討します。

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

2.1 アプリケーションの動作

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

デフォルトでは、BDR は INSERT、UPDATE、DELETE、TRUNCATE からのすべての変更をソースノードから他のノードに複製します。すべてのトリガーとルールが処理された後、最終的な変更のみが送信されます。例、INSERT ... ON CONFLICT UPDATE は、オリジンで発生した内容に応じて INSERT または UPDATE を送信します。UPDATE または DELETE がゼロ行に影響する場合、変更は送信されません。

INSERT は、前提条件なしで複製できます。

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

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

INSERT、UPDATE、および DELETE コマンドによって暗黙的に取得された行レベルのロックは、変更が行われると複製されます。INSERT、UPDATE、DELETE、および TRUNCATE コマンドによって暗黙的に取得されたテーブルレベルのロックも複製されます明示的行レベルロック (SELECT ... FOR UPDATE / FOR SHARE) はユーザセッションによって複製されず、アドバイザリロックも複製されません。SERIALIZABLE モードで実行されているトランザクションによって保存された情報は、他のノードに複製されません。SERIALIZABLE のトランザクション隔離レベルはサポートされていますが、マルチプルのノードで同時トランザクションが存在する場合、トランザクションはノード間でシリアル化されません。

DML がマルチプルのノードで同時に実行される場合、非同期レプリケーションで実行する場合、潜在的な競合が発生する可能性があります。これら処理または回避する必要があります。さまざまな回避メカニズムが可能です。Conflicts の章でも説明されています。

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

BYTEA 列のバイナリデータは通常どおり複製され、最大 1GB のデータの「ブロブ」が許可されます。PostgreSQL の「ラージオブジェクト」機能の使用は、BDR ではサポートされていません。

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

レプリケーションは、ベーステーブルからベーステーブルにのみ可能です。つまり、サブスクリプション側のソースおよびターゲットのテーブルは、ビュー、マテリアライズドビュー、または外部テーブルではなく、テーブルでなければなりません。ベーステーブル以外のテーブルをレプリケートしようとすると、エラーが発生します。更新可能ビューを介して行われた DML 変更は、オリジンの tobase テーブルを介して解決され、ターゲットの同じベーステーブル名前に適用されます。

BDR はテーブルパーティションを透過的にサポートします。つまり、パーティションテーブルをレプリケーションセットに追加でき、パーティションに関する変更がダウンストリームにレプリケートされます。

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

一部のタイプのトリガーは、atable に存在し、現在有効になっている場合でも、適用時に実行されません。実行されないトリガータイプは

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

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

BDR は、テキストまたはその他のデフォルト可能なデータ型に関連する問題がないことを前提としていることに注意して照合順序。変更のレプリケーションでは、等値検索を使用してレプリカ ID 値を特定します。したがって、一致しない照合順序修飾子で一意性インデックスが明示的に定義されている場合を除き、これは効果がありません。照合可能な式が使用されている場合、行フィルターは照合の違いの影響を受ける可能性があります。

PostgreSQL 内の非常に長い「トースト」データの BDR ハンドリングは、ユーザに対して透過的です。TOAST の「chunkid」値は、異なるノードの同じ行で異なる可能性が高いことに注意してください。ただし、問題は発生しません。

レプリカ ID 列が「外部」としてマークされている場合、BDR は正しく機能できません。

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

BDR は外部キーの使用を制限しません。カスケード FK は許可されます。

BDR は現在、非 ASCII スキーマまたはリレーション名の使用をサポートしていません。後のバージョンではこの制約がなくなります。

2.2 複製されていないステートメント

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

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

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

2.3 DML および DDL レプリケーション

BDR は DML ステートメントを複製せず、DML ステートメントによって引き起こされた変更を複製することに注意してください。したがって、例、2つの行を変更した UPDATE は2つの変更を複製しますが、行を削除しなかった DELETE は何も複製しません。これは、揮発性ステートメントの実行結果が複製されることを意味し、ステートメントベースのレプリケーションで発生する可能性のあるノード間の相違がないことを保証します。

DDL レプリケーションの動作は DML とは異なります。DDL の場合、BDR はステートメントを複製し、それがすべてのノードで実行されます。そのため、DROP TABLE IF EXISTS はローカルノードで何も複製しない場合がありますが、DDL レプリケーションが有効になっている場合、ステートメントはまだ他のノードに送信されて実行 DDL レプリケーションします。

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

2.4 異なるリリースレベル間の複製

BDR は、異なるメジャーバージョンの PostgreSQL を持つノード間で複製するように設計されています。これは、ダウンタイムなしでメジャーバージョンをアップグレードできるように設計された機能です。

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

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

上記の機能は両方とも特定の制限の影響を受ける可能性があります。既知の非互換性については、リリースノートで説明します。

2.5 相違点のあるノード間の複製

デフォルトでは、DDL はすべてのノードに自動的に送信されます。これは、[DDL Replication](#) で説明されているように、手動で制御できます。これは、ノード間でデータベーススキーマの違いを作成するために使用できます。BDR は、ノード間にわずかな違いがある場合でもレプリケーションを続行できるように設計されています。これらの機能は、ダウンタイムなしでアプリケーションスキーマを移行したり、レポートまたはテスト用に論理スタンバイノードを許可したりするように設計されています。

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

ターゲットのパーティションを変更する更新のサポートを含む、分割されたテーブルに複製する通常のテーブルであるソースなど、異なるパーティション定義を持つテーブル間で複製することが可能です。ソースとターゲットのパーティショニング定義が同じ場合、適用時に動的パーティションルーティングを実行する必要がないため、高速になります。詳細については、レプリケーションセットの章を参照してください。

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

BDR は、列の数が異なるテーブル間の複製をサポートしています。

ターゲットにソースからの列が欠落している場合、BDR は `target_column_missing` の競合を発生させます。この競合については、デフォルトの競合が `ignore_if_null` を解決します。NULL 以外の値が到着した場合、これは **ERROR** をスローします。または、ノードは競合リゾルバ `ignore` を使用して構成することもできます。この設定は **ERROR** をスローせず、単に追加列を無視します。

ターゲットにソースレコードにない追加の列がある場合、BDR は `source_column_missing` の競合を発生させます。この競合のデフォルトの競合は `use_default_value` です。追加の列にデフォルト、NULL (null 可能の場合) またはデフォルト式がある場合、レプリケーションは続行しますが、そうでない場合はエラーをスローし、レプリケーションを停止します。

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

ソースとターゲットのコンストレインが異なる場合、レプリケーションが試行されますが、ソースからの行をターゲットに適用できない場合は失敗する可能性があります。ここで行フィルターがヘルプ場合があります。

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

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

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

複製されるテーブルは、各ノードの同じユーザ/ロールが所有する必要があります。詳細については、Security and Roles を参照してください。

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

2.6 違いのあるノード間の比較

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

Livecompare 1.3 から、all_bdr_nodes を設定して構成できました。これにより、クラスター内の各ノードに関連するすべての DSN を明確にする必要がなくなります。BDR クラスターには、接続情報を持つ N 個のノードがありますが、livecompare 1.3+ がジョブを完了するためにニーズ初期接続と出力接続のみです。logical_replication_mode を設定すると、すべてのノードの通信方法が示されます。

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

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

このツールには、多くのカスタマイズとフィルターがあります。テーブル、スキーマ、replication_sets など。LiveCompare は、コンテキスト情報を失うことなく stop-start を使用できるため、都合の良いときに実行できます。比較後、ユーザが確認できるように、要約と DML スクリプトが生成されます。見つかった違いがある場合は、修正するために DML を適用してください。

2.7 アプリケーションの一般規則

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

同様に、BDR は変更をリプレイするテーブルを識別するためにテーブル名を使用するため、同じオブジェクト名を作成してからドロップし、後で再利用するアプリケーションにも同様の ABA 問題が存在します。

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

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

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

2.8 タイミングの考慮事項と同期複製

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

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

Postgres の同期レプリケーション機能は、BDRas でも利用できます。さらに、BDR は、より多くの同期複製のためにマルチプルのバリエーションを提供します。利用可能なすべてのバリエーションとその異なるモードの概要と比較については、[Durability & Performance Options](#) chapter を参照してください。

2.9 アプリケーションのテスト

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

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

2.9.1 TPAexec

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

TPAexec には、各リファレンスアーキテクチャのテストスイートが含まれています。また、次の例のような構文を使用して、TPA クラスターに対して実行されるテストのローカルコレクションの作成と管理を簡素化します。

```
tpaexec test mycluster mytest
```

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

2.9.2 CAMO /フェイルオーバーオプションを備えた pgbench

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

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

--retry
retry transactions on failover
```

上記のオプションに加えて、フェイルオーバーのピアノードに関するノード情報を [DSNform](#) で指定する必要があります。

- `-m camo` または `-m failover` を使用して、pgbench のモードを指定します。`-m failover` 仕様を使用して、フェイルオーバーをテストできます。通常の BDR 展開。
- `--retry` を使用して、次の場合にトランザクションを再試行するかどうかを指定します。フェイルオーバーは `-m failover` モードで発生します。これはデフォルトで有効になっています `-m camo` モードの場合。

CAMO 環境での呼び出しの例を次に示します。

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

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

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

2.9.3 マルチノードアクセスの分離テスター

isolationtester は、ユーザーが複数のセッションとマルチプルのノードでテストを実行できるように拡張しました。これは内部 BDR テストに使用されますが、ユーザアプリケーションテストで使用することもできます。

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

分離テストは、PostgreSQL の同時動作を調べるために実行される一連のテストです。これらのテストでは、マルチプルの相互作用するトランザクションを実行する必要があり、マルチプルの同時接続を管理する必要があります。そのため、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隔離レベルをテストすることであったという事実由来しています。しかし、他のソートの並行動作のテストも追加されています。

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

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

重複するトランザクションでテストを定義するには、次のセクションで説明するカスタム構文のテスト仕様ファイルを使用します。新しいテストを追加するには、`specs` /サブディレクトリに `spec` ファイルを配置し、`expected` /サブディレクトリに `expectedoutput` を追加し、メークファイルにテストの名前を追加します。

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

仕様は、次のオーダーでテストされた5つの部分で構成されています。

`server "<name>"`

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

`setup { <SQL> }`

- 指定された SQL ブロックは、テストを実行する前に、1つのセッションでのみ1回実行されます。ここでテストテーブルまたはその他の必要なオブジェクトを作成します。このパートはオプション。必要に応じて、複数のセットアップブロックを使用できます。それぞれは、指定されたオーダーで

個別に実行されます。(マルチプルのセットアップブロックを許可する理由は、各ブロックが単一の PQexec 送信として実行され、VACUUM などの一部のステートメントがそのようなブロック内の他のステートメントと結合できないためです。)

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

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

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

- 通常、spec ファイルにはいくつかの「セッション」部分があります。各セッションは、独自の接続で実行されます。セッションパートは、セットアップ、分解、および 1 つ以上の「ステップ」の 3 つの部分で構成されます。セッションごとのセットアップとティアダウンの部分の構文は、上記のテストごとのセットアップとティアダウンと同じですが、各セッションで実行されます。通常、セットアップパートには、トランザクションを開始するための「BEGIN」コマンドが含まれています。
- さらに、セッションパートも `connect_to` 仕様で構成されます。これは、このセッションが実行されるサーバーを示す最初に指定されたサーバー名前を指します。
- `connect_to "<name>"`
- 各ステップには構文があります
- `step "<name>" { <SQL> }`
- ここで、<name>はこのステップを識別する名前であり、SQL はステップで実行される SQL ステートメント（またはセミコロンで区切られたステートメント）です。ステップ名は、spec ファイル全体で一貫である必要があります。

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

- 順列行は、そのオーダーで実行されるステップのリストを指定します。順列行はいくつでも表示できます。順列が指定されていない場合、テストプログラムは各セッションからステップの可能なすべての順序付けを自動的に生成します（1 つのセッションのステップをオーダーに実行します）。手動で指定された「順列」行のステップのリストは、実際には使用可能なステップの順列である必要はないことに注意してください。インスタンス、いくつかの手順を複数回繰り返したり、他の手順を省略したりできます。

#で始まる行はコメントと見なされます。

セッションステップの順列（仕様ファイルで手動で指定されるか、自動生成されるか）ごとに、`isolationtester` はメインセットアップパート、セッションごとのセットアップパート、選択されセッションステップ、セッションごとの分解を実行します。maintardown スクリプト。選択された各ステップは、そのセッションに関連付けられた接続に送信されます。

すべての前提条件 `make` コマンドを実行した BDR3 環境で分離テストを実行するには、次の手順に従います。

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

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

A. `isolationcheck-installcheck` を実行するには、2 つ以上の `postgreslserver`s を実行する必要があります。サーバーの `conninfo` を BDR 3.0 メークファイルの `pg_isolation_regress` に渡します。例：`pg_isolation_regress --server 'd1=host=myhost dbname=mydb port=5434' --server 'd2=host=myhost1 dbname=mydb port=5432'`

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

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

B. `Isolationcheck-makecheck` は現在、マルチプルのデータベース間で BDR を設定することにより、単一インスタンスでの分離テストの実行をサポートしています。

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

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

各ステップには、さらにアクションが実行されるまでブロックするコマンドが含まれる場合があります（おそらく、他のセッションはブロックを解除するかデッドロックを引き起こすステップを実行します）。この機能を使用するテストでは、有効な置換、つまりブロックされたセッションがコマンドを実行ことを期待しないものを手動で指定する必要があります。テストがそのルールに従わない場合、`isolationtester` は 300 秒後にキャンセルします。キャンセルが機能しない場合、合計 375 秒の待機時間の後、`isolationtester` は異常終了します。無効な置換のテストは、分離テストの実行に非常に長い時間がかかる可能性があり、有用なテスト目的に役立たない `make`、避ける必要があります。

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

2.10 パフォーマンスのテストとチューニング

BDR を使用すると、マルチプルのマスタノードに書き込みトランザクションを発行できコスト。これらの書き込みを各ノードに戻すと、パフォーマンスが低下することに注意してください。

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

次に、変更を再生すると、ローカルワークロードに対する競合を引き起こす可能性のあるテーブルレベルおよび行レベルロックが保持されます。CRDT（競合のないレプリケートされたデータ型）および CLCD（列レベルの競合検出）機能により、同時更新でも正しい答えが得られますが、通常のロックオーバーヘッドは削除されません。ロックの競合が発生する場合は、更新の競合を回避するか、トランザクションをできるだけ短くしてください。大きなトランザクション内で頻繁に更新される行は、そのトランザクション

のパフォーマンスにボトルネックを引き起こします。複雑なアプリケーションでは、スケーラビリティを維持するためにいくつかの考えが必要です。

パフォーマンスに問題があると思われる場合は、上記のベンチマークツールを使用してパフォーマンステストを開発することをお勧めします。pgbench を使用すると、ユースケースに固有のカスタムテストスクリプトを記述できます SQL のオーバーヘッドを理解し、同時実行の影響を測定できます。

したがって、「BDR の実行速度が遅い」場合は、次のことをお勧めします。

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

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

2.11 適合性の評価

BDR は PostgreSQL と互換性がありますが、すべての PostgreSQL アプリケーションが分散データベースでの使用に適しているわけではありません。ほとんどのアプリケーションはすでに、または簡単に変更して BDR 準拠にすることができます。ユーザーは、自分のアプリケーションを BDR 対応セットアップに hướngることができる評価アクティビティを実行できます。BDR は、評価期間中に設定できるいくつかのノブを提供します。これらは、BDR 対応環境でのアプリケーションの適合性を判断するプロセスを支援します。

2.11.1 主キー/レプリカ ID の更新の評価

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

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

これらの構成パラメーターは、評価のみに使用する必要があることに注意してください。これらは単一ノード BDR インスタンスで使用できますが、2つ以上のノードが相互に複製されている production 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、ノードのスタートに失敗するか、新しいノードがクラスターに結合できません。

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

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

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

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

適用ワーカプロセスは、このパラメータの設定を常に無視します。

2.11.2 テーブルまたは SELECT クエリでの LOCK の使用の評価

BDR ライタプロセスは通常のユーザセッションとほとんど同じように動作するため、行とテーブルのロックに関する通常のルールにサブジェクトます。これにより、BDR ライタプロセスがユーザトランザクションによって、または相互に保持されているロックを待機する場合があります。

EDB Postgres Extended で実行する場合、BDR は次の構成パラメータを提供して、アプリケーションが明示的なロックを取得しているかどうかを評価します。

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

追跡できるロックには次の 2 種類があります。

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

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

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

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

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

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

第3章 PostgreSQL Configuration for BDR

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

3.1 BDR の PostgreSQL 設定

BDR を正しく実行するには、次の PostgreSQL 設定が必要です。

- `wal_level`- BDR はロジカルデコーディングに依存するため、`logical` に設定する必要があります。
- `shared_preload_libraries`-`pglogical`, `bdr` を (このオーダーで) 含める必要があります。ただし、必要に応じて、前後に他のエントリを含めることもできます。
- `track_commit_timestamp`-競合を解決するには、「on」に設定する必要があります競合する行のタイムスタンプを取得します。

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

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

N 個の同等なを持つグループの通常の実行では、BDR には N 個の slots と WAL センダが必要になることに注意してください。同期中、BDR は another N-1 slots と WAL センダを一時的に使用するため、この時々のピーク需要に対応するために上記のパラメータを十分に高く設定するよう注意してください。

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

Decoding Worker が有効になっている場合、このプロセスには 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使用する 4 ノード BDR グループの一般に安全な推奨値は、`max_replication_slots` および `max_worker_processes` を 50 および `max_wal_senders` に少なくとも 10 に設定することです。

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

アプリケーションでは、これらのパラメーターを設定することもできます。詳細については、[耐久性とパフォーマンスオプション] の章を参照してください。

- `synchronous_commit`- BDR レプリケーションの耐久性とパフォーマンスに影響します [physical replication](#) と同様の方法で。
- `synchronous_standby_names`-上記と同じ

3.2 BDR の 2ndQPostgres / EDB Postgres 拡張設定

次の Postgres 設定は、最大 2 回のコミット (CAMO) で考慮する必要があります。これは、2ndQPostgres との BDR の組み合わせでのみ使用可能な機能です。これらのいくつかは、2ndQPostgres でのみ利用可能です。他のコミュニティバージョンには既に存在しますが、CAMO と組み合わせて BDR にのみ関連するようになります。

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

3.3 BDR の pglogical 設定

BDR は、内部で `pglogical` を使用して基本的なレプリケーションを実装するため、`pglogical` 設定の一部の影響も受けます。

- `pglogical.track_subscription_apply`-の適用統計を追跡各サブスクリプション。
- `pglogical.track_relation_apply`-それぞれの適用統計を追跡リレーションの
- `pglogical.track_apply_lock_timing`-追跡時の追跡ロックタイミングリレーションの統計。
- `pglogical.standby_slot_names`-物理的スタンバイノードを使用する場合フェイルオーバーを目的とするため、レプリケーションに設定する必要があります対象となる各スタンバイのスロット。
- `pglogical.writers_per_subscription`-デフォルトのライターあたりの数サブスクリプション (BDR では、これも変更できますグループの場合は `bdr.alter_node_group_config`)。
- `pglogical.max_writers_per_subscription`-ライターの最大数サブスクリプションごと (上記の設定の上限を設定)。

3.4 BDR 固有の設定

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

3.4.1 競合処理

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

3.4.2 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。

3.4.3 DDL 処理

- `bdr.default_replica_identity`-REPLICA IDENTITY のデフォルト値を設定します新しく作成されたテーブル。REPLICA IDENTITY は、どの情報が更新または削除された行を識別するために、ログ先行書き込みに書き込まれます。
 - 受け入れられる値は次のとおりです。
 - `-DEFAULT`-プライマリキーの列の古い値があれば、それを記録します（これはデフォルトの PostgreSQL の動作です）。`-FULL`-行のすべての列の古い値を記録します。`-NOTHING`-古い行に関する情報を記録しません。
 - 詳細については、[PostgreSQL documentation](#) を参照してください。
 - BDR は、テーブル固有の構成または `bdr.default_replica_identity` を介して、テーブルのレプリカアイデンティティが `FULL` でない限り、PRIMARY KEY または UNIQUE 制約のないテーブルで `UPDATEs` および `DELETEs` を複製できません。
 - `bdr.default_replica_identity` が `DEFAULT` であり、テーブルに `UNIQUE` 制約がある場合、`REPLICA IDENTITY` として自動的に選択されません。テーブルの作成時に、または上記の文書で説明されているように、後で明示的に設定ニーズがあります。
 - テーブルのレプリカアイデンティティを `FULL` に設定すると、書き込まれる WAL のボリュームと、テーブルのワイヤ上で複製されるデータの量が増加します。
- `bdr.ddl_replication`-ノード間で DDL を自動的に複製します（デフォルト「オン」）。
 - このパラメータは、`bdr_superuser` ロールまたはスーパーユーザロールによってのみ設定できます。
 - `bdr.ddl_replication = off` を使用して DDL を実行するか、BDR 管理関数を呼び出すと、管理者が介入できるまでレプリケーションが停止する状況が発生する可能性があります。詳細については、[the DDL replication chapter](#) を参照してください。

- `bdr.ddl_replication` が `off` に設定されると、LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定により、キャプチャされた DDL コマンドまたは BDR レプリケーション機能のレプリケーションがスキップされたときに、`WARNING-level` メッセージが書き込まれます。
- `bdr.role_replication`-ノード間で `ROLE` コマンドを自動的に複製します (デフォルトは「オン」)。このパラメータは、スーパーユーザのみが設定可能です。この設定 `bdr.ddl_replication` もオンになっている場合にのみ機能します。
 - 外部メソッドを使用せずにこれをオフにして、保証のノード間でロールを同期させると、管理者が介入するまで複製 DDL がレプリケーションを中断する可能性があります。
 - 詳細については、[Role manipulation statements in the DDL replication chapter](#) を参照してください。
- `bdr.ddl_locking`-DDL のグローバルロックのオペレーションモードを構成します。
 - このパラメータは、`bdr_superuser` ロールまたはスーパーユーザロールによってのみ設定できます。
 - 可能なオプションは次のとおりです。
 - オフ-DDL 操作にグローバルロックを使用しない-オン-すべての DDL 操作にグローバルロックを使用する DDL レプリケーションのグローバル DML ロックを取得して書き込みを防止する必要がある DDL 操作にのみグローバルロックを使用する
 - `bdr.ddl_replication` が `off` に設定されると、LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定が原因でグローバルロック手順がスキップされたときに、`WARNING` メッセージが書き込まれます。一部のステートメントでは、2 つの `WARNING` が発生するのが正常です。1 つは DML ロックをスキップし、もう 1 つは DDL ロックをスキップします。
- `bdr.truncate_locking`-デフォルトでは `False`、この構成オプションは `TRUNCATE` コマンドのロック動作。(真の場合) `TRUNCATE` するかどうかを決定します `bdr.ddl_locking` 設定に従います。

3.4.4 グローバルロック

- `bdr.ddl_locking`-上記で説明。
- `bdr.global_lock_max_locks`-可能なグローバルロックの最大数ノードで保持されます (デフォルトは 1000)。Postgres サーバーのスタートにのみ設定できます。
- `bdr.global_lock_timeout`-待機の最大許容期間を設定しますグローバルロックの場合 (デフォルトは 1 分)。値がゼロの場合、このタイムアウトは無効になります。
- `bdr.global_lock_statement_timeout`-の最大許容期間を設定しますグローバルロックを保持するステートメント (デフォルトは 10 分)。値がゼロの場合、このタイムアウトは無効になります。
- `bdr.global_lock_idle_timeout`-最大許容持続時間を設定しますグローバルロックを保持しているトランザクションのアイドル時間 (デフォルトは 10 分)。値がゼロの場合、このタイムアウトは無効になります。

3.4.5 ノード管理

- `bdr.replay_progress_frequency`-レプリケーション位置を送信する間隔残りのクラスターへの情報（デフォルトは1分）。

3.4.6 汎用レプリケーション

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

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

!!! Warning * `bdr.xact_replication` をオフにすると、ノード間でデータの不整合が発生します。ノード間のデータの相違から回復する場合、またはレプリケーションを続行するには単一ノードの変更が必要なレプリケーション状況でのみ使用してください。ご自身のリスクで使用してください。

- `bdr.permit_unsafe_commands`-コマンドの安全性チェックを無効にするオプション
 - 一般的な使用には安全でないと見なされます。
 - `bdr_superuser` または PostgreSQL スーパーユーザが必要です。

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

- `bdr.batch_inserts`-内の1つのテーブルへの連続挿入数単一のトランザクションは、そのテーブルの挿入のバッチ処理をオンにします。
 - このオプションを使用すると、挿入のセットではなく、ラージのデータを内部的に COPY としてレプリケーションできます。また、ノード結合中の初期データのコピー方法も示します。
- `bdr.maximum_clock_skew`
 - これにより、`bdr.maximum_clock_skew_action` をトリガーする前に、着信トランザクションコミットタイムスタンプとサブスクライバーの現在の時刻との最大の差と見なされる値が指定されます。
 - これにより、現在再生されているトランザクションのタイムスタンプが、サブスクライバーの現在の時刻と比較して将来にあるかどうかを確認されます。そして、その差が `bdr.maximum_clock_skew` よりも大きい場合、`bdr.maximum_clock_skew_action` 設定で指定されたアクションを実行します。

デフォルトは-1 です。つまり、クロックスキューを無視します（チェックはオフになります）。すべてのサーバーのクロックが同期されているときに 0 を設定することは有効です。トランザクションを再生しているという事実は、過去にコミットされたことを意味します。

- `bdr.maximum_clock_skew_action`
 - これは、`bdr.maximum_clock_skew` よりも高いクロックスキューが検出された場合に実行するアクションを指定します。
 - このオプションには 2 つの可能な値があります。
 - `-WARN`-この事実に関するワーニングを記録します。サーバーログのフラッディングを防ぐため、警告は最大で毎分 1 回（デフォルト）記録されます。`-WAIT`-現在のローカルタイムスタンプが、リモートコミットタイムスタンプから `bdr.maximum_clock_skew` を引いたものよりも古くなるまで待機します。

3.4.7 CRDT

- `bdr.crdt_raw_value`-[CRDT Data Types](#) の出力フォーマットを設定します。デフォルトの出力（この設定が結果の場合）は、現在の基本 CRDT タイプの値（例、`crdt_pncounter` の `bigint`）。`on` に設定すると、戻り値は CRDT 値。例、マルチプルのノードからの状態を含めることができます。

3.4.8 最大準備トランザクション

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

3.4.9 熱心な複製

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

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

3.4.10 最大 1 回コミットする

- `bdr.enable_camo`-CAMO 機能を有効にして制御するために使用されます。デフォルトは `off` です。CAMO はトランザクションごとにオンに切り替えることができますこれを `remote_write`、`remote_commit_async` に設定する、または `remote_commit_flush`。下位互換性のために、値 `on`、`true`、および `1` は、最も安全な `remote_commit_flush` モードを設定します。 `false` または `0` も CAMO を無効にします。
- `bdr.camo_partner_of`-データベースごとに CAMO パートナーを指定できます。コロンで結合されたデータベース名前とノード名前のペアが必要です。複数ペアを指定できますが、データベースごとに最初に現れるもののみがアカウントされます。例: `'db1:node_4 test_db:test_node_3'`。Postgres サーバーのスタートにのみ設定できます。
- `bdr.camo_origin_for`-オリジンのデータベースごとのノード名前 CAMO ペアリングのトランザクション。データベースごとに、これはマッチニーズがあります対応するオリジンノードで `bdr.camo_partner_of` 設定を使用します。Postgres サーバーのスタートにのみ設定できます。
- `bdr.standby_dsn`-接続のマニュアルオーバーライドを許可します以来変更された場合に CAMO パートナーに到達するための文字列 (DSN) ローカルノードのクラッシュ。通常は設定しないでください。Postgres サーバーのスタートにのみ設定できます。
- `bdr.camo_local_mode_delay`-適用されるコミット遅延通常発生するオーバーヘッドをエミュレート CAMO のローカルモード CAMO パートナーはトランザクションを確認する必要があります。デフォルトは 5 ミリ秒です。0 に設定すると、この機能が無効になります。
- `bdr.camo_enable_client_warnings`-アクティビティが CAMO プロパティができないデータベースで実行保証されています。これはデフォルトで有効になっています。十分な情報に基づいたユーザーが選択できますこれを無効にして、ログに記録される警告の量を減らします。

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

3.4.11 タイムスタンプベースのスナップショット

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

3.4.12 監視とログ

- `bdr.debug_level`-BDR が書き込みに使用するログレベルを定義しますデバッグメッセージ。デフォルト値は `debug2` です。見たいなら詳細な BDR デバッグ出力、`bdr.debug_level = 'log'` を設定します。
- `bdr.trace_level`-上記と同様に、これはログレベルを定義します BDR トレースメッセージに使用します。のすべてのノードでトレースを有効にする BDR クラスターは、2ndQuadrant サポートが問題を診断するのにヘルプます。Postgres サーバーのスタートにのみ設定できます。

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

3.4.13 内部

- `bdr.raft_keep_min_entries`-保持するエントリの最小数ログ圧縮を行うときにログをラフティンクする（デフォルトは 100）。値 0 は無効になりますログ圧縮。 **警告：ログの圧縮が無効になっている場合、ログは永遠にサイズが大きくなります。** Postgres サーバーのスタートにのみ設定できます。
- `bdr.raft_response_timeout`-ネットワーク障害をアカウントして、実装されたラフトコンセンサスプロトコルは、一定の時間。このタイムアウトのデフォルトは 30 秒です。
- `bdr.raft_log_min_apply_duration`-ステートマシンを移動するには転送すると、Raft は内部ログにエントリを追加します。通常時オペレーション、追加には数ミリ秒しかかかりません。これはその追加アクションの期間の上限しきい値。 **INFO** メッセージが記録されます。これは、実際の問題。このパラメータのデフォルト値は 3000 ミリ秒です。
- `bdr.raft_log_min_message_duration`-コンセンサスリクエストをログするタイミング。ログコンセンサスリクエストの往復時間を測定し、時間がこのパラメータを超える場合の **INFO** メッセージ。デフォルト値このパラメータの最大値は 5000 ミリ秒です。
- `bdr.raft_group_max_connections`-接続の最大数 Postgres サーバーのすべての BDR グループ全体。これらの接続は運ぶグループのノード間の `bdr` コンセンサスリクエスト。このデフォルト値パラメータは 100 接続です。Postgres サーバーのスタートにのみ設定できます。
- `bdr.backwards_compatibility`-バージョンを指定します下位互換性があり、同じ数値フォーマットで `bdr.bdr_version_num`、たとえば 30618。の正確な動作を可能にします以前の BDR バージョン、これが一般に望ましくない効果をもたらす場合でも。デフォルトは現在の BDR バージョンです。これはリリースから変更されるためリリースするには、構成内で明示的に使用しないことをお勧めします値が現在のバージョンと異なる場合を除き、ファイル。
- `bdr.track_replication_estimates`-レプリケーションの見積もりを用語で追跡する同等な適用率とキャッチアップ間隔。この情報は CAMO などのプロトコルで使用して、同等なノード。このパラメータはデフォルトで有効になっています。（EDB Postgres Extended）
- `bdr.lag_tracker_apply_rate_weight`-同等なからの遅れをモニタしますローカルノードから WAL を適用するという点で、移動を計算します遅延追跡の適用率の平均。このパラメータは、この移動平均における新しい計算値の寄与度計算。デフォルト値は 0.1 です。（EDB Postgres Extended）

第4章 Node Management

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

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

BDR グループには、0 個以上のサブグループを含めることもでき、さまざまな異なるアーキテクチャを作成できます。

4.1 BDR グループの作成と参加

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

SQL ファンクション `bdr.create_node_group()` を使用して、ローカルノードから BDR グループを作成します。これにより、そのノードで BDR がアクティブになり、他のノードが BDR グループ（その時点で1つのノードのみで構成される）に結合できるようになります。作成時には、他のノードがこのノードへの接続に使用する接続文字列を指定する必要があります。

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

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

`bdr_init_physical` ユーティリティは、BDR1 および BDR2 の `bdr_init_copy` ユーティリティの機能を置き換えます。これは、`pglogical pglogical_create_subscriber` ユーティリティに相当する BDR3 です。

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

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

結合プロセス中に、BDR 拡張機能は、提供されたソースノードをベースとして使用して既存のデータを同期し、BDR グループのメッシュトポロジで自分自身を確立するために必要なすべてのメタデータ情報を作成します。この初期コピー中にソースと新しいノード間の接続が切断された場合、結合プロセスを最初から再開する必要があります。

クラスターに参加しているノードには、BDR グループのデータベースに既に存在するスキーマまたはデータを含めることはできません。BDR と pglogical 拡張を除き、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。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を確認してください。ただし、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。

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

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

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

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

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

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

4.1.1 異種クラスターへの参加

BDR 3.7 ノードは、特定の最小メンテナンスリリース（3.6.25 など）または 3.6 ノードと 3.7 ノードの混合で 3.6.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 BDRmajor バージョンだけでなく基礎となるバージョンもアップグレードする場合に役立ちます。PostgreSQL（または 2ndQPostgres）のメジャーバージョン。これは、PostgreSQL（または 2ndQPostgres）12 または 13 で実行される 3.7 ノードを、PostgreSQL（または 2ndQPostgres）11 で 3.6.x を実行する BDR クラスターに参加させることで実現できます。もちろん、新しいノードは、すべての PostgreSQL メジャーリリースで実行することもできます既存のクラスター内のノードの

一部のノードが PostgreSQL メジャーリリースで 3.6 を実行し、他のノードが別の PostgreSQL メジャーリリースで 3.7 を実行している場合でも、BDR はレプリケーションがすべての方向で正しく機能することを保証します。ただし、十分な数の新しいノードがクラスターに参加したら、ユーザは古いノードを切り離すことで、クラスターを迅速に不均質な状態にすることをお勧めします。古いバージョンでは使用できない DDL を実行しないように注意する必要があります。逆の場合も同様です。

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

3.7 ノードが 3.6 ノードをソースとして使用してクラスターに参加する場合、競合解決構成などの特定の構成はソースノードからコピーされないことに注意してください。ノードは、クラスターに参加した後に構成する必要があります。

4.1.2 接続 DSN と SSL (TLS)

ノードは `libpq` を使用して接続するため、ノードの DSN は単純に `libpq` 接続文字列です。そのため、許可された `libpq` 接続パラメーター（SSL の接続パラメーターを含む）を含めることができます。DSN は、指定されたノードに接続するクライアントからの接続文字列として機能する必要があることに注意してください。クライアント証明書を使用したこのような一連のパラメーターの例を次に示します。

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

このセットアップでは、ファイル `bdr_client.crt`、`bdr_client.key`、`root.crt` が各ノードのデータディレクトリに適切な権限で存在する必要があります。`verify-full` モードの場合、サーバーの SSL 証明書がチェックされ、`root.crt` CertificateAuthority で直接または間接的に署名されます。接続で使用されるホスト名前またはアドレスが証明書の内容と一致すること。名前の場合、これは SubjectAlternative Name にマッチするか、証明書にそのような名前がない場合、Subject の Common Name (CN) フィールドに一致します。Postgres は現在 IP アドレスのサブジェクトの別名をサポートしていないため、接続が名前ではなくアドレスによって行われる場合、CN フィールドと一致する必要があります。

クライアント証明書の CN は、BDR 接続を行うユーザの名前である必要があります。これは通常、ユーザ `postgres` です。各ノードには、`pg_hba.conf` ファイルでの接続を許可する一致するラインが必要です。例：

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

別の設定としては、`clientcertificates` の代わりに `SCRAM-SHA-256` パスワードを使用し、証明書が適切に署名されている限りサーバーの身元を確認する必要はありません。ここで、DSN パラメーターは次のとおりです。

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

…対応する `pg_hba.conf` 行は次のようになります。

