

2016年度WG3活動報告書
レプリケーション調査編

目次

目次	2
1. ライセンス	5
2. はじめに	6
2.1. PostgreSQLエンタープライズコンソーシアムとWG3について	6
2.2. 本資料の概要と目的	6
2.3. 本資料の構成	6
2.4. 想定読者	6
2.5. 参考文献	7
3. PostgreSQLにおけるレプリケーション	8
3.1. レプリケーションの目的	8
3.1.1. 可用性の向上	8
3.1.2. 性能向上	8
3.1.2.1. 参照の同期	8
3.1.2.2. 更新による一貫性の維持	8
3.2. 代表的なレプリケーションの手法	8
3.2.1. シングルマスタ	8
3.2.1.1. ストレージレプリケーション	9
3.2.1.2. トリガベースレプリケーション	9
3.2.1.3. クエリベースレプリケーション	9
3.2.1.4. ストリーミングレプリケーション	9
3.2.2. マルチマスタ	10
3.2.2.1. Bi-Directional Replication	10
3.2.2.2. Bucardo	10
3.3. 参考文献	10
4. ストリーミングレプリケーション	11
4.1. はじめに	11
4.1.1. PostgreSQLのストリーミングレプリケーションの特徴	11
4.1.2. 想定構成	13
4.2. SR環境構築時の設定項目、推奨値	13
4.2.1. SRにおける目的別の設定	13
4.2.2. SR環境の設定手順	13
4.2.3. アーカイブモード運用	17
4.2.4. レプリケーションスロット	18
4.2.5. 遅延レプリケーション	20
4.2.6. WAL圧縮	22
4.3. SR環境の監視	22
4.3.1. レプリケーション操作ログの監視	22
4.3.1.1. 調査の目的	22
4.3.1.2. 調査結果	22
4.3.1.3. 調査結果	23
4.3.2. 同期状況の監視	23
4.3.2.1. 同期遅延監視	23
4.3.3. 更新、読み取りの監視	24
4.3.3.1. 更新・読取監視	24
4.3.4. スプリットブレインの監視	25
4.3.4.1. スプリットブレインの定義と調査の目的	25
4.3.4.2. 監視方法	25
4.3.4.3. サーバログの監視	26
4.3.4.4. pg_controldataコマンドによる監視	26
4.3.4.5. pg_control_recoveryコマンドによる監視	27
4.3.4.6. pg_is_in_recovery関数による監視	27
4.4. SR環境の障害時運用	27
4.4.1. 本文書における用語の定義	27
4.4.1.1. フェイルオーバー	27
4.4.1.2. フェイルバック	27
4.4.1.3. スイッチオーバー	28
4.4.1.4. スイッチバック	28

4.4.2. 障害時運用手順	29
4.4.2.1. フェイルオーバー	30
4.4.2.2. フェイルバック	31
4.4.2.3. スイッチオーバー	35
4.4.2.4. スイッチバック	36
4.4.2.5. スレーブ障害による対処	39
4.5. まとめ	40
4.5.1. ストリーミングレプリケーション機能拡張の歴史	40
4.5.2. PGConsの今後の取り組み	42
5. Bi-Directional Replication (BDR)	43
5.1. はじめに	43
5.1.1. BDRの特徴	43
5.1.1.1. BDRとは	43
5.1.1.2. ユースケース	43
5.1.1.3. メカニズム	43
5.1.1.4. 整合性	44
5.1.1.5. シーケンス	44
5.1.1.6. 比較表	45
5.1.1.7. サポート	45
5.2. 調査、検証の目的	45
5.3. 調査、検証の前提	45
5.4. BDR環境構築時の設定と設定手順	46
5.4.1. BDR環境の設定手順	46
5.4.1.1. 概要	46
5.4.1.2. 対象試験サーバ	46
5.4.1.3. 事前確認	46
5.5. BDR動作検証	52
5.5.1. ノードの追加・切り離し	52
5.5.1.1. 検証の目的	52
5.5.1.2. 検証内容	52
5.5.1.3. 検証環境	52
5.5.1.4. 検証手順	52
5.5.1.4.1. ノード切り離し	52
5.5.1.4.2. ノード追加	54
5.5.1.5. 検証結果	58
5.5.2. グローバルシーケンス設定	58
5.5.2.1. 検証の目的	58
5.5.2.2. 検証環境	58
5.5.2.3. 検証手順	58
5.5.2.4. 検証結果	60
5.5.3. 選択的レプリケーション	61
5.5.3.1. 検証目的	61
5.5.3.2. 検証内容	61
5.5.3.3. 検証環境	61
5.5.3.4. 検証手順	61
5.5.3.5. 検証結果	65
5.5.4. 更新処理競合時の動作	65
5.5.4.1. 検証目的	65
5.5.4.2. 検証内容	65
5.5.4.3. 検証環境	66
5.5.4.4. 検証手順	66
5.5.4.4.1. PRIMARY KEYまたはUNIQUE制約	67
5.5.4.4.2. 外部キー制約	75
5.5.4.4.3. 排他制約	76
5.5.4.4.4. グローバルなデータ	79
5.5.4.4.5. ロックの競合とデッドロックの中断	80
5.5.4.4.6. その他	81
5.5.4.5. 検証結果	83
5.5.5. ノード障害と復旧	83

5.5.5.1. 検証の目的	83
5.5.5.2. 検証内容	83
5.5.5.3. 検証環境	83
5.5.5.4. 検証手順	83
5.5.5.4.1. ノード障害	83
5.5.5.4.2. ノード復旧	86
5.5.5.5. 検証結果	89
5.5.5.6. 備考	89
5.6. BDR性能検証	89
5.6.1. 更新性能検証結果	89
5.6.1.1. 検証目的	89
5.6.1.1.1. 検証内容	89
5.6.1.1.2. 検証環境	90
5.6.1.1.3. 検証手順	90
5.6.1.1.4. 検証結果	92
5.7. まとめ	93
5.7.1. BDR検証まとめ	93
5.8. 参考文献	93
5.8.1. 関数一覧	93
5.8.2. システムカタログ一覧	94
6. まとめ	95
7. 著者	96

1. ライセンス

本作品はCC-BYライセンスによって許諾されています。ライセンスの内容を知りたい方は [こちら](#) でご確認ください。文書の内容、表記に関する誤り、ご要望、感想等につきましては、[PGEConsのサイト](#) を通じてお寄せいただきますようお願いいたします。

- Eclipseは、Eclipse Foundation Incの米国、およびその他の国における商標もしくは登録商標です。
- IBMおよびDB2は、世界の多くの国で登録されたInternational Business Machines Corporationの商標です。
- Intel、インテルおよびXeonは、米国およびその他の国における Intel Corporation の商標です。
- Javaは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。
- Linux は、Linus Torvalds 氏の日本およびその他の国における登録商標または商標です。
- Red HatおよびShadowman logoは、米国およびその他の国におけるRed Hat,Inc.の商標または登録商標です。
- Microsoft、Windows Server、SQL Server、米国 Microsoft Corporationの米国及びその他の国における登録商標または商標です。
- MySQLは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。
- Oracleは、Oracle Corporation 及びその子会社、関連会社の米国及びその他の国における登録商標です。文中の社名、商品名等は各社の商標または登録商標である場合があります。
- PostgreSQLは、PostgreSQL Community Association of Canadaのカナダにおける登録商標およびその他の国における商標です。
- Windows は米国 Microsoft Corporation の米国およびその他の国における登録商標です。
- TPC, TPC Benchmark, TPC-C, TPC-E, tpmC, TPC-H, QphHは米国Transaction Processing Performance Councilの商標です
- その他、本資料に記載されている社名及び商品名はそれぞれ各社が 商標または登録商標として使用している場合があります。

2. はじめに

2.1. PostgreSQLエンタープライズコンソーシアムとWG3について

PostgreSQLエンタープライズコンソーシアム(略称 PGECons)は、PostgreSQL本体および各種ツールの情報収集と提供、整備などの活動を通じて、ミッションクリティカル性の高いエンタープライズ領域へのPostgreSQLの普及を推進することを目的として設立された団体です。

PGECons 技術部会ではPostgreSQLの普及に資する課題を活動テーマとし、3つのワーキンググループで具体的な活動を行っています。

- WG1(新技術検証ワーキンググループ)
- WG2(移行ワーキンググループ)
- WG3(課題検討ワーキンググループ)

これら3つのワーキンググループのうち、WG1、WG3については2015年度まではそれぞれ、「性能ワーキンググループ」、「設計運用ワーキンググループ」という名称で活動してきました。2016年度は、従来の活動領域を広げる意図のもとでそれらを再定義し、上記のような名称に改めました。

これに伴い、WG3ではPostgreSQLの設計運用を中心としたさまざまな課題の解決のための調査検証を行い、PostgreSQLが広く活用される事を推進していくこととしました。

2.2. 本資料の概要と目的

本資料はWG3の2016年度の活動としてPostgreSQLにおけるレプリケーション機能について、調査検討した結果をまとめたものです。

これまでも、WG3では2度にわたってレプリケーションについて取り上げています。

WG3が発足した2013年には、PostgreSQLをエンタープライズ領域で活用するにあたって、業務要件とコストとのバランスを考えて可能な限り業務を継続できるように、システムの運用や保守サービスをPostgreSQLで実現する手法についてまとめました [\[PGECons WG3 2013\]](#)。報告書では、可用性を向上させるための手段としてレプリケーションを取り上げています。今年度の報告書では取り上げなかった、ストレージレプリケーション(DRBD)、トリガベースレプリケーション(Slony-I)についても紹介しています。

2014年度の報告書では、可用性のうち「災害対策」に焦点を当て、ITサービス継続を可能とするPostgreSQLの構成について調査検討しています [\[PGECons WG3 2014\]](#)。ストリーミングレプリケーションを含む代表的なPostgreSQLのシステム構成を挙げて、データベースのデータ保全性、サービス継続性の観点から各構成の得失を示しました。

この間、PostgreSQLや周辺のOSSツールの開発が進み、ストリーミングレプリケーションの機能改善やPostgreSQLをベースとして双方向レプリケーションがリリースされています。

これらの開発動向と過去の検討結果とを踏まえて、2016年度は機能拡充が進むストリーミングレプリケーションの運用ノウハウの獲得と、2nd Quadrant 社が開発・リリースしているPostgreSQLベースの双方向レプリケーションであるPostgres-BRD [\[PostgresBRD\]](#) の基本機能の調査を実施しました。

2.3. 本資料の構成

- はじめに
- PostgreSQLにおけるレプリケーション
 - レプリケーションの目的
 - 代表的なレプリケーションの手法
- ストリーミングレプリケーション
 - はじめに
 - SR環境構築時の設定項目、推奨値
 - SR環境の監視
 - SR環境の障害時運用
 - スレーブのアーカイブ保存
 - まとめ
- Bi-Directional Replication (BDR)
 - はじめに
 - BDR環境構築時の設定項目、推奨値
 - BDR動作検証
 - BDR性能検証
 - まとめ
- おわりに

2.4. 想定読者

本資料の読者は以下のような知識を有していることを想定しています。

- DBMSを操作してデータベースの構築、保守、運用を行うDBAの知識
- PostgreSQLを利用する上での基礎的な知識

2.5. 参考文献

- [[PGECons_WG3_2013](#)] PostgreSQL Enterprise Consortium. 2013年度WG3活動成果報告書. 2014. <https://www.pgecons.org/wp-content/uploads/PGECons/2013/WG3/pgecons-wg3-2013-report.pdf>
- [[PGECons_WG3_2014](#)] PostgreSQL Enterprise Consortium. 2014年度WG3活動成果報告書 (可用性編). 2014. https://www.pgecons.org/wp-content/uploads/PGECons/2014/WG3/PGECons_2014_WG3_Availability.pdf
- [[PostgresBDR](#)] 2ndQuadrant. Postgres-BDR. Version Postgres-BDR94 1.0.2, November 14, 2016. <https://2ndquadrant.com/en-us/resources/bdr/>

3. PostgreSQLにおけるレプリケーション

データベースにおいて、レプリケーションとは複数のデータベースサーバの間で、何らかの一貫性を保ちながら、その内容を複製する手法を指します。PostgreSQLにおいては、9.0以降の各バージョンの基本機能として、レプリケーションが実現されています。また、PostgreSQLに付加するツールによってもレプリケーションが実現されています。この章では、各種のレプリケーション機能を目的に応じて使い分けるために、機能と特性について簡単に紹介します。

3.1. レプリケーションの目的

レプリケーションによって複数のデータベースサーバの複製を作ることによって、シングルサーバでは対応が難しい要件にも対応できるようになります。それらは、データの冗長性(複製があること)と複数サーバによる処理の分散の結果です。

3.1.1. 可用性の向上

可用性はITシステムの非機能要件の一つで、システムを継続的に利用可能とすることです [\[IPA\]](#)。可用性は「継続性」「耐障害性」「災害対策」「回復性」という4つの要素から構成されますが、レプリケーションによってデータベースサーバを冗長化することで、1つのサーバで故障や災害が生じたときにもレプリカが格納されている残りのサーバでサービスを継続することができるようになります。また、運用上の停止が必要な場合であっても、各サーバを順次停止して作業することで、サービス全体としては停止させないようにできます。

先に挙げた可用性の4つの要素をどの程度満足するかはデータベースの構成によって変わってきます。詳しくは『2013年度WG3活動成果報告書』[\[PGECCons WG3 2013\]](#)を参考にしてください。

3.1.2. 性能向上

レプリケーションによって、同じ情報を格納しているデータベースサーバが複数存在することになります。それらのサーバでアプリケーションからの要求にこたえることが出来れば、システム全体としての性能状況が期待できます(スケールアウト)。アプリケーションからの要求を複数のサーバに分散させる際には、更新(削除・挿入を含む)クエリを特定の1サーバに集約する「シングルマスタ」構成と、複数のサーバに分散する「マルチマスタ」構成があります。また、参照クエリを複数のサーバに分散させることを参照負荷分散と呼びます。

レプリケーションクラスタを性能向上に用いる場合、複数のデータをベースを同時に運用することから生じる特有の課題があります。レプリケーション方式を選択する際には、それらの課題をどの程度解決しているのかについても考慮する必要があります。

3.1.2.1. 参照の同期

ある瞬間に同一の参照クエリを異なるサーバに送った時に、まったく同じ結果が返ってくるものと、そうでないものとがあります。同じ結果が得られる場合、サーバは同期している、同期レプリケーションであると言えます。

3.1.2.2. 更新による一貫性の維持

マルチマスタ構成の場合には同期の問題に加えて、更新の衝突と一貫性の維持が問題となります。

更新の衝突

複数のアプリケーションプログラムから同一の更新操作を実行しようとした場合、単一のDBサーバであればどちらかの操作が遅延され、場合によってはエラーとなります。マルチマスタ構成では、レプリケーションの送信側でなされた更新と、そのサーバに直接要求された更新とが衝突した場合に、どちらが優先するかが問題となります。

一貫性の維持

更新が衝突した結果、レプリケーションの送信側と受信側でデータベースの内容が異なってしまうと、レプリケーションによって構成されるクラスタ全体でデータベースの一貫性が維持されなくなります。この問題を適切に対処する必要があります。

3.2. 代表的なレプリケーションの手法

ここでは、本報告書で取り上げるレプリケーション手法を中心に、PostgreSQLで利用できる代表的なレプリケーション手法を紹介します。レプリケーションを利用する立場からは、シングルマスタとマルチマスタに二分することができます。その上で、レプリカを生成する方法に着目して代表例を挙げ、そのメリット・デメリットを説明します。

なお、『2013年度WG3活動成果報告書』[\[PGECCons WG3 2013\]](#)では可用性向上の観点から、レプリケーションを含めて様々なPostgreSQLの構成を取り上げていますので、併せてご覧ください。

コミュニティのWikiページには、PostgreSQL上で動作するクラスタソフトウェアについての解説があり、その中にレプリケーションも含まれています [\[PGWiki replica\]](#)。ここで紹介する紹介するレプリケーションソフトウェアについても紹介されています。

3.2.1. シングルマスタ

シングルマスタ構成の場合、レプリカを生成する手法には以下のようなものがあります。

- ストレージレプリケーション
- トリガベースレプリケーション
- クエリベースレプリケーション
- ストリーミングレプリケーション

3.2.1.1. ストレージレプリケーション

PostgreSQLやその上で動作するツールを介することなく、データを格納するストレージのレベルでデータを複製します。ストレージ装置自体がレプリカを生成するものや、DRBD [\[DRBD\]](#) のようにLinux上で動作するソフトウェアによる実現があります。

メリット

- PostgreSQLからは単一のサーバに見えるので、単一サーバと同じように運用できます

デメリット

- 受信側のサーバはデータベースとしては動作していないので、負荷分散に利用することができません

3.2.1.2. トリガベースレプリケーション

PostgreSQLのデータベース内に更新によって起動されるトリガを設定しておき、更新による変分を受信側のサーバに送り出すもの。代表的な製品に Slony-I があります。以下では Slony-I での主なメリット・デメリットを紹介します。

メリット

- PostgreSQLのデータベースクラスタに含まれる表全体だけでなく、任意のテーブルについてだけ複製を作成することができる
- 更新される表については参照負荷分散、それ以外の表については更新負荷分散が可能です

デメリット

- 比較的オーバーヘッドが大きいので、後述のストリーミングレプリケーション方式に比べて、性能が低い傾向があります [\[pglogical\]](#)

3.2.1.3. クエリベースレプリケーション

アプリケーションプログラムとDBサーバ(PostgreSQL)の間に入るミドルウェアによって、発行されたクエリを複製して複数のDBサーバに送信することで、データベースを複製します。代表的な製品に Pgpool-II [\[pgpool2\]](#) があります。

メリット

- 複数のDBサーバを用いて負荷分散を実現する際に、参照クエリ・更新クエリともに適切なサーバにクエリが自動的に振り分けられるため、アプリケーションから見ると単一のDBサーバを利用しているように見える

デメリット

- 一部のSQL文に対する挙動が単一のDBサーバとは異なる

3.2.1.4. ストリーミングレプリケーション

PostgreSQL データベースでは、更新をコミットした際にその結果をクラッシュ等で失わないように更新情報をファイルに書きこむログ先行書込み(Write Ahead Logging; WAL)を用いています。このWALファイルにはデータベースに対する更新を全て復元することができる情報が含まれていますから、これを他のDBサーバに転送することでデータベースを複製することができる --- これがストリーミングレプリケーションの基本的な考え方です。

