

EDB Postgres Distributed

リリース *5.2.0*

EnterpriseDB Corporation

2023 年 08 月 15 日

目 次

第 1 章 Compatibility matrix **3**

第 2 章 EDB Postgres Distributed Release notes **7**
2.1 Release notes for EDB Postgres Distributed version 5.2.0 . . . 7
2.1.1 EDB Postgres Distributed 5.2 のハイライト . . . 7
2.2 Release notes for EDB Postgres Distributed version 5.1.0 . . . 9
2.2.1 EDB Postgres Distributed 5.1 のハイライト . . . 9
2.3 Release notes for EDB Postgres Distributed version 5.0.1 . . . 12
2.4 Release notes for EDB Postgres Distributed version 5.0.0 . . . 13

第 3 章 Known issues **17**

第 4 章 Introducing PGD quick starts **19**
4.1 クイックスタート . . . 19
4.2 PGD と TPA の紹介 . . . 19
4.3 Docker、Linux ホスト、または AWS クイックスタートの選択 . . . 20
4.3.1 Docker クイックスタート . . . 20
4.3.2 Linux ホストクイックスタート . . . 20
4.3.3 AWS クイックスタート . . . 20
4.4 クラスターでのさらなる探索 . . . 21
4.4.1 Deploying an EDB Postgres Distributed example cluster on Docker . . . 21
4.4.2 Deploying an EDB Postgres Distributed example cluster on Linux hosts . . . 28
4.4.3 Deploying an EDB Postgres Distributed example cluster on AWS . . . 37
4.4.4 Deploying an EDB Postgres Distributed example cluster on Azure and Google cloud platforms . . . 44
4.4.5 Connecting to your database . . . 45
4.4.6 Exploring failover handling with PGD . . . 50
4.4.7 Exploring conflict handling with PGD . . . 59
4.4.8 Next Steps with PGD . . . 64

第 5 章 Concepts **67**
5.1 Terminology . . . 67
5.1.1 非同期レプリケーション . . . 67
5.1.2 可用性 . . . 67
5.1.3 CAMO または commit-at-most-once . . . 67
5.1.4 クラスタリング . . . 67
5.1.5 データシャーディング . . . 68
5.1.6 PGD の Eager Replication . . . 68
5.1.7 結果整合性 . . . 68
5.1.8 フェールオーバー . . . 68
5.1.9 水平スケーリングまたはスケールアウト . . . 68

5.1.10 論理レプリケーション . . . 68
5.1.11 ナイン . . . 69
5.1.12 ノード . . . 69
5.1.13 物理レプリケーション . . . 69
5.1.14 読み取りのスケーラビリティ . . . 69
5.1.15 目標復旧時点（RPO） . . . 69
5.1.16 目標復旧時間（RTO） . . . 69
5.1.17 単一障害点（SPOF） . . . 70
5.1.18 スイッチオーバー . . . 70
5.1.19 同期レプリケーション . . . 70
5.1.20 2 フェーズコミット（2PC） . . . 70
5.1.21 垂直スケーリングまたはスケールアップ . . . 70
5.1.22 書き込みスケーラビリティ . . . 70
5.1.23 リーダーを書く . . . 70
5.2 PGD overview . . . 71
5.2.1 基本的なアーキテクチャ . . . 71
5.2.2 アーキテクチャのオプションとパフォーマンス . . . 73

第 6 章 Planning **77**
6.1 Choosing your architecture . . . 77
6.1.1 標準 EDB Always On アーキテクチャ . . . 77
6.1.2 アーキテクチャの詳細 . . . 78
6.1.3 アーキテクチャの選択 . . . 79
6.1.4 標準アーキテクチャに柔軟性を追加する . . . 80
6.2 Choosing a Postgres distribution . . . 81
6.3 Choosing your deployment method . . . 82
6.4 Other considerations . . . 83
6.4.1 データの一貫性 . . . 83
6.4.2 展開 . . . 83
6.4.3 サイジングの考慮事項 . . . 83
6.4.4 時計とタイムゾーン . . . 84
6.5 Limitations . . . 84
6.5.1 ノード . . . 84
6.5.2 単一インスタンス上の複数のデータベース . . . 84
6.5.3 CAMO . . . 85
6.5.4 その他の制限 . . . 86

第 7 章 Installing **87**
7.1 Deploying EDB Postgres Distributed using Trusted Postgres Architect . . . 87
7.1.1 前提条件：TPA をインストールする . . . 87
7.1.2 構成する . . . 87
7.1.3 規定 . . . 92
7.1.4 展開する . . . 92
7.1.5 テスト . . . 92
7.2 Upgrading . . . 93
7.2.1 アップグレード計画 . . . 93
7.2.2 アップグレードの準備 . . . 95

7.2.3 サーバーソフトウェアのアップグレード . . . 96
7.2.4 アップグレードのチェックと検証 . . . 97
7.2.5 HARP から PGD プロキシへの移行 . . . 97

第 8 章 Using **107**
8.1 Application use . . . 107
8.1.1 アプリケーションの動作 . . . 107
8.1.2 複製されないステートメント . . . 109
8.1.3 DML および DDL レプリケーション . . . 109
8.1.4 異なるリリースレベル間の複製 . . . 110
8.1.5 差分のあるノード間のレプリケーション . . . 110
8.1.6 差分のあるノード同士の比較 . . . 111
8.1.7 アプリケーションの一般的なルール . . . 112
8.1.8 タイミングの考慮事項と同期レプリケーション . . . 112
8.1.9 アプリケーションテスト . . . 113
8.1.10 パフォーマンステストとチューニング . . . 117
8.1.11 PGD でのテーブルアクセスメソッド（TAM）の使用 . . . 118
8.2 Postgres configuration . . . 118
8.2.1 Postgres の設定 . . . 118
8.2.2 時間ベースのスナップショット . . . 119
8.2.3 最大準備トランザクション . . . 120
8.3 Node management . . . 120
8.3.1 PGD グループの作成と参加 . . . 120
8.3.2 Witness ノード . . . 123
8.3.3 ロジカルスタンバイノード . . . 123
8.3.4 フィジカルスタンバイノード . . . 125
8.3.5 サブグループ . . . 127
8.3.6 デコードワーカー . . . 128
8.3.7 ノードの再起動とダウンしたノードの回復 . . . 129
8.3.8 PGD グループからノードを削除する . . . 132
8.3.9 PGD グループ全体を削除する . . . 132
8.3.10 PGD トポロジーのリスト . . . 133
8.4 DDL replication . . . 134
8.4.1 DDL レプリケーションオプション . . . 135
8.4.2 PGD システムで DDL を実行する . . . 136
8.4.3 DDL ロックの詳細 . . . 136
8.4.4 DDL の影響の最小化 . . . 138
8.4.5 ダウンノードでの DDL の処理 . . . 139
8.4.6 ステートメント固有の DDL レプリケーションに関する懸念 . . . 140
8.4.7 PGD DDL コマンド処理マトリックス . . . 140
8.5 Security and roles . . . 156
8.5.1 カタログテーブル . . . 157
8.5.2 PGD 関数と演算子 . . . 157
8.5.3 カタログオブジェクトに対する特権の付与 . . . 157
8.5.4 ロール管理 . . . 158
8.5.5 ロールとレプリケーション . . . 158
8.5.6 接続ロール . . . 159

8.5.7 特権制限 . . . 159
8.5.8 PGD のデフォルト/事前定義されたロール . . . 161
8.5.9 検証 . . . 164
8.6 Sequences . . . 166
8.6.1 PGD グローバルシーケンス . . . 166
8.6.2 UUID、KSUUID、およびその他のアプローチ . . . 173
8.6.3 こちらもご覧ください . . . 175
8.7 Durability and performance options . . . 175
8.7.1 概要 . . . 175
8.7.2 用語と定義 . . . 176
8.7.3 比較 . . . 176
8.7.4 操作の内部タイミング . . . 178
8.7.5 構成 . . . 179
8.7.6 計画的なシャットダウンと再起動 . . . 180
8.7.7 PGD を使用したレガシー同期レプリケーション . . . 180
8.8 Consistency . . . 202
8.8.1 Conflicts . . . 203
8.8.2 autovacuum_freeze_max_age の 90 % . . . 217
8.8.3 Column-level conflict detection . . . 227
8.8.4 Conflict-free replicated data types . . . 233
8.8.5 Eager conflict resolution . . . 245
8.9 Parallel Apply . . . 247
8.9.1 Parallel Apply とは何ですか？ . . . 247
8.10 Replication sets . . . 249
8.10.1 レプリケーションセットの使用 . . . 249
8.10.2 パーティション分割テーブルの動作 . . . 250
8.10.3 外部キーを使用した動作 . . . 250
8.10.4 レプリケーションセットのメンバーシップ . . . 251
8.10.5 レプリケーションセットのリスト . . . 252
8.10.6 DDL レプリケーションのフィルタリング . . . 253
8.11 Application connection management . . . 254
8.11.1 構成 . . . 254
8.12 Backup and recovery . . . 269
8.12.1 バックアップ . . . 269
8.12.2 復元する . . . 273
8.13 Monitoring . . . 274
8.13.1 OpenTelemetry integration . . . 275
8.13.2 Monitoring through SQL . . . 278
8.13.3 SELECT node_group_name, origin_name, sub_enabled, sub_slot_name, . . . 283
8.13.4 subscription_status . . . 283
8.13.5 FROM bdr.subscription_summary; . . . 283
8.13.6 SELECT lock_type、relation、pid FROM bdr.global_locks; . . . 285
8.14 EDB Postgres Distributed Command Line Interface . . . 292
8.14.1 要件 . . . 292
8.14.2 PGD CLI の使用 . . . 292
8.14.3 構成ファイルの指定 . . . 292
8.14.4 データベース接続文字列を渡す . . . 292

8.14.5 出力フォーマットの指定 . . . 293
8.14.6 コマンドラインヘルプへのアクセス . . . 293
8.15 Transaction streaming . . . 328
8.15.1 PGD の機能強化 . . . 328
8.15.2 構成 . . . 329
8.15.3 モニタリング . . . 331
8.16 Stream triggers . . . 331
8.16.1 Apply 中に実行をトリガーする . . . 332
8.16.2 行型の用語 . . . 334
8.16.3 競合トリガー . . . 334
8.16.4 トリガーの変換 . . . 335
8.16.5 行の内容 . . . 336
8.16.6 実行順序 . . . 336
8.16.7 ストリームトリガーの例 . . . 336
8.17 AutoPartition . . . 339
8.17.1 パーティションの自動作成 . . . 339
8.17.2 自動パーティションの例 . . . 340
8.17.3 RANGE パーティションテーブル . . . 340
8.17.4 パーティションの自動作成の停止 . . . 341
8.17.5 パーティションの作成を待っています . . . 341
8.17.6 パーティションを見つける . . . 342
8.17.7 AutoPartitioning の有効化または無効化 . . . 342
8.18 Explicit two-phase commit (2PC) . . . 342
8.18.1 使用する . . . 343
8.19 Timestamp-based snapshots . . . 343

第 9 章 Reference **345**
9.1 PGD Reference . . . 345
9.1.1 ストリームトリガーリファレンス . . . 354

EDB Postgres Distributed（PGD）は、マルチマスターレプリケーションとデータ分散を、高度な競合管理、データ損失保護、およびネイティブ論理レプリケーションよりも最大 5 倍高速なスループットを提供します。これにより、最大ファイブ・ナインの高可用性を備えた分散 Postgres クラスターが有効になります。

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

Group Commit 、*CAMO/failover* オプションを使用した *pgbench* 、または *Eager conflict resolution*

レプリケーションを構成することにより、異なるノード、ノードのグループ、またはすべてのノード間の追加レベルの同期性を構成できます。

第1章　Compatibility matrix

EDB Postgres Distributed	Community PostgreSQL	EDB Postgres Extended Server	EDB Postgres Advanced Server	PGD CLI	PGD Proxy	BDR sion	Exten-	pgLogical3 Ex- tension	HARP
5	12-15	12-15	12-15	5	5	5		n/a	n/a
4	12-14	12-14	12-14	1	n/a	4		n/a	2
3.7	11-13	11r2-13	11-13	n/a	n/a	3.7		3.7	2

- PGD CLI 1 は BDR 4.1 以降でサポートされています。
- HARP 2 の BDR DCS は、 BDR 3.7.15 以降および 4.0.1 以降でサポートされています。

第2章　EDB Postgres Distributed Release notes

EDB Postgres Distributed のドキュメントには、マイナーリリースとパッチを含む EDB Postgres Distributed 5 の最新バージョンが記載されています。リリースノートには、各リリースの新機能に関する情報が記載されています。マイナーリリースまたはパッチリリースで導入された新機能の場合、コンテンツには、その機能が導入されたリリースも示されています。

Release Date	EDB Postgres Distributed	BDR extension	PGD CLI	PGD Proxy
2023 Aug 4	*Release notes for EDB Postgres Distributed version 5.2.0*	5.2.0	5.2.0	5.2.0
2023 May 16	*Release notes for EDB Postgres Distributed version 5.1.0*	5.1.0	5.1.0	5.1.0
2023 Mar 21	*Release notes for EDB Postgres Distributed version 5.0.1*	5.0.0	5.0.1	5.0.1
2023 Feb 21	*Release notes for EDB Postgres Distributed version 5.0.0*	5.0.0	5.0.0	5.0.0

2.1 Release notes for EDB Postgres Distributed version 5.2.0

EDB Postgres Distributed バージョン 5.2.0 は、 EDB Postgres Distributed のマイナーバージョンです。

2.1.1 EDB Postgres Distributed 5.2 のハイライト

- パラレル適用は、PGD の Commit at Most once（CAMO）同期コミットスコープで利用可能になり、レプリケーションパフォーマンスが向上します。
- ネイティブ Postgres へのパラレル適用非同期および同期レプリケーションは、コミットシーケンスを維持し、デッドロックを回避するために複数のトランザクションによって同じキーが同時に変更されているワークロードのために改善されました
- PGD プロキシは HTTP(S) API を追加して、プロキシの状態を直接監視して、準備状況とライブネスを確認できるようにしました。プロキシヘルスチェック を参照

Component	Version	Type	Description
BDR	5.2.0	Feature	CAMO を使用した同期コミットスコープのパラレル適用
BDR	5.2.0	Feature	コミットスコープルールで複数の SYNCHRONOUS_COMMIT 句を許可する

次のページに続く

表 2 – 前のページからの続き

Component	Version	Type	Description
BDR	5.2.0	Enhancement	SYNCHRONOUS_COMMIT および LAG CONTROL コミットスコープを使用したトランザクションストリーミングを許可します
BDR	5.2.0	Enhancement	パラレル適用で同時ワークロードの処理を改善する
BDR	5.2.0	Enhancement	新しい列の `bdr.stat_subscription`を変更します
BDR	5.2.0	Bug fix	外部キー制約違反がある場合、ノードの論理結合を許可します。 (RT91745)
BDR	5.2.0	Bug fix	デッドロックの可能性を回避するために`group_raft_details`ビューに変更します
BDR	5.2.0	Bug fix	拡張機能のアップグレードをログに記録する
BDR	5.2.0	Bug fix	競合するノード名のチェックを追加
BDR	5.2.0	Bug fix	Raft 手動スナップショット復元中のクラッシュを修正
BDR	5.2.0	Bug fix	分割または分割されたノードへのコンセンサス接続を確立しようとしないでください
BDR	5.2.0	Bug fix	正しいユニットを使用するように `tcp_user_timeout` GUC を修正
BDR	5.2.0	Bug fix	PGD 3.7 とのコンセンサススナップショットの互換性を修正します。 (RT93022)
BDR	5.2.0	Bug fix	BDR が pgaudit と一緒に使用されたときのクラッシュに対処します
BDR	5.2.0	Bug fix	監視ノードへの別れの同期をスキップする
BDR	5.2.0	Bug fix	接続文字列に正しいキープアライブパラメーターを生成する
BDR	5.2.0	Bug fix	グループとそのサブグループ間でノードを切り替えるさまざまなシナリオを有効にします。たとえば、グループからネストされたサブグループのいずれかにノードを移行します
BDR	5.2.0	Bug fix	アイドル状態のサーバーでのコンセンサスによって生成される WAL の量を削減します
BDR	5.2.0	Bug fix	`DROP EXTENSION`の同時実行時の自動パーティションカタログのデッドロックを修正しました
BDR	5.2.0	Bug fix	ノードの再起動後に拡張機能をドロップするときの散発的な障害を修正
BDR	5.2.0	Bug fix	pgaudit バグ 212 によるクラッシュの回避策を追加
BDR	5.2.0	Bug fix	コンセンサスとグローバルモニタリングクエリ間のデッドロックを修正
BDR	5.2.0	Bug fix	`bdr.run_on_all_nodes`全体でのクエリキャンセルの伝播を修正
BDR	5.2.0	Bug fix	parted ノードでの `bdr.run_on_nodes()`、*bdr.run_on_group()*、および `bdr.run_on_all_nodes()`の呼び出しを禁止する

次のページに続く

表 2 – 前のページからの続き

Component	Version	Type	Description
CLI	5.2.0	Enhancement	`verify-settings`コマンドに新しい GUCs 検証を追加します
CLI	5.2.0	Bug fix	`verify-settings`の表形式出力で GUC の long 値を切り捨てます
CLI	5.2.0	Bug fix	`sslmode=allow`または `sslmode=prefer`の場合に `connect_timeout`問題を修正するデータベースドライバーライブラリバージョンをアップグレードします
Proxy	5.2.0	Feature	プロキシヘルスチェックに HTTP(S) API を追加
Proxy	5.2.0	Enhancement	ルート変更イベント処理メカニズムを改善する
Proxy	5.2.0	Enhancement	コンセンサスクエリエラー時のリトライメカニズムを追加
Proxy	5.2.0	Bug fix	`sslmode=allow`または `sslmode=prefer`の場合に `connect_timeout`問題を修正するデータベースドライバーライブラリバージョンをアップグレードします

2.2 Release notes for EDB Postgres Distributed version 5.1.0

EDB Postgres Distributed バージョン 5.1.0 は、 EDB Postgres Distributed のマイナーバージョンです。このバージョンは、PGD プロキシおよび PGD CLI の依存関係におけるセキュリティの脆弱性に対処します。

2.2.1 EDB Postgres Distributed 5.1 のハイライト

*** 同期コミット**は、PGD の統一された COMMIT SCOPE 構文で利用できるようになりました。Postgres のレガシー同期コミットオプションをモデルにした PGD 同期コミットにより、DBA はより粒度の細かいコミットと同期管理を利用できます。この追加は、既存の Group Commit 、 CAMO、および Lag Control コミットスコープオプションを補完します。

*** 優先度ベースのプロキシルーティング**の修正により、フェイルオーバーの処理が改善されました。PGD CLI を介してプロキシの猶予期間を構成できるようになり、Raft リーダーを失った場合のプロキシ応答を調整できます。それに伴い、Raft イベントが PGD CLI の `show-events` コマンドで表示され、イベント、ソース、およびサブタイプが表示されます。

*** `bdr.drop_node_group()` **は、PGD SQL インターフェースを使用して空のノードグループを削除するためのサポートを追加します。

Component	Version	Type	Description
BDR	5.1.0	Feature	ライタプロセスが終了したときに出力されるログメッセージに pid を追加しました。
BDR	5.1.0	Feature	bdr.event_summary ビューにグループ名を追加しました。

次のページに続く

表 3 – 前のページからの続き

Component	Version	Type	Description
BDR	5.1.0	Feature	SLES サポートが追加されました。 SLES 15sp4 がサポートされるようになりました。
BDR	5.1.0	Feature	グループサブスクリプションの概要ビューにサブスクリプションステータス列を追加しました。この機能により、 NULL 値がノードがダウンしているか、サブスクリプションが無効であるかを区別できます。
BDR	5.1.0	Feature	Group Commit に NOT グループフィルターを追加しました。この機能により、グループフィルターの意味を逆にして、指定したグループ内のノードを除くすべてのノードを含めることができます。
BDR	5.1.0	Feature	`bdr.drop_node_group`を追加しました。空のノードグループをドロップできるようになりました。
BDR	5.1.0	Feature	スコープをコミットするために SYNCHRONOUS_COMMIT を追加しました。この機能により、レプリケーションの動的 synchronous_commit のような動作が可能になります。
BDR	5.1.0	Feature	`bdr.event_summary`に event_node_name 列を追加しました。
BDR	5.1.0	Feature	イベント履歴に書き込みリーダーの選択を追加しました。event_history カタログ内のグループごとに書き込みリーダーとして選択されたノードに関する情報を追加しました。また、event_history の event_detail で raft インスタンス ID のレポートを改善しました。
BDR	5.1.0	Feature	他の Raft インスタンスのスナップショットのエクスポートおよびインポートを許可する機能を追加しました。この機能により、トップの Raft インスタンスだけでなく他のインスタンスのスナップショットをエクスポートおよびインポートできます。
BDR	5.1.0	Bug fix	コンセンサスプロセスでのメモリリークを修正しました。 (RT91830)
BDR	5.1.0	Bug fix	再参加後にノードがグループと矛盾する可能性がある問題を修正しました。ノードがサブグループの一部であり、別れてからグループに再参加した場合、グループと矛盾する可能性があります。グループの一部のノードからの変更が間違った開始点から再生され、データが失われる可能性があります。
BDR	5.1.0	Bug fix	SDW と standby_slot_names が設定されているときの結合とレプリケーションを修正しました。 (RT89702、RT89536)
BDR	5.1.0	Bug fix	シーケンスオプションを処理するときのスプリアス警告が修正されました。 (RT90668)
BDR	5.1.0	Bug fix	CRDT を使用するノードのアップグレードを修正しました。

次のページに続く

表 3 – 前のページからの続き

Component	Version	Type	Description
BDR	5.1.0	Bug fix	pg_stat_replication からの LCR の調整されたラグトラッキングパラメーター。
BDR	5.1.0	Bug fix	ノードの種類に基づいて調整されたノードルーティングオプションのデフォルト。この変更は、動作ではなく情報の表示にのみ関連します。たとえば、監視ノードは、書き込みを受信する候補としてマークされません。
BDR	5.1.0	Bug fix	すべてのシーケンスは、ノードの作成中に「分散」に変換されるようになりました。
BDR	5.1.0	Bug fix	`standby_slot_names`を使用したストリーミングトランザクションを修正しました。これにより、サブスクライバー専用ノードがデータノードよりも先に進んでいた可能性があります。
BDR	5.1.0	Bug fix	ルーティング選択の優先度が適切に機能するようになりました。以前は、ノードの優先度は、前のリーダーがいない場合にのみ機能していました。これは、フェールオーバー時には当てはまりません。
BDR	5.1.0	Bug fix	最初のノードの結合の記録が完了として修正されました。
BDR	5.1.0	Bug fix	デフォルトでトレースを無効にしました。トレースは初期デバッグでのみ有効であり、5.0 リリースより前に無効にする予定でした。
BDR	5.1.0	Bug fix	pglogical レシーバーのリロード構成のサポートを追加しました。サーバーがリロード信号を受信すると、pglogical レシーバーはリロードして構成の変更を適用します。
BDR	5.1.0	Bug fix	`bdr.monitor_group_raft()`の欠落しているインスタンスのエラーメッセージを改善しました。
BDR	5.1.0	Bug fix	すべての BDR 接続で tcp キープアライブの一貫した使用を実装しました。この変更により、次の GUC が追加されました。`bdr.global_connection_timeout``bdr.global_keepalives``bdr.global_keepalives_>`bdr.global_tcp_user_timeout`デフォルトはかなり保守的な値に設定されており、将来のパッチで調整される可能性があります。
BDR	5.1.0	Bug fix	タイムアウト後、アクティビティがない場合に Raft 接続を閉じました。これは、下で wal_sender_timeout/wal_receiver_timeout を使用します。
BDR	5.1.0	Bug fix	Raft メッセージを受信するバックエンドを簡単に識別可能にしました。Raft バックエンドに関連するログメッセージに情報を追加しました。

次のページに続く

表 3 – 前のページからの続き

Component	Version	Type	Description
BDR	5.1.0	Bug fix	Parallel Apply が高負荷時に遅くなることがある問題を修正しました。
BDR	5.1.0	Enhancement	同期ワーカーの再起動が回避されるようになりました。この修正は、サブスクリプションワーカーの再起動を通知する構成の変更が行われたときに、ノード参加が失敗しないようにすることです。
PGD Proxy	5.1.0	Enhancement	`application_name`は、内部 db 接続の接続文字列でユーザーが設定しなかった場合、プロキシ名に設定されるようになりました。
PGD Proxy	5.1.0	Enhancement	これは、Raft リーダーを失ったときにプロキシが現在の書き込みリーダー（利用可能な場合）にルーティングし続ける期間です。この期間に新しい Raft リーダーが選出されない場合、プロキシはルーティングを停止します。 `0s`に設定されている場合、プロキシはルーティングをすぐに停止します。
PGD Proxy	5.1.0	Bug fix	書き込みリーダーが利用できない場合にブロックすることから、新しいクライアント接続を閉じることに変更されました。
CLI	5.1.0	Enhancement	Raft イベント、イベントソース、およびサブタイプを表示するように`show-events`コマンドを拡張しました。
CLI	5.1.0	Enhancement	`show-clockskew`および`check-health`コマンドのクロックスキュー推定を改善。
CLI	5.1.0	Feature	`consensus_grace_period`プロキシオプションの表示と設定のサポートが追加されました。
Utilities	1.1.0	Bug fix	初期化されていない物理レプリケーションスロットの問題の処理を実装しました。

2.3 Release notes for EDB Postgres Distributed version 5.0.1

EDB Postgres Distributed バージョン 5.0.1 は、EDB Postgres Distributed のパッチバージョンです。このバージョンは、PGD-Proxy および PGD-CLI の依存関係におけるセキュリティの脆弱性を解決します。

Component	Version	Type	Description
CLI	5.0.1	Change	サードパーティの依存関係をアップグレードして Githubdependabot アラートを修正
Proxy	5.0.1	Change	サードパーティの依存関係をアップグレードして Githubdependabot アラートを修正

2.4 Release notes for EDB Postgres Distributed version 5.0.0

EDB Postgres Distributed バージョン 5.0.0 は、 EDB Postgres Distributed の新しいメジャーバージョンです。このバージョンでは、主要な新機能と互換性の変更が行われます。

このリリースのハイライトは次のとおりです。

柔軟な展開アーキテクチャ 強化されたルーティング機能**ユニファイドレプリケーションの耐久性構成** EDB Advanced Storage Pack のサポート *EDBPostgresAdvanced15* **および** *EDBPostgresExtended15* **を使用した** *TDE* **のサポート** OpenTelemetry との統合**トランザクション追跡パフォーマンスの向上（***GroupCommit***、***CAMO***）** Postgres 12〜15 の互換性

Component	Version	Type	Description
PGD	5.0.0	Feature	柔軟な展開アーキテクチャ Always-ON を再定義して、より幅広い展開をサポートしました。
BDR	5.0.0	Feature	強化されたルーティング機能 BDR クラスターは、グループに対してルーティングが有効になっている場合、グループごとの Raft を使用して、すべてのグループ（および関連付けられた場所）の書き込みリーダーを選出します。書き込みリーダーのフェールオーバーを処理し、書き込みリーダーを変更するための SQL コマンドを提供します。
BDR	5.0.0	Feature	EDB Advanced Storage Pack のサポート EDB Advanced Storage Pack は、テーブルアクセスメソッド（TAM）拡張機能の形式で PostgreSQL データベースの高度なストレージオプションを提供します。これらのストレージオプションにより、アプリケーションを変更することなく、データベースのパフォーマンスと信頼性を向上させることができます。
BDR	5.0.0	Feature	ユニファイドレプリケーションの持続性構成 Group Commit、 CAMO、 Eager Replication 、Lag Control などの持続性オプションは、すべてコミットスコープ構成を介して構成されるようになりました。
BDR	5.0.0	Feature	EDB Postgres Advanced および EDB Postgres Extended TDE サポート EDB Postgres Distributed 5 は、EDB Postgres Advanced および EDB Postgres Extended の Transparent Data Encryption 機能を完全にサポートします。
BDR	5.0.0	Feature	OpenTelemetry との統合 BDR 拡張機能は、既存のモニタリングソリューションとのより良い統合のために、モニタリングメトリックとトレースを OpenTelemetry コレクターに送信できるようになりました。

次のページに続く

表 5 – 前のページからの続き

Component	Version	Type	Description
BDR	5.0.0	Feature	Postgres 15 の互換性 EDB Postgres Distributed 5 は Postgres 12〜15 と互換性があります。
BDR	5.0.0	Feature	クラスターイベント管理の改善 `bdr.worker_errors`および `bdr.state_journal_details`ビューは、統合された`bdr.event_summary`に置き換えられました。これには、ローカルノードの Raft ロールの変更も含まれます。将来的には、追加のイベントがそれに追加される可能性があります。
BDR	5.0.0	Change	トランザクション追跡パフォーマンスの向上トランザクション追跡は、`bdr.internal_node_pre_commit`カタログの代わりに共有メモリを使用するようになりました。追加の I / O が発生しないため、パフォーマンスが大幅に向上します。
BDR	5.0.0	Feature	Decoding Worker でデフォルト以外のレプリケーションセットをサポート非対称レプリケーションセットアップのようなデフォルト以外のレプリケーションセットを使用して、クラスターで Decoding Worker 機能を使用できるようにします。
BDR	5.0.0	Feature	Autopartition に HASH パーティショニングのサポートを追加 HASH パーティショニングをサポートするように autopartition / autoscale を拡張します。 RANGE パーティショニングに必要なものの多くは、HASH パーティショニングには必要ありません。たとえば、すべての HASH パーティションを一度に作成することを想定しています (少なくとも現在の作業については。後でこれを変更する可能性があります)。 HASH パーティションが別の表領域に移動したり削除されたりすることは想定していません。したがって、データ保持ポリシーは HASH パーティション分割には適用されません。
BDR	5.0.0	Feature	新しいベンチマークユーティリティ`pgd_bench`を追加しますこのユーティリティは、CAMO トランザクションのベンチマークをサポートしており、将来のリリースでは PGD 特定のワークロードのベンチマークに使用されます。
BDR	5.0.0	Change	ノードにノードの種類が追加されましたこれにより、データ、ウィットネス、サブスクライバー専用、スタンバイなど、さまざまな種類のノードがより適切に区別されます。

次のページに続く

表 5 – 前のページからの続き

Component	Version	Type	Description
BDR	5.0.0	Change	自動パーティションからのタスク管理の分離このリリースでは、自動パーティションワークキューメカニズムがタスクマネージャー（taskmgr）と呼ばれる別のモジュールに移動されました。タスクマネージャーは、新しいタスクを作成し、ローカルノードまたはタスクマネージャーリーダーノードによって作成されたタスクを実行します。したがって、このリリースでは、自動パーティションワーカーワーカーに変更されました。 古い PGD リリースでは、Raft リーダーは新しい作業項目の作成を担当していました。ただし、監視ノードはクラスターオブジェクトの完全なビューがないときに Raft リーダーになることができるため、問題が発生します。このリリースでは、タスクマネージャーリーダーノードの概念を導入しました。ノードは PGD によって自動的に選択されますが、アップグレードされたクラスターの場合、クラスター内のすべてのノードに `node_kind` プロパティを設定することが重要です。ユーザーは、最新の PGD バージョンにアップグレードした後、各ノードの bdr.alter_node_kind() SQL 関数を呼び出して、これを手動で行う必要があります。
BDR	5.0.0	Deprecation	`bdr.assess_lock_statement` および `bdr.assess_update_replica_identity` は非推奨です。
Proxy	5.0.0	Feature	PGD ビルトインプロキシ TCP レイヤー 4、BDR のルーティング機能を使用した PGD クラスターのパススループロキシ。
CLI	5.0.0	Feature	PGD クラスターの検証 CLI は、2 つの新しいコマンド `verify-settings` および `verify-cluster` をサポートします。 `verify-settings` は、PGD クラスター内の各ノードの PostgreSQL 構成を推奨事項に対して検証します。 `verify-cluster` は、柔軟なアーキテクチャ展開の推奨事項に対して PGD クラスターアーキテクチャを検証します。
CLI	5.0.0	Feature	プロキシの管理と構成 `pgd` は、グループごとにプロキシを構成および管理するために、 *create-proxy*、*delete proxy*、*set-group-options*、*set-node-options*、*set-proxy-options*、*show-proxies*、`show-groups` および `switchover` をサポートします。
CLI	5.0.0	Change	`show-camo` コマンドを削除し、`check-health` コマンドから CAMO チェックを削除します。 CLI での `commit scopes` のサポートは、将来のリリースで追加される予定です。
CLI	5.0.0	Change	`show-nodes` および `show-raft` コマンドの出力を変更して、ルーティング機能に対応します。

第3章　Known issues

これらは、 EDB Postgres Distributed 5 の現在の既知の問題です。これらの既知の問題は、PGD のチケットシステムで追跡されており、将来のリリースで解決される予定です。

- update_origin_change 競合のリゾルバーが skip に設定され、synchronous_commit=remote_apply が使用され、同じ行の同時更新が 2 つの異なるノードに繰り返し適用される場合、更新ステートメントの 1 つが PGD ライタとのデッドロックが原因でハングする場合があります。*CRDT データ型を使用した列の競合の処理* で述べたように、 skip は update_origin_change 競合のデフォルトのリゾルバーではありません。この組み合わせは、本番環境での使用を意図したものではありません。そのノードへの到着順序に基づいて、2 つの競合する更新のうちの 1 つを破棄します。これにより、クラスターが発散する可能性があります。skip 競合リゾルバーの使用を選択するまれな状況では、remote_apply モードの使用に関する問題に注意してください。
- Decoding Worker 機能は、CAMO/Eager/Group Commit では動作しません。CAMO/Eager/Group Commit を使用したインストールでは、enable_wal_decoder を無効にしておく必要があります。
- ラグ制御は、完全に分離されたノードで、つまり、他のすべてのノードが到達不能または動作不能である場合、コミット遅延を調整しません。少なくとも 1 つのノードが接続されるとすぐに、レプリケーションラグコントロールが作業を開始し、PGD コミット遅延を再度調整します。
- 時間ベースのラグ制御の場合、PGD は現在、履歴適用率に基づいた推定キャッチアップタイムではなく、ラグタイム（コミットタイムスタンプで測定）を使用します。

CAMO ペアの CAMO パートナーを変更することは現在不可能です。ペアの追加・削除のみ可能です。ペアを追加または削除するには、Postgres の再起動や、構成のリロードさえ必要ありません。

- Group Commit は *CAMO/failover オプションを使用した pgbench* または *Eager All-Node Replication* と組み合わせることはできません。 Eager Replication は現在、グローバルな PGD コミットスコープを使用する場合にのみ機能します。
- Eager Replication を使用するトランザクションはまだ DDL を実行できません。TRUNCATE コマンドは許可されています。

CAMO または Group Commit のいずれかを使用すると、すべての DDL を実行できるわけではありません。

- パラレル適用は現在、Group Commit と組み合わせてサポートされていません。 Group Commit を使用するときは、次のいずれかの方法で無効にしてください。 - GUC bdr.writers_per_subscription を使用。*Configuration of generic replication* を参照してください。
- 現在、コミットスコープの変更または削除に対する保護はありません。同時に変更または削除されているコミットスコープでトランザクションを実行すると、トランザクションを適用しようとしているダウンストリームノードでエラーが発生したため、トランザクションのブロックまたはレプリケーションの完全な停止につながる可能性があります。特定のコミットスコープを使用するトランザクションを変更または削除する前に、それが終了していることを確認してください。

- *PGD CLI* の使用 は、以前にクラスターから分離されたノードにまだ接続している場合、クラスターの状態に関する古いデータを返す可能性があります。*pgd-cli-config.yml* ファイルを編集するか、--dsn 設定を変更して、クラスター内のアクティブノードのみが接続用にリストされるようにします。

他のデザインまたは実装 *CAMO* の制限 の詳細も利用できます。

第4章　Introducing PGD quick starts

4.1 クイックスタート

EDB Postgres Distributed（PGD）は、高いパフォーマンスと可用性を実現するように設計された Postgres のマルチマスターレプリケーション実装です。多数の双方向同期データベースノードで構成されるデータベースクラスターを作成できます。クラスターには、クエリトラフィックを最も利用可能なノードに転送する多数のプロキシサーバーを含めることができ、クラスター構成にさらに回復力を追加します。

- Trusted Postgres Architect（TPA）の簡単な紹介と、 EDB Postgres Distributed の構成、展開、管理にどのように役立つか

Docker、*Linux* **ホスト、または** *AWS* **クイックスタートを選択するためのガイド** Docker クイックスタート *Linux* **ホストクイックスタート** AWS クイックスタート

*アプリケーションをクラスターに接続する

** 競合*フェールオーバーを含むクラスターでのさらなる探索

4.2 PGD と TPA の紹介

PGD は、高いパフォーマンスと可用性を実現するように設計された Postgres のマルチマスター複製実装です。PGD のインストールは、TPA によって調整されます。

さまざまな Postgres 構成のインストールと管理を簡単に繰り返し行えるようにするために、TPA を作成しました。TPA は、Postgres の作成とデプロイを調整します。

これらのクイックスタートは、単一のリージョンクラスターをすばやく取得できるように設計されています。

これらのクイックスタートでは、最初に TPA をインストールします。TPA が既にインストールされている場合は、これらの手順をスキップできます。TPA は、単純なインストーラーというよりもツールです。TPA の同じインストールを使用して、Postgres クラスターのさまざまな構成をデプロイできます。

TPA を使用して、PGD デモクラスターの構成ファイルを生成します。このクラスターには 3 つの複製データベースノードがあり、3 つの高可用性プロキシと 1 つのバックアップノードを共ホスティングします。

次に、TPA を使用して、必要な構成とソフトウェアを各ノードにプロビジョニングおよび展開します。

4.3 Docker、Linux ホスト、または AWS クイックスタートの選択

現在、3 つのクイックスタートが利用できます。

- Docker -単一のマシン上の Docker コンテナーでクラスターをプロビジョニング、デプロイ、およびホストします。
- Linux ホストオペレーティングシステムと SSH 接続で既にプロビジョニングされている Linux サーバーにクラスターをデプロイしてホストします。これらは、オンプレミスまたはクラウドに展開された実際の物理サーバーまたは仮想マシンです。
- AWS - AWS でクラスターをプロビジョニング、デプロイ、およびホストします。

4.3.1 Docker クイックスタート

Docker クイックスタートは、最初に PGD とその機能を探索しようとしている人に最適です。PGD のこの構成は、運用環境での使用には適していませんが、PGD クラスターの機能と動作をテストする場合に役立ちます。また、PGD コマンドと API に精通して、クラウド、VM、または Linux ホストに展開する準備をするときにも役立つ場合があります。

- *Begin the Docker quick start*

4.3.2 Linux ホストクイックスタート

Linux ホストのクイックスタートは、ハードウェアとソフトウェアを完全に制御できる独自のホスト、またはプライベートクラウドに PGD をインストールしようとしている人に適しています。全体的な構成は Docker 構成に似ていますが、システムの再起動後もより永続的であり、PGD のシングルリージョン運用展開に近いです。

- *Begin the Linux host quick start*

4.3.3 AWS クイックスタート

AWS クイックスタートはより広範囲で、Amazon のクラウド上の EC2 ノードに PGD クラスターをデプロイします。クラスターの全体的な構成は、Docker クイックスタートに似ています。ただし、Docker コンテナを使用する代わりに、Amazon EC2 の t3.micro インスタンスを使用してコンピューティング能力を提供します。AWS デプロイはより永続的であり、Docker クイックスタートデプロイの制限の影響を受けません。ただし、AWS CLI を設定するには、より多くの初期セットアップが必要です。

- *Begin the AWS quick start*

4.4 クラスターでのさらなる探索

- *Connect applications to your PGD cluster*
- *Find out how a PGD cluster stands up to downtime of data nodes or proxies*
- *Learn about how EDB Postgres Distributed manages conflicting updates*
- *Moving beyond the quick starts*

4.4.1 Deploying an EDB Postgres Distributed example cluster on Docker

このクイックスタートでは、TPA を使用して、ローカルの Docker コンテナーを使用して Always On Single Location アーキテクチャで PGD をセットアップします。

TPA と PGD の紹介

さまざまな Postgres 構成のインストールと管理を簡単に繰り返し行えるようにするために、TPA を作成しました。TPA は、Postgres の作成とデプロイを調整します。このクイックスタートでは、最初に TPA をインストールします。TPA が既にインストールされている場合は、これらの手順をスキップできます。TPA を使用して、Postgres クラスターのさまざまな構成をデプロイできます。

PGD は、高いパフォーマンスと可用性を実現するように設計された Postgres のマルチマスター複製実装です。PGD のインストールは、TPA によって調整されます。TPA を使用して、PGD デモンストレーションクラスターの構成ファイルを生成します。

このクラスターは、ローカルの Docker コンテナーを使用してクラスターのノードをホストします。3 つのレプリケートデータベースノード、3 つの共ホスト接続プロキシ、および 1 つのバックアップノード。次に、TPA を使用して、必要な構成とソフトウェアを各ノードにプロビジョニングおよび展開できます。

PGD のこの構成は、運用環境での使用には適していませんが、PGD クラスターの機能と動作をテストする場合に役立ちます。また、PGD コマンドと API に精通して、クラウド、VM、または Linux ホストへの展開を準備するときにも役立つ場合があります。

注釈: この一連の手順は、特に Intel / AMD プロセッサー上の Ubuntu 22.04 LTS を対象としています。 !!!

前提条件

この例を完了するには、十分な RAM と空きストレージを備えたシステムが必要です。`curl` と Docker もインストールする必要があります。

RAM 要件

システムには最低 4GB の RAM が必要です。4 つのコンテナを実行し、そのうち 3 つが Postgres データベースをホストするため、これだけの RAM が必要です。

空きディスク領域

この例で説明するクラスターをデプロイするには、Docker からアクセス可能な少なくとも 5GB の無料ストレージが必要です。もう少し多めが賢明でしょう。

curl ユーティリティ

このクイックスタート中に、デフォルトではインストールされない場合がある `curl` ユーティリティを使用して、スクリプトをダウンロードして実行します。`curl` がインストールされていることを確認するには、次を実行します。

```
sudo apt update
sudo apt install curl
```

Docker エンジン

この PGD 展開のターゲットプラットフォームとして Docker コンテナを使用します。Docker エンジンをインストールします。

```
sudo apt update
sudo apt install docker.io
```

```
sudo usermod -aG docker <username>
newgrp docker
```

準備

EDB アカウント

TPA と PGD の両方をインストールするには、EDB アカウントが必要です。

Sign up for a free EDB account まだお持ちでない場合。サインアップすると、EDB のソフトウェアリポジトリへの試用版サブスクリプションが提供されます。

登録したら、

EDB Repos 2.0 ページに移動し、リポジトリトークンを取得できます。

このページに初めてアクセスしたときに、[**RequestAccess**] を選択してリポジトリトークンを生成します。**CopyToken** アイコンを使用してトークンをコピーし、安全に保存します。

環境変数の設定

まず、 EDB_SUBSCRIPTION_TOKEN 環境変数を、

EDB アカウント ステップで取得した EDB リポジトリトークンの値に設定します。

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

これを.bashrc スクリプトまたは同様のシェルプロファイルに追加して、常に設定されるようにすることができます。

リポジトリを構成する

この例に必要なすべてのソフトウェアは、 EDB Postgres 分散パッケージリポジトリから入手できます。次のコマンドは、スクリプトをダウンロードして実行し、EDB Postgres 分散リポジトリを構成します。このリポジトリには、TPA パッケージも含まれています。

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_
↪distributed/setup.deb.sh" | sudo -E bash
```

edb_notranlate_5 出力またはエラーが生成されない場合は、トークンを正しく入力したことを再確認してください。問題が解決しない場合は、 contact Support で支援をお願いします。

Trusted Postgres Architect（TPA）のインストール

TPA を使用して、PGD をプロビジョニングおよびデプロイします。以前に TPA をインストールしたことがある場合は、

Next Steps with PGD に進むことができます。

TPA をインストールするための完全な手順は、 *Trusted Postgres Architect documentation* 、これもここに含まれています。

Linux 環境

ホストプラットフォームとしての *TPA supports several distributions of Linux* 。これらの例は Ubuntu 22.04 用に書かれていますが、サポートされている他のプラットフォームでも手順は似ています。

重要: 使用している Linux ホストプラットフォームで cgroups v2 が実行されている場合は、TPA を使用して Docker にデプロイしながら cgroups v1 を無効にして有効にする必要があります。 cgroup v2 を確認するには、次を実行します。

```
mount | grep cgroup | head -1
```

行頭が表示されない場合**：`tmpfs on /sys/fs/cgroup type tmpfs`

次に、cgroup v2 を無効にする必要があります。これを行うには、次を実行します。

```
echo GRUB_CMDLINE_LINUX=systemd.unified_cgroup_hierarchy=false | sudo tee /etc/
↪default/grub.d/cgroup.cfg
```

次に、次を実行して grub ブートローダーを更新し、再起動します。

```
sudo update-grub
sudo reboot
```

TPA パッケージをインストールする

```
sudo apt install tpaexec
```

TPA の構成

次に、TPA の Python 環境を構成する TPA を構成する必要があります。 setup コマンドで tpaexec を呼び出します。

```
sudo /opt/EDB/TPA/bin/tpaexec setup
export PATH=$PATH:/opt/EDB/TPA/bin
```

export コマンドをシェルのプロファイルに追加できます。

TPA インストールのテスト

selftest を実行して、TPA が正しくインストールされたことを確認できます。

```
  tpaexec selftest
  ``` TPA がインストールされました。

  ##  TPA を使用した PGD のインストール

  ###  構成ファイルの生成

  :ref:`tpaexec configure <Deploying EDB Postgres Distributed using Trusted Postgres␣
↪Architect>` コマンドを実行して、構成フォルダーを生成します。

  ```shell-session
  tpaexec configure democluster \
    --architecture PGD-Always-ON \
    --platform docker \
    --edb-postgres-advanced 15 \
    --redwood \
    --location-names dc1 \
```

(次のページに続く)

(前のページからの続き)

```
    --pgd-proxy-routing global \
    --no-git \
    --hostnames-unsorted

:ref:`PGD 5's Always On architectures <Choosing your architecture>` の構成を設定する
PGD-Always-ON アーキテクチャ（\ ``--architecture PGD-Always-ON``
```

）を指定します。デフォルトアーキテクチャの一部として、3 つのデータノードでクラスターを構成し、3 つの *Installing PGD Proxy* サーバーと
Barman

バックアップ用のノード。

Docker（`--platform docker`）を使用していることを指定します。デフォルトでは、TPA は Rocky Linux をすべてのノードのデフォルトイメージとして構成します。

TPA の実行に使用しているのと同じプラットフォームに PGD を展開する必要はないことに注意してください。

データノードが Oracle 互換性（`--redwood` ）で EDB Postgres Advanced Server v15（`--edb-postgres-advanced 15`）を実行することを指定します。

`--location-names` を使用して、ノードの概念的な場所を `dc1` に設定します。次に、`--pgd-proxy-routing` を `global` に設定して、プロキシルーティングがクラスター内のすべてのノードにトラフィックをルーティングできるようにします。

デフォルトでは、TPA は構成の変更を Git リポジトリにコミットします。この例では、それを行う必要はないため、`--no-git` フラグを渡します。

最後に、 `--hostnames-unsorted` を渡して、ノードの反復可能なホスト名を生成するように TPA に依頼します。それ以外の場合は、事前に定義された適切な単語のリストからホスト名をランダムに選択します。

このコマンドは、現在の作業ディレクトリに `democluster` というサブディレクトリを作成します。TPA がクラスターの作成に使用する `config.yml` 構成ファイルが含まれています。次を使用して表示できます。

```
less democluster/config.yml
```

edb_notranlate_14 - *Deploying with TPA* の PGD-Always-ON 構成オプションの詳細

- Trusted Postgres Architect のドキュメントの *PGD-Always-ON* - Trusted Postgres Architect のドキュメントの *tpaexec configure* - Trusted Postgres Architect のドキュメントの *Docker platform*

クラスターのプロビジョニング

次に、

tpaexec provision コマンドを使用して、作成したばかりの構成を実行するために必要なリソースを割り当てます。

```
tpaexec provision democluster
```

プラットフォームとして Docker を指定したため、TPA は Docker イメージ、コンテナー、ネットワークなどを作成します。

Trusted Postgres Architect ドキュメントの.. SeeAlso "Further reading"::

- *tpaexec provision*

クラスターのデプロイ

構成が整い、インフラストラクチャがプロビジョニングされたら、分散クラスターを *Choosing your deployment method* できます。

```
tpaexec deploy democluster
```

TPA は構成を適用し、必要なパッケージをインストールして、実際の EDB Postgres 分散クラスターをセットアップします。

これで、SSH を使用してクラスターのノードのいずれかにログインし、データベースに接続する準備が整いました。構成プロセスの一部では、キーを使用して、すべてのノードの SSH ログインを設定します。SSH 構成を使用するには、前に `tpaexec configure` コマンドで作成した `democluster` ディレクトリにいる必要があります。

```
cd democluster
```

そこから `ssh -F ssh_config <hostname>` を実行して SSH 接続を確立できます。クラスター内の最初のデータベースノードである kaboom に接続します。

```
ssh -F ssh_config kaboom
__OUTPUT__
[root@kaboom ~]#
```

`kaboom` に `root` としてログインしていることに注意してください。

ここで、enterprisedb ユーザーの ID を採用する必要があります。このユーザーは、クラスターのノードに接続するように事前構成され、承認されています。

```
sudo -iu enterprisedb
__OUTPUT__
enterprisedb@kaboom:~ $
```

`psql` コマンドを実行して、 `bdrdb` データベースにアクセスできます。

```
psql bdrdb
__OUTPUT__
psql (15.2.0, server 15.2.0)
Type "help" for help.

bdrdb=#
```

kaboom ノードで実行されている Postgres データベースに直接接続され、SQL コマンドの発行を開始できます。

SQL クライアントを終了するには、 `exit` と入力します。

PGD CLI の使用

PGD CLI としても知られる pgd ユーティリティを使用すると、 EDB Postgres 分散クラスターを制御および管理できます。すでにノードにインストールされています。

`pgd check-health` を実行して、クラスターの状態を確認するために使用できます。

```
pgd check-health
__OUTPUT__
Check      Status Message
- ----      ------ -------
ClockSkew  Ok     All BDR node pairs have clockskew within permissible limit
Connection Ok     All BDR nodes are accessible
Raft       Ok     Raft Consensus is working correctly
Replslots  Ok     All BDR replication slots are working correctly
Version    Ok     All nodes are running same BDR versions
enterprisedb@kaboom:~ $
```

または、 `pgd show-nodes` を使用して、クラスター内のデータを保持するノードを表示するように PGD に依頼できます。

```
pgd show-nodes
__OUTPUT__
Node   Node ID    Group        Type Current State Target State Status Seq ID
- ---   -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     1
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     2
enterprisedb@kaboom:~ $
```

同様に、 `pgd show-proxies` を使用してプロキシ接続ノードを表示します。

```
pgd show-proxies
__OUTPUT__
```

(次のページに続く)

(前のページからの続き)

```
Proxy  Group       Listen Addresses Listen Port
- ----  -----       ---------------- -----------
kaboom democluster [0.0.0.0]        6432
kaftan democluster [0.0.0.0]        6432
kaolin democluster [0.0.0.0]        6432
```

プロキシは、アプリケーションのデータノードのクラスターへの高可用性接続を提供します。コマンド `psql -h kaboom,kaftan,kaolin -p 6432 bdrdb` を使用して、プロキシに接続し、さらにデータベースに接続できます。

```
psql -h kaboom,kaftan,kaolin -p 6432 bdrdb
__OUTPUT__
psql (15.2.0, server 15.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

bdrdb=#
```

クラスターを探索する

- *Connect to your database* アプリケーションへ
- 実践的な演習を含む *Explore failover*
- *Understand conflicts* 作成して監視することで

*クラスターでの *Next Steps with PGD*

4.4.2 Deploying an EDB Postgres Distributed example cluster on Linux hosts

TPA と PGD の紹介

さまざまな Postgres 構成のインストールと管理を簡単に繰り返し行えるようにするために、TPA を作成しました。TPA は、Postgres の作成とデプロイを調整します。このクイックスタートでは、最初に TPA をインストールします。TPA が既にインストールされている場合は、これらの手順をスキップできます。TPA を使用して、Postgres クラスターのさまざまな構成をデプロイできます。

PGD は、高いパフォーマンスと可用性を実現するように設計された Postgres のマルチマスター複製実装です。PGD のインストールは、TPA によって調整されます。TPA を使用して、PGD デモンストレーションクラスターの構成ファイルを生成します。

TPA Linux ホストオプションを使用すると、任意のクラウドまたは VM プラットフォームのユーザーが TPA を使用して EDB Postgres Distributed を構成できます。TPA から必要なのは、ターゲットシステムを Linux オペレーティングシステムで構成し、SSH を使用してアクセスすることだけです。他の TPA プラッ

トフォーム（Docker および AWS）とは異なり、Linux ホスト構成はターゲットマシンをプロビジョニングしません。展開することを決定した場所にそれらをプロビジョニングするのはあなた次第です。

このクラスターは、Linux サーバーインスタンスを使用してクラスターのノードをホストします。ノードには、3 つの複製データベースノード、3 つの共ホスト接続プロキシ、および 1 つのバックアップノードが含まれます。TPA は、必要な EDB Postgres Distributed ソフトウェアと構成を各ノードにプロビジョニング、準備、デプロイできます。

前提条件

Linux ホストを構成する

このクイックスタートでは、4 つのホストをプロビジョニングする必要があります。各ホストには *supported Linux operating system* がインストールされている必要があります。パスワードのプロンプトをなくすには、各ホストが証明書キーペアを使用して SSH アクセス可能である必要もあります。

どちらのクラウドまたは VM プラットフォームを使用する場合でも、各インスタンスに SSH でアクセスできること、および各インスタンスが他のインスタンスに接続できることを確認する必要があります。パブリックネットワークまたはクラウドプラットフォームの VPC を介して接続できます。オンプレミス VM は、ローカルネットワークを介して接続できます。

これができない場合は、Docker または AWS のクイックスタートを検討することをお勧めします。これらの構成は、セットアップが簡単で、解体が迅速です。たとえば、
Docker、*Linux* ホスト、または *AWS* クイックスタートの選択

は、コンピューティングインスタンスを自動的にプロビジョニングし、それらのインスタンスの VPC を作成します。

Host name	Public IP	Private IP
linuxhost-1	172.19.16.27	192.168.2.247
linuxhost-2	172.19.16.26	192.168.2.41
linuxhost-3	172.19.16.25	192.168.2.254
linuxhost-4	172.19.16.15	192.168.2.30

これらは IP アドレスの例です。クイックスタートを進めながら、自分のパブリック IP アドレスとプライベート IP アドレスに置き換えてください。

ホスト管理ユーザーを設定する

各マシンには、インストールに使用するユーザーアカウントが必要です。簡単にするために、すべてのホストで同じ名前のユーザーを使用します。各ホストで、パスワードの入力を求められずにホストに SSH できるようにユーザーも構成します。そのユーザーにホストで sudo 特権を付与してください。4 つのホストで、ユーザー rocky は sudo 特権ですでに構成されています。

準備

EDB アカウント

TPA と PGD の両方をインストールするには、EDB アカウントが必要です。

Sign up for a free EDB account

まだお持ちでない場合。サインアップすると、EDB のソフトウェアリポジトリへの試用版サブスクリプションが提供されます。

登録したら、
EDB Repos 2.0 ページに移動し、リポジトリトークンを取得できます。

このページに初めてアクセスしたときに、[**RequestAccess**] を選択してリポジトリトークンを生成します。**CopyToken** アイコンを使用してトークンをコピーし、安全に保存します。

環境変数の設定

まず、 `EDB_SUBSCRIPTION_TOKEN` 環境変数を、
EDB アカウント ステップで取得した EDB リポジトリトークンの値に設定します。

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

これを`.bashrc` スクリプトまたは同様のシェルプロファイルに追加して、常に設定されるようにすることができます。

リポジトリを構成する

この例に必要なすべてのソフトウェアは、 EDB Postgres 分散パッケージリポジトリから入手できます。スクリプトをダウンロードして実行し、EDB Postgres 分散リポジトリを構成します。このリポジトリには、TPA パッケージも含まれています。

curl -1sLf "https://downloads.enterprisedb.com/\protect\T1\textdollarEDB_SUBSCRIPTION_TOKEN/postgres_distributed/setup.deb.sh" | sudo -E bash

Trusted Postgres Architect（TPA）のインストール

TPA を使用して、PGD をプロビジョニングおよびデプロイします。以前に TPA をインストールしたことがある場合は、
Next Steps with PGD に進むことができます。

TPA をインストールするための完全な手順は、 *Trusted Postgres Architect documentation* 、これもここに含まれています。

Linux 環境

ホストプラットフォームとしての *TPA supports several distributions of Linux* 。これらの例は Ubuntu 22.04 用に書かれていますが、サポートされている他のプラットフォームでも手順は似ています。

TPA パッケージをインストールする

```
sudo apt install tpaexec
```

TPA の構成

次に、TPA の Python 環境を構成する TPA を構成する必要があります。 `setup` コマンドで `tpaexec` を呼び出します。

```
sudo /opt/EDB/TPA/bin/tpaexec setup
export PATH=$PATH:/opt/EDB/TPA/bin
```

`export` コマンドをシェルのプロファイルに追加できます。

TPA インストールのテスト

`selftest` を実行して、TPA が正しくインストールされたことを確認できます。

```
tpaexec selftest
``` TPA がインストールされました。

##  TPA を使用した PGD のインストール

###  構成ファイルの生成

:ref:`tpaexec configure <Deploying EDB Postgres Distributed using Trusted Postgres␣
↪Architect>` コマンドを実行して、構成フォルダーを生成します。
```

tpaexec configure democluster –architecture PGD-Always-ON –platform bare –edb-postgres-advanced 15 –redwood –no-git –location-names dc1 –pgd-proxy-routing global –hostnames-unsorted

```
:ref:`PGD 5's Always On architectures <Choosing your architecture>` の構成を設定する
PGD-Always-ON アーキテクチャ（`--architecture PGD-Always-ON` ）を指定します。デフォルトアーキテクチャの一部として、3 つのデータノードでクラスターを構成し、3 つの `PGD Proxy <https://
↪www.enterprisedb.com/docs/pgd/latest/routing/proxy/>`_ サーバーとバックアップ用の 1 つの :ref:`Barman <Backup and recovery>` ノードを共ホストします。

Linux ホストの場合、「ベア」プラットフォーム（`--platform bare` ）をターゲットとしていることを指定します。 TPA は、展開中に各ホストで実行されている Linux バージョンを判断します。サポートさ
```

(次のページに続く)
```

(前のページからの続き)

れているオペレーティングシステムの詳細については、 :ref:`the EDB Postgres Distributed␣
↪compatibility table <Compatibility changes>` を参照してください。

データノードが Oracle 互換性（`--redwood` ）で `EDB Postgres Advanced Server v15 <https:/
↪/www.enterprisedb.com/docs/epas/latest/>`_ （`--edb-postgres-advanced 15` ）を実行す
ることを指定します。

`--location-names` を使用して、ノードの概念的な場所を `dc1` に設定します。次に、 `--pgd-
↪proxy-routing` を `global` に設定して、プロキシルーティングがクラスター内のすべてのノード
にトラフィックをルーティングできるようにします。

デフォルトでは、TPA は構成の変更を Git リポジトリにコミットします。この例では、それを行う必要は
ないため、`--no-git` フラグを渡します。

最後に、`--hostnames-unsorted` を渡して、ノードの反復可能なホスト名を生成するように TPA に依
頼します。それ以外の場合は、事前に定義された適切な単語のリストからホスト名をランダムに選択しま
す。

このコマンドは、現在の作業ディレクトリに `democluster` というサブディレクトリを作成します。␣
↪TPA がクラスタの作成に使用する `config.yml` 構成ファイルが含まれています。次を使用して表示で
きます。

```shell
less democluster/config.yml
```

構成ファイルを編集して、管理ユーザー名、パブリックおよびプライベート IP アドレスなど、Linux ホス
トに関連する詳細を追加する必要があります。

構成の編集

好みのエディターを使用して、democluster/config.yml を開きます。

ansible_user: root を含む行を検索します。 root を、SSH アクセスと sudo 特権で構成したユーザー
の名前に変更します。次の行に従ってください。

```
manage_ssh_hostkeys: yes
```

instance_defaults セクションは次のようになります。

edb_notranlate_7 次に、最初のノード kaboom の構成設定である node: 1 を検索します。

node: 1 行の後に、ノードのパブリック IP アドレスとプライベート IP アドレスを追加します。このノー
ドのホストとして linuxhost-1 を使用します。以下をファイルに追加し、IP アドレスを置き換えます。各
行の先頭を node: 行の先頭に揃えます。

```
public_ip: 172.19.16.27
private_ip: 192.168.2.247
```

kaboom のエントリ全体は次のようになりますが、独自の IP アドレスを使用します。

```
- Name: kaboom
  backup: kapok
  location: dc1
  node: 1
  public_ip: 172.19.16.27
  private_ip: 192.168.2.247
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: dc1_subgroup
    bdr_node_options:
      route_priority: 100
``` 他の 3 つのノードでこのプロセスを繰り返します。

ノード kaftan の構成設定である `node: 2` を検索します。このノードには `linuxhost-2` を使用します。 IP アドレスを置き換えて、以下を追加します。

```yaml
  public_ip: 172.19.16.26
  private_ip: 192.168.2.41
```

ノード kaolin の構成設定である node: 3 を検索します。このノードには linuxhost-3 を使用します。IP アドレスを置き換えて、以下を追加します。

```
public_ip: 172.19.16.25
private_ip: 192.168.2.254
```

最後に、ノード kapok の構成設定である node: 4 を検索します。このノードには linuxhost-4 を使用します。IP アドレスを置き換えて、以下を追加します。

```
public_ip: 172.19.16.15
private_ip: 192.168.2.30
```

クラスターのプロビジョニング

次を実行できます。

```
tpaexec provision democluster
```

このコマンドは、クラスターをデプロイする準備をします。(Docker や AWS などの他のプラットフォームでは、このコマンドも必要なホストを作成します。Linux ホストを使用する場合、ホストはすでに構成されている必要があります。)

Trusted Postgres Architect ドキュメントの.. SeeAlso "Further reading"::

- *tpaexec provision*

Linux ホストのこのプロセスの 1 つは、後で SSH 操作のためにホストのキーペアを作成することです。これらのキーペアを作成したら、キーペアの公開部分をホストにコピーする必要があります。`ssh-copy-id` でこれを行うことができ、democluster ID（`-i` ）と各ホストへのログインを提供します。この例では、これらはコマンドです。

```
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.27
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.26
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.25
ssh-copy-id -i democluster/id_democluster rocky@172.19.16.15
```

これで、ホストを検証できる `tpa_known_hosts` ファイルを作成できます。各ホスト（`-H` ）で `ssh-keyscan` を使用し、その出力を `tpa_known_hosts` に追加します。

```
ssh-keyscan -H 172.19.16.27 >> democluster/tpa_known_hosts
ssh-keyscan -H 172.19.16.26 >> democluster/tpa_known_hosts
ssh-keyscan -H 172.19.16.25 >> democluster/tpa_known_hosts
ssh-keyscan -H 172.19.16.15 >> democluster/tpa_known_hosts
```

クラスターをデプロイする

これで、クラスターをデプロイする準備が整いました。展開するには、次を実行します。

```
tpaexec deploy democluster
```

TPA は構成を適用し、必要なパッケージをインストールして、実際の EDB Postgres 分散クラスターをセットアップします。

Trusted Postgres Architect ドキュメントの.. SeeAlso "Further reading"::

- *tpaexec deploy*

クラスターへの接続

これで、SSH を使用してクラスターのノードのいずれかにログインし、データベースに接続する準備が整いました。構成プロセスの一部では、キーを使用して、すべてのノードの SSH ログインを設定します。SSH 構成を使用するには、前に `tpaexec configure` コマンドで作成した `democluster` ディレクトリにいる必要があります。

```
cd democluster
```

そこから `ssh -F ssh_config <hostname>` を実行して SSH 接続を確立できます。クラスター内の最初のデータベースノードである kaboom に接続します。

```
ssh -F ssh_config kaboom
__OUTPUT__
[rocky@kaboom ~]#
```

kaboom で、以前に構成した admin ユーザーと ansible ユーザー rocky としてログインしていることに注意してください。

ここで、enterprisedb ユーザーの ID を採用する必要があります。このユーザーは、クラスターのノードに接続するように事前構成され、承認されています。

```
sudo -iu enterprisedb
__OUTPUT__
enterprisedb@kaboom:~ $
```

`psql` コマンドを実行して、 `bdrdb` データベースにアクセスできます。

```
psql bdrdb
__OUTPUT__
psql (15.2.0, server 15.2.0)
Type "help" for help.

bdrdb=#
```

kaboom ノードで実行されている Postgres データベースに直接接続され、SQL コマンドの発行を開始できます。

SQL クライアントを終了するには、 `exit` と入力します。

PGD CLI の使用

PGD CLI としても知られる pgd ユーティリティを使用すると、 EDB Postgres 分散クラスターを制御および管理できます。すでにノードにインストールされています。

pgd check-health を実行して、クラスターの状態を確認するために使用できます。

```
pgd check-health
__OUTPUT__
Check      Status Message
- ----      ------ -------
ClockSkew  Ok     All BDR node pairs have clockskew within permissible limit
Connection Ok     All BDR nodes are accessible
Raft       Ok     Raft Consensus is working correctly
Replslots  Ok     All BDR replication slots are working correctly
Version    Ok     All nodes are running same BDR versions
enterprisedb@kaboom:~ $
```

または、pgd show-nodes を使用して、クラスター内のデータを保持するノードを表示するように PGD に依頼できます。

```
pgd show-nodes
__OUTPUT__
Node   Node ID    Group        Type Current State Target State Status Seq ID
- ---   -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     1
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     2
enterprisedb@kaboom:~ $
```

同様に、 pgd show-proxies を使用してプロキシ接続ノードを表示します。

```
pgd show-proxies
__OUTPUT__
Proxy  Group       Listen Addresses Listen Port
- ----  -----       ---------------- -----------
kaboom democluster [0.0.0.0]        6432
kaftan democluster [0.0.0.0]        6432
kaolin democluster [0.0.0.0]        6432
```

プロキシは、アプリケーションのデータノードのクラスターへの高可用性接続を提供します。コマンド psql -h kaboom,kaftan,kaolin -p 6432 bdrdb を使用して、プロキシに接続し、さらにデータベースに接続できます。

```
psql -h kaboom,kaftan,kaolin -p 6432 bdrdb
__OUTPUT__
psql (15.2.0, server 15.2.0)
```

(次のページに続く)

(前のページからの続き)

```
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

bdrdb=#
```

クラスターを探索する

- *Connect to your database* アプリケーションへ
- 実践的な演習付きの *Explore failover*
- *Understand conflicts* を作成して監視することにより

*クラスターでの *Next Steps with PGD*

4.4.3 Deploying an EDB Postgres Distributed example cluster on AWS

このクイックスタートでは、Amazon EC2 を使用して、Always On 単一ロケーションアーキテクチャで EDB Postgres Distributed をセットアップします。

TPA と PGD の紹介

さまざまな Postgres 構成のインストールと管理を簡単に繰り返し行えるようにするために、TPA を作成しました。TPA は、Postgres の作成とデプロイを調整します。このクイックスタートでは、最初に TPA をインストールします。TPA が既にインストールされている場合は、これらの手順をスキップできます。TPA を使用して、Postgres クラスターのさまざまな構成をデプロイできます。

PGD は、高いパフォーマンスと可用性を実現するように設計された Postgres のマルチマスター複製実装です。PGD のインストールは、TPA によって調整されます。TPA を使用して、PGD デモンストレーションクラスターの構成ファイルを生成します。このクラスターは Amazon EC2 インスタンスを使用し、3 つのデータノードでクラスターを構成し、3 つの *Installing PGD Proxy* サーバーと *Barman* を共ホスティングします

バックアップ用のノード。次に、TPA を使用して、必要な構成とソフトウェアを各ノードにプロビジョニングおよび展開できます。

準備

注釈: この一連の手順は、特に Intel / AMD プロセッサー上の Ubuntu 22.04 LTS を対象としています。 !!!

EDB アカウント

TPA と PGD の両方をインストールするには、EDB アカウントが必要です。

Sign up for a free EDB account

まだお持ちでない場合。サインアップすると、EDB のソフトウェアリポジトリへの試用版サブスクリプションが提供されます。

登録したら、

EDB Repos 2.0 ページに移動し、リポジトリトークンを取得できます。

このページに初めてアクセスしたときに、[**RequestAccess**] を選択してリポジトリトークンを生成します。**CopyToken** アイコンを使用してトークンをコピーし、安全に保存します。

curl をインストールする

curl コマンドを使用して、リポジトリからインストールスクリプトを取得します。Ubuntu では、curl はデフォルトではインストールされません。存在するかどうかを確認するには、ターミナルで curl を実行します。

```
$ curl
Command curl not found, but can be installed with:
sudo apt install curl
```

見つからない場合は、次を実行します。

```
sudo apt -y install curl
```

環境変数の設定

まず、 EDB_SUBSCRIPTION_TOKEN 環境変数を、

EDB アカウント ステップで取得した EDB リポジトリトークンの値に設定します。

```
export EDB_SUBSCRIPTION_TOKEN=<your-repo-token>
```

これを.bashrc スクリプトまたは同様のシェルプロファイルに追加して、常に設定されるようにすることができます。

リポジトリを構成する

この例に必要なすべてのソフトウェアは、 EDB Postgres 分散パッケージリポジトリから入手できます。次のコマンドは、スクリプトをダウンロードして実行し、EDB Postgres 分散リポジトリを構成します。このリポジトリには、TPA パッケージも含まれています。

```
curl -1sLf "https://downloads.enterprisedb.com/$EDB_SUBSCRIPTION_TOKEN/postgres_
↪distributed/setup.deb.sh" | sudo -E bash
```

edb_notranlate_4 出力またはエラーが生成されない場合は、トークンを正しく入力したことを再確認してください。問題が解決しない場合は、 contact Support で支援を求めてください。

Trusted Postgres Architect（TPA）のインストール

TPA を使用して、PGD をプロビジョニングおよびデプロイします。以前に TPA をインストールしたことがある場合は、
Next Steps with PGD に進むことができます。

TPA をインストールするための完全な手順は、 *Trusted Postgres Architect documentation* 、これもここに含まれています。

Linux 環境

ホストプラットフォームとしての *TPA supports several distributions of Linux* 。これらの例は Ubuntu 22.04 用に書かれていますが、サポートされている他のプラットフォームでも手順は似ています。

TPA パッケージをインストールする

```
sudo apt install tpaexec
```

TPA の構成

TPA の Python 環境を構成する TPA を構成する必要があります。 `setup` コマンドで `tpaexec` を呼び出します。

```
sudo /opt/EDB/TPA/bin/tpaexec setup
export PATH=$PATH:/opt/EDB/TPA/bin
```

`export` コマンドをシェルのプロファイルに追加できます。

TPA インストールのテスト

selftest を実行して、TPA が正しくインストールされたことを確認できます。

```
  tpaexec selftest
  ``` TPA がインストールされました。

  ###  AWS CLI

  TPA は AWS CLI を使用してプロビジョニングとデプロイします。 `Install the CLI <https://
↪docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>`_ 、次に十分な
アクセス権を持つ IAM ユーザーの `configure credentials <https://docs.aws.amazon.com/cli/
↪latest/userguide/cli-configure-files.html>`_ 。 :ref:`TPA AWS platform <Deploying␣
↪EDB Postgres Distributed using Trusted Postgres Architect>` には、必要な権限が詳しく説
明されています。

  ##  TPA を使用した PGD のインストール

  ###  構成ファイルの生成

  :ref:`tpaexec configure <Deploying EDB Postgres Distributed using Trusted Postgres␣
↪Architect>` コマンドを実行して、構成フォルダーを生成します。

  ```shell-session
  tpaexec configure democluster \
    --architecture PGD-Always-ON \
    --platform aws \
    --region eu-west-1 \
    --edb-postgres-advanced 15 \
    --redwood \
    --location-names dc1 \
    --pgd-proxy-routing global \
    --no-git \
    --hostnames-unsorted

:ref:`PGD 5's Always On architectures <Choosing your architecture>` の構成を設定する
PGD-Always-ON アーキテクチャ（\ ``--architecture PGD-Always-ON``
```

）を指定します。デフォルトアーキテクチャの一部として、これにより、3 つのデータノードでクラスターが構成され、3 つの *Installing PGD Proxy* サーバーと、バックアップ用の *Barman* ノードが共ホスティングされます。

AWS（`--platform aws` ）を使用していることを指定し、リージョン（`--region eu-west-1` ）として eu-west-1 を指定します。

TPA のデフォルトは、AWS の t3.micro インスタンスです。このデモンストレーションにはこれで十分であり、
AWS free tier アカウントでの使用にも適しています。

デフォルトでは、TPA は Debian を AWS 上のすべてのノードのデフォルト OS として構成します。

TPA の実行に使用しているのと同じプラットフォームに PGD を展開する必要はないことに注意してください。

データノードが Oracle 互換性（`--redwood` ）で EDB Postgres Advanced Server v15（`--edb-postgres-advanced 15` ）を実行することを指定します。

`--location-names` を使用して、ノードの概念的な場所を `dc1` に設定します。次に、`--pgd-proxy-routing` を `global` に設定して、プロキシルーティングがクラスター内のすべてのノードにトラフィックをルーティングできるようにします。

デフォルトでは、TPA は構成の変更を Git リポジトリにコミットします。この例では、それを行う必要はないため、`--no-git` フラグを渡します。

最後に、 `--hostnames-unsorted` を渡して、ノードの反復可能なホスト名を生成するように TPA に依頼します。それ以外の場合は、事前に定義された適切な単語のリストからホスト名をランダムに選択します。

このコマンドは、現在の作業ディレクトリに `democluster` というサブディレクトリを作成します。TPA がクラスターの作成に使用する `config.yml` 構成ファイルが含まれています。次を使用して表示できます。

```
less democluster/config.yml
```

edb_notranlate_10 - *Deploying with TPA* の PGD-Always-ON 構成オプションの詳細

- Trusted Postgres Architect のドキュメントの *PGD-Always-ON* - Trusted Postgres Architect のドキュメントの *tpaexec configure* - Trusted Postgres Architect のドキュメントの *AWS platform*

クラスターのプロビジョニング：

次に、

tpaexec provision コマンドを使用して、作成したばかりの構成を実行するために必要なリソースを割り当てます。

shell tpaexec provision democluster

プラットフォーム（デフォルトプラットフォーム）として AWS を指定したため、TPA は EC2 インスタンス、VPC、サブネット、ルーティングテーブル、インターネットゲートウェイ、セキュリティグループ、EBS ボリューム、Elastic IP などをプロビジョニングします。

設定時に既存のものを指定しなかったため、TPA は S3 バケットの作成の確認も求めます。

Trusted Postgres Architect ドキュメントの.. SeeAlso "Further reading"::

- *tpaexec provision*

クラスターのデプロイ

構成が整い、インフラストラクチャがプロビジョニングされたら、分散クラスターを *Choosing your deployment method* できるようになりました。

```
tpaexec deploy democluster
```

TPA は構成を適用し、必要なパッケージをインストールして、実際の EDB Postgres 分散クラスターをセットアップします。

Trusted Postgres Architect ドキュメントの.. SeeAlso "Further reading"::

- *tpaexec deploy*

クラスターへの接続

これで、SSH を使用してクラスターのノードのいずれかにログインし、データベースに接続する準備が整いました。構成プロセスの一部では、キーを使用して、すべてのノードの SSH ログインを設定します。SSH 構成を使用するには、前に `tpaexec configure` コマンドで作成した `democluster` ディレクトリにいる必要があります。

```
cd democluster
```

そこから `ssh -F ssh_config <hostname>` を実行して SSH 接続を確立できます。クラスター内の最初のデータベースノードである kaboom に接続します。

```
ssh -F ssh_config kaboom
__OUTPUT__
[admin@kaboom ~]#
```

kaboom で admin としてログインしていることに注意してください。

ここで、enterprisedb ユーザーの ID を採用する必要があります。このユーザーは、クラスターのノードに接続するように事前構成され、承認されています。

```
sudo -iu enterprisedb
__OUTPUT__
enterprisedb@kaboom:~ $
```

`psql` コマンドを実行して、 bdrdb データベースにアクセスできます。

```
psql bdrdb
__OUTPUT__
psql (15.2.0, server 15.2.0)
Type "help" for help.

bdrdb=#
```

kaboom ノードで実行されている Postgres データベースに直接接続され、SQL コマンドの発行を開始できます。

SQL クライアントを終了するには、 `exit` と入力します。

PGD CLI の使用

PGD CLI としても知られる pgd ユーティリティを使用すると、 EDB Postgres 分散クラスターを制御および管理できます。すでにノードにインストールされています。

`pgd check-health` を実行して、クラスターの状態を確認するために使用できます。

```
pgd check-health
__OUTPUT__
Check      Status Message
- ----      ------ -------
ClockSkew  Ok     All BDR node pairs have clockskew within permissible limit
Connection Ok     All BDR nodes are accessible
Raft       Ok     Raft Consensus is working correctly
Replslots  Ok     All BDR replication slots are working correctly
Version    Ok     All nodes are running same BDR versions
enterprisedb@kaboom:~ $
```

または、 `pgd show-nodes` を使用して、クラスター内のデータを保持するノードを表示するように PGD に依頼できます。

```
pgd show-nodes
__OUTPUT__
Node   Node ID    Group        Type Current State Target State Status Seq ID
- ---   -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     1
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     2
enterprisedb@kaboom:~ $
```

同様に、 `pgd show-proxies` を使用してプロキシ接続ノードを表示します。

```
pgd show-proxies
__OUTPUT__
Proxy  Group       Listen Addresses Listen Port
- ----  -----       ---------------- -----------
kaboom democluster [0.0.0.0]        6432
kaftan democluster [0.0.0.0]        6432
kaolin democluster [0.0.0.0]        6432
```

プロキシは、アプリケーションのデータノードのクラスターへの高可用性接続を提供します。コマンド `psql -h kaboom,kaftan,kaolin -p 6432 bdrdb` を使用して、プロキシに接続し、さらにデータベースに接続できます。

```
psql -h kaboom,kaftan,kaolin -p 6432 bdrdb
__OUTPUT__
psql (15.2.0, server 15.2.0)
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
Type "help" for help.

bdrdb=#
```

クラスターを探索する

- *Connect to your database* アプリケーションへ
- *Explore failover* ハンズオンエクササイズ付き
- *Understand conflicts* を作成して監視することにより

*クラスターでの *Next Steps with PGD*

4.4.4 Deploying an EDB Postgres Distributed example cluster on Azure and Google cloud platforms

Azure および Google クラウドへのデプロイ

Azure や Google Cloud Platform などのほとんどのクラウドプラットフォームでは、使用しているクラウドプラットフォームで Linux ホストを作成します。その後、

Deploying on Linux hosts クイックスタートを使用して、PGD をこれらの Linux ホストに展開できます。

- Azure ユーザーは、Linux が読み込まれた Azure VM をプロビジョニングする方法について、a Microsoft guide に従ってください。
- Google Cloud Platform ユーザーは、Linux がロードされた状態で GCP VM をプロビジョニングする方法について、 a Google guide をフォローできます。

  Deploying on Linux hosts に進みます
- AWS ユーザーは、*Deploying on AWS* クイックスタートに移動する必要があります。ここでは、TPA がクラスターの作成に必要なホストを自動的にプロビジョニングします。

4.4.5 Connecting to your database

アプリケーションを接続するか、新しい EDB Postgres 分散クラスターにリモート接続する手順には、次のものが含まれます。

*資格情報を取得し、オプションで.pgpass ファイルを作成する

*接続したい PGD プロキシホストの IP アドレスの確立

*その IP アドレスに接続できることを確認する

*適切な Postgres クライアントの取得

*クライアントをクラスターに接続する

資格情報の取得

デフォルトユーザー enterprisedb は、tpaexec によってクラスターに作成されました。また、プロビジョニングの一環として、そのユーザーのパスワードを生成しました。パスワードを取得するには、次を実行します。

```
tpaexec show-password democluster enterprisedb
```

このコマンドは、enterprisedb ユーザーのパスワードである文字列を返します。必要に応じて、パスワードの入力を求められたときにその文字列を使用できます。

.pgpass ファイルの作成

ホームディレクトリに a `.pgpass file <Monitoring through SQL>` を作成することにより、 psql および他の Postgres クライアントのパスワードの入力を避けることができます。これには、アプリケーションが接続時に検索できるパスワードの詳細が含まれています。パスワードを取得した後（

> 資格情報の取得 を参照)、お好みのエディターを使用して.pgpass

ファイルを開くことができます。

ファイルに、次のように入力します。

```
*:* :bdrdb:enterprisedb:<your password>
```

ファイルを保存してエディターを終了します。ファイルを保護するには、次のコマンドを実行します。このコマンドは、あなただけに読み取りおよび書き込みアクセスを与えます。

```
chmod 0600 ~/.pgpass
```

IP アドレスの確立

Docker

Docker クイックスタートクラスターは、デフォルトで、IP アドレス 172.17.0.2（kaboom）、172.17.0.3（kaftan）、172.17.04（kaolin）、および 172.17.0.5（kapok）でアクセスできます。Docker はこれらのアドレスを生成します。

AWS

`democluster/ssh_config` の IP アドレスを参照できます。または、次を実行します。

```
aws ec2 --region eu-west-1 describe-instances --query Reservations[ *].Instances[* ].
↪{PublicIpAddress:PublicIpAddress,Name:Tags[?Key==`Name`]|[0].Value}
__OUTPUT__
[
    [
        {
            "PublicIpAddress": "54.217.130.13",
            "Name": "kapok"
        }
    ],
    [
        {
            "PublicIpAddress": "54.170.119.101",
            "Name": "kaolin"
        }
    ],
    [
        {
            "PublicIpAddress": "3.250.235.130",
            "Name": "kaftan"
        }
    ],
    [
        {
            "PublicIpAddress": "34.247.188.211",
            "Name": "kaboom"
        }
    ]
]
```

このコマンドは、クラスターインスタンスの EC2 のパブリック IP アドレスのリストを表示します。

Linux ホスト

クイックスタートでクラスターを構成したときに、Linux サーバーの IP アドレスを設定します。それらのアドレスを使用します。

IP アドレスに接続できることを確認します

Linux ホストと Docker

これらを接続するために構成を実行する必要はありません。

AWS

AWS は、SSH エンドポイントへの外部アクセスのみを許可するように構成されています。Postgres クライアントが AWS クラウドの外部から接続できるようにするには、ポート 6432 でのトラフィックの転送を有効にする必要があります。

独自の外部 IP アドレスまたはクラスターに接続するシステムの外部 IP アドレスを取得します。これを行う 1 つの方法は、次を実行することです。

```
curl https://checkip.amazonaws.com
__OUTPUT__
89.97.100.108
```

クラスターのセキュリティグループ ID も必要です。実行：

```
aws ec2 --region eu-west-1 describe-security-groups --filter Name=group-name,Values="␣
↪*democluster* " | grep GroupId
__OUTPUT__
            "GroupId": "sg-072f996360ba20d5c",
```

構成時に設定した、クラスターの正しいリージョンを入力します。

```
aws ec2 authorize-security-group-ingress --group-id <SECURITYGROUPID> --protocol tcp -
↪-port 6432 --cidr <YOURIP>/32 --region eu-west-1
```

繰り返しますが、クラスターの正しいリージョンに配置していることを確認してください。

このコマンドの詳細については、AWS CLI ガイドの *Add rules to your security group* をご覧ください。

適切な Postgres クライアントの取得

ローカルシステムに Postgres をインストールしていない限り、データベースクラスターに接続するには、おそらく psql などのクライアントアプリケーションをインストールする必要があります。

たとえば、Ubuntu では、次を実行できます。

```
sudo apt install postgresql-client
```

このコマンドは、 psql を他のいくつかのツールとともにインストールしますが、Postgres データベースをローカルにインストールせずに。

クライアントをクラスターに接続する

psql または同様のクライアントをインストールした後、クラスターに接続できます。実行：

```
  psql -h <ipaddressofnode> -p 6432 -U enterprisedb bdrdb
  __OUTPUT__
  psql (15.2, server 15.2.0 (Debian 15.2.0-2.buster))
  SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,␣
↪compression: off)
  Type "help" for help.

  bdrdb=#
```

:ref:`Use the `.pgpass` file <*#creating-a-pgpass-file*>` をサポートするクライアントで使用するか、ホスト、ポート、ユーザー、パスワード、およびデータベース名を使用して他のアプリケーションと接続します。

プロキシフェールオーバーを使用してクライアントをクラスターに接続する

プロキシのすべてのアドレスをホストとしてリストすることにより、プロキシに障害が発生した場合、クライアントが常にフェールオーバーし、最初に使用可能なプロキシに接続することを保証できます。

```
psql -h <ipaddressofnode1>,<ipaddressofnode2>,<ipaddressofnode3> -U enterprisedb -p␣
↪6432 bdrdb
__OUTPUT__
psql (15.2, server 15.2.0 (Debian 15.2.0-2.buster))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,␣
↪compression: off)
Type "help" for help.

bdrdb=#
```

接続 URL の作成

多くのアプリケーションは、 Connection URL を使用してデータベースに接続します。接続 URL を作成するには、次の形式の文字列を組み立てる必要があります。

```
postgresql://<user>@<ipaddressofnode1>:6432,<ipaddressofnode2>:6432,<ipaddressofnode3>
↪:6432/bdrdb
```

この文字列形式は psql コマンドで使用できるため、データベースノードが IP アドレス 192.168.9.10、192.168.10.10、および 192.168.10.11 にある場合、次を使用できます。

```
psql postgresql://enterprisedb@192.168.9.10:6432,192.168.10.10:6432,192.168.11.
↪10:6432/bdrdb
```

作成した URL にパスワードを埋め込むこともできます。enterprisedb ユーザーを使用していて、enterprisedb ユーザーのパスワードが notasecret の場合、URL は次のようになります。

```
psql postgresql://enterprisedb:notasecret@192.168.9.10:6432,192.168.10.10:6432,192.
↪168.11.10:6432/bdrdb
```

実際のパスワードはそれよりも複雑で、特殊文字が含まれている場合があります。パスワードを urlencode して、使用しているシェル、コマンド、またはドライバーをトリップしないようにする必要があります。

Java 接続 URL の作成

最後に、作成した URL は多くの Postgres アプリケーションとクライアントで問題ありませんが、Java ベースのものでは、Java が適切なドライバーを呼び出せるように 1 つの変更が必要です。URL の前に jdbc: を付けて、Java 互換の URL を作成します。

```
jdbc:postgresql://enterprisedb@192.168.9.10:6432,192.168.10.10:6432,192.168.11.
↪10:6432/bdrdb
```

移動

これで、必要なすべての接続資格情報と URL を使用して、アプリケーションをクラスターに接続する準備が整いました。

4.4.6 Exploring failover handling with PGD

高可用性クラスターでは、フェールオーバー機能はクラスターの全体的な復元力にとって重要です。リードデータノードが何らかの理由で動作を停止した場合、アプリケーションはほとんど、またはまったく中断せずにデータベースの操作を継続できる必要があります。PGD の場合、これは、アプリケーションを新しいリードデータノードに転送することを意味し、自動的に引き継ぎます。これが PGD プロキシが役立つ場所です。クラスターと連携し、トラフィックをリードデータノードに自動的に転送します。

この演習では、データベースに定期的にデータを送信するアプリケーションを作成します。次に、PGD CLI で変更を要求することにより、最初にリードデータノードをソフトに切り替えます。次に、データベースインスタンスを強制的にシャットダウンし、PGD がそれをどのように処理するかを確認します。

クイックスタート構成

この探索では、 *quick start for Docker* 、 *quick start for AWS* 、または *quick start for Linux hosts* を使用して PGD クラスターを作成したことを前提としています。

各クイックスタートの最後には、4 つのノードと次のロールを持つクラスターが作成されます。

Host name	Host role
kaboom	PGD data node and pgd-proxy co-host
kaftan	PGD data node and pgd-proxy co-host
kaolin	PGD data node and pgd-proxy co-host
kapok	Barman backup node

この演習では、これらのホスト名を使用します。

xpanes のインストール

xpanes を使用します。これは、簡単に切り替えることができる複数のターミナルセッションをすばやく作成できるユーティリティです。デフォルトではインストールされないため、インストールする必要があります。この演習では、tpaexec を実行してクイックスタートクラスターを構成したシステムから xpanes を起動します。

システムで Ubuntu を実行している場合は、次を実行します。

```
sudo apt install software-properties-common
sudo add-apt-repository ppa:greymd/tmux-xpanes
sudo apt update
sudo apt install tmux-xpanes
```

これらは the xpanes repository からのインストール手順です。Ubuntu を使用していない場合、リポジトリには他のシステムのインストール手順も含まれています。

4 つのサーバーへの接続

xpanes がインストールされたら、次を実行して、4 つすべてのサーバーとの SSH セッションを作成できます。

```
cd democluster
xpanes -d -c "ssh -F ssh_config {}" "kaboom" "kaolin" "kaftan" "kapok"
```

これらのコマンドを実行すると、4 つのペインが表示されます。4 つのペインは kaboom、kaolin、kaftan、および kapok に接続され、それぞれに root ユーザーとしてログインします。演習の後半でサービスを簡単に停止および開始できるように、この特権が必要です。

Control-b を押してから **q** を押して、各ペインの数値を簡単に表示します。

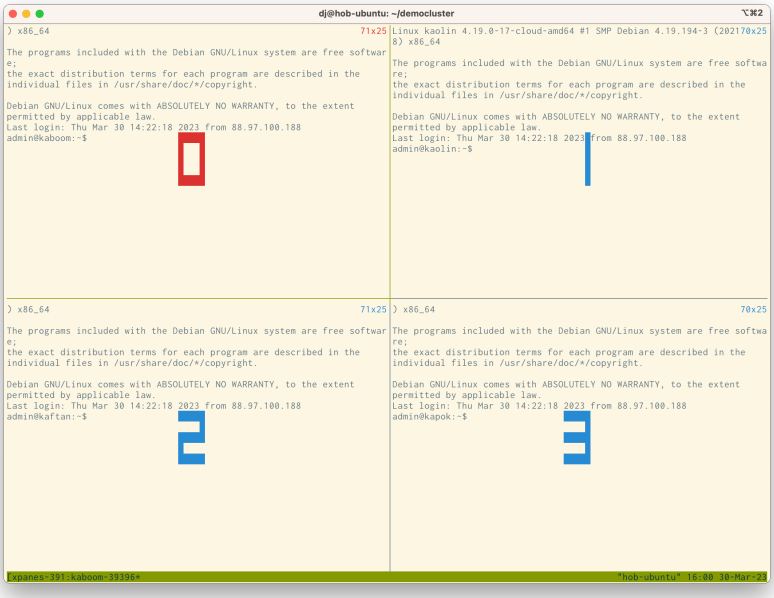


図 1: 4 SSH Sessions showing numbers

ペイン間でフォーカスを切り替えるには、 **Control-b** とカーソルキーを使用してペイン間を移動できます。または、 **Control-b** を使用し、その後に **q** とフォーカスするペインの番号を入力します。両方の方法を示します。

Control-b ↓を使用します。Control-b→ または **Control-bq3** して、フォーカスを kapok ホストである右下のペインに移動します。このサーバーは、バックアップの実行を担当します。これは、デモアプリケーションの操作のベースとして使用します。Barman の資格情報を使用して、データベースサーバーとプロキシに接続できます。

```
sudo -iu barman
psql -h kaboom -p 6432 bdrdb
```

このコードは kaboom ホスト上のプロキシに接続します。ホストはクラスターの一部として Postgres インスタンスも実行します。

次のステップでは、アプリケーションが書き込むテーブルを作成します。

```
drop table if exists ping cascade;
CREATE TABLE ping (id SERIAL PRIMARY KEY, node TEXT, timestamp TEXT) ;
```

このコードは、最初に ping テーブルを削除します。次に、id 主キーと、ノードとタイムスタンプ用の 2 つのテキストフィールドを使用して、ping テーブルを再作成します。これでテーブルの準備ができました。確認するには、**Control-b ←を使用します。Control-b ↑** または **Control-bq0** して左上のペインに移動し、kaboom サーバーに接続します。このペインで、enterprisedb ユーザーになり、データベースに簡単に接続できます。

```
sudo -iu enterprisedb
```

次を実行して、ローカルデータベースに接続できます。

```
psql bdrdb
```

このコマンドは、kaboom 上のローカルデータベースインスタンスに直接接続します。\dt を使用して、使用可能なテーブルを表示します。

```
bdrdb=# \dt
       List of relations
 Schema | Name | Type  | Owner
- -------+------+-------+--------
 public | ping | table | barman
(1 row)
```

\d ping を実行すると、ping を作成する DDL が kaboom サーバー上にあることが示されます。

edb_notranlate_7 このテーブルが複製されていることを確認したい場合は、クラスター内の別のノードに接続して見ることができます。psql の\c コマンドを使用すると、別のサーバーに接続できます。Kaftan ノードに接続するには、次を実行します。

```
\c - - kaftan
```

次のようなログインメッセージが表示されます。

```
psql.bin (15.2.0 (Debian 15.2.0-2.buster), server 15.2.0 (Debian 15.2.0-2.buster))␣
↪SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
You are now connected to database "bdrdb" as user "enterprisedb" on host "kaftan"␣
↪(address "10.33.25.233") at port "5444".
bdrdb=#
```

\dt および\d ping を実行すると、kaftan ノードで同じ結果が表示されます。

kaboom ノードに再接続するには、次を実行します。

```
\c - - kaboom
```

モニターのセットアップ

次に、ping テーブルのアクティビティを監視します。次の SQL を入力して、最新の 10 件のエントリを表示します。

```
select * from ping order by timestamp desc limit 10;
```

このコマンドを複数回実行するには、シェルで\watch コマンドを使用します。これにより、定期的に最後のクエリが実行されます。毎秒更新するには、次のように入力します。

```
\watch 1
```

これまでのところ、見るものは何もありません。すぐにアクティビティを追加します。

ping の作成

Control-b ↓を使用して、Barman ホストカポックに戻ります。Control-b→ または **Control-bq3** 。

このセッションはまだ psql セッションにログインしています。次にシェルスクリプトを実行するため、psql を終了する必要があります。 **Control-d** を押します。

シェルプロンプトに次のように表示されます。

barman@kapok:~$

admin@kapok または root@kapok と書かれている場合は、sudo -iu barman を実行して Barman ユーザーに戻ります。

作成するアプリケーションはシンプルです。書き込むノードと ping のタイムスタンプを取得します。次に、できるだけ早く、新しい ping を ping テーブルに書き込みます。

シェルで、次のように入力します。

```
while true; do psql -h kaftan,kaolin,kaboom -p 6432 bdrdb -c "INSERT INTO ping(node,␣
↪timestamp) select node_name, current_timestamp from bdr.local_node_summary;"; done
```

より読みやすい形式で、つまり：

```
while true;
    do psql -h kaftan,kaolin,kaboom -p 6432 bdrdb -c \
        "INSERT INTO ping(node, timestamp) select node_name, current_timestamp from␣
↪bdr.local_node_summary;"
    done
```

定数ループでは、psql コマンドを呼び出し、ホストとして 3 つのプロキシのいずれかに接続するように指示し、プロキシポートを提供し、bdrdb データベースを選択します。また、2 つの値を ping テーブルに挿入するコマンドを渡します。値の 1 つは、実際に接続しているノードの名前を含む bdr.local_node_summary から取得されます。もう 1 つの値は現在の時刻です。

ループが実行されると、新しいエントリがテーブルに表示されます。モニターをセットアップする左上のペインにそれらが表示されます。

これで、フェールオーバーのテストを開始できます。

書き込みリーダーの表示

プロセスのこの部分では、左下隅にあるホスト kaftan に切り替えます。**Control-b ←** または **Control-bq2** を使用して、フォーカスを切り替えます。

pgd を実行するための適切な特権を取得するには、PGD コマンドラインインターフェイスで次を実行します。

```
sudo -iu enterprisedb
```

クラスターの状態を表示するには、次を実行します。

```
pgd show-groups
```

次のような出力が表示されます。

```
Group        Group ID   Type   Parent Group Location Raft Routing Write Leader
- ----        --------   ----   ------------ -------- ---- ------- ------------
democluster  1935823863 global                       true false
dc1_subgroup 1302278103 data   democluster  dc1      true true    kaboom
```

グローバルグループ democluster には、すべてのサブグループが含まれます。 dc1_subgroup は、使用しているデータクラスターです。そのグループ名の値は、このクラスターを構成したときにクイックスタートで指定された場所から派生します。各ロケーションは独自のサブグループを取得するため、他のロケーションまたはクラスターとは無関係に管理できます。

表の右側にスキップすると、グループの現在の書き込みリーダー（すべてのプロキシが更新を送信するサーバー）が kaboom であることがわかります。

switchover コマンドをクラスターグループに送信して、リーダーを変更します。次のコマンドを実行します。

edb_notranlate_33 ノード名は、dc1_subgroup グループ内の別のデータノードのホスト名です。

2 つの応答のいずれかが表示されます。 show-groups コマンドを実行したとき、書き込みリーダーとして kaolin が表示された場合、次のように表示されます。

```
Error: "kaolin" is already a write leader
```

これは、カオリンがすでに書き込みリーダーに選出されたことを意味するため、切り替えても効果はありません。この演習では、ノード名を kaboom または kaftan に置き換えて、別のホストへのスイッチオーバーを再試行します。

現在の書き込みリーダーではないホストを選択すると、他の応答が表示されます。

```
switchover is complete
```

左上のペインを見ると、スクリプトからの挿入が切り替えられ、切り替えたばかりのノードに書き込まれていることがわかります。

!!!Info ID 番号を観察する生成されている ID 番号は、値の完全に異なる範囲からのものであることに注意してください。これは、システムが ID を生成するシーケンスを透過的にグローバルシーケンスにしたためです。グローバルシーケンスとそれらが

> *bdr.* **シーケンス** でどのように機能するかについてもっと読む。

ノードを失う

リーダーを切り替えることができることは、計画的なメンテナンスに役立ちます。クラスターに構成を変更するように指示します。予期せぬ変化が起こったら？ここで、そのシナリオを作成します。

左下のペインで、リーダーをカオリンに設定します。