```
hostssl all          postgres 10.1.2.3/24 scram-sha-256
hostssl replication postgres 10.1.2.3/24 scram-sha-256
```

このようなシナリオでは、postgres ユーザは正しいパスワードを含む `.pgpass` ファイルを必要とします。

4.2 監視ノード

クラスターのノード数が偶数のイベント、ネットワーク分割（またはネットワークパーティションとも呼ばれます）が発生した場合に追加のノードを作成して結び付けを解除すると便利な場合がありますヘルプ。

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

4.3 ロジカルスタンバイノード

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

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

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

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

高可用性にはマルチプルのオプションがあります。

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

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

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

PostgreSQL では、スロットを昇格する前にスタンバイでスロットの同期が完了していることを保証することが重要です。次のクエリ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。このクエリが `true` を返すときに、プロモーションを続行できます。

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

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

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

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

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

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

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

ロジカルスタンバイノードに DDL の変更を `make` ことはできますが、それらは複製されず、グローバル DDL ロックの取得も試みません。DDL と同様に機能する BDR 関数も複製されません。[DDL Replication]

を参照してください。ロジカルスタンバイに対して互換性のない DDL 変更を行った場合、データベースは分岐ノードと呼ばれます。現在、ダイバージェントノードの昇格はレプリケーションの失敗につながります。そのため、スタンバイとして使用する保証は、ロジカルスタンバイノードにダイバージェントな変更がフリーようにするか、ダイバージェントノードが昇格しないようにする必要があります。

4.4 フィジカルスタンバイノード

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

ただし、レプリケーションスロットの使用および BDR の他の機能要件により、これが適切に機能するための最小限の前提条件がいくつかあります。

BDR プライマリとスタンバイ間の接続はストリーミングを使用します物理的レプリケーションレプリケーションスロットを介した複製。 - スタンバイには次のものがあります。 - `recovery.conf` (PostgreSQL < 12 の場合、PostgreSQL 12+ の場合、これらの設定は `postgres.conf` にある必要があります) : - プライマリを指す `primary_conninfo` - `primary_slot_name` は、このスタンバイでのみ使用されるプライマリの物理的レプリケーションスロットの名前を指定します - `postgresql.conf` : - 少なくとも `shared_preload_libraries = 'pglogical, bdr'` - `hot_standby = on` - `hot_standby_feedback = on` - プライマリには次のものがあります。 - `postgresql.conf` : - `pglogical_standby_slot_names` は物理的を指定する必要があります スタンバイの `primary_slot_name` に使用されるレプリケーションスロット。

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

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

その結果、フェイルオーバーが正常に機能すると想定する前に、書き込みアクティビティの量が少ない新しく初期化された BDR クラスタの物理的スタンバイノードをチェックする必要があります。この予防措置を怠ると、BDR ノードとしてファンクションするために必要なレプリケーションスロットの不完全なサブセットがスタンバイに存在する可能性があり、フェイルオーバーが中止されます。

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

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

保証-enterprise 拡張機能をインストールすると、次の SQL 構文をスタンバイして、物理的スロットでの BDR 関連のレプリケーションスロットの作成を早めるのに役立つように、すべてのノードでグループスロットを（可能な限り）進めることができます。

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

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

1. Primary.2 と同じ IP アドレスまたはホスト名の制御を想定します。他のすべての BDR ノードで `bdr.alter_node_interface` ファンクションを実行して、アドレスの変更を BDR クラスターに通知します。

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

`pglogical.standby_slot_names` パラメータにも特別なノートを払ってください。これは `pglogical` 設定パラメータですが、プライマリ->フィジカルスタンバイのリレーション BDR クラスターに設定することが重要です。保証はこれを使用して、物理的スタンバイサーバがロジカルレプリカの前に常に WAL トラフィックを受信するようにしますが、BDR の使用例は異なります。

BDR は、アウトバウンドレプリケーションの最大のラグを示すクラスターノードの状態を常に反映するメンバスロットを維持しノード。グループスロットの状態。

スタンバイは他の BDR ノードと直接通信しないため、`standby_slot_names` パラメータは BDR に名前付けスロットをグループスロットの必要なコンストレインとして考慮するよう通知します。設定すると、スタンバイが遅れを示す場合、通常はグループスロットが進んでいたとしても、グループスロットが保持されます。

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

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

`insynchronous_standby_names` では、物理的スタンバイと他の BDR ノードを混在させることができます。CAMO および Eager All Node Replication は異なる同期メカニズムを使用し、同期レプリケーションでは機能しません。`synchronous_standby_names` に CAMO パートナー（CAMO が使用されている場合）または BDR ノードがまったく含まれていない（BDR All Node Replication が使用されている）`make`を確認してください。

4.5 サブグループ

グループには、0 個以上のサブグループを含めることもできます。各サブグループは、最上位の親グループ内の特定の目的に割り当てることができます。node_group_type は、サブグループが作成されときのタイプを指定します。

4.5.1 サブスクリバード専用グループ

BDR 3.7.5 以降では、subscriber-only ノードと呼ばれる新しい種類のノードがサポートされています。名前が示すように、このタイプのノードはクラスター内の他のノードからのレプリケーションの変更を受け取りません。これはロジカルスタンバイノードと多少似ていますが、ロジカルスタンバイとは対照的に、subscriber-only ノードはクラスターに完全に参加しています。クラスター内の他のすべてのノードからレプリケーションの変更を受け取ることができるため、クラスター内のいずれかのノードの使用不可または分離の影響を受けません。

また、pglogical ノードとは異なり、subscriber-only ノードは完全に結合された BDR ノードであるため、すべての複製された DDL を受け取り、それらに作用します。また、Raft を使用して、クラスター内のすべてのノードにそのステータスを一貫してレポートします。subscriber-only ノードには Raft の投票権がないため、Raft リーダーになることもリーダー選挙に参加することもできません。また、複製された DDL を受け取りますが、DDL または DML ロックの取得には参加しません。つまり、現在ダウンしている subscriber-only ノードは、DML ロックの取得を停止しません。

subscriber-only ノードは、BDR Treetopology のビルディングブロックを形成します。このトポロジでは、少数の完全にアクティブなノードがあり、すべての方向に変更を複製し、変更を受信するだけでクラスター内の他のノードに変更を送信しない subscriber-only ノードが多数あります。このトポロジは、ラージのノードに起因する接続の爆発を防ぎますが、データを消費するために使用できる非常にラージの leaf ノードを提供します。

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

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

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

注

subscriber-only ノードはクラスター内のノードに変更をレプリケートしないため、ノードがクラスターから分離されると、レプリケーションの変更を同期するためのソースとして機能できません。ただし、**subscriber-onlynode** が、クラスター内の他のノードが現在持っていない分割されたノードからレプリケーションの変更を既に受信して適用している場合、ノード間で不整合が発生します。

現時点では、`pglogical.standby_slot_names` および `pglogical.standby_slots_min_confirmed` を適切に設定して、**subscriber-onlynodes** の前に常に完全にアクティブな BDR ノードが存在するようにすることで、これを解決できます。これらの構成パラメーターを効果的に使用する方法については、`pglogical` の文書を参照してください。

これは将来のリリースで改善される予定です。レプリケーションで先に `b_tran_2` ノードを許可してから、同期のためにマストレプリケーションソースを使用するか、別の完全に結合されたノードが分離されたときにクラスターから矛盾する **subscriber-only** ノードをオプションで削除する方法を提供できます。RM20306 は開発タスクを追跡します。

4.6 デコードワーカー

BDR3.7 には、データが送信されているノードの数に関係なく、デコードを 1 回実行するワーカープロセスのデコードを有効にするオプションがあります。これにより、各 BDR ノードに新しいプロセス、`wal` デコーダーが導入されます。接続ごとに 1 つの WAL Sender プロセスがまだ存在しますが、これらのプロセスはデータの送信と受信のタスクを実行するだけです。これらの変更を総合すると、より大きな BDR グループの CPU オーバーヘッドが削減され、WAL Sender プロセスが通信により多くの時間を費やすようになるため、レプリケーションのスループットも向上します。

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

デコードワーカーが有効になっている場合、BDR は Logical Change Record (LCR、略称) ファイルを保存して、デコードとすべてのサブスクリाइブノードがデータを受信したときの変更のバッファリングを可能にします。LCR ファイルは、各ローカルノードのデータディレクトリ内の `pg_logical` ディレクトリの下に保存されます。LCR ファイルの数とサイズは、レプリケーションの遅延が増加するにつれて変化するため、モニタリングも必要になります。BDR ノードのいずれにも不要な LCR は定期的にクリーニングされます。2 つの連続したクリーンアップのインターバルは、`bdr.lcr_cleanup_interval` によって制御されます。デフォルトは 3 分です。`bdr.lcr_cleanup_interval` がゼロの場合、クリーンアップは無効になります。

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

デコードワーカーは BDR グループに対して有効になっていますが、次の GUC はノードごとの LCR の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 の稼動と使用のために、BDRBDR の各ノードで `true` に設定する必要があります。

- `pglogical.enable_wal_decoder=false` に変更すると、すべての WAL LCR を使用する送信者は、WAL を直接使用するために再起動されます。`true` の場合 BDR グループの構成とともに、デコードワーカープロセスは LCR の作成を開始し、WAL センダは LCR を使用します。

- `bdr.receive_lcr`-サブスクリプションしているノードの `true` が WAL を要求する場合利用可能な場合、LCR を使用するためにパブリッシャーノードで送信します。

4.6.1 メモ

現在、Decoding Worker は、実行中のノードに対応する変更をデコードします。単一のソースを介して、BDR グループのすべてのノードから変更がロジカルスタンバイに送信されます。したがって、ロジカルスタンバイを提供する WAL 送信者は、現在 LCR を使用できません。

購読者専用ノードはそれぞれのノードから直接変更を受信するため、購読者専用ノードにサービスを提供する WAL 送信者は LCR を使用できます。

LCR が生成されても、対応する WAL は、デコードワーカーが有効になっていない場合と同様に保持されます。将来、LCR が必要でない場合は、LCR に対応する WAL を削除できる可能性があります。

リファレンスまでに、LCR ファイル名の最初の 24 文字は WAL ファイル名のものと似ています。現在、名前の最初の 8 文字はすべて「0」です。将来的には、WAL セグメントファイル名の最初の 8 文字と同様に、`TimeLineId` を表すと予想されます。名前の 16 文字の次のシーケンスは、WAL ストリームに対して `LCRchanges` を追跡するのに使用される WAL セグメント番号と同様です。ノート、ロジカル変更は、下位のトランザクションのコミットオーダーに従って並べ替えられます。したがって、LCR セグメントでの配置は、WAL セグメントでの対応する WAL の配置とマッチしません。最後の 16 文字のセットは、LCR セグメント内のサブセグメント番号を表します。各 LCR ファイルはサブセグメントに対応しています。LCR ファイルはバイナリサイズで変数サイズです。LCR ファイルの最大サイズは、`pglogical.max_lcr_segment_file_size` で制御できます。デフォルトは 1GB です。

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

4.7 ノードの再起動とノードの復旧

BDR は、ノードのリスタートまたはノードの切断から回復するように設計されています。切断されたノードは、各同等なノードに再接続し、そのノードから不足しているデータを複製することにより、グループに自動的に再参加しノード。

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

各ノードの書き込みアクティビティの量が均一でない場合、データの多いノードからのキャッチアップ期間は他のノードよりもかなり長くなる可能性があります。最終的に、スロットの状態は `bdr.node_slots.state = streaming` に変わります。

数時間や数日など、長期間オフラインになっているノードは、さまざまな理由でリソースの問題を引き起こす可能性があります。ユーザーは、次の問題を理解せずに拡張の停止を計画しないでください。

ノードノードはオフラインノードを保持するため（同等なごとに 1 つを使用）、後で一時的にストレージできないノードへの変更をリプレイスペースノード。で* WAL *)、データベースサーバーが次のようなエラーでシャットダウンする可能性があります：

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

…またはその他のディスク外関連の症状をレポートします。

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

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

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

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

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

4.7.1 BDR によって作成された複製スロット

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

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

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

一方、Barman or pglogical などのソフトウェアに必要なレプリケーションスロットは、BDR に影響を与えることなく、ソフトウェアの適切なコマンドを使用して作成またはドロップできます。

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

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

- `bdr_mydb_mygroup` 名前付けの 1 つのグループスロット
- `bdr_mydb_mygroup_alpha` 名前付けの 2 つのノードスロットと `bdr_mydb_mygroup_beta`

グループ複製スロット

グループスロットは、主に BDR グループ内のノード（すべての論理スタンバイを含む）がこのノードからのアウトバウンドレプリケーションに対してキャッチした最も古い安全な位置を追跡するために BDR によって使用される内部スロットです。

グループスロット名前は、`bdr.local_group_slot_name()` ファンクションによって指定されます。

グループスロットは次のことができます。

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

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

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

4.7.2 ロング識別子のハッシュ

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

この例として、`group20xxxxxxxxxxxxxx`（20 バイト長）という名前の BDR グループを使用して、`db20xxxxxxxxxxxxxx`（20 バイト長）という名前のデータベースを複製するクラスターを考えます。ノード `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
```

…3597186、be9cbd0、7f304a2 はそれぞれ `db20xxxxxxxxxxxxxx`、`group20xxxxxxxxxxxxxx`、`a30xxxxxxxxxxxxxxxxxxxxxxxxxx` のハッシュであるため。

4.8 BDR グループからノードを削除する

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

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

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

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

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

4.8.1 BDR のアンインストール

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

```
DROP EXTENSION bdr;
```

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

- `timeshard` や `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削除する必要がありインスタンス、依存オブジェクトを削除する、列タイプを非 BDR に変更する、シーケンスタイプを `toLocal` に戻すなどです。

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

!!! Warning * ローカル BDR ノード、またはローカルデータベースの BDR 拡張機能をドロップすると、既存のセッションが BDR 固有のワークフローを実行しようとして失敗する可能性があります。この問題は、セッションを切断してからクライアントを再接続するか、インスタンスを再起動することで解決できます。さらに、(1) リレーションクラスターからノードを BDR し、(2) BDR 拡張機能をドロップ (3) それを再作成し、最後に (4) `pglogical.replication_set_add_all_tables()` を実行しています。インスタンスを再起動すると、問題が解決します。

BDR が必要とする pglogical 拡張にも同様の考慮事項が適用されます。

pglogical が BDR でのみ使用される場合、単一のステートメントで両方の拡張子を削除することができます。

```
DROP EXTENSION pglogical, bdr;
```

逆に、ノードが BDR から独立して pglogical も使用している場合、たとえば、一部のテーブルをリモートデータベースに一方向でレプリケーション場合、BDR 拡張のみを削除する必要があります。

!!! Warning * pglogical を単独で使用するデータベースから BDR をドロップすると、既存の pglogical サブスクリプションが「BDR グローバルロックマネージャがまだ初期化されていません」エラーでさらに動作するのをブロックできます。インスタンスを再起動すると、問題が解決します。

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

4.9 BDR トポロジーのリスト

4.9.1 BDR グループのリスト

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

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

各ノードグループの構成は、より複雑なクエリーを使用して表示できます。

```
SELECT g.node_group_name
, ns.pub_repsets
, ns.sub_repsets
, g.node_group_default_repset      AS default_repset
, node_group_check_constraints     AS check_constraints
FROM bdr.local_node_summary ns
JOIN bdr.node_group g USING (node_group_name);
```

4.9.2 BDR グループ内のノードのリスト

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

```
SELECT node_name      AS name
, node_seq_id         AS ord
, peer_state_name     AS current_state
, peer_target_state_name AS target_state
```

(次のページに続く)

(前のページからの続き)

```
, interface_connstr      AS dsn
FROM bdr.node_summary
WHERE node_group_name = 'mygroup';
```

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

4.9.3 ノード状態のリスト

- **NONE**：ワーカーの起動時にノードの状態が設定されず、すぐに設定されることが期待されます現在の既知の状態に。
- **CREATED**：bdr.create_node() は実行されましたが、ノードは BDR クラスターのメンバ。
- **JOIN_START**：結合は、ローカルノードを既存の BDR クラスター。
- **JOINING**：ノードの結合が開始され、現在は初期同期フェーズにあります。ノードでスキーマとデータを作成します。
- **CATCHUP**：初期同期フェーズが完了しました。結合は最後のステップになりましたアップストリームで実行されたトランザクションの取得と適用の結合が開始されノードからの同等な。
- **STANDBY**：ノードの結合は終了しましたが、変更のブロードキャストはまだ開始されていません。すべての参加はこの状態でしばらく時間がかかりますが、ロジカルスタンバイとして定義されている場合、ノードはこの状態を継続します。
- **PROMOTE**：ノードはロジカルスタンバイであり、bdr.promote_node を呼び出してノードの状態を **ACTIVE** に移動します。これらの 2 つの PROMOTEstates は一貫している必要があります事実、STANDBY よりも高い状態にできるノードは 1 つだけですが、ACTIVE よりも低い。
- **PROMOTING**：ロジカルスタンバイからフル BDR ノードへの昇格が進行中です。
- **ACTIVE**：ノードは完全な BDR ノードであり、現在は **ACTIVE** です。これは最も一般的なノードのステータス。
- **PART_START**：ノードは **ACTIVE** または **STANDBY** で、bdr.part_node を呼び出しました BDR クラスターからノードを削除します。
- **PARTING**：ノードは他のノードから切断され、それ以上のパートを果たしませんコンセンサスまたはレプリケーション。
- **PART_CATCHUP**：分割されていないノードは、最近別れたノード。
- **PARTED**：ノード分割オペレーションがすべてのノードで完了しました。

一度に 1 つのノードのみが **PROMOTE** または **PROMOTING** の状態になります。

4.10 シャードグループの管理

BDR クラスターには、オートスケール機能用のシャードグループの配列が含まれる場合があります。これらは、シャードグループと呼ばれるサブサブノードグループの配列で構成されるサブノードグループとして表示されます。

シャードグループで実行できる操作は次のとおりです。

- シャード配列の作成
- ドロップシャードアレイ
- 修復-故障したノードを置き換えるために新しいノードを追加します
- 展開-新しいシャードグループの追加
- 再バランス-シャードグループ間でのデータの再分散

4.10.1 作成/ドロップ

4.10.2 展開

例 4 つのシャードグループから 8 つのシャードグループに展開する

このオペレーションは、ユーザの操作を妨げることなく実行できます。

4.10.3 リバランス

たとえば、4 ノード配列のデータを、8 ノード配列に理想的に配置する方法に移動します。

データの一部は 1 つのシャードグループから別のシャードグループに移動されるため、移動するデータの量によっては、このアクションに拡張かかる場合があります。データは一度に 1 パーティションずつ移動されるため、無駄な労力をかけずに再起動できます。

再バランスはオプション。

このオペレーションは、これに書き込みトランザクションが含まれる場合でも、ユーザ操作を妨げることなく実行できます。

4.11 ノード管理インターフェース

ノードは、SQL インターフェースを使用して動的に追加および削除できます。

4.11.1 bdr.create_node

このファンクションはノードを作成します。

あらすじ

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

パラメーター

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

注

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

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

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

4.11.2 bdr.drop_node

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

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

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

あらすじ

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

パラメーター

- **node_name**-既存のノードの名前。
- **cascade**-依存オブジェクトにカスケードするかどうか、これも削除します関連する pglogical ノード。このオプションは注意して使用する必要があります！
- **force**-すべての健全性チェックを回避し、危険の可能性があるにもかかわらず、特定の BDR ノードのすべてのメタデータ矛盾を引き起こします。強制ノードドロップは、関連する緊急事態の場合のみのテクニカルサポート 別れ。

注

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

このファンクションは、特定のノードのメタデータをローカルデータベースから削除します。ノードはローカルノードにすることができます。この場合、リモートノードに関する情報を含むすべてのノードメタデータが削除されます。または、リモートノードにすることもできます。その場合、その特定のノードのメタデータのみが削除されます。

!!! Note * BDR3 は、一度に最大 1024 のノードレコード (ACTIVE と PARTED の両方) を持つことができます。これは、各ノードにタイムハードシーケンスで使用するための一意のシーケンス番号が割り当てられているためです。PARTED ノードは現時点では自動的にクリーンアップされません。これが問題になった場合、このファンクションを使用してそれらのレコードを削除できます。

4.11.3 bdr.create_node_group

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

あらすじ

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

パラメーター

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

注

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

ファンクションはトランザクションではありません。グループの作成はバックグラウンドプロセスであるため、ファンクションが終了すると、変更をロールバックできません。また、変更は現在のトランザクションにすぐに可視されない場合があります。

グループの作成はロックを保持しません。

4.11.4 bdr.alter_node_group_config

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

あらすじ

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

(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)

```

パラメーター

- **node_group_name**-既存の BDR グループの名前。ローカルノードはパートでなければなりませんグループの。
- **insert_to_update**-後方互換性のために予約されています。
- **update_to_insert**-後方互換性のために予約されています。BDR のバージョン。代わりに **bdr.alter_node_set_conflict_resolver** を使用してください。 * *
- **ignore_redundant_updates**-後方互換性のために予約されています。
- **check_full_tuple**-後方互換性のために予約されています。
- **apply_delay**-後方互換性のために予約済み
- **check_constraints**-適用プロセスがコンストレインをチェックするかどうか複製されたデータを書き込むとき。このオプションは**非推奨**であり、**今後無効または削除される可能性があります BDR のバージョン**
- **num_writers**-サブスクリプションバックギング用の並列ライターの数このノードグループ、-1 はデフォルトを意味します (pglogical で指定) GUC pglogical.writers_per_subscription) が使用されます。有効な値
- 1 または正の整数です。
- **enable_wal_decoder**-デコードワーカプロセスを有効/無効にします。

すべてのオプションパラメータは、単に pglogical ライタを制御するために使用されることに注意してください。

注

このファンクションは、グループコンセンサスメカニズムにデフォルトを変更する要求を渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

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

このファンクションはロックを保持しません。

!!! Warning * このファンクションを使用して **apply_delay** 値を変更する場合、変更はすでにグループのメンバーであるノードには適用されません。通常、この値はテスト以外では使用されないため、この制約は稼働環境の使用にはほとんど影響しないことに注意してください。

4.11.5 bdr.join_node_group

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

あらすじ

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

パラメーター

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

wait_for_completion が false として指定されている場合、これは、参加プロシージャが開始されるとすぐに返される非同期呼び出しです。結合の進行状況は、ログおよび `thebdr.state_journal_details` 情報ビューで確認できます。または、`bdr.join_node_group()` が戻ったら、`bdr.wait_for_join_completion()` ファンクションを呼び出すことで確認できます。

注

このファンクションは、**join_target_dsn** 接続文字列が指すノードを介してグループコンセンサスメカニズムに要求を渡します。加えられた変更は、コンセンサスメカニズムを介してグローバルに複製されます。

このファンクションはトランザクションではありません。参加プロセスはバックグラウンドで行われるため、ロールバックできません。**wait_for_completion** が true に設定された場合、または後で `bdr.wait_for_join_completion` を呼び出して、変更がローカルトランザクションにのみ表示されます。

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

ノード結合は、BDR グループのロックを保持しません。

4.11.6 bdr.promote_node

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

あらすじ

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

注

このファンクションは、グループコンセンサスメカニズムにデフォルトを変更する要求を渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

このファンクションはトランザクションではありません。プロモーションプロセスはバックグラウンドで行われるため、ロールバックできません。wait_for_completion が true に設定された場合、または後で bdr.wait_for_join_completion を呼び出して、変更がローカルトランザクションにのみ表示されます。

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

4.11.7 bdr.wait_for_join_completion

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

あらすじ

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

パラメーター

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

注

このファンクションは、ローカルノードのチェック状態が `bdr.create_node_group`、`bdr.join_node_group` または `bdr.promote_node` によって設定された `targetstate` に到達するまで待機します。

4.11.8 bdr.part_node

BDR グループからノードを削除（「パーツ」）しますが、ノードからデータを削除しません。

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

!!! Note * あなたはローカルノードを「分割」しているため、`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。

!!! Warning * のアクションは永続的です。ノードのレプリケーションを一時的に停止する場合は、`bdr.alter_subscription_disable()` を参照してください。

あらすじ

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

パラメーター

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

!!! Warning * `force = true` を使用すると、BDR グループが一貫性のない状態になる可能性があり、他の方法でノードを削除できないビザンチン障害から回復する場合にのみ使用する必要があります。 * *

注

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

`force` が `true` に設定されている場合、コンセンサスエラーが発生すると、このファンクションはローカルノードでのみ指定されたノードの状態を設定しノード。このような場合、ファンクションはトランザクション関数であり（ファンクション自分自身がノードの状態を変更するため）、ロールバックできます。ファンクションが、`force` が `true` に設定されて既に分割中のノードで呼び出された場合、指定されたノードをローカルに分割してマークし、終了します。これは、クラスターでコンセンサスに到達できない場合（つまり、ノードの大部分がダウンしている場合）、または何らかの理由で分割プロセスが停止した場合にのみ役立ちます。ただし、他のノードが特定のノードから欠落しているデータを再同期する必要があるため、書き込みを受信していた分離ノードが書き込みを行うときに、スタックすることなく分離プロセスに時間がかかる可能性があることに注意することが重要です。強制分割はこの再同期を完全にスキップするため、他のノードが一貫性のない状態のままになる可能性があります。

分離プロセスはロックを保持しません。

4.11.9 bdr.alter_node_interface

このファンクションは、指定されたノードの接続文字列（DSN）を変更します。

あらすじ

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

パラメーター

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

注

このファンクションはローカルノードでのみ実行され、変更はローカルノードでのみ行われノード。これは、通常、変更中のノードを含む BDR グループ内のすべてのノードで実行する必要があることを意味します。

このファンクションはトランザクショナルです。ロールバックでき、変更は現在のトランザクションに表示されます。

このファンクションは、ローカルノードをロックします。

4.11.10 bdr.alter_subscription_enable

このファンクションは、指定されたサブスクリプションまたはローカル BDR ノードのすべてのサブスクリプションを有効にします。サブスクリプションの再開とも呼ばれます。サブスクリプションが既に有効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

あらすじ

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

パラメーター

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

注

このファンクションは複製されず、ローカルノードのサブスクリプション（特定のノードまたはすべてのノード）にのみ影響します。

このファンクションはトランザクションです。ロールバックでき、現在のトランザクションでカタログの変更を確認できます。サブスクリプションワーカーは、トランザクションがコミットされた後、バックグラウンドプロセス開始されます。

4.11.11 bdr.alter_subscription_disable

このファンクションは、指定されたサブスクリプションまたはローカル BDR ノードのすべてのサブスクリプションを無効にします。必要に応じて、無効なサブスクリプションに関連付けられているすべてのワーカーをすぐに停止することもできます。pausesubscription と呼ばれます。サブスクリプションがすでに無効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

あらすじ

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

パラメーター

- **subscription_name**-無効にするサブスクリプションの名前。NULL の場合（デフォルト）、ローカルノード上のすべてのサブスクリプションは無効になります。
- **immediate**-即時を使用してアクションをすぐに強制し、停止します無効サブスクリプションに関連付けられているすべてのワーカー。このオプションで **true**、このファンクションはトランザクションブロック内で実行できません。

注

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

このファンクションはトランザクションです。ロールバックでき、現在のトランザクションでカタログの変更を確認できます。ただし、subscriptionworker が停止するタイミングは **immediate** の値に依存します。**true** に設定されている場合、ワーカーはすぐに停止します。**false** に設定すると、COMMIT の時点で停止します。

4.12 ノード管理コマンド

BDR は、既存ノードの物理コピー（pg_basebackup）を介してノードを BDR グループに追加し、既存ノードの物理スタンバイを BDR グループの新しいノードに変換するためのコマンドラインユーティリティも提供します。

4.12.1 bdr_init_physical

これは、PostgreSQL の bin ディレクトリに追加される通常のコマンドです。

ユーザはデータディレクトリを指定する必要があります。このデータディレクトリが空の場合、pg_basebackup -X stream コマンドを使用して、高速ブロックレベルコピーオペレーションを使用してディレクトリを埋めます。

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

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

このコマンドは、すべての pglogical のみのサブスクリプションと構成をデータベースから削除し、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もデータベースから削除します（または、レプリケーション起点オプションが使用されている場合は単に無効にします）。

これは、pglogical_create_subscriber ユーティリティの BDR3 バージョンです。

あらすじ

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

オプション

一般オプション

- `-D, --pgdata=DIRECTORY`-新しいノードに使用されるデータディレクトリ。それ空のディレクトリまたは存在しないディレクトリのいずれか、または `pg_basebackup -X stream` コマンド（必須）。
- `-l, --log-file=FILE`-ロギングに FILE を使用します。デフォルトは `bdr_init_physical_postgres`。ログ。
- `-n, --node-name=NAME`-新しく作成されたノードの名前（必須）。
- `--replication-sets=SETS`-レプリケーションのコンマ区切りリストの名前使用する名前を設定します。指定しない場合、すべてのレプリケーションセットが使用されます。
- `--standby`-完全ではなく、ロジカルスタンバイ（ノードのみを受信）を作成します送受信ノード。
- `--node-group-name`-結合グループ。デフォルトはソースノードと同じグループです。
- `-s, --stop`-初期化が完了したらサーバーを停止します。
- `-v`-ロギングの詳細度を上げます。
- `-L`-と組み合わせて使用すると、選択的な `pg_basebackup` を実行します空/存在しないデータディレクトリ（`-D` オプション）。（EDB Postgres Extended）
- `-S`-ロジカルレプリケーションサブスクリプションを削除する代わりに、単に無効にしますそれら。

接続オプション

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

設定ファイルの上書き

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

注

コマンドで指定されたレプリケーションセット名は、ノードが BDR グループに参加する前にデータディレクトリに存在するデータに影響しません。これは、`bdr_init_physical` が独自の `basebackup` を作成するか、既存の `basebackupis` が新しい BDR ノードに昇格されるかに関係なく当てはまります。したがって、`--replication-sets` オプションは、ノードが BDR ノードグループに参加した後だけにのみ、公開およびサブスクライブされるデータに影響します。この動作は、レプリケーションセットが非論理結合で使用される方法、つまり `bdr.join_node_group()` を使用する場合は異なります。

不要なテーブルは、結合の完了後に演算子によって切り捨てられる場合があります。`bdr.tables` カタログを参照して、レプリケーションセットのメンバーシップを確認し、サブスクライブ先のレプリケーションセットのメンバーではないテーブルを識別します。次の理由により、テーブルを削除するのではなく、切り捨てることを強くお勧めします。

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

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

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

第5章 DDL Replication

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

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

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

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

デフォルトのレプリケーションセットでは、DDL はデフォルトですべてのノードに複製されます DDLDDL レプリケーションを追加する必要があります。[DDL Replication Filtering] を参照してください。

BDR は、DDL レプリケーションに関しては、スタンドアロン PostgreSQL とは大きく異なり、同じものとして扱うことは、BDR の最も一般的な運用上の問題です。

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

5.1 DDL レプリケーションオプション

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

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

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

は rep セット（たとえば、site1 のすべてのノード）のサブセットに存在するインデックスまたはパーティションが必要な場合に役立ちます。

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

自動 DDL レプリケーションをスキップして、bdr.ddl_replication 構成パラメーターを使用して各ノードで手動で実行することは可能ですが、推奨されません。

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

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

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

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

5.2 BDR システムでの DDL の実行

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

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

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

BDR は、これらの問題に対する 3 つのレベルの保護を提供します。

ddl_locking = 'dml' は操作に最適なオプションで、一度に 1 つのノードからのみ DDL を実行するときに使用できます。これはデフォルトではありませんが、DDL の実行元を制御できる場合は、この設定を使用して、ノード間の競合が発生しないようにすることをお勧めします。ノード内の競合は、PostgreSQL によってすでに処理されています。

ddl_locking = on は最も厳密なオプションであり、DDL が任意のノードから同時に実行される可能性があります。正確性を保証したい場合に最適です。

ddl_locking = off は最も厳密ではないオプションであり、一般的な使用では危険です。このオプションはロックを完全にスキップするため、パフォーマンスのオーバーヘッドを回避し、新しい空のデータベーススキーマを作成する際に便利なオプションになります。

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

`bdr.replicate_ddl_command` を使用する場合、その関数に渡される DDL コマンドに対してのみ指定された `bdr.ddl_locking` 設定を使用して、3 番目の引数を介してこのパラメーターを直接設定できます。

5.3 DDL ロックの詳細

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

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

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

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

DDL を実行しているノードがクラスター内で実行されているすべての PriorDDL のオーダーを保証に、以前の DDL を実行していたノードに追いつくまで待機しノード。現在の DDL を実行しているノードが、前の DDL を実行したノードとの関係で遅れている場合、ロックを取得するのに非常に長い時間がかかります。したがって、単一のノードまたは他のノードで発生するレプリケーションの変更にはほぼ追いついたノードから DDL を実行することをお勧めします。

2 番目の種類は、リレーション DML ロックと呼ばれます。この種類のロックは、`ddl_locking = on` または `ddl_locking = dml` のいずれかであり、一意性制約、検査制約、NOT NULL 制約などの制約を追加または変更する場合など、DDL ステートメントによって飛行中の DML ステートメントが失敗する場合に使用されます。一度に 1 つのリレーション。リレーション保証は、キューに変更があり、エラーでレプリケーションが停止する可能性がある間、DDL が実行されないようにします。

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

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

保留中の DML 操作が排出されるのを待つには時間がかかるか、レプリケーションが現在遅れている場合は長くなります。つまり、行の表現形式とコンストレインに影響するスキーマ変更は、データの変更とは

異なり、構成されたすべてのノードが到達可能であり、合理的に維持されている場合にのみ実行できます。ノードがダウンしている間にそのような DDL コマンドを絶対に実行する必要がある場合、ダウンノードを最初に構成から削除する必要があります。

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

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

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

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

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

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

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

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

一部の BDR 管理機能は DDL のように機能することに注意してください。つまり、DDLreplication がアクティブな場合、グローバルロックの取得を試み、そのアクションが複製されます。複製された関数の完全なリストは、*DDL のように動作する BDRFunctions* にリストされています。

一時テーブルで実行される DDL は、グローバルロックを必要としません。

現在のトランザクションで作成されたオブジェクトの ALTER または DROP は、グローバル DML ロックを必要としません。

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

5.4 DDL の影響を最小限に抑える

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

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

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

長時間実行される DDL を実行する別の方法は、DDL レプリケーションを無効にしてから、各ノードで個別に DDL ステートメントを実行です。以下の例に示すように、単一の SQL ステートメントを使用して実行できますが、グローバルロックルールが適用されることに注意してください。

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

上記の bdr.run_on_all_nodes() テクニックを CREATEINDEX CONCURRENTLY で使用することをお勧めし稼働。CREATEINDEX CONCURRENTLY はマルチトランザクションコマンドであるため、DDL レプリケーションをセッション全体で無効にする必要があることに注意してください。REINDEX は BDR3.6 までのバージョンで複製されますが、BDR3.7 以降では複製されません。REINDEX の使用は、AccessExclusiveLocks が保持されているため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。

無効なインデックスに対する REINDEX または REINDEX CONCURRENTLY は、BDR ノードでの実行に失敗します。無効なインデックスは、ドロップして再度作成する必要があります。無効なインデックスは、DDL BDR .. を使用してドロップする必要があります。

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

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

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

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

(オプションの) グローバルロックタイムアウト設定で、グローバルロックをロックする `wait-for` がキャンセル長さトランザクションまでの時間を制限します。グローバルロックを保持し、`bdr.global_lock_idle_timeout` は、グローバルロックを保持するトランザクションの最大許容アイドル時間(ステートメント間の時間)を設定します。これらのタイムアウトはすべて、値をゼロに設定することで無効にできます。

発信元ノードで DDL オペレーションがコミットされると、キャンセルまたは中止できません。BDR グループは、グローバルロックを確認した他のノードに正常に適用され、リプレイを確認するまで待機する必要があります。これが、DDL トランザクションを短く高速に保つことが重要な理由です。

5.5 ダウンノードでの DDL の処理

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

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

グローバルロックを確認した後、それを取得するコマンドが実行される前に、他のノードの 1 つがダウンした場合、ロックを要求するコマンドの実行は、ノードがアップしたかのように続きます。

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

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

5.6 ステートメント固有の DDL レプリケーションの懸念

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

BDR は、データベース上でアクティブなときに一部の DDL ステートメントが実行されないようにします。これにより、正しく複製できないステートメントやレプリケーションがまだサポートされていないステートメントを禁止することにより、システムの一貫性が保護されます。一部の制限付きでサポートされているステートメントは、[制限付きの DDL ステートメント] で説明されています。一方、BDR で完全に禁止されているコマンドは、禁止されている DDL ステートメントでカバーされています。

ステートメントが BDR で許可されていない場合、同じことを行う別の方法を見つけることがしばしば可能です。例、揮発性のデフォルト値を持つ列を追加する `ALTER TABLE` を実行することはできませんが、通常は、動作するシリーズの独立した `ALTER TABLE` および `UPDATE` ステートメントとして言い換えることができます。

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

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

5.7 BDR DDL コマンド処理マトリックス

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

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

Command	Allowed	Replicated	Lock
ALTER AGGREGATE	Y	Y	DDL
ALTER CAST	Y	Y	DDL
ALTER COLLATION	Y	Y	DDL
ALTER CONVERSION	Y	Y	DDL
ALTER DATABASE	Y	N	N
ALTER DATABASE LINK	Y	Y	DDL
ALTER DEFAULT PRIVILEGES	Y	Y	DDL
ALTER DIRECTORY	Y	Y	DDL
ALTER DOMAIN	Y	Y	DDL
ALTER EVENT TRIGGER	Y	Y	DDL
ALTER EXTENSION	Y	Y	DDL
ALTER FOREIGN DATA WRAPPER	Y	Y	DDL
ALTER FOREIGN TABLE	Y	Y	DDL
ALTER FUNCTION	Y	Y	DDL
ALTER INDEX	Y	Y	DDL
ALTER LANGUAGE	Y	Y	DDL
ALTER LARGE OBJECT	N	N	N
ALTER MATERIALIZED VIEW	Y	N	N
ALTER OPERATOR	Y	Y	DDL
ALTER OPERATOR CLASS	Y	Y	DDL
ALTER OPERATOR FAMILY	Y	Y	DDL
ALTER PACKAGE	Y	Y	DDL
ALTER POLICY	Y	Y	DDL
ALTER PROCEDURE	Y	Y	DDL
ALTER PROFILE	Y	Y	DDL
ALTER PUBLICATION	Y	Y	DDL
ALTER QUEUE	Y	Y	DDL
ALTER QUEUE TABLE	Y	Y	DDL
ALTER REDACTION POLICY	Y	Y	DDL

[次のページに続く](#)

表 1 – 前のページからの続き

Command	Allowed	Replicated	Lock
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	` Details <#bdr_ddl_allowed_AlterSeqStmt>`__	Y	DML
ALTER SERVER	Y	Y	DDL
ALTER SESSION	Y	N	N
ALTER STATISTICS	Y	Y	DDL
ALTER SUBSCRIPTION	Y	Y	DDL
ALTER SYNONYM	Y	Y	DDL
ALTER SYSTEM	Y	N	N
ALTER TABLE	<i>Details</i>	Y	<i>Details</i>
ALTER TABLESPACE	Y	N	N
ALTER TEXT SEARCH CONFIGURATION	Y	Y	DDL
ALTER TEXT SEARCH DICTIONARY	Y	Y	DDL
ALTER TEXT SEARCH PARSER	Y	Y	DDL
ALTER TEXT SEARCH TEMPLATE	Y	Y	DDL
ALTER TRIGGER	Y	Y	DDL
ALTER TYPE	Y	Y	DDL
ALTER USER MAPPING	Y	Y	DDL
ALTER VIEW	Y	Y	DDL
ANALYZE	Y	N	N
BEGIN	Y	N	N
CHECKPOINT	Y	N	N
CLOSE	Y	N	N
CLOSE CURSOR	Y	N	N
CLOSE CURSOR ALL	Y	N	N
CLUSTER	Y	N	N
COMMENT	Y	<i>Details</i>	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

次のページに続く

表 1 – 前のページからの続き

Command	Allowed	Replicated	Lock
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
CREATE ROLE	Y	Y	DDL
CREATE ROUTINE	Y	Y	DDL
CREATE RULE	Y	Y	DDL
CREATE SCHEMA	Y	Y	DDL
CREATE SEQUENCE	<i>Details</i>	Y	DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	<i>Details</i>	Y	DDL
CREATE TABLE AS	<i>Details</i>	Y	DDL
CREATE TABLESPACE	Y	N	N
CREATE TEXT SEARCH CONFIGURATION	Y	Y	DDL
CREATE TEXT SEARCH DICTIONARY	Y	Y	DDL
CREATE TEXT SEARCH PARSER	Y	Y	DDL
CREATE TEXT SEARCH TEMPLATE	Y	Y	DDL
CREATE TRANSFORM	Y	Y	DDL
CREATE TRIGGER	Y	Y	DDL
CREATE TYPE	Y	Y	DDL

次のページに続く

表 1 – 前のページからの続き

Command	Allowed	Replicated	Lock
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
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

次のページに続く

表 1 – 前のページからの続き

Command	Allowed	Replicated	Lock
DROP REDACTION POLICY	Y	Y	DDL
DROP RESOURCE GROUP	Y	N	N
DROP ROLE	Y	Y	DDL
DROP ROUTINE	Y	Y	DDL
DROP RULE	Y	Y	DDL
DROP SCHEMA	Y	Y	DDL
DROP SEQUENCE	Y	Y	DDL
DROP SERVER	Y	Y	DDL
DROP STATISTICS	Y	Y	DDL
DROP SUBSCRIPTION	Y	Y	DDL
DROP SYNONYM	Y	Y	DDL
DROP TABLE	Y	Y	DML
DROP TABLESPACE	Y	N	N
DROP TEXT SEARCH CONFIGURATION	Y	Y	DDL
DROP TEXT SEARCH DICTIONARY	Y	Y	DDL
DROP TEXT SEARCH PARSER	Y	Y	DDL
DROP TEXT SEARCH TEMPLATE	Y	Y	DDL
DROP TRANSFORM	Y	Y	DDL
DROP TRIGGER	Y	Y	DDL
DROP TYPE	Y	Y	DDL
DROP TYPE BODY	Y	Y	DDL
DROP USER MAPPING	Y	Y	DDL
DROP VIEW	Y	Y	DDL
EXECUTE	Y	N	N
EXPLAIN	Y	<i>Details</i>	<i>Details</i>
FETCH	Y	N	N
GRANT	Y	<i>Details</i>	DDL
GRANT ROLE	Y	Y	DDL
IMPORT FOREIGN SCHEMA	Y	Y	DDL
LISTEN	Y	N	N
LOAD	Y	N	N
LOAD ROW DATA	Y	Y	DDL
LOCK TABLE	Y	N	N
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

次のページに続く

表 1 – 前のページからの続き

Command	Allowed	Replicated	Lock
REVOKE	Y	<i>Details</i>	DDL
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	<i>De tails</i>	DDL
SELECT INTO	<i>Detai ls</i>	Y	DDL
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	<i>De tails</i>	<i>Details</i>
UNLISTEN	Y	N	N
VACUUM	Y	N	N

</ div>

5.7.1 ALTER SEQUENCE

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

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

5.7.2 ALTER TABLE

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

</ div>

ALTER TABLE 禁止コマンド

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

- ADD COLUMN ... DEFAULT (non-immutable expression)-これは許可されていません現在、異なるノードで異なるデータが生成されます。見る推奨される回避策については、*Adding a Column*。
- ADD CONSTRAINT ... EXCLUDE-現時点では、除外コンストレインはサポートされていません。非同期システムでは、除外コンストレインはあまり意味がありません。make。リプレイできない変更につながります。

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

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

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

BDR 3.7.4 以降では、`CHECK` や `FOREIGN KEY` コンストレインなどの特定の種類のコンストレインを、`DML` ロックを取得せずに追加できます。ただし、これには最初に `NOT VALID` 制約を作成し、次に `ALTER TABLE ... VALIDATE CONSTRAINT` コマンドを介して別のトランザクションで制約を検証する 2 段階のプロセスが必要です。詳細については、*Adding a CONSTRAINT* を参照してください。

</div>

ALTER TABLE ロック

`ALTER TABLE` の次のバリエーションは、`DDL` ロックのみを取得し、`aDML` ロックは「**」ではありません：

- `ALTER TABLE ... ADD COLUMN ... (immutable) DEFAULT`
- `ALTER TABLE ... ALTER COLUMN ... SET DEFAULT expression`
- `ALTER TABLE ... ALTER COLUMN ... DROP DEFAULT`
- 書き換えが不要な場合は `ALTER TABLE ... ALTER COLUMN ... TYPE` (現在、EDB Postgres Extended および EDB Postgres Advanced でのみ利用可能)
- `ALTER TABLE ... ALTER COLUMN ... SET STATISTICS`

- ALTER TABLE ... VALIDATE CONSTRAINT
- ALTER TABLE ... ATTACH PARTITION
- ALTER TABLE ... DETACH PARTITION
- ALTER TABLE ... ENABLE TRIGGER (ENABLE REPLICA TRIGGER は引き続き DML ロックを取得します)
- ALTER TABLE ... CLUSTER ON
- ALTER TABLE ... SET WITHOUT CLUSTER
- ALTER TABLE ... SET (storage_parameter = value [, ...])
- ALTER TABLE ... RESET (storage_parameter = [, ...])
- ALTER TABLE ... OWNER TO

ALTER TABLE の他のすべてのバリエーションは、変更対象のテーブルで DML ロックを取得します。ALTER TABLE の一部のバリエーションには、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、したがってテーブルの書き換えを行わないため、カタログのみの変更として実行されるため、機能します。

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

ただし、VARCHAR (128) から VARCHAR (20) への変更はバイナリ強制互換ではないため、上記のコマンドを逆にするためにこの変更を行うことはできません。

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

推奨される回避策については、後で参照してください。

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

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

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

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

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

ただし、col2 のサイズを縮小する場合は、基になるテーブルデータの書き換えにつながります。通常、テーブルの書き換えは制限されています。

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

非テキスト型の例を与えるには、typeINTEGER を使用した上記の col3 を検討してください。SMALLINT または BIGINT に変換しようとする ATCT オペレーションは、上記と同様の方法で失敗します。

```
ALTER TABLE sample ALTER COLUMN col3 TYPE bigint;
ERROR: ALTER TABLE ... ALTER COLUMN TYPE that rewrites table data may not affect
↳ replicated tables on a BDR node
```

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

このような場合、制御された DBA 環境では、すべてのノードで非複製環境でこの列のタイプにローリングアップグレードを1つずつ採用することにより、列のタイプを自動的にキャスト可能なものに変更することができます。DDL がレプリケートされず、列タイプの変更が上記のように自動的にキャスト可能なものである場合、同じテーブルの他のノードでの同時アクティビティとともに、変更を実行するノードでローカルに書き換えを許可することができます。その後、この複製されない ATCT オペレーションをすべてのノードで1つずつ繰り返して、BDR クラスター全体にわたって列タイプの望ましい変更をもたらすことができます。これは書き換えを伴うため、アクティビティはまだ短い期間 DML ロックを取得するため、クラスター全体が使用可能である必要があります。上記の仕様が整ったら、次のように非複製者アクティビティのローリングアップグレードを実行できます。

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

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

```
-- foreach node in BDR cluster do:
SET bdr.ddl_replication TO FALSE;
ATCT operations to-from {INTEGER, SMALLINT, BIGINT}
ATCT operations to-from {CHAR(n), VARCHAR(n), VARCHAR, TEXT}
ATCT operations from numeric types to text types
RESET bdr.ddl_replication;
-- done
```

上記は、レプリケートされていない環境で許容される可能性のある ATCT 操作の完全なリストではありません

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

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

レプリケートされない環境での ATCT 操作はさまざまな型変換をサポートしますが、基になるテーブルデータに新しいデータタイプに割り当てることができない値が含まれる場合、書き換えが失敗する可能性があることにノートすることが重要です。例、列の現在の型は `VARCHAR(256)` である可能性があり、複製されない ATCToperation を試して `VARCHAR(128)` に変換しました。テーブルに 128 バイトよりも広い既存のデータがある場合、書き換え操作はローカルで失敗します。