ストリーミングレプリケーションには、物理レプリケーションと論理レプリケーションとがあります。いずれの方法でも参照負荷分散が可能です。

物理レプリケーションは、WALファイルに書かれた内容をほぼそのまま受信側(スレーブサーバ)に送り出すことで、送信側(マスタサーバ)のDBを複製します。送信されるデータにDBサーバ内の記憶装置レベルのデータが含まれるため、物理レプリケーションと呼ばれます。

メリット

- 受信側サーバに送信側サーバと同一のデータベースを複製することができる
- 送信側サーバでコミット済みのデータを受信側で確実に書き込み済みにするため、高信頼化に適している
- 参照負荷分散ができる

デメリット

- 特定の表だけを複製することはできない
- メジャーバージョンが異なるPostgreSQLの間では利用できない

論理レプリケーションは、送信側でWALファイルを元にSQL文やタプル(レコード)を生成して受信側に送ります。これらのデータの形式には記憶装置レベルのデータを含まないことから、論理レプリケーションと呼ばれます。

メリット

- 送信側サーバの一部の表に対する更新だけを受信側に送ることができる
- 複数の送信側サーバの出力を1つの受信側サーバで受け取ることができる
- メジャーバージョンが異なるPostgreSQLの間でも利用できる

デメリット

- 後日記入予定(27.Feb.2017)

3.2.2. マルチマスタ

マルチマスタ構成の場合、レプリカを生成する手法には以下のようなものがあります。

- Bi-Directional Replication
- Bucardo

3.2.2.1. Bi-Directional Replication

2nd Quadrant社が公開している Bi-Directional Replication は、先に紹介した論理レプリケーションを用いてデータを複製しつつ、複数のサーバでデータの更新を可能としたものです。主な用途としては地理的に離れた場所にある複数のサーバ間で、データを共有する利用形態を想定しています [\[PostgresBDR\]](#)。

メリット

- 論理レプリケーションを利用しているため、他の方式によるマルチマスタに比べてオーバーヘッドが小さい

デメリット

- レプリケーション自体は非同期に行われるため、複数サーバに同時に発行した参照クエリの結果が異なることがあります。

3.2.2.2. Bucardo

Bucardoは、トリガベースレプリケーションを使ってデータを複製します [\[Bucardo\]](#)。そのため、メリット、デメリットはシングルマスタのトリガベースレプリケーションと同じです。

3.3. 参考文献

- [\[IPA\]](#) 独立行政法人情報処理推進機構. 非機能要求グレード 利用ガイド[活用編]. 2010. <http://www.ipa.go.jp/files/000026853.pdf>
- [\[PGWiki_replica\]](#) Smith, G.; Grittner, K.; Pino, Conrad T.; Ringer, C.; Simon, R. et al. Replication, Clustering, and Connection Pooling. 2017. https://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling
- [\[DRBD\]](#) LINBIT, Inc. <http://www.drbd.org/en/>
- [\[pglogical\]](#) 2ndQuadrant. pglogical. <https://2ndquadrant.com/en/resources/pglogical/>
- [\[pgpool2\]](#) PgPool Global Development Group. http://www.pgpool.net/mediawiki/index.php/Main_Page
- [\[Bucardo\]](#) Jensen, G.; Sabino, G. M. et al. https://bucardo.org/wiki/Main_Page

4. ストリーミングレプリケーション

4.1. はじめに

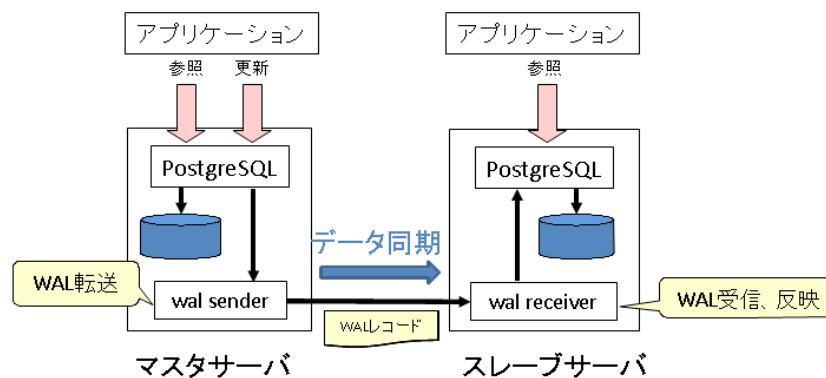
4.1.1. PostgreSQLのストリーミングレプリケーションの特徴

PostgreSQLのストリーミングレプリケーションは以下を目的とした構成です。

- 高可用性
- バックアップ
- 参照負荷分散

以下の特徴があります。

- 変更履歴が格納されたWALを操作単位でマスタ側からスレーブ側へ転送することでデータを同期

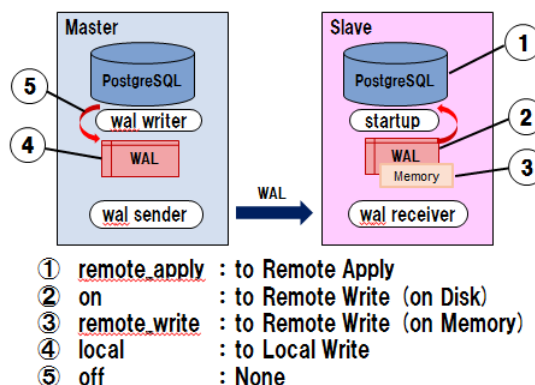


ストリーミングレプリケーション基本構成図(スレーブ1台)

PostgreSQL 9.6における主な機能を示します。主に9.3以降の新機能を対象としています。

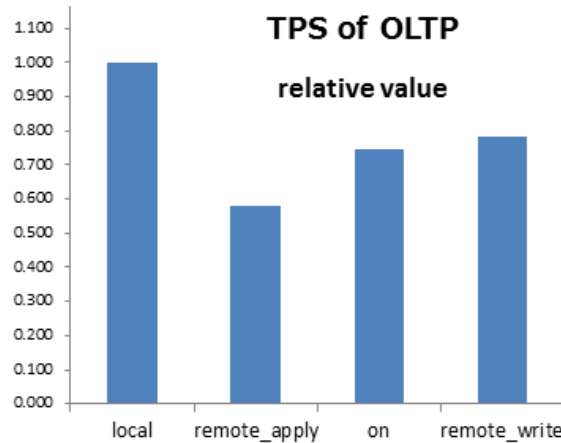
- **完全同期方式**により、スレーブ活用度の向上が期待できる
 - 従来の同期式はWALの転送までの保証であり、データ保護が目的。
 - `synchronous_commit`には以下の設定が可能であり、要件に応じて選択。
 - `remote_apply` : 完全同期。WAL適用まで保証。昇格時の時間短縮と参照負荷分散の拡充が目的。
 - `on` : 同期。WAL転送(ディスク書き込み)まで保証。データ保護が目的。
 - `remote_write` : 準同期。WAL転送(メモリ書き込み)まで保証。データ保護とパフォーマンスのバランスが目的。
 - `local` : 非同期。ローカルのWAL書き込みまで保証。パフォーマンス優先が目的。
 - `off` : 完全非同期。ローカルのWAL書き込みすら保証しない。最も高パフォーマンスだが非現実的。
 - 以下の図は`synchronous_commit`の設定による保証時点を示す。

synchronous_commit parameter setting List

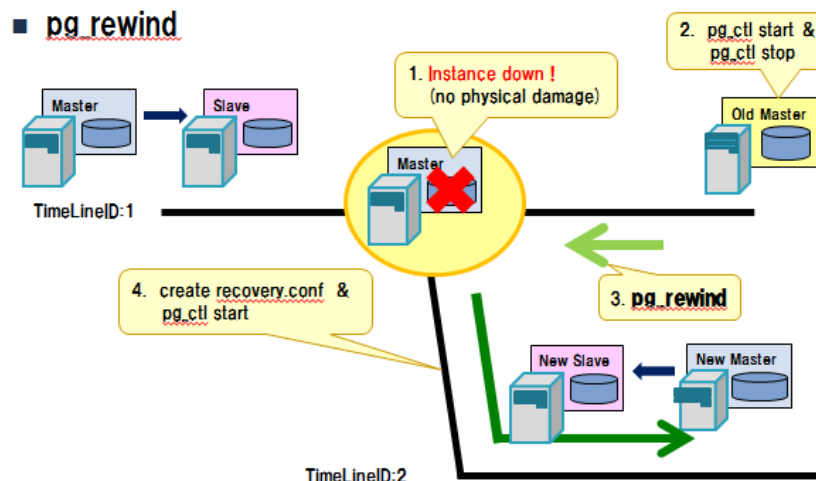


- 以下の図は`synchronous_commit`の各設定によるパフォーマンス比較(`local`の値を1.00とした相対値)。
- サーバスペック
 - CPU : Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz 16Core
 - メモリ : 64GB
 - OS : Red Hat Enterprise Linux Server release 6.5 (Santiago)
 - PostgreSQLのバージョン : PostgreSQL 9.6.1
- パラメータ設定
 - キャッシュヒット率の影響を受けないようほぼ100%となるように調整

- shared_buffers : 8GB
- 処理中に自動VACUUMが発生しないように無効化
 - autovacuum : off
- 処理中にチェックポイントが発生しないように調整
 - checkpoint_timeout : 1h
 - max_wal_size : 10GB
 - 処理前に毎回手動でチェックポイント実行
- トランザクションツールはpgbench を使用。
 - 初期化スケール 100
 - 100セッション、25ワーカーレッドのトランザクションを実施(pgbench -c 100 -j 25)
- 通常のアプリケーションではこれほど低下しないと考えられるが、一定の甘受は必要。



- レプリケーションスロット により、スレーブに必要なWALをマスタが確保し続ける
 - スレーブの障害時(ネットワーク不調含む)等にマスタでWALを保持し切れず、WALファイルの再利用によりロストする懸念がある。
 - 従来はwal_keep_segmentsパラメータによる調整、またはアーカイブモード運用が必要であった - wal_keep_segmentsはWAL数を指定するものであるため、見積もりは困難。 - アーカイブモードも複数スレーブ構成における不要WALファイルの判断はやはり困難。
 - レプリケーションスロットはWALの不要WALファイルの判断をシステムに任せる事ができる。
 - レプリケーションのWALファイル管理のためにアーカイブモード運用は必要はなく、過去の時点に復旧する(PITR)要件がある場合に設定する。
- 複数同期スレーブ構成が可能
 - 従来も複数スレーブ構成は可能であったが、同期スレーブはその内の1台までという制限があったがそれが取り払われた。
- 遅延レプリケーション機能 により、オペレーションミスを反映させない事が可能に
 - スレーブでの適用を一定時間待機する機能である。マスタでのオペミスが即座にスレーブに伝搬されるのを防ぐのが主な目的。
 - WALファイルの転送は遅延なく処理されるため、データ保全(RPO)の観点では問題ない。同期モードで設定する事もできる。
 - recovery.conf のrecovery_min_apply_delayパラメータに遅延時間を設定。
 - 従来からの手法として、PITRを使用してオペミス直前時点を指定してリカバリすることで障害回復を図る方法もある。
- 巻き戻し機能 (pg_rewind)により、効率の良いレプリケーション再構成が可能に
 - 旧マスタを分岐時点へ巻き戻す事で、新マスタに追いつく事が可能な状態とする機能である。
 - 巻き戻し後、旧マスタに新マスタのWALを適用(追いつき)する事で旧マスタを新スレーブとして構成。
 - 従来はpg_basebackup等による再作成が必要であった。
 - 大規模データベースでは、大幅な時間短縮が期待できる。
 - 下図に巻き戻し機能のイメージを示す。



ストリーミングレプリケーションの運用上の注意点

- 死活監視と障害発生時のフェイルオーバーはPostgreSQL機能ではできないため、クラスタソフトを利用する必要があります。商用クラスタソフトが使用される場合もありますが、オープンソースソフトウェアのpgpool-IIと呼ばれるクラスタソフトを使用した例も多く報告されています。特に参照負荷分散を行う場合は、pgpool-IIを使用します。

4.1.2. 想定構成

今回はストリーミングレプリケーションの基本機能の検証を目的としています。以下の構成を想定しています。

- 対象バージョンは PostgreSQL 9.6
 - 2017/03時点の最新版
- 2ノード構成に限定
 - 基本機能の検証が目的であるため、2ノードを想定
 - 複数同期スレーブ構成などは来季以降の課題とする
- PostgreSQLの機能に限定
 - pgpool-II等のクラスタソフトに関するテーマは対象外とする

4.2. SR環境構築時の設定項目、推奨値

4.2.1. SRにおける目的別の設定

PostgreSQLのSR(ストリーミングレプリケーション)には様々な機能があり、対処する障害に応じて適切に設定する必要があります。また各機能は組み合わせることが可能です。

表 4.1 ストリーミングレプリケーションの機能

機能	内容	目的	注意点
アーカイブログ	WALファイルが循環して再利用される前に、アーカイブログとして保存する。これにより全ての変更履歴が保持される。	● データの損失がないよう、障害発生直前まで復旧できるようにする。 ● 操作ミス対策に任意の時点まで巻き戻せるようにする。 ● スレーブとの同期を保証させる。	● アーカイブログが領域を圧迫するため、定期的な削除が必要になる。
レプリケーションスロット	スレーブ側で適用されていないWALファイルを保持する。	● スレーブとの同期を保証させる。	● スレーブへWALファイル転送ができないと、WALが溜まり続け領域を圧迫する。
遅延レプリケーション	スレーブ側のWAL適用を意図して遅らせることで、スレーブを過去の状態に維持する。	● 操作ミス対策に任意の時点まで巻き戻せるようにする。	● スレーブは常に過去の状態であるため、参照活用には一定の制限がある。 ● フェイルオーバー時には遅延分の適用が必要になる。
WAL圧縮	Full Page Write時(チェックポイント後の最初の更新時)に、WALファイルに書き出すフルページイメージを圧縮する。圧縮されたWALは適用時に解凍される。	● 回復不可能なデータ破壊のリスクを増やさずに、WALの量を減らせる。	● 圧縮時と解凍時に通常より余分にCPUを使用する。

4.2.2. SR環境の設定手順

基本的なSR環境の設定手順を紹介します。尚、マスタとスレーブの両サーバにPostgreSQLはインストール済みであり、マスタ側ではデータベースクラスタを構築していることを前提としています。

1. 関連パラメータ

ストリーミングレプリケーション構成に最低限必要な設定は以下の通りです。スレーブはマスタのベースバックアップから作成されるため、マスタに設定したパラメータは全てスレーブも同様の値に設定されます。

- "ストリーミングレプリケーション"を"SR構成"と表記します。
- "レプリケーションスロット"を"スロット"と表記します。

説明の便宜上、以下の設定とします。

表 4.2 構成データ

項目	内容
マスタのIPアドレス およびサーバ名	192.168.100.101/24 server1
スレーブのIPアドレスおよびサーバ名	192.168.100.102/24 server2
ポート番号	5432
レプリケーション用ユーザ/パスワード	repuser/password

表 4.3 ストリーミングレプリケーションの設定(マスタのpg_hba.conf)

設定値	内容
host replication repuser 192.168.100.101/32 md5 host replication repuser 192.168.100.102/32 md5 またはCIDR指定にて host replication repuser 192.168.102.0/24 md5	レプリケーション用ユーザの接続を許可する。 ・pg_hba.confは両ノード共通の設定となる事から、 どちらがマスタになっても使用できる設定とする。 ・IPアドレスはWAL転送を行うネットワークLANを指定。 パブリックLAN以外に、ハートビート系LANも有力。 ・認証方式は任意。

表 4.4 ストリーミングレプリケーションの設定(マスタのpostgresql.conf)

パラメータ	設定値	内容
listen_addresses	'*' または '0.0.0.0'	接続を受け入れるIPを指定する。
wal_level	replica (9.5以前の場合: hot_standby)	WALに書かれる情報量を指定する。
wal_keep_segments	適切な数	最低限保持するWALファイル数を指定する。 スロットを作成する場合は設定不要。
synchronous_commit	以下の設定が可能。 remote_apply: 完全同期 on: 同期 remote_write: 準同期 local: 非同期 off: 完全非同期	目的に応じて設定する。 remote_apply: WAL転送から適用までを保証。 on: WAL転送後ディスクへの書き込みまで保証。 remote_write: WAL転送後メモリ書き込みまで保証。 local: ローカルへの書き込みまで保証。 off: ローカルへの書き込みも保証しない。 同期式におけるスレーブ障害時および復旧時の同期/非同期の切り替えはsynchronous_standby_namesにて行う。本パラメータは固定。
synchronous_standby_names	非同期の場合: '' 同期の場合: '*' あるいは特定スレーブのapplication_name	レプリケーションの非同期と同期を切り替えに使用。 リロードで反映可能。 スレーブが1台であれば '*' を設定。
max_wal_senders	スレーブ数以上を設定	起動するwal senders数を指定。 pg_basebackupでWALストリーミングを指定する場合はその分も考慮
max_replication_slot	スレーブ数以上を設定	レプリケーションスロットを作成可能な最大数
restart_after_crash	off	onの場合、インスタンス障害(postgresプロセスは残存)の場合に、自動的に再起動を行う。 シングル構成では便利な機能であるが、SR構成では管理が複雑化するため、offに設定するのが一般的。

表 4.5 ストリーミングレプリケーションの設定(スレーブのpostgresql.conf)

パラメータ	設定値	内容
hot_standby	on	ホットスタンバイとして参照可能な状態で起動する。 スレーブに対しても監視SQLを実行できるように通常は有効化する。
hot_standby_feedback	on	onの場合、スレーブが現在処理している問い合わせについて、プライマリへフィードバックを送る。 通常は有効化する。特にスロットを作成する場合には有効化が必須。

表 4.6 ストリーミングレプリケーションの設定(スレーブのrecovery.conf)

パラメータ	設定値	内容
standby_mode	on	スレーブとして起動か、PITRのリカバリ処理かを区別をする設定。 onの場合は、スレーブとして起動する。WALファイルの最後に達してもリカバリを終了せず、マスタへ接続して新WALセグメント取得を継続。
primary_conninfo	'host=server1 port=5432 user=repuser password=password application_name=<アプリ名>'	マスタへの接続文字列を指定。 hostはWAL転送を行うLANを指定。2サーバ間の転送のみである事から、ハートビートLANでも良い。 userはreplication仮想データベースへ接続できるユーザを指定(ユーザのreplication属性およびpg_hba.confの設定)。 passwordはパスワードが必要な場合に設定。パスワードの記載を避けたい場合は、~/.pgpassを使用。 application_nameは視認性を高めるのが目的であり必須ではない。デフォルトで'wal_receiver'。
recovery_target_timeline	latest	リカバリが作成する個別のタイムラインを指定。 SR構成ではlatestを指定して最新タイムラインを追従。
primary_slot_name	<スロット名>	当該スレーブ用のスロットを作成している場合は、スロット名を設定。 最新バージョンではスロットを作成するのが一般的。
restore_command	アーカイブWALファイルのリストア(コピー)方法を指定	スロットを使用している場合は設定不要