```
pgd switchover --group-name dc1_subgroup --node-name kaolin
```

次に、**Control-b ↑を使用して、フォーカスを右上のペインに変更します。Control-b→** または **Control-bq1** 、これはカオリンホストでのセッションです。

次を実行して、Postgres サーバーをオフにします。

```
sudo systemctl stop postgres.service
```

左上のペインには、クラスターサブグループが新しいリーダーを選択すると、監視対象テーブルがカオリンから別のノードに切り替わるのが表示されます。更新がキャンセルされると、右下ペインのスクリプトにエラーが表示される場合があります。ただし、新しいリーダーが選出されるとすぐに、そのリーダーへのトラフィックのルーティングが開始されます。

ノードの状態を表示する

Control-b ↓を使用して左下のペインに切り替えます Control-b ← または **Control-bq2** 、実行：

```
pgd show-nodes
```

次のようなものが表示されます。

```
Node   Node ID    Group        Type Current State Target State Status      Seq ID
- ---   --------    -----        ---- ------------- ------------ ------      ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up          3
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up          2
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Unreachable 1
```

カオリンノードがダウンしており、更新は別の書き込みリーダーに送られます。

監視ラグ

カオリンがダウンしている間、PGD の中心にある論理レプリケーションは、カオリンがクラスターとどの程度同期していないかを追跡しています。それを自分で確認するには、次を実行します。

```
psql bdrdb -c "select * from bdr.node_replication_rates;"
```

このコマンドは、サーバー間の現在のレプリケーション速度を表示します。

```
 peer_node_id | target_name | sent_lsn   | replay_lsn |   replay_lag    | replay_lag_
↪bytes | replay_lag_size | apply_rate | catchup_interval
- -------------+-------------+-----------+------------+-----------------+-------------
↪-----+-----------------+------------+------------------
   2710197610 | kaboom      | 0/769F650 | 0/769F650  | 00:00:00        |
  0 | 0 bytes         |       1861 | 00:00:00
              | kaolin      | 0/7656648 | 0/7656648  | 00:03:07.252266 |          ␣
↪299016 | 292 kB          |            |
(2 rows)
```

この出力を見ると、カオリンには 3 分のリプレイラグと、今戻ってきた場合に追いつくための約 292KB のデータがあることがわかります。カオリンがダウンしている時間が長いほど、リプレイラグが大きくなります。モニタリングコマンドを再実行すると、数値が上昇していることがわかります。

```
 peer_node_id | target_name | sent_lsn   | replay_lsn |   replay_lag    | replay_lag_
↪bytes | replay_lag_size | apply_rate | catchup_interval
- -------------+-------------+-----------+------------+-----------------+-------------
↪-----+-----------------+------------+------------------
   2710197610 | kaboom      | 0/76B1D28 | 0/76B1D28  | 00:00:00        |
  0 | 0 bytes         |       1743 | 00:00:00
              | kaolin      | 0/7656648 | 0/7656648  | 00:03:53.045704 |          ␣
↪374496 | 366 kB          |            |
(2 rows)
```

さらに 46 秒が経過し、ラグは 74KB 拡大しました。次に、ノードを元に戻し、システムがどのように回復するかを確認します。

ノードの再起動

カオリンで Postgres サービスを戻すことができます。**Control-b ↑を使用して、右上のペインに戻ります。Control-b→** または **Control-bq1** 、実行：

```
sudo systemctl start postgres.service
```

変化は見られません。データベースサービスは復旧して実行されていますが、クラスターは選挙を行っていないため、リーダーはそのままです。**Control-b ↓を使用して左下のペインに切り替えます Control-b ←** または **Control-bq2** 、実行：

```
pgd show-nodes
```

次の内容が表示されます。

```
Node   Node ID    Group        Type Current State Target State Status Seq ID
- ---   -------    -----        ---- ------------- ------------ ------ ------
kaboom 2710197610 dc1_subgroup data ACTIVE        ACTIVE       Up     3
kaftan 3490219809 dc1_subgroup data ACTIVE        ACTIVE       Up     2
kaolin 2111777360 dc1_subgroup data ACTIVE        ACTIVE       Up     1
```

カオリンがクラスターに戻るとすぐに、クラスターとの同期が開始されます。それは、そのリプレイデータに追いつくことによってそれを行います。実行：

```
psql bdrdb -c "select * from bdr.node_replication_rates;"
```

出力は次のようになります。

```
 peer_node_id | target_name | sent_lsn  | replay_lsn | replay_lag | replay_lag_bytes␣
↪| replay_lag_size | apply_rate | catchup_interval
- -------------+-------------+-----------+------------+------------+------------------
↪+-----------------+------------+------------------
   2710197610 | kaboom      | 0/8092938 | 0/8092938  | 00:00:00   |                0␣
↪| 0 bytes         |       2321 | 00:00:00
   2111777360 | kaolin      | 0/8092938 | 0/8092938  | 00:00:00   |                0␣
↪| 0 bytes         |     337426 | 00:00:00
(2 rows)
```

ご覧のとおり、カオリンは完全に自動的に追いついたので、リプレイラグはなくなりました。

カオリンが完全にサービスを再開すると、すべてをそのままにすることができます。ライトリーダーであるサーバーを変更する必要はありません。フェイルオーバーメカニズムは、必要に応じて別のサーバーを書き込みリーダーに引き上げる準備ができています。

必要に応じて、次を実行してカオリンリーダーを再度作成できます。

```
pgd switchover --group-name dc1_subgroup --node-name kaolin
```

このコマンドは、リードを書き込むカオリンを返します。プロキシが書き込みリーダーを追跡するため、アプリケーションの更新が続きます。

プロキシフェイルオーバー

プロキシはフェールオーバーすることもできます。これを体験するには、フォーカスがまだ左下のペインにあることを確認して、次を実行します。

```
pgd show-proxies
```

次の内容が表示されます。

```
Proxy  Group        Listen Addresses Listen Port
- ----  -----        ---------------- -----------
kaboom dc1_subgroup [0.0.0.0]        6432
kaftan dc1_subgroup [0.0.0.0]        6432
kaolin dc1_subgroup [0.0.0.0]        6432
```

`exit` と入力して enterprisedb ユーザーを終了し、管理/ルートシェルに戻ります。次を実行して、このノードのプロキシサービスを停止できます。

```
sudo systemctl stop pgd-proxy.service
```

スクリプトが別のプロキシに切り替えると、右下のウィンドウに簡単なエラーが表示されます。ただし、書き込みリーダーは変更されないため、プロキシのスイッチは、モニタークエリが実行されている左上のペインに表示されません。

次を実行して、Kaftan のプロキシサービスを元に戻します。

```
sudo systemctl start pgd-proxy.service
```

!!!Tip Tmux の終了 Tmux と関連するすべてのセッションをすばやく終了できます。セッションが強制終了された後も実行を継続するため、最初に実行中のプロセスを終了します。**Control-B** を押してから、`:kill-session` と入力します。このアプローチは、**Control-D** または `exit` を使用して各ペインのセッションを 1 つずつ終了するよりも簡単です。

この例では、3 つのデータノードと 1 つのバックアップノードのクイックスタート構成を使用します。2 つのデータノードと 1 つの監視ノードを持つようにクラスターを構成できます。これは、ノードの障害に対する回復性が低くなります。または、5 つのデータノードを構成できます。これは、ノードの障害に対する回復力がはるかに高くなります。この構成では、アプリケーションでフェールオーバーがどのように機能するかを調べることができます。複数の場所を持つクラスターの場合、同じ基本ルールが適用されます。サーバーをダウンすると、プロキシが指す新しい書き込みリーダーが選択されます。

さらに読む

- *PGD CLI の使用* の管理機能の詳細を読む。
- *monitoring replication using SQL* の詳細

4.4.7 Exploring conflict handling with PGD

PGD のようなマルチマスターアーキテクチャでは、競合が発生します。PGD はそれらを処理するように構築されています。

1 つのデータベースノードでアプリケーションから行への更新があり、別のノードで同じ行への別の更新がある場合、競合が発生する可能性があります。このタイプの競合は、**行レベルの競合**と呼ばれます。競合はエラーではありません。それらを効果的に解決することは、Postgres Distributed が一貫性を維持する方法の中核です。

コンフリクトを処理する最良の方法は、そもそもコンフリクトを持たないことです！プロキシで PGD の Always-On アーキテクチャを使用して、アプリケーションがクラスター内の同じサーバーに書き込むようにします。

ただし、競合が発生した場合、PGD が競合を解決する方法、その解決を制御する方法、および競合が発生していることを確認する方法を知っておくと役立ちます。行の挿入と行の更新は、競合を引き起こす可能性のある 2 つのアクションです。

それがどのように機能するかを確認するには、すべてのサーバーのコマンドラインビューを開く必要があります。

クイックスタート構成

この探索では、 *quick start for Docker* 、 *quick start for AWS* 、または *quick start for Linux hosts* を使用して PGD クラスターを作成したことを前提としています。

各クイックスタートの最後には、4 つのノードと次のロールを持つクラスターが作成されます。

Host name	Host role
kaboom	PGD data node and pgd-proxy co-host
kaftan	PGD data node and pgd-proxy co-host
kaolin	PGD data node and pgd-proxy co-host
kapok	Barman backup node

この演習では、これらのホスト名を使用します。

xpanes のインストール

xpanes を使用します。これは、簡単に切り替えることができる複数のターミナルセッションをすばやく作成できるユーティリティです。デフォルトではインストールされないため、インストールする必要があります。ssh で kaboom ノードに接続することから始めます。

```
cd democluster && ssh -F ssh_config kaboom
```

Docker でクイックスタートを実行している場合、Red Hat の派生物である Rocky Linux を使用します。 xpanes のインストールを実行するには、次を実行します。

```
dnf -y install xpanes
```

AWS でクイックスタートを実行している場合、Debian Linux を使用します。xpanes のインストールを実行するには、次を実行します。

```
wget https://github.com/greymd/tmux-xpanes/releases/download/v4.1.4/tmux-xpanes_v4.1.
↪4.deb
sudo apt -y install ./tmux-xpanes*.deb
rm tmux-xpanes*.deb
```

4 つのサーバーへの接続

認証を機能させるには、enterprisedb ユーザーとしてログインする必要があります。

```
sudo -iu enterprisedb
```

次に、次のコマンドを実行して、3 つのデータベースサーバーとプロキシサーバーに接続します。

```
xpanes -d -c "psql postgresql://enterprisedb@{}/bdrdb?sslmode=require" "kaboom:5444"
↪"kaftan:5444" "kaolin:5444" "kaboom:6432"
```

xpanes は-c の後のコマンドを受け取り、続く引数の値を使用して、実行するコマンドを作成します。つまり、実行すると 4 つのペインが表示されます。3 つのペインがポート 5444 のデータベースノード kaboom、kaftan、および kaolin に接続されます。1 つはポート 6432 の kaboom で実行されている pgd-proxy に接続されます。それぞれが enterprisedb としてデータベースにログインします。

Control-b を押してから **q** を押して、各ペインの数値を簡単に表示します。

ペイン間でフォーカスを切り替えるには、 **Control-b** とカーソルキーを使用してペイン間を移動できます。または、 **Control-b** を使用し、その後に **q** とフォーカスするペインの番号を入力します。両方の方法を示します。

Control-b ↓を使用して左下のペインに移動します。Control-b→ または Control-b q 3**。

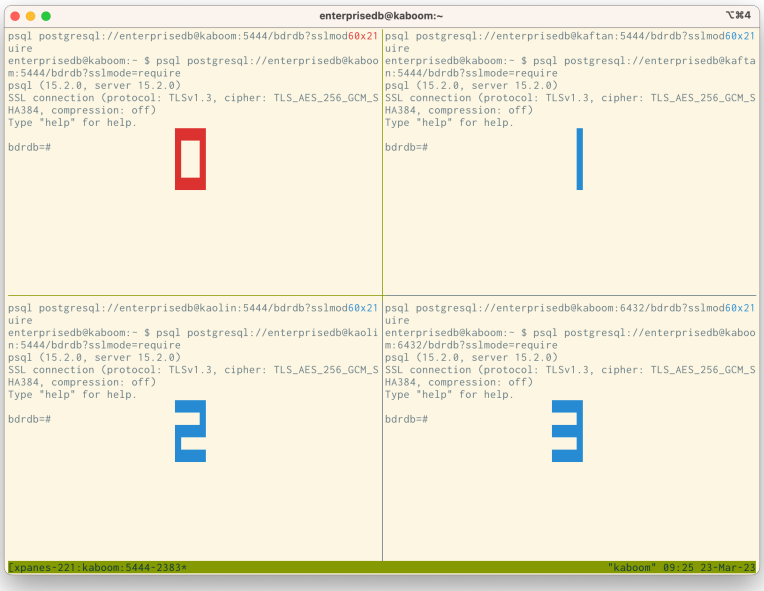


図 2: 4 Sessions showing numbers

衝突の準備

競合を起こすには、最初に単純なテーブルが必要です。現在フォーカスがあるペインで、次のように入力します。

```
drop table if exists test_conflict;
create table test_conflict(
  id integer primary key ,
  value_1 text);
```

競合の監視

現在フォーカスがあるペインで、次のように入力します。

```
select * from bdr.conflict_history_summary
\watch 1
```

select コマンドは、クラスターの競合履歴を表示します。 \watch 1 コマンドは、前述のコマンドを 1 秒ごとに再実行する psql コマンドです。

これで、競合を生成する準備が整いました。

競合の作成

最も基本的な競合の形態は、2 つの異なるノード上のテーブルに挿入が発生し、両方が同じ主キーを持つ場合です。そのシナリオを作成して観察できるようになりました。

Control-b ↑を使用して左上のペインに移動します Control-b ← または **Control-bq0** 。このペインは kaboom ノードです。そこでトランザクションを開始し、行を挿入します。

```
start transaction;
insert into test_conflict values (1, from kaboom);
```

次に、**Control-b→** または **Control-bq1** を使用して、右上のペインに移動します。このペインは kaftan ノードです。ここでも、トランザクションを開始し、異なるデータを同じ行に挿入します。

```
start transaction;
insert into test_conflict values (1, from kaftan);
```

これで、異なるサーバーで 2 つのトランザクションが開かれ、挿入操作が既に正常に実行されました。この時点で両方のトランザクションをコミットする必要があります。

*** Control-b ← **または** Control-b q 0**を使用して、 `commit;` と入力します。

*** Control-b → **または** Control-b q 1**を使用し、 `commit;` と入力します。

両方のコミットが機能していることがわかります。ただし、右下のペインでは、競合が検出されていることを確認できます。



図 3: 4 Sessions showing conflict detected

競合履歴の行に、 `insert_exists` . また、この競合の解決は、コミットのタイミングに基づいて、より新しいレコードが保持されることであることに注意してください。この競合は、INSERT / INSERT 競合と呼ばれます。このタイプの競合の詳細については、

INSERT / INSERT の競合 を参照してください。

更新の競合の作成

同じレコードに対する異なる更新が異なるノードで行われると、競合が発生します。現在の構成でそのシナリオを作成することもできます。右下のペインで`\watch 1`を実行したままにします。

Control-b ← または **Control-bq0** を使用して、左上のペインに移動します。このペインは kaboom ノードです。ここで、トランザクションを開始し、行を更新します。

```
start transaction;
update test_conflict set value_1 = update from kaboom where id = 1;
```

次に、**Control-b→** または **Control-bq1** を使用して、右上のペインに移動します。このペインは kaftan ノードです。ここでも、トランザクションを開始し、同じ行を異なるデータで更新します。

```
start transaction;
update test_conflict set value_1 = update from kaftan where id = 1;
```

これで、異なるサーバーで 2 つのトランザクションが開かれ、更新操作が既に正常に実行されました。この時点で両方のトランザクションをコミットする必要があります。

*** Control-b cursor-left **または** Control-b q 0**を使用し、`commit;`と入力します。

*** Control-b cursor-right **または** Control-b q 1**を使用し、`commit;`と入力します。

繰り返しますが、両方のコミットが機能していることがわかります。そして、再び、右下のペインで、更新の競合が検出されていることを確認できます。

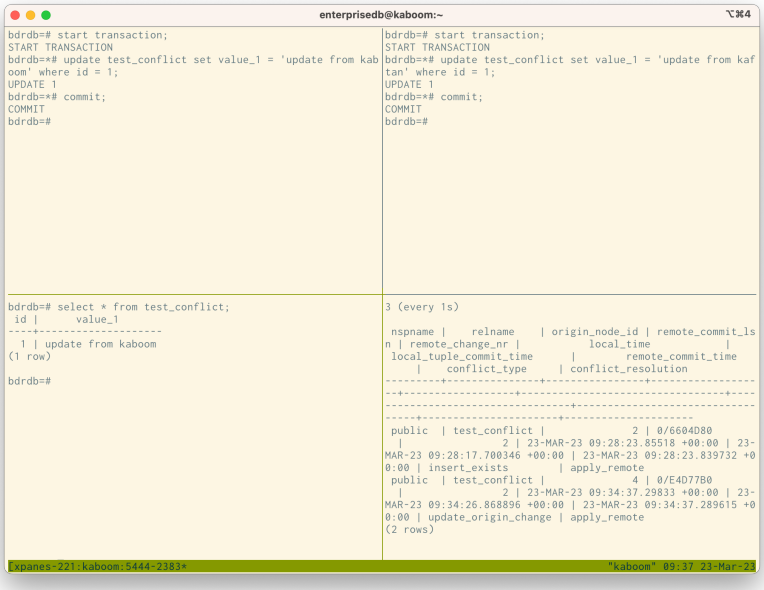


図 4: 4 Sessions showing update conflict detected

競合履歴の追加の行は、`update_origin_change` 競合が発生し、解決が `apply_remote` であったことを示しています。この解決は、リモート変更が適用され、レコードが更新されたことを意味します。この競合は UPDATE / UPDATE 競合と呼ばれ、 *UPDATE / UPDATE* **の競合** で詳しく説明されています。

!!!Tip Tmux の終了 Tmux と関連するすべてのセッションをすばやく終了できます。セッションが強制終了された後も実行を継続するため、最初に実行中のプロセスを終了します。**Control-b** を押してから、`:kill-session` と入力します。このアプローチは、**Control-D** または `exit` を使用して各ペインのセッションを 1 つずつ終了するよりも簡単です。

これで、考えられるすべての競合シナリオと発生する可能性のある解決策を探索する準備が整いました。競合の管理方法の詳細については、ドキュメントの *CRDT* **データ型を使用した列の競合の処理** を参照してください。理想的には、競合を避けるべきですが、競合が発生すると、Postgres Distributed の統合され構成可能な競合リゾルバーによって記録および管理されることを知っておくことが重要です。

4.4.8 Next Steps with PGD

PGD クラスターをさらに活用する

アーキテクチャ

このクイックスタートでは、高可用性 Postgres データベースの単一リージョンクラスターを作成しました。これは、Always On Single Location アーキテクチャで、使用可能な PGD アーキテクチャの範囲の 1 つです。他のアーキテクチャには、複数のデータセンターのクラスターが連携する Always On Multi-Location、および復元力を強化する監視ノードを備えた両方のバリエーションが含まれます。詳細については、

> **アーキテクチャのオプションとパフォーマンス** を参照してください。

Postgres のバージョン

このクイックスタートでは、 EDB Postgres Advanced Server （EPAS）をデータベースノードにデプロイしました。PGD は、3 種類の Postgres ディストリビューション、EPAS、 EDB Postgres Extended Server、およびオープンソースの PostgreSQL を展開できます。データベースの選択は PGD に影響し、サーバーに依存する *different capabilities* を提供します。

*オープンソースの PostgreSQL は CAMO をサポートしていません

- EDB Postgres Extended Server は CAMO をサポートしますが、Oracle 互換性は提供しません
- EDB Postgres Advanced Server は CAMO をサポートし、オプションの Oracle 互換性を提供します

続きを読む

- PGD の 行型の用語 を学ぶ-非同期レプリケーションから書き込みスケーラビリティまで
- PGD で *applications work* を使用する方法と、*bdr.* シーケンス などの Postgres の一般的な機能がグローバルに配布される方法を調べる
- PGD がクラスターの rolling upgrades をサポートする方法を発見する
- *bdr.node_group_routing_config_summary* を制御し、 SQL を使用して PGD プロキシを制御する
- *PGD CLI* の使用 と連携して、クラスターを管理および監視します

クラスターのプロビジョニング解除

クラスターのテストが完了したら、プロビジョニングを解除します。

```
tpaexec deprovision democluster
```

- Docker デプロイメントでは、プロビジョニングを解除すると、Docker コンテナ、ネットワーク、およびその他のローカル構成が破棄されます。
- AWS デプロイでは、EC2 インスタンス、VPC 構成、およびその他の関連リソースが削除されます。作成した S3 バケットはそのままにします。これは手動で削除する必要があります。

第5章　Concepts

5.1 Terminology

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

5.1.1 非同期レプリケーション

オリジンノードでトランザクションが完了した後、クラスターメンバーにデータをコピーします。非同期レプリケーションは、同期レプリケーションよりも高いパフォーマンスと低いレイテンシーを提供できます。ただし、複数の同時変更が原因で競合が発生する可能性があります。発生した競合を管理する必要があります。

5.1.2 可用性

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

5.1.3 CAMO または commit-at-most-once

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

5.1.4 クラスタリング

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

5.1.5 データシャーディング

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

5.1.6 PGD の Eager Replication

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

5.1.7 結果整合性

異なるクラスターメンバー内の同じ項目への変更を述べる分散コンピューティング一貫性モデルは、同じ値に収束します。PGD では、これは、競合解決と競合のないレプリケートされたデータ型を備えた非同期論理レプリケーションによって実現されます。

5.1.8 フェールオーバー

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

5.1.9 水平スケーリングまたはスケールアウト

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

5.1.10 論理レプリケーション

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

5.1.11 ナイン

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

5.1.12 ノード

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

5.1.13 物理レプリケーション

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

5.1.14 読み取りのスケーラビリティ

1 つ以上のリードレプリカノードをクラスターに導入し、アプリケーションにプライマリノードへの直接書き込みとレプリカノードへの読み取りを行わせることによって達成できます。読み取りワークロードが増加するにつれて、パフォーマンスを維持するためにリードレプリカノードの数を増やすことができます。

5.1.15 目標復旧時点（RPO）

アプリケーションの配信の中断によりデータが失われる可能性のある最大対象期間。非常に低いまたは最小限の RPO は、非常に高可用性のドライバーです。

5.1.16 目標復旧時間（RTO）

中断されたアプリケーションを復元するための目標時間。非常に低いまたは最小限の RTO は、非常に高可用性のドライバーです。

5.1.17 単一障害点（SPOF）

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

5.1.18 スイッチオーバー

アプリケーションとクラスター内のアクティブなデータベースノード間の接続の計画的な変更。通常はメンテナンスのために行われます。

5.1.19 同期レプリケーション

参加しているすべてのノードで変更が同時に更新される場合、通常は 2 フェーズコミットを活用します。このアプローチは即時の一貫性を提供し、競合を回避しますが、ノード間で必要な調整が原因でレイテンシーのパフォーマンスコストが発生します。

5.1.20 2 フェーズコミット（2PC）

複数のデータベースノード間で一貫性を達成するためのマルチステッププロセス。

5.1.21 垂直スケーリングまたはスケールアップ

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

5.1.22 書き込みスケーラビリティ

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

5.1.23 リーダーを書く

常時接続アーキテクチャでは、ノードはアプリケーションの正しい接続エンドポイントとして選択されます。このノードは書き込みリーダーと呼ばれます。使用するアプリケーションの書き込みリーダーを選択することにより、意図しないマルチノード書き込みを回避できます。書き込みリーダーは、プロキシノードのクォーラムのコンセンサスによって選択されます。書き込みリーダーが利用できなくなった場合、プロキシノードは別のノードを選択して書き込みリーダーになります。書き込みリーダーではないノードは、**シャドウノード** と呼ばれます。

5.2 PGD overview

EDB Postgres Distributed（PGD）は、マルチマスターレプリケーションとデータ分散を、高度な競合管理、データ損失保護、およびネイティブ論理レプリケーションよりも最大 5 倍高速なスループットを提供します。また、最大ファイブ・ナインの高可用性を備えた分散 Postgres クラスターを有効にします。

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

デフォルトでは、PGD は非同期レプリケーションを使用し、ローカルコミット後にのみピアノードに変更を適用します。複数の同期レプリケーションオプションも利用できます。

5.2.1 基本的なアーキテクチャ

複数のグループ

PGD ノードは、少なくとも 1 つの **ノードグループ**のメンバーです。最も基本的なアーキテクチャでは、PGD クラスター全体に単一のノードグループがあります。

マルチプルマスター

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

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

常にすべてのマスターに書き込む必要はありません。頻繁な構成では、主に
書き込みリーダーの表示 と呼ばれる 1 つのマスターのみに書き込みを指示します。

非同期、デフォルト

1 つの PGD 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レプリケートされません。その結果、データは常にすべてのノードで完全に同じではありません。一部のノードには、他のノードにまだ到着していないデータがあります。PostgreSQL のブロックベースのレプリケーションソリューションも、デフォルトで非同期レプリケーションになります。PGD には、複数のマスターがあり、その結果、複数のデータストリームがあります。そのため、`synchronous_commit` および `synchronous_standby_names` が使用されても、異なるノード上のデータが異なる場合があります。

メッシュトポロジ

PGD 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、PGD にはデータの転送はありません。データは、EDB Postgres 分散クラスターの外部から到着するか、ネイティブの PostgreSQL 論理レプリケーションを使用して送信できます。

論理レプリケーション

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

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

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

レプリケートされたデータは、安全な場合にバイナリ形式で送信されます。

接続管理

> **接続管理** は、コンセンサス駆動のクォーラムを活用して、半排他的な方法で正しい接続エンドポイントを決定し、アプリケーションからの意図しないマルチノード書き込みを防ぎます。このアプローチにより、データの競合の可能性が減少します。任意の時点で正しい接続エンドポイントとして選択されたノードは、**書き込みリーダーの表示** と呼ばれます。
>
> *Installing PGD Proxy* は、 EDB Postgres

Distributed の一部として提供されるアプリケーション接続管理用のツールです。

高可用性

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

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

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

DDL は、デフォルトでノード間でレプリケートされます。必要に応じて、DDL の実行をユーザー制御して、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。

5.2.2 アーキテクチャのオプションとパフォーマンス

Always On アーキテクチャ

多数の異なるアーキテクチャを構成でき、それぞれに異なるパフォーマンスとスケーラビリティ特性があります。

グループは、2 つ以上のノード（サーバー）で構成される基本的なビルディングブロックです。グループでは、各ノードは専用のルーターとバックアップを使用して異なるアベイラビリティーゾーンにあり、即時のスイッチオーバーと高可用性を提供します。各グループには、専用のレプリケーションセットが定義されています。グループがノードを失った場合、グループから既存のノードをコピーすることにより、簡単に修復または交換できます。

Always On アーキテクチャは、1 つの場所にある 1 つのグループ、または 2 つの別々の場所にある 2 つのグループから構築されます。各グループは高可用性を提供します。2 つのグループがリモートロケーションで活用される場合、それらは一緒に災害復旧（DR）も提供します。

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

各グループ内の 1 つのノードが、グループの書き込みリーダーとして選択されます。書き込みリーダーは、メインアプリケーションの書き込みとクエリのターゲットです。他のすべてのノードはシャドウノード（または「読み取り/書き込みレプリカ」）と呼ばれ、必要に応じて引き継ぐのを待っています。ノードが接続を失った場合、処理はすぐにシャドウノードに切り替わり、処理を続行します。グループに障害が発生した場合、処理は他のグループに切り替えることができます。スケーラビリティは、このアーキテクチャの目標ではありません。

書き込みは主に 1 つのノードのみに行われるため、ノード間の競合の可能性はほぼゼロに削減されます。その結果、パフォーマンスへの影響が大幅に削減されます。

セカンダリアプリケーションはシャドウノードに対して実行される場合がありますが、メインアプリケーションがそのノードの使用を開始すると、これらは削減または中断されます。

将来的には、1 つのノードが他のグループのメインレプリケーターとして選択され、クラスターの成長に合わせてレプリケーションの CPU オーバーヘッドが制限され、他のグループへの帯域幅が最小限に抑えられます。

サポートされている Postgres データベースサーバー

PGD は PostgreSQL 、 EDB Postgres Extended Server および EDB Postgres Advanced Server と互換性があり、BDR という名前の標準の Postgres 拡張機能としてデプロイされます。サポートされているバージョンの組み合わせの詳細については、

> *Compatibility matrix* を参照してください。

一部の主要な PGD 機能は、ターゲットの Postgres データベースサーバーで使用可能な特定のコア機能に依存します。したがって、PGD ユーザーは、ビジネスニーズに最適な Postgres データベースサーバーディストリビューションも採用する必要があります。たとえば、PGD 機能の Commit At Most Once （CAMO）を使用することがユースケースにとってミッションクリティカルである場合は、コミュニティの PostgreSQL ディストリビューションを採用しないでください。CAMO を処理するために必要なコア機能がありません。

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

PGD は、ネイティブに近い Postgres 互換性を提供します。ただし、一部のアクセスパターンは、マルチノードセットアップで必ずしも単一インスタンスで機能するとは限りません。マルチノード設定で安全に複製できるものにもいくつかの制限があります。

> *Application usage* は、アプリケーション開発の観点から PGD がどのように動作するかについて詳しく説明します。

パフォーマンスに影響する特性

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

データのコピーはどこにでもあるため、SELECT はローカルノードにアクセスするだけで済みます。読み取り専用クラスターでは、1 つのノードのパフォーマンスはノード数の影響を受けず、長時間実行される SELECT クエリによって引き起こされる他のノードでのレプリケーションの競合の影響を受けません。したがって、ノードを追加すると、可能な合計 SELECT スループットが直線的に増加します。

INSERT、UPDATE および DELETE（DML）がローカルで実行される場合、変更はグループ内のすべてのノードに伝播します。DML apply のオーバーヘッドは、元の実行よりも少ないです。そのため、複数のノードで純粋な書き込みワークロードを同時に実行する場合、マルチノードクラスターは単一ノードよりも多くの TPS を処理できます。

競合処理には、スループットを低下させるように作用するコストがあります。スループットは、アプリケーションが実際に表示する競合の量によって異なります。競合が非常に少ないアプリケーションは、単一ノードよりも優れたパフォーマンスを発揮します。競合の多いアプリケーションは、単一ノードよりもパフォーマンスが低下する可能性があります。これらの結果は、マルチマスターテクノロジーと一致しており、PGD に固有のものではありません。

同期レプリケーションオプションは、レプリケーションラグを最小限に抑えるように、複数のノードに同時に変更を送信できます。ノードを追加すると、レプリケーションにより多くの CPU を使用することになるため、ノードが追加されるたびにピーク TPS がわずかに低下します。

ワークロードがすべての CPU リソースを使用しようとする場合、このリソースはレプリケーションを制約し、レプリケーションラグに影響を与える可能性があります。

要約すると、PGD グループにマスターノードを追加しても、ほとんどのテーブルがレプリケートされる場合、すべての書き込みがすべてのノードでリプレイされるため、書き込みスループットが大幅に向上することはありません。PGD 書き込みは、一般に、SQL を介して Postgres クライアントからの書き込みよりも効果的であるため、パフォーマンスを向上させることができます。読み取りスループットは、通常、ノード数に比例して増加します。

第6章　Planning

6.1 Choosing your architecture

Always On アーキテクチャは、EDB の Trusted Postgres アーキテクチャを反映しています。これらはプラクティスをカプセル化し、複数の構成で可能な限り最高のサービス可用性を達成するのに役立ちます。これらの構成は、単一ロケーションアーキテクチャから、ハードウェア障害やデータセンター障害から保護する複雑な分散システムにまで及びます。このアーキテクチャは、EDB Postgres Distributed のマルチマスター機能と、メンテナンス操作中でも 99.999 %の可用性を達成するその機能を活用しています。

ここで説明されている例以外のアーキテクチャにも、 EDB Postgres Distributed を使用できます。ユースケース固有のバリエーションが本番環境に正常にデプロイされました。ただし、これらのバリエーションは、最初に厳密なアーキテクチャレビューを受ける必要があります。また、Always On アーキテクチャ用の EDB の標準デプロイツールでは、Trusted Postgres Architect（TPA）を有効にしてバリエーションをサポートしてからでないと、運用環境でサポートできません。

6.1.1 標準 EDB Always On アーキテクチャ

EDB は、目標復旧時点（RPO）と目標復旧時間（RTO）の要件に応じて、さまざまなレベルの冗長性を備えた単一または複数の場所の展開をサポートするための一連の標準化されたアーキテクチャを特定しました。

Always ON アーキテクチャは、基本的なビルディングブロックとして 3 つのデータベースノードグループを使用します。余分な冗長性のために 5 ノードグループを使用することもできます。

EDB Postgres Distributed は、次の主要なビルディングブロックで構成されています。

- 双方向レプリケーション（BDR）-マルチマスターメッシュネットワークを作成する Postgres 拡張機能
- PGD プロキシ-アプリケーションが適切なデータノードに接続されていることを確認する接続ルーター。

Always On アーキテクチャはすべて、ますます多様化する障害状況を保護します。たとえば、2 つのデータノードを持つ 1 つのアクティブな場所は、ローカルのハードウェア障害から保護しますが、場所（データセンターまたはアベイラビリティーゾーン）の障害からの保護は提供しません。別の場所にあるバックアップでそのアーキテクチャを拡張すると、場所の壊滅的な損失が発生した場合に備えて、ある程度の保護が保証されます。ただし、最初にバックアップからデータベースを復元する必要があります。これは、RTO 要件に違反する可能性があります。マルチマスターメッシュネットワークに接続された 2 番目のアクティブロケーションを追加すると、ロケーションがオフラインになった場合でもサービスが利用可能なままになります。最後に、3 番目の場所（これは witness 専用の場所にすることができます）を追加すると、1 つの場所がオフラインになった場合でも、グローバル Raft 機能が機能できます。グローバル Raft は、主に管理コマンドを実行するために必要です。また、DDL やシーケンス割り当てなどの一部の機能は、それ

なしでは動作しない場合がありますが、DML レプリケーションはグローバル Raft がなくても引き続き動作します。

データを少なくとも 1 つのローカルマスターに同期的にストリーミングできるため、各アーキテクチャはゼロ RPO を提供でき、ローカルハードウェアに障害が発生した場合のデータ損失ゼロを保証します。

可用性の保証を高めると、常にハードウェアとライセンス、ネットワーク要件、運用の複雑さのコストが増加します。アーキテクチャを選択する前に、可用性とコンプライアンスの要件を慎重に検討することが重要です。

6.1.2 アーキテクチャの詳細

デフォルトでは、アプリケーショントランザクションは DML（選択、挿入、更新、および削除）のクラスター全体のコンセンサスを必要としないため、レイテンシーが低下し、パフォーマンスが向上します。ただし、新しいグローバルシーケンスの生成や分散 DDL の実行などの特定の操作では、EDB Postgres Distributed は Raft ベースのコンセンサスモデルを使用して決定を行うために奇数のノードを必要とします。したがって、より単純なアーキテクチャでも、すべてがデータを格納しているわけではない場合でも、常に 3 つのノードがあります。

アプリケーションは、マルチホスト接続文字列を介して標準の Always On アーキテクチャに接続します。各 PGD プロキシサーバーは、マルチホスト接続文字列内の個別のエントリです。高可用性を確保するには、各場所に常に少なくとも 2 つのプロキシノードが必要です。プロキシをデータベースインスタンスと同じ場所に配置できます。その場合、すべてのデータノードにプロキシを配置することをお勧めします。

他の接続メカニズムは、本番環境に正常に展開されています。ただし、これらは標準の Always On アーキテクチャの一部ではありません。

Always On 1 つの場所

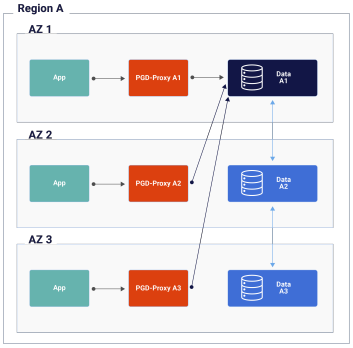


図 1: Always On 1 Location, 3 Nodes Diagram

*データノード 1 と 3 間の追加のレプリケーションは表示されませんが、レプリケーションメッシュの一部として発生します

ローカル障害から迅速に復元する冗長ハードウェア 3 つの PGD ノード *3* **つのデータノードにすることができます（推奨）** 2 つのデータノードとデータを保持しない 1 つのウィットネスにすることができます（図示せず）**アプリケーションへの親和性** データノードと併置可能

バックアップとリカバリの *Barman*（図示されていません）オフサイトはオプションですが、推奨 *複数のクラスターで共有できます

モニタリング用の *PostgresEnterpriseManager*（*PEM*）（図示せず）複数のクラスターで共有可能

Always On マルチロケーション

*アプリケーションは、各場所でアクティブ/アクティブにするか、1 つの場所のみが書き込みを行うアクティブ/パッシブまたはアクティブ DR にすることができます。

*データノード 1 と 3 間の追加のレプリケーションは表示されませんが、レプリケーションメッシュの一部として発生します。

ローカル障害から迅速に復元する冗長ハードウェア。合計 6 つの PGD ノード、各場所に 3 つ *3* **つのデータノードにすることができます（推奨）**2 つのデータノードとデータを保持しない 1 つの witness にできます（図示されていません）アプリケーション*データノードと同じ場所に配置できます

バックアップとリカバリの *Barman*（図示せず）。複数のクラスターで共有できます

監視用の *PostgresEnterpriseManager*（*PEM*）（図示せず）。複数のクラスターで共有できます

ロケーション障害の耐性を高めるには、オプションの監視ノードを *3* 番目のリージョンに配置する必要があります。そうでない場合、ロケーションに障害が発生すると、新しいノードの追加や分散 DDL など、グローバルコンセンサスを必要とするアクションがブロックされます。

6.1.3 アーキテクチャの選択

すべてのアーキテクチャは以下を提供します。

*ハードウェア障害保護

*ゼロダウンタイムのアップグレード

*パブリック/プライベートクラウドでのアベイラビリティーゾーンのサポート

これらの基準を使用して、適切な Always On アーキテクチャを選択できます。

	Single Data Location	Two Data Locations	Two Data Locations + Witness	Three or More Data Locations
Locations needed	1	2	3	3
Fast restoration of local HA after data node failure	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes	Yes - if 3 PGD data nodes No - if 2 PGD data nodes
Data protection in case of location failure	No (unless offsite backup)	Yes	Yes	Yes

次のページに続く

表 1 – 前のページからの続き

	Single Data Location	Two Data Locations	Two Data Locations + Witness	Three or More Data Locations
Global consensus in case of location failure	N/A	No	Yes	Yes
Data restore required after location failure	Yes	No	No	No
Immediate failover in case of location failure	No - requires data restore from backup	Yes - alternate Location	Yes - alternate Location	Yes - alternate Location
Cross-location network traffic	Only if backup is offsite	Full replication traffic	Full replication traffic	Full replication traffic
License cost	2 or 3 PGD data nodes	4 or 6 PGD data nodes	4 or 6 PGD data nodes	6+ PGD data nodes

6.1.4 標準アーキテクチャに柔軟性を追加する

必要なデータ回復性と、アプリケーションおよびデータを維持するユーザーへの近接性を提供するために、単一ロケーションアーキテクチャは、必要な数の場所に展開できます。EDB Postgres Distributed にはさまざまな競合処理アプローチが利用できますが、地理的に異なる場所からの書き込みアクティビティを許可する場合は、予想される衝突の数を最小限に抑えるように注意してください。

2 種類のノードを追加して、標準アーキテクチャを拡張することもできます。

- **サブスクライバー専用ノード**。これを使用すると、読み取りのスケーラビリティが向上し、アプリケーションのワークロードの大部分が読み取り集中型で書き込み頻度が低い場合に、データをユーザーの近くに配置できます。また、それらを活用して、レポート、アーカイブ、および分析のニーズに合わせてデータのサブセットを公開することもできます。
- **論理スタンバイ** 。PGD クラスター内の別のノードから複製されたデータを受信しますが、レプリケーションメッシュまたはコンセンサスには参加しません。これらには、他の PGD データノードと同じデータがすべて含まれており、データノードの 1 つがクラスターをフルキャパシティー/コンセンサスに戻せない場合、すぐにマスターに昇格できます。データセンター間のネットワークトラフィックが懸念される環境で使用できます。それ以外の場合、場所ごとに 3 つの PGD データノードが常に優先されます。

6.2 Choosing a Postgres distribution

EDB Postgres Distributed は、 PostgreSQL、 EDB Postgres Extended Server、または EDB Postgres Advanced Server の 3 つの異なる Postgres ディストリビューションでデプロイできます。特定の EDB Postgres Distributed 機能を利用できるかどうかは、使用されている Postgres ディストリビューションによって異なります。したがって、ビジネスニーズに最適な Postgres ディストリビューションを採用することが不可欠です。たとえば、Commit At Most Once （CAMO）機能を使用することがユースケースにとってミッションクリティカルである場合は、CAMO を処理するために必要なコア機能を備えていないオープンソースの PostgreSQL を採用しないでください。

次の表に、Postgres ディストリビューションとバージョンに依存する EDB Postgres Distributed の機能をリストします。

Feature	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
Rolling application and database upgrades	Y	Y	Y
Upgrading within version 3.7	Y	Y	Y
DML および DDL レプリケーション	Y	Y	Y
Upgrading within version 3.7	Y	Y	Y
Upgrading within version 3.7	v14+	v13+	v14+
Upgrading within version 3.7	Y	Y	Y
Upgrading within version 3.7	Y	Y	Y
Monitoring	Y	Y	Y
Upgrading within version 3.7	Y	Y	Y
Parallel Apply	Y	Y	Y
Upgrading within version 3.7	Y	Y	Y
列レベルの競合解決の有効化と無効化	Y	Y	Y
トリガーの変換	Y	Y	Y
競合トリガー	Y	Y	Y
非同期レプリケーション	Y	Y	Y
PGD **を使用したレガシー同期レプリケーション**	Y	Y	Y
Group Commit	N	Y	14+

次のページに続く

表 2 – 前のページからの続き

Feature	PostgreSQL	EDB Postgres Extended	EDB Postgres Advanced
Upgrading within version 3.7	N	Y	14+
Eager conflict resolution	N	Y	14+
Lag Control	N	Y	14+
デコードワーカー	N	13+	14+
Upgrading within version 3.7	N	Y	14+
Missing partition conflict	N	Y	14+
No need for UPDATE Trigger on tables with TOAST	N	Y	14+
Automatically hold back FREEZE	N	Y	14+
Transparent Data Encryption	N	15+	15+

6.3 Choosing your deployment method

次の方法を使用して、 EDB Postgres 分散製品を展開およびインストールできます。

- 信頼できる *Postgres* アーキテクト （TPA）は、Ansible を使用して、さまざまなシナリオで Postgres をセットアップおよび操作する方法を文書化したリファレンスアーキテクチャのセットを使用して Postgres クラスターを構築するオーケストレーションツールです。TPA は、EDB が続くベストプラクティスを表しており、その推奨事項は、実稼働環境と同じように、クイックテストベッドのセットアップに適用されます。詳細については、 *Deploying with TPA* を参照してください。

- BigAnimal は、Oracle との互換性が組み込まれたフルマネージドの Database-as-a-Service であり、Postgres の専門家によって運用されているクラウドアカウントで実行されます。BigAnimal を使用すると、データベースのセットアップ、管理、およびスケーリングが簡単になります。EDB Postgres Distributed（PGD）を介した超高可用性サポートを追加すると、シングルリージョンの Always On Gold クラスターが可能になります。これらのクラスターは、単一のクラウドリージョン内の異なるアベイラビリティーゾーンにある 2 つの PGD グループであり、3 番目のアベイラビリティーゾーンに監視ノードがあります。詳細については、 BigAnimal documentation の Extreme high availability を参照してください。

近日公開：

- EDB Postgres Distributed for Kubernetes は、EDB によって設計、開発、およびサポートされる Kubernetes オペレーターです。PGD レプリケーションを使用して、マルチマスターアーキテクチャを備えた高可用性 Postgres データベースクラスターのライフサイクル全体をカバーします。オープンソースの CloudNativePG オペレーターに基づいており、EDB Postgres Advanced Server を使用した Oracle との互換性や、IBM Power や OpenShift を含む追加のサポートされているプラットフォームなど、追加の価値を提供します。

6.4 Other considerations

展開を計画するときは、これらの他の考慮事項を確認してください。

6.4.1 データの一貫性

CRDT データ型を使用した列の競合の処理 について読んで、データの一貫性の観点からの非同期操作モードの影響を理解してください。

6.4.2 展開

PGD は、 信頼できる *Postgres* アーキテクトまたはテクニカルサポートによって承認された構成管理アプローチと展開アーキテクチャのいずれかを使用して、少数の既知の良好な構成の 1 つで展開されることを目的としています。

手動展開はお勧めできず、サポートされていない場合があります。

ログメッセージとドキュメントは現在、英語でのみ提供されています。

6.4.3 サイジングの考慮事項

本番デプロイメントの場合、EDB は、Postgres データノードごとに最小 4 コアをお勧めします。Witness ノードはデータレプリケーションオペレーションに参加しないため、この要件を満たす必要はありません。サブグループ Raft なしで 1 つのコアで十分です。サブグループ Raft を使用する場合、2 つのコアで十分です。ノード昇格の場合のパフォーマンスの低下を回避するために、ロジカルスタンバイのサイズは常にデータノードとまったく同じにしてください。運用展開では、PGD プロキシノードには少なくとも 1 つのコアが必要であり、データベースコアの数を約 1:10 の比率で増加させながら段階的に増やす必要があります。特定のパフォーマンス要件の詳細なベンチマークを行って、ワークロードに基づいて適切なサイジングを決定することをお勧めします。必要に応じて、EDB プロフェッショナルサービスチームが支援します。

開発目的では、2 コア未満の Postgres データノードを割り当てないでください。Barman ノードのサイズは、データベースのサイズとデータ変更率によって異なります。

Postgres データノード、 Barman ノード、および PGD Proxy ノードを仮想マシンに、またはベアメタル展開モードで展開できます。ただし、復元性が低下するため、同じ物理ハードウェア上にある VM に複数のデータノードを展開しないでください。また、同じ物理ハードウェア上の VM に複数の PGD プロキシノードを展開しないでください。これも、復元性が低下するためです。

単一の PGD プロキシノードは、単一の PGD データノードと同じ場所に配置できます。

6.4.4 時計とタイムゾーン

EDB Postgres Distributed は、マルチプルのタイムゾーンのノードで動作するように設計されているため、真にワールドワイドなデータベースクラスターが可能になります。個々のサーバーを一致するタイムゾーンで構成する必要はありませんが、`log_timezone = UTC` を使用して、人間が読めるサーバーログにアクセスしやすく、比較できるようにすることをお勧めします。

NTP またはその他のソリューションを使用して、サーバーのクロックを同期します。

他のいくつかのソリューションの場合と同様、クロック同期はパフォーマンスにとって重要ではありません。クロックスキューは、オリジンの競合検出に影響を与える可能性がありますが、EDB Postgres Distributed は、存在するスキューを報告および管理するコントロールを提供します。

Column-level conflict detection で説明されているように、 EDB Postgres Distributed は、行バージョンの競合検出も提供します。

6.5 Limitations

展開を計画するときは、これらの EDB Postgres Distributed（PGD）の設計制限を考慮に入れてください。

6.5.1 ノード

- PGD は、適切なハードウェアとネットワークを前提に、数百のノードを実行できます。ただし、メッシュベースの展開の場合、通常、1 つのクラスターで 48 を超えるノードを実行することはお勧めしません。48 ノードの制限を超える追加の読み取りスケーラビリティが必要な場合は、メッシュネットワークに接続を追加せずにサブスクライバー専用ノードを追加できます。
- PGD には現在、1000 アクティブノード以下のハード制限があります。これは、これが許可される現在の最大 Raft 接続であるためです。
- グループ内のノードの推奨最小数は 3 です。PGD のコンセンサスメカニズムにフォールトトレランスを提供します。2 つのノードだけで、ノードの 1 つが応答しないとコンセンサスは失敗します。分散シーケンス生成など、一部の PGD 操作にはコンセンサスが必要です。EDB Postgres Distributed によって使用されるコンセンサスメカニズムの詳細については、*Architectural details* を参照してください。

6.5.2 単一インスタンス上の複数のデータベース

同じ Postgres インスタンス上の複数のデータベースに PGD を使用するためのサポートは、PGD 5 以降で **非推奨** になり、PGD 6 でサポートされなくなります。マルチデータベース設計でより長く実行可能。

これがベストプラクティスであり、PGD インスタンスごとに 1 つのデータベースのみを構成することをお勧めします。

TPA による展開の自動化と、CLI や PGD プロキシなどのツールは、既にその推奨事項を成文化しています。

1 つのインスタンスで最大 10 個のデータベースをホストすることは引き続き可能ですが、そうすると多くの即時のリスクと現在の制限が発生します。

- PGD 構成の変更が必要な場合、データベースごとに管理コマンドを実行する必要があります。そうすると、潜在的な不整合やエラーのリスクが高まります。
- 各データベースを個別に監視する必要があり、オーバーヘッドが追加されます。
- TPAexec は 1 つのデータベースを想定しています。より多くのデータベースのレプリケーションを設定するには、顧客または EDB プロフェッショナルサービスチームが展開後のフックで追加のコーディングが必要です。
- PGD プロキシは、データベースレベルではなく Postgres インスタンスレベルで動作します。つまり、リーダーノードはすべてのデータベースで同じです。
- データベースが追加されるたびに、サーバーのリソース要件が増加します。それぞれには、レプリケーションを維持する独自のワーカープロセスのセット、たとえば、論理ワーカー、WAL 送信者、WAL 受信者が必要です。また、それぞれには、レプリケーションクラスター内の他のインスタンスへの独自の接続セットが必要です。これらのニーズは、すべてのデータベースのパフォーマンスに深刻な影響を与える可能性があります。

・ノードの再構築または追加を行う場合、1 つのデータベースの物理初期化方法 (`bdr_init_physical`) は 1 ノードのみ利用できます。他のすべてのデータベースは論理レプリケーションで初期化する必要があります。これは、時間がかかるため大規模なデータベースでは問題になる可能性があります。

CAMO や Group Commit などの同期レプリケーション方法は、期待どおりに機能しません。Postgres WAL はデータベース間で共有されるため、同期コミット確認は、必ずしも正しいコミットの順序でなく、任意のデータベースから来ることができます。

- CLI と OTEL の統合（v5 の新機能）は 1 つのデータベースを前提としています。

6.5.3 CAMO

> *Commit At Most Once* （CAMO）は、アプリケーションが複数回コミットするのを防ぐことを目的とした機能です。この機能を使用する場合は、計画時に次の制限を考慮してください。

CAMO は、オリジンノードで最近失敗した COMMIT の結果を照会するように設計されています。そのため、切断された場合には、アプリケーションをコーディングして、CAMO パートナーからトランザクションステータスをすぐに要求します。障害が発生してからステータスを要求するまでの遅延をできるだけ短くします。アプリケーションは、CAMO の決定が 15 分以上保存されることに依存してはなりません。

- アプリケーションが、たとえば、再起動の結果として、割り当てられたグローバル識別子を忘れた場合、それを回復する簡単な方法はありません。したがって、アプリケーションは、未処理のトランザクションが終了するのを待ってからシャットダウンすることをお勧めします。
- クライアントが適切なチェックを適用するには、CAMO によって保護されたトランザクションを、暗黙的なトランザクション制御を持つ単一のステートメントにすることはできません。トランザクション制御プロシージャまたはトランザクションを開始または終了しようとする `DO` ブロックで CAMO を使用することもできません。

CAMO はコミットステータスを解決しますが、コミット時に保留中の通知を解決しません。CAMO および Eager Replication オプションでは、 `NOTIFY` SQL コマンドまたは `pg_notify()` 関数は許可されません。`LISTEN` または `UNLISTEN` も許可しません。

- 変更をリプレイするとき、CAMO トランザクションは他のトランザクションと同じように競合を検出する場合があります。タイムスタンプ競合検出を使用する場合、CAMO トランザクションは、オリジンノードでの準備のタイムスタンプを使用します。これは、トランザクションがオリジンノード自分自身で可視になる前です。
- CAMO は現在、トランザクションストリーミングと互換性がありません。CAMO の使用を計画している場合は、トランザクションストリーミングを必ず無効にしてください。このオプションは、グローバルに構成するか、PGD ノードグループで構成できます。*Transaction streaming configuration* を参照してください。

6.5.4 その他の制限

この非包括的なリストには、設計による予想されるその他の制限が含まれています。将来的にそれらを解決する予定はありません。展開を計画するときは、次の制限を考慮してください。

- ノードをフィジカルスタンバイに置き換えても、 CAMO/ Eager / Group Commit を使用するノードでは機能しません。たとえ可能であっても、フィジカルスタンバイと EDB Postgres Distributed を組み合わせることはお勧めしません。
- `galloc` シーケンスは、ロールバックされたトランザクションでシーケンスを作成し、同じ名前で再度作成すると、一部のチャンクをスキップする場合があります。DDL レプリケーションがアクティブでないときにシーケンスを作成して削除し、DDL レプリケーションがアクティブなときに再度作成した場合にも、チャンクのスキップが発生する可能性があります。シーケンスの保証に違反しないため、問題の影響は軽度です。シーケンスは、一部の最初のチャンクのみをスキップします。また、回避策として、シーケンスの開始値を `bdr.alter_sequence_set_kind()` 関数への引数として指定できます。
- レガシー同期レプリケーションは、 CAMO、 Eager、および Group Commit で使用されるものとは異なるトランザクション確認のメカニズムを使用します。この 2 つは互換性がありませんので、一緒に使用しないでください。
- Postgres の 2 フェーズコミット（2PC）トランザクション（つまり、*PREPARE TRANSACTION* ）は、CAMO、グループコミット、または Eager Replication と一緒に使用できません。これらの機能は下で 2 フェーズコミットを使用するためです。

第7章　Installing

7.1 Deploying EDB Postgres Distributed using Trusted Postgres Architect

自己管理設定で EDB Postgres Distributed を展開する標準的な方法は、EDB の展開ツール： **信頼できる *Postgres* アーキテクト**（TPA）を使用することです。これは、セルフホストとクラウド（EC2）の両方の物理マシンと仮想マシンに適用されます。

無料トライアルアカウントと
: **信頼できる *Postgres* アーキテクト**（TPA）から *deploy directly to your own bare metal servers* を使用することもできます。

7.1.1 前提条件：TPA をインストールする

TPA を使用して PGD を展開する前に、TPA をインストールする必要があります。続行する前に *installation instructions in the Trusted Postgres Architect documentation* に従ってください。

7.1.2 構成する

tpaexec configure コマンドは、選択したオプションに基づいて、クラスターを説明する単純な YAML 構成ファイルを生成します。構成はすぐに使用する準備ができており、ニーズに合わせて変更できます。構成ファイルの編集は、クラスターの作成前と作成後の両方に構成を変更する通常の方法です。

構文は次のとおりです。

```
tpaexec configure <cluster_dir> --architecture <architecture_name> [options]
```

使用可能な構成オプションは次のとおりです。

表 1: :header: Flags,Description :widths: 10,30 :align: left :class: longtable

--architecture	必須。 EDB Postgres 分散展開の場合は *PGD-Always-ON* に設定します。
–-postgresql <version> or *--edb-postgres-advanced <version>* or *--edb-postgres-extended <version>*	必須。使用する Postgres のディストリビューションとバージョンを指定します。詳細については、 *Upgrading within version 3.7* を参照してください。
--redwood or *--no-redwood*	`--edb-postgres-advanced`フラグが存在する場合は必須。 Oracle データベースの互換性機能が必要かどうかを指定します。
--location-names l1 l2 l3	必須。 PGD を展開する場所の名前を指定します。
--data-nodes-per-location N	ロケーションごとのデータノードの数を指定します。デフォルトは 3 です。
--add-witness-node-per-location	ロケーションごとに偶数のデータノードの場合、witness ノードを追加してローカルコンセンサスを可能にします。 2 つのデータノードの場所でデフォルトで有効になっています。
--add-proxy-nodes-per-location	PGD プロキシをデータノードから分離するかどうか、および構成する数。デフォルトでは、データノードごとに 1 つのプロキシが構成され、共ホストされます。
--pgd-proxy-routing globalIlocal	PGD プロキシルーティングがグローバルまたはローカル（ロケーションごと）で処理されるかどうか。
--add-witness-only-location loc	クラスターの場所の 1 つを監視専用として指定します（その場所にデータノードは存在しません）。
--enable-camo	各ロケーションに CAMO ペアをセットアップします。ロケーションごとに 2 つのデータノードでのみ機能します。
PGD-Always-ON の TPA ドキュメントには、より多くの構成オプションがリストされています。	

例：

```
[tpa]$ tpaexec configure ~/clusters/speedy \
         --architecture PGD-Always-ON \
         --platform aws \
         --edb-postgres-advanced 15 \
         --redwood \
         --location-names eu-west-1 eu-north-1 eu-central-1 \
         --data-nodes-per-location 3 \
         --pgd-proxy-routing global
```

最初の引数は、クラスターディレクトリ、たとえば、`speedy` または`~/clusters/speedy` である必要があります。（クラスターの名前はどちらの場合も `speedy` です。）すべてのクラスターを`~/clusters` などの共通ディレクトリに保持することをお勧めします。アーキテクチャを選択するには、次の引数が`--architecture` である必要があり、その後にオプションが続きます。

このコマンドは、 `~/clusters/speedy` という名前のディレクトリを作成し、PGD-Always-ON アーキテ

クチャのレイアウトに従った config.yml という名前の構成ファイルを生成します。tpaexec configure --architecture PGD-Always-ON --help コマンドを使用して、このアーキテクチャの構成オプションでサポートされている値を確認できます。

この例では、オプションは以下を選択します。

- AWS デプロイメント（--platform aws ）
- EDB Postgres Advanced Server、バージョン 15 および Oracle 互換性（--edb-postgres-advanced 15 および--redwood ）
- 3 か所 (--location-names eu-west-1 eu-north-1 eu-central-1)
- 各場所に 3 つのデータノード（--data-nodes-per-location 3 ）
- グローバルのプロキシルーティングポリシー (--pgd-proxy-routing global)

一般的な構成オプション

その他の構成オプションには次のものがあります。

所有者

すべてのクラスターは、プロビジョニングされたリソースの責任者を直接追跡できる必要があります。

デフォルトでは、クラスターは、 tpaexec provision を実行しているユーザーのログイン名が所有しているとしてタグ付けされます。この名前で個人を識別できない場合（たとえば、postgres 、ec2-user ）、--owner SomeId を指定して識別可能な所有者を設定する必要があります。

イニシャル、「Firstname Lastname」、またはあなたを一意に識別するテキストを使用できます。

プラットフォームオプション

- -platform のデフォルト値は aws で、これは PGD-Always-ON アーキテクチャでサポートされているプラットフォームです。

--region を指定して、アクセス権があり、必要な数のインスタンスを作成できる既存の AWS リージョンを指定します。デフォルトのリージョンは eu-west-1 です。

AWS の有効なインスタンスタイプで--instance-type を指定します。デフォルトは t3.micro です。

サブネット選択

デフォルトでは、各クラスターには 10.33/16 の下のランダムな/28 サブネットが割り当てられます。ただし、アーキテクチャによっては、1 つ以上のサブネットが存在する可能性があり、各サブネットは/24 と/29 の間のどこにでも存在できます。

`--subnet 192.0.2.128/27` など、特定のサブネットを使用するには`--subnet` を指定します。

ディスクスペース

- -root-volume-size を指定して、ルートボリュームのサイズを GB 単位で設定します。例、–root-volume-size 64。デフォルトは 16GB です。インスタンスの作成に使用されるイメージによっては、ルートボリュームの最小サイズがある場合があります。

Postgres と Barman の別々のボリュームをサポートするアーキテクチャの場合：

- `--postgres-volume-size` を指定して、Postgres ボリュームのサイズを GB 単位で設定します。デフォルトは 16GB です。
- `--barman-volume-size` を指定して、Barman ボリュームのサイズを GB 単位で設定します。デフォルトは 32GB です。

配布

- -os または–distribution を指定して、クラスターのインスタンスで使用する OS を指定します。値では大文字と小文字が区別されます。

選択したプラットフォームによって、使用可能なディストリビューションとデフォルトで使用されるディストリビューションが決まります。詳細については、`tpaexec info platforms/<platformname>` を参照してください。

一般に、 `Debian` 、 `RedHat` 、および `Ubuntu` を使用して、Postgres およびその他のソフトウェアがプリインストールされている TPA イメージを選択できます（展開時間を短縮するため）。代わりに在庫分布画像を使用するには、`-minimal` を値に追加します（例、`--distribution Debian-minimal` ）。

リポジトリ

TPA を使用して PDG 5 以降を展開する場合、TPA は EDB Repos 2.0 からリポジトリを選択します。すべてのソフトウェアはこれらのリポジトリからソースされています。

> EDB Repos 2.0 を使用するには、 tpaexec を実行する前に

`export EDB_SUBSCRIPTION_TOKEN=xxx` を使用する必要があります。 the web interface からサブスクリプショントークンを取得できます。

必要に応じて、 `--edb-repositories repository …`を使用して、各インスタンスにインストールするデフォルトのリポジトリに加えて EDB リポジトリを指定します。

ソフトウェアバージョン

デフォルトでは、TPA は Postgres の最新メジャーバージョンを使用します。–postgres-version を指定して、サポートされている以前のメジャーバージョンをインストールするか、

リポジトリを構成する

で説明されているフラグのいずれかを使用してバージョンとディストリビューションの両方を指定します。

デフォルトでは、TPA はすべてのパッケージの最新バージョンをインストールしますが、これは通常望ましい動作です。ただし、一部のテストシナリオでは、特定のパッケージバージョンを選択する必要がある場合があります。例：

```
- --postgres-package-version 10.4-2.pgdg90+1
- --repmgr-package-version 4.0.5-1.pgdg90+1
- --barman-package-version 2.4-1.pgdg90+1
- --pglogical-package-version 2.2.0*
- --bdr-package-version 3.0.2*
- --pgbouncer-package-version 1.8*
```

--extra-packages または--extra-postgres-packages を指定して、さらにパッケージをインストールします。前者は、システムパッケージとともにインストールするパッケージをリストします。後者には、Postgres パッケージとともに後でインストールするパッケージがリストされています。（前のリストで Postgres に依存するパッケージに言及した場合、Postgres はまだインストールされていないため、インストールは失敗します。）引数は、変更なしでインストールのためにパッケージマネージャーに渡されます。

--extra-optional-packages オプションは--extra-packages と同様に動作しますが、指定されたパッケージをインストールできない場合、エラーではありません。

ホスト名

デフォルトでは、 tpaexec configure は、数十の名前の事前承認リストから必要な数のホスト名をランダムに選択します。これは、ほとんどのクラスターで十分です。

--hostnames-from を指定して、提供されたリストで使用可能なより多くの名前が必要な場合など、別のリストから名前を選択します。ファイルには、1 行に 1 つのホスト名を含める必要があります。

--hostnames-pattern を指定して、ホスト名を egrep-syntax パターンに一致するものに制限します。これを行う場合、パターンが有効なホスト名（[a-zA-Z0-9-]）のみと一致し、十分なホスト名を見つけることを確認する必要があります。

ロケーション

デフォルトでは、 tpaexec configure は、選択したアーキテクチャで使用される場所に first、second などの名前を使用します。

`--location-names` を指定して、各場所により意味のある名前を提供します。

7.1.3 規定

tpaexec provision コマンドは、クラスターに必要なインスタンスとその他のリソースを作成します。プロセスの詳細は、クラスターの構成中に選択したアーキテクチャ（PGD-Always-ON など）とプラットフォーム（AWS など）によって異なります。

たとえば、必要な権限を持つ AWS アクセスが与えられると、TPA は EC2 インスタンス、VPC、サブネット、ルーティングテーブル、インターネットゲートウェイ、セキュリティグループ、EBS ボリューム、Elastic IP などをプロビジョニングします。

`bare` プラットフォームを選択し、接続の詳細を提供することにより、既存のサーバーをプロビジョニングすることもできます。これらがベアメタルサーバーであろうと、クラウドプラットフォームで個別にプロビジョニングされたものであろうと、それらは TPA によって作成されたかのように使用できます。

単一のプラットフォームに制限されません。必要に応じて、マルチプルリージョンの一部の AWS インスタンスと、他のデータセンターの一部のオンプレミスサーバーまたはサーバーにクラスターを分散できます。

プロビジョニング段階の最後に、基本的なオペレーティングシステムがインストールされた必要な数のインスタンスが作成されます。TPA は SSH を使用してアクセスできます（ルートに sudo を使用）。

7.1.4 展開する

tpaexec deploy コマンドは、プロビジョニングされたサーバーに Postgres およびその他のソフトウェアをインストールおよび構成します。TPA はサーバーを作成できますが、SSH および sudo アクセスが利用可能な限り、誰がそれらを作成したかは関係ありません。これには、レプリケーション、バックアップなどの設定が含まれます。

展開段階の最後に、 EDB Postgres Distributed が稼働しています。

7.1.5 テスト

tpaexec test コマンドは、デプロイされたクラスターに対してさまざまなアーキテクチャおよびプラットフォーム固有のテストを実行し、期待どおりに動作していることを確認します。

テスト段階の最後には、完全に機能するクラスターがあります。

詳細については、 信頼できる *Postgres* アーキテクト を参照してください。

7.2 Upgrading

EDB Postgres Distributed は複数のソフトウェアコンポーネントで構成されているため、アップグレード戦略は、アップグレードされるコンポーネントによって部分的に異なります。

一般に、ローリングアップグレードと呼ばれるアプローチを使用すると、ダウンタイムをほぼゼロでクラスターをアップグレードできます。このアプローチを使用すると、ノードが 1 つずつアップグレードされ、アプリケーション接続が既にアップグレードされているノードに切り替えられます。

すべてのノードを停止し、すべてのノードでアップグレードを実行してから、クラスター全体を再起動することもできます。このアプローチは、標準の PostgreSQL セットアップと同じです。すべてのノードを同時にアップグレードするこの戦略は、ソフトウェアの混合バージョンでの実行を回避するため、最も簡単です。ただし、ダウンタイムが発生するため、何らかの理由でローリングアップグレードを実行できない場合を除き、お勧めしません。

EDB Postgres 分散クラスターをアップグレードするには：

1. アップグレードを計画します。

    2. アップグレードの準備をします。

3. サーバーソフトウェアをアップグレードします。

    4. アップグレードを確認および検証します。

7.2.1 アップグレード計画

各ノードをアップグレードするには、大きく 2 つの方法があります。

***ノードを新しいソフトウェアバージョンにアップグレードします。**
ローリングサーバーソフトウェアのアップグレード を参照してください。

***ノードを新しいバージョンがインストールされているものに置き換えます。**
ノード参加を使用したローリングアップグレード を参照してください。

これらのアプローチの両方をローリング方式で使用できます。

ローリングアップグレードの考慮事項

クラスターのローリングアップグレード中に、ソフトウェアの混合バージョンがクラスターで実行されています。たとえば、nodeA に PGD 3.7.16 があり、nodeB と nodeC に 4.1.0 があるとします。この状態では、レプリケーションとグループ管理は最も古いバージョン（この例では 3.7.16）のプロトコルと機能を使用するため、プロトコルの変更を必要とする新しいバージョンが提供する新機能は無効になります。すべてのノードが同じバージョンにアップグレードされると、新機能が有効になります。

同様に、WAL デコーダー対応ノードを含むクラスターがローリングアップグレードを行っている場合、より高いバージョンの PGD ノードの WAL デコーダーは、より高い pglogical バージョンの LCR を生成します。PGD ノードのより低いバージョンの WAL デコーダーは、より低い pglogical バージョンの LCR を生成します。その結果、上位バージョンの PGD ノード上の WAL 送信者は、プロトコルバージョンの不一致のため、LCR を使用すると予想されません。PGD ノードのより低いバージョンでは、WAL 送信者は LCR

を使い続ける場合があります。すべての PGD ノードが同じ PGD バージョンになると、WAL 送信者は LCR を使用します。

ローリングアップグレードは、すべてのノードが以前のリリースであるクラスターから始まります。次に、すべてのノードが新しいリリースになるまで、一度に 1 つのノードを新しいリリースにアップグレードします。コンポーネントの 3 つ以上のバージョンを同時に実行してはなりません。この要件は、以前のアップグレードプロセスがすべてのノードで完全に完了するまで、新しいアップグレードを開始しないことを意味します。

ビジネスリスクを軽減するために注意が必要な場合、アップグレードプロセスにはさらに時間がかかる場合があります。ただし、ソフトウェアの混合バージョンを無期限に実行することはお勧めしません。

ソフトウェアのメジャーバージョンのアップグレードにローリングアップグレードを使用できますが、1 つのクラスターでの PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server の混合はサポートされていません。したがって、このアプローチを使用して Postgres バリアントを変更することはできません。

警告: EDB Postgres Distributed のダウングレードはサポートされていません。クラスターを手動でリビルドする必要があります。

ローリングサーバーソフトウェアのアップグレード

ローリングアップグレードとは、クラスターの残りの部分の動作を維持しながら、クラスター内の各ノードで **ローリングサーバーソフトウェアのアップグレード** プロセスが順に実行されるプロセスです。

実際の手順は、Postgres コンポーネントが新しいメジャーバージョンにアップグレードされているかどうかによって異なります。

アップグレードプロセス中に、現在アップグレードされていないノードにアプリケーションを切り替えて、アプリケーションのデータベースの継続的な可用性を提供できます。

ノード参加を使用したローリングアップグレード

サーバーソフトウェアをアップグレードする別の方法は、新しいノードをクラスターに参加させ、後で古いバージョンのソフトウェアを実行している既存のノードの 1 つを削除することです。

このアプローチの場合、手順は常に同じです。ただし、ノードジョインが含まれるため、潜在的に大規模なデータ転送が必要になります。

すべてのノードが Postgres のより新しい同じリリースにアップグレードされるまで、新しい Postgres バージョンでのみ利用可能な機能を使用しないように注意してください。これは、Postgres の新しいリリースに追加された新しい DDL 構文に特に当てはまります。

注釈: *bdr_init_physical* はソースノードのバイトごとのコピーを作成するため、Postgres のメジャーバージョンから別のメジャーバージョンへのアップグレード中に使用することはできません。実際、現在 `bdr_init_physical` では、ソースの PGD バージョンと参加ノードがまったく同じである必要があります。新

しいノードに参加する方法によるローリングアップグレードには使用できません。代わりに、論理結合を使用してください。

CAMO 対応クラスターのアップグレード

CAMO 対応クラスターをアップグレードするには、CAMO グループを 1 つずつアップグレードしながら、アップグレードするグループの CAMO 保護を無効にし、新しい

> *commit* スコープへの移行 ベースの設定を使用して再構成する必要があります。

CAMO ペアを構成する 2 つの BDR ノードを PGD 5.0 にアップグレードするには、次のアプローチをお勧めします。

- 2 つのノードのいずれかでのトランザクションで `bdr.enable_camo` が `off` のままであることを確認するか、クライアントを 2 つのノードからリダイレクトします。CAMO を使用しようとしているときに CAMO ペアリングを削除すると、エラーが発生し、それ以上のトランザクションができなくなります。
- `bdr.camo_origin_for` および `bdr.camo_parter_of`（ BDR 3.7.x からアップグレードする場合）をリセットするか、（ BDR 4.x で）`bdr.remove_camo_pair` を使用して、CAMO を構成解除して、ペアを分離します。
- 2 つのノードを PGD 5.0 にアップグレードします。

・2 つのノード専用のノードグループを作成し、そのノードグループに移動します。

- このノードグループに *commit* スコープへの移行 を作成し、CAMO を使用するノードのペアを作成します。
- `default_commit_scope` を設定するか、セッションまたはトランザクションに `bdr.enable_camo` ではなく `bdr.commit_scope` を明示的に設定するようにクライアントを変更して、CAMO 保護を再度アクティブにします。
- 必要に応じて、クライアントが CAMO 保護ノードに再度接続できるようにします。

7.2.2 アップグレードの準備

ソフトウェアの各メジャーリリースには、以前のリリースとの互換性性に影響を与える可能性のあるいくつかの変更が含まれています。これらは、Postgres 構成、展開スクリプト、および PGD を使用するアプリケーションに影響を与える可能性があります。アップグレードの前に、これらの変更を検討し、必要な調整を行うことをお勧めします。

> *EDB Postgres Distributed Release notes* およびバージョン固有のアップグレードノートに記載されている個々の変更を参照してください。

7.2.3 サーバーソフトウェアのアップグレード

個々のノードに分散された EDB Postgres のアップグレードは、その場で行われます。BDR 拡張機能をアップグレードするときに、バックアップと復元する必要はありません。

BDR 拡張機能のアップグレード

BDR 拡張機能のアップグレードプロセスは、いくつかの手順で構成されています。

Postgres を停止します

バイナリパッケージのアップグレード中は、通常、実行中の Postgres サーバーを最初に停止するのが最善です。そうすることで、アップグレード中に予期せず再起動した場合に混合バージョンがロードされなくなります。

パッケージのアップグレード

アップグレードの最初の手順は、新しいバージョンの BDR パッケージをインストールすることです。このインストールにより、新しいバイナリと拡張機能 SQL スクリプトの両方がインストールされます。この手順は、オペレーティングシステムに固有です。

Postgres を起動します

パッケージをアップグレードしたら、 Postgres インスタンスを起動できます。BDR 拡張機能は、新しいバイナリが拡張機能の古いバージョンを検出すると、起動時にアップグレードされます。

Postgres のアップグレード

Postgres のインプレースアップグレードのプロセスは、Postgres の新しいマイナーバージョンにアップグレードするか、Postgres の新しいメジャーバージョンにアップグレードするかによって異なります。

マイナーバージョン Postgres アップグレード

Postgres の新しいマイナーバージョンへのアップグレードは、 *upgrading the BDR extension* に似ています。通常、必要なのは Postgres を停止し、パッケージをアップグレードして、Postgres を再起動することだけです。

ただし、特定のマイナーバージョンのアップグレードには、インデックスの再作成などのより多くの手順が推奨される場合があります。アップグレードする Postgres のバージョンのリリースノートを参照してください。

メジャーバージョン Postgres アップグレード

Postgres の新しいメジャーバージョンへのアップグレードは、マイナーバージョンへのアップグレードよりも複雑です。

EDB Postgres Distributed は、 *In-place Postgres major version upgrades* を実行するために使用できる bdr_pg_upgrade コマンドラインユーティリティを提供します。

注釈: Postgres、 BDR 拡張機能などを含むソフトウェアの新しいメジャーバージョンにアップグレードするときは、アップグレードするソフトウェアのターゲットバージョンとアプリケーションの互換性を確保することが常に重要です。

7.2.4 アップグレードのチェックと検証

PGD ノードをアップグレードした後、バイナリの現在のバージョンを確認できます。

```
SELECT bdr.bdr_version();
```

ノードをアップグレードした後は常に *Monitoring* をチェックして、アップグレードされたノードが期待どおりに動作していることを確認します。

7.2.5 HARP から PGD プロキシへの移行

HARP は、しばらくの間、新しい **接続管理** 構成と共存できます。これは、次のことができることを意味します。

- 5 より前のクラスター全体を PGD 5 クラスターにアップグレードします。
- 接続ルーティングを設定します。
- HARP プロキシを PGD プロキシに置き換えます。
- アプリケーション接続を PGD プロキシインスタンスに移動します。
- すべてのサーバーから HARP Manager を削除します。

ノードを PGD 5 にアップグレードした後、できるだけ早くこれを行うことを強くお勧めします。HARP は、PGD 5 での長期使用が保証されていません。

TPA は、これに役立つツールをいくつか提供し、最終的には PGD 4 と PGD 5 間の単一コマンドのアップグレードパスを提供します。

Application schema upgrades

EDB Postgres Distributed のアップグレードと同様に、アプリケーションスキーマをアップグレードするには 2 つのアプローチがあります。より簡単なオプションは、影響を受けるすべてのアプリケーションを停止し、スキーマのアップグレードを実行し、アップグレードされたアプリケーションを再起動して新しいスキーマバリアントを使用することです。このアプローチでは、ダウンタイムが発生します。

このダウンタイムを解消するために、 EDB Postgres Distributed は、アプリケーションスキーマのローリングアップグレードを実行するための便利なツールを提供します。

次の推奨事項とヒントは、クラスターに対するアプリケーションスキーマのアップグレードの影響を軽減します。

ローリングアプリケーションスキーマのアップグレード

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

PGD は、ノード間のわずかな違いがあってもレプリケーションを継続できるように設計されています。これらの機能は、ダウンタイムなしでアプリケーションスキーマを移行できるように、またはロジカルスタンバイノードでレポートまたはテストを行えるように設計されています。

> **警告:** PGD の外部で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管理する必要があります。

これを実稼働クラスターで正しく機能させるには、慎重なスクリプト作成が必要です。広範なテストをお勧めします。

詳細は 差分のあるノード間のレプリケーション を参照してください。

1 つのノードが新しいテーブルを追加する DDL を実行すると、最新の DDL をまだ受け取っていないノードは余分なテーブルを処理する必要があります。この観点から、ローリングスキーマアップグレードの適切な設定は、`target_table_missing` の競合が発生した場合に `skip` リゾルバーを適用するようにすべてのノードを構成することです。ノードにテーブルを追加する前に、この構成を実行します。この設定は永続的です。

次のクエリを **各ノードで** 個別に実行します。 `node1` を実際のノード名に置き換えます。

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
       target_table_missing, skip);
```

1 つのノードがテーブルに列を追加する DDL を実行すると、最新の DDL をまだ受け取っていないノードは余分な列を処理する必要があります。この観点から、ローリングスキーマアップグレードの適切な設定は、`target_column_missing` の競合が発生した場合に `ignore` リゾルバーを適用するようにすべてのノードを構成することです。これは、1 つのノードに列を追加する前に実行します。この設定は永続的です。

次のクエリを **各ノードで** 個別に実行します。 `node1` を実際のノード名に置き換えます。

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
       target_column_missing, ignore);
```

1 つのノードがテーブルから列を削除する DDL を実行すると、最新の DDL をまだ受け取っていないノードは不足している列を処理する必要があります。この状況により、use_default_value リゾルバーを使用する source_column_missing の競合が発生します。したがって、 NULL を受け入れず、 DEFAULT 値を持たない列には、2 段階のプロセスが必要です。

1. NOT NULL 制約を削除するか、すべてのノードの列に DEFAULT 値を追加します。
2. カラムを取り外します。

ローリング方式でコンストレインを削除できます。現在、一度に 1 ノードずつローリングしてテーブル制約の追加を処理する方法はサポートされていません。

1 つのノードが既存の列の型を変更する DDL を実行する場合、現在の型とターゲット型の間にバイナリ強制性の存在によっては、操作で基になるテーブルデータが書き換えられない場合があります。その場合、それは基になる列タイプのメタデータの更新のみです。通常、テーブルの書き換えは制限されています。ただし、制御された DBA 環境では、すべてのノードの非レプリケート環境でこの列の型のローリングアップグレードを 1 つずつ採用することにより、列の型を自動的にキャスト可能なものに変更できます。詳細については、 *ALTER* テーブル を参照してください。

In-place Postgres major version upgrades

コマンドラインユーティリティ bdr_pg_upgrade を使用して、PGD ノードを Postgres の新しいメジャーバージョンにアップグレードできます。

bdr_pg_upgrade は内部的に標準の pg_upgrade を使用します

スムーズなアップグレードを保証する PGD 固有のロジックを使用。

用語

この用語は、アップグレードプロセスと関連するコンポーネントを説明するときに使用されます。

古いクラスターアップグレードする既存の Postgres クラスターノード、データの移行元。

新しいクラスターデータの移行先の新しい Postgres クラスター。このクラスターノードは、古いクラスターより 1 つ前のメジャーバージョンである必要があります。

注意事項

両方のクラスターが pg_upgrade のすべての要件を満たす必要があるという事実を含め、標準の Postgres メジャーバージョンアップグレードの予防措置が適用されます。

さらに、他のツールがレプリケーションスロットとレプリケーションオリジンを使用している場合は、bdr_pg_upgrade を使用しないでください。アップグレード後に復元されるのは、PGD スロットとオリジンのみです。

bdr_pg_upgrade のいくつかの前提条件を満たす必要があります。

- 古いクラスターを使用しているアプリケーションを切断します。たとえば、クラスター内の別のノードにリダイレクトできます。
- 両方のクラスターのピア認証を構成します。bdr_pg_upgrade はピア認証を必要とします。
- 両方のクラスターの PGD バージョンはまったく同じであり、バージョン 4.1.0 以降である必要があります。
- 新しいクラスターはシャットダウン状態である必要があります。
- 新しいクラスターに PGD パッケージをインストールする必要があります。
- 新しいクラスターは、古いクラスター構成と一致するように、必要に応じて既に初期化および構成されている必要があります。
- 新しいクラスターにデータベース、テーブル、およびその他のオブジェクトが存在しないでください。

bdr_pg_upgrade を実行する前に古いクラスターを起動することもお勧めします。CLI は、古いクラスターがシャットダウンされている場合、起動します。

使用法

Postgres の新しいメジャーバージョンにアップグレードするには、最初に新しいバージョンをインストールする必要があります。

bdr_pg_upgrade コマンドライン

bdr_pg_upgrade は、すべてのパラメーターを pg_upgrade に渡します。したがって、
pg_upgrade でサポートされているパラメーターを指定できます。

概要

```
bdr_pg_upgrade [OPTION] ...
```

オプション

pg_upgrade のオプションに加えて、次のパラメーターを bdr_pg_upgrade に渡すことができます。

必須パラメーター

これらのパラメーターは、コマンドラインで指定するか、 `--database` パラメーターを除くすべての場合、同等の環境変数で指定します。それらは bdr_pg_upgrade によって使用されます。

- `-b, --old-bindir` —古いクラスター bin ディレクトリ。
- `-B, --new-bindir` —新しいクラスター bin ディレクトリ。
- `-d, --old-datadir` —古いクラスターデータディレクトリ。
- `-D, --new-datadir` —新しいクラスターデータディレクトリ。
- `--database` — PGD データベース名。

オプションパラメーター

これらのパラメーターはオプションであり、bdr_pg_upgrade によって使用されます。

- `-p, --old-port` —古いクラスターのポート番号。
- `-s, --socketdir` —アップグレード中に postmaster ソケットに使用するディレクトリ。
- `--check` —チェックのみを実行し、クラスターを変更しないことを指定します。

その他のパラメーター

上記のいずれでもない他のパラメーターは pg_upgrade に渡されます。pg_upgrade は次のパラメーターを受け入れます。

- `-j, --jobs` —使用する同時プロセスまたはスレッドの数。
- `-k, --link` —ファイルを新しいクラスターにコピーする代わりに、ハードリンクを使用します。
- `-o, --old-options` —古い postgres コマンドに渡すオプション。複数の呼び出しが追加されます。
- `-O, --new-options` —新しい postgres コマンドに渡すオプション。複数の呼び出しが追加されます。
- `-N, --no-sync` —アップグレードされたクラスター内のすべてのファイルがディスクに書き込まれるのを待たないでください。
- `-P, --new-port` —新しいクラスターポート番号。
- `-r, --retain` —正常に完了した後でも、SQL ファイルとログファイルを保持します。
- `-U, --username` —クラスターのインストールユーザー名。
- `--clone` —効率的なファイルの複製を使用します。

環境変数

コマンドラインパラメーターの代わりに次の環境変数を使用できます。

- PGBINOLD —古いクラスター bin ディレクトリ。
- PGBINNEW —新しいクラスター bin ディレクトリ。
- PGDATAOLD —古いクラスターデータディレクトリ。
- PGDATANEW —新しいクラスターデータディレクトリ。
- PGPORTOLD —古いクラスターのポート番号。
- PGSOCKETDIR —アップグレード中に postmaster ソケットに使用するディレクトリ。

例

次のシナリオが与えられたとします。

- 古いクラスター bin ディレクトリは/usr/lib/postgresql/13/bin です。
- 新しいクラスター bin ディレクトリは/usr/lib/postgresql/14/bin です。

・旧クラスタデータディレクトリは/var/lib/postgresql/13/main です。

- 新しいクラスタデータディレクトリは/var/lib/postgresql/14/main です。

・ データベース名は bdrdb です。

次のコマンドを使用して、クラスターをアップグレードできます。

```
bdr_pg_upgrade \
- -old-bindir /usr/lib/postgresql/13/bin \
- -new-bindir /usr/lib/postgresql/14/bin \
- -old-datadir /var/lib/postgresql/13/main \
- -new-datadir /var/lib/postgresql/14/main \
- -database bdrdb
```

実行された手順

これらの手順は、 bdr_pg_upgrade の実行時に実行されます。

注釈: *--check* が bdr_pg_upgrade への引数として指定されている場合、CLI はデータベースを変更する手順をスキップします。

PGD Postgres チェック

Steps	*--check* supplied
Collecting pre-upgrade new cluster control data	*run*
Checking new cluster state is shutdown	*run*
Checking PGD versions	*run*
Starting old cluster (if shutdown)	*skip*
Connecting to old cluster	*skip*
Checking if bdr schema exists	*skip*
Turning DDL replication off	*skip*
Terminating connections to database.	*skip*
Disabling connections to database	*skip*
Waiting for all slots to be flushed	*skip*
Disconnecting from old cluster	*skip*
Stopping old cluster	*skip*
Starting old cluster with PGD disabled	*skip*
Connecting to old cluster	*skip*
Collecting replication origins	*skip*
Collecting replication slots	*skip*
Disconnecting from old cluster	*skip*
Stopping old cluster	*skip*

pg_upgrade の手順

標準の pg_upgrade ステップが実行されます。

注釈: 提供されている場合、 *--check* は pg_upgrade に渡されます。

PGD アップグレード後の手順

Steps	*--check* supplied
Collecting old cluster control data	*skip*
Collecting new cluster control data	*skip*
Advancing LSN of new cluster	*skip*
Starting new cluster with PGD disabled	*skip*
Connecting to new cluster	*skip*
Creating replication origin, repeated for each origin	*skip*
Advancing replication origin, repeated for each origin	*skip*
Creating replication slot, repeated for each slot	*skip*

次のページに続く

表 3 – 前のページからの続き

Steps	*--check* supplied
Stopping new cluster	*skip*

Compatibility changes

PGD 5 の多くの変更は、PGD 4 または PGD 3.7 との下位互換性がありません。

接続ルーティング

HARP Manager はもう存在しません。新しい **接続管理** 構成に置き換えられました。

HARP プロキシは、同様に機能する PGD プロキシに置き換えられます。これにより、非推奨の機能が削除され、接続管理構成で構成されます。

Commit At Most Once

CAMO 構成は、 *commit* **スコープへの移行** を介して行われます。 `bdr.camo_pairs` カタログおよび関連する操作関数はもう存在しません。 `bdr.enable_camo` GUC が削除されました。`synchronous_replication_availability` GUC は CAMO に影響しなくなりました。コミットスコープの `DEGRADE ON ... TO ASYNC` 句を使用します。

Eager All-Node Replication

`global` スコープは存在しなくなりました。同じ動作でスコープを作成するには、

Group Commit を使用します。

```
SELECT bdr.add_commit_scope(
    commit_scope_name := eager_scope,
    origin_node_group := top_group,
    rule := ALL (top_group) GROUP COMMIT (conflict_resolution = eager, commit_
↪decision = raft) ABORT ON (timeout = 60s),
    wait_for_ready := true
);
```

`bdr.global_commit_timeout` GUC が削除されました。コミットスコープには `ABORT ON` 句を使用します。

ラグコントロール

CAMO と Eager と同様に、より柔軟な耐久性構成のために、ラグコントロール構成も *commit* スコープへの移行 に移動されました。

カタログ

- bdr.workers は、worker_commit_timestamp のようなワーカー固有の情報を表示しません。
- bdr.worker_errors は非推奨であり、ほとんどの情報を失いました。
- bdr.state_journal_details は非推奨であり、ほとんどの情報を失いました。
- bdr.event_summary は、bdr.worker_errors および bdr.state_journal_details を Raft ロールの変更などの追加情報で置き換えます。
- テーブル bdr.node_catchup_info には、ユーザーが消費可能なビュー bdr.node_catchup_info_details が含まれるようになりました。これにより、よりフレンドリーな方法で情報が表示されます。
- Witness ノードは、複製するレプリケーションセットではなく、bdr.node_summary の node_kind 値を使用して区別されなくなりました。
- Raft（コンセンサス）関連のすべてのテーブルと関数は、複数の Raft インスタンス（サブグループ Raft）をサポートするように調整されました。
- bdr.node_pre_commit ビューと、基になるテーブルが削除されました。情報はテーブルに格納されなくなりました。
- bdr.commit_decisions ビューが追加され、bdr.node_pre_commit ビューを置き換えます。
- その背後にあるメカニズムが一般化されたため、複数の内部自動パーティションテーブルが taskmgr のものに置き換えられました。
- bdr.network_monitoring ビューは、基になるテーブルと関数とともに削除されました。
- *Internal catalogs and views* で説明されているように、多くのカタログが追加され、一部には新しい列があります。これらは厳密には破壊的変更ではありませんが、アップグレード時に確認することをお勧めします。

Supported PGD upgrade paths

バージョン 5 内でのアップグレード

5.0.0	5.0.1	5.1.0	Target PGD version
x	x	x	5.2.0
x	x		5.1.0

次のページに続く

表 4 – 前のページからの続き

5.0.0	5.0.1	5.1.0	Target PGD version
x			5.0.1

バージョン 4 からバージョン 5 へのアップグレード

PGD 4 から PGD 5 へのアップグレードは、バージョン 4.3.0 からサポートされています。古いバージョンの場合、5 にアップグレードする前に 4.3.0 にアップグレードしてください。詳細については、

Upgrading within 4 を参照してください。

4.3.0 以降へのアップグレード後、以下の組み合わせが許可されます。

4.3.0	4.3.0-1	4.3.1	4.3.1-1	Target PGD version
x	x	x	x	5.2.0
x	x	x		5.1.0
x	x			5.0.1
x				5.0.0

バージョン 3.7 からバージョン 5 へのアップグレード

現在、3.7 から 5 への直接アップグレードパスはありません。5 にアップグレードする前に、最初にクラスターを 4.3.0 以降にアップグレードする必要があります。詳細については、
Upgrading from version 3.7 to version 4 ドキュメントを参照してください。

バージョン 3.7 からバージョン 5 へのアップグレード

PGD 3.7 から PGD 5 へのアップグレードは、バージョン 3.7.20 からサポートされています。5 にアップグレードする前に 3.7.20 にアップグレードします。

Upgrading within version 3.7 を参照してください。

3.7.20 以降にアップグレードした後、次の組み合わせが許可されます。

3.7.20	3.7.21	Target PGD version
X	X	5.2.0
X	X	5.1.0
X		5.0.0

第8章　Using

8.1 Application use

ユーザーの観点から PGD アプリケーションについて学びます。

8.1.1 アプリケーションの動作

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

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

前提条件なしで INSERT をレプリケートできます。

他のノードでレプリケートする更新と削除の場合、PGD は、影響を受ける一意の行を識別できる必要があります。PGD では、テーブルに PRIMARY KEY が定義されていること、UNIQUE 制約、または特定の列に定義されている明示的な REPLICA IDENTITY が必要です。これらのいずれかが定義されていない場合、警告が生成され、後の更新または削除は明示的にブロックされます。REPLICA IDENTITY FULL がテーブルに定義されている場合、ユニークインデックスは必要ありません。その場合、更新と削除が許可され、ライブ、有効、遅延されておらず、式または WHERE 句を持たない最初の非一意のインデックスを使用します。それ以外の場合は、シーケンシャルスキャンが使用されます。

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

> *Replication sets* を参照してください。影響を受けるすべてのテーブルが同じサブスクリプションの一部である場合、これは正しく機能します。ただし、サブスクライバーで切り捨てるテーブルの一部に、同じ（または任意の）レプリケーションセットの一部ではないテーブルへの外部キーリンクがある場合、サブスクライバーに切り捨てアクションを適用すると失敗します。

INSERT、UPDATE、および DELETE コマンドによって暗黙的に取得される行レベルのロックは、変更が行われるとレプリケートされます。INSERT、UPDATE、DELETE、および TRUNCATE コマンドによって暗黙的に取得されたテーブルレベルのロックもレプリケートされます。ユーザーセッションによる明示的な行レベルロック（`SELECT ... FOR UPDATE/FOR SHARE`）は複製されず、勧告ロックも複製されません。SERIALIZABLE モードで実行されているトランザクションによって保存された情報は、他のノードにレプ

リケートされません。SERIALIAZABLE のトランザクション分離レベルはサポートされていますが、マルチプルのノード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 が複数のノードで同時に実行され、非同期レプリケーションで実行すると競合が発生する可能性があります。これらを処理するか、回避する必要があります。

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

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

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

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

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

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

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

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

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

PGD レプリケーションの適用は、システムレベルのデフォルトの search_path を使用します。レプリカトリガー、ストリームトリガー、およびインデックス式関数は、他の search_path 設定を想定できますが、適用時に実行すると失敗します。これが発生しないようにするには、次のいずれかの手法を使用します。

- デフォルトの search_path のみを使用して、オブジェクト参照を明確に解決します。
- 常にオブジェクトへの完全修飾参照を使用します。例、schema.objectname。
- 影響を受ける関数に `ALTER FUNCTION ... SET search_path = ...` を使用して、関数の検索パスを設定します。

PGD は、テキストまたはその他の照合可能なデータ型に関連する問題がないことを前提としています。つまり、使用中のすべての照合はすべてのノードで使用でき、デフォルトの照合はすべてのノードで同じです。変更のレプリケートでは、等式検索を使用してレプリカ ID 値を特定するため、一致しない照合修飾子で一意のインデックスが明示的に定義されている場合を除き、これは効果がありません。照合可能な式が使用された場合、行フィルターは照合の違いの影響を受ける可能性があります。

PostgreSQL での非常に長い「トースト」データの PGD 処理は、ユーザーには透過的です。TOAST の「chunkid」値は、異なるノードの同じ行の間で異なる可能性がありますが、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、PGD は正しく動作しません。

PostgreSQL は、揮発性関数を含む CHECK() 制約を許可します。PGD は適用時に CHECK() 制約を再実行するため、以前と同じ結果を返さない後続の再実行により、データの相違が発生します。

PGD は外部キーの使用を制限しません。 FK のカスケードが許可されます。

8.1.2 複製されないステートメント

次のユーザーコマンドは PGD によって複製されないため、効果はローカル/オリジンノードでのみ発生します。

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

`NOTIFY` SQL コマンドと `pg_notify()` 関数は複製されないため、フェールオーバーが発生した場合の通知は信頼できません。これは、サーバーがクラッシュしたときにトランザクションがコミットされた場合、フェールオーバー時に通知が簡単に失われる可能性があることを意味します。`LISTEN` を実行しているアプリケーションは、フェイルオーバー時に通知を逃す場合があります。

これは標準の PostgreSQL レプリケーションにも当てはまり、PGD はまだこれを改善していません。CAMO および Eager Replication オプションでは、 `NOTIFY` SQL コマンドまたは `pg_notify()` 関数は許可されません。

8.1.3 DML および DDL レプリケーション

PGD は DML ステートメントをレプリケートしません。DML ステートメントによって引き起こされた変更をレプリケートします。たとえば、2 つの行を変更した UPDATE は 2 つの変更をレプリケートしますが、行を削除しなかった DELETE は何もレプリケートしません。これは、volatile ステートメントの実行結果がレプリケート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、 PGD はステートメントを複製し、すべてのノードで実行します。したがって、`DROP TABLE IF EXISTS` はローカルノードで何も複製しない場合がありますが、DDL レプリケーションが有効になっている場合、ステートメントは実行のために他のノードに送信されます。詳細は

DML および DDL レプリケーション で説明されています。

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

8.1.4 異なるリリースレベル間の複製

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

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

ただし、クラスター内のメジャーバージョンのノードに参加することはできますが、クラスターが新しいプロトコルバージョンを使用している場合、マイナーバージョンのノードを追加することはできません。これを行うと、エラーが返されます。

これらの機能はどちらも、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、
EDB Postgres Distributed Release notes を参照してください。

8.1.5 差分のあるノード間のレプリケーション

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

PGD は、ノード間のわずかな違いがあってもレプリケーションを継続できるように設計されています。これらの機能は、ダウンタイムなしでアプリケーションスキーマを移行できるように、またはロジカルスタンバイノードでレポートまたはテストを行えるように設計されています。

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

ターゲット上のパーティションを変更する更新のサポートを含む、パーティションテーブルにレプリケート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レプリケートすることが可能です。動的パーティションルーティングは適用時に実行する必要がないため、ソースとターゲットでパーティション定義が同じである場合、より高速になります。詳細は、
Replication sets を参照してください。

デフォルトでは、すべての列が複製されます。

PGD は、列名に基づいてデータ列を複製します。列の名前は同じでデータ型が異なる場合、PGD はソース型からターゲット型にキャストしようとします（それを許可するキャストが定義されている場合）。

PGD は、列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欠落している列がある場合、PGD は `target_column_missing` conflict を発生させます。デフォルトの競合リゾルバーは `ignore_if_null` です。これにより、非 NULL 値が到着するとエラーがスローされます。または、`ignore` の競合リゾルバーでノードを構成することもできます。この設定はエラーをスローしませんが、追加の列を静かに無視します。

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、PGD は source_column_missing conflict を発生させます。デフォルトの競合リゾルバーは use_default_value です。追加の列に NULL（NULL 可能な場合）またはデフォルトの式のデフォルトがある場合、レプリケーションは続行されます。エラーをスローし、そうでない場合はレプリケーションを停止します。

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

ソースとターゲットに異なる制約がある場合、レプリケーションが試行されますが、ソースからの行をターゲットに適用できない場合、失敗する可能性があります。ここで行フィルターが役立ちます。

あるスキーマからよりリラックスしたスキーマにデータをレプリケートしても、エラーは発生しません。スキーマからより制限の厳しいスキーマにデータをレプリケートすると、潜在的な障害の原因になる可能性があります。これを解決する正しい方法は、よりリラックスした側に制約を配置して、悪いデータを入力できないようにすることです。こうすれば、レプリケーションによって不良データが到着することはないため、より制限の厳しいスキーマへの変換が失敗することはありません。たとえば、あるスキーマに TEXT 型の列があり、別のスキーマが同じ列を XML として定義している場合、TEXT 列に CHECK 制約を追加して、テキストが XML であることを強制します。

各ノードで異なるインデックスを持つテーブルを定義できます。デフォルトでは、インデックス定義は複製されます。
DML および DDL レプリケーション を参照して、ノードのサブセットのみまたはローカルにインデックスを作成する方法を指定します。

fillfactor や toast_tuple_target などの記憶域パラメーターは、テーブルのノード間で問題なく異なる場合があります。その例外は、テーブルの記憶域パラメーター user_catalog_table の値がすべてのノードで同一でなければならないことです。

レプリケートされるテーブルは、各ノードで同じユーザー/ロールが所有する必要があります。さらなる議論については *Security and roles* を参照してください。

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

8.1.6 差分のあるノード同士の比較

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

LiveCompare 1.3 以降、 all_bdr_nodes set で設定できます。この設定により、クラスター内の個別のノードごとに関連するすべての DSN を明確にする手間が省けます。EDB Postgres 分散クラスターには、接続情報を持つ N 個のノードがありますが、LiveCompare 1.3+がジョブを完了するために必要なのは初期接続と出力接続のみです。logical_replication_mode を設定すると、すべてのノードがどのように通信しているかが示されます。

たとえば、すべての構成は bdrLC.ini という名前の .ini ファイルで行われます。 /etc/2ndq-livecompare/ でこの構成ファイルのテンプレートを見つけます。

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

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

8.1.7 アプリケーションの一般的なルール

PGD はレプリカの ID 値を使用して、変更する行を識別します。アプリケーションで同じ一意の識別子を挿入、削除、および後で再利用すると、問題が発生する可能性があります。これは
ABA problem と呼ばれます。

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

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

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

- 行に一意の識別子を使用します（INSERT）。
- 一意の識別子（UPDATE）の変更を避けます。
- 削除された一意の識別子の再利用は避けてください。
- ドロップされたオブジェクト名の再利用を避けます。

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

8.1.8 タイミングの考慮事項と同期レプリケーション

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

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

Postgres の同期レプリケーション機能は、PGD でも利用できます。さらに、PGD は、より同期的なレプリケーションのために複数のバリアントを提供します。利用可能なすべてのバリアントとそのさまざまなモードの概要と比較については、 *Durability and performance options* を参照してください。

8.1.9 アプリケーションテスト

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

- 信頼できる *Postgres* アーキテクト
- *CAMO/failover* オプションを使用した *pgbench*
- マルチノードアクセスを備えた *isolationtester*

信頼できる Postgres アーキテクト

信頼できる *Postgres* アーキテクト は、 EDB Postgres

Distributed ベースのものを含むリファレンスアーキテクチャを展開するために EDB が使用するシステムです。

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