```
INSERT INTO sample VALUES (1, repeat('a', 200), 10);
SET bdr.ddl_replication TO FALSE;
ALTER TABLE sample ALTER COLUMN col2 TYPE VARCHAR(128);
INFO:  in rewrite
ERROR:  value too long for type character varying(128)
```

基になるテーブルデータが新しい型の特性を満たす場合、書き換えは成功します。ただし、他のノード（まだ複製されていないローリングデータタイプアップグレードを実行していない）がこのローカル ATCT オペレーションに 128 バイトを超える新しいデータを同時に導入すると、複製が失敗する可能性があります。これにより、クラスター内でレプリケーションが停止します。したがって、これらのレプリケートされないローリングデータタイプのアップグレード操作を実行する際には、データベースレベルおよびアプリケーションレベルでのデータタイプの制限と特性を認識することが重要です。制御された完全に認識された DBA 環境で、このような ATCT 操作を実行およびテストすることを強くお勧めします。これらの ATCT 操作は非対称であり、失敗するとテーブルの書き換えが長時間続く可能性がある特定の変更をバックアウトすることに注意する必要があります。

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

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

5.7.3 ALTER TYPE

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

</ div>

5.7.4 コメントオン

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

</ div>

5.7.5 シーケンスの作成

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

</ div>

5.7.6 テーブルの作成

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

</ div>

5.7.7 CREATE TABLE AS および SELECT INTO

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

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

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

5.7.8 説明

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

</ div>

EXPLAIN ANALYZE レプリケーション

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

</ div>

EXPLAIN ANALYZE ロック

EXPLAIN ANALYZE は、分析されたステートメントのロック規則に従います。

</ div>

5.7.9 付与および取り消し

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

5.7.10 ロックテーブル

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

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

</ div>

5.7.11 セキュリティラベル

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

</ div>

5.7.12 切り捨てレプリケーション

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

</ div>

5.7.13 TRUNCATE ロック

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

5.7.14 ロール操作ステートメント

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

BDR では、複製された DDL によって被参照されるロールがすべてのノードに存在する必要があります。ロールは、同じ付与、パスワードなどを持つ必要はありませんが、存在する必要があります。

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

ロール操作ステートメントには、次のステートメントが含まれます。

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

一般的に、次のいずれかです。

- システムは `bdr.role_replication = off` およびすべてのロール（ユーザおよびグループ）の変更は、外部オーケストレーションによって展開する必要があります Ansible、Puppet、Chef などのツール、または経由で明示的に複製されたツール `bdr.replicate_ddl_command(...)`; または

- システムは、BDR 対応データベースが1つだけになるように構成する必要があります PostgreSQL インスタンスでは `bdr.role_replication = on` とすべてを持っていますロール管理 DDL はそのデータベースで実行する必要があります。

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

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

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

5.7.15 制限され DDL の回避策

BDR DDL オペレーションハンドリングの制限のいくつかは回避できます。多くの場合、オペレーションを小さな変更に分割すると、単一のステートメントとして許可されないか、過度のロックが必要な望ましい結果が生成されます。

制約の追加

BDR 3.7.4 以降、DML ロックを必要とせずに、`CHECK` および `FOREIGN KEY` 制約を追加できます。これには2段階のプロセスが必要です。

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

これらのステップは、2つの異なるトランザクションで実行する必要があります。これらのステップは両方ともテーブルの DDL ロックのみを取得するため、1つ以上のノードがダウンしている場合でも実行できます。しかし、制約を検証するオーダーに、BDR はクラスター内のすべてのノードが `ADD CONSTRAINT` コマンドを認識し、制約を検証するノードがそれらのノードに `NOT VALID` 制約を作成する前に他のすべてのノードから `replicationchanges` を適用することを保証する必要があります。そのため、新しいメカニズムでは、制約の検証中にすべてのノードを起動する必要はありませんが、すべてのノードが `ALTER TABLE ... ADD CONSTRAINT ... NOT VALID` コマンドを適用し、十分な進歩を遂げている必要があります。BDR は、制約を検証する前に、`consistentstate` に到達するのを待ちます。

新しい機能を使用するには、RAFT protocolversion 24 以降でクラスターを実行する必要があります。RAFT プロトコルがまだアップグレードされていない場合、古いメカニズムが使用され、DML ロック要求が発生します。

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

列を追加する

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

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

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

最良の結果を得るには、一度に数万または数十万を超える行を更新しないように、更新をチャンクにバッチします。これは、埋め込みトランザクションで PROCEDURE を使用して実行できます。

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

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

列のタイプを変更する

PostgreSQL は、次の例に、回避できる場合にテーブルの書き換えを引き起こします。

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

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

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

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

列タイプを変更する一般的なケースでは、最初に目的のタイプの列を追加します。

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

BEFORE INSERT OR UPDATE ON mytable FOR EACH ROW .. として定義されたトリガーを作成し、NEW.newcolumn を NEW.oldcolumn に割り当てて、テーブルへの新しい書き込みが新しい列を自動的に更新するようにします。

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

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

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

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

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

他のタイプの変更

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

この DDL を使用する場合、ユーザは新しいタイプを使用する前に、すべてのノードでステートメントが正常に実行されたことを保証する必要があります。これは、bdr.wait_slot_confirm_lsn() ファンクションを使用して実現できます。

例、

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

DML ステートメントで新しい値を使用する前に、DDL がすべてのノードに書き込まれていることを保証します。

5.7.16 DDL のように動作する BDRFunctions

次の BDR 管理機能は DDL のように機能します。これは、DDLreplication がアクティブで、DDL フィルター設定で許可されている場合、グローバルロックの取得を試み、そのアクションが複製されることを意味します。詳細については、個々の機能の文書を参照してください。

複製セット管理

- bdr.create_replication_set
- bdr.alter_replication_set
- bdr.drop_replication_set

- `bdr.replication_set_add_table`
- `bdr.replication_set_remove_table`
- `bdr.replication_set_add_ddl_filter`
- `bdr.replication_set_remove_ddl_filter`

紛争管理

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

シーケンス管理

- `bdr.alter_sequence_set_kind`

ストリームトリガー

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

第6章 Security and Roles

BDR3 拡張機能はスーパーユーザーのみが作成できますが、必要に応じて、pgextwlist 拡張機能をセットアップし、非スーパーユーザーが BDR3 を作成できるように構成することもできます。

デフォルトのデフォルトと管理には、スーパーユーザーアクセスは必要ありません。また、推奨もさ PostgreSQL ません。

- - bdr_superuser *-すべての BDR テーブルと機能にアクセスできる最高特権ロール。
- - bdr_read_all_stats * BDR の状態を理解するのに十分な、テーブル、ビュー、および関数への読み取り専用アクセス権を持つロール。
- - bdr_monitor *-現時点では bdr_read_all_stats と同じで、後で拡張します。
- - bdr_application * BDR を実行するアプリケーションに必要な最小限の特権。
- - bdr_read_all_conflicts -bdr.conflict_log ですべて*の競合をビューできます。

これらの BDR ロールは、BDR3 拡張機能がインストールされるときに作成されます。詳細については、以下の [BDR デフォルトロール] を参照してください。

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

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

6.1 カタログ表

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

さらに、拡張機能が所有するテーブルはレプリケーションから除外されます。

6.2 BDRFunctions と演算子

すべての BDR 関数は bdr スキーマで公開されます。これらの関数の呼び出しは、bdr を search_path に入れるのではなく、スキーマで修飾する必要があります。

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

6.3 カタログオブジェクトに対する権限の付与

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

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

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

6.4 ロール管理

ユーザーは PostgreSQL インスタンスのグローバルオブジェクト BDR が実行され、`bdr.role_replication` がオンになっているデータベースで実行された場合、`.CREATE USER` および `CREATE ROLE` コマンドは自動的に複製されます。ただし、これらのコマンドが同じ PostgreSQL インスタンスの他のデータベースで実行される場合、それらのユーザーが BDR データベースに対する権限を持っている場合でも、それらは複製されません。

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

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

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

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

6.5 ロールとレプリケーション

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

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

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

行単位セキュリティポリシーが有効になっているテーブルでは、適用時にポリシーを再適用せずに変更が複製されます。これは、FORCE ROW LEVEL SECURITY が指定されている場合でも、asNO FORCE ROW LEVEL SECURITY に適用される変更と同等です。すべてのノードの行セキュリティポリシーが同一であるか、少なくとも互換性があることが推奨されますが、強制されません。

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

6.6 接続ロール

新しい BDR ノードを割り当てる場合、bdr.create_node の local_dsn 引数と join_target_dsn of bdr.join_node_group の DSN で指定されたユーザは、データベースオブジェクトの参照、作成、および管理に頻繁に使用されます。これは、指定されたアカウントがデータベースオブジェクトの操作を直接または BDR ではなく pglogicalmodule を通じて呼び出す最初のブートストラッププロセス中に特に関連します。

BDR は、これらの DSN で SUPERUSER 権限を持つロールを使用する場合でも、権限エスカレーション攻撃を防ぐために慎重に記述されています。

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

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

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

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

6.7 特権の制限

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

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

6.7.1 外部キー特権

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

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

6.7.2 トリガー

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

`BDR` は、誰がテーブルにトリガーを作成できるかについてより厳しいルールを使用することにより、この問題を軽減します。

- スーパーユーザ
- `bdr_superuser`
- テーブルの所有者は、PostgreSQL と同じルールに従ってトリガーを作成できます。(トリガーによって使用されるファンクションに対する `EXECUTE` 権限が必要です)。
- テーブルに対する `TRIGGER` 権限を持つユーザーは、次の場合にのみトリガーを作成できます。それらは、同じ所有者が所有するファンクションを使用してトリガーを作成しますテーブルで、標準の PostgreSQL ルールを満たします(ここでも `EXECUTE` が必要です) ファンクションに対する権限)。したがって、テーブルとファンクションの両方が同じ所有者を持ち、所有者は、テーブルに対する `TRIGGER` 権限と `EXECUTE` の両方をユーザに与えることにしましたファンクションに対する権限、そのユーザが作成することは問題ないと想定されます。このファンクションを使用したそのテーブルのトリガー。
- テーブルに対する `TRIGGER` 権限を持つユーザーは、次を使用してトリガーを作成できます。 `EXECUTE` がある場合、`SECURITY DEFINER` 句で定義されている関数それらの権限。この句により、ファンクションは常にコンテキストで実行されます標準の PostgreSQL と `BDR` の両方におけるファンクション自分自身の所有者の。

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

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

これらのチェックは `BDR 3.6.19` で追加されました。以前のバージョンの動作に依存するアプリケーションは、`edbdr.backwards_compatibility` を 30618 (またはそれ以下) に設定して、以前のバージョンのように動作させることができます。

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

6.8 BDR デフォルト/事前定義ロール

BDR3 拡張機能をインストールすると、BDR 定義済みロールが作成されます。BDR3 拡張機能がデータベースから削除された後も、ロールは引き続き存在するため、必要に応じて手動で削除する必要があります。これにより、同じ PostgreSQL インスタンス上の複数のデータベースで BDR を問題なく使用できます。

`GRANT ROLE DDL` ステートメントは BDR レプリケーションに参加しないため、クラスターの各ノードでこれを実行必要があることに注意してください。

6.8.1 bdr_superuser

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

6.8.2 bdr_read_all_stats

SELECT 権限

- `bdr.conflict_history_summary`
- `bdr.ddl_epoch`
- `bdr.ddl_replication`
- `bdr.global_consensus_journal_details`
- `bdr.global_lock`
- `bdr.global_locks`
- `bdr.local_consensus_state`
- `bdr.local_node_summary`
- `bdr.node`
- `bdr.node_catchup_info`
- `bdr.node_conflict_resolvers`
- `bdr.node_group`
- `bdr.node_local_info`

- `bdr.node_peer_progress`
- `bdr.node_slots`
- `bdr.node_summary`
- `bdr.replication_sets`
- `bdr.sequences`
- `bdr.state_journal_details`
- `bdr.stat_relation`
- `bdr.stat_subscription`
- `bdr.subscription`
- `bdr.subscription_summary`
- `bdr.tables`
- `bdr.worker_errors`

EXECUTE 権限

- `bdr.bdr_edition`
- `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`

6.8.3 bdr_monitor

bdr_read_all_stats からのすべての特権、プラス

EXECUTE 権限

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

6.8.4 bdr_application

EXECUTE 権限

- column_timestamps データ型のすべての関数
- CRDT データ型のすべての関数
- bdr.alter_sequence_set_kind
- bdr.create_conflict_trigger
- bdr.create_transform_trigger
- bdr.drop_trigger
- bdr.get_configured_camo_partner_of
- bdr.get_configured_camo_origin_for
- bdr.global_lock_table
- bdr.is_camo_partner_connected
- bdr.is_camo_partner_ready
- bdr.logical_transaction_status
- bdr.ri_fkey_trigger
- bdr.seq_nextval
- bdr.seq_currval
- bdr.seq_lastval
- bdr.trigger_get_committs
- bdr.trigger_get_conflict_type
- bdr.trigger_get_origin_node_id
- bdr.trigger_get_row
- bdr.trigger_get_type

- `bdr.trigger_get_xid`
- `bdr.wait_for_camo_partner_queue`
- `bdr.wait_slot_confirm_lsn`

上記の関数の多くには、使用する前に追加の特権が必要であること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ファンクション。

6.8.5 `bdr_read_all_conflicts`

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

6.9 検証

BDR は、次のツールとアプローチを使用して検証されています。

6.9.1 コベリティ

Coverity Scan は、次のルールとコーディング標準を使用して、脆弱性に対するカバレッジを提供する BDR スタックを検証するために使用されています。

- MISRA C
- ISO 26262
- ISO / IEC TS 17961
- OWASP トップ 10
- CERT C
- CWE トップ 25
- AUTOSAR

6.9.2 CIS ベンチマーク

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

	Result	Description
1.4	PASS	systemd サービスファイルが有効になっていることを確認する
1.5	PASS	データクラスターが正常に初期化されたことを確認する
2.1	PASS	ファイル権限マスクが正しいことを確認します
2.2	PASS	PostgreSQL グループメンバシップが正しいことを確認します
3.1.2	PASS	ログの宛先が正しく設定されていることを確認します
3.1.3	PASS	ログ収集機構が有効になっていることを確認します
3.1.4	PASS	ログファイルの宛先ディレクトリが正しく設定されていることを確認します
3.1.5	PASS	ログファイルのファイル名パターンが正しく設定されていることを確認します
3.1.6	PASS	ログファイルの権限が正しく設定されていることを確認します
3.1.7	PASS	' log_truncate_on_rotation' が有効になっていることを確認します
3.1.8	PASS	ログファイルの最大有効期間が正しく設定されていることを確認します
3.1.9	PASS	ログファイルの最大サイズが正しく設定されていることを確認します
3.1.10	PASS	正しい syslog 機能が選択されていることを確認します
3.1.11	PASS	PostgreSQL メッセージのプログラム名前が正しいことを確認してください
3.1.14	PASS	' debug_print_parse' が無効になっていることを確認します
3.1.15	PASS	' debug_print_rewritten ' が無効になっていることを確認します
3.1.16	PASS	' debug_print_plan' が無効になっていることを確認します
3.1.17	PASS	' debug_preTTYprint' 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
3.1.19	PASS	' log_disconnections' が有効になっていることを確認します
3.1.21	PASS	' log_hostname' が正しく設定されていることを確認します
3.1.23	PASS	' log_statement' が正しく設定されていることを確認します
3.1.24	PASS	' log_timezone' が正しく設定されていることを確認します
3.2	PASS	PostgreSQL Audit Extension (pgAudit) が有効になっていることを確認します
4.1	PASS	sudo が正しく構成されていることを確認します
4.2	PASS	過度の管理者権限が取り消されていることを確認します
4.3	PASS	過剰なファンクション権限が取り消されていることを確認します
4.4	PASS	テスト済み過剰な DML 特権が取り消されていることを確認します
5.2	Not Tested	' ホスト' TCP/IP ソケットを介したログインが正しく構成されていることを確認します
6.2	PASS	' バックエンド' ランタイムパラメータが正しく構成されていることを確認します
6.7	Not Tested	FIPS 140-2 OpenSSL 暗号化が使用されていることを確認します
6.8	PASS	SSL が有効になっていて正しく構成されていることを確認します
7.3	PASS	WAL アーカイピングが構成され機能していることを確認します

手動で監査した場合、テスト 5.2 は合格できますが、自動化可能なテストはありません。

テスト 6.7 は、CentOS を使用したデフォルトの展開で成功しますが、Debian バリエーションの追加パッケージソフトが必要です。

第7章 Conflicts

BDR は、アクティブ/アクティブまたはマルチマスター DBMS です。非同期で使用する場合、マルチプルの異なるノードから同じ行または関連する行に書き込むと、標準データ型を使用するときにデータの競合が発生する可能性があります。

競合はエラーではありません-ほとんどの場合、BDR によって発生したときに自動的に検出および解決できるイベントです。解決はアプリケーションの性質とデータの意味に依存するため、競合を解決する方法に関して、BDR がアプリケーションにレンジ選択肢を提供することが重要です。

デフォルトでは、競合行レベルで解決されます。つまり、2つのノードからの変更が競合する場合、ローカルタプルまたはリモートタプルを選択し、他のタプルを破棄します。例、2つの競合する変更のコミットタイムスタンプを比較し、新しいものを保持する場合があります。これにより、すべてのノードが同じ結果に収束し、クラスター全体でコミット順のようなセマンティクスが確立されます。

この章では、標準データ型と行レベルの競合について詳しく説明します。

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

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

競合を回避したい場合は、これらの機能を BDR で使用できます。

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

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

一部のクラスタリングシステムは、データへの同時アクセスを防ぐために分散ロックメカニズムを使用します。これらは、サーバーが非常に近い場合に合理的に実行できますが、許容可能なパフォーマンスのために非常に低いレイテンシが重要である地理的に分散したアプリケーションをサポートできません。

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

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

7.1 競合の発生方法

ノード間の競合は、関連するすべてのトランザクションが同じノードで同時に発生した場合に起こり得ない一連のイベントの結果として発生します。ノードはトランザクションのコミット後にのみ変更を交換するため、各トランザクションはコミットしたノードで個別に有効ですが、同時に他の競合する作業を行った別のノードに適用した場合は無効になります。

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

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

BDR には分散トランザクションマネージャまたはロックマネージャがありません。これが、レイテンシとネットワークパーティションで適切に機能する理由のパートです。その結果、デフォルトの遅延レプリケーションを使用する場合、異なるノード上のトランザクションは互いに完全に独立して実行されます。ノード間の独立性が低いと、競合を完全に回避できます。これが、BDR がこれが重要な場合のレプリケーションを積極的に提供する理由です。

7.2 競合の種類

7.2.1 主キーまたは一意の競合

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

行の競合は次のとおりです。

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

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

INSERT / INSERT の競合

最も一般的な競合 INSERT / INSERT は、2つの異なるノード上の INSERTs が同じ PRIMARY KEY 値（または noPRIMARY KEY が存在する場合、単一の UNIQUE 制約に同じ値）を持つタプルを作成する場合に発生します。

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

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

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

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

複数の一意の Constraints に違反する挿入

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

このような競合が発生した場合、レプリケーションを続行オーダーには一部の行を削除する必要があります。`multiple_unique_conflicts` のリゾルバ設定に応じて、適用プロセスはエラーで終了するか、着信行をスキップするか、行の一部を自動的に削除します。自動削除では、常に正しい PRIMARY KEY で行を保持し、他の行を削除しようとします。

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

競合トリガーを使用して異なる動作を定義することもできます。

更新/更新の競合

異なるノード上の2つの同時 UPDATE が同じタプルを変更する場合（ただし、PRIMARY KEY は変更しない）、リプレイ時に UPDATE / UPDATE の競合が発生する可能性があります。

これらは、構成と状況に基づいて異なる競合の種類を生成できます。テーブルが [行バージョン競合検出] で構成されている場合、オリジナルの（キー）行がローカル行と比較されます; それらが異なる場合、`update_differing` 競合が生成されオリジン。行がチェックされます（オリジンは、現在の行が由来するノードです）。変更された場合、`update_origin_change` の競合が生成されます。他のすべての場合、UPDATE は通常、競合が発生することなく適用されます。

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

プライマリキーの更新の競合

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

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

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

説明するために、非常にシンプルスキーマの例を作成しましょう。

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

主キー列の更新が可能のため、この SQL は成功します。

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

…しかし、テーブルにマルチプルの行がある場合、例えば：

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

…その後、いくつかの更新が成功します：

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

```
SELECT * FROM pktest;
```

```
pk | val
```

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

```
* 2 | 1
```

```
* 4 | 3
```

```
(2 rows)
```

…しかし、他の UPDATE は制約エラーで失敗します：

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

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

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

そのため、PK を更新する PostgreSQL アプリケーションの場合、BDR がなくてもランタイムエラーを回避するように非常に注意してください。

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

これらの 2 つの変更を同時に実行すると機能します。

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

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

(次のページに続く)

(前のページからの続き)

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

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

```
node1: UPDATE pktest SET pk=1 WHERE pk = 3;
node2: UPDATE pktest SET pk=2 WHERE pk = 3;
```

…各ノードで異なるデータを残す：

```
node1:
SELECT * FROM pktest;
pk | val
----+-----
* 1 | 1
* 5 | 3
(2 rows)

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

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

競合が同時に発生すると問題が発生します。これら2つの変更を同時に実行することは簡単には解決できません。

```
node1: UPDATE pktest SET pk=6, val=8 WHERE pk = 5;
node2: UPDATE pktest SET pk=6, val=9 WHERE pk = 5;
```

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

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

update_pkey_exists を update_if_newer に設定することにより、sameold キーが同じ新しいキーに

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

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

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

複数の一意の Constraints に違反する更新

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

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

更新/削除の競合

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

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

削除され行は、ローカルノードがレプリケーションで遅れている場合に UPDATE を受信するまでに、データベースからクリーンアップできます。この場合、BDR は UPDATE / DELETE conflicts と [INSERT / UPDATE Conflicts] を区別できず、単に、`update_missing` の競合を生成します。

競合する別のタイプの DELETE と UPDATE は、行がローカルで UPDATED になった後に実行される DELETE 操作です。このシチュエーションでは、結果は使用される競合検出のタイプに依存します。デフォルトの [Origin Conflict Detection] を使用すると、競合はまったく検出されず、DELETE が適用され、行が削除されます。行バージョンの競合検出を有効にすると、`delete_recently_updated` の競合が生成されます。この競合タイプのデフォルトの解決策は、DELETE を適用して行を削除することですが、これは競合トリガーを介して構成または処理できます。

INSERT / UPDATE の競合

オペレーションのデフォルトの非同期モードを使用する場合、ノードはオリジナルの INSERT が受信される前に行の anUPDATE を受信する場合があります。これは、3 つ以上のノードがアクティブな場合にのみ発生します（以下の **3 つ以上のノードとの競合** を参照）。

これが発生すると、update_missing の競合が生成されます。デフォルトの競合リゾルバは insert_or_skip ですが、代わりに insert_or_error または skip を使用できます。insert-or-action を実行するリゾルバーは、可能であれば（行全体が受信されたとき）UPDATE からのデータに基づいて新しい行を INSERT に最初に試行します。行の再構築を可能にするには、テーブルに haveREPLICA IDENTITY FULL がニーズであるか、行に TOAST されたデータが含まれていない必要があります。

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

挿入/削除の競合

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

現在、BDR はこの競合タイプを検出できません。INSERT 操作は競合タイプを生成せず、INSERT が適用されます。

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

削除/削除の競合

2 つの異なるノードが同じタブルを同時に削除すると、DELETE / DELETE の競合が発生します。

これにより、常に delete_missing の競合が生成されます。これは、デフォルトではオペレーションをスキップすることで解決されます。

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

3 つ以上のノードとの競合

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

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

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

また、ノード 1 からノード 3 へのレプリケーションラグは、次のシーケンスのアクションを許可するのに十分な大きさである必要があります。

1. ノード 2 はノード 12 から INSERT を受信します。ノード 2 は UPDATE3 を実行します。ノード 3 はノード 24 から UPDATE を受け取ります。ノード 3 はノード 1 から INSERT を受け取ります

`insert_or_error`（または、場合によっては `update_missing` 競合タイプの場合は `insert_or_skip` 競合リゾルバー）を使用することは、競合に対する実行可能な緩和戦略です。ただし、このオプションを有効にすると、INSERT / DELETE の競合が発生することに注意してください。以下を参照してください。

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

これらが問題である場合、`update_recently_deleted` として正しく検出されるように、tableor データベースの凍結設定を調整することをお勧めします。

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

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

BDR は、`update_missing` の競合で発生する make に、各 INSERT を最近削除されたチェックに入れることを選択できます。ただし、これを行うと、大部分のユーザーにペナルティが科せられるため、現時点では `delete_missing` をログに記録するだけです。

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

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

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

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

BDR は、レプリケーションが正しく機能する make に識別子の BDR 性に依存しているため、後者の場合に問題があります。

同一一意の識別子を挿入、削除し、後で再利用するアプリケーションは、問題を引き起こす可能性があります。これは、ABA 問題として知られています。BDR には、行が現在の行、最後の行、またははるかに古い行であるかどうかを知る方法がありません。< https://en.wikipedia.org/wiki/ABA_problem >

一意の識別子の再利用もまた、ビジネス上の問題です。これは、時間の経過とともに一意の識別ができないため、監査、トレーサビリティ、および適切なデータ品質が妨げられるためです。アプリケーションは一意の識別子を再利用する必要はありません。

変更がシステムを通過するのにかかる時間インターバル内に発生する識別子の再利用は、困難を引き起こします。通常のオペレーションではその時間は短いかもしれませんが、ダウンロードはその間隔を数時間または数日に延長する場合があります。

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

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

7.2.2 外部キー制約の競合

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

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

最初に問題を説明してから、解決ソリューションを提供します。

シングルマスター PostgreSQL では、参照されるテーブルの値を参照する INSERT / UPDATE は、行レベルのロックを取得する前に DELETE が完了するまで待機する必要があります。DELETE が被参照した値を削除すると、INSERT / UPDATE は FK チェックに失敗します。

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

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

親子リレーション (Orders-> OrderItems など) では、これを行うのは一般的ではありません。OrderItem を完全に削除するよりも、OrderItem をキャンセル済みとしてマークする可能性が高くなります。リファレンス/ルックアップデータの場合、新しいファクトデータに同じ値を使用すると同時にエントリを完全に削除するのは奇妙です。

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

最初の解決策は、FK の使用を、通常一度に 1 つのノードのみから変更されるか、まれにしか変更されない、または変更の並行性がアプリケーションを介する密接に関連するエンティティに制限することです。これにより、アプリケーションレベルでの FK 違反が回避されます。

2 番目の解決策は、BDR が提供する関数 `bdr.ri_fkey_trigger()` および `bdr.ri_fkey_on_del_trigger()` を使用して、このケースから保護するトリガーを追加することです。BEFORE トリガーとして呼び出されると、これらの関数は FOREIGN KEY 情報を使用して、SET NULL 制約があるかのように、参照列を NULL に設定することで FK の異常を回避します。テーブルごとに FK 違反を防ぎます。

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

RefData の被参照された row がすでに削除されている場合、Fact で列を NULL に設定するトリガーを追加します。

```
CREATE TRIGGER bdr_replica_fk_iu_trg
*   BEFORE INSERT OR UPDATE ON fact
*   FOR EACH ROW
*   EXECUTE PROCEDURE bdr.ri_fkey_trigger();

ALTER TABLE fact
*   ENABLE REPLICA TRIGGER bdr_replica_fk_iu_trg;
```

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

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

ALTER TABLE refdata
*   ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

両方のトリガーを追加すると、外部キーがぶら下がるのを防ぎます。

7.2.3 競合を打ち切る

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

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

ユーザーは、次のいずれかのアクションを実行できます。

- TRUNCATE が他の同時 DML と一緒に実行されないようにします
 - LiveCompare を使用して、このような矛盾を強調します。
- TRUNCATE を WHERE 句のない DELETE ステートメントに置き換え、これはパフォーマンスが非常に低い可能性が高いことに注意してくださいより大きなテーブル。
- `bdr.truncate_locking = 'on'` を設定して TRUNCATE コマンドを設定しますロック動作。TRUNCATE が `bdr.ddl_locking` に従うかどうかを決定します設定。これすべてのノードが必要なため、TRUNCATE のデフォルトの動作ではありません起きているので、すべての場合において可能または望ましいとは限りません。

7.2.4 除外制約の競合

BDR は除外コンストレインをサポートせず、その作成を妨げます。

既存のスタンドアロンデータベースを BDR データベースに変換する場合、すべての除外コンストレインを手動で削除する必要があります。

分散非同期システムでは、異なるノード上のすべてのトランザクションが完全に分離されているため、制約に違反する行が存在しないことを保証ことはできません。除外コンストレインは、排他制約違反のためにノードから他のノードにリプレイが進まないリプレイデッドロックを引き起こします。

BDR で強制的に排他制約を作成する場合、またはスタンドアロンデータベースを BDR に変換するときに既存の制約を削除しない場合は、レプリケーションが中断することを予想する必要があります。再度進行させるには、リモートトランザクションを適用できるように、着信リモートタプルが競合するローカルタプルを削除または変更します。

7.2.5 ロールとテーブルスペースの違いによるデータの競合

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

例、**node1** には **fred** 名前付けのユーザがいますが、そのユーザは **node2** で作成されていません。**node1** の **fred** がテーブルを作成する場合、所有者が **fred** に設定されて複製されます。DDL コマンドが **node2** に適用されると、**fred** 名前付けのユーザがないため、DDL は失敗します。この失敗により、PostgreSQL ログに **ERROR** が出力されます。

BDR が実行されているデータベースにユーザ **fred** を作成することにより、この競合を解決するには管理者の介入が必要です。今後これを解決するには、`bdr.role_replication = on` を設定することをお勧めします。

7.2.6 ロックの競合とデッドロックの中止

BDR ライタプロセスは通常のユーザセッションとほとんど同じように動作するため、行とテーブルのロックに関する通常のルールに従います。これにより、BDR ライタプロセスが、ユーザトランザクションによって、または相互に保持されているロックを待機する場合があります。

関連するロックには以下が含まれます。

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

BDR ライタプロセスは、ユーザトランザクションでデッドロックすることもできます。この場合、ユーザトランザクションは、ライタプロセスが保持するロックを待機しています。2 つのライタプロセスも互いにデッドロックする可能性があります。PostgreSQL のデッドロックディテクタは、問題のあるトランザ

クシヨンの 1 つに介入して終了します。BDR ライタプロセスが終了した場合、単にリトライし、通常は成功します。

これらの問題はすべて一時的なものであり、通常は管理者の操作は必要ありません。ライタプロセスがアイドル状態のユーザセッションのロックの背後で長時間停滞している場合、管理者はユーザセッションを終了してレプリケーションを再度フローさせることができますが、これは別のユーザセッションに影響する長いロックを保持しているユーザーと同じです。

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

7.2.7 発散する対立

異なるノードで同じであるはずのデータが予期せず異なる場合、分岐の競合が発生します。さまざまな競合が発生することはありませんが、このような競合のすべてが執筆時点で確実に防止できるわけではありません。

すべてのノードが変更をリプレイする前に別のノードが同じ行のキーを変更すると、行の **PRIMARY KEY** を変更すると、競合が生じる可能性があります。プライマリキーを変更しないか、指定された 1 つのノードでのみ変更してください。

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

管理者は、このような競合を手動で解決する必要があります。競合の性質によっては、`bdr.ddl_replication` や `bdr.ddl_locking` などの高度なオプションの使用が必要になる場合があります。ただし、これらのオプションを不注意に使用 `make` と事態がさらに悪化する可能性があり、あらゆる種類の競合を解決するための一般的な指示を与えることはできません。

7.2.8 TOAST サポートの詳細

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

ロジカルデコーディング (BDR はその上に構築されます) およびロジカルレプリケーションでの **TOAST** 値のハンドリングは、テーブルのメイン行のパートとして格納されるインラインデータとは異なります。

TOAST 値は、値が変更された場合にのみトランザクションログ (WAL) にログオン。これは、特に、更新された列の値を変更しなかった **UPDATE** ステートメントがその列のない行を生成するため、**UPDATE** の競合をハンドリングするときに問題を引き起こす可能性があります。[**INSERT / UPDATE Conflicts**] で述べたように、`update_missingconflict` が `insert_or_error` を使用して解決され、**TOAST** 列が欠落している場合、BDR はエラーを生成します。

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

1. ノード A : `txn A1` は `UPDATE SET col1 = ' トースト data ... '` を実行し、`first2` をコミットします。ノード B : `txn B1` は `UPDATE SET other_column = ' anything else '` ;そして A13 以降にコミットします。ノード C : ノード A への接続が遅れる 4。ノード C : `txn B1` が最初に適用され、`col1` の **TOAST** された列が欠

落しますが、競合せずに適用されます 5。ノード C: txn A1 は (update_origin_change で) 競合し、スキップされます 6。ノード C は A1 からのトーストされたデータを永遠に失います

BDR が最近複製された行にローカルの UPDATE を検出すると、BDR が TOAST 列の独自のロギングを追加するため、BDR を使用する場合 (ビルトインロジカルレプリケーションまたはマルチマスターのプレーン pglogical を使用する場合)、通常は上記の問題はありません TOAST 列の変更。localUPDATE は TOAST を変更していません。したがって、BDR は、マルチプルのノードで更新が発生した場合 (つまり、タプルのオリジンが変更された場合) に WAL ロギングが増加する代わりに、TOAST されたデータの不整合を防ぎます。すべての更新が単一のノードから行われた場合、追加の WAL オーバーヘッドはゼロになります BDR AlwaysOn アーキテクチャでは通常そうです。

!!! Note * メインテーブルでも同じことを行わずに、TOAST テーブルでのみ VACUUM FULL または CLUSTER を実行すると、余分なロギングが機能するために必要なメタデータが削除されます。存在しません。

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

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

TOASTed 列に関連するこれらの問題は、行全体がキーのパートであると見なされるため、この設定は常にキーの一部として TOAST された値をログするため、edREPLICA IDENTITY FULL のあるテーブルに影響しません。BDR と pglogical の両方は、新しい行を再構築するのに十分スマートであり、キー行の欠落データを埋めます。その結果、REPLICA IDENTITY FULL を使用すると WAL サイズが大幅に増加する可能性があることに注意してください。

7.3 競合の回避または許容

ほとんどの場合、競合を回避するか、競合を許容するようにアプリケーションを設計できます。

競合は複数のノードで同時に発生している場合にのみ発生する可能性があるため、競合を回避する最も簡単な方法は、1つのノードにのみ書き込みを行うか、特定の行から一度に特定のノードから特定の行にのみ書き込みを行うことです。

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

ノードがダウンする直前に変更を make と、変更が失われたように見える場合があります。その後、同じ変更を再度行って、異なるノードを介して2つの更新を make ことができます。ダウンしたノードが復旧すると、古い変更を他のノードに送信しようとしませんが、データの最後の更新が保持されるため拒否されます。

INSERT / INSERT の競合の場合、Global Sequences を使用すると、このタイプの競合を完全に防ぐことができます。

ルーム予約アプリケーションなど、オブジェクト間に関係を割り当てるアプリケーションの場合、update_if_newer を適用しても、許容できるビジネス結果が得られない場合があります。最も簡単な解決策は、

保証を使用して、1つの予約のみが成功するようにすることです。アプリケーションに応じて、より複雑な方法が可能になる場合があります。たとえば、各ノードに 100 シートを割り当て、そのノードのライタがそれらを予約できるようにします。

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

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

最善のアクションコースは、頻繁に競合を発生させ、BDR の競合解決メカニズムと連携して競合に対処するようにアプリケーションを設計することです。

7.4 競合検出

BDR は、競合検出のために次のメカニズムを提供します。

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

7.4.1 オリジン競合検出

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

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

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

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

オリジン情報は、行が凍結されるまでしか使用できません。凍結された行に到着する更新は競合を引き起こさないため、すべての場合に適用されます。これは、`bdr_init_physical` で新しいノードを追加する通常のケースです。そのため、競合を発生させると、多くの誤検知ケースが発生します。

しばらくオフラインになっていたノードが再接続してデータ変更の送信を開始すると、新しく到着した更新が更新した凍結行よりも実際に古い場合、これにより発散エラーが発生する可能性があります。挿入と削除は、このシチュエーションの影響を受けません。

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

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

Postgres の他の亜種では、ユーザーはこのシチュエーションを慎重に管理する必要がある場合があります。

フリーズは通常、バキュームされる行が現在の xid の `vacuum_freeze_min_age` xid よりも古い場合に発生します。つまり、これらのパラメーターに適切な高い値を設定する必要があります。

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

トランザクションレートに基づいて値を選択する必要があります。競合データがデータベースサーバから削除されるまでの猶予期間はダウンタイムです。例、ディスク上が 5 億に設定されている場合、競合データが削除される前に、1000 TPS を実行しているノードがわずか 5.5 日ギガバイトする可能性があります。fromlower 設定。

最初に推奨される設定は次のとおりです。

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

vacuum_freeze_min_age = 500000000

# 90% of autovacuum_freeze_max_age
vacuum_freeze_table_age = 900000000
```

以下に注意してください。

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

7.4.2 行バージョンの競合検出

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

行バージョンの競合検出では、3つの項目を有効にする必要があります。これらの手順のいずれかが正しく実行されない場合、[Origin Conflict Detection] が使用されます。

1. BDR ノードグループに対して `check_full_tuple` を有効にする必要があります。
2. 行バージョンの競合検出を使用するすべてのテーブルで `REPLICA IDENTITY FULL` を有効にする必要があります。

3. `bdr.alter_table_conflict_detection` を使用して、テーブルで行バージョン追跡を有効にする必要があります。この関数は、新しい列（ユーザ定義名前）と新しい列の値を管理する UPDATE トリガーを追加します。列は INTEGER タイプとして作成されます。

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

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

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

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

7.4.3 `bdr.alter_table_conflict_detection`

テーブル所有者は、特定のテーブルの競合検出の動作を変更できます。

あらすじ

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

パラメーター

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

競合検出のために認識されている方法は次のとおりです。

- **row_origin**-タプルに対して行われた以前の変更のオリジン（参照 [元の競合検出] 上記）。これは、サポートされている唯一のメソッドですテーブルに追加の列は必要ありません。
- **row_version**-行バージョン列（**行バージョンの競合検出**を参照）上記）。
- **column_commit_timestamp**-列ごとのコミットタイムスタンプ（章）。
- **column_modify_timestamp**-列ごとの変更タイムスタンプ（章）。

注

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

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

このファンクションは、列レベルの競合解決が有効になっているリレーションで 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に可視されます。

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

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

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

7.4.4 競合タイプのリスト

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

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

- `target_table_missing`-ターゲットテーブルがありません。
- `apply_error_ddl`-適用時に PostgreSQL によりエラーがスローされました複製された DDL コマンド。

7.5 競合の解決

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

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

7.5.1 `bdr.alter_node_set_conflict_resolver`

このファンクションは、特定のノードでの競合解決の動作を設定します。

あらすじ

```
bdr.alter_node_set_conflict_resolver(node_name text,  
    *                               conflict_type text,  
    *                               conflict_resolver text)
```

パラメーター

- `node_name`-変更されるノードの名前
- `conflict_type`-設定が適用される競合タイプ（[競合タイプのリスト](#)を参照）
- `conflict_resolver`-指定された競合タイプに使用するリゾルバ（[コンフリクトリゾルバのリスト](#)を参照）

注

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

このファンクションによる構成の変更は、`[bdr.create_node_group]` または `bdr.alter_node_group` で指定された競合解決のデフォルトの動作をオーバーライドすることに注意してください。

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

7.5.2 コンフリクトリゾルバのリスト

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

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

insert_exists、update_differing、update_origin_change、update_missing、multiple_unique_conflicts、update_recently_deleted、update_pkey_exists、delete_recently_updated、および delete_missing の競合タイプは、[Conflict Triggers](#) を使用するユーザー定義ロジックでも解決できます。

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

	insert_exists	update_differing	update_origin_change	update_origin_missing	update_recently_deleted	update_pkey_exists	delete_recently_updated	delete_pkey_missing	target_column_missing	source_column_missing	target_table_missing	multiple_unique_conflicts
error	X	X	X	X	X	X	X	X	X	X	X	X
skip	X	X	X	X	X	X	X	X	X	X	X	X
skip_if_recently_dropped											X	
update_if_newer	X	X	X			X						
update	X	X	X			X	X					X
insert_or_skip				X	X							
insert_or_error				X	X							
ignore									X			
ignore_if_null									X			
use_default_value										X		
conflict_trigger	X	X	X	X	X	X	X	X				X

7.5.3 デフォルトの競合解決

Conflict Type	Resolver
insert_exists	update_if_newer
update_differing	update_if_newer
update_origin_change	update_if_newer
update_missing	insert_or_skip
update_recently_deleted	skip
update_pkey_exists	update_if_newer
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

7.5.4 競合解決のリスト

競合解決は、競合リゾルバによって選択された解決の種類を表し、競合を解決するために行われた特定のアクションに対応します。

現在、`conflict_resolution` パラメータでは次の競合解決がサポートされています。

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

7.6 競合ログ

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

7.6.1 bdr.alter_node_set_log_config

ノードの競合ロギング構成を設定します。

あらすじ