2. 構築手順

ここでは基本的な構成における手順を記載します。

表 4.7 ストリーミングレプリケーション構成条件

項目	内容
同期式	非同期(synchronous_commit = off)
レプリケーションスロット	使用しない

(1)レプリケーションユーザ作成

マスタサーバでレプリケーション用のユーザを作成します。

```
postgres=# CREATE ROLE repuser LOGIN REPLICATION PASSWORD 'repuser';
```

(2)設定ファイルの変更

マスタで\$PGDATA配下のpg_hba.confとpostgresql.confを変更します。

- pg_hba.conf

レプリケーションユーザがデータベース接続できるようpg_hba.confを次のように設定します。

```
$ vi $PGDATA/pg_hba.conf

[pg_hba.conf]
host    replication  repuser  127.0.0.1/32    md5
host    replication  repuser  192.168.100.101/32  md5
host    replication  repuser  192.168.100.101/32  md5
```

- postgresql.conf

マスタとして稼働させるため、postgresql.confを次のように設定します。

```
$ vi $PGDATA/postgresql.conf

[postgresql.conf]
port = 5432
listen_addresses = '*'
wal_level = replica
synchronous_commit = off
synchronous_standby_names = ''
max_wal_senders = 3
max_replication_slots = 3
restart_after_crash = off
```

パラメータの記述後、設定ファイルの変更を反映するためにPostgreSQLを再起動します。

```
$ pg_ctl restart
```

(3)スレーブの構築

スレーブでpg_basebackupを実行し、スレーブデータベースを構築します。

```
$ pg_basebackup -h <マスタIP> -p 5432 -U repuser -D $PGDATA --xlog --progress -
-verbose
```

(4)スレーブの設定

pg_basebackupで取得したデータベースクラスタの設定を変更し、スレーブとして稼働するようにします。そのために取得先(\$PGDATA)配下で、postgresql.confを変更し、さらにrecovery.confを作成します。

- postgresql.conf

次のパラメータを変更します。

```
$ vi $PGDATA/postgresql.conf

[postgresql.conf]
hot_standby = on
hot_standby_feedback = on
```

- recovery.conf

\$PGDATA配下にrecovery.confを作成し、以下を記述します。

```
$ vi $PGDATA/recovery.conf

[recovery.conf]
standby_mode = 'on'
primary_conninfo = 'host=server1 port=5432 user=repuser
password=repuser'
recovery_target_timeline = latest
```

(5)スレーブの起動

スレーブを起動します。

```
$ pg_ctl start
```

ストリーミングレPLICATIONが構築され、データが同期されていることを確認します。

[マスタ側で確認]

```
postgres=# \x
postgres=# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]-----+-----
pid                | 40724
usesysid           | 16385
username           | repuser
application_name   | walreceiver
client_addr        | <スレーブのIP>
client_hostname    |
client_port        | 48077
backend_start      | 2017-03-15 14:29:51.283395+09
backend_xmin       | 548
state              | streaming
sent_location      | 0/7000840
write_location     | 0/7000840
flush_location     | 0/7000840
replay_location    | 0/7000840
sync_priority      | 0
sync_state         | async
```

[スレーブ側で確認]

```
postgres=# \x
postgres=# SELECT * FROM pg_stat_wal_receiver;
-[ RECORD 1 ]-----+-----
pid                | 40723
status             | streaming
receive_start_lsn  | 0/7000000
receive_start_tli  | 1
received_lsn       | 0/7000920
received_tli       | 1
last_msg_send_time | 2017-03-15 14:33:38.605953+09
last_msg_receipt_time | 2017-03-15 14:33:38.606003+09
latest_end_lsn     | 0/7000920
latest_end_time    | 2017-03-15 14:32:08.44417+09
slot_name          |
conninfo           | user=repuser password=***** dbname=replication
                   | host=<マスタのIP> port=<マスタのポート番号>
                   | fallback_application_name=walreceiver
                   | sslmode=prefer sslcompression=1
```

4.2.3. アーカイブモード運用

PostgreSQLは全ての変更履歴をWALファイルに記録しています。データベースクラスタのクラッシュ時は、WALに記録されている最新の変更履歴を用いることで自動復旧しています。ただしWALファイルは循環使用するため、古いWALは再利用(上書き)されます。アーカイブモード運用により、WALが再利用される前にアーカイブWALファイルとして保存することが可能です。これにより、データベースクラスタに物理障害が発生した場合にも、バックアップのリストア後にアーカイブWALファイルと最新WALファイルを適用することで障害発生直前まで復旧することが可能です。

また任意の時点までデータを戻すことも可能です。PITR (Point In Time Recovery)という技法です。ただしアーカイブWALファイルは蓄積され続け領域を圧迫するため、定期的に削除するなどの管理が必要です。

1. 関連パラメータ

既存のレプリケーションスロットの設定に加え、次のパラメータを指定します。

サーバ	設定ファイル	パラメータ	設定値	内容
マスタ	postgresql.conf	archive_mode	on	アーカイブを有効化する。
マスタ	postgresql.conf	archive_command	cp %p <アーカイブの保存先>/%f	アーカイブの実行コマンドを指定する。スクリプトを指定する事も可能であるため、複雑な処理を組み込む事が可能。
スレーブ	recovery.conf	restore_command	'scp <マスタのユーザ名>@<マスタのホスト名>:<マスタのアーカイブ・ディレクトリ>%f %p'	マスタのアーカイブを取得するコマンドを指定する。WAL同期を保証する場合に必要な。ただしレプリケーションスロットを使用する場合は不要である。

注釈

- restore_commandを使用する場合、スレーブはマスタのPostgreSQLのOSユーザに sshにてパスワードなしで接続できるようになる必要があります。またはアーカイブ・ディレクトリをNFSマウント等、マスタ/スレーブ間で共有可能なパスにする事で、アーカイブの扱いが容易になります。
- 障害時スレーブをマスタにする場合に備え、マスタに設定したパラメータはスレーブ側でも事前に有効化することを推奨します。

2. 設定手順

(1)マスタ側のpostgresql.confのパラメータを以下のように設定する。

```
$ vi $PGDATA/postgresql.conf

[postgresql.conf]
archive_mode = on
archive_command = 'cp %p <アーカイブの保存先>/%f'
```

(2)必要ならばスレーブ側のrecovery.confのパラメータに以下を追加する。

```
$ vi $PGDATA/postgresql.conf

[postgresql.conf]
restore_command = 'scp <マスタのユーザ名>@<マスタのホスト名>:<マスタのアーカイブ・ディレクトリ>%f %p'
```

(3)マスタ側を再起動し、その後スレーブ側を再起動する。

```
[マスタ/スレーブの両方で実施]
$ pg_ctl restart
```

(4)マスタ側で強制的にWALファイルを切り替え、アーカイブWALファイルが出力されることを確認する。

```
postgres=# SELECT pg_switch_xlog();
pg_switch_xlog
-----
0/7017008
(1 row)

$ <アーカイブ・ディレクトリ>
00000001000000000000000007
```

4.2.4. レプリケーションスロット

スレーブに未転送のWALを保持することで、レプリケーションの同期を保証します。アーカイブ運用でも同期の保証は可能ですが、アーカイブログの管理などが問題となり、ノーアーカイブ運用をしている環境も多くあります。そのような環境において、レプリケーションの同期を保証するためには、レプリケーションスロットの設定が必要です。またアーカイブ運用においても、アーカイブWALファイルの削除にレプリケーションの考慮が不要になるため、有用な設定です。

1. 関連パラメータ

既存のレプリケーションスロットの設定に加え、次のパラメータを指定します。

表 4.8 レプリケーションスロットの設定

サーバ	設定ファイル	パラメータ	設定値	内容
マスタ	postgresql.conf	max_replication_slots	スレーブ数以上	作成可能なレプリケーションスロット数を指定する。
スレーブ	recovery.conf	primary_slot_name	レプリケーションスロット名	使用するレプリケーションスロット名を指定する。

注釈

- 障害時スレーブをマスタにする場合に備え、マスタに設定したパラメータはスレーブ側でも事前に有効化することを推奨します。

2. レプリケーションスロットの作成方法

レプリケーションスロットに関連する関数は以下の通りです。

表 4.9 レプリケーションスロットの関数

関数名	説明
pg_create_physical_replication_slot(スロット名[, true/false])	レプリケーションスロットを作成する。 ● スロット名: 作成するレプリケーションスロット名を指定する。 ● true/false: 9.6から追加された。trueの場合、レプリケーションスロットは即座にWALを保持する。falseの場合従来通り、スレーブがレプリケーションスロットに繋いだ時点からWALを保持する。
pg_drop_replication_slot(スロット名)	レプリケーションスロットを削除する。 ● スロット名: 削除するレプリケーションスロット名を指定する。

3. 検証

(1)マスタ側でpostgresql.confにmax_replication_slotsを加え、設定反映のため再起動する。

```
$ vi $PGDATA/postgresql.conf

[postgresql.conf]
max_replication_slots = 3

$ pg_ctl restart
```

(2)マスタ側でレプリケーションスロットを作成する。

```
postgres=# SELECT pg_create_physical_replication_slot('slot1', true);
pg_create_physical_replication_slot
-----
(slot1,0/B000220)
(1 row)
```

(3)マスタ側でレプリケーションスロットの作成を確認する。

```
postgres=# SELECT slot_name, restart_lsn, active FROM pg_replication_slots;
 slot_name | restart_lsn | active 
-----+-----+-----
 slot1     | 0/B000220   | f
(1 row)
```

(4)スレーブ側でrecovery.confにprimary_slot_nameを加え、設定反映のため再起動する。

```
$ vi $PGDATA/postgresql.conf

[recovery.conf]
primary_slot_name = 'slot1'

$ pg_ctl restart
```

(5) マスタ側でレプリケーションスロットが使用されていることを確認する。

```
postgres=# SELECT slot_name, restart_lsn, active FROM pg_replication_slots;
 slot_name | restart_lsn | active 
-----+-----+-----
 slot1     | 0/B0002C8   | t
(1 row)
```

4.2.5. 遅延レプリケーション

スレーブの適用を一時的に遅延させます。マスタの操作ミスが即座に伝搬されるのを防ぐのが目的です。WAL転送は遅延なく処理されるため、データ保全の観点では問題ありません。PITRでも過去の任意時点へ戻せますが、アーカイブ運用が必要となります。またベースバックアップから戻す時点までWALを適用するため、時間がかかる場合があります。WAL遅延はノーアーカイブ運用でも実施可能であり、遅延している時点からWALを適用できます。

1. 関連パラメータ

既存のレプリケーションスロットの設定に加え、次のパラメータを指定します。

表 4.10 WAL遅延の設定

サーバ	設定ファイル	パラメータ	設定値
スレーブ	recovery.conf	recovery_min_apply_delay	遅延させる時間を指定する。

2. 検証

(1) テストに使用するテーブルを作成する。

```
postgres=# CREATE TABLE delay_test(id integer, time1 timestamp);
postgres=# \d delay_test
Table "public.delay_test"
Column | Type | Modifiers 
-----+-----+-----
 id    | integer | 
 time1 | timestamp without time zone |
```

(2) スレーブ側でrecovery.confにrecovery_min_apply_delayを追加し、再起動する。

```
$ vi $PGDATA/recovery.conf

[recovery.conf]
recovery_min_apply_delay = '60min'

※今回は検証のため60分遅延する。

$ pg_ctl restart
```

(2) マスタでテストテーブル1分間隔でデータを追加する。

```

postgres=# INSERT INTO delay_test VALUES (1, localtimestamp);
~約1分後~
postgres=# INSERT INTO delay_test VALUES (2, localtimestamp);

postgres=# SELECT * FROM delay_test ;
 id |          time1
-----+-----
  1 | 2017-03-27 18:05:46.29221
  2 | 2017-03-27 18:06:46.728155
(2 rows)

```

(3)id=2の更新を操作ミスとみなし、スレーブを用いて更新前へ戻す作業を開始する。

(4)マスタを停止する。

```
$ pg_ctl stop
```

(5)スレーブのrecovery.confを修正する。

```

$ vi $PGDATA/recovery.conf

[recovery.conf]
#recovery_min_apply_delay = '60min'
recovery_target_time = '2017-3-27 18:06:00'

$ pg_ctl restart

```

(6)スレーブのデータでid=2の更新が適用されていないことを確認する。

```

postgres=# SELECT * FROM delay_test ;
 id |          time1
-----+-----
  1 | 2017-03-27 18:05:46.29221
(1 row)

```

(7)スレーブをマスタに昇格するため設定ファイルを変更する。

```

$ vi $PGDATA/postgresql.conf

[postgresql.conf]
#hot_standby = on
#hot_standby_feedback = on

$ vi $PGDATA/recovery.conf

[recovery.conf]
#standby_mode = 'on'
#primary_conninfo = 'host=<マスタのIP> port=<マスタのポート番号> user=repuser password=repuser'
recovery_target_timeline = 'latest'
#recovery_min_apply_delay = '60min'
recovery_target_time = '2017-3-27 18:06:00'
restore_command = ''

```

(8)スレーブを再起動し、マスタにする。

```
$ pg_ctl restart
```

(9)id=2の更新が適用されておらず、マスタとなったことを確認する。

```
postgres=# SELECT * FROM delay_test ;
 id |          time1
-----+-----
  1 | 2017-03-27 18:05:46.29221
(1 row)

postgres=# SELECT pg_is_in_recovery();
 pg_is_in_recovery
-----
 f
(1 row)
```

4.2.6. WAL圧縮

Full Page Write時(チェックポイント後の最初の更新時)に、WALに書き出すフルページイメージを圧縮します。圧縮されたWALは適用時に解凍されます。WALファイルのサイズが小さくなるため、**書き込みや転送の時間短縮** が期待されます。注意点として、圧縮処理および解凍処理が発生するため、通常より余分にCPUを使用します。適用についてはそれらを総合的に判断します。一般的な適用場面として、ストリーミングレプリケーション構成において効果的と考えられます。以下の構成の適用を容易にします。

- 同期モード(転送まで)または完全同期モード(適用まで)
- スレーブの遠隔地配置(ディザスタ・リカバリ)

1. 関連パラメータ

既存のレプリケーションスロットの設定に加え、次のパラメータを指定します。

表 4.11 WAL圧縮の設定

サーバ	設定ファイル	パラメータ	設定値
マスタ	postgresql.conf	wal_compression	'on'

4.3. SR環境の監視

4.3.1. レプリケーション操作ログの監視

4.3.1.1. 調査の目的

SR構成のサーバログ監視について有益な情報を提示する事を目的としています。以下の挙動時にサーバログに出力されるメッセージを確認しました。

- マスタの起動/停止
- スレーブの起動/停止
- マスタ側のWAL再利用によるロス(スロット不使用)

以下のパターンで検証しました。

- レプリケーション操作ログ出力無効時(デフォルト)
 - log_replication_command = off
- レプリケーション操作ログ出力有効時
 - log_replication_command = on

4.3.1.2. 調査結果

マスタ側またはスレーブ側のサーバログに出力されるレプリケーション情報についてまとめます。ホスト名、IPアドレス、ポート番号、ユーザ名、データベース名は例です。

- レプリケーション操作ログ出力無効時(デフォルト)

表 4.12 デフォルトで出力される情報

タイミング	サイト	メッセージ
スレーブ停止時	マスタ	LOG: unexpected EOF on standby connection この時のSQLSTATEは " 08P01 protocol_violation" です。
マスタ停止時	スレーブ	エラーメッセージが繰り返し出力されます。 LOG: invalid record length at 132/EF8AD7F0: wanted 24, got 0 FATAL: could not connect to the primary server: could not connect to server: Connection refused Is the server running on host "master" (192.168.100.100) and accepting TCP/IP connections on port 5432 この時のSQLSTATEは " XX000 internal_error" です。
WAL再利用によるロス	マスタ	エラーメッセージが繰り返し出力されます。 ERROR: requested WAL segment 0000001100000132000000EF has already been removed この時のSQLSTATEは " 58P01 undefined_file" です。
WAL再利用によるロス	スレーブ	エラーメッセージが繰り返し出力されます。 LOG: started streaming WAL from primary at 132/EF000000 on timeline 17 FATAL: could not receive data from WAL stream: ERROR: requested WAL segment 0000001100000132000000EF has already been removed この時のSQLSTATEは " XX000 internal_error" です。

- レプリケーション操作ログ出力有効時

デフォルトで出力される情報に加えて、以下が出力されます。

表 4.13 監視強化時に出力される情報

タイミング	サイト	メッセージ
WAL再利用によるロス	マスタ	エラーメッセージが繰り返し出力されます。 LOG: received replication command: IDENTIFY_SYSTEM LOG: received replication command: START_REPLICATION 134/4A000000 TIMELINE 17 ERROR: requested WAL segment 0000001100000132000000EF has already been removed この時のSQLSTATEは " XX000 internal_error" です。

4.3.1.3. 調査結果

- エラーについては、デフォルト設定で出力があります。
エラーの監視という意味では、デフォルトで問題ないと言えます。
- 表の太字の文字が監視キーワード候補です。
- マスターまたはスレーブを停止すると、反対側にエラー出力があります。
- log_replication_command による出力はそれほど多くなく、監視に対する影響は限定的です。
ただリロードで反映できることと合わせて、気軽に有効化できると言えます。
- log_min_message = debug とすることでレプリケーション関連のメッセージが出力されます。
ただし影響が大きいため、通常はlog_replication_commandを使用します。
- log_replication_commandによる出力は、スレーブのwal receiverプロセスからフィードバックされた情報です。
log_line_prefixにapplication_name(%a)を設定する事で確認できます。

4.3.2. 同期状況の監視

非同期レプリケーションを構成した場合、ハードやNW 構成、利用状況によって
マスターとスレーブの同期状況が変化します。
この状態を監視し、同期の遅延が許容範囲内であるかを確認します。