```
tpaexec test mycluster mytest
```

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

CAMO/failover オプションを使用した pgbench

EDB Postgres Extended では、pgbench が拡張され、ユーザーが CAMO または通常の PGD デプロイメントを使用しながら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
```

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

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

CAMO 環境の例を次に示します。

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

このコマンドは 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 への接続が失われると、pgbench は node2 に接続します。node2 に問い合わせて、進行中のトランザクションのステータスを取得します。中止および飛行中のトランザクションは、迷彩モードで再試行されます。

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

マルチノードアクセスを備えた isolationtester

`isolationtester` が拡張され、ユーザーが複数のセッションおよび複数のノードでテストを実行できるようになりました。このツールは、内部 PGD テストに使用されますが、ユーザー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 の同時動作を検査するための一連のテストです。これらのテストでは、相互作用する複数のトランザクションを実行する必要があります。これには、複数の同時接続を管理する必要があるため、通常の `pg_regress` プログラムを使用してテストすることはできません。「分離」という名前は、当初の動機がシリアライズ可能な分離レベルをテストすることであったという事実に由来します。他の種類の同時動作のテストも追加されました。

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

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

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

Isolationtester は、libpq を使用してマルチプルの接続を開き、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 5 つの部分で構成され、次の順序でテストされます。

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

この部分は、セッションを実行するサーバーの名前を定義します。ゼロ個以上のサーバー"<name>" 仕様があります。名前に対応する conninfo は、 isolationtester を実行するコマンドによって提供されます。これは quickstart_isolationtest.md で説明されています。この部分はオプションです。

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

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

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

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

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

通常、スペックファイルにはいくつかの「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、ティアダウン、および 1 つ以上の「ステップ」の 3 つのパートで構成されています。セッションごとのセットアップとティアダウンの部分は、テストごとのセットアップとティアダウンと同じ構文ですが、セッションごとに実行されます。セットアップ部分には、通常、トランザクションを開始するための BEGIN コマンドが含まれています。

セッション部分も connect_to 仕様で構成され、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をポイントします。

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

各ステップの構文は次のとおりです。

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

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

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

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

”＃”で始まる行はコメントです。

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

1. メインセットアップ部

    1. セッションごとのセットアップパーツ

1. 選択されたセッションステップ

1. セッションごとのティアダウン

1. メインのティアダウンスクリプト

選択された各ステップは、そのセッションに関連付けられた接続に送信されます。

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

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

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

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

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

次に、次を実行します。

`make isolationcheck-installcheck`

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

次のように、適切なデータベース名と bdr インスタンスの conninfo を PGD makefile 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、次を実行します。

`make isolationcheck-makecheck`

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

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

8.1.10 パフォーマンステストとチューニング

PGD を使用すると、複数のマスターノードに書き込みトランザクションを発行できます。これらの書き込みを各ノードに戻すと、パフォーマンスが低下します。

まず、別のノードからの変更の再生には CPU コスト、I / O コストがあり、WAL レコードを生成します。SQL を再実行する必要がないため CPU オーバーヘッドが少ないため、リソース使用は通常元のトランザクションよりも少なくなります。UPDATE および DELETE トランザクションの場合、データがキャッシュされていない場合、再生時に I/O コストが発生する場合があります。

第二に、変更をリプレイすると、ローカルワークロードに対して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のロックが保持されます。競合のないレプリケートデータ型（CRDT）と列レベルの競合検出（CLCD）機能により、同時更新でも正しい答えが得られますが、通常のロックオーバーヘッドが削除されるわけではありません。ロックの競合が発生した場合は、更新の競合を避けるか、トランザクションをできるだけ短くしてください。大規模なトランザクションで頻繁に更新される行は、そのトランザクションのパフォーマンスにボトルネックを引き起こします。複雑なアプリケーションでは、スケーラビリティを維持するためにある程度の配慮が必要です。

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

PGD の実行が遅い場合は、次のことをお勧めします。

1. 本番システムの問題ケースに可能な限り近づけて、pgbench のカスタムテストスクリプトを作成します。

2. 1 つのノードでスクリプトを実行して、ベースラインの数値を取得します。

3. 1 つのノードで実行したのと同じ合計数のセッションを使用して、実稼働で発生するのと同じ数のノードでスクリプトを実行します。このテクニックは、マルチプルのノードに移動した場合の効果を示しています。

4. これら 2 つのテストのセッション数を増やして、アプリケーションでの競合の増加の影響をプロットできるようにします。

5. テストがレプリケーションの遅延を考慮に入れるのに十分な長さであることを確認します。

6. テスト中にレプリケーションの遅延が拡大していないことを確認します。

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

8.1.11 PGD でのテーブルアクセスメソッド（TAM）の使用

PGD 5.0 は、 EDB Postgres 15.0 でリリースされた 2 つのテーブルアクセス方法をサポートしています。これら 2 つのテーブルアクセス方法は、PGD 5.0 で認定および許可されています。

※オートクラスター※参考データ

他の TAM は、EDB によって認定されるまで制限されます。テーブルでテーブルアクセス方法のいずれかを使用する予定がある場合は、PGD クラスター内の各参加ノードでその TAM を構成する必要があります。自動クラスターまたは参照データ TAM を構成するには、各ノードで次の手順に従います。

1. `postgresql.conf` を更新して、`shared_preload_libraries` パラメーターに TAM `autocluster` または `refdata` を指定します。

1. サーバーを再起動し、`CREATE EXTENSION autocluster;` または `CREATE EXTENSION refdata;` を実行します。

拡張機能を作成した後、TAM で `CREATE TABLE test USING autocluster;` または `CREATE TABLE test USING refdata;` を使用してテーブルを作成できます。これは、すべての PGD ノードに複製されます。これらのテーブルアクセス方法の詳細については、

> テーブルを作成する を参照してください。

8.2 Postgres configuration

いくつかの Postgres 構成パラメーターは、PGD ノードに影響します。これらのパラメーターは、ノードごとに異なる方法で設定できますが、通常はお勧めしません。

PGD 独自の設定については、 *PGD settings reference* を参照してください。

8.2.1 Postgres の設定

正しく実行するには、PGD に次の Postgres 設定が必要です。

- `wal_level` — PGD は論理デコードに依存しているため、 `logical` に設定する必要があります。
- `shared_preload_libraries` — `bdr` を含める必要がありますが、必要に応じて、その前後に他のエントリを含めることができます。ただし、`pglogical` を含めないでください。
- `track_commit_timestamp` —競合する各行のタイムスタンプを取得する競合解決のために `on` に設定する必要があります。

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

- `logical_decoding_work_mem` —論理デコードで使用されるメモリバッファーサイズ。このサイズより大きいトランザクションはバッファーをオーバーフローし、ローカルディスクに一時的に保存されます。デフォルトは 64MB ですが、はるかに大きく設定できます。

- max_worker_processes —PGD はレプリケーションとメンテナンスのタスクにバックグラウンドワーカーを使用するため、正しく動作するには十分なワーカースロットが必要です。各データベースの正しい最小ワーカー数の式は、これらの値を合計することです。- 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 - PGD 対応データベースごとに 4 つ - PGD グループ内のピアノードごとに 1 つ - PGD グループ内のピアノードごとに有効になっている各ライターノードが PGD グループから削除されるときに、一時的により多くのワーカープロセスが必要になる場合があります。

- max_wal_senders —すべてのピアノードに 2 つ必要。

- max_replication_slots —すべてのピアノードに 2 つ必要。

- wal_sender_timeout および wal_receiver_timeout —ノードが CAMO パートナーを切断または再接続と見なす速さを決定します。詳細は *CAMO failure scenarios* を参照してください。

N 個のピアノードを持つグループの通常の実行では、PGD には N 個のスロットと WAL 送信者が必要です。同期中、PGD は一時的に別の N-1 スロットと WAL 送信者を使用するため、この時折のピーク需要に合わせてパラメーターを十分に高く設定するように注意してください。

パラレル適用をオンにした状態で、スロット数を式* writers から N スロットに増やす必要があります。これは、max_replication_slots もレプリケーション起点の最大数を設定し、パラレル適用の機能の一部がライタごとに追加の起点を使用するためです。

> デコードワーカー が有効になっている場合、このプロセスには PGD 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。

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

レガシー同期レプリケーションモードは、次のパラメーターを使用してサポートされます。詳細と制限事項については、
Durability and performance options を参照してください。

- synchronous_commit and synchronous_standby_names —PGD 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 。

8.2.2 時間ベースのスナップショット

snapshot_timestamp

> タイムスタンプベースのスナップショット の使用をオンにし、使用するタイムスタンプを設定します。

8.2.3 最大準備トランザクション

max_prepared_transactions

明示的な 2 フェーズコミット、CAMO、または Eager トランザクションによるクラスター全体の同時準備トランザクションの最大数に対処するには、十分に高く設定する必要があります。制限を超えると、ノードでローカル 2 フェーズコミットまたは CAMO トランザクションを実行できなくなり、クラスター上のすべての Eager トランザクションができなくなります。このパラメーターは、Postgres サーバーの起動時にのみ設定できます。

8.3 Node management

PGD グループのメンバーである各データベースは、独自のノードで表される必要があります。ノードは、PGD グループ内の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みのメンバーになれます。(これは後のリリースで拡張される可能性があります。) 各ノードは 1 つ以上のレプリケーションセットに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。

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

8.3.1 PGD グループの作成と参加

PGD の場合、すべてのノードは他のすべてのノードに接続する必要があります。構成を簡単にするために、新しいノードが参加すると、既存のすべてのノードが接続するように構成されます。このため、作成された最初の PGD ノードを含むすべてのノードは、他のノードが接続に使用できる
PostgreSQL connection string

（データソース名（DSN）とも呼ばれます）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。したがって、`host=myhost port=5432 dbname=mydb` のようなキー値形式、または `postgresql://myhost:5432/mydb` のような URI 形式のいずれかを使用できます。

SQL 関数 *bdr.create_node_group()* は、ローカルノードから PGD グループを作成します。これにより、そのノードで PGD がアクティブになり、他のノードがその時点で 1 つのノードのみで構成される PGD グループに参加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に接続するために使用する接続文字列を指定する必要があります。

ノードグループが作成されると、以降のすべてのノードは *bdr.join_node_group()* 関数を使用して PGD グループに参加できます。

または、コマンドラインユーティリティ
bdr_init_physical を使用して、既存のノードの `pg_basebackup`

（またはフィジカルスタンバイ）を使用して新しいノードを作成します。`pg_basebackup` を使用している場合、bdr_init_physical ユーティリティは、オプションでターゲットデータベースのみのベースバックアップを指定できます。以前の動作は、データベースクラスター全体をバックアップすることでした。このユーティリティを使用すると、アクティビティはより速く完了し、不要なデータベースを除外するため、使用す

る領域も少なくなります。ターゲットデータベースのみを指定すると、除外されたデータベースがクリーンアップされ、新しいノードで削除されます。

新しい PGD ノードが既存の PGD グループに参加する場合、またはノードがアップストリームピアにサブスクライブする場合、レプリケーションを開始する前に、システムは既存のデータをピアノードからローカルノードにコピーする必要があります。このコピーは、ローカルデータとリモートデータが最初は同じになるように慎重に調整する必要があります。自分で pg_dump を使用するだけでは不十分です。BDR 拡張機能は、この初期コピーを作成するための組み込み機能を提供します。

参加プロセス中に、BDR 拡張機能は、提供されたソースノードを基礎として使用して既存のデータを同期し、PGD 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は、参加プロセスを最初からやり直します。

クラスターに参加するノードには、PGD 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が含まれていてはなりません。BDR 拡張子を除き、新しく参加するデータベースは空にすることをお勧めします。ただし、必要なすべてのデータベースユーザーとロールが作成されていることが重要です。

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

参加するソースノードとして、接続が最適な（最も近い）ソースノードを選択することをお勧めします。そうすることで、結合が完了するまでに必要な時間が短縮されます。

Raft コンセンサスアルゴリズムを使用して、参加手順を調整します。これには、ほとんどの既存のノードがオンラインで到達可能であることが必要です。

論理結合プロシージャ（*bdr.join_node_group* ファンクションを使用する）は、`COPY` 操作を実行してデータ同期を実行し、有効になっている場合は複数のライター（並列適用）を使用します。

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

参加プロセスはソースとして 1 つのノードのみを使用するため、ノードの過半数が利用可能な場合、ノードがダウンしたときに実行できます。これにより、論理結合の実行が複雑になる可能性があります。論理結合中に、ソースノードからコピーされた行のコミットタイムスタンプは、ソースノードの最新のコミットタイムスタンプに設定されます。これより前のコミットタイムスタンプを持つノードでコミットされた変更（ノードがダウンしているか、大幅なラグがあるため）は、他のノードからの変更と競合する可能性があります。この場合、新しく参加したノードは他のノードとは異なる方法で解決される可能性があり、発散が発生します。その結果、ノード間に重大なレプリケーションラグが存在する場合は、ノード参加を実行しないことをお勧めします。これが必要な場合は、新しく参加したノードで LiveCompare を実行して、すべてのノードが利用可能で追いついたら、データの相違を修正します。

`pg_dump` は、キャッシュルックアップの失敗が原因で、ソースノードで同時 DDL アクティビティがある場合に失敗する可能性があります。

bdr.join_node_group は内部で `pg_dump`

を使用するため、ソースノードに同時 DDL アクティビティがあると失敗する場合があります。その場合、結合を再試行すると機能します。

異種クラスターへの参加

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

PGD は、一部のノードが 1 つの PostgreSQL メジャーリリースで 3.6 を実行していて、他のノードが別の PostgreSQL メジャーリリースで 3.7 を実行している場合でも、レプリケーションがすべての方向で正しく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、古いノードを分割することにより、クラスターをすばやく同種の状態にすることをお勧めします。古いバージョンで利用できない可能性のある DDL、またはその逆を実行しないでください。

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

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と、競合解決などの特定の構成がソースノードからコピーされません。ノードは、クラスターに参加した後に構成する必要があります。

接続 DSN と SSL（TLS）

ノードは `libpq` を使用して接続するため、ノードの DSN は単なる `libpq` 接続文字列です。そのため、SSL 用のパラメーターを含む、許可された `libpq` 接続パラメーターを含めることができます。DSN は、指定されたノードに接続するクライアントからの接続文字列として機能する必要があります。クライアント証明書を使用するこのようなパラメーターのセットの例は次のとおりです。

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

このセットアップでは、ファイル `bdr_client.crt` 、`bdr_client.key` 、および `root.crt` が、適切な権限を持つ各ノードのデータディレクトリに存在する必要があります。`verify-full` モードの場合、サーバーの SSL 証明書がチェックされ、`root.crt` 認証局で直接または間接的に署名されていること、および接続で使用されるホスト名またはアドレスが証明書の内容と一致することを確認します。名前の場合、これはサブジェクトの別名と一致するか、証明書にそのような名前がない場合はサブジェクトの共通名（CN）フィールドと一致します。Postgres は現在、IP アドレスのサブジェクトの別名をサポートしていないため、名前ではなくアドレスで接続が行われる場合、CN フィールドと一致する必要があります。

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

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

別のセットアップは、クライアント証明書の代わりに SCRAM-SHA-256 パスワードを使用し、証明書が適切に署名されている限りサーバー ID を検証しないことです。ここで DSN パラメーターは次のようになります。

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

対応する pg_hba.conf 行は次のとおりです。

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

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

8.3.2 Witness ノード

クラスターのノード数が偶数の場合、ネットワーク分割（またはネットワークパーティションと呼ばれることもあります）が発生した場合に関係を断ち切るのに役立つ追加のノードを作成すると役立つ場合があります。

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

8.3.3 ロジカルスタンバイノード

PGD を使用すると、オフロードノード、読み取り専用ノード、受信専用ノード、または論理読み取りレプリカとも呼ばれる**論理スタンバイノード**を作成できます。マスターノードは、ゼロ、1 つ以上のロジカルスタンバイノードを持つことができます。

注釈: ロジカルスタンバイノードは、データセンター間のネットワークトラフィックが懸念される環境で使用できます。それ以外の場合、場所ごとにより多くのデータノードを持つことが常に好まれます。

フィジカルスタンバイノードでは、ノードが完全に起動しないため、継続的リカバリモードのままになります。PGD は似たようなことを許可します。

> *bdr.join_node_group* には、ノードをロジカルスタンバイノードとして途中で結合したままにする pause_in_standby

オプションがあります。ロジカルスタンバイノードは変更を受け取りますが、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フルの通常の送信/受信ノードに移動します。

ロジカルスタンバイには、 *bdr.join_node_group* の DSN で定義された 1 つのソースノードからデータが送信されます。他のすべてのノードからの変更はこの 1 つのソースノードから受信され、複数のサイト間の帯域幅を最小限に抑えます。

高可用性には複数のオプションがあります。

- ソースノードが死亡した場合、1 つのフィジカルスタンバイをマスターに昇格できます。この場合、新しいマスターは、任意またはすべてのロジカルスタンバイノードに電力を供給し続けることができます。
- ソースノードが停止した場合、1 つのロジカルスタンバイをフルノードに昇格させ、シングルマスター操作と同様のフェールオーバー操作でソースを置き換えることができます。複数のロジカルスタンバイノードがある場合、他のノードは新しいマスターをフォローできないため、この手法の有効性は 1 つのロジカルスタンバイに制限されます。

既存の PGD ノードから新しいスタンバイが作成された場合、操作に必要なレプリケーションスロットは、グループスロットが最後に進められてから少なくとも 16 MB の LSN が経過するまで、新しいスタンバイに同期されません。極端な場合、これには、スロットがストリーミングレプリカで同期または作成される前に、完全な 16 MB が必要になる場合があります。この間隔中にフェールオーバーまたはスイッチオーバーが発生した場合、グループスロットと他の依存スロットがまだ存在しないため、ストリーミングスタンバイをその PGD ノードを置き換えるように昇格させることはできません。

スタンバイのスロット同期アッププロセスは、アップストリームの関数を呼び出すことによりこれを解決します。この機能は、WAL スイッチを実行し、すべての PGD ピアノードに進行状況の更新を再生するように要求することにより、EDB Postgres 分散クラスター全体のグループスロットを移動します。これにより、グループスロットが短時間で先に進みます。これにより、スタンバイが最初のスロットの同期に必要な時間が短縮され、必要に応じて初期スロットへのフェールオーバーが高速化されます。

PostgreSQL では、昇格する前に、スロットの同期がスタンバイで完了していることを確認することが重要です。ターゲットデータベースのスタンバイで次のクエリを実行して、スロットがアップストリームと同期していることを監視および確認できます。このクエリが `true` を返すと、プロモーションを進めることができます。

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

WAL スイッチを手動で実行し、すべての PGD ピアノードに進行状況の更新を再生するように要求することにより、PGD クラスター全体でスロット同期プロセスを微調整することもできます。このアクティビティにより、グループスロットが短時間で先に進み、スタンバイでのスロット同期アクティビティが早められます。このために、ターゲットデータベース内の PGD ピアノードで次のクエリを実行できます。

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

スタンバイでモニタリングクエリを使用して、これらのクエリがそのスタンバイでのスロット同期の高速化に役立つことを確認します。

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

ロジカルスタンバイでは書き込みトランザクションが許可されるため、フィジカルスタンバイの制限は適用されません。これを使用すると、ロジカルスタンバイに追加のインデックス、データ、中間作業表、LISTEN/NOTIFY、一時テーブル、マテリアライズドビュー、およびその他の違いのより長い保持期間を設定できるため、大きなメリットがあります。

昇格前にコミットされたロジカルスタンバイに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。昇格後にコミットされるすべてのトランザクションは、以降に送信されます。ロジカルスタンバイへの書き込みを実行する場合は、昇格する前にデータベースを静止するように注意してください。

ロジカルスタンバイノードに DDL 変更を加える場合がありますが、それらはレプリケートされず、グローバル DDL ロックを取得しようとしません。DDL と同様に動作する PGD 関数もレプリケートされません。

> *DML および DDL レプリケーション* を参照してください。ロジカルスタンバイに互換性のない DDL 変更を加えた場合、データベースは

ダイバージェントノードです。現在、発散ノードの昇格はレプリケーションに失敗します。その結果、ロジカルスタンバイノードをスタンバイとして使用する場合は、ロジカルスタンバイノードが発散の変更を受けないようにするか、発散ノードが昇格しないことを保証することを計画してください。

8.3.4 フィジカルスタンバイノード

PGD では、従来のフィジカルスタンバイフェールオーバーノードを作成することもできます。これらは一般に、短い昇格手順の後、クラスター内の PGD ノードを直接置き換えることを目的としています。標準の Postgres クラスターと同様に、ノードはこれらの物理レプリカを任意の数持つことができます。

ただし、PGD でのレプリケーションスロットおよびその他の機能要件の使用により、これが適切に機能するための最小前提条件がいくつかあります。

- PGD プライマリとスタンバイ間の接続は、物理レプリケーションスロットを介したストリーミングレプリケーションを使用します。
- スタンバイには以下があります。 - `recovery.conf` （PostgreSQL <12 の場合、PostgreSQL 12+の場合、これらの設定は `postgres.conf` にあります）： - プライマリを指す `primary_conninfo` ： - `shared_preload_libraries = 'bdr'` 、リストには他のプラグインもありますが、pglogical を含めないでください - `hot_standby = on` - `hot_standby_feedback = on`
- プライマリには以下があります。 - `postgresql.conf` ： - *bdr.standby_slot_names* は、スタンバイの `primary_slot_name` に使用される物理レプリケーションスロットを指定します。

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

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

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

フィジカルスタンバイノードが最新の状態であり、昇格できることを保証する保護メカニズム（

bdr.standby_slot_names

で構成）は、PGD グループの全体的なレプリケーションレイテンシーに影響します。これは、グループレプリケーションがフィジカルスタンバイノードが最新の場合にのみ発生するためです。

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

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

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

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

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

これが完了すると、他の PGD ノードは、新しく昇格したスタンバイ->プライマリノードとの通信を再確立します。レプリケーションスロットは定期的にのみ同期されるため、この新しいプライマリは、既存の PGD 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、PGD は各遅延スロットを各 PGD ノードが使用した最後の場所に早送りします。

bdr.alter_node_interface

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

PGD は、アウトバウンドレプリケーションで最も遅延が大きいクラスターノードの状態を常に反映するグループスロットを維持します。物理レプリカを追加すると、グループスロットの状態に関係なく、参加していないノードメンバーがあることを PGD に通知する必要があります。

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

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

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

`synchronous_standby_names` にフィジカルスタンバイノードと他の PGD ノードを混在させることができます。CAMO と Eager All-Node Replication は異なる同期メカニズムを使用し、同期レプリケーションでは動作しません。CAMO または Eager All-Node Replication のいずれかを使用する場合、

synchronous_standby_names に PGD ノードが含まれていないことを確認してください。代わりに、フィジカルスタンバイなどの非 PGD ノードのみを使用します。

8.3.5 サブグループ

グループには、ゼロ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で特定の目的に割り当てることができます。node_group_type は、サブグループが作成されるときにタイプを指定します。

購読者専用グループ

名前が示すように、このタイプのノードは、クラスター内の他のノードからのレプリケーションの変更のみにサブスクライブします。ただし、他のノードは subscriber-only ノードからレプリケーションの変更を受け取りません。これは、ロジカルスタンバイノードに多少似ています。ただし、ロジカルスタンバイとは対照的に、subscriber-only ノードはクラスターに完全に参加しています。彼らは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 1 つのノードの非可用性または分離の影響を受けません。

subscriber-only ノードは完全に参加した PGD ノードであるため、複製されたすべての DDL を受信し、それらに動作します。また、Raft を使用して、クラスター内のすべてのノードにステータスを一貫して報告します。subscriber-only ノードには 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、レプリケートされた DDL を受信しますが、DDL または DML ロックの取得には参加しません。言い換えれば、現在ダウンしている subscriber-only ノードは、DML ロックの取得を停止しません。

subscriber-only ノードは、PGD ツリートポロジのビルディングブロックを形成します。このトポロジでは、少数の完全にアクティブなノードがすべての方向に変更をレプリケートしています。多数の subscriber-only ノードは変更のみを受け取りますが、クラスター内の他のノードに変更を送信することはありません。このトポロジは、多数のノードによる接続の爆発を回避しますが、データの消費に使用できる非常に多数の leaf ノードを提供します。

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

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

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

注意事項

subscriber-only ノードはクラスター内のノードに変更をレプリケートしないため、ノードがクラスターから分離されると、レプリケーションの変更を同期するソースとして機能できません。ただし、subscriber-only ノードがすでにクラスター内の他のノードが持っていない分離されたノードからレプリケーションの変更を受信して適用している場合、ノード間で矛盾が発生します。

今のところ、 *bdr.standby_slot_names* を設定することでこれを解決できます

および *bdr.standby_slots_min_confirmed* により、 subscriber-only ノードよりも前に完全にアクティブな PGD ノードが常に存在します。

8.3.6 デコードワーカー

PGD4 は、データが送信されるノードの数に関係なく、1 回のデコードを実行するデコードワーカープロセスを有効にするオプションを提供します。これにより、各 PGD ノードに新しいプロセス、WAL デコーダーが導入されます。接続ごとに 1 つの WAL 送信プロセスが引き続き存在しますが、これらのプロセスはデータの送受信のタスクのみを実行するようになりました。まとめると、これらの変更は、より大きな PGD グループの CPU オーバーヘッドを削減し、WAL 送信者プロセスが通信により多くの時間を費やすようになったため、レプリケーションスループットを向上させます。

enable_wal_decoder は各 PGD 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()* **を使用したグループ構成** を使用して、PGD 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、PGD は論理変更レコード（LCR）ファイルを保存して、デコードとすべてのサブスクライブノードがデータを受信したときとの間の変更をバッファリングできるようにします。LCR ファイルは、各ローカルノードのデータディレクトリの pg_logical ディレクトリの下に保存されます。レプリケーションラグが増加すると、LCR ファイルの数とサイズが変化するため、これも監視が必要です。PGD ノードのいずれかで必要とされない LCR は、定期的にクリーニングされます。2 つの連続するクリーンアップの間隔は *bdr.lcr_cleanup_interval* によって制御され、デフォルトは 3 分です。

> *bdr.lcr_cleanup_interval* がゼロの場合、クリーンアップは無効になります。

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

PGD グループのデコードワーカーが有効になっている場合でも、次の GUC はノードごとの LCR の生成と使用を制御します。デフォルトでは、これらは false です。LCR の生成と使用には、PGD グループのデコードワーカーを有効にし、これらの GUC を PGD グループ内の各ノードで true に設定します。

- *bdr.enable_wal_decoder* — false にすると、LCR を使用するすべての WAL 送信者は再起動して WAL を直接使用します。PGD グループ構成とともに true を使用すると、デコードワーカープロセスが開始されて LCR が生成され、WAL 送信者は LCR を使用します。
- *bdr.receive_lcr* & mdash; サブスクライブノードで true を使用する場合、パブリッシャーノード上の WAL センダーに LCR が利用可能であれば使用するように要求します。

注意事項

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

サブスクライバー専用ノードは、各ノードから直接変更を受け取ります。したがって、加入者専用ノードにサービスを提供する WAL 送信者は、LCR を使用できます。

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

参考までに、LCR ファイル名の最初の 24 文字は WAL ファイル名と似ています。名前の最初の 8 文字はすべて「0」です。将来的には、WAL セグメントファイル名の最初の 8 文字に似た TimeLineId を表すことが期待されます。次の名前の 16 文字のシーケンスは、WAL ストリームに対して LCR の変更を追跡するために使用される WAL セグメント番号に似ています。

ただし、論理変更は、それらが属するトランザクションのコミット順序に従って並べ替えられます。したがって、LCR セグメント内のそれらの配置は、WAL セグメント内の対応する WAL の配置と一致しません。

最後の 16 文字のセットは、LCR セグメントのサブセグメント番号を表します。各 LCR ファイルはサブセグメントに対応します。LCR ファイルはバイナリで可変サイズです。 LCR ファイルの最大サイズは、`bdr.max_lcr_segment_file_size` で制御できます。デフォルトは 1 GB です。

8.3.7 ノードの再起動とダウンしたノードの回復

PGD は、ノードの再起動またはノードの切断から回復するように設計されています。切断されたノードは、各ピアノードに再接続し、そのノードから欠落しているデータを複製することにより、グループに再参加します。

ノードが起動すると、各接続は `bdr.node_slots.state = catchup` で FROM bdr.node_slots; に表示され始め、欠落しているデータの複製を開始します。キャッチアップは、各ピアノードからの欠落データの量に応じて一定期間続き、サーバーのワークロードに応じて、時間の経過とともに増加する可能性があります。

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

数時間や数日など、長期間オフラインになっているノードは、さまざまな理由でリソースの問題を引き起こし始める可能性があります。次の問題を理解せずに、長期にわたる停止を計画しないでください。

各ノードは変更情報を保持します（1 つの replication slot を使用）

ピアノードごと）により、一時的に到達不能なノードへの変更を後でリプレイできます。ピアノードが無期限にオフラインのままである場合、この蓄積された変更情報により、最終的にノードで PostgreSQL トランザクションログ（`pg_wal` の *WAL* ）のストレージスペースが不足し、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、他のディスク不足関連の症状を報告する場合があります。

さらに、オフラインノードのスロットもカタログ xmin を保持し、カタログテーブルのバキュームを防ぎます。

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

オリジンの競合検出 を参照)。

管理者は、ノードの停止を監視し（ *Monitoring* を参照)、ノードに十分な空きディスク領域があることを確認する必要があります。ワークロードが予測可能な場合は、経時的に使用される領域量を計算できる場合があり、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。

PGD によって作成されたレプリケーションスロットを手動で削除しないでください。その場合、クラスターが損傷し、
PGD によって作成されたレプリケーションスロット

で説明されているように、スロットを使用していたノードをクラスターから分離する必要があります。

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

PGD によって作成されたレプリケーションスロット

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

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

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

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

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

- ノード alpha には 3 つのスロットがあります。 - bdr_mydb_mygroup という名前の 1 つのグループスロット - bdr_mydb_mygroup_beta および bdr_mydb_mygroup_gamma という名前の 2 つのノードスロット
- ノード beta には 3 つのスロットがあります。 - bdr_mydb_mygroup という名前の 1 つのグループスロット - bdr_mydb_mygroup_alpha および bdr_mydb_mygroup_gamma という名前の 2 つのノードスロット

- ノード gamma には 3 つのスロットがあります。 - bdr_mydb_mygroup という名前の 1 つのグループスロット - bdr_mydb_mygroup_alpha および bdr_mydb_mygroup_beta という名前の 2 つのノードスロット

グループレプリケーションスロット

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

グループスロット名は、関数 *bdr.local_group_slot_name()* によって与えられます。

グループスロットは次のことができます。

- 既存のすべてのノードを起動して実行せずに（ノードの大部分は稼働している必要がありますが)、新しいノードを PGD グループに参加させます。参加中にダウンしたノードが再び複製を開始した場合にデータ損失を招くことはありません。
- 一部のノードが分割されたノードに完全に追いついていない場合でも、一貫してクラスターからノードを分割します。
- いくつかの競合を見逃さないように、フリーズポイントを保持します。
- タイムスタンプベースのスナップショットの履歴スナップショットを保持します。

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

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

長い識別子のハッシュ

レプリケーションスロットの名前は、他の PostgreSQL 識別子と同様に、63 バイト以下でなければなりません。PGD は、結果のスロット名がその制限に対して長すぎる場合に、データベース名、PGD グループ名、およびノードの名前を短縮することにより、これを処理します。識別子の短縮は、文字列の最後の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。

たとえば、 `group20xxxxxxxxxxxx` （長さ 20 バイト）という名前の PGD グループを使用して、`db20xxxxxxxxxxxxxxx` （長さ 20 バイト）という名前のデータベースをレプリケートするクラスターについて考えます。`3597186` 、`be9cbd0` 、および `7f304a2` は、それぞれ `db20xxxxxxxxxxxxxxx` 、`group20xxxxxxxxxxxx` 、および `a30xxxxxxxxxxxxxxxxxxxxxxxxxxxx` のハッシュであるため、ノード `a30xxxxxxxxxxxxxxxxxxxxxxxxxxxx` （30 バイト長）に関連付けられた論理レプリケーションスロットが呼び出されます。

```
bdr_db20xxxx3597186_group20xbe9cbd0_a30xxxxxxxxxxxx7f304a2
```

8.3.8 PGD グループからノードを削除する

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

parting とも呼ばれるノードの削除は、 *bdr.part_node()*

機能。ノードを削除するには、（ノードの作成中に渡された）ノード名を指定する必要があります。削除するノードを含む、PGD グループ内のアクティブなノードから *bdr.part_node()* ファンクションを呼び出すことができます。

参加手順と同じように、別れは Raft コンセンサスを使用して行われ、動作するにはノードの大部分がオンラインである必要があります。

別れのプロセスはすべてのノードに影響します。ラフトリーダーは、ノード間の投票を管理して、どのノードが分離ノードからの最新のデータを持っているかを確認します。次に、残りのすべてのノードは、最新のノードへのセカンダリの一時的な接続を確立して、欠落しているデータをキャッチできるようにします。

別れたノードはまだ PGD に認識されていますが、リソースを消費しません。別れたノードと同じ名前でノードが再度追加される場合があります。まれに、関数
bdr.drop_node()

を使用して、分割されたノードのすべてのメタデータをクリアしたい場合があります。

8.3.9 PGD グループ全体を削除する

PGD グループは通常、場所にマッピングします。ロケーションが展開されなくなったら、そのロケーションの PGD グループも削除する必要がある可能性があります。

削除する PGD グループは空である必要があります。グループを削除する前に、グループ内のすべてのノードを分離する必要があります。

PGD のアンインストール

PGD 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 PGD オブジェクトが削除されます。次のコマンドでこれを行うことができます。

```
DROP EXTENSION bdr;
```

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

- `SnowflakeId` や `galloc` などの PGD 固有のシーケンスを使用したテーブル

- CRDT データ型を使用する列
- 一部の PGD カタログテーブルに依存するビュー

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

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

警告:

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

bdr.drop_node() 関数もあります。この機能は、別れに困ったときなど、緊急時にのみ使用してください。

8.3.10 PGD トポロジーのリスト

PGD グループのリスト

次の単純なクエリは、現在のノードがメンバーであるすべての PGD ノードグループをリストします。現在、
bdr.local_node_summary から 1 行のみを返します。

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

より複雑なクエリを使用して、各ノードグループの構成を表示できます。

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

PGD グループ内のノードのリスト

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

```
SELECT node_name          AS name
, node_seq_id             AS ord
, peer_state_name         AS current_state
, peer_target_state_name AS target_state
, interface_connstr       AS dsn
FROM bdr.node_summary
WHERE node_group_name = mygroup;
```

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

8.4 DDL replication

DDL はデータ定義言語の略で、データベースオブジェクトを作成、変更、および削除する SQL 言語のサブセットです。

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

- 一時的またはログされていない関係
- 特定の DDL ステートメント（ほとんどが長時間実行）
- ロックコマンド（LOCK ）
- テーブルメンテナンスコマンド（VACUUM 、ANALYZE 、CLUSTER 、REINDEX ）
- autovacuum のアクション
- 操作コマンド（CHECKPOINT 、ALTER SYSTEM ）

・ データベースまたはテーブルスペースに関するアクション

自動 DDL レプリケーションにより、DDL 変更をすべてのノードに手動で配布し、それらの一貫性を保証することなく、特定の DDL 変更が簡単になります。

デフォルトのレプリケーションセットでは、デフォルトで DDL がすべてのノードにレプリケートされます。DDL をレプリケートするには、DDL レプリケーションフィルタをレプリケーションセットに追加する必要があります。

DDL replication filtering を参照してください。

PGD は、DDL レプリケーションに関しては、スタンドアロンの PostgreSQL とは大きく異なります。それを同じように扱うことは、PGD の最も一般的な問題です。

テーブルレプリケーションとの主な違いは、DDL レプリケーションは DDL の結果ではなくステートメント自分自身をレプリケートすることです。これはほとんどの場合非常にうまく機能しますが、DDL はすべ

てのノードで同様に実行する必要があるという要件が導入されます。より微妙な点は、DDL は、拡張機能（ビルトインではない）によって導入されたデータ型を含む、すべてのデータ型固有のパラメーター設定に関して不変でなければならないということです。たとえば、DDL ステートメントは、各ノードで使用されるデフォルトのエンコーディングで正しく実行する必要があります。

8.4.1 DDL レプリケーションオプション

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

bdr.ddl_replication = on はデフォルトであり、DDL をデフォルトのレプリケーションセット、デフォルトではすべてのノードを意味します。デフォルト以外のレプリケーションセットは、

DDL filter がない限り DDL をレプリケートしません

それらのために定義されています。

関数 bdr.replicate_ddl_command() を使用して、特定のレプリケーションセットに DDL をレプリケートすることもできます。これは、ノードがダウンしているときに DDL コマンドを実行する場合、またはノードまたは担当者セットのサブセット、たとえば site1 のすべてのノードに存在するインデックスまたはパーティションが必要な場合に役立ちます。

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

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

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

設定すると、PGD はグローバルロックと実行された DDL コマンドのレプリケーションの両方をスキップします。次に、すべてのノードで DDL を手動で実行する必要があります。

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

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

8.4.2 PGD システムで DDL を実行する

PGD グループは、スタンドアロンの PostgreSQL サーバーと同じではありません。これは、中央ロックとトランザクションコーディネーターのない非同期マルチマスターレプリケーションに基づいています。これは、DDL の実行時に重要な意味を持ちます。

並列実行される DDL は、PGD でも引き続き実行されます。DDL 実行は、実行時に各ノードでのパラレル操作に影響を与えるパラメーターを尊重するため、ノード間の設定の違いに気付く場合があります。

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

PGD は、これらの問題に対して 3 つのレベルの保護を提供します。

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

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

`ddl_locking = off` は最も厳密でないオプションであり、一般的な使用では危険です。このオプションはロックを完全にスキップし、パフォーマンスのオーバーヘッドを回避するため、新しい空のデータベーススキーマを作成するときに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、スーパーユーザー、または `config file` でのみ設定できます。

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

8.4.3 DDL ロックの詳細

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

最初の種類はグローバル DDL ロックと呼ばれ、 `ddl_locking = on` の場合にのみ使用されます。グローバル DDL ロックは、各 DDL ステートメントの実行中に、他の DDL がクラスターで実行されるのを防ぎます。これにより、一般的なケースでは完全な正確さが保証されますが、多くの単純なケースでは厳しすぎます。PGD は、スキーマが変更されるトランザクションで初めて、DDL 操作のグローバルロックを取得します。これにより、クラスター内の DDL 実行トランザクションが効果的にシリアル化されます。言い換えれば、DDL の実行中は、異なるテーブルに影響を与える場合でも、ノード上の他の接続は別の DDL コマンドを実行できません。

DDL 操作のロックを取得するために、DDL を実行する排他的権利を付与するように要求します。ロック要求は通常のレプリケーションストリームによって送信され、ノードもレプリケーションストリームによって応答します。したがって、ノード（または少なくとも大部分のノード）がレプリケーションの遅延を大きくせずに実行することが重要です。そうしないと、ノードが DDL ロックを取得するまでに時間がかかる場合があります。大多数のノードが同意すると、DDL 実行が実行されます。

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

DDL を実行しているノードがクラスターで実行された以前のすべての DDL の効果を確認したことを確認するために、以前の DDL を実行したノードに追いつくまで待機します。現在の DDL を実行しているノードが、前の DDL を実行したノードに対してレプリケーションが遅れている場合、ロックの取得に時間がかかる場合があります。したがって、単一のノード、または他のノードで発生したレプリケーションの変更に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、グローバル DML ロックまたはリレーション DML ロックと呼ばれます。この種のロックは、`ddl_locking = on` または `ddl_locking = dml` のいずれかに使用され、DDL ステートメントにより、進行中の DML ステートメントが失敗する可能性があります。これらのエラーは、ユニーク制約、チェック制約、NOT NULL 制約などの制約を追加または変更したときに発生する可能性があります。リレーション DML ロックは、一度に 1 つのリレーションにのみ影響します。リレーション DML ロックは、レプリケーションがエラーで停止する可能性のある変更がキューにある間、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 ロックを取得するには、DDL を実行している PGD ノードが PGD 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し、そのテーブルへの保留中の変更がすべて排出されるまで待機するように要求します。すべてのノードが完全に追いつくと、DML ロックの発信者は自由にテーブルのスキーマ変更を実行し、他のノードにレプリケートできます。

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

保留中の DML 操作が排出されるのを待機すると、時間がかかり、レプリケーションが現在遅れている場合はさらに時間がかかる場合があります。これは、データの変更とは異なり、行の表現と制約に影響を与えるスキーマの変更は、構成されたすべてのノードに到達でき、現在の書き込み速度にかなりうまく対応している場合にのみ実行できることを意味します。ノードがダウンしているときにそのような DDL コマンドを実行する必要がある場合は、最初にダウンしているノードを構成から削除します。

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

> *PGD システムで DDL を実行する* で説明されているように、ロック動作は `bdr.ddl_locking`

パラメーターで指定されます。

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

一部の PGD 関数は、DDL を変更します。これらの関数には、各関数のドキュメントに記載されている DDL ロック動作が適用されます。

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

`ddl_locking = off` は、競合する DDL がなく、DDL が実行されるデータベースオブジェクトで競合する DML 操作がないことを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、

データへの進行中の変更が失われる可能性があります。ユーザーアプリケーションチームは、発生した問題を解決する必要があります。

場合によっては、同時に実行される DDL を適切にシリアル化できます。これらのシリアル化の失敗が発生すると、DDL が再実行される場合があります。

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

一部の PGD 管理機能は DDL のように動作します。つまり、グローバルロックを取得しようとし、DDL レプリケーションがアクティブな場合、アクションがレプリケートされます。レプリケートされた関数の完全なリストは、

DDL のように動作する PGD 関数 にリストされています。

一時テーブルで実行される DDL は、グローバルロックを必要としません。

現在のトランザクションで作成されたオブジェクトの ALTER または DROP には、グローバル DML ロックは必要ありません。

グローバル DDL ロックおよびグローバル DML ロックのモニタリングは、
Monitoring に示されています。

8.4.4 DDL の影響の最小化

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

- DDL の影響を最小限に抑えるには、DDL を実行するトランザクションを短くし、多くの行の変更と組み合わせないで、長時間実行される外部キーまたはその他の制約の再チェックを避けます。
- `ALTER TABLE` の場合、`ADD CONSTRAINT` を単独で使用するのではなく、`ADD CONSTRAINT NOT VALID` を使用してから、`VALIDATE CONSTRAINT` を使用した別のトランザクションを使用します。 `VALIDATE CONSTRAINT` は、すべてのノードで再生されるまで待機します。これにより、確認を受け取るまでに顕著な遅延が発生します。
- インデックスを作成するときは、可能な限り `CONCURRENTLY` オプションを使用します。

実行時間の長い DDL を実行する別の方法は、DDL レプリケーションを無効にしてから、各ノードで DDL ステートメントを個別に実行することです。次の例に示すように、単一の SQL ステートメントを使用してこれを行うこともできます。グローバルロックルールは引き続き適用されるため、このタイプの使用で自分自身をロックアウトしないように注意してください。これはより多くの回避策です。

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

`bdr.run_on_all_nodes()` テクニックを `CREATE INDEX CONCURRENTLY` で使用することをお勧めします。`CREATE INDEX CONCURRENTLY` はマルチトランザクションコマンドであるため、セッション全体で DDL レプリケーションを無効にする必要があることに注意してください。実行中の書き込みを防止するため、実稼働システムで `CREATE INDEX` を避けてください。 `REINDEX` は 3.6 までのバージョンで複製されましたが、PGD 3.7 以降では複製されません。`REINDEX` が保持する AccessExclusiveLocks のため、 `REINDEX` の使用を避けてください。

代わりに、PG12+または 2QPG11+で利用可能な `REINDEX CONCURRENTLY` （または `reindexdb --concurrently`）を使用してください。

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

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

複数の DDL ステートメントは、個々のステートメントとして起動されるのではなく、単一のトランザクションにまとめる方が役立つ場合があるため、DDL ロックは 1 回だけ取得します。テーブルレベルのロックが通常の操作を妨げる場合、これは望ましくない場合があります。

DDL がシステムを長時間保持している場合は、psql または `pg_cancel_backend()` で **Control-C** を使用して、元のノードで DDL を安全にキャンセルできます。他のノードから DDL ロックをキャンセルすることはできません。

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

DDL 操作が発信元ノードでコミットされると、キャンセルまたは中止することはできません。PGD グループは、グローバルロックを確認した他のノードに正常に適用され、再生を確認応答するまで待機する必要があります。このため、DDL トランザクションは短く高速に保ちます。

8.4.5 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なままです。タイムアウトが設定されていても、グローバルロックはタイムアウトしません。ノードが復旧すると、保持しているすべてのグローバルロックを解放します。

長時間または無期限にダウンしたままの場合は、ノードを PGD グループから削除して、グローバルロックを解放します。これが、bdr_superuser として `SET` コマンドを使用して緊急 DDL を実行し、`bdr.ddl_locking` 値を更新する理由の 1 つです。

他のノードの 1 つが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ダウンした場合、ロックを要求するそのコマンドの実行は、ノードが稼働しているかのように続行されます。

前述のように、グローバル DDL ロックはノードの過半数のみが応答する必要があるため、クラスターの一部がダウンしていても、過半数が実行されていて到達可能である限り機能します。ただし、クラスター全体が利用可能でない限り、DML ロックを取得することはできません。

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

8.4.6 ステートメント固有の DDL レプリケーションに関する懸念

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

PGD は、データベースでアクティブになっているときに、一部の DDL ステートメントの実行を防ぎます。これにより、正しくレプリケートできないステートメント、またはレプリケーションがまだサポートされていないステートメントを禁止することにより、システムの一貫性が保護されます。

PGD でステートメントが許可されていない場合、多くの場合、同じことを行う別の方法を見つけることができます。たとえば、揮発性のデフォルト値を持つ列を追加する ALTER TABLE を実行することはできません。ただし、一般的には、動作する一連の独立した ALTER TABLE および UPDATE ステートメントとしてそれを言い換えることができます。

一般に、サポートされていないステートメントの実行は防止され、feature_not_supported (SQLSTATE 0A000) エラーが発生します。

一時オブジェクトを参照または依存する DDL は、PGD によってレプリケートできず、DDL レプリケーションを有効にして実行するとエラーがスローされます。

8.4.7 PGD DDL コマンド処理マトリックス

次の表では、許可されるユーティリティまたは DDL コマンド、レプリケートされるコマンド、およびレプリケート時にかかるグローバルロックの種類について説明します。

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

Command	Allowed	Replicated	Lock
ALTER AGGREGATE	Y	Y	DDL
ALTER CAST	Y	Y	DDL
ALTER COLLATION	Y	Y	DDL
ALTER CONVERSION	Y	Y	DDL
ALTER DATABASE	Y	N	N
ALTER DATABASE LINK	Y	Y	DDL
ALTER DEFAULT PRIVILEGES	Y	Y	DDL
ALTER DIRECTORY	Y	Y	DDL
ALTER DOMAIN	Y	Y	DDL
ALTER EVENT TRIGGER	Y	Y	DDL
ALTER EXTENSION	Y	Y	DDL
ALTER FOREIGN DATA WRAPPER	Y	Y	DDL
ALTER FOREIGN TABLE	Y	Y	DDL
ALTER FUNCTION	Y	Y	DDL
ALTER INDEX	Y	Y	DDL
ALTER LANGUAGE	Y	Y	DDL
ALTER LARGE OBJECT	N	N	N

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 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	アーキテクチャの詳細
ALTER PUBLICATION	Y	Y	DDL
ALTER QUEUE	Y	Y	DDL
ALTER QUEUE TABLE	Y	Y	DDL
ALTER REDACTION POLICY	Y	Y	DDL
ALTER RESOURCE GROUP	Y	N	N
ALTER ROLE	Y	Y	DDL
ALTER ROUTINE	Y	Y	DDL
ALTER RULE	Y	Y	DDL
ALTER SCHEMA	Y	Y	DDL
ALTER SEQUENCE	アーキテクチャの詳細	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	アーキテクチャの詳細	Y	アーキテクチャの詳細
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

次のページに続く

表 1 – 前のページからの続き

Command	Allowed	Replicated	Lock
CLUSTER	Y	N	N
COMMENT	Y	アーキテクチャの詳細	DDL
COMMIT	Y	N	N
COMMIT PREPARED	Y	N	N
COPY	Y	N	N
COPY FROM	Y	N	N
CREATE ACCESS METHOD	Y	Y	DDL
CREATE AGGREGATE	Y	Y	DDL
CREATE CAST	Y	Y	DDL
CREATE COLLATION	Y	Y	DDL
CREATE CONSTRAINT	Y	Y	DDL
CREATE CONVERSION	Y	Y	DDL
CREATE DATABASE	Y	N	N
CREATE DATABASE LINK	Y	Y	DDL
CREATE DIRECTORY	Y	Y	DDL
CREATE DOMAIN	Y	Y	DDL
CREATE EVENT TRIGGER	Y	Y	DDL
CREATE EXTENSION	Y	Y	DDL
CREATE FOREIGN DATA WRAPPER	Y	Y	DDL
CREATE FOREIGN TABLE	Y	Y	DDL
CREATE FUNCTION	Y	Y	DDL
CREATE INDEX	Y	Y	DML
CREATE LANGUAGE	Y	Y	DDL
CREATE MATERIALIZED VIEW	Y	N	N
CREATE OPERATOR	Y	Y	DDL
CREATE OPERATOR CLASS	Y	Y	DDL
CREATE OPERATOR FAMILY	Y	Y	DDL
CREATE PACKAGE	Y	Y	DDL
CREATE PACKAGE BODY	Y	Y	DDL
CREATE POLICY	Y	Y	DML
CREATE PROCEDURE	Y	Y	DDL
CREATE PROFILE	Y	Y	アーキテクチャの詳細
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

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 SEQUENCE	アーキテクチャの詳細	Y	DDL
CREATE SERVER	Y	Y	DDL
CREATE STATISTICS	Y	Y	DDL
CREATE SUBSCRIPTION	Y	Y	DDL
CREATE SYNONYM	Y	Y	DDL
CREATE TABLE	アーキテクチャの詳細	Y	DDL
CREATE TABLE AS	アーキテクチャの詳細	Y	DDL
CREATE TABLESPACE	Y	N	N
CREATE TEXT SEARCH CONFIGURATION	Y	Y	DDL
CREATE TEXT SEARCH DICTIONARY	Y	Y	DDL
CREATE TEXT SEARCH PARSER	Y	Y	DDL
CREATE TEXT SEARCH TEMPLATE	Y	Y	DDL
CREATE TRANSFORM	Y	Y	DDL
CREATE TRIGGER	Y	Y	DDL
CREATE TYPE	Y	Y	DDL
CREATE TYPE BODY	Y	Y	DDL
CREATE USER MAPPING	Y	Y	DDL
CREATE VIEW	Y	Y	DDL
DEALLOCATE	Y	N	N
DEALLOCATE ALL	Y	N	N
DECLARE CURSOR	Y	N	N
DISCARD	Y	N	N
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

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 FOREIGN DATA WRAPPER	Y	Y	DDL
DROP FOREIGN TABLE	Y	Y	DDL
DROP FUNCTION	Y	Y	DDL
DROP INDEX	Y	Y	DDL
DROP LANGUAGE	Y	Y	DDL
DROP MATERIALIZED VIEW	Y	N	N
DROP OPERATOR	Y	Y	DDL
DROP OPERATOR CLASS	Y	Y	DDL
DROP OPERATOR FAMILY	Y	Y	DDL
DROP OWNED	Y	Y	DDL
DROP PACKAGE	Y	Y	DDL
DROP PACKAGE BODY	Y	Y	DDL
DROP POLICY	Y	Y	DDL
DROP PROCEDURE	Y	Y	DDL
DROP PROFILE	Y	Y	DDL
DROP PUBLICATION	Y	Y	DDL
DROP QUEUE	Y	Y	DDL
DROP QUEUE TABLE	Y	Y	DDL
DROP REDACTION POLICY	Y	Y	DDL
DROP RESOURCE GROUP	Y	N	N
DROP ROLE	Y	Y	DDL
DROP ROUTINE	Y	Y	DDL
DROP RULE	Y	Y	DDL
DROP SCHEMA	Y	Y	DDL
DROP SEQUENCE	Y	Y	DDL
DROP SERVER	Y	Y	DDL
DROP STATISTICS	Y	Y	DDL
DROP SUBSCRIPTION	Y	Y	DDL
DROP SYNONYM	Y	Y	DDL
DROP TABLE	Y	Y	DML
DROP TABLESPACE	Y	N	N
DROP TEXT SEARCH CONFIGURATION	Y	Y	DDL
DROP TEXT SEARCH DICTIONARY	Y	Y	DDL
DROP TEXT SEARCH PARSER	Y	Y	DDL
DROP TEXT SEARCH TEMPLATE	Y	Y	DDL
DROP TRANSFORM	Y	Y	DDL
DROP TRIGGER	Y	Y	DDL
DROP TYPE	Y	Y	DDL
DROP TYPE BODY	Y	Y	DDL
DROP USER MAPPING	Y	Y	DDL
DROP VIEW	Y	Y	DDL
EXECUTE	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
EXPLAIN	Y	アーキテクチャの詳細	アーキテクチャの詳細
FETCH	Y	N	N
GRANT	Y	アーキテクチャの詳細	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	アーキテクチャの詳細
MOVE	Y	N	N
NOTIFY	Y	N	N
PREPARE	Y	N	N
PREPARE TRANSACTION	Y	N	N
REASSIGN OWNED	Y	Y	DDL
REFRESH MATERIALIZED VIEW	Y	N	N
REINDEX	Y	N	N
RELEASE	Y	N	N
RESET	Y	N	N
REVOKE	Y	アーキテクチャの詳細	DDL
REVOKE ROLE	Y	Y	DDL
ROLLBACK	Y	N	N
ROLLBACK PREPARED	Y	N	N
SAVEPOINT	Y	N	N
SECURITY LABEL	Y	アーキテクチャの詳細	DDL
SELECT INTO	アーキテクチャの詳細	Y	DDL
SET	Y	N	N
SET CONSTRAINTS	Y	N	N
SHOW	Y	N	N
START TRANSACTION	Y	N	N
TRUNCATE TABLE	Y	アーキテクチャの詳細	アーキテクチャの詳細
UNLISTEN	Y	N	N
VACUUM	Y	N	N

ALTER シーケンス

一般に ALTER SEQUENCE がサポートされていますが、グローバルシーケンスを使用する場合、一部のオプションは効果がありません。

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

ALTER テーブル

一般に、 ALTER TABLE コマンドは許可されています。ただし、いくつかのサブコマンドはサポートされていません。

ALTER TABLE は許可されないコマンド

ALTER TABLE のいくつかのバリアントは現在、PGD ノードでは許可されていません。

- ADD COLUMN ... DEFAULT (non-immutable expression) —現在、異なるノードで異なるデータが得られるため、これは許可されません。推奨される回避策については、列の追加 を参照してください。
- ADD CONSTRAINT ... EXCLUDE —除外制約は現在サポートされていません。除外制約は非同期システムではあまり意味がなく、再生できない変更につながります。
- ALTER TABLE ... SET WITH[OUT] OIDS — CREATE TABLE と同じ理由でサポートされていません。
- ALTER COLUMN ... SET STORAGE external —列がテーブルのレプリカ ID の列の 1 つである場合、拒否されます。
- RENAME —自動パーティションテーブルの名前を変更することはできません。
- SET SCHEMA —自動パーティション分割テーブルのスキーマを設定できません。
- ALTER COLUMN ... TYPE —コマンドによってテーブル全体が書き換えられる場合、列の型の変更はサポートされていません。これは、変更がバイナリ強制ではない場合に発生します。バイナリ強制変更は、一方向にのみ許可される場合があります。たとえば、VARCHAR(128) から VARCHAR(256) への変更はバイナリ強制的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的ではないため、通常は許可されません。レプリケートされない ALTER COLUMN ... TYPE は、列が新しい型に自動的にキャスト可能な場合に許可されます（USING 句が含まれていません）。次に例を示します。テーブルの書き換えは、より大きなテーブルで長期間 AccessExclusiveLock を保持するため、このようなコマンドは、いずれの場合も高可用性データベースでは実行できない可能性があります。推奨される回避策については、列のタイプを変更する を参照してください。
- ALTER TABLE ... ADD FOREIGN KEY —現在のユーザーに参照先テーブルを読み取る権限がない場合、または参照先テーブルで現在のユーザーがバイパスできない RLS 制限が有効になっている場合、はサポートされません。

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

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

PGD 3.7.4 以降、DML ロックを取得せずに、 CHECK および FOREIGN KEY 制約などの特定のタイプの制約を追加できます。ただし、これには、最初に NOT VALID 制約を作成し、次に ALTER TABLE ... VALIDATE CONSTRAINT コマンドを使用して別のトランザクションで制約を検証するという 2 段階のプロセスが必要です。

CONSTRAINT の追加 を参照

詳細については。

ALTER TABLE ロック

ALTER TABLE の次のバリアントは、DDL ロックのみを取得し、DML ロックは取得しません。

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

ALTER TABLE の他のすべてのバリアントは、変更されるテーブルで DML ロックを取得します。ALTER TABLE の一部のバリアントには、以下に示す制限があります。

ALTER TABLE の例

この次の例は、型の変更がバイナリ強制的であり、テーブルの書き換えを引き起こさないため、機能します。カタログのみの変更として実行されます。

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

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

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

回避策については、 制限付き *DDL* 回避策 を参照してください。

一般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役立ちます。

一部の ATCT 操作は、基になる列タイプのメタデータのみを更新し、基になるテーブルデータの書き換えを必要としません。これは通常、既存の列の型とターゲットの型がバイナリ強制型である場合に当てはまります。例：

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

バイナリ強制性のため、列の型を VARCHAR または TEXT データ型に変更することもできます。繰り返しますが、これは基になる列タイプのメタデータの更新にすぎません。

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

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

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

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

これらの両方の失敗したケースでは、現在の型からターゲット型への自動割り当てキャストがあります。ただし、バイナリ強制性はなく、基になるテーブルデータの書き換えが発生します。

このような場合、制御された DBA 環境では、すべてのノードの非レプリケート環境でこの列の型のローリングアップグレードを 1 つずつ採用することにより、列の型を自動的にキャスト可能なものに変更できます。DDL がレプリケートされず、列の型が自動的にキャスト可能なものに変更されたとします。次に、変更を実行しているノードでローカルに書き換えを許可し、この同じテーブルの他のノードでの同時アクティビティを許可できます。次に、すべてのノードでこのレプリケートされない ATCT 操作を 1 つずつ繰り返して、EDB Postgres 分散クラスター全体で列タイプの目的の変更をもたらすことができます。これには書き換えが含まれるため、アクティビティは引き続き短期間 DML ロックを取得するため、クラスター全体が使用可能である必要があります。これらの詳細が整ったら、次のようにレプリケートされない変更アクティビティのローリングアップグレードを実行できます。

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

多くのデータ型で自動割り当てキャストを使用できるため、このローカルのレプリケートされない ATCT 操作は、さまざまな変換をサポートします。また、USING 句を使用する ATCT 操作は、自動割り当てキャストがないため失敗する可能性があります。この例は、自動割り当てキャストを使用したいくつかの一般的な変換を示しています。

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

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

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

レプリケートされない環境での ATCT 操作はさまざまな型変換をサポートしていますが、基になるテーブルデータに新しいデータ型に割り当てることができない値が含まれている場合、書き換えが失敗する可能性があります。たとえば、列の現在の型が VARCHAR(256) であり、レプリケートされない ATCT 操作を試

行して VARCHAR(128) に変換するとします。テーブルに 128 バイトを超える既存のデータがある場合、書き換え操作はローカルで失敗します。

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

基になるテーブルのデータが新しい型の特性を満たしている場合、書き換えは成功します。ただし、レプリケートされないローリングデータ型のアップグレードをまだ実行していない他のノードが 128 バイトを超える新しいデータをこのローカル ATCT 操作に同時に導入した場合、レプリケーションは失敗する可能性があります。これにより、クラスター内のレプリケーションが停止します。したがって、これらのレプリケートされないローリングデータ型アップグレード操作を実行するときは、データベースおよびアプリケーションレベルでのデータ型の制限と特性に注意してください。このような ATCT 操作を、制御され完全に認識した DBA 環境で実行およびテストすることを強くお勧めします。これらの ATCT 操作は非対称であり、失敗した特定の変更をバックアウトすると、テーブルの書き換えに時間がかかる場合があります。

また、トランザクションブロックで暗黙的なキャスト可能な ALTER アクティビティを実行することはできません。

ALTER タイプ

ALTER TYPE は複製されますが、PostgreSQL はこれらの依存関係を記録しないため、そのデータ型を使用するすべてのテーブルにグローバル DML ロックが適用されるわけではありません。

> 制限付き *DDL* 回避策 を参照してください。

コメント

COMMENT ON のすべてのバリアントが許可されますが、COMMENT ON TABLESPACE/DATABASE/LARGE OBJECT は複製されません。

CREATE PROFILE または ALTER PROFILE

機能が SECURITY DEFINER の場合、プロファイルに関連付けられた PASSWORD_VERIFY_FUNCTION は IMMUTABLE でなければなりません。このような CREATE PROFILE または ALTER PROFILE コマンドは複製されますが、このプロファイルを使用する後続の CREATE USER または ALTER USER コマンドは、writer ワーカーがエラー：cannot change current role within security-restricted operation をスローしたため、レプリケーションを中断します。

シーケンスを作成する

一般に CREATE SEQUENCE がサポートされていますが、グローバルシーケンスを使用する場合、一部のオプションは効果がありません。

テーブルを作成する

一般に、 CREATE TABLE はサポートされていますが、CREATE TABLE WITH OIDS は PGD ノードでは許可されていません。

CREATE TABLE AS と SELECT INTO

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

説明する

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

EXPLAIN ANALYZE レプリケーション

EXPLAIN ANALYZE は、分析されたステートメントの複製ルールに従います。

EXPLAIN ANALYZE ロック

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

GRANT と REVOKE

一般に、GRANT および REVOKE ステートメントはサポートされていますが、GRANT/REVOKE ON TABLESPACE/LARGE OBJECT は複製されません。

ロックテーブル

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

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

セキュリティラベル

SECURITY LABEL のすべてのバリアントが許可されますが、SECURITY LABEL ON TABLESPACE/DATABASE/LARGE OBJECT は複製されません。

TRUNCATE レプリケーション

TRUNCATE コマンドは、DDL ステートメントとしてではなく、DML としてレプリケートされます。テーブルの TRUNCATE がレプリケートされるかどうかは、影響を受ける各テーブルのレプリケーション設定によって異なります。

TRUNCATE ロック

TRUNCATE は他の DDL と同じように複製されませんが、bdr.truncate_locking が on に設定されていると、グローバル DML ロックを取得できます。

ロール操作ステートメント

ユーザーは PostgreSQL インスタンスのグローバルオブジェクトです。つまり、PGD は個々のデータベースレベルで動作しますが、ユーザーは複数のデータベースにまたがります。これは、ロール操作ステートメントの処理には特別な配慮が必要であることを意味します。

PGD では、レプリケートされた DDL によって参照されるロールがすべてのノードに存在する必要があります。ロールは、同じ付与、パスワードなどを持つ必要はありませんが、存在している必要があります。

bdr.role_replication が有効になっている場合（デフォルト）、PGD はロール操作ステートメントをレプリケートし、ロール操作ステートメントが PGD 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ものが含まれます。

- CREATE ROLE
- ALTER ROLE
- DROP ROLE
- GRANT ROLE
- CREATE USER
- ALTER USER
- DROP USER
- CREATE GROUP
- ALTER GROUP
- DROP GROUP

一般に、次のいずれかです。

- bdr.role_replication = off でシステムを構成し、Ansible、Puppet、Chef などの外部オーケストレーションツールで、または bdr.replicate_ddl_command(...) によって明示的に複製されたすべてのロールの変更（ユーザーとグループ）を展開します。
- PostgreSQL インスタンス上のちょうど 1 つの PGD 対応データベースが bdr.role_replication = on を持つようにシステムを構成し、そのデータベースですべてのロール管理 DDL を実行します。

すべてのロール管理コマンドを 1 つのデータベースで実行することをお勧めします。

ロールのレプリケーションがオフになっている場合、管理者は、1 つのノードで DDL が使用するロールが他のノードにも存在することを確認する必要があります。そうしないと、他のノードでロールが作成されるまで、PGD apply はエラーでストールします。

注釈: PGD は、PGD 対応の PostgreSQL インスタンス内の PGD 非対応のデータベースで実行される場合、ロール管理ステートメントをキャプチャおよびレプリケートしません。たとえば、DB 'bdrdb'（bdr グループメンバー）と'postgres'（ベア db）、および `bdr.role_replication = on` がある場合、 *bdrdb* で実行される `CREATE USER` は複製されますが、 *postgres* で実行される `CREATE USER` は複製されません。

制限付き DDL 回避策

PGD DDL 操作処理の制限の一部を回避できます。多くの場合、操作を小さな変更に分割すると、単一のステートメントとして許可されないか、過度のロックが必要な目的の結果が生成される可能性があります。

CONSTRAINT の追加

DML ロックを必要とせずに CHECK および FOREIGN KEY 制約を追加できます。これには 2 段階のプロセスが含まれます。

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

2 つの異なるトランザクションでこれらの手順を実行します。これらの手順はどちらもテーブルでのみ DDL ロックを取得するため、1 つ以上のノードがダウンしている場合でも実行できます。ただし、制約を検証するには、PGD で次のことを保証する必要があります。

- クラスター内のすべてのノードは ADD CONSTRAINT コマンドを参照します。
- 制約が適用されたレプリケーションを検証するノードは、それらのノードに NOT VALID 制約を作成する前に、他のすべてのノードからの変更を行います。

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稼働させる必要はありませんが、すべてのノードが ALTER TABLE .. ADD CONSTRAINT ... NOT VALID コマンドを適用し、十分に進行したことは引き続き必要です。PGD は、制約を検証する前に、一貫した状態に到達するのを待ちます。

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

列の追加

揮発性のデフォルトを持つ列を追加するには、別のトランザクションで次のコマンドを実行します。

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

ALTER TABLE mytable ALTER COLUMN newcolumn DEFAULT volatile-expression;

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

このアプローチは、スキーマの変更と行の変更を個別のトランザクションに分割し、PGD が実行できるため、PGD グループ内のすべてのノードで一貫したデータが得られます。

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

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

必要に応じて、UPDATE が終了した後に 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

列の型を変更すると、PostgreSQL がテーブルを書き換える可能性があります。場合によっては、この書き換えを回避できます。例：

```
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を書き換えて、制限をデータ型の変更ではなくテーブル制約にすることにより、テーブルの書き換えを回避できます。必要に応じて、後続のコマンドで制約を検証して、長いロックを回避できます。

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

検証が失敗した場合、失敗した行だけを UPDATE できます。 length() を使用して TEXT および VARCHAR 、または scale() を使用して NUMERIC データ型でこの手法を使用できます。

列の種類を変更する一般的な場合、最初に目的の種類の列を追加します。

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

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

テーブルをバッチで UPDATE 、埋め込みトランザクションで PROCEDURE を使用して oldcolumn の値を newcolumn にコピーします。作業をバッチ処理すると、大きなテーブルの場合、レプリケーションのラグを減らすことができます。ID の範囲または任意の方法で更新しても問題ありません。または、小さなテーブルの場合、テーブル全体を 1 回のパスで更新できます。

CREATE INDEX ... 新しい列に必要なインデックス。ロック期間を短縮するために、各ノードで DDL レプリケーションなしで CREATE INDEX ... CONCURRENTLY 実行を個別に使用しても安全です。

ALTER NOT NULL および CHECK 制約を追加する列、必要に応じて。

1. BEGIN トランザクション。

1. 追加したトリガーを DROP します。

1. ALTER TABLE を使用して、列に必要な DEFAULT を追加します。

1. 古い列を DROP 。

1. ALTER TABLE mytable RENAME COLUMN newcolumn TO oldcolumn 。

2. COMMIT 。

注釈: 列を削除するため、テーブルに依存するビュー、プロシージャなどの再作成が必要になる場合があります。列を `CASCADE` ドロップする場合は注意してください。それを参照していたものをすべて再作成する必要があるためです。

他のタイプの変更

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

この DDL を使用する場合は、新しいタイプを使用する前に、ステートメントがすべてのノードで正常に実行されたことを確認してください。bdr.wait_slot_confirm_lsn() 関数を使用してこれを実現できます。

この例では、DML ステートメントで新しい値を使用する前に、DDL がすべてのノードに書き込まれるようにします。

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

DDL のように動作する PGD 関数

次の PGD 管理関数は、 DDL のように機能します。これは、DDL レプリケーションがアクティブであり、DDL フィルター設定で許可されている場合、グローバルロックを取得しようとし、アクションがレプリケートされることを意味します。詳細については、個々の機能の文書を参照してください。

レプリケーションセット管理

- bdr.create_replication_set
- bdr.alter_replication_set
- bdr.drop_replication_set
- bdr.replication_set_add_table
- bdr.replication_set_remove_table
- bdr.replication_set_add_ddl_filter
- bdr.replication_set_remove_ddl_filter

コンフリクトマネジメント

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

シーケンス管理

- bdr.alter_sequence_set_kind

ストリームトリガー

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

8.5 Security and roles

スーパーユーザーのみが BDR 拡張機能を作成できます。ただし、必要に応じて、pgextwlist 拡張機能を設定し、非スーパーユーザーが BDR 拡張機能を作成できるように構成できます。

PGD の構成と管理にスーパーユーザーアクセスは必要ありません。PGD に必要な特権は、PostgreSQL のデフォルトおよび事前定義ロールと同様に名前付けされた、次のデフォルトおよび事前定義ロールに分割されます。

- bdr_superuser —すべての PGD テーブルと関数へのアクセスを持つ最も高い特権のロール。
- bdr_read_all_stats —PGD 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を持つロール。

- bdr_monitor —現時点では、 bdr_read_all_stats と同じ。後日延長予定。
- bdr_application & mdash; PGD を実行するアプリケーションに必要な最小限の特権。
- bdr_read_all_conflicts — `bdr.conflict_history` のすべての競合を表示できます。

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

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

紛争を保護するための手配は、 競合をテーブルに記録する で議論されています。

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

8.5.1 カタログテーブル

システムカタログおよび情報スキーマテーブルは、PGD によるレプリケーションから常に除外されます。

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。

8.5.2 PGD 関数と演算子

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

すべての PGD 演算子は `pg_catalog` スキーマを介して利用可能であるため、ユーザーは search_path から `public` スキーマを問題なく除外できます。

8.5.3 カタログオブジェクトに対する特権の付与

管理者は、テーブル、ビュー、関数などのカタログオブジェクトに対する明示的な特権を付与してはなりません。

PGD default roles で説明されているロールのいずれかを付与して、これらのオブジェクトへのアクセスを管理します。

この要件は、参加のいずれかの側のノードが完全に同じバージョンの PGD、したがって PGD カタログのバージョン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た場合、`bdr.join_node_group()` プロシージャは失敗する可能性があります。結合されるノードから抽出された対応する GRANT ステートメントは、参加するノードに適用されない可能性があるためです。

8.5.4 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です。CREATE USER および CREATE ROLE コマンドは、PGD が実行されており、 bdr.role_replication がオンになっているデータベースで実行される場合、自動的に複製されます。ただし、これらのコマンドが同じ PostgreSQL インスタンス内の他のデータベースで実行された場合、それらのユーザーが PGD データベースに対する権限を持っている場合でも、それらはレプリケートされません。

新しい PGD ノードが PGD 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。この動作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、ユーザーが複数のデータベースにアクセスできます。デフォルトでは、任意のデータベースにアクセスできます。PGD は、各ユーザーがデータベースに持っているアクセスを知らないため、新しいノードにコピーするユーザーを安全に決定できません。

PostgreSQL では、次のコマンドを使用してすべてのユーザーをダンプできます。

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

次に、新しく作成したノードで再実行する前に、ファイル roles.sql を編集して不要なユーザーを削除できます。ID とアクセス管理ソリューション（IAM）に応じて、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。

8.5.5 ロールとレプリケーション

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

テーブルへの DML 変更は、ターゲットノードでテーブル所有ユーザーとしてレプリケートされます。各ノードで同じユーザーがテーブルを所有することをお勧めしますが、強制はしません。

テーブル A が node1 のユーザー X が所有し、node2 のユーザー Y が所有しているとします。ユーザー Y がユーザー X よりも高い特権を持っている場合、これは特権エスカレーションと見なされる場合があります。一部のノードにはさまざまなユースケースがあるため、これを許可しますが、セキュリティ管理者がこの状況を計画および監査できるように警告します。

行レベルのセキュリティポリシーが有効になっているテーブルでは、適用時にポリシーを再適用せずに変更がレプリケートされます。この動作は、FORCE ROW LEVEL SECURITY が指定されている場合でも、NO FORCE ROW LEVEL SECURITY として適用される変更と同等です。これが望ましくない場合は、すべての行の複製を回避する row_filter を指定します。すべてのノードの行セキュリティポリシーが同一であるか、少なくとも互換性があることをお勧めしますが、強制はしません。

ユーザー bdr_superuser は、PGD のレプリケーションを制御し、レプリケーションセットにテーブルを追加または削除できます。bdr_superuser には、個々のテーブルに対する特権は必要なく、お勧めしません。レプリケーションセットファンクションへのアクセスを制限する必要がある場合は、これらのファンクションの制限バージョンを SECURITY DEFINER ファンクションとして実装し、適切なユーザーに付与できます。

8.5.6 接続ロール

新しい PGD ノードを割り当てるとき、bdr.create_node の local_dsn 引数および bdr.join_node_group の join_target_dsn に DSN で指定されたユーザーは、データベースオブジェクトの参照、作成、管理に頻繁に使用されます。

PGD 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、特権エスカレーション攻撃を防ぐように慎重に作成されています。

攻撃対象領域をさらに減らすには、上記の DSN でより制限されたユーザーを指定できます。少なくとも、そのようなユーザーには、次の規定が満たされるように、すべてのノードに対する権限を付与する必要があります。

- ユーザーが REPLICATION 属性を持っています。
- データベースに対する CREATE 権限が付与されています。

・ bdr_superuser ロールを継承します。

- 直接または所有者ロールの権限から、レプリケートするすべてのデータベースオブジェクトを所有しています。

すべてのノードが参加したら、権限をさらに次のように減らして、DML および DDL レプリケーションを引き続き許可できます。

- ユーザーが REPLICATION 属性を持っています。

・ bdr_superuser ロールを継承します。

8.5.7 特権制限

PGD は追加の制限を適用し、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。

GRANT ALL は引き続き TRIGGER 特権と REFERENCES 特権の両方を付与するため、特権を明示的に述べることをお勧めします。たとえば、ALL の代わりに GRANT SELECT, INSERT, UPDATE, DELETE, TRUNCATE を使用します。

外部キー特権

ALTER TABLE ... ADD FOREIGN KEY は、ユーザーが参照先のテーブルに対する SELECT 権限を持っている場合、または参照先のテーブルで現在のユーザーがバイパスできない RLS 制限が有効になっている場合にのみサポートされます。

したがって、REFERENCES 特権は、PGD で外部キーを作成するのに十分ではありません。REFERENCES 特権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検証チェックを実行するため、通常は役に立ちません。通常、正常に使用するには高すぎます。

トリガー

PostgreSQL では、テーブルの所有者と TRIGGER 権限を付与された人の両方がトリガーを作成できます。非テーブル所有者によって付与されたトリガーは、PGD でテーブル所有者として実行され、セキュリティ問題が発生する可能性があります。TRIGGER 権限はめったに使用されません。PostgreSQL コアチームは、「個別の TRIGGER 権限は、廃止されたと考えるものです」と述べています。

PGD は、テーブルでトリガーを作成できるユーザーに関するより厳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。

- スーパーユーザー。
- bdr_superuser。
- テーブルの所有者は、PostgreSQL と同じルールに従ってトリガーを作成できます（トリガーで使用される関数の EXECUTE 権限が必要です）。
- テーブルの TRIGGER 権限を持つユーザーは、テーブルと同じ所有者が所有する関数を使用してトリガーを作成し、標準の PostgreSQL ルールを満たしている場合にのみトリガーを作成できます。（ファンクションに対する EXECUTE 特権が必要です。）テーブルとファンクションの両方が同じ所有者を持ち、所有者がユーザにテーブルに対する TRIGGER 特権とファンクションに対する EXECUTE 特権の両方を与えることにしたとします。その場合、そのユーザーはこの関数を使用してそのテーブルにトリガーを作成してもよいと想定されます。
- 表の TRIGGER 権限を持つユーザーは、EXECUTE 権限を持っている場合、SECURITY DEFINER 句で定義された関数を使用してトリガーを作成できます。この句により、標準の PostgreSQL と PGD の両方で、関数が常に関数の所有者のコンテキストで実行されます。

このロジックは、PostgreSQL では、トリガーの所有者はそれを作成したユーザーではなく、そのトリガーで使用される関数の所有者であるという事実に基づいています。

同じルールが既存のテーブルに適用され、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、レプリケーションセットに追加できません。

これらのチェックは PGD 3.6.19 で追加されました。以前のバージョンの動作に依存するアプリケーションは、`bdr.backwards_compatibility` を 30618 以下に設定して、以前のバージョンと同様に動作できます。

PGD レプリケーションの適用は、システムレベルのデフォルトの search_path のみを使用します。レプリカトリガー、ストリームトリガー、およびインデックス式関数は、他の search_path 設定を想定し、適用時に実行すると失敗する場合があります。これが発生しないようにするには、デフォルトの search_path のみを使用してオブジェクト参照を明確に解決するか、影響を受ける関数の `ALTER FUNCTION ... SET search_path = ...` を使用して関数の検索パスを設定します。デフォルトの search_path を使用する場合は、常にオブジェクトへの完全修飾参照（schema.objectname など）を使用します。

8.5.8 PGD のデフォルト/事前定義されたロール

PGD の事前定義された役割は、 BDR 拡張機能のインストール時に作成されます。BDR 拡張機能がデータベースから削除された後、ロールは引き続き存在します。ドロップが必要な場合は、手動でドロップする必要があります。このプラクティスにより、PGD を同じ PostgreSQL インスタンス上の複数のデータベースで問題なく使用できます。

`GRANT ROLE` DDL ステートメントは、PGD レプリケーションに参加しません。したがって、クラスターの各ノードでこれを実行します。

bdr_superuser

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

bdr_read_all_stats

以下に対する SELECT 特権：

- `bdr.conflict_history_summary`
- `bdr.ddl_epoch`
- `bdr.ddl_replication`
- `bdr.global_consensus_journal_details`
- `bdr.global_lock`
- `bdr.global_locks`
- `bdr.local_consensus_state`
- `bdr.local_node_summary`
- `bdr.node`
- `bdr.node_catchup_info`
- `bdr.node_conflict_resolvers`
- `bdr.node_group`
- `bdr.node_local_info`
- `bdr.node_peer_progress`
- `bdr.node_slots`
- `bdr.node_summary`
- `bdr.replication_sets`
- `bdr.sequences`

- bdr.stat_relation
- bdr.stat_subscription
- bdr.subscription
- bdr.subscription_summary
- bdr.tables

以下に対する EXECUTE 特権：

- bdr.bdr_version
- bdr.bdr_version_num
- bdr.decode_message_payload
- bdr.get_global_locks
- bdr.get_raft_status
- bdr.get_relation_stats
- bdr.get_slot_flush_timestamp
- bdr.get_sub_progress_timestamp
- bdr.get_subscription_stats
- bdr.peer_state_name
- bdr.show_subscription_status

bdr_monitor

bdr_read_all_stats からのすべての特権。

また、以下に対する EXECUTE 特権：

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

bdr_application

以下に対する EXECUTE 特権：

- column_timestamps データ型のすべての関数
- CRDT データ型のすべての関数
- bdr.alter_sequence_set_kind
- bdr.create_conflict_trigger

- bdr.create_transform_trigger
- bdr.drop_trigger
- bdr.get_configured_camo_partner
- bdr.global_lock_table
- bdr.is_camo_partner_connected
- bdr.is_camo_partner_ready
- bdr.logical_transaction_status
- bdr.ri_fkey_trigger
- bdr.seq_nextval
- bdr.seq_currval
- bdr.seq_lastval
- bdr.trigger_get_committs
- bdr.trigger_get_conflict_type
- bdr.trigger_get_origin_node_id
- bdr.trigger_get_row
- bdr.trigger_get_type
- bdr.trigger_get_xid
- bdr.wait_for_camo_partner_queue
- bdr.wait_slot_confirm_lsn

これらの機能の多くは、使用する前に追加の特権が必要です。たとえば、bdr.alter_sequence_set_kind を正常に実行するには、テーブルの所有者である必要があります。これらの追加ルールについては、特定の機能ごとに説明します。

bdr_read_all_conflicts

PGD は、競合を bdr.conflict_history テーブルにログ記録します。競合はテーブルの所有者にのみ表示されるため、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。すべてのテーブルの競合を確認できるユーザーが便利な場合は、オプションでそのユーザーにロール bdr_read_all_conflicts を付与できます。

8.5.9 検証

PGD は、次のツールとアプローチを使用して検証されました。

コベリティ

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

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

CIS ベンチマーク

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

	Result	Description
1.4	PASS	systemd サービスファイルが有効になっていることを確認します
1.5	PASS	データクラスターが正常に初期化されたことを確認する
2.1	PASS	ファイルのアクセス許可マスクが正しいことを確認します
2.2	PASS	PostgreSQL pg_wheel グループメンバーシップが正しいことを確認します
3.1.2	PASS	ログの宛先が正しく設定されていることを確認します
3.1.3	PASS	ロギングコレクター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

次のページに続く

表 2 – 前のページからの続き

	Result	Description
3.1.6	PASS	ログファイルのアクセス許可が正しく設定されていることを確認します
3.1.7	PASS	「log_truncate_on_rotation」が有効になっていることを確認します
3.1.8	PASS	ログファイルの最大存続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	最大ログファイルサイズが正しく設定されていることを確認します
3.1.10	PASS	正しい syslog 機能が選択されていることを確認します
3.1.11	PASS	PostgreSQL syslog メッセージのプログラム名が正しいことを確認します
3.1.14	PASS	「debug_print_parse」が無効になっていることを確認します
3.1.15	PASS	「debug_print_rewriting」が無効になっていることを確認します
3.1.16	PASS	「debug_print_plan」が無効になっていることを確認します
3.1.17	PASS	「debug_pretty_print」が有効になっていることを確認します
3.1.18	PASS	「log_connections」が有効になっていることを確認します
3.1.19	PASS	「log_disconnections」が有効になっていることを確認します
3.1.21	PASS	'log_hostname' が正しく設定されていることを確認します
3.1.23	PASS	「log_statement」が正しく設定されていることを確認します
3.1.24	PASS	「log_timezone」が正しく設定されていることを確認します
3.2	PASS	PostgreSQL Audit Extension（pgAudit）が有効になっていることを確認します
4.1	PASS	sudo が正しく構成されていることを確認します
4.2	PASS	過剰な管理権限が取り消されていることを確認する
4.3	PASS	過剰な機能特権が取り消されていることを確認する
4.4	PASS	テスト済み 過剰な DML 権限が取り消されていることを確認する
5.2	Not Tested	「host」TCP / IP ソケットを介したログインが正しく構成されていることを確認します
6.2	PASS	「バックエンド」ランタイムパラメーターが正しく構成されていることを確認します

次のページに続く

表 2 – 前のページからの続き

	Result	Description
6.7	Not Tested	FIPS 140-2 OpenSSL 暗号化が使用されていることを確認する
6.8	PASS	SSL が有効になっており、正しく構成されていることを確認します
7.3	PASS	WAL アーカイブが構成され、機能していることを確認します

テスト 5.2 は、手動で監査すると PASS する可能性がありますが、自動テストはありません。

テスト 6.7 は、CentOS を使用したデフォルトの展開で成功しますが、Debian バリアントには追加のパッケージが必要です。

8.6 Sequences

多くのアプリケーションでは、一意のサロゲート ID をデータベースエントリに割り当てる必要があります。多くの場合、データベース `SEQUENCE` オブジェクトを使用してこれらを生成します。PostgreSQL では、これらは次のいずれかです。

- `CREATE SEQUENCE` コマンドを使用して手動で作成され、`nextval()` 関数を呼び出すことによって取得されたシーケンス
- `serial` および `bigserial` 列、または `GENERATED BY DEFAULT AS IDENTITY` 列

ただし、PostgreSQL の標準シーケンスはマルチノードを認識せず、ローカルノードでのみ一意の値を生成します。このようなシーケンスによって生成された一意の ID は、マルチマスターレプリケーションで破棄された `INSERT` アクションにより競合とデータ損失を引き起こすため、これは重要です。

8.6.1 PGD グローバルシーケンス

このため、PGD は、**グローバルシーケンス**と呼ばれる PGD グループ全体で 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。

PGD 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、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。ほとんどの場合に機能しますが、必ずしもすべての場合とは限りません。

PGD グローバルシーケンスを使用すると、挿入の競合に関する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、ノードは他のノードと同じ値を取得できません。PGD 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。

PGD 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考えられます。

- SnowflakeId シーケンス

- グローバルに割り当てられた範囲シーケンス

SnowflakeId シーケンスは、任意の時点でノード間通信を必要としないアルゴリズムを使用して値を生成します。より高速で堅牢で、値が作成されたときの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。

SnowflakeId シーケンスには、64 ビット BIGINT データ型でのみ機能し、最大 19 桁の長さの値を生成するという制限があります。これは、Javascript Integer 型などの一部のホスト言語データ型で使用するには長すぎる場合があります。グローバルに割り当てられたシーケンスは、ノード間のコンセンサスによって必要に応じて補充できる値のローカル範囲を割り当て、BIGINT または INTEGER シーケンスに適したものにします。

bdr.alter_sequence_set_kind() を使用してグローバルシーケンスを作成できます

機能。この関数は、標準の PostgreSQL シーケンスを取り、それを PGD グローバルシーケンスとしてマークします。また、シーケンスを標準の PostgreSQL シーケンスに変換して戻すこともできます。

PGD は、構成変数 *bdr.default_sequence_kind* も提供します。これは、 `CREATE SEQUENCE` コマンドの実行時、または `serial` 、`bigserial` 、または `GENERATED BY DEFAULT AS IDENTITY` 列の作成時に作成す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` 、新しく作成されたシーケンスが標準の PostgreSQL（ローカル）シーケンスであることを意味します。
- `galloc` 、常にグローバルに割り当てられた範囲シーケンスを作成します。
- `snowflakeid` 、時間、ノード ID、およびカウンターコンポーネントで構成される BIGINT シーケンスのグローバルシーケンスを作成します。INTEGER シーケンスでは使用できません（したがって、`bigserial` には使用できますが、 `serial` には使用できません）。
- `timeshard` 、これは SnowflakeId シーケンスの古いバージョンであり、下位互換性のためにのみ提供されています。SnowflakeId が優先されます。
- `distributed`（デフォルト）、:ref:` <reference/pgdpostgres-settings/#global-sequence-parameters>` にのみ使用できる特別な値。 `int8` シーケンス（つまり `bigserial` ）には `snowflakeid` を選択し、 `int4`（つまり `serial` ）および `int2` シーケンスには `galloc` シーケンスを選択します。

bdr. **シーケンス** ビューは、個々のシーケンスの種類に関する情報を表示します。

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

SnowflakeId シーケンス

SnowflakeId シーケンスによって生成された ID は大まかな時間順序付けであるため、標準の PostgreSQL シーケンスのように、それらを使用してデータ挿入のおおよその順序を取得できます。同じミリ秒内に生成された値は、1 つのノードでも、順不同の場合があります。緩い時間順序のプロパティは、レンジパーティションキーとしての使用に適していることを意味します。

SnowflakeId シーケンスは 1 つ以上のノードで機能し、ノード参加プロセスが完了した後にノード間通信を必要としません。そのため、ネットワークパーティションが拡張されるリスクがあっても、引き続き使用できます。レプリケーションラグやノード間のレイテンシーの影響を受けません。

SnowflakeId シーケンスは、標準シーケンスとは異なる方法で一意の ID を生成します。このアルゴリズムでは、シーケンス番号に 3 つのコンポーネントを使用します。シーケンスの最初のコンポーネントは、シー

ケンス番号生成時のタイムスタンプです。シーケンス番号の 2 番目のコンポーネントは、各 PGD ノードに割り当てられた一意の ID です。これにより、さまざまなノードの ID が常に異なることが保証されます。3 番目のコンポーネントは、ローカルシーケンスによって生成された番号です。

シーケンス番号に一意のノード ID を追加するだけで競合がないことを保証できますが、シーケンスの別の便利なプロパティも維持したいでしょう。シーケンス番号の順序は、データがテーブルに挿入された順序にほぼ対応しています。タイムスタンプを最初に置くと、これが保証されます。

SnowflakeId シーケンスには、いくつかの制限と警告が適用されます。

SnowflakeId シーケンスは 64 ビット幅で、 `bigint` または `bigserial` が必要です。生成される値は最大 19 桁です。実用的な 32 ビット `integer` バージョンはないため、 `serial` シーケンスでは使用できません。代わりに、グローバルに割り当てられた範囲シーケンスを使用します。

SnowflakeId の場合、特定のノードでミリ秒ごとに生成されるシーケンス値は 4096 個（1 秒あたり約 400 万シーケンス値）の制限があります。シーケンス値の生成が指定されたミリ秒以内にラップアラウンドする場合、SnowflakeId シーケンスは次のミリ秒まで待機し、そのミリ秒の新しい値を取得します。

SnowflakeId シーケンスはタイムスタンプをシーケンス値にエンコードするため、指定されたタイムフレーム内でのみ（システムクロックに応じて）新しいシーケンス値を生成できます。使用できる最も古いタイムスタンプは 2016-10-07 で、これは SnowflakeId のエポックタイムです。値は 2086 年に負の値にラップし、2156 までに数が完全に不足します。

タイムスタンプは SnowflakeId シーケンスの重要な部分であるため、Postgres プロセスの有効期間中に使用された最新のものよりも古いタイムスタンプを持つシーケンスを生成しないようにする追加の保護があります（ただし、Postgres の再起動間ではありません）。

SnowflakeId シーケンスの入力として使用されるシーケンスの `INCREMENT` オプションは、効果的に無視されます。これは、多くのオブジェクトリレーショナルマッパー（ORM）ツール、特に Hibernate のように、シーケンス ID キャッシングを行うアプリケーションに関連する場合があります。シーケンスは時間ベースであるため、アプリケーションがキャッシュされた値を使用して何かを実行できるようになるまでに、シーケンスは新しい非衝突値に進むため、これは実際的な効果はほとんどありません。

同様に、基になるシーケンスの `START` 、 `MINVALUE` 、 `MAXVALUE` 、および `CACHE` 設定を変更する場合がありますが、変更してもメリットはありません。シーケンスの下位 14 ビットが使用され、残りは破棄されるため、値の範囲の制限は関数の結果に影響しません。同じ理由で、 `setval()` は SnowflakeId シーケンスには役立ちません。

Timeshard シーケンス

Timeshard シーケンスは、既存のインストールとの下位互換性のために提供されていますが、新しいアプリケーションでの使用はお勧めしません。代わりに SnowflakeId シーケンスを使用することをお勧めします。

Timeshard は SnowflakeId と非常に似ていますが、制限が異なり、保護が少なく、パフォーマンスが低下します。

timeshard と SnowflakeId の違いは次のとおりです。

- Timeshard は、SnowflakeId よりも多い、ミリ秒あたり最大 16384 個（1 秒あたり約 1600 万個）を生成できます。ただし、指定されたミリ秒内のラップアラウンドに対する保護はありません。タイムハードシーケンスを使用するスキーマは、特定の列にタイムハード値を使用する場合、 `UNIQUE` 制約の使

用を保護する必要があります。-timeshard シーケンスのタイムスタンプコンポーネントは、2050 年に値を使い果たし、bigint と組み合わせて使用すると、値は 2033 年に負の数にラップします。これは、2033 年以降に生成されたシーケンスが負の値を持つことを意味します。これは SnowflakeId よりもかなり短い時間スパンであり、これが SnowflakeId が好ましい主な理由です。- タイムシャードシーケンスでは、時折ディスク書き込みが必要です（標準のローカルシーケンスと同様）。SnowflakeId はメモリ内で計算されるため、SnowflakeId シーケンスは一般的に timehard シーケンスよりも少し高速です。

グローバルに割り当てられた範囲シーケンス

グローバルに割り当てられた範囲（または `galloc` ）シーケンスは、各ノードに値の範囲（チャンク）を割り当てます。ローカル範囲が使い果たされると、新しい範囲が他のノード間のコンセンサスによってグローバルに割り当てられます。これはキースペースを効率的に使用しますが、現在割り当てられているローカル範囲が使い果たされたときにシーケンスジェネレーターが進行するには、ローカルノードをクラスター内の大部分のノードに接続する必要があります。

SnowflakeId シーケンスとは異なり、 `galloc` シーケンスは、PostgreSQL が提供するすべてのシーケンスデータ型：`smallint` 、`integer` 、および `bigint` をサポートします。これは、64 ビットシーケンスに問題がある環境で `galloc` シーケンスを使用できることを意味します。例には、53 ビット値のみをサポートするため、またはシーケンスが限られたスペースで出力に表示されるため、JavaScript で整数を使用することが含まれます。

各投票によって割り当てられる範囲は、シーケンスが使用しているデータ型に基づいて現在事前に決定されています。

- smallint - smallint 1 000 の数字
- 整数 - 1 000 000 番号
- bigint — 1 000 000 000 番号

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、もう 1 つは将来の使用のために予約されたチャンクのため、1 つのノードから生成される値は単調に増加します。ただし、グローバルに見ると、生成される値はまったく順序付けられていません。これは、b ツリーインデックスへの影響によりパフォーマンスの低下を引き起こす可能性があり、通常は、生成された値が範囲パーティションキーとして役に立たないことを意味します。

`galloc` シーケンスの主な欠点は、割り当てられた範囲が使い果たされると、シーケンスジェネレーターは、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求める必要があることです。PGD グループの大部分にアクセスできない場合、これにより、遅延または運用上の問題が発生する可能性があります。（これは、後のリリースでは回避される可能性があります。）

`CACHE` 、`START` 、`MINVALUE` 、および `MAXVALUE` オプションは、 `galloc` シーケンスで正しく機能します。ただし、シーケンスを `galloc` 種類に変換する前に設定する必要があります。 `INCREMENT BY` オプションも正しく動作します。ただし、シーケンスデータ型ごとに割り当てられた上記の範囲以上のインクリメント値を割り当てることはできません。`setval()` は `galloc` シーケンスのグローバル状態をリセットしません。使用しないでください。

`galloc` シーケンスには、いくつかの制限が適用されます。PGD は、特別な PGD カタログ *bdr.sequence_alloc* で `galloc` シーケンスを追跡します。このカタログは、 `galloc` シーケンスに現在割り当てられているチャ

ンクを追跡するために必要です。シーケンス名と名前空間は、このカタログに格納されます。シーケンスチャンクの割り当ては Raft によって管理されますが、シーケンス名前/名前空間の変更はレプリケーションストリームによって管理されるため、PGD は現在、galloc シーケンスの名前変更、別の名前空間への移動、または galloc シーケンスを含む名前空間の名前変更をサポートしていません。アプリケーションスキーマを設計する際は、この制限に注意してください。

ローカルシーケンスから galloc シーケンスへの変換

ローカルシーケンスを galloc に変換する前に、いくつかの前提条件を満たす必要があります。

1. シーケンスと列のデータ型が一致することを確認します

シーケンスのデータ型が、使用される列のデータ型と一致することを確認します。たとえば、bigint シーケンスを作成し、integer 列のデフォルトをそのシーケンスによって返される nextval() に割り当てることができます。 bigint に 1 000 000 000 のブロックで割り当てられる galloc シーケンスを使用すると、3 つ以上のノードが使用されている場合、すぐに nextval() によって返される値が int4 の範囲を超えます。

次の例は、何が起こるかを示しています。