```
bdr.alter_node_set_log_config(node_name text,  
    *                               log_to_file bool DEFAULT true,  
    *                               log_to_table bool DEFAULT true,  
    *                               conflict_type text[] DEFAULT NULL,  
    *                               conflict_resolution text[] DEFAULT NULL)
```

パラメーター

- **node_name**-変更されるノードの名前
- **log_to_file**-サーバーログログに記録するかどうか
- **log_to_table**-bdr.conflict_history テーブルにログを記録するかどうか
- **conflict_type**-ログに記録する競合タイプ。NULL（デフォルト）はすべてを意味します
- **conflict_resolution**-ログの競合解決。NULL（デフォルト）はすべてを意味します

注

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

このファンクションはトランザクションに対応しています。変更はロールバックでき、現在のトランザクションに可視されます。

競合ログ設定の一覧表示

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

テーブルへの競合のログイン

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

このテーブルは、`local_time` 列でパーティション分割されたレンジです。テーブルは自動パーティションによって管理されます。デフォルトでは、毎日新しいパーティションが作成され、過去 1 か月間の競合が維持されます。その後、古いパーティションは自動的に削除されます。自動パーティションは、7~14 個のパーティションを事前に作成します。`bdr_superuser` はこれらのデフォルトを変更する場合があります。

BDR によって管理されるすべてのテーブルに対して生成された競合はこのテーブルに記録されるため、正当なユーザーのみが競合するデータを読み取れるように保証ことが重要です。これを行うには、`bdr.conflict_history` テーブルで ROW LEVEL SECURITY ポリシーを定義します。テーブルの所有者のみが、それぞれのテーブルの競合を読み取ることができます。基になるテーブル自体に RLS ポリシーが定義、有効化、および適用されている場合、所有者でさえ競合を読み取ることができません。FORCE オプションで作成された RLS ポリシーは、テーブルの所有者にも適用されます。その場合、基礎となるテーブルの一部またはすべての行は、所有者でさえも読み取りできない場合があります。したがって、競合ログテーブルに対してより厳しいポリシーを適用します。

デフォルトのロール `bdr_read_all_conflicts` は、`bdr_superuser` ロールを付与せずに、`bdr.conflict_history` テーブルに記録されたすべての競合の詳細を表示する必要があるユーザーに付与できます。

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

7.6.2 競合の報告

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

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

7.7 LiveCompare によるデータ検証

LiveCompare は、2 つのデータベースを比較して、それらが同一であることを確認するように設計されたユーティリティプログラムです。

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

LiveCompare は、受信行を継続的にモニタするためにも使用できます。コンテキスト情報を失うことなく停止および開始できるため、都合の良い時間に実行できます。

LiveCompare は、マルチプルのテーブルの同時チェックを許可し、いくつかのテーブルまたはテーブル内の行のセクションのチェックを許可するように設定できます。チェックは、最初に全行ハッシュを比較してから実行されます。有用なサイズのバッチの行。

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

詳細については、LiveCompare のドキュメントを参照してください。

第8章 Sequences

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

ただし、PostgreSQL の標準シーケンスはマルチノード対応ではなく、ローカルノードで一意の値のみを生成します。このようなシーケンスによって生成された一意の ID は、マルチマスターレプリケーションで競合とデータロス（INSERTs の破棄による）を引き起こすため、これは重要です。

8.1 BDR グローバルシーケンス

このため、BDR は、**グローバルシーケンス**と呼ばれる、BDR グループ全体の `bigint` または `bigserial` データ型のシーケンスを使用して、一意の ID を生成するアプリケーション透過的な方法を提供します。

BDR グローバルシーケンスは、アプリケーションがデータベースを使用して、ほとんどの場合（必ずしもすべてではない）に機能する非同期分散システムで一意の合成キーを生成する簡単な方法を提供します。

BDR グローバルシーケンスを使用すると、挿入の競合の問題を回避できます。グローバルシーケンスを使用している列に `PRIMARY KEY` または `UNIQUE` 制約を定義した場合、ノードが他のノードと同じ値を取得することはできません。BDR がノード間の挿入を同期する場合、競合することはありません。

BDR グローバルシーケンスは PostgreSQL シーケンスを拡張するため、クラッシュセーフです。それらを使用するには、`bdr_application` ロールが付与されている必要があります。

グローバルシーケンスにはさまざまなアルゴリズムがあります。

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

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

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

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

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

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

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

8.1.1 タイムシャードシーケンス

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

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

Timeshard シーケンスは、標準シーケンスとは異なる方法で一意の ID を生成します。アルゴリズムは、シーケンス番号に3つのコンポーネントを使用します。シーケンスの最初のコンポーネントは、シーケンス番号生成時のタイムスタンプです。シーケンス番号の2番目のコンポーネントは、各 BDR ノードに割り当てられた一意の ID です。これにより、異なるノードからの ID は常に異なることが保証されます。最後に、3番目のコンポーネントは、ローカルシーケンス自分自身によって生成された番号です。

シーケンス番号に一意のノード ID を追加オーダーことで、競合がないことを保証できますが、順序付けの別の有用なプロパティを保持する必要もあります。タイムスタンプを最初に置くことでこれが保証されます。

タイムハードシーケンスには、いくつかの制限と注意事項が適用されます。

タイムシャードシーケンスは 64 ビット幅で、`bigint` または `bigserial` が必要です。生成される値は少なくとも 19 桁です。実用的な 32 ビット `integer` バージョンはないため、`serialsequences` では使用できません。代わりにグローバルに割り当てられたレンジシーケンスを使用してください。

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

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

おそらくもっと重要なのは、タイムスタンプコンポーネントが 2050 年に値を使い果たし、`bigint` と組み合わせて使用した場合、値は 2033 年に負の数にラップれることを意味します。この日付を超えてアプリケー

ションをデプロイする予定がある場合は、下記の *UUID*、*KSUUID*、およびその他のアプローチのいずれかを試すか、代わりにグローバルに割り当てられたレンジシーケンスを使用してください。

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

同様に、*START*、*MINVALUE*、*MAXVALUE*、および *CACHE* の設定は、基になるシーケンスで変更される場合がありますが、そうすることには利点がありません。シーケンスの下位 14 ビットが使用され、残りは破棄されるため、値のレンジの制限は関数の結果に影響しません。同じ理由で、*setval()* はタイムハードシーケンスには役立ちません。

8.1.2 グローバルに割り当てられたレンジのシーケンス

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

タイムハードシーケンスとは異なり、*galloc* シーケンスは、PostgreSQL が提供するすべてのシーケンスデータ型（*smallint*、整数、*bigint*）をサポートします。つまり、*gallocsequences* は、JavaScript で整数を使用するなど、64 ビットシーケンスが問題のある環境で使え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ています。

- *smallint*-1000 個の数字
- 整数-1 000 000 の数字
- *bigint*-1000 000 000 の数字

各ノードは *seq_chunk_size* の 2 つのチャンクを割り当てます。1 つは現在の使用に加え、将来の使用のために予約したチャンクです。したがって、1 つのノードから生成される値は単調に増加します。ただし、グローバルに表示すると、生成された値はまったく順序付けられません。これにより、*b-tree* インデックスへの影響によりパフォーマンスが低下する可能性があり、通常、生成された値はレンジパーティションキーとして有用ではないことを意味します。

galloc シーケンスの主な欠点は、割り当てられたレンジが使い果たされると、シーケンスジェネレータはノード間通信を必要とするローカルノードの次の範囲についてコンセンサスを要求する必要があることです。BDR グループにアクセスできません。これは今後のリリースで回避される可能性があります。

CACHE、*START*、*MINVALUE*、および *MAXVALUE* オプションは *galloc* シーケンスで正しく機能しますが、シーケンスを *galloc* 種類に変換する前にそれらを設定する必要があります。*INCREMENT BY* オプションも正しく機能しますが、各シーケンスデータ型に割り当てられた上記の範囲以上のインクリメントを割り当てることはできません。*setval()* は *galloc* シーケンスのグローバル状態をリセットしないため、使用しないでください。

galloc シーケンスにはいくつかの制限が適用されます。BDR は、特別な BDR カタログ `bdr.sequence_alloc` の galloc シーケンスを追跡します。このカタログは、galloc シーケンスに現在割り当てられているチャンクを追跡するために必要です。シーケンス名前と名前空間はこのカタログに保存されます。シーケンスチャンクのアロケーションは RAFT で管理され、シーケンス名前/ namespace への変更はレプリケーションストリームで管理されるため、BDR は現在、gallocsequences の名前変更、別の名前空間への移動、galloc シーケンスを含む名前空間の名前変更をサポートしていません。ユーザは、アプリケーションスキーマの設計中にこの制限に留意する必要があります。

使用法

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

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

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

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

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

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

```
SELECT * FROM bdr.sequence_alloc
    * WHERE seqid = 'public.categories_category_seq'::regclass;
    *      seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |
    seq_last_alloc

-----+-----+-----+-----+
↳-----
categories_category_seq |      1000000 |      4000333 |          4 | 2020-
↳05-21 20:02:15.957835+00
(1 row)
```

各ノードの特定のシーケンスに現在割り当てられている範囲を表示するには、次のクエリを使用します。

- ノード Node1 は、333 から 2000333 までのレンジを使用しています。

```
SELECT last_value AS range_start, log_cnt AS range_end
* FROM categories_category_seq WHERE ctid = '(0,2)'; -- first range
range_start | range_end
-----+-----
*          334 |   1000333
(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 までのレンジを使用しています。

```
SELECT last_value AS range_start, log_cnt AS range_end
* FROM categories_category_seq WHERE ctid = '(0,2)'; -- first range
range_start | range_end
-----+-----
*    2000334 |   3000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
* FROM categories_category_seq WHERE ctid = '(0,3)'; -- second range
range_start | range_end
-----+-----
*    3000334 |   4000333
```

注単一のクエリー（WHERE ctid IN ('(0,2)', '(0,3)' ）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るためです。

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

8.2 UUID、KSUUID、およびその他のアプローチ

BDR で使用できる `globalsequences` を使用せずにグローバルに一意の ID を生成する方法は他にもあります。例：

- UUID、およびその BDR バリエント、KSUUID
- ノードごとに異なるオフセットを持つローカルシーケンス（マニュアル）
- 外部で調整された自然キー

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

8.2.1 UUID と KSUUID

代わりに、UUID キーはシーケンスを完全に回避し、128 ビットのユニバーサル意識別子を使用します。これらは非常にラージランダム値または擬似ランダム値であり、同じ値を 2 回生成することはほぼ不可能です。UUID キーを使用する場合、ノードが継続的に通信する必要はありません。

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

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

BDR は、KSUUID と呼ばれる UUID データの K-Sortable バリエントを操作するための関数を提供します。これは、PostgreSQL の標準の UUID データタイプを使用して保存できる値を生成します。KSUUID の値は、UUID 標準に従って、タイムスタンプとランダムデータの両方を格納するという点で `UUIDv1` と似ています。違いは、KSUUID が K- タイムスタンプであるということです。これにより、よりコンパクトな `btree` インデックスが生成され、検索の効率が向上し、結果データの自然な時間ソートが可能になるため、データベースキーとしてより便利になります。UUIDv1 とは異なり、KSUUID の値にはそれらが生成されたコンピュータの MAC が含まれないため、KSUUIDs を使用してもセキュリティ上の問題はありません。

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

BDR には KSUUID の 2 つのバージョン、v1 および v2 があります。従来の KSUUID v1 は非推奨になりましたが、既存のインストールをサポートするオーダーに保持されており、新規インストールには使用しないでください。v1 と v2 の内部コンテンツには互換性がないため、それらを操作する機能にも互換性はありません。KSUUID の v2 も UUID バージョン番号を格納しなくなりました。

8.2.2 ステップとオフセットのシーケンス

オフセットステップシーケンスでは、各ノードで通常の PostgreSQL シーケンスが使用されます。各シーケンスは同じ量だけ増加し、異なるオフセットで開始されます。例、ステップ 1000 では、node1 のシーケンスは 1001、2001、3001 などを作成し、node2 は 1002 を作成します、2002、3002 など。このスキームは、ノードが拡張通信できない場合でもうまく機能しますが、設計者はスキーマを確立するときにノードの最大数を指定する必要があります、ノードごとの構成が必要です。ただし、間違いは簡単に重複するシーケンスにつながる可能性があります。

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

```
CREATE TABLE some_table (
    *   generated_value bigint primary key
);

CREATE SEQUENCE some_seq INCREMENT 1000 OWNED BY some_table.generated_value;

ALTER TABLE some_table ALTER COLUMN generated_value SET DEFAULT nextval('some_seq');
```

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

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

-- On node 2
SELECT setval('some_seq', 2);
* -- ... etc
```

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

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

BDR は現在、このようなステップ/オフセットシーケンスでのノードごとのオフセットの構成の自動化を提供していません。

複合キー

ステップ/オフセットシーケンスのバリエーションは、`PRIMARY KEY (node_number, generated_value)` で構成される複合キーを使用することです。通常、ノード番号は、各ノードで異なる番号を返す関数から取得されます。このような関数は、一時的に DDL レプリケーションを無効にして定数 SQL 関数を作成するか、レプリケーションセットのパートではない 1 行のテーブルを使用して各ノードに異なる値を格納することで作成できます。

8.3 グローバルシーケンス管理インターフェイス

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

次の関数は DDL と見なされるため、DDL レプリケーションとグローバルロックが適用されることに注意してください。

8.3.1 bdr.alter_sequence_set_kind

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

BDR はこのファンクションを DDL として扱うため、DDL レプリケーションとグローバルロックが適用されます（現在アクティブな場合）。[DDL Replication] を参照してください。

あらすじ

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text)
```

パラメーター

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

注

シーケンスの種類を galloc に変更すると、そのシーケンスに最初に割り当てられたレンジは、シーケンスのスタート値を開始点として使用します。シーケンスが galloc に変更される前にすでに使用されている既存の値がある場合、次のコマンドを使用して、新しく生成された値が既存の値と競合しないように開始点を移動することをお勧めします。

```
ALTER SEQUENCE seq_name START starting_value RESTART
```

このファンクションは、DDL ステートメントと同じレプリケーションメカニズムを使用します。つまり、レプリケーションは ddl filters 構成の影響を受けます。

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

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

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

8.3.2 bdr.extract_timestamp_from_timesthard

このファンクションは、`timesthard` シーケンスのタイムスタンプコンポーネントを抽出します。結果値のタイプは「`timestamp_tz`」です。

あらすじ

```
bdr.extract_timestamp_from_timesthard(timesthard_seq bigint)
```

パラメーター

- `timesthard_seq`-タイムハードシーケンスの値

注

このファンクションは、ローカルノードでのみ実行されます。

8.3.3 bdr.extract_nodeid_from_timesthard

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

あらすじ

```
bdr.extract_nodeid_from_timesthard(timesthard_seq bigint)
```

パラメーター

- `timesthard_seq`-タイムハードシーケンスの値

注

このファンクションは、ローカルノードでのみ実行されます。

8.3.4 bdr.extract_localseqid_from_timeshard

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

あらすじ

```
bdr.extract_localseqid_from_timeshard(timeshard_seq bigint)
```

パラメーター

- timeshard_seq-タイムハードシーケンスの値

注

このファンクションは、ローカルノードでのみ実行されます。

8.3.5 bdr.timestamp_to_timeshard

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

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

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

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

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

あらすじ

```
bdr.timestamp_to_timeshard(ts timestamptz)
```

パラメーター

- ts-タイムハードシーケンス生成に使用されるタイムスタンプ

注

このファンクションはローカルノードでのみ実行されます。

8.4 KSUUID v2Functions

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

8.4.1 bdr.gen_ksuuid_v2

このファンクションは、引数として渡されたタイムスタンプの値または NULL が渡された場合は現在のシステム時刻を使用して、新しい KSUUID v2 値を生成します。システム時刻を使用して KSUUID を自動的に生成する場合は、NULL 引数を渡します。

結果値のタイプは「UUID」です。

あらすじ

```
bdr.gen_ksuuid_v2(timestampz)
```

注

このファンクションは、ローカルノードでのみ実行されます。

8.4.2 bdr.ksuuid_v2_cmp

このファンクションは、KSUUID v2 値を比較します。

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

あらすじ

```
bdr.ksuuuid_v2_cmp(uuid, uuid)
```

パラメーター

- UUID-比較する KSUUUID v2

注

このファンクションはローカルノードでのみ実行されます。

8.4.3 bdr.extract_timestamp_from_ksuuuid_v2

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

あらすじ

```
bdr.extract_timestamp_from_ksuuuid_v2(uuid)
```

パラメーター

- UUID-タイムスタンプを抽出する KSUUUID v2 値

注

このファンクションは、ローカルノードでのみ実行されます。

8.5 KSUUUID v1Functions

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

8.5.1 bdr.gen_ksuuid

このファンクションは、現在のシステム時刻を使用して、新しい KSUUID v1 値を生成します。結果値の型は「UUID」です。

あらすじ

```
bdr.gen_ksuuid()
```

注

このファンクションは、ローカルノードでのみ実行されます。

8.5.2 bdr.uuid_v1_cmp

このファンクションは、KSUUID v1 の値を比較します。

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

あらすじ

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

注

このファンクションは、ローカルノードでのみ実行されます。

パラメーター

- UUID-比較する KSUUID v1

8.5.3 bdr.extract_timestamp_from_ksuuid

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

あらすじ

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

パラメーター

- UUID-タイムスタンプを抽出する KSUUID v1 値

注

このファンクションは、ローカルノードでのみ実行されます。

第9章 Column-Level Conflict Detection

デフォルトでは、競合行レベルで解決されます。つまり、2つのノードからの変更が競合する場合、ローカルまたはリモートのタプルのいずれかを選択し、もう一方を破棄します。例、2つの競合する変更のコミットタイムスタンプを比較し、新しいものを保持する場合があります。これにより、すべてのノードが同じ結果に収束することが保証され、クラスター全体でコミット順序のようなセマンティクスが確立されます。

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

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

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

…別のノードで同時に (precedUPDATE を受信する前に) 実行ます：

```
UPDATE t SET b = 100
```

これにより、UPDATE-UPDATE の競合が発生します。update_if_newerconflict 解決では、コミットタイムスタンプを比較し、newrow バージョンを保持します。2 番目のノードが最後にコミットされたと仮定すると、(1,100) になり、列「a」への変更を事実上破棄します。

多くのユースケースでは、これは望ましい動作であり期待される動作ですが、場合によってはこれが問題になる可能性があります。例、アプリケーションの各パートが異なるノードに接続され、共有テーブルの列の専用サブセットを更新するマルチノードクラスターを検討してください。その場合、異なるコンポーネントが互いのつま先を踏んで、変更を上書きする可能性があります。

このようなユースケースでは、列レベルで特定のテーブルの競合を解決の方が適切な場合があります。これを実現するために、BDR は各列の最後の変更のタイムスタンプを個別に追跡し、それを使用して最新の値（本質的に update_if_newer）を選択します。

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

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

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

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

9.1 列レベルの競合解決の有効化と無効化

列レベルの競合解決は、`bdr.alter_table_conflict_detection()` 関数によって管理されます。

9.1.1 例

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

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

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

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

db=# \d my_app.test_table
```

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

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

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

9.1.2 列レベルの競合解決を使用したテーブルの一覧表示

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

```
SELECT nc.nspname, c.relname
FROM pg_attribute a
JOIN (pg_class c JOIN pg_namespace nc ON c.relnamespace = nc.oid)
    * ON a.attrelid = c.oid
JOIN (pg_type t JOIN pg_namespace nt ON t.typnamespace = nt.oid)
    * ON a.atttypid = t.oid
```

(次のページに続く)

(前のページからの続き)

```

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');

```

9.1.3 bdr.column_timestamps_create

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

あらすじ

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestamptz)
```

パラメーター

- `p_source`-2つのオプションは 'current' または 'コミット' です。
- `p_timestamp`-タイムスタンプは選択したソースに依存します: 'コミット' の場合、次に `TIMESTAMP_SOURCE_COMMIT`; 'current' の場合、`TIMESTAMP_SOURCE_CURRENT`。

9.2 DDL ロック

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

9.3 現在とコミットのタイムスタンプ

重要な質問は、変更された列に割り当てられるタイムスタンプです。

デフォルトでは、変更された列に割り当てられたタイムスタンプは、`clock_timestamp` から取得されたかのように `currenttimestamp` です。これはシンプルで、多くの場合、完全に正しいです（たとえば、競合する行が重複しない列のサブセットを変更する場合）。

ただし、予期しないさまざまな影響がある場合があります。

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

注：将来、ステートメントおよびトランザクションのタイムスタンプを追加する可能性があります。これにより、同時ステートメントまたはトランザクションの混合効果に関する問題に対処できます。それでも、これらのオプションはいずれも、コミットオーダーと同等の結果を生成することはできません。

実際のコミットタイムスタンプを使用することもできますが、この機能は現在実験的と見なされています。コミットタイムスタンプを使用するには、列レベルの競合解決を有効にするときに最後のパラメータを `true` に設定します。

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

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

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

9.4 列のタイムスタンプの検査

変更された列のタイムスタンプを保存する列は、トリガーによって自動的に維持されます。直接変更しないでください。例、特定の競合がどのように解決されたかを調査する際に、現在のタイムスタンプ値を調べると便利です。

この目的には3つの機能があります。

- `bdr.column_timestamps_to_text(bdr.column_timestamps)`
 - このファンクションは、タイムスタンプマッピングの人間が読み取れる表現形式を返し、値を `text` にキャストするときに使用されます。

```
db=# select cts::text from test_table;
```

```
  *                               cts
```

```
→-----
```

```
{source: current, default: 2018-09-23 19:24:52.118583+02, map: [2 : 2018-09-23 19:25:02.590677+02]}
```

```
(1 row)
```

- `bdr.column_timestamps_to_jsonb(bdr.column_timestamps)`
 - このファンクションはタイムスタンプマッピングの JSONB 表現形式を有効にし、値を `jsonb` にキャストするときに使用されます。

```
db=# select jsonb_pretty(cts::jsonb) from test_table;
*
          jsonb_pretty
-----
{
*      "map": {
*          "2": "2018-09-23T19:24:52.118583+02:00"
*      },
*      "source": "current",
*      "default": "2018-09-23T19:24:52.118583+02:00"
}
(1 row)
```

- `bdr.column_timestamps_resolve(bdr.column_timestamps, xid)`

- このファンクションは、最新のトランザクション（既にコミットされている場合）によって変更された属性のコミットタイムスタンプでマッピングを更新します。これは、コミットタイムスタンプを使用する場合にのみ重要です。例、この場合、最後のトランザクションは2番目の属性を（`attnum = 2`で）更新しました：

```
test=# select cts::jsonb from test_table;
*
          cts
-----
→ {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
→ "2018-09-23T19:29:55.581823+02:00", "modified": [2]}
(1 row)

db=# select bdr.column_timestamps_resolve(cts, xmin)::jsonb from test_table;
*
          column_timestamps_resolve
-----
→ {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
→ "2018-09-23T19:29:55.581823+02:00"}
(1 row)
```

9.5 CRDT データ型を使用した列の競合の処理

デフォルトでは、列レベルの競合解決は、単にタイムスタンプの高い値を選択し、他の値を破棄します。ただし、たとえば、情報を破棄せずに競合する値を「マージ」できる CRDT タイプを使用するなど、さまざまな（より複雑な）方法で競合を調整することができます。

`pglogical` にはそのようなデータ型は含まれていませんが、それらを個別に追加してカタログ `crdt_handlers` に登録することができます。通常のデータタイプ関数（入力/出力、…）の他に、各ローカル型はマージファンクションを実装リモート必要がありリモート。。

9.6 制限

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

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

… 1 つのノードが以下を行うと仮定します。

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

…別のノードが同時に行う場合：

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

- これらの更新はそれぞれ、最初の行で実行されると有効になるため、各ノードに渡されます。ただし、他のノードに複製する場合、結果の行は **CHECK (A > b)** 制約に違反し、問題が手動で解決されるまでレプリケーションは停止します。
- タイムスタンプマッピングを格納する列は自動的に管理されます。しないでくださいクエリで値を指定またはオーバーライドします。予測不可能な効果（とにかく可能な場合は値を無視します）。
- タイムスタンプマッピングはトリガーによってオーダーされますが、トリガーの実行は重要です。したがって、変更するカスタムトリガーがある場合ダブルおよび **pgl_c1cd** トリガーの後に実行され、変更された列は正しく検出されません。
- 変更/コミットをオーダーに通常のタイムスタンプを使用する場合、可能です競合する変更のタイムスタンプがまったく同じであること（なぜなら 2 つ以上のノードがたまたま同じタイムスタンプを生成しました）。このリスク発生する可能性があるため、列レベルの競合解決に固有ではありません通常の行レベルの競合解決でも、ノード ID をこのシチュエーションでのタイブレーカー（より高いノード ID が勝つ）。同じ変更がすべてのノードに適用されます。
- 異なるノード間にクロックスキューがある可能性があります。それながら
 - やや予期しない動作を引き起こす可能性があります

- タイムスタンプが反転しているために変更されます)、ノード間のクロックスキューは
- パラメーター `bdr.maximum_clock_skew` を使用して管理され、
- `bdr.maximum_clock_skew_action`。
- 基礎となる pglogical サブスクリプションは、変更を破棄してはなりません。発散エラーを簡単に引き起こす可能性があります (特に CRDT タイプ)。サブスクリプションには `ignore_redundant_updates` が必要です `false` (デフォルト) に設定します。
 - `ignore_redundant_updates` のデフォルト値以外で作成された既存のグループは、次のように変更できます。

```
SELECT bdr.alter_node_group_config('group', ignore_redundant_updates := false);
```


第10章 Conflict-free Replicated Data Types

競合のない複製されたデータ型（CRDT）は、行の1つを破棄する代わりに、同時に変更された行からの値のマージをサポートします（これは従来の競合解決が行うことです）。

各 CRDT タイプは、`bdr.crdt_handlers` カタログに追加のコールバックが追加された個別の PostgreSQL データタイプとして実装されます。マージプロセスは、適用側の `pglogical` 内で発生します。追加のユーザー操作は必要ありません。

CRDT では、[CLCD](#) の章で説明されているように、テーブルで列レベルの競合解決を有効にする必要があります。

ユーザが行う唯一のアクションは、整数などの標準のビルトインではなく、`CREATE / ALTER TABLE` で特定のデータ型を使用することです。たとえば、1つの通常の整数カウンタと1つの行がある次のテーブルを考えてください。

```
CREATE TABLE non_crdt_example (
  *   id          integer          PRIMARY KEY,
  *   counter integer          NOT NULL DEFAULT 0
);

INSERT INTO non_crdt_example (id) VALUES (1);
```

2つのノードで次の SQL を同時に発行した場合：

```
UPDATE non_crdt_example
  * SET counter = counter + 1    -- "reflexive" update
WHERE id = 1;
```

…結果の値は、両方の更新が適用された後、このクエリーを使用して表示できます。

```
SELECT * FROM non_crdt_example WHERE id = 1;
  *   id | counter
-----+-----
  *   1 |       1
(1 row)
```

…`update_if_newerconflict` リゾルバにより、増分の1つが失われたことを示しています。代わりに CRDT カウンタデータタイプを使用する場合は、次のようなことを確認する必要があります。

```
CREATE TABLE crdt_example (
  *   id          integer          PRIMARY KEY,
```

(次のページに続く)

(前のページからの続き)

```

* counter bdr.crdt_gcounter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection('crdt_example',
* 'column_modify_timestamp', 'cts');

INSERT INTO crdt_example (id) VALUES (1);

```

ここでも、2つのノードで次の SQL を同時に発行し、変更が適用されるのを待ちます。

```

UPDATE crdt_example
* SET counter = counter + 1 -- "reflexive" update
WHERE id = 1;

SELECT id, counter FROM crdt_example WHERE id = 1;
* id | counter
-----+-----
* 1 | 2
(1 row)

```

これは、CRDT がアキュムレーター列を正しく許可することを示しています。そうでなければ、競合する非同期の同時更新に直面してもです。

`crdt_gcounter` 型は、上記のように $x = x + 1$ などの再帰的な UPDATE SQL でのみ機能する状態ベースの CRDT 型の例です。

`bdr.crdt_raw_value` 構成オプションは、クエリが CRDT タイプの現在の値または完全な内部状態を返すかどうかを決定します。デフォルトでは、現在の数値のみが返されます。`true` に設定すると、クエリは完全な状態の表現形式を結果ます。特別なハッシュ演算子 (#) を使用して、特別な演算演算子を使用せずに現在の数値のみを要求できます（これはデフォルトの動作です）。`bdr.crdt_raw_value = on` を使用して完全な状態がダンプされる場合、値は `bdr.crdt_raw_value = on` でのみ再取得できます。

注：`bdr.crdt_raw_value` は、クライアントに返されるデータのフォーマットのみを適用します。つまり、選択リスト内のシンプル列参照。クエリーの他の部分の列参照（たとえば、`WHERE` 句または選択リストの偶数式）でも、#演算子の使用が必要になる場合があります。

別のクラスの CRDT データ型が存在します。これは「デルタ CRDT」型と呼ばれます（後で説明するように、操作ベースの CRDT の特別なサブクラスです）。

デルタ CRDT を使用すると、値の更新は同じノードの以前の値と自動的に比較され、その後、他のすべてのノードで変更がデルタとして適用されます。

```

CREATE TABLE crdt_delta_example (
* id integer PRIMARY KEY,
* counter bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

```

(次のページに続く)

(前のページからの続き)

```
ALTER TABLE crdt_delta_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection('crdt_delta_example',
*   'column_modify_timestamp', 'cts');

INSERT INTO crdt_delta_example (id) VALUES (1);
```

2つのノードで次の SQL を同時に発行した場合：

```
UPDATE crdt_delta_example
*   SET counter = 2          -- notice NOT counter = counter + 2
WHERE id = 1;
```

結果の値は、両方の更新が適用された後、このクエリーを使用して表示できます。

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
*   id | counter
-----+-----
*   1 |      4
(1 row)
```

通常の integer 列では、結果はもちろん 2 になります。しかし、デルタ CRDT カウンタで行を更新する場合、OLDrow バージョンからスタートし、新しい行バージョンを make して、両方をリモートノードに送信し、そこで見つかったバージョンと比較します (LOCAL バージョンと呼びましょう)。標準 CRDT は NEW バージョンと LOCAL バージョンをマージしますが、デルタ CRDT は OLD バージョンと NEW バージョンを比較して、デルタを LOCAL バージョンに適用します。

CRDT タイプは bdr のパートとして bdr スキーマにインストールされます。便宜上、基本的な演算子 (+、#および!) および多くの一般的な集計関数 (min、max、sum、make、edcreate_tran_8 tweaksearch_path) を使用する必要はありません。

重要な質問は、クエリーの計画と最適化がこれらの新しいデータ型でどのように機能するかです。CRDT タイプは透過的に処理されます-ANALYZE とオプティマイザの両方が機能するため、他に何もしなくても推定とクエリープランニングが正常に機能します。

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

10.1 状態ベースおよび操作ベースの CRDT

[1] の表記に従って、操作ベースと状態ベースの両方の CRDT を実装します。

10.1.1 操作ベースの CRDT タイプ (CmCRDT)

オペレーションは明示的に転送されるのではなく、リモートノードから受信した古い行と新しい行から計算されるため、操作ベースのタイプの実装は非常に簡単です。

現在、これらの操作ベースの CRDT を実装しています。

- `crdt_delta_counter-bigint` カウンタ (増分/減分)
- `crdt_delta_sum-numeric` 総和 (増分/減分)

これらのタイプは、デルタを計算するためのビットのコードを使用して、既存のデータタイプ (`bigint` の `crdt_delta_counter` isa ドメインなど) を活用します。

このアプローチは、デルタの計算方法がわかっている型に対してのみ可能ですが、結果は非常にシンプルで安価であり (スペースと CPU の両方の点で)、いくつかの追加の利点があり構文 (たとえば、横たわっているデータタイプ)。

主な欠点は、非同期環境と同時環境ではこの値を確実にリセットできないことです。

注: カスタムデータ型を作成し、状態と最後のオペレーションを保存することにより、より複雑な操作ベースの型を実装することもできます (個々の変更をすべてデコードして転送するため、複数の操作は必要ありません)。ただし、その時点では、スペース要件を除いて、状態ベースのタイプ (リセットする機能はまだありません) に比べて利点を得ることなく、主な利点 (単純さ、既存のデータ型の再利用) を失いますノードごとの状態)。

10.1.2 状態ベースの CRDT タイプ (CvCRDT)

状態ベースの型はより複雑な内部状態を必要とするため、操作ベースの型のように通常の変数型を直接使用することはできません。

現在、4 つの状態ベースの CRDT を実装しています。

- `crdt_gcounter-bigint` カウンタ (増分のみ)
- `crdt_gsum-numeric` 総和/ カウンタ (増分のみ)
- `crdt_pncounter-bigint` カウンタ (増分/減分)
- `crdt_pnsum-numeric` 総和/ カウンタ (increments / decrements)

通常、内部状態にはノードごとの情報が含まれディスク上のサイズは増加しますが、追加の利点があります。カスタムデータ型を実装する必要性は、より多くのコード (入力/出力関数および演算子) を意味します。

利点は、値を確実にリセットする機能、失われた変更 (適切に運用されたクラスターでは発生しないはず) が存在する場合の自己回復機能、およびソースノード以外からの変更を受信する機能です。

例、ノード A で値が変更され、変更ゲットが B に複製され、C ではなく (A と C 間のネットワークの問題のため)、B が値を変更し、この変更が C に複製される場合、A からのオリジナルの変更。操作ベースの CRDT では、AC ネットワーク接続が再び機能するまでノード C は変更を受信しませんでした。

CvCRDT の主な欠点は、ディスクスペースの点でコストが高いことです-クラスターから既に削除されたノードを含む各ノードの情報がビット必要です)。状態の複雑な性質（varlena 型にシリアル化）は、CPU 使用率の増加を意味します。

10.2 ディスクスペースの要件

重要な考慮事項は、CRDT タイプ、特にディスク上のサイズに関連するオーバーヘッドです。

操作ベースのタイプの場合、タイプは他のタイプの上にあるドメインにすぎず、同じディスクスペース要件を持っているため（ノードがいくつあっても）、これはかなり簡単です。

- `crdt_delta_counter-bigint` と同じ（8 バイト）
- `crdt_delta_sum-numeric` と同じ（変数、精度に応じておおよそスケール）

操作ベースの CRDT タイプはノードごとの情報を保存しないため、ノードの数に依存しません。

状態ベースのタイプの場合、シチュエーションはより複雑です。すべてのタイプは可変長であり（本質的に `bytea` 列として格納されます）、ヘッダーと、値を「変更」した各ノードの一定量のノードごとの情報で構成されます。

`bigint` バリエーションの場合、おおよそのサイズを計算する式は次のとおりです（`N` は、この値を変更したノードの数を示します）。

- `crdt_gcounter`-32B (header) + $N * 12B$ (per-node)
- `crdt_pncounter`-48B (header) + $N * 20B$ (per-node)

`numeric` バリエーションの場合、ヘッダー部分とノードごとの部分の両方に `numeric` 可変長値が含まれるため、正確な式はありません。このような値をいくつ保持する必要があるかをお考えください。

- `crdt_gsum`
- 修正： $20B$ (header) + $N * 4B$ (per-node)
- 変数： $(2 + N)$ `numeric` 値
- `crdt_pnsum`
- 修正： $20B$ (header) + $N * 4B$ (per-node)
- 変数： $(4 + 2 * N)$ `numeric` 値
- 注*：クラスター内にいくつかのノードがあるかは関係ありません。値がマルチプルのノードで更新されることはありません。また、それは重要ではありません更新は同時（競合の原因）であったかどうか。
- 注*：また、それらのノードがすでにいくつあったかは関係ありません。クラスターから削除されました。状態を圧縮する方法はまだありません。

10.3 CRDT タイプと競合処理

テーブルには CRDT 列と非 CRDT 列の両方が含まれる可能性があるため（実際、ほとんどの列は非 CRDT であると予想されます）、通常の競合解決と CRDT マージの両方を行う必要があります。

競合解決が最初に発生し、保持するタプル（適用するタプル）と破棄するタプルを決定する責任があります。次にマージフェーズが発生し、CRDT 列のデータが破棄されたタプルから適用タプルにマージされます。

- 注*：これにより、CRDT タイプはプレーンに比べてやや高価になります競合の解決。マージは毎回発生ニーズがあるため、競合解決が高速パスのいずれかを使用できる場合（変更現在のトランザクションなどで）。

10.4 CRDT タイプと競合レポート

デフォルトでは、検出された競合はサーバーログに書き込まれます。CRDT タイプがなければ、これは完全に理にかなっていません。なぜなら、競合の解決は本質的に利用可能な情報の半分を破棄するからです（設定によってはローカルまたはリモート行）。これはデータロスを示します。

CRDT タイプを使用すると、何も捨てずに情報の両方の部分を組み合わせて、データロスの問題を解消できます。これにより、競合レポートが不要になります。

このため、CRDT マージによって競合を完全に解決できる場合、つまり各列が次の 2 つの条件の少なくとも 1 つを満たす場合、競合レポートをスキップします。

- 1) ローカルとリモートのタプルの値が同じ（NULL または等しい）
- 2) CRDT データタイプを使用します（したがって、マージできます）

- 注*：これは、競合がない場合にも競合レポートをスキップすることを意味します CRDT 列。ただし、ローカル/リモートタプルのすべての値は等しい。

10.5 CRDT 値のリセット

CRDT 値のリセットは可能ですが、特別なハンドリングが必要です。クラスターの非同期性は、異なるノードが変更ストリームの異なる場所でリセットオペレーションを見ることができることを意味します（実装方法は関係ありません）。異なるノードも同時にリセットを開始できます。すなわち、他のノードからのリセットを観察する前に。

言い換えれば make、リセットオペレーションが正しく動作するためには、通常の操作に対して可換であるニーズがあります。値をリセットするための多くの単純な方法（単一ノードで完全に機能する場合があります）は、まさにこの理由で失敗します。

例、値をリセットする最も簡単な方法は次のとおりです。

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

状態ベースの CRDT では、これは機能しません。他のノードの状態は破棄しますが、ローカルのみです。リモートノードのマージ機能により追加され、値が発散し、最終的に他のノードの変更によりそれを受け取ります。

操作ベースの CRDT では、更新が `-cnt` の減算として解釈されるため、これは機能しているように見える場合があります。ただし、同時リセットがない場合にのみ機能します。2つのノードが同時にリセットを実行しようとする、デルタを2回適用し、負の値を取得します（リセットから予想したものではありません）。

`DELETE + INSERT` をリセットとして使用することもできますが、これにはいくつかの弱点もあります。行が同じキーで再挿入された場合、すべてのノードが（他のノードからの変更に関して）操作のストリームの同じ位置に表示されることは保証されません。BDR は、同じプライマリキー値を再利用することを特に推奨します。これは、同時ケースではデータの異常を引き起こす可能性があるためです。

状態ベースの CRDT タイプは、次のような特別な `!` 演算子を使用して、確実にリセットをハンドルできます。

```
UPDATE tab SET counter = !counter WHERE ...;
```

「信頼できる」とは、値に上記の2つの問題（マルチプルの同時リセットと発散）がないことを意味します。

操作ベースの CRDT タイプは、[Eager Replication](#) を使用してのみ確実にリセットできます。これにより、マルチプルの同時リセットが回避されるためです。

10.6 実装された CRDT データ型

現在、6つの CRDT データ型が実装されています-成長のみのカウンタと総和、正負のカウンタと総和、およびデルタカウンタと総和。「総和」タイプは10進数ベース（`numeric`）です。

[1] で説明されている追加の CRDT タイプは、後で実装される可能性があります。

現在実装されている CRDT データ型は、次のクエリーでリストできます。

```
SELECT n.nspname, t.typname
FROM bdr.crdt_handlers c
JOIN (pg_type t JOIN pg_namespace n ON t.typnamespace = n.oid)
* ON t.oid = c.crdt_type_id;
```

10.6.1 成長専用カウンタ（`crdt_gcounter`）

- 負でない値の増分のみをサポート（`value + int`
 - および `counter + bigint` 演算子）
- カウンタの現在の値は、`#` 演算子を使用して取得できます。または `bigint` にキャストして
- `counter = value` のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション）
- `!` 演算子を使用して、カウンタをシンプルにリセットできます。（`counter = !counter`）

- `crdt_gcounter_to_text` を使用して内部状態を検査できます

```
CREATE TABLE crdt_test (  
  *   id          INT PRIMARY KEY,  
  *   cnt          bdr.crdt_gcounter NOT NULL DEFAULT 0  
);  
  
INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0  
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824  
INSERT INTO crdt_test VALUES (3, -4531);  -- error: negative value  
  
-- enable CLCD on the table  
ALTER TABLE crdt_test REPLICA IDENTITY FULL;  
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts  
→');  
  
-- increment counters  
UPDATE crdt_test SET cnt = cnt + 1 WHERE id = 1;  
UPDATE crdt_test SET cnt = cnt + 120 WHERE id = 2;  
  
-- error: minus operator not defined  
UPDATE crdt_test SET cnt = cnt - 1 WHERE id = 1;  
  
-- error: increment has to be non-negative  
UPDATE crdt_test SET cnt = cnt + (-1) WHERE id = 1;  
  
-- reset counter  
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;  
  
-- get current counter value  
SELECT id, cnt::bigint, cnt FROM crdt_test;  
  
-- show internal structure of counters  
SELECT id, bdr.crdt_gcounter_to_text(cnt) FROM crdt_test;
```

10.6.2 成長のみの総和 (`crdt_gsum`)

- 負でない値の増分のみをサポートします (`sum + numeric`)
- 総和の現在の値は、`#`演算子を使用して取得できます。または `numeric` にキャストして
- `sum = value` のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション）
- `!演算子` (`sum = !sum`) を使用して、総和をシンプルにリセットできます。
- `crdt_gsum_to_text` を使用して内部状態を検査できます

```

CREATE TABLE crdt_test (
  *   id          INT PRIMARY KEY,
  *   gsum        bdr.crdt_gsum NOT NULL DEFAULT 0.0
);

INSERT INTO crdt_test VALUES (1, 0.0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- error: negative value

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts
→');

-- increment sum
UPDATE crdt_test SET gsum = gsum + 11.5 WHERE id = 1;
UPDATE crdt_test SET gsum = gsum + 120.33 WHERE id = 2;

-- error: minus operator not defined
UPDATE crdt_test SET gsum = gsum - 15.2 WHERE id = 1;

-- error: increment has to be non-negative
UPDATE crdt_test SET gsum = gsum + (-1.56) WHERE id = 1;

-- reset sum
UPDATE crdt_test SET gsum = !gsum WHERE id = 1;

-- get current sum value
SELECT id, gsum::numeric, gsum FROM crdt_test;

-- show internal structure of sums
SELECT id, bdr.crdt_gsum_to_text(gsum) FROM crdt_test;

```

10.6.3 正負カウンタ (crdt_pncounter)

- 正と負の両方の値を持つ増分をサポート（を通じて
 - `counter + int` および `counter + bigint` 演算子）
- カウンタの現在の値は、`#`演算子を使用して取得できます。または `bigint` にキャストすることにより
- `counter = value` のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション）
- `!`演算子を使用して、カウンタをシンプルにリセットできます。（`counter = !counter`）
- `crdt_pncounter_to_text` を使用して内部状態を検査できます