■前提

構成 : マスター、スレーブの2台構成

バージョン : PostgreSQL 9.6

同期モード : 問わない

4.3.2.1. 同期遅延監視

(1)WALの書き込み位置による同期遅延量を用いて監視することは可能か
「2014年度WG3活動報告書- 可用性編 -」での検証結果をベースに実施します。

マスタのWALとスタンバイが適応したWALの2つの差分は、マスタの最新WALの書き込み位置
(LSN: Log Sequence Number)とスタンバイが適用したWALの位置(LSN)との差分より算出します。
マスタのLSNは、pg_current_xlog_insert_location()関数が返す現在のWALの挿入位置とみなします。
スタンバイのLSNは、統計情報pg_stat_replicationビューのreplay_locationを使用します。
この2つのLSNの差分をバイト単位で表示するために、pg_xlog_location_diff関数の引数にそれぞれを格納し
その結果が許容値を超過するかどうかで、監視を行います。

検証には、以下シェルスクリプトで実施しました。
監視通知部については、テストの為、標準出力のみとしています。

```

1  #!/bin/sh
2  HOST=localhost
3  LIMIT=1
4  SLEEPTIME=2
5  ALERT="echo"
6
7  function send_alert(){
8      ${ALERT} $1
9  }
10
11  while :
12  do
13      val1=$(psql -h $HOST -p 5432 -U postgres -q -t -c "select current_time;")
14      val2=$(psql -h $HOST -p 5432 -U postgres -q -t -c "select client_addr,
15          pg_xlog_location_diff(master,replay_location)as
16          replaydiff from (select pg_current_xlog_insert_location() master)as m,
17          pg_stat_replication;")
18
19      if [ ! -n "${val2}" ]; then
20          send_alert "Slave Down |${val1}"
21      else
22          byte=`echo ${val2} |awk '{print $NF}'`
23          if [ ${byte} -gt ${LIMIT} ]; then
24              send_alert "Replication Delay |${val1},${val2}"
25          fi
26      fi
27      sleep ${SLEEPTIME}
28  done

```

監視シェルを実行中にinsert文を実行した結果を示します。
以下の通り、遅延状況を検知でき、メールサーバ等に連携することで監視することが可能です。
また間接的に更新待ち監視も行うことができます。詳細は、更新・読取監視の項に記載します。

```

$ ./kanshi_No8.sh
Replication Delay | 03:03:39.701249+09, ***.***.***.*** | 104
Replication Delay | 03:03:41.725377+09, ***.***.***.*** | 104
Replication Delay | 03:03:43.745755+09, ***.***.***.*** | 104
Replication Delay | 03:03:47.790622+09, ***.***.***.*** | 56
Replication Delay | 03:03:49.813697+09, ***.***.***.*** | 56
Replication Delay | 03:03:51.834757+09, ***.***.***.*** | 56

```

4.3.3. 更新、読み取りの監視

非同期レプリケーションを構成した場合、ハードやNW構成、利用状況によって
マスターとスレーブの同期状況が変化します。
この状態を監視し、同期の遅延が許容範囲内であるかを確認します。

■前提

構成 : マスター、スレーブの2台構成
バージョン : PostgreSQL 9.6
同期モード : 同期または完全同期 (synchronous_commit = on / remote_apply)

4.3.3.1. 更新・読取監視

(1)PostgreSQLの機能により監視することは可能か

- 検証方法

PostgreSQLの内部パラメータ(statement_timeout)により、マスターへの更新処理に対して
スレーブ停止によって一定時間応答がない場合に接続を切断し、アラート通知することは可能かを検証を行いました。
statement_timeoutは、postgresql.conf内での設定は推奨されていない為、SQL発行時に設定しました。
また、pg_stat_replicationビューにより間接的に、検知できないかについて検証を行いました。

- 検証結果

statement_timeoutを設定しSQLを発行しましたが、有効に機能せず
Ctrl+Cをキーインするまで停止しないという結果となりました。

```
[postgres@test_pg02 ~]$ psql -U postgres -d testdb -q -t << EOF
> \timing on
> SET statement_timeout TO 5000;
> insert into test(id) values(1);
> delete from test where id=1;
> EOF
時間: 0.192 ms
^Ccancel request sent
WARNING: canceling wait for synchronous replication due to user request
DETAIL: The transaction has already committed locally, but might not have been
replicated to the standby.
時間: 13626.496 ms
```

pg_stat_replicationビューについては、スレーブサーバ停止時にレコードが取得できなくなる為
この点を利用することで間接的に、同期待ちを検知することが可能であることがわかります。

```
testdb=# select * from pg_stat_replication;
 pid | usesysid | username | application_name | client_addr | client_hostname |
 client_port | backend_start | backend_xmin | state | sent_location | write_location | flush_location | replay_location | sync_priority |
 sync_state
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
--+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----
(0 行)
```

(2)OSコマンドなどの外部機能により、更新処理に対して一定時間応答がない場合を監視する

timeoutコマンドにより、指定秒数で強制切断に成功し、スタンバイと同期が取れていないメッセージが出力されます。
また終了値として、124が返却されている為、メッセージ内容または、終了値を制御することで
スレーブサーバ障害による同期待ちを検知することが可能です。

```
$ timeout -sINT 5 psql -U postgres -d testdb -q -t << EOF
> \timing on
> insert into test(id) values(1);
> delete from test where id=1;
> EOF
Cancel request sent
WARNING: canceling wait for synchronous replication due to user request
DETAIL: The transaction has already committed locally, but might not have been
replicated to the standby.
時間: 4995.329 ms

$ echo $?
124
```

4.3.4. スプリットブレインの監視

4.3.4.1. スプリットブレインの定義と調査の目的

SR構成においては、通常はマスターの障害を検知した場合にスレーブを昇格させます。つまりマスターは常に1台のみです。ただしオペレーションミスにより、マスターが正常な状態にもかかわらずスレーブを昇格させてしまうという事も有り得ます。稼働中のマスター（シングル）が2台という危険な状態になります。その状態をスプリットブレインと定義します。

その場合でも全てのアプリケーションが元のマスターにのみ接続していれば問題ありませんが、2台目のマスターにも接続が発生するとデータの整合性が損なわれてしまいます。それを避けるため、スプリットブレイン状態になっていないかの監視の方法を検討します。

4.3.4.2. 監視方法

以下の監視についてまとめます。

- サーバログの監視
- pg_controldataコマンドによる監視
- pg_control_recovery関数による監視
- pg_is_in_recovery関数による監視

4.3.4.3. サーバログの監視

pg_ctl promote コマンドにより昇格した場合は、昇格したノードのサーバログに以下のメッセージが出力されます。**timeline IDが変化**した事が分かります。

```
received promote request
selected new timeline ID: XX
```

マスター稼働中にスレーブ側にこのようなメッセージが出力されていないかを監視します。

注意点

recovery.confを削除(または改名)した後に再起動する事でマスター(シングル)として起動する方法もあります。

ただし timeline ID の変化がなく、特徴的なメッセージは出力されません。状態変更の捕捉が困難です。

4.3.4.4. pg_controldataコマンドによる監視

pg_controldataコマンドは制御ファイル(\$PGDATA/global/pg_control)の状態を表示します。

以下に実行例を示します。ここでは環境変数LANGの設定により英語で表示しています。

(実行例)

```
$ export LANG=C
$ pg_controldata
pg_control version number:      960
Catalog version number:        201608131
Database system identifier:     6356464040634181169
Database cluster state:         in archive recovery
pg_control last modified:       Tue Dec 13 14:39:49 2016
～以下略～
```

多数の項目がありますが、ここでは"Database cluster state"に着目します。必要な項目のみ抽出する例です。

(スレーブでの実行例)

```
$ pg_controldata | grep "Database cluster state"
Database cluster state:         in archive recovery
```

4種類の状態があります。

- マスター/スレーブ
- 稼働中/停止中

表 4.14 Database cluster stateで表示される値

表示される値(英語)	表示される値(日本語)	意味
in production	運用中	マスターとして稼働中
in archive recovery	アーカイブリカバリ中	スレーブとして稼働中
shut down	シャットダウン	マスターとして停止中
shut down in recovery	リカバリしながらシャットダウン中	スレーブとして停止中

"in archive recovery"はPITRとしてのリカバリ中の場合も含まれますが、ここではスレーブ状態の意味です。両ノードとも"in production"状態でない事を確認します。

sshコマンドでリモートの状態を容易に取得できます。

注意点としては、リモート側の環境変数を認識しないため、明示的に指定する必要があります。

以下はローカルとリモートで同一設定の前提です。

```
$ ssh <remote> $PGHOME/bin/pg_controldata $PGDATA | \
> grep "Database cluster state"
Database cluster state:                in production
```

4.3.4.5. pg_control_recoveryコマンドによる監視

pg_control_recoveryは制御ファイルの情報をpg_controldataコマンドに替わり取得する方法です。取得できる項目は限られていますが、SELECT文で取得できるという特徴があります。状態監視の選択肢の一つとしてご認識下さい。
(スレーブでの実行例)

```
=# SELECT pg_control_recovery();
pg_control_recovery
-----
(0/E0EAFB50,5,0/0,0/0,f)
(1 行)
```

カンマ区切りにより5項目から構成されています。何れもpg_controldataコマンドでも取得できます。

表 4.15 pg_control_recovery関数の表示内容例

項目	マスタ	スレーブ
min_recovery_end_location	0/0	0/E0EAFB50
min_recovery_end_timeline	0	5
backup_start_location	0/0	0/0
backup_end_location	0/0	0/0
end_of_backup_record_required	f	f

min_recovery_end_locationおよび min_recovery_end_timelineにマスターとスレーブの違いが表れます。

4.3.4.6. pg_is_in_recovery関数による監視

リカバリ中かどうかを示します。マスターであれば false 、スレーブであれば true を表示します。

(スレーブでの実行例)

```
=# SELECT pg_is_in_recovery();
pg_is_in_recovery
-----
t
(1 行)
```

両ノードともfalse状態でない事を確認します。

4.4. SR環境の障害時運用

4.4.1. 本文書における用語の定義

4.4.1.1. フェイルオーバー

マスタへの障害発生時やアクセス不能時に、スレーブをマスタへ昇格させる処理を指します。新マスタはシングル構成であり、レプリケーションされていない片系運用の状態です。

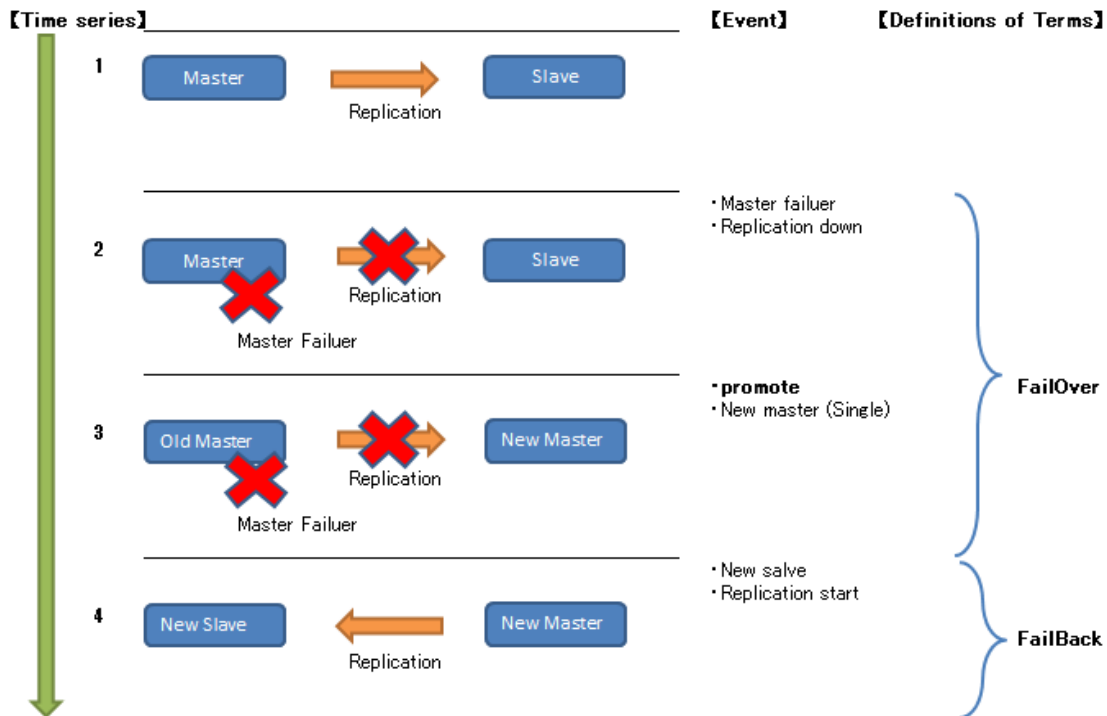
レプリケーション構成への復旧には後述の「フェイルバック」を実行しますが、マスタの障害が物理的なものでない場合は、「スイッチオーバー」が可能な場合もあります。

4.4.1.2. フェイルバック

フェイルオーバーにより片系運用になった後、再度レプリケーション構成に復帰させる処理を指します。
 シングル構成となった新マスタに、新スレーブを追加します。
 新マスタと新スレーブを交代させてフェイルオーバー前に戻すことまでは含まない事とします。

複数スレーブ構成では、不要な処理となります。

※Image of failover / failback



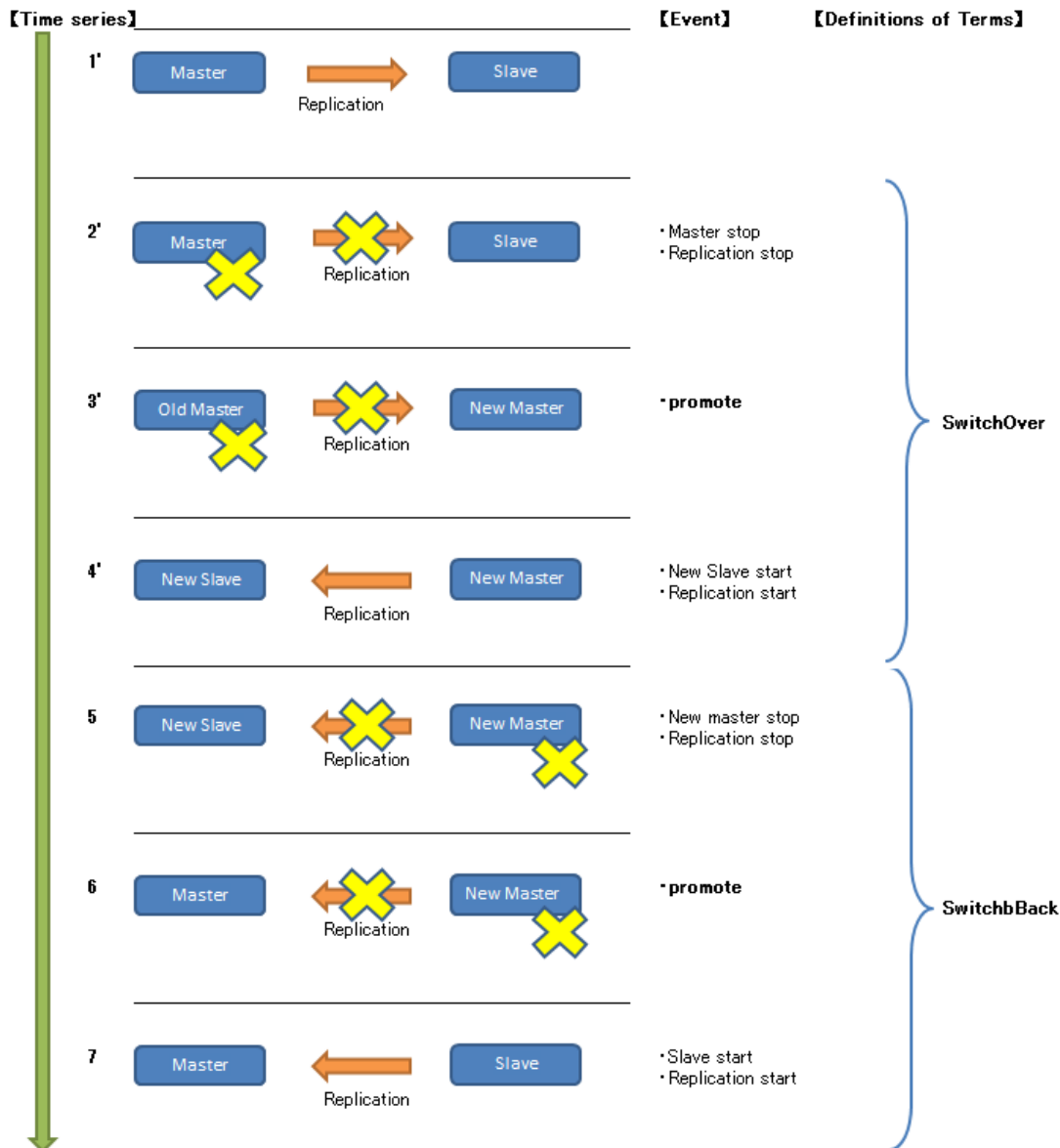
4.4.1.3. スイッチオーバー

マスタとスレーブを入れ替える処理を指します。
 マスタを停止し、スレーブを新マスタへ昇格させ旧マスタを新スレーブとして再追加します。
 マスタのメンテナンスのための一時的な入れ替えなどが目的です。
 フェイルオーバーとの違いは、マスタ停止が障害による停止か計画停止かであり、スレーブに対する処理は同様です。

4.4.1.4. スイッチバック

スイッチオーバー後に、マスタ/スレーブを入れ替えることを指します。
 マスタ/スレーブとの関係性を除き処理内容は、「スイッチオーバー」と同様です。

※Image of switchover / switchback



4.4.2. 障害時運用手順

■前提 以降の手順では次の前提とします。

- PostgreSQL 9.6
- マスタ、スレーブの2台構成(ホスト名をそれぞれ server1,server2と表記する)
- 同期モードは同期(synchronous_commit = on または remote_apply)
- スレーブはホットスタンバイ機能により参照可能(hot_standby = on)
- レプリケーション用のユーザは rep_user
- マスタ/スレーブとも、portは5432を使用
- レプリケーションスロット使用
- マスタ/スレーブとも、環境変数PGDATA,PGPORTは設定済み
- 死活監視は実際の運用ではクラスソフトを使用するのが一般的ですが、ここでは便宜上手動で実施
- サーバのNICはパブリックのみ
- 仮想IPについては考慮しない