```
CREATE SEQUENCE int8_seq;

SELECT sequencename, data_type FROM pg_sequences;
 sequencename | data_type
- -------------+-----------
 int8_seq     | bigint
(1 row)

CREATE TABLE seqtest (id INT NOT NULL PRIMARY KEY);

ALTER SEQUENCE int8_seq OWNED BY seqtest.id;

SELECT bdr.alter_sequence_set_kind(public.int8_seq::regclass, galloc, 1);
 alter_sequence_set_kind
- ------------------------

(1 row)

ALTER TABLE seqtest ALTER COLUMN id SET DEFAULT nextval(int8_seq::regclass);
```

2 つのノードで INSERT INTO seqtest VALUES(DEFAULT) を実行した後、テーブルには次の値が含まれます。

```
SELECT * FROM seqtest;
     id
- -----------
```

(次のページに続く)

(前のページからの続き)

```
          2
 2000000002
(2 rows)
```

ただし、シーケンスが値 4000000002 を生成したため、3 番目のノードで同じ操作を試みると、integer out of range エラーで失敗します。

Tip: PostgreSQL からシーケンスの現在のデータ型を取得できます *pg_sequences*

ビュー。現在の値が新しいデータ型の最大値を超えない限り、ALTER SEQUENCE ... AS ...、たとえば ALTER SEQUENCE public.sequence AS integer を使用してシーケンスのデータ型を変更できます。

2. シーケンスの新しい開始値を設定します

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開始値からリスタートします。これが本番データベース内の既存のシーケンスで発生した場合は、現在の値を照会してから、開始値を適切に設定する必要があります。このユースケースを支援するために、PGD では、ユーザーが関数

> *bdr.alter_sequence_set_kind()*

を使用して開始値を渡すことができます。すでにオフセットを使用していて、複数のノードからの書き込みがある場合は、使用された最大の値を確認し、少なくとも次の値までシーケンスをリスタートする必要があります。

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

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

ユーザーはシーケンスをロックできないため、 max() 値が照会されている間、操作を続行できるように$MARGIN 値を残す必要があります。

> *bdr.sequence_alloc* テーブルは、チャンクサイズとクラスター全体に割り当てられた範囲に関する情報を提供します。

この例では、シーケンスは 333 で始まり、クラスターには 2 つのノードがあります。割り当て数は 4 つ、1 ノードあたり 2 つ、チャンクサイズは 1000000 整数列関連です。

```
SELECT * FROM bdr.sequence_alloc
    WHERE seqid = public.categories_category_seq::regclass;
          seqid          | seq_chunk_size | seq_allocated_up_to | seq_nallocs |      ␣
↪seq_last_alloc
- ------------------------+----------------+---------------------+-------------+------
↪-----------------------
 categories_category_seq |        1000000 |             4000333 |           4 | 2020-
↪05-21 20:02:15.957835+00
(1 row)
```

各ノードの特定のシーケンスに現在割り当てられている範囲を表示するには、次のクエリを使用します。

- ノード Node1 は 333 から 2000333 の範囲を使用しています。

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,2); -- first range
 range_start | range_end
- ------------+-----------
         334 |   1000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,3); -- second range
 range_start | range_end
- ------------+-----------
     1000334 |   2000333
(1 row)
```

- ノード Node2 は 2000004 から 4000003 の範囲を使用しています。

```
SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,2); -- first range
 range_start | range_end
- ------------+-----------
     2000334 |   3000333
(1 row)

SELECT last_value AS range_start, log_cnt AS range_end
    FROM categories_category_seq WHERE ctid = (0,3); -- second range
 range_start | range_end
- ------------+-----------
     3000334 |   4000333
```

注釈: 最初の範囲のみが表示されるため、単一のクエリ（*WHERE ctid IN ('(0,2)', '(0,3)')* など）に結合することはできません。

ノードがチャンクを終了すると、新しいチャンクのコンセンサスを求め、最初に利用可能なチャンクを取得します。この例では、4000334 から 5000333 です。これは新しい予約済みチャンクであり、古い予約済みチャンクの消費を開始します。

8.6.2 UUID、KSUUID、およびその他のアプローチ

PGD で使用できるグローバルシーケンスを使用せずに、他の方法でグローバルに一意の ID を生成できます。例：

- UUID とその PGD バリアント、KSUUID
- ノードごとに異なるオフセットを持つローカルシーケンス（つまり、手動）
- 外部調整されたナチュラルキー

PGD アプリケーションは他の方法を安全に使用できません。シーケンス生成に `SELECT ... FOR UPDATE`、`UPDATE ... RETURNING ...` などに依存するカウンターテーブルベースのアプローチは、PGD で正しく機能しません。複数のノードで同じ値が生成されます。同じ理由で、「ギャップレス」シーケンス生成の通常の戦略は PGD では機能しません。ほとんどの場合、アプリケーションは、2 フェーズコミットを使用して、外部ソースからギャップレスでなければならないシーケンスの生成を調整します。または、PGD グループ内の 1 つのノードでのみ生成します。

UUID

`UUID` キーは、代わりにシーケンスを完全に回避し、128 ビットのユニバーサル一意識別子を使用します。これらはランダムまたは擬似ランダム値であり、非常に大きいため、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、ノードが継続的な通信を行う必要はありません。

万が一、衝突が発生した場合、競合検出は、挿入された 2 つのレコードのうち新しい方を選択して保持します。競合ログが有効になっている場合、そのようなイベントが記録されます。ただし、衝突は約 2 ^ 64 個のキーが生成された後にのみ実際に発生する可能性があるため、発生する可能性は非常に低いです。

`UUID` キーの主な欠点は、スペースとネットワークの点でやや非効率的であることです。主キーとしてだけでなく、外部キーで参照される場所やネットワークで送信される場合にも、より多くのスペースを消費します。また、すべてのアプリケーションが `UUID` キーにうまく対応するとは限りません。

KSUUID

PGD は、`UUID` データの K-Sortable バリアントを操作するための関数を提供します。KSUUID は、PostgreSQL 標準の `UUID` データ型を使用して保存できる値を生成します。`KSUUID` 値は、`UUID` 標準に従って、タイムスタンプとランダムデータの両方を格納するという点で `UUIDv1` に似ています。違いは、`KSUUID` は K-Sortable、つまりタイムスタンプで弱くソートできることです。これにより、よりコンパクトな `btree` インデックスが生成され、検索の効率が向上し、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ーの MAC が含まれないため、使用してもセキュリティ上の懸念はありません。

現在、すべての場合に KSUUID v2 をお勧めします。通常の比較演算子で生成された値を直接ソートできます。

PGD には KSUUID の 2 つのバージョンがあります。v1 と v2 です。レガシー KSUUID v1 は非推奨ですが、既存のインストールをサポートするために保持されます。新規インストールには使用しないでください。v1 と v2 の内部コンテンツには互換性がありません。そのため、それらを操作する機能も互換性がありません。KSUUID の v2 は、 UUID バージョン番号も保存しなくなりました。

PGD リファレンスの *KSUUID v2* **関数** および *KSUUID v1* **関数** を参照してください。

ステップとオフセットのシーケンス

オフセットステップシーケンスでは、各ノードで通常の PostgreSQL シーケンスが使用されます。各シーケンスは同じ量でインクリメントし、異なるオフセットで始まります。たとえば、ステップ 1000 では、node1 のシーケンスは 1001、2001、3001 などを生成します。node2 のシーケンスは 1002、2002、3002 などを生成します。このスキームは、ノードが長期間通信できない場合でもうまく機能します。ただし、デザイナーはスキーマを確立するときにノードの最大数を指定する必要があり、ノードごとの構成が必要です。間違いは、シーケンスの重複に簡単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、PGD でこのアプローチを構成するのは比較的簡単です。

```
CREATE TABLE some_table (
    generated_value bigint primary key
);

CREATE SEQUENCE some_seq INCREMENT 1000 OWNED BY some_table.generated_value;

ALTER TABLE some_table ALTER COLUMN generated_value SET DEFAULT nextval(some_seq);
```

次に、 setval() を呼び出す各ノードで、各ノードに異なるオフセット開始値を指定します。例：

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

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

 -- ... etc
```

将来的に変更することは困難で混乱を招くため、追加する可能性のあるすべてのノード用のスペースを確保できるように、十分な大きさの INCREMENT を必ず許可してください。

bigint 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質的な懸念はありません。何百もの機械が毎秒何百万ものインサートを処理し、枯渇に近づくチャンスを得るには、何百年もかかるでしょう。

PGD は現在、このようなステップ/オフセットシーケンスでノードごとのオフセットを構成するため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アントは、`PRIMARY KEY (node_number, generated_value)` で構成される複合キーを使用することです。ノード番号は通常、各ノードで異なる番号を返す関数から取得されます。DDL レプリケーションを一時的に無効にし、定数 SQL ファンクションを作成することにより、このようなファンクションを作成できます。または、レプリケーションセットの一部ではない 1 行のテーブルを使用して、各ノードに異なる値を格納できます。

8.6.3 こちらもご覧ください

- *Global sequence management interfaces*
- *KSUUID v2* 関数
- *KSUUID v1* 関数

8.7 Durability and performance options

8.7.1 概要

EDB Postgres Distributed を使用すると、 **コミットスコープ**を使用して、耐久性、一貫性、可用性、およびパフォーマンスのニーズに基づいて、いくつかのレプリケーション構成から選択できます。

基本的な構成では、 EDB Postgres Distributed は非同期レプリケーションを使用します。ただし、コミットスコープは、デフォルトとトランザクションごとの動作の両方を変更できます。組み込みの物理または論理レプリケーションと同じ方法で、標準の `synchronous_standby_names` を使用して、レガシー Postgres 同期レプリケーションを構成することもできます。ただし、コミットスコープを使用すると、レプリケーション動作をはるかに柔軟に制御できます。

さまざまな同期設定は、関連するアプリケーションに関連する 3 つのプロパティに影響しますが、すべて個別に実装できます。

- 耐久性：複数のノードに書き込むと、クラッシュ回復力が向上し、クラッシュして再起動した後にデータを回復できます。
- 可視性：クライアントへのコミット確認により、データベースはいくつかのノードセットでコミットされたトランザクションの即時の可視性を保証します。
- 競合処理：競合はコミット後に楽観的に処理でき、コミットタイムスタンプに基づいてトランザクションがレプリケートされるときに競合が解決されます。または、プリコミットで悲観的に処理できます。クライアントは、トランザクションが最終的にすべてのノードに適用されることを信頼できます。これ以上競合することはありません。アボートして、クライアントにエラーを直接通知できます。

コミットスコープを使用すると、トランザクションの持続性を制御する 2 つの方法を使用できます。

- *Group Commit* 。このオプションは、トランザクションがコミット可能と見なされる前にコンセンサスに達する必要があるノードとノード数、およびレプリケーションのどの段階でコミットされたと見なされるかを制御します。このオプションを使用すると、トランザクションの可視性の順序を制御することもできます。
- *CAMO/failover* オプションを使用した *pgbench* 。このオプションは、クライアントがコンセンサスの一部である Group Commit のバリアントです。
- *Lag Control* 。このオプションは、コミットの続行を許可する前に、レプリケーションに関してノードからどれだけ遅れることができるかを制御します。

Postgres は *physical streaming replication* を提供します

（PSR）単方向ですが、 *synchronous variant* を提供します。下位互換性のため、PGD は引き続き `synchronous_commit` および `synchronous_standby_names` を使用した同期レプリケーションの構成をサポートしています。

> *PGD* を使用したレガシー同期レプリケーション を参照してくださいが、代わりに *Group Commit*

の使用を検討してください。

8.7.2 用語と定義

PGD ノードは、トランザクションのレプリケーション中にさまざまな役割を果たします。これらはトランザクションごとに暗黙的に割り当てられ、同時トランザクションでも関係ありません。

** オリジン*は、クライアントまたはアプリケーションからトランザクションを受信するノードです。最初にトランザクションを処理するのはノードであり、他の PGD ノードへのレプリケーションを開始し、確認またはエラーでクライアントに応答します。

** パートナー*ノードは、グループコミット要件に従ってトランザクションを確認することが期待される PGD ノードです。

** コミットグループ*は、コミットに関与するすべての PGD ノード、つまり、オリジンとそのすべてのパートナーノードのグループであり、少数またはすべてのピアノードになります。

8.7.3 比較

PGD で利用可能な同期レプリケーションのほとんどのオプションは、さまざまなレベルの同期を可能にし、パフォーマンスと、ノードまたはネットワークの停止に対する保護との間のさまざまなトレードオフを提供します。

次の表は、トランザクションが発行されたオリジンノードから COMMIT 確認を受信した後、クライアントがレプリケートされたピアノードに期待できることをまとめたものです。Mode 列は、バリアントに応じて異なる意味をとります。PSR と PGD を使用したレガシー同期レプリケーションの場合、`synchronous_commit` 設定を指します。コミットスコープの場合、

> *commit scope configuration* の確認要件を指します。

Variant	Mode	Received	Visible	Durable
PSR Async	off (default)	no	no	no
PGD Async	off (default)	no	no	no
PGD Lag Control	'ON received' nodes	no	no	no
PGD Lag Control	'ON replicated' nodes	no	no	no
PGD Lag Control	'ON durable' nodes	no	no	no
PGD Lag Control	'ON visible' nodes	no	no	no
PSR Sync	remote_write (2)	yes	no	no (1)
PSR Sync	on (2)	yes	no	yes
PSR Sync	remote_apply (2)	yes	yes	yes
PGD Group Commit	'ON received' nodes	yes	no	no
PGD Group Commit	'ON replicated' nodes	yes	no	no
PGD Group Commit	'ON durable' nodes	yes	no	yes
PGD Group Commit	'ON visible' nodes	yes	yes	yes
PGD CAMO	'ON received' nodes	yes	no	no
PGD CAMO	'ON replicated' nodes	yes	no	no
PGD CAMO	'ON durable' nodes	yes	no	yes
PGD CAMO	'ON visible' nodes	yes	yes	yes
PGD Legacy Sync (3)	remote_write (2)	yes	no	no
PGD Legacy Sync (3)	on (2)	yes	yes	yes
PGD Legacy Sync (3)	remote_apply (2)	yes	yes	yes

- （1）OS に書き込まれ、OS が実行されたままで Postgres のみがクラッシュする場合。
- （2）`synchronous_replication_availability` を `async'` に設定してローカルモード（許可されている場合）に切り替えない限り、それ以外の場合は非同期 PGD のデフォルトの値が適用されます。

（*3*）代わりに *GroupCommit* の使用を検討してください。

受信により、正常に動作しているピアが、ネットワーク全体または部分的な停止に直面した場合でも、それ以上の通信を必要とせずに最終的にトランザクションを適用できることが保証されます。この確認には永続ストレージが関係しないため、ピアノードのクラッシュには引き続きトランザクションの再送信が必要な場合があります。同期と見なされるすべてのモードがこの保護を提供します。

可視性は、トランザクションがリモートで適用されたことを意味します。他のすべてのクライアントは、すべてのノードでトランザクションの結果を確認し、コミットがオリジンノードによって確認された直後にこの保証を提供します。可視性がなければ、接続されている他のクライアントはトランザクションの結果を表示できず、古い読み取りが発生する可能性があります。

耐久性はピアノードのストレージに関連し、ピアノードのクラッシュと回復後のデータの損失に対する保護を提供します。これは、データの受信（物理ストリーミングレプリケーションなど）または可視性（Group Commit など）に関連する可能性があります。前者はクラッシュ後の再送信の必要性を排除し、後者は再起動後も可視性が維持されることを保証します。

8.7.4 操作の内部タイミング

さまざまなモードがどのように機能するかをよりよく理解するには、PSR と PGD がトランザクションを異なる方法で適用することを理解すると役立ちます。

物理ストリーミングレプリケーション、操作の順序は次のとおりです。

- Origin はコミットレコードを WAL にフラッシュし、トランザクションをローカルで見えるようにします。
- ピアノードは変更を受け取り、書き込みを発行します。
- ピアは、受け取った変更をディスクにフラッシュします。
- ピアは変更を適用し、トランザクションをローカルで見えるようにします。

PGD では、操作の順序が異なります。

- Origin はコミットレコードを WAL にフラッシュし、トランザクションをローカルで見えるようにします。
- ピアノードは、メモリ内の適用キューへの変更を受け取ります。
- ピアは変更を適用し、トランザクションをローカルで見えるようにします。
- ピアは、ディスクにフラッシュすることによりトランザクションを永続化します。

Group Commit、CAMO、および Eager の場合、オリジンノードはトランザクションをローカルで可視化する前に特定の数の確認を待機します。操作の順序は次のとおりです。

- Origin は、準備またはプリコミットレコードを WAL にフラッシュします。
- ピアノードは、メモリ内の適用キューへの変更を受け取ります。
- ピアは変更を適用し、トランザクションをローカルで見えるようにします。
- ピアは、ディスクにフラッシュすることによりトランザクションを永続化します。
- Origin はコミットしてトランザクションをローカルで可視にします。

次の表は、違いをまとめたものです。

Variant	Order of apply vs persist	Replication before or after commit
PSR	persist first	after WAL flush of commit record
PGD Async	apply first	after WAL flush of commit record
PGD Lag Control	apply first	after WAL flush of commit record
PGD Group Commit	apply first	before COMMIT on origin
PGD CAMO	apply first	before COMMIT on origin

8.7.5 構成

PGD での他の管理操作と同様に、SQL 関数を使用してコミットスコープを構成します。

たとえば、 example_group PGD グループの大部分のノードでグループコミットを行う基本的なコミットスコープを定義できます。

```
SELECT bdr.add_commit_scope(
  commit_scope_name := example_scope,
  origin_node_group := example_group,
  rule := ANY MAJORITY (example_group) GROUP COMMIT,
  wait_for_ready := true
);
```

次に、トランザクションごとにまたはグローバルに構成変数（GUC）bdr.commit_scope をそのコミットスコープに設定することにより、コミットスコープを使用できます。

```
BEGIN;
SET LOCAL bdr.commit_scope = example_scope;
...
COMMIT;
```

特定の PGD グループのデフォルトのコミットスコープを設定することもできます。

```
SELECT bdr.alter_node_group_option(
  node_group_name := example_group,
  config_key := default_commit_scope,
  config_value := example_scope
);
```

default_commit_scope は、指定された起点ノードが属するグループツリーで下から上にチェックされます。default_commit_scope は特別な値 local に設定できません。これは、コミットスコープが bdr.commit_scope 構成パラメーターを使用する方法がないことを意味します。

説明されているすべてのオプションを含むコミットスコープ言語の完全な詳細については、
commit **スコープへの移行** を参照してください。

Postgres 構成パラメーター

次の表に、デフォルト以外の値（req）に設定する必要がある構成設定と、オプション（opt）であるが特定のバリアントに影響を与える構成設定の概要を示します。

Setting (GUC)	Group Commit	Lag Control	PSR (1)	Legacy Sync
synchronous_standby_names	n/a	n/a	req	req
synchronous_commit	n/a	n/a	opt	opt

次のページに続く

表 5 – 前のページからの続き

Setting (GUC)	Group Commit	Lag Control	PSR (1)	Legacy Sync
synchronous_replication_availability	n/a	n/a	opt	opt
bdr.commit_scope	opt	opt	n/a	n/a

8.7.6 計画的なシャットダウンと再起動

受信確認で Group Commit を使用する場合は、計画的なシャットダウンまたは再起動に注意してください。デフォルトでは、適用キューはシャットダウン前に消費されます。ただし、`immediate` シャットダウンモードでは、シャットダウン時にキューが破棄され、停止したノードがキュー内のトランザクションを「忘れ」ます。起点ノードの同時障害は、両方のノードに障害が発生したかのように、データの損失につながる可能性があります。

適用キューがディスクにフラッシュされるようにするには、メンテナンスタスクに `smart` または `fast` shutdown を使用します。このアプローチにより、必要な同期レベルが維持され、データの損失が防止されます。

8.7.7 PGD を使用したレガシー同期レプリケーション

注釈: 代わりに *Group Commit* の使用を検討してください。

使用法

PGD を使用して同期レプリケーションを有効にするには、関連する PGD ピアノードのアプリケーション名を `synchronous_standby_names` に追加する必要があります。 `FIRST x` または `ANY x` の使用は、非 PGD スタンバイノードの要件と競合しない場合、ある程度の柔軟性を提供します。

追加したら、 `synchronous_commit` を使用して、トランザクションごとの同期レベルを構成できます。デフォルトは on です。この設定は、アプリケーション名を `synchronous_standby_names` に追加すると既に同期レプリケーションが有効になっていることを意味します。`synchronous_commit` を `local` または `off` に設定すると、同期レプリケーションがオフになります。

PGD はトランザクションを永続化する前に適用するため、値 on および `remote_apply` 論理レプリケーションと同等です。

commit スコープへの移行

synchronous_commit および synchronous_standby_names とは無関係に、PGD のグループコミット機能を構成します。代わりに、bdr.commit_scope GUC を使用すると、トランザクションごとにスコープを選択できます。また、各ノードで synchronous_standby_names を個別に構成する代わりに、Group Commit はグローバルに同期されたコミットスコープを使用します。

注釈: *synchronous_standby_names* と commit スコープの文法は似ていますが、前者は起点ノードを説明しませんが、後者は説明します。したがって、たとえば、 *synchronous_standby_names = 'ANY 1 (..)'* は `ANY 2 (...)` のコミットスコープと同等です。この選択により、多数決についての推論が容易になり、オリジンノードがトランザクションの耐久性と可視性にも貢献することを反映しています。

Commit scopes

コミットスコープにより、アプリケーションは EDB Postgres Distributed の耐久性と一貫性をきめ細かく制御できます。

コミットスコープは、 COMMIT レプリケーションの動作を説明する名前付きルールです。実際の動作は、コミットスコープが

Group Commit 、 *Commit At Most Once* 、 *Lag Control*

、またはこれらの組み合わせを使用するかどうかによって異なります。

構成

Group Commit を使用するには、最初にコミットスコープを定義します。これにより、トランザクションのコミットに関与する PGD ノードが決定されます。スコープを確立したら、次のように Group Commit を使用するようにトランザクションを構成できます。

```
BEGIN;
SET LOCAL bdr.commit_scope = example_scope;
...
COMMIT;
```

トランザクションがデータを書き込む前に、コミットスコープを設定する必要があります。

この例では、以前にコミットスコープを次のように定義している可能性があります。

```
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := example_bdr_group,
    rule := ANY 2 (example_bdr_group) GROUP COMMIT,
    wait_for_ready := true
);
```

この例では、 example_bdr_group という名前付けの **ノードグループ**が存在し、少なくとも 2 つの PGD ノードをメンバーとして直接またはサブグループに含めることを前提としています。example_scope でコミットされたトランザクションでは、グループ内の PGD ノードからの追加の確認が 1 つ必要です。起点ノードとともに、これはグループ外の「ANY 2」ノードを占め、トランザクションはコミット後に可視で耐久性があることが保証されます。

オリジングループ

コミットスコープのルールは、トランザクションがコミットされるノード、つまりトランザクションの起点として機能するノードによって異なる場合があります。これをアプリケーションに対して透過的にするために、PGD では、コミットスコープでトランザクションの発生元に応じて異なるルールを定義できます。

たとえば、ノードが左側と右側の 2 つのデータセンターにまたがる EDB Postgres 分散クラスターを検討します。トップレベルの PGD ノードグループの名前が top_group であると仮定します。次のコマンドを使用して、サブグループを設定し、ローカルデータセンター内のすべてのノードがトランザクションを確認する必要があるコミットスコープを作成しますが、リモートからは 1 つのノードのみを確認できます。

```
- - create sub-groups
SELECT bdr.create_node_group(
    node_group_name := left_dc,
    parent_group_name := top_group,
    join_node_group := false
);
SELECT bdr.create_node_group(
    node_group_name := right_dc,
    parent_group_name := top_group,
    join_node_group := false
);

- - create a commit scope with individual rules
- - for each sub-group
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) GROUP COMMIT AND ANY 1 (right_dc) GROUP COMMIT,
    wait_for_ready := true
);
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) GROUP COMMIT AND ALL (right_dc) GROUP COMMIT,
    wait_for_ready := true
);
```

現在、 left_dc の一部であるノードで example_scope を使用すると、最初のスコープが使用されます。right_dc の一部であるノードで同じスコープを使用すると、2 番目のスコープが使用されます。これは、アプリケーションでスコープ名をいじることなく、逆スコープを作成する効果的な方法です。

各グループは、 bdr.alter_node_group_option 管理インターフェイスを使用して指定されたデフォルトのコミットスコープを持つこともできます。

上記のスコープをこれらのグループのノードで発生するすべてのトランザクションのデフォルトのスコープにすることは次のとおりです。

```
SELECT bdr.alter_node_group_option(
  node_group_name := left_dc,
  config_key := default_commit_scope,
  config_value := example_scope
);
SELECT bdr.alter_node_group_option(
  node_group_name := right_dc,
  config_key := default_commit_scope,
  config_value := example_scope
);
```

確認レベル

PGD ノードは、トランザクションの確認をさまざまなタイミングで送信できます。保護レベルを上げる際に、確認ノードの観点から、これらは次のとおりです。

- received —リモート PGD ノードは、ローカルアプリケーションを開始する前に、トランザクションを受信した直後に確認します。
- replicated —トランザクションの変更を適用した後、ディスクにフラッシュする前に確認します。
- durable —すべての変更がディスクにフラッシュされた後、トランザクションを確認します。
- visible （デフォルト）& mdash; すべての変更がディスクにフラッシュされ、同時実行トランザクションから見えるようになった後、トランザクションを確認します。

コミットスコープのルールでは、次のように、これらの確認レベルを ON を括弧で囲んでノードグループ定義に追加できます。

- ANY 2 (right_dc) ON replicated
- ALL (left_dc) ON visible （デフォルト）
- ALL (left_dc) ON received AND ANY 1 (right_dc) ON durable

リファレンス

コミットスコープの文法

コミットスコープの文法は次のように構成されています。

```
commit_scope:
    confirmation [AND ...]
```

(次のページに続く)

(前のページからの続き)

```
commit_scope_operation:
    commit_scope_group [ ON { received|replicated|durable|visible } ] commit_scope_
↪kind

commit_scope_group:
{ ANY num (node_group [, ...])
  | MAJORITY (node_group [, ...])
  | ALL (node_group [, ...]) }

commit_scope_kind:
{ GROUP COMMIT [ ( group_commit_parameter = value ) ] [ ABORT ON ( abort_on_parameter␣
↪= value ) ]
  | CAMO [ DEGRADE ON ( degrade_on_parameter = value ) TO ASYNC ]
  | LAG CONTROL [ ( lag_control_parameter = value ) ] }
```

パラメーター

- node_group —ノードグループの名前。

``(group_commit_parameter=value)``—*GroupCommit* **のオプション。**transaction_tracking（ブール値）& mdash; トランザクションのステータスを追跡するかどうかを指定します。``*conflict_resolution*`` **（列挙）& *mdash;* 競合の処理方法を指定します。可能な値は、**``*async*``**、競合解決ポリシーを使用してレプリケーション中に競合を非同期的に解決することを意味する、または** ``*eager*``**、競合するトランザクションの *1* つを中止することにより、*COMMIT* 中に競合が積極的に解決されることを意味します。**commit_decision（列挙）& mdash; COMMIT の決定がどのように行われるかを指定します。値 group は、commit_scope_group 仕様が耐久性だけでなく COMMIT の決定にも影響することを意味します。値 partner は、パートナーノードがトランザクションをコミットできるかどうかを決定することを意味します。この値は 2 つのデータノードグループでのみ許可されます。値 raft は、 commit_scope_group コンセンサスとは独立して Raft コンセンサスを使用して COMMIT 決定が行われることを意味します。

``*ABORTON(abort_on_parameter=value)*``**—タイムアウト時の自動トランザクションアボートを許可します。**timeout （インターバル）& mdash; ミリ秒単位のタイムアウト（他の単位を受け入れます）。

``*DEGRADEON(degrade_on_parameter=value)*``**—タイムアウト時に非同期操作への低下を許可します**timeout （インターバル）ミリ秒単位のタイムアウト（他の単位を受け入れます）。その後、操作は非同期になります。* require_write_lead（ブール値）& mdash; ノードが非同期モードに切り替えることができる書き込みリードでなければならないかどうかを指定します。

``*(lag_control_parameter=value)*``**—ラグ制御のオプション：**max_lag_size （int） WAL バイトに基づいた最大許容ラグ。``*max_lag_time*`` **（インターバル）& *mdash;* ウォールクロックサンプリングに基づく最大許容ラグ。**max_commit_delay（インターバル）& mdash; ラグ制限内に維持しようとするためにコミットに注入できる最大遅延。

注釈: *CAMO* コミットスコープの種類は、ほとんどが `GROUP COMMIT (transaction_tracking = true, commit_decision = partner)` の構文糖衣で、 *DEGRADE ON* 句が追加されています。 *GROUP COMMIT* は最終

的に `DEGRADE ON` 句も取得することが予想され、*CAMO* 構文は非推奨になります。

注釈: *synchronous_standby_names* と commit スコープの文法は似ていますが、前者は起点ノードを説明しませんが、後者は説明します。したがって、たとえば *synchronous_standby_names = 'ANY 1 (..)'* は `ANY 2 (...)` のコミットスコープと同等です。この選択により、多数決についての推論が容易になり、オリジンノードがトランザクションの耐久性と可視性にも貢献することを反映しています。

コミットスコープルールの追加

関数 bdr.add_commit_scope は、指定されたコミットスコープ名とオリジンノードグループのルールを作成します。ルールが EDB Postgres 分散クラスター内のすべてのノードで同じである場合、トップレベルのノードグループに対してこの関数を 1 回呼び出すだけで、コミットスコープを完全に定義できます。

または、同じ commit_scope_name で複数回呼び出すことができますが、起点ノードグループと、トランザクションの起点によって異なるコミットスコープのルールが異なります。

概要

```
bdr.add_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT
    wait_for_ready boolean DEFAULT true)
```

コミットスコープルールの変更

コミットスコープ内の単一のオリジンノードグループの特定のルールを変更するには、関数 bdr.alter_commit_scope を使用できます。

概要

```
bdr.alter_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME,
    rule TEXT)
```

コミットスコープルールの削除

bdr.remove_commit_scope を使用して、コミットスコープ内の単一のルールを削除できます。コミットスコープに複数のルールを定義する場合、ルールごとに 1 回この関数を呼び出して、コミットスコープ全体を完全に削除する必要があります。

概要

```
bdr.remove_commit_scope(
    commit_scope_name NAME,
    origin_node_group NAME)
```

注釈: ノードグループによってまだデフォルトとして使用されているコミットスコープを削除することは許可されていません。

Group Commit

Group Commit の目標は、単一ノードの障害または一時的な停止が発生した場合のデータ損失から保護することです。これを実現するには、COMMIT 時にトランザクションを正常に受信して確認するために複数の PGD ノードを要求します。

要件

通常の操作中、Group Commit はアプリケーションに対して透過的です。フェールオーバー中に進行中であったトランザクションは、アプリケーションまたは中間のプロキシによってトリガーまたは統合される調整フェーズを必要とします。このアクティビティは現在、オリジンノードが回復したとき、またはクラスターから分離したときにのみ発生します。この動作は、Postgres のレガシービルトイン同期レプリケーションの場合と同じです。

Group Commit でコミットされたトランザクションは、その下で 2 フェーズコミットを使用します。したがって、ノードごとに発生するそのようなすべてのトランザクションを処理するのに十分な高さに max_prepared_transactions を構成します。

構成

Group Commit を使用するには、最初に *commit* スコープへの移行を定義します。コミットスコープは、トランザクションのコミットに関与する PGD ノードを決定します。

動作

Group Commit の動作は、コミットスコープによって適用される構成によって異なります。

決定をコミットする

Group Commit を構成して、`group` 、`partner` 、および `raft` の 3 つの異なる方法でコミットを決定できます。

グループコミットの決定は、耐久性の目的で使用されるのと同じコミットスコープグループ設定を使用して指定されたコンセンサスを通じて行われます。違いは、コミットの決定は PREPARE レプリケーションに基づいて行われるのに対し、耐久性は COMMIT（PREPARED）レプリケーションをチェックすることです。

パートナーの決定は、
: *Commit At Most Once* が使用するものです。このアプローチは、ノードグループに 2 つのデータノードがある場合にのみ機能します。これら 2 つのノードはお互いのパートナーであり、オリジンではなくレプリカが何かをコミットするかどうかを決定します。このアプローチでは、アプリケーションが何らかの形でコンセンサスの一部であるため、

CAMO トランザクションプロトコルを使用して正しく動作するようにアプリケーションを変更する必要があります。このアプローチの詳細については、
: *CAMO/failover オプションを使用した pgbench* を参照してください。

raft オプションは、ビルトインの Raft コンセンサスを使用して、コミットを実行できるかどうかを決定します。現在、グローバル Raft が使用されています。これが機能するには、クラスター全体のノードの大部分が機能する必要があります。

競合の解決

競合解決は `async` または `eager` です。

非同期とは、PGD が特定のノードに構成された行レベルの解決を使用して、レプリケーション中に楽観的な競合解決を行うことを意味します。これは、元のトランザクションがコミットされたか進行中かに関係なく発生します。非同期競合解決がどのように機能するかの詳細は、
: *CRDT データ型を使用した列の競合の処理* を参照してください。

Eager は、競合が（COMMIT に関する合意の一部として）熱心に解決され、競合するトランザクションがシリアル化エラーで中止されることを意味します。このアプローチは、パフォーマンスを犠牲にして、非同期解決よりも優れた分離を提供します。Eager 競合解決の仕組みの詳細については、 *Eager conflict resolution* を参照してください。

中止

COMMIT でコンセンサスを取得できないトランザクションが永久にハングするのを防ぐために、ABORT ON 句でタイムアウトを指定できます。タイムアウト後、トランザクションアボートが要求されます。アボート要求が送信された時点でトランザクションのコミットがすでに決定されている場合、クライアントがアボートメッセージを受信する場合でも、トランザクションは最終的に COMMIT します。

制限事項

Group Commit トランザクションはまだ DDL を実行できず、明示的な 2 フェーズコミットをサポートしていません。将来のリリースでは、それらが許可される可能性があります。ただし、TRUNCATE コマンドは許可されます。

可用性が低下した場合に非同期操作に切り替えるための DEGRADE TO 句と CAMO トランザクションのみを組み合わせることができます。

Eager および CAMO トランザクションは、現在、デコードワーカー機能またはトランザクションストリーミングとの組み合わせでサポートされていません。Eager を使用したインストールでは、PGD ノードグループで enable_wal_decoder および streaming_mode を無効にしておく必要があります。

同期レプリケーションは、Group Commit とは異なるトランザクション確認のメカニズムを使用します。この 2 つは互換性がありません。一緒に使用しないでください。したがって、グループコミットトランザクションを行うときは常に、synchronous_standby_names で PGD ノードが構成されていないことを確認してください。

現在、 Raft のコミットの決定は非常に遅く、非常に低い TPS を生成します。 PGD 4 以前の Eager All-Node Replication 動作を取得するには、 eager conflict resolution 設定でのみ使用することをお勧めします。

同じトランザクションで異なるコミット決定オプションを組み合わせることはサポートされていません。同じトランザクションで異なる競合解決オプションを組み合わせることもサポートされていません。

Commit At Most Once

Commit At Most Once（CAMO）機能の目的は、アプリケーションが複数回コミットするのを防ぐことです。

CAMO がないと、 COMMIT が送信された後にクライアントが接続を失うと、アプリケーションはサーバーから応答を受信しない可能性があるため、トランザクションがコミットされたかどうかがわかりません。

アプリケーションは、次の 2 つのオプションのいずれかを簡単に決定できません。

- 場合によってはデータが 2 回入力される可能性があるため、同じデータでトランザクションを再試行します
- トランザクションを再試行せず、データがまったく処理されないリスク

これらはいずれも、価値の高いデータに関する重大なエラーです。

この状況を回避する 1 つの方法は、トランザクションが一意のインデックスを持つテーブルに少なくとも 1 つの INSERT を含めるようにすることです。ただし、それはアプリケーションの設計によって異なり、アプリケーション固有のエラー処理ロジックが必要になるため、すべての場合に効果的であるとは限りません。

PGD の CAMO 機能は、より一般的なソリューションを提供し、 INSERT を必要としません。 bdr.commit_scope によってアクティブ化されると、アプリケーションは、トランザクション識別子が既に割り当てられている場合、トランザクション識別子を含むメッセージを受け取ります。それ以外の場合、トランザクションの最初の write ステートメントはその情報をクライアントに送信します。

アプリケーションが明示的な COMMIT を送信する場合、プロトコルは、アプリケーションが COMMIT が送信される前にトランザクション識別子の通知を受信することを保証します。サーバーが COMMIT に応答しない場合、アプリケーションはトランザクション識別子を使用して別の PGD ノードからトランザクションの最終ステータスを要求することにより、このエラーを処理できます。以前のトランザクションのステータスがわかっている場合、アプリケーションはトランザクションを再試行するかどうかを安全に決定できます。

CAMO は、同じ PGD グループの 2 つの PGD ノードであるパートナーノードのペアを作成することによって機能します。この動作モードでは、ペアの各ノードは他のピアで実行された最近のトランザクションの結果を知っており、特に（必要に応じて）COMMIT 中に切断されたトランザクションの結果を知っています。アプリケーションからトランザクションを受信するノードを「オリジン」、これらのトランザクションを確認するノードを「パートナー」と呼ぶ場合があります。ただし、CAMO ペア内のノードの CAMO 構成に違いはありません。ペアは対称です。

> **警告:** CAMO では、高度なエラー処理を利用するには、ユーザーのアプリケーションを変更する必要があります。パラメーターを有効にするだけでは、保護を得るには十分ではありません。リファレンスクライアント実装は、要求に応じて顧客に提供されます。

要件

CAMO を使用するには、アプリケーションはマルチステートメントリクエストの一部としてではなく、別のリクエストとして明示的な COMMIT メッセージを発行する必要があります。CAMO は、プロシージャから発行されたトランザクション、または暗黙的なコミットを使用する単一ステートメントトランザクションのステータスを提供できません。

構成

CAMO の構成は *commit* スコープへの移行 を介して行われます。

失敗シナリオ

異なる構成では、さまざまな障害シナリオが発生します。

受信側でのデータの永続化

デフォルトでは、PGL ライタは、リモートノードからトランザクションを適用するときに `bdr.synchronous_commit = off` モードで動作します。これは CAMO にも当てはまります。つまり、CAMO パートナーのディスクに到達する前に、オリジンノードへのトランザクションが確認されることを意味します。クラッシュまたはハードウェア障害が発生した場合、確認されたトランザクションは、CAMO パートナーで単独では回復できない場合があります。CAMO パートナーノードが回復するとトランザクションを再分散するため、CAMO オリジンノードが動作し続けている限り、これは問題ではありません。

これは、CAMO が単一ノード障害から保護できることを意味します。これは、ローカルモードだけでなく、リモート書き込みとの組み合わせにも適しています。

CAMO ペアの両方のノードの停止をカバーするには、`bdr.synchronous_commit = local` を使用して、コミット前の確認の前にフラッシュを強制します。これはリモート書き込みモードでもローカルモードでも機能せず、レイテンシーの影響を受けやすいコミットパスにある CAMO パートナーの I / O 要件により、パフォーマンスに影響します。

非同期モード

`DEGRADE ON ... TO ASYNC` 句がコミットスコープで使用される場合、ノードはその CAMO パートナーの準備ができているかどうかを検出します。そうでない場合、一時的に非同期（ローカル）モードに切り替わります。このモードの場合、ノードは CAMO モードに戻るまでトランザクションをローカルでコミットします。

これにより、 COMMIT ステータスを取得することはできませんが、一貫性よりも可用性を選択できます。このモードは、単一ノードの障害に耐えることができます。CAMO ペアの両方のノードに障害が発生した場合、可用性を維持するために一致しないコミット決定を選択し、データの不整合につながる可能性があります。

CAMO パートナーが準備完了に切り替えるには、接続する必要があり、推定キャッチアップ間隔が `TO ASYNC` の `timeout` 値を下回る必要があります。 `bdr.is_camo_partner_ready` で CAMO パートナーの現在の準備状況を確認できます。`bdr.node_replication_rates` は、キャッチアップ時間の現在の見積もりを提供します。

CAMO 保護から非同期モードへの切り替えは、コミットが `TO ASYNC` の `timeout` 値を超えるため、または CAMO パートナーがすでにわかっている場合はコミット時に切断されるため、実際の CAMO トランザクションによってのみトリガーされます。このスイッチは、推定キャッチアップ間隔とは無関係です。`enable_proxy_routing` ノードグループオプションで構成されたように、現在のノードがグループの書き込みリードであることを要求するように CAMO ペアが構成されている場合。構文については、

> *commit* スコープへの移行 を参照してください。これにより、分離されたノードが非同期モードに切り替わることによるスプリットブレインの状況を防ぐことができます。

CAMO グループに `enable_proxy_routing` が設定されていない場合、起点ノードはすぐに非同期モードに切り替わります。

非同期モードから CAMO モードへの切り替えは、接続を開始する CAMO パートナーノードによって異なります。CAMO パートナーは、少なくとも 30 秒ごとに再接続を試行します。したがって、接続が再確立された後、CAMO パートナーが元のノードに接続するまでに最大 30 秒かかる場合があります。CAMO パートナーに蓄積されたラグは、CAMO 保護モードへの切り替えをさらに遅らせます。

通常の CAMO 操作中とは異なり、非同期モードでは、追加のコミットオーバーヘッドはありません。CAMO ペアが通常処理できるよりも多くのトランザクションをノードが継続的に処理できるようになるため、これは問題になる可能性があります。CAMO パートナーが最終的に再接続してトランザクションを適用した場合でも、そのような状況ではラグが増加するだけであり、CAMO 保護を再確立できません。トランザクションスループットを人為的に調整するために、PGD は `bdr.camo_local_mode_delay` 設定を提供します。これにより、ローカルモードで `COMMIT` を任意の時間遅延させることができます。予想されるワークロード中に通常の CAMO モードでコミット時間を測定し、それに応じてこの遅延を構成することをお勧めします。デフォルトは 5 ミリ秒で、これは非同期ネットワークと比較的迅速な CAMO パートナーの応答を反映しています。

アーキテクチャと可用性の要件を考慮して、非同期モードを許可するかどうかの選択を検討します。次の例は、いくつかの詳細を提供します。

例

この例では、互いの CAMO パートナーである 2 つの PGD ノードを使用したセットアップを検討します。

edb_notranlate_0 この CAMO コミットスコープが正当であるためには、グループ内のノードの数がちょうど 2 に等しい必要があります。非定量化されたグループ式は明確なペアに解決されないため、複数のノードで構成されるグループで ALL または ANY 2 を使用するとエラーになりますノードの。

非同期モードの場合

非同期モードが許可されている場合、単一障害点はありません。1 つのノードに障害が発生した場合：

- 他のノードは、障害が発生したノードの `COMMIT` 中に切断されたすべてのトランザクションのステータスを特定できます。
- 新しい書き込みトランザクションが許可されます。2 番目のノードにも障害が発生した場合、その時点でコミットされていたトランザクションの結果は不明です。

非同期モードなし

非同期モードが許可されていない場合、各ノードはトランザクションをコミットするために他のノードを必要とします。つまり、各ノードは単一障害点です。1 つのノードに障害が発生した場合：

- 他のノードは、障害が発生したノードの `COMMIT` 中に切断されたすべてのトランザクションのステータスを特定できます。
- ノードが回復するまで、新しい書き込みトランザクションは防止されます。

アプリケーション使用

概要と要件

CAMO は、クライアント側での再試行ループと特定のエラー処理に依存しています。それには 3 つの側面があります。

- トランザクションの COMMIT の結果を確認する必要があり、一時的なエラーの場合、クライアントはトランザクションをリトライする必要があります。
- COMMIT より前は、クライアントはノード ID とトランザクション ID（両方とも 32 ビット整数）で構成されるトランザクションのグローバル識別子を取得する必要があります。
- トランザクションの COMMIT を試行中に現在のサーバーに障害が発生した場合、アプリケーションは CAMO パートナーに接続し、そのトランザクションのステータスを取得し、応答に応じて再試行する必要があります。

アプリケーションは、 COMMIT 中に切断された場合のトランザクションステータスを確認する目的でのみ、グローバルトランザクション識別子を保存する必要があります。特に、アプリケーションは別の永続化レイヤーを必要としません。アプリケーションが失敗した場合、再起動するのに必要なのはデータベース内の情報のみです。

CAMO パートナー接続ステータス

ファンクション bdr.is_camo_partner_connected を使用すると、ペアモードで構成された CAMO パートナーノードの接続ステータスを確認できます。現在、Eager Replication で使用される CAMO に相当するものはありません。

概要

```
bdr.is_camo_partner_connected()
```

戻り値

CAMO パートナーが現在ローカルノード上の WAL 送信者プロセスに接続しているため、トランザクションデータを受信して確認を送信できるかどうかを示すブール値。

CAMO パートナーの準備

ファンクション bdr.is_camo_partner_ready を使用すると、ペアモードで構成された CAMO パートナーノードの準備ステータスを確認できます。下では、これによりローカルモードとの間の切り替えがトリガーされます。

概要

```
bdr.is_camo_partner_ready()
```

戻り値

CAMO パートナーが、ローカルノードから発信されるトランザクションをタイムリーに、つまり TO ASYNC の timeout が期限切れになる前に確認することが合理的に期待できるかどうかを示すブール値。

注釈: この関数は、過去または現在の状態を照会します。正の戻り値は、 CAMO パートナーが将来の取引を確認できるかどうかを示すものではありません。

CAMO パートナーを取得する

この機能は、ローカルノードの CAMO パートナー（ペアモードで構成）を表示します。

```
bdr.get_configured_camo_partner()
```

CAMO パートナーからの適用キューが消費されるのを待ちます

関数 bdr.wait_for_camo_partner_queue は、デフォルトで CAMO パートナーノードを照会する bdr.wait_for_apply_queue のラッパーです。ローカルノードが CAMO ペアの一部でない場合、エラーを返します。

概要

```
bdr.wait_for_camo_partner_queue()
```

CAMO ノード間のトランザクションステータス

この機能により、CAMO トランザクションが完全に解決されるのを待つことができます。

```
bdr.camo_transactions_resolved()
```

取引状況照会機能

ノードに障害が発生したときにコミットされていたトランザクションのステータスを確認するには、アプリケーションで次の関数を使用する必要があります。

```
bdr.logical_transaction_status(node_id OID, xid OID,
                               require_camo_partner boolean)
```

ペアモードで使用する CAMO では、CAMO ペアの一部であるノードでのみこの機能を使用します。 Eager Replication とともに 、すべてのノードで使用できます。

どちらの場合も、コミットが発行されてから 15 分以内に関数を呼び出す必要があります。CAMO パートナーは、このようなメタ情報を定期的に消去する必要があるため、古いトランザクションに正しい答えを提供できません。

トランザクションのステータスを照会する前に、この関数は受信キューが消費されて完全に適用されるのを待ちます。このメカニズムにより、受信されたがまだ適用されていないトランザクションに対する早期の否定的な回答が防止されます。

その名前にもかかわらず、それは常に読み取り専用の操作ではありません。ステータスが不明な場合、CAMO パートナーはトランザクションをコミットするかアボートするかを決定し、その決定をローカルに保存して、今後の一貫性を保証します。

クライアントは、オリジンでコミットする前にこの関数を呼び出さないでください。そうしないと、トランザクションが強制的にロールバックされる可能性があります。

概要

```
bdr.logical_transaction_status(node_id OID, xid OID,
                               require_camo_partner boolean DEFAULT true)
```

パラメーター

- node_id —トランザクションの発信元である PGD ノードのノード ID。通常、PQ パラメーター bdr.local_node_id から COMMIT の前にクライアントによって取得されます。
- xid —オリジンノードのトランザクション ID。通常、PQ パラメーター transaction_id から COMMIT の前にクライアントによって取得されます。

- require_camo_partner —デフォルトは true で、構成チェックを有効にします。これらのチェックを無効にし、CAMO トランザクションではないトランザクションのステータスを照会するには、false に設定します。

戻り値

この関数は、次のいずれかの結果を返します。

- 'committed'::TEXT —トランザクションはコミットされ、CAMO ペアの両方のノードで可視であり、最終的に他のすべての PGD ノードにレプリケートされます。クライアントが再試行する必要はありません。
- 'aborted'::TEXT —トランザクションは中止され、他の PGD ノードにレプリケートされません。クライアントは、トランザクションをコミットするために再試行するか、失敗をエスカレートする必要があります。
- 'in progress'::TEXT —トランザクションはこのローカルノードでまだ進行中であり、まだコミットまたは中止されていません。トランザクションは COMMIT フェーズにあり、CAMO パートナーがコミットを確認または拒否するのを待っている可能性があります。推奨されるクライアントの反応は、オリジンノードから切断し、CAMO パートナーに再接続して、代わりに照会することです。ロードバランサーまたはプロキシが間にあり、クライアントが照会するノードを制御できない場合、クライアントは、ステータスが'committed' または'aborted' に切り替わるまでのみ繰り返しポーリングできます。

Eager All-Node Replication の場合、ピアノードは、まだコミットまたは中止されていないトランザクションに対してこの結果を生成します。この場合、まだ複製されていない（またはオリジンノードで開始されていない）トランザクションでも、ピア PGD ノードで in progress 結果が得られる場合があります。ただし、クライアントは、オリジンでコミットする前にトランザクションステータスを照会してはなりません。

- 'unknown'::TEXT —指定されたトランザクションは不明です。それは未来のものであるか、その特定のノードにまだレプリケートされていないか、過去すぎます。そのようなトランザクションのステータスはまだないか、わかっていません。この戻り値は、クライアントによる不適切な使用の兆候です。

クライアントは、エラー時に関数呼び出しを再試行する準備ができている必要があります。

接続プールとプロキシ

CAMO クラスターを設計するときは、接続プールとプロキシの影響を考慮してください。プロキシは、コミットグループ内のすべてのノード、つまり CAMO ペアの両方のノード、または Eager All-Node Replication の場合はすべての PGD ノードにトランザクションを自由に分散する場合があります)。

アプリケーションが適切なノード ID をフェッチするように注意してください。セッションプーリングを使用する場合、クライアントは同じノードに接続されたままであるため、ノード ID はクライアントセッションの存続期間中一定のままです。ただし、より粒度の高いトランザクションプーリングを使用すると、クライアントはすべてのトランザクションのノード ID を取得する必要があります（次の例のように。

PGD ノードに直接接続されていないクライアントは、フェールオーバーまたはスイッチオーバーに気付かない場合があります。ただし、bdr.local_node_id パラメーターを常に使用して、現在接続されている

ノードを判断できます。COMMIT 中に切断するという重大な状況では、プロキシはその切断をエラーとして CAMO プロトコルを適用するクライアントに適切に転送する必要があります。

`received` モードの CAMO の場合、CAMO ペアを切り替える可能性のあるプロキシは、`bdr.wait_for_camo_partner_queue` ファンクションを使用して古い読み取りを防ぐ必要があります。

例

この例は、CAMO 対応クライアントアプリケーションの再試行ループが C のような擬似コードでどのように見えるかを示しています。接続情報を提供する 2 つの DSN `origin_dsn` および `partner_dsn` が必要です。これらは通常、`bdr.create_node` への最初の呼び出しに使用されたものと同じ DSN であり、`bdr.node_summary` 、列 `interface_connstr` で検索できます。

```
PGconn *conn = PQconnectdb(origin_dsn);

loop {
    // start a transaction
    PQexec(conn, "BEGIN");

    // apply transactional changes
    PQexec(conn, "INSERT INTO ...");
    ...

    // store a globally unique transaction identifier
    node_id = PQparameterStatus(conn, "bdr.local_node_id");
    xid = PQparameterStatus(conn, "transaction_id");

    // attempt to commit
    PQexec(conn, "COMMIT");
    if (PQresultStatus(res) == PGRES_COMMAND_OK)
        return SUCCESS;
    else if (PQstatus(res) == CONNECTION_BAD)
    {
        // Re-connect to the partner
        conn = PQconnectdb(partner_dsn);
        // Check if successfully reconnected
        if (!connectionEstablished())
            panic();

        // Check the attempted transactions status
        sql = "SELECT bdr.logical_transaction_status($node_id, $xid)";
        txn_status = PQexec(conn, sql);
        if (txn_status == "committed")
            return SUCCESS;
        else
            continue;   // to retry the transaction on the partner
```

(次のページに続く)

(前のページからの続き)

```
    }
    else
    {
        // The connection is intact, but the transaction failed for some
        // other reason.  Differentiate between permanent and temporary
        // errors.
        if (isPermanentError())
            return FAILURE;
        else
        {
            // Determine an appropriate delay to back-off to account for
            // temporary failures due to congestion, so as to decrease
            // the overall load put on the servers.
            sleep(increasing_retry_delay);

            continue;
        }
    }
}
```

この例は、再試行とエラー処理を含む、接続のための適切なロジックで拡張する必要があります。PgBouncer などのロードバランサーを使用している場合、 `PQreset` を使用して再接続を実装できます。ロードバランサーがクライアントのみを CAMO パートナーにリダイレクトし、他の PGD ノードにはリダイレクトしないことを確認します。

実際には、再試行の上限をお勧めします。トランザクションで実行されるアクションによっては、他の一時的なエラーが発生する可能性があります。`SERIALIZABLE` 分離モードでのデッドロックまたはシリアル化の失敗に関するベストプラクティスと同様に、エラーコードに応じてトランザクションを再試行することによって処理する必要があります。

DDL とグローバルロックとの相互作用

CAMO によって保護されたトランザクションには、DDL 操作を含めることができます。ただし、DDL はグローバルロックを使用します。これにより、ノード間の同期がすでに提供されます。詳細については、*DDL* ロックの詳細 を参照してください。

CAMO と DDL を組み合わせると、待ち時間が長くなり、グローバルデッドロックの可能性も高くなります。したがって、比較的低い DDL を使用することをお勧めします。DDL を中止し、妥当な時間内にデッドロックを解決します。

非トランザクション DDL

次の DDL 操作はトランザクションブロックで許可されていないため、CAMO 保護の恩恵を受けることはできません。これらの場合、CAMO は内部で自動的に無効になります。

*すべての同時インデックス操作（`CREATE` 、`DROP` 、および `REINDEX` ）

- `REINDEX DATABASE` 、`REINDEX SCHEMA` 、および `REINDEX SYSTEM`
- `VACUUM`

*パラメーターなしの `CLUSTER`

- `ALTER TABLE DETACH PARTITION CONCURRENTLY`
- `ALTER TYPE [enum] ADD VALUE`
- `ALTER SYSTEM`
- `CREATE` および `DROP DATABASE`
- `CREATE` および `DROP TABLESPACE`
- `ALTER DATABASE [db] TABLESPACE`

CAMO の制限

CAMO の制限については、 *PGD limitations* ページで説明されています。

パフォーマンスへの影響

CAMO は、コミット時にメッセージラウンドトリップを追加することにより、Postgres レプリケーションプロトコルを拡張します。アプリケーションは、非同期レプリケーションよりもコミットレイテンシーが高く、主に関連するノード間のラウンドトリップ時間によって決まります。同時セッションの数を増やすと、並列処理が増えて、適切なトランザクションスループットが得られます。

トランザクションを確認する CAMO パートナーは、トランザクション状態を保存する必要があります。非 CAMO オペレーションと比較して、これには、オリジンから適用されるトランザクションごとに追加のシークが必要になる場合があります。

クライアントアプリケーションのテスト

クライアント側で CAMO を適切に使用することは簡単ではありません。ノードのクラッシュやネットワークの停止などの障害シナリオに対して、PGD クラスターを使用してアプリケーションの動作をテストすることを強くお勧めします。

Lag Control

PGD オリジンノード上のデータベースアプリケーションのデータスループットは、コミットされたデータがダウンストリームピアノードに安全にレプリケートできる速度を超える場合があります。この不均衡が高可用性アプリケーションで一定期間を超えてまたは慢性的に続く場合、ディザスターリカバリーまたはビジネス継続性計画に関連する組織の目標が満たされない可能性があります。

レプリケーションラグコントロール（RLC）機能は、動的レート制限デバイスを使用してこの不均衡を調整し、PGD グループノード間のデータフローがこれらの組織の目標に準拠するように設計されています。これは、PGD ノード間のレプリケーションラグの程度を制御することによって行います。

これらの目的の一部には以下が含まれます。

- 目標復旧時点（RPO）は、計画外のイベントが原因で失われる可能性のあるデータの最大耐量を指定し、通常は時間量として表されます。レプリケートされないシステムでは、RPO を使用してバックアップ間隔を設定し、コミットされたデータが障害によって失われるリスクを制限します。レプリケートされたシステムの場合、RPO は、1 つ以上のピアノードに安全に適用されていないコミットされたデータの許容量を決定します。
- リソース制約目標（RCO）は、有限のストレージ制約があることを認めています。このストレージには、データベースファイル、WAL、および操作を継続するために必要な一時ファイルまたは中間ファイルが含まれます。レプリケートされたシステムの場合、ラグが増加すると、これらのストレージリソースに対する要求も増加します。
- グループ弾性目標（GEO）は、ノードがピアノードに受け入れられない形で保存できないクリップで新しいデータを発信していないことを保証します。その場合、その条件の検出は、データベースノードの数を拡張する決定における 1 つのメトリックとして使用できます。同様に、その状態が緩和すると、データベースノードの数を縮小する決定に影響を与える可能性があります。

ラグ制御は、クライアント接続が READ WRITE トランザクションをコミットできる速度を制御することにより、レプリケーションの遅延を管理します。レプリケーションラグは、達成する目的に応じて、ラグタイムまたはラグサイズとして測定されます。トランザクションのコミット率は、構成された PGD コミット遅延時間を使用して調整されます。

要件

ラグ制御の使用を開始するには：

- すべてのデータベースアプリケーションが許容できる最大許容コミット遅延時間 `max_commit_delay` を決定します。
- 使用するラグメジャーを決定します。ラグサイズ `max_lag_size` またはラグタイム `max_lag_time` のいずれかを選択します。
- 関与するグループまたはサブグループ、および確認を満たすために必要な各コレクション内のノードの最小数を決定します。この情報は、コミットスコープルールの定義の基礎を形成します。

構成

コミットスコープでラグ制御を指定すると、コミットスコープルールが及ぶノード全体で一貫性のある調整されたパラメーター設定が可能になります。トップグループのデフォルトのコミットスコープ、またはオリジングループのコミットスコープの一部として、ラグ制御仕様を含めることができます。

この例では、*commit* スコープへの移行のサンプルノードグループを使用して、2 つのデータセンターのラグ制御ルールを示します。

```
- - create a Lag Control commit scope with individual rules
- - for each sub-group
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := left_dc,
    rule := ALL (left_dc) LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s) AND␣
↪ANY 1 (right_dc) LAG CONTROL (max_commit_delay=500ms, max_lag_time=30s),
    wait_for_ready := true
);
SELECT bdr.add_commit_scope(
    commit_scope_name := example_scope,
    origin_node_group := right_dc,
    rule := ANY 1 (left_dc) LAG CONTROL (max_commit_delay=0.250ms, max_lag_
↪size=100MB) AND ALL (right_dc) LAG CONTROL (max_commit_delay=0.250ms, max_lag_
↪size=100MB),
    wait_for_ready := true
);
```

パラメーター値は、GUC パラメーター規則と互換性のあるユニット仕様を許可します。

グループコミットと CAMO ルールの仕様も含む既存のコミットスコープルールに、ラグ制御コミットスコープルールを追加できます。

max_commit_delay パラメーターは、必要に応じて、サブミリ秒の設定を含む、小数部を含むミリ秒の指定を許可および推奨します。

概要

ラグ制御は、クライアント接続レベルで動作する動的 TPS のレート制限メカニズムです。構成されたラグコントロールの制約を満たしながら、できるだけ目立たないように設計されています。これは、十分な PGD ノードが構成されたラグメジャーしきい値を下回るのに十分な速度で変更をレプリケートできる場合、PGD ランタイムのコミット遅延は 0 ミリ秒に固定されたままであることを意味します。

そうでない場合は、PGD ランタイムのコミット遅延を必要に応じて最小限に調整しますが、適合ノードの数が最小しきい値に戻るまで、それ以上には調整しません。

最小ノードしきい値に達した後でも、ラグ制御は、PGD ランタイムのコミット遅延をゼロに戻そうとし続けます。PGD コミット遅延は、ほとんどの場合、平衡レベル付近で上下する可能性がありますが、データスループットまたはラグ適用率が向上すると、コミット遅延は時間の経過とともに減少します。

PGD コミット遅延は、コミット後の遅延です。これは、トランザクションがコミットされた後、トランザクションによってロックまたは取得されたすべての Postgres リソースが解放された後に発生します。したがって、遅延は、同時アクティブなトランザクションによる値の監視または変更、またはリソースの取得を妨げることはありません。Postgres 拡張機能で管理される外部リソースについては、同じ保証はできません。拡張機能の依存関係に関係なく、PGD 拡張機能が `postgresql.conf` の拡張機能ベースのリソースマネージャーの前にリストされている場合、同じ保証を行うことができます。

厳密に言えば、PGD コミット遅延はトランザクションごとの遅延ではありません。これは、特定のクライアント接続のトランザクションのストリームでのコミット遅延の平均値です。この手法により、コミット遅延と値のきめ細かな調整により、OS スケジューラの粗い粒度、クロック割り込み、およびシステム負荷による変動を回避できます。また、PGD ランタイムのコミット遅延を可能な限り最小の期間のマイクロ秒以内に解決して、ラグメジャーのしきい値を維持することもできます。

注釈: PGD のコミット遅延と Postgres のコミット遅延を混同しないでください。それらは無関係であり、異なる機能を実行します。一方を他方で置き換えないでください。 !!!

取引申請

PGD コミット遅延は、ユーザーアプリケーションのデータを変更するすべての READ WRITE トランザクションに適用されます。この動作は、宣言された READ WRITE トランザクションを含む、データを変更しないトランザクションはコミット遅延から免除されることを意味します。

非同期トランザクションのコミットは、PGD コミット遅延も実行します。これは直感に反するように見えるかもしれませんが、非同期コミットは、そのパフォーマンスのおかげで、レプリケーションラグの最大の原因の 1 つになる可能性があります。

Postgres および PGD 補助プロセスは、トランザクションのコミットで遅延しません。最も注目すべきことに、PGD ライターは、ローカルノードでリモートトランザクションを適用するときにコミット遅延を実行しません。PGD ライターは送信レプリケーションラグに何も寄与せず、トランザクションコミットが遅延によって調整されないことにより、受信レプリケーションラグを最も削減できるため、これは設計によるものです。

制限事項

最大コミット遅延は、ハード制限を表す上限値であり、コミット遅延が設定値を超えることはありません。逆に、最大ラグはサイズと時間の両方で測定され、超えることができるソフト制限です。最大コミット遅延に達した場合、継続的な増加を防ぐための遅延対策に追加のバックプレッシャーはありません。

PGD レプリケーションセットを変更しないオリジントランザクションをコミット遅延から免除する方法はありません。これらのトランザクションでは、最大トランザクション遅延を 0 に SET LOCAL すると便利です。

注意事項

アプリケーション TPS は、レプリケーションラグに影響を与える可能性のある多くの要因の 1 つです。その他の要因には、PGD コミット遅延の効果が低下する可能性のあるトランザクションの平均サイズが含まれます。特に、バルクロード操作はレプリケーションラグの上昇を引き起こす可能性があります。

同様に、非常に高い OLTP 要件と適度なデータ変更を伴うアプリケーションは、許容可能な PGD コミット遅延設定によって不当に制限される可能性があります。

このような場合、`SET [SESSION|LOCAL]` コマンドを使用して、これらのアプリケーションのラグ制御設定をカスタム構成したり、それらのアプリケーションを変更したりすると便利です。たとえば、バルクロード操作は、トランザクションスナップショットの期間と WAL 保持サイズを制限したり、バルクロードが失敗した場合のリスタートポイントを確立したりするために、複数の小さなトランザクションに分割される場合があります。ラグ制御に従い、これらのトランザクションコミットは、非常に長い PGD コミット遅延をスケジュールして、以前の部分的なバルクロードによって寄与されたラグを消化できるようにすることもできます。

組織目標の達成

前述の目標リストの例では：

- RPO は、適切な最大ラグタイムを設定することで満たすことができます。
- RCO は、適切な最大ラグサイズを設定することで満たすことができます。
- GEO は、PGD ランタイムコミット遅延と PGD ランタイムラグメジャーを監視することによって満たすことができます。

前述のように、最大 PGD ランタイムコミット遅延が PGD で構成されたコミット遅延制限に固定され、ラグメジャーが一貫して PGD で構成された最大レベルを超える場合、このシナリオは PGD グループ拡張のマーカーになります。

8.8 Consistency

EDB Postgres Distributed は、アクティブ/アクティブまたはマルチマスターの DBMS です。非同期で使用すると、複数の異なるノードから同じ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データ *CRDT データ型を使用した列の競合の処理* が得られ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、それらは PGD が発生時に検出および解決できるイベントです。解像度はアプリケーションの性質とデータの意味によって異なるため、PGD がアプリケーションにそれらを解決する方法に関する幅広い選択肢を提供することが重要です。

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、ローカルまたはリモートのタプルが選択され、もう一方は破棄されます。たとえば、2 つの競合する変更のコミットタイムスタンプを比較し、新しい方を保持する場合があります。このアプローチにより、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスが確立されます。

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

競合を回避したい場合は、 ***Group Commit*** **を** ***Eager conflict resolution*** **または**
CRDT **データ型を使用した列の競合の処理** で説明されている競合のないデータ型（CRDT）とともに使用できます。

8.8.1 Conflicts

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

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

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、PGD はローカルまたはリモートのタプルを選択し、もう一方を破棄します。たとえば、2 つの競合する変更のコミットタイムスタンプを比較し、新しい方を保持する場合があります。このアプローチにより、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスが確立されます。

列レベルの競合解決の有効化と無効化 で説明されているように、競合処理は構成可能です。PGD は、ストリームトリガーリファレンス

で説明されている競合トリガーを使用して、競合を検出し、テーブルごとに異なる方法で処理できます。

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

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、表の所有者は競合を監視して回避方法を分析するか、アプリケーションタスクとして定期的に処理する計画を立てる必要があります。

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

一部のクラスタリングシステムは、分散ロックメカニズムを使用して、データへの同時アクセスを防ぎます。これらは、サーバーが互いに非常に近い場合は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ーが重要である地理的に分散したアプリケーションをサポートできません。

分散ロックは本質的に悲観的なアプローチです。PGD は楽観的なアプローチを提唱しています。これは、可能な場合は競合を回避しますが、特定の種類の競合の発生を許可し、発生したときに解決することです。

競合はどのように発生するか

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

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

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

PGD には、分散トランザクションマネージャーまたはロックマネージャーがありません。これが、レイテンシーとネットワークパーティションでうまく機能する理由の一部です。その結果、デフォルトの遅延レプリケーションを使用すると、異なるノード上のトランザクション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。そのため、PGD はこれが重要な場合に Eager Replication も提供します。

紛争の種類

PRIMARY KEY または UNIQUE の競合

最も一般的な競合は行の競合です。2 つの操作が、単一のノードではできない方法で同じキーを持つ行に影響します。PGD はそれらのほとんどを検出でき、 update_if_newer 競合リゾルバーを適用できます。

行の競合には次のものがあります。

- INSERT 対 INSERT
- UPDATE 対 UPDATE
- UPDATE 対 DELETE
- INSERT 対 UPDATE
- INSERT 対 DELETE
- DELETE 対 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競合解決が現在どのように構成されているかに関する情報を提供します。

INSERT / INSERT の競合

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

PGD は、発信元ノードのタイムスタンプに従って、2 つのうちの最後に挿入されたタプルを保持することにより、この状況を処理します。（ユーザー定義の競合ハンドラーは、この動作をオーバーライドできます。）

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

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

PGD グローバルシーケンス を使用すると、このタイプの競合を効果的に解消できます。

複数の UNIQUE 制約に違反する INSERT 操作

INSERT /INSERT の競合は、複数の UNIQUE 制約に違反する可能性があります。そのうちの 1 つが PRIMARY KEY である可能性があります。新しい行が複数の UNIQUE 制約に違反し、その結果他の複数の行と競合が発生する場合、レプリケーション変更を適用すると multiple_unique_conflicts 競合が発生します。

このような競合が発生した場合、レプリケーションを続行するには、いくつかの行を削除する必要があります。multiple_unique_conflicts のリゾルバー設定に応じて、適用プロセスはエラーで終了するか、入力行をスキップするか、一部の行を削除します。削除により、正しい PRIMARY KEY で行を保持し、他の行を削除しようとします。

> **警告:** この方法で競合する複数の行がある場合、競合解決の結果が挿入操作を続行する場合、一部のデータは常に削除されます。

競合トリガーを使用して、別の動作を定義することもできます。

UPDATE / UPDATE の競合

異なるノードでの 2 つの同時 UPDATE 操作が同じタプルを変更しますが、その PRIMARY KEY は変更しない場合、再生時に UPDATE /UPDATE の競合が発生する可能性があります。

これらは、構成と状況に基づいてさまざまな種類の競合を生成できます。テーブルが
行バージョンの競合検出

で構成されている場合、元の（キー）行がローカル行と比較されます。それらが異なる場合、update_differing 競合が生成されます。 **オリジンの競合検出** を使用する場合、行の原点がチェックされます。（起点は、現在のローカル行の元のノードです。）それが変更された場合、update_origin_change 競合が生成されます。他のすべての場合、UPDATE は通常、競合を生成せずに適用されます。

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

PRIMARY KEY での UPDATE の競合

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

主キーの更新に関する競合は **発散する競合** であり、手動介入が必要です。

Postgres で主キーを更新することは可能ですが、Postgres と PGD の両方に問題があります。

単純なスキーマは、以下を説明する例を提供します。

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

主キー列を更新できるため、次の SQL は成功します。

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

ただし、テーブルに複数の行があるとします。

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

一部の UPDATE 操作は成功します。

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

SELECT * FROM pktest;
 pk | val
- ---+-----
  2 |   1
  4 |   3
(2 rows)
```

他の UPDATE 操作は、制約エラーで失敗します。

```
UPDATE pktest SET pk=4 WHERE pk=2;
ERROR:  duplicate key value violates unique constraint "pktest_pkey"
DETAIL:  Key (pk)=(4) already exists
```

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

PGD では、UPDATE 操作が同時に複数の場所から許可されている場合、状況はより複雑になります。

これら 2 つの変更を同時に実行すると機能します。

```
node1: UPDATE pktest SET pk=pk+1 WHERE pk = 2;
node2: UPDATE pktest SET pk=pk+1 WHERE pk = 4;

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

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

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

このシナリオでは、各ノードで異なるデータを残します。

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

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

:ref:`LiveCompare によるデータ検証 <LiveCompare によるデータ検証>` を使用して、この状況を特定して解決できます。

同時競合は問題を引き起こします。これら 2 つの変更を同時に実行することは、解決するのは簡単ではありません。

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

両方の変更がローカルに適用されるため、ノード間で相違が発生します。ただし、ターゲットへの適用は、重複キー値違反エラーで両方のノードで失敗します。このエラーにより、レプリケーションが停止し、手動で解決する必要があります。

conflict_type update_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、レプリケーションは中断しません。このアプローチは、更新の性質によっては、依然として発散につながる可能性があります。

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

その結果、特に PGD では、アプリケーションで主キーの UPDATE 操作を許可しないことを強くお勧めします。アプリケーションの一部が主キーを変更する場合、同時変更を避けるために、Eager Replication を使用してこれらの変更を行います。

警告: *update_pkey_exists* の競合の競合解決の結果が更新の場合、行の 1 つが常に削除されます。

複数の UNIQUE 制約に違反する UPDATE 操作

複数の *UNIQUE* 制約に違反する *INSERT* 操作 と同様に、着信 UPDATE が複数の UNIQUE

インデックス（または PRIMARY KEY ）に違反すると、PGD は multiple_unique_conflicts 競合を発生させます。

PGD は、据え置きの一意の制約をサポートしています。トランザクションがソースでコミットできる場合、競合が見つからない限り、ターゲットにクリーンに適用されます。ただし、遅延主キーを REPLICA IDENTITY として使用することはできないため、ユースケースはすでにそのことと、複数の一意の制約の使用に関する警告によって制限されています。

UPDATE/DELETE の競合

あるノードは、別のノードが同時に削除する行を更新できます。この場合、再生時に UPDATE /DELETE の競合が発生する可能性があります。

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

ローカルノードのレプリケーションが遅れている場合、データベースは UPDATE を受け取るまでに削除された行をクリーンアップできます。この場合、PGD は UPDATE /DELETE conflicts と *INSERT / UPDATE* の競合 を区別できません。update_missing 競合を生成します。

DELETE と UPDATE の競合の別のタイプは、行がローカルで更新された後に来る DELETE です。この状況では、結果は使用される競合検出の種類によって異なります。デフォルトの

オリジンの競合検出 を使用する場合、競合は検出されず、 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。ただし、これを構成するか、競合トリガーで処理できます。

INSERT / UPDATE の競合

デフォルトの非同期操作モードを使用する場合、ノードは元の INSERT を受け取る前に行の UPDATE を受け取る場合があります。これは、3 つ以上のノードがアクティブな場合にのみ発生します（

3 つ以上のノードとの競合 を参照）。

これが発生すると、 update_missing の競合が生成されます。デフォルトの競合リゾルバーは insert_or_skip ですが、代わりに insert_or_error または skip を使用できます。insert-or-action を行うリゾルバーは、可能な場合（行全体を受け取ったとき）、最初に UPDATE からのデータに基づいて新しい行を INSERT しようとします。行の再構築を可能にするには、テーブルに REPLICA IDENTITY FULL が必要であるか、行にトーストデータが含まれていてはなりません。

トーストされたデータの詳細については、

TOAST サポート詳細 を参照してください。

INSERT/DELETE の競合

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

3 つ以上のノードとの競合 を参照)。

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

DELETE 操作は常に delete_missing 競合を生成します。デフォルトでは、操作をスキップすることで解決されます。

DELETE/DELETE の競合

DELETE /DELETE の競合は、2 つの異なるノードが同じタプルを同時に削除すると発生します。

このシナリオでは常に delete_missing 競合が生成されます。これは、デフォルトでは、操作をスキップすることで解決されます。

両方の DELETE 操作は同じ効果があるため、この競合は無害です。そのうちの 1 つを安全に無視できます。

3 つ以上のノードとの競合

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

これらの競合は、 UPDATE 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。これらは

発散する競合 です。

この競合タイプは、3 つ以上のマスターでのみ発生します。少なくとも 2 つのマスターがアクティブに書き込んでいる必要があります。

また、ノード 1 からノード 3 へのレプリケーションラグは、次の一連のアクションを許可できるように十分に高くする必要があります。

1. ノード 2 はノード 1 から INSERT を受け取ります

2. ノード 2 が UPDATE を実行します

3. ノード 3 はノード 2 から UPDATE を受信します

4. ノード 3 はノード 1 から INSERT を受け取ります

insert_or_error （または場合によっては update_missing 競合タイプの insert_or_skip 競合リゾルバー）を使用することは、これらの競合の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 の競合の扉が開きます。

1. ノード 1 は UPDATE を実行します

2. ノード 2 は DELETE を実行します

3. ノード 3 はノード 2 から DELETE を受信します

4. ノード 3 はノード 1 から UPDATE を受け取り、それを INSERT

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

別の方法は、 **一般的な積極的な *Eager Replication* の効果** 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 に置き換えられたことを除いて、 INSERT /UPDATE と同じです。これにより、 delete_missing の競合が発生する可能性があります。

PGD は、 update_missing の競合で発生するように、各 INSERT を最近削除されたチェックにすることを選択できます。ただし、これを行うと大多数のユーザーにペナルティが科されるため、現時点では代わりに delete_missing をログに記録します。

将来のリリースでは、 delete_missing の競合が発生した場合、
LiveCompare **によるデータ検証** を使用した再チェックにより、 INSERT /DELETE

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

これらの競合は、2 つの主な問題のユースケースで発生する可能性があります。

- INSERT の後にすぐに DELETE が続きます。キューイングアプリケーションで使用できます

・ テーブルの主キー識別子を再利用する場合

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

PGD は、レプリケーションを正しく動作させるために識別子の一意性に依存しているため、後者の場合に問題があります。

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

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

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

変更がシステムを通過するまでにかかる時間内に識別子の再利用が発生すると、問題が発生します。通常の操作ではその時間は短いかもしれませんが、ノードがダウンしていると、その間隔が数時間または数日に延長される可能性があります。

アプリケーションで一意の識別子を再利用しないことをお勧めします。その場合は、1 年以内に再利用を避けるための措置を講じます。

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

外部キー制約の競合

適用されてい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間の競合は、FOREIGN KEY （FK）制約でも発生する可能性があります。

PGD は session_replication_role = 'replica' で変更を適用するため、変更を適用するときに外部キー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、参照元テーブルへの挿入と同時に参照先テーブルで削除が発生すると、この状況により FK 違反が発生する可能性があります。このシナリオは、INSERT /DELETE の競合に似ています。

シングルマスター Postgres では、参照先テーブルの値を参照する INSERT /UPDATE は、行レベルロックを取得する前に DELETE 操作が完了するのを待つ必要があります。 DELETE が参照値を削除する場合、 INSERT /UPDATE は FK チェックに失敗します。

マルチマスター PGD では。ノード間の行レベルのロックはありません。参照元のテーブルの INSERT は、参照先のテーブルの DELETE の後ろで待機しないため、両方のアクションを同時に実行できます。したがって、参照元テーブルのあるノードの INSERT /UPDATE は、別のノードの参照先テーブルの DELETE と同時に値を使用できます。結果は、参照元テーブルに存在しなくなった参照元テーブルの値です。

実際には、この状況は、 DELETE 操作が参照テーブルでの DELETE 操作とは別のトランザクションで参照テーブルで発生する場合に発生しますが、これは一般的な操作ではありません。

Orders -> OrderItems などの親子関係では、これを行うのは一般的ではありません。OrderItem を完全に削除するよりも、キャンセル済みとしてマークする可能性が高くなります。参照/ルックアップデータの場合、新しいファクトデータに同じ値を使用すると同時にエントリを完全に削除することは珍しいことです。

ぶら下がっている FK は可能ですが、一般的にこのリスクは非常に低いです。したがって、PGD はこのケースをカバーするための一般的な解決策を課しません。これが発生する状況を理解したら、2 つの解決策が可能です。

最初の解決策は、FK の使用を、一般に一度に 1 つのノードからのみ変更される、まれに変更される、または変更の同時実行がアプリケーションによって仲介される密接に関連するエンティティに制限することです。このアプローチにより、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、PGD 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た場合、これらの関数は FOREIGN KEY 情報を使用して、参照列を NULL に設定することにより、FK の異常を回避します。SET NULL 制約があるかのように。このアプローチでは、1 つのトリガーですべての FK を再チェックするため、FK 違反を防ぐには、テーブルごとに 1 つのトリガーのみを追加する必要があります。

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

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

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

ALTER TABLE fact
    ENABLE REPLICA TRIGGER bdr_replica_fk_iu_trg;
```

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

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

ALTER TABLE refdata
    ENABLE REPLICA TRIGGER bdr_replica_fk_d_trg;
```

両方のトリガーを追加すると、外部キーのダングリングが回避されます。

TRUNCATE の競合

TRUNCATE は、すべての行の DELETE と同様に動作しますが、行ごとの削除ではなくテーブルデータを物理的に削除することにより、このアクションを実行します。その結果、行レベルの競合処理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、別の DML が TRUNCATE への他のノードで同時に実行された場合、リプレイの順序によって異なる変更が発生する可能性があります。

次のいずれかのアクションを実行できます。

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

除外制約の競合

PGD は除外制約をサポートしていないため、作成できません。

既存のスタンドアロンデータベースを PGD データベースに変換する場合は、すべての除外制約を手動で削除します。

分散非同期システムでは、さまざまなノード上のすべてのトランザクションが完全に分離されるため、制約に違反する行のセットが存在しないことを保証できません。除外制約は、除外制約違反のために再生を任意のノードから他のノードに進めることができない再生デッドロックにつながります。

PGD に除外制約の作成を強制した場合、またはスタンドアロンデータベースを PGD に変換するときに既存の制約を削除しない場合、レプリケーションが破損することが予想されます。再び進行させるには、着信リモートタプルが競合するローカルタプルを削除または変更して、リモートトランザクションを適用できるようにします。

ロールとテーブルスペースの違いによるデータの競合

競合は、ノードがロールなどの異なるグローバル（Postgres システム全体の）データを持つ場合にも発生する可能性があります。この競合により、オペレーション（主に DDL ）が発生する可能性があります。オペレーションは正常に実行され、1 つのノードでコミットできますが、他のノードに適用できません。

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

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

ロックの競合とデッドロックの中止

PGD ライタプロセスは通常のユーザーセッションとほとんど同じように動作するため、行とテーブルのロックに関する通常のルールに従います。これにより、PGD ライタプロセスが、ユーザートランザクションまたは相互に保持されているロックで待機する場合があります。

関連するロックには以下が含まれます。

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

PGD ライタプロセスは、ユーザートランザクションでデッドロックする可能性があります。ユーザー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。その逆も同様です。2 つのライタプロセスが互いにデッドロックする場合もあります。Postgres のデッドロック検出器は、問題のあるトランザクションの 1 つに介入して終了します。PGD ライタプロセスが終了した場合、再試行し、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、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ーセッションのロックの背後で長時間スタックしている場合、管理者はユーザーセッションを終了して、レプリケーションが再び流れるようにすることができます。ただし、これは、ユーザーが別のユーザーセッションに影響を与える長いロックを保持することと変わりありません。

Postgres で *log_lock_waits* 機能を使用すると、ロック関連のリプレイストールの特定に役立ちます。

発散する競合

発散競合は、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に発生します。発散する競合は発生しないはずですが、執筆時点でそのような競合のすべてを確実に防止できるわけではありません。

行の `PRIMARY KEY` を変更すると、すべてのノードが変更を再生する前に別のノードが同じ行のキーを変更した場合、発散競合が発生する可能性があります。主キーの変更を避けるか、指定された 1 つのノードでのみ変更します。

行データが関係する発散コンフリクトは、一般に、一方のノードのデータを他方のノードと一貫性を保つように手動で調整する管理者のアクションを必要とします。文書化されたとおりに PGD を使用し、安全でないとマークされた設定または機能を回避する限り、このような競合は発生しません。

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

TOAST サポート詳細

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

論理デコード（PGD がその上に構築されます）および論理レプリケーションで処理する TOAST 値は、テーブルのメイン行の一部として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）に記録されます。これは、特に UPDATE の競合を処理するときに問題を引き起こす可能性があります。トーストされた列の値を変更しなかった `UPDATE` ステートメントは、その列のない行を生成するためです。

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

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

ただし、これよりも微妙な問題は、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に発生します。（積極的なトランザクションは影響を受けません。）たとえば、A、B、および C という 3 つのノードを持つ EDB Postgres 分散クラスターでの次のワークロードを想像してください。

1. ノード A で： txn A1 は UPDATE SET col1 = ’toast data…’ を行い、最初にコミットします。

2. ノード B で： txn B1 は UPDATE SET other_column = ‘anything else’; を行います。 A1 の後にコミットします。

3. ノード C：ノード A への接続が遅れます。

4. ノード C： txn B1 が最初に適用され、col1 の TOASTed 列を見逃しますが、競合することなく適用されます。

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逃します。

PGD を使用する場合、このシナリオは通常問題になりません。（ビルトイン論理レプリケーションまたはマルチマスターのプレーンな pglogical を使用する場合です。）PGD は、TOAST 列の変更を最近レプリケートした行にローカル TOAST を検出し、ローカル UPDATE が t TOAST を変更します。したがって、PGD は、異なるノード間でトーストされたデータの不整合を防ぎます。この状況により、マルチプルのノードで更新が発生した場合、つまりタプルのオリジンが変更された場合、WAL ロギングが増加します。PGD AlwaysOn アーキテクチャの通常の場合と同様に、すべての更新が単一のノードから行われた場合、追加の WAL オーバーヘッドはゼロです。

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

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

insert_or_error の競合解決のために、 REPLICA IDENTITY FULL の使用はまだ必要です。

トーストされた列に関連するこれらの問題は、 REPLICA IDENTITY FULL を持つテーブルには影響しません。この設定では、行全体がキーの一部と見なされるため、トーストされた値が常にキーの一部としてログに記録されます。PGD は、新しい行を再構築し、キー行から欠落しているデータを埋めることができます。その結果、REPLICA IDENTITY FULL を使用すると、WAL サイズが大幅に増加する可能性があります。

競合の回避または許容

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

競合は、複数のノードで同時に発生している場合にのみ発生します。競合を回避する最も簡単な方法は、1 つのノードにのみ書き込むか、一度に 1 つの特定のノードから特定の方法で特定の行にのみ書き込むことです。

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

ノードがダウンする直前に変更を加える可能性があるため、変更が失われたように見えます。その後、同じ変更を再度行う可能性があり、異なるノードで 2 つの更新が発生します。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す。データの最終更新が保持されるため、拒否されます。

INSERT/INSERT の競合の場合、*PGD グローバルシーケンス*を使用して、このタイプの競合を防止します。

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

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

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

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

競合検出

PGD は、競合検出のために次のメカニズムを提供します。

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

オリジンの競合検出

オリジン競合検出は、トランザクションが発生したノードに記録されたコミットタイムスタンプを使用および依存します。これには、クロックが正しく動作するか、2 つのノード間の最速メッセージの許容範囲内にあることが必要です。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。パラメーター bdr.maximum_clock_skew および bdr.maximum_clock_skew_action を使用して、ノード間のクロックスキューを管理できます。

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

競合は、レプリケーション起点が変更されたかどうかに基づいて最初に検出されるため、競合とならない可能性のある状況で競合トリガーが呼び出されます。したがって、このメカニズムは誤検出の競合を生成する可能性があるため、正確ではありません。

Origin 情報は、行がフリーズするポイントまでのみ利用できます。フリーズされた後に行に到着した更新は競合を発生させないため、すべての場合に適用されます。これは、bdr_init_physical で新しいノードを追加するときの通常のケースであるため、その場合、競合を発生させると、多くの誤検知の結果が発生します。

オフラインだったノードが再接続してデータ変更の送信を開始すると、新しく到着した更新が更新するフリーズされた行よりも古い場合、発散エラーが発生する可能性があります。挿入と削除は、この状況の影響を受けません。

> ノードの再起動とダウンしたノードの回復 で説明されているように、長期にわたる停止のためにノードをダウンさせないことをお勧めします。

EDB Postgres Extended Server および EDB Postgres Advanced Server では、PGD はノードがダウンしているときに行のフリーズを抑制します。このメカニズムはこの状況を適切に処理するため、パラメーター設定を変更する必要はありません。

Postgres の他のバリアントでは、この状況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、バキュームされている行が現在の xid の vacuum_freeze_min_age xid より古い場合に発生します。これは、これらのパラメーターに適切に高い値を構成する必要があることを意味します。

- vacuum_freeze_min_age
- vacuum_freeze_table_age
- autovacuum_freeze_max_age

トランザクション率に基づいて値を選択し、データベースノードから競合データを削除する前にダウンタイムの猶予期間を提供します。たとえば、vacuum_freeze_min_age が 5 億に設定されている場合、1000 TPS を実行するノードは、競合データが削除されるまでに 5.5 日強ダウンする可能性があります。CommitTS データ構造は、その設定で 5 GB のディスク領域を必要とするため、トランザクションレートの低いシステムは、低い設定のメリットを享受できます。

最初に推奨される設定は次のとおりです。

sql # 10 億= 10GB

autovacuum_freeze_max_age = 1000000000

Vacuum_freeze_min_age = 500000000

8.8.2 autovacuum_freeze_max_age の 90 %

Vacuum_freeze_table_age = 900000000

次のことに注意してください。

・ autovacuum_freeze_max_age はノードスタート時にのみ設定できます。

- vacuum_freeze_min_age を設定できるため、低い値を使用すると行が早期にフリーズされ、競合が無視される可能性があります。個々のテーブルに autovacuum_freeze_min_age および toast.autovacuum_freeze_min_age を設定することもできます。
- CLUSTER または VACUUM FREEZE コマンドを実行すると、行が早期にフリーズされ、競合が無視される場合があります。

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

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

- check_full_tuple または PGD ノードグループを有効にします。
- 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 を有効にします。
- bdr.alter_table_conflict_detection を使用して、テーブルで行バージョンの追跡を有効にします。この関数は、指定した名前の列と、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。

カウンターは UPDATE でのみインクリメントされますが、この手法により、 UPDATE と DELETE の両方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。

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

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

テーブルの所有者は、特定のテーブルの競合検出の動作を変更できます。

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

- relation —新しい競合検出方法を設定するリレーションの名前。
- method —使用する競合検出方法。
- column_name —列検出データの格納に使用する列。これはスキップできます。その場合、競合検出方法に基づいて列名が選択されます。row_origin メソッドは、メタデータストレージ用の追加の列を必要としません。

認識されている競合検出方法は次のとおりです。

- row_origin —タプルに対して行われた以前の変更の起点（オリジンの競合検出を参照）。これは、テーブルに追加の列を必要としないサポートされている唯一の方法です。
- row_version —行バージョン列（ 行バージョンの競合検出 を参照）。
- column_commit_timestamp —列ごとのコミットタイムスタンプ（*CLCD* で説明）。
- column_modify_timestamp —列ごとの変更タイムスタンプ（*CLCD* で説明）。

column_commit_timestamp と column_modify_timestamp の競合検出方法の違いの詳細については、 現在とコミットのタイムスタンプを参照してください。

この関数は、 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。

この関数は、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。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して効果をロールバックでき、変更は現在のトランザクションで表示されます。

bdr.backwards_compatibility が 30618 以下に設定されていない限り、relation の所有者のみが bdr.alter_table_conflict_detection 関数を実行できます。

警告: 競合検出方法を追加の列を使用してメタデータを格納する方法から変更すると、その列が削除されます。

警告: この関数は、警告が `bdr.camo_enable_client_warnings` で無効になっていない限り、 CAMO を無効にして警告を出します。

PGD は、 conflict_type パラメーターとして使用できる次の競合タイプを認識します。

- insert_exists —着信挿入は、主キーまたは一意のキー/インデックスを介して既存の行と競合します。
- update_differing —着信更新のキー行がローカル行と異なります。これは、 行バージョンの競合検出を使用する場合にのみ発生します。
- update_origin_change —着信更新は、別のノードによって最後に変更された行を変更しています。
- update_missing —着信更新が、存在しない行を変更しようとしています。
- update_recently_deleted —着信更新が、最近削除された行を変更しようとしています。
- update_pkey_exists —着信更新により、 PRIMARY KEY が、変更を適用しているノードに既に存在する値に変更されました。
- multiple_unique_conflicts —着信行は、ターゲットテーブル内の複数の UNIQUE 制約/インデックスと競合します。
- delete_recently_updated —現在のノードでの行の最新の更新よりも古いコミットタイムスタンプを持つ着信削除、または行バージョンの競合検出 を使用する場合。
- delete_missing —着信削除は、存在しない行を削除しようとしています。
- target_column_missing —ターゲットテーブルには、受信行にある 1 つ以上の列がありません。
- source_column_missing —受信行には、ターゲットテーブルに存在する 1 つ以上の列がありません。
- target_table_missing —ターゲットテーブルがありません。
- apply_error_ddl —レプリケートされた DDL コマンドを適用すると、Postgres によってエラーがスローされました。

競合の解決

ほとんどの競合は自動的に解決できます。PGD は、デフォルトで last-update-wins メカニズム、より正確には update_if_newer 競合リゾルバーになります。このメカニズムは、競合検出に使用されるのと同じコミットタイムスタンプに基づいて、競合する 2 つの行のうちの最後に挿入または変更された行を保持します。特定のコーナーケースシナリオでの動作は、bdr.create_node_group 、または bdr.alter_node_group に使用される設定によって異なります。

PGD では、次の関数を使用して、競合解決のデフォルトの動作をオーバーライドできます。

bdr.alter_node_set_conflict_resolver

この関数は、指定されたノードでの競合解決の動作を設定します。

概要

```
bdr.alter_node_set_conflict_resolver(node_name text,
                                     conflict_type text,
                                     conflict_resolver text)
```

パラメーター

- node_name —変更されるノードの名前。
- conflict_type —設定を適用する競合タイプ（ 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ー（競合リゾルバーのリスト を参照)。

注意事項

現在、ローカルノードのみを変更できます。関数呼び出しは複製されません。複数のノードの設定を変更する場合は、ノードごとに関数を実行する必要があります。

この関数による構成の変更は、 bdr.create_node_group または bdr.alter_node_group で指定された競合解決のデフォルトの動作をオーバーライドします。

この関数はトランザクションです。変更をロールバックでき、現在のトランザクションで表示されます。

競合リゾルバーのリスト

PGD ではいくつかの競合リゾルバーを使用できますが、処理できる競合タイプのカバレッジは異なります。

- 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、次の変更でレプリケーションを続行します。insert_exists 、update_differing 、update_origin_change 、update_missing 、update_recently_deleted 、update_pkey_exists 、delete_recently_updated 、delete_missing 、target_table_missing 、target_column_missing 、および source_column_missing の競合に使用できます。
- skip_if_recently_dropped —ダウンストリームで最近（1 日以内）に削除されたため、ダウンストリームに存在しないテーブルの場合、リモート変更をスキップします。それ以外の場合はエラーをスローします。target_table_missing 競合タイプに使用できます。skip_if_recently_dropped 競合リゾルバーは、同じ名前のテーブルが削除された直後に再作成されると、課題を引き起こす可能性があります。その場合、ノードの 1 つが、テーブルを再作成するための DDL を認識する前に、再作成されたテーブルの DML を認識する可能性があります。次に、テーブルが最近削除されたと仮定して、リモートデータを誤ってスキップし、データ損失を引き起こします。したがって、この競合リゾルバーとともにオブジェクト名を削除した直後にオブジェクト名を再利用しないことをお勧めします。
- skip_transaction —競合を生成したトランザクション全体をスキップします。apply_error_ddl の競合に使用できます。
- update_if_newer —リモート行が競合するローカル行よりも後にコミットされた場合（発信元ノードの壁時計によって決定される）を更新します。タイムスタンプが同じ場合、ノード ID がタイブレーカーとして使用され、すべてのノードで同じ行が確実に選択されます（ノード ID が高いほど勝ちます）。insert_exists 、update_differing 、update_origin_change 、および update_pkey_exists の競合タイプに使用できます。
- update —常にレプリケートされたアクションを実行します。insert_exists （INSERT を UPDATE に変換します）、update_differing 、update_origin_change 、update_pkey_exists 、および delete_recently_updated （削除を実行します）に使用できます。
- insert_or_skip —オリジンから送信された利用可能な情報から新しい行を作成し、 INSERT を試みます。完全な行を作成するのに十分な情報がない場合、変更をスキップします。update_missing および update_recently_deleted 競合タイプに使用できます。
- insert_or_error —origin から送信された利用可能な情報から新しい行を作成して挿入しようとします。完全な行を構築するのに十分な情報がない場合、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 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、処理を続行します。target_column_missing 競合タイプに使用できます。
- ignore_if_null —リモート行の余分な列に NULL 値が含まれる場合、欠落しているターゲット列を無視します。それ以外の場合、エラーをスローし、レプリケーションを停止します。target_column_missing 競合タイプに使用できます。
- use_default_value —欠落している列値をデフォルト（それが列のデフォルトの場合は NULL を含む）で埋めて、処理を続行します。デフォルトの処理中のエラーまたは制約の違反（つまり、NOT

NULL 列の NULL デフォルト）は、レプリケーションを停止します。source_column_missing 競合タイプに使用できます。

insert_exists 、 update_differing 、 update_origin_change 、update_missing 、multiple_unique_conflicts 、update_recently_deleted 、 update_pkey_exists 、delete_recently_updated 、および delete_missing 競合タイプは、

競合トリガー を使用してユーザー定義ロジックで解決することもできます。

このマトリックスは、競合リゾルバーが処理できる競合タイプを区別するのに役立ちます。

	insert exists	update differing	update origin change	update missing	update recently deleted	update pkey exists	delete recently updated	delete missing	target column missing	source column missing	target table missing	multiple unique conflicts
error	X	X	X	X	X	X	X	X	X	X	X	X
skip	X	X	X	X	X	X	X	X	X	X	X	X
skip_if_recently_dr											X	
update_if_newer	X	X	X			X						
update	X	X	X			X	X					X
insert_or_skip				X	X							
insert_or_error				X	X							
ignore									X			
ignore_if_null									X			
use_default_value										X		
conflict_trigger	X	X	X	X	X	X	X	X				X

デフォルトの競合リゾルバー

Conflict type	Resolver
insert_exists	update_if_newer
update_differing	update_if_newer
update_origin_change	update_if_newer
update_missing	insert_or_skip
update_recently_deleted	skip
update_pkey_exists	update_if_newer
multiple_unique_conflicts	error
delete_recently_updated	skip
delete_missing	skip
target_column_missing	ignore_if_null
source_column_missing	use_default_value
target_table_missing	skip_if_recently_dropped
apply_error_ddl	error

競合解決のリスト

競合解決は、競合リゾルバーによって選択された解決の種類を表し、競合を解決するために取られた特定のアクションに対応します。

`conflict_resolution` パラメーターでは、現在、次の競合解決がサポートされています。

- `apply_remote` —リモート（着信）行が適用されました。
- `skip` —行の処理がスキップされました（ローカルで変更はありませんでした）。
- `merge` —リモート行とローカル行からの情報をマージして、新しい行が作成されました。
- `user` —ユーザーコード（競合トリガー）は、ターゲットテーブルに書き込まれた行を生成しました。

競合ログ

マルチマスターの競合の診断と処理を容易にするために、PGD はデフォルトで、すべての競合を `bdr.conflict_history` テーブルに記録します。次の関数を使用して、この動作をより詳細に変更できます。

bdr.alter_node_set_log_config

ノードの競合ログ構成を設定します。

概要