```

CREATE TABLE crdt_test (
    *   id          INT PRIMARY KEY,
    *   cnt          bdr.crdt_pncounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts
→');

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET cnt = cnt - 73     WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283  WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12)  WHERE id = 3;

-- get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

-- show internal structure of counters
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;

-- reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

-- get current counter value after the reset
SELECT id, cnt::bigint, cnt FROM crdt_test;

```

10.6.4 正負の総和 (crdt_pnsum)

- 正と負の両方の値を持つ増分をサポート（を通じて
 - `sum + numeric`）
- 総和の現在の値は、`#`演算子を使用して取得できます。または `numeric` にキャストすることにより
- `sum = value` のようシンプル割り当てと互換性がない（これは、新しい値がどこかで計算されるとき一般的なパターンですアプリケーション）

- !演算子 (sum = !sum) を使用して、総和をシンプルにリセットできます。
- crdt_pnsum_to_text を使用して内部状態を検査できます

```
CREATE TABLE crdt_test (
  *   id          INT PRIMARY KEY,
  *   pnsum       bdr.crdt_pnsum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24);    -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);     -- initialized to -45.31

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts
→');

-- increment sums
UPDATE crdt_test SET pnsum = pnsum + 1.44      WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum + 12.20     WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum + (-24.34)  WHERE id = 3;

-- decrement sums
UPDATE crdt_test SET pnsum = pnsum - 7.3       WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum - 192.83    WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum - (-12.22)  WHERE id = 3;

-- get current sum value
SELECT id, pnsum::numeric, pnsum FROM crdt_test;

-- show internal structure of sum
SELECT id, bdr.crdt_pnsum_to_text(pnsum) FROM crdt_test;

-- reset sum
UPDATE crdt_test SET pnsum = !pnsum WHERE id = 1;

-- get current sum value after the reset
SELECT id, pnsum::numeric, pnsum FROM crdt_test;
```

10.6.5 デルタカウンタ (crdt_delta_counter)

- bigint ドメインとして定義されているため、bigint 列とまったく同じように機能します
- 正と負の両方の値を持つ増分をサポート
- counter = value のようシンプル割り当てと互換性があります（新しい値がアプリケーションのどこかで計算される場合に一般的）
- 値をリセットするシンプル方法はありません（確実）

```
CREATE TABLE crdt_test (
  *   id          INT PRIMARY KEY,
  *   cnt          bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824);     -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);      -- initialized to -4531

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts
→');

-- increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET cnt = cnt - 73     WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283  WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12)  WHERE id = 3;

-- get current counter value
SELECT id, cnt FROM crdt_test;
```

10.6.6 総和 (crdt_delta_sum)

- numeric ドメインとして定義されているため、numeric 列とまったく同じように機能します
- 正と負の両方の値を持つ増分をサポート
- sum = value のようシンプル割り当てと互換性があります（新しい値がアプリケーションのどこかで計算される場合に一般的）
- 値をリセットするシンプル方法はありません（確実）

```

CREATE TABLE crdt_test (
  *   id          INT PRIMARY KEY,
  *   dsum        bdr.crdt_delta_sum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);           -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129.824);    -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4.531);    -- initialized to -4531

-- enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection('crdt_test', 'column_modify_timestamp', 'cts
→');

-- increment counters
UPDATE crdt_test SET dsum = dsum + 1.32 WHERE id = 1;
UPDATE crdt_test SET dsum = dsum + 12.01 WHERE id = 2;
UPDATE crdt_test SET dsum = dsum + (-2.4) WHERE id = 3;

-- decrement counters
UPDATE crdt_test SET dsum = dsum - 7.33 WHERE id = 1;
UPDATE crdt_test SET dsum = dsum - 19.83 WHERE id = 2;
UPDATE crdt_test SET dsum = dsum - (-1.2) WHERE id = 3;

-- get current counter value
SELECT id, cnt FROM crdt_test;

```

[1] < https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type >

第11章 Durability & Performance Options

Durability & Performance Options

第12章 Eager Replication

Eager Replication

第13章 Commit At Most Once

“Commit At Most Once”

第14章 Database Scaling

Database Scaling

第15章 Timestamp-based Snapshots

Timestamp-based Snapshots

第16章 Replication Sets

Replication Sets

第17章 Stream Triggers

Stream Triggers

第18章 Backup and Recovery

Backup and Recovery

第19章 Upgrades

Upgrades

第20章 Explicit Two-Phase Commit (2PC)

Explicit Two-Phase Commit (2PC)

第21章 Catalogs and Views

Catalogs and Views

第22章 Monitoring

レプリケーションセットアップの監視は、システムが最適に実行され、ディスクスペースが不足したり、操作を停止する可能性のある他の障害が発生したりしないように保証に重要です。

例、レプリケーションスロットが大幅に遅れスタートした場合に管理者に警告が発せられ、事前に対策を講じることが保証に、自動モニタリングを実施することが重要です。

EDB は、バージョン 8.1 の BDR をサポートする Postgres Enterprise Manager (PEM) を提供します。また、ツールまたはユーザーは、以下で説明する機能を使用して BDR に独自の呼び出しを make ことができます。

22.1 監視の概要

BDR グループは、多くの場合ノードと呼ばれるマルチプルのサーバーで構成されます。グループ全体の健全性を保証には、すべてのノードを監視する必要があります。

bdr_monitor ロールは、`bdr.monitor` 関数を実行して、3つのレベルのいずれかを使用して BDR の健全性の評価を提供します。

- OK-しばしば緑と表示される
- WARNING-しばしば黄色として表示される
- CRITICAL-しばしば赤と表示される
- UNKNOWN と同様に-認識されない状況の場合、多くの場合赤と表示されます

BDR は、さまざまな内部メトリックの瞬間的な状態を示す動的カタログビュー、およびユーザが要求した構成のデフォルトや構成の変更を保存する BDR メタデータカタログも提供します。これらのビューとテーブルには `bdr_monitor` または `bdr_read_all_stats` でアクセスできるものもありますが、セキュリティ要件が高いユーザーまたは内部情報を含むものもあります。

BDR を使用すると、各ノードを個別にモニタしたり、単一のノードにアクセスしてグループ全体をモニタしたりできます。各ノードを個別にモニタ場合は、各ノードに接続してモニタリング要求を発行するだけです。単一のノードからグループを監視する場合は、`bdr.group` で始まるビューを使用します。これらの要求は他のノードを呼び出してグループレベルの情報セットを組み立てるためです。

`bddr_superuser` によって `bdr.run_on_all_nodes()` ファンクションへのアクセスが許可されている場合、すべてのノードに対して独自の呼び出しを行うことが make ます。

22.2 ノードの参加と削除の監視

デフォルトでは、ノード管理機能は、結合またはパート操作が完了するまで待機します。これは、それぞれの `wait_for_completion` ファンクションの引数を使用してオフにすることができます。待機がオフになっている場合、結合またはパートオペレーションがいつ終了するかを確認するには、`bdr.node_summary` および `bdr.state_journal_details` を介して間接的にノードの状態を確認します。

呼び出されると、ヘルパファンクション `bdr.wait_for_join_completion()` は、すべての未処理のノード結合操作が完了するまで、PostgreSQL セッションを一時停止させます。

次に、2つのノードがアクティブで別のノードが参加していることを示す `bdr.node_summary` からの SELECT クエリーの出力例を示します。

```
# SELECT node_name, interface_connstr, peer_state_name,
#       node_seq_id, node_local_dbname
# FROM bdr.node_summary;
- [ RECORD 1 ]-----+-----
node_name      | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name | ACTIVE
node_seq_id     | 1
node_local_dbname | postgres
- [ RECORD 2 ]-----+-----
node_name      | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name | ACTIVE
node_seq_id     | 2
node_local_dbname | postgres
- [ RECORD 3 ]-----+-----
node_name      | node3
interface_connstr | host=localhost dbname=postgres port=7434
peer_state_name | JOINING
node_seq_id     | 3
node_local_dbname | postgres
```

また、テーブル `bdr.node_catchup_info <catalogs>`__` は、キャッチアップ状態に関する情報を提供します。これは、ノードの結合またはノードの分離に関連する可能性があります。

ノードが分離されると、クラスター内の一部のノードがその分離ノードからすべてのデータを受信しなかった可能性があります。そのため、すでにそのデータを受信して転送できるノードから一時スロットを作成します。

`catchup_state` は次のいずれかです。

```
10 = setup
20 = start
30 = catchup
40 = done
```

22.3 レプリケーションピアの監視

レプリケーションアクティビティのモニタリングに使用される 2 つの主なビューがあります。

- 発信レプリケーションをモニタリングするための ``bdr.node_slots <catalogs>`__`
- 着信レプリケーションをモニタリングするための ``bdr.subscription_summary <catalogs>`__`

`bdr.node_slots` によって提供される情報のほとんどは、標準の PostgreSQL レプリケーションモニタリングビューへのクエリによっても取得できます。

各ノードには BDR グループスロットが 1 つありますが、これには接続しないでください。また、アクティブとしてマークされることはほとんどありません。これは正常であり、何かがダウンしたり切断されたりすることを意味するものではありません。``Replication Slots created by BDR <nodes>`__` を参照してください。

22.3.1 送信レプリケーションの監視

発信レプリケーションアクティビティのモニタリングに使用される追加のビューがあります。

- 発信レプリケーションをモニタリングするための ``bdr.node_replication_rates <catalogs>`__`

`bdr.node_replication_rates` ビューは、特に、発信レプリケーションアクティビティの全体像と、同等なキャッチアップ推定値を提供します。

```
# SELECT * FROM bdr.node_replication_rates;
- [ RECORD 1 ]-----+-----
peer_node_id      | 112898766
target_name       | node1
sent_lsn          | 0/28AF99C8
replay_lsn        | 0/28AF99C8
replay_lag        | 00:00:00
replay_lag_bytes  | 0
replay_lag_size   | 0 bytes
apply_rate        | 822
catchup_interval  | 00:00:00
- [ RECORD 2 ]-----+-----
peer_node_id      | 312494765
target_name       | node3
sent_lsn          | 0/28AF99C8
replay_lsn        | 0/28AF99C8
replay_lag        | 00:00:00
replay_lag_bytes  | 0
replay_lag_size   | 0 bytes
apply_rate        | 853
catchup_interval  | 00:00:00
```

上記の `apply_rate` は、1 秒あたりのバイト数のレートを示します。それは同等ながローカルノードからデータを消費している `rateat` です。ノードがクラスターに再接続するときの `replay_lag` はすぐにゼロに

設定されます。この情報の修正に取り組んでいます。回避策として、同等ながローカルノードデータに追いつくのに必要な時間を参照する `catchup_intervalcolumn` を使用することをお勧めしノード。以下に説明するように、他のフィールドも `bdr.node_slotsview` を介して利用できます。

!!! Note * このカタログは、bdr-enterprise 拡張機能がインストールされている場合にのみ存在します。

管理者は、ローカルノードからの発信レプリケーションについて `bdr.node_slots` をクエリーできます。現在のノードで認識されているグループ内の他のすべてのノードのレプリケーションステータスに関する情報と、現在のノードで BDR によって作成された追加の複製スロットが表示されます。

```
# SELECT node_group_name, target_dbname, target_name, slot_name, active_pid,
#        catalog_xmin, client_addr, sent_lsn, replay_lsn, replay_lag,
#        replay_lag_bytes, replay_lag_size
# FROM bdr.node_slots;
- [ RECORD 1 ]-----+-----
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node3
slot_name       | bdr_postgres_bdrgroup_node3
active_pid      | 15089
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 120
replay_lag_size | 120 bytes
- [ RECORD 2 ]-----+-----
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node2
slot_name       | bdr_postgres_bdrgroup_node2
active_pid      | 15031
catalog_xmin    | 691
client_addr     | 127.0.0.1
sent_lsn        | 0/23F7B70
replay_lsn      | 0/23F7B70
replay_lag      | [NULL]
replay_lag_bytes| 84211
replay_lag_size | 82 kB
```

BDR はメッシュネットワークであるため、クラスターの遅延を完全にビューするには、参加しているすべてのノードでこのクエリーを実行する必要があります。

`replay_lag_bytes` は、ローカルサーバーの現在の WAL 書き込み位置と、同等なが最後に確認した位置である `replay_lsn` との WAL 位置の違いを報告しノード。`replay_lag_size` は、人間が読める形式の単なるものです。WAL には通常、複製されないが、`VACUUM` アクティビティ、インデックスの変更、同じノード上の他のデータベースに関連付けられた書き込み、レプリケーションセットのパートではないテーブル

の書き込みなど、`replay_lag_bytes` でカウントされる書き込みが多数含まれることを理解することが重要です。ここでも報告されるバイトの遅れは、同等な最新のノードにするためにワイヤ上で複製する必要があるデータの量ではなく、処理する必要があるサーバサイドの WAL の量のみです。

同様に、ノードは、同等ながキャッチアップするのにかかる時間、または `bdr.node_slots` が照会されたときに現在の位置から書き込み位置にリプレイするのにかかる時間の測定値ではありません。同等なが最新のコミットと現在の壁時計時間を確認したときの遅延を測定します。`bdr.node_replication_rates` の `replay_lag_bytes` および `replay_lag_size` または `catchup_interval` をモニタすることをお勧めします。これは、ノードが再接続した直後にこの列がゼロに設定されるためです。

ロジカルレプリケーションがトランザクションをストリーミングしている間、バイトと時間の両方の遅れは進みません。コミットがレプリケートされるときにのみ変更されます。そのため、遅延は「鋸歯状」になる傾向があり、トランザクションがストリーミングされると上昇し、同等ながコミットしてフラッシュし、確認をノードすると再び下降します。報告された LSN 位置は、同様の理由でスムーズに進むのではなく「階段状」になります。

レプリケーションが切断されると (`active = 'f'`)、`active_pid` 列は NULL になり、`client_addr` およびアクティブ make 接続でのみ意味のある他のフィールドになります。`state` フィールドは `'disconnected'` になります。`_lsn` フィールドは `confirmed_flush_lsn` と同じになります。これは、クライアントが最後に再生および保存されたことがわかっている最後の位置であるためです。`_lag_size` と `_lag_bytes` フィールドは、`confirmed_flush_lsn` とローカルサーバの現在の WAL 挿入位置との間の距離をレポートします。

注：`restart_lsn` が他のノード列の背後にあるのは正常です。これは、レプリケーションまたは同等なの遅延の問題を示すものではありません。`restart_lsn` は、PostgreSQL の内部ロジカルデコーディングが中断された場合に WAL を読み取る必要がある位置であり、通常、まだ複製およびフラッシュされていない最も古いトランザクションの位置を反映します。非常に古い make は、切断後のレプリケーションのリスタートを遅くし、望ましい以上の WAL を保持することができますが、そうでなければ無害になります。

22.3.2 着信レプリケーションの監視

受信レプリケーション（サブスクリプションとも呼ばれます）は、`bdr.subscription_summary` ビューをクエリすることで監視できます。これにより、BDR クラスタ内の他のノードへの既知のサブスクリプションのリストとレプリケーションワーカーの状態が表示されます。例：

```
# SELECT node_group_name, origin_name, sub_enabled, sub_slot_name,
#        subscription_status
# FROM bdr.subscription_summary;
- [ RECORD 1 ]-----+-----
node_group_name      | bdrgroup
origin_name          | node2
sub_enabled          | t
sub_slot_name        | bdr_postgres_bdrgroup_node1
subscription_status  | replicating
- [ RECORD 2 ]-----+-----
node_group_name      | bdrgroup
origin_name          | node3
sub_enabled          | t
```

(次のページに続く)

(前のページからの続き)

sub_slot_name	bdr_postgres_bdrgroup_node1
subscription_status	replicating

22.3.3 LCR を使用して WAL 送信者を監視する

が有効になっている場合、各 WAL 送信者の現在の LCR () ファイルに関する情報は、ファンクションを介して監視できます。例：

```
postgres=# SELECT * FROM bdr.wal_sender_stats();
   * pid   | is_using_lcr | decoder_slot_name | lcr_file_
   name
-----+-----+-----+-----
   2059904 | f            |                   | 
   2059909 | t            | bdr_postgres_bdrgroup_decoder | 
   000000000000000000000000000000001400000000000000000
   2059916 | t            | bdr_postgres_bdrgroup_decoder | 
   000000000000000000000000000000001400000000000000000
(3 rows)
```

is_using_lcr が FALSE の場合、decoder_slot_name / lcr_file_name は NULL になります。これは、デコードワーカーが有効になっていない場合、または WAL 送信者が[ロジカルスタンバイ] (nodes.md # Logical Standby Nodes) を提供している場合です。

22.4 BDR レプリケーションワーカーの監視

すべての BDR ワーカーは、bdr.stat_activity と同じ列と情報コンテンツを持つシステムビュー bdr.stat_activity に表示されます。したがって、このビューは BDR システムの状態に関するこれらの洞察を提供します。

- 次の場合、wait_event 列には拡張情報があります。待機理由は BDR に関連しています。
- query 列は、BDR ワーカーでは空白になります。ただし、ライタプロセスが DDL を実行しているとき

bdr.workers ビューには、BDR ワーカー固有の詳細が表示されますが、bdr.stat_activity からは利用できません。

ビュー bdr.worker_errors は、作業の継続に問題があるワーカーによって報告されたエラー（ある場合）を示します。このビューではアクティブなエラーのみが可視されるため、ワーカーに一時的な問題があったが回復した場合、ビューは空になります。

22.5 グローバルロックの監視

現在 DDL レプリケーションにのみ使用されているグローバルロックは、BDR グループ全体に存在する重量ロックです。

現在、グローバルロックには2つのタイプがあります。

DDL ロック、永続的なすべての DDL 操作のシリアル化に使用データベース内の（一時的ではない）オブジェクト（テーブル）DDL 中にリレーションへの書き込みをロックアウトするために使用される DML リレーションロックリレーション定義を変更する操作

DDL オペレーションのタイプと `bdr.ddl_locking` 設定の値に応じて、同じトランザクションに対してどちらかまたは両方のエンタリタイプを作成できます。

ローカルノードで保持されているグローバルロックは、に可視されます。このビューには、ロックのタイプが表示されます。関係ロックは、どのリレーションがロックされているか、ロックを保持している PID（ローカルの場合）、およびロックがグローバルに許可されているかどうかを示します。グローバルアドバイザリロックの場合、列はロックが取得されるアドバイザリキーを示しキー。

以下は、`bdr.ddl_locking = on` で `anALTER TABLE` ステートメントを実行しているときの `bdr.global_locks` の出力例です。

```
# SELECT lock_type, relation, pid FROM bdr.global_locks;
- [ RECORD 1 ]-----
lock_type | GLOBAL_LOCK_DDL
relation  | [NULL]
pid       | 15534
- [ RECORD 2 ]-----
lock_type | GLOBAL_LOCK_DML
relation  | someschema.sometable
pid       | 15534
```

ロックタイミング情報を含むすべてのフィールドの詳細については、カタログの文書を参照してください。

22.6 競合の監視

レプリケーション `conflicts` は、マルチプルのノードが相互に対話できる方法で同じ行に影響を与える変更を行う場合に発生する可能性があります BDR システムを保証して、競合を特定し、可能な場合はアプリケーションの変更を行って、競合を排除または頻度を減らします。

デフォルトでは、すべての競合は `bdr.conflict_history` に記録されます。これには競合するデータの完全な詳細が含まれるため、行は行レベルのセキュリティによって保護され、レプリケートされたテーブルの所有者のみが可視保証にします。所有者は競合を予想し、それらを分析して、もしあれば、解決すべき問題と見なされる可能性があるものを確認する必要があります。

モニタリングのために、ユーザデータを含まない `bdr.conflict_history_summary` を使用します。効率的なクエリプランを使用して、現在の日に発生した競合の数をカウントするクエリーの例は次のとおりです。

```
SELECT count(*)
FROM bdr.conflict_history_summary
WHERE local_time > date_trunc('day', current_timestamp)
      * AND local_time < date_trunc('day', current_timestamp + '1 day');
```

22.7 外部モニタリング

ユーザーが提供したメタデータを保存して、モニタリングツールが BDR クラスターを理解およびモニタできるようにすることができます。この情報を集中化することにより、externaltools は任意の単一ノードにアクセスし、特定の接続のネットワークコストやワーニング/アラームしきい値など、クラスター全体に関する詳細を読み取ることができます。

bdr_superuser にはこれらの関数とテーブルに対する特権があります。ビュー bdr.network_monitoring には、bdr_read_all_stats ロールからもアクセスできます。

22.7.1 bdr.set_node_location

このファンクションは、ノードメタデータを bdr.node_location に挿入します

あらすじ

```
bdr.set_node_location(
    node_group_name text,
    node_name text,
    node_region text,
    node_location text);
```

パラメーター

- node_group_name- BDR グループの名前
- node_name- ノードの名前
- node_region- データセンターサイトまたはリージョン
- node_location- サーバー名前、可用性ゾーンなど。

22.7.2 bdr.set_network_path_info

このファンクションは、ノード間のネットワーク経路のネットワークパスメタデータをテーブル `bdr.network_path_info` に挿入します。

あらすじ

```
bdr.set_network_path_info(  
    node_group_name text,  
    region1 text,  
    region2 text,  
    location1 text,  
    location2 text,  
    network_cost numeric,  
    warning_threshold numeric,  
    alarm_threshold numeric)
```

パラメーター

- `node_group_name`- BDR グループの名前
- `region1`-オリジンサーバー名前
- `region2`-リモートサーバ名前
- `location1`-オリジンのデータセンター名前
- `location2`-リモートデータセンター名前
- `network_cost`-ネットワーク転送のコストを表すアブストラクト値
- `warning_threshold`-しきい値を超えると遅延が発生する
- `alarm_threshold`-アラームが発生するまでの遅延

22.7.3 bdr.network_monitoring ビュー

このビューは、ノード間のネットワークパスに関する情報を収集します。

ロギングの構成は、`bdr.alter_node_set_log_config` 関数によって定義されます。

22.8 統計を適用する

BDR は、各サブスクリプションと各テーブルの両方に適用されるレプリケーションに関する統計を収集します。

2つのモニタリングビューがあります。サブスクリプション統計用の `bdr.stat_subscription` と、リレーション統計用の `bdr.stat_relation` です。これらのビューは両方とも提供します：

- 複製された INSERT / UPDATE / DELETE / TRUNCATE の数
- ブロックアクセスとキャッシュヒット率
- 読み取り/書き込みの合計 I/O 時間

リレーションのみの場合、これらの統計：

- リレーションのレプリケーションの処理にかかった合計時間
- リレーションのロック（ある場合）を取得するための合計ロック待機時間（のみ）

サブスクリプションの場合のみ、これらの統計：

- サブスクリプションに対して複製された COMMIT / DDL の数
- このサブスクリプションがアップストリームに接続した回数

これらの統計の追跡は、`pglogical GUC` `pglogical.track_subscription_apply` と `pglogical.track_relation_apply` によってそれぞれ制御されます。詳細については、[BDR] を参照してください。

これらの出力例は次のようになります。

```
# SELECT sub_name, nconnect, ninsert, ncommit, nupdate, ndelete, ntruncate, nddl
FROM pglogical.stat_subscription;
- [ RECORD 1 ]-----
sub_name | bdr_regression_bdrgroup_node1_node2
nconnect | 3
ninsert  | 10
ncommit  | 5
nupdate  | 0
ndelete  | 0
ntruncate | 0
nddl     | 2
```

この場合、サブスクリプションはアップストリームに3回接続し、10行を挿入し、5つのトランザクション内で2つのDDLコマンドを実行しました。

これらのビューの統計カウンターは、関数 `bdr.reset_subscription_stats` および `bdr.reset_relation_stats` を使用してゼロにリセットできます。

22.9 標準の PostgreSQL 統計 Views

テーブルおよびインデックスの使用に関する統計は、通常、downstreammaster によって更新されます。これは、autovacuum の正しいファンクションに不可欠です。ダウンストリームマスタにローカル書き込みがなく、統計がリセットされていない場合、これらの2つのビューはアップストリームとダウンストリームの対応する結果を表示する必要があります。

- pg_stat_user_tables
- pg_statio_user_tables

!!! Note * アップストリームテーブルの統計がダウンストリーム統計と同様であるとは必ずしも期待していません。同じ量だけ変化することを期待しています。統計が 1M の挿入と 1M の更新を示すテーブルの例を考えてください。新しいノードが BDR グループに参加すると、新しいノードの同じテーブルの統計には 1M の挿入とゼロの更新が表示されます。ただし、その瞬間から、アップストリームとダウンストリームのテーブル統計は同じ量だけ変化します。一方の側のすべての変更が他方の側に複製されるためです。

インデックスは変更の適用に使用されるため、下流側の識別インデックスは、非識別インデックスよりも b_tran_1s および DELETEs を実行したワークロードでより頻繁に使用されるように見える場合があります。

ビルトインのインデックスモニタリングビューは次のとおりです。

- pg_stat_user_indexes
- pg_statio_user_indexes

これらのビューはすべて、PostgreSQL documentation on the statistics views で詳細に説明されています。

22.10 BDR バージョンの監視

BDR では、同じクラスター内のノード間で異なる Postgres バージョンと異なる BDR バージョンを実行できます。これはアップグレードに役立ちます。

ビュー bdr.group_versions_details は functionbdr.run_on_all_nodes() を使用して、すべてのノードから BDR バージョン、エディション、および pglogical バージョンを同時に取得します。例：

```
bdrdb=# SELECT node_name, postgres_version, pglogical_version,
bdr_version FROM bdr.group_versions_details;
```

node_name	postgres_version	pglogical_version	bdr_version
node1	13.3	3.7.10	3.7.10
node2	13.3	3.7.10	3.7.10

推奨されるセットアップは、すべてのノードでできるだけ早く同じ最新バージョンを実行することです。クラスターが異なる BDR バージョンを長時間実行しないことをお勧めします。BDR と pglogical バージョンは同じノードで問題なく異なる場合があります。

モニタリングのために、次のアラートレベルをお勧めします。

- status = UNKNOWN、メッセージ=このノードは BDR グループのパートではありません

- status = OK、メッセージ=すべてのノードが同じ pglogical および BDR バージョンを実行しています
- status = WARNING、メッセージ=アクセスできないノードが少なくとも 1 つあります
- status = WARNING、メッセージ=異なる BDR バージョンを実行しているノードがあります他のノードと比較した場合
- status = WARNING、メッセージ=異なる BDR エディションを実行しているノードがあります他のノードと比較した場合

説明されている動作は、`bdr.group_version_details` ビューから返された BDR / pglogical バージョン情報を使用してクラスタ全体のバージョンチェックを提供する `functionbdr.monitor_group_versions()` に実装されています。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status | message
-----+-----
 OK      | All nodes are running same pglogical and BDR versions
```

22.11 ラフトコンセンサスの監視

Raft Consensus は常にクラスタ全体で機能する必要があります。Raft Consensus が機能しない状態で BDR クラスタを実行した場合の影響は次のとおりです。

BDR データ変更のレプリケーションはまだ正常に機能している可能性があります- グローバル DDL/ DML ロックは機能しません - Galloc シーケンスは最終的にチャンクを使い果たします - Eager Replication は機能しません - クラスタメンテナンス操作（結合ノード、パートノード、スタンバイのプロモート）まだ許可されていますが、終了しない可能性があります（単にハングします）BDR ノード間でノードステータスが正しく同期されない場合があります BDR グループレプリケーションスロットは LSN を進めないため、WAL ファイルを保持しますディスク

ビュー `bdr.group_raft_details` は、関数 `bdr.run_on_all_nodes()` と `bdr.get_raft_status()` を使用して、すべてのノードから RaftConsensus ステータスを同時に取得します。例：

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
 * node_id | node_name | state | leader_id
-----+-----+-----+-----
 1148549230 | node1 | RAFT_LEADER | 1148549230
 3367056606 | node2 | RAFT_FOLLOWER | 1148549230
```

以下のすべての条件が満たされている場合、ラフトコンセンサスは正しく機能していると言えます。

- すべてに対して有効な状態（RAFT_LEADER または RAFT_FOLLOWER）が定義されていますノード
- ノードの 1 つだけが RAFT_LEADER です
- leader_id はすべての行で同じであり、node_id とマッチする必要があります state = RAFT_LEADER がある行の

Raft Consensus は、随時、新しい選挙をスタートして、新しい RAFT_LEADER を定義します。選挙中に、RAFT_LEADER が存在せず、一部のノードが自分自身を RAFT_CANDIDATE と見なす仲介者がいる場合があります。選挙全体が `bdr.raft_election_timeout` より長くかかることはありません（デフォルトでは 6 秒に設定されています）。上記のクエリで選挙中のシチュエーションが返された場合は、`edbdr.raft_election_timeout` を待ってから再度クエリを実行します。`afterbdr.raft_election_timeout` が過ぎても上記の条件のいくつかはまだ満たされていない場合、Raft Consensus は機能していません。

Raft Consensus は、単一のノードでのみ正常に動作しない場合があります。例、ノードの 1 つが現在のリーダーを認識せず、自分自身を RAFT_CANDIDATE と見なします。この場合、次のことを確認することが重要です。

- すべての BDR ノードは、通常およびレプリケーション接続（ファイル `pg_hba.conf` を確認）BDR および pglogical バージョンはすべてのノードで同じです
- `bdr.raft_election_timeout` はすべてのノードで同じです

場合によっては、特にノードが地理的に互いに離れていたり、ネットワークのレイテンシが長い場合は、`bdr.raft_election_timeout`（6 秒）のデフォルト値では不十分な場合があります。すべてが正しいことを確認しても RaftConsensus がまだ機能しない場合は、すべてのノードで `bdr.raft_election_timeout` をたとえば 30 秒に増やすことを検討してください。BDR 3.6.11 以降、`settingbdr.raft_election_timeout` の設定に必要なのはサーバーのリロードのみです。

Raft Consensus がクラスターの運用タスクにどのように影響し、asRaft Consensus がグループスロットを進める責任があるかを考えると、モニタリングアラートレベルを次のように定義できます。

- `status = UNKNOWN`、メッセージ=このノードは BDR グループのパートではありません
- `status = OK`、メッセージ= Raft Consensus は正常に機能しています
- `status = WARNING`、メッセージ=アクセスできないノードが少なくとも 1 つあります
- `status = WARNING`、メッセージ= RAFT_CANDIDATE としてノードがあり、選挙が進行中の可能性があります
- `status = WARNING`、メッセージ= RAFT_LEADER はありません。選挙は進行中
- `status = CRITICAL`、メッセージ= Raft Consensus に単一のノードがあります
- `status = CRITICAL`、メッセージ= RAFT_CANDIDATE としてノードがありますが、RAFT_LEADER が定義されています
- `status = CRITICAL`、メッセージ=異なるリーダーに続くノードがあります RAFT_LEADER としてノードセットよりも

説明した動作は、`bdr.group_raft_details` ビューから返された Raft コンセンサスステータス情報を使用してクラスタ全体の Raft チェックを提供する `functionbdr.monitor_group_raft()` に実装されています。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
 status | message
-----+-----
 OK      | Raft Consensus is working correctly
```

22.12 レプリケーションスロットの監視

各 BDR ノードは以下を保持します。

- アクティブな BDR 同等なごとに 1 つのレプリケーションスロット
- 1 つのグループレプリケーションスロット

例：

```
bdrdb=# SELECT slot_name, database, active, confirmed_flush_lsn
FROM pg_replication_slots ORDER BY slot_name;
```

* slot_name	database	active	confirmed_flush_lsn
bdr_bdrdb_bdrgroup	bdrdb	f	0/3110A08
bdr_bdrdb_bdrgroup_node2	bdrdb	t	0/31F4670
bdr_bdrdb_bdrgroup_node3	bdrdb	t	0/31F4670
bdr_bdrdb_bdrgroup_node4	bdrdb	t	0/31F4670

ピアスロット名は `bdr_<DATABASE>_<GROUP>_<PEER>` の規則に従い、BDR グループスロット名前は `functionbdr.local_group_slot_name()` を使用してアクセスできる Convention `bdr_<DATABASE>_<GROUP>` に従います。

ピアレプリケーションスロットは、すべてのノードで常にアクティブである必要があります。同等なレプリケーションスロットがアクティブでない場合は、次のことを意味します。

- 対応する同等ながシャットダウンされているか、アクセスできません。または BDR レプリケーションが壊れています。ERROR のログファイルを Grep または FATAL およびすべてのノードで `bdr.worker_errors` もチェックします。ルートな原因は、例、互換性のない DDL でした。いずれかのノードで DDL レプリケーションを無効にして実行されます。

一方、BDR グループレプリケーションスロットは、ほとんどの場合非アクティブです。他のすべてのピアがすでに対応するトランザクションを消費しているため、BDR はこのスロットを保持し、LSNを進めます。そのため、グループスロットのステータス（アクティブまたは非アクティブ）をモニタすることはできません。

次のモニタリングアラートレベルをお勧めします。

- status = UNKNOWN、メッセージ=このノードは BDR グループのパートではありません
- status = OK、メッセージ=すべての BDR レプリケーションスロットが正常に機能しています
- status = CRITICAL、メッセージ=少なくとも 1 つの BDR レプリケーションがあります非アクティブなスロット
- status = CRITICAL、メッセージ=少なくとも 1 つの BDR レプリケーションがあります欠落しているスロット

説明されている動作は `functionbdr.monitor_local_replslots()` で実装され、ビュー `bdr.node_slots`（スロットのアクティブまたは非アクティブ）から返される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 BDR ノード複製スロットを考慮したローカルチェックを提供します。

```
bdrdb=# SELECT * FROM bdr.monitor_local_replslots();
 status | message
-----+-----
 OK     | All BDR replication slots are working correctly
```

22.13 トランザクションのコミットの監視

デフォルトでは、BDR トランザクションはローカルノードでのみコミットします。その場合、トランザクション COMMIT は迅速に処理されます。

BDR は標準の PostgreSQL 同期レプリケーションで使用できますが、BDR は CAMO と Eagerreplication の 2 つの新しいトランザクションコミットモードも提供します。これらの各モードは、COMMIT での追加のレイテンシを犠牲にして、追加の堅牢性機能を提供します。COMMIT での追加時間は、レポートカタログを使用して動的に監視できます。1 つ以上の同期的スタンバイからの確認を待機している COMMIT のトランザクションは、SyncRep 待機イベントを報告しますが、2 つの新しいモードは EagerRep をレポートします。

第23章 BDR System Functions

BDR 管理は、主に SQL 呼び出し可能関数を介して実行されます。BDR のすべての関数は、BDR スキーマで公開されます。これらの関数の呼び出しは、`bdr` を `search_path` に入れるのではなく、スキーマで修飾する必要があります。

このページには、文書の他のセクションで説明されていない追加のシステム機能が含まれています。

`pglogical` 関数を使用して BDR が所有するオブジェクトを操作することはできません。以下の提供された関数のみを使用してください。

23.1 バージョン情報 Functions

23.1.1 `bdr.bdr_edition`

このファンクションは、BDR エディションのテキスト表現形式を返します。BDR3 は、Standard Edition (SE) または Enterprise Edition (EE) のいずれかで配布されます。このファンクションを使用して、現在インストールされているものを確認できます。非推奨。

23.1.2 `bdr.bdr_version`

このファンクションは、現在使用中の BDR バージョンのテキスト表現形式を取得します。

23.1.3 `bdr.bdr_version_num`

このファンクションは、現在使用されている BDR バージョンの数値表現形式を取得します。バージョン番号は単調に増加しているため、この値を小なりと大なりの比較に使用できます。

次の式は、メジャーバージョン、マイナーバージョン、およびパッチリリースで構成されるバージョン番号を単一の数値に変換するために使用されます。

$\text{MAJOR_VERSION} * 10000 + \text{MINOR_VERSION} * 100 + \text{PATCH_RELEASE}$

23.1.4 bdr.wal_sender_stats

が有効になっている場合、このビューには、各 WAL センダによって読み取られているデコーダスロットと現在の LCR () セグメントファイルに関する情報が表示されます。

あらすじ

```
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。

23.2 システムおよび進捗情報のパラメーター

BDR は、psql の SHOW を介して、またはクライアントアプリケーションから PQparameterStatus (または同等の) を使用して照会できるいくつかのパラメーターを公開します。このセクションでは、BDR レポートの対象となるこのようなパラメーターをすべてリストします。

23.2.1 bdr.local_node_id

セッションの初期化時に、これはクライアントが接続されているノード ID に設定されます。これにより、アプリケーションは、透過プロキシの背後であっても接続されているノードを把握できます。

また、CAMO と組み合わせて使用されます。 [Connection pools and proxies](#) セクションを参照してください。

23.2.2 bdr.last_committed_lsn

非同期トランザクションのすべての COMMIT の後、このパラメータは更新され、オリジンノードのコミットレコードの終わりを指します。bdr.wait_for_apply_queue との組み合わせにより、アプリケーションはマルチプルのノード間で因果読み取りを実行できます。つまり、トランザクションがリモートで可視されるまで待機できます。

23.2.3 transaction_id

Postgres がトランザクション ID を割り当てるとすぐに、CAMO が有効になっている場合、このパラメータは更新され、割り当てられたばかりのトランザクション ID が表示されます。

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

23.3 ユーティリティ Functions

23.3.1 bdr.wait_slot_confirm_lsn

ユーザは、このセッションの最後の書き込みが 1 つまたはすべてのノードにリプレイされるまで待つことができます。

スロットが特定の LSN を通過するまで待機します。位置が指定されていない場合、現在の書き込み位置がローカルノードで使用されます。

スロット名前が渡されない場合、すべての BDR スロットが LSN を通過するまで待機します。これは、pglogical が提供する機能とは別のファンクションであるため、すべての pglogical スロットではなく、他の BDR ノードに登録されたスロットのみを待機できます。BDR グループスロットではありません。

このファンクションは、他のノードからの変更を 1000 ミリ秒ごとにポーリングします。

スロットが同時にドロップされると、そのスロットの待機が終了します。ノードが現在ダウンしており、スロットを更新していない場合、待機は続行されます。

あらすじ

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

パラメーター

- slot_name-レプリケーションスロットの名前、または NULL の場合、すべての BDR スロット（のみ）
- target_lsn-待機する LSN、または NULL の場合、現在の書き込み LSN を使用しますローカルノード

23.3.2 bdr.wait_for_apply_queue

ファンクション `bdr.wait_for_apply_queue` を使用すると、BDR ノードは、特定の BDR ノードから発信される特定のトランザクションのローカルアプリケーションを待つことができノード。その `peernode` からのすべてのトランザクションがローカルに適用された後にのみ結果れます。アプリケーションまたはプロキシは、この関数を使用して古い読み取りを防ぐことができます。

便宜上、BDR は、CAMO および CAMO パートナーノード（`edbdr.wait_for_camo_partner_queue` を参照）に対してこのファンクションの特別なバリエーションを提供します。

特定の LSN が指定されている場合、それは同等なからのリカバリストリーム内の待機ポイントです。これは、以前の接続または同時接続でその同等なから取得したノードと組み合わせて使用できます。

指定された `target_lsn` が NULL の場合、このファンクションは `localreceive` バッファをチェックし、指定された同等なから受信した最後のトランザクションの LSN を使用しノード。すでに受信したすべてのトランザクションが適用されるのを効果的に待機しています。これは、`peernode` に障害が発生し、送信されたトランザクションが不明な場合に特に役立ちます。この場合、送信側でまだ送信中またはバッファリングされているトランザクションは待機されません。

あらすじ

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

パラメーター

ノード着信元の同等なの名前トランザクションはキューに入れられ、待機する必要があります のために。NULL の場合、すべての同等な適用キューが消費されるまで待機します。ノード同等なからのレプリケーションストリームの LSN 待つために、通常は `bdr.last_committed_lsn` から 同等なノード。

23.3.3 bdr.get_node_sub_receive_lsn

サブスクライバーでこのファンクションを使用して、指定されたオリジンから受け取った最後の LSN を取得できます。適用または非フィルタリングされるトランザクションに関連する LSN 増分のみを考慮に入れるためにフィルター処理されます。

このファンクションの出力と `bdr.get_node_sub_apply_lsn()` の出力の差により、対応する適用キューのサイズが測定されます。

あらすじ

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

パラメーター

- `node_name`-のソースであるノードの名前 LSN を取得するレプリケーションストリーム/
- `committed`-デフォルト (true) の場合、このファンクションはアカウントは、最後ではなく、受信したトランザクションのコミットのみ LSN 全体。サブスクライバーに影響を与えないアクションを含む ノード

23.3.4 bdr.get_node_sub_apply_lsn

サブスクライバーでこのファンクションを使用して、指定されたオリジンから受信および適用された最後の LSN を取得できます。

あらすじ

```
bdr.get_node_sub_apply_lsn(node_name name)
```

パラメーター

- `node_name`-のソースであるノードの名前 LSN を取得するレプリケーションストリーム。

23.3.5 bdr.run_on_all_nodes

すべてのノードでクエリーを実行するファンクション。

!!! Warning * このファンクションは、ノードの DSN で指定されたノード間接続に使用されるユーザの権限を使用して、リモートノードで任意のクエリーを実行します。このファンクションに特権を付与するときは注意が必要です。

あらすじ

```
bdr.run_on_all_nodes(query text)
```

パラメーター

- `query`-実行される任意のクエリー。

注

このファンクションは他のノードに接続してクエリーを実行し、各ノードから json フォーマットで結果を返します。json 配列としてエンコードされた各ノードから複数の行が返される場合があります。ノードがダウンしているために接続できないなどのエラーは、応答フィールドに表示されます。明示的な `statement_timeout` またはその他のランタイムパラメータは設定されていないため、デフォルトが使用されます。

このファンクションは通常のレプリケーションを経由せず、既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドはすでにクラスター内のすべてのノードに送信されているため、`bdr.ddl_replication = off` で接続が作成されます。

このファンクションを使用する場合は、レプリケーションを中断し、ノード間で不整合を引き起こすリスクがあるため注意してください。透過的な DDLreplication または `bdr.bdr_replicate_ddl_command()` を使用して DDL を複製します。DDL は将来のリリースでブロックされる可能性があります。

例

このファンクションは、例次のクエリのモニタリングで使用すると便利です。

```
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"
  *        }
  *      ]
  *    }
  *  }
]
```

(次のページに続く)

(前のページからの続き)