■対処一覧 大別すると3種類の対処方法が考えられます。

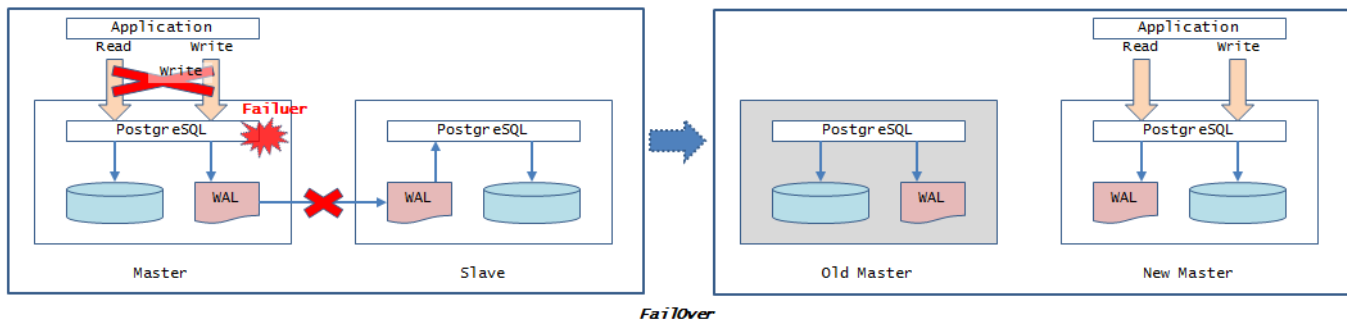
表 4.16 障害別の対処

ID	障害箇所	障害状況	pg_basebackupとpg_rewindの使い分け
1	マスタ	マスタとスレーブの関係が崩れており再構成が必要 - 物理障害 - promoteを伴わないスレーブのマスタ化(recovery.conf削除)	pg_basebackupコマンドを使用してフェイルバック
2	マスタ	マスタとスレーブの関係は巻き戻しで復旧可能 - 非同期レプリケーションのインスタンス障害(差異がある状態で昇格) - スプリットブレイン状態での旧マスタへの更新	pg_rewindコマンドを使用してスイッチバック
3	マスタ	マスタとスレーブの切り替え可能 - 計画停止 - 同期レプリケーションのインスタンス障害(物理障害なし)	pg_rewindコマンドを使用しないでスイッチバック
4	スレーブ	マスタとスレーブの連携再開可能 スレーブの障害	同期式の場合は非同期式に切り替え

4.4.2.1. フェイルオーバー

フェイルオーバーについて記載します。

マスタにて、障害が発生した場合のスレーブを新マスタへ昇格したシングル構成図



(1) マスタの疑似障害発生

immediateオプションにて停止、またはpostgresプロセスのkillする事で疑似障害を発生させます。
マスタにて実行します。

```
$ pg_ctl -w -m immediate stop
```

```
$ kill -9 `head -1 $PGDATA/postmaster.pid`
```

(2) マスタの死活監視にて異常を検知

死活監視の方法には多数ありますが、ここではpg_isreadyコマンドを使用します。
pg_isreadyコマンドはクライアントツールですので、任意のノードから実行できます。

```
$ pg_isready -h server2 -U postgres -d postgres
server2:5432 - no response
```

(3) スレーブを新マスタへ昇格

旧スレーブで実行します。

```
$ pg_ctl promote
```

※サーバログに下記内容が記載されること

```
"database system is ready to accept connections"
```

ただしこの時点ではsynchronous_standby_namesパラメータに値が設定されているため、新マスタで更新処理ができない状態です。

(4) 新マスタを非同期に切り替え

`synchronous_standby_names` パラメータの設定を''に設定し、リロードで反映します。
新マスタで実行します。

```
$ vi $PGDATA/postgresql.conf

[編集前]
synchronous_standby_names = '*'

[編集後]
synchronous_standby_names = ''

$ pg_ctl reload
```

これで更新処理ができる状態になりました。

(5) 新マスタの死活監視にて正常を確認

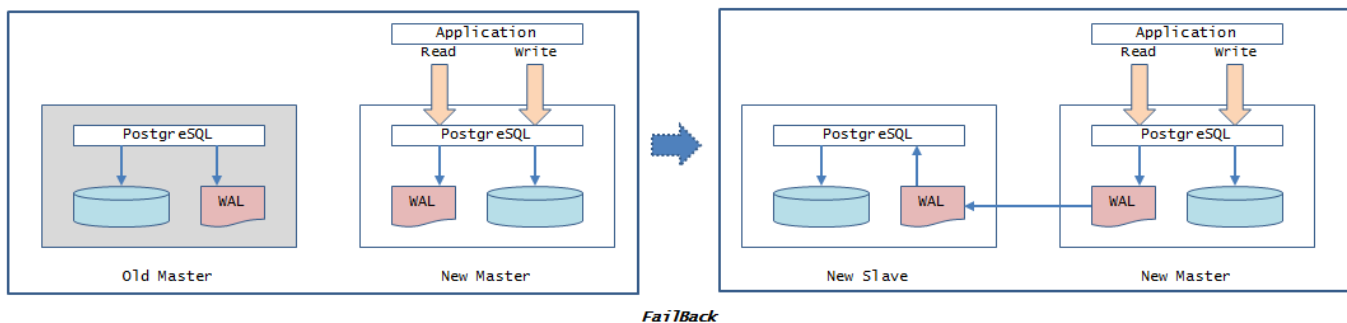
```
$ pg_isready -h server2 -U postgres -d postgres
server2:5432 - accepting connections
```

以上でファイルオーバーは完了です。

4.4.2.2. フェイルバック

`pg_basebackup` を使用したフェイルバックについて記載します。初期構築手順とほぼ同じです。

マスタ障害発生によるフェイルオーバー後、旧マスタを新スレーブとしたレプリケーション構成図



■パラメータ

`pg_basebackup` に必要な設定を記載します。

表 4.17 `pg_basebackup` に必要な設定

サーバ	設定ファイル	パラメータ	設定値	内容
マスタ	postgresql.conf	listen_address	0.0.0.0	全てのIPアドレス(v4)からの接続を受け付ける
マスタ	postgresql.conf	max_wal_senders	2	WALストリームオプションを付与する場合は、2以上を設定

■`pg_basebackup` コマンド

`pg_basebackup` コマンドの主なオプションは次の通りです。

表 4.18 pg_basebackup のオプション

オプション	内容
-D <directory>	出力を書き出すディレクトリを指定。
-X <method> --xlog-method=<method>	必要なWALファイルをバックアップに含める。method(収集方式)は以下から選択。 ● fetch : WALファイルは最後に収集 ● stream : バックアップ作成中に同時にWALをストリームで収集 運用中にpg_basebackupを実行する場合には stream を指定する。 fetch (最後に収集)では、必要なWALファイルが削除される可能性があるため。
-S <slot_name> --slot=<slot_name>	WALストリーミングの収集に指定したレプリケーションスロットを使用。 -X stream とセットで指定。 必要なWALファイルが削除されるのを防ぐ事を目的とする。 運用中にpg_basebackupを実行する場合に使用を検討する。 以下に注意する。 ● 事前にレプリケーションスロットを作成する必要がある ● マスタのWAL領域の空きが十分である事を確認する
-R --write-recovery-conf	最低限のrecovery.confを作成。必要に応じてrecovery.confを加筆修正。
-r <rate> --max-rate=<rate>	サーバから転送されるデータの最大転送速度を指定。 運用中にpg_basebackupを実行する場合に使用を検討する。 転送速度を抑える事で、マスタに対する影響を制限する事を目的とする。
-P	進行状況報告を有効化。 pg_basebackup 処理中におよその進行状況を報告する。 運用中に実行した場合、データベースクラスタのサイズが増加して進行状況が100%を超える場合がある。
-v	冗長モードを有効化。 進行状況報告も有効な場合、現在処理中のファイル名を出力。

pg_basebackupでレプリケーションスロットが使用できます。WAL収集方式に stream を指定する事でWALをほぼ確保できますが、スロットを指定する事でより確実になります。スロットを使用する運用であれば、この段階で作成するのが有力です。

■ファイルバック手順

pg_basebackupにスロットを指定する場合を記載します。

(1) 旧マスタのデータベースクラスタを削除
旧マスタにて実行します。

```
$ rm -rf $PGDATA/*
```

※\$PGDATA以外に表領域を作成している場合、そのファイルも削除します。
※\$PGDATAディレクトリを削除する場合は、postgresユーザで\$PGDATAディレクトリを作成できるよう
親ディレクトリのオーナーまたはパーミッションを設定します。

(2) レプリケーションスロットの作成
新マスタにて実行します。

```
$ psql

=# SELECT pg_create_physical_replication_slot('slot_server1',true); -- 第2パラメータにtrueを指定
pg_create_physical_replication_slot
-----
(slot_server1,144/EEFC8940)
(1 行)

=# \x on
拡張表示は on です。

=# SELECT * FROM pg_replication_slots ;
-[ RECORD 1 ]-----+-----
slot_name           | slot_server1
plugin              |
slot_type           | physical
datoid              |
database            |
active              | f              -- まだスロットは使用されてないため false
active_pid          |
xmin               |
catalog_xmin        |
restart_lsn         | 144/EEFC8940   -- trueの指定により、作成直後からrestat_lsnを認識
confirmed_flush_lsn |
```

(3) pg_basebackupコマンドにて、新マスタからデータベースクラスタをコピー
旧マスタにて実行します。

```
$ pg_basebackup -h server2 -U rep_user -D $PGDATA -X stream -S slot_server1 -P -v -R
transaction log start point: 144/F3000028 on timeline 17
pg_basebackup: starting background WAL receiver
10503022/10503022 kB (100%), 1/1 tablespace
transaction log end point: 144/F3000130
pg_basebackup: waiting for background process to finish streaming ...
pg_basebackup: base backup completed
```

(4) recovery.confの修正

旧マスタにて実行します。

pg_basebackupにて-Rオプションを指定した事で、recovery.confが作成されます。
スロットを指定している事から、primary_slot_nameの指定があります。以下を追記します。

- recovery_target_timeline/パラメータ
- primary_conninfoパラメータにapplication_nameを追加(任意/デフォルトはwalreceiver)

```
$ vi $PGDATA/recovery.conf

[編集前]
standby_mode = 'on'
primary_conninfo = 'user=rep_user host=server2 port=5432 sslmode=prefer sslcompression=1'
primary_slot_name = 'slot_server1'

[編集後]
standby_mode = 'on'
primary_conninfo = 'user=rep_user host=server2 port=5432 application_name=slave_server1
sslmode=prefer sslcompression=1'
primary_slot_name = 'slot_server1'
recovery_target_timeline = latest
```

(5) 旧マスタのpostgresql.confの修正

旧マスタにて、パラメータの調整を行います。

- synchronous_standby_namesパラメータの無効化
- pg_statsinfoの無効化(有効化されている場合)

同期レプリケーションの設定を無効化します。

新スレーブのshared_preload_librariesパラメータにpg_statsinfoが設定されている場合は、書き込みができずにエラーが発生します。
pg_statsinfoの指定を削除します。

```
$ vi $PGDATA/postgresql.conf

[編集前]
synchronous_standby_names = '*'
shared_preload_libraries = 'pg_stat_statements,pg_statsinfo'

[編集後]
synchronous_standby_names = ''
shared_preload_libraries = ''
```

(6) 新スレーブの起動
新スレーブを起動します。

```
$ pg_ctl start
```

(7) 新マスタでのレプリケーション確認

pg_stat_replicationsビューを参照して、レプリケーション構成であることを確認します。
pg_replication_slotsビューを参照して、スロットがアクティブであることを確認します。

```
$ psql

=# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]-----+-----
 pid          | 11262
 usesysid     | 16384
 username     | rep_user
 application_name | slave_server1
 client_addr   | <新スレーブIP>
 client_hostname |
 client_port   | 44548
 backend_start | 2017-03-22 06:36:06.362576+09
 backend_xmin  | 1781
 state        | streaming          --- ストリーミング中
 sent_location | 0/10000060
 write_location | 0/10000060
 flush_location | 0/10000060
 replay_location | 0/10000000
 sync_priority | 2
 sync_state    | async              --- 非同期

=# SELECT slot_name,active FROM pg_replication_slots;
-[ RECORD 1 ]-----+-----
 slot_name      | slot_server1
 active         | t              --- アクティブ
```

(8) レプリケーション方式を同期式に変更
新マスタにて、synchronous_standby_namesパラメータを設定し、リロードで反映します。

```
$ vi $PGDATA/postgresql.conf

[編集前]
synchronous_standby_names = ''

[編集後]
synchronous_standby_names = '*'

$ pg_ctl reload
```

```
$ psql -h server2 -U postgres postgres -c "SELECT * FROM pg_stat_replication" -x
-[ RECORD 1 ]-----+-----
pid              | 11262
usesysid         | 16384
username         | rep_user
application_name | s1
client_addr      | <新スレーブIP>
client_hostname  |
client_port      | 44548
backend_start    | 2017-03-22 06:36:06.362576+09
backend_xmin     | 1781
state            | streaming          --- ストリーミング中
sent_location    | 0/10000060
write_location   | 0/10000060
flush_location   | 0/10000060
replay_location  | 0/10000000
sync_priority    | 2
sync_state       | sync              --- 同期
```

これにて、以下の構成に復旧しました。

- レプリケーションスロット使用
- 同期式レプリケーション構成

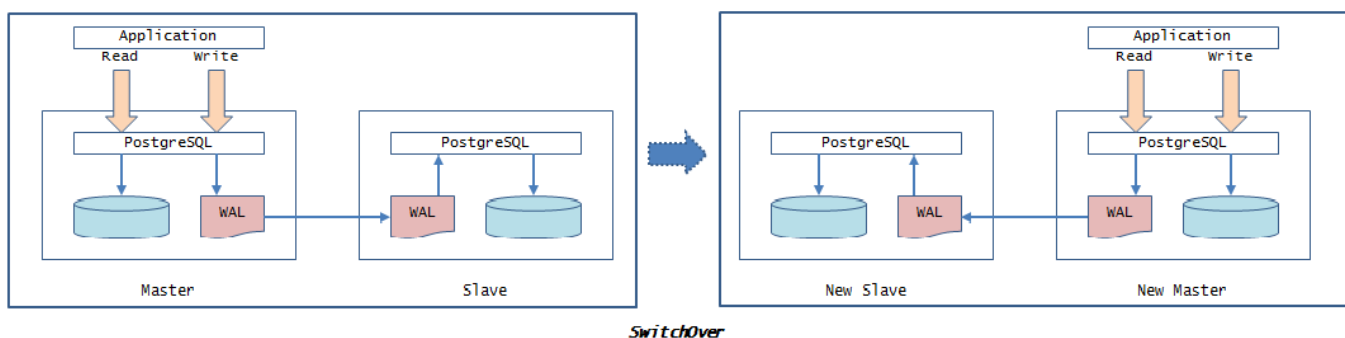
4.4.2.3. スイッチオーバー

スイッチオーバーについて記載します。

■スイッチオーバー手順

計画停止におけるマスタ/スレーブの切り替え手順です。pg_basebackupやpg_rewindが不要であるためシンプルな手順です。

計画停止にてマスタとスレーブの役割を切り替えたレプリケーション構成図



(1) マスタの正常停止
マスタにて実行します。

```
$ pg_ctl stop -m fast
```

(2) スレーブの昇格
スレーブにて実行します。

以下の場合にはpg_rewind不要です。

- 計画停止
- 同期レプリケーションのインスタンス障害(物理障害なし)

```
$ pg_ctl promote
```

以降の手順はフェイルオーバーの場合と同様であるため省略します。

- 3. レプリケーションスロットの作成
- 4. recovery.confの作成

- 5. 旧マスタのpostgresql.confの修正
- 6. 新スレーブの起動
- 7. 新マスタでのレプリケーション確認
- 8. レプリケーション方式を同期式に変更

(9) レプリケーションスロットの削除(スレーブ)

フェイルオーバーとの違いとしては、新スレーブに旧マスタ時代のスロットが残る事があります。
restart_lsn 列に値が残っている状態では、マスタのVACUUM 処理を阻害するなどの
悪影響の可能性があるため、削除します。
スロットの削除は関数で行うため、スレーブでも実行可能です。

```
$ psql

=# SELECT pg_drop_replication_slot('slot_server2');
pg_create_physical_replication_slot
-----
(1 row)

=# SELECT slot_name FROM pg_replication_slots ;
(0 rows)
```

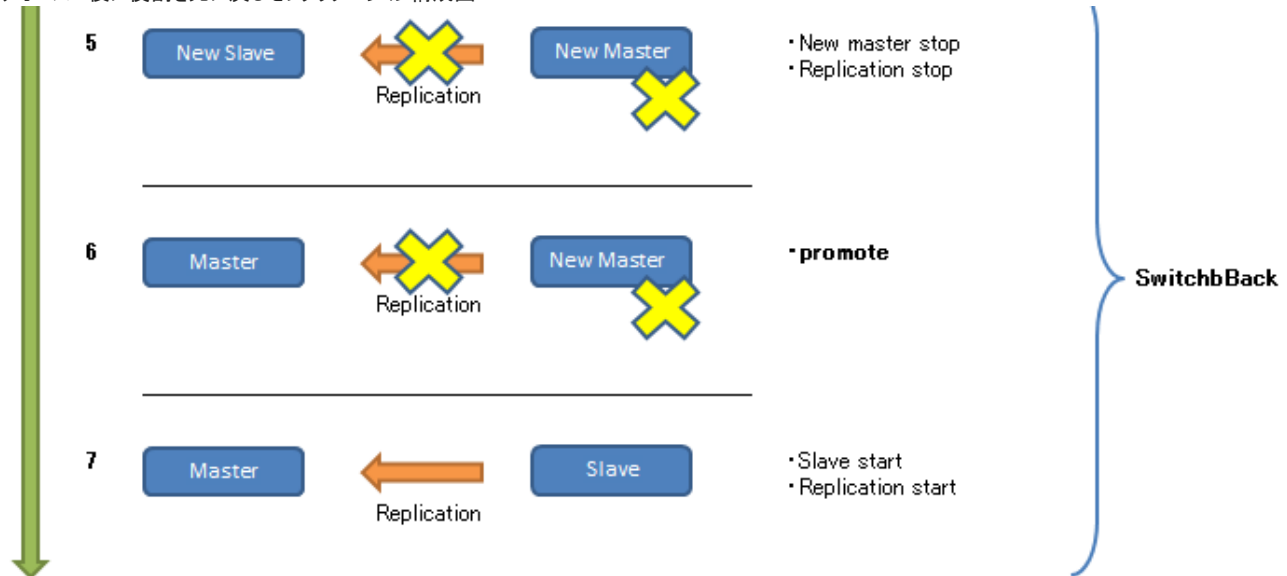
以上でスイッチオーバーが完了しました。

4.4.2.4. スイッチバック

pg_rewindを使用したスイッチバックについて記載します。

pg_rewindはタイムラインのずれたレプリケーションを再同期させる機能です。実行後、ターゲットクラスタはソースクラスタと置き換えられた状態になります。そのため
pg_rewind後の操作は、通常のフェイルオーバー時と同じです。タイムラインの分岐点からソースクラスタのWALを適用するため、更新量が少なければpg_basebackupによる
複製より高速です。これによりフェイルオーバー時、旧マスタを容易に新スレーブとして起動させることができます。