```
bdr.alter_node_set_log_config(node_name text,
                              log_to_file bool DEFAULT true,
                              log_to_table bool DEFAULT true,
                              conflict_type text[] DEFAULT NULL,
                              conflict_resolution text[] DEFAULT NULL)
```

パラメーター

- node_name —変更されるノードの名前。
- log_to_file —ノードログファイルに記録するかどうか。
- log_to_table — bdr.conflict_history テーブルにログを記録するかどうか。
- conflict_type —ログに記録する競合タイプ。NULL（デフォルト）はすべてを意味します。
- conflict_resolution —ログに記録する競合の解決。NULL（デフォルト）はすべてを意味します。

注意事項

ローカルノードのみ変更できます。関数呼び出しは複製されません。複数のノードの設定を変更する場合は、ノードごとに関数を実行する必要があります。

この関数はトランザクションです。変更をロールバックでき、現在のトランザクションで表示されます。

競合ログ構成のリスト

ビュー bdr.node_log_config は、すべてのロギング構成を表示します。ログ構成の名前、ログに記録する場所、ログに記録する競合の種類と解決がリストされます。

競合をテーブルに記録する

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

このテーブルは、列 local_time でレンジパーティション分割されています。テーブルは autopartition で管理されています。デフォルトでは、毎日新しいパーティションが作成され、過去 1 か月の競合が維持されます。その後、古いパーティションは自動的に削除されます。自動パーティションは、7〜14 個のパーティションを事前に作成します。bdr_superuser はこれらのデフォルトを変更できます。

PGD によって管理されるすべてのテーブルで生成された競合はこのテーブルに記録されるため、正当なユーザーのみが競合するデータを読み取れるようにすることが重要です。PGD は、 bdr.conflict_history テーブルで ROW LEVEL SECURITY ポリシーを定義することにより、これを行います。テーブルの所有者のみが、それぞれのテーブルの競合を読み取ることが許可されています。基になるテーブルに RLS ポリシーが定義、有効化、および適用されている場合、所有者でさえ競合を読み取ることはできません。FORCE オプションで作成された RLS ポリシーは、テーブルの所有者にも適用されます。その場合、基になるテーブルの一部またはすべての行が、所有者でさえ読み取れない場合があります。そのため、PGD は競合ログテーブルにより厳しいポリシーも適用します。

デフォルトのロール bdr_read_all_conflicts は、 bdr_superuser ロールを付与せずに、 bdr.conflict_history テーブルに記録されたすべての競合の詳細を表示する必要があるユーザーに付与できます。

デフォルトのロール bdr_read_all_stats は、bdr.conflict_history_summary というカタログビューにアクセスできます。これにはユーザーデータが含まれていないため、ログに記録された競合を監視できます。

競合レポート

レポートのテーブルに記録された競合を要約できます。レポートにより、アプリケーション所有者は競合を特定、理解、解決し、アプリケーションの変更を導入して競合を防ぐことができます。

```
SELECT nspname, relname
, date_trunc(day, local_time) :: date AS date
, count(*)
FROM bdr.conflict_history
WHERE local_time > date_trunc(day, current_timestamp)
GROUP BY 1,2,3
ORDER BY 1,2;

 nspname | relname |    date    | count
- --------+---------+------------+-------
 my_app  | test    | 2019-04-05 |     1
(1 row)
```

LiveCompare によるデータ検証

LiveCompare は、2 つのデータベースを比較して同一であることを確認するように設計されたユーティリティプログラムです。

LiveCompare は PGD スタックの一部として含まれ、PGD ノードの任意のペアを対象とすることができます。デフォルトでは、レプリケートされたすべてのテーブルを比較し、違いを報告します。LiveCompare は、Postgres や Oracle などの非 PGD データソースでも動作します。

LiveCompare を使用して、着信行を継続的に監視することもできます。コンテキスト情報を失うことなく停止および開始できるため、都合の良いときに実行できます。

LiveCompare を使用すると、複数のテーブルを同時にチェックできます。いくつかのテーブルまたはテーブル内の行のセクションのみをチェックできるように構成できます。チェックは、最初に行ハッシュ全体を比較することによって実行されます。異なる場合、LiveCompare は行全体を比較します。LiveCompare は、有用なサイズのバッチで行を比較することにより、オーバーヘッドを回避します。

違いが見つかった場合、それらは時間をかけて再チェックでき、最終的な一貫性の遅れを考慮に入れることができます。

詳細については、 *LiveCompare* **によるデータ検証** のドキュメントを参照してください。

8.8.3 Column-level conflict detection

デフォルトでは、競合は行レベルで解決されます。2 つのノードからの変更が競合する場合、ローカルまたはリモートのタプルが選択され、もう一方は破棄されます。たとえば、2 つの競合する変更のコミットタイムスタンプが比較され、新しい方が保持される場合があります。このアプローチにより、すべてのノードが同じ結果に収束し、クラスター全体でコミットオーダーのようなセマンティクスが確立されます。

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

テーブル t に 2 つの整数列 a および b と単一の行 (1,1) がある簡単な例を考えます。 1 つのノードで次を実行します。

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

別のノードで、先行する UPDATE を受け取る前に、同時に実行します。

```
UPDATE t SET b = 100
```

このシーケンスにより、UPDATE-UPDATE の競合が発生します。update_if_newer 競合解決により、コミットタイムスタンプが比較され、新しい行バージョンが保持されます。2 番目のノードが最後にコミットされたと仮定すると、結果は (1,100) になり、列 a への変更が効果的に破棄されます。

多くのユースケースでは、この動作は望ましいものであり、予期されています。ただし、ユースケースによっては、これが問題になる場合があります。たとえば、アプリケーションの各部分が異なるノードに接続され、共有テーブル内の列の専用サブセットを更新するマルチノードクラスターを考えてみます。その場合、さまざまなコンポーネントが競合し、変更を上書きする可能性があります。

このようなユースケースでは、特定のテーブルの競合を列レベルで解決する方が適切な場合があります。それを達成するために、PGD は各列の最終変更のタイムスタンプを個別に追跡し、それを使用して最新の値を選択し、基本的に update_if_newer を実行します。

前の例に適用すると、どちらのノードもそのような行を見たことがないにもかかわらず、結果は両方のノードで (100,100) です。

列レベルの競合解決について考えるときは、テーブルを垂直にパーティション分割されたものとして見ると便利です。そうすると、各更新は 1 つのスライスのデータのみに影響します。このアプローチにより、列の異なるサブセットに対する変更間の競合が排除されます。実際、垂直パーティション分割は、列レベルの競合解決の実用的な代替手段になることさえあります。

列レベルの競合解決では、テーブルに REPLICA IDENTITY FULL が必要です。 bdr.alter_table_conflict_detection 関数はそれをチェックし、この設定が欠落している場合はエラーで失敗します。

列レベルの競合解決の有効化と無効化

bdr.alter_table_conflict_detection() ファンクションは、列レベルの競合解決を管理します。

例

この例では、テーブル test_table を作成し、そのテーブルで列レベルの競合解決を有効にします。

```
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,
db(# column_modify_timestamp, cts);
 alter_table_conflict_detection
- -------------------------------
 t

db=# \d my_app.test_table
```

この関数は、関数呼び出しで指定されたとおりに cts 列を追加します。また、各変更の前に新しい列のタイムスタンプを維持する責任がある 2 つのトリガー（BEFORE INSERT および BEFORE UPDATE ）を作成します。

新しい列は、デフォルト値で NOT NULL を指定します。これは、ALTER TABLE ... ADD COLUMN はテーブルの書き換えを実行しないことを意味します。

注釈: *bdr.column_timestamps* データ型の列を他の目的に使用しないでください。悪影響を与える可能性があります。たとえば、テーブルを列レベルの競合解決に切り替えますが、これはトリガーがないと正しく機能しません。

列レベルの競合解決を備えたリストテーブル

次のクエリを使用して、列レベルの競合解決が有効になっているテーブルをリストできます。このクエリは、bdr.column_timestamp 型の列の存在を検出します。

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

bdr.column_timestamps_create

このファンクションは、列レベルの競合解消を作成します。column_timestamp_enable 内で呼び出されます。

概要

```
bdr.column_timestamps_create(p_source cstring, p_timestamp timestampstz)
```

パラメーター

- p_source — 2 つのオプションは current または commit です。
- p_timestamp —タイムスタンプは、選択したソースによって異なります。commit の場合、TIMESTAMP_SOURCE_COMMIT 。 current の場合、TIMESTAMP_SOURCE_CURRENT 。

DDL ロック

テーブルの列のタイムスタンプを有効または無効にする場合、コードは DDL ロックを使用して、切り替え前からの保留中の変更がないことを保証します。このアプローチにより、両方のタプルのタイムスタンプとの競合のみが保証されるか、どちらも表示されません。そうしないと、コードは予期せずにローカルタプルにタイムスタンプ、リモートタプルに NULL を表示する可能性があります。また、すべてのノードで変更が同じ方法（列レベルまたは行レベル）で解決されるようにします。

現在とコミットのタイムスタンプ

重要な決定は、変更された列に割り当てるタイムスタンプです。

デフォルトでは、変更された列に割り当てられるタイムスタンプは、あたかも clock_timestamp から取得したかのように現在のタイムスタンプです。これは簡単で、多くの場合正しいです（たとえば、競合する行が列の重複しないサブセットを変更する場合）。

ただし、さまざまな予期しない影響を与える可能性があります。

・ ステートメント実行中にタイムスタンプが変化する。したがって、UPDATE が複数の行に影響を与える場合、それぞれはわずかに異なるタイムスタンプを取得します。これは、異なるノードで実行された変更がどのようにインターリーブするかに応じて、同時変更の効果がさまざまな方法で「混合」される可能性があることを意味します。

- タイムスタンプはコミットタイムスタンプとは関係ありません。それを使用して競合を解決することは、結果がコミット順序と同等ではないことを意味し、つまり、シリアル化できない可能性が高いことを意味します。

注釈: ステートメントとトランザクションのタイムスタンプは将来的に追加される可能性があり、同時実行のステートメントまたはトランザクションの混合効果に関する問題に対処します。それでも、これらのオプションはどちらも、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、この機能は実験的と見なされます。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、最後のパラメーターを true に設定します。

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

bdr.column_timestamps_disable を使用して無効にできます。

コミットタイムスタンプには現在、 注意事項で説明されている制限があります。

列のタイムスタンプの検査

変更された列のタイムスタンプを格納する列は、トリガーによって維持されます。直接変更しないでください。たとえば、競合がどのように解決されたかを調査しているときに、現在のタイムスタンプ値を検査すると便利です。

この目的には、次の 3 つの関数が役立ちます。

- bdr.column_timestamps_to_text(bdr.column_timestamps)

この関数は、タイムスタンプマッピングの人間が読める形式の表現を返し、値を text にキャストするときに使用されます。

```
db=# select cts::text from test_table;
                                                cts
```

(次のページに続く)

(前のページからの続き)

```
- ------------------------------------------------------------------------------
↪----------------
 {source: current, default: 2018-09-23 19:24:52.118583+02, map: [2 : 2018-09-23␣
↪19:25:02.590677+02]}
(1 row)
```

- bdr.column_timestamps_to_jsonb(bdr.column_timestamps)

この関数は、タイムスタンプマッピングの JSONB 表現を変換し、値を jsonb にキャストするときに使用されます。

```
db=# select jsonb_pretty(cts::jsonb) from test_table;
                    jsonb_pretty
- ------------------------------------------------
 {                                                +
     "map": {                                     +
         "2": "2018-09-23T19:24:52.118583+02:00"  +
     },                                           +
     "source": "current",                         +
     "default": "2018-09-23T19:24:52.118583+02:00"+
 }
(1 row)
```

- bdr.column_timestamps_resolve(bdr.column_timestamps, xid)

このファンクションは、最新のトランザクションがすでにコミットされている場合、それ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。たとえば、この場合、最後のトランザクションは 2 番目の属性を更新しました (attnum = 2 を使用)。

```
test=# select cts::jsonb from test_table;
                                                                        cts
- ------------------------------------------------------------------------------
↪--------------------------------------------------
 {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
↪"2018-09-23T19:29:55.581823+02:00", "modified": [2]}
(1 row)

db=# select bdr.column_timestamps_resolve(cts, xmin)::jsonb from test_table;
                                                 column_timestamps_resolve
- ------------------------------------------------------------------------------
↪-----------------------------------
 {"map": {"2": "2018-09-23T19:29:55.581823+02:00"}, "source": "commit", "default":
↪"2018-09-23T19:29:55.581823+02:00"}
(1 row)
```

CRDT データ型を使用した列の競合の処理

デフォルトでは、列レベルの競合解決はより高いタイムスタンプを持つ値を選択し、もう一方を破棄します。ただし、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。たとえば、情報を破棄せずに競合する値をマージできる CRDT タイプを使用できます。

注意事項

- UPDATE によって変更される属性は、トリガーの古い行と新しい行を比較することによって決定されます。これは、属性の値が変更されない場合、明示的に設定されていても、変更されたものとして検出されないことを意味します。たとえば、UPDATE t SET a = a は a をどの行でも変更済みとしてマークしません。同様に、UPDATE t SET a = 1 は、 a が既に 1 に設定されている行の変更済みとしてマークしません。
- INSERT ステートメントの場合、新しい行と比較する古い行がないため、すべての属性が変更されたと見なされ、新しいタイムスタンプが割り当てられます。この条件は、INSERT ステートメントに含まれず、デフォルト値を受け取った列にも適用されます。PGD は、デフォルト値を持つ属性を検出できますが、それが自動的に含まれたのか、明示的に指定されたのかを知ることはできません。

この状況は、 INSERT ステートメントが列の異なるサブセットを指定していても、列レベルの競合解決が INSERT-INSERT 競合に対して機能しないことを効果的に意味します。新しい行には、古い行よりもすべて新しいタイムスタンプがあります。

- 列を個別に扱うことにより、すべての変更が同じノードで発生する場合に不可能な方法で制約に違反するのは簡単です。たとえば、次のようなテーブルを考えます。

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

1 つのノードが次のことを行うと仮定します。

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

別のノードが同時に行います：

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

これらの各更新は、最初の行で実行されたときに有効であるため、各ノードに渡されます。ただし、他のノードに複製すると、結果の行は CHECK (A > b) 制約に違反し、問題が手動で解決されるまで複製は停止します。

・タイムスタンプマッピングを格納するカラムは自動管理されます。結果が予測不能になる可能性があるため、クエリで値を指定またはオーバーライドしないでください。(値は可能な場合は無視されます。)

- タイムスタンプマッピングはトリガーによって維持されますが、トリガーが実行される順序が重要です。そのため、タプルを変更し、pgl_clcd_ トリガーの後に実行されるカスタムトリガーがある場合、変更された列は正しく検出されません。

- 通常のタイムスタンプを使用して変更またはコミットを注文する場合、2 つ以上のノードがたまたま同じタイムスタンプを生成したため、競合する変更のタイムスタンプがまったく同じになる場合があります。このリスクは、通常の行レベルの競合解決でも発生する可能性があるため、列レベルの競合解決に固有のものではありません。ノード ID は、この状況のタイブレーカーです。高いノード ID が勝ちます。このアプローチにより、すべてのノードに同じ変更が適用されます。
- 異なるノード間でクロックスキューが発生する可能性があります。タイムスタンプが逆になるため、一見新しい変更を破棄し、やや予期しない動作を引き起こす可能性があります。ただし、パラメーター bdr.maximum_clock_skew および bdr.maximum_clock_skew_action を使用して、ノード間のクロックスキューを管理できます。

```
SELECT bdr.alter_node_group_config(group, ignore_redundant_updates := false);
```

8.8.4 Conflict-free replicated data types

競合のないレプリケートデータ型（CRDT）は、従来の解決のように行の 1 つを破棄するのではなく、同時に変更される行の値のマージをサポートします。

各 CRDT 型は、bdr.crdt_handlers カタログに追加された追加のコールバックを備えた個別の PostgreSQL データ型として実装されます。マージプロセスは、ユーザーアクションを必要とせずに、適用側の PGD ライタ内で発生します。

> *CLCD* で説明されているように、CRDT では、テーブルで列レベルの競合解決を有効にする必要があります。

あなたが行う必要がある唯一のアクションは、整数などの標準の組み込みデータ型ではなく、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ーと 1 つの行がある次のテーブルについて考えます。

```
CREATE TABLE non_crdt_example (
    id      integer             PRIMARY KEY,
    counter integer             NOT NULL DEFAULT 0
);

INSERT INTO non_crdt_example (id) VALUES (1);
```

2 つのノードで次の SQL を同時に発行するとします。

```
UPDATE non_crdt_example
   SET counter = counter + 1    -- "reflexive" update
 WHERE id = 1;
```

両方の更新を適用した後、次のクエリを使用して結果の値を確認できます。

```
SELECT * FROM non_crdt_example WHERE id = 1;
   id |   counter
 -----+-----------
```

(次のページに続く)

(前のページからの続き)

```
    1 |         1
(1 row)
```

このコードは、 update_if_newer 競合リゾルバーが原因でインクリメントの 1 つを失ったことを示しています。代わりに CRDT カウンターデータ型を使用すると、結果は次のようになります。

```
CREATE TABLE crdt_example (
    id      integer             PRIMARY KEY,
    counter bdr.crdt_gcounter   NOT NULL DEFAULT 0
);

ALTER TABLE crdt_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_example,
            column_modify_timestamp, cts);

INSERT INTO crdt_example (id) VALUES (1);
```

再び 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)
```

この例は、CRDT が、そうでなければ競合する非同期同時更新に直面しても、アキュムレータ列が正しく機能できることを示しています。

crdt_gcounter タイプは、例が示すように、 x = x + 1 などの再帰 UPDATE SQL でのみ機能する状態ベースの CRDT タイプの例です。

bdr.crdt_raw_value 構成オプションは、クエリが現在の値を返すか、CRDT 型の完全な内部状態を返すかを決定します。デフォルトでは、現在の数値のみが返されます。true に設定すると、クエリは完全な状態の表現を返します。特別なハッシュ演算子（# ）を使用すると、特別な演算子を使用せずに現在の数値のみを要求できます（デフォルトの動作）。bdr.crdt_raw_value = on を使用して完全な状態がダンプされる場合、値は bdr.crdt_raw_value = on でのみリロードできます。

注釈: *bdr.crdt_raw_value* は、クライアントに返されるデータ、つまり選択リスト内の単純な列参照のフォーマットのみを適用します。クエリの他の部分（*WHERE* 句や選択リスト内の式など）での列参照では、引き続き`#` 演算子の使用が必要になる場合があります。

CRDT データ型の別のクラスは、**デルタ** *CRDT* 型と呼ばれます。これらは、オペレーションベースの CRDT の特別なサブクラスです。

デルタ CRDT では、値の更新は同じノード上の以前の値と比較されます。次に、変更は他のすべてのノードにデルタとして適用されます。

```
CREATE TABLE crdt_delta_example (
    id       integer            PRIMARY KEY,
    counter  bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

ALTER TABLE crdt_delta_example REPLICA IDENTITY FULL;

SELECT bdr.alter_table_conflict_detection(crdt_delta_example,
            column_modify_timestamp, cts);

INSERT INTO crdt_delta_example (id) VALUES (1);
```

2 つのノードで次の SQL を同時に発行するとします。

```
UPDATE crdt_delta_example
   SET counter = 2          -- notice NOT counter = counter + 2
 WHERE id = 1;
```

両方の更新を適用した後、次のクエリを使用して結果の値を確認できます。

```
SELECT id, counter FROM crdt_delta_example WHERE id = 1;
   id | counter
 -----+---------
    1 |       4
(1 row)
```

通常の integer 列の場合、結果は 2 です。ただし、デルタ CRDT カウンターで行を更新する場合、古い行バージョンから始めて新しい行バージョンを作成し、両方をリモートノードに送信します。そこで、そこにあるバージョン（ローカルバージョンなど）と比較します。標準 CRDT は NEW バージョンと LOCAL バージョンをマージしますが、デルタ CRDT は OLD バージョンと NEW バージョンを比較し、デルタを LOCAL バージョンに適用します。

CRDT タイプは、 bdr の一部として bdr スキーマにインストールされます。便宜上、基本的な演算子（+ 、# および! ）と多くの一般的な集計関数（min 、max 、sum 、および avg ）が pg_catalog で作成されます。したがって、search_path を調整せずに利用できます。

重要な質問は、クエリの計画と最適化がこれらの新しいデータ型でどのように機能するかです。CRDT タイプは透過的に処理されます。 ANALYZE とオプティマイザーの両方が機能するため、見積もりとクエリプランニングは、他に何もしなくても正常に機能します。

状態ベースおよびオペレーションベースの CRDT

[1] の表記に従って、操作ベースと状態ベースの両方の CRDT が実装されます。

オペレーションベースの CRDT タイプ（CmCRDT）

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け取った新旧の行から計算されるため、オペレーションベースの型の実装は簡単です。

現在、次のオペレーションベースの CRDT が実装されています。

- `crdt_delta_counter` — `bigint` カウンター（インクリメント/デクリメント）
- `crdt_delta_sum` — `numeric` sum（インクリメント/デクリメント）

これらの型は、デルタを計算するための少しのコードで既存のデータ型を活用します。たとえば、`crdt_delta_counter` は `bigint` 上のドメインです。

このアプローチは、デルタの計算方法がわかっているタイプでのみ可能ですが、結果はシンプルで安価（スペースと CPU の両方）であり、いくつかの利点があります。たとえば、基になるデータ型の演算子/構文を活用できます。

主な欠点は、非同期および並行環境でこの値を確実にリセットできないことです。

注釈: カスタムデータ型を作成することにより、より複雑な操作ベースの型を実装し、状態と最後の操作を保存することが可能です。（すべての変更がデコードされて転送されるため、複数の操作は必要ありません）。しかし、その時点で、主な利点（単純さ、既存のデータ型の再利用）は失われ、領域要件を除き、状態ベースの型と比較して利点を得ることはありません（たとえば、まだリセットする機能はありません）。（ノードごとの状態は必要ありません。）

状態ベースの CRDT タイプ（CvCRDT）

状態ベースの型はより複雑な内部状態を必要とするため、操作ベースの型のように通常のデータ型を直接使用することはできません。

現在、4 つの状態ベースの CRDT が実装されています。

- `crdt_gcounter` — `bigint` カウンター（インクリメントのみ）
- `crdt_gsum` — `numeric` 合計/カウンター（インクリメントのみ）
- `crdt_pncounter` — `bigint` カウンター（インクリメント/デクリメント）
- `crdt_pnsum` — `numeric` 合計/カウンター（インクリメント/デクリメント）

内部状態には通常、ノードごとの情報が含まれ、ディスク上のサイズが大きくなりますが、追加の利点が得られます。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（入出力関数と演算子）を意味します。

利点は、値を確実にリセットできること、変更が失われた場合の自己修復性（適切に動作するクラスターでは発生しません）、およびソースノード以外から変更を受信できることです。

たとえば、ノード A で値が変更され、A と C 間のネットワーク問題が原因で、変更が B にレプリケートされますが、C にはレプリケートされないとします。B が値を変更し、この変更が C にレプリケートされる場合、それには A からの元の変更。オペレーションベースの CRDT では、ノード C は A-C ネットワーク接続が再び機能し始めるまで変更を受け取りません。

CvCRDT の主な欠点は、ディスク領域と CPU 使用率のコストが高いことです。クラスターから既に削除されたノードを含む、各ノードに関する少しの情報が必要です。状態の複雑な性質（varlena タイプにシリアル化）は、CPU 使用率の増加を意味します。

ディスク容量要件

重要な考慮事項は、CRDT タイプ、特にディスク上のサイズに関連するオーバーヘッドです。

オペレーションベースのタイプの場合、タイプは他のタイプの上の単なるドメインであるため、これは簡単です。ノードの数に関係なく、ディスク領域の要件は同じです。

- `crdt_delta_counter` — `bigint` と同じ（8 バイト）
- `crdt_delta_sum` — `numeric` と同じ（可変、精度とスケールに応じて）

オペレーションベースの CRDT タイプはノードごとの情報を格納しないため、ノード数に依存しません。

状態ベースの型の場合、状況はより複雑です。すべての型は可変長（基本的に `bytea` 列として格納）であり、ヘッダーと、値を変更した各ノードの一定量のノードごとの情報で構成されています。

`bigint` バリアントの場合、おおよそのサイズを計算する式は次のとおりです。

- `crdt_gcounter` — `32B (header) + N * 12B (per-node)`
- `crdt_pncounter` -— `48B (header) + N * 20B (per-node)`

`N` は、この値を変更したノードの数を示します。

`numeric` バリアントの場合、ヘッダー部分とノードごとの部分の両方に `numeric` 可変長値が含まれるため、正確な式はありません。そのような値をいくつ保持する必要があるかを知るには：

- `crdt_gsum` -固定：`20B (header) + N * 4B (per-node)` -変数：`(2 + N)` `numeric` 値
- `crdt_pnsum` -固定：`20B (header) + N *4B(per-node)`-変数：`(4+2* N)` `numeric` 値

注釈: 値が複数のノードで更新されない場合、クラスター内にノードがいくつあっても構いません。また、更新が同時に行われたかどうか（競合が発生したかどうか）も関係ありません。

さらに、これらのノードのうちのいくつがクラスターから既に削除されていても構いません。状態をコンパクトにする方法はまだありません。

CRDT タイプと競合処理

テーブルには CRDT 列と非 CRDT 列の両方を含めることができるため（ほとんどの列は非 CRDT であることが予想されます）、通常の競合解決と CRDT マージの両方を行う必要があります。

競合の解決が最初に行われ、保持するタプル（applytuple）と破棄するタプルを決定します。次にマージフェーズが発生し、破棄されたタプルの CRDT 列のデータが applytuple にマージされます。

注釈: この処理により、マージは毎回発生する必要があるため、CRDT タイプはプレーンな競合解決と比較してやや高価になります。これは、競合解決でいずれかの高速パス（現在のトランザクションで変更されたパスなど）を使用できる場合でも当てはまります。

CRDT タイプと競合レポート

デフォルトでは、検出された競合は個別に報告されます。CRDT タイプがなければ、これは理にかなっています。競合解決は基本的に利用可能な情報の半分（構成に応じてローカルまたはリモートの行）を破棄するためです。これは、データの損失を示します。

CRDT タイプでは、何も捨てずに情報の両方の部分を結合できるため、データ損失の問題が解消されます。このアプローチにより、競合レポートが不要になります。

このため、CRDT マージで競合を完全に解決できる場合、競合レポートはスキップされます。各列は、次の 2 つの条件の少なくとも 1 つを満たす必要があります。

- ローカルタプルとリモートタプルの値が同じ（NULL または等しい）。
- CRDT データ型を使用しているため、マージできます。

注釈: CRDT 列がないが、ローカル/リモートのタプルのすべての値が等しい場合、競合レポートもスキップされます。

CRDT 値のリセット

CRDT 値をリセットすることは可能ですが、特別な処理が必要です。クラスターの非同期の性質は、実装方法に関係なく、さまざまなノードが変更ストリームのさまざまな場所でリセットオペレーションを見る可能性があることを意味します。異なるノードが同時に、つまり他のノードからのリセットを監視する前に、リセットを開始する場合もあります。

言い換えれば、リセット操作を正しく動作させるには、通常の操作に対して可換である必要があります。単一ノードでうまくいく可能性のある値をリセットする多くの素朴な方法は、この理由で失敗します。

たとえば、値をリセットする最も簡単なアプローチは次のとおりです。

```
UPDATE crdt_table SET cnt = 0 WHERE id = 1;
```

状態ベースの CRDT では、これは機能しません。他のノードの状態を破棄しますが、ローカルのみです。リモートノードのマージ関数によって追加され、値が発散し、最終的に他のノードの変更により値が返されます。

操作ベースの CRDT では、更新が-cnt の減算として解釈されるため、これは機能するように見える場合があります。ただし、同時リセットがない場合にのみ機能します。2 つのノードが同時にリセットを試行すると、デルタが 2 回適用され、負の値が取得されます（これはリセットでは予期されません）。

DELETE + INSERT をリセットとして使用できるように思われるかもしれませんが、このアプローチにはいくつかの弱点もあります。行が同じキーで再挿入された場合、すべてのノードが他のノードからの変更に関して、操作のストリーム内の同じ位置でそれを見ることは保証されません。PGD は、同時発生のデータ異常につながる可能性があるため、同じ主キー値の再利用を特にお勧めしません。

状態ベースの CRDT 型は、次のような特別な ! 演算子を使用して、リセットを確実に処理できます。

```
UPDATE tab SET counter = !counter WHERE ...;
```

「信頼できる」とは、値に複数の同時リセットと発散の 2 つの問題がないことを意味します。

操作ベースの CRDT タイプは、**一般的な積極的な** *Eager Replication* **の効果**を使用してのみ確実にリセットできます。これにより、複数の同時リセットが回避されるためです。Eager Replication を使用して、いずれかの種類の CRDT を特定の値に設定することもできます。

実装された CRDT データ型

現在、6 つの CRDT データ型が実装されています。

- 成長専用のカウンターと合計
- 正負のカウンターと合計
- デルタカウンターと合計

カウンターと合計は、カウンター型が整数ベース（bigint ）であり、合計タイプが 10 進ベース（numeric ）であることを除いて、ほとんど同じように動作します。

[1] で説明されている追加の CRDT タイプは、後で実装される可能性があります。

次のクエリを使用して、現在実装されている CRDT データ型をリストできます。

```
SELECT n.nspname, t.typname
FROM bdr.crdt_handlers c
JOIN (pg_type t JOIN pg_namespace n ON t.typnamespace = n.oid)
  ON t.oid = c.crdt_type_id;
```

成長専用カウンター（crdt_gcounter）

- 非負の値を持つインクリメントのみをサポート（value + int および counter + bigint 演算子）。
- # 演算子を使用するか、bigint にキャストすることにより、カウンターの現在の値を取得できます。
- counter = value のような単純な割り当てと互換性がありません（これは、アプリケーションのどこかで新しい値が計算される場合の一般的なパターンです）。
- ! 演算子（ counter = !counter ）を使用して、カウンターの単純なリセットを許可します。
- crdt_gcounter_to_text を使用して内部状態を検査できます。

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_gcounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- error: negative value

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET cnt = cnt + 1 WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120 WHERE id = 2;

- - error: minus operator not defined
UPDATE crdt_test SET cnt = cnt - 1 WHERE id = 1;

- - error: increment has to be non-negative
UPDATE crdt_test SET cnt = cnt + (-1) WHERE id = 1;

- - reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

- - get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

- - show internal structure of counters
SELECT id, bdr.crdt_gcounter_to_text(cnt) FROM crdt_test;
```

成長のみの合計（crdt_gsum）

- 非負の値を持つインクリメントのみをサポートします（sum + numeric ）。
- # 演算子を使用するか、numeric にキャストして、合計の現在の値を取得できます。
- 新しい値がアプリケーションのどこかで計算されるときの一般的なパターンである sum = value のような単純な割り当てと互換性がありません。
- ! 演算子（sum = !sum ）を使用して、合計の単純なリセットを許可します。
- crdt_gsum_to_text を使用して内部状態を検査できます。

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    gsum     bdr.crdt_gsum NOT NULL DEFAULT 0.0
);

INSERT INTO crdt_test VALUES (1, 0.0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- error: negative value

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment sum
UPDATE crdt_test SET gsum = gsum + 11.5 WHERE id = 1;
UPDATE crdt_test SET gsum = gsum + 120.33 WHERE id = 2;

- - error: minus operator not defined
UPDATE crdt_test SET gsum = gsum - 15.2 WHERE id = 1;

- - error: increment has to be non-negative
UPDATE crdt_test SET gsum = gsum + (-1.56) WHERE id = 1;

- - reset sum
UPDATE crdt_test SET gsum = !gsum WHERE id = 1;

- - get current sum value
SELECT id, gsum::numeric, gsum FROM crdt_test;

- - show internal structure of sums
SELECT id, bdr.crdt_gsum_to_text(gsum) FROM crdt_test;
```

ポジティブネガティブカウンター（crdt_pncounter）

- 正と負の両方の値でインクリメントをサポート（counter + int および counter + bigint オペレーターを介して）。
- # 演算子を使用するか、bigint にキャストすることにより、カウンターの現在の値を取得できます。
- 新しい値がアプリケーションのどこかで計算されるときの一般的なパターンである counter = value のような単純な割り当てと互換性がありません。
- ! 演算子（counter = !counter ）を使用して、カウンターの単純なリセットを許可します。
- crdt_pncounter_to_text を使用して内部状態を検査できます。

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_pncounter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

- - get current counter value
SELECT id, cnt::bigint, cnt FROM crdt_test;

- - show internal structure of counters
SELECT id, bdr.crdt_pncounter_to_text(cnt) FROM crdt_test;

- - reset counter
UPDATE crdt_test SET cnt = !cnt WHERE id = 1;

- - get current counter value after the reset
SELECT id, cnt::bigint, cnt FROM crdt_test;
```

正負の合計（crdt_pnsum）

- sum + numeric を介して正と負の両方の値を持つインクリメントをサポートします。
- then # 演算子を使用するか、numeric にキャストすることにより、合計の現在の値を取得できます。
- 新しい値がアプリケーションのどこかで計算されるときの一般的なパターンである sum = value のような単純な割り当てと互換性がありません。
- ! 演算子（sum = !sum ）を使用して、合計の単純なリセットを許可します。
- crdt_pnsum_to_text を使用して内部状態を検査できます。

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    pnsum    bdr.crdt_pnsum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 1298.24); -- initialized to 1298.24
INSERT INTO crdt_test VALUES (3, -45.31);  -- initialized to -45.31

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment sums
UPDATE crdt_test SET pnsum = pnsum + 1.44     WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum + 12.20    WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum + (-24.34) WHERE id = 3;

- - decrement sums
UPDATE crdt_test SET pnsum = pnsum - 7.3      WHERE id = 1;
UPDATE crdt_test SET pnsum = pnsum - 192.83   WHERE id = 2;
UPDATE crdt_test SET pnsum = pnsum - (-12.22) WHERE id = 3;

- - get current sum value
SELECT id, pnsum::numeric, pnsum FROM crdt_test;

- - show internal structure of sum
SELECT id, bdr.crdt_pnsum_to_text(pnsum) FROM crdt_test;

- - reset sum
UPDATE crdt_test SET pnsum = !pnsum WHERE id = 1;

- - get current sum value after the reset
SELECT id, pnsum::numeric, pnsum FROM crdt_test;
```

デルタカウンター（crdt_delta_counter）

- bigint ドメインとして定義されているため、bigint 列とまったく同じように機能します。
- 正負両方の値のインクリメントをサポートします。
- 新しい値がアプリケーションのどこかで計算されるときに一般的な counter = value のような単純な割り当てと互換性があります。
- 値を確実にリセットする簡単な方法はありません。

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    cnt      bdr.crdt_delta_counter NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);      -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4531);  -- initialized to -4531

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET cnt = cnt + 1      WHERE id = 1;
UPDATE crdt_test SET cnt = cnt + 120    WHERE id = 2;
UPDATE crdt_test SET cnt = cnt + (-244) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET cnt = cnt - 73    WHERE id = 1;
UPDATE crdt_test SET cnt = cnt - 19283 WHERE id = 2;
UPDATE crdt_test SET cnt = cnt - (-12) WHERE id = 3;

- - get current counter value
SELECT id, cnt FROM crdt_test;
```

デルタサム（crdt_delta_sum）

- numeric ドメインとして定義されるため、numeric 列とまったく同じように機能します。
- 正負両方の値のインクリメントをサポートします。
- 新しい値がアプリケーションのどこかで計算されるときに一般的な sum = value のような単純な割り当てと互換性があります。
- 値を確実にリセットする簡単な方法はありません。

```
CREATE TABLE crdt_test (
    id       INT PRIMARY KEY,
    dsum     bdr.crdt_delta_sum NOT NULL DEFAULT 0
);

INSERT INTO crdt_test VALUES (1, 0);       -- initialized to 0
INSERT INTO crdt_test VALUES (2, 129.824); -- initialized to 129824
INSERT INTO crdt_test VALUES (3, -4.531);  -- initialized to -4531

- - enable CLCD on the table
ALTER TABLE crdt_test REPLICA IDENTITY FULL;
SELECT bdr.alter_table_conflict_detection(crdt_test, column_modify_timestamp, cts);

- - increment counters
UPDATE crdt_test SET dsum = dsum + 1.32   WHERE id = 1;
UPDATE crdt_test SET dsum = dsum + 12.01  WHERE id = 2;
UPDATE crdt_test SET dsum = dsum + (-2.4) WHERE id = 3;

- - decrement counters
UPDATE crdt_test SET dsum = dsum - 7.33   WHERE id = 1;
UPDATE crdt_test SET dsum = dsum - 19.83  WHERE id = 2;
UPDATE crdt_test SET dsum = dsum - (-1.2) WHERE id = 3;

- - get current counter value
SELECT id, cnt FROM crdt_test;
```

[1] https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

8.8.5 Eager conflict resolution

積極的な競合解決（ 積極的な Eager Replication とも呼ばれます）は、 COMMIT 決定プロセス中にシリアル化可能なエラーで互いに競合するトランザクションを中止することにより、競合を防ぎます。

> *Group Commit* の競合解決オプションの 1 つとして *commit* スコープへの移行

を使用して構成します。

使用法

Eager 競合解決を有効にするには、クライアントはコミットスコープに切り替える必要があります。これは、次に示すように、セッションレベルまたは個々のトランザクションで使用します。

```
BEGIN;

SET LOCAL bdr.commit_scope = eager_scope;

... other commands possible...
```

クライアントは、トランザクションの最後に COMMIT の発行を継続し、PGD に 2 つのフェーズを管理させることができます。

```
COMMIT;
```

この場合、 eager_scope コミットスコープは次のように定義されます。

```
SELECT bdr.add_commit_scope(
    commit_scope_name := eager_scope,
    origin_node_group := top_group,
    rule := ALL (top_group) GROUP COMMIT (conflict_resolution = eager, commit_
↪decision = raft) ABORT ON (timeout = 60s),
    wait_for_ready := true
);
```

エラー処理

PGD がトランザクションを管理することを考えると、クライアントは COMMIT の結果のみを確認する必要があります。これは、シングルノードの Postgres を含む、どのような場合でもお勧めします。

起点ノードに障害が発生した場合、残りのノードは最終的に（少なくとも ABORT ON timeout の後）、グローバルに準備されたトランザクションをロールバックすることを決定します。Raft は、一貫性のないコミットとロールバックの決定を防ぎます。ただし、これには接続されたノードの大部分が必要です。切断されたノードは、必要に応じてトランザクションを最終的にコミット（またはロールバック）する準備を整え、決定してさらに進んだ可能性のあるノードの大部分と調整します。

一般的な積極的な Eager Replication の効果

コミットレイテンシーの増加

同期ステップを追加するということは、ノード間の通信が増えることを意味し、その結果、コミット時の待機時間が長くなります。Eager All-Node Replication は、約 2 つのネットワークラウンドトリップを追加します（最悪の場合、最も遠いピアノードへ）。ロジカルスタンバイノードと、まだ参加中またはキャッチアップ中のノードは含まれませんが、最終的に変更を受け取ります。

ピアノードがトランザクションのローカル準備を確認できる前に、ローカルに適用する必要もあります。これにより、トランザクションのサイズによっては、コミットレイテンシーがさらに増加します。この設定は、synchronous_commit 設定とは無関係です。

中止率の増加

シングルノード Postgres、またはデフォルトの非同期レプリケーションモードの PGD を使用しても、COMMIT 時のエラーはまれです。追加された同期ステップによりエラーのソースが追加されるため、アプリケーションはこのようなエラーを適切に処理する準備をする必要があります（通常は再試行ループを適用することにより）。

アボートの割合は、ワークロードのみに依存します。多くの行を変更する大規模なトランザクションは、他の同時実行トランザクションと競合する可能性がはるかに高くなります。

8.9 Parallel Apply

8.9.1 Parallel Apply とは何ですか？

パラレル適用は、PGD ノードがサブスクリプションごとに複数のライターを使用できるようにする PGD の機能です。通常、この動作により、サブスクリプションのスループットが向上し、レプリケーションのパフォーマンスが向上します。

パラレル適用が動作している場合、サブスクリプションからのトランザクションの変更は、複数のライターによって書き込まれます。ただし、各ライタは、トランザクションの最終コミットがオリジンノードで実行されるコミット順序に違反しないことを保証します。違反がある場合、エラーが生成され、トランザクションをロールバックできます。

このメカニズムは、以前は、トランザクションがコミットを保留しており、別のトランザクションが変更する必要がある行を変更し、保留中のトランザクションが実行する前に起点ノードで他のトランザクションが実行された場合、結果のエラーがデッドロックとして現れる可能性があることを意味していました。保留中のトランザクションがロック要求を取り出しました。

さらに、エラーを処理すると、デッドロックの検出にかかる時間、クライアントがトランザクションのロールバックにかかる時間、既に適用されている変更の間接ガベージコレクションにかかる時間、および時間の組み合わせにより、レプリケーションラグが増加する可能性があります。仕事のやり直しに。

これが、PGD 5.2 で導入された Parallel Apply のデッドロック緩和が役立つ場所です。任意のトランザクションについて、パラレル適用は今後の書き込みを確認し、潜在的な競合がある場合は、競合がなくなるまでトランザクションを保留します。

Parallel Apply の構成

PGD 5 のパラレル適用を制御する 2 つの変数、 *bdr.max_writers_per_subscription* （デフォルトは 8）および *bdr.writers_per_subscription* （デフォルトは 2)。

```
bdr.max_writers_per_subscription = 8
bdr.writers_per_subscription = 2
```

この構成では、各サブスクリプションに 2 つのライターが提供されます。ただし、状況によっては、システムがサブスクリプションに最大 8 つのライターを割り当てる場合があります。

> *bdr.max_writers_per_subscription* は、サーバーの再起動でのみ変更できます。

次の方法で、再起動せずに特定のサブスクリプションの *bdr.writers_per_subscription* を変更できます。

1. *bdr.alter_subscription_disable* を使用してサブスクリプションを停止します。

1. 新しい値を設定します。

1. *bdr.alter_subscription_enable* を使用してサブスクリプションを再開します。

ただし、最初に、 `select * from bdr.subscription` を使用してサブスクリプションの名前を確立します。この例では、サブスクリプション名は `bdr_bdrdb_bdrgroup_node2_node1` です。

```
SELECT bdr.alter_subscription_disable (bdr_bdrdb_bdrgroup_node2_node1);

UPDATE bdr.subscription
SET num_writers = 4
WHERE sub_name = bdr_bdrdb_bdrgroup_node2_node1;

SELECT bdr.alter_subscription_enable (bdr_bdrdb_bdrgroup_node2_node1);
```

Parallel Apply をいつ使用するか

パラレル適用はデフォルトで常にオンになっており、ほとんどの操作で、オンのままにすることをお勧めします。

PGD 5.2 から、 Parallel Apply は CAMO で動作します。 Group Commit または Eager Replication を使用している場合は無効にする必要があります。

PGD 5.1 以前は、グループコミット、 CAMO 、および一括レプリケーションでパラレル適用を使用しないでください。これらのシナリオでは、Parallel Apply を無効にする必要があります。 PGD 5.1 以前を使用しているときに多数のデッドロックが発生している場合は、パラレル適用を無効にするか、アップグレードを検討することもできます。

パラレル適用のモニタリング

Parallel Apply のデッドロック緩和をサポートするために、PGD 5.2 は
bdr.stat_subscription に列を追加します。新しい列は nprovisional_waits

、ntuple_waits 、および ncommmit_waits です。nprovisional_waits 値は、同時適用トランザクションによって実行されている同じタプルに対する操作の数を反映します。これらは暫定的な待機であり、実際にはまだ待機していませんが、待機できます。

適用操作がデッドロックの緩和策の影響を受ける場合、それらは安全に適用され、ntuple_waits でカウントされるまで待機します。コミットされる前に待機した完全に適用されたトランザクションは、ncommit_waits でカウントされます。

パラレル適用の無効化

Parallel Apply を無効にするには、 *bdr.writers_per_subscription* を 1 に設定します。

8.10 Replication sets

レプリケーションセットは、PGD ノードがサブスクライブできるテーブルのグループです。レプリケーションセットを使用して、各ノードが他のノードの正確なコピーである通常の対称マルチマスターよりも複雑なレプリケーショントポロジを作成できます。

すべての PGD グループは、グループと同じ名前の複製セットを作成します。このレプリケーションセットはデフォルトのレプリケーションセットであり、すべてのユーザーテーブルと DDL レプリケーションに使用されます。すべてのノードがサブスクライブしています。言い換えれば、デフォルトでは、すべてのユーザーテーブルがすべてのノード間で複製されます。

8.10.1 レプリケーションセットの使用

bdr.create_replication_set

を使用して、挿入、更新、削除、または切り捨てのアクションを含めるかどうかを指定して、レプリケーションセットを作成できます。1 つのオプションでは既存のテーブルをセットに追加でき、2 番目のオプションではテーブルの作成時にテーブルを追加するかどうかを定義します。

レプリケーションセットに追加または削除するテーブルを手動で定義することもできます。

レプリケーションセットに含まれるテーブルは、ノードがクラスターに参加したとき、およびその後に維持されます。

ノードが参加したら、レプリケーションセットからテーブルを削除できますが、再同期操作を使用して新しいテーブルを追加する必要があります。

デフォルトでは、新しく定義されたレプリケーションセットは DDL または PGD 管理ファンクションコールをレプリケートしません。
bdr.replication_set_add_ddl_filter を使用

複製するコマンドを定義します。

PGD は、すべてのノードで複製セット定義を作成します。次に、 *bdr.alter_node_replication_sets* を使用して、各レプリケーションセットをパブリッシュまたはサブスクライブするように各ノードを定義できます。

関数を使用して、これらの定義を後で変更したり、複製セットを削除したりできます。

注釈: 選択的複製にデフォルトの複製セットを使用しないでください。クラスター内の PGD ノードでデフォルトのレプリケーションセットを削除または変更しないでください。デフォルトで DDL レプリケーションおよび管理ファンクション呼び出しにも使用されます。

8.10.2 パーティション分割テーブルの動作

PGD はパーティションテーブルを透過的にサポートします。つまり、パーティションテーブルをレプリケーションセットに追加できます。

いずれかのパーティションを含む変更は、ダウンストリームに複製されます。

注釈: パーティションがパーティションテーブルを介してレプリケートされる場合、パーティションで直接実行されるステートメントは、親テーブルで実行されるのと同じようにレプリケートされます。例外は `TRUNCATE` コマンドで、常に影響を受けるテーブルまたはパーティションのリストで複製されます。

個々のパーティションをレプリケーションセットに追加できます。その場合、それらは通常のテーブルのように、つまり、ダウンストリームのパーティションと同じ名前のテーブルに複製されます。パーティショニング定義がプロバイダーとサブスクライバーの両方で同じである場合、パーティショニングロジックを実行する必要がないため、この動作にはいくつかのパフォーマンス上の利点があります。

注釈: ルートパーティションテーブルがレプリケーションセットの一部である場合、個々のパーティションのメンバーシップは無視されます。そのルートテーブルのメンバーシップのみが考慮されます。

8.10.3 外部キーを使用した動作

外部キー制約は、参照元のテーブルの各行が参照先のテーブルの行と一致することを保証します。したがって、参照元のテーブルがレプリケーションセットのメンバーである場合、参照されるテーブルも同じレプリケーションセットのメンバーである必要があります。

PGD の現在のバージョンは、この条件を自動的にチェックまたは強制しません。レプリケーションセットにテーブルを追加するとき、データベース管理者は、外部キーによって参照されるすべてのテーブルも追加されることを確認する必要があります。

次のクエリを使用して、この要件を満たさないすべての外部キーとレプリケーションセットをリストできます。参照元のテーブルはレプリケーションセットのメンバーですが、参照先のテーブルはそうではありません。

```
SELECT t1.relname,
       t1.nspname,
       fk.conname,
       t1.set_name
  FROM bdr.tables AS t1
  JOIN pg_catalog.pg_constraint AS fk
    ON fk.conrelid = t1.relid
   AND fk.contype = f
 WHERE NOT EXISTS (
  SELECT *
    FROM bdr.tables AS t2
   WHERE t2.relid = fk.confrelid
     AND t2.set_name = t1.set_name
);
```

このクエリの出力は次のようになります。

```
 relname | nspname |  conname  | set_name
- --------+---------+-----------+----------
 t2      | public  | t2_x_fkey | s2
(1 row)
```

この出力は、テーブル t2 はレプリケーションセット s2 のメンバーですが、外部キー t2_x_fkey によって参照されるテーブルはそうではないことを意味します。

TRUNCATE CASCADE コマンドは、コマンドを複製する前に複製セットのメンバーシップを考慮します。例：

```
TRUNCATE table1 CASCADE;
```

これは、レプリケーションセットのみの一部であるすべてのテーブルでカスケードのない TRUNCATE になります。

```
TRUNCATE table1, referencing_table1, referencing_table2 ...
```

8.10.4 レプリケーションセットのメンバーシップ

1 つ以上のレプリケーションセットにテーブルを追加したり、テーブルを削除したりできます。これを行うと、それらのテーブルの変更のみのレプリケーション（DML）に影響します。スキーマの変更（DDL）は、DDL レプリケーションセットフィルターによって処理されます（ *DDL replication filtering* を参照）。

レプリケーションは、ノードレプリケーションセット構成を持つレプリケーションセットのテーブルメンバーシップを使用して、どのノードにレプリケートするアクションを決定します。決定は、すべてのメンバーシップとレプリケーションセットオプションの和集合を使用して行われます。テーブルが、INSERT アクションのみをレプリケートするレプリケーションセット A と、UPDATE アクションのみをレプリケートするレプリケーションセット B のメンバーであるとします。ターゲットノードがレプリケーションセッ

ト A と B の両方にサブスクライブしている場合、INSERT アクションと UPDATE アクションの両方がレプリケートされます。

bdr.replication_set_add_table および *bdr.replication_set_remove_table*

を使用してメンバーシップを制御できます。

8.10.5 レプリケーションセットのリスト

次のクエリを使用して、既存のレプリケーションセットをリストできます。

```
SELECT set_name
FROM bdr.replication_sets;
```

このクエリを使用して、特定のレプリケーションセット内のすべてのテーブルをリストできます。

```
  SELECT nspname, relname
  FROM bdr.tables
  WHERE set_name = myrepset;

:ref:`外部キーを使用した動作 <外部キーを使用した動作>` は、参照先テーブルが参照元テーブルと同じレプリケーションセットに含まれていないすべての外部キーをリストするクエリを示しています。
```

次の SQL を使用して、現在のノードがパブリッシュおよびサブスクライブするレプリケーションセットを表示します。

```
SELECT node_id,
       node_name,
       pub_repsets,
       sub_repsets
  FROM bdr.local_node_summary;
```

このコードは、次のような出力を生成します。

```
  node_id   | node_name |    pub_repsets     |   sub_repsets
- -----------+-----------+--------------------------------------
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
(1 row)
```

クラスター内のすべてのノードに対して同じクエリを実行するには、次のクエリを使用できます。このアプローチは、すべてのノードに関連付けられたレプリケーションセットを同時に取得します。

```
WITH node_repsets AS (
  SELECT jsonb_array_elements(
    bdr.run_on_all_nodes($$
        SELECT
          node_id,
```

(次のページに続く)

(前のページからの続き)

```
          node_name,
          pub_repsets,
          sub_repsets
        FROM bdr.local_node_summary;
    $$)::jsonb
  ) AS j
)
SELECT j->response->command_tuples->0->>node_id AS node_id,
       j->response->command_tuples->0->>node_name AS node_name,
       j->response->command_tuples->0->>pub_repsets AS pub_repsets,
       j->response->command_tuples->0->>sub_repsets AS sub_repsets
FROM node_repsets;
```

これは、たとえば次のことを示しています。

```
  node_id   | node_name |    pub_repsets     |    sub_repsets
- ----------+-----------+--------------------------------------
 933864801  | s02db01   | {bdrglobal,bdrs02} | {bdrglobal,bdrs02}
 1834550102 | s01db01   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
 3898940082 | s01db02   | {bdrglobal,bdrs01} | {bdrglobal,bdrs01}
 1102086297 | s02db02   | {bdrglobal,bdrs02} | {bdrglobal,bdrs02}
(4 rows)
```

8.10.6 DDL レプリケーションのフィルタリング

デフォルトでは、サポートされているすべての DDL のレプリケーションは、デフォルトの PGD グループレプリケーションセットを介して行われます。このレプリケーションは、PGD グループと同じ名前の DDL フィルターを使用して実現されます。このフィルターは、PGD グループが作成されるときに、デフォルトの PGD グループレプリケーションセットに追加されます。

これを調整するには、既存のすべてのレプリケーションセットの DDL レプリケーションフィルターを変更します。これらのフィルターは、レプリケーションセットのテーブルメンバーシップとは無関係です。データの変更と同じように、各 DDL ステートメントは、複数のレプリケーションセットの複数のフィルターに一致した場合でも、1 回だけレプリケートされます。

次のクエリを使用して、既存の DDL フィルターをリストできます。フィルターごとに、コマンドタグとロール名に適用される正規表現を示します。

```
  SELECT * FROM bdr.ddl_replication;

:ref:`bdr.replication_set_add_ddl_filter <bdr.replication_set_add_ddl_filter>` および␣
↪:ref:`bdr.replication_set_remove_ddl_filter <bdr.replication_set_remove_ddl_filter>
↪` を使用して、
```

DDL フィルターを操作できます。それらは DDL と見なされるため、DDL レプリケーションとグローバルロックの対象となります。

8.11 Application connection management

アプリケーション接続の管理は、高可用性の重要な部分です。

特に非同期レプリケーションでは、競合を回避し、アプリケーションの可用性を保証するために、一貫した書き込みリードノードを持つことが重要です。

EDB Postgres Distributed は、PGD プロキシと呼ばれるプロキシレイヤーを提供します。これは、通常、高可用性構成（リージョンごとに少なくとも 2 つのインスタンス）でインストールされます。

PGD プロキシは、 EDB Postgres 分散ノードの 1 つに接続し、 EDB Postgres 分散クラスターによって決定されたルーティング構成の変更を監視します。これにより、接続が一貫して正しいノードにルーティングされます。

8.11.1 構成

ルーティングの構成は、SQL インターフェイスまたは PGD-CLI のいずれかを介して行われます。

`bdr.alter_node_group_option()` 関数を呼び出すことで、ルーティングの決定を有効にできます。例：

```
SELECT bdr.alter_node_group_option(region1-group, enable_proxy_routing, true)
```

同じオプションを `false` に設定することで無効にできます。

追加のグループレベルのオプションは、ルーティングの決定に影響します。

- `route_writer_max_lag` —書き込みリーダーとして選択される新しい書き込み候補の最大ラグ（バイト単位）。これに合格する候補者がない場合、ライタは自動的に選択されません。
- `route_reader_max_lag` —ノードが実行可能な読み取り専用ノードと見なされるための最大ラグ（バイト単位）。現在、将来の使用のために予約されています。

ルーティングのノードごとの構成は、 `bdr.alter_node_option()` を使用して設定されます。ルーティングに影響する使用可能なオプションは次のとおりです。

- `route_dsn` —このノードに接続するためにプロキシが使用する dsn。
- `route_priority` —同じノードグループ内の他のノードに対するノードの相対的なルーティング優先度。
- `route_fence` —ノードがルーティングからフェンスされているかどうか、つまり、PGD プロキシから接続を受信できないかどうか。
- `route_writes` —書き込みをこのノードにルーティングできるかどうか、つまり、ノードが書き込みリーダーになれるかどうか。
- `route_reads` —読み取り専用接続をこのノードにルーティングできるかどうか。現在、将来の使用のために予約されています。

SQL インターフェイスを使用してプロキシを構成することもできます。bdr.create_proxy を使用してプロキシ構成を追加できます。たとえば、SELECT bdr.create_proxy('region1-proxy1', 'region1-group'); は、PGD グループ region1-group のメンバーである region1-proxy1 という名前のプロキシのデフォルト構成を追加します。ここで指定されたプロキシの名前は、プロキシ構成ファイルで指定された名前と同じである必要があります。SELECT bdr.drop_proxy('region1-proxy1') を使用してプロキシ構成を削除できます。その結果、プロキシは非アクティブ化されます。

bdr.alter_proxy_option() 関数を使用して、各プロキシのオプションを構成できます。使用可能なオプションは次のとおりです。

- listen_address —リッスンするプロキシのアドレス。
- listen_port —プロキシがリッスンするポート。
- max_client_conn —プロキシが受け入れる最大接続数。
- max_server_conn —プロキシが Postgres ノードに作成できる接続の最大数。
- server_conn_timeout —サーバー接続の接続タイムアウト。
- server_conn_keepalive —サーバー接続のキープアライブ間隔。
- consensus_grace_period —Raft リーダーを失ってもプロキシがルーティングを継続する期間。 0s に設定されている場合、プロキシはルーティングをすぐに停止します。

すべてのグループの現在の構成は、bdr.node_group_routing_config_summary ビューに表示されます。同様に、bdr.node_routing_config_summary ビューには、現在のノードごとのルーティング構成が表示されます。bdr.proxy_config_summary は、プロキシごとの構成を示しています。

スイッチオーバー操作を実行して、書き込みリーダーであるノードを明示的に変更することもできます。これを行うには、bdr.routing_leadership_transfer() 関数を使用します。

EDB Postgres Distributed Proxy

EDB Postgres 分散プロキシは、クライアントアプリケーションと Postgres 間の抽象化レイヤーとして機能するデーモンです。PGD コンセンサスメカニズムとインターフェイスして、現在の書き込みリーダーノードの ID を取得し、トラフィックをそのノードにリダイレクトします。

PGD クラスターには、常に少なくとも 1 つのグローバルグループと 1 つのデータグループがあります。PGD は、 enable_proxy_routing および enable_raft オプションが true に設定されている各データグループの書き込みリーダーを選択します。プロキシをグローバルグループまたはデータグループにアタッチできます。各グループに複数のプロキシをアタッチできます。

PGD プロキシは、TCP レイヤー 4 プロキシです。

仕組み

起動時に、PGD プロキシはローカル構成ファイルで指定されたエンドポイントのいずれかに接続します。それはフェッチします：

・全ノードの DB 接続情報

- リッスンアドレス、リッスンポートなどのプロキシオプション
- 現在の書き込みリーダーなどのルーティングの詳細

構成ファイルで指定されたエンドポイントは、起動時にのみ使用されます。その後、実際のエンドポイントは `bdr.node_routing_config_summary` の PGD カタログの `route_dsn` フィールドから取得されます。

PGD は書き込みリーダーの選択を管理します。PGD プロキシは PGD と対話して、Postgres 通知/リッスンチャネルで書き込みリーダー変更イベント通知を取得し、クライアントトラフィックを現在の書き込みリーダーにルーティングします。PGD プロキシは、書き込みリーダーの変更時、または書き込みリーダーが利用できない場合、既存のすべてのクライアント接続を切断します。リーダー選挙の書き込みは Raft が支援するアクティビティであり、Raft リーダーの可用性の対象となります。書き込みリーダーが利用できない場合、PGD プロキシは新しいクライアント接続を閉じます。

PGD プロキシは、書き込みリーダー変更イベントに応答します。これは、**フェールオーバー** と **スイッチオーバー** の 2 つのモードに分類できます。

Postgres またはオペレーティングシステムがクラッシュした場合に、現在の書き込みリーダーノードから新しいノードへの書き込みリーダーシップの自動転送は、フェールオーバーと呼ばれます。PGD は、現在の書き込みリーダーがダウンするか応答しなくなると、新しい書き込みリーダーを選択します。新しい書き込みリーダーが PGD によって選択されると、PGD プロキシは古い書き込みリーダーへの既存のクライアント接続を閉じ、新しいクライアント接続を新しく選択された書き込みリーダーにリダイレクトします。

現在の書き込みリーダーから新しいターゲットリーダーへの書き込みリーダーシップのユーザー制御の手動転送は、スイッチオーバーと呼ばれます。スイッチオーバーは、
: *PGD CLI switchover* コマンドを介してトリガーされます。コマンドは PGD に送信され、PGD は指定されたターゲットノードを新しい書き込みリーダーとして選択しようとします。フェールオーバーと同様に、PGD

Proxy は既存のクライアント接続を閉じ、新しいクライアント接続を新しく選択された書き込みリーダーにリダイレクトします。これは、サーバーのメンテナンス中に、たとえば、サーバーの更新や OS パッチの更新などのメンテナンスのために現在の書き込みリーダーノードを停止する必要がある場合に役立ちます。

コンセンサス猶予期間

PGD プロキシは、Raft リーダーを失ったときのルーティング動作を構成するために使用できるコンセンサス_grace_period プロキシオプションを提供します。PGD プロキシは、この間、現在の書き込みリーダー（使用可能な場合）にルーティングし続けます。この期間に新しい Raft リーダーが選出されない場合、プロキシはルーティングを停止します。`0s` に設定されている場合、PGD プロキシはルーティングをすぐに停止します。

このオプションの主な目的は、ラフトリーダーが失われたときの書き込み動作をユーザーが構成できるようにすることです。Raft リーダーがクラスターに存在しない場合、プロキシによって見られる現在の書き込みリーダーが正しいものであるとは必ずしも保証されません。場合によっては、次の例のネットワーク

パーティションのように、2 つの書き込みリーダーが同じグループに接続されている 2 つの異なるプロキシによって認識され、書き込み競合の可能性が高まる可能性があります。これが目的の動作でない場合は、前述の `consensus_grace_period` を 0 に設定できます。この設定は、ラフトリーダーが失われたことを検出するとすぐに、ルーティングを停止し、既存の開いている接続を閉じるようにプロキシを構成します。

ネットワークパーティションの例

各データノードにプロキシがある 3 データノードグループを検討します。この場合、現在の書き込みリーダーがネットワークパーティション分割または分離されると、マジョリティパーティションに存在するデータノードが新しい書き込みリーダーを選択します。`consensus_grace_period` がゼロ以外の値、たとえば `10s` に設定されている場合、前の書き込みリーダーに存在するプロキシは、この期間中、書き込みのルーティングを継続します。

この場合、猶予期間が長すぎると、2 つの書き込みリーダーで書き込みが引き続き発生します。この条件により、書き込みの競合が発生する可能性が高くなります。

とはいえ、ほとんどの場合、現在の Raft リーダーが失われると、半分以上のノード（クォーラム）がまだ稼働している場合、新しい Raft リーダーは数秒以内に BDR によって選出されます。したがって、Raft リーダーはダウンしているが書き込みリーダーはまだ稼働している場合、`consensus_grace_period` をゼロ以外の正の値に維持することにより、ルーティングを許可するようにプロキシを構成できます。プロキシは、ルーティングを停止する前に、この期間中に Raft リーダーが選出されるのを待ちます。これは、可用性がより重要な場合に役立つ場合があります。

マルチホスト接続文字列

PostgreSQL C クライアントライブラリ（libpq）を使用すると、単純なフェールオーバーのために、単一の接続文字列で複数のホスト名を指定できます。これは、他のいくつかのプログラミング言語のクライアントライブラリ（ドライバー）でもサポートされています。ダウンまたはアクセスできない PGD プロキシインスタンス全体のフェールオーバーに適しています。

ただし、PGD プロキシインスタンスはアクセス可能だが書き込みリーダーへのアクセス権がない場合、または特定のインスタンスの書き込みリーダーが存在しない場合（つまり、特定の PGD グループの書き込みリーダーがないため）、接続単に失敗します。マルチホスト接続文字列内の他のホストは試行されません。この動作は、HAProxy や pgbouncer などの他のプロキシを使用した PostgreSQL クライアントライブラリの動作と一貫しています。

PGD プロキシの管理

PGD CLI には、 `create-proxy` 、 `delete-proxy` 、 `set-proxy-options` 、 `show-proxies` など、PGD クラスター内のプロキシを管理するためのいくつかのコマンドが用意されています。詳細については、 *PGD CLI の使用* を参照してください。

PGD プロキシの構成と管理の PGD 側の詳細については、
接続管理 を参照してください。

プロキシヘルスチェック

PGD プロキシは、次の HTTP(s) ヘルスチェック API エンドポイントを提供します。API エンドポイントは、 GET リクエストに応答します。使用する前に有効にして構成する必要があります。これを行う方法については、 競合ログ構成のリスト を参照してください。

```
GET /health/is-ready
GET /health/is-live
```

準備

プロキシは、有効な「GET」リクエストを受信すると、現在の書き込みリーダーに接続を正常にルーティングできるかどうかを確認します。チェックが正常に終了した場合、API は true を含む本文と HTTP ステータスコード 200 (OK) で応答します。それ以外の場合、HTTP ステータスコード 500 (Internal Server Error) とともに false を含む本文を返します。

活気

ライブネスチェックは、HTTP ステータスコード 200 (OK) またはエラーで true を返しますが、false は返しません。これは、プロキシサービスの開始または終了に失敗すると、要求をリッスンしている HTTP サーバーが自動的に停止するためです。

プロキシログの場所

syslog

- Debian ベース- /var/log/syslog
- Red Hat ベース- /var/log/messages

journalctl コマンドを使用して、プロキシのトラブルシューティング用にログをフィルタリングおよび表示します。以下は、クイックリファレンスのためのいくつかのサンプルコマンドです。

```
journalctl -u pgd-proxy -n100 -f
journalctl -u pgd-proxy --since today
journalctl -u pgd-proxy --since "10 min ago"
journalctl -u pgd-proxy --since "2022-10-20 16:21:50" --until "2022-10-20 16:21:55"
```

Installing PGD Proxy

PGD プロキシのインストール

2 つの方法を使用して、PGD プロキシをインストールおよび構成して、 EDB Postgres 分散クラスターを管理できます。PGD プロキシをインストールして構成するための推奨される方法は、クラスターの展開と管理に EDB Trusted Postgres Architect（TPA）ユーティリティを使用することです。

TPA を介したインストール

PGD クラスターが TPA を介して展開されている場合、TPA は推奨アーキテクチャに従って PGD プロキシを自動的にインストールおよび構成します。PGD クラスター内の他のノードに PGD プロキシをインストールする場合は、TPA 構成ファイルでそのインスタンスに pgd-proxy ロールをアタッチする必要があります。また、この例に示すように、展開する前に bdr_child_group パラメーターを設定します。詳細については、 信頼できる *Postgres* アーキテクト を参照してください。

```
- Name: proxy-a1
  location: a
  node: 4
  role:
  - pgd-proxy
  vars:
    bdr_child_group: group_a
  volumes:
  - device_name: /dev/sdf
    volume_type: none
```

構成

PGD プロキシは、プロキシオプションの取得や書き込みリーダーの詳細の取得など、内部操作のために PGD データベースに接続します。したがって、PGD ノードに接続するにはエンドポイント/ DSN のリストが必要です。PGD プロキシは、ローカル構成ファイル pgd-proxy-config.yml でこれらの構成を想定しています。以下は、 pgd-proxy-config.yml ファイルの作業例です。

```
log-level: debug
cluster:
  name: cluster-name
  endpoints:
    - "host=bdr-a1 port=5432 dbname=bdrdb user=pgdproxy "
    - "host=bdr-a3 port=5432 dbname=bdrdb user=pgdproxy "
    - "host=bdr-a2 port=5432 dbname=bdrdb user=pgdproxy "
  proxy:
    name: "proxy-a1"
```

デフォルトでは、TPA で作成されたクラスターでは、 pgd-proxy-config.yml は /etc/edb/pgd-proxy ディレクトリにあります。PGD プロキシは、次の場所で pgd-proxy-config.yml を検索します。優先順位は高いものから低いものです。

1. /etc/edb/pgd-proxy （デフォルト） 2. $HOME/.edb/pgd-proxy

ファイルの名前を変更するか、別の場所に移動する場合は、サービスの開始時にオプションの-f または--config-file フラグを使用して、新しい名前と場所を指定します。

sample service file を参照してください。

サンプル構成に示すように、トップレベルの構成パラメーター log-level を使用して、PGD プロキシサービスのログレベルを設定できます。log-level の有効な値は、debug 、info 、warn 、および error です。

cluster.endpoints および cluster.proxy.name は、構成ファイルの必須フィールドです。PGD プロキシは、常にリスト内の最初のエンドポイントに接続しようとします。失敗した場合、次のエンドポイントを試行します。

PGD プロキシは、プロキシの起動時にのみローカル構成ファイルで指定されたエンドポイントを使用します。その後、PGD プロキシは PGD プロキシカタログから実際のエンドポイント（route_dsn）のリストを取得します。したがって、ノードオプション route_dsn を PGD プロキシノードごとに設定する必要があります。詳細については、 *route_dsn* を参照してください。

ヘルスチェックの構成

PGD プロキシは *HTTP(S) health check APIs* を提供します。ヘルスチェックが必要な場合、以下の構成パラメーターを pgd-proxy 構成ファイルに追加することにより、それらを有効にできます。デフォルトでは無効になっています。

```
cluster:
  name: cluster-name
  endpoints:
    - "host=bdr-a1 port=5432 dbname=bdrdb user=pgdproxy "
    - "host=bdr-a3 port=5432 dbname=bdrdb user=pgdproxy "
    - "host=bdr-a2 port=5432 dbname=bdrdb user=pgdproxy "
  proxy:
    name: "proxy-a1"
    endpoint: "host=proxy-a1 port=6432 dbname=bdrdb user=pgdproxy "
    http:
      enable: true
      host: "0.0.0.0"
      port: 8080
      secure: false
      cert_file: ""
      key_file: ""
      probes:
        timeout: 10s
```

config cluster.proxy.http.enable: true を追加することで、API を有効にできます。有効にすると、HTTP サーバーはデフォルトのポート 8080 で 10 秒の timeout をリッスンし、HTTPS はサポートしません。

HTTPS を有効にするには、構成パラメーター cluster.proxy.http.secure: true を設定します。 true に設定されている場合、 cert_file および key_file も設定する必要があります。

cluster.proxy.endpoint は、プロキシがチェックの一環として現在の書き込みリーダーに接続するために使用するエンドポイントです。cluster.proxy.http.enable が true の場合、cluster.proxy.endpoint も設定する必要があります。ホストは listen_address 、ポートは listen_port *proxy options* 、 BDR ノード

routing_dsn

と同じになる可能性があります。必要に応じて、ユーザーはこのエンドポイントに sslmode 、sslrootcert 、user などの接続文字列パラメーターを追加できます。

PGD プロキシユーザー

エンドポイントで指定されたデータベースユーザーは、スーパーユーザーである必要はありません。通常、TPA 環境では、pgdproxy は OS ユーザーであり、 bdr_superuser ロールを持つデータベースユーザーでもあります。

PGD プロキシサービス

PGD Proxy を systemd サービスとして実行することをお勧めします。pgd-proxy サービスユニットファイルは、デフォルトで/etc/systemd/system/pgd-proxy.service にあります。以下は、TPA によって作成されたサンプルサービスファイルです。

```
[Unit]
Description=PGD Proxy

[Service]
Type=simple
User=pgdproxy
Group=pgdproxy
Restart=on-failure
RestartSec=1s
ExecStart=/usr/bin/pgd-proxy -f /etc/edb/pgd-proxy/pgd-proxy-config.yml
StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=pgd-proxy

[Install]
WantedBy=multi-user.target
```

次のコマンドを使用して、 pgd-proxy サービスを管理します。

```
systemctl status pgd-proxy
systemctl stop pgd-proxy
systemctl restart pgd-proxy
```

手動インストール

PGD リポジトリから入手可能な.deb および.rpm パッケージを使用して、任意の Linux マシンに PGD プロキシを手動でインストールできます。パッケージ名は edb-pgd5-proxy です。例：

sh # Debian の場合

sudo apt-get install edb-pgd5-proxy

Proxies, Raft, and Raft subgroups

PGD は、最上位グループが PGD インストール内のすべてのデータノードにまたがる Raft モデルを使用してメタデータを管理します。Raft リーダーはトップレベルグループによって選出され、トップレベルグループの状態をグループ内の他のすべてのノードに伝播します。

トップレベルグループの特定のオペレーションでは、Raft リーダーを確立し、接続する必要があります。これらの操作の例には、ノードの追加と削除、
ローカルシーケンスから *galloc* シーケンスへの変換 シーケンスの範囲の割り当てが含まれます。

また、トップレベルグループ内のノードの絶対多数（それらの半分と 1 つ）が互いに到達できる必要があることも意味します。したがって、5 つのノードを持つトップレベルグループでは、Raft リーダーを確立するには、少なくとも 3 つのノードが互いに到達可能である必要があります。

プロキシルーティング

Raft も使用する機能の 1 つはプロキシルーティングです。プロキシルーティングでは、プロキシがノードグループ内のデータノードへの書き込みを調整できる必要があります。このデータノードは書き込みリーダーです。書き込みリーダーがオフラインになった場合、接続されたアプリケーションの継続性を維持するために、プロキシはデータノードによって選択された新しい書き込みリーダーに切り替えられる必要があります。

PGD 5 ではノードグループごとにプロキシルーティングを構成できますが、推奨される構成は**グローバル**および **ローカル** ルーティングです。

グローバルルーティング

グローバルルーティングは、トップレベルグループを使用してプロキシルーティングを管理します。グループ内のすべての書き込み可能なデータノード（監視またはサブスクライブ専用ノードではない）は、すべてのプロキシの書き込みリーダーになる資格があります。トップレベルグループ内のプロキシへの接続は、トップレベルグループ内のデータノードにルーティングされます。

グローバルルーティングでは、トップレベルグループ全体に 1 つの書き込みリーダーしかありません。

ローカルルーティング

ローカルルーティングは、多くの場合場所にマッピングされるサブグループを使用して、サブグループ内のプロキシルーティングを管理します。ローカルルーティングは、書き込みを地理的に分離するためによく使用されます。トップレベルのコンセンサスが失われた場合でも、ルーティングを継続することが重要です。

これは、PGD では、トップレベルのコンセンサスが失われた場合でも、クエリと非同期データ操作（DML）を機能させるためです。ただし、グローバルルーティングの場合と同様に、トップレベルのコンセンサスを使用すると、そのコンセンサスが失われたときに新しい書き込みリーダーを選択できないことを意味します。ローカルグループは、独立したコンセンサスメカニズムとその複雑さを追加せずに、トップレベルのコンセンサスに頼ることはできません。

PGD 5 では、この問題にエレガントに対処するために、サブグループ Raft サポートが導入されました。サブグループラフトのサポートにより、PGD トップレベルグループのサブグループは、必要なリーダーを独立して選出できます。彼らは、他のサブグループまたはトップレベルの Raft コンセンサスとは無関係に書き込みリーダーを選択できる委任された Raft グループを形成することによってこれを行います。サブグループ内のプロキシへの接続は、サブグループ内のデータノードにルーティングされます。

ローカルルーティングでは、サブグループごとに書き込みリーダーがあります。

詳細情報

- *Raft subgroups and TPA* は、Trusted Postgres Architect でデプロイするときに PGD で Raft サブグループを有効にする方法を示しています。
- *Working with Raft subgroups and PGD CLI* は、PGD CLI が Raft サブグループの存在とステータスについてどのようにレポートするかを示します。
- *Migrating to Raft subgroups* は、既存のインストールを移行し、TPA なしで Raft サブグループを有効にするためのガイドです。
- *Raft elections in depth* は、Raft を使用して書き込みリーダーが選出される方法を詳細に調べます。

Creating Raft subgroups using TPA

TPAexec configure コマンドは、 --enable_proxy_routing local オプションが設定されている場合、Raft サブグループを有効にします。 TPA は、**ロケーション**という用語を使用して、物理/リージョンドメインにマッピングされるサブグループの一般的なユースケースを反映しています。構成が生成されると、指定されたロケーション名は、ロケーション名に基づいて、生成されたグループ名の下に保存されます。

TPA を使用して Raft サブグループを作成する

この例では、各場所に 3 つのデータノードを持つ 2 ロケーションクラスターを作成します。各場所のノードは、その場所の PGD Raft サブグループの一部です。

最上位グループの名前は pgdgroup です。

トップレベルグループには、 us_east と us_west の 2 つの場所があります。これらの場所は、2 つのサブグループ us_east_subgroup および us_west_subgroup にマップされます。

各ロケーションには 4 つのノードがあります。3 つのデータノードと 1 つの barman backup ノードです。3 つのデータノードは、PGD プロキシも共同ホストします。構成は次のように視覚化できます。

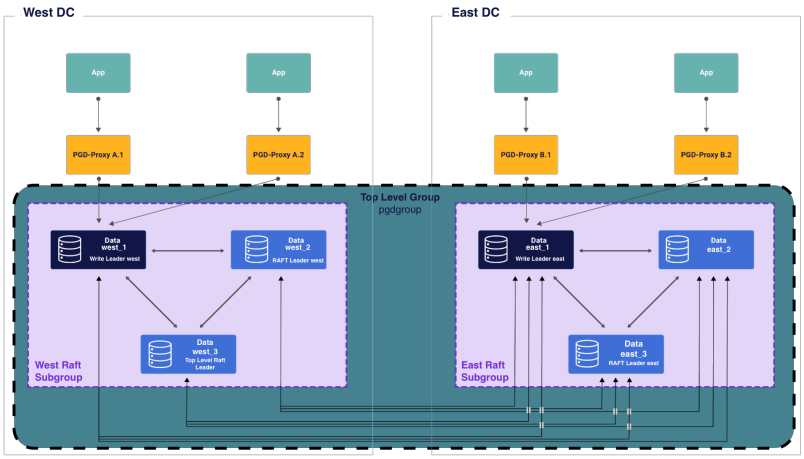


図 1: 6 Node Cluster with 2 Raft Subgroups

barman ノードはサブグループに参加せず、ひいては Raft グループにも参加しません。したがって、それらは表示されません。この図は、West Raft グループが west_1 を書き込みリーダーとして、west_2 を自分の Raft リーダーとして選択したクラスターの潜在的な状態のスナップショットです。東では、east_1 は書き込みリーダーであり、east_3 は Raft リーダーです。クラスター全体は、トップレベルの Raft グループ内に含まれます。そこでは、west_3 が現在 Raft リーダーです。

この構成を作成するには、次を実行します。

```
tpaexec configure pgdgroup --architecture PGD-Always-ON --location-names us_east us_
↪west --data-nodes-per-location 3 --epas 15 --no-redwood --enable_proxy_routing␣
↪local --hostnames-from hostnames.txt
```

hostnames.txt には以下が含まれます。

```
east1
east2
east3
eastbarman
west1
west2
west3
westbarman
```

構成ファイル

生成された config.yml ファイルには、トップレベルグループ pgdgroup と 2 つのサブグループ us_east_subgroup および us_west_subgroup を含む bdr_node_groups セクションがあります。これらの各サブグループには、ロケーションセット（us_east および us_west ）と true に設定されている他の 2 つのオプションがあります。

- enable_raft 、サブグループ内のサブグループ Raft をアクティブにします
- enable_proxy_routing 、pgd_proxy ルーターがトラフィックをサブグループの書き込みリーダーにルーティングできるようにします

サンプルの tpaexec コマンドによって生成された例を次に示します。

```
cluster_vars:
  apt_repository_list: []
  bdr_database: bdrdb
  bdr_node_group: pgdgroup
  bdr_node_groups:
  - name: pgdgroup
  - name: us_east_subgroup
    options:
      enable_proxy_routing: true
      enable_raft: true
      location: us_east
    parent_group_name: pgdgroup
  - name: us_west_subgroup
    options:
      enable_proxy_routing: true
      enable_raft: true
      location: us_west
    parent_group_name: pgdgroup
  bdr_version: 5
```

すべてのノードインスタンスには、インスタンスリストにエントリがあります。そのエントリでは、bdr_child_group が変数セクションに表示され、ノードが属するサブグループに設定されます。サンプルの tpaexec コマンドによって生成された例を次に示します。

```
instances:

- Name: east1
  backup: eastbarman
  location: us_east
  node: 1
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: us_east_subgroup
    bdr_node_options:
      route_priority: 100

- Name: east2
  location: us_east
  node: 2
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: us_east_subgroup
    bdr_node_options:
      route_priority: 100

- Name: east3
  location: us_east
  node: 3
  role:
  - bdr
  - pgd-proxy
  vars:
    bdr_child_group: us_east_subgroup
    bdr_node_options:
      route_priority: 100

- Name: eastbarman
  location: us_east
  node: 4
  role:
  - barman
```

bdr_child_group 設定がないこの場所の 1 つのノードは、Raft 意思決定プロセスに参加しないため、barman ノードです。

Working with Raft subgroups and PGD CLI

PGD でのテーブルアクセスメソッド（TAM）の使用

CLI コマンドを使用して、ノードとサブグループのステータスを表示できます。ここの例では、
Creating Raft subgroups with TPA で構成されたクラスターを想定しています。

pgd でノードを表示する

pgd コマンドは show-nodes です。

```
$pgd show-nodes
__OUTPUT__
Node     Node ID    Node Group Type Current State Target State Status Seq ID
- ---     -------    ---------- ---- ------------- ------------ ------ ------
east1    916860695  us_east    data ACTIVE        ACTIVE       Up     1
east2    2241077170 us_east    data ACTIVE        ACTIVE       Up     2
east3    1093023575 us_east    data ACTIVE        ACTIVE       Up     3
west1    1668243030 us_west    data ACTIVE        ACTIVE       Up     4
west2    2311995928 us_west    data ACTIVE        ACTIVE       Up     5
west3    4162758468 us_west    data ACTIVE        ACTIVE       Up     6
```

pgd でグループ（およびサブグループ）を表示する

PGD デプロイメント内のグループを名前と属性とともに表示するには、PGD CLI コマンド show-groups. を使用します

```
$pgd show-groups
Group    Group ID   Type   Parent Group Location Raft Routing Write Leader
- ----    --------   ----   ------------ -------- ---- ------- ------------
pgdgroup 1360502012 global                       true false
us_east  1806884964 data   pgdgroup     us_east  true true    east2
us_west  3098768667 data   pgdgroup     us_west  true true    west1
```

Migrating to Raft subgroups

実行中の PGD インストールに Raft サブグループを導入できます。

Raft サブグループへの移行（ SQL のみを使用）

Raft サブグループを既存のクラスターで有効にするには、次の構成手順が必要です。

- PGD クラスター内のすべてのノードのトップレベルグループを識別します。既存のクラスターには、すべてのノードが属する最上位グループが既にあります。

*場所ごとにサブグループを作成します。 bdr.create_node_group を、その値としてトップレベルグループを与える parent_group_name 引数とともに使用します。

- bdr.switch_node_group() を使用して、各場所の各ノードをその場所のサブグループに追加します。

*場所の各サブグループを変更して、グループの Raft を有効にします。bdr.alter_node_group_option() を使用し、 enable_raft オプションを true に設定します。

サブグループ Raft ノードグループを有効にする（ SQL のみを使用）

```
SELECT bdr.alter_node_group_option($group_name, enable_raft, true);
```

Raft elections in depth

PGD での書き込みリーダーの選択は、PGD の Raft メカニズムに依存しています。Raft メカニズムは、PGD の BDR Postgres 拡張機能の完全に内部であり、透過的に動作します。グループ内のノードは、グループのノード内に Raft リーダーを確立することから始まります。

ノードの相互作用

Raft リーダーが確立されると、リーダーはカタログを照会して、プロキシルーティングの書き込みリーダーが指定されたかどうかを確認します。

書き込みリーダーが指定されていない場合、ラフトリーダーは新しい書き込みリーダーを指定する手順を実行します。プロセスは、グループ内のすべてのノードを照会して状態を確立することから始まります。結果のノードのリストは、不適格なノード（たとえば、監視ノード）でフィルタリングされ、優先順位が付けられます。リストの最初/一番上のエントリーは、Raft ログの新しい書き込みリーダーとして設定されます。

プロキシ相互作用

すべてのプロキシは、最初にグループ内のデータノードに接続します。この動作により、現在の書き込みリーダーのカタログを照会し、そのノードへの接続のルーティングを開始できます。

それらはラフトリーダーに接続し、書き込みリーダーのカタログエントリへの変更をリッスンします。書き込みリーダーの変更が通知されると、ルーティングを再構成し、新しい書き込みリーダーに接続を送信します。

ノードとプロキシの相互作用の両方を次のシーケンス図に示します。2 つのノードと 1 つのプロキシが関与し、どのノードが書き込みリーダーであるかを調整し、プロキシはどのノードが書き込みリーダーであるかを学習するのを待っています。

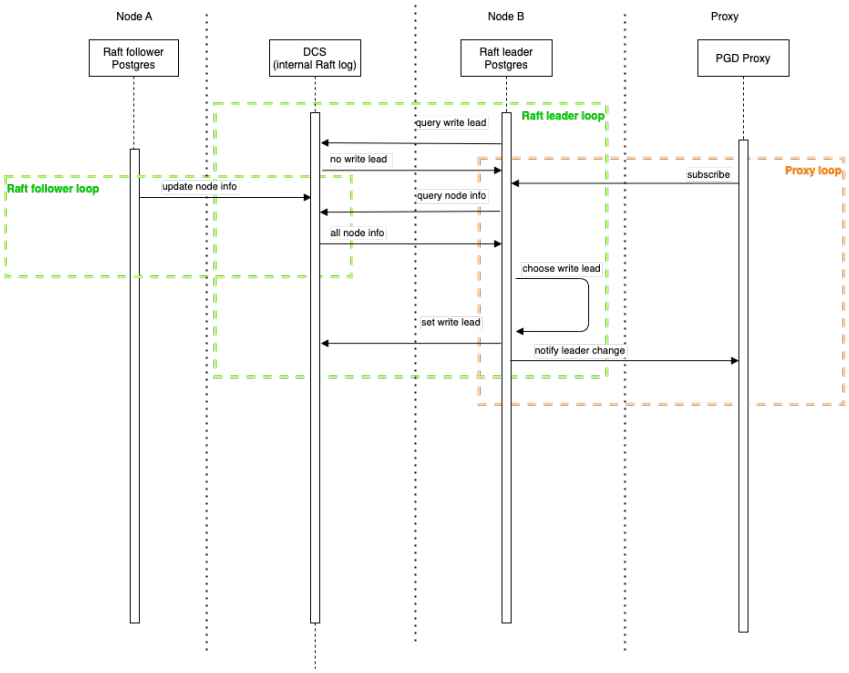


図 2: Sequence Diagram

8.12 Backup and recovery

PGD は、分散型の高可用性システムとして設計されています。クラスターの 1 つ以上のノードが失われた場合、それらを置き換える最善の方法は、残りのノードから新しいノードを直接複製することです。

PGD におけるバックアップとリカバリの役割は、次のような状況でのディザスターリカバリー（DR）を提供することです。

- クラスター内のすべてのノードの損失
- データ破損、アプリケーションエラー、またはセキュリティ侵害の結果としての、複数のノードにわたる重大で訂正不能なデータ破損

8.12.1 バックアップ

pg_dump

pg_dump は、通常は PGD で **論理バックアップ** とも呼ばれます。

pg_dump は、ローカルシーケンスとグローバルシーケンスの両方をローカルシーケンスであるかのようにダンプします。この動作は意図的なもので、PGD スキーマをダンプして他の PostgreSQL データベースに移植できるようにするためのものです。これは、シーケンスの種類のメタデータがダンプ時に失われることを

意味するため、復元により、復元時にすべてのシーケンスの種類が効果的に bdr.default_sequence_kind の値にリセットされます。

復元後のスクリプトを作成して、シーケンスごとに正確なシーケンスの種類をリセットするには、次のような SQL スクリプトを使用できます。

```
SELECT SELECT bdr.alter_sequence_set_kind(||
        nspname||.||relname||,||seqkind||);
FROM bdr.sequences
WHERE seqkind != local;
```

pg_dump が bdr.crdt_raw_value = on を使用して実行されている場合、bdr.crdt_raw_value = on でのみダンプをリロードできます。

テクニカルサポートは、PGD のバックアップと回復に物理バックアップ技術を使用することをお勧めします。

物理バックアップ

Barman などの標準の PostgreSQL ソフトウェアを使用して、 EDB

Postgres 分散クラスター内のノードの物理バックアップを取得できます。

PostgreSQL ノードに適用される同じ手順を使用して、PGD ノードの物理バックアップを実行できます。PGD ノードは、 BDR 拡張機能を実行している PostgreSQL ノードです。

PostgreSQL バックアップテクニックを PGD に適用する際には、次の特定の点を考慮してください。

- PGD は単一のデータベースのレベルで動作しますが、物理バックアップにはインスタンス内のすべてのデータベースが含まれます。データベースを計画して、簡単にバックアップと復元ができるようにします。
- バックアップは 1 つのノードのみのコピーを作成します。最も単純なケースでは、すべてのノードにすべてのデータのコピーがあるため、1 つのノードのみをバックアップしてすべてのデータをキャプチャする必要があります。ただし、その単一のコピーを含むサイトがダウンした場合、PGD の目標は達成されないため、最小はサイトごとに少なくとも 1 つのノードバックアップ（多数のコピーなど）です。
- ただし、各ノードに複製されていないローカルデータがある場合、または複製セットの定義が複雑なため、すべてのノードがすべての複製セットにサブスクライブしない場合があります。このような場合、バックアップ計画には、レプリケートされていないローカルデータをバックアップする方法の計画と、各レプリケーションセットにサブスクライブする少なくとも 1 つのノードのバックアップも含める必要があります。

最終的な一貫性

EDB Postgres 分散クラスター内のノードは **最終的に一貫性がある** ですが、**完全に一貫性がある**わけではありません。特定のノードの物理バックアップは、そのノードが実際に想定する状態に制限されたポイントインタイムリカバリ機能を提供します。

次の例は、同じ EDB Postgres 分散クラスター内の 2 つのノードがどのように同じ順序の状態を経ない（そして通常は経ない）かを示しています。

最初は S 状態である 2 つのノード N1 と N2 を持つクラスターを考えます。トランザクション W1 がノード N1 に適用され、同時に非競合のトランザクション W2 がノード N2 に適用された場合、ノード N1 は次の状態を経ます。

```
(N1)   S  -->  S + W1  -->  S + W1 + W2
```

ノード N2 は、次の状態を経ます。

```
(N2)   S  -->  S + W2  -->  S + W1 + W2
```

つまり、ノード N1 は状態 S + W2 を **決して**取りません。ノード N2 も同様に状態 S + W1 を取りません。ただし、両方のノードは同じ状態 S + W1 + W2 になります。この状況を考慮することは、バックアップ戦略の決定に影響を与える可能性があります。

ポイントインタイムリカバリ（PITR）

前の例では、変更が時間的に一貫していないことも示しました。 W1 と W2 は両方とも T1 時間に発生しますが、W1 の変更は T2 まで N2 に適用されません。

PostgreSQL PITR は、単一のマスターから COMMIT オーダーで変更が届くことを前提に設計されています。したがって、PITR は、特定の時点（PIT）に到達するまで変更をスキャンすることにより可能です。このスキームを使用すると、1 つのノードをその観点から単一の時点に復元できます。たとえば、T1 。ただし、その状態には、その時間近くにコミットされたがまだノードに到着していない他のノードからの他のデータは含まれません。その結果、リカバリーは部分的に一貫性がない、または少なくとも 1 つの複製起点についてのみ一貫していると考えられる場合があります。

これを要求するには、標準の構文を使用します。

```
recovery_target_time = T1
```

PGD は、複数のマスターからの変更を許可します。すべてが 1 つのノードの WAL ログに記録され、レプリケーションオリジン識別子を使用して個別に識別されます。

PGD は、特定の時点へのすべてまたは一部の複製起点の PITR を許可し、ノードのすべてのサブセットにわたって完全に一貫した視点を提供します。

したがって、マルチオリジンの場合、WAL ストリームは、複数のストリームがすべて 1 つの大きなストリームに混合されて含まれていると見ることができます。PIT はまだ 1 つしかありませんが、オリジンごとに異なるポイントとして到達します。

WAL ストリームは、要求されたオリジンが PIT を見つけるまで読み取られます。オリジンの PIT に到達した後、オリジンのコミット済みとしてマークされないトランザクションレコードを除いて、すべての変更はその時点までに適用されます。

WAL には 1 つの LSN「ストップポイント」ができますが、シングルオリジンの PITR と同じように、1 つのタイムスタンプが一貫して適用されます。

定義された PIT に達すると、必要に応じて、リカバリを続行できるように後の PIT も設定される場合があります。

目的の停止ポイントに到達した後、回復したサーバーを昇格する場合は、最初にシャットダウンします。`pg_resetwal` を使用して、このサーバー上のタイムラインで使用されるよりも高い LSN 値に LSN を転送します。このアプローチにより、論理デコードによって生成された重複した LSN がないことが保証されます。

示されている特定の例では、 `N1` は `T1` に復元されます。また、後でまで `N1` に適用されなかった場合でも、`T1` によってコミットされた他のノードからの変更も含まれます。

マルチオリジン PITR を要求するには、 `recovery.conf` ファイルで標準構文を使用します。

```
recovery_target_time = T1
```

2 つの方法のいずれかで `T1` に復元される複製起点のリストを指定する必要があります。新しいパラメーター `recovery_target_origins` を使用して、別の `multi_recovery.conf` ファイルを使用できます。

```
recovery_target_origins = *
```

または、 `recovery_target_origins` のリストとして起点サブセットを指定できます。

```
recovery_target_origins = 1,3
```

指定された `recovery_target_time` へのローカル WAL アクティビティリカバリは、常に暗黙的に実行されます。`recovery_target_origins` で指定されていないオリジンの場合、`recovery_target_origins` で言及されたリストのターゲットがいつ達成されるかに応じて、リカバリは任意の時点で停止できます。

`multi_recovery.conf` ファイルがない場合、回復はデフォルトで元の PostgreSQL PITR の動作になります。これは、変更が COMMIT オーダーで単一のマスターから到着することを前提に設計されています。

注釈: この機能は、 EDB Postgres Extended でのみ使用できます。 Barman は `multi_recovery.conf` ファイルを作成しません。

8.12.2 復元する

標準の PostgreSQL ノードと同じ手順で物理バックアップを作成できますが、PGD ノードの物理バックアップを復元するのは少し複雑です。

EDB Postgres 分散クラスターの障害またはバックアップからの新しいクラスターのシード

物理バックアップを復元する最も一般的なユースケースには、データセンターに障害が発生した場合など、クラスター内のすべての PGD ノードの障害または交換が含まれます。

この手順を実行して、 EDB Postgres 分散クラスターの現在のコンテンツのクローンを作成し、QA または開発インスタンスをシードすることもできます。

その場合、単一の PGD ノードの物理バックアップと、オプションで WAL アーカイブに基づいて、PGD 機能を復元できます。

- まだいくつかの PGD ノードがライブで実行されている場合は、PGD ノードを復元したホストをフェンスオフして、生き残った PGD ノードに接続できないようにします。このプラクティスにより、新しいノードが既存のクラスターを混同しないようにします。
- PGD ノードの 1 つの物理バックアップから単一の PostgreSQL ノードを復元します。
- バックアップに関連付けられた WAL アーカイブがある場合は、適切な `recovery.conf` を作成し、PostgreSQL をリカバリで起動して、最新の状態まで再生します。必要に応じて、ここで代替の `recovery_target` を指定できます。
- 復元されたノードを起動するか、スタンバイリカバリの場合は読み取り/書き込みに昇格します。生き残ったノードからフェンスしたままにしてください！
- 物理バックアップに含まれていた残りの PGD メタデータをクリーンアップします。
- PostgreSQL インスタンスを完全に停止して再起動します。
- `bdr.join_node_group()` 関数呼び出しに基づいた標準手順で、さらに PGD ノードを追加します。

PGD メタデータのクリーンアップ

残った PGD メタデータをクリーンアップするには：

1. `bdr.drop_node` を使用して PGD ノードを削除します。
2. PostgreSQL を完全に停止して再起動します（重要！）。

複製起点のクリーンアップ

レプリケーションオリジンはシステムカタログに永続的に記録されるため、別の手順で明示的に削除する必要があります。したがって、それらはバックアップと復元されたインスタンスに含まれます。それらは依存関係として明示的に記録されないため、BDR 拡張機能を削除しても自動的に削除されません。

クラッシュセーフな方法で着信レプリケーションの進行状況を追跡するために、PGD はリモートマスターノードごとに 1 つのレプリケーションオリジンを作成します。したがって、前のクラスター内のノードごとに、これを 1 回実行します。

```
SELECT pg_replication_origin_drop(bdr_dbname_grpname_nodename);
```

次のようにレプリケーションオリジンをリストできます。

```
SELECT * FROM pg_replication_origin;
```

PGD によって作成されたものは、名前で簡単に認識できます。

レプリケーションスロットのクリーンアップ

pg_basebackup で物理バックアップが作成された場合、レプリケーションスロットはバックアップから省略されます。

他の一部のバックアップ方法では、古い状態または無効な状態である可能性が高いレプリケーションスロットを保持する場合があります。バックアップを復元したら、これを使用してすべてのレプリケーションスロットを削除します。