```

*         ]
*         },
*         "node_name": "node2"
*     }
]

```

23.3.6 bdr.global_lock_table

このファンクションは、指定されたテーブルのグローバル DML ロックを取得します。グローバル DML ロックについては、[DDL Locking Details](#) を参照してください。

あらすじ

```
bdr.global_lock_table(relation regclass)
```

パラメーター

- **relation**-ロックされるリレーションの名前または OID。

注

このファンクションは、ddl_locking 設定とは無関係にグローバル DML ロックを取得します。

bdr.backwards_compatibility isset が 30618 以下に設定されていない限り、bdr.global_lock_table ファンクションでは、ロックされた relation で UPDATE、DELETE、または TRUNCATE privilege が必要です。

23.3.7 bdr.wait_for_xid_progress

このファンクションは、特定のノード（ノード ID で識別）で発生した特定のトランザクション（XID で識別）がクラスターで十分に進行 make のを待つために使用できます。進行状況は、ノードに適用されるトランザクションとして定義され、このノードは、トランザクションが適用される前に他のすべてのレプリケーションの変更が行われたことを確認します。

あらすじ

```
bdr.wait_for_xid_progress(origin_node_id oid, origin_topxid int4, allnodes boolean,  
→DEFAULT true)
```

パラメーター

- **origin_node_id**-トランザクションがあったノードのノード ID 発生しました。
- **origin_topxid**-トランザクションの XID。
- **allnodes**-true の場合、トランザクションが進行するのを待つすべてのノード。そうでなければ、現在のノードを待つだけです。

注

これらのトランザクションのみが現在追跡されているため、このファンクションは、DDL コマンドを複製したトランザクションに対してのみ使用できます。間違った **origin_node_id** または **origin_topxid** が指定された場合、ファンクションは永遠に、または **statement_timeout** がヒットするまで待機する場合があります。

23.3.8 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
```

23.3.9 bdr.node_group_type

指定されたノードグループのタイプを返します。戻り値は、ノードグループの作成時に **bdr.create_node_group()** に渡されたものと同じですが、グループの作成時に **node_group_type** が NULL として渡された場合は **normal** が返されます。

例

```
bdrdb=# SELECT bdr.node_group_type('bdrgroup');
node_group_type
-----
normal
```

23.4 グローバルアドバイザリーロック

BDR はグローバルアドバイザリーロックをサポートしています。これらのロックは、BDR がサポートするアドバイザリーロックが本質的にグローバルであることを除いて、PostgreSQL で使用可能なアドバイザリーロックに非常に似ています。DDL ロックと同様のセマンティクスに従います。したがって、すべてのノードの過半数が連携できる限り、1 つまたは複数のノードがダウンまたは遅れている場合でも、多数決コンセンサスによってアドバイザリーロックを取得できます。

現在、排他ロックのみをサポートしています。そのため、同じノード上の別のノードまたは別のバックエンドがオブジェクトのアドバイザリーロックを既に取得している場合、他のノードまたはバックエンドはロックが解除されるのを待つ必要があります。

アドバイザリーロックは本質的にトランザクションです。そのため、トランザクションの終了前に明示的に解放されない限り、トランザクションが終了すると自動的にロックが解除されレベル。その場合、リリースされるとすぐに使用可能になります。

グローバルアドバイザリーロックは再入可能です。したがって、同じリソースが 3 回ロックされている場合、他のセッションで使用するために 3 回ロック解除する必要があります。

23.4.1 bdr.global_advisory_lock

このファンクションは、提供されたオブジェクトの排他ロックを取得します。ロックが使用可能になるか、bdr.global_lock_timeout に達するまで待機します。

あらすじ

```
bdr.global_advisory_lock(key bigint)
```

パラメーター

- key-アドバイザリロックが取得されるオブジェクト。

あらすじ

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

パラメーター

- key1-複合キーの最初のパート。
- key2-複合キーの 2 番目のパート。

23.4.2 bdr.global_advisory_unlock

このファンクションは、アプリケーションが定義したソースで以前に取得したロックを解放しました。ロックは、アプリケーションによって同じトランザクションで以前に取得されている必要があります。そうでない場合は、エラーが発生します。

あらすじ

```
bdr.global_advisory_unlock(key bigint)
```

パラメーター

- key-アドバイザリロックが取得されるオブジェクト。

あらすじ

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

パラメーター

- `key1`-複合キーの最初のパート。
- `key2`-複合キーの 2 番目のパート。

第24章 Credits and Licence

BDR3 は、このチームによって設計、開発、テストされています。

- ペトル・イエリネク
- クレイグリナー
- マーカス・ワナー
- パヴァン・デオラシー
- トマス・ボンドラ
- サイモン・リッグス
- ニキル・ソントัก
- パラヴィ・ソントัก
- アムルタデオラシー
- ラヒラ・サイード
- Ashutosh Bapat
- アビジット・メノン・セン
- フローリンイリオン
- オリバー・リッグス

Copyright © 2018-2020 2ndQuadrant LtdCopyright © 2021 EnterpriseDB UK Ltd

BDR3 は EDB 使用ライセンスの下で提供されます。

これらの文書の複製は禁止されています。

第25章 ‘ Appendix A

25.1 BDR 3.7.12

これは BDR 3.7 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。

BDR にも影響する解決済みの問題については、pglogical 3.7.12 のリリースノートも確認してください。

25.1.1 改善

- キャッシュとロックの改善による単一デコードパフォーマンスの調整 (BDR-1311、BDR-1312) BDR のキャッシュを追加-デコードに関する内部カタログ情報労働者。単一のグローバルロックをマルチプルのロックに分割します (WAL ごとに 1 つ WAL 送信者の内部ステータス情報へのアクセス用。これにより、特に多くの同時 WAL 送信者のパフォーマンスが向上しますプロセス。
- 新しいビュー `bdr.replication_status` を追加します (BDR-1412) これはビュー `pglogical.replication_status` に似ており、ローカルノードのレプリケーションステータスに関する情報クラスター内の他のすべての BDR ノードを尊重します。
- ウォルセンダーのデコーダーモニタリングパラメーターを追加 WAL 送信者統計を拡張して、LCR を使用するかどうかをレポートするデコードワーカーと正確な LCR ファイルによって出力される該当する場合、現時点で読み取ります。
- CAMO をデコードワーカーと組み合わせて使用しないようにします (BDR-792) 現在、これらの機能を組み合わせて使用することはできません。このリリース多くの場合、両方を有効にできません。これはただ設定ミスを防ぐためのベストエフォート戦略。
- `bdr_init_physical` の `postgresql.auto.conf` ファイルの指定を許可 (RT72989、BDR-1400) コマンドライン引数を `bdr_init_physical` に追加して、`postgresql.auto.conf` に使用するカスタムファイルを提供します。

25.1.2 解決された問題

- `bdr_init_physical` の潜在的なデータロスの問題を修正 (RT71888) スロット名前を再利用すると、以前の状態が適切にクリーンアップされませんでしたすべての場合。これにより、物理的にデータロスされる可能性がありますスロットは `bdr_init_physical` によって事前に作成されるため、結合します同じ名前で。物理的からロジカルへの移行レプリケーションはレプリケーションストリームのパートを見逃す可能性があります。スロットを再作成します。このリリースはスロットを適切にクリーニングします情報をドロップすると、データロスを防ぎます。

- `bdr.camo_local_mode_delay` を修正して、実際に起動します (BDR-1352) この人為的な遅延により、そうでない CAMO ノードを調整できます現在 CAMO パートナーに接続して、生産を防止しています CAMO パートナーが適用できるよりも速いトランザクション。で以前のバージョンでは、後に適切に起動しませんでした `bdr.global_commit_timeout` 量のラグ、ただし 1000 回のみ後で (誤って秒とミリ秒を比較したため)。
- サードパーティの出力プラグインと組み合わせたセグメンテーション違反の防止 (BDR -1424, RT72006) BDR の Eager All に固有のロジカル WAL メッセージのハンドリングを調整する BDR とは無関係の出力プラグインのノード複製モード。これ例、Debezium の `decoderbufs` 出力プラグインが機能するようにします BDR とともに。

Postgres 13 との互換性を改善 (BDR-1396) `ReplicationSlotAcquire` の API の変更に合わせて調整してください。非ブロッキングが要求された場合の意図しないブロッキングへ、およびその逆その逆。 PGLogical のこのバージョンは、この潜在的な問題を排除し、これは稼動システムではこれまで観察されていません。

- コミット決定を含む Raft スナップショットのシリアル化を修正 (CAMO、BDR-1454) タプル数の不一致がシリアル化につながる可能性がありますまたは後に撮影された `Raftsnapshot` の逆シリアル化エラー CAMO または Eager All Node レプリケーションを使用したトランザクションが使用されました最近、コミット決定を保存しました。
- `bdr_init_physical` の `--recovery-conf` オプションを修正

25.1.3 アップグレード

このリリースでは、次のバージョンの BDR からのアップグレード処理がサポートされています。

- 3.7.9 以降
- 3.6.27

25.2 BDR 3.7.11

これは BDR 3.7 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。

BDR にも影響する解決済みの問題については、`pglogical 3.7.11` のリリースノートも確認してください。

25.2.1 改善

- デコードワーカーのデバッグログを削減 (BDR-1236、 BDR-1239)
- コンセンサスの最大接続の構成を許可 (BDR-1005) これにより、非常にラージクラスターをセットアップできます。

25.2.2 解決された問題

- autopatition およびエグゼキュータの snapshot ハンドリングを修正 PostgreSQL の最新バージョンとの互換性のため
- CAMO のデッドロックハンドリングを修正これにより、クロス CAMO での競合の解決が非常に遅いという問題が解決しますセットアップ。
- 競合をロギングするときにスロットタブルのコピーを取得します (BDR-734) そうしないと、行を早期にマテリアライズして、存在するときに誤った更新を引き起こす可能性があります下流の追加の列の。
- LCR セグメント削除ロジックの改善 (BDR-1180、BDR-1183、BDR-993、BDR-1181) 間のより小さいすべての LSN の LCR セグメントを保持することを確認してくださいグループスロット LSN およびデコードワーカースロット LSN。
- 内部コネクションプーラへの同時接続のハンドリングを修正プールの所有者 (コンセンサスワーカー) がリストを再作成しています (BDR-1113)

25.2.3 アップグレード

このリリースでは、次のバージョンの BDR からのアップグレード処理がサポートされています。

- 3.7.9 以降
- 3.6.27

25.3 BDR 3.7.10

これは BDR 3.7 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。

25.3.1 改善

- `bdr.monitor_group_raft()` のラフトクォーラムを確認 (BDR-960) 少なくとも次の場合、`bdr.monitor_group_raft()` に「CRITICAL」ステータスを返します投票ノードの半分は到達不能です。
- `bdr_monitor` ロールが追加の情報ビューを読み取ることを許可します。 (BDR-732)
- `bdr.group_camo_details`
- `bdr.group_versions_details`
- `bdr.group_raft_details`
- `bdr.group_replslots_details`
- `bdr.group_subscription_summary`

- `is_decoder_slot` を `bdr.node_slots` に追加して、
 - デコーダーワーカー

25.3.2 解決された問題

- ポストマスターが死亡した場合、コンセンサスワーカーを常に終了させる (BDR1063、RT70024)
- リスタート後のデコードワーカーの LSN の開始を修正デコードワーカーが再起動すると、既存の LCR セグメントをスキャンして検索します LSN、それまでのトランザクション、完全にデコードされます。この LSN がスロットの確認済みの LSN よりも高い場合、デコードする前にスロットを更新しますトランザクション。これにより、トランザクションがマルチプルデコードおよび複製されることを回避できます。回。 (BDR-876、RT71345)
- 物理的スタンバイのデコードワーカーのレプリケーションスロットを同期しない WAL デコーダーが初めて起動するとき、デコードワーカーのスロットにはすべての WAL センダスロットの背後にあるため、必要な WAL をデコードします。WAL 送信者。しかし、プライマリのスロットはすべての WAL 送信者よりも先に移動しましたそのため、同期化は役に立ちません。それは物理的後に新しく作成されますスタンバイが昇格されます。 (BDR-738)
- デコードワーカーが有効な場合の結合パフォーマンスの改善 `fsync = on` の場合、新しいノードをクラスターに参加させるには、デコードワーカーが有効になりました。また、ノードビルドアップは、結合のソース。これは、デコードワーカーが LCR セグメントを同期したためです。頻繁に。頻度を減らすことで問題を修正しました。 (BDR-1160、RT71345)
- デコードワーカーが使用されている場合の UPDATE / UPDATE 競合の TOAST ハンドリングを修正
- デコードワーカーが使用されている場合の追加オリジンのフィルタリングを修正これは主に、BDR とデコードワーカーおよび個別の `pglogical` の混合に影響しますレプリケーション。
- `bdr.raft_leadership_transfer` のハングの可能性を排除 (BDR -1039) `wait_for_completion` との組み合わせで、ベストエフォートアプローチはオリジナルのリクエストが適切に送信された場合の無限ループへ、しかし、実際のリーダーシップの移転は依然として失敗しました。
- PGL マネージャがワーカーをスタートできないときにエラーをスローしない (RT71345) PGL マネージャがエラーをスローした場合、再起動されます。責任があるのでノードの状態と他の BDR 管理タスクを維持するためこのようなエラーで再起動すると、BDR クラスターの状態に影響します。代わりに、警告をログします。
- 結合中の `repset` 構成ハンドリングをより決定的にします (RT71021) `autoadd_tables` オプションは、以前のすべてのケースで尊重されるとは限りません。
- `bdr.node_summary` の `pub_repsets` および `sub_repsets` を廃止 (BDR-702、RT70743) 偽の情報ではなく、常に NULL と表示されるようになり、完全に削除されます次のメジャーバージョンで。
- オリジンノードとターゲットノードが含まれる場合、`bdr.node_slots` にノードとグループの情報を表示異なるグループ。
- `bdr.monitor_local_replslots()` がスタンバイノードを理解し、加入者専用グループ構成であり、スロットはチェックされませんこれらの状況では必要ありません (BDR-720)
- フリー接続スロットを再利用しない可能性のある内部コネクションプーラを修正 (BDR-1068)

- 不足している列のエラーメッセージで報告されたスキーマ名前を修正 (BDR-759)

25.4 BDR 3.7.9

25.4.1 改善

- グループスロットを返す `bdr.local_group_slot_name()` ファンクションを追加名前 (BDR-931) 主にモニタリングに役立ちます。

BDR ワーカーに関する追加情報を表示する `bdr.workers` ビューを追加します (BDR-725) BDR 固有のアクティビティのモニタリングを支援します。特に参加時に有用 `bdr.stat_activity` を使用します。

- 転送されたトランザクションのロジカルスタンバイでの並列適用を許可 (BDR-852) 以前は、レプリケートされた変更にのみ並列適用を使用できました。ロジカルスタンバイのアップストリームから直接、ただし変更なし別のノードから来ます。
- `bdr.batch_inserts` 構成変数の導入 (RT71004, RT70727) これは、(同じトランザクション内の) 同じテーブルにいくつかの行が入った後に設定します BDR はマルチ挿入戦略にスイッチします。
 - これにより、通常、INSERTs または COPY コマンドを使用して、ラージのデータのレプリケーションのパフォーマンスが向上します。ただし、BDR 3.7.8 は常にこの戦略を使用しようとするため、多くの単行の挿入のみを行うワークロードのパフォーマンスが低下します。

25.4.2 解決された問題

- ノードパート/ドロップで WAL デコーダーインフラストラクチャを破棄します (BDR-1107) これにより、ノードが次の場合に WAL デコーダーインフラが削除されます。クラスターから分離されました。LCR ディレクトリと同様に、デコーダースロット。これにより、ノードがクラスターに再びきれいに結合できるようになります後で、必要な場合。
- サブスクライバー専用ノードで WAL デコーダーをスタートしないでください (BDR-821) サブスクライバー専用ノードは、変更を他のノードに送信しませんクラスター。したがって、WAL デコーダーインフラおよび WAL デコーダーは必要ありませんプロセス自分自身。これを修正すると、加入者のみがノードは未使用のスロットのため WAL を抑制しません。
- PROMOTE 状態に達した後にのみ WAL デコーダーを起動します (BDR-1051) ノードが結合を開始するときに、WAL デコーダーインフラストラクチャを作成していましたプロセス。それは早すぎて、WAL の蓄積につながる可能性がありますロジカルスタンバイ。代わりに、WAL デコーダーインフラストラクチャを作成しますノードが PROMOTE 状態に達したときのみ。それが他のときの状態でノードはノードへの接続をスタートする可能性があるため、WAL デコーダーが必要です。
- サブスクライバーのみのノードでのグループスロットアドバンスの修正 (BDR-916, BDR-925, RT71182) これにより、サブスクライバーのみのノードでの過剰な WAL ログ保持が解決されます。
- を使用して加入者専用ノードに参加するときに正しいスロット名前を使用する `bdr_init_physical` (BDR-895, BDR-898, RT71124) `bdr_init_physical` が間違っただのスロットを作成するために使用されたため、2 スロットになりました加入者専用ノードが次を使用して参加されたときに、結合ソースノードに存在するこのメソッド。これにより、結合ソースに過剰な WAL が保持されます。ノード

- グループモニタリングビューを修正して、ノードごとに複数の行を許可する (BDR-848) グループモニタリングビューは、以前に任意のノードから情報を切り捨てました複数行の情報を報告します。これにより、例 `bdr.group_replslots_details` にスロットがありません。
- CAMO の正しいコミットキャンセル (BDR-962) これにより、ユーザがクエリーをキャンセルしたときの CAMO の動作が再び修正されます。
- レシーバのリスタート後にグローバルロックカウンターの状態を復元する (BDR-958) すでにロックを復元しましたが、原因となる可能性のあるカウンターは復元していません並列適用を使用する場合、グローバルロック中にデッドロックが発生します。
- マルチプルある場合の `skip_transaction` 競合リゾルバのハンドリングを修正 `skip_transaction` を引き起こしたトランザクションの後のトランザクションの変更 (BDR-886)
- 自動パーティションテーブルの Raftsnapshot 作成を修正 (RT71178, BDR-955) 以前は、Raftsnapshot は自動パーティションの状態をアカウントしていませんでした情報を書き込むときのすべてのノードでのタスク。これにより、長時間のダウンタイムの後、ノードはパーティション作成をスキップします。
- 自動パーティションでトランザクションと snapshot のハンドリングを調整します (BDR-903) これにより、自動パーティション処理中に有効な snapshot が使用されます回。以前のアプローチは、将来のポイントリリースで問題を引き起こす可能性があります PostgreSQL の。
- 自動パーティションでの KSUUID 列の検出を修正
- `bdr.drop_node()` ファンクションによるノードステータスの誤報告を修正
- 正しいシーケンスタイプが常にグローバル `galloc` に設定されていることを確認してくださいシーケンス状態。
- いくつかのコマンドの DDL レプリケーションとロック管理を修正 (BDR-874) `ANALYZE`、`CHECKPOINT`、`CLUSTER`、`PREPARE / COMMIT / ABORT TRANSACTION`、`MOVE`、`RELEASE`、`ROLLBACK` は複製されたものとして文書化されており、これらのいくつかはすべきではない DDL ロックを取得しようとしていました。
- 複製されていないユーティリティコマンドのロギングを削減 (BDR-874) `PREPARE` と `EXECTUE` は、誰としても複製されないことについて spam する必要はありません。彼らがいることを期待しています。
- `ALTER TABLE ... SET` のグローバルロックを修正 (BDR-653) グローバル DML ロックを取得しないでください。
- `TRUNCATE` コマンドの複製方法文書を修正 (BDR-874) `TRUNCATE` はグローバルロックを取得できませんが、他の方法では複製されません DDL コマンドは、レプリケーションセットに従って、DML のように複製されます設定。
- CAMO と Eager が現在、Decoding Worker で動作しないことを文書化 (BDR-584)
- ドキュメントの複数のタイポと文法の修正。

25.5 BDR 3.7.8

これは、BDR 3.7 の最初の安定 (stable) リリースです。3.7.7 で特定された問題の新しい主要機能と修正の両方が含まれています。

25.5.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

このリリースでは、BDR 3.6.25 および 3.7.7 からのアップグレードがサポートされています。

25.5.2 BDR 3.7 のハイライト

PostgreSQL 11、12、および 13 のサポート

EDB Advanced Server のサポート Standard Edition と Enterprise Edition の両方が使用可能になりました EDB Advanced Server

- 並列適用 レプリケーションを適用する並列ライターの数を構成できますストリームこれは、Enterprise Edition でのみサポートされる機能です。
- 自動パーティション自動作成により、テーブルパーティションの自動管理が可能になり、設定可能な保持期間などを備えた自動クリーンアップ。

BDR WAL デコードワーカーを分離するオプションを導入これにより、数に関係なく、各ノードで単一のデコードプロセスを使用できます。それに接続されたサブスクリプションの。デコードされた情報は、ロジカル変更レコード (LCR) ファイルに保存されます。従来の WAL と同様の方法で他のノードにストリーミングされます。デコードと walsender のオプションの分離。これは、Enterprise Edition でのみサポートされる機能です。

- **subscriber-only** ノードの概念を実装するこれらは完全に結合されたノードですが、レプリケーションを送信しませんクラスター内の他の BDR ノードへの変更。しかし、彼らは変更を受け取りますクラスター内のすべてのノードから（もちろん、他のサブスクライバーのみを除くノード）。彼らは RAFT 投票プロトコルに参加していないため、彼らの存在（または不在）は RAFT リーダーの選挙を決定しません。これらのノードにはレプリケーションスロットを作成する必要はありません。レプリケーションの変更を送信しないでください。同様に、何も作成する必要はありません他の BDR ノード上のこれらのノードのサブスクリプション。
- **CREATE TABLE ... AS** および **SELECT INTO** ステートメントをサポート
 - この機能は、Enterprise Edition でのみサポートされるようになりました。
- 物理的表現を改善するオーダーに BDR サブグループを定義する新しい機能 BDR クラスターの構成。これにより、BDR クラスターが分散している構成も簡素化されます。マルチプルデータセンターとデータベースのパートのみが複製されます各サブグループが新しいデフォルトのレプリケーションを自動的に持つデータセンターそれに割り当てられたセット。

- 複数の新しいモニタリングビュー主にグループレベルのモニタリングと進行中のモニタリングに焦点を当てています適用側。
- 競合はデフォルトで `bdr.conflict_history` に記録されます行単位セキュリティを使用してパーティションテーブルにログを記録し、簡単にアプリケーションユーザーの競合へのアクセス。
- 新しい競合タイプ `multiple_unique_conflicts` および `apply_error_ddl` よりエッジケースの状況でレプリケーションを継続できます
- 一部の DDL ステートメントのロックレベルを削減また、文書化されたロックヘルプは、他のマルチプルの DDL ステートメント。
- 更新と削除を適用するときに利用可能な最適なインデックスを使用するこれにより、`REPLICA IDENTITY FULL` テーブルのパフォーマンスが大幅に向上します。プライマリキーがありません。

以下は 3.7.7 以降の変更点です。

25.5.3 改善

EDB Advanced Server (EE) での並列適用のサポート

- 必要に応じて進捗レポートの頻度を増やします (BDR-436、BDR-522) これにより、DML を使用せずに `VALIDATE CONSTRAINT` のパフォーマンスを高速化できます。ロック。

SQL セッションから設定可能なすべての BDR 構成オプションを変更して、Postgres スーパーユーザだけでなく、`bdr_superuser` によって設定可能。

- `bdr.run_on_all_nodes()` で `bdr.ddl_replication` をオフに設定 (BDR-445) 通常、DDL を使用して実行された DDL を複製することは望ましくありません。すべてのノードですでに実行されているため、`bdr.run_on_all_nodes()` ファンクション。
- 適用側で進行中のトランザクションのモニタリングを改善する (BDR-690、BDR -691) DDL およびいくつかの追加の適用時に `pg_stat_activity` にクエリーを追加します最新の受信の LSN を示す `bdr.subscription_summary` ビューのフィールド変更、最新の受信コミットの LSN、コミット LSN の適用、LSN のフラッシュ、およびタイムスタンプを適用しました。
 - これは、特にラージトランザクションに関しては、レプリケーションの進行状況をモニタリングのに役立ちます。
- `pg_stat_activity` と同様ですが、BDR を表示するビュー `bdr.stat_activity` を追加します特定の待機状態。
- 初期データ同期以外のバッチ挿入を許可する既存の BDR グループへのビッグデータのロードのパフォーマンスを改善します。
- `DROP INDEX` で取得したグローバルロックレベルを DML Global Lock から DDL グローバルロック (BDR-652)

25.5.4 解決された問題

- いくつかの DDL コマンドのレプリケーション設定を修正一般 make に、実際の動作と文書化された動作を確認してください許可されるもの、複製されるもの、DDL 中に保持されるロックレプリケーションマッチ。

例、TABLESPACE 関連のコマンドは複製しないでください。

- 同時結合の競合条件を修正します。（BDR-644、BDR -645）ローカルノードに既にある場合は、常に最初に有効なサブスクリプションを作成します PROMOTING 状態を超えました。
- すでに保持されているロックのグループリーダーを設定する（BDR-418、BDR-291）これにより、「グローバルロックタイムアウトによるステートメントのキャンセル」が解決されます。ライタが既にテーブルを開いていた場合の DDL 操作。これはパーティショニングまたは並列適用が関係する場合は特に問題です。
- WAL デコーダー入力に基づいて進行中の WAL 送信者のスロット（BDR-567）これがないと、サーバーは最終的に単一のデコードワーカーで動作しなくなる可能性があります。
- bdr_init_physical の TEMPORARY レプリケーションスロットに切り替えます（BDR-191）これにより、bdr_init_physical の実行後に適切にクリーンアップされます。
- 不要になった XID 進行状況レコードをクリーンアップします（BDR-436、BDR-532）xid progress snapshot のサイズを縮小します。
- BDR ライターで apply_timestamp を正しく追跡する（BDR BDR）3.7.7 では更新されませんでした
- EPAS での BDR ストリームトリガーの作成を修正（BDR-581）以前は、間違ったトリガータイプとして作成されていました。
- LCR ファイルに保存され、walsender に渡されるオプションのエラーハンドリングを改善異なる（BDR-551）
- 最上位ノードグループに対してのみ WAL デコーダー構成を有効にします（BDR-566）一般に、トップノードグループのグループ構成の変更のみを許可します。
- 特定の BDR カタログ列に「C」照合順序または「名前」タイプを使用します（BDR -561）これにより、BDR カタログの潜在的なインデックス照合順序の問題が解決されます。
- CAMO の正しいコミットキャンセルこれにより、ユーザがクエリーをキャンセルしたときの CAMO の動作が修正されます。
- 既存のパーティションを持つテーブルの自動パーティションハンドリングを修正（BDR-668）
- BDRWrite でリモート ID のないリレーションをキャッシュしない（BDR-620）いくつかの形式の TRUNCATE コマンドの後のレプリケーションの破損を修正します。
- Wal デコーダーのアップストリーム dbname を考慮したアップストリームデコーダースロット名前の作成（BDR-460）wal デコーダーが使用するスロット名を修正します。
- WAL デコーダーと LCR を使用する WAL 送信者が使用する正しい BDR 出力オプションを使用する（BDR-714）
- 壊れたクラスターでのモニタ機能のクラッシュを修正。（BDR-580、BDR-696）

- `bdr.node_slots` ビューで `PARTED` に存在しないスロットを表示しない
- ノードのドロップ時にストリームトリガーをドロップ (BDR-692) これにより、ストリームトリガーで `bdr_init_physical` を使用できるようになります。
- `SIGUSR2` シグナルのハンドリング中にセグメンテーション違反が発生しないようにしますシグナルはプロセスのライフタイムのどの時点でも到着する可能性がある `make`、現在の状態に関する仮定。
- 自動パーティションの欠落につながる可能性のあるテーブルの同時ドロップを処理するルール
- 異なるエラーの処理中にエラーが発生してもクラッシュしないことを確認してください
- バックグラウンドワーカーの場合、グローバル `xid` をクライアントに送信しないでくださいこれを送る人はいません。

25.5.5 その他の変更

- `bdr.permit_unsafe_commands` がオンの場合、セッションレベルの `bdr.xact_replication = off` を許可 `pg_restore` を使用してデータベースに手動でデータを入力するときに役立ちます。
- さまざまなドキュメントの改善
- テーブルがレプリケーションセットから削除されている場合、テーブルが既に `repset` にありません (BDR-562)
- `check_constraints` オプションを再度許可しますが、適切にマークされていることを確認してください非推奨 (BDR-26) BDR 4.0 で削除されます。
- WAL デコーダーが有効/無効になっている場合、WAL 送信者の管理をマネージャプロセス (BDR-612) コンセンサスワーカーでそれらを管理すると、コンセンサスサブシステム。
- より多くの場所で割り込みをチェックします
 - 暴走ループの可能性を減らす必要があります

25.6 BDR 3.7.7

これは、BDR 3.7 のベータリリースです。3.7.6 で特定された問題の新しいメジャー機能と修正の両方が含まれています。

25.6.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

ベータ版ソフトウェアは稼働環境ではサポートされていません-アプリケーションテストのみ

このリリースでは、BDR 3.6.25 および 3.7.6 からのアップグレードがサポートされています。

25.6.2 改善

EDB Advanced Server で Enterprise Edition 機能をサポートこれは、特に CAMO と Eager レプリケーションを除外します。

EDB Advanced Server DDL コマンドのほとんどをサポート (EBC-45) キューに関連する DDL は複製されますが、キューの内容は複製されることに注意してください。複製されません。

DDL レプリケーションのハンドリングを調整して、コマンドレベルではなく、内部表現形式 (BDR-275) これにより、主にフィルタリングと文書が容易になります。

- Enterprise Edition で SELECT INTO ステートメントを許可 (BDR-306)
- COPY FROM で BDR シーケンスを処理する (BDR-466) COPY FROM は、列のデフォルトを独自に処理します。クエリープランナフックで 사용되는ため、キャッチされません式プランナ。悲しいことに、式プランナにはフックがありませんしたがって、実際の COPY FROM コマンド自分自身を処理する必要があります。
- bdr.run_on_all_nodes の改善 (BDR-326、 BDR-303) 戻り値の型を jsonb に変更し、常に各コマンドのステータスを結果ます。受信した実際のエラーメッセージを返すことにより、エラー報告を改善します。リモートサーバ。
- conflict_history にさらに情報を追加 (BDR-440) これにより、競合履歴テーブルにいくつかの新しいフィールドが追加され、簡単になります。実際のデータを見なくてもタプルを識別できます。
 - 最初の 1 つは origin_node_id で、これはサブスクリプションのオリジンとは異なる可能性のある変更のオリジンを指します。状況によっては、異なるオリジナルのノードから変更を転送するためです。
 - 2 つ目は change_nr で、これはトランザクションの (カウンタに基づく) 変更の数を表します。1 つの変更は、1 つのオリジナルのコマンドではなく、1 つの行を表します。
 - これらは、競合履歴サマリテーブルにも追加されます。
 - local_time を bdr.conflict_history_summary に追加しますモニタリングは bdr.conflict_history のパーティションキーです。これは、クエリを効率的に実行できるようにするために必要です。
- bdr_init_physical に -node-group-name オプションを追加 bdr.join_node_group の node_group_name と同じ-参加を許可しますノードのサブグループ。
- LCR を WAL デコーダスロット (BDR -60) にちなんでも名前付けたディレクトリの下に保存する Pglogical は LCR をレプリケーションスロットにちなんでも名前付けたディレクトリに保存しますそれらを生成するために使用。
- WAL デコーダーと送信者の調整におけるさまざまな改善 (BDR-232、BDR-335、 BDR-342) WALDecoder waitlsn に関する情報を公開し、WALSender その情報を使用して待機し、必要なときに WALDecoder にシグナルを送ります WAL が利用可能です。これにより、不要なポーリングが回避され、改善されます 2 つのコーディネーター。
- 単一のデコーダーワーカー GUC オプションの変更。 (BDR-222) bdr.receive_logical_change_records を bdr.receive_lcr に変更し、bdr.logical_change_records_cleanup_interval から bdr.lcr_cleanup_interval

- CAMO / Eager コードのほとんどを BDR に移動します (BDR-330) CAMO および Eager All Node を Postgres パッチに依存しにくくします。
- 初期同期の並列化をサポートします。並列適用が有効な場合、ロジカル結合中の初期同期は同様に並列化されます。
- `bdr.set_ddl_replication` および `bdr.set_ddl_locking` を廃止します。

25.6.3 解決された問題

- `bdr_stop_wal_decoder_senders()` (BDR-232) のロジックを修正 `bdr_stop_wal_decoder_senders()` が待機する期間を増やします WAL 送信者のステータスを再度確認する前に。
- 場合によっては `ALTER TABLE..ADD FOREIGN KEY` の実行を許可しない (EBC-38、BDR-155) 現在のユーザに読み取り権限がない場合 被参照テーブル、`ALTER TABLE ADD FOREIGN KEY` を許可しない そのようなテーブルに
- 一時オブジェクトと永続オブジェクトが混在するクエリの検出を改善そうしないと、レプリケーションを中断する可能性があります。
- `INTO TABLE` 句を使用する場合の `EXPLAIN` ステートメントを修正。
- ユーティリティコマンドと DML が混在する場合の `bdr.run_on_all_nodes()` のクラッシュを修正 (BDR-305)
- EPAS の古いマイナーバージョンでの CTAS ハンドリングを修正
- テーブル定義チェックの統合 (BDR-24) これにより、チェックや作成を見逃してしまういくつかの隠れたバグが修正されます。余分なオブジェクトの
- 無効なインデックスの `REINDEX` および `DROP` インデックスを修正 (BDR-155、EBC-41) インデックスが無効な場合、`REINDEX` はエラーをスローします。ユーザーは無効にドロップできます `DROP` インデックス `if_exists` を使用したインデックス。
- ローカルノードグループメンバシップのチェックを改善します (BDR-271) いくつかの関数、つまり `bdr_wait_for_apply_queue` と `bdr_resynchronize_table_from_node` はこのチェックを行わなかったため、潜在的にクラッシュを引き起こす。
- 誤解を招く CTAS エラーを修正基になるサポートされていないユーティリティまたは複製されていないユーティリティの場合、エラーが発生し、基礎となるユーティリティについて言及する必要があります。
- WAL デコーダーの有効化に関する修正と改善 (BDR-272、BDR-427)
- `pglogical manager` の WAL デコーダーインフラストラクチャの削除を修正 (BDR-484)

25.7 BDR 3.7.6

これは、BDR 3.7 のベータリリースです。3.7.5 で特定された問題の新しいメジャー機能と修正の両方が含まれています。

25.7.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

ベータ版ソフトウェアは稼動環境ではサポートされていません-アプリケーションテストのみ

このリリースでは、BDR 3.6.25 からのアップグレードがサポートされています。

25.7.2 改善

BDR WAL デコードワーカーを分離するオプションを導入 (RM18868、BDR-51、BDR-58) これにより、数に関係なく、各ノードで単一のデコードプロセスを使用できます。それに接続されたサブスクリプションの。デコードされた情報は、ロジカル変更レコード (LCR) ファイルに保存されます。従来の WAL と同様の方法で他のノードにストリーミングされます。

- CAMO と Eager の並列適用を有効にします (RM17858)
- BDRWriter でリレーションキャッシングをリワークこれにより、キャッシュルックアップの間に発生した無効化の欠落が修正されます。とテーブルの開口部。また、ハッシュテーブルルックアップの量を減らしました (パフォーマンスの向上)。
- 単一の DDL コマンドで一時オブジェクトと永続オブジェクトを混在させないでください (BDR-93) 一時オブジェクトで機能する DDL を複製しようとしなことが重要です。そのため、DDL はレプリケーションを確実に中断します。
- `bdr.alter_subscription_skip_changes_upto ()` を追加 (BDR-76) 指定された LSN までのレプリケーションの変更をスキップできますサブスクリプション。同様のファンクションが `pglogical` にすでに存在します。
- `snapshot` エントリーハンドラのルックアップをより堅牢にします (BDR -86) これにより、コンセンサス `snapshot` で将来のバグを導入 `make` ことが難しくなります。ハンドリング
- `bdr.consensus_snapshot_verify ()` を追加 (BDR-124) 前に提供されたコンセンサス `snapshot` が正しいことを確認するために使用できますそれを `bdr.consensus_snapshot_import ()` に渡します。
- 特定のほとんどの DDL コマンドのサポートを追加 EDB Postgres Advanced Server (EBC-39、EBC-40)
- 複製されることを想定していない複製されていないコマンドの警告 `spam` を削減する最初の場所で複製されます (VACUUM など)
- CAMO 構成に関する警告とヒントを改善する

25.7.3 解決された問題

- ライタでリレーションを開く前に、xid が割り当てられていることを確認してくださいこれにより、並列適用のデッドロック検出が改善されます。
- ファンクション `drop_trigger` でテーブル oid をチェック (BDR -35) 無効な oid がファンクションに渡されたときにクラッシュする問題を修正しました。
- 古いコンセンサススナップショットの適用を修正 (BDR -231) 欠落しているグループ UUID を正しくハンドルせずに 3.7 ノードが発生していました 3.6 クラスタに結合できません。
- デフォルトの切り捨てハンドリングの再調整 (BDR -25) デフォルトではロックしないでください。これは故障の可能性を引き起こす可能オーダーがありますが切り捨て、後方互換性が向上します。
- `CREATE FOREIGN TABLE` ステートメントで `OPTION` 句を使用するとクラッシュする問題を修正 (EBC-37)
- 古いノードとの通信中に余分なデータを送信しないようにしますコンセンサスプロトコル (BDR-135)
- 混在バージョングループのコンセンサス snapshot の `kv_data` パートを読み取ります (BDR-130) 両方の BDR 3.6. 3.7 コンセンサス snapshot のこのパートを記述しますが、BDR 3.7 snapshot も 3.7 で書き込まれた場合にのみそれを読み取ります。
- `bdr` を移動します。EE スクリプトへの制約 (EBC-36) エンタープライズ版のみの機能なので、カタログのみをインストールする必要がありますエンタープライズ版。
- `TABLESPACE` および `Large` で `GRANT / REVOKE` コマンドを複製しようとしないでください オブジェクトこれらのオブジェクトは複製されないため、`GRANT` と `REVOKE` を複製しようとするとレプリケーションを中断します。
- CAMO がリプレイの進行をブロックしないようにします (RT69493)
- 失敗した CAMO 接続ハンドリングを修正 (RT69493、RM19924) 状態マシンを修正して、適切にクリーンアップし、これから回復する失敗し、未使用およびアイドル状態にリセットされます。
- 不明なノードからの Raft リクエストを受け入れない
 - コンセンサスリーダーは、知らないノードからのいかなる要求を受け入れてはいけません。
- 未知のノードの進行状況についてコンセンサスプロトコルをネゴシエートしないでください (RT69779) ノードが強制的にドロップされても、まだ進行状況メッセージを受信する場合がありますそれ。これは、そのようなメッセージを優雅に無視する必要があります。そのようなシチュエーションで。

25.7.4 その他の変更

- サポートされていないコンセンサスプロトコルのコードを削除 (BDR-86)

25.8 BDR 3.7.5

これは、BDR 3.7 のベータリリースです。3.7.4 で特定された問題の新しいメジャー機能と修正の両方が含まれています。

25.8.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

ベータ版ソフトウェアは稼働環境ではサポートされていません-アプリケーションテストのみ

このリリースでは、BDR 3.6.22 からのアップグレードがサポートされています。

25.8.2 改善

- 「コンセンサスプロトコルをサポートする」ログ spam を削減します。(RT69557)
- `node_state` チェックで `bdr.drop_node` を拡張します。(RM19280) 新しい引数「force」を `bdr.drop_node` に追加します。デフォルトは false です。この場合、次の追加チェックが実行されます。`bdr.run_on_all_nodes` を介して、ノードの現在の `node_state` 落とすことが問い合わせられます。分離されるノードが完全ではない場合すべてのノードで分離されたため、これでエラーが発生します。force 引数により、このチェックを無視できます。この機能は、`cascade` が以前に持っていた「強制」動作も削除します。現在、健全性チェック（強制）をスキップオプションと、1 つは依存オブジェクトにカスケードします（カスケード）。
- `pg2q.enable_camo` の廃止 (RM19942、RT69521) パラメータは 3.7 で新しい `bdr.enable_camo` に変更されました。
- 新しいパラメータ `detector_args` を `bdr.alter_table_conflict_detection` に追加 (RT69677) 個々の検出器に追加のパラメーターを許可します。現在、`row_version` に `atttype` を追加するだけで、列のデフォルトの整数だけでなく、`smallint` および `bigint` タイプ。
- `bdr.raft_leadership_transfer` の追加 (RM20159) 特定のノードを Raft リーダーとして昇格させます。Raft の論文によれば、リーダーシップを特定のノードに移すには 次の手順：
 - 現在のリーダーは新しい要求の受け入れを停止します-現在のリーダーはすべての保留中の追加エントリを指定されたリーダーに送信します-現在のリーダーは指定されたリーダーに選挙タイムアウトを強制し、次のリーダーになるためのより良い機会を与えます
 - この機能は、ほぼその概要に従っています。指定されたリーダーだけに追加エントリを送信する代わりに、ハートビートとしても機能するため、すべてのノードに送信します。それは現在のリーダーが指定されたノードにパワーを委任している間、他のノードがタイムアウトしないのを保証にすべきです。また、指定されたノードのステータスを確認し、ノードがアクティブなノードでない場合、または投票権がない場合はリクエストを受け入れません。
- `subscriber-only` ノードの概念を実装するこれらは完全に結合されたノードですが、レプリケーションを送信しませんクラスター内の他の BDR ノードへの変更。しかし、彼らは変更を受け取りますクラスター内のすべてのノードから（もちろん、他のサブスクライバーのみを除くノード）。彼らは

RAFT 投票プロトコルに参加していないため、彼らの存在（または不在）は RAFT リーダーの選挙を決定しません。これらのノードにはレプリケーションスロットを作成する必要はありません。レプリケーションの変更を送信しないでください。同様に、何も作成する必要はありません他の BDR ノード上のこれらのノードのサブスクリプション。これを実装するには、BDR ノードグループの新しいタイプを定義し、「加入者専用」グループ。サブスクライバー専用ノードであると想定されるノードトップレベルの BDR グループではなく、このノードグループに結合必要があります。もちろん、誰かが最初にサブスクライバのみの BDR ノードグループを作成するニーズがあります。の機能は自動的に作成しようとしません。