スイッチオーバー後に役割を元に戻したレプリケーション構成図



■関連パラメータ

pg_rewindに必要な設定を記載します。

表 4.19 pg_rewindに必要な設定

サーバ	設定ファイル	パラメータ	設定値	内容
マスタ	postgresql.conf	full_page_writes	on	チェックポイント後の更新時、ディスクページの全内容をWALに書き込む。
マスタ	postgresql.conf	wal_log_hints	on	ヒントビット更新時もfull_page_writesを実行する。

■pg_rewindコマンド

pg_rewindコマンドのの主なオプションは次の通りです。

表 4.20 pg_rewindのオプション

オプション	内容
D <ターゲットクラスタ>	pg_rewindを実行し、ソースクラスタの内容に置き換えるクラスタを指定する。
source-server="<ソースクラスタ>"	同期対象であるソースクラスタを指定します。主に次の接続文字列を使用します。 ● host: ソースクラスタのホスト名またはIPアドレス ● port: ソースクラスタのポート番号 ● dbname: ソースクラスタの接続先データベース名 ● user: ソースクラスタの接続先ユーザ
P	進行状況をレポートとして表示する。

■スイッチバック手順

※事前にマスタ/スレーブで(1) 関連パラメータの設定がされていることを前提とします。

(1) 旧マスタの正常停止

pg_rewindを使用するには正常停止する必要があります。

停止した旧マスタを一旦起動した後、正常停止させます。

障害により正常に起動や停止ができない状態ではpg_rewindは使用できません。

その場合は、pg_basebackupを使用します。

```
$ pg_ctl start -w
$ pg_ctl stop -m fast -w
```

(2) pg_rewindの実行

旧マスタでpg_rewindを実行する。

```
$ pg_rewind -D $PGDATA --source-server="host=server2 port=5432"
servers diverged at WAL position 0/5015B70 on timeline 1
rewinding from last common checkpoint at 0/5015B00 on timeline 1
Done!
```

(3) スレーブのrecovery.confを編集

旧マスタの\$PGDATA配下にrecovery.confを作成し、以下を設定します。

```
$ vi $PGDATA/recovery.conf

[編集後]
standby_mode = 'on'
primary_conninfo = 'host=server2 port=5432 user=rep_user'
recovery_target_timeline = 'latest'
```

(4) 新スレーブを起動し、新マスタとタイムラインIDが揃っていることを確認する。

タイムラインIDの取得には、pg_controldataコマンドを使用する。

厳密には最新チェックポイント実行時のタイムラインIDであるため、

タイムラインIDが揃っていないときは、マスタにてチェックポイント実行後、再確認します。

ssh経由で実行する事で任意のノードから全データベースクラスタの情報が取得できる。

[新マスタ]

```
$ export LANG=C
$ pg_controldata | grep " TimeLineID"
Latest checkpoint's TimeLineID:          2

$ ssh server1 $PGHOME/bin/pg_controldata $PGDATA | grep " TimeLineID"
Latest checkpoint's TimeLineID:          2
```

■pg_rewind使用時の注意点

● 正常停止が必要

pg_rewindを実行するデータベースクラスタは正常終了しなければいけません。
物理障害等により正常停止できない場合、pg_rewindは使用できません。pg_basebackupを使用します。

● 同一タイムラインの場合は実施不可

pg_promoteを実行せずに旧スレーブを新マスタにした場合、新マスタのタイムラインIDは変わらないため、
新マスタと旧マスタのタイムラインIDは同じ状態です。
この場合は、pg_rewindは実行できません。

● 実行時期と所要時間の関係

pg_rewindによるフェイルバックの所要時間は2つの要素から構成されます。

Step1. pg_rewindによる巻き戻し（旧マスタのWALを使用）

Step2. WAL適用による追い付き（新マスタのWALを使用）

新マスタで大量更新がある場合は、Step1は短時間で終了してもStep2で時間がかかります。

結果として、pg_basebackupの方が効率が良い場合もあり得ます。

また新マスタ昇格時のWALが削除されている場合は、後述するようにStep2でエラーとなる可能性もあります。その場合はpg_basebackupが必要となります。

pg_rewindはフェイルオーバー後、あまり時間を置かずに実行する事がポイントです。

● 旧マスタのWAL削除

旧マスタの巻き戻しに必要な旧マスタのWALが削除されている場合、pg_rewindは失敗します。

例えば旧マスタが障害により大量更新の途中で異常終了した場合などに発生します。

pg_rewind実行時に次のようなエラーが発生します。

```
could not open file "/home/pg96/pg96_data/pg_xlog/00000000D000000020000000CF": No
such file or directory
could not find previous WAL record at 2/CF000140
Failure, exiting
```

pg_rewindが成功するかどうかは検証(dry-runオプション)にて事前に確認する事ができます。

```
$ pg_rewind -D $PGDATA --source-server="host=server2 port=5432" --dry-run
servers diverged at WAL position 0/5015B70 on timeline 1
rewinding from last common checkpoint at 0/5015B00 on timeline 1
Done!
```

| メッセージはdry-runオプションが無い場合と同じです。
| pg_rewindでエラーが発生する場合（スレーブのWAL削除）は、この検証にて確認できますが、
| pg_rewindでエラーが発生しないで、後から発生する場合（マスタのWAL削除）は検知できません。
| 検証の仕様について認識下さい。

● 新マスタのWAL削除

pg_rewind後、新スレーブは新マスタのWALを適用することで、新マスタと同期します。

新マスタに昇格時のWALファイルが残っていない場合、新スレーブは追い付きができず、次のエラーがサーバログに出力され続けます。

対策としてレプリケーションスロットの有効化が有力ですが、新マスタのWAL領域の枯渇にご注意下さい。

```
ERROR: requested WAL segment 0000000D0000000000000000F3 has already been removed
```

- タイムラインの巻き戻し

pg_rewindはPostgreSQL9.6からタイムラインの巻き戻しができるよになっています。
これによりスプリットブレインが発生しても、新マスタをスレーブに戻すことが可能です。

- pg_rewindが不要な場合

pg_rewindはターゲットとソースクラスタのタイムラインIDが分岐した場合に実行が必要です。
そのためタイムラインが枝分かれしなかった場合、pg_rewindを実行する必要はありません。
例えばpg_rewind実行時に次のようなメッセージが出た場合、pg_rewindは実行せずに、以降の操作を継続します。

```
servers diverged at WAL position 0/503A428 on timeline 2  
no rewind required
```

4.4.2.5. スレーブ障害による対処

スレーブ障害時の緊急対応の必要性は、非同期モードが同期モードかによって異なります。

非同期モードの場合は、スレーブ障害がマスタの更新処理を阻害しないため、緊急対応は必要ありません。
とはいえ、シングル状態であるため早期にレプリケーション構成に復旧します。

同期モードの場合は、スレーブ障害によりマスタの更新処理が阻害されハング状態となります。

```
$ psql -h server1 -U postgres postgres -c "INSERT INTO test1_t VALUES ( 1 )"
Cancel request sent
WARNING: canceling wait for synchronous replication due to user request
DETAIL:  ** The transaction has already committed locally, but might not have been replicated to
the standby. **
INSERT 0 1
※Ctrl+Cをキーインする等、意図的にキャンセルしない限り、応答が返ってきません。
```

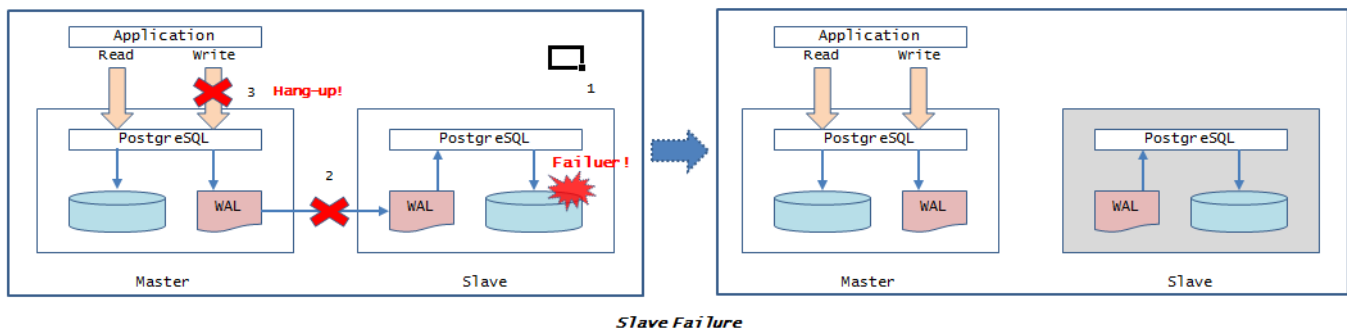
そのため、直ちに非同期に設定する必要があります。

非同期への切り替え処理は以降に示す様に再起動不要であるため、即時対応が可能です。

非同期への切り替えではなく、スレーブの再起動を試みる方法も考えられます。その方が効率的のようにも考えられます。
即座に起動できればその通りですが、起動に時間がかかる、あるいは物理的な障害で起動できない状態である事も考えられます。
そのような試行錯誤より、まずは確実にマスタのハング状態解消を優先します。

マスタの復旧後は、スレーブの復旧を試みます。

物理障害により起動できない場合は、フェイルバック処理と同様の作業を行います。



- (1) スレーブの疑似障害発生

immediateオプションにて停止、またはpostgresプロセスのkillする事で疑似障害を発生させます。
スレーブにて実行します。

```
$ pg_ctl -w -m immediate stop
```

- (2) スレーブの死活監視にて異常を検知

死活監視の方法には多数ありますが、ここではpg_isreadyコマンドを使用します。

pg_isreadyコマンドはクライアントツールですので、任意のノードから実行できます。

```
$ pg_isready -h server2 -U postgres -d postgres
server2:5432 - no response
```

(3) 非同期モードに切り替え

synchronous_standby_names/パラメータを''に設定する事で、非同期の設定となります。

```
$ vi $PGDATA/postgresql.conf

[変更前]
synchronous_standby_names = '*'

[変更後]
synchronous_standby_names = ''

$ pg_ctl reload
```

(4) 非同期モードの確認

レプリケーションモードが非同期(async)に変更された事を確認します。

```
$ psql -At -c "SELECT sync_state FROM pg_stat_replication;"
async
```

これでマスタが更新処理が可能な状態に復旧しました。ただしシングル状態であるため、フェイルオーバーと同様の作業を行います。

4.5. まとめ

4.5.1. ストリーミングレプリケーション機能拡張の歴史

PostgreSQL 9.0でストリーミングレプリケーション機能が実装されて以降、メジャーバージョン毎にストリーミングレプリケーション関連の新機能を実装しています。以下に経緯をまとめます。

- "ミスオペレーション"を"ミスオペ"と略記する場合があります。
- "レプリケーションスロット"と表記した場合は物理型を指します。論理型の場合は明記します。
- "レプリケーションスロット"を"スロット"と略記する場合があります。

表 4.21 ストリーミングレプリケーション機能拡張の歴史

バージョン	分野または目的	概要	関連パラメータ
9.1	管理性の向上 データ保護	同期モードの実装 COMMIT時にWALの転送完了までを保証する。 データ保護とパフォーマンスとのトレードオフ	[マスタのpostgresql.conf] synchronous_standby_names
9.1	管理性向上 昇格処理の明瞭化	pg_ctl promote コマンドによる昇格 明確なコマンドとなり分かり易くなった。 従来はtrigger_fileで設定したパスにtouchコマンド等でファイルを作成する方式であった。	
9.1	管理性向上 SR状況把握の簡易化	pg_stat_replicationビューの追加 マスタでの、スレーブへのWAL転送および適用の状況把握方法が簡易になった。wal_receiver_status_interval間隔で反映。	[スレーブのpostgresql.conf] wal_receiver_status_interval
9.1	管理性向上 SR状況把握の簡易化	pg_last_xact_replay_timestamp関数の追加 スレーブにて最終適用された時間を取得する	
9.2	データ保護とパフォーマンス の調整	同期方式にremote_writeの追加 synchronous_commitの選択肢として、on/off/localに加え、remote_writeが追加された。 スレーブのメモリに書き込むまでを保証する。その時点でOSがハングした場合、WALは損失する。	[マスタのpostgresql.conf] synchronous_commit

バージョン	分野または目的	概要	関連パラメータ
9.2	高可用性の向上	カスケードレプリケーションの実装 スレーブにぶら下がる2段目のレプリケーション構成が可能となった。 マスタの負荷増加を最小限とするレプリケーション強化。	
9.2	管理性の向上 SR状況把握の簡易化	pg_xlog_location_diff関数の追加 スレーブの転送や適用がどの程度遅れているかを取得するのが容易になった。 従来はpg_stat_replicationビューを参照していたが、ログの位置をバイト数に換算する計算が必要だった。 本関数により16進の差分をバイト数として取得できる。	
9.3	管理性の向上 役割交換の効率化	スイッチバックの実装 historyファイルの転送により実装	
9.3	管理性の向上	カスケードレプリケーションの管理性向上 カスケードレプリケーション構成において、スレーブの新マスタ昇格後に、スレーブとレプリケーション継続が可能に	
9.3	障害時間の短縮	昇格処理の高速化 昇格処理にて、リカバリのみ実行し、チェックポイントを省略する事で時間短縮を実現。 従来はチェックポイントも昇格処理にて実行していた。	
9.4	管理性の向上 ミスオペ対策	遅延レプリケーションの実装 スレーブでの適用を一定時間(recovery_min_apply_delay)遅らせる事で、マスタで発生したミスオペの伝搬を防ぐ。以下の事項に注意。 - 一定時間内にマスタでのミスオペを検知して対応が必要 - 昇格時には遅延分の適用が必要であるため時間を要する	[スレーブのpostgresql.conf] recovery_min_apply_delay
9.4	管理性の向上 WAL保持の保証	レプリケーションスロットの実装 スレーブに必要なWALをマスタで保持し続ける事を保証。 特に複数スレーブ構成にて効果的。以下の事項に注意。 - マスタのWAL領域が溢れないように監視を検討 - 不要スロットは削除(残存しているとWALを保持)	[マスタのpostgresql.conf] max_replication_slots [スレーブのpostgresql.conf] hot_standby_feedback
9.4	論理レプリケーション	論理レプリケーションの関数の実装 行レベルの変更内容出力する関数が実装された。 - pg_logical_slot_get_changes関数 - pg_logical_slot_peek_changes関数 必要な設定 - wal_levelをlogicalに設定 - 論理レプリケーションスロットを作成	[マスタのpostgresql.conf] wal_level max_replication_slot
9.5	WAL転送効率の向上	WAL圧縮機能の実装 WALを圧縮する事で、転送効率の向上する。 圧縮/解凍によるオーバーヘッドとのトレードオフだが、通常はメリットの方が大きい。 レプリケーションに特化した機能では無いが、特に次のレプリケーション構成での適用が効果的と考えられる。 - 同期モードまたは完全同期モード - スレーブを遠隔地に配置(ディザスタ・リカバリ)	[マスタのpostgresql.conf] wal_compression
9.5	管理性の向上 SR構築の効率化	pg_rewindコマンドの実装 昇格した新マスタと旧マスタを再同期する事で、レプリケーション構成を実装。 物理的な障害でない場合は、pg_rewindにて対応可能な可能性がある。 新マスタから旧マスタへ差分バックアップを転送および旧マスタのWALを適用。 差分バックアップの転送はpg_basebackupと同様であるため、同様の設定が必要。	[マスタのpostgresql.conf] wal_log_hints full_page_writes max_wal_senders [マスタのpg_hba.conf] replication 疑似データベースとの認証設定
9.5	管理性の向上	レプリケーション関連メッセージの出力 ログ監視の利便性が向上。ただしあまり多くは出力されない。	[マスタのpostgresql.conf] log_replication_commands
9.5	管理性の向上 継続的アーカイブ	スレーブでのアーカイブ出力機能 スレーブにおいて自分のWALを出力する事で、昇格時に途切れることなく継続的なアーカイブが可能。	[スレーブのpostgresql.conf] archive_mode
9.6	スケールアウト 参照負荷分散	完全同期レプリケーション機能の実装 スレーブへの適用完了までを保証。 スレーブ参照時にマスタと同一データが保証される事で、参照負荷分散によるスケールアウトが期待される。	[マスタのpostgresql.conf] synchronous_commit

バージョン	分野または目的	概要	関連パラメータ
9.6	データ保護の多重化	複数同期スレーブ構成機能 従来は複数スレーブの内、同期モードが設定できるのは1台のみであったが、その制限が無くなった。 同時に複数スレーブに対して同期モードが設定できる。	[マスタのpostgresql.conf] synchronous_standby_names
9.6	管理性の向上 SR状況把握の簡易化	pg_stat_wal_receiverビューの追加 従来はマスタにてpg_stat_replicationビューが参照できたが、スレーブにてpg_stat_wal_receiverビューが参照できる。 スレーブからレプリケーションの状況が把握が容易に。	
9.6	管理性の向上 監視手法の多様化	pg_control_recovery関数の追加 制御ファイル情報の内、リカバルに関する情報を取得。 従来はpg_controldataコマンドで制御ファイルの情報を取得したが、SELECT文で取得できる事で管理手法の選択肢が増えた。	
9.6	管理性の向上 役割交換の簡易化	pg_rewindの機能拡張 タイムラインIDの変更後にも対応できるようになった。 マスタに適用する事でスレーブに戻す事が可能。	

4.5.2. PGECsの今後の取り組み

ストリーミングレプリケーションは、以下に示す複数のテーマを包含しています。

- データ保護
- 高可用性
- 参照負荷分散

PGECs は今後も各WGにて、さまざまな観点から PostgreSQL の調査・検証を続けていきます。次期メジャーバージョンの PostgreSQL 10 でもレプリケーション機能の拡充が予定されています。今後もストリーミングレプリケーションを重要分野と位置づけ、新機能の検証を継続していきたいと考えています。