```
SELECT pg_drop_replication_slot(slot_name)
FROM pg_replication_slots;
```

いくつかのスロットを保持する理由がある場合は、`WHERE slot_name LIKE 'bdr%'` 句を追加できますが、これが役立つことはめったにありません。

> **警告:** ライブ PGD ノードでこれを実行しないでください。

8.13 Monitoring

レプリケーション設定の監視は、システムが次のことを保証するために重要です。

- 最適に実行します
- ディスク容量が不足しない
- 操作を停止する可能性のある他の障害に遭遇しません

自動モニタリングを導入して、管理者にアラートが届き、問題が発生したときにプロアクティブなアクションを取れるようにすることが重要です。たとえば、レプリケーションスロットがひどく遅れ始めた場合、管理者にアラートを通知できます。

EDB は、バージョン 8.1 以降の PGD をサポートする Postgres Enterprise Manager（PEM）を提供します。詳細については、 *Monitoring EDB Postgres Distributed* を参照してください。

または、ツールまたはユーザーは、BDR 拡張機能が提供する情報ビューおよび機能を独自に呼び出すことができます。詳細は *Monitoring through SQL* を参照してください。

EDB Postgres Distributed は OpenTelemetry とも統合されているため、既存のレポート設定を使用して、EDB Postgres Distributed クラスターの状態を追跡できます。詳細は *OpenTelemetry integration* を参照してください。

8.13.1 OpenTelemetry integration

モニタリング情報とトレースを OpenTelemetry コレクターに報告するように EDB Postgres Distributed を構成できます。

EDB Postgres Distributed OTEL コレクターは、いくつかのリソース属性を入力します。これらはすべてのメトリックとトレースに添付されます。

- `service.name` は `bdr.otel_service_name` 構成設定で構成できます。 - `service.namespace` は常に `edb_postgres_distributed` に設定されます。 - `service.instance.id` は、常に Postgres インスタンスのシステム識別子に設定されます。 - `service.version` は、Postgresql インスタンスにロードされた BDR 拡張機能の現在のバージョンに設定されます。

メトリック収集

構成オプション `bdr.metrics_otel_http_url` を空でない URL に設定すると、メトリック収集が有効になります。

次の表に示すように、さまざまな種類のメトリックが収集されます。

一般的な指標

Metric name	Type	Labels	Description
pg_backends_by_state	gauge	conn_state - idle, active, idle in transaction, fast-path functioncall, idle in transaction (aborted), disabled, undefined	特定の状態のバックエンドの数

次のページに続く

表 8 – 前のページからの続き

Metric name	Type	Labels	Description
pg_oldest_xact_start	gauge		最も古いトランザクション開始時刻
pg_oldest_activity_start	gauge		最も古いクエリ開始時刻
pg_waiting_backends	gauge	wait_type - LWLock, Lock, BufferPin, Activity, Client, Extension, IPC, Timeout, IO, ??? (for unknown)	待機タイプ別の現在待機中のバックエンドの数
pg_start_time	gauge		サーバーが起動したタイムスタンプ
pg_reload_time	gauge		サーバーが最後に構成をリロードしたタイムスタンプ

レプリケーションメトリック

Metric name	Type	Labels	Description
bdr_slot_sent_lag	gauge	slot_name - name of a slot	各レプリケーションスロットの現在の送信ラグ（バイト単位）
bdr_slot_write_lag	gauge	slot_name - name of a slot	各レプリケーションスロットの現在の書き込みラグ（バイト単位）
bdr_slot_flush_lag	gauge	slot_name - name of a slot	各レプリケーションスロットの現在のフラッシュラグ（バイト）
bdr_slot_apply_lag	gauge	slot_name - name of a slot	各レプリケーションスロットのバイト単位の現在の適用ラグ
bdr_subscription_receive_l	gauge	sub_name - name of subscription	サブスクリプションごとに現在受信している LSN
bdr_subscription_flush_lsn	gauge	sub_name - name of subscription	各サブスクリプションの現在フラッシュされた LSN
bdr_subscription_apply_lsr	gauge	sub_name - name of subscription	各サブスクリプションに現在適用されている LSN
bdr_subscription_receiver	gauge	sub_name - name of subscription	サブスクリプションレシーバーが現在実行されている（1）かどうか（0）

コンセンサスメトリック

Raft コンセンサスのモニタリング も参照

Metric name	Type	Labels	Description
bdr_raft_state	gauge	state_str - RAFT_FOLLOWER, RAFT_CANDIDATE RAFT_LEADER, RAFT_STOPPED	このノードでのコンセンサスのラフト状態
bdr_raft_protocol_version	gauge		このノードが使用するコンセンサスプロトコルバージョン
bdr_raft_leader_node	gauge		このノードが現在のリーダーであると見なすノードの ID
bdr_raft_nodes	gauge		コンセンサスに参加するノードの総数（学習者/非投票ノードを含む）
bdr_raft_voting_nodes	gauge		コンセンサスにおける実際の投票ノードの数
bdr_raft_term	gauge		このノードがいる現在の raft 用語
bdr_raft_commit_index	gauge		このノードによってコミットされた Raft コミットインデックス
bdr_raft_apply_index	gauge		このノードによって適用された Raft コミットインデックス

トレース

OpenTelemetry へのコレクションのトレースには、`bdr.trace_otel_http_url` を構成し、 `bdr.trace_enable` を使用してトレースを有効にする必要があります。

トレースは、現時点では一部のサブシステム、主にクラスター管理機能に制限されています。トレースには、次のスパンが表示されます。

Span name	Description
create_node_group	グループ作成
alter_node_group_config	グループ設定オプションの変更
alter_node_config	node config オプションの変更
join_node_group	グループに参加するノード
join_send_remote_request	参加ノードに代わって参加要求を送信する参加ソース
add_camo_pair	CAMO ペアを追加
alter_camo_pair	CAMO ペアを変更する
remove_camo_pair	CAMO ペアの削除
alter_commit_scope	コミットスコープ定義を変更する（新規作成または既存を更新）

次のページに続く

表 11 – 前のページからの続き

Span name	Description
alter_proxy_config	PGD-Proxy インスタンスの構成を変更します（新規作成または既存の更新のいずれか）
walmsg_global_lock_send	グローバルロック WAL メッセージを送信する
walmsg_global_lock_recv	グローバルロック WAL メッセージを受信
ddl_epoch_apply	グローバルロックエポック適用（新しいエポックを開始するためにクラスターが十分に同期されていることを確認します）
walmsg_catchup	ノード削除中のキャッチアップ WAL メッセージ
raft_send_appendentries	内部ラフトブックキーピングメッセージ
raft_recv_appendentries	内部ラフトブックキーピングメッセージ
raft_request	ラフトリクエストの実行
raft_query	Raft クエリの実行
msgb_send	コンセンサスメッセージングレイヤーメッセージ
msgb_recv_receive	コンセンサスメッセージングレイヤーメッセージ
msgb_recv_deliver	コンセンサスメッセージングレイヤーのメッセージ配信
preprocess_ddl	DDL コマンドの前処理

TLS サポート

メトリックとトレースのエンドポイントは、HTTP または HTTPS にすることができます。`bdr.otel_https_ca_path`、`bdr.otel_https_key_path`、および `bdr.otel_https_cert_path` 構成オプションを使用して、CA バンドル、クライアントキー、およびクライアント証明書へのパスを構成できます。

8.13.2 Monitoring through SQL

EDB Postgres Distributed は、その分散性に固有のいくつかのモニタリングおよび統計ビューを提供します。標準の Postgres モニタリングは、EDB Postgres Distributed のモニタリングにも役立ちます。

モニタリングの概要

PGD グループは、ノードと呼ばれることが多い複数のサーバーで構成されています。すべてのノードを監視して、グループ全体の状態を確認します。

bdr_monitor ロールは、`bdr.monitor` ファンクションを実行して、3 つのレベルのいずれかを使用して PGD の健全性を評価できます。

- `OK` —多くの場合、緑色で示されます。
- `WARNING` —黄色で表示されることが多いです。
- `CRITICAL` —赤く表示されることが多いです。
- `UNKNOWN` —認識されない状況の場合、多くの場合赤で示されます。

PGD は、さまざまな内部メトリックの瞬間的な状態を表示する動的カタログビューも提供します。また、構成のデフォルトとユーザーが要求する構成の変更を格納するメタデータカタログも提供します。これらのビューとテーブルの一部は、bdr_monitor または bdr_read_all_stats でアクセスできますが、一部には、より高いセキュリティ要件を持つユーザーまたは内部情報が含まれています。

PGD を使用すると、各ノードを個別に監視したり、単一のノードにアクセスしてグループ全体を監視したりできます。各ノードを個別に監視したい場合は、各ノードに接続して監視要求を発行します。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てください。これらの要求は他のノードを呼び出してグループレベルの情報セットを組み立てるためです。

bdr_superuser 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た場合、すべてのノードに対して独自の呼び出しを行うことができます。

モニタリングノードの参加と削除

デフォルトでは、ノード管理機能は結合または部分操作が完了するまで待機します。それぞれの wait_for_completion 関数引数を使用して、待機をオフにできます。待機がオフになっている場合、結合または部分操作がいつ終了するかを確認するには、bdr.node_summary および bdr.event_summary を使用して間接的にノードの状態を確認します。

呼び出されると、ヘルパー関数 bdr.wait_for_join_completion() は、すべての未処理のノード参加操作領域が完了するまで PostgreSQL セッションを一時停止します。

この例は、 bdr.node_summary からの SELECT クエリの出力を示しています。これは、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しています。

```
# SELECT node_name, interface_connstr, peer_state_name,
#     node_seq_id, node_local_dbname
# FROM bdr.node_summary;
-[ RECORD 1 ]-----+----------------------------------------
node_name         | node1
interface_connstr | host=localhost dbname=postgres port=7432
peer_state_name   | ACTIVE
node_seq_id       | 1
node_local_dbname | postgres
-[ RECORD 2 ]-----+----------------------------------------
node_name         | node2
interface_connstr | host=localhost dbname=postgres port=7433
peer_state_name   | ACTIVE
node_seq_id       | 2
node_local_dbname | postgres
-[ RECORD 3 ]-----+----------------------------------------
node_name         | node3
interface_connstr | host=localhost dbname=postgres port=7434
peer_state_name   | JOINING
node_seq_id       | 3
node_local_dbname | postgres
```

また、テーブル *bdr.node_catchup_info* は、キャッチアップ状態に関する情報を提供します。これは、ノードの参加またはノードの分離に関連します。

ノードが分離されると、クラスター内の一部のノードがその分離ノードからすべてのデータを受信しない場合があります。したがって、ノードを分割すると、そのデータをすでに受信して転送できるノードから一時的なスロットが作成されます。

`catchup_state` は、次のいずれかになります。

```
10 = setup
20 = start
30 = catchup
40 = done
```

レプリケーションピアのモニタリング

レプリケーションアクティビティの監視には、2 つのメインビューを使用します。

- 発信レプリケーションを監視するための FROM bdr.node_slots;
- 着信レプリケーションを監視するための *FROM bdr.subscription_summary;*

標準の PostgreSQL レプリケーションモニタリングビュー
: *pg_catalog.pg_stat_replication* を照会することにより、 `bdr.node_slots`

が提供する情報のほとんどを取得することもできます

そして *pg_catalog.pg_replication_slots* 。

各ノードには 1 つの PGD グループスロットがあります。これは決して接続してはならず、アクティブとしてマークされることはめったにありません。これは正常であり、何かがダウンまたは切断されていることを意味するものではありません。
: Replication slots created by PGD <Node management>` を参照してください。

発信レプリケーションのモニタリング

送信レプリケーションアクティビティの監視には、別のビューを使用できます。

- 発信レプリケーションを監視するための SELECT * FROM bdr.node_replication_rates;

  :ref:` <User visible catalogs and views>` ビューは、特にピアノードのキャッチアップ推定とともに、発信レプリケーションアクティビティの全体像を提供します。

```
# SELECT * FROM bdr.node_replication_rates;
-[ RECORD 1 ]----+-----------
peer_node_id     | 112898766
target_name      | node1
sent_lsn         | 0/28AF99C8
replay_lsn       | 0/28AF99C8
```

(次のページに続く)

(前のページからの続き)

```
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
```

apply_rate は、1 秒あたりのバイト単位の速度を指します。これは、ピアがローカルノードからのデータを消費しているレートです。ノードがクラスターに再接続したときの replay_lag は、すぐにゼロに設定されます。この情報は、将来のリリースで修正される予定です。回避策として、ピアノードがローカルノードのデータに追いつくために必要な時間を参照する catchup_interval 列を使用することをお勧めします。他のフィールドも FROM bdr.node_slots; から利用できます

ビュー。

注釈: このカタログは、 bdr-enterprise 拡張機能がインストールされている場合にのみ存在します。

管理者は、ローカルノードからの送信レプリケーションについて `

FROM bdr.node_slots; にクエリを実行できます。現在のノードで認識されているグループ内の他のすべてのノードのレプリケーションステータスと、現在のノードで PGD によって作成された追加のレプリケーションスロットに関する情報を表示します。

```
# SELECT node_group_name, target_dbname, target_name, slot_name, active_pid,
#     catalog_xmin, client_addr, sent_lsn, replay_lsn, replay_lag,
#     replay_lag_bytes, replay_lag_size
# FROM bdr.node_slots;

-[ RECORD 1 ]---+----------------------------
node_group_name | bdrgroup
target_dbname   | postgres
target_name     | node3
slot_name       | bdr_postgres_bdrgroup_node3
active_pid      | 15089
catalog_xmin    | 691
client_addr     | 127.0.0.1
```

(次のページに続く)

(前のページからの続き)

```
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
```

PGD はメッシュネットワークであるため、クラスター内のラグの完全なビューを取得するには、参加しているすべてのノードでこのクエリを実行する必要があります。

replay_lag_bytes は、ローカルサーバーの現在の WAL 書き込み位置と、ピアノードによって確認された最後の位置である replay_lsn との WAL 位置の差を報告します。 replay_lag_size は、人間が読める形式のものです。WAL には通常、複製されないが replay_lag_bytes でカウントされる多くの書き込みが含まれることを理解することが重要です。例：

- 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一部ではないテーブルへの書き込み

したがって、ここで報告されるバイト単位のラグは、ピアノードを最新の状態にするためにネットワークで複製する必要があるデータの量ではなく、処理する必要があるサーバーサイド WAL の量のみです。

同様に、 replay_lag は、ピアノードが追いつくまでにかかる時間や、bdr.node_slots が照会されたときの現在の位置から書き込み位置まで再生するのにかかる時間の尺度ではありません。ピアが最新のコミットを確認してから現在の実時間までの遅延を測定します。ノードが再接続した直後にこの列がゼロに設定されるため、replay_lag_bytes および bdr.node_replication_rates の replay_lag_size または catchup_interval を監視することをお勧めします。

論理レプリケーションがトランザクションをストリーミングしている間、バイトと時間の両方のラグは進行しません。コミットがレプリケートされる場合にのみ変更されます。したがって、ラグは「のこぎり」になる傾向があり、トランザクションがストリーミングされると上昇し、ピアノードがコミットしてフラッシュし、確認を送信すると再び低下します。報告された LSN は、同様の理由で、スムーズに進むのではなく「階段」の位置にあります。

レプリケーションが切断されると（active = 'f' ）、active_pid 列は NULL になり、client_addr およびアクティブな接続でのみ意味を持つ他のフィールドも同様です。state フィールドは'disconnected' です。 _lsn フィールドは confirmed_flush_lsn と同じです。これは、クライアントが再生および保存したことが確実にわかっている最後の位置であるためです。_lag フィールドは、クライアントで確認された最新のフラッシュと現在時刻の間の経過時間を示します。_lag_size および_lag_bytes フィールドは、confirmed_flush_lsn とローカルサーバーの現在の WAL 挿入位置との間の距離を報告します。

注釈: *restart_lsn* が他の `lsn` 列の後ろにあるのは正常です。これは、レプリケーションの問題やピアノードの遅延を示しているわけではありません。 *restart_lsn* は、PostgreSQL の内部論理デコードが中断された場合に WAL を読み取っていなければならない位置です。通常、まだレプリケートおよびフラッシュされていない最も古いトランザクションの位置を反映します。非常に古い `restart_lsn` は、切断後のレプリケーションの再開を遅くし、必要以上の WAL の保持を強制する可能性がありますが、それ以外は無害です。心配な場合は、非常に長時間実行されるトランザクションと忘れられた準備済みトランザクションを探してください。

着信レプリケーションのモニタリング

bdr.subscription_summary ビューを照会することにより、着信レプリケーション（サブスクリプションとも呼ばれます）を監視できます。このクエリは、EDB Postgres 分散クラスター内の他のノードへの既知のサブスクリプションのリストとレプリケーションワーカーの状態を表示します。

8.13.3 SELECT node_group_name, origin_name, sub_enabled, sub_slot_name,

8.13.4 subscription_status

8.13.5 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 | bdr_postgres_bdrgroup_node1 複製 -[RECORD 2]—— -+———————— - node_group_name | bdrgroup origin_name | node3 sub_enabled | t sub_slot_name | bdr_postgres_bdrgroup_node1 subscription_status | bdr_postgres_bdrgroup_node1 複製する

  デコードワーカー が有効になっている場合、関数 *bdr.wal_sender_stats()*

を使用して、WAL 送信者ごとの現在の論理変更レコード（LCR）ファイルに関する情報をモニタできます。例：

```
postgres=# SELECT * FROM bdr.wal_sender_stats();
   pid   | is_using_lcr |       decoder_slot_name       |             lcr_file_name
- --------+--------------+-------------------------------+---------------------------
↪--------------
 2059904 | f            |                               |
```

(次のページに続く)

(前のページからの続き)

```
 2059909 | t             | bdr_postgres_bdrgroup_decoder |␣
↪0000000000000000000001400000000000000000
 2059916 | t             | bdr_postgres_bdrgroup_decoder |␣
↪0000000000000000000001400000000000000000
(3 rows)
```

is_using_lcr が FALSE の場合、decoder_slot_name /lcr_file_name は NULL です。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ノード を提供している場合です。

また、関数 *bdr.get_decoding_worker_stat()* を使用して、デコードワーカーに関する情報をモニタできます。例：

```
postgres=# SELECT * FROM bdr.get_decoding_worker_stat();
   pid   | decoded_upto_lsn | waiting | waiting_for_lsn
- --------+------------------+---------+-----------------
 1153091 | 0/1E5EEE8        | t       | 0/1E5EF00
(1 row)
```

PGD レプリケーションワーカーのモニタリング

すべての PGD ワーカーは、システムビュー bdr.stat_activity に表示されます。これには、*pg_stat_activity* と同じ列と情報コンテンツがあります。したがって、このビューは、PGD システムの状態に関する次の洞察を提供します。

- 待機の理由が PGD に関連する場合、wait_event 列の情報が強化されました。
- ライタプロセスが DDL を実行している場合を除き、PGD ワーカーの query 列は空白です。

bdr.workers ビューには、 bdr.stat_activity から利用できない PGD ワーカー固有の詳細が表示されます。

ビュー bdr.event_summary は、作業の続行に問題があるワーカーによって報告された最後のエラー（ある場合）を示します。この情報は永続的であるため、エラーの存在だけでなく、エラーの時刻に注意することが重要です。ほとんどのエラーは一時的なものであり、PGD ワーカーは失敗した操作を再試行します。

PGD ライターの監視

別のシステムビューである bdr.writers は、ライタのアクティビティを監視します。このビューには、ライタワーカーの現在のステータスのみが表示されます。それには以下が含まれます：

- ライタが属するサブスクリプションを識別する sub_name
- ライタプロセスの pid
- ライタが進行中のストリーミングトランザクションの適用をサポートしているかどうかを知るための streaming_allowed

- ライタが現在ストリーミングトランザクションを適用しているかどうかを知るための is_streaming
- commit_queue_position は、コミットキュー内のライタの位置を確認します

PGD は、オリジンで発生したのと同じコミットオーダーに従うことにより、コミットオーダーを尊重します。並列ライターの場合、複数のライターが同時に異なるトランザクションを適用する場合があります。commit_queue_position は、コミットする順序を示しています。値 0 は、ライタが最初にコミットすることを意味します。値-1 は、コミット位置がまだわかっていないことを意味します。これは、ストリーミングトランザクション、またはライタが現在トランザクションを適用していないときに発生する可能性があります。

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、PGD グループ全体に存在する重量ロックです。

現在、2 種類のグローバルロックがあります。

- DDL ロック、データベース内の永続的な（一時ではない）オブジェクト（つまり、テーブル）に対するすべての DDL 操作をシリアル化するために使用されます
- DML リレーションロック、リレーション定義を変更する DDL 操作中にリレーションへの書き込みをロックアウトするために使用されます

DDL 操作のタイプと bdr.ddl_locking 設定の値に応じて、同じトランザクションに対していずれかまたは両方のエント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、

SELECT lock_type、relation、pid FROM bdr.global_locks; ビューで表示されます。このビューは、ロックの種類を示しています。リレーションロックの場合、ロックされているリレーション、ロックを保持している PID（ローカルの場合）、およびロックがグローバルに付与されたかどうかを示します。グローバルなアドバイザリーロックの場合、

lock_type 列は GLOBAL_LOCK_ADVISORY を示し、 relation 列はロックが取得されるアドバイザリーキーを示します。

この例は、bdr.ddl_locking = on で ALTER TABLE ステートメントを実行中の bdr.global_locks の出力を示しています。

8.13.6 SELECT lock_type、relation、pid FROM bdr.global_locks;

- [RECORD 1]———— – lock_type | GLOBAL_LOCK_DDL リレーション | [NULL] pid | 15534 -[RECORD 2]———— – lock_type | GLOBAL_LOCK_DML リレーション| 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、

Internal catalogs and views を参照してください。

競合の監視

レプリケーション *CRDT* データ型を使用した列の競合の処理 は、複数のノードが互いに相互作用できる方法で同じ行に影響を与える変更を加えた場合に発生する可能性があります。PGD システムを監視して競合を特定し、可能な場合は、アプリケーションを変更して競合を排除するか、競合の頻度を減らします。

デフォルトでは、すべての競合は bdr.conflict_history に記録されます。このログには競合するデータの完全な詳細が含まれているため、行は行レベルのセキュリティによって保護され、レプリケートテーブルの所有者のみが参照できるようにします。所有者は、競合を予想し、競合を分析して、解決すべき問題と見なされる可能性のある問題を確認する必要があります。

モニタリングには、ユーザーデータを含まない bdr.conflict_history_summary を使用します。この例は、効率的なクエリプランを使用して、当日に発生した競合の数をカウントするクエリを示しています。

```
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);
```

統計を適用する

PGD は、サブスクリプションごととテーブルごとの両方について、レプリケーションの適用に関する統計を収集します。

2 つのモニタリングビューが存在します。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。これらのビューはどちらも以下を提供します。

- レプリケートされた INSERT / UPDATE / DELETE / TRUNCATE の数
- ブロックアクセスとキャッシュヒット率
- 読み取り/書き込みの合計 I / O 時間
- ファイルにストリーミングされた進行中のトランザクションの数
- ライターにストリーミングされた進行中のトランザクションの数
- コミット/中止された進行中のストリーミングトランザクションの数

リレーションの場合のみ、 bdr.stat_relation には以下も含まれます。

- リレーションのレプリケーションの処理にかかった合計時間
- リレーション（のみ）のロック（ある場合）を取得するための合計ロック待機時間

サブスクリプションの場合のみ、 bdr.stat_subscription には以下が含まれます。

- サブスクリプションにレプリケートされた COMMIT / DDL の数
- このサブスクリプションがアップストリームに接続した回数

これらの統計の追跡は、それぞれ PGD GUC bdr.track_subscription_apply および bdr.track_relation_apply によって制御されます。

次に、これらからの出力例を示します。

sql # SELECT sub_name、nconnect、ninsert、ncommit、nupdate、ndelete、ntruncate、nddl

FROM bdr.stat_subscription; -[RECORD 1]————————————— - sub_name | bdr_regression_bdrgroup_node1_node2 nconnect | 3 n 挿入 | 10 ncommit | 5 nupdate | 0 ndelete | 0 トランケート | 0nddl | 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。

関数 `bdr.reset_subscription_stats` および `bdr.reset_relation_stats` を使用して、これらのビューの統計カウンターをゼロにリセットできます。

標準の PostgreSQL 統計ビュー

テーブルとインデックスの使用に関する統計は、通常、ダウンストリームマスターによって更新されます。これは、
autovacuum

の正しい機能に不可欠です。ダウンストリームマスターにローカル書き込みがなく、統計がリセットされていない場合、次の 2 つのビューには、アップストリームとダウンストリームの間で対応する結果が表示されます。

- `pg_stat_user_tables`
- `pg_statio_user_tables`

注釈: 上流のテーブル統計が必ずしも下流のものと*似ている*とは限りません。私たちはそれらが同じ量だけ*変化*することを期待しています。統計が 1M の挿入と 1M の更新を示すテーブルの例を考えます。新しいノードが PGD 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、片側でのすべての変更が反対側にレプリケートされるため、アップストリームとダウンストリームのテーブル統計は同じ量だけ変化します。

インデックスは変更を適用するために使用されるため、ダウンストリーム側の識別インデックスは、非識別インデックスよりも `UPDATE` および `DELETE` を実行するワークロードで頻繁に使用されるように見える場合があります。

組み込みのインデックス監視ビューは次のとおりです。

- `pg_stat_user_indexes`
- `pg_statio_user_indexes`

これらすべてのビューについては、 *PostgreSQL documentation on the statistics views* で詳細に説明されています。

PGD バージョンの監視

PGD を使用すると、同じクラスター内のノード間で異なる Postgres バージョンと異なる BDR 拡張バージョンを実行できます。この機能は、アップグレードに役立ちます。

ビュー bdr.group_versions_details は、ファンクション bdr.run_on_all_nodes() を使用して、すべてのノードから Postgres および BDR 拡張機能バージョンを同時に取得します。例：

```
bdrdb=# SELECT node_name, postgres_version, bdr_version
        FROM bdr.group_versions_details;
 node_name | postgres_version | bdr_version
- ----------+------------------+-------------
 node1     | 15.2.0           | 5.0.0
 node2     | 15.2.0           | 5.0.0
```

推奨されるセットアップは、すべてのノードで同じ（および最新の）バージョンをできるだけ早く実行することです。クラスターで異なるバージョンの BDR 拡張機能を長時間実行しないことをお勧めします。

モニタリングのために、次のアラートレベルをお勧めします。

- status=UNKNOWN、message=このノードは PGD グループの一部ではありません
- status = OK、message =すべてのノードは同じ PGD バージョンを実行しています
- status=WARNING、message=アクセスできないノードが少なくとも 1 つあります
- status=WARNING、message=他のノードと比較すると、異なる PGD バージョンを実行しているノードがあります

説明されている動作は関数 bdr.monitor_group_versions() で実装されています。関数は bdr.group_version_details ビューから返された PGD バージョン情報を使用して、クラスター全体のバージョンチェックを提供します。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_versions();
 status |                message
- -------+---------------------------------------
 OK     | All nodes are running same BDR versions
```

Raft コンセンサスのモニタリング

Raft コンセンサスは、常にクラスター全体で機能している必要があります。Raft コンセンサスを機能させずに EDB Postgres 分散クラスターを実行した場合の影響は次のとおりです。

- PGD データ変更のレプリケーションは、引き続き正常に動作する場合があります。
- グローバル DDL/ DML ロックが機能しません。
- Galloc シーケンスは最終的にチャンクを使い果たします。
- Eager Replication が機能しません。

- クラスターのメンテナンス操作（ノードに参加、ノードの一部、スタンバイの昇格）は引き続き許可されますが、完了しない（代わりにハングする）場合があります。
- PGD ノード間でノードのステータスが正しく同期されない場合がある。
- PGD グループレプリケーションスロットは LSN を進めないため、WAL ファイルをディスク上に保持します。

ビュー bdr.group_raft_details は、関数 bdr.run_on_all_nodes() および bdr.get_raft_status() を使用して、すべてのノードから同時に Raft コンセンサスステータスを取得します。例：

```
bdrdb=# SELECT node_id, node_name, state, leader_id
FROM bdr.group_raft_details;
  node_id   | node_name | node_group_name |     state      | leader_id
- -----------+-----------+-----------------+----------------+------------
 1148549230 | node1     | top_group       | RAFT_LEADER    | 1148549230
 3367056606 | node2     | top_group       | RAFT_FOLLOWER | 1148549230
```

これらの条件がすべて満たされている場合、Raft コンセンサスは正しく機能しています。

- すべてのノードで有効な状態（RAFT_LEADER または RAFT_FOLLOWER ）が定義されています。
- ノードのうち 1 つだけが RAFT_LEADER です。
- leader_id はすべての行で同じであり、state = RAFT_LEADER が含まれる行の node_id と一致する必要があります。

時々、Raft コンセンサスは、新しい RAFT_LEADER を定義するための新しい選挙を開始します。選挙中に、RAFT_LEADER がなく、一部のノードが自分自身を RAFT_CANDIDATE であると考える中間状況が発生する可能性があります。選択全体は 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状況が返された場合、bdr.raft_election_timeout を待って、クエリを再度実行します。bdr.raft_election_timeout が合格した後、リストされている条件の一部がまだ満たされていない場合、Raft コンセンサスは機能していません。

Raft コンセンサスは、単一のノードでのみ正しく機能しない場合があります。たとえば、ノードの 1 つが現在のリーダーを認識せず、自分自身を RAFT_CANDIDATE と見なします。この場合、次のことを確認することが重要です。

- すべての PGD ノードは、通常接続とレプリケーション接続の両方を介して互いにアクセスできます（ファイル pg_hba.conf を確認してください）。
- PGD バージョンはすべてのノードで同じです。
- bdr.raft_election_timeout はすべてのノードで同じです。

場合によっては、特にノードが地理的に離れている場合、またはネットワークの待機時間が長い場合、デフォルト値の bdr.raft_election_timeout （6 秒）では十分でない場合があります。すべてが正しいことを確認しても Raft コンセンサスがまだ機能しない場合は、すべてのノードで bdr.raft_election_timeout を 30 秒に増やすことを検討してください。 PGD 3.6.11 以降の場合、bdr.raft_election_timeout を設定するにはサーバーのリロードのみが必要です。

Raft コンセンサスがクラスター運用タスクにどのように影響するか、および Raft コンセンサスがグループスロットを進めることに直接責任があることを考えると、モニタリングアラートレベルは次のように定義

されます。

- status=UNKNOWN、message=このノードは PGD グループの一部ではありません
- status=OK、message=Raft Consensus は正しく機能しています
- status=WARNING、message=アクセスできないノードが少なくとも 1 つあります
- status=WARNING、message=RAFT_CANDIDATE のノードがあります。選挙が進行中の可能性があります
- status=WARNING, message=RAFT_LEADER はありません。選挙が進行中の可能性があります
- status=CRITICAL、message=Raft Consensus には単一のノードがあります
- status=CRITICAL、message=RAFT_LEADER が定義されているときに RAFT_CANDIDATE としてノードがあります
- status=CRITICAL、message=RAFT_LEADER として設定されたノードとは異なるリーダーをフォローしているノードがあります

説明されている動作は関数 `bdr.monitor_group_raft()` に実装されています。関数 `bdr.group_raft_details` はビュー `bdr.group_raft_details` から返された Raft コンセンサスステータス情報を使用して、クラスター全体の Raft チェックを提供します。例：

```
bdrdb=# SELECT * FROM bdr.monitor_group_raft();
node_group_name | status |               message
- ---------------/--------+-------------------------------------
myroup           | OK     | Raft Consensus is working correctly
```

レプリケーションスロットのモニタリング

各 PGD ノードは以下を保持します。

- アクティブな PGD ピアごとに 1 つの複製スロット
- 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
```

ピアスロット名は規則 bdr_<DATABASE>_<GROUP>_<PEER> に従いますが、PGD グループスロット名は規則 bdr_<DATABASE>_<GROUP> に従います。関数 bdr.local_group_slot_name() を使用して、グループスロットにアクセスできます。

ピアレプリケーションスロットは、すべてのノードで常にアクティブである必要があります。ピアレプリケーションスロットがアクティブでない場合、次のいずれかを意味する可能性があります。

- 対応するピアがシャットダウンされているか、アクセスできません。
- PGD レプリケーションが壊れています。

ERROR または FATAL のログファイルを grep し、すべてのノードで bdr.event_summary も確認します。根本的な原因は、たとえば、ノードの 1 つで DDL レプリケーションを無効にして互換性のない DDL が実行されたことが考えられます。

ただし、PGD 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。PGD はこのスロットを維持し、他のすべてのピアが対応するトランザクションをすでに消費しているときに LSN を進めます。したがって、グループスロットのステータスを監視する必要はありません。

関数 bdr.monitor_local_replslots() は、すべての PGD ノードレプリケーションスロットが期待どおりに機能しているかどうかのサマリーを提供します。例：

```
bdrdb=# SELECT * FROM bdr.monitor_local_replslots();
 status |                    message
- -------+-----------------------------------------------
 OK     | All BDR replication slots are working correctly
```

次のステータス概要のいずれかが返されます。

- UNKNOWN：This node is not part of any BDR group
- OK：All BDR replication slots are working correctly
- OK：This node is part of a subscriber-only group
- CRITICAL：There is at least 1 BDR replication slot which is inactive
- CRITICAL：There is at least 1 BDR replication slot which is missing

トランザクションの COMMIT の監視

デフォルトでは、PGD 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。

標準の PostgreSQL 同期レプリケーションで PGD を使用できます。PGD は、2 つの新しいトランザクションコミットモード、CAMO と eager レプリケーションも提供します。これらの各モードは堅牢性機能を追加しますが、COMMIT での追加のレイテンシーを犠牲にします。 bdr.stat_activity カタログを使用して、COMMIT での追加時間を動的に監視できます。プロセスはさまざまな wait_event 状態を報告します。1 つ以上の同期スタンバイからの確認を待っている COMMIT のトランザクションは SyncRep 待機イベントを報告しますが、2 つの新しいモードは EagerRep を報告します。

8.14 EDB Postgres Distributed Command Line Interface

EDB Postgres 分散コマンドラインインターフェイス（PGD CLI）は、 EDB Postgres 分散クラスターを管理するためのツールです。 EDB Postgres 分散クラスターに対してコマンドを実行できます。

クラスターリソースに関する情報を検査、管理、および取得するために使用可能なコマンドについては、
Command reference を参照してください。

Trusted Postgres Architect が PGD CLI を展開する方法、スタンドアロンサーバーに PGD CLI を手動でインストールする方法、および接続文字列を指定する方法については、

Installing PGD CLI を参照してください。

8.14.1 要件

PGD CLI を実行するには、Postgres のスーパーユーザー権限が必要です。

8.14.2 PGD CLI の使用

pgd は、PGD コマンドラインインターフェイスのコマンド名です。コマンドオプションの説明については、コマンドリファレンスの *PGD* **でのテーブルアクセスメソッド（*TAM*）の使用** を参照してください。

8.14.3 構成ファイルの指定

ファイルの名前を変更するか、別の場所に移動する場合は、オプションの`-f` または`--config-file` フラグを使用して、新しい名前と場所を指定します。例：

```
pgd show-nodes -f /opt/my-config.yml
```

8.14.4 データベース接続文字列を渡す

`--dsn` フラグを使用して、データベース接続文字列をコマンドに直接渡します。このフラグで接続文字列を渡す場合、構成ファイルは必要ありません。構成ファイルが存在する場合、フラグは優先されます。例：

```
pgd show-nodes --dsn "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
```

8.14.5 出力フォーマットの指定

PGD CLI は、次の出力形式をサポートしています。

Format	Considerations
tabular	デフォルトのフォーマット。データを表形式で表示します。
json	フォーマットなしで生データを表示します。一部のコマンドでは、json 出力に、追加のフィールドやより詳細なメッセージなど、表形式の出力よりも多くのデータが表示される場合があります。
yaml	フィールドオーダーがアルファベット順であることを除いて json と同じです。実験的であり、将来のバージョンでは完全にサポートされない可能性があります。

`-o` または`--output` フラグを使用して、デフォルトの出力形式を json または yaml に変更します。例：

```
pgd show-nodes -o json
```

8.14.6 コマンドラインヘルプへのアクセス

サポートされているコマンドをリストするには、次のように入力します。

```
pgd help
```

特定のコマンドとそのパラメーターのヘルプを表示するには、`pgd help <command_name>` と入力してください。例：

```
pgd help show-nodes
```

Installing PGD CLI

デフォルトでは、Trusted Postgres Architect は各 PGD ノードに PGD CLI をインストールして構成します。クラスター内の非 PGD インスタンスに PGD CLI をインストールする場合は、デプロイする前に Trusted Postgres Architect の構成ファイルのそのインスタンスに pgdcli ロールをアタッチします。詳細については、信頼できる *Postgres* アーキテクト を参照してください。

手動インストール

EDB リポジトリから入手可能な.deb および.rpm パッケージを使用して、任意の Linux マシンに PGD CLI を手動でインストールできます。パッケージ名は edb-pgd5-cli です。例：

sh # Debian の場合

sudo apt-get install edb-pgd5-cli

Trusted Postgres Architect が PGD CLI を構成すると、自動的に接続されます。スタンドアロン EDB Postgres 分散クラスターへの手動インストールでは、接続文字列を提供する必要があります。

データベース接続文字列の指定

構成ファイルを使用して、クラスターのデータベース接続文字列を指定できます。または、接続文字列をコマンドに直接渡すこともできます。詳細については、
sample use case を参照してください。

構成ファイルの使用

pgd-cli-config.yml 構成ファイルを使用して、クラスターのデータベース接続文字列を指定します。構成ファイルには、クラスター内の少なくとも 1 つの PGD ノードのデータベース接続文字列が含まれている必要があります。クラスター名はオプションであり、検証されません。

例：

```
cluster:
  name: cluster-name
  endpoints:
  - "host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
  - "host=bdr-b1 port=5432 dbname=bdrdb user=postgres "
  - "host=bdr-c1 port=5432 dbname=bdrdb user=postgres "
```

デフォルトでは、pgd-cli-config.yml は/etc/edb/pgd-cli ディレクトリにあります。 PGD CLI は、次の場所で pgd-cli-config.yml を検索します。優先順位は高いものから低いものです。

1. /etc/edb/pgd-cli （デフォルト） 2. $HOME/.edb/pgd-cli

ファイルの名前を変更するか、別の場所に移動する場合は、コマンドを入力するときにオプションの-f または--config-file フラグを使用して、新しい名前と場所を指定します。

sample use case を参照してください。

Command reference

pgd は、PGD コマンドラインインターフェイスのコマンド名です。

概要

EDB Postgres 分散コマンドラインインターフェイス（PGD CLI）は、 EDB Postgres 分散クラスターを管理するツールです。 EDB Postgres 分散クラスターに対してコマンドを実行できます。これを使用して、クラスターリソースを検査および管理できます。

オプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-h, --help                 help for pgd
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

こちらもご覧ください

- *check-health*
- *create-proxy*
- *delete-proxy*
- *set-group-options*
- *set-node-options*
- *set-proxy-options*
- *show-clockskew*
- *show-events*
- *show-groups*
- *show-nodes*
- *show-proxies*
- *show-raft*
- *show-replslots*
- *show-subscriptions*

- *show-version*
- *switchover*
- *verify-cluster*
- *verify-settings*

check-health

EDB Postgres 分散クラスターのヘルスを確認します。

概要

すべてのノードにアクセスでき、すべてのレプリケーションスロットが機能しているかどうかなど、さまざまなチェックを実行します。

このコマンドの実行中にクラスターの負荷が高い場合、または多数のノードが含まれる場合、クロックスキューの現在の実装は不正確なスキュー値を返す可能性があることに注意してください。

```
pgd check-health [flags]
```

例

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

$ pgd check-health

Check      Status   Message
-----      ------   -------
ClockSkew  Critical Clockskew cannot be determined for at least 1 BDR node pair
Connection Critical The node bdr-b1 is not accessible
Raft       Warning  There is at least 1 node that is not accessible
Replslots  Critical There is at least 1 BDR replication slot which is inactive
Version    Warning  There is at least 1 node that is not accessible

Example 2 (3 node cluster, all nodes are up but system clocks are not in sync)

$ pgd check-health

Check      Status  Message
-----      ------  -------
ClockSkew  Warning At least 1 BDR node pair has clockskew greater than 2 seconds
Connection Ok      All BDR nodes are accessible
Raft       Ok      Raft Consensus is working correctly
```

(次のページに続く)

(前のページからの続き)

```
Replslots  Ok      All BDR replication slots are working correctly
Version    Ok      All nodes are running same BDR versions

Example 3 (3 node cluster, all nodes are up and all checks are Ok)

$ pgd check-health

Check      Status Message
-----      ------ -------
ClockSkew  Ok     All BDR node pairs have clockskew within permissible limit
Connection Ok     All BDR nodes are accessible
Raft       Ok     Raft Consensus is working correctly
Replslots  Ok     All BDR replication slots are working correctly
Version    Ok     All nodes are running same BDR versions
```

オプション

```
-h, --help   help for check-health
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

create-proxy

EDB Postgres 分散クラスターにプロキシを作成します。

概要

EDB Postgres 分散クラスターにプロキシを作成し、指定されたグループにアタッチします。プロキシ名は、クラスター全体で一意であり、対応するプロキシ構成ファイルで指定された名前と一致する必要があります。

```
pgd create-proxy [flags]
```

例

```
Example 1 (attach new proxy called proxy-a1 to group bdrgroup)

$ pgd create-proxy --proxy-name proxy-a1 --group-name bdrgroup
proxy created successfully
```

オプション

```
    --group-name string   group name
-h, --help                help for create-proxy
    --proxy-name string   proxy name
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

delete-proxy

EDB Postgres 分散クラスターからプロキシを削除します。

概要

EDB Postgres 分散クラスターからプロキシを削除します。

```
pgd delete-proxy [flags]
```

例

```
Example 1 (delete proxy proxy-a1)

$ pgd delete-proxy --proxy-name proxy-a1
proxy deleted successfully
```

オプション

```
-h, --help                help for delete-proxy
    --proxy-name string   proxy name
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

set-group-options

「enable_raft」、「enable_proxy_routing」、「location」などのグループオプションを設定します。

概要

このコマンドで次のグループオプションを設定できます。

- ‘enable_raft’
- ‘enable_proxy_routing’
- ‘場所’
- ‘route_writer_max_lag’

プロキシがグループにアタッチされている場合、「enable_raft」と「enable_proxy_routing」の両方が true である必要があります。

’pgd show-groups -o json’を使用して、各グループのオプション値を表示します。

```
pgd set-group-options [flags]
```

例

```
Example 1 (comma separated multiple options, spaces are not allowed)

$ pgd set-group-options --group-name bdrgroup --option enable_proxy_routing=true,
↪route_writer_max_lag=1000000
group options updated successfully

Example 2 (multiple --option flags are allowed)

$ pgd set-group-options --group-name bdrgroup --option enable_proxy_routing=true --
↪option route_writer_max_lag=1000000
group options updated successfully

Example 3 (use double quote if option value has spaces or special characters)

$ pgd set-group-options --group-name bdrgroup --option "location = mumbai" --option
↪"route_writer_max_lag = 1000000"
group options updated successfully
```

オプション

```
    --group-name string   group name
-h, --help                help for set-group-options
    --option strings      option in name=value format
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

set-node-options

「route_fence」、「route_priority」、「route_writes」などのノードオプションを設定します。

概要

このコマンドで次のノードオプションを設定できます。

- ‘route_dsn’
- ‘route_fence’
- ‘route_priority’
- ‘route_writes’

’pgd show-nodes -o json’を使用して、各ノードのオプション値を表示します。

```
pgd set-node-options [flags]
```

例

```
Example 1 (comma separated multiple options, spaces are not allowed)

$ pgd set-node-options --node-name bdr-a1 --option route_priority=100,route_fence=true
node options updated successfully
```

(次のページに続く)

(前のページからの続き)

```
Example 2 (multiple --option flags are allowed)

$ pgd set-node-options --node-name bdr-a1 --option route_priority=100 --option route_
↪fence=true
node options updated successfully

Example 3 (use double quote if option value has spaces or special characters)

$ pgd set-node-options --node-name bdr-a1 --option "route_priority = 100" --option
↪"route_fence = true"
node options updated successfully
```

オプション

```
-h, --help               help for set-node-options
    --node-name string   node name
    --option strings     option in name=value format
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

set-proxy-options

「listen_address」、「listen_port」、「max_client_conn」などのプロキシオプションを設定します。

概要

このコマンドで次のプロキシオプションを設定できます。

- ‘listen_address’
- ‘listen_port’
- ‘max_client_conn’
- ‘max_server_conn’

- ‘server_conn_keepalive’
- ‘server_conn_timeout’
- ‘consensus_grace_period’

これらのオプションのいずれかを更新した後、プロキシを再起動します。

’pgd show-proxies -o json’を使用して、各プロキシのオプション値を表示します。

```
pgd set-proxy-options [flags]
```

例

```
Example 1 (comma separated multiple options, spaces are not allowed)

$ pgd set-proxy-options --proxy-name proxy-a1 --option listen_address=0.0.0.0,listen_
↪port=6432
proxy options updated successfully, please restart proxy service

Example 2 (multiple --option flags are allowed)

$ pgd set-proxy-options --proxy-name proxy-a1 --option listen_address=0.0.0.0 --
↪option listen_port=6432
proxy options updated successfully, please restart proxy service

Example 3 (use double quote if option value has spaces or special characters)

$ pgd set-proxy-options --proxy-name proxy-a1 --option "listen_address = 0.0.0.0" --
↪option "listen_port = 6432"
proxy options updated successfully, please restart proxy service
```

オプション

```
-h, --help                help for set-proxy-options
    --option strings      option in name=value format
    --proxy-name string   proxy name
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-clockskew

各 BDR 間のクロックスキューのステータスを示します。

概要

クラスター内の各 BDR 間のクロックスキューのステータスを表示します。

このコマンドの実行中にクラスターの負荷が高い場合、または多数のノードが含まれる場合、クロックスキューの現在の実装は不正確なスキュー値を返す可能性があることに注意してください。

記号 意味 —— - ——– * ok ~ warning (skew > 2 秒) !クリティカル（スキュー> 5 秒）×ダウン/到達不能？不明 - 該当なし

```
pgd show-clockskew [flags]
```

例

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

$ pgd show-clockskew

Node   bdr-a1 bdr-b1 bdr-c1 Current Time
----   ------ ------ ------ ------------
bdr-a1  *?*       2022-03-30 07:02:21.334472
bdr-b1 x      *      x      x
```

(次のページに続く)

(前のページからの続き)

```
bdr-c1  *?*       2022-03-30 07:02:21.186809

Example 2 (3 node cluster, all nodes are up)

$ pgd show-clockskew

Node   bdr-a1 bdr-b1 bdr-c1 Current Time
----   ------ ------ ------ ------------
bdr-a1  **       *      2022-03-30 07:04:54.147017
bdr-b1  **       *      2022-03-30 07:04:54.340543
bdr-c1  **       *      2022-03-30 07:04:53.90451
```

オプション

```
-h, --help   help for show-clockskew
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-events

バックグラウンドワーカーエラーやノードメンバーシップの変更などのイベントを表示します。

概要

バックグラウンドワーカーエラーやノードメンバーシップの変更などのイベントを表示します。出力は Time 列の降順でソートされます。メッセージ列は、数行後に切り捨てられます。完全なメッセージを表示するには、json 出力形式（‘-o json’）を使用します。

各ノードの状態の詳細については、 show-nodes コマンドのヘルプ（ ‘pgd show-nodes -h’）を参照してください。

```
pgd show-events [flags]
```

例

```
Example 1 (3 node cluster)

$ pgd show-events --lines 10

Time                          Observer Node Subject Node Source    Type   Subtype
 Message
----                          ------------- ------------ ------    ----   -------
 -------
2023-03-23 05:38:25.243257+00 witness-a1    witness-a1   consensus RAFT   STATE_
↪CHANGE RAFT_LEADER
2023-03-23 05:38:25.23815+00  witness-a1    witness-a1   consensus RAFT   STATE_
↪CHANGE RAFT_CANDIDATE
2023-03-23 05:38:21.197974+00 bdr-a1        bdr-a1       consensus RAFT   STATE_
↪CHANGE RAFT_FOLLOWER
2023-03-23 05:38:21.197107+00 witness-a1    witness-a1   consensus RAFT   STATE_
↪CHANGE RAFT_FOLLOWER
2023-03-23 05:38:21.169781+00 bdr-a2        bdr-a2       consensus RAFT   STATE_
↪CHANGE RAFT_FOLLOWER
2023-03-23 05:38:17.949669+00 witness-a1    bdr-a1       consensus NODE   STATE_
↪CHANGE ACTIVE
2023-03-23 05:38:17.949544+00 bdr-a1        bdr-a1       consensus NODE   STATE_
↪CHANGE ACTIVE
2023-03-23 05:38:17.946857+00 bdr-a2        bdr-a1       consensus NODE   STATE_
↪CHANGE ACTIVE
2023-03-23 05:38:17.91628+00  bdr-a1        bdr-a2       receiver  WORKER ERROR
 pglogical worker received fast finish request, exiting
2023-03-23 05:38:17.915236+00 witness-a1    bdr-a1       consensus NODE   STATE_
↪CHANGE PROMOTING
```

オプション

```
-h, --help        help for show-events
-n, --lines int   show top n lines
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-groups

EDB Postgres 分散クラスター内のすべてのグループとその概要を表示します。

概要

EDB Postgres 分散クラスター内のすべてのグループとその概要を表示します。

場合によっては、raft が適切に機能していないか、グループ raft リーダーが存在しない場合、このコマンドはそのグループの古いまたは間違った書き込みリーダーを表示する場合があります。

```
pgd show-groups [flags]
```

例

```
Example 1 (multi-group cluster, with special witness-only data group
group_c and subscriber-only group group_so)
Note:
1. For group write leader election both enable_raft and
enable_proxy_routing options should be true for that group
2. enable_raft is always set to true for global group

$ pgd show-groups

Group    Group ID   Type            Parent Group Location Raft  Routing Write Leader
-----    --------   ----            ------------ -------- ----  ------- ------------
bdrgroup 1360502012 global                                true  false
group_a  3618712053 data            bdrgroup     a        true  true    bdr-a1
group_b  402614658  data            bdrgroup     b        true  true    bdr-b1
group_c  2808307099 data            bdrgroup     c        false false
group_so 2123208041 subscriber-only bdrgroup     c        false false
```

オプション

```
-h, --help   help for show-groups
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-nodes

EDB Postgres 分散クラスター内のすべてのノードとその概要を表示します。

概要

EDB Postgres 分散クラスター内のすべてのノードと、名前、ノード ID、グループ、現在/ターゲットの状態を含むその概要を表示します。

ノードの状態

- NONE ：ワーカーの起動時にノードの状態は設定解除され、現在の既知の状態にすばやく設定されることが期待されます。
- CREATED：bdr.create_node() が実行されましたが、ノードはまだ EDB Postgres 分散クラスターのメンバーではありません。
- JOIN_START：bdr.join_node_group() は、ローカルノードを既存の EDB Postgres 分散クラスターに参加させ始めます。
- JOINING ：ノード参加が開始され、現在初期同期フェーズにあり、ノード上にスキーマとデータを作成しています。
- CATCHUP ：初期同期フェーズが完了しました。これで、参加は、参加が開始されてから上流のピアノードで実行されたトランザクションを取得および適用する最後のステップになりました。
- STANDBY ：ノード参加は完了しましたが、変更のブロードキャストはまだ開始されていません。すべての参加はこの状態でしばらく時間を費やしますが、ロジカルスタンバイとして定義されている場合、ノードはこの状態で続行されます。
- PROMOTE ：ノードはロジカルスタンバイでした。bdr.promote_node を呼び出して、ノードの状態を ACTIVE に移動しました。これら 2 つの PROMOTE 状態は、1 つのノードのみが STANDBY よりも高く、ACTIVE よりも低い状態にできるという事実と一貫している必要があります。

- PROMOTING ：ロジカルスタンバイからフル BDR ノードへのプロモーションが進行中です。
- ACTIVE ：ノードはフル BDR ノードであり、現在 ACTIVE です。これは、最も一般的なノードのステータスです。
- PART_START ：ノードは ACTIVE または STANDBY でした。bdr.part_node を呼び出して、 EDB Postgres 分散クラスターからノードを削除しました。
- PARTING ：ノードは他のノードから切断し、コンセンサスまたはレプリケーションでそれ以上の役割を果たしません。
- PART_CATCHUP ：非分割ノードは、最近分割されたノードから欠落しているデータを同期します。
- PARTED ：ノードの分割操作がすべてのノードで完了しました。

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

[現在の状態] または [ターゲット状態] 列に示されるように、ノードの読み取り専用状態は STANDBY と示されることに注意してください。

```
pgd show-nodes [flags]
```

例

```
Example 1 (3 node cluster, bdr-a1 and witness-a are up, bdr-a2 is down)

$ pgd show-nodes

Node      Node ID    Group   Type    Current State Target State Status      Seq ID
----      -------    -----   ----    ------------- ------------ ------      ------
bdr-a1    3136956818 group_a data    ACTIVE        ACTIVE       Up          1
bdr-a2    2133699692 group_a data    ACTIVE        ACTIVE       Unreachable 2
witness-a 3889635963 group_a witness ACTIVE        ACTIVE       Up          3

Example 2 (multi-group cluster with witness, logical standby and
subscriber-only nodes)
Note: In contrast to logical standby, the subscriber-only nodes are fully
joined node to the cluster

$ pgd show-nodes

Node               Node ID    Group    Type            Current State Target State␣
↪Status Seq ID
----               -------    -----    ----            ------------- ------------ ----
↪-- ------
bdr-a1             3136956818 group_a  data            ACTIVE        ACTIVE       Up
   6
bdr-a2             2133699692 group_a  data            ACTIVE        ACTIVE       Up
```

(次のページに続く)

(前のページからの続き)

```
   3
logical-standby-a1 1140256918 group_a  standby         STANDBY       STANDBY      Up
   9
witness-a          3889635963 group_a  witness         ACTIVE        ACTIVE       Up
   7
bdr-b1             2380210996 group_b  data            ACTIVE        ACTIVE       Up
   1
bdr-b2             2244996162 group_b  data            ACTIVE        ACTIVE       Up
   2
logical-standby-b1 3541792022 group_b  standby         STANDBY       STANDBY      Up
   10
witness-b          661050297  group_b  witness         ACTIVE        ACTIVE       Up
   5
witness-c          1954444188 group_c  witness         ACTIVE        ACTIVE       Up
   4
subscriber-only-c1 2448841809 group_so subscriber-only ACTIVE        ACTIVE       Up
   8
```

オプション

```
-h, --help   help for show-nodes
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-proxies

EDB Postgres 分散クラスター内のすべてのプロキシとその概要を表示します。

概要

EDB Postgres 分散クラスター内のすべてのプロキシとその概要を表示します。

同じグループに接続されているすべてのプロキシに同じプロキシオプション値を指定することをお勧めします。

```
pgd show-proxies [flags]
```

例

```
Example 1 (multi-group cluster, with 2 proxies attached to each data group)

$ pgd show-proxies

Proxy    Group   Listen Addresses Listen Port
-----    -----   ---------------- -----------
proxy-a1 group_a [0.0.0.0]        6432
proxy-a2 group_a [0.0.0.0]        6432
proxy-b1 group_b [0.0.0.0]        6432
proxy-b2 group_b [0.0.0.0]        6432
```

オプション

```
-h, --help   help for show-proxies
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-raft

BDR Raft（コンセンサスプロトコル）の詳細を表示します。

概要

Raft インスタンス ID、Raft 状態（リーダー、フォロワー）、Raft 用語などの BDR Raft（コンセンサスプロトコル）の詳細を表示します。Raft がサブグループレベルで有効になっている場合、そのサブグループの Raft インスタンスも表示されます。

ネットワークパーティションなど、場合によっては、CLI が接続されているノードによって出力が異なる場合があります。

```
pgd show-raft [flags]
```

例

```
Example 1 (multi-group cluster with subgroup Raft and with witness,
logical standby, subscriber-only nodes)
Note: Unlike data or witness node, logical standby and subscriber-only
nodes dont have Raft voting rights (see Voting Nodes column).

$ pgd show-raft

Instance Group    Node               Raft State    Raft Term Commit Index Nodes␣
↪Voting Nodes Protocol Version
-------- -----    ----               ----------    --------- ------------ ----- ------
↪------ -----------------
1        bdrgroup bdr-b1             RAFT_LEADER   0         383          10    7
       5000
1        bdrgroup bdr-a1             RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup bdr-a2             RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup bdr-b2             RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup logical-standby-a1 RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup logical-standby-b1 RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup subscriber-only-c1 RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup witness-a          RAFT_FOLLOWER 0         383          10    7
       5000
```

(次のページに続く)

(前のページからの続き)

```
1        bdrgroup witness-b          RAFT_FOLLOWER 0         383          10    7
       5000
1        bdrgroup witness-c          RAFT_FOLLOWER 0         383          10    7
       5000
2        group_a  witness-a          RAFT_LEADER   1         2            4     3
       0
2        group_a  bdr-a1             RAFT_FOLLOWER 1         2            4     3
       0
2        group_a  bdr-a2             RAFT_FOLLOWER 1         2            4     3
       0
2        group_a  logical-standby-a1 RAFT_FOLLOWER 1         2            4     3
       0
3        group_b  witness-b          RAFT_LEADER   1         2            4     3
       0
3        group_b  bdr-b1             RAFT_FOLLOWER 1         2            4     3
       0
3        group_b  bdr-b2             RAFT_FOLLOWER 1         2            4     3
       0
3        group_b  logical-standby-b1 RAFT_FOLLOWER 1         2            4     3
       0
```

オプション

```
-h, --help   help for show-raft
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-replslots

BDR レプリケーションスロットの状態を表示します。

概要

BDR レプリケーションスロットの状態を表示します。verbose フラグを使用した出力は、スロットがアクティブかどうか、レプリケーション状態（切断、ストリーミング、キャッチアップ）、おおよそのラグなどの詳細を提供します。

記号 意味 —— - ——– * ok ~ warning (lag > 10M) !クリティカル（ラグ> 100M OR スロットが「非アクティブ」または「切断」）x ダウン/到達不能-n/a

マトリックスビューでは、バイトラグが括弧内に表示される場合があります。maxOf(WriteLag, FlushLag, ReplayLag, SentLag) です。

```
pgd show-replslots [flags]
```

例

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

$ pgd show-replslots

Node   bdr-a1 bdr-b1  bdr-c1
----   ------ ------  ------
bdr-a1  *!(6.6G)*
bdr-b1 x      *       x
bdr-c1  *!(6.9G)*

$ pgd show-replslots --verbose

Origin Node Target Node Status (active/state) Write Lag (bytes/duration)    Flush Lag␣
↪(bytes/duration)    Replay Lag (bytes/duration)   Sent Lag (bytes)
----------- ----------- -------------------- -------------------------    ----------
↪-----------------    ---------------------------   ----------------
bdr-a1      bdr-b1      f / disconnected      6.6G / 8 days 02:58:36.243723 6.6G / 8␣
↪days 02:58:36.243723 6.6G / 8 days 02:58:36.243723 6.6G
bdr-a1      bdr-c1      t / streaming         0B / 00:00:00                 0B /␣
↪00:00:00                 0B / 00:00:00                 0B
bdr-c1      bdr-a1      t / streaming         0B / 00:00:00.000812          0B /␣
↪00:00:00.000812          0B / 00:00:00.000812          0B
bdr-c1      bdr-b1      f / disconnected      6.9G / 8 days 02:58:36.004415 6.9G / 8␣
↪days 02:58:36.004415 6.9G / 8 days 02:58:36.004415 6.9G
```

(次のページに続く)

(前のページからの続き)

```
Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)

$ pgd show-replslots

Node   bdr-a1 bdr-b1  bdr-c1
----   ------ ------  ------
bdr-a1  *!(6.9G)*
bdr-b1  **        *
bdr-c1  *!(5.8G)*

$ pgd show-replslots --verbose

Origin Node Target Node Status (active/state) Write Lag (bytes/duration) Flush Lag␣
↪(bytes/duration) Replay Lag (bytes/duration) Sent Lag (bytes)
----------- ----------- -------------------- ------------------------- -------------
↪------------- -------------------------- ----------------
bdr-a1      bdr-b1      t / catchup           6.9G / 00:00:00.000778     6.9G /␣
↪00:00:00.000778     6.9G / 00:00:00.000778      6.9G
bdr-a1      bdr-c1      t / streaming         0B / 00:00:00.104121       0B /␣
↪00:00:00.104133       0B / 00:00:00.104133        0B
bdr-b1      bdr-a1      t / streaming         0B / 00:00:00              0B /␣
↪00:00:00              0B / 00:00:00               0B
bdr-b1      bdr-c1      t / streaming         0B / 00:00:00              0B /␣
↪00:00:00              0B / 00:00:00               0B
bdr-c1      bdr-a1      t / streaming         6.8K / 00:00:00            6.8K /␣
↪00:00:00            6.8K / 00:00:00             6.8K
bdr-c1      bdr-b1      t / catchup           5.5G / 00:00:00.008257     5.5G /␣
↪00:00:00.008257     5.5G / 00:00:00.008257      5.5G

Example 3 (3 node cluster, all nodes are up and in streaming state)

$ pgd show-replslots

Node   bdr-a1 bdr-b1 bdr-c1
----   ------ ------ ------
bdr-a1  **       *
bdr-b1  **       *
bdr-c1  **       *

$ pgd show-replslots --verbose

Origin Node Target Node Status (active/state) Write Lag (bytes/duration) Flush Lag␣
↪(bytes/duration) Replay Lag (bytes/duration) Sent Lag (bytes)
----------- ----------- -------------------- ------------------------- -------------
↪------------- -------------------------- ----------------
```

(次のページに続く)

(前のページからの続き)

```
bdr-a1      bdr-b1      t / streaming         0B / 00:00:00              0B /␣
↪00:00:00              0B / 00:00:00               0B
bdr-a1      bdr-c1      t / streaming         0B / 00:00:00              0B /␣
↪00:00:00              0B / 00:00:00               0B
bdr-b1      bdr-a1      t / streaming         0B / 00:00:00              0B /␣
↪00:00:00              0B / 00:00:00               0B
bdr-b1      bdr-c1      t / streaming         0B / 00:00:00              0B /␣
↪00:00:00              0B / 00:00:00               0B
bdr-c1      bdr-a1      t / streaming         0B / 00:00:00              528B /␣
↪00:00:00            528B / 00:00:00             0B
bdr-c1      bdr-b1      t / streaming         528B / 00:00:00            528B /␣
↪00:00:00            528B / 00:00:00             0B

Example 4 (cluster with witness, logical standby and subscriber-only nodes;
upstream for logical-standby-a1 is bdr-a1 and for logical-standby-b1 it is bdr-b1)
Note:
1. A logical standby is sent data only by one source node, but no other
nodes receive replication changes from it
2. Subscriber-only node subscribes to replication changes from other nodes
in the cluster, but no other nodes receive replication changes from it

$ pgd show-replslots

Node               bdr-a1 bdr-b1 logical-standby-a1 logical-standby-b1 subscriber-
↪only-c1 witness-c1
----               ------ ------ ------------------ ------------------ ---------------
↪--- -----------
bdr-a1              **        *-*                   *
bdr-b1              **       -                     **                   *
logical-standby-a1 -      -      *                  -                    -
    -
logical-standby-b1 -      -      -                  *                    -
    -
subscriber-only-c1 -      -      -                  -                    *
    -
witness-c1          **       -                   -                    **
```

オプション

```
-h, --help      help for show-replslots
-v, --verbose   verbose output (default true)
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-subscriptions

BDR サブスクリプション（受信レプリケーション）の詳細を表示します。

概要

オリジン/ターゲットノード、最後にリプレイされたトランザクションのタイムスタンプ、現在から最後にリプレイされたトランザクションのタイムスタンプまでのラグなど、BDR サブスクリプション（受信レプリケーション）の詳細を表示します。

```
pgd show-subscriptions [flags]
```

例

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

$ pgd show-subscriptions

Origin Node Target Node Last Transaction Replayed At  Lag Duration (seconds)
----------- ----------- ----------------------------  ----------------------
bdr-a1      bdr-c1      2022-04-23 13:13:40.854433+00 0.514275
bdr-b1      bdr-a1
bdr-b1      bdr-c1
bdr-c1      bdr-a1      2022-04-23 13:13:40.852233+00 0.335464

Example 2 (3 node cluster, bdr-b1 was down and it has just been restarted)
```

(次のページに続く)

(前のページからの続き)

```
$ pgd show-subscriptions

Origin Node Target Node Last Transaction Replayed At  Lag Duration (seconds)
----------- ----------- ----------------------------  ----------------------
bdr-a1      bdr-b1      2022-04-23 13:14:45.669254+00 0.001686
bdr-a1      bdr-c1      2022-04-23 13:14:46.157913+00 -0.002009
bdr-b1      bdr-a1
bdr-b1      bdr-c1
bdr-c1      bdr-a1      2022-04-23 13:14:45.698472+00 0.259521
bdr-c1      bdr-b1      2022-04-23 13:14:45.667979+00 0.002961

Example 3 (3 node cluster, all nodes are up and in streaming state)

$ pgd show-subscriptions

Origin Node Target Node Last Transaction Replayed At  Lag Duration (seconds)
----------- ----------- ----------------------------  ----------------------
bdr-a1      bdr-b1      2022-04-23 13:15:39.732375+00 0.034462
bdr-a1      bdr-c1      2022-04-23 13:15:40.179618+00 0.002647
bdr-b1      bdr-a1      2022-04-23 13:15:39.719994+00 0.305814
bdr-b1      bdr-c1      2022-04-23 13:15:40.180886+00 0.001379
bdr-c1      bdr-a1      2022-04-23 13:15:39.714397+00 0.311411
bdr-c1      bdr-b1      2022-04-23 13:15:39.714397+00 0.052440

Example 4 (cluster with witness, logical standby and subscriber-only nodes;
upstream for logical-standby-a1 is bdr-a1 and for logical-standby-b1 it is bdr-b1)
Note: Logical standby and subscriber-only nodes receive changes but do not
send changes made locally to other nodes

$ pgd show-subscriptions

Origin Node Target Node        Last Transaction Replayed At  Lag Duration (seconds)
----------- -----------        ----------------------------  ----------------------
bdr-a1      bdr-b1             2022-04-23 13:40:49.106411+00 0.853665
bdr-a1      logical-standby-a1 2022-04-23 13:40:50.72036+00  0.138430
bdr-a1      logical-standby-b1
bdr-a1      subscriber-only-c1 2022-04-23 13:40:50.72036+00  0.016226
bdr-a1      witness-c1         2022-04-23 13:40:50.470142+00 0.001514
bdr-b1      bdr-a1             2022-04-23 13:40:49.10174+00  1.095422
bdr-b1      logical-standby-a1
bdr-b1      logical-standby-b1 2022-04-23 13:40:50.713666+00 0.271213
bdr-b1      subscriber-only-c1 2022-04-23 13:40:50.713666+00 0.022920
bdr-b1      witness-c1         2022-04-23 13:40:50.471789+00 -0.000133
witness-c1  bdr-a1             2022-04-23 13:40:49.107706+00 1.089456
```

(次のページに続く)

(前のページからの続き)

```
witness-c1  bdr-b1             2022-04-23 13:40:49.107706+00 0.852370
witness-c1  logical-standby-a1
witness-c1  logical-standby-b1
witness-c1  subscriber-only-c1 2022-04-23 13:40:50.719844+00 0.016742
```

オプション

```
-h, --help   help for show-subscriptions
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

show-version

各ノードにインストールされている BDR と Postgres のバージョンを表示します。

概要

クラスター内の各ノードにインストールされている BDR と Postgres のバージョンを表示します。

```
pgd show-version [flags]
```

例

```
Example 1 (3 node cluster, bdr-a1 and bdr-c1 are up, bdr-b1 is down)

$ pgd show-version

Node   BDR Version Postgres Version
----   ----------- ----------------
bdr-c1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-
↪1.buster+1)
```

(次のページに続く)

(前のページからの続き)

```
bdr-a1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-
↪1.buster+1)
bdr-b1

Example 2 (3 node cluster, all nodes are up)

$ pgd show-version

Node   BDR Version Postgres Version
----   ----------- ----------------
bdr-c1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-
↪1.buster+1)
bdr-a1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-
↪1.buster+1)
bdr-b1 4.1.0       14.2 (EDB Postgres Extended Server 14.2.0) (Debian 2:14.2.0edbpge-
↪1.buster+1)
```

オプション

```
-h, --help   help for show-version
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

switchover

新しい書き込みリーダーに切り替えます。

概要

新しい書き込みリーダーに切り替えます。即時スイッチオーバーには、スイッチオーバー方法「fast」を使用します。「strict」を使用して、指定されたターゲットノードでラグが「route_writer_max_lag」未満になるまで待機します。タイムアウトまたはその他の問題が原因でスイッチオーバーが失敗した場合、BDR は、指定されたターゲットノードとは異なる書き込みリーダーを選択する場合があります。

```
pgd switchover [flags]
```

例

```
Example 1 (with required arguments, default method is strict and default
timeout is 10s)

$ pgd switchover --group-name group_a --node-name bdr-a1
switchover is complete

Example 2 (with optional arguments)

$ pgd switchover --group-name group_a --node-name bdr-a1 --method strict --timeout 15s
switchover is complete

Example 3 (immediate switchover)

$ pgd switchover --group-name group_a --node-name bdr-a1 --method fast
switchover is complete
```

オプション

```
    --group-name string   group name
-h, --help                help for switchover
    --method string       switchover method (strict, fast)
                          strict - waits until lag on given node is less than route_
↪writer_max_lag
                          fast - immediate switchover, route_writer_max_lag is␣
↪ignored (default "strict")
    --node-name string    node name
    --timeout duration    timeout period when switchover method is strict (default␣
↪10s)
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

verify-cluster

クラスターが AlwaysOn アーキテクチャに従ってルールに従うかどうかを確認します。

概要

クラスターが AlwaysOn アーキテクチャに従ってルールに従うかどうかを確認します。

```
pgd verify-cluster [flags]
```

例

```
Example 1 (cluster with not recommended architecture)

$ pgd verify-cluster

Check                                                                          ␣
↪Status   Groups
-----                                                                           --
↪----   ------
There is always at least 1 Global Group and 1 Data Group                        Ok
There are at least 2 data nodes in a Data Group (except for the witness-only group)␣
↪Critical group_b
There is at most 1 witness node in a Data Group                                ␣
↪Warning  group_a
Witness-only group does not have any child groups                               Ok
There is at max 1 witness-only group iff there is even number of local Data Groups ␣
↪Warning  bdrgroup
There are at least 2 proxies configured per Data Group if routing is enabled   ␣
↪Warning  group_a, group_b

Example 2 (cluster with recommended architecture)
```

(次のページに続く)

(前のページからの続き)

```
$ pgd verify-cluster

Check                                                                          ␣
↪Status Groups
-----                                                                           --
↪---- ------
There is always at least 1 Global Group and 1 Data Group                       Ok
There are at least 2 data nodes in a Data Group (except for the witness-only group) Ok
There is at most 1 witness node in a Data Group                                Ok
Witness-only group does not have any child groups                              Ok
There is at max 1 witness-only group iff there is even number of local Data Groups  Ok
There are at least 2 proxies configured per Data Group if routing is enabled   Ok
```

オプション

```
-h, --help   help for verify-cluster
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

verify-settings

EDB Postgres 分散クラスター設定を確認します。

概要

EDB Postgres 分散クラスター設定を確認します。

```
pgd verify-settings [flags]
```

例

```
Example 1

$ pgd verify-settings

# bdr.accept_connections
Node                Status   Pending Restart Value Message
----                ------   --------------- ----- -------
bdr-a1              Critical false           off   must be set to on
bdr-a2              Ok       false           on
bdr-b1              Ok       false           on
bdr-b2              Ok       false           on
logical-standby-a1 Ok       false           on
logical-standby-b1 Ok       false           on
subscriber-only-c1 Ok       false           on
witness-a           Ok       false           on
witness-b           Ok       false           on
witness-c           Ok       false           on
Warning: value must be same on all primary nodes

# bdr.ddl_locking
Ok: all node values are ok

# bdr.ddl_replication
Node                Status  Pending Restart Value Message
----                ------  --------------- ----- -------
bdr-a1              Warning false           0     must be set to on
bdr-a2              Ok      false           on
bdr-b1              Ok      false           on
bdr-b2              Ok      false           on
logical-standby-a1 Ok      false           on
logical-standby-b1 Ok      false           on
subscriber-only-c1 Ok      false           on
witness-a           Ok      false           on
witness-b           Ok      false           on
witness-c           Ok      false           on
Warning: value must be same on all primary nodes

# bdr.max_writers_per_subscription
Ok: all node values are ok

# bdr.raft_group_max_connections
Ok: all node values are ok
```

(次のページに続く)

(前のページからの続き)

```
# bdr.replay_progress_frequency
Node                Status  Pending Restart Value Message
----                ------  --------------- ----- -------
bdr-a1              Warning false           61000 must be <= 60000
bdr-a2              Ok      false           60000
bdr-b1              Ok      false           60000
bdr-b2              Ok      false           60000
logical-standby-a1 Ok      false           60000
logical-standby-b1 Ok      false           60000
subscriber-only-c1 Ok      false           60000
witness-a           Ok      false           60000
witness-b           Ok      false           60000
witness-c           Ok      false           60000
Warning: value must be same on all primary nodes

# bdr.role_replication
Node                Status  Pending Restart Value Message
----                ------  --------------- ----- -------
bdr-a1              Warning false           off   must be set to on
bdr-a2              Ok      false           on
bdr-b1              Ok      false           on
bdr-b2              Ok      false           on
logical-standby-a1 Ok      false           on
logical-standby-b1 Ok      false           on
subscriber-only-c1 Ok      false           on
witness-a           Ok      false           on
witness-b           Ok      false           on
witness-c           Ok      false           on
Warning: value must be same on all primary nodes

# bdr.standby_slot_names
Node                Status  Pending Restart Value         Message
----                ------  --------------- -----         -------
bdr-a1              Warning false           bdr_bdrdb_ja… must contain valid logical␣
↪slots of peer data nodes only
bdr-a2              Warning false           bdr_bdrdb_ja… must contain valid logical␣
↪slots of peer data nodes only
bdr-b1              Warning false                         must contain valid logical␣
↪slots of peer data nodes only
bdr-b2              Warning false                         must contain valid logical␣
↪slots of peer data nodes only
logical-standby-a1 Ok      false
logical-standby-b1 Ok      false
subscriber-only-c1 Ok      false
witness-a           Ok      false
```

(次のページに続く)

(前のページからの続き)

```
witness-b          Ok      false
witness-c          Ok      false

# bdr.standby_slots_min_confirmed
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           -1    must be >= 1
bdr-a2             Warning false           -1    must be >= 1
bdr-b1             Warning false           -1    must be >= 1
bdr-b2             Warning false           -1    must be >= 1
logical-standby-a1 Ok      false           -1
logical-standby-b1 Ok      false           -1
subscriber-only-c1 Ok      false           -1
witness-a          Ok      false           -1
witness-b          Ok      false           -1
witness-c          Ok      false           -1

# bdr.start_workers
Ok: all node values are ok

# bdr.xact_replication
Ok: all node values are ok

# max_prepared_transactions
Node               Status  Pending Restart Value Message
----               ------  --------------- ----- -------
bdr-a1             Warning false           16    must be >= 250
bdr-a2             Warning false           16    must be >= 250
bdr-b1             Warning false           16    must be >= 250
bdr-b2             Warning false           16    must be >= 250
logical-standby-a1 Warning false           16    must be >= 250
logical-standby-b1 Warning false           16    must be >= 250
subscriber-only-c1 Warning false           16    must be >= 250
witness-a          Warning false           16    must be >= 250
witness-b          Warning false           16    must be >= 250
witness-c          Warning false           16    must be >= 250

# max_replication_slots
Node               Status   Pending Restart Value Message
----               ------   --------------- ----- -------
bdr-a1             Critical false            8     must be >= 10
bdr-a2             Ok       false           12
bdr-b1             Ok       false           12
bdr-b2             Ok       false           12
logical-standby-a1 Ok       false           12
```

(次のページに続く)

(前のページからの続き)

```
logical-standby-b1 Ok       false           12
subscriber-only-c1 Ok       false           12
witness-a          Ok       false           12
witness-b          Ok       false           12
witness-c          Ok       false           12
Warning: value must be same on all primary nodes

# max_wal_senders
Ok: all node values are ok

# max_worker_processes
Ok: all node values are ok

# shared_preload_libraries
Node               Status  Pending Restart Value         Message
----               ------  --------------- -----         -------
bdr-a1             Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
bdr-a2             Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
bdr-b1             Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
bdr-b2             Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
logical-standby-a1 Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
logical-standby-b1 Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
subscriber-only-c1 Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
witness-a          Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
witness-b          Warning false           pg_stat_stat… must contain bdr as first␣
↪entry
witness-c          Warning false           pg_stat_stat… must contain bdr as first␣
↪entry

# track_commit_timestamp
Ok: all node values are ok

# wal_level
Ok: all node values are ok
```

オプション

```
-h, --help   help for verify-settings
```

親コマンドから継承されたオプション

```
-f, --config-file string   config file; ignored if
                           --dsn flag is present (default "/etc/edb/pgd-cli/pgd-cli-
↪config.yml")
    --dsn string           database connection string
                           e.g."host=bdr-a1 port=5432 dbname=bdrdb user=postgres "
-L, --log-level string     logging level: debug, info, warn, error (default "error")
-o, --output string        output format: json, yaml
```

8.15 Transaction streaming

論理レプリケーションでは、トランザクションはパブリッシャーで同時にデコードされますが、トランザクションがコミットされるまでサブスクライバーには送信されません。変更が `logical_decoding_work_mem`（PostgreSQL 13 以降）を超える場合、それらはディスクに書き込まれます。つまり、特に大規模なトランザクションでは、サブスクライバーに到達するまでに遅延があり、パブリッシャーで追加の I / O が必要になる場合があります。

PostgreSQL 14 以降、トランザクションは、パブリッシャーでコミットされる前に、オプションでデコードしてサブスクライバーに送信できます。サブスクライバーは、着信変更をステージングファイル（またはファイルのセット）に保存し、トランザクションがコミットするときにそれらを適用します（またはトランザクションがアボートする場合は破棄します）。これにより、トランザクションがコミットされるとすぐに、サブスクライバーにトランザクションを適用できます。

8.15.1 PGD の機能強化

PostgreSQL の組み込みトランザクションストリーミングには、次の制限があります。

- パブリッシャーのディスクに変更を書き込む必要はなくなりましたが、各サブスクライバーのディスクに変更を書き込む必要があります。
- トランザクションが中止された場合、作業（各サブスクライバーが受け取った変更と関連するストレージ I / O）は無駄になります。

ただし、バージョン 3.7 以降、PGD は並列適用をサポートし、各サブスクライバーで複数のライタプロセスを有効にします。この機能を活用して、次の拡張機能を提供します。

- デコードされたトランザクションは、サブスクライバーのライタに直接ストリーミングできます。
- デコードされたトランザクションは、サブスクライバーのディスクに保存する必要はありません。

- サブスクライバーでトランザクションの適用を開始する前に、トランザクションがコミットするのを待つ必要はありません。

注意事項

- パラレル適用を有効にする必要があります。
- 多くの小さくて競合するトランザクションで構成されるワークロードは、ライター間で頻繁にデッドロックを引き起こす可能性があります。

注釈: ライタへのダイレクトストリーミングはまだ実験的な機能です。注意して使用してください。具体的には、ストリーミングのコミットタイムスタンプを利用できない場合があるため、競合解決ではうまく機能しない場合があります。（トランザクションはオリジンでまだコミットされていない可能性があります。）

8.15.2 構成

次の 2 つの場所でトランザクションストリーミングを構成します。

- ノードレベルで、GUC *bdr.default_streaming_mode* を使用
- グループレベルで、関数 *bdr.alter_node_group_config()* **を使用したグループ構成** を使用

bdr.default_streaming_mode を使用したノード構成

許可される値は次のとおりです。

- `off`
- `writer`
- `file`
- `auto`

デフォルト値は `auto` です。

この設定を変更するには、設定を有効にするには、サブスクリプションごとに pglogical レシーバープロセスを再起動する必要があります。サーバーの再起動でこれを実現できます。

`bdr.default_streaming_mode` が `off` 以外の値に設定されている場合、サブスクライバーはパブリッシャーからのトランザクションストリーミングを要求します。これがどのように提供されるかは、グループ構成設定によっても異なります。詳細は *bdr.default_streaming_mode* **を使用したノード構成** を参照してください。

bdr.alter_node_group_config() を使用したグループ構成

関数 *bdr.alter_node_group_config()* **を使用したグループ構成** でパラメーター `streaming_mode` を使用できます

グループトランザクションストリーミング構成を設定します。

許可される値は次のとおりです。

- `off`
- `writer`
- `file`
- `auto`
- `default`

デフォルト値は `default` です。

現在の設定の値は、ビュー *bdr.node_group_config* の列 `node_group_streaming_mode` に含まれています。返される値は単一の char 型で、可能な値は次のとおりです

構成設定の効果

トランザクションストリーミングは、GUC `bdr.default_streaming_mode` によってサブスクライバーレベルで制御されます。トランザクションストリーミングを無効にする `off` に設定されていない限り、サブスクライバーはトランザクションストリーミングを要求します。

パブリッシャーがトランザクションストリーミングを提供できる場合、トランザクションサイズが `logical_decoding_work_mem` で設定されたしきい値を超えるたびにトランザクションをストリーミングします。パブリッシャーは通常、トランザクションがファイルにストリーミングされるかライタにストリーミングされるかを制御できません。一部の状況（COPY など）を除き、サブスクライバーがトランザクションをライタにストリーミングすることを示唆する場合があります（可能な場合）。

サブスクライバーは、パブリッシャーから受け取ったトランザクションをライタまたはファイルにストリーミングできます。この決定は、いくつかの要因に基づいています。

- 並列適用がオフ（`num_writers = 1`）の場合、ファイルにストリーミングされます。（ライタ 0 は常に非ストリーミングトランザクション用に予約されています。）
- 並列適用がオンになっているが、すべてのライターがストリーミングされたトランザクションの処理ですでにビジー状態である場合、新しいトランザクションはファイルにストリーミングされます。PGD ライタのステータスを確認するには、*PGD* **ライターの監視** を参照してください。

ライタへのストリーミングが可能な場合（つまり、無料のライタが利用可能である場合）、トランザクションをライタにストリーミングするかファイルにストリーミングするかは、次の表のようにグループとノードの設定の組み合わせに基づいて決定されます。

Group	Node	Streamed to
off	(any)	(none)
(any)	off	(none)
writer	file	file
file	writer	file
default	writer	writer
default	file	file
default	auto	writer
auto	(any)	writer

グループ構成が `auto` に設定されている場合、またはグループ構成が `default` でノード構成が `auto` の場合、パブリッシャーがこれを行うことを示唆した場合にのみ、トランザクションはライタにストリーミングされます。

現在、パブリッシャーは、サブスクライバーが次のトランザクションタイプのライタにストリーミングするようにヒントしています。これらは競合がないことが知られており、ライタが安全に処理できます。

- `COPY`
- `CREATE INDEX CONCURRENTLY`

8.15.3 モニタリング

bdr.stat_subscription を使用して、トランザクションストリーミングの使用を監視できます

サブスクライバーノードの機能。

- `nstream_writer` —ライタにストリーミングされたトランザクションの数。
- `nstream_file` —ファイルにストリーミングされたトランザクションの数。
- `nstream_commit` —コミットされたストリーミングトランザクションの数。
- `nstream_abort` —中止されたストリーミングトランザクションの数。
- `nstream_start` —開始されたストリーミングトランザクションの数。
- `nstream_stop` —完全に受信されたストリーミングトランザクションの数。

8.16 Stream triggers

PGD には、ダウンストリーム/ターゲットノードでの追加のデータ処理に使用できる新しいタイプのトリガーが導入されています。

- コンフリクトトリガー
- 変身トリガー

これらのタイプのトリガーは、 **ストリームトリガー** と呼ばれます。

ストリームトリガーは、構文がトリガーのように設計されています。これらは PostgreSQL BEFORE トリガーアーキテクチャを活用しており、PostgreSQL BEFORE トリガーと同様のパフォーマンス特性を持つ可能性があります。

通常の PostgreSQL トリガーと同様に、複数のトリガー定義で 1 つのトリガー関数を使用できます。トリガー関数は、次の形式で定義されたプログラムです：`CREATE FUNCTION ... RETURNS TRIGGER`。トリガーを作成するために、 `CREATE TRIGGER` コマンドを使用する必要はありません。代わりに、特別な PGD 関数

> *bdr.create_conflict_trigger* および *bdr.create_transform_trigger* を使用してストリームトリガーを作成します。

作成されると、トリガーはカタログ表 `pg_trigger` で可視になります。ストリームトリガーは `tgisinternal = true` および `tgenabled = 'D'` としてマークされ、名前サフィックス'_bdrc'または'_bdrt'が付いています。ビュー `bdr.triggers` は、テーブルに関連するトリガー、実行されているプロシージャの名前、それをトリガーするイベント、およびトリガータイプに関する情報を提供します。

ストリームトリガーは、通常の SQL 処理では有効になっていません。このため、`ALTER TABLE ... ENABLE TRIGGER` は、その特定の名前バリアントと ALL バリアントの両方でストリームトリガーに対してブロックされます。このメカニズムにより、トリガーが通常の SQL トリガーとして実行されなくなります。

これらのトリガーは、ダウンストリームまたはターゲットノードで実行されます。オリジンノードで実行するオプションはありません。ただし、オリジンでの `row_filter` 式の使用を検討することをお勧めします。

また、ストリームトリガーの実行中に適用される DML は他の PGD ノードに複製されず、標準のローカルトリガーの実行をトリガーしません。これは意図的です。これを使用して、たとえば、ストリームトリガーによってキャプチャされた変更または競合を、クラッシュセーフでそのノードに固有のテーブルにログ記録できます。実例については、
ストリームトリガーの例 を参照してください。

8.16.1 Apply 中に実行をトリガーする

変換トリガーは、トリガーテーブル内の受信変更ごとに最初に 1 回実行されます。これらのトリガーは、一致するターゲット行を見つけようとする前に起動されるため、非常に幅広い変換を効率的かつ一貫して適用できます。

次に、 UPDATE および DELETE の変更については、ターゲット行を見つけます。ターゲット行がない場合、それらの変更タイプに対してそれ以上の処理は発生しません。

次に、以前にテーブルレベルでレプリカトリガーとして明示的に有効にされていた通常のトリガーを実行します。

```
ALTER TABLE tablename
ENABLE REPLICA TRIGGER trigger_name;
```

次に、潜在的な競合が存在するかどうかを判断します。その場合、そのテーブルに存在する競合トリガーを呼び出します。

欠落している列の競合の解決

変換トリガーが実行される前に、PostgreSQL は受信タプルをターゲットテーブルの行タイプと一致させようとします。

入力行に存在し、ターゲットテーブルに存在しない列は、target_column_missing 型の競合をトリガーします。逆に、ターゲットテーブルに存在するが受信行にない列は、source_column_missing 競合をトリガーします。これら 2 つの競合タイプのデフォルトの解決は、それぞれ ignore_if_null と use_default_value です。

これは、ローリングスキーマアップグレードのコンテキスト、たとえば、スキーマの新しいバージョンで新しい列が導入される場合に関連します。スキーマの古いバージョンから新しいバージョンにレプリケートする場合、ソース列が欠落しており、新しく導入された列にデフォルト値を設定するため、use_default_value 戦略が適切です。

ただし、新しいスキーマバージョンを持つノードから古いスキーマバージョンを持つノードに複製すると、ターゲットテーブルに列がありません。ignore_if_null リゾルバーは、ユーザーがアップグレードされたノードの新しい列に NULL 以外の値を持つタプルを挿入するとすぐにレプリケーションを中断するため、ローリングアップグレードには適していません。

この例を考慮すると、ローリングスキーマアップグレードの適切な設定は、target_column_missing の競合が発生した場合に ignore リゾルバーを適用するように各ノードを構成することです。

これを行うには、次のクエリを使用します。各ノードで個別に実行する必要があります。node1 を実際のノード名に置き換えます。

```
SELECT bdr.alter_node_set_conflict_resolver(node1,
    target_column_missing, ignore);
```

データ損失と発散リスク

競合リゾルバーを ignore に設定すると、データの損失とクラスターの発散につながる可能性があります。

次の例を考えます。表 t はノード 1 および 2 に存在しますが、その列 col はノード 1 にのみ存在します。

競合リゾルバーが ignore に設定されている場合、ノード 1 に c が null でない行、たとえば、(pk=1, col=100) が存在する可能性があります。その行はノード 2 に複製され、列 c の値は破棄されます（例、(pk=1) ）。

c 列がノード 2 のテーブルに追加される場合、それは最初に既存のすべての行で NULL に設定され、上記で検討された行は (pk=1, col=NULL) になります。 pk=1 を持つ行は、すべてのノードで同一ではなくなったため、クラスターは発散します。

ノード 2 にレプリケートされる行には col=NULL があるため、デフォルトの ignore_if_null リゾルバーはこのリスクの影響を受けません。

この例に基づいて、 ignore リゾルバーが使用されたスキーマのローリングアップグレードの最後に、クラスター全体に対して LiveCompare を実行することをお勧めします。このプラクティスは、発散を確実に検出して修正するのに役立ちます。

8.16.2 行型の用語

これらの行タイプを使用します。

・ SOURCE_OLD は更新前の行、つまりキーです。

- SOURCE_NEW は、別のノードからの新しい行です。
- TARGET は、ノードにすでに存在する行、つまり競合している行です。

8.16.3 競合トリガー

競合トリガーは、PGD によって競合が検出されると実行されます。彼らは、紛争が発生したときにどうするかを決定します。

- トリガー関数が行を返す場合、アクションはターゲットに適用されます。
- トリガー関数が NULL 行を返した場合、アクションはスキップされます。

たとえば、トリガーが DELETE に対して呼び出された場合、DELETE をスキップする場合、トリガーは NULL を返します。 DELETE を続行したい場合は、行の値を返します。SOURCE_OLD または TARGET が機能します。競合する操作が INSERT または UPDATE のいずれかであり、選択された解決策が競合する行を削除することである場合、トリガーは明示的に削除を実行し、NULL を返す必要があります。トリガー関数は、選択に応じて他の SQL アクションを実行できますが、これらのアクションはローカルにのみ適用され、レプリケートはされません。

2 つ以上のノード間で実際のデータの競合が発生する場合、2 つ以上の同時変更が発生しています。変更が適用されると、競合の解決は各ノードで独立して行われます。これは、競合の解決が各ノードで 1 回発生し、ノード間に大きな時間差がある場合に発生する可能性があることを意味します。その結果、競合トリガーの複数の実行間の通信はできません。トリガーが関連するすべてのイベントに対してまったく同じ結果をもたらすことを保証するのは、競合トリガーの作成者の責任です。そうしないと、データの相違が発生します。テクニカルサポートは、PGD で提供される isolationtester ツールを使用して、すべての競合トリガーを正式にテストすることをお勧めします。

警告:

- 1 つのテーブルに複数の競合トリガーを指定できますが、個別のイベントと一致する必要があります。つまり、各競合は単一の競合トリガーのみと一致する必要があります。 -同じテーブルの同じイベントに一致する複数のトリガーはお勧めしません。それらは一貫性のない動作を引き起こす可能性があり、将来のリリースでは許可されません。

同じ競合トリガーが複数のイベントと一致する場合、トリガーで TG_OP 変数を使用して、競合を生成した操作を特定できます。

デフォルトでは、PGD は行のレプリケーション起点の変更を監視することにより競合を検出します。したがって、1 つの変更のみが発生している場合でも、競合トリガーを呼び出すことができます。この場合、実際の競合はないため、この競合検出メカニズムは誤検出の競合を生成する可能性があります。競合トリガーは、これらすべてを同じように処理する必要があります。

場合によっては、タイムスタンプの競合検出で競合がまったく検出されません。たとえば、DELETE が UPDATE の直後に発生する同時実行 UPDATE /DELETE では、最初に UPDATE 、次に DELETE を見るノードは競合を見ません。競合が見られない場合、競合トリガーは呼び出されません。同じ状況で、行バージョンの競合検出を使用すると、競合が発生し、競合トリガーで処理できます。

トリガー関数は、操作の種類に応じて、追加の状態情報と競合に関連するデータ行にアクセスできます。

- INSERT では、競合トリガーはソースから SOURCE_NEW 行および TARGET 行にアクセスできます。
- UPDATE では、競合トリガーはソースおよび TARGET 行から SOURCE_OLD および SOURCE_NEW 行にアクセスできます。
- DELETE では、競合トリガーはソースから SOURCE_OLD 行および TARGET 行にアクセスできます。

その操作の値が存在する場合、ファンクション bdr.trigger_get_row() を使用して、 SOURCE_OLD 、SOURCE_NEW 、または TARGET 行を取得できます。

競合トリガーへの変更はトランザクション的に発生し、 ALTER TABLE の一部のバリアントが処理される方法と同様に、構成変更のレプリケーション中にグローバル DML ロックによって保護されます。

競合トリガー内で主キーが更新されると、実行タイミングの違いによりユニーク制約違反エラーにつながる場合があります。したがって、競合トリガーで主キーを更新しないでください。

8.16.4 トリガーの変換

これらのトリガーは、特定のテーブルに対してデータストリームのすべての行に対して実行されることを除いて、競合トリガーに似ています。戻り値と公開された変数の動作は似ていますが、変換トリガーはターゲット行が識別される前に実行されるため、TARGET 行はありません。

PGD の各テーブルに複数の変換トリガーを指定できます。変換トリガーはアルファベット順に実行されます。

変換トリガーは、行をフィルター処理して、必要に応じて追加の操作を実行できます。任意の列の値を変更したり、それらを NULL に設定したりできます。戻り値は、実行されるさらなるアクションを決定します。

- トリガー関数が行を返す場合、ターゲットに適用されます。
- トリガー関数が NULL 行を返した場合、これ以上実行するアクションはありません。未実行のトリガーは実行されません。
- トリガー機能は、必要に応じて他のアクションを実行できます。

トリガー関数は、追加の状態情報と競合に関連する行にアクセスできます。

- INSERT では、変換トリガーはソースから SOURCE_NEW 行にアクセスできます。
- UPDATE では、変換トリガーはソースから SOURCE_OLD および SOURCE_NEW 行にアクセスできます。
- DELETE では、変換トリガーはソースから SOURCE_OLD 行にアクセスできます。

関数 bdr.trigger_get_row() を使用して、 SOURCE_OLD または SOURCE_NEW 行を取得できます。このタイプのトリガーは、そのようなターゲット行が識別される前に実行されるため、TARGET 行は利用できません。

変換トリガーは、通常の BEFORE 行トリガーと非常によく似ていますが、次の重要な違いがあります。

- 変換トリガーは、着信変更ごとに呼び出されます。 BEFORE UPDATE に対してトリガーはまったく呼び出されません。テーブル内に一致する行が見つからない場合、DELETE が変更されます。
- パーティションテーブルのルーティングが発生する前に、変換トリガーが呼び出されます。
- 変換トリガーは、通常の SQL トリガーでは使用できない SOURCE_OLD を介してルックアップキーにアクセスできます。

8.16.5 行の内容

SOURCE_NEW 、SOURCE_OLD 、および TARGET の内容は、オペレーション、テーブルの REPLICA IDENTITY 設定、およびターゲットテーブルの内容に依存します。

TARGET 行は、競合トリガーでのみ使用できます。TARGET 行には、ターゲットテーブルに UPDATE または DELETE を適用したときに行が見つかった場合にのみデータが含まれます。行が見つからない場合、TARGET は NULL です。

8.16.6 実行順序

トリガーの実行順序：

- トリガーを変換する -ターゲットの受信行ごとに 1 回実行します。
- 通常のトリガー - 1 行に 1 回実行します。
- 競合トリガー -競合が存在する行ごとに 1 回実行します。

8.16.7 ストリームトリガーの例

update_if_newer 競合リゾルバーと同様の動作を提供する競合トリガー：

```
CREATE OR REPLACE FUNCTION update_if_newer_trig_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF (bdr.trigger_get_committs(TARGET) >
        bdr.trigger_get_committs(SOURCE_NEW)) THEN
    RETURN TARGET;
    ELSIF
        RETURN SOURCE;
    END IF;
END;
$$;
```

カウンター列にデルタ変更を適用し、他のすべての列に SOURCE_NEW を使用する競合トリガー。

```
CREATE OR REPLACE FUNCTION delta_count_trg_func
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    DELTA bigint;
    SOURCE_OLD record;
    SOURCE_NEW record;
    TARGET record;
BEGIN
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    TARGET := bdr.trigger_get_row(TARGET);

    DELTA := SOURCE_NEW.counter - SOURCE_OLD.counter;
    SOURCE_NEW.counter = TARGET.counter + DELTA;

    RETURN SOURCE_NEW;
END;
$$;
```

すべての変更を適用する代わりにログテーブルに記録する変換トリガー。

```
CREATE OR REPLACE FUNCTION log_change
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE_NEW record;
    SOURCE_OLD record;
    COMMITTS timestamptz;
BEGIN
    SOURCE_NEW := bdr.trigger_get_row(SOURCE_NEW);
    SOURCE_OLD := bdr.trigger_get_row(SOURCE_OLD);
    COMMITTS := bdr.trigger_get_committs(SOURCE_NEW);

    IF (TG_OP = INSERT) THEN
        INSERT INTO log SELECT I, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = UPDATE) THEN
        INSERT INTO log SELECT U, COMMITTS, row_to_json(SOURCE_NEW);
    ELSIF (TG_OP = DELETE) THEN
        INSERT INTO log SELECT D, COMMITTS, row_to_json(SOURCE_OLD);
    END IF;

    RETURN NULL; -- do not apply the change
```

(次のページに続く)

(前のページからの続き)

```
END;
$$;
```

この例は、信頼できるソースの競合検出を実装する競合トリガーを示しています。信頼できるサイト、優先ノード、または Always Wins 解決とも呼ばれます。 bdr.trigger_get_origin_node_id() 関数を使用して、3 つ以上のノードで動作するソリューションを提供します。

```
CREATE OR REPLACE FUNCTION test_conflict_trigger()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
DECLARE
    SOURCE  record;
    TARGET  record;

    TRUSTED_NODE    bigint;
    SOURCE_NODE     bigint;
    TARGET_NODE     bigint;
BEGIN
    TARGET := bdr.trigger_get_row(TARGET);
    IF (TG_OP = DELETE)
        SOURCE := bdr.trigger_get_row(SOURCE_OLD);
    ELSE
        SOURCE := bdr.trigger_get_row(SOURCE_NEW);
    END IF;

    TRUSTED_NODE := current_setting(customer.trusted_node_id);

    SOURCE_NODE := bdr.trigger_get_origin_node_id(SOURCE_NEW);
    TARGET_NODE := bdr.trigger_get_origin_node_id(TARGET);

    IF (TRUSTED_NODE = SOURCE_NODE) THEN
        RETURN SOURCE;
    ELSIF (TRUSTED_NODE = TARGET_NODE) THEN
        RETURN TARGET;
    ELSE
        RETURN NULL; -- do not apply the change
    END IF;
END;
$$;
```