PostgreSQL 13 の DDL レプリケーションサポートの改善 `ALTER STATISTICS` および `ALTER TYPE ... SET` コマンドがサポートされるようになりました。

25.8.3 解決された問題

- `bdr.drop_node` の安全性チェックを緩和します。（RT69639）同等なでノードが既にドロップされているノード、その同等なを削除するノードのステータスを確認します。それはまだ大丈夫でなければなりませんそのノードをドロップします。
- 削除された自動パーティションルールを再挿入しないでください。1つのノードで自動パーティションルールが削除され、アクションが他のノードが1つを実行する場合、他のノードで複製されるテーブルの保留中のタスクが多くなると、誤ってルールが削除されました。それは私たちが落ちない場所としての問題につながります自動パーティションの依存関係チェックのため、リモートノードのテーブルルールが失敗します。
- `node_summary` および `local_node_summary` ビューの定義を修正（RT69564）基礎となるマルチプルカタログは、ノード、BDR は、ノードと同じ名前付けの1つのみを使用します。これらのビューはそれを反映せず、間違った情報を表示しました-もしノードにマルチプルのインターフェースがあり、`node_summary` ビューにマルチプルが表示されます結果および `local_node_summary` は必ずしも選択しませんどちらかを修正してください。
- `bdr.node_log_config` の修正（RM20318）ビュー `bdr.node_log_config` を調整して、正しく結果競合解決。
- ライタ内のテーブルアクセス統計レポートを修正 これにより、アクセスと I/O を示す PostgreSQL モニタリングビューが修正されます。以前のベータ版で壊れていたテーブルの統計。
- 3.6 からのアップグレード後の `bdr.conflict_history` のパーティショニングを修正以前は 3.6 定義を保持していましたが、今は自動新しい 3.7 インストールと同じ方法でパーティショニングします。
- `snapshot` から初期化されるノードのノード名前の再利用を修正（RM20111）これらのノードは、以前にキャッチアップを引き起こす可能性のある初期状態情報を逃していました結合プロセスのフェーズをスキップされたし、新しいノードが同時に欠落する結果として書き込まれたデータ。これは正しく機能するようになりました。
- Standard Edition でのテーブルの書き換え（`VACUUM FULL`）で発生する可能性のあるクラッシュを修正（EBC-34）Standard Edition でトリガーを確認すると、テーブルの書き換え時にクラッシュする可能性があります以前。
- Standard でノードを削除するときに Enterprise Edition オブジェクトを削除しようとししないでくださいエディション（RM19581）

- 文書言語の改善

25.9 BDR 3.7.4

これは、BDR 3.7 のベータリリースです。3.7.3 で特定された問題の新しいメジャー機能と修正の両方が含まれています。

25.9.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

ベータ版ソフトウェアは稼動環境ではサポートされていません-アプリケーションテストのみ

このリリースでは、BDR 3.6.22 からのアップグレードがサポートされています。

25.9.2 改善

PostgreSQL 13 のサポートを追加

- オプションの `committed` 引数で `bdr.get_node_sub_receive_lsn` を拡張デフォルトの動作が結果され、最後の適用（フィルタリング）するコミット済みトランザクションの LSN を受信しました。関数のオリジナルの意図と使用です（例：HARP による）。`false` を渡すと、この関数はフィルタリングされていないほとんどを結果します。以前のバージョンの動作とマッチング、最近受信した LSN。これ変更は `bdr.wait_for_apply_queue` のハングに関連しています以下を参照してください。
- INCREMENT BY が `galloc` チャンクレンジを超えている場合のエラー (RM18519) `smallint`, `int`, `bigint` の `galloc` シーケンスは、1000、1000000、各チャンクにそれぞれ 1000000000 個の値が割り当てられます。場合エラー INCREMENT 値はこれらの範囲を超えています。
- グローバル DML ロックなしでコンストレインを検証するためのサポートを追加 (RM12646) DDL オペレーション `ALTER TABLE ... ADD CONSTRAINT` は、かなり時間がかかる場合がありますバリデーションが実行されるための時間。BDR で許可されるようになりましたバリデーションの延期と `ALTER TABLE ... VALIDATE` の実行 長時間にわたって DML ロックを保持しない制約パートバリデーション期間。
 - 詳細については、文書の「DDL レプリケーション」の章の「制約の追加」セクションを参照してください。
- `ALTER TABLE ... VALIDATE CONSTRAINTS` は完了を待機しますユーザが完了するのを明示的に待つことを期待する代わりにこの DDL オペレーション、BDR は進行状況を確認し、完了を待ちます自動的に。
- 新しい競合の種類 `apply_error_ddl` とリゾルバ `skip_transaction` を追加 (RM19351) DDL レプリケーションによって `ERROR` が発生するトランザクションをスキップために使用できます。例、同じ DDL がマルチプルのノードに手動で適用された場合。

- `bdr.stat_subscription` に新しい統計を追加します (RM18548)
- `nabort`-ライタが取得したアボートの数
- ライタが見たエラーの数 (現在は上記と同じ)
- `nskippedtx`-ライタがスキップした TX の数 (`skip_transaction` 競合リゾルバ)
- `nretries`-ライタが再リスタート/再接続せずに再リトライした回数
- 特にグローバルロックの場合、SystemTAP 統合を改善します。

25.9.3 解決された問題

- `bdr.wait_for_apply_queue` のハングを修正 (RM11416、CAMO にも影響) キープアライブメッセージにより、LSN が前方に移動する可能性があります。そうでなければ静止システム (処理されたトランザクションなし)、これには `bdr.wait_for_apply_queue` がハングする原因になりました。対応する PGL ライタに適用するものであれば何でも `apply_lsn` は `receive_lsn` に到達しません。適切な CAMO クライアント実装は `bdr.logical_transaction_status` を使用します。次に、影響を受けるファンクションを内部的に使用します。したがって、CAMO スイッチ-またはフェイルオーバーもハングにつながる可能性があります。このリリースでは LSN インクリメントを破棄することによるハング 加入者に適用します。
- 分離されたノードにもかかわらず、合意プロトコルバージョンのアップグレードを許可 (RM19041) コンセンサスプロトコルバージョンから既に分離されたノードを除外するそのようなノードはコンセンサスに参加しないため、ネゴシエーションもうプロトコル。中で最新のプロトコルバージョンを保証しますアクティブノードのセットが使用されます。
- `galloc` シーケンスの多数の修正 (RM18519、RM18512) `galloc` シーケンスの「`nextval`」コードには多くの問題がありました。
- 大きな INCREMENT BY 値 (+ve または -ve) が正しく機能していませんでした
- 大きな CACHE 値が適切に処理されませんでした
- 場合によっては MINVAL / MAXVAL が尊重されない問題の核心は、ラージ増分またはキャッシュ呼び出しがマルチプルの RAFT フェッチ呼び出しを `make` 必要があります。これにより、ループリトライコードが発生しましたマルチプル回呼び出されます。ループを追跡するためのさまざまな変数必要な調整。
- CAMO および Eager トランザクションの最後にコミットされた LSN の追跡を修正 (RM13509) `GUC bdr.last_committed_lsn` は標準用にのみ更新されました非同期 BDR トランザクション。CAMO または Eager トランザクション用ではありません。
- `bdr.ddl_epoch` カタログの NULL 値の問題を修正 (RM19046、RM19072) リリース 3.7 では、新しい `epoch_consumed_lsn` 列が追加されました `bdr.ddl_epoch` カタログ。新しい列を追加すると、列が設定されますテーブル内のすべての既存の行の値を NULL にします。しかし、コードは失敗しました NULL 値を適切にハンドルします。これはゴミの読み取りにつながる可能性があります値またはメモリアクセスエラーですらあります。ガベージ値は潜在的にバックエンドがはるかに遠い LSN で待機する可能性があるため、グローバルロックタイムアウトにつながる未来へ。

- これを修正するには、すべての NULL 値を「0/0」LSN に更新しますが、これは LSN の無効な値表現形式です。列は明示的に NOT NULL とマークされ、コードは列の新しい NULL 値を生成しないように修正されています。

BDR 3.6.22 からのアップグレード処理に関する修正サブスクリプションライターと競合ハンドラーを適切に移行します PGLLogical。この情報は BDR 3.6 で使用されていました。確認する `bdr.conflict_history` は、アップグレード後に適切に処理されます。

- コンセンサスリクエストタイムアウト時の JOINING 状態ハンドリングを修正 (RT69076) JOINING 状態ハンドリング中の timeout により、ノードが BDR グループに結合します。リトライロジックは、この状態を正しく処理するようになりました。
- `replication_set_remove_table` への入力を検証する (RT69248, RM19620)
- `ALTER COLUMN TYPE` の欠落した列を適切に処理します (RM19389, RT69114)
 - これが発生したときにクラッシュするのではなく、標準エラーをスローします。
- 競合検索中のタブルロットのメモリハンドリングを修正しました (RM18543) 見つかったタブルが競合ログテーブルにログオンもクラッシュしなくなりました。
- ローカルノードキャッシュの無効化ハンドリングを修正 (RM13821) 以前は、BDR は競合によりノードの作成またはノードのドロップに気付かない場合があります条件、およびユーザバックエンド内の誤った動作を選択します。

25.10 BDR 3.7.3

これは、BDR 3.7 のベータリリースです。3.7.2 で特定された問題の新しいメジャー機能と修正の両方が含まれています。

25.10.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

ベータ版ソフトウェアは稼働環境ではサポートされていません-アプリケーションテストのみ

このリリースでは、3.6 からのアップグレードはまだサポートされていません。

25.10.2 改善

- 並列適用 (RM6503) `pglogical 3.7.3` の新しいインフラストラクチャを使用して、並列のサポートを追加します作家。デフォルトは、同じ `pglogical` 設定オプション (およびしたがって、この機能は現在デフォルトでオフになっています) 並列ライターの数は、グループごとに変更できます `bdr.alter_node_group_config()` 管理の `num_writers` パラメータインタフェース
- `resynchronize_table_from_node()` は生成された列で動作します (RM14876) リモートノードから生成された列を除くすべての列をコピーします生成された列の値をローカルで計算します。

- `resynchronize_table_from_node()` freezes ターゲットノード上のテーブル (RM15987) このファンクションを使用すると、最初にターゲットテーブルが切り捨てられ、次にコピーされます宛先ノードに。このアクティビティはさらにダブルをフリーズします再同期が発生したとき。これにより、大量の WAL アクティビティが回避されます。I/O + WAL に関連するヒントビットが画像に含まれる場合に発生する可能性があります将来、この宛先ノードで。

BDR 拡張機能がインストールされているが、インストールされていないデータベースでの CRDT の使用を許可するノード (RM17470)。BDR 拡張のあるノードで CRDT 値を以前に復元しますが、何もありませんノードは、CRDT データタイプがノード識別子。現在、`InvalidOid` 値を保存することにより修正されています。ノード識別子は利用できません。その後、ノードが BDR に追加される場合クラスタおよび CRDT 値が更新されると、`InvalidOid` は UPDATE オペレーションのパートとして適切なノード識別子。

HARP プロジェクトで使用するための一貫した KV ストアの実装を追加 (RM17825) これはユーザが直接消費するためのものではありませんが、HARP を機能させることができます追加のコンセンサス設定なしで BDR を使用します。

25.10.3 解決された問題

- 「`local_only`」レプリケーションオリジンを再度追加します (RT68021) `bdr_init_physical` を使用すると、リリース 3.6.19 まで存在していたバグ。このリリースでは、欠落している場合は、再作成します。
- `bdr.alter_node_set_log_config()` への `NULL` 引数を適切に処理します (RT68375、RM17994) このファンクションは、これに対する最初の引数がファンクションは `NULL` です。適切なエラーメッセージを提供するように修正されました代わりに。
- `galloc` シーケンスを使用した `MAXVALUE` および `MINVALUE` の修正 (RM14596)
 - 事前に値を取得している間に、制限に達した可能性があります。今私たち
 - 制限に達する前に取得した値のみを使用します。
- オプションで、以前のエポックによってトリガーされたレプリケーションの変更を待機します (RM17594、RM17802) これにより、BDR 全体でのマルチプルの同時 DDL 操作のハンドリングが改善されます。以前はグローバルロックタイムアウトになるグループでしたが、現在はノード間のレプリケーションラグがラージすぎない限り通過することが許可されました。

メンバーシップは、ターゲットノードによってサブスクライブされたレプリケーションセット内の再同期されたテーブル (RM17621) これは、権限のないユーザがテーブルをコピーできないようにオーダーに重要でそれ以外の場合はアクセスする能力がありません。

- 以前の試行が失敗した後、新しいグループ作成要求が機能するようにします (RM17482) 以前は、BDR が有効になるまで、一部の設定で新しいリクエストは常に失敗していました。最初のグループ作成時にノードから完全に削除され、再インストールされた失敗しました。
- 自動パーティション機能が有効な場合、コンセンサスワーカーの CPU 消費を削減使用済み (RM18002)
- 初期データ同期中のメモリーリークを修正しました (RM17668)
- `update_recently_deleted` の競合検出を修正 (RM16471)
 - この競合は 3.7.2 では正しく検出されませんでした。

- `galloc` シーケンスを変更するときにオプションを確認します (RM18301、RT68470) `Galloc` シーケンスは一部の変更を受け入れないため、ユーザに警告します許可されていないオプションが使用されます。
- `bdr_wait_slot_confirm_lsn` がすべてのスロットを待機していることを確認します (RM17478) この関数は、ダウンストリームをチェックするときにスロットの一部をスキップするために使用されましたすべてを複製しました。
- `PART_CATCHUP` ノードの状態ハンドリングの改善 (RM17418) ノードの状態が原因で `PART_CATCHUP` が永久に残る場合を解決ノード間の競合条件。
- 分割されたノードが欠落している場合のコンセンサスプロセスの弾力性を高めるノードの状態を `PARTED` に更新しようとしても失敗しないでください。長く存在します。
- `bdr_init_physical` から `--recovery-conf` 引数を削除 (RM17196) 以前は動作していませんでしたが、PostgreSQL12 にはもうありません `recovery.conf`。

25.10.4 その他の改善

- デフォルトで `bdr.truncate_locking` を有効にするこれは、常に一貫した結果を生成するために `TRUNCATE` 操作に必要です BDR グループで同時 DML が発生している場合。これは以前のベータでは見逃されていました。
- 他のノードで仮想シーケンスレコードを作成する RM16008 `galloc` シーケンスを作成し、同じ値でその値を使用しようとするトランザクションブロック、それはまだ他のノードに存在しないため、他方で「次のシーケンスチャンクをフェッチできませんでした」というエラーで使われるノード。これを解決するには、他のノードに仮想レコードを作成します。
- 文書の言語が大幅に改善されました。

25.11 BDR 3.7.2

これは、BDR 3.7 のベータリリースです。

25.11.1 重要な注意

BDR 3.7 では、いくつかのメジャー新機能と、既存のアプリケーションとの下位互換性に影響を与えるアーキテクチャ上の変更が導入されています。詳細については、[Upgrades](#) を参照してください。

ベータ版ソフトウェアは稼働環境ではサポートされていません-アプリケーションテストのみ

このリリースでは、3.6 からのアップグレードはまだサポートされていません。

25.11.2 BDR 3.7 のハイライト

- 並列適用 レプリケーションを適用する並列ライターの数で構成できますストリーム
- 自動パーティション
 - 詳細については、[AutoPartition](#) を参照してください。
- CREATE TABLE ... AS ステートメントのサポート (RM9696)
 - この機能は、Enterprise Edition でのみサポートされるようになりました。
- 物理的表現を改善するオーダーに BDR サブグループを定義する新しい機能 BDR クラスターの構成。これにより、BDR クラスターが分散している構成も簡素化されます。マルチプルのデータセンターとデータベースのパートのみが複製されます各サブグループとしてのデータセンターには、自動的に新しいデフォルトのレプリケーションがありますそれに割り当てられたセット。
- 競合はデフォルトで `bdr.conflict_history` に記録されます行単位セキュリティを使用してパーティションテーブルにログを記録し、簡単にアプリケーションユーザーの競合へのアクセス。
- 新しい競合タイプ `multiple_unique_conflicts` マルチプルの UNIQUE を含む複雑な競合の解決を許可します INSERT と UPDATE の両方のコンストレイン。
- ビュー `bdr.node_replication_rates` と `bdr.node_estimate` をマージして
 - `bdr.node_replication_rates`、`bdr.node_estimate` は削除されました (RM13523)
- REINDEX コマンドを複製しないでください。メンテナンスコマンドとして扱われます。
- デフォルト設定に対するその他のさまざまな変更

25.11.3 その他の改善

- ノード接続と地理的な説明のためのオプションのモニタリングテーブルディストリビューション
- `bdr.resynchronize_table_from_node` ファンクションを追加 (RM13565、RM14875) このファンクションは、リモートノードからのリレーションを再同期します。これリレーションのグローバル DML ロックを取得し、リレーションを切り捨てますローカルで、リモートノードからデータをコピーします。リレーションは同じ名前と定義を持つ両方のノードに存在します。
- で使用されるファンクション `bdr.trigger_get_origin_node_id` を追加します競合トリガー (RM15105、RT67601) これにより、ユーザーは次のように競合トリガーを定義できます。DML の競合が発生した場合、信頼できるノードが常に勝ちます。
- `bdr.wait_for_apply_queue` を拡張して特定の LSN を待つ (RM11059、RT65827)
- `bdr.last_committed_lsn` によるコミット済み LSN レポートの追加 (RM11059、RT65827)

BDR は接続文字列の URI も受け入れるようになりました (RM14588) また、フォーマット URI “`postgresql://...`” も指定できます。接続文字列。

25.11.4 解決された問題

- `idle_in_transaction_session_timeout` に対する回復力 (RM13649、RT67029、RT67688) `idle_in_transaction_session_timeout` を 0 に設定して、ユーザ設定を回避します接続を閉じ、snapshot を無効にする可能性があります。

BDR WAL メッセージの正しいパース (RT67662) まれに、BDR クラスター全体に複製され、必要な DDL グローバルロックは、「無効なメモリ割り当て要求サイズ」などのエラーを引き起こす可能性がありますまたは「直接の不適切なパースによる「メッセージにデータが不十分です」WAL メッセージ。このような WAL を解析してハンドルようにコードが修正されましたメッセージを正しく。

- マルチプルのサブコマンドを使用した ALTER TABLE のロックを修正 (RM14771) 複数の ALTER TABLE サブコマンドはロックを尊重する必要がありますセット全体の要件。1 つのサブコマンドがロックをニーズとする場合、その後、ALTER TABLE コマンド全体で同様にニーズになります。

25.12 BDR 3.6.19

これは BDR 3.6 のセキュリティおよびメンテナンスリリースであり、さまざまなマイナー機能も含まれています。

25.12.1 解決された問題

- セキュリティ：内部 BDR SQL ステートメントの `search_path` を空に設定します (RM15373) また、内部で使用されるすべての演算子を完全に修飾します。BDR は現在保護されています CVE-2018-1058 で特定された攻撃リスクから、ユーザアプリケーションそこで特定された安全でないコーディング慣行を回避します。詳細については、BDR セキュリティの章を参照してください。
- セキュリティ：BDR 管理機能に必要な特権を上げる (RM15542) スーパーユーザまたは `bdr_superuser` 以外のロールによって実行される場合：
 - `-bdr.alter_table_conflict_detection` にはテーブル所有者がニーズです-`bdr.column_timestamps_enable` にはテーブル所有者がニーズです-`bdr.column_timestamps_disable` にはテーブル所有者がニーズです-`bdr.drop_trigger` にはテーブル所有者がニーズです-`bdr.alter_sequence_set_kind` にはシーケンス所有者がニーズです-`bdr.create_conflict_trigger` にはテーブルの TRIGGER パーミッションと関数の EXECUTE パーミッションがニーズです-`bdr.create_transform_trigger` にはテーブルの TRIGGER パーミッションと関数の EXECUTE パーミッションがニーズです
 - 新しい GUC `bdr.backwards_compatibility` では、以前の動作を必要とする既存のクライアントについて、この新しく導入されたチェックをスキップできます。
- マルチプルのグローバルロックを解放した後にハングする可能性を解決しました (RT67570、RM14994) タイムアウト後にグローバルロックを解除するためのコードパスのバグ-1 の値でバックエンドの PID をオーバーライドすることになり、また `bdr.global_locks` の `waiters` リストに表示されます。これで順番にウェイターリストを不自由にし、最終的に無限につながったループこのリリースでは、オリジナルの原因であるオーバーライドが修正されていますこのハングのとエントリをロック待機リストから正しく削除します。

BDR WAL メッセージの正しいパース (RT67662) まれに、BDR クラスター全体に複製され、必要な DDL グローバルロックは、「無効なメモリ割り当て要求サイズ」などのエラーを引き起こす可能性がありますまたは「直接の不適切なパースによる「メッセージにデータが不十分です」WAL メッセージ。このような WAL を解析してハンドルようにコードが修正されましたメッセージを正しく。

- マルチプルのサブコマンドを使用した ALTER TABLE のロックを修正 (RM14771) 複数の ALTER TABLE サブコマンドはロックを尊重する必要がありますセット全体の要件。1つのサブコマンドがロックをニーズとする場合、その後、ALTER TABLE コマンド全体で同様にニーズになります。
- ALTER TABLE ... ADD COLUMN 回避策の例のバグを修正 (RT67668) 同時実行を回避するために bdr.global_lock_table () が必要な理由を説明する場合、問題を引き起こす変更。
- 物理的スタンバイの昇格後のハングを修正 (RM15728) BDR ノードに昇格された物理的スタンバイがスタートに失敗した可能性があります内部カタログからの古いデータの使用による複製 キャッシュ。
- bdr.trigger_get_type () が自分自身で呼び出されたときのクラッシュを修正 (RM15592) ストリーミングトリガー関クションの外側で bdr.trigger_get_type () を呼び出すと、クラッシュを引き起こす。外部で呼び出されたときに NULL を結果ように関クションを修正しましたストリーミングトリガー関クション。

25.12.2 改善

- bdr.trigger_get_origin_node_id () は優先ノードの解決を許可します (RM15105、RT67601) 一部のお客様には、ノードに基づいて競合を解決する必要がありますそれが変更のソースです。これは信頼できるソースとも呼ばれます。信頼済みサイトまたは AlwaysWins 解決。以前のバージョンの BDR が許可されました2つのノードを持つこれらのメカニズム。この新しい関クションでは、このオプションを使用できます任意の数のノード。例が文書化されています。

BDR は接続文字列で URI を受け入れるようになりました (RM14588) すべての接続文字列で、URI「postgres://...」のフォーマットを使用できるようになりました

- 新しい関クション bdr.resynchronize_table_from_node () (RM13565、RT67666、RT66968) 単一のテーブルを切り捨ててから、選択したノードから再同期できます。グローバル dml ロックを保持しながら。これにより、テーブルを再同期できます次のことを必要とせずに、データの相違またはデータの破損を追跡するノード全体を再生成します。外部キーは削除され、その後再び有効になります。
- 明示的に複製されないトランザクションによって行われた変更のフィルタリングを改善します (RM15557) 以前は bdr.xact_replication = off を使用してトランザクションによって行われた変更または bdr.difference_fix によってトランザクションがリモートに送信されますノード、偽の競合を生成し、労力を浪費します。変更点代わりにソースノードでフィルタリングされ、パフォーマンスが向上します。
- 初期および定期的なトランザクションステータスチェックで非同期 libpq を使用 (RM13504) (EE) CAMO を有効にすると、処理中のトランザクションのステータスが確認されますパートナーノードに対して。これは標準の Postgres 接続を使用します PGL マネージャプロセスをブロックするために使用される libpq を介して。これリリースは、非同期 libpq を使用して PGL を許可するようにロジックを変更します他のタスクを実行するマネージャ (例: Raft メッセージの処理) そのステータスチェックが実行されます。これにより、タイムアウトまたはより応答性の高い PGL マネージャプロセスによるデッドロック。

- 追加のメッセージフィールドは、DDL レプリケーションの問題の診断に役立ちます (RM15292)

BDR ユーザーに必要な特権に関する文書を明確化 (RT67259、RM15533)

25.13 BDR 3.6.18

これは BDR 3.6 のメンテナンスリリースであり、マイナー機能と以前に特定された問題の修正が含まれています。

25.13.1 改善

- `synchronize_structure` オプションを `join_node_group` に追加 (RM14200、RT67243) 新しい `synchronize_structure` オプションは、`'all'` または `'none'`。スキーマ全体を同期するか、DDL をコピーしません。これにより、アプリケーションスキーマのローリングアップグレードを実行できます。ユーザー管理スキーマ (DDL) 変更ステップを使用します。
- `bdr * difference_fix` 関数で事前に作成されたローカルオリジンを使用するようにしました (RM14189) にローカルオリジンを作成するために使用される関数の `bdr * difference_fix` ファミリー競合の修正を実行します。「`bdr_local_only_origin`」ローカルを事前作成します拡張機能作成時のオリジン。これと同じローカルオリジンが上記で使用されています今すぐ機能します。
- `bdr.monitor_group_versions ()` で監視値を調整します (RM14494) `pglogical` バージョンと `bdr` バージョンが異なる場合、`CRITICAL` はレポートされなくなりました。実際には重要ではありません。BDR のエディションが異なる場合、警告をレポートするようになりましたノード間。
- 文書の大幅な書式修正とスペルチェック

25.13.2 解決された問題

- ノードの結合を修正して、`bdr_superuser` 権限のみを使用するようにしました (RM14121、RT67259) これは、の `join_target_dsn` 接続に影響します `join_node_group` ファンクション、でのみ動作するように修正されました接続に使用されるロールの `bdr_superuser` 権限。
- `bdr.show_subscription_status` で `EXEC` を `bdr_real_all_stats` に付与 (RT67360、RM14624) これにより、`bdr_read_all_stats` ロールと `bdr_monitor` ロールの両方が `bdr.subscription_summary` ビュー

BDR タイプを使用してコピーデータ `bdr_init_physical` の失敗を修正 (RM14522) `bdr_init_physical` は `DROP` ではなく `bdr.drop_node ()` を使用するようになりました `EXTENSION`。CRDT などの BDR データ型を使用するすべての列が発生します。静かにテーブルから削除されます。

- ビューが CRDT を参照するために発生する 3.6.17 アップグレードスクリプトの失敗を修正しました (RT67505) アップグレードスクリプトは、テーブルとマットビューでのみ実行されるようになりました。アップグレードの失敗は「エラー：BDR グローバルロックマネージャがまだ初期化されていません」などの偽のエラーを与える
- 非結合サブスクリプションを、スタートアップに `INIT` 状態ではなく `CATCHUP` 状態に設定しますノードの結合中にメタデータを 2 回コピーする、まれではあるが可能性のあるケースを回避します。

BDR カタログが存在しない場合の `galloc` シーケンスの検索を修正。(RT67455、RM14564) これにより、シーケンスのクエリーがエラーをスローする可能性があります `bdr` ライブラリを追加すると、「リレーションのキャッシュ検索に失敗しました…」など `shared_preload_libraries` が BDR 拡張機能は作成されません。

BDR がロードされているが初期化されていない `ALTER TABLE ALTER COLUMN` を許可する (RM14435) BDR 拡張機能はロードされていますが、ローカル BDR ノードは作成されていないため、DDL レプリケーションロジックは、`ALTER TABLE` の実行を引き続き許可するようになりました `ALTER COLUMN` オペレーション。

BDR ノードが作成されていない場合、`LOCK TABLE` ワーニングが表示されない (RM14613) `LOCK TABLE` ステートメントの評価が `bdr.assess_lock_statement` のときに表示されない BDR ノードが作成されるまで、「無視」以外の値に設定されます。

- `consensus_disable` の NULL デリファレンスを防止します (RM14618) `bdr.consensus_disable` は、コンセンサスプロセスが実行中。そうでない場合にセグメンテーション違反を防ぐために修正しますファンクションが呼び出されます。

25.14 BDR 3.6.17

これは BDR 3.6 のメンテナンスリリースであり、マイナー機能と以前に特定された問題の修正が含まれています。

25.14.1 改善

DDL を複製しない場合、`ALTER TABLE ALTER COLUMN TYPE with rewrite` を許可します (EE) (RM13244) 場合によっては、制御された DBA 環境では、変更することが可能ですローリングを採用することにより、暗黙的にキャスト可能なものにカラムのタイプ非複製環境でのこの列のタイプのアップグレードすべてのノードを 1 つずつ。このテーブルで他の同時アクティビティを許可します書き換え中のノード。また、このような `ALTER` コマンドは実行できないことにノートしてくださいトランザクションブロック内。

- 競合ロギング設定ビューを追加 (RM13691、RT66898) 競合に関する情報を表示する `bdr.node_log_config` ビューを追加しますロギング構成。
- 新しいグループモニタリングビューと機能を追加 (RM14014) これらのビューと機能は、BDR インストールの状態をレポートします。BDR グループ内のすべてのノードにわたるレプリケーションスロットとコンセンサス。
- 構成パラメーターに関連する DDL レプリケーションの現在の状態をログに追加しますコンテキスト (RM13637) トラブルシューティングを改善します。

25.14.2 解決された問題

- 参加ノードの既存のスロットをドロップしないでください (RM13310、RT67289、RT66797) これにより、ノードを使用してノードを結合したときに矛盾が発生する可能性があります `bdr_init_physical` は、新しいノードのスロットを事前作成したため結合中に再利用されることになっており、代わりに削除されて再作成されました。これで、スロットが正しく保持され、矛盾がなくなります。
- パートナーノード (EE) が欠落しているにもかかわらず CAMO ノードのリスタートを修正 (RM13899、RT67161) CAMO として設定された BDR ノードの検索エラーを防止しますオリジンノードがドロップされた場合、存在しませんが、CAMO 用にまだ構成されています。
- `bdr.column_timestamps_enable()` (EE) のロックを修正 (RT67150) 同じトランザクションおよびセッションレベルのロックを保持しないでください。そうでない場合、`PREPARE`、CAMO および Eager レプリケーションは、これが使用されているトランザクションでは機能しません。

BDR 競合解決を PGL のみのサブスクリプションに適用しようとししないでください (RT67178) BDR は pglogical サブスクリプションではなく、BDR サブスクリプションでのみアクティブにする必要があります。

- CAMO パートナーに、学んだら最終決定を結果せる (RM13520) オリジンノードがローカルモードに切り替わった場合、CAMO を一時的にドロップします保護、CAMO パートナーが暫定的にすることは可能ですトランザクションを内部でアボートしますが、実際には最終的にコミットします (オリジンノードと同期するため)。以前のリリースでは、これは記録されていくエリーフアクショントランザクションの「中止」結果を結果ます。このリリースでは暫定アボートをオーバーライドする最終コミット決定内部的に (カタログテーブル `bdr.node_pre_commit`)。
- CLCD / CRDT データ型を適切に TOAST 可能にする (EE) (RM13689) CLCD / CRDT データ型は、PLAIN ストレージを使用するものとして定義されました。これはなることができます列が多すぎるテーブルの問題として、またはラージのノードがある場合関与しています。これは、これらのデータ型を使用するように変換することで解決されました拡張ストレージにより、ラージサイズの値が可能になります。
- ノードの昇格中に重複メッセージが受信されないようにします (RM13972) キャッチアップ段階で、結合ソースから結合ノードに透かしを送信します他のすべてのレプリケーションの現在の位置について保証に学習するための結合キャッチアップ中に転送するデータがない場合でも、ノード。それ以外の場合は、プロモーション中に古い `lsns` を要求して受け取ることがありますメッセージを複製し、結合に失敗します。
- 非トランザクション DDL 操作 (EE) の CAMO を自動的に無効にするトランザクションブロック内ではいくつかの DDL 操作は許可されていませんそして、そのように CAMO の保護から合理的に恩恵を受けることはできません提供しています。回避するために、これらの CAMO を自動的に無効にします COMMIT 時の「準備できません」エラー。
- `bdr.move_group_slot_all_nodes` が BDR ノードなしで呼び出されたときのエラーを修正データベースに存在する (RT67245) 作成する前に、将来の BDR マスタの物理的スタンバイを設定できます。BDR ノード。
- CLCD がオンの場合、テーブルに PRIMARY KEY があることを確認します (EE) これは、ユーザがテーブルで CLCD を有効にできないようにする健全性チェックです。PRIMARY KEY を使用すると、このようなテーブルの競合検出が中断されます。

25.15 BDR 3.6.16

BDR 3.6.16 は、BDR 3.6 シリーズの 16 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.15.1 改善

- `bdr.alter_table_conflict_detection()` の追加 (RM13631) このファンクションは、競合検出メソッドを変更するための UI を統合します。個々のテーブル。オリジンベース、`row_version` ベースの選択が可能同じインタフェースを使用した列レベルベース (EE のみ) の競合検出。古い機能は引き続きサポートされますが、考慮される必要があります廃止され、BDR 3.7 で削除されます。
- `bdr.default_conflict_detection` 構成オプションの追加 (RM13631) 上記の `bdr.alter_table_conflict_detection()` ファンクションに関連して、新しい構成オプションにより、デフォルトの競合検出を設定できます新しく作成されたテーブルのメソッド。
- 強制パートノードの動作を変更する (RM13447) 強制ノードパートは、最初に別れのコンセンサスを得ようとしています。コンセンサスが失敗した場合、またはノードに対して呼び出された場合、ローカル変更を行いますコンセンサスベースのパートプロセスをすでに開始しているが、プロセスが停止している手順の 1 つ。
- `bdr` をドロップすると、`bdr-enterprise` 拡張機能が自動的にドロップされます拡張 (RM13703) これにより、`bdr` 拡張機能をドロップせずに使用する際の使いやすさが向上します。カスケード。ユーザが `pglogical` を保持したい場合例に便利です。BDR に関連付けられたノード。
- 既存のアクティブと同じ名前のノードに参加する際のエラー報告を改善ノード (RM13447、RT66940) 前のエラーメッセージは、BDR のように見えるため、混乱を招きましたノード名前の再利用を許可しません (これは行います)。
- `bdr_init_physical` に `application_name` を設定しますこのツールで問題を診断するときに役立ちます。
- `ALTER TABLE` の制限文書の改善 (RM13512、RM13244、RT66940) `varchar` 列の長さを変更するための新しい回避策を含む。

25.15.2 解決された問題

BDR `galloc` シーケンスの `pg_dump` を修正 (RM13462、RT67051) `Galloc` シーケンスは、シーケンスヒープに追加データを内部的に格納します。BDR が非表示になりました `SELECT` からの追加データ。これにより、シーケンスに対するクエリ (通常のユーザクエリまたは `pg_dump` からのクエリ例) は、通常のシーケンス情報。

- CLCD の `REPLICA IDENTITY FULL` の強制を修正
 - 高度な競合処理アプローチ (CLCD、CRDT) では、
 - レプリカ ID が完全です。しかし、機能が最初にどのように
 - 独立して進化したが、これは適切に実施 (および文書化) されなかった

- そして一貫して。現在、REPLICA IDENTITY FULL を正しく実施しています。
- すべてのテーブルの CLCD。
- 他のノードの結合ソースとして使用されたノードのノード名前の再利用を修正 BDR グループの既存のノード (RM12178、RM13447) ソースノードには特別なハンドリングがあるため、新しく追加 make 必要があります。参加ノードは、分離された同じ名前のノードと混同されません。
- 新しく参加するノードの既存のノードにローカル状態を適用する (RT66940) そうしないと、結合プロセス中の意思決定で誤った状態が使用される可能性があります情報を取得し、スロットの作成やサブスクリプションの作成などのタスクを逃します。
- ノードレベルのログフィルターと競合リゾルバの構成を正しくクリーンアップする (RM13704) これにより、関連するドロップを行わずに BDR ノードをドロップしようとするときの問題が解決します pglogical ノードと後で BDR ノードを再作成します。
- 分離された BDR ノードの Raft でセグメンテーション違反を防ぐ (RM13705)

25.16 BDR 3.6.15

BDR 3.6.15 は、BDR 3.6 シリーズの 15 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.16.1 改善

- 解決されたすべての CAMO 決定の永続的なログを保持する (RM12712) CAMO パートナーが行ったすべての決定を記録する `bdr.logical_transaction_status`、つまりフェイルオーバーの場合。
- 行バージョン追跡を有効/無効にする機能を追加 (RM12930) 列を手動で追加するよりも使いやすく、エラーが発生しにくいインタフェースそしてトリガー。
- `timeshard` および `galloc` の `currval()` および `lastval()` サポートを追加
 - シーケンス (RM12059)
- レート制限の背景に `pglogical.min_worker_backoff_delay` 設定を追加ワーカーの再起動、および `pglogical.worker_tasks` 診断ビューバックグラウンドワーカのアクティビティ。 `pglogical 3.6.15` リリースノートと 詳細については文書。

25.16.2 解決された問題

- `walsender` 内で TOAST 列値をコピーする際のバッファオーバーランを防止出力プラグイン (RT66839) これにより、特定のタイプの `walsender` がクラッシュする問題が修正されます TOAST された列に触れるワークロード。
- `bdr-enterprise` 拡張機能時の「`type bdr.column_timestamps not found`」エラーを修正 `bdr` エンタープライズライブラリが `shared_preload_libraries` にある場合はインストールされません (RT66758、RM13110)

25.17 BDR 3.6.14

BDR 3.6.14 は、BDR 3.6 シリーズの重要なメンテナンスリリースです。このリリースには、CAMO およびその他の機能のメジャー修正と、マイナーな新機能が含まれています。

25.17.1 改善

- `bdr.camo_local_mode_delay` を追加して、CAMO ローカルモードでの調整を許可します (RM12402) CAMO ローカルモードでトランザクションスループットをシンプルに調整します。オリジンノードがより多くのトランザクションを処理するのを防ぐためペアは CAMO を有効にしてハンドルできます。
- `bdr.camo_enable_client_warnings` を追加して、CAMO モードで警告を制御します (RM12558) CAMO の対象となるデータベースでアクティビティが実行されると、警告が発行されますプロパティは保証できません。知識のあるユーザーは、これを無効にすることを選択できますそのような警告がログをいっぱいにしないようにしたい場合。
- 認識されない構成設定に関する警告
- スタートアップメッセージの「BDR のローディング」メッセージを前に移動する
- 行バージョンの競合検出のドキュメントを大幅に強化 (RT66493)
- NOTIFY が CAMO / Eager では不可能であることをドキュメントに明確化
- `global_lock_request_time`、`local_lock_request_time` を追加および `last_state_change_time` 列から `bdr.global_locks` ビューロックのモニタリングと診断の使用。
- コンセンサス snapshot のエクスポート/インポート用の SQL 関数を追加 (RM11433) これらの機能により、BDR システムカタログのマニュアル同期が可能になります。破損またはユーザの間違いの場合。

25.17.2 解決された問題

- `remote_commit_ts` の設定が正しくないため、パートナーノードで UPDATE がスキップされたされました (RM12476) 一部の CAMO メッセージでコミットタイムスタンプが設定されていなかったため、`last-update-wins` が失われていました。彼らが勝ったはずの比較。つまり、一部の UPDATE がスキップされたされたとき、更新は別のマスタから同時に発生しました。これは通常は発生しませんでした、AlwaysOn クラスタでは、パッシブマスタノード経由で書き込みが発生した場合に発生する可能性があります。
- 制御バックエンドがなくなった準備済みトランザクションのみを解決する (RM12388) これにより、`pglogical` マネージャプロセスとユーザバックエンド間の競合条件が修正されます。CAMO トランザクションの実行。マネージャプロセスによる早すぎる解決プリペアドトランザクションたトランザクションは、トランザクションが中断されたとしてマークされる可能性がありますパートナーノード。一方、オリジンはトランザクションをコミットすることになります。これによりデータの相違。この修正により、マネージャプロセスは解決しようとするだけになります。制御ユーザバックエンドが終了したか、またはそうでない準備されたトランザクション CAMO トランザクションをより積

極的に管理します。改訂されたコードは、ProcArrayLock。競合を減らし、パフォーマンスとスループットを改善します。

- CAMO パートナーのコミット決定の時期尚早なクリーンアップを防止します。(RM12540) CAMO または Eager All Node でコミットまたはアボートの決定を維持するようにします次の場合、`bdr.node_pre_commit` のトランザクションは 15 分以上決定を学習していないノードが少なくとも 1 つあり、まだクエリーします。これにより、矛盾の可能性が排除されます CAMO オリジンとパートナーノードの間。
- デッドロックされた CAMO または Eager トランザクションを解決します (RM12903、RM12910) `lock_timeout` とアボートフィードバックをオリジンノードに追加しますプライマリキーの競合による分散デッドロックを解決する更新。これにより、頻繁に再起動したり、Eager All Node および同期 CAMO トランザクション用の PGL ライタプロセス。
- マルチプルノードからのトーストデータの同時更新による潜在的な相違の修正 (RM11058) これは、UPDATE が 1 つ以上のトーストされた列を変更するときに発生しますが、同時ですが、後で別のノードで UPDATE コミットします。これは、PostgreSQL は、UPDATE によって変更されなかった場合、TOAST データを WAL ログに記録しませんコマンド。その結果、論理的にデコードされた行にはこれらの列があります変更されていない TOAST としてマークされ、実際の値が含まれていません。修正が処理されます BDR -EE では自動的に、BDR -SE では追加のトリガーを作成する必要があります更新をパブリッシュし、トースト可能なデータを持つテーブルで (これも行われます自動的に)。追加のチェックのパフォーマンスは小さくても測定可能ですオーバーヘッド。記録されたデータは、影響を受けるケースでのみ増加します。お勧めストレージを最適化するための `toast_tuple_target` のチューニング。REPLICA IDENTITY FULL を含むテーブルは、この問題または修正の影響を受けません。
- 迷惑メールを回避するためにリークパートナーに問い合わせた後、接続を適切に閉じます。(RM12572)
- `bdr.wait_for_apply_queue` を修正して、指定された LSN を尊重します (RM12552) 以前のリリースでは、`target_lsn` 引数がオーバーライドされ、ファンクションは、`target_lsn` が指定されていない場合と同じように動作しました。
- 不明なノードからの進捗メッセージを無視する (RT66461) ノードの分離中の問題を回避します。
- ALTER TABLE とパラレルクエリで `bdr.xact_replication` を動作させる (RM12489)