5. Bi-Directional Replication (BDR)

5.1. はじめに

5.1.1. BDRの特徴

5.1.1.1. BDRとは

BDR(Bi-Directional Replication)は2ndQuadrant社によって開発された、オープンソース(PostgreSQL License)のマルチマスタ・レプリケーションシステムです。双方向の非同期論理レプリケーションを使用し、地理的に分散したクラスタで使用するために設計されています。

マルチマスタについて

一般的なRDBMSの冗長化構成(マスタスレーブ構成)においては、更新処理を実行可能なサーバをマスタと呼称します。マルチマスタとは、同一のデータを保持している複数のDB間(クラスタ)において、更新処理を実施可能なサーバが複数台存在する構成を指します。BDRでは双方向にレプリケーションを実施することで、複数のサーバへの更新を可能にしています。

5.1.1.2. ユースケース

BDRは以下のようなケースで有用です。

1. 遠隔地や高レイテンシ環境でクラスタを構成している場合。
2. 各ノードが書き込み処理を実施する場合。
3. クラスタ間でデータが非同期であることを許容できる場合。

例としては以下のようにレスポンスを向上させるために各地でアプリケーション及びDBを動作させるようなケースが考えられます。

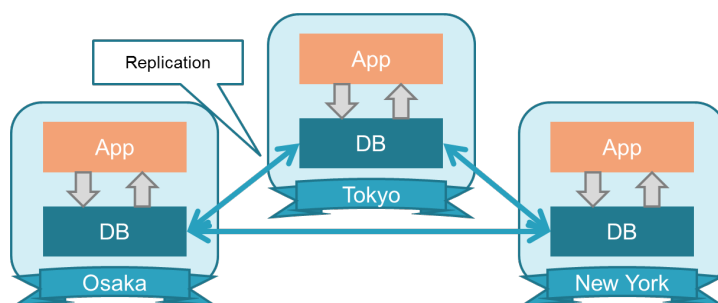


図 5.1 BDRユースケース

5.1.1.3. メカニズム

BDRでは「Logical Decoding」により、WALから論理的な変更点を抽出し、各ノードで適用することで双方向レプリケーションを実現しています。(Logical Decoding及びWALの送受信はバックグラウンドワーカプロセスが実施します)

従来のトリガーを用いた双方向レプリケーションの場合、下図のように書き込みが余分に発生(変更記録、変更反映)してしまいます。

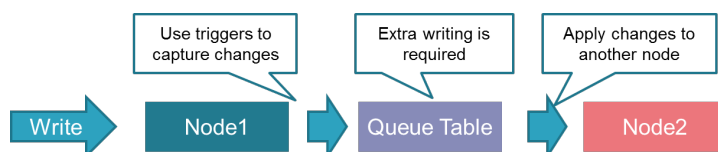


図 5.2 トリガベースメカニズム

一方、BDRでは余分な書き込みが発生せずパフォーマンス的に有利となっています。

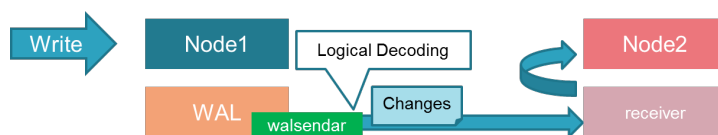


図 5.3 BDRメカニズム

利用されているPostgreSQLのメカニズムの一覧です。

表 5.1 利用メカニズムの一覧

No.	機能	概要
1	Event Triggers	一つのテーブルに接続され、DMLイベントのみを補足する通常のトリガとは異なり、特定データベースのDDLイベントを捕捉可能。
2	Logical Decoding	SQLを介して実行された更新処理を外部コンシューマへストリーミングするための機能。更新結果はロジカルレプリケーションスロットで識別され、ストリームに送出される。
3	Replication Slots	マスタのデータベースの変更をスレーブ側に同じ順序で適用するための機能。
4	Background Workers	ユーザ提供のコードを別プロセスで実行するように拡張する機能。Background Workerプロセスはpostgresプロセスによって起動、終了、監視される。
5	Commit Timestamps	トランザクションがいつコミットされたかを確認するための機能。
6	Replication Origins	レプリケーションの進行状況を追跡するための機能。双方向レプリケーションにてループの防止などが可能。
7	DDL event capture	実行されるDDLコマンドを返す機能。
8	generic WAL messages for logical decoding	テキストあるいはバイナリのメッセージをWALに挿入できる仕組み（API）。Logical Decoding機能によって読み出されることを想定している。

5.1.1.4. 整合性

マルチマスタ構成を取る場合、各ノードが持つ情報に不整合が発生しないように管理する仕組みが必要となります。BDRでは結果整合性(eventually consistent)と呼ばれる一貫性モデルを採用し、整合性を確保します。

- デフォルトでは競合が発生した場合、最後の更新処理が適用されます。(last_update_wins)
- 競合結果はテーブル「[bdr.bdr_conflict_history](#)」で確認可能です。

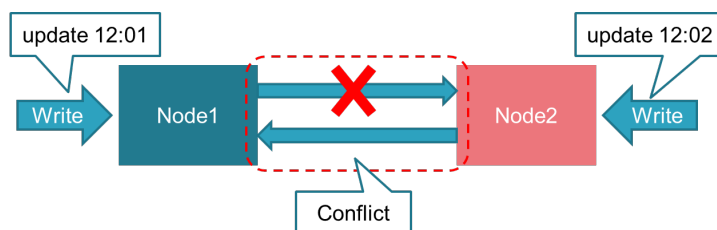


図 5.4 結果整合性

5.1.1.5. シーケンス

BDRでは一意な値を払い出すために、「グローバルシーケンス」、「ステップ/オフセットシーケンス」という2つの手法を紹介しています。

1. グローバルシーケンス

グローバルシーケンスでは、各ノードに予め値の塊(chunk)を一定数ずつ割り振ることで値の重複を回避しています。

- chunkを消費するとvoting処理(下記参照)を行い、新たにchunkを割り振ります。
- グローバルシーケンスは廃止予定です。(下記のステップ/オフセットシーケンスを推奨)

voting 処理

"chunk"と呼ばれるシーケンス番号のまとまりをノードに割り当てる処理をvoting処理と呼びます。"chunk"が複数のノードに割り当てられないことを確認するため、ノード間で投票処理が行われており、正常に機能させるためには奇数台のノードが必要です。過半数のノードが停止している場合は、投票処理にて過半数に到達しなくなるため、新しい"chunk"がノードに割り当てられません。そして、"chunk"が枯渇した場合、nextvalの実行に失敗してしまいます。

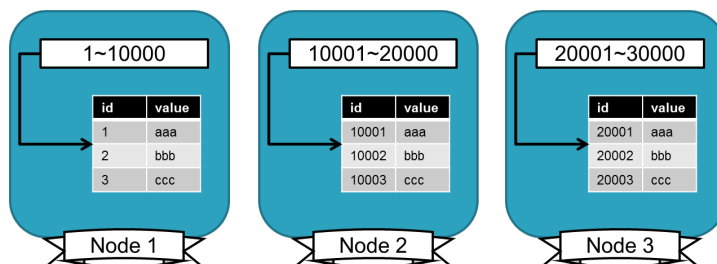


図 5.5 グローバルシーケンス

2. ステップ/オフセットシーケンス

ステップ/オフセットシーケンスでは、各ノードで通常のPostgreSQLシーケンスを使用します。各シーケンスを同じ量だけ増分されるようにし、値が重複しないように設定します。

- 設計時には注意が必要です。
- 増分する値をある程度大きく取らないとノードの追加に対応できません。(例: 10の増分であればノード数の追加は10台まで)

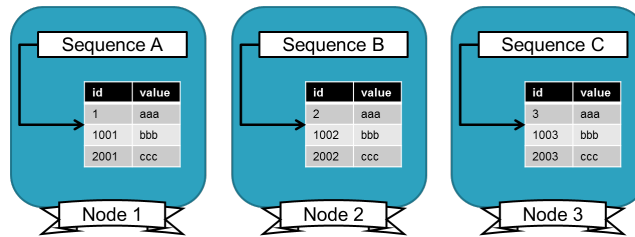


図 5.6 ステップ/オフセットシーケンス

5.1.1.6. 比較表

類似機能及び製品との机上比較の結果です。

表 5.2 類似機能及び製品との机上比較

No.	比較項目	BDR	SR(Hot Standby)	Slony
1	マルチマスタ	○	×	×
2	選択的レプリケーション	○	×	○
3	競合検知	○	×	×
4	カスケードイング	×	○	○
5	WALベースレプリケーション	○	○	×
6	DDLレプリケーション	○	○	×
7	自動レプリカ新テーブル	○	○	×
8	シーケンスレプリケーション	○	○	○
9	プライマリキー更新	○	○	×
10	同期コミット	○	○	×
11	外部デーモン不使用	○	○	×
12	レプリカへの書き込み	○	×	○

<https://2ndquadrant.com/en/resources/bdr/>

5.1.1.7. サポート

サポートについては以下が存在します。

表 5.3 サポート

No.	サポート	概要
1	無償サポート	BDRコミュニティへのメール、BDRのGoogleグループが存在。 email: bdr-list@2ndquadrant.com Googleグループ: https://groups.google.com/a/2ndquadrant.com/forum/#!forum/bdr-list
2	有償サポート	2ndQuadrant社によるサポートを受けることが可能です。 ※ 2ndQuadrant社について BDRの製造元で、PostgreSQLの専門家(コミッター等)が多数在籍する企業です。 PostgreSQLのコンサルティングサービス等を提供しています。 https://2ndquadrant.com/en/

5.2. 調査、検証の目的

机上の情報整理および検証について、以下を主な目的としています。

- 情報整理
 - BDRを使用したマルチマスタ環境を構築、運用するにあたり、各種機能やパラメータの理解に必要な情報の提供
- 検証
 - BDRの動作(競合時等)や障害発生時の対処法を調査
 - 更新性能の検証

5.3. 調査、検証の前提

表 5.4 調査、検証の前提

項目	説明
PostgreSQLバージョン	9.4.10 ※ 調査時点ではBDRは9.6に対応していなかったため
BDRバージョン	1.0.2 ソースコード: https://github.com/2ndQuadrant/bdr
OS	CentOS 7.1
構成	2ノード構成

ダウンロードモジュールは以下になります。

- postgresql-bdr94-bdr.x86_64 0:1.0.2-1_2ndQuadrant.el7.centos
- postgresql-bdr94.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos
- postgresql-bdr94-contrib.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos
- postgresql-bdr94-libs.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos
- postgresql-bdr94-server.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos

以下は構成図になります。

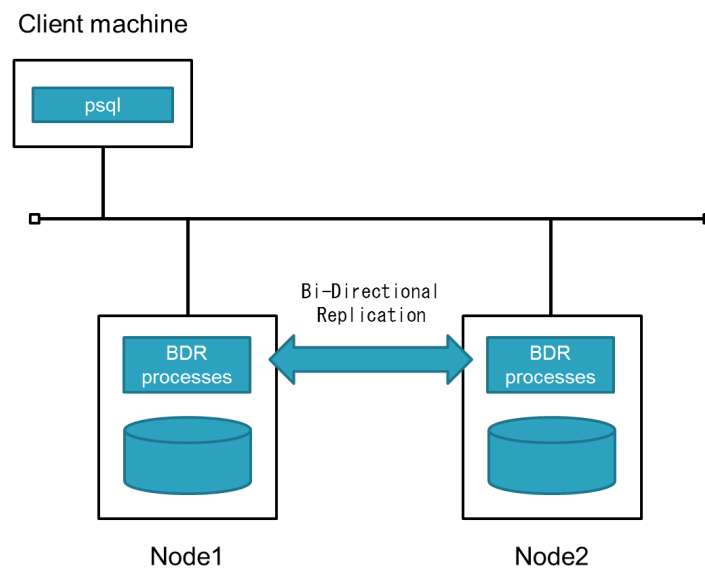


図 5.7 調査、検証環境構成図

5.4. BDR環境構築時の設定と設定手順

5.4.1. BDR環境の設定手順

5.4.1.1. 概要

PostgreSQL BDR (Bi-Directional Replication)を利用したマルチマスタ環境の構築手順について確認します。

また、マルチマスタ環境構築後、各ノードに対して更新が実行可能かを確認します。

<http://bdr-project.org/docs/next/installation-packages.html#INSTALLATION-PACKAGES-REDHAT>

[こちらの環境](#) で検証を実施しました。

5.4.1.2. 対象試験サーバ

表 5.5 対象試験サーバ

サーバ名	IPアドレス
node1	192.168.0.10
node2	192.168.0.12

5.4.1.3. 事前確認

1. yumが利用可能な状態であること

- パッケージを入手する際にyumを使用

設定手順

指定がない部分は、node1・node2両方で実施します。

1. 検証環境準備

BDR検証を実施するための環境を準備します。

```
ssh接続を用いて、該当環境へ接続
ユーザ: root
パスワード: xxxxxxxx
```

2. hostsの設定

DNSの名前解決のために各サーバのホスト名を設定します。

```
# vi /etc/hosts
[下記をファイル末尾に追加]
192.168.0.10 node1
192.168.0.12 node2
```

3. 関連ポートの開放

PostgreSQL間のBDR接続のために5432ポートの開放をします。

```
# firewall-cmd --permanent --add-port=5432/tcp
# firewall-cmd --reload
```

以下のコマンドで確認します。

```
# firewall-cmd --list-ports
5432/tcp
```

4. BDRレポジトリの登録

BDRレポジトリ用のRPMをインストールします。

```
# RHEL/CentOS users only:
# yum install http://packages.2ndquadrant.com/postgresql-bdr94-2ndquadrant/yum-repo-
rpms/postgresql-bdr94-2ndquadrant-redhat-latest.noarch.rpm
```

以下のコマンドでインストール済み一覧を確認します。

```
# yum list installed postgresql-bdr94-2ndquadrant-redhat.noarch 0:1.0-3
[省略]
インストール済みパッケージ
postgresql-bdr94-2ndquadrant-redhat.noarch          1.0-3          installed
```

5. BDRインストール

登録したBDRレポジトリからBDRをインストールします。

```
# yum install postgresql-bdr94-bdr
[省略]
インストール:
  postgresql-bdr94-bdr.x86_64 0:1.0.2-1_2ndQuadrant.el7.centos

依存性関連をインストールしました:
  postgresql-bdr94.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos
  postgresql-bdr94-contrib.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos
  postgresql-bdr94-libs.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos
  postgresql-bdr94-server.x86_64 0:9.4.10_bdr1-1_2ndQuadrant.el7.centos

完了しました！
```

以下のコマンドでインストール済み一覧を確認します。

```
# yum list installed postgresql-bdr94-bdr.x86_64 0:1.0.2-1_2ndQuadrant.el7.centos
[省略]
インストール済みパッケージ
  postgresql-bdr94-bdr.x86_64 1.0.2-1_2ndQuadrant.el7.centos
@postgresql-bdr94-2ndquadrant-redhat
```

6. 環境変数設定

データベースクラスタとコマンド実行ファイルに環境変数の設定をします。

```
$ vi ~/.bash_profile
[下記の修正を加える]
#PGDATA=/var/lib/pgsql/9.4-bdr/data
PGDATA=/var/lib/pgsql/2ndquadrant_bdr/data/
[下記をファイル末尾に追加]
export PATH=/usr/pgsql-9.4/bin:$PATH
```

以下のコマンドで設定した環境変数を確認します。

```
$ exit
ログアウト
# su - postgres
$ echo $PATH
/usr/pgsql-9.4/bin:/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin
$ echo $PGDATA
/var/lib/pgsql/2ndquadrant_bdr/data/
```

7. データベースクラスタ作成

BDR検証用に新規でデータベースクラスタを作成します。

```
$ mkdir -p $HOME/2ndquadrant_bdr/
$ initdb -D $HOME/2ndquadrant_bdr/data -A trust -U postgres --no-locale
```

8. BDRのパラメータ設定(postgresql.conf)

BDRのパラメータ設定(postgresql.conf)をします。

```
$ vi /var/lib/pgsql/2ndquadrant_bdr/data/postgresql.conf
[下記の修正を加える]
listen_addresses = '*'
shared_preload_libraries = 'bdr'
wal_level = 'logical'
track_commit_timestamp = on
max_connections = 100
max_wal_senders = 10
max_replication_slots = 10
# Make sure there are enough background worker slots for BDR to run
max_worker_processes = 10
log_line_prefix = '%m d=%d p=%p a=%a%q '

# These aren't required, but are useful for diagnosing problems
#log_error_verbosity = verbose
#log_min_messages = debug1
# Useful options for playing with conflicts
#bdr.default_apply_delay=2000 # milliseconds
#bdr.log_conflicts_to_table=on
```

9. BDRのクライアント認証設定(pg_hba.conf)

BDRのクライアント認証設定(pg_hba.conf)をします。

```
$ vi /var/lib/pgsql/2ndquadrant_bdr/data/pg_hba.conf
[下記の修正を加える]
host      all             all             192.168.0.10/32      trust
host      all             all             192.168.0.12/32      trust
local     replication     postgres        trust
host      replication     postgres        0.0.0.0/0            trust
host      replication     postgres        ::1/128              trust
```

10. PostgreSQL起動

設定が環境しましたらPostgreSQLを起動します。

起動時にBDRのバックグラウンドワーカーの登録メッセージが出力される事を確認します。

```
$ pg_ctl start
サーバは起動中です。
-bash-4.2$ < 2016-10-19 01:47:01.891 JST >LOG:   バックグラウンドワーカー"bdr supervisor"を登録しています
< 2016-10-19 01:47:02.041 JST >LOG:   ログ出力をログ収集プロセスにリダイレクトしています
< 2016-10-19 01:47:02.041 JST >ヒント:   ここからのログ出力はディレクトリ"pg_log"に現れます。
```

以下のコマンドでPostgreSQLの起動状況を確認します。

```
$ pg_ctl status
pg_ctl: サーバが動作中です (PID: 10948)
/usr/pgsql-9.4/bin/postgres
```

11. データベース作成

BDRの動作確認用のデータベースを作成します。

```
$ createdb bdrtest
```

以下のコマンドで作成したデータベースへの接続を確認します。

```
$ psql bdrtest
psql (9.4.9)
"help" でヘルプを表示します。

bdrtest=# \q
-bash-4.2$
```

12. BDRモジュール登録

BDRに必要な拡張モジュールを登録します。

```
$ psql -U postgres bdrtest

=# CREATE EXTENSION btree_gist;
=# CREATE EXTENSION bdr;
```