8.17 AutoPartition

AutoPartition を使用すると、テーブルをいくつかのパーティションに分割できます。自動パーティション管理を使用して、テーブルを大きなサイズに簡単に拡張できます。この機能では、パーティションの作成と削除の競合の少ないロックなど、PGD の機能を使用します。

新しいパーティションを定期的に作成し、データ保持期間が終了したときに削除できます。

PGD 管理は、主に SQL で呼び出すことができる関数によって達成されます。PGD のすべての関数は、bdr スキーマで公開されています。search_path に入れない限り、各関数の名前をスキーマ修飾する必要があります。

8.17.1 パーティションの自動作成

bdr.autopartition() は、表の自動レンジパーティション化の定義を作成または変更します。定義が存在しない場合は、作成されます。そうしないと、後の実行で定義が変更されます。

bdr.autopartition() は実際のテーブルをロックしません。新しいパーティションのメンテナンスアクションをいつ、どのように行うかの定義を変更します。

bdr.autopartition() は、テーブルの残りの部分をロックせずにパーティションをアタッチまたはデタッチ/ドロップできる機能を活用します（基になる Postgres バージョンがサポートしている場合）。

テーブルが RANGE パーティション分割されていない場合、またはマルチカラムパーティションキーが使用されている場合、ERROR が発生します。

デフォルトでは、AutoPartition はパーティションをグローバルに管理します。言い換えれば、1 つのノードでパーティションが作成されると、クラスター内の他のすべてのノードに同じパーティションが作成されます。デフォルトを使用すると、すべてのパーティションに一貫性があり、使用できることが保証されます。この機能のために、AutoPartition は Raft を利用します。

`managed_locally` を `true` に設定することで、この動作を変更できます。その場合、すべてのパーティションは各ノードでローカルに管理されます。パーティションをローカルで管理することは、パーティションテーブルがレプリケートテーブルでない場合に役立ちます。たとえば、組み込みの *bdr.conflict_history* テーブルはレプリケートテーブルではありません。AutoPartition によってローカルに管理されています。各ノードはこのテーブルのパーティションをローカルに作成し、十分に古くなると削除します。

次の点も考慮してください。

- `managed_locally` としてマークされたテーブルをグローバルに管理するように後で変更することはできません。その逆も同様です。
- アクティビティは、エントリが `enabled = on` とマークされている場合にのみ実行されます。
- AutoPartition で管理されるテーブルのパーティションを手動で作成または削除しないことをお勧めします。これを行うと、AutoPartition メタデータに一貫性がなくなり、失敗する可能性があります。

8.17.2 自動パーティションの例

デイリーパーティション、1 か月間データを保持：

```
CREATE TABLE measurement (
logdate date not null,
peaktemp int,
unitsales int
) PARTITION BY RANGE (logdate);

bdr.autopartition(measurement, 1 day, data_retention_period := 30 days);
```

残りのパーティションが 2 つだけになったら、5 つのアドバンスパーティションを作成します。各パーティションは 10 億オーダーを保持できます。

```
bdr.autopartition(Orders, 1000000000,
        partition_initial_lowerbound := 0,
        minimum_advance_partitions := 2,
        maximum_advance_partitions := 5
    );
```

8.17.3 RANGE パーティションテーブル

partition_increment の値の範囲ごとに新しいパーティションが追加され、下限と上限が partition_increment 離れています。 timestamp または date 型のパーティションキーを持つテーブルの場合、partition_increment は interval 型の有効な定数でなければなりません。たとえば、1 Day を指定すると、新しいパーティションが毎日追加され、パーティション境界が 1 日離れます。

パーティション列が snowflakeid 、timeshard 、または ksuuid シーケンスに接続されている場合、partition_increment を interval タイプとして指定する必要があります。それ以外の場合、パーティションキーが整数または数値の場合、partition_increment は同じデータ型の有効な定数でなければなりません。たとえば、 1000000 を指定すると、100 万個の値ごとに新しいパーティションが追加されます。

テーブルに既存のパーティションがない場合、指定された partition_initial_lowerbound が最初のパーティションの下限として使用されます。partition_initial_lowerbound を指定しない場合、システムはパーティション列タイプと指定された partition_increment から値を導き出そうとします。たとえば、 partition_increment が 1 Day として指定されている場合、partition_initial_lowerbound は CURRENT DATE に設定されます。partition_increment が 1 Hour として指定されている場合、partition_initial_lowerbound は現在の日付の現在の時間に設定されます。後続のパーティションの境界は、partition_increment 値を使用して設定されます。

システムは常に、特定の最小数のアドバンスパーティションを作成しようとします。新しいパーティションを作成するかどうかを決定するために、指定された partition_autocreate_expression を使用します。これは、チェックが実行されるたびに評価される SQL で評価できる式にすることができます。たとえば、列型 date のパーティションテーブルの場合、 partition_autocreate_expression が DATE_TRUNC('day',CURRENT_DATE) として指定され、partition_increment が 1 Day として指定され、minimum_advance_partitions が 2 として指定されている場合、最後のパーティションの上限が

DATE_TRUNC('day', CURRENT_DATE) + '2 Days'::interval 未満になるまで新しいパーティションが作成されます。

式は、システムが新しいパーティションをチェックするたびに評価されます。

列型 integer のパーティションテーブルの場合、partition_autocreate_expression を SELECT max(partcol) FROM schema.partitioned_table として指定できます。次に、システムは、パーティション分割された列の最大値が最後のパーティションの上限の minimum_advance_partitions * partition_increment の距離内にあるかどうかを定期的にチェックします。 partcol にインデックスを作成して、クエリが効率的に実行されるようにします。列型 integer 、smallint 、または bigint のパーティションテーブルに partition_autocreate_expression を指定しない場合、システムはそれを max(partcol) に設定します。

data_retention_period が設定されている場合、この期間の後にパーティションが削除されます。ロックを最小限に抑えるために、新しいパーティションが追加されると同時にパーティションが削除されます。この値が設定されていない場合、パーティションを手動で削除する必要があります。

data_retention_period パラメーターは、タイムスタンプ（および関連する）ベースのパーティションでのみサポートされています。期間は、パーティションの上限を考慮して計算されます。上限に対して、指定された期間が満了すると、パーティションは削除されます。

8.17.4 パーティションの自動作成の停止

bdr.drop_autopartition() を使用して、指定されたリレーションの自動パーティションルールを削除します。リレーションの保留中のすべての作業項目が削除され、新しい作業項目は作成されません。

8.17.5 パーティションの作成を待っています

パーティションの作成は非同期プロセスです。AutoPartition は、ローカルまたはすべてのノードでパーティションが作成されるのを待機する一連の機能を提供します。

bdr.autopartition_wait_for_partitions() を使用して、ローカルノードでパーティションが作成されるのを待ちます。この関数は、パーティション分割テーブル名とパーティションキー列の値を受け取り、その値を保持するパーティションが作成されるまで待機します。

この関数は、パーティションがローカルに作成されるまでのみ待機します。パーティションがリモートノードにも存在することを保証するものではありません。

すべての PGD ノードでパーティションが作成されるのを待つには、

bdr.autopartition_wait_for_partitions_on_all_nodes() 関数を使用します。この関数は、ローカルノードとすべてのリモートノードを内部的にチェックし、パーティションがどこにでも作成されるまで待機します。

8.17.6 パーティションを見つける

bdr.autopartition_find_partition() ファンクションを使用して、指定されたパーティションキー値のパーティションを見つけます。その値を保持するパーティションが存在しない場合、関数は NULL を返します。それ以外の場合は、パーティションの Oid が返されます。

8.17.7 AutoPartitioning の有効化または無効化

bdr.autopartition_enable() を使用して、指定されたテーブルで AutoPartitioning を有効にします。

AutoPartitioning がすでに有効になっている場合、アクションは発生しません。同様に、
bdr.autopartition_disable()

を使用して、指定されたテーブルの AutoPartitioning を無効にします。

8.18 Explicit two-phase commit (2PC)

アプリケーションは、PGD で 2 フェーズコミットを使用することを明示的に選択できます。
Distributed Transaction Processing: The XA Specification を参照してください。

X/Open Distributed Transaction Processing (DTP) モデルは、3 つのソフトウェアコンポーネントを想定しています。

- トランザクション境界を定義し、トランザクションを構成するアクションを指定するアプリケーションプログラム (AP)
- 共有リソースへのアクセスを提供するリソースマネージャー（データベースやファイルアクセスシステムなどの RM）
- トランザクションに識別子を割り当て、その進行状況を監視し、トランザクションの完了と障害回復に責任を負うトランザクションマネージャー（TM）と呼ばれる別個のコンポーネント

PGD は、 `PREPARE TRANSACTION` および `COMMIT PREPARED` /`ROLLBACK PREPARED` コマンドを使用した明示的な外部 2PC をサポートしています。外部的には、 EDB Postgres 分散クラスターは、単一セッションのトランザクションマネージャーにとって単一のリソースマネージャーに見えます。

`bdr.commit_scope` が `local` の場合、トランザクションはローカルノードでのみ準備されます。コミットされると、変更がレプリケートされ、PGD はコミット後の競合解決を適用します。

`bdr.commit_scope` を `local` に設定して使用すると、明示的な 2 フェーズコミットでは意味がないように見える場合があります。ただし、トランザクションの待機時間と堅牢性の間のトレードオフを制御できるオプションが提供されています。

明示的な 2 フェーズコミットは、CAMO またはグローバルコミットスコープでは機能しません。将来のリリースでは、この組み合わせが有効になる可能性があります。

8.18.1 使用する

ローカルコミットスコープを持つ 2 フェーズコミットは、標準の PostgreSQL とまったく同じように機能します。ローカルコミットスコープを使用し、CAMO を無効にします。

```
BEGIN;

SET LOCAL bdr.enable_camo = off;
SET LOCAL bdr.commit_scope = local;

... other commands possible...
```

コミットの最初のフェーズを開始するには、クライアントはグローバルトランザクション ID を割り当てる必要があります。これは、トランザクションを識別する一意の文字列です。

```
PREPARE TRANSACTION some-global-id;
```

最初のフェーズが成功すると、すべてのノードが変更を適用し、トランザクションをコミットする準備が整います。次に、クライアントは同じノードから 2 番目のフェーズを呼び出す必要があります。

```
COMMIT PREPARED some-global-id;
```

8.19 Timestamp-based snapshots

タイムスタンプベースのスナップショットを使用すると、通常の MVCC スナップショットではなく、ユーザー指定のタイムスタンプを使用して、一貫した方法でデータを読み取ることができます。この機能を使用して、共通の時点で異なる PGD ノード上のデータにアクセスできます。たとえば、データ品質チェックのために複数のノードのデータを比較できます。

この機能は現在、書き込みトランザクションでは動作しません。

snapshot_timestamp パラメーターを使用して、タイムスタンプベースのスナップショットの使用を有効にします。このパラメーターは、タイムスタンプ値または現在のタイムスタンプ（現在）を表す特別な値'current' を受け入れます。 snapshot_timestamp が設定されている場合、クエリはそのタイムスタンプを使用して、通常の MVCC セマンティクスではなく行の可視性を判断します。

たとえば、次のクエリは 2018-12-08 02:28:30 GMT における customers テーブルの状態を返します。

```
SET snapshot_timestamp = 2018-12-08 02:28:30 GMT;
SELECT count(*) FROM customers;
```

PGD がなければ、このクエリは将来のタイムスタンプまたは特別な'current' 値でのみ機能するため、履歴クエリには使用できません。

PGD は、マルチノード環境でその機能と連携し、改善します。まず、PGD は、他のノードへのすべての接続が、指定されたタイムスタンプより前にデータベースに追加された未処理のデータを複製することを確認します。これにより、タイムスタンプベースのスナップショットがマルチマスターグループ全体で一貫

していることが保証されます。次に、PGD は bdr.timestamp_snapshot_keep というパラメーターを追加します。このパラメーターは、そのノードで最近の履歴に対してクエリを実行できる時間枠を指定します。

任意の間隔を指定できますが、VACUUM（自動バキュームを含む）は、指定された間隔の最大 2 倍より新しいデッド行をクリーニングしないことに注意してください。これは、トランザクション ID が同じ時間解放されないことも意味します。その結果、これを使用すると、ユーザーテーブルにより多くの膨張が残る可能性があります。最初は、一般的な設定として 10 秒をお勧めしますが、必要に応じて変更できます。

クエリの実行が受け入れられると、クエリは通常どおり、問題なく bdr.timestamp_snapshot_keep より長く実行される場合があります。

また、スナップショットがどの程度保持されているかに関する情報は、サーバーの再起動後も存続しません。タイムスタンプベースのスナップショットで使用可能な最も古いタイムスタンプは、PostgreSQL インスタンスの最後の再起動の時刻です。

bdr.timestamp_snapshot_keep の使用と postgres_fdw 拡張機能を組み合わせて、PGD グループ内の複数のノードで一貫した読み取りを取得できます。この組み合わせを使用して、外部テーブルで使用する場合、ノード間で並列クエリを実行できます。

この機能を使用する場合、マルチノードクエリのノード数に制限はありません。

タイムスタンプベースのスナップショットを使用しても、ノード間のトラフィックや帯域幅は増加しません。クエリデータに加えて、タイムスタンプ値のみが渡されます。

第9章　Reference

9.1 PGD Reference

リファレンスセクションは、 EDB Postgres Distributed で利用可能なすべての機能、ビュー、およびコマンドの決定的なリストです。

User visible catalogs and views

- *bdr.camo_decision_journal*
- *bdr.commit_scopes*
- *bdr.conflict_history*
- *bdr.conflict_history_summary*
- *bdr.consensus_kv_data*
- *bdr.crdt_handlers*
- *bdr.ddl_replication*
- *bdr.depend*
- *bdr.global_consensus_journal*
- *bdr.global_consensus_journal_details*
- *bdr.global_consensus_response_journal*
- *SELECT lock_type、relation、pid FROM bdr.global_locks;*
- *SELECT lock_type、relation、pid FROM bdr.global_locks;*
- *bdr.group_camo_details*
- *bdr.group_raft_details*
- *bdr.group_replslots_details*
- *bdr.group_subscription_summary*
- *bdr.group_versions_details*
- *bdr.local_consensus_snapshot*
- *bdr.local_consensus_state*
- *bdr.local_node*

- *bdr.local_node_summary*
- *bdr.local_sync_status*
- FROM bdr.node_summary;
- *bdr.node_catchup_info*
- *bdr.node_catchup_info_details*
- *bdr.node_conflict_resolvers*
- *bdr.node_group_config*
- *bdr.node_group_replication_sets*
- *bdr.node_local_info*
- *bdr.node_log_config*
- *bdr.node_peer_progress*
- SELECT * FROM bdr.node_replication_rates;
- FROM bdr.node_slots;
- FROM bdr.node_summary;
- *bdr.queue*
- *bdr.replication_set*
- *bdr.replication_set_table*
- *bdr.replication_set_ddl*
- *bdr.replication_sets*
- *bdr.schema_changes*
- *bdr.sequence_alloc*
- *bdr.* シーケンス
- *bdr.stat_activity*
- *bdr.stat_relation*
- *bdr.stat_subscription*
- *FROM bdr.subscription_summary;*
- *FROM bdr.subscription_summary;*
- *bdr.tables*
- *bdr.taskmgr_work_queue*
- *bdr.taskmgr_workitem_status*
- *bdr.taskmgr_local_work_queue*

- *bdr.taskmgr_local_workitem_status*
- *bdr.trigger*
- *bdr.triggers*
- *bdr.workers*
- *bdr.writers*
- *bdr.worker_tasks*

System functions

バージョン情報機能

- *bdr.bdr_version*
- *bdr.bdr_version_num*

システム情報機能

- *bdr.get_relation_stats*
- *bdr.get_subscription_stats*

システムと進捗情報パラメーター

- *bdr.local_node_id*
- *bdr.last_committed_lsn*
- *transaction_id*
- *bdr.is_node_connected*
- *bdr.is_node_ready*

コンセンサス機能

- *bdr.consensus_disable*
- *bdr.consensus_enable*
- *bdr.consensus_proto_version*
- *bdr.consensus_snapshot_export*
- *bdr.consensus_snapshot_import*
- *bdr.consensus_snapshot_verify*
- *bdr.get_consensus_status*
- *bdr.get_raft_status*

- *bdr.raft_leadership_transfer*

  ユーティリティ関数

- *bdr.wait_slot_confirm_lsn*
- *bdr.wait_for_apply_queue*
- *bdr.get_node_sub_receive_lsn*
- *bdr.get_node_sub_apply_lsn*
- *bdr.run_on_all_nodes*
- *bdr.run_on_nodes*
- *bdr.run_on_group*
- *bdr.global_lock_table*
- *bdr.wait_for_xid_progress*
- *bdr.local_group_slot_name*
- *bdr.node_group_type*
- *bdr.alter_node_kind*
- *bdr.alter_subscription_skip_changes_upto*

  グローバルアドバイザリーロック

- *bdr.global_advisory_lock*
- *bdr.global_advisory_unlock*

  監視機能

- *bdr.monitor_group_versions*
- *bdr.monitor_group_raft*
- *bdr.monitor_local_replslots*
- *bdr.wal_sender_stats*
- *bdr.get_decoding_worker_stat*
- *bdr.lag_control*

  PGD settings

  Exploring conflict handling with PGD

- *bdr.default_conflict_detection*

グローバルシーケンスパラメーター

- *bdr.default_sequence_kind*

DDL ハンドリング

- *bdr.default_replica_identity*
- *bdr.ddl_replication*
- *bdr.role_replication*
- *bdr.ddl_locking*
- *bdr.truncate_locking*

グローバルロック

- *bdr.global_lock_max_locks*
- *bdr.global_lock_timeout*
- *bdr.global_lock_statement_timeout*
- *bdr.global_lock_idle_timeout*
- *bdr.predictive_checks*

Node management

- *bdr.replay_progress_frequency*
- *bdr.standby_slot_names*

一般的なレプリケーション

- *bdr.writers_per_subscription*
- *bdr.max_writers_per_subscription*
- *bdr.xact_replication*
- *bdr.permit_unsafe_commands*
- *bdr.batch_inserts*
- *bdr.maximum_clock_skew*
- *bdr.maximum_clock_skew_action*
- *bdr.accept_connections*
- *bdr.standby_slot_names*
- *bdr.standby_slots_min_confirmed*

- *bdr.writer_input_queue_size*
- *bdr.writer_output_queue_size*
- *bdr.min_worker_backoff_delay*

状態ベースおよびオペレーションベースの *CRDT*

- *bdr.crdt_raw_value*

一般的な積極的な *Eager Replication* の効果

- *bdr.commit_scopes*

Commit At Most Once

- *bdr.camo_local_mode_delay*
- *bdr.camo_enable_client_warnings*

トランザクションストリーミング

- *bdr.default_streaming_mode*

Lag Control

- *bdr.lag_control_max_commit_delay*
- *bdr.lag_control_max_lag_size*
- *bdr.lag_control_max_lag_time*
- *bdr.lag_control_min_conforming_nodes*
- *bdr.lag_control_commit_delay_adjust*
- *bdr.lag_control_sample_interval*
- *bdr.lag_control_commit_delay_start*

タイムスタンプベースのスナップショット

- *bdr.timestamp_snapshot_keep*

モニタリングとロギング

- *bdr.debug_level*
- *bdr.trace_level*
- *bdr.track_subscription_apply*

- *bdr.track_relation_apply*
- *bdr.track_apply_lock_timing*

デコードワーカー

- *bdr.enable_wal_decoder*
- *bdr.receive_lcr*
- *bdr.lcr_cleanup_interval*

接続設定

- *bdr.global_connection_timeout*
- *bdr.global_keepalives*
- *bdr.global_keepalives_idle*
- *bdr.global_keepalives_interval*
- *bdr.global_keepalives_count*
- *bdr.global_tcp_user_timeout*

内部設定

- *bdr.raft_keep_min_entries*
- *bdr.raft_response_timeout*
- *bdr.raft_log_min_apply_duration*
- *bdr.raft_log_min_message_duration*
- *bdr.raft_group_max_connections*
- *bdr.backwards_compatibility*
- *bdr.track_replication_estimates*
- *bdr.lag_tracker_apply_rate_weight*

Node management

- ノード状態のリスト
- ノード管理コマンド
- *bdr_init_physical*

Node management interfaces

- *bdr.create_node*
- *bdr.drop_node*
- *bdr.create_node_group*
- *bdr.alter_node_group_config*
- *bdr.join_node_group*
- *bdr.switch_node_group*
- *bdr.promote_node*
- *bdr.wait_for_join_completion*
- *bdr.part_node*
- *bdr.alter_node_interface*
- *bdr.alter_subscription_enable*
- *bdr.alter_subscription_disable*
- *bdr.alter_node_group_option*
- *bdr.alter_node_option*

Routing functions

- *bdr.create_proxy*
- *bdr.alter_proxy_option*
- *bdr.drop_proxy*
- *bdr.routing_leadership_transfer*

Replication set management

- *bdr.create_replication_set*
- *bdr.alter_replication_set*
- *bdr.drop_replication_set*
- *bdr.alter_node_replication_sets*

Replication set membership

- *bdr.replication_set_add_table*
- *bdr.replication_set_remove_table*

DDL replication filtering

- *bdr.replication_set_add_ddl_filter*
- *bdr.replication_set_remove_ddl_filter*

Global sequence management interfaces

シーケンス関数

- *bdr.alter_sequence_set_kind*
- *bdr.extract_timestamp_from_snowflakeid*
- *bdr.extract_nodeid_from_snowflakeid*
- *bdr.extract_localseqid_from_snowflakeid*
- *bdr.timestamp_to_snowflakeid*
- *bdr.extract_timestamp_from_timeshard*
- *bdr.extract_nodeid_from_timeshard*
- *bdr.extract_localseqid_from_timeshard*
- *bdr.timestamp_to_timeshard*

KSUUID v2 **関数**

- *bdr.gen_ksuuid_v2*
- *bdr.ksuuid_v2_cmp*
- *bdr.extract_timestamp_from_ksuuid_v2*

KSUUID v1 **関数**

- *bdr.gen_ksuuid_v2*
- *bdr.uuid_v1_cmp*
- *bdr.extract_timestamp_from_ksuuid_v2*

Autopartition

- *bdr.autopartition*
- *bdr.drop_autopartition*
- *bdr.autopartition_wait_for_partitions*
- *bdr.autopartition_wait_for_partitions_on_all_nodes*
- *bdr.autopartition_find_partition*

- *bdr.autopartition_enable*
- *bdr.autopartition_disable*
- 内部関数
- *bdr.autopartition_create_partition*
- *bdr.autopartition_drop_partition*

9.1.1 ストリームトリガーリファレンス

Stream triggers manipulation interfaces

- *bdr.create_conflict_trigger*
- *bdr.create_transform_trigger*
- *bdr.drop_trigger*

Stream triggers row functions

- *bdr.trigger_get_row*
- *bdr.trigger_get_committs*
- *bdr.trigger_get_xid*
- *bdr.trigger_get_type*
- *bdr.trigger_get_conflict_type*
- *bdr.trigger_get_origin_node_id*
- *bdr.ri_fkey_on_del_trigger*

Stream triggers row variables

- *TG_NAME*
- *TG_WHEN*
- *TG_LEVEL*
- *TG_OP*
- *TG_RELID*
- *TG_TABLE_NAME*
- *TG_TABLE_SCHEMA*
- *TG_NARGS*
- *TG_ARGV[]*

Internal catalogs and views

- *bdr.ddl_epoch*
- *bdr.event_history*
- *bdr.event_summary*
- *bdr.node_config*
- *bdr.node_group_config*
- *bdr.node_group_routing_config_summary*
- *bdr.node_group_routing_info*
- *bdr.node_group_routing_summary*
- *bdr.node_routing_config_summary*
- *bdr.proxy_config*
- *bdr.proxy_config_summary*
- *bdr.sequence_kind*

Internal system functions

一般的な内部機能

- *bdr.decode_message_response_payload*
- *bdr.decode_message_payload*
- *bdr.get_global_locks*
- *bdr.get_slot_flush_timestamp*
- *bdr.internal_alter_sequence_set_kind*
- *bdr.internal_replication_set_add_table*
- *bdr.internal_replication_set_remove_table*
- *bdr.internal_submit_join_request*
- *bdr.isolation_test_session_is_blocked*
- *bdr.local_node_info*
- *bdr.msgb_connect*
- *bdr.msgb_deliver_message*
- *bdr.peer_state_name*
- *bdr.request_replay_progress_update*

- *bdr.seq_nextval*
- *bdr.show_subscription_status*
- *bdr.get_node_conflict_resolvers*
- *bdr.reset_subscription_stats*
- *bdr.reset_relation_stats*
- *bdr.pg_xact_origin*
- *bdr.difference_fix_origin_create*
- *bdr.difference_fix_session_setup*
- *bdr.difference_fix_session_reset*
- *bdr.difference_fix_xact_set_avoid_conflict*
- *bdr.resynchronous_table_from_node*
- *bdr.consensus_kv_store*
- *bdr.consensus_kv_fetch*

タスクマネージャー機能

- *bdr.taskmgr_set_leader*
- *bdr.taskmgr_get_last_completed_workitem*
- *bdr.taskmgr_work_queue_check_status*
- *bdr.node_catchup_state_name*
- *bdr.bdr_track_commit_decision*
- *bdr.bdr_get_commit_decisions*
- *bdr.show_workers*
- *bdr.show_writers*
- *bdr.node_kind_name*

User visible catalogs and views

ここでは、カタログとビューをアルファベット順に提示しています。

bdr.camo_decision_journal

bdr.logical_transaction_status が呼び出された場合、フェールオーバー後に CAMO パートナーノードによって解決された決定の永続的なジャーナル。bdr.node_pre_commit とは異なり、これは、通常の動作条件（つまり、CAMO ペアの両方のノードが実行および接続されている）で処理されるトランザクションをカバーしません。このジャーナルのエントリは、自動的にクリーンアップされることはありません。これは、システムが依存しない診断ツールです。

bdr.camo_decision_journal 列

Name	Type	Description
origin_node_id	oid	トランザクションが実行されたノードの OID
origin_xid	oid	リモートオリジンノードのトランザクション ID
decision	char	コミットの場合は「c」、中止の場合は「a」
decision_ts	timestamptz	デシジョンタイム

bdr.commit_scopes

Group Commit を有効にするために bdr.commit_scope に使用できるすべての可能なコミットスコープを保存するカタログ。

bdr.commit_scopes 列

Name	Type	Description
commit_scope_id	oid	参照するスコープの ID
commit_scope_name	name	参照するスコープの名前
commit_scope_origin_node_	oid	ID で参照される、ルールが適用されるノードグループ
sync_scope_rule	text	範囲の定義

bdr.conflict_history

このテーブルは、競合がログに記録されるデフォルトのテーブルです。テーブルは列 local_time で RANGE パーティション分割され、Autopartition によって管理されます。デフォルトのデータ保持期間は 30 日です。

このテーブルへのアクセスは、行レベルのセキュリティによって制限され、所有するテーブルのすべての競合を表示できるテーブルの所有者すべてが可能です。

詳細については、 競合をテーブルに記録する を参照してください。

bdr.conflict_history 列

Name	Type	Description
sub_id	oid	この競合を引き起こしたサブスクリプション。`bdr.subscription`テーブルに結合できます
local_xid	xid	競合時のレプリケーションプロセスのローカルトランザクション
local_lsn	pg_lsn	競合時のローカル LSN
local_time	timestamp with time zone	紛争の現地時間
remote_xid	xid	リモートノード（オリジン）で競合する変更を生成したトランザクション
remote_commit_lsn	pg_lsn	リモートノード（オリジン）で競合する変更を生成したトランザクションの LSN をコミットします
remote_commit_time	timestamp with time zone	リモートノード（オリジン）で競合する変更を生成したトランザクションのコミットタイムスタンプ
conflict_type	text	検出された競合のタイプ
conflict_resolution	text	選択された競合の解決
conflict_index	regclass	競合するインデックス（インデックスが削除されていない場合にのみ有効）
reloid	oid	競合する関係（インデックスが削除されていない場合にのみ有効）
nspname	text	競合時に競合が発生したリレーションのスキーマの名前（名前変更に従いません）
relname	text	競合時に競合が発生したリレーションの名前（名前変更には従いません）
key_tuple	json	行のマッチングに使用されるキーの Json 表現
remote_tuple	json	着信する競合する行の Json 表現
local_tuple	json	ローカルの競合する行の Json 表現
apply_tuple	json	結果の（適用された）行の Json 表現
local_tuple_xmin	xid	ローカルの競合する行を生成したトランザクション（ *local_tuple* が設定され、行が凍結されていない場合）
local_tuple_node_id	oid	ローカルの競合する行を生成したノード（ *local_tuple* が設定され、行がフリーズされていない場合）
local_tuple_commit_time	timestamp with time zone	ローカルの競合する行の最終既知の変更タイムスタンプ（ *local_tuple* が設定され、行がフリーズされていない場合）

bdr.conflict_history_summary

行の競合に関するユーザーが判読可能な詳細を含むビュー。

bdr.conflict_history_summary 列

Name	Type	Description
nspname	text	スキーマの名前
relname	text	テーブルの名前
local_time	timestamp with time zone	紛争の現地時間
local_tuple_commit_time	timestamp with time zone	ローカルコミットの時間
remote_commit_time	timestamp with time zone	リモートコミットの時間
conflict_type	text	紛争の種類
conflict_resolution	text	採択された決議

bdr.consensus_kv_data

`bdr.consensus_kv_store()` および `bdr.consensus_kv_fetch()` インターフェースによって使用される Raft ベースの内部 KV ストア用の永続ストレージ。

bdr.consensus_kv_data 列

Name	Type	Description
kv_key	text	ユニークキー
kv_val	json	json 形式の任意値
kv_create_ts	timestamptz	最終書き込みタイムスタンプ
kv_ttl	int	ミリ秒単位の値の存続時間
kv_expire_ts	timestamptz	有効期限タイムスタンプ（*kv_create_ts* + *kv_ttl*）

bdr.crdt_handlers

この表は、すべての CRDT データ型のマージ (「ハンドラー」) 関数をリストしています。

bdr.crdt_handlers 列

Name	Type	Description
crdt_type_id	regtype	CRDT データ型 ID
crdt_merge_id	regproc	このデータ型のマージ機能

bdr.ddl_replication

このビューには、現在の *DDL filters* によって設定された DDL レプリケーション構成がリストされます。

bdr.ddl_replication 列

Name	Type	Description
set_ddl_name	name	DDL フィルターの名前
set_ddl_tag	text	適用されるコマンドタグ（正規表現）
set_ddl_role	text	適用されるロール （正規表現）
set_name	name	このフィルターが定義されているレプリケーションセットの名前

bdr.depend

このテーブルは、PGD カタログ内の内部オブジェクト依存関係を追跡します。

bdr.global_consensus_journal

このカタログテーブルは、グローバルコンセンサスの管理中に送信されたすべての Raft メッセージをログに記録します。

`bdr.global_consensus_response_journal` カタログに関しては、ペイロードはバイナリエンコードされた形式で保存され、`bdr.decode_message_payload()` ファンクションでデコードできます。詳細については、

bdr.global_consensus_journal_details ビューを参照してください。

bdr.global_consensus_journal 列

Name	Type	Description
log_index	int8	仕訳の ID
term	int8	ラフトターム
origin	oid	リクエストが発生したノードの ID
req_id	int8	リクエストの ID
req_payload	bytea	リクエストのペイロード
trace_context	bytea	リクエストのトレースコンテキスト

bdr.global_consensus_journal_details

このビューは、 `bdr.decode_message_payload()` 関数を使用してペイロードをデコードし、送信された Raft メッセージと対応する応答を表示します。

bdr.global_consensus_journal_details 列

Name	Type	Description
node_group_name	name	ノードグループの名前
log_index	int8	仕訳の ID
term	int8	ラフトターム
request_id	int8	リクエストの ID
origin_id	oid	リクエストが発生したノードの ID
req_payload	bytea	リクエストのペイロード
origin_node_name	name	要求が発生したノードの名前
message_type_no	oid	リクエストの PGD メッセージタイプの ID
message_type	text	要求の PGD メッセージタイプの名前
message_payload	text	リクエストの PGD メッセージペイロード
response_message_type_n	oid	応答の PGD メッセージタイプの ID
response_message_type	text	応答の PGD メッセージタイプの名前
response_payload	text	応答の PGD メッセージペイロード
response_errcode_no	text	応答の SQLSTATE
response_errcode	text	レスポンスのエラーコード
response_message	text	レスポンスのエラーメッセージ

bdr.global_consensus_response_journal

このカタログテーブルは、グローバルコンセンサスの管理中に受信した Raft メッセージへのすべての応答を収集します。

bdr.global_consensus_journal カタログに関しては、ペイロードはバイナリエンコード形式で格納され、bdr.decode_message_payload() ファンクションでデコードできます。詳細については、

bdr.global_consensus_journal_details ビューを参照してください。

bdr.global_consensus_response_journal 列

Name	Type	Description
log_index	int8	仕訳の ID
res_status	oid	レスポンスのステータスコード
res_payload	bytea	レスポンスのペイロード
trace_context	bytea	応答のトレースコンテキスト

bdr.global_lock

このカタログテーブルは、サーバーの再起動時にグローバルロック状態を回復するために必要な情報を格納します。

bdr.global_lock の可視行は必ずしもすべてのグローバルロックアクティビティを反映するとは限らないため、使用状況の監視には、

SELECT lock_type、relation、pid FROM bdr.global_locks; ビューが推奨されます。

このテーブルの内容を変更しないでください。重要な PGD カタログです。

bdr.global_lock 列

Name	Type	Description
ddl_epoch	int8	ロックの DDL エポック
origin_node_id	oid	グローバルロックが発生したノードの OID
lock_type	oid	ロックのタイプ（DDL または DML）
nspname	name	ロックされたリレーションのスキーマ名
relname	name	ロックされたリレーションのリレーション名
groupid	oid	トップレベルグループの OID（アドバイザリーロックの場合）

次のページに続く

表 11 – 前のページからの続き

Name	Type	Description
key1	integer	最初の 32 ビットキーまたは 64 ビットキーの下位 32 ビット（勧告ロック用）
key2	integer	2 番目の 32 ビットキーまたは上位 32 ビットの 64 ビットキー（勧告ロック用）
key_is_bigint	boolean	64 ビット整数キーが使用されている場合は true（勧告ロック用）

bdr.global_locks

このノード上のアクティブなグローバルロックを含むビュー。

SELECT lock_type、relation、pid FROM bdr.global_locks; ビューは、PGD の共有メモリロック状態の追跡を公開し、管理者が PGD のグローバルロックアクティビティと進行状況をより深く理解できるようにします。

グローバルロックの監視 を参照

グローバルロックの詳細については。

bdr.global_locks 列

Name	Type	Description
origin_node_id	oid	グローバルロックが発生したノードの OID
origin_node_name	name	グローバルロックが発生したノードの名前
lock_type	text	ロックのタイプ（DDL または DML）
relation	text	ロックされたリレーション名（DML ロックの場合）またはキー（勧告ロックの場合）
pid	int4	ロックを保持しているプロセスの PID
acquire_stage	text	ロック取得プロセスの内部状態
waiters	int4	同じグローバルロックを待っているバックエンドのリスト
global_lock_request_tim	timestamptz	このグローバルロックの取得が起点ノードによって開始された時刻
local_lock_request_time	timestamptz	ローカルノードがローカルロックの取得を開始した時刻
last_state_change_time	timestamptz	`acquire_stage`の最終変更時刻

列の詳細：

・ `relation` ： DML ロックの場合、 `relation` は、DML ロックを取得する関係を示します。グローバル勧告ロックの場合、`relation` 列には、実際にロックが取得された 2 つの 32 ビット整数または 1 つの 64 ビット整数が表示されます。

- origin_node_id および origin_node_name ：これらがローカルノードの ID および名前と同じである場合、ローカルノードはグローバル DDL ロックのイニシエータ、つまり、取得トランザクションを実行しているノードです。これらのフィールドが別のノードを指定している場合、ローカルノードは代わりにローカル DDL ロックを取得して、リモートノードからのグローバル DDL ロック要求を満たすことを試みています。
- pid ：ローカルノードが要求元ノードである場合、グローバル DDL ロックを要求したプロセスのプロセス ID。他のノードでは null。オリジンノードを照会して、ロッカー pid を特定します。
- global_lock_request_time ：グローバルロック要求イニシエータがグローバルロックの取得プロセスを開始したタイムスタンプ。現在のノードで不明な場合、null になる可能性があります。この時間は DDL ロック要求の先頭にスタンプされ、DDL エポック管理と保留中のレプリケーションキューの必要なフラッシュにかかった時間が含まれます。現在、オリジンノードでのみ知られています。
- local_lock_request_time ：ローカルノードがこのグローバルロックのローカルロックの取得を試行し始めたタイムスタンプ。これには、重量セッションロックの取得にかかった時間が含まれますが、DDL エポックまたはキューのフラッシュにかかった時間は含まれません。自ノード再起動後に再度ロックを取得すると、ノード再起動時間となります。
- last_state_change_time ：このグローバルロックエントリの bdr.global_locks.acquire_stage フィールドが最後に変更されたタイムスタンプ。

bdr.group_camo_details

bdr.run_on_all_nodes を使用して、すべてのノードから CAMO 関連の情報を収集します。

bdr.group_camo_details 列

Name	Type	Description
node_id	text	内部ノード ID
node_name	text	ノードの名前
camo_partner	text	カモパートナーのノード名
is_camo_partner_connec	text	接続状態
is_camo_partner_ready	text	準備状況
camo_transactions_resol	text	保留中および未解決の CAMO トランザクションはありますか
apply_lsn	text	リプレイとして報告された最新の位置（表示）
receive_lsn	text	受信した変更またはメッセージの最新の LSN（リスタートの場合は前に戻ることができます）
apply_queue_size	text	apply_lsn と receive_lsn のバイト差

bdr.group_raft_details

bdr.run_on_all_nodes を使用して、すべてのノードから Raft Consensus ステータスを収集します。

bdr.group_raft_details 列

Name	Type	Description
node_id	oid	内部ノード ID
node_name	name	ノードの名前
node_group_name	name	所属しているグループの名前
state	text	ノード上のラフトワーカーの状態
leader_id	oid	RAFT_LEADER のノード ID
current_term	int	ラフト選挙内部 ID
commit_index	int	Raft スナップショットの内部 ID
nodes	int	アクセス可能なノード数
voting_nodes	int	投票するノード数
protocol_version	int	このノードのプロトコルバージョン

bdr.group_replslots_details

bdr.run_on_all_nodes を使用して、すべてのノードから PGD スロット情報を収集します。

bdr.group_replslots_details 列

Name	Type	Description
node_group_name	text	PGD グループの名前
origin_name	text	起点ノードの名前
target_name	text	対象ノードの名前
slot_name	text	このサブスクリプションで使用されるオリジンノードのスロット名
active	text	スロットはアクティブですか（接続が接続されていますか）
state	text	レプリケーションの状態（キャッチアップ、ストリーミングなど）またはオフラインの場合は「切断」
write_lag	interval	報告された書き込みのおおよそのラグタイム
flush_lag	interval	報告されたフラッシュのおおよその遅延時間
replay_lag	interval	報告されたリプレイのおおよそのラグタイム

次のページに続く

表 15 – 前のページからの続き

Name	Type	Description
sent_lag_bytes	int8	sent_lsn と現在の WAL 書き込み位置のバイト差
write_lag_bytes	int8	write_lsn と現在の WAL 書き込み位置のバイト差
flush_lag_bytes	int8	flush_lsn と現在の WAL 書き込み位置のバイト差
replay_lag_byte	int8	replay_lsn と現在の WAL 書き込み位置のバイト差

bdr.group_subscription_summary

bdr.run_on_all_nodes を使用して、すべてのノードからサブスクリプションステータスを収集します。

bdr.group_subscription_summary 列

Name	Type	Description
origin_node_name	text	購読元の名前
target_node_name	text	サブスクリプションのターゲットの名前
last_xact_replay_timesta	text	最後に再生されたトランザクションのタイムスタンプ
sub_lag_seconds	text	now と last_xact_replay_timestamp の間のラグ

bdr.group_versions_details

bdr.run_on_all_nodes を使用して、すべてのノードから PGD 情報を収集します。

bdr.group_versions_details 列

Name	Type	Description
node_id	oid	内部ノード ID
node_name	name	ノードの名前
postgres_version	text	ノード上の PostgreSQL バージョン
bdr_version	text	ノード上の PGD バージョン

bdr.local_consensus_snapshot

このカタログテーブルには、ローカルノードによって作成または受信されたコンセンサススナップショットが含まれています。

bdr.local_consensus_snapshot 列

Name	Type	Description
log_index	int8	仕訳の ID
log_term	int8	ラフトターム
snapshot	bytea	いかだスナップショットデータ

bdr.local_consensus_state

このカタログテーブルは、ローカルノード上の Raft の現在の状態を保存します。

bdr.local_consensus_state 列

Name	Type	Description
node_id	oid	ノードの ID
current_term	int8	ラフトターム
apply_index	int8	ラフトアプライインデックス
voted_for	oid	今期このノードからの投票
last_known_leader	oid	最後にわかっているラフトリーダーの node_id

bdr.local_node

このテーブルは、現在の Postgres インスタンスの現在のデータベース内のローカルノードを識別します。

bdr.local_node 列

Name	Type	Description
node_id	oid	ノードの ID
pub_repsets	text[]	パブリッシュされた複製セット
sub_repsets	text[]	サブスクライブされた複製セット

次のページに続く

表 20 – 前のページからの続き

Name	Type	Description

bdr.local_node_summary

FROM bdr.node_summary; （および pub_repsets および sub_repsets ）と同じ情報を含むが、ローカルノードのみを対象とするビュー。

bdr.local_sync_status

サブスクリプションまたはテーブル同期プロセスのステータスに関する情報。

bdr.local_sync_status 列

Name	Type	Description
sync_kind	char	行われた同期の種類
sync_subid	oid	同期を行うサブスクリプションの ID
sync_nspname	name	同期テーブルのスキーマ名（ある場合）
sync_relname	name	同期されたテーブルの名前（ある場合）
sync_status	char	同期の現在の状態
sync_remote_relid	oid	アップストリームの同期テーブル（ある場合）の ID
sync_end_lsn	pg_lsn	同期状態が最後に変化した位置

bdr.node

この表には、クラスター内のすべての PGD ノードがリストされています。

bdr.node 列

Name	Type	Description
node_id	oid	ノードの ID
node_name	name	ノードの名前
node_group_id	oid	ノードグループの ID
source_node_id	oid	ソースノードの ID
synchronize_structure	"char"	参加中に行われるスキーマの同期
node_state	oid	ノードの一貫した状態

次のページに続く

表 22 – 前のページからの続き

Name	Type	Description
target_state	oid	ノードが到達しようとしている状態（参加中または昇格中）
seq_id	int4	一意のシーケンス番号を生成するために使用されるノードのシーケンス識別子
dbname	name	ノードのデータベース名
node_dsn	char	ノードの接続文字列
proto_version_ranges	int[]	ノードがサポートするプロトコルバージョン範囲
node_join_finished	boolean	結合が終了したかどうかを確認する

bdr.node_catchup_info

このカタログテーブルは、結合または部分的なプロシージャに関連している場合、各ノードに関連するキャッチアップ情報を記録します。

bdr.node_catchup_info 列

Name	Type	Description
node_id	oid	ノードの ID
node_source_id	oid	データのソースとして使用されるノードの ID
slot_name	name	このソースに使用されるスロット
min_node_lsn	pg_lsn	ノードがピアノードからの直接再生に切り替えることができる最小 LSN
catchup_state	oid	キャッチアップ状態のステータスコード
origin_node_id	oid	トランザクションが必要なノードの ID

ノード（node_id）が別れノード（origin_node_id）から欠損データを必要とする場合、転送により、すでにそれを持っているノード（node_source_id）から取得できます。このテーブルのレコードは、ノード（node_id）が EDB Postgres 分散クラスターのメンバーになるまで持続します。

bdr.node_catchup_info_details

よりフレンドリーな方法で情報を表示する `bdr.node_catchup_info` カタログのビュー

bdr.node_conflict_resolvers

既知のすべての競合タイプに対して現在構成されている競合解決。

bdr.node_conflict_resolvers 列

Name	Type	Description
conflict_type	text	紛争の種類
conflict_resolver	text	この競合タイプに使用されるリゾルバー

bdr.node_group

このカタログテーブルには、すべての PGD ノードグループがリストされています。

bdr.node_group 列

Name	Type	Description
node_group_id	oid	ノードグループの ID
node_group_name	name	ノードグループの名前
node_group_default_rep	oid	このノードグループのデフォルトのレプリケーションセット
node_group_default_rep	oid	このノードグループのデフォルトのレプリケーションセット
node_group_parent_id	oid	親グループの ID（これがルートグループの場合は 0）
node_group_flags	int	グループフラグ
node_group_uuid	uuid	グループの uuid
node_group_apply_delay	interval	プロバイダーからの変更を適用するまでにサブスクライバーが待機する時間
node_group_check_cons	bool	データの適用時に適用プロセスが制約をチェックするかどうか
node_group_num_writer	int	このノードグループをバッキングするサブスクリプションに使用するライターの数
node_group_enable_wal	bool	グループに enable_wal_decoder が設定されているかどうか
node_group_streaming_	char	トランザクションストリーミング設定：「O」-オフ、「F」-ファイル、「W」-ライタ、「A」-自動、「D」-デフォルト
node_group_location	char	ノードグループに関連付けられた場所の名前

次のページに続く

表 25 – 前のページからの続き

Name	Type	Description
node_group_enable_pro:	char	ノードグループが `pgd-proxy`からのルーティングを許可するかどうか
node_group_enable_raft	bool	ノードグループが Raft Consensus を許可するかどうか

bdr.node_group_replication_sets

PGD グループ用に作成されるデフォルトの複製セットを示すビュー。bdr.replication_sets も参照してください。

bdr.node_group_replication_sets 列

Name	Type	Description
node_group_name	name	PGD グループの名前
def_repset	name	デフォルトの repset の名前
def_repset_ops	text[]	デフォルトの repset によって複製されるアクション
def_repset_ext	name	デフォルトの「外部」repset の名前（通常は def_repset と同じ）
def_repset_ext_ops	text[]	デフォルトの「外部」repset によって複製されるアクション（通常は def_repset_ops と同じ）

bdr.node_local_info

ローカルノードに固有のノードごとの構成を格納するために使用されるカタログテーブル（ノードごとの構成のグローバルビューとは対照的に）。

bdr.node_local_info 列

Name	Type	Description
node_id	oid	ノードの OID（ローカルノードを含む）
applied_state	oid	ノード状態の内部 ID
ddl_epoch	int8	ノードによって処理された最後のエポック番号
slot_name	name	そのノードへの接続に使用されるスロットの名前（ローカルノードの場合は NULL）

bdr.node_log_config

競合ログ構成に関する情報を格納するカタログビュー。

bdr.node_log_config 列

Name	Description
log_name	ロギング構成の名前
log_to_file	サーバーログファイルに記録するかどうか
log_to_table	テーブルにログを記録するかどうか、およびどのテーブルがターゲットか
log_conflict_type	ログに記録する競合タイプ（NULL がすべてを意味する場合）
log_conflict_res	NULL がすべてを意味する場合、ログに記録する競合解決

bdr.node_peer_progress

レプリケーションストリーム内のすべてのノードの進行状況を追跡するために使用されるカタログ。クラスター内のすべてのノードは、`bdr.replay_progress_frequency` ミリ秒ごとに他のすべてのノードに進行状況を定期的にブロードキャストします（デフォルトは 60000 ミリ秒、つまり 1 分です）。このリレーションには N * (N-1) 行が必要です。

モニタリング目的で `bdr.node_slots` ビューにもっと興味があるかもしれません。

> *Monitoring* も参照してください。

bdr.node_peer_progress 列

Name	Type	Description
node_id	oid	この位置情報を報告した発信ノードの OID
peer_node_id	oid	この位置情報が報告されたノードのピア（リモートノード）の OID
last_update_sent_time	timestamptz	発信元ノードによってレポートが送信された時間
last_update_recv_time	timestamptz	ローカルサーバーがレポートを受信した時刻
last_update_node_lsn	pg_lsn	レポート時の発信元ノードの LSN
peer_position	pg_lsn	発信元ノードから見たノードのピアの最新の LSN
peer_replay_time	timestamptz	レポートノードによって見られるピアの最新のリプレイ時間
last_update_horizon_xid	oid	内部解像度ホライズン：より低いすべての xid は、レポートノードで解決されていることがわかっています
last_update_horizon_lsn	pg_lsn	内部解決期間：報告ノードの LSN に関して同じ

bdr.node_replication_rates

このビューには、特定のノードからの発信レプリケーションアクティビティに関する情報が含まれています。

bdr.node_replication_rates 列

Column	Type	Description
peer_node_id	oid	この情報が報告されたノードのピア（リモートノード）の OID
target_name	name	ターゲットピアノードの名前
sent_lsn	pg_lsn	最新の送信位置
replay_lsn	pg_lsn	リプレイとして報告された最新の位置（表示）
replay_lag	interval	報告されたリプレイのおおよそのラグタイム
replay_lag_bytes	int8	replay_lsn と原点上の現在の WAL 書き込み位置のバイト差
replay_lag_size	text	replay_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
apply_rate	bigint	ピアノードで 1 秒あたりに適用されている LSN
catchup_interval	interval	ピアノードがまだ適用されていないすべての変更に追いつくために必要なおおよその時間

注釈: *replay_lag* は、再接続後すぐにゼロに設定されます。回避策として、*replay_lag_bytes* 、*replay_lag_size* 、または `catchup_interval` を使用します。

bdr.node_slots

このビューには、PGD によって現在のデータベースで使用されるレプリケーションスロットに関する情報が含まれています。

発信レプリケーションのモニタリング を参照

このビューのフィールドの使用と解釈に関するガイダンス。

bdr.node_slots 列

Name	Type	Description
target_dbname	name	ターゲットノードのデータベース名
node_group_name	name	PGD グループの名前
node_group_id	oid	PGD グループの OID

次のページに続く

表 31 – 前のページからの続き

Name	Type	Description
origin_name	name	起点ノードの名前
target_name	name	対象ノードの名前
origin_id	oid	オリジンノードの OID
target_id	oid	ターゲットノードの OID
local_slot_name	name	PGD によるレプリケーションスロットの名前
slot_name	name	Postgres によるスロットの名前（上記と同じ）
is_group_slot	boolean	スロットがこのノードのノードグループクラッシュリカバリスロットである場合は true ([「Group Replication Slot」](nodes#Group Replication Slot) を参照）
is_decoder_slot	boolean	このスロットはデコードワーカー機能によって使用されますか
plugin	name	このスロットを使用した論理デコードプラグイン（ pglogical_output または bdr であるべきです）
slot_type	text	スロットのタイプ（論理的である必要があります）
datoid	oid	現在のデータベースの OID
database	name	現在のデータベースの名前
temporary	bool	スロットは一時的ですか
active	bool	スロットはアクティブですか（接続が接続されていますか）
active_pid	int4	スロットに接続されているプロセスの PID
xmin	xid	スロットに必要な XID
catalog_xmin	xid	スロットに必要なカタログ XID
restart_lsn	pg_lsn	スロットがデコードを再開できる LSN
confirmed_flush_lsn	pg_lsn	最新に確認された複製位置
usesysid	oid	レプリケーションセッションが実行されているユーザーの sysid
usename	name	レプリケーションセッションを実行しているユーザーのユーザー名
application_name	text	クライアント接続のアプリケーション名（`synchronous_standby_names`が使用）
client_addr	inet	クライアント接続の IP アドレス
client_hostname	text	クライアント接続のホスト名
client_port	int4	クライアント接続のポート
backend_start	timestamptz	接続が開始されたとき
state	text	レプリケーションの状態（キャッチアップ、ストリーミングなど）またはオフラインの場合は「切断」
sent_lsn	pg_lsn	最新の送信位置
write_lsn	pg_lsn	書かれたとおりに報告された最新の位置
flush_lsn	pg_lsn	ディスクにフラッシュされたと報告された最新の位置

次のページに続く

表 31 – 前のページからの続き

Name	Type	Description
replay_lsn	pg_lsn	リプレイとして報告された最新の位置（表示）
write_lag	interval	報告された書き込みのおおよそのラグタイム
flush_lag	interval	報告されたフラッシュのおおよその遅延時間
replay_lag	interval	報告されたリプレイのおおよそのラグタイム
sent_lag_bytes	int8	sent_lsn と現在の WAL 書き込み位置のバイト差
write_lag_bytes	int8	write_lsn と現在の WAL 書き込み位置のバイト差
flush_lag_bytes	int8	flush_lsn と現在の WAL 書き込み位置のバイト差
replay_lag_bytes	int8	replay_lsn と現在の WAL 書き込み位置のバイト差
sent_lag_size	text	sent_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
write_lag_size	text	write_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
flush_lag_size	text	flush_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い
replay_lag_size	text	replay_lsn と現在の WAL 書き込み位置の人間が読めるバイトの違い

注釈: *replay_lag* は、再接続後すぐにゼロに設定されます。回避策として、 *replay_lag_bytes* または `replay_lag_size` を使用します。

bdr.node_summary

このビューには、ローカルノードが認識しているすべての PGD ノードに関する概要情報が含まれています。

bdr.node_summary 列

Name	Type	Description
node_name	name	ノードの名前
node_group_name	name	ノードが属する PGD グループの名前
interface_connstr	text	ノードへの接続文字列
peer_state_name	text	人間が読める形式でのノードの一貫した状態
peer_target_state_name	text	ノードが到達しようとしている状態（参加中または昇格中）
node_seq_id	int4	一意のシーケンス番号を生成するために使用されるノードのシーケンス識別子
node_local_dbname	name	ノードのデータベース名
node_id	oid	ノードの OID

次のページに続く

表 32 – 前のページからの続き

Name	Type	Description
node_group_id	oid	PGD ノードグループの OID
node_kind_name	oid	ノード種類名

bdr.queue

このテーブルには、レプリケートされた DDL ステートメントの履歴レコードが保存されます。

bdr.queue 列

Name	Type	Description
queued_at	timestamptz	ステートメントがキューに入れられたのはいつですか
role	name	ステートメントを実行したロール
replication_sets	text[]	ステートメントはどのレプリケーションセットに発行されましたか
message_type	char	メッセージの種類。可能な値： A - Table sync D - DDL S - Sequence T - Truncate Q - SQL ステートメント
message	json	ステートメントのレプリケーションに必要なメッセージのペイロード

bdr.replication_set

レプリケーションセットの構成を格納するテーブル。ユーザークエリの場合、代わりに `bdr.replication_sets` ビューを確認することをお勧めします。

bdr.replication_set 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_nodeid	oid	ノードの OID（現在は常にローカルノード oid）
set_name	name	複製セットの名前
replicate_insert	boolean	レプリケーションセットが INSERT をレプリケートするかどうかを示します
replicate_update	boolean	レプリケーションセットが UPDATE をレプリケートするかどうかを示します

次のページに続く

表 34 – 前のページからの続き

Name	Type	Description
replicate_delete	boolean	レプリケーションセットが DELETE をレプリケートするかどうかを示します
replicate_truncate	boolean	レプリケーションセットが TRUNCATE をレプリケートするかどうかを示します
set_isinternal	boolean	予約済み
set_autoadd_tables	boolean	このレプリケーションセットに新しいテーブルが自動的に追加されるかどうかを示します
set_autoadd_seqs	boolean	この複製セットに新しいシーケンスが自動的に追加されるかどうかを示します

bdr.replication_set_table

レプリケーションセットテーブルメンバーシップを格納するテーブル。ユーザークエリの場合、代わりに `bdr.tables` ビューを確認することをお勧めします。

bdr.replication_set_table 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_reloid	regclass	テーブルのローカル ID
set_att_list	text[]	予約済み
set_row_filter	pg_node_tree	コンパイルされた行フィルタリング式

bdr.replication_set_ddl

レプリケーションセット ddl レプリケーションフィルターを格納するテーブル。ユーザークエリの場合、代わりに `bdr.ddl_replication` ビューを確認することをお勧めします。

bdr.replication_set_ddl 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_ddl_name	name	DDL フィルターの名前
set_ddl_tag	text	DDL フィルターのコマンドタグ
set_ddl_role	text	DDL を実行するロール

bdr.replication_sets

PGD グループで定義されたレプリケーションセットを示すビュー（現在どのノードでも使用されていない場合でも）。

bdr.replication_sets 列

Name	Type	Description
set_id	oid	レプリケーションセットの OID
set_name	name	複製セットの名前
replicate_insert	boolean	レプリケーションセットが INSERT をレプリケートするかどうかを示します
replicate_update	boolean	レプリケーションセットが UPDATE をレプリケートするかどうかを示します
replicate_delete	boolean	レプリケーションセットが DELETE をレプリケートするかどうかを示します
replicate_truncate	boolean	レプリケーションセットが TRUNCATE をレプリケートするかどうかを示します
set_autoadd_tables	boolean	このレプリケーションセットに新しいテーブルが自動的に追加されるかどうかを示します
set_autoadd_seqs	boolean	この複製セットに新しいシーケンスが自動的に追加されるかどうかを示します

bdr.schema_changes

スキーマに対するすべての変更を表示する単純なビューが PGD に勝ちます。

bdr.schema_changes 列

Name	Type	Description
schema_changes_ts	timestampstz	トリガーの ID
schema_changes_change	char	変更タイプのフラグ
schema_changes_classid	oid	クラス ID
schema_changes_objecti	oid	オブジェクト ID
schema_changes_subid	smallint	サブスクリプション
schema_changes_descr	text	オブジェクトが変更されました
schema_changes_addrna	text[]	スキーマ変更の場所

bdr.sequence_alloc

galloc シーケンスの割り当て詳細を表示するビュー。

bdr.sequence_alloc 列

Name	Type	Description
seqid	regclass	シーケンスの ID
seq_chunk_size	bigint	その値内のチャンクのシーケンス番号
seq_allocated_up_to	bigint	該当なし
seq_nallocs	bigint	該当なし
seq_last_alloc	timestamptz	最後に割り当てられたシーケンス

bdr. シーケンス

このビューには、内部 PGD ブックキーピングのシーケンスを除く、すべてのシーケンスがその種類とともにリストされます。

bdr.sequences 列

Name	Type	Description
nspname	name	シーケンスを含む名前空間
relname	name	シーケンスの名前
seqkind	text	シーケンスのタイプ（「local」、「timeshard」、「galloc」）

bdr.stat_activity

各バックエンドまたはワーカープロセスの動的アクティビティ。

これには、`pg_stat_activity` と同じ情報が含まれますが、待機が PGD に関連する場合に `wait_event` が正しく設定されます。

bdr.stat_relation

各関係に統計を適用します。追跡が有効になっていて、特定のリレーションに対して何かがレプリケートされた場合にのみデータが含まれます。

bdr.stat_relation 列

Column	Type	Description
nspname	name	リレーションのスキーマの名前
relname	name	関係の名前
relid	oid	リレーションの OID
total_time	double precision	リレーションのレプリケーションの処理にかかった合計時間
ninsert	bigint	リレーションに対して複製された挿入の数
nupdate	bigint	リレーションのために複製された更新の数
ndelete	bigint	リレーションのために複製された削除の数
ntruncate	bigint	リレーションのために複製された切り捨ての数
shared_blks_hit	bigint	リレーションの共有ブロックキャッシュヒットの合計数
shared_blks_read	bigint	リレーションのために読み取られた共有ブロックの合計数
shared_blks_dirtied	bigint	リレーションのためにダーティされた共有ブロックの総数
shared_blks_written	bigint	リレーションに書き込まれた共有ブロックの総数
blk_read_time	double precision	ミリ秒単位のリレーションのブロックの読み取りに費やされた合計時間（`track_io_timing`が有効になっている場合、それ以外の場合はゼロ）
blk_write_time	double precision	リレーションのブロックの書き込みに費やされた合計時間、ミリ秒単位（`track_io_timing`が有効になっている場合、そうでない場合はゼロ）
lock_acquire_time	double precision	リレーションのロックの取得に費やされた合計時間、ミリ秒単位（`bdr.track_apply_lock_timing`が有効になっている場合、それ以外の場合はゼロ）

bdr.stat_subscription

サブスクリプションごとに統計を適用します。トラッキングが有効になっている場合にのみデータが含まれます。

bdr.stat_subscription 列

Column	Type	Description
sub_name	name	サブスクリプションの名前
subid	oid	サブスクリプションの OID
mean_apply_time	double precision	ミリ秒単位の適用トランザクションあたりの平均時間
nconnect	bigint	このサブスクリプションがアップストリームに接続した回数
ncommit	bigint	このサブスクリプションが行ったコミットの数
nabort	bigint	このサブスクリプションに対してライタが行ったアボートの数
nerror	bigint	このサブスクリプションでライタがヒットしたエラーの数
nskippedtx	bigint	このサブスクリプションのライタによってスキップされたトランザクションの数（`skip_transaction`競合リゾルバーによる）
ninsert	bigint	このサブスクリプションが行った挿入の数
nupdate	bigint	このサブスクリプションが行った更新の数
ndelete	bigint	このサブスクリプションが行った削除の数
ntruncate	bigint	このサブスクリプションが行った切り捨ての数
nddl	bigint	このサブスクリプションが実行した DDL 操作の数
ndeadlocks	bigint	デッドロックが原因のエラーの数
nretries	bigint	ライタが行った再試行の数（完全な再起動/再接続を行わずに）
nstream_writer	bigint	ライタにストリーミングされたトランザクションの数
nstream_file	bigint	ファイルにストリーミングされたトランザクションの数
nstream_commit	bigint	コミットされたストリーミングトランザクションの数
nstream_abort	bigint	中止されたストリーミングトランザクションの数
nstream_start	bigint	処理された STREAM START メッセージの数
nstream_stop	bigint	処理された STREAM STOP メッセージの数
nstream_commit	bigint	コミットされたストリーミングトランザクションの数
nstream_abort	bigint	中止されたストリーミングトランザクションの数
nstream_prepare	bigint	準備されたストリーミングトランザクション数
nstream_insert	bigint	処理されたストリーミング挿入の数
nstream_update	bigint	処理されたストリーミング更新の数
nstream_delete	bigint	処理されたストリーミング削除の数

次のページに続く

表 42 – 前のページからの続き

Column	Type	Description
nstream_truncate	bigint	処理されたストリーミング切り捨ての数
shared_blks_hit	bigint	サブスクリプションによる共有ブロックキャッシュヒットの合計数
shared_blks_read	bigint	サブスクリプションによって読み取られた共有ブロックの合計数
shared_blks_dirtied	bigint	サブスクリプションによってダーティー化された共有ブロックの合計数
shared_blks_written	bigint	サブスクリプションによって書き込まれた共有ブロックの合計数
blk_read_time	double precision	サブスクリプションがブロックの読み取りに費やした合計時間、ミリ秒単位（`track_io_timing`が有効になっている場合、それ以外の場合はゼロ）
blk_write_time	double precision	ミリ秒単位のブロックの書き込みにサブスクリプションが費やした合計時間（`track_io_timing`が有効になっている場合、それ以外の場合はゼロ）
connect_time	timestamp with time zone	現在のアップストリーム接続が確立された時間、接続されていない場合は NULL
last_disconnect_time	timestamp with time zone	最後のアップストリーム接続がドロップされた時間
start_lsn	pg_lsn	このサブスクリプションがアップストリームからのレプリケーションの開始を要求した LSN
retries_at_same_lsn	bigint	サブスクリプションが同じ LSN 値からリスタートされた回数
curr_ncommit	bigint	現在の接続が確立された後にこのサブスクリプションが行ったコミットの数
npre_commit_confirmati	bigint	CAMO パートナーによるプレコミット確認の数
npre_commit	bigint	プリコミットの数
ncommit_prepared	bigint	準備されたトランザクションのコミット数
nabort_prepared	bigint	準備されたトランザクションのアボートの数
nprovisional_waits	bigint	同時適用トランザクションによる同じタプルに対する更新/削除操作の数。これらは暫定的な待機です。 *Parallel Apply* を参照
ntuple_waits	bigint	安全に適用されるのを待っていた更新/削除操作の数。 *Parallel Apply* を参照
ncommit_waits	bigint	コミットされる前に待機する必要があった完全に適用されたトランザクションの数。 *Parallel Apply* を参照
stats_reset	timestamp with time zone	これらのサブスクリプション統計がリセットされた時刻

bdr.subscription

このカタログテーブルには、ローカル PGD ノードが所有するすべてのサブスクリプションとそのモードがリストされています。

bdr.subscription 列

Name	Type	Description
sub_id	oid	サブスクリプションの ID
sub_name	name	サブスクリプションの名前
nodegroup_id	oid	ノードグループの ID
origin_node_id	oid	起点ノードの ID
source_node_id	oid	ソースノードの ID
target_node_id	oid	対象ノードの ID
subscription_mode	char	サブスクリプションのモード
sub_enabled	bool	サブスクリプションが有効になっているかどうか（レプリケーションである必要があります）
apply_delay	interval	このサブスクリプションの変更の適用はどれくらい遅れますか（通常は 0）
slot_name	name	このサブスクリプションで使用されるアップストリームのスロット
origin_name	name	このサブスクリプションで使用されるローカルオリジン
num_writers	int	このサブスクリプションが使用するライタプロセスの数
streaming_mode	char	サブスクリプションのストリーミング構成
replication_sets	text[]	このサブスクリプションによってレプリケートされるレプリケーションセット（NULL = all）
forward_origin	text[]	このサブスクリプションによって転送されるオリジン（NULL = all）

bdr.subscription_summary

このビューには、ローカルノードが他のノードに対して持っているすべての PGD サブスクリプションに関する概要情報が含まれています。

bdr.subscription_summary 列

Name	Type	Description
node_group_name	name	ノードが属する PGD グループの名前
sub_name	name	サブスクリプションの名前
origin_name	name	起点ノードの名前
target_name	name	ターゲットノードの名前（通常はローカルノード）
sub_enabled	bool	サブスクリプションは有効ですか
sub_slot_name	name	このサブスクリプションで使用されるオリジンノードのスロット名
sub_replication_sets	text[]	サブスクライブしたレプリケーションセット
sub_forward_origins	text[]	サブスクリプションは、オリジン以外の他のノードから転送された変更を受け入れますか
sub_apply_delay	interval	オリジンと比較してこれだけトランザクションを遅らせる
sub_origin_name	name	このサブスクリプションで使用されるレプリケーションオリジン名
bdr_subscription_mode	char	サブスクリプションモード
subscription_status	text	サブスクリプションワーカーのステータス
node_group_id	oid	ノードが属する PGD グループの OID
sub_id	oid	サブスクリプションの OID
origin_id	oid	オリジンノードの OID
target_id	oid	ターゲットノードの OID
receive_lsn	pg_lsn	受信した変更またはメッセージの最新の LSN（これはリスタートの場合に戻ることができます）
receive_commit_lsn	pg_lsn	受信した最後の COMMIT の最新の LSN（これはリスタートの場合に戻ることができます）
last_xact_replay_lsn	pg_lsn	このサブスクリプションで再生された最後のトランザクションの LSN
last_xact_flush_lsn	timestamptz	ディスクに永続的にフラッシュされる、このサブスクリプションで再生された最後のトランザクションの LSN
last_xact_replay_timesta	timestamptz	このサブスクリプションで再生された最後のトランザクションのタイムスタンプ

bdr.tables

このビューには、レプリケーションセットのテーブルメンバーシップに関する情報が一覧表示されます。テーブルが複数のレプリケーションセットに存在する場合、このテーブルに複数回表示されます。

bdr.tables 列

Name	Type	Description
relid	oid	リレーションの OID
nspname	name	スキーマリレーションの名前は
relname	name	関係の名前
set_name	name	複製セットの名前
set_ops	text[]	レプリケートされたオペレーションのリスト
rel_columns	text[]	レプリケート列のリスト（NULL =すべての列）(*)
row_filter	text	行フィルタリング式
conflict_detection	text	使用される競合検出方法：row_origin（デフォルト）、row_version または column_level

(*) これらの列は将来の使用のために予約されており、現在 NULL である必要があります

bdr.taskmgr_work_queue

タスクマネージャーによって作成および処理された作業項目が含まれています。作業項目は 1 つのノードでのみ作成され、さまざまなノードで処理されます。

bdr.taskmgr_work_queue 列

Column	Type	Description
ap_wq_workid	bigint	作業項目の一意の ID
ap_wq_ruleid	int	autopartition_rules にリストされているルールの ID。ルールは bdr.autopartition コマンドを使用して指定されます
ap_wq_relname	name	タスクが属するリレーションの名前
ap_wq_relnamespace	name	この作業項目のルールで指定されたテーブルスペースの名前
ap_wq_partname	name	ワークアイテムによって作成されたパーティションの名前

次のページに続く

表 46 – 前のページからの続き

Column	Type	Description
ap_wq_work_category	char	作業カテゴリは、「c」（パーティションの作成）、「m」（パーティションの移行）、「d」（パーティションの削除）、「a」（パーティションの変更）のいずれかです。
ap_wq_work_sql	text	作業項目の SQL クエリ
ap_wq_work_depends	Oid[]	作業項目が依存するノードの OID

bdr.taskmgr_workitem_status

各ノードでローカルに更新される作業項目のステータス。

bdr.taskmgr_workitem_status 列

Column	Type	Description
ap_wi_workid	bigint	ワークアイテムの ID
ap_wi_nodeid	Oid	作業項目が処理されているノードの OID
ap_wi_status	char	ステータスは、「q」（キュー）、「c」（完了）、「f」（失敗）、「u」（不明）のいずれかです
ap_wi_started_at	timestamptz	作業項目の開始 timestamptz
ap_wi_finished_at	timestamptz	作業項目の終了 timestamptz

bdr.taskmgr_local_work_queue

タスクマネージャーによって作成および処理された作業項目が含まれています。これは `bdr.taskmgr_work_queue` に似ていますが、これらの作業項目はローカル管理テーブル用です。各ノードは、クラスター内の他のノードとは独立して、独自のローカル作業項目を作成して処理します。

bdr.taskmgr_local_work_queue 列

Column	Type	Description
ap_wq_workid	bigint	作業項目の一意の ID
ap_wq_ruleid	int	autopartition_rules にリストされているルールの ID。ルールは bdr.autopartition コマンドを使用して指定されます
ap_wq_relname	name	タスクが属するリレーションの名前
ap_wq_relnamespace	name	この作業項目のルールで指定された表領域の名前。

次のページに続く

表 48 – 前のページからの続き

Column	Type	Description
ap_wq_partname	name	ワークアイテムによって作成されたパーティションの名前
ap_wq_work_category	char	カテゴリは、「c」（パーティションの作成）、「m」（パーティションの移行）、「d」（パーティションの削除）、「a」（パーティションの変更）のいずれかです。
ap_wq_work_sql	text	作業項目の SQL クエリ
ap_wq_work_depends	Oid[]	常に NULL

bdr.taskmgr_local_workitem_status

ローカル管理テーブルの作業項目のステータス。

bdr.taskmgr_local_workitem_status 列

Column	Type	Description
ap_wi_workid	bigint	ワークアイテムの ID
ap_wi_nodeid	Oid	作業項目が処理されているノードの OID
ap_wi_status	char	ステータスは、「q」（キュー）、「c」（完了）、「f」（失敗）、「u」（不明）のいずれかです
ap_wi_started_at	timestamptz	作業項目の開始 timestamptz
ap_wi_finished_at	timestamptz	作業項目の終了 timestamptz

bdr.trigger

このビューでは、作成されたすべてのストリームトリガーを表示できます。多くの場合、ここでのトリガーは `bdr.create_conflict_trigger` から作成されます。

bdr.trigger 列

Name	Type	Description
trigger_id	oid	トリガーの ID
trigger_reloid	regclass	関連する機能の名前
trigger_pgtgid	oid	Postgres トリガー ID
trigger_type	char	トリガー呼び出しの種類
trigger_name	name	トリガーの名前

bdr.triggers

読みやすい列を含む `bdr.trigger` の展開図。

Name	Type	Description
trigger_name	name	トリガーの名前
event_manipulation	text	操作
trigger_type	bdr.trigger_type	トリガーの種類
trigger_table	bdr.trigger_reloid	それを呼ぶテーブル
trigger_function	name	使用した機能

bdr.workers

実行中の PGD ワーカープロセスに関する情報。

これを pid を使用して `bdr.stat_activity` と結合して、PGD ワーカーの状態をさらに詳しく知ることができます。

bdr.workers 列

Name	Type	Description
worker_pid	int	ワーカープロセスのプロセス ID
worker_role	int	ワーカーロールの数値表現
worker_role_name	text	ワーカーロールの名前
worker_subid	oid	ワーカーが 1 つに関連付けられている場合のサブスクリプション ID

bdr.writers

PGD ライタプロセスに関する特定の情報。

bdr.writers 列

Name	Type	Description
sub_name	name	サブスクリプションの名前
pid	int	ワーカープロセスのプロセス ID
syncing_rel	int	同期されるリレーションの OID（存在する場合）
streaming_allowed	text	このライタは、ライタへの直接ストリーミングのターゲットになりますか

次のページに続く

表 53 – 前のページからの続き

Name	Type	Description
is_streaming	bool	このライタにストリーミングされているトランザクションはありますか
remote_xid	xid	処理中のトランザクションのリモートトランザクション ID（ある場合）
remote_commit_lsn	pg_lsn	処理された最後のコミットの LSN
commit_queue_position	int	内部コミットキュー内の位置
nxacts	bigint	このライタによって処理されたトランザクションの数
ncommits	bigint	このライタによってコミットされたトランザクションの数
naborts	bigint	このライタによって中止されたトランザクションの数
nstream_file	bigint	このライタによって処理されたファイルへのストリーミングトランザクションの数
nstream_writer	bigint	このライタによって処理されたライターへのストリーミングトランザクションの数

bdr.worker_tasks

`bdr.worker_tasks` ビューは、PGD の現在のワーカー起動率制限状態と、バックグラウンドでのワーカー起動と登録アクティビティに関する基本的な統計を示します。

ここにリストされている他のビューとは異なり、現在のデータベースと PGD ノードに固有ではありません。現在の PostgreSQL インスタンス上のすべての PGD ノードの状態が表示されます。現在のデータベースに参加して、フィルタリングします。

`bdr.worker_tasks` は、walsender と出力プラグインを追跡しません。

bdr.worker_tasks 列

Column	Type	Description
task_key_worker_role	integer	ワーカーロール識別子
task_key_worker_role_n	text	Worker ロール名
task_key_dboid	oid	データベース識別子（利用可能な場合）
datname	name	データベースの名前（利用可能な場合）
task_key_subid	oid	サブスクリプション識別子（利用可能な場合）
sub_name	name	サブスクリプションの名前（利用可能な場合）
task_key_ext_libname	name	ライブラリの名前（おそらく bdr）
task_key_ext_funcname	name	関数エントリポイントの名前
task_key_ext_workernan	name	ワーカーに割り当てられた名前

次のページに続く

表 54 – 前のページからの続き

Column	Type	Description
task_key_remoterelid	oid	利用可能な場合、リモート同期関係の識別子
task_pid	integer	ワーカーのプロセス ID
task_registered	timestamp with time zone	ワーカー登録のタイムスタンプ
since_registered	interval	ワーカーが登録してからの間隔
task_attached	timestamp with time zone	ワーカーアタッチタイムスタンプ
since_attached	interval	ワーカーがアタッチされてからの間隔
task_exited	timestamp with time zone	ワーカー出口のタイムスタンプ
since_exited	interval	ワーカーが終了してからの間隔
task_success	boolean	ワーカーはまだ実行されていますか？
task_next_launch_not_b	timestamp with time zone	ワーカーが再び再起動されるタイムスタンプ
until_launch_allowed	interval	次の打ち上げまでの残り時間
task_last_launch_request	integer	起動を要求したプロセス ID
task_last_launch_request	timestamp with time zone	リクエストが行われたときのタイムスタンプ
since_last_request	interval	最後の要求からの間隔
task_last_launch_request	boolean	最後のリクエストは成功しましたか？
task_nrequests	integer	リクエスト数
task_nregistrations	integer	登録数
task_prev_pid	integer	前世代のプロセス ID
task_prev_registered	timestamp with time zone	前回登録したタスクのタイムスタンプ
since_prev_registered	interval	前回登録からの間隔
task_prev_launched	timestamp with time zone	前回の起動のタイムスタンプ
since_prev_launched	interval	前回の打ち上げからの間隔
task_prev_exited	timestamp with time zone	前のタスクが終了したときのタイムスタンプ
since_prev_exited	interval	前のタスクが終了してからの間隔
task_first_registered	timestamp with time zone	最初の登録が行われたときのタイムスタンプ
since_first_registered	interval	初回登録からの間隔

System functions

主に SQL から呼び出す関数を使用して、PGD 管理を実行します。PGD のすべての関数は、 `bdr` スキーマで公開されています。スキーマは、`search_path` に `bdr` を置く代わりに、これらの関数への呼び出しを修飾します。

バージョン情報機能

bdr.bdr_version

この関数は、現在使用されている BDR 拡張機能のバージョンのテキスト表現を取得します。

bdr.bdr_version_num

この関数は、現在使用されている BDR 拡張機能のバージョン番号を取得します。バージョン番号は単調に増加するため、この値を小なりと大なりの比較に使用できます。

次の式は、メジャーバージョン、マイナーバージョン、およびパッチリリースで構成されるバージョン番号を 1 つの数値に返します。

```
MAJOR_VERSION  *10000+MINOR_VERSION*  100 + PATCH_RELEASE
```

システム情報機能

bdr.get_relation_stats

リレーション情報を返します。

bdr.get_subscription_stats

現在のサブスクリプション統計を返します。

システムと進捗情報パラメーター

PGD は、 `psql` の `SHOW` を使用して、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）を使用してクエリできるいくつかのパラメーターを公開します。

bdr.local_node_id

セッションを初期化すると、これはクライアントが接続しているノード 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。

> 接続プールとプロキシ でも使用されます。

bdr.last_committed_lsn

非同期トランザクションの `COMMIT` ごとに、このパラメーターは更新され、起点ノードのコミットレコードの最後をポイントします。それを `bdr.wait_for_apply_queue` と組み合わせると、アプリケーションは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になるまで待機できます。

transaction_id

Postgres がトランザクション ID を割り当てるとすぐに、CAMO が有効になっている場合、このパラメーターが更新され、割り当てられたトランザクション ID が表示されます。

bdr.is_node_connected

概要

```
bdr.is_node_connected(node_name name)
```

特定のピアのウォルセンダーがこのノードでアクティブかどうかを確認して、ブール値を返します。

bdr.is_node_ready

概要

```
bdr.is_node_ready(node_name name, span interval DEFAULT NULL)
```

ラグが指定されたスパンよりも低いかどうか、または TO ASYNC の timeout よりも低いかどうかを確認して、ブール値を返します。

コンセンサス機能

bdr.consensus_disable

サーバーが再起動するまで、または bdr.consensus_enable を使用して再度有効になるまで（どちらか早い方）、ローカルノードのコンセンサスワーカーを無効にします。

> **警告:** コンセンサスを無効にすると、PGD の一部の機能が無効になり、長期間無効のままにした場合、EDB Postgres 分散クラスターの可用性に影響します。この機能は、テクニカルサポートと協力する場合にのみ使用してください。

bdr.consensus_enable

ローカルノードで無効になっているコンセンサスワーカーを再度有効にします。

bdr.consensus_proto_version

ローカルノードで現在使用されているコンセンサスプロトコルバージョンを返します。

PGD グループの再構成内部メカニズムによって必要とされます。

bdr.consensus_snapshot_export

概要

```
bdr.consensus_snapshot_export(version integer DEFAULT NULL)
```

ローカルノードの現在コミットおよび適用された状態から新しい PGD コンセンサススナップショットを生成し、bytea として返します。

デフォルトでは、サポートされている最も高い Raft バージョンのスナップショットがエクスポートされます。ただし、明示的な version 番号を渡すことにより、それをオーバーライドできます。

エクスポートするノードは現在の Raft リーダーである必要はなく、リーダーの最新の状態で完全に最新である必要もありません。ただし、bdr.consensus_snapshot_import() はそのようなスナップショットを受け入れない場合があります。

新しいスナップショットは、ローカルノードの bdr.local_consensus_snapshot テーブルに自動的に保存されません。呼び出し元にのみ返されます。

生成されたスナップショットは、エクスポートノードの Raft ログ位置の背後にある同じ PGD ノードグループ内の他のノードの bdr.consensus_snapshot_import() に渡される可能性があります。

この機能を動作させるには、ローカル PGD コンセンサスワーカーを無効にする必要があります。典型的な使用法は次のとおりです。

```
SELECT bdr.bdr_consensus_disable();
\copy (SELECT * FROM bdr.consensus_snapshot_export()) TO my_node_consensus_snapshot.
↪data
SELECT bdr.bdr_consensus_enable();
```

PGD コンセンサスワーカーが無効になっている間：

- ノードでの DDL ロックの試行が失敗するか、タイムアウトします。
- galloc シーケンスは新しい値を取得しません。
- Eager および CAMO トランザクションは一時停止またはエラー。
- 分散コンセンサスシステムを必要とする他の機能が中断されます。必要なダウンタイムは一般的に非常に短時間です。

ユースケースによっては、 bdr.local_consensus_snapshot テーブルの snapshot フィールドから既に存在するスナップショットを抽出し、代わりに使用する方が実用的な場合があります。そうすることで、コンセンサスワーカーを停止する必要はありません。

bdr.consensus_snapshot_import

概要

```
bdr.consensus_snapshot_import(IN snapshot bytea)
```

通常は同じ PGD ノードグループ内の別のノードから、bdr.consensus_snapshot_export() によってエクスポートされたコンセンサススナップショットをインポートします。

別のノード上の bdr.local_consensus_snapshot テーブルの snapshot フィールドから直接抽出したスナップショットを使用することもできます。

この関数は、破損またはユーザーエラーが発生した場合に、PGD ノードのカタログ状態を既知の良好な状態にリセットするのに役立ちます。

スナップショットが生成されたときにインポートノードの apply_index がスナップショットのエクスポートノードの commit_index 以下である場合、スナップショットをインポートできます。(bdr.get_raft_status() を参照。) ログがすでにはるか先にあるためにスナップショットを受け入れることができないノードは、エラーを発生させ、変更を行いません。スナップショットがインポートされると、ノードは現在のリーダーから残りの変更をフェッチするため、インポートされたスナップショットは完全に最新である必要はありません。

この機能を動作させるには、インポートノードで PGD コンセンサスワーカーを無効にする必要があります。詳細については、bdr.consensus_snapshot_export() のノートを参照してください。

次を実行することにより、この関数を使用して、ローカルノードに新しい Raft スナップショットを生成させることができます。

```
SELECT bdr.consensus_snapshot_import(bdr.consensus_snapshot_export());
```

このアプローチは、現在適用されているログ位置まで Raft ログを切り捨てる場合もあります。

bdr.consensus_snapshot_verify

概要

```
bdr.consensus_snapshot_verify(IN snapshot bytea)
```

bdr.consensus_snapshot_export() によってエクスポートされた特定のコンセンサススナップショットを確認します。スナップショットヘッダーには、それが生成されたバージョンが含まれており、ノードは同じバージョンに対して検証しようとします。

スナップショットは、同じノードまたはクラスター内の他のノードでエクスポートされた可能性があります。スナップショットを検証するノードが、エクスポートされたスナップショットのバージョンをサポートしていない場合、エラーが発生します。

bdr.get_consensus_status

現在のコンセンサス（Raft）ワーカーに関するステータス情報を返します。

bdr.get_raft_status

現在のコンセンサス（Raft）ワーカーに関するステータス情報を返します。bdr.get_consensus_status のエイリアス。

bdr.raft_leadership_transfer

概要

```
bdr.raft_leadership_transfer(IN node_name text, IN wait_for_completion boolean)
```

node_name で識別されるノードが Raft リーダーになることを要求します。要求は、任意の PGD ノードから開始でき、現在のリーダーに内部的に転送され、指定されたノードにリーダーシップを転送します。指定されたノードは、完全な投票権を持つ ACTIVE PGD ノードである必要があります。

wait_for_completion が false の場合、リクエストはベストエフォートベースで処理されます。bdr.raft_election_timeout 期間中にノードがリーダーになれない場合、他の能力のあるノードが再びリーダーになります。また、リーダーシップは、ラフトプロトコルごとに期間にわたって変わる可能性があります。true の戻り結果は、要求が正常に送信されたことのみを示します。

wait_for_completion が true である場合、関数は、指定されたノードが新しいリーダーになるまで待機し、要求されたノードが Raft リーダーにならない場合（ネットワークの問題などにより）、おそらく無限に待機します。したがって、無限ループを防ぐために、常に wait_for_completion で statement_timeout を設定することをお勧めします。

ユーティリティ関数

bdr.wait_slot_confirm_lsn

このセッションの最後の書き込みが 1 つまたはすべてのノードで再生されるまで待機できます。

スロットが特定の LSN を通過するまで待機します。位置が指定されていない場合、ローカルノードで現在の書き込み位置が使用されます。

スロット名が渡されない場合、すべての PGD スロットが LSN を渡すまで待機します。

この関数は、他のノードからの変更について 1000 ミリ秒ごとにポーリングします。

スロットが同時にドロップされた場合、そのスロットの待機は終了します。ノードが現在ダウンしており、スロットを更新していない場合、待機は継続されます。その場合、statement_timeout をより早く完了するように設定することをお勧めします。

概要

```
bdr.wait_slot_confirm_lsn(slot_name text DEFAULT NULL, target_lsn pg_lsn DEFAULT NULL)
```

パラメーター

- slot_name —レプリケーションスロットの名前、または、NULL の場合、すべての PGD スロット(のみ)。
- target_lsn —待機する LSN。または、NULL の場合、ローカルノード上の現在の書き込み LSN を使用します。

bdr.wait_for_apply_queue

関数 bdr.wait_for_apply_queue を使用すると、PGD ノードは、特定の PGD 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機できます。そのピアノードからのすべてのトランザクションがローカルに適用された後にのみ戻ります。アプリケーションまたはプロキシは、この機能を使用して、古い読み取りを防ぐことができます。

便宜上、PGD は CAMO および CAMO パートナーノードにこの機能のバリアントを提供します。
bdr.wait_for_camo_partner_queue を参照してください。

特定の LSN が指定されている場合、それはピアが待機する復旧ストリームのポイントです。これを、以前の接続または同時接続でそのピアノードから取得した bdr.last_committed_lsn で使用できます。

指定された target_lsn が NULL の場合、この関数はローカル受信バッファーをチェックし、指定されたピアノードから受信した最後のトランザクションの LSN を使用し、既に受信したすべてのトランザクションが適用されるのを効果的に待機します。これは、ピアノードに障害が発生し、どのトランザクションが送信されたかがわからない場合に特に役立ちます。この場合、送信側でまだ転送中またはバッファリングされているトランザクションは待機されません。

概要

```
bdr.wait_for_apply_queue(peer_node_name TEXT, target_lsn pg_lsn)
```

パラメーター

- peer_node_name —着信トランザクションがキューに入れられて待機するピアノードの名前。NULL の場合、すべてのピアノードの適用キューが消費されるのを待ちます。
- target_lsn —待機するピアノードからのレプリケーションストリーム内の LSN。通常、ピアノードから bdr.last_committed_lsn を介して学習されます。

bdr.get_node_sub_receive_lsn

サブスクライバーでこの関数を使用して、特定のオリジンから受信した最後の LSN を取得できます。適用するトランザクションに関連する LSN インクリメントのみを考慮に入れるように、フィルター処理しないか、フィルター処理することができます。

この関数の出力と `bdr.get_node_sub_apply_lsn()` の出力の差は、対応する適用キューのサイズを測定します。

概要

```
bdr.get_node_sub_receive_lsn(node_name name, committed bool default true)
```

パラメーター

- `node_name` —LSN を取得するレプリケーションストリームのソースであるノードの名前。
- `committed` —; デフォルト（true）では、この関数は最後の LSN 全体ではなく、受信したトランザクションのコミットのみを考慮します。これには、サブスクライバーノードに影響を与えないアクションが含まれます。

bdr.get_node_sub_apply_lsn

サブスクライバーでこの関数を使用して、特定のオリジンから受信および適用された最後の LSN を取得できます。

概要

```
bdr.get_node_sub_apply_lsn(node_name name)
```

パラメーター

- `node_name` —LSN が取得されるレプリケーションストリームのソースであるノードの名前。

bdr.run_on_all_nodes

すべてのノードでクエリを実行する関数。

> **警告:** この関数は、ノードの DSN で指定されたノード間接続に使用されるユーザーの特権で、リモートノードで任意のクエリを実行します。このファンクションに権限を付与するときは注意してください。

概要

```
bdr.run_on_all_nodes(query text)
```

パラメーター

- query —実行する任意のクエリ。

注意事項

この関数は、他のノードに接続してクエリを実行し、それぞれからの結果を JSON 形式で返します。JSON 配列としてエンコードされた、各ノードから複数の行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 または他の実行時パラメーターは設定されていないため、デフォルトが使用されます。

この関数は、通常のレプリケーションを行いません。既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドは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、接続は bdr.ddl_replication = off で作成されます。

レプリケーションを壊し、ノード間の矛盾を引き起こす危険があるため、この機能を使用するときは注意してください。透過的な DDL レプリケーションまたは bdr.replicate_ddl_command() を使用して DDL をレプリケートします。DDL は、将来のリリースでブロックされる可能性があります。

例

次のクエリのように、監視でこの関数を使用すると便利です。

```
SELECT bdr.run_on_all_nodes($$
    SELECT local_slot_name, origin_name, target_name, replay_lag_size
      FROM bdr.node_slots
     WHERE origin_name IS NOT NULL
$$);
```

このクエリは、2 ノードクラスターで次のようなものを返します。

```
[
    {
        "dsn": "host=node1 port=5432 dbname=bdrdb user=postgres ",
        "node_id": "2232128708",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
                    "origin_name": "node1",
```

(次のページに続く)

(前のページからの続き)


```
                    "target_name": "node2",
                    "local_slot_name": "bdr_bdrdb_bdrgroup_node2",
                    "replay_lag_size": "0 bytes"
                }
            ]
        },
        "node_name": "node1"
    },
    {
        "dsn": "host=node2 port=5432 dbname=bdrdb user=postgres ",
        "node_id": "2058684375",
        "response": {
            "command_status": "SELECT 1",
            "command_tuples": [
                {
                    "origin_name": "node2",
                    "target_name": "node1",
                    "local_slot_name": "bdr_bdrdb_bdrgroup_node1",
                    "replay_lag_size": "0 bytes"
                }
            ]
        },
        "node_name": "node2"
    }
]
```


bdr.run_on_nodes

指定されたノードのリストでクエリを実行する関数。

> **警告:** この関数は、ノードの DSN で指定されたノード間接続に使用されるユーザーの特権で、リモートノードで任意のクエリを実行します。このファンクションに権限を付与するときは注意してください。

概要

```
bdr.run_on_nodes(node_names text[], query text)
```

パラメーター

- node_names —クエリが実行されるノード名の Text ARRAY。
- query —実行する任意のクエリ。

注意事項

この関数は、他のノードに接続してクエリを実行し、それぞれからの結果を JSON 形式で返します。JSON 配列としてエンコードされた、各ノードから複数の行を返すことができます。ノードがダウンしているため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 または他の実行時パラメーターは設定されていないため、デフォルトが使用されます。

この関数は、通常のレプリケーションを行いません。既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドは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、接続は bdr.ddl_replication = off で作成されます。

レプリケーションを壊し、ノード間の矛盾を引き起こす危険があるため、この機能を使用するときは注意してください。透過的な DDL レプリケーションまたは bdr.replicate_ddl_command() を使用して DDL をレプリケートします。DDL は、将来のリリースでブロックされる可能性があります。

bdr.run_on_group

ノードのグループでクエリを実行する関数。

> **警告:** この関数は、ノードの DSN で指定されたノード間接続に使用されるユーザーの特権で、リモートノードで任意のクエリを実行します。このファンクションに権限を付与するときは注意してください。

概要

```
bdr.run_on_group(node_group_name text, query text)
```

パラメーター

- node_group_name —クエリが実行されるノードグループの名前。
- query —実行する任意のクエリ。

注意事項

この関数は、他のノードに接続してクエリを実行し、それぞれからの結果を JSON 形式で返します。JSON 配列としてエンコードされた、各ノードから複数の行を返すことができます。ノードがダウンしているため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 または他の実行時パラメーターは設定されていないため、デフォルトが使用されます。

この関数は、通常のレプリケーションを行いません。既知のすべてのノードへの直接クライアント接続を使用します。デフォルトでは、コマンドは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、接続は bdr.ddl_replication = off で作成されます。

レプリケーションを壊し、ノード間の矛盾を引き起こす危険があるため、この機能を使用するときは注意してください。透過的な DDL レプリケーションまたは bdr.replicate_ddl_command() を使用して DDL をレプリケートします。DDL は、将来のリリースでブロックされる可能性があります。

bdr.global_lock_table

このファンクションは、指定されたテーブルのグローバル DML ロックを取得します。グローバル DML ロックの詳細は、
DDL ロックの詳細 を参照してください。

概要

```
bdr.global_lock_table(relation regclass)
```

パラメーター

- relation —ロックするリレーションの名前または oid。

注意事項

このファンクションは、 ddl_locking 設定とは無関係にグローバル DML ロックを取得します。

bdr.global_lock_table ファンクションには、bdr.backwards_compatibility が 30618 以下に設定されていない限り、ロックされた relation に対する UPDATE 、DELETE 、または TRUNCATE 特権が必要です。

bdr.wait_for_xid_progress

この関数を使用して、特定のノード（ノード ID で識別）で発生した特定のトランザクション（XID で識別）がクラスターで十分に進行するのを待つことができます。進行状況は、ノードに適用されているトランザクションとして定義され、このノードは、トランザクションが適用される前に行われた他のすべてのレプリケーション変更を確認しています。

概要

```
bdr.wait_for_xid_progress(origin_node_id oid, origin_topxid int4, allnodes boolean␣
↪DEFAULT true)
```

パラメーター

- origin_node_id —トランザクションが発生したノードのノード ID。
- origin_topxid —トランザクションの XID。
- allnodes — true の場合、すべてのノードでトランザクションが進行するのを待ちます。それ以外の場合は、現在のノードのみを待ちます。

注意事項

DDL コマンドをレプリケートしたトランザクションのみが現在追跡されているため、この関数を使用できます。間違った origin_node_id または origin_topxid が指定された場合、関数は永久に、または statement_timeout が発生するまで待機する場合があります。

bdr.local_group_slot_name

ローカルノードのグループスロットの名前を返します。

例

```
bdrdb=# SELECT bdr.local_group_slot_name();
 local_group_slot_name
- ----------------------
 bdr_bdrdb_bdrgroup
```

bdr.node_group_type

指定されたノードグループのタイプを返します。戻り値は、ノードグループの作成時に bdr.create_node_group() に渡されたものと同じですが、グループの作成時に node_group_type が NULL として渡された場合、 normal が返されます。

例

```
bdrdb=# SELECT bdr.node_group_type(bdrgroup);
 node_group_type
- ----------------
 normal
```

bdr.alter_node_kind

PGD5 は、タスクマネージャーリーダーノードの概念を導入しました。ノードは PGD によって自動的に選択されますが、アップグレードされたクラスターの場合、クラスター内のすべてのノードに対して node_kind を適切に設定することが重要です。ユーザーは、最新の PGD バージョンにアップグレードした後、各ノードの bdr.alter_node_kind() SQL 関数を呼び出して、これを手動で行う必要があります。

概要

```
bdr.alter_node_kind(node_name text,
                    node_kind text);
```

パラメーター

- node_name —種類を変更するノードの名前。
- node_kind —ノードの種類。 data 、 standby 、 witness 、または subscriber-only のいずれか。

bdr.alter_subscription_skip_changes_upto

論理レプリケーションはバージョン間でレプリケートでき、ロールのようなグローバルな変更をレプリケートせず、選択的にレプリケートできるため、論理レプリケーション適用プロセスでエラーが発生し、変更の適用が停止する場合があります。

可能な限り、ターゲット側に変更を加えて、このような問題を修正します。CREATE レプリケーションをブロックしている欠落しているテーブル、CREATE 必要なロール、 GRANT 必要な権限など。ただし、場合によっては問題をその方法で修正できず、トランザクションを完全にスキップする必要がある場合があります。変更は、トランザクション全体としてスキップされます。スキップする場所を決定するには、次の例のようにログ出力を使用してコミット LSN を見つけるか、論理デコード関数で変更ストリームをピークします。

トランザクションが 1 つだけ変更を加えない限り、多くの場合、ターゲット側にトランザクションの効果を手動で適用する必要があるため、次の例に示すように、可能な限り問題のあるトランザクションを保存することが重要です。

pg_catalog.pg_logical_slot_get_binary_changes を使用して目的の LSN にスキップすることにより、bdr.alter_subscription_skip_changes_upto なしで変更をスキップすることが可能であるため、これは便利な関数です。論理デコードでいくつかの種類のエラーをバイパスする可能性がありますが、より高速にスキップします。

この機能は、無効になっているサブスクリプションでのみ機能します。

通常の手順は次のとおりです。

1. 問題のサブスクリプションと問題のコミットの LSN を特定します。

1. サブスクリプションを無効にします。

1. 可能であれば、ソースノードで pg_catalog.pg_logical_slot_peek_changes を使用してトランザクションのコピーを保存します。

1. ターゲットノード上の bdr.alter_subscription_skip_changes_upto 。

1. 必要に応じて、修復した変更または同等の変更をターゲットに手動で適用します。

1. サブスクリプションを再度有効にします。

> **警告:** この関数を使用すると、問題を悪化させるのは簡単です。それが唯一の選択肢であると確信できる場合を除き、何もしないでください。

概要

```
bdr.alter_subscription_skip_changes_upto(
  subname text,
  skip_upto_and_including pg_lsn
);
```

例

トランザクションの適用がエラーで失敗し、ターゲット側での変更などの影響の少ない修正ではこの問題を解決できないと判断しました。あなたは、トランザクションをスキップする必要があると判断します。

次の例のように、エラーログで、スキップするコミットレコード LSN を見つけます。

```
ERROR:  XX000: CONFLICT: target_table_missing; resolver skip_if_recently_dropped␣
↪returned an error: table does not exist
CONTEXT:  during apply of INSERT from remote relation public.break_me in xact with␣
↪commit-end lsn 0/300AC18 xid 131315
committs 2021-02-02 15:11:03.913792+01 (action #2) (effective sess origin id=2 lsn=0/
```