25.18 BDR 3.6.12

BDR 3.6.12 は、BDR 3.6 シリーズの 12 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.18.1 改善

- DELETE 操作に `check_full_row` を適用する (RT66493) これにより、`delete_recently_updated` 競合の検出が可能になります。DELETE オペレーションは、満杯のテーブルで実時間後に行われました。行チェックが有効になりました。
- グローバル DML ロックトレースの改善グローバル DML ロックトレースに情報を追加して、グローバルデバッグをヘルプします。問題をより効果的にロックします。
- 結合時にレプリケーションセットを検証します。(RM12020、RT66310) 参加ノードが次の場合、`bdr.join_node_group()` からエラーを発生させますを使用してデフォルト以外のレプリケーションセットをサブスクライブするように構成参加前の結合が、サブスクライブ先の一部レプリケーションセットがありません。
 - 以前のリリースでは、結合プロセスの後半で参加ノードが失敗し、強制的に分割する必要がありますがありました。または、成功したように見えても、空のテーブルと結合する場合があります。

25.18.2 解決された問題

- `bdr.run_on_all_nodes` のクラッシュを修正 (RM12114、RT66515) 初期化が正しくないため、`bdr.run_on_all_nodes` は以前は、PARTED ノードの存在下でセグメンテーション違反でクラッシュしていました。
- エポック消費 WAL メッセージをブロードキャストしないでください (RM12042) メッセージをすべてのノードにブロードキャストすると、一部のノードが Global DDL Lock Epoch は、まだ安全ではない状況で前進しています。実行時の同時 DML ステートメントに対する保護が低下するグローバル DML ロックを必要とするステートメント。
- 単一のマルチプルの BDR ノードがあるインストールのグローバルロックを修正 PostgreSQL インスタンス グローバルロックは、ロックメッセージ単一のノードに複数の BDR ノードがある場合、誤ったノード ID が含まれていた PostgreSQL インスタンス。
- ドキュメントのいくつかの例 SQL のタイプミスを修正

25.19 BDR 3.6.11

BDR 3.6.11 は、BDR 3.6 シリーズの 11 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.19.1 改善

PostgreSQL 11.6 のサポート API

- ノード名前に「-」（ハイフン）文字の使用を許可する（RM11567、RT65945）ノード名前にハイフンが含まれる pglogical3 ノードが作成された場合 BDR はそのデータベースにノードを作成できませんでした。
- 更新ノードがわかっている場合、`update_origin_change` の競合を生成しない最新のローカル変更を確認しました（RM11556、RT66145）競合解決のオーバーヘッドと `update_origin_change` のロギングを削減します競合が偽陽性であることを示すことができる場合。これは完全ではありません `update_origin_change` から誤検出を削除しますが、古い行に UPDATES がある場合に発生します。これにより、古い行のオリジンが変更された場合の競合ログ spam が減少します。また、競合トリガーの呼び出し回数は大幅に少なくなります。
- `bdr.wait_for_apply_queue` を拡張して特定の LSN を待つ（RM11059、RT65827）
- 新しいパラメータ `bdr.last_committed_lsn` を追加（RM11059、RT65827）各 COMMIT の後に値がクライアントに報告され、アプリケーションが許可されますマルチプルのノードで因果的読み取りを実行します。
- 適用および受信した LSN のステータスクエリー機能を追加（RM11059、RM11664）新しい関数 `bdr.get_node_sub_receive_lsn` および `bdr.get_node_sub_apply_lsn` は内部の取得を簡素化します HAproxy ヘルスチェックスクリプトに必要な情報。
- CRDT タイプの総和（）および avg（）集約を追加（RM11592、RT66168）
- 物理的スタンバイのレプリケーションスロットの初期同期を高速化（RM6747）
- `bdr.pg_xact_origin` ファンクションを追加して、xid のオリジンをリクエストします（RM11971）
- TRUNCATE を設定する `bdr.truncate_locking` 構成オプションを追加コマンドのロック動作（RT66326）この設定オプションは、true が TRUNCATE に従うかどうかを決定します `bdr.ddl_locking` 設定は、新しい安全な動作です。（デフォルトの false の場合）ロックは行われません。潜在的に安全でない動作。
- 競合トリガーによる `update_recently_deleted` のコミットタイムスタンプの表示を許可ターゲット行（RM11808、RT66182）

25.19.2 解決された問題

- `column_timestamps` データタイプのハッシュ/equality opclass を追加 (RT66207) `REPLICA IDENTITY FULL` では、タブルのすべての列の比較が必要です。したがって、`column_timestamps` データタイプは等値比較をサポートする必要があります。

BDR-EE の競合ドキュメントを修正 (RT66239、RM9670) BDR3.5 で行われた変更は、競合ドキュメントに正しく反映されませんでした

- 初期同期中に `galloc` シーケンスのプロトコルバージョンをチェックしない (RM11576、RT65660) `galloc` シーケンスが既に存在する場合、`bdr_init_physical` は必要ありませんプロトコルのバージョンを `recheck` してください。
- 遅延ノードでの `galloc` シーケンスチャンクトラッキングの破損を修正 (RM11933、RT66294) コンセンサスが遅れているノードがある場合、チャンク追跡テーブルは異なるノードで分岐し、間違っただけのチャンクが割り当てられたコンセンサスリーダーの変更について。その結果、ノードは既に生成をスタート可能性があります使用されたシーケンス番号。この修正により、テーブルが分岐しないことが保証されます。
- `galloc` シーケンスのローカルチャンク情報の破損を修正 (RM11932、RT66294) すべての場合において、新しいチャンクのリクエスト時にエラーが正しく発生することを確認してください失敗した場合、そうでない場合は、ローカルにシーケンスに偽のチャンクを割り当てる可能性がありますシーケンス番号が重複して生成される可能性があります異なるノード。
- コンセンサスワーカイベントループがメッセージで停止する場合がある問題を修正到達不能または応答しない同等な再ノードしようとした場合のブローカーソケットの読み取り可能性/書き込み可能性チェックをより防御的にすることにより `libpq` 非同期接続確立フェーズ。 (RM11914)
 - この問題は、同等な基盤となるホストがハード障害を起こし、すべての TCP 要求への応答を停止した場合、またはピアのネットワークがタイムリーな `ICMP Destination Unreachable` メッセージを報告する代わりに同等なトラフィックをブラックホール化した場合に発生する可能性が最も高くなります。
 - 影響を受けるノードでの問題の影響により、ストールまたはまったく機能しないコンセンサスが必要な操作が発生します DDL ロックの取得、Eager トランザクションコミット、`bdr.get_consensus_status()` ファンクションの呼び出し、`galloc` シーケンスチャンクアロケーション、リーダー選出、BDR グループスロットの前進。これは、偽のロックタイムアウトエラーまたは BDR グループスロットの遅延の増加としてユーザーに可視された可能性があります。
- グローバルロックと DML の競合条件を修正 (RM12042) DML に対するロック操作の順序付けの不一致を防ぐ 3 つ以上の同時書き込みノード。これにより、適切にマルチプルライタからの同時 DML から TRUNCATE を保護します ノード。
- 透過プロキシ (EE) をサポートするために `local_node_id` のレポートを繰り返します (RM12025、RM12033) CAMO を有効にすると、BDR は `bdr.local_node_id` GUC をレポートします クライアント。HAproxy などの透過的なプロキシを完全にサポートするために、BDR はこれを報告するようになりました `transaction_id` と組み合わせてトランザクションごとに 1 回、クライアントが保証 () から誤った結果を結果ないようにする透過的接続スイッチの欠落によって引き起こされた古いキャッシュのため。
- インスタンスのリスタートまたはクラッシュ後のグローバル DDL および DML ロックのリカバリを

修正 (RM12042) BDR の以前のバージョンは、グローバルに対する書き込みを正しくブロックしない可能性がありますロックの取得後にノードまたは適用ワーカーが再起動した場合はロックします。これにより、保護されたコマンドの場合に、データの相違が生じる可能性がありますデータを同時に変更していました。

- レプリケーション変更のグローバル DDL および DML ロックブロックを修正 (RM12042) BDR の以前のバージョンは、ロックされたサーバーへの変更のレプリケーションを続行します他のノードからのテーブル。これにより、一時的なレプリケーションエラーが発生したり、グローバルを取得したトランザクションが遅延のあるノードにはロックが適用されます。
- 適用ワーカーで取得したグローバルロックのクリーンアップ/リカバリのハングを修正別のノードのグローバルロックを取得した適用ワーカーは、終了時にハングしているロックをリークし、異なるバックエンドによって「盗まれる」可能性があります。これにより、適用ワーカーは同じロックのロック取得を待機する可能性がありますリスタート後は永久に。
- 永久に freezeLimit を保持しない (EE) (RM11783) BDR の Enterprise Edition は、十分な情報を保証にフリーズポイントを保持します競合の解決に常に利用できます。しないでください可用性が失われないように、xid 周回ワーニング制限を超えてフリーズを保持します。バキュームフリーズストームのリスクを回避するために、制限をスムーズに進めることができます。
- `bdr.run_on_all_nodes` の接続を適切に閉じる `bdr.run_on_all_nodes` のときに同等なによってリセットされた接続に関するログ spam を削除します。使用されます。
- CREATE MATERIALIZED VIEW がまだサポートされていないドキュメントを明確にします。(RT66363)

25.20 BDR 3.6.10

BDR 3.6.10 は、BDR 3.6 シリーズの 10 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.20.1 改善

- CAMO の新しいオプションのパフォーマンスモードを追加-remote_write (EE) (RM6749) このリリースでは、CAMO remote_write モードが有効になり、より迅速に CAMO からのコミットメッセージの受信時のフィードバックトランザクションの適用後のみではなく、パートナー。堅牢性のわずかな損失と引き換えに、大幅に向上しパフォーマンス。
- パートナーが追いつくまで (EE) (RM9605 / RT65000 / RT65827) まで CAMO モードへの切り替えを延期する可用性を向上させる非同期モードでは、スイッチは CAMO パートナーに到達できない場合のローカルモード。いつ切り替えて、合理的にできる前に追いつく必要があるかもしれませんオリジンからの新しいトランザクションを確認します。オリジンは現在、CAMO モードへのスイッチを遅らせるためのキャッチアップ時間の推定まだ CAMO パートナーによるタイミングのトランザクションを排除する追いつく。
- 関数 wait_for_apply_queue および wait_for_camo_partner_queue (EE) を追加します既に受信したが現在はトランザクションを待つことができますアプリケーションのキューに入れられました。これらは、古い読み取りを防ぐために使用できます remote_write モードで複製された BDR ノード。

- レプリケーションのモニタリング、特に同等な (EE) のキャッチアップ推定に関する改善 (RM9798) 2つの新しいビュー `bdr.node_replication_rates` および `bdr.node_estimates` を導入同等なのノードからの妥当な推定値を取得するにはこのローカルノードから WAL を適用します。`bdr.node_replication_rates` ビュー発信レプリケーションアクティビティの全体像を提供します `bdr.node_estimates` はキャッチアップに焦点を当てているのに対し、平均適用レート同等なの推定。
- テーブルパーティション (EE) の列レベルの競合解決をサポート (RM10098、RM11310) 新しく作成または接続されたパーティションが CLCD 用に設定されていることを確認してください親テーブルの CLCD が有効になっています。
- 少ないケースでグローバル DML ロックを取得します (RM9609)。現在のトランザクションで作成されたりレシーションをグローバルにロックしないでください。また、テーブルではないリレーション (ビュー例) レプリケーション経由でデータを取得します。

25.20.2 解決された問題

- プライマリキーのパートである列に `external` ストレージパラメータを設定することを禁止します (RM11336)。このような設定では、UPDATE をプライマリキーとして複製できませんでした PostgreSQL によってデコードされません。
- `check_full_tuple = true` の ABA 問題を防止します。(RM10940、RM11233) `bdr.inc_row_version()` トリガーが存在する場合にのみ、行全体のチェックを行います UPDATE の競合を検出する際の ABA の問題を防ぐための今後のテーブル `check_full_tuple` が `true` に設定されている場合、データは変更されませんでした。

25.21 BDR 3.6.9

BDR 3.6.9 は、BDR 3.6 シリーズの 9 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.21.1 改善

25.21.2 解決された問題

- `gsum` および `pnsum` CRDT タイプのクラッシュ MIN/MAX を修正しました (RM11049)
- 別れたノードが他のノードで `bdr.part_node()` を要求できないようにします。(RM10566、RT65591)

25.22 BDR 3.6.8

BDR 3.6.8 は、BDR 3.6 シリーズの 8 番目のマイナーリリースです。このリリースには、重大なデータロスの問題の修正と、以前のリリースで特定された他の問題の修正が含まれています。

25.22.1 改善

- `bdr.triggers` ビューの作成 (EE) (RT65773) (RM10874) テーブル名前に関連するトリガーの詳細、それを使用しているファンクション、どのイベントがトリガーされ、何がトリガータイプ。

25.22.2 解決された問題

- 更新 (EE) を複製するリモートノードでの TOAST データの損失 (RM10820、RT65733) 変換トリガーコードパスのバグが特定されましたトースト列を設定する可能性 (特定の非常に長い値列)、リモートノードに UPDATE を適用する場合は NULL 変換トリガーは使用されていません。BDR-EE のみが影響を受けますテーブルにトーストテーブルが定義されていて、使用していない場合のみ REPLICA IDENTITY FULL。BDR3 SE はこの問題の影響を受けません。LiveCompare は、損傷評価とデータリカバリによって強化されました機能、詳細は別の Tech Alert で提供されます 既知の影響を受けるユーザー。このリリースでは、この問題によるさらなるデータロスを防ぎます。
- CAMO：不注意によるタイムアウト (EE) につながるレースを排除します (RM10721) 競合条件により、CAMO パートナーからのコミット前確認が行われました無視されます。これにより、CAMO 保護の不注意なタイムアウトが発生しましたトランザクションとパフォーマンスの低下との組み合わせ `synchronous_replication_availability` を `async` に設定します。この修正リリース 3.6.7 で導入された問題。
- CAMO：コミット時にトランザクションのキャンセルを適切にハンドル (EE) (RM10741) CAMO で保護されたトランザクションの COMMIT の中止を許可します (詳細ノードのリスタートまたは PANIC を使用した場合よりも優雅に)。ランタイムを有効にする CAMO ペアを `make` の CAMO パートナーとの調整 最後には一貫性を持つます。
- CAMO：ステータスクエリーファンクションで CAMO が有効になっていることを確認します。(EE) (RM10803) `logical_transaction_status` ファンクションを使用すると、CAMO が無効になりました単にクエリーのためではなく、セッション全体のために。に依存 CAMO クライアント (またはその間のプロキシ) がどのようにセッションを使用したか、これ CAMO が誤って無効になる可能性があります。これは修正され、CAMO はこの呼び出しとは無関係に有効のままになりますファンクション
- Eager：古いトランザクションをクリーンアップします。(EE) (RM10595)
 - COMMIT フェーズ中に中断されたトランザクションが確実に消去される
 - 最終的にすべてのノードで起動します。
- VACUUM フリーズ制限を設定する際の TransactionId 比較を修正。これはエラーにつながる可能性があります：コミットされた xmax をフリーズできません xid ラップで短期間、VACUUM が失敗します。(EE) (RT65814、RT66211)

25.23 BDR 3.6.7.1

これは、3.6.7 のホットフィックスリリースです。

25.23.1 解決された問題

- 削除されたオリジンからの偽のトランザクション転送を防止します。(RT65671、RM10605) オリジンの削除後、そのオリジンからトランザクションをフィルタリングします出力プラグインではなく、それらを転送しようとするのではなくオリジン情報。

25.24 BDR 3.6.7

BDR 3.6.7 は、BDR 3.6 シリーズの 7 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.24.1 改善

- CAMO と Eager は、内部で二相コミット (2PC) を使用するように切り替えました。これは内部的な変更であり、デッドロックと 2 つのデータ相違の問題 (以下を参照)。これはトランザクションのオリジンノードに影響するノードローカルな変更排他的であり、BDR 間のネットワークプロトコルには影響しません ノード。CAMO を実行する BDR ノードでは、構成の変更が必要になりました十分な `max_prepared_transactions` を許可します。アップグレードを参照してください詳細。の使用には制約がないことに注意してください PostgreSQL の明示的な 2PC に存在する一時テーブル。
- グローバルに割り当てられたレンジシーケンスを追加 (RM2125) ノード間のコンセンサスを使用し、範囲を割り当てる新しいシーケンスの種類必要に応じて、各シーケンスの個々のノードへのシーケンス番号。smallint、整数、bigint のすべてのシーケンスをサポート (シリアルを含む 列タイプ)。
- Multi-Origin PITR Recovery (EE) の実装 (RM5826) BDR は、すべてまたは一部のレプリケーション起点元の PITR を許可します完全に一貫した視点を提供する特定の時点ノードのすべてのサブセットにわたって。マルチオリジンの場合、WAL をビューしますすべてが 1 つの大きなストリームに混在するマルチプルのストリームを含むストリームストリーム PIT は 1 つだけですが、それは次のように到達しますオリジンごとに別々のポイント。したがって、物理的を使用しますマルチプルの個別のロジカル停止ポイントを使用した WAL リカバリ各オリジン。WAL では 1 つの LSN 「停止ポイント」になりますが、また、1 つのタイムスタンプが一貫して適用されます。「単一オリジン PITR」で行います。
- トランザクションレプリケーション制御用の `bdr.xact_replication` オプションを追加何に似た方法でトランザクション全体のレプリケーションをスキップすることができます `bdr.ddl_replication` は DDL ステートメントに対して実行しますが、すべての変更に影響します INSERT/UPDATE/DELETE を含む。SET LOCAL を介してのみ設定できます。注意して使用してください！

BDR によって作成および管理されるレプリケーションスロットの偶発的なマニュアルドロップを防止

- `bdr.permit_unsafe_commands` オプションを追加して、それ以外の場合は無効にしますコマンド (RM10148) 現在、BDR レプリケーションスロットのマニュアルドロップのチェックをオーバーライドしますエンタープライズ版。
- `bdr.ddl_replication` および `bdr.ddl_locking` を `bdr_superuser` として設定可能 SET コマンドを使用するこれは、以前はラッパー関数を介してのみ可能でした。まだ `bdr.set_ddl_replication()` と `bdr.set_ddl_locking()` 利用できます。
- コンセンサスメッセージングレイヤーのパフォーマンスの改善 (RM10319、RT65396)

25.24.2 解決された問題

- `bdr.drop_node` の追加のメタデータを削除 (RT65393、RM10346) 以前はローカルノード情報の一部を保持していたため、ノード名前の再利用。
- 使用時にノード依存のメタデータを正しく同期する `bdr_init_physical` (RT65221、RM10409) 追加のレプリケーションセットとそれらのテーブルメンバーシップを次のように同期します。`bdr_init_physical` のストリームトリガーおよびシーケンス情報と同様に、ロジカル同期と同様の方法で。
- ネットワークの不具合による CAMO との潜在的なデータの相違を修正 (RM # 10147) これにより、ノード間で発生する可能性のあるデータの不整合が修正されます到達不能または応答しない場合の CAMO ペアの (ただし操作可能) CAMO パートナーと CAMO オリジンの同時クラッシュノード CAMO によって保護されているトランザクションの実行中の COMMIT は、1 つのノードでコミットされましたが、その他、両方のノードが動作して接続された後。
- クロス CAMO ペア間の潜在的なデッドロックを修正しました (RM # 7907) 2 つのノードを対称 CAMO ペアとして構成すると、可能だったペアがデッドロックするために、両方のノードがダウンして再起動した場合、ただし、どちらも CAMO トランザクションを実行中です。
- クラッシュに直面した Eager トランザクションの潜在的なデータの相違を修正 (RM # 9907) Eager トランザクションのオリジンノードがクラッシュした場合最終的なローカルコミットの前に、そのようなトランザクションは終了した可能性があります up はオリジンで中止されましたが、他のすべてのノードでコミットされました。これはオリジンノードでも 2PC を使用して適切に解決することにより修正オリジンノードのリスタート後の実行中の Eager トランザクション。
- 実行中の CAMO トランザクションによる高速シャットダウンの正しいハンドリング (RM # 9556) 処理中の BDR ノードでの Postgres の高速シャットダウン CAMO で保護されたトランザクションをコミットすると、以前は PANIC になりました。これは、機内 CAMO トランザクションを使用して適切に処理されるようになりましたそのノードのリスタート後も適切に回復されます。

25.25 BDR 3.6.6

BDR 3.6.6 は、BDR 3.6 シリーズの 6 番目のマイナーリリースです。このリリースには、マイナーな新機能と以前に特定された問題の修正が含まれています。

25.25.1 改善

- `bdr.drop_node()` を追加 (RM9938) ローカルデータベースからノードメタデータを削除し、再利用できるようにするクラスター内のノード名前の。
- `bdr_init_physical` を BDR に含める-SE (RM9892) ラージノード結合時のパフォーマンスを改善します BDR-EE にはこのツールが含まれていますしばらくの間。

BDR -EE で `bdr_init_physical` ユーティリティを強化 (RM9988) `bdr_init_physical` を変更して、オプションで選択的な `pg_basebackup` のみを使用するようにします全体のバックアップの以前の動作とは対照的に、ターゲットデータベースデータベースクラスター。このアクティビティをより速く完了させ、許可する必要があります不要なデータベースが除外されるため、使用するスペースが少なくなります。

- 切捨て複製トランザクション中に `TRUNCATE` が許可されるようになりました (RM9812)
- 新しい `bdr.global_lock_table()` ファンクション (RM9735)。リレーションのグローバル DML ロックを明示的に取得できます。特に便利同時書き込みで `TRUNCATE` を使用する場合の競合を回避するためトランザクション。
- 新しい競合タイプ `update_pkey_exists` (RM9976) `PRIMARY KEY` が更新された場合の競合解決を許可します変更を適用しているノードに既に存在します。

DDL ロックをスキップしてメッセージをスキップし、ログレベルを下げる以前の動作はあまりにも邪魔でした。

- `bdr.apply_log_summary` を追加 (RM6596) 人間が読み取れる競合タイプを示す `bdr.apply_log` を表示し、内部 ID ではなくリゾルバ文字列。
- `bdr.maximum_clock_skew` と `bdr.maximum_clock_skew_action` を追加します構成オプション (RM9379) ノード間のクロックスキューをチェックし、ワーニングまたは遅延を適用するためクロックスキューが大きすぎる場合。

25.25.2 解決された問題

- CRDT 型演算子をパブリックスキーマから `pg_catalog` に移動します (RT65280、RM10027) 以前は、BDR 演算子はパブリックスキーマにインストールされていたため、より厳しいセキュリティポリシーを実装するサーバーで使います。アクションは不要です。
- サポートされていない Eager Replication コマンドが現在実行されていたかどうかを覚えておいてくださいトランザクションのこれにより、サポートされていないコマンドが実行された状況を防ぐことができます Eager Replication がオフになり、トランザクションの後半で Eager Replication がオンになっています。

- `crdt_pnsum` データタイプの「!」演算子を修正 (RM10156) 列の値をリセットするための演算子ですが、前のバージョンでは、リセットオペレーションはこのタイプでは機能していませんでした。

25.26 BDR 3.6.5

BDR 3.6.5 は、BDR 3.6 シリーズの 5 番目のマイナーリリースです。このリリースには、3.6.4 で特定された問題の修正に加えて、マイナーな新機能が含まれています。

25.26.1 改善

- CAMO の遅延有効化を許可 (RM8886) `bdr.enable_camo` の設定は、トランザクションの ID がすでにある場合でも、コミットの前の時間割り当てられた。
- シンプル比較演算子を使用して比較できるバージョン-2 KSUUID を追加します (RM9662)
- 新しい `delete_recently_updated` 競合タイプ (RM9673 / RT65063) オーダーで到着する DELETE オペレーションによってトリガーされます-DELETE には行の最新のローカル UPDATE よりも古いコミットタイムスタンプ。することができます常に勝利する DELETE のデフォルトポリシーをオーバーライドするために使用されます。
- `bdr` 管理関数機能のレプリケーションを DDL レプリケーションフィルターに準拠させる (RT65174) `bdr.replication_set_add_table` のようなコマンドが複製されないように最初に CREATE TABLE を複製しなかったノードに。
- すべてのノードでグループレプリケーションセットを常にサブスクライブする必要はありません (RT65161) DDL レプリケーションフィルターを管理機能に適用するようになったため、グループレプリケーションセットを強制的にすべてのノードによってサブスクライブさせるために必要管理関数を複製するように他のレプリケーションセットを構成できます呼び出します。
- さらにいくつかの DDL 操作でグローバル DML ロックをスキップできるようにする次の DDL 操作は、グローバル DDL ロック、DML ロックではない：

* - ALTER TABLE .. ALTER COLUMN .. SET STATISTICS
--

- -ALTER TABLE .. VALIDATE CONSTRAINT-ALTER TABLE .. CLUSTER ON-ALTER TABLE .. RESET-CREATE TRIGGER
- DELETE で外部キーの異常を解決する新しい BDR トリガーを追加 (RM9580)
- 新しい関数 `bdr.run_on_all_nodes()` を追加して、モニタリングと診断を支援します (RM9945)
- C で CAMO リファレンスクライアントを拡張するテストトランザクションの `bdr.commit_scope` の設定を許可します。
- CLUSTER コマンドのレプリケーションを防止して、運用へのインパクトを回避
- セキュリティおよび一般的な診断を支援するため、スキップする DDL ユーザ要求でのレプリケーションまたはグローバル DDL ロックがログに記録されます。定期的にレプリケートされていないおよび/またはログに記録されていない DDL のユーザーは、これによりログの量を増やすことができま

す。一部のログメッセージのフォーマットが変更されました。この変更は、次の場合に有効になります。bdr.ddl_locking = off および/または bdr.ddl_replication = off。

- (RM8345) に関する BDR 管理機能の説明を大幅に強化運用上のインパクト、レプリケーション、ロック、トランザクションの性質
- 同時主キー更新シナリオを説明する詳細なドキュメント (RM9873 / RT65156)
- bdr.replication_set_add_ddl_filter () の使用例のドキュメント (RM9691)

25.26.2 解決された問題

- レプリケーションセット定義のレプリケーションのリワーク (RT64451) レプリケーションセットが一部のノードから消える問題を解決します。特定の状況で発生する可能性があります。
- これらの DDL コマンドのグローバル DML ロックを取得して、正確性を確保する (RM9650)
- 一意のインデックスを同時に作成
- インデックスの同時削除
- bdr.drop_trigger () 管理ファンクション制約を追加または削除すると、レプリケーションを停止する DML が許可される可能性があるため
- Raft コードで別れている、または別れているノードを正しく無視します (RM9666 / RT64891) ノードメンバーシップの変更の過剰なロギングを削除します。
- update_recently_deleted で CRDT / CLCD マージを試行しないでください (RM9674) 厳密に行レベルの競合です。マージを行うと、間違った結果が生成されます。
- standard_conforming_strings = off を必要とする BDR アプリを許可する (RM9573 / RT64949)
- レプリケーションスロットメタデータを使用して行の凍結を延期する (RM9670) (EE のみ) そうしないと、一定期間後に update_origin_change の競合が検出されない場合がありますノードのダウンタイムまたは切断。SE バージョンでは、パラメーターを使用してのみこれを回避できます。
- bdr_wait_slot_confirm_lsn () を修正して、最後のコミットの LSN を待つ現在の書き込み位置の LSN より。いくつかのケースでは、それが解放された可能性があります適切なよりも早く待機し、他の場合は遅延する可能性があります。

25.27 BDR 3.6.4

BDR 3.6.4 は、BDR 3.6 シリーズの 4 番目のマイナーリリースです。このリリースには、3.6.3 で特定された問題の修正に加えて、マイナーな新機能が含まれています。

25.27.1 BDR 3.6.4 のハイライト

- 統計追跡の適用 (RM9063) 個人のレプリケーションとリソース使用に関する統計を追跡するようになりましたサブスクリプションとリレーションを使用して、`pglogical.stat_subscription` および `pglogical.stat_relation` ビュー。追跡は `pglogical.stat_track_subscription` を介して設定できますおよび `pglogical.stat_track_relation` 構成パラメーター。
- Eager All Node Replication で CAMO クライアントプロトコルをサポートクエリーを拡張して、グローバルコミットスコープで複製されたトランザクションのステータス (Eager All ノード複製)。Eager All Node Replication のサポートを追加 Java CAMO リファレンスクライアント。

25.27.2 解決された問題

- マルチレベルテーブルパーティションの初期データコピーを修正 (RT64809) 単一レベルのパーティショニングのみをサポートするために使用される初期データコピー。マルチプルレベルのパーティショニングがサポートされるようになりました。
- 状況によっては、パーティションの初期データを 2 回コピーしないでください (RT64809) にあるすべてのテーブルからコピーデータしようとするために使用される初期データコピーパーティショニングを適切に考慮しないレプリケーションセット。これにより、ルートパーティションと個別の両方の場合、パーティションデータが 2 回コピーされるレプリケーションセットを介してパーティションが公開されました。これは解決されました。私たちだけルートパーティションが公開されている場合は、最初のコピーを実行します。
- INSERT をパーティションに複製するときのインデックスのハンドリングを修正 (RT64809) すべての状況でインデックスを正しく成約します。
- パーティショニングテストのカバレッジを改善 (RM9311) パーティショニング関連の問題を考慮して、テーブルパーティションに対して行われた自動テスト。
- `crdt_pnsum` データタイプのマージを修正 (RT64975) 内部インデックスが誤って処理されたため、セグメンテーションが発生する可能性があります障害; これは解決されました。BDR 拡張機能がインストールされていないデータベースでキャッシュルックアップが失敗する問題を修正 (RM9217) 以前は、PostgreSQL でテーブルを削除するときにエラーが発生する可能性がありました BDR ライブラリがロードされているが、拡張子がないインスタンスインストール済み。
- `bdr.subscription_summary` のパーミッションの問題を修正 (RT64945) `pglogical.get_sub_progress_timestamp()` に対する権限は必要ありませんこのビューはもう使用しません。
- 障害発生後に準備された Eager All Node トランザクションのクリーンアップ (RM8996) 未完了のトランザクションによる不整合とハングを防止しますノードまたはネットワークの障害後。Raft を使用して一貫性を保証しますこのようなダングリングのクリーンアップのためにノード間で準備トランザクション。

25.27.3 その他の改善

- `replicate_inserts` オプションが初期 COPY に影響するようになりましたテーブルが挿入を複製する場合にのみ、データの初期コピーを行います。
- Raft ワーカー内の内部ノード管理のログレベルを下げる (RT64891) これは、ノードへの結合または分離中に不必要にログをスパミングしていました。DDL レプリケーションまたは DDL ロックなしで DDL を実行するときに警告する DDL レプリケーションまたはロックなしで実行される DDL コマンドは、異なるデータベースとレプリケーションエラーの原因となるため、慎重に警告するそれらについて。
- `dml` ロックなしで統計の作成を許可 (RM9507)
- ドキュメントの正しいデフォルト設定を反映するように文書を変更します `update_missing` 競合タイプ。

25.28 BDR 3.6.3

BDR 3.6.3 は、BDR 3.6 シリーズの 3 番目のマイナーリリースです。このリリースには、3.6.2 で特定された問題の修正に加えて、マイナーな新機能が含まれています。

25.28.1 BDR 3.6.3 のハイライト

- CRDT タイプの `btree` /ハッシュ演算子クラスを追加 (EE、RT64319) これにより、CRDT 列にインデックスを構築できます (スカラ値を使用) そして、シンプル等式/不等式句を使用してそれらをクエリし、`in GROUP BY` 句など。
- CRDT 総和タイプ (EE、RT64600) の `int4` / `int8` からの暗黙的なキャストの追加整数型と CRDT 総和型の式と一緒に使用して入力できるようにします。例：

25.28.2 解決された問題

- グローバル DML ロックにグループロックを使用する (RT64404) これにより、グローバル DML ロッカーとライター間の連携が向上します。残りの変更に追いついています。

25.28.3 その他の改善

- レガシー CRDT タイプと `bdr` スキーマにある新しい CRDT タイプの混在使用をサポート暗黙的に 2 つの間をキャストするため、混在使用と潜在的な移行 透明です。
- 静的コードスキャンの改善すべてのビルドは、Coverity と Clang スキャンビルドの両方でスキャンされます。
- `bdr.ddl_replication` および `bdr.ddl_locking` の変更のログ異なる DDL が実行されたときのトラブルシューティングに役立ちます。文書と PDF 文書。これは主にドキュメントの構造化方法を変更しますチャプターリストの単一のソースと順序付けのために PDF と HTML の両方のドキュメント。

25.29 BDR 3.6.2

BDR 3.6.2 は、BDR 3.6 シリーズの 2 番目のマイナーリリースです。このリリースには、マイナーな新機能と 3.6.1 で特定された問題の修正が含まれています。

25.29.1 BDR 3.6.2 のハイライト

- すべての SQL 可視インターフェースが `bdr` スキーマ (EE) に移動しました CRDT タイプと列ごとの競合解決インターフェースは、`bdr_crdt` および `bdr_conflicts` ではなく、`bdr` スキーマ。タイプとパブリックインターフェースは、既存のスキーマとの互換性のためにそれらのスキーマにまだ存在していますインストール、ただし、現在は非推奨となっているため、使用は推奨されませんまた、将来のリリースで削除される可能性があります。`bdr` スキーマにあるものを使用してください。ドキュメントには現在、`bdr` スキーマへの参照のみが含まれています。
- `bdr.node_conflict_resolvers` ビューを追加 (RT64388) ローカルの競合タイプごとに現在の競合リゾルバ設定を表示しますデフォルトを含むノード。
- C および Java の CAMO リファレンスクライアント実装を文書に追加します。
- DEFERRED UNIQUE インデックスをサポートこのリリース以前は、限られた場合にのみ機能していました。

25.29.2 解決された問題

- リーダー選出中のコンセンサスリクエストのタイムアウトを修正 (RT64569) リーダーが不明な場合にタイムアウトが適用されず、即時につながりましたコンセンサスを必要とするアクションの失敗 (グローバル DDL ロック例)。これは解決されました。DDL ロックオペレーション中の障害時のクリーンアップを改善します。これにより、DDL が高速化されます。エラーがカスケードを作成しないように、エラー後にサブシステムのリカバリをロックする効果。
- 管理ファンクションコマンドのレプリケーションを統合 (RT64544) これにより、管理ファンクションのレプリケーションおよびロック動作が行われます。コマンドは、ロジカルスタンバイを含むすべての状況で DDL とよりインラインになります。
- UNIQUE インデックスのカバーをサポート (RT64650) 以前は、UNIQUE インデックスをカバーするとあいまいなエラーが発生する可能性があります場合によってはメッセージ。
- いかだタイミングの単調なタイムソースへの切り替え (RM6390) これにより、時間のジャンプがある場合の Raft 内部タイミングの信頼性が向上します。NTPd などが原因です。その結果、一般的にいかなだの信頼性が向上します。
- 内部コネクションプーラのロックを改善ノード間のメッセージングの信頼性を高めるため。
- リスタートにコンセンサスプロトコルバージョンを復元する (RT64526) これにより、コンセンサスワーカーまたはノードが毎回再交渉する必要がなくなります。再起動され、機能が一貫して新しいプロトコルバージョンに依存するようになります再起動後も利用可能です。
- 使用時の `bdr.enable_camo` の自動無効化と再有効化を修正トランザクションの DDL。同じ内で手動で再度有効にできないことを確認してくださいトランザクションの

- オリジンが始まる前に早く到着する CAMO 確認のハンドリングを修正待つ。これにより、このような確認が無視されることによるタイムアウトが防止されます。

25.30 BDR 3.6.1

BDR 3.6.1 は、BDR 3.6 シリーズの最初のマイナーリリースです。このリリースには、3.6.0.1 および 3.6.0.2 のすべての修正を含む、マイナーな新機能と修正が含まれています。

25.30.1 3.6.1 のハイライト

- `bdr.role_replication` 構成オプションの追加 (RT64330) 新しいオプションは、ロール管理ステートメント (`CREATE/ALTER/DROP/GRANT ROLE`) のレプリケーションを制御します。ロール管理ステートメントはまだ DDL レプリケーションの標準ルールに従っているため、このオプションは `bdr.ddl_replication` に依存しています。デフォルトでは、これは `on` に設定されています。つまり、これらのステートメントは、BDR 対応データベースで実行されると複製されます。
- `--standby` オプションを `bdr_init_physical` に追加 (RM8543、EE) `bdr_init_physical` を使用してロジカルスタンバイを作成できます。以前は、この方法では完全な送信/受信ノードのみを作成できました。
- `last_xact_replay_timestamp` を `bdr.subscription_summary` に追加 (RT63881) サブスクリプションによって最後に再生されたトランザクションのコミットタイムスタンプを示します。
- 回復不能なエラーで結合を停止する (RT64463) 構造の同期中に結合が失敗する可能性があります、現在は回復不能なエラーです。他の (一時的な) エラーのように再試行する代わりに、パートノードを分離して、エラーがあったことをユーザに通知します。

25.30.2 解決された問題

- トリガーセキュリティチェックの改善 (RT64412) トリガーが `bdr` または `pglogical` トリガー関数、セキュリティ定義関数 (とにかくセキュリティを再定義する関数) を使用する場合、トリガーがテーブルとは異なる所有者を持つことを許可し、ノード結合中の初期レプリケーションセット同期中にレプリケーションセットメンバーシップの変更を常に許可します。BDR の複製されたコマンドを `bdr.ddl_replication` に従う (RT64479) いくつかの BDR ファンクション呼び出し (`bdr.conflicts.column_timestamps_enable` など) は、必要に応じて DDL ロックを含む通常の DDL コマンドと同様の方法で複製されます。ただし、BDR の以前のバージョンのこれらのコマンドは、`bdr.ddl_replication` 設定を無視し、常に複製されていました。これは修正されました。さらに、通常の DDL と同様に、これらのコマンドはロジカルスタンバイから複製されません。
- グローバルオブジェクトで汎用コマンドを複製しようとしないでくださいグローバルオブジェクトのいくつかのコマンドは、それらが内部的にどのように表現されているかによって、本来あるべきではない状況でも複製されます。次のコマンドの処理が修正されました。
 - `ALTER ROLE/DATABASE/TABLESPACE ... RENAME TO`
 - `ALTER DATABASE/TABLESPACE ... OWNER TO`

- COMMENT ON ROLE/DATABASE/TABLESPACE
- SECURITY LABEL ON ROLE/DATABASE/TABLESPACE
- CAMO パートナーで適切にタイムアウトし、ローカルモードにスイッチます (RT64390、EE) 他の BDR ノードの接続ステータスを無視し、指定された CAMO パートナーノードに障害が発生したらすぐにローカルモードにスイッチます。4 ノード以上のクラスターでローカルモードへのスイッチを機能させます。

25.31 BDR 3.6.0.2

BDR 3.6.0.2 リリースは、BDR 3.6 シリーズの 2 番目のバグ修正リリースです。

25.31.1 解決された問題

- 最初の DDL での CAMO の動的な無効化 (EE、RT64403)
- Raftsnapshot を復元する際のタイミングの問題が原因でノード結合がハングする問題を修正 (RT64433)
- トリガーファンクションの所有権チェックを修正 (RT64412)
- `wait_for_completion := false` で `promote_node` および `join_node_group` の動作を改善

25.32 BDR 3.6.0.1

BDR 3.6.0.1 は、BDR 3.6 シリーズの最初のバグ修正リリースです。

25.32.1 解決された問題

- 透過パーティショニング (EE) の `target_table_missing` の競合をサポート (RT64389)
- メッセージブローカーがメッセージを破棄することがある問題を修正 (一般的な副作用は DDL ロックタイムアウトです)
- ラフトプロトコルネゴシエーションの改善
- トレースコードのメモリーリークを修正
- 同期的 `remote_write` レプリケーションパフォーマンスの改善 (RT64397)
- CLCD (EE) のコミットタイムスタンプバリエーションハンドリングを修正
- バイナリプロトコルのサポートを再追加
- `synchronous_replication_availability = 'async'` (EE) を使用した CAMO の正しいローカルモード
- サポートされていない操作を CAMO (EE) と組み合わせて禁止し、ヒントを提供します。

- `logical_transaction_status` のデッドロックを修正 (EE)

25.33 BDR 3.6.0

BDR3 のバージョン 3.6 は、CAMO の改善、パフォーマンスの改善、競合ハンドリングの改善、バグ修正をもたらすメジャーアップデートです。

25.33.1 BDR 3.6 のハイライト

BDR RemoteWrite モードを区別し、`write_lsn` を設定します - レプリケーションパフォーマンスの大幅な改善 - キャッシュテーブルの同期状態 - 必要な場合にのみキープアライブを送信します - 必要な場合にのみフラッシュを行う - wal sender (2ndQPostgres) でより少ないケースでトランザクションをシリアル化します - 物理的ストリーミングレプリケーションがレポートする方法により沿った、レプリケーション位置レポートの改善 - 競合の検出と解決の改善 - 新しいタイプの競合 (`target_table_missing` など) を追加する - 新しいタイプの競合リゾルバーを追加します - ノードおよび競合タイプごとに競合解決を構成可能にします - 更新の競合検出を改善する - CAMO 構成の簡素化 (EE) - CAMO (EE) のパフォーマンスの改善

25.33.2 解決された問題

- リプレイラグの報告を修正 (RT63866)
- トランザクション内の同じ行の繰り返し UPDATE に対する CRDT と競合トリガーを修正します (RT64297)
- 一時インデックスの REINDEX を複製しようとししないでください

25.33.3 その他の改善

- いかだテーブルのバキュームハンドリングの改善
- CAMO 文書 (EE) の改善と明確化

第26章 ‘ Appendix B

Appendix B: Conflict Details

第27章 ‘ Appendix C

Appendix C: Known Issues

第28章 ‘Appendix D

Appendix D: Libraries

第29章 ‘ Appendix E

Appendix E: CAMO Reference Client Implementations