(次のページに続く)

(前のページからの続き)

```
↪300AC18)
while consuming I message from receiver for subscription bdr_regression_bdrgroup_
↪node1_node2 (id=2667578509)
on node node2 (id=3367056606) from upstream node node1 (id=1148549230, reporiginid=2)
```

ログのこの部分には、必要な情報があります。the_target_lsn： **0/300AC18** the_subscription： **bdr_regression_bdrgroup_node1_node2**

次に、サブスクリプションを無効にして、適用ワーカーがレプリケーションスロットに接続しようとしないようにします。

```
SELECT bdr.alter_subscription_disable(the_subscription);
```

トランザクションの一部だけをスキップすることはできません。それはすべてか無かです。そのため、サブスクリプションのスロット名を使用して、最初にプロバイダー側でコピーして、そのレコードを保存することを強くお勧めします。

```
\copy (SELECT * FROM pg_catalog.pg_logical_slot_peek_changes(the_slot_name,
    the_target_lsn, NULL, min_proto_version, 1, max_proto_version, 1,
    startup_params_format, 1, proto_format, json))
 TO transaction_to_drop.csv WITH (FORMAT csv);
```

この例は、読みやすくするために複数行に分割されていますが、1 行で発行してください。\copy は複数行のコマンドをサポートしていません。

peek を get に変更することで変更をスキップできますが、bdr....skip_changes_upto はより高速にスキップし、すべてのデータのデコードと出力を回避します。

```
SELECT bdr.alter_subscription_skip_changes_upto(subscription_name,
    the_target_lsn);
```

ダンプされたトランザクションの内容をガイドとして使用して、同じ変更（またはそれらの修復バージョン）をターゲットノードに手動で適用できます。

最後に、サブスクリプションを再度有効にします。

```
SELECT bdr.alter_subscription_enable(the_subscription);
```

グローバルアドバイザリーロック

PGD は、グローバル勧告ロックをサポートしています。これらのロックは、PGD でサポートされる勧告ロックがグローバルであることを除いて、PostgreSQL で利用可能な勧告ロックに似ています。これらは、DDL ロックと同様のセマンティクスに従います。したがって、アドバイザリーロックは多数決コンセンサスによって取得され、すべてのノードの過半数が連携できる限り、1 つ以上のノードがダウンまたは遅れている場合でも使用できます。

現在、 EXCLUSIVE ロックのみがサポートされています。したがって、別のノードまたは同じノード上の別のバックエンドがオブジェクトの勧告ロックをすでに取得している場合、他のノードまたはバックエンドはロックが解放されるまで待つ必要があります。

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。したがって、トランザクションの終了前に明示的に解放されない限り、トランザクションの終了時にロックは自動的に解放されます。この場合、リリースされ次第利用可能になります。セッションレベルのアドバイザリロックは現在サポートされていません。

グローバル勧告ロックはリエントラントです。したがって、同じリソースが 3 回ロックされている場合、他のセッションで使用できるように解放するには、3 回ロック解除する必要があります。

bdr.global_advisory_lock

このファンクションは、指定されたオブジェクトの EXCLUSIVE ロックを取得します。ロックが利用できない場合、ロックが利用可能になるか、bdr.global_lock_timeout に到達するまで待機します。

概要

```
bdr.global_advisory_lock(key bigint)
```

パラメーター

- key —アドバイザリロックが取得されるオブジェクト。

概要

```
bdr.global_advisory_lock(key1 integer, key2 integer)
```

パラメーター

- key1 —複合キーの最初の部分。
- key2 —複合キーの 2 番目の部分。

bdr.global_advisory_unlock

この関数は、アプリケーション定義のソースで以前に取得したロックを解放します。ロックは、アプリケーションが同じトランザクションで取得している必要があります。それ以外の場合、エラーが発生します。

概要

```
bdr.global_advisory_unlock(key bigint)
```

パラメーター

- key —アドバイザリロックが取得されるオブジェクト。

概要

```
bdr.global_advisory_unlock(key1 integer, key2 integer)
```

パラメーター

- key1 —複合キーの最初の部分。
- key2 —複合キーの 2 番目の部分。

監視機能

bdr.monitor_group_versions

クラスター全体のバージョンチェックを提供するために、この関数はビュー bdr.group_version_details から返される PGD バージョン情報を使用します。

概要

```
bdr.monitor_group_versions()
```

注意事項

Monitoring で説明されているように、この関数は、フィールド status および message を持つレコードを返します。

この関数は bdr.run_on_all_nodes() を呼び出します。

bdr.monitor_group_raft

クラスター全体の Raft チェックを提供するために、この関数はビュー bdr.group_raft_details から返された PGD Raft 情報を使用します。

概要

```
bdr.monitor_group_raft()
```

パラメーター

- node_group_name —チェックするノードグループ名。

注意事項

Monitoring で説明されているように、この関数は、フィールド status および message を持つレコードを返します。

この関数は bdr.run_on_all_nodes() を呼び出します。

bdr.monitor_local_replslots

この関数は、ビュー pg_replication_slots（スロットアクティブまたは非アクティブ）から返されたレプリケーションスロットステータス情報を使用して、PGD グループスロットを除くすべてのレプリケーションスロットを考慮したローカルチェックを提供します。

概要

```
bdr.monitor_local_replslots()
```

注意事項

レプリケーションスロットのモニタリング で説明されているように、この関数は、フィールド status および message を持つレコードを返します。

bdr.wal_sender_stats

デコードワーカー が有効になっている場合、この関数は、デコーダースロットと各 WAL 送信者によって読み取られている現在の LCR（論理変更レコード）セグメントファイルに関する情報を表示します。

概要

```
bdr.wal_sender_stats()
```

出力列

- pid — WAL 送信者の PID（ pg_stat_replication の pid 列に対応）。
- is_using_lcr — WAL 送信者が LCR ファイルを送信しているかどうか。is_using_lcr が FALSE の場合、次の列は NULL です。
- decoder_slot_name —デコーダー複製スロットの名前。
- lcr_file_name —現在の LCR ファイルの名前。

bdr.get_decoding_worker_stat

デコードワーカー が有効になっている場合、この関数は、現在のデータベースに関連付けられているデコードワーカーの状態に関する情報を表示します。これはまた、

pg_replication_slots を介して利用可能なものよりも、デコードワーカーの進行状況に関するより詳細な情報を提供します。

概要

```
bdr.get_decoding_worker_stat()
```

出力列

- pid —デコードワーカーの PID（ pg_replication_slots の active_pid 列に対応）。
- decoded_upto_lsn —デコードワーカーがトランザクションログを読み取るまでの LSN。
- waiting —デコードワーカーが新しい WAL を待っているかどうか。
- waiting_for_lsn —次に予想される WAL の LSN。

注意事項

詳細については、 LCR を使用した WAL 送信者のモニタリング を参照してください。

bdr.lag_control

Lag Control が有効になっている場合、この機能は、コミット遅延と、ローカルノードと現在のデータベースに構成されているラグメジャーに準拠するノードの数に関する情報を表示します。

概要

```
bdr.lag_control()
```

出力列

- commit_delay —現在のランタイムコミット遅延（ミリ秒未満）。
- commit_delay_maximum —構成された最大コミット遅延（ミリ秒未満）。
- commit_delay_adjustment —サンプル間隔中に可能なランタイムコミット遅延への変更（ミリ秒未満）。
- conforming_nodes —ラグメジャーに準拠するノードの現在のランタイム数。
- conforming_nodes_minimum —ラグメジャーに準拠するために必要なノードの構成された最小数。それ以下ではコミット遅延調整が適用されます。
- lag_bytes_threshold —コミット遅延が適用されるラグサイズ（キロバイト）。
- lag_bytes_maximum —構成された最大ラグサイズ（キロバイト単位）。
- lag_time_threshold —ミリ秒単位のコミット遅延が適用されるラグタイム。
- lag_time_maximum —構成された最大ラグタイム（ミリ秒）。
- sample_interval —ミリ秒単位のラグサンプルと可能なコミット遅延調整間の構成された最小時間。

PGD settings

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

コンフリクトハンドリング

bdr.default_conflict_detection

新しく作成されたテーブルのデフォルトの競合検出方法を設定します。

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

グローバルシーケンスパラメーター

bdr.default_sequence_kind

デフォルトの *sequence kind* を設定します。

デフォルト値は `distributed` です。 これは、 `snowflakeid` が `int8` シーケンス（つまり `bigserial` ）に使用され、`galloc` シーケンスが `int4` （つまり `serial` ）および `int2` シーケンスに使用されることを意味します。

DDL ハンドリング

bdr.default_replica_identity

新しく作成されたテーブルの `REPLICA IDENTITY` のデフォルト値を設定します。 `REPLICA IDENTITY` は、更新または削除された行を特定するログ先行書き込みに書き込まれる情報を定義します。

受け入れられる値は次のとおりです。

- `DEFAULT` —主キーの列の古い値があれば記録します（これは PostgreSQL のデフォルトの動作です）。
- `FULL` —行内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。

PGD は、 `PRIMARY KEY` または `UNIQUE` 制約のないテーブルで `UPDATE` および `DELETE` 操作をレプリケートできません。例外は、テーブル固有の構成または `bdr.default_replica_identity` のいずれかにより、テーブルのレプリカ ID が `FULL` である場合です。

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

テーブルのレプリカ ID を `FULL` に設定すると、書き込まれる WAL の量とテーブルのワイヤーで複製されるデータの量が増加します。

bdr.ddl_replication

ノード間で DDL を自動的に複製します（デフォルトは on ）。

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

bdr.ddl_replication = off で DDL を実行するか、PGD 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。詳細は *DML および DDL レプリケーション* を参照してください。

bdr.ddl_replication が off に設定されるたびに、LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定が原因でキャプチャされた DDL コマンドまたは PGD 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るたびに、WARNING-level メッセージが書き込まれます。

bdr.role_replication

ノード間で ROLE コマンドを自動的に複製します（デフォルトは on ）。このパラメーターを設定できるのはスーパーユーザーのみです。この設定は、bdr.ddl_replication 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、すべてのノードでロールが同期されていることを確認すると、複製された DDL により、管理者が介入するまで複製が中断される場合があります。

> ロール操作ステートメント を参照

詳細については。

bdr.ddl_locking

DDL のグローバルロックの動作モードを構成します。

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

可能なオプションは次のとおりです。

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

bdr.ddl_replication が off に設定されるたびに、LOG レベルのログメッセージが PostgreSQL サーバーログに出力されます。さらに、この設定によりグローバルロックステップがスキップされるたびに、WARNING メッセージが書き込まれます。一部のステートメントの結果、2 つの WARNING メッセージが表示されるのは正常です。1 つは DML ロックをスキップするため、もう 1 つは DDL ロックをスキップするためです。

> グローバルロック も参照

bdr.truncate_locking

この構成オプションは、 TRUNCATE コマンドのロック動作を設定します。(true の場合) TRUNCATE が `bdr.ddl_locking` 設定に従うかどうかを判断します。デフォルトで false に設定されています。

グローバルロック

DDL ロックは *bdr.ddl_locking* によって制御されます。その他のグローバルロック設定には、次のものがあります。

bdr.global_lock_max_locks

ノードで保持できるグローバルロックの最大数を設定します (デフォルト 1000)。Postgres サーバーの起動時にのみ設定できます。

bdr.global_lock_timeout

グローバルロックの待機の最大許容時間を設定します (デフォルトは 10 分)。ゼロの値はこのタイムアウトを無効にします。

bdr.global_lock_statement_timeout

グローバルロックを保持するステートメントの最大許容期間を設定します (デフォルト 60 分)。ゼロの値はこのタイムアウトを無効にします。

bdr.global_lock_idle_timeout

グローバルロックを保持しているトランザクションのアイドル時間の最大許容時間を設定します (デフォルト 10 分)。ゼロの値はこのタイムアウトを無効にします。

bdr.predictive_checks

予測チェックのログレベルを設定します (現在、グローバルロックでのみ使用されます)。`DEBUG` 、`LOG` 、`WARNING` (デフォルト)、または `ERROR` のいずれかです。予測チェックは、特定の操作を実行するときに予想されるクラスター状態の早期検証です。タイムアウトを待つのではなく、早期に失敗するための操作に使用できます。グローバルロックの用語では、PGD は、十分なノードが接続されており、グローバルロックに必要なクォーラムを取得するための合理的なラグ制限があることを確認します。

ノード管理

bdr.replay_progress_frequency

クラスターの残りの部分にレプリケーション位置情報を送信する間隔を設定します（デフォルト 1 分）。

bdr.standby_slot_names

他のスロットよりも前にレプリケーションの変更を受信および確認するために必要なスロットを設定します。この設定は、主にフェイルオーバーにフィジカルスタンバイを使用する場合、またはサブスクライブ専用ノードを使用する場合に役立ちます。

一般的なレプリケーション

bdr.writers_per_subscription

サブスクリプションあたりのデフォルトのライター数を設定します（PGD では、グループの `bdr.alter_node_group_config` でこれを変更することもできます）。

bdr.max_writers_per_subscription

サブスクリプションあたりのライターの最大数（上記の設定の上限を設定します）。

bdr.xact_replication

現在のトランザクションを複製します（デフォルトは on ）。

これをオフにすると、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、PGD および論理デコードの他のすべてのダウンストリームターゲットによる論理デコードから可視ではありません。データは、ロジカルスタンバイノードを含む他のノードに転送されません。

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

このパラメーターは、`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。

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

警告: *bdr.xact_replication* をオフにすると、ノード間のデータの矛盾につながります。ノード間のデータ相違から回復するため、またはレプリケーションを続行するために単一ノードでの変更が必要なレプリケーション状況でのみ使用してください。ご自身の責任で使用してください。

bdr.permit_unsafe_commands

一般的な使用には安全でないとみなされるコマンドの安全性チェックをオーバーライドするオプション。

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

警告: 通常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。ご自身の責任で使用してください。

bdr.batch_inserts

単一のトランザクションで 1 つのテーブルへの連続挿入の数は、そのテーブルへの挿入のバッチ処理をオンにします。

この設定により、大規模なデータのレプリケーションは、挿入のセットではなく、内部的に COPY としてロードできます。ノードジョイン時の初期データもコピーされます。

bdr.maximum_clock_skew

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

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

デフォルトは-1 で、これはクロックスキューを無視することを意味します（チェックはオフになっています)。すべてのサーバーの時計を同期する場合は 0 を設定すると有効です。トランザクションをリプレイ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

この設定は、 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。

この設定には 2 つの値があります。

- WARN —この事実に関する警告をログに記録します。警告は、サーバーログのフラッディングを防ぐために、最大で毎分 1 回（デフォルト）ログに記録されます。
- WAIT —現在のローカルタイムスタンプが、リモートコミットタイムスタンプから bdr.maximum_clock_skew を引いたものより古くなくなるまで待ちます。

bdr.accept_connections

この設定は、PGD への接続を有効または無効にします。デフォルトは on です。

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

bdr.standby_slot_names

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

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

PGD の論理レプリケーションウォルセンダーは、ノードが他の PGD レプリケーションクライアントにこれらの変更を送信する前に、すべてのローカル変更が bdr.standby_slot_names のレプリケーションスロットに送信されてフラッシュされることを保証します。効果的に、スロットの名前付きリストと他のすべてのレプリケーションクライアントの間に同期レプリケーションバリアを提供します。

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

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

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

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

bdr.standby_slot_names が役立つ別のユースケースは、サブスクライバー専用ノードを使用して、通常の PGD ノードよりも先に移動しないようにする場合です。これは、bdr.standby_slots_min_confirmed を少なくとも 1 に設定することと組み合わせて、すべての通常の PGD ピアノードの論理スロットをリストすることで最適に達成できます。

bdr.standby_slots_min_confirmed

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

bdr.writer_input_queue_size

この設定は、受信側がライタプロセスにデータを送信するために使用する共有メモリキューのサイズを指定します。ライタプロセスが停止しているか、進行が遅い場合、キューがいっぱいになり、受信側プロセスも停止する可能性があります。したがって、このキューに十分な共有メモリを提供することが重要です。デフォルトは 1MB で、許可される最大サイズは 1GB です。任意のストレージサイズ指定子を使用して GUC を設定できますが、デフォルトは KB です。

bdr.writer_output_queue_size

この設定は、受信側がライタプロセスからデータを受信するために使用する共有メモリキューのサイズを指定します。ライタは大量のデータを送信することは想定されていないため、比較的小さなサイズのキューで十分です。デフォルトは 32 KB で、許可される最大サイズは 1 MB です。任意のストレージサイズ指定子を使用して GUC を設定できますが、デフォルトは KB です。

bdr.min_worker_backoff_delay

この設定により、特定のワーカーが bdr.min_worker_backoff_delay ミリ秒ごとよりも頻繁に再起動されるのを防ぐことにより、PGD バックグラウンドワーカーの起動のレート制限が可能になります。エラーが繰り返されると、最大 bdr.max_worker_backoff_delay までジッターが追加され、バックオフが指数関数的に増加します。

時間単位のサフィックスがサポートされています。

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

---

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

バックオフ遅延設定が変更され、PostgreSQL 構成がリロードされると、現在のすべてのバックオフはリセットを待機します。さらに、bdr.worker_task_reset_backoff_all() ファンクションが提供されているため、管理者はすべてのバックオフ間隔を強制的にすぐに期限切れにすることができます。

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

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

レート制限の目的で、ワーカーはタスク別に分類されます。このキーは、ワーカーロール、データベース OID、サブスクリプション ID、サブスクリプションライタ ID、拡張機能ライブラリの名前と関数名、拡張機能が提供するワーカー名、および同期ライターのリモートリレーション ID で構成されています。NULL は、特定の分類子が適用されない場合に使用されます。たとえば、マネージャーワーカーにはサブスクリプション ID がなく、レシーバーにはライタ ID がありません。

CRDT

bdr.crdt_raw_value

CRDT データ型を使用した列の競合の処理 の出力形式を設定します。

デフォルトの出力（この設定が off の場合）は、ベース CRDT タイプの現在の値（たとえば、crdt_pncounter の bigint）のみを返します。 on に設定すると、戻り値は CRDT 値の完全な表現を表し、たとえば、複数のノードからの状態を含めることができます。

Eager Replication

bdr.commit_scope

現在の（またはデフォルトの） *commit* スコープへの移行 （デフォルト local ）を設定します。

Commit At Most Once

bdr.camo_local_mode_delay

CAMO パートナーがトランザクションを確認する必要がある場合に通常発生するオーバーヘッドをエミュレートするために、CAMO の非同期モードで適用されるコミット遅延。デフォルトは 5 ミリ秒です。この機能を無効にするには 0 に設定します。 -

bdr.camo_enable_client_warnings

CAMO プロパティを保証できないデータベースでアクティビティが実行された場合、警告を発します。これはデフォルトで有効になっています。十分な知識のあるユーザーは、これを無効にして、ログに記録される警告の量を減らすことができます。

トランザクションストリーミング

bdr.default_streaming_mode

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

トランザクションストリーミング を参照してください。

ラグコントロール

bdr.lag_control_max_commit_delay

許容できる最大許容ポストコミット遅延（ミリ秒単位）。

bdr.lag_control_max_lag_size

許容できる最大許容ラグサイズ（キロバイト単位）。

bdr.lag_control_max_lag_time

許容できる最大の許容ラグタイム（ミリ秒）。

bdr.lag_control_min_conforming_nodes

許容可能なラグメジャーを下回るために必要なノードの最小数。

bdr.lag_control_commit_delay_adjust

コミット遅延のマイクロ調整は、最大コミット遅延時間の割合として測定されます。デフォルト値 0.01 % では、最大コミット遅延に到達するには 100 正味増分が必要です。

bdr.lag_control_sample_interval

ラグサンプルとコミット遅延マイクロ調整間の最小時間（ミリ秒）。

bdr.lag_control_commit_delay_start

コミット遅延増分の適用が開始されるラグしきい値。許容可能なラグメジャーの割合として表されます。デフォルト値の 1.0 %では、許容可能なラグ測定値を超えるまで、コミット遅延の増分は開始されません。

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

タイムスタンプベースのスナップショット

bdr.timestamp_snapshot_keep

タイムスタンプベースのスナップショットで使用する有効なスナップショットを保持する時間（デフォルトは `0`、過去のスナップショットを保持しないことを意味します）。

モニタリングとロギング

bdr.debug_level

PGD がデバッグメッセージを書き込むために使用するログレベルを定義します。デフォルト値は debug2 です。詳細な PGD デバッグ出力を見たい場合は、 `bdr.debug_level = 'log'` を設定します。

bdr.trace_level

上記と同様に、これは PGD トレースメッセージに使用するログレベルを定義します。EDB Postgres 分散クラスターのすべてのノードでトレースを有効にすると、EDB Support が問題を診断するのに役立つ場合があります。これは、Postgres サーバーの起動時にのみ設定できます。

> **警告:** *bdr.debug_level* または `bdr.trace_level` を値>= *log_min_messages* に設定すると、非常に大量のログ出力が生成される可能性があります。

bdr.track_subscription_apply

各サブスクリプションの適用統計を追跡します。

bdr.track_relation_apply

各関係の適用統計を追跡します。

bdr.track_apply_lock_timing

リレーションの統計を追跡するときにロックのタイミングを追跡します。

デコードワーカー

bdr.enable_wal_decoder

> デコードワーカー を使用して、単一ノードでの論理変更レコード（LCR）の送信を有効にします。デフォルトでは、この設定は false です。

true に設定すると、デコードワーカープロセスが開始され、WAL 送信者は生成した LCR を送信します。false に戻した場合、LCR を使用する WAL 送信者は再起動され、WAL を直接使用します。

注釈: PGD グループ内のすべてのノードでこの設定を有効にし、グループで `enable_wal_decoder` オプションを true に設定する必要があります。

---

bdr.receive_lcr

別のノードにサブスクライブする場合、この設定により、ノードはサブスクリプションの論理変更レコード（LCR）の使用を要求できます。デフォルトでは、この設定は false です。この設定がダウンストリームノードで true の場合、ノードはアップストリームノードに送信時に LCR を使用するように要求します。ノードで `bdr.enable_wal_decoder` を true に設定する場合、この設定も `true` に設定します。

注釈: PGD グループ内のすべてのノードでこの設定を有効にし、グループで `enable_wal_decoder` オプションを true に設定する必要があります。 !!!

---

bdr.lcr_cleanup_interval

論理変更レコード（LCR）ファイルのクリーンアップ間隔。

デコードワーカー が有効になっている場合、デコードワーカーは LCR ファイルをバッファーとして保存します。これらのファイルは定期的にクリーニングされ、この設定は 2 つの連続するクリーンアップの間隔を制御します。デフォルトは 3 分です。ゼロに設定すると、クリーンアップが無効になります。

接続設定

以下は、すべてのクロスノード `libpq` 接続に影響を与える一連の接続設定です。デフォルトはかなり控えめな値に設定され、ほとんどの実稼働ニーズをカバーします。すべての変数には `SIGHUP` コンテキストがあります。つまり、リロード時に変更が適用されます。

bdr.global_connection_timeout

接続中に待機する最大時間（秒単位）。10 進整数として記述します。例、10。ゼロ、負、または指定されていない場合は、無限に待機することを意味します。許容される最小タイムアウトは 2 秒であるため、値 1 は 2 と解釈されます。

デフォルトは 15 秒です。

bdr.global_keepalives

TCP キープアライブを使用するかどうかを制御します。デフォルト値は 1、つまりオンです。キープアライブが必要ない場合は、これを 0、つまりオフに変更できます。このパラメーターは、Unix ドメインソケットによる接続では無視されます。

デフォルトは 1（オン）です。

bdr.global_keepalives_idle

TCP がサーバーにキープアライブメッセージを送信するまでの非アクティブの秒数を制御します。ゼロの値は、システムのデフォルトを使用します。このパラメーターは、Unix ドメインソケットによる接続、またはキープアライブが無効になっている場合、無視されます。`TCP_KEEPIDLE` または同等のソケットオプションが利用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

デフォルトは 1 秒です。

bdr.global_keepalives_interval

サーバーによって承認されない TCP キープアライブメッセージを再送信するまでの秒数を制御します。ゼロの値は、システムのデフォルトを使用します。このパラメーターは、Unix ドメインソケットによる接続、またはキープアライブが無効になっている場合、無視されます。TCP_KEEPINTVL または同等のソケットオプションが利用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

デフォルトは 2 秒です。

bdr.global_keepalives_count

サーバーへのクライアントの接続が切断されたと見なされる前に失われる可能性のある TCP キープアライブの数を制御します。ゼロの値は、システムのデフォルトを使用します。このパラメーターは、Unix ドメインソケットによる接続、またはキープアライブが無効になっている場合、無視されます。TCP_KEEPCNT または同等のソケットオプションが利用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

デフォルトは 3 です。

bdr.global_tcp_user_timeout

接続が強制的に閉じられるまでに、送信されたデータが未確認のままになるミリ秒数を制御します。ゼロの値は、システムのデフォルトを使用します。このパラメーターは、Unix ドメインソケットによる接続では無視されます。TCP_USER_TIMEOUT が利用可能なシステムでのみサポートされています。他のシステムでは、効果はありません。

デフォルトは 5 秒です（ミリ秒単位、5000）。

内部設定

bdr.raft_keep_min_entries

ログコンパクションを行うときに Raft ログに保持するエントリーの最小数（デフォルト 100）。値 0 はログ圧縮を無効にします。これは、Postgres サーバーの起動時にのみ設定できます。

> **警告:** ログコンパクションが無効になっている場合、ログのサイズは永久に大きくなります。 !!!

bdr.raft_response_timeout

ネットワーク障害を説明するために、実装された Raft コンセンサスプロトコルは、一定時間後にリクエストをタイムアウトします。このタイムアウトのデフォルトは 30 秒です。

bdr.raft_log_min_apply_duration

ステートマシンを前進させるために、Raft はエントリーをその内部ログに追加します。通常の操作中、追加には数ミリ秒しかかかりません。これにより、その追加アクションの継続時間に上限しきい値が設定され、それを超えると `INFO` メッセージがログに記録されます。これは問題を示している可能性があります。このパラメーターのデフォルト値は 3000 ミリ秒です。

bdr.raft_log_min_message_duration

コンセンサスリクエストをログに記録するタイミング。bdr コンセンサスリクエストのラウンドトリップ時間を測定し、時間がこのパラメーターを超える場合は `INFO` メッセージをログに記録します。このパラメーターのデフォルト値は 5000 ミリ秒です。

bdr.raft_group_max_connections

Postgres サーバーのすべての PGD グループにわたる接続の最大数。これらの接続は、グループのノード間で bdr コンセンサス要求を運びます。このパラメーターのデフォルト値は 100 接続です。Postgres サーバーの起動時にのみ設定できます。

bdr.backwards_compatibility

`bdr.bdr_version_num` 、たとえば `30618` で使用されるのと同じ数値形式で、下位互換性を持つバージョンを指定します。これが一般的に望ましくない効果をもたらす場合でも、以前の PGD バージョンの正確な動作を有効にします。デフォルトは現在の PGD バージョンです。これはリリースごとに異なるため、値が現在のバージョンと異なる場合を除き、構成ファイルで明示的に使用しないことをお勧めします。

bdr.track_replication_estimates

ピアノードの適用率とキャッチアップ間隔の観点からレプリケーションの見積もりを追跡します。CAMO のようなプロトコルは、この情報を使用して、ピアノードの準備ができているかどうかを推定できます。このパラメーターはデフォルトで有効になっています。

bdr.lag_tracker_apply_rate_weight

ローカルノードから WAL を適用する点でピアノードがどれだけ遅れているかを監視し、ラグトラッキングの適用率の移動平均を計算します。このパラメーターは、この移動平均計算における新しい計算値の寄与度を指定します。デフォルト値は 0.1 です。

Node management

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は設定解除され、現在の既知の状態に迅速に設定されることが期待されます。
- CREATED ：bdr.create_node() が実行されましたが、ノードはまだ EDB Postgres 分散クラスターのメンバーではありません。
- JOIN_START ： bdr.join_node_group() は、ローカルノードを既存の EDB Postgres 分散クラスターに参加させ始めます。
- JOINING ：ノード参加が開始され、現在初期同期フェーズにあり、ノード上にスキーマとデータを作成しています。
- CATCHUP ：初期同期フェーズが完了しました。これで、参加は、参加が開始されてから上流のピアノードで実行されたトランザクションを取得して適用する最後のステップになりました。
- STANDBY ：ノード参加は終了しましたが、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を費やしますが、ロジカルスタンバイとして定義されている場合、ノードはこの状態を継続します。
- PROMOTE ：ノードはロジカルスタンバイであり、bdr.promote_node を呼び出してノードの状態を ACTIVE に移動しました。これら 2 つの PROMOTE 状態は、1 つのノードのみが STANDBY よりも高く、ACTIVE よりも低い状態にできるという事実と一貫している必要があります。
- PROMOTING ：ロジカルスタンバイからフル PGD ノードへの昇格が進行中です。
- ACTIVE ：ノードは完全な PGD ノードであり、現在 ACTIVE です。これは、最も一般的なノードのステータスです。
- PART_START ：ノードは ACTIVE または STANDBY で、bdr.part_node を呼び出して、 EDB Postgres 分散クラスターからノードを削除しました。
- PARTING ：ノードは他のノードから切断し、コンセンサスまたはレプリケーションにおいてそれ以上の役割を果たしません。
- PART_CATCHUP ：非分割ノードは、最近分割されたノードから欠落しているデータを同期します。
- PARTED ：ノードの分割操作がすべてのノードで完了しました。

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

ノード管理コマンド

PGD は、既存のノードの物理コピー（pg_basebackup ）を使用して PGD グループにノードを追加し、既存のノードのフィジカルスタンバイを PGD グループ内の新しいノードに変換するためのコマンドラインユーティリティも提供します。

bdr_init_physical

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

データディレクトリを指定する必要があります。このデータディレクトリが空の場合、pg_basebackup -X stream を使用して、高速なブロックレベルのコピー操作を使用してディレクトリを埋めます。

指定されたデータディレクトリが空でない場合、これは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フィジカルスタンバイノードとして既にアクティブな場合は、Postgres を管理する bdr_init_physical を実行する前にスタンバイを停止する必要があります。最初はキャッチアップを待ってから、マスターノードに昇格してから PGD グループに参加します。--standby オプションを使用すると、既存のフィジカルスタンバイがロジカルスタンバイノードになります。これは、指定されたデータディレクトリの開始状態ではなく、新しい PGD ノードの終了状態を参照します。

このコマンドは、データベースからすべての PostgreSQL ネイティブ論理レプリケーションサブスクリプションを削除（または-S オプションが使用されている場合は無効にします)、およびレプリケーションオリジンとスロットを削除します。

概要

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

オプション

一般オプション

- -D, --pgdata=DIRECTORY —新しいノードに使用するデータディレクトリ。空または存在しないディレクトリ、または pg_basebackup -X stream コマンド（必須）を使用して移入されたディレクトリのいずれかです。
- -l, --log-file=FILE — ロギングには FILE を使用します。デフォルトは bdr_init_physical_postgres.log です。
- -n, --node-name=NAME —新しく作成されたノードの名前（必須)。
- --replication-sets=SETS —使用するレプリケーションセット名のコンマ区切りリストの名前。指定しない場合、すべての複製セットが使用されます。
- --standby —完全な送信/受信ノードではなく、ロジカルスタンバイ (受信専用ノード) を作成します。
- --node-group-name —参加するグループ。デフォルトはソースノードと同じグループです。

- -s, --stop —初期化が完了したら、サーバーを停止します。
- -v —ログの冗長性を高めます。
- -L —空/存在しないデータディレクトリ（-D オプション）で使用する場合、選択的な pg_basebackup を実行します。これは EDB Postgres Extended Server のみの機能です。
- -S —論理レプリケーションサブスクリプションを削除する代わりに、無効にします。

接続オプション

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

構成ファイルのオーバーライド

- --hba-conf —新しい pg_hba.conf へのパス。
- --postgresql-conf —新しい postgresql.conf へのパス。
- --postgresql-auto-conf —新しい postgresql.auto.conf へのパス。

注意事項

コマンドで指定されたレプリケーションセット名は、ノードが PGD グループに参加する前にデータディレクトリに存在するデータには影響しません。これは、bdr_init_physical が独自のベースバックアップを作成するか、既存のベースバックアップが新しい PGD ノードに昇格されるかに当てはまります。したがって、--replication-sets オプションは、ノードが PGD ノードグループに参加した後にパブリッシュおよびサブスクライブされたデータにのみ影響します。この動作は、bdr.join_node_group() を使用する場合など、論理結合でレプリケーションセットを使用する方法とは異なります。

オペレーターは、結合の完了後に不要なテーブルを切り捨てることができます。bdr.tables カタログを参照して、レプリケーションセットのメンバーシップを判断し、サブスクライブ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特定します。次の理由から、テーブルは削除せずに切り捨てることを強くお勧めします。

DDL レプリケーションセットは、必ずしも行（DML）レプリケーションセットと同じではないため、他のノード上のテーブルを誤って削除する可能性があります。

- 後でテーブルをレプリケーションセットに追加し、ノードの一部のサブセットで削除した場合は、レプリケーションセットに追加する前に、DDL 競合を作成せずにそれらのノードでのみ再作成する必要があります。

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

Node management interfaces

SQL インターフェースを使用して、ノードを動的に追加および削除できます。

bdr.create_node

この関数はノードを作成します。

概要

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

パラメータ

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

注意事項

この関数は、関連付けられたパブリック接続文字列を使用してローカルノードのレコードを作成します。ローカルレコードは 1 つしか存在できないため、作成されると、関数は再度実行されるとエラーを報告します。

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

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

bdr.drop_node

ノードをドロップします。

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

この関数は、指定されたノードのメタデータをローカルデータベースから削除します。ノードは次のいずれかです。

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

概要

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

パラメータ

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

注意事項

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

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

注釈: PGD4 は、スノーフレーク ID およびタイムハードシーケンスで使用するために、各ノードに一意のシーケンス番号が割り当てられているため、一度に最大 1024 個のノードレコード（ACTIVE と PARTED の両方）を持つことができます。 PARTED ノードは自動的にクリーンアップされません。これが問題になった場合は、この機能を使用してこれらのレコードを削除できます。

---

bdr.create_node_group

この関数は、ローカルノードをグループの唯一のメンバーとして持つ PGD グループを作成します。

概要

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

パラメータ

- node_group_name —新しい PGD グループの名前。ノード名と同様に、有効なグループ名は、小文字、数字、アンダースコアのみで構成されている必要があります。
- parent_group_name —サブグループの親グループの名前。
- join_node_group —このパラメーターは、ノードが作成中のグループに参加するかどうかを決定するのに役立ちます。デフォルト値は true です。これは、ノードが参加したくないシャードグループを作成しているときに使用されます。これは、parent_group_name を指定した場合にのみ false にすることができます。
- node_group_type —有効な値は、NULL 、subscriber-only 、datanode 、read coordinator 、および write coordinator です。 subscriber-only タイプは、クラスター内の完全に参加したノードからのみ変更を受信しますが、レプリケーションの変更を他のノードに送信しないノードのグループを作成するために使用されます。詳細については、購読者専用グループ を参照してください。Datanode は、グループがシャードを表すことを意味しますが、他の値は、グループがそれぞれのコーディネーターを表すことを意味します。subscriber-only を除き、他の値は将来の使用のために予約されています。 NULL は、通常の汎用ノードグループが作成されることを意味します。

注意事項

この関数は、ローカルノードで実行されているローカルコンセンサスワーカーにリクエストを渡します。

この関数はトランザクションではありません。グループの作成はバックグラウンドプロセスであるため、機能が終了すると、変更をロールバックできません。また、変更は現在のトランザクションにすぐに表示されない場合があります。bdr.wait_for_join_completion を呼び出して、それらが完了するまで待つことができます。

グループの作成にはロックはありません。

bdr.alter_node_group_config

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

概要

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

(次のページに続く)

(前のページからの続き)

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

パラメータ

- node_group_name —既存の PGD グループの名前。ローカルノードはグループの一部である必要があります。
- insert_to_update —下位互換性のために予約されています。
- update_to_insert —下位互換性のために予約されています。このオプションは非推奨であり、PGD の将来のバージョンでは無効または削除されます。代わりに bdr.alter_node_set_conflict_resolver を使用してください。
- ignore_redundant_updates —下位互換性のために予約されています。
- check_full_tuple —下位互換性のために予約されています。
- apply_delay —下位互換性のために予約されています。
- check_constraints —レプリケート・データを書き込むときに適用プロセスが制約をチェックするかどうか。このオプションは非推奨であり、PGD の将来のバージョンでは無効または削除されます。
- num_writers —このノードグループをバッキングするサブスクリプションの並列ライターの数。-1 は、デフォルト（GUC bdr.writers_per_subscription で指定）が使用されることを意味します。有効な値は、-1 または正の整数です。
- enable_wal_decoder —デコードワーカープロセスを有効/無効にします。streaming_mode が既に有効になっている場合、デコードワーカープロセスを有効にすることはできません。
- streaming_mode —大規模なトランザクションのストリーミングを有効/無効にします。off に設定すると、ストリーミングは無効になります。他の値に設定されている場合、大規模なトランザクションは進行中にデコードされ、変更はダウンストリームに送信されます。値が file に設定されている場合、ストリーミングトランザクションの着信変更はファイルに保存され、トランザクションがアップストリームでコミットされた後にのみ適用されます。値が writer に設定されている場合、着信変更はライターの 1 つに直接送信されます（使用可能な場合）。パラレル適用が無効になっているか、ライタがストリーミングトランザクションを自由に処理できない場合、変更はファイルに書き込まれ、トランザクションがコミットされた後に適用されます。値が auto に設定されている場合、PGD は、トランザクションプロパティと使用可能なリソースに応じて、file と writer の間でインテリジェントに選択しようとします。WAL デコーダーが既に有効になっている場合、 streaming_mode を有効にすることはできません。

詳細については、 トランザクションストリーミング を参照してください。

- default_commit_scope —デフォルトで使用するコミットスコープ、最初は local コミットスコープ。これは、トップレベルのノードグループにのみ適用されます。同じコミットスコープの異なるオリジングループに個々のルールを使用できます。詳細については、 オリジングループ を参照してください。

注意事項

この関数は、デフォルトを変更するリクエストをグループコンセンサスメカニズムに渡します。行われた変更は、コンセンサスメカニズムを使用してグローバルに複製されます。

この関数はトランザクションではありません。要求はバックグラウンドで処理されるため、関数呼び出しをロールバックすることはできません。また、変更は現在のトランザクションにすぐに表示されない場合があります。

この関数はロックを保持しません。

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

bdr.join_node_group

この関数は、ローカルノードを既存の PGD グループに参加させます。

概要

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

パラメータ

- `join_target_dsn` —ローカルノードを追加する PGD グループ内の既存の（ソース）ノードへの接続文字列を指定します。
- `node_group_name` —PGD グループのオプションの名前。デフォルトは NULL で、ソースノードに存在する情報からグループ名を検出しようとします。
- `pause_in_standby` —オプションで、ロジカルスタンバイノードとしてのみ参加するように参加プロセスに指示します。これは、後でフルメンバーに昇格できます。
- `wait_for_completion` —参加プロセスが完了するまで待ってから、戻ります。デフォルトは `true` です。

- synchronize_structure —結合中に実行する構造（スキーマ）同期の種類を設定します。有効なオプションは、完全なデータベース構造を同期する all 、および構造を同期しない none です。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、これは非同期呼び出しであり、参加プロシージャーが開始されるとすぐに戻ります。結合の進行状況は、ログと bdr.event_summary 情報ビューで確認するか、 bdr.join_node_group() が戻った後に bdr.wait_for_join_completion() 関数を呼び出すことで確認できます。

注意事項

この関数は、 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リクエストを渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

この関数はトランザクションではありません。参加プロセスはバックグラウンドで行われ、ロールバックすることはできません。wait_for_completion が true に設定されている場合、または後で bdr.wait_for_join_completion を呼び出すことによって、変更はローカルトランザクションにのみ可視です。

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

ノード参加は、PGD グループでロックを保持しません。

bdr.switch_node_group

この関数は、ローカルノードを現在のサブグループから同じ既存の PGD ノードグループ内の別のサブグループに切り替えます。

概要

```
bdr.switch_node_group (
    node_group_name text,
    wait_for_completion boolean DEFAULT true
)
```

パラメータ

- node_group_name — PGD グループまたはサブグループの名前。
- wait_for_completion —戻る前に、切り替えプロセスが完了するのを待ちます。デフォルトは true です。

wait_for_completion が false として指定されている場合、これは非同期呼び出しであり、切り替え手続きが開始されるとすぐに戻ります。スイッチの進行状況は、ログと bdr.event_summary 情報ビューで確認するか、 bdr.switch_node_group() が戻った後に bdr.wait_for_join_completion() 関数を呼び出すことで確認できます。

注意事項

この関数は、グループコンセンサスメカニズムにリクエストを渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

この関数はトランザクションではありません。切り替えプロセスはバックグラウンドで行われ、ロールバックすることはできません。wait_for_completion が true に設定されている場合、または後で bdr.wait_for_join_completion を呼び出すことによって、変更はローカルトランザクションにのみ可視です。

ローカルノードは、クラスターを分割することなく、現在のサブグループから同じ PGD ノードグループ内の別のサブグループにメンバーシップを変更します。ノードの種類は、ターゲットサブグループ内の既存のノードの種類と一致する必要があります。

ノード切り替えは、PGD グループでロックを保持しません。

制限：現在、この機能はサブグループとその PGD ノードグループ間の切り替えのみを許可します。サブグループ間の移動を行うには、2 つの個別の呼び出しを行う必要があります。1）サブグループからノードグループに切り替える、2）ノードグループから他のサブグループに切り替える。

bdr.promote_node

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

概要

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

注意事項

この関数は、デフォルトを変更するリクエストをグループコンセンサスメカニズムに渡します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。

この関数はトランザクションではありません。昇格プロセスはバックグラウンドで行われ、ロールバックすることはできません。wait_for_completion が true に設定されている場合、または後で bdr.wait_for_join_completion を呼び出すことによって、変更はローカルトランザクションにのみ可視です。

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

bdr.wait_for_join_completion

この関数は、ローカルノードの join 手続きが完了するのを待ちます。

概要

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

パラメータ

- verbose_progress —オプションで、結合手順中に実行された個々のステップに関する情報を出力します。

注意事項

この関数は、ローカルノードのチェック状態が bdr.create_node_group 、bdr.join_node_group 、または bdr.promote_node で設定されたターゲット状態に到達するまで待機します。

bdr.part_node

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

削除するノードを含む、PGD グループ内のアクティブなノードから関数を呼び出すことができます。ただし、ノードが分割されると、クラスター内の他のノードを分割することはできません。

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

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

概要

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

パラメータ

- node_name —分割する既存のノードの名前。
- wait_for_completion — true の場合、ノードがクラスターから完全に分離されるまで、関数は戻りません。それ以外の場合、関数は別れ手続きを開始し、待たずにすぐに戻ります。ローカルノードで実行する場合、または force を使用する場合は、常に false に設定します。
- force —ローカルノード上のノードを強制的に削除します。コンセンサスに到達できない場合、またはノード分割プロセスがスタックしている場合、これによりノードの状態がローカルに設定されます。

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

注意事項

この関数は、グループコンセンサスメカニズムに要求を渡し、指定されたノードを分割します。行われた変更は、コンセンサスメカニズムによってグローバルに複製されます。別れのプロセスはバックグラウンドで行われ、元に戻すことはできません。wait_for_completion が true に設定されている場合、分割プロセスによる変更はローカルトランザクションでのみ可視です。

force を true に設定すると、コンセンサスが失敗すると、この関数はローカルノードでのみ指定されたノードの状態を設定します。このような場合、関数はトランザクション的であり（関数はノードの状態を変更するため)、ロールバックできます。force が true に設定された状態で既に別れの処理中のノードで関数が呼び出された場合、また、指定されたノードをローカルで parted としてマークして終了します。これは、クラスターでコンセンサスに到達できない場合 (つまり、ノードの大部分がダウンしている場合)、または分割プロセスがスタックしている場合にのみ役立ちます。ただし、受信していた分割ノードが書き込みを行うと、分割プロセスはスタックせずに長い時間がかかる可能性があることを考慮することが重要です。他のノードは、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ります。強制的な別れはこの再同期を完全にスキップし、他のノードを矛盾した状態のままにする可能性があります。

分割プロセスはロックを保持しません。

bdr.alter_node_interface

この関数は、指定されたノードの接続文字列（DSN ）を変更します。

概要

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

パラメータ

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

注意事項

この関数を実行し、ローカルノードでのみ変更を加えます。これは、通常、変更中のノードを含む PGD グループ内のすべてのノードで実行することを意味します。

この関数はトランザクションです。ロールバックでき、変更は現在のトランザクションで表示されます。

この関数は、ローカルノードでロックを保持します。

bdr.alter_subscription_enable

この関数は、指定されたサブスクリプションまたはローカル PGD ノードのすべてのサブスクリプションを有効にします。これは、レジュームサブスクリプションとしても知られています。サブスクリプションが既に有効になっている場合、エラーはスローされません。このオペレーションの影響を受けるサブスクリプションの数を返します。

概要

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

パラメータ

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

注意事項

この関数はレプリケートされず、ローカルノードのサブスクリプション（特定のノードまたはすべてのノード）にのみ影響します。

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

bdr.alter_subscription_disable

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

概要

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

パラメータ

- subscription_name —無効にするサブスクリプションの名前。NULL（デフォルト）の場合、ローカルノード上のすべてのサブスクリプションが無効になります。
- immediate —アクションをすぐに強制し、無効になっているサブスクリプションに関連付けられているすべてのワーカーを停止するために使用されます。このオプションが true の場合、トランザクションブロック内でこの関数を実行することはできません。
- fast —この引数は、 immediate の動作に影響します。 true（デフォルト）に設定されている場合、現在の作業が完了するのを待たずに、無効になっているサブスクリプションに関連付けられているすべてのワーカーを停止します。

注意事項

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

この関数はトランザクションです。ロールバックでき、現在のトランザクションでカタログの変更を確認できます。ただし、サブスクリプションワーカーが停止するタイミングは、immediate の値に依存します。true に設定されている場合、ワーカーは COMMIT を待たずにストップを受け取ります。 fast 引数が true に設定されている場合、ワーカーの中断は現在の作業が完了するまで待機しません。

bdr.alter_node_group_option

PGD ノードグループのルーティング構成を変更する

概要

```
bdr.alter_node_group_option(node_group_name text,
    config_key text,
    config_value text);
```

パラメータ

- node_group_name —変更するグループの名前。
- config_key —変更するノードグループ内のオプションのキー。
- config_value —指定されたキーに設定される新しい値。

bdr.alter_node_option

PGD ノードのルーティング構成を変更する

概要

```
bdr.alter_node_option(node_name text,
    config_key text,
    config_value text);
```

パラメータ

- node_name —変更するノードの名前。
- config_key —変更するノードのオプションのキー。
- config_value —指定されたキーに設定される新しい値。

Routing functions

bdr.create_proxy

プロキシを作成する

概要

```
bdr.create_proxy(proxy_name text, node_group text);
```

パラメーター

- proxy_name —新しいプロキシの名前。
- node_group —プロキシが使用するグループの名前。

bdr.alter_proxy_option

プロキシを変更する

概要

```
bdr.alter_proxy_option(proxy_name text, config_key text, config_value text);
```

パラメーター

- proxy_name —変更するプロキシの名前。
- config_key —変更するプロキシのオプションのキー。
- config_value —指定されたキーに設定される新しい値。

bdr.drop_proxy

プロキシをドロップする

概要

```
bdr.drop_proxy(proxy_name text);
```

パラメーター

- proxy_name —削除するプロキシの名前。

bdr.routing_leadership_transfer

ルーティングリーダーを変更すると、ノードグループのリーダーシップが別のノードに転送されます

概要

```
bdr.routing_leadership_transfer(node_group_name text,
               leader_name text,
               transfer_method text DEFAULT strict,
               transfer_timeout interval DEFAULT 10s);
```

パラメーター

- node_group_name —リーダーシップの異動を要求するグループの名前。
- leader_name —書き込みリーダーになるノードの名前。
- transfer_method —転送のタイプ。「fast」または最大ラグをチェックするデフォルトの「strict」になります。
- transfer_timeout —リーダーシップ転送のタイムアウト、デフォルトは 10 秒です。

Replication set management

bdr.create_replication_set

この関数は、レプリケーションセットを作成します。

このコマンドのレプリケーションは、 DDLDDL 構成の影響を受けます。

概要

```
bdr.create_replication_set(set_name name,
                           replicate_insert boolean DEFAULT true,
                           replicate_update boolean DEFAULT true,
                           replicate_delete boolean DEFAULT true,
                           replicate_truncate boolean DEFAULT true,
                           autoadd_tables boolean DEFAULT false,
                           autoadd_existing boolean DEFAULT true)
```

パラメータ

- set_name —新しい複製セットの名前。PGD グループ全体で一意である必要があります。
- replicate_insert —このレプリケーションセット内のテーブルへの挿入をレプリケートするかどうかを示します。
- replicate_update —このレプリケーションセット内のテーブルの更新をレプリケートするかどうかを示します。
- replicate_delete —このレプリケーションセット内のテーブルからの削除をレプリケートするかどうかを示します。
- replicate_truncate —このレプリケーションセット内のテーブルの切り捨てをレプリケートするかどうかを示します。
- autoadd_tables —新しく作成された（将来）テーブルをこのレプリケーションセットにレプリケートするかどうかを示します
- autoadd_existing —このレプリケーションセットに既存のすべてのユーザーテーブルを追加するかどうかを示します。このパラメーターは、autoadd_tables が true に設定されている場合にのみ効果があります。

注意事項

デフォルトでは、新しいレプリケーションセットは DDL または PGD 管理ファンクションコールをレプリケートしません。レプリケーションセットの DDL レプリケーションを設定する方法については、 *ddl filters* を参照してください。すべての DDL および管理ファンクションコールをレプリケートするデフォルトのグループレプリケーションセット用に、既存の DDL フィルターが設定されています。グループの作成時に作成されますが、PGD グループのデフォルトのレプリケーションセットで DDL または PGD 管理関数呼び出しをレプリケートしたくない場合は、削除できます。

この関数は、 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* の影響を受けることを意味します

構成。

この関数は、 DDL グローバルロックを取ります。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

bdr.alter_replication_set

この機能は、既存のレプリケーションセットのオプションを変更します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.alter_replication_set(set_name name,
                          replicate_insert boolean DEFAULT NULL,
                          replicate_update boolean DEFAULT NULL,
                          replicate_delete boolean DEFAULT NULL,
                          replicate_truncate boolean DEFAULT NULL,
                          autoadd_tables boolean DEFAULT NULL)
```

パラメータ

- set_name —既存の複製セットの名前。
- replicate_insert —このレプリケーションセット内のテーブルへの挿入をレプリケートするかどうかを示します。
- replicate_update —このレプリケーションセット内のテーブルの更新をレプリケートするかどうかを示します。
- replicate_delete —このレプリケーションセット内のテーブルからの削除をレプリケートするかどうかを示します。
- replicate_truncate —このレプリケーションセット内のテーブルの切り捨てをレプリケートするかどうかを示します。
- autoadd_tables —新しく作成された（将来）テーブルをこのレプリケーションセットに追加するかどうかを示します。

NULL（デフォルト）に設定されているオプションはすべて、以前と同じままです。

注意事項

この関数は、 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* の影響を受けることを意味します

構成。

この関数は、 DDL グローバルロックを取ります。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

bdr.drop_replication_set

この関数は、既存のレプリケーションセットを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.drop_replication_set(set_name name)
```

パラメータ

- set_name —既存の複製セットの名前。

注意事項

この関数は、 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* の影響を受けることを意味します

構成。

この関数は、 DDL グローバルロックを取ります。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

警告: 少なくとも別のノードで使用されているレプリケーションセットを削除しないでください。そのノードでレプリケーションが停止するためです。その場合は、影響を受けるノードをそのレプリケーションセットからサブスクライブ解除します。同じ理由から、参加するノードがそのレプリケーションセットのメンバーである場合、参加操作が進行中のレプリケーションセットを削除しないでください。レプリケーションセットのメンバーシップは、結合の開始時にのみチェックされます。これは、レ

プリケーションセットの使用に関する情報は各ノードにローカルであるため、グループに参加する前にノードで構成できるために発生します。

alter_node_replication_sets を使用して、ノードのレプリケーションセットサブスクリプションを管理できます。

bdr.alter_node_replication_sets

この関数は、ノードがパブリッシュおよびサブスクライブするレプリケーションセットを変更します。

概要

```
bdr.alter_node_replication_sets(node_name name,
                                set_names text[])
```

パラメータ

- node_name —変更するノード。現在、ローカルノードである必要があります。
- set_names —指定されたノードにレプリケートするレプリケーションセットの配列。配列が空の場合、グループのデフォルトのレプリケーションセットが使用されます。

注意事項

この関数はローカルノードでのみ実行され、いかなる方法でも複製されません。

この関数はノードが参加する前に実行されるように設計されているため、リストされているレプリケーションセットの存在はチェックされません。エラーを回避するために、レプリケーションセット名を正しく指定するように注意してください。

これにより、PGD グループの一部であるノードだけでなく、まだグループに参加していないノードでも関数を呼び出すことができます。このアプローチでは、結合中に同期されるデータが制限されます。ただし、スキーマは、レプリケーションセットの設定に関係なく、常に完全に同期されます。レプリケーションセットで指定されたテーブルだけでなく、すべてのテーブルがコピーされます。bdr.tables カタログテーブルを参照して、不要なテーブルを削除できます。これらは、PGD の以降のバージョンで自動的に削除される可能性があります。これは、

ddl filters

そうしないと、構成により DDL のレプリケーションが防止されます。

この呼び出しの後にノードがサブスクライブするレプリケーションセットは、他のノードによって公開され、これらのノードからこの関数が実行されるノードに変更を実際に複製します。

Replication set membership

bdr.replication_set_add_table

この関数は、テーブルをレプリケーションセットに追加します。

これにより、テーブルがレプリケーションセットに追加され、この瞬間（またはむしろトランザクションコミット）から変更の複製が開始されます。テーブルがノード上に持つ可能性のある既存のデータは同期されません。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.replication_set_add_table(relation regclass,
                              set_name name DEFAULT NULL,
                              columns text[] DEFAULT NULL,
                              row_filter text DEFAULT NULL)
```

パラメータ

- relation —テーブルの名前または Oid。
- set_name —複製セットの名前。NULL（デフォルト）の場合、PGD グループのデフォルトの複製セットが使用されます。
- columns —将来の使用のために予約されています（現在は何も行わず、NULL である必要があります）。
- row_filter —レプリケートされた行のフィルタリングに使用される SQL 式。この式が定義されていない（つまり、デフォルトの NULL に設定されている）場合、すべての行が送信されます。

row_filter は、 NULL を含むブール結果を生成する式を指定します。True または Unknown と評価される式は、行を複製します。False 値は行を複製しません。式にサブクエリを含めたり、レプリケートされる現在の行の列以外の変数を参照したりすることはできません。システム列を参照することはできません。

row_filter は、ターゲットノードではなく、オリジンノードで実行されます。これにより、この特定のテーブルのレプリケーションに追加の CPU オーバーヘッドがかかりますが、フィルタリングされた行のデータの送信は完全に回避されます。したがって、ネットワーク帯域幅が削減され、ターゲットノードのオーバーヘッドが適用されます。

row_filter は、特定のテーブルの TRUNCATE コマンドを削除しません。レプリケーションセットレベルで TRUNCATE コマンドをフィルタリングできます。

テーブルの一部の列のみをレプリケートできます。

差分のあるノード間のレプリケーション を参照してください。

注意事項

この関数は、 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* の影響を受けることを意味します

構成。

row_filter が NULL でない場合、関数は、レプリケーションセットに追加されるリレーションで DML グローバルロックを取得します。それ以外の場合は、 DDL グローバルロックのみを使用します。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

bdr.replication_set_remove_table

この関数は、レプリケーションセットからテーブルを削除します。

このコマンドのレプリケーションは、DDL フィルタリング設定を含む DDL レプリケーション構成の影響を受けます。

概要

```
bdr.replication_set_remove_table(relation regclass,
                                 set_name name DEFAULT NULL)
```

パラメータ

- relation —テーブルの名前または Oid。
- set_name —複製セットの名前。 NULL（デフォルト）の場合、PGD グループのデフォルトの複製セットが使用されます。

注意事項

この関数は、 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* の影響を受けることを意味します

構成。

この関数は、 DDL グローバルロックを取ります。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

DDL replication filtering

DDL replication filtering も参照

bdr.replication_set_add_ddl_filter

このファンクションは、 DDL フィルタをレプリケーションセットに追加します。

指定されたフィルターと一致する DDL は、そのセットにサブスクライブしているノードにレプリケートされます。これは、PGD 管理機能のレプリケーションにも影響します。

これにより、ノードでの DDL の実行が妨げられることはありません。DDL が他のノードにレプリケートされるかどうかのみが変更されます。2 つのノードの間に、すべてのインデックスコマンドを除外するレプリケーションフィルターがあるとします。Index コマンドは、各ノードに直接接続し、そのノードで目的の DDL を実行することにより、引き続き自由に実行できます。

DDL フィルターは command_tag および role_name を指定して、一部の DDL ステートメントのみのレプリケーションを許可できます。command_tag は、 *EVENT TRIGGERs* で使用されるものと同じです

通常の PostgreSQL コマンドの場合。典型的な例は、ロジカルスタンバイでの追加のインデックスコマンドが他のすべてのノードにレプリケートされないようにするフィルターを作成することです。

修飾機能名と一致するタグ名を使用して、使用される PGD 管理機能をフィルタリングできます。たとえば、bdr.replication_set_add_table は、同じ名前の関数のコマンドタグです。この場合、このタグにより、すべての PGD 機能を bdr.* を使用してフィルタリングできます。

role_name は、コマンドを実行している現在のロールとの照合に使用されます。command_tag と role_name はどちらも正規表現として評価され、大文字と小文字が区別されます。

概要

```
bdr.replication_set_add_ddl_filter(set_name name,
                                   ddl_filter_name text,
                                   command_tag text,
                                   role_name text DEFAULT NULL,
                                   base_relation_name text DEFAULT NULL,
                                   query_match text DEFAULT NULL,
                                   exclusive boolean DEFAULT FALSE)
```

パラメーター

- set_name —複製セットの名前。NULL の場合、PGD グループのデフォルトのレプリケーションセットが使用されます
- ddl_filter_name —DDL フィルターの名前。これは PGD グループ全体で一意である必要があります
- command_tag —コマンドタグを一致させるための正規表現。NULL はすべてに一致することを意味します
- role_name —ロール名に一致する正規表現。NULL は、すべてのロールに一致することを意味します
- base_relation_name —将来の使用のために予約されており、NULL である必要があります
- query_match —クエリに一致する正規表現。NULL はすべてのクエリに一致することを意味します
- exclusive —true の場合、他の一致フィルターは考慮されません（つまり、排他フィルターのみが適用されます）。複数の排他フィルターが一致すると、エラーがスローされます。これは、メインの複製セットでデフォルトの複製を維持しながら、特定のコマンドを特定の複製セットにルーティングする場合に役立ちます。

注意事項

この関数は、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* の影響を受けることを意味します

構成。これは、ddl フィルター構成への変更のレプリケーションが、既存の ddl フィルター構成の影響を受けることも意味します。

この関数は、DDL グローバルロックを取ります。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

定義されたレプリケーションフィルターを表示するには、ビュー bdr.ddl_replication を使用します。

例

PGD 管理機能のみを含めるには、次のようなフィルターを定義します。

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, mygroup_admin, $$bdr\..*$$);
```

インデックス DDL 以外をすべて除外するには：

```
SELECT bdr.replication_set_add_ddl_filter(mygroup, index_filter,
                '^(?!(CREATE INDEX|DROP INDEX|ALTER INDEX)).*');
```

テーブルとインデックスに対するすべてのオペレーションを含め、他のすべてのオペレーションを除外するには、テーブル用とインデックス用の 2 つのフィルターを追加します。これは、複数のフィルターが、許可されたすべての DDL コマンドの和集合を提供することを示しています。

```
SELECT bdr.replication_set_add_ddl_filter(bdrgroup,index_filter, ^((?!INDEX).)*$);
SELECT bdr.replication_set_add_ddl_filter(bdrgroup,table_filter, ^((?!TABLE).)*$);
```

bdr.replication_set_remove_ddl_filter

このファンクションは、レプリケーションセットから DDL フィルタを削除します。

このコマンドのレプリケーションは、 DDL フィルタリング設定自分自身を含む、DDL レプリケーション構成の影響を受けます。

概要

```
bdr.replication_set_remove_ddl_filter(set_name name,
                                      ddl_filter_name text)
```

パラメーター

- set_name —複製セットの名前。 NULL の場合、PGD グループのデフォルトの複製セットが使用されます。
- ddl_filter_name —削除する DDL フィルターの名前。

注意事項

この関数は、 DDL ステートメントと同じレプリケーションメカニズムを使用します。これは、レプリケーションが *ddl filters* 構成の影響を受けることを意味します。これは、DDL フィルター構成への変更のレプリケーションが既存の DDL フィルター構成の影響を受けることも意味します。

この関数は、 DDL グローバルロックを取ります。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

Global sequence management interfaces

PGD は、標準の PostgreSQL シーケンスと PGD グローバルシーケンスの間で変換するためのインターフェイスを提供します。

以下の関数は DDL とみなされるため、DDL レプリケーションとグローバルロックが適用されます。

シーケンス関数

bdr.alter_sequence_set_kind

シーケンスの所有者がシーケンスの種類を設定できるようにします。設定すると、seqkind は bdr.sequences ビューを介してのみ表示されます。他のすべての方法では、シーケンスは通常のシーケンスとして表示されます。

PGD はこの関数を DDL として扱うため、現在アクティブな場合、DDL レプリケーションとグローバルロックが適用されます。

DML および DDL レプリケーション を参照してください。

概要

```
bdr.alter_sequence_set_kind(seqoid regclass, seqkind text)
```

パラメーター

- seqoid —変更するシーケンスの名前または Oid。
- seqkind —標準の PostgreSQL シーケンスの場合は local 、グローバルに一意の PGD シーケンスの場合は snowflakeid または galloc 、またはレガシーのグローバルに一意のシーケンスの場合は timeshard 。

注意事項

シーケンスの種類を galloc に変更すると、そのシーケンスに最初に割り当てられた範囲はシーケンス開始値を開始点として使用します。galloc に変更される前にシーケンスで使用されていた既存の値がある場合は、次のコマンドを使用して、新しく生成された値が既存の値と競合しないように、開始点を移動することをお勧めします。

```
ALTER SEQUENCE seq_name START starting_value RESTART
```

この関数は、 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* の影響を受けることを意味します

構成。

この関数は、グローバルな DDL ロックを取得します。また、シーケンスをローカルにロックします。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

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

bdr.extract_timestamp_from_snowflakeid

この関数は、snowflakeid シーケンスのタイムスタンプコンポーネントを抽出します。戻り値は timestamptz 型です。

概要

sql bdr.extract_timestamp_from_snowflakeid（snowflakeid bigint）

パラメーター

- snowflakeid -スノーフレークイドシーケンスの値。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.extract_nodeid_from_snowflakeid

この関数は、 snowflakeid シーケンスの nodeid コンポーネントを抽出します。

概要

sql bdr.extract_nodeid_from_snowflakeid（snowflakeid bigint）

パラメーター

- snowflakeid -スノーフレークイドシーケンスの値。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.extract_localseqid_from_snowflakeid

この関数は、 snowflakeid シーケンスのローカルシーケンス値コンポーネントを抽出します。

概要

sql bdr.extract_localseqid_from_snowflakeid（snowflakeid bigint）

パラメーター

- snowflakeid -スノーフレークイドシーケンスの値。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.timestamp_to_snowflakeid

この関数は、タイムスタンプ値をダミーのスノーフレーク ID シーケンス値に変換します。

これは、snowflakeid 列の値と特定のタイムスタンプのインデックス付き検索または比較を行う場合に役立ちます。

たとえば、 `snowflakeid` シーケンスを使用している列 `id` を持つテーブル `foo` が与えられた場合、次のように昨日の午前 0 時からの変更数を取得できます。

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

このように定式化されたクエリは、列 `id` でインデックススキャンを使用します。

概要

sql bdr.timestamp_to_snowflakeid（ts timestamptz）

パラメーター

- ts - Snowflakeid シーケンス生成に使用するタイムスタンプ。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.extract_timestamp_from_timeshard

この関数は、 timeshard シーケンスのタイムスタンプコンポーネントを抽出します。戻り値は timestamptz 型です。

概要

```
bdr.extract_timestamp_from_timeshard(timeshard_seq bigint)
```

パラメーター

- timeshard_seq —タイムハードシーケンスの値。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.extract_nodeid_from_timeshard

この関数は、 timeshard シーケンスの nodeid コンポーネントを抽出します。

概要

```
bdr.extract_nodeid_from_timeshard(timeshard_seq bigint)
```

パラメーター

- timeshard_seq —タイムハードシーケンスの値。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.extract_localseqid_from_timeshard

この関数は、 timeshard シーケンスのローカルシーケンス値コンポーネントを抽出します。

概要

```
bdr.extract_localseqid_from_timeshard(timeshard_seq bigint)
```

パラメーター

- timeshard_seq —タイムハードシーケンスの値。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.timestamp_to_timeshard

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

これは、timeshard 列の値と特定のタイムスタンプのインデックス付き検索または比較を行う場合に役立ちます。

たとえば、 timeshard シーケンスを使用している列 id を持つテーブル foo が与えられた場合、次のように昨日の午前 0 時からの変更数を取得できます。

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

このように定式化されたクエリは、列 id でインデックススキャンを使用します。

概要

```
bdr.timestamp_to_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するタイムスタンプ。

注意事項

この関数は、ローカルノードでのみ実行されます。

KSUUID v2 関数

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

KSUUID in the sequences documentation も参照してください。

bdr.gen_ksuuid_v2

この関数は、引数として渡された timestamp の値、または NULL が渡された場合は現在のシステム時刻を使用して、新しい KSUUID v2 値を生成します。システム時刻を使用して KSUUID を自動的に生成する場合は、NULL 引数を渡します。

戻り値は UUID 型です。

概要

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

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.ksuuid_v2_cmp

この関数は、 KSUUID v2 の値を比較します。

最初の値の方が新しい場合は 1、2 番目の値の方が小さい場合は-1、等しい場合は 0 を返します。

概要

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

パラメーター

- UUID —比較する KSUUID v2。

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.extract_timestamp_from_ksuuid_v2

この関数は、 KSUUID v2 のタイムスタンプコンポーネントを抽出します。戻り値は timestamptz 型です。

概要

```
bdr.extract_timestamp_from_ksuuid_v2(uuid)
```

パラメーター

- UUID —タイムスタンプを抽出する KSUUID v2 値。

注意事項

この関数は、ローカルノードでのみ実行されます。

KSUUID v1 関数

KSUUID v1 データ、K-Sortable UUID データ（v1）を操作するための関数。非推奨 - 詳細は *KSUUID in the sequences documentation* を参照してください。

bdr.gen_ksuuid

この関数は、現在のシステム時刻を使用して、新しい KSUUID v1 値を生成します。戻り値は UUID 型です。

概要

```
bdr.gen_ksuuid()
```

注意事項

この関数は、ローカルノードでのみ実行されます。

bdr.uuid_v1_cmp

この関数は、 KSUUID v1 の値を比較します。

最初の値の方が新しい場合は 1、2 番目の値の方が小さい場合は-1、等しい場合は 0 を返します。

概要

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

注意事項

この関数は、ローカルノードでのみ実行されます。

パラメーター

- UUID —比較する KSUUID v1。

bdr.extract_timestamp_from_ksuuid

この関数は、 KSUUID v1 または UUIDv1 値のタイムスタンプコンポーネントを抽出します。戻り値は timestamptz 型です。

概要

```
bdr.extract_timestamp_from_ksuuid(uuid)
```

パラメーター

- UUID —タイムスタンプを抽出する KSUUID v1 値。

注意事項

この関数はローカルノードで実行されます。

Autopartition

自動パーティションを使用すると、テーブルをいくつかのパーティションに分割できます。メインの 水平スケーリングまたはスケールアウト セクションで詳細を読んでください。

bdr.autopartition

bdr.autopartition ファンクションは、テーブルの自動 RANGE パーティション化を構成します。

概要

```
bdr.autopartition(relation regclass,
        partition_increment text,
        partition_initial_lowerbound text DEFAULT NULL,
        partition_autocreate_expression text DEFAULT NULL,
        minimum_advance_partitions integer DEFAULT 2,
        maximum_advance_partitions integer DEFAULT 5,
        data_retention_period interval DEFAULT NULL,
        managed_locally boolean DEFAULT false,
        enabled boolean DEFAULT on);
```

パラメーター

- relation —テーブルの名前または Oid。
- partition_increment —次のパーティション作成の間隔またはインクリメント。
- partition_initial_lowerbound —テーブルにパーティションがない場合、この下限と partition_increment 離れた上限を持つ最初のパーティションが作成されます。
- partition_autocreate_expression —新しいパーティションを作成する時期かどうかを検出するために使用されます。
- minimum_advance_partitions —システムは、常に少なくとも minimum_advance_partitions パーティションを持つように試みます。

- maximum_advance_partitions —事前パーティションの数が minimum_advance_partitions を下回った後に、一度に作成するパーティションの数。
- data_retention_period —定義されている場合、古いパーティションが削除されるまでの間隔。この値は migrate_after_period より大きい必要があります。
- managed_locally —true の場合、パーティションはローカルで管理されます。
- enabled —アクティビティを無効または一時停止し、後で再開または再度有効にできます。

例

デイリーパーティション、1 か月間データを保持：

```
CREATE TABLE measurement (
logdate date not null,
peaktemp int,
unitsales int
) PARTITION BY RANGE (logdate);

bdr.autopartition(measurement, 1 day, data_retention_period := 30 days);
```

残りのパーティションが 2 つだけになったら、5 つのアドバンスパーティションを作成します。各パーティションは 10 億オーダーを保持できます。

```
bdr.autopartition(Orders, 1000000000,
        partition_initial_lowerbound := 0,
        minimum_advance_partitions := 2,
        maximum_advance_partitions := 5
    );
```

bdr.drop_autopartition

bdr.drop_autopartition() を使用して、指定されたリレーションの自動パーティションルールを削除します。リレーションの保留中のすべての作業項目が削除され、新しい作業項目は作成されません。

```
bdr.drop_autopartition(relation regclass);
```

パラメーター

- relation —テーブルの名前または Oid。

bdr.autopartition_wait_for_partitions

パーティションの作成は非同期プロセスです。AutoPartition は、ローカルまたはすべてのノードでパーティションが作成されるのを待機する一連の機能を提供します。

bdr.autopartition_wait_for_partitions() を使用して、ローカルノードでパーティションが作成されるのを待ちます。この関数は、パーティション分割テーブル名とパーティションキー列の値を受け取り、その値を保持するパーティションが作成されるまで待機します。

この関数は、パーティションがローカルに作成されるまでのみ待機します。パーティションがリモートノードにも存在することを保証するものではありません。

すべての PGD ノードでパーティションが作成されるのを待つには、bdr.autopartition_wait_for_partitions_on_all_nodes() 関数を使用します。この関数は、ローカルノードとすべてのリモートノードを内部的にチェックし、パーティションがどこにでも作成されるまで待機します。

概要

```
bdr.autopartition_wait_for_partitions(relation regclass, upperbound text);
```

パラメーター

- relation —テーブルの名前または Oid。
- upperbound —パーティションキー列の値。

bdr.autopartition_wait_for_partitions_on_all_nodes

概要

```
bdr.autopartition_wait_for_partitions_on_all_nodes(relation regclass, upperbound␣
↪text);
```

パラメーター

- relation —テーブルの名前または Oid。
- upperbound —パーティションキー列の値。

bdr.autopartition_find_partition

bdr.autopartition_find_partition() ファンクションを使用して、指定されたパーティションキー値のパーティションを見つけます。その値を保持するパーティションが存在しない場合、関数は NULL を返します。それ以外の場合は、パーティションの Oid が返されます。

概要

```
bdr.autopartition_find_partition(relname regclass, searchkey text);
```

パラメーター

- relname —パーティションテーブルの名前。
- searchkey —検索するパーティションキーの値。

bdr.autopartition_enable

bdr.autopartition_enable を使用して、指定されたテーブルで AutoPartitioning を有効にします。AutoPartitioning がすでに有効になっている場合、アクションは発生しません。指定されたテーブルで AutoPartitioning を無効にするには、

bdr.autopartition_disable を参照してください。

概要

```
bdr.autopartition_enable(relname regclass);
```

パラメーター

- relname — AutoPartitioning を有効にするリレーションの名前。

bdr.autopartition_disable

bdr.autopartition_disable を使用して、指定されたテーブルの AutoPartitioning を無効にします。AutoPartitioning がすでに無効になっている場合、アクションは発生しません。

概要

```
bdr.autopartition_disable(relname regclass);
```

パラメーター

- relname — AutoPartitioning を無効にするリレーションの名前。

内部関数

bdr.autopartition_create_partition

AutoPartition は、内部関数 bdr.autopartition_create_partition を使用して、親テーブルにスタンドアロンの AutoPartition を作成します。

概要

```
bdr.autopartition_create_partition(relname regclass,
                                   partname name,
                                     lowerb text,
                                     upperb text,
                                     nodes oid[]);
```

パラメーター

- relname —アタッチする親テーブルの名前または Oid。
- partname —新しい AutoPartition の名前。
- lowerb —パーティションの下限。
- upperb —パーティションの上限。
- nodes —新しいパーティションが存在するノードのリスト。このパラメーターは PGD の内部であり、将来の使用のために予約されています。

注意事項

これは、AutoPartition がパーティション管理のために使用する内部機能です。関数を直接使用しないことをお勧めします。

bdr.autopartition_drop_partition

AutoPartition は、内部関数 bdr.autopartition_drop_partition を使用して、データ保持ポリシーに従って、不要になったパーティションを削除します。パーティションテーブルが正常に削除された場合、関数は true を返します。

概要

```
bdr.autopartition_drop_partition(relname regclass)
```

パラメーター

- relname —削除するパーティションテーブルの名前。

注意事項

このファンクションは、選択したパーティションテーブルで DROP TABLE を使用する前に、親テーブルに DDL ロックを設定します。この関数は、AutoPartition がパーティション管理のために使用する内部関数です。関数を直接使用しないことをお勧めします。

Stream Triggers Reference

ストリームトリガーの概要は、.. SeeAlso::

ストリームトリガーリファレンス。 !!!

競合トリガー と **トリガーの変換** の両方は、トリガー API が提供する事前定義された変数および PGD が提供する追加情報関数を介して、行とメタデータに関する情報にアクセスできます。

PL / pgSQL では、以下の事前定義された変数と関数を使用できます。

- *Stream triggers row variables*
- *Row Information functions*
- *bdr.trigger_get_row*
- *bdr.trigger_get_committs*
- *bdr.trigger_get_xid*
- *bdr.trigger_get_type*

- *bdr.trigger_get_conflict_type*
- *bdr.trigger_get_origin_node_id*
- *bdr.ri_fkey_on_del_trigger*

ストリームトリガーの作成とドロップは、操作インターフェースを介して管理されます。

- *Stream triggers manipulation interfaces*
- *bdr.create_conflict_trigger*
- *bdr.create_transform_trigger*
- *bdr.drop_trigger*

Stream triggers manipulation interfaces

REPLICA IDENTITY FULL を持つテーブル、または TOAST が適用される列のないテーブルでのみストリームトリガーを作成できます。

bdr.create_conflict_trigger

この関数は、新しい競合トリガーを作成します。

概要

```
bdr.create_conflict_trigger(trigger_name text,
                            events text[],
                            relation regclass,
                            function regprocedure,
                            args text[] DEFAULT {})
```

パラメータ

- trigger_name —新しいトリガーの名前。
- events —このトリガーを起動するイベントの配列。有効な値は’INSERT ‘、’UPDATE ‘、および’DELETE ’です。
- relation —このトリガーを起動するリレーション。
- function —実行する関数。
- args —オプション。トリガー関数が実行時に受け取るパラメーターの配列を指定します（TG_ARGV 変数の内容）。

注意事項

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

このファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

通常の PostgreSQL トリガーと同様に、 bdr.create_conflict_trigger 関数には、関数に対する relation および EXECUTE 特権に対する TRIGGER 特権が必要です。これは、30619 以上の bdr.backwards_compatibility に適用されます。追加のセキュリティルールは、PGD で競合トリガーを含むすべてのトリガーに適用されます。

> *Security and roles* を参照してください。

bdr.create_transform_trigger

この関数は、変換トリガーを作成します。

概要

```
bdr.create_transform_trigger(trigger_name text,
                             events text[],
                             relation regclass,
                             function regprocedure,
                             args text[] DEFAULT {})
```

パラメータ

- trigger_name —新しいトリガーの名前。
- events —このトリガーを起動するイベントの配列。有効な値は’INSERT ‘、’UPDATE ‘、および’DELETE ’です。
- relation —このトリガーを起動するリレーション。
- function —実行する関数。
- args —オプション。トリガー関数が実行時に受け取るパラメーターの配列を指定します（TG_ARGV 変数の内容）。

注意事項

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

このファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

通常の PostgreSQL トリガーと同様に、bdr.create_transform_trigger 関数には、関数に対する relation および EXECUTE 特権に対する TRIGGER 特権が必要です。追加のセキュリティ規則は、PGD で変換トリガーを含むすべてのトリガーに適用されます。

> *Security and roles* を参照してください。

bdr.drop_trigger

この関数は、既存のストリームトリガー（競合と変換の両方）を削除します。

概要

```
bdr.drop_trigger(trigger_name text,
                 relation regclass,
                 ifexists boolean DEFAULT false)
```

パラメータ

- trigger_name —既存のトリガーの名前。
- relation —トリガーが定義されている関係。
- ifexists — true に設定すると、この関数は欠落しているトリガーを無視します。

注意事項

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

このファンクションは、トリガーが作成されているリレーションでグローバル DML ロックを取得します。

この関数はトランザクションです。トランザクションの ROLLBACK を使用して、効果をロールバックできます。変更は現在のトランザクションに表示されます。

relation の所有者のみが bdr.drop_trigger 関数を実行できます。

Stream triggers row functions

bdr.trigger_get_row

この関数は、識別子で指定されたトリガー行の内容を RECORD として返します。この関数は、不適切に呼び出された場合、つまり、操作タイプ（TG_OP）が DELETE の場合に SOURCE_NEW で呼び出された場合、NULL を返します。

概要

```
bdr.trigger_get_row(row_id text)
```

パラメーター

- row_id —行の識別子。トリガーのタイプと操作に応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

bdr.trigger_get_committs

このファンクションは、識別子で指定されたトリガー行のコミットタイムスタンプを返します。行がフリーズしているか利用できないために利用できない場合、NULL を返します。行識別子 SOURCE_OLD に対して常に NULL を返します。

概要

```
bdr.trigger_get_committs(row_id text)
```

パラメーター

- row_id —行の識別子。トリガーのタイプと操作に応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

bdr.trigger_get_xid

このファンクションは、識別子で指定された TARGET 行のローカルトランザクション ID を返します。行がフリーズしているか利用できないために利用できない場合、NULL を返します。 SOURCE_OLD および SOURCE_NEW 行識別子に対して常に NULL を返します。

競合トリガーでのみ使用できます。

概要

```
bdr.trigger_get_xid(row_id text)
```

パラメーター

- row_id —行の識別子。トリガーのタイプと操作に応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

bdr.trigger_get_type

この関数は、現在のトリガータイプを返します。これは、 CONFLICT または TRANSFORM のいずれかです。ストリームトリガーの外部で呼び出された場合は null を返します。

概要

```
bdr.trigger_get_type()
```

bdr.trigger_get_conflict_type

この関数は、競合トリガー内で呼び出された場合、現在の競合タイプを返します。それ以外の場合は、NULL を返します。

競合タイプのリスト を参照

この関数の可能な戻り値について。

概要

```
bdr.trigger_get_conflict_type()
```

bdr.trigger_get_origin_node_id

この関数は、引数として渡されたトリガー row_id のオリジンに対応するノード ID を返します。オリジンが有効でない場合（つまり、行がローカルで発生した場合）、トリガー行引数に応じて、ソースまたはターゲットノードのノード ID を返します。行識別子 SOURCE_OLD に対して常に NULL を返します。この関数を使用して、常に信頼できるソースノードを優先する競合トリガーを定義できます。

概要

```
bdr.trigger_get_origin_node_id(row_id text)
```

パラメーター

- row_id —行の識別子。トリガーのタイプと操作に応じて、SOURCE_NEW、SOURCE_OLD、および TARGET のいずれかになります（個々のトリガータイプの説明を参照）。

bdr.ri_fkey_on_del_trigger

BEFORE トリガーとして呼び出された場合、この関数は FOREIGN KEY 情報を使用して FK の異常を回避します。

概要

```
bdr.ri_fkey_on_del_trigger()
```

Stream triggers row variables

TG_NAME

データ型名。この変数には、実際に起動されたトリガーの名前が含まれています。実際のトリガー名には、トリガーの作成中に指定された名前と比較して、‘_bdrt’または’_bdrc’サフィックス（トリガータイプによって異なります）が付いています。

TG_WHEN

データ型テキスト。この変数は、競合トリガーと変換トリガーの両方に対して BEFORE を示します。 bdr.trigger_get_type() 情報関数を呼び出すことで、ストリームトリガータイプを取得できます。

bdr.trigger_get_type を参照してください。

TG_LEVEL

データ型テキスト：ROW の文字列。

TG_OP

データ型テキスト: トリガーが起動された操作を識別する INSERT 、UPDATE 、または DELETE の文字列。

TG_RELID

データ型 oid：トリガー呼び出しの原因となったテーブルのオブジェクト ID。

TG_TABLE_NAME

データ型名：トリガー呼び出しの原因となったテーブルの名前。

TG_TABLE_SCHEMA

データ型名: トリガーの呼び出しの原因となったテーブルのスキーマの名前。パーティションテーブルの場合、これはルートテーブルの名前です。

TG_NARGS

データ型整数： bdr.create_conflict_trigger() または bdr.create_transform_trigger() ステートメントでトリガー関数に与えられた引数の数。

TG_ARGV[]

テキストのデータ型配列: bdr.create_conflict_trigger() または bdr.create_transform_trigger() ステートメントからの引数。インデックスは 0 からカウントされます。無効なインデックス（0 未満または TG_NARGS 以上）は、 NULL 値になります。

Internal catalogs and views

ここでは、カタログとビューをアルファベット順に提示しています。

bdr.ddl_epoch

DDL エポックごとの状態を保持する内部カタログテーブル。

bdr.ddl_epoch 列

Name	Type	Description
ddl_epoch	int8	単調増加エポック番号
origin_node_id	oid	このエポックの作成を要求したノードの内部ノード ID
epoch_consume_timeout	timestamptz	このエポックのタイムアウト
epoch_consumed	boolean	ローカルノードがエポックを完全に処理するとすぐに true に切り替わります
epoch_consumed_lsn	boolean	ローカルノードがエポックを処理した LSN

bdr.event_history

特定の PGD ノードのクラスターメンバーシップイベントを追跡する内部カタログテーブル。具体的には、以下を追跡します。

*ノードが参加します（クラスターに）

- Raft の状態が変化する（つまり、ノードがコンセンサスプロトコルでの役割を変更するたび-リーダー、フォロワー、または候補者からリーダーへ）- *Raft* **コンセンサスのモニタリング** を参照

*ワーカーがエラーアウトしたとき（ *bdr.workers* を参照）

および *PGD* **レプリケーションワーカーのモニタリング** ）

bdr.event_history 列

Name	Type	Description
event_node_id	oid	イベントが参照するノードの ID
event_type	int	イベントの種類（ノード、raft、またはワーカー関連のイベント）
event_sub_type	int	イベントのサブタイプ、つまり、参加、状態変更、またはエラーの場合
event_source	text	イベントがソースされたワーカープロセスの名前
event_time	timestamptz	イベントが発生したタイムスタンプ
event_text	text	イベントのテキスト表現（ワーカーのエラーなど）
event_detail	text	イベントのより詳細な説明（今のところ、ワーカーエラーにのみ関連）

bdr.event_summary

より人間に優しい形式で情報を表示する bdr.event_history カタログのビュー。具体的には、イベントタイプとサブタイプを整数ではなくテキスト表現として表示します。

bdr.node_config

ノードごとの構成オプションを備えた内部カタログテーブル。

bdr.node_config 列

Name	Type	Description
node_id	oid	ノード ID
node_route_priority	int	このノードに割り当てられた優先度
node_route_fence	boolean	このノードをフェンスするように切り替えます
node_route_writes	boolean	書き込みを許可するように切り替えます
node_route_reads	boolean	読み取りを許可するように切り替えます
node_route_dsn	text	このノードのインターフェース

bdr.node_group_config

ノードグループごとの構成オプションを備えた内部カタログテーブル。

bdr.node_group_config 列

Name	Type	Description
node_group_id	oid	ノードグループ ID
route_writer_max_lag	bigint	受け入れられる最大書き込み遅延
route_reader_max_lag	bigint	受け入れられる最大読み取り遅延
route_writer_wait_flush	boolean	フラッシュを待つ必要がある場合は切り替えます

bdr.node_group_routing_config_summary

ノードグループごとのルーティング構成オプション。

bdr.node_group_routing_info

プロキシの現在のルーティング情報を保持する内部カタログテーブル。

bdr.node_group_routing_info 列

Name	Type	Description
node_group_id	oid	このプロキシが使用するノードグループ ID
write_node_id	oid	現在の書き込みノード
prev_write_node_id	oid	前の書き込みノード
read_node_ids	oid[]	読み込んだノード ID のリスト

bdr.node_group_routing_summary

よりフレンドリーな方法で情報を示す `bdr.node_group_routing_info` カタログのビュー

bdr.node_routing_config_summary

ノードごとのルーティング構成オプション。

bdr.proxy_config

プロキシ固有の構成を保持する内部カタログテーブル。

bdr.proxy_config 列

Name	Type	Description
proxy_name	name	プロキシの名前
node_group_id	oid	このプロキシが使用するノードグループ ID
listen_port	int	プロキシが使用するポート
max_client_conn	int	プロキシが受け入れる最大クライアント接続数
max_server_conn	int	サーバーが受け入れる最大接続数

次のページに続く

表 60 – 前のページからの続き

Name	Type	Description
server_conn_timeout	interval	サーバー接続のタイムアウト
server_conn_keepalive	interval	サーバー接続キープアライブの間隔
fallback_group_timeout	interval	フォールバックに必要なタイムアウト
fallback_group_ids	oid[]	フォールバックに使用するグループ ID のリスト
listen_addrs	text[]	リッスンアドレス

bdr.proxy_config_summary

プロキシインスタンスごとの特定の構成オプション。

bdr.sequence_kind

各非ローカルシーケンスのタイプを格納する内部状態テーブル。診断目的には、ビュー `bdr.sequences` をお勧めします。

bdr.sequence_kind 列

Name	Type	Description
seqid	oid	シーケンスの内部 OID
seqkind	char	内部シーケンスの種類（ 'l'=ローカル、't'=タイムハード、's'=スノーフレークイド、'g'=galloc）

Internal system functions

以下にリストされているのは、内部システム関数です。多くはさまざまなビューの作成に使用され、関数を直接使用せずに、関数が提供するビューを使用することをお勧めします。

一般的な内部機能

bdr.decode_message_response_payload

コンセンサスペイロードをより人間が読める出力にデコードする PGD メッセージペイロード関数。主に `bdr.global_consensus_journal_details` デバッグビューによって使用されます。

bdr.decode_message_payload

コンセンサスペイロードをより人間が読める出力にデコードする PGD メッセージペイロード関数。主に `bdr.global_consensus_journal_details` デバッグビューによって使用されます。

bdr.get_global_locks

この機能は、ローカルノードで保持されているグローバルロックに関する情報を表示します。

`bdr.global_locks` ビューを実装して、ロックのより詳細な概要を提供するために使用されます。

bdr.get_slot_flush_timestamp

指定されたレプリケーションスロットの最後のフラッシュ位置確認のタイムスタンプを取得します。

`bdr.node_slots` ビューを実装するために内部的に使用されます。

bdr.internal_alter_sequence_set_kind

さまざまなファンクションコールのレプリケーションのために内部的に使用される PGD 内部ファンクション。

PGD の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためにのみ存在します。

bdr.internal_replication_set_add_table

さまざまなファンクションコールのレプリケーションのために内部的に使用される PGD 内部ファンクション。

PGD の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためにのみ存在します。

bdr.internal_replication_set_remove_table

さまざまなファンクションコールのレプリケーションのために内部的に使用される PGD 内部ファンクション。

PGD の現在のバージョンでは使用されなくなりました。ローリングアップグレード中の下位互換性のためにのみ存在します。

bdr.internal_submit_join_request

新しいノードに参加するためのコンセンサスリクエストを送信します。

PGD グループの再構成内部メカニズムによって必要とされます。

bdr.isolation_test_session_is_blocked

グローバルロック上のブロックのチェックが追加されて、元の `pg_isolation_test_session_is_blocked` を拡張（および実際に呼び出し）するヘルパー関数。

分離/並行性テストに使用されます。

bdr.local_node_info

この機能は、PGD グループ再構成の内部メカニズムが必要とするローカルノードの情報を表示します。

ビュー `bdr.local_node_summary` は、ユーザーの消費に役立つ同様の情報を提供します。

bdr.msgb_connect

コンセンサスプロトコルが使用する、別のノードのコネクションプーラーに接続するための機能。

bdr.msgb_deliver_message

コンセンサスプロトコルによって使用される、別のノードの接続プーラーにメッセージを送信するための関数。

bdr.peer_state_name

この関数は、ノード状態（`node_state` ）をテキスト表現に変換し、主に `bdr.node_summary` ビューを実装するために使用されます。

bdr.request_replay_progress_update

「リプレイ進行状況更新」Raft メッセージの即時書き込みを要求します。これは主にテスト目的で使用されますが、コンセンサスメカニズムが機能しているかどうかのテストにも使用できます。

bdr.seq_nextval

シーケンスインクリメントの内部実装。

PGD グローバルシーケンス と対話するクエリでは、標準の `nextval`

の代わりにこの関数を使用します。

注意事項

以下は、内部 PGD シーケンス操作関数でもあります。 `bdr.seq_currval` と `bdr.sql_lastval` は自動的に使用されます。

bdr.show_subscription_status

サブスクリプションステータスに関する情報を取得し、主に `bdr.subscription_summary` ビューを実装するために使用されます。

bdr.get_node_conflict_resolvers

ローカルノード上のすべての競合リゾルバーのテキスト文字列を表示します。

bdr.reset_subscription_stats

`bdr.stat_subscription` で表示される、サブスクリプションによって作成された統計をリセットした後、ブール値の結果を返します。

bdr.reset_relation_stats

`bdr.stat_relation` から見た、リレーション統計をリセットした後、ブール値の結果を返します。

bdr.pg_xact_origin

指定されたトランザクションのオリジン ID を返します。

概要

```
bdr.pg_xact_origin(xmin xid)
```

パラメーター

- xid —オリジンが返されるトランザクション ID、

bdr.difference_fix_origin_create

引数として渡された指定された名前で bdr_ プレフィックスを追加して、レプリケーションオリジンを作成します。オリジンの内部 ID を返します。これは pg_replication_origin_create() と同じ機能を実行しますが、これには postgres のスーパーユーザー権限ではなく bdr_superuser が必要です。

bdr.difference_fix_session_setup

現在のセッションを現在のオリジンからのリプレイとしてマークします。この関数は、セッションに対して暗黙的に事前に作成された bdr_local_only_origin ローカルレプリケーション起点を使用します。これにより、リプレイの進行状況を報告でき、void を返します。この関数は、この関数が postgres スーパーユーザー権限ではなく bdr_superuser を必要とすることを除いて、pg_replication_origin_session_setup() と同じ機能を実行します。関数の以前の形式である bdr.difference_fix_session_setup(text) は非推奨であり、今後のリリースで削除されます。

概要

```
bdr.difference_fix_session_setup()
```

bdr.difference_fix_session_reset

現在のセッションをどのオリジンからも再生していないものとしてマークし、基本的に bdr.difference_fix_session_setup() の効果をリセットします。void を返します。この関数は、この関数が postgres のスーパーユーザー権限ではなく bdr_superuser を必要とすることを除いて、pg_replication_origin_session_reset() と同じ機能を持っています。

概要

```
bdr.difference_fix_session_reset()
```

bdr.difference_fix_xact_set_avoid_conflict

LSN ’0/0’ およびタイムスタンプ ’2000-01-01’ でコミットされたトランザクションをリプレイするものとして、現在のトランザクションをマークします。この関数は、postgres のスーパーユーザー権限ではなく bdr_superuser が必要であることを除いて、`pg_replication_origin_xact_setup('0/0', '2000-01-01')` と同じ機能を持っています。

概要

```
bdr.difference_fix_xact_set_avoid_conflict()
```

bdr.resynchronous_table_from_node

リモートノードからリレーションを再同期します。

概要

```
bdr.resynchronize_table_from_node(node_name name, relation regclass)
```

パラメーター

- node_name —リレーションデータをコピーまたは再同期するノード。
- relation —リモートノードからコピーするリレーション。

注意事項

このファンクションは、リレーションのグローバル DML ロックを取得し、リレーションをローカルで切り捨て、リモートノードからリレーションにデータをコピーします。

リレーションは、同じ名前と定義で両方のノードに存在する必要があります。

以下がサポートされています。

- 同じパーティション定義を持つパーティションテーブルの再同期
- パーティションテーブルから非パーティションテーブルへ、およびその逆の再同期
- 外部キー制約を一時的に削除して再作成することによる参照テーブルの再同期

参照先テーブルで関数を実行した後、参照先列のデータが参照元列の値と一致しなくなった場合、エラーがスローされます。参照テーブルデータを再同期した後、関数を再実行します。

さらに、リモートノードからデータをコピーした後にローカルで生成された列の値を計算することにより、生成された列を含むテーブルの再同期をサポートします。

現在、row_filters はこの関数によって無視されます。

bdr.resynchronize_table_from_node ファンクションは、所有者が bdr_superuser 権限を持っている場合、テーブルの所有者のみが実行できます。

bdr.consensus_kv_store

一貫性のある KV ストアに値を保存します。

値の有効期限のタイムスタンプを返します。これは ttl に依存します。ttl が NULL の場合、これは infinity を返します。値が削除された場合、-infinity を返します。

概要

```
bdr.consensus_kv_store(key text, value jsonb,
        prev_value jsonb DEFAULT NULL, ttl int DEFAULT NULL)
```

パラメーター

- key —挿入、更新、または削除するための任意の一意のキー。
- value —格納する JSON 値。NULL の場合、既存のレコードは削除されます。
- prev_value —設定されている場合、書き込み操作は現在の値が prev_value に等しい場合にのみ実行されます。
- ttl —新しい値の存続時間（ミリ秒）。

注意事項

これは内部関数であり、主に HARP によって使用されます。

> **警告:** ユーザアプリケーションでこの関数を使用しないでください。

bdr.consensus_kv_fetch

一貫性のある KV ストアから JSON 形式で値を取得します。

概要

```
bdr.consensus_kv_fetch(IN key text) RETURNS jsonb
```

パラメーター

- key —取得する任意のキー。

注意事項

これは内部関数であり、主に HARP によって使用されます。

> **警告:** ユーザアプリケーションでこの関数を使用しないでください。

タスクマネージャー機能

bdr.taskmgr_set_leader

指定された node がタスクマネージャーリーダーノードになるように要求します。リーダーノードは、新しいタスクの作成を担当します（現在、自動パーティションのみがこの機能を利用します)。監視ノード、ロジカルスタンバイ、またはサブスクライバー専用ノードは、リーダーになることはできません。このような要求は ERROR で失敗します。

概要

```
bdr.taskmgr_set_leader(node name, wait_for_completion boolean DEFAULT true);
```

bdr.taskmgr_get_last_completed_workitem

クラスター内のすべてのノードで正常に完了した最後の作業項目の id を返します。

概要

```
bdr.taskmgr_get_last_completed_workitem();
```

bdr.taskmgr_work_queue_check_status

この機能を使用すると、タスクを生成して終了するジョブを実行しているバックグラウンドワーカーのステータスを確認できます。

ステータスは、次のビューで見ることができます。bdr.taskmgr_work_queue_local_status bdr.taskmgr_work_queue_global_status

概要

```
bdr.taskmgr_work_queue_check_status(workid bigint
                            local boolean DEFAULT false);
```

パラメーター

- workid —タスクのキー。
- local —ローカルステータスのみを確認します。

注意事項

Taskmgr ワーカーは、 bdr.autopartition 関数が初めて呼び出される前でも、常にバックグラウンドで実行されています。無効な workid が使用された場合、関数は unknown を返します。 In-progress は典型的なステータスです。

bdr.node_catchup_state_name

キャッチアップ州コードを名前に変換する

概要

```
bdr.node_catchup_state_name(catchup_state oid);
```

パラメーター

- catchup_state —キャッチアップ状態の Oid コード。

bdr.bdr_track_commit_decision

トランザクションのコミットステータスを共有メモリハッシュテーブルに保存します。これは、アップグレードスクリプトによって使用され、bdr.node_pre_commit カタログに保存されているコミットの決定を共有メモリハッシュテーブルに転送します。これは WAL にも記録されるため、WAL からリロードできます。

概要

```
bdr.bdr_track_commit_decision(OID, xid, xid, "char", timestamptz, boolean);
```

bdr.bdr_get_commit_decisions

共有メモリの状態を検査するコンビニエンスルーチン

概要

```
bdr.bdr_get_commit_decisions(dbid OID,
            origin_node_id OID,
            origin_xid xid,
            local_xid xid,
            decision "char",
            decision_ts timestamptz,
            is_camo boolean)
```

bdr.show_workers

bdr ワーカーに関する情報。

概要

```
bdr.show_workers(
    worker_pid int,
    worker_role int,
    worker_role_name text,
    worker_subid oid
```

bdr.show_writers

bdr.writers ビューで使用される関数。

bdr.node_kind_name

ノードの種類のわかりやすい名前を返します（data|standby|witness|subscriber-only）