以下のコマンドでインストール済みの拡張モジュールを確認します。

```
=# \dx
                                インストール済みの拡張の一覧
 名前      | バージョン | スキーマ | 説明
-----+-----+-----+-----
 bdr        | 1.0.1.0    | pg_catalog | Bi-directional replication for PostgreSQL
 btree_gist | 1.0        | public   | support for indexing common datatypes in GiST
 plpgsql    | 1.0        | pg_catalog | PL/pgSQL procedural language
(3 行)
```

13. BDRグループ作成

node1でBDRグループを作成するため、「[bdr.bdr_group_create](#)」関数を実行します。

```
$ psql -U postgres bdrtest
=# SELECT bdr.bdr_group_create(
    local_node_name := 'node1',
    node_external_dsn := 'host=node1 port=5432 dbname=bdrtest');
```

グループが作成されたことを確認するため、「[bdr.bdr_node_join_wait_for_ready](#)」関数を実行します。

```
=# SELECT bdr.bdr_node_join_wait_for_ready();
 bdr_node_join_wait_for_ready
-----
(1 行)
```

14. ノード登録

node2でノード登録するために、「[bdr.bdr_group_join](#)」関数を実行します。

```
$ psql -U postgres bdrtest
=# SELECT bdr.bdr_group_join(
    local_node_name := 'node2',
    node_external_dsn := 'host=node2 port=5432 dbname=bdrtest',
    join_using_dsn := 'host=node1 port=5432 dbname=bdrtest'
);
```

ノードが登録されたことを確認するため、「[bdr.bdr_node_join_wait_for_ready](#)」関数を実行します。

```
=# SELECT bdr.bdr_node_join_wait_for_ready();
    bdr_node_join_wait_for_ready
-----
(1 行)
```

15. ノード状態の確認

ノードの状態を「[bdr.bdr_nodes](#)」テーブルの情報から確認します。

```
$ psql -U postgres bdrtest
=# \x
=# SELECT * FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_sysid          | 6369044607312522907
node_timeline       | 1
node_dboid          | 16385
node_status         | r
node_name           | node1
node_local_dsn      | host=node1 port=5432 dbname=bdrtest
node_init_from_dsn  |
node_read_only      | f
-[ RECORD 2 ]-----+-----
node_sysid          | 6369045470046586386
node_timeline       | 1
node_dboid          | 16385
node_status         | r
node_name           | node2
node_local_dsn      | host=node2 port=5432 dbname=bdrtest
node_init_from_dsn  | host=node1 port=5432 dbname=bdrtest
node_read_only      | f
```

16. 簡易動作検証(node1)

node1でBDRの簡易動作検証を実施します。

node1で作成されたテーブルとデータがnode2にレプリケーションされていることを確認します。

```
$ psql -U postgres bdrtest
=# CREATE TABLE t1bdr (c1 INT, PRIMARY KEY (c1));
=# INSERT INTO t1bdr VALUES (1);
=# INSERT INTO t1bdr VALUES (2);
```

17. 簡易動作検証(node2)

node2でBDRの簡易動作検証の結果を確認します。

node1と同じテーブルとデータが表示できていれば、構築したBDR環境に問題はありません。

node1と同様の簡易動作検証をnode2からも実施して下さい。

```
$ psql -U postgres bdrtest
=# SELECT * FROM t1bdr;
 c1
----
 1
 2
(2 行)
=# INSERT INTO t1bdr VALUES (3);
```

5.5. BDR動作検証

5.5.1. ノードの追加・切り離し

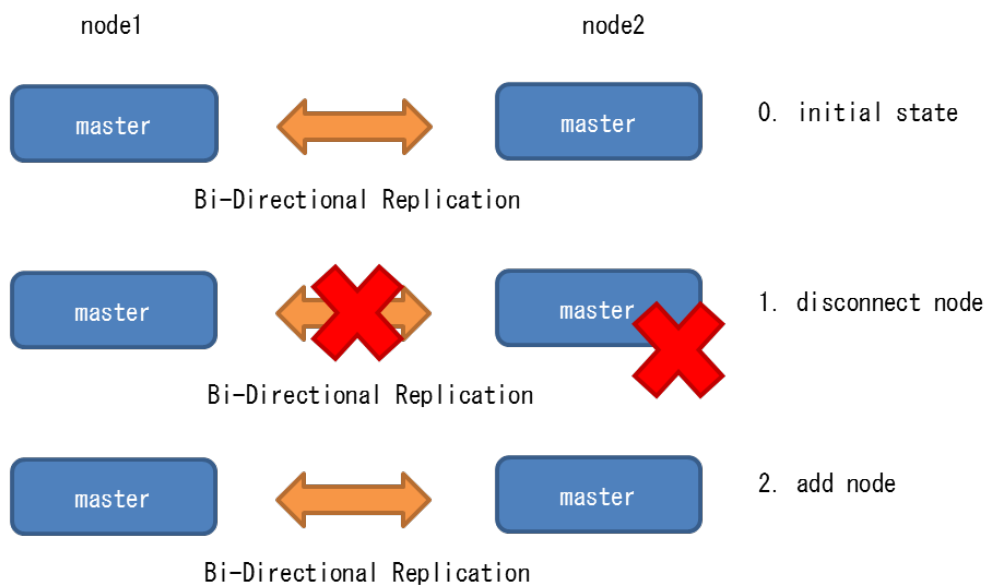
5.5.1.1. 検証の目的

ノードの追加/切り離しをオンライン(DB停止)なしで実行できるか否かを確認します。

5.5.1.2. 検証内容

本検証では、2台で構成されたBDRクラスタに対して、下記を実施しました。

1. ノード切り離し
2. ノード追加



「ノード切り離し」および「ノード追加」時に他ノード(上図のnode1)にトランザクションを実行し、トランザクションにエラーが発生するか否かを確認しました。

5.5.1.3. 検証環境

[こちらの環境](#) で検証を実施しました。

5.5.1.4. 検証手順

5.5.1.4.1. ノード切り離し

1. BDRの各ノード状態確認

BDRの各ノード間のレプリケーションが正常に動作していることを「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r                                     <-- r (正常) であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | r                                     <-- r (正常) であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest <-- node1から作成されていることを確認
```

2. 検証用テーブルとデータの作成

pgbenchを利用して、検証時に利用するテーブルとデータを作成します。

```
(node1にて実施)
$ pgbench -i -s 10 bdrtest
```

3. トランザクション実行

pgbenchを利用して、ノード1にトランザクションを継続的に実行します。ノード2を切り離した発生した場合に、ノード1に対して実行したトランザクションにエラーが発生するか否かを確認します。

```
$ pgbench -h node1 -c 10 -t 100000 bdrtest
starting vacuum...end.
```

4. ノード切り離し

ノード2を切り離すため、「[bdr.bdr_part_by_node_names](#)」関数を実行します。

```
(node1のデータベースに接続)
=# SELECT bdr.bdr_part_by_node_names(ARRAY['node2']);
 bdr_part_by_node_names
-----
(1 row)
```

5. ノードの切り離し結果確認

ノードの切り離し結果を「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | k                                     <-- k (削除) であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest
```

6. システムカタログの残データ削除

本手順は推奨される手順ではありませんが、削除状態のノードが「[bdr.bdr_nodes](#)」テーブルに残っていると削除したノードの追加が実行できないため暫定的な対処です。BDRのIssuesで「[bdr.bdr_connections](#)」のデータも削除する事が提案されていたので、こちらも暫定的な手順ですが実行します。詳細は下記をご参照下さい。

- <https://github.com/2ndQuadrant/bdr/issues/97>
- <https://github.com/2ndQuadrant/bdr/issues/126>

```
(node1のデータベースに接続)
=# DELETE FROM bdr.bdr_connections USING bdr.bdr_nodes WHERE node_status = 'k' AND
(node_sysid, node_timeline, node_dboid) = (conn_sysid, conn_timeline, conn_dboid);
=# DELETE FROM bdr.bdr_nodes where node_status = 'k';
```

7. ノードの切り離し結果確認

ノードの切り離し結果を「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r
node_init_from_dsn |
```

8. BDRの無効化

ノード2のデータベースから「[bdr.remove_bdr_from_local_node](#)」関数を用いてBDRを削除し、BDR拡張機能を削除します。

```
(node2のデータベースに接続)
=# SELECT bdr.remove_bdr_from_local_node(true);
WARNING:  forcing deletion of possibly active BDR node
NOTICE:  removing BDR from node
NOTICE:  BDR removed from this node. You can now DROP EXTENSION bdr and,
if this is the last BDR node on this PostgreSQL instance, remove bdr from
shared_preload_libraries.
remove_bdr_from_local_node
-----

=# DROP EXTENSION bdr;
DROP EXTENSION
```

9. トランザクション状態実行状態確認

手順1で実行したpgbenchにエラーが発生していないことを確認します。本検証ではエラーは発生しませんでした。

5.5.1.4.2. ノード追加

切り離したBDRノードを再度追加する場合、既存ノードのデータベースと追加するノードのデータベースのスキーマおよびデータを同期させる必要があります。ノード間のデータコピーには、論理コピーと物理コピーの2つの手法があります。

- <http://bdr-project.org/docs/stable/functions-node-mgmt.html#FUNCTION-BDR-GROUP-JOIN>

表 5.6 コピー取得方法

項番	コピー取得	説明	備考
1	bdr.bdr_group_join 関数実行	ユーザが指定したノード内データベースのスキーマとデータダンプを取得	pg_dumpコマンドに相当
2	bdr_init_copyコマンド	ユーザが指定したノード上の全てのデータベースのコピーを取得	pg_basebackupコマンドに相当

■ 論理コピーによるノード追加

1. トランザクション実行

pgbenchを利用して、ノード1にトランザクションを継続的に実行します。ノード2に復旧する際に、ノード1に対して実行したトランザクションの停止が必要か否かを確認します。

```
$ pgbench -h node1 -c 10 -t 100000 bdrtest
starting vacuum...end.
```

2. ノード2のPostgreSQL起動

ノード2上で動作するPostgreSQLを起動させます。

```
$ pg_ctl start
```

3. データベース削除

BDRで利用したデータベースを削除します。

```
(node2のデータベースに接続)
=# DROP DATABASE bdrtest ;
DROP DATABASE
```

※ 接続が残っており、削除できない場合はPostgreSQLを再起動

4. データベースの再作成

BDRで利用するデータベースを再度作成します。

```
(node2のデータベースに接続)
=# CREATE DATABASE bdrtest;
```

5. BDR有効化

無効化したBDRを再度有効化します。

```
(node2のデータベースに接続)
=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
=# CREATE EXTENSION bdr;
CREATE EXTENSION
```

6. ノードの追加

ノードを追加(復旧)させるため、「[bdr.bdr_group_join](#)」関数を実行します。

```
(node2のデータベースに接続)
=# SELECT bdr.bdr_group_join(
      local_node_name := 'node2',
      node_external_dsn := 'host=node2 port=5432 dbname=bdrtest',
      join_using_dsn := 'host=node1 port=5432 dbname=bdrtest'
    );

bdr_group_join
-----
(1 行)
```

7. ノード追加の確認待ち

ノードが追加されたことを確認するため、「[bdr.bdr_node_join_wait_for_ready](#)」関数を実行します。

```
(node2のデータベースに接続)
=# SELECT bdr.bdr_node_join_wait_for_ready();
※ トランザクションが実行中の場合、上記関数の結果が戻りません。
```

node1のPostgreSQLログファイルに下記メッセージが出力された後、復旧の処理が開始されません。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found initial starting point at 0/BB399BF0
DETAIL:  10 transactions need to finish.
```

pgbenchコマンドで実行中のトランザクションを停止すると、下記ログメッセージが出力され、復旧処理が開始されます。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found consistent point at 0/B2895648
DETAIL:  There are no running transactions.
LOG:  exported logical decoding snapshot: "00046FE0-1" with 0 transaction IDs
```

8. BDRの各ノード状態確認

ノード2が追加されたことを「[bdr.bdr_nodes](#)」テーブルの情報から確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r              <-- r(正常)であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | r              <-- r(正常)であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest      <-- node1から作成されていることを確認
```

■ 物理コピーによるノード追加

1. トランザクション実行

pgbenchを利用して、ノード1にトランザクションを継続的に実行します。ノード2に追加する際に、ノード1に対して実行したトランザクションの停止が必要か否かを確認します。

```
$ pgbench -h node1 -c 10 -t 100000 bdrtest
starting vacuum...end.
```

2. ノード2のPostgreSQL停止確認

ノード2上で動作するPostgreSQLが停止していることを確認します。

```
(node2にて実施)
$ pg_ctl status
pg_ctl: no server running
```

3. 物理コピーの取得

ノード2上で「bdr_init_copy」コマンドを実行し、ノード1上のコピーを取得します。

```
(node2にて実施)
$ rm -rf $PGDATA/*
$ bdr_init_copy -D $PGDATA -n node2 -h node1 -p 5432 -d bdrtest
--local-host=node2 --local-port=5432 --local-dbname=bdrtest
bdr_init_copy: starting ...
Getting remote server identification ...
Detected 1 BDR database(s) on remote server
Updating BDR configuration on the remote node:
bdrtest: creating replication slot ...
bdrtest: creating node entry for local node ...
Creating base backup of the remote node...
50357/50357 kB (100%), 1/1 tablespace
Creating restore point on remote node ...
Bringing local node to the restore point ...
トランザクションログをリセットします。
Initializing BDR on the local node:
bdrtest: adding the database to BDR cluster ...
All done
```

node1のPostgreSQLログファイルに下記メッセージが出力された後、追加の処理が開始されません。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found initial starting point at 0/BB399BF0
DETAIL:  10 transactions need to finish.
```

pgbenchコマンドで実行中のトランザクションを停止すると、下記ログメッセージが出力され、復旧処理が開始されます。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found consistent point at 0/B2895648
DETAIL:  There are no running transactions.
STATEMENT:  SELECT
pg_create_logical_replication_slot('bdr_25434_6369931070716042622_2_25434__', 'bdr');
```

4. BDRの各ノード状態確認

ノード2が追加されたことを「[bdr.bdr_nodes](#)」テーブルの情報から確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r                                <-- r (正常)であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | r                                <-- r (正常)であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest          <-- node1から作成されていることを確認
```

5.5.1.5. 検証結果

切り離れたノードの情報がシステムカタログに残ってしまうため、データ操作が禁止されているシステムカタログのデータ削除が必要でした。

システムカタログのデータ操作は禁止されていますが、切り離れたノードの情報がシステムカタログに残っている場合、ノード追加時に下記のメッセージが出力され、ノードが追加が実施できないため、本検証では暫定対処としてシステムカタログのデータ削除を実施しております。

```
ERROR: System identification mismatch between connection and slot
詳細: Connection for bdr (6369029438929838565,1,16385,) resulted in slot on node bdr
(6369034270871134883,2,16385,) instead of expected node
LOG: ワークプロセス: bdr db: bdrtest (PID 17232)は終了コード1で終了しました
```

ノード追加にはプロセスの再起動は必要ありませんが、トランザクションの停止が必要でした。

5.5.2. グローバルシーケンス設定

5.5.2.1. 検証の目的

BDRに実装されたグローバルシーケンスの利用方法について確認します。グローバルシーケンスを利用することでノード毎に払い出されるシーケンス番号を独立させ、ノード間のシーケンス番号の競合を防ぐことができるか否かを確認します。

5.5.2.2. 検証環境

[こちらの環境](#)で検証を実施しました。

5.5.2.3. 検証手順

1. 初期状態

BDRの各ノード間のレプリケーションが正常に動作していることを「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r                                <-- r (正常)であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | r                                <-- r (正常)であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest          <-- node1から作成されていることを確認
```

2. グローバルシーケンス動作確認

node1のデータベースに接続し、serial型を列に持つgstestテーブルを作成します。serial型を持つテーブルを作成すると、自動的にシーケンスが作成

されます。グローバルシーケンス作成のため、テーブル作成時に「SET LOCAL default_sequenceam = 'bdr';」を指定する必要があります。

```
(node1のデータベースに接続)
=#
BEGIN;
    SET LOCAL default_sequenceam = 'bdr';                <-- グローバルシーケンス作成用の設定
    CREATE TABLE gstest (
        id serial primary key,
        hogehoge text
    );
COMMIT;
```

参考． 上記DDL実行時にnode1に出力されるログメッセージ

```
LOG:  DDL LOCK TRACE: attempting to acquire in mode <ddl_lock> for (bdr (xxxxxxxx))
-----
STATEMENT:  CREATE TABLE gstest (
                id serial primary key,
                hogehoge text
            );
LOG:  DDL LOCK TRACE: attempting to acquire in mode <write_lock> (upgrading from
<ddl_lock>) for (bdr (xxxxxxxx))
>STATEMENT:  CREATE TABLE gstest (
                id serial primary key,
                hogehoge text
            );
-----
```

3. グローバルシーケンス作成確認

node1およびnode2のデータベースに接続し、グローバルシーケンスが作成されたことを確認します。

(node1のデータベースに接続)	(node2のデータベースに接続)
=# \ds	=# \ds
List of relations	List of relations
-[RECORD 1]-----	-[RECORD 1]-----
Schema public	Schema public
Name gstest_id_seq	Name gstest_id_seq
Type sequence	Type sequence
Owner postgres	Owner postgres

※ 下記のSQLで詳細情報(pg_classの情報)を確認可能です。

```
=# SELECT * FROM pg_class
    INNER JOIN pg_seqam ON (pg_class.relam = pg_seqam.oid)
    WHERE pg_seqam.seqamname = 'bdr' AND relkind = 'S';
```

4. グローバルシーケンス利用

node1のデータベースに接続し、1件のデータを投入します。

```
(node1のデータベースに接続)
=# INSERT INTO gstest(hogehoge) VALUES ('test1');
INSERT 0 1
```

node2のデータベースに接続し、1件のデータを投入します。

```
(node2のデータベースに接続)
=# INSERT INTO gstest(hogehoge) VALUES ('test2');
INSERT 0 1
```

5. シーケンス値の競合有無確認

node1およびnode2のシーケンスの値に競合が発生していないことを確認します。本検証では、node1に2が割り当てられ、node2には100001が割り当てられました。

```
(node1のデータベースに接続)
=# SELECT * FROM gstest;
   id   | hogehoge
-----+-----
      2 | test1
 100001 | test2
(2 rows)
```

<-- node1には2のシーケンスが割り当てられる
<-- node2には100001のシーケンスが割り当てられる

6. グローバルシーケンス利用

node1に200,000件のデータを投入し、シーケンスの値が100,000を超えた場合、node2の100,001と競合しないか否かを確認しました。

```
(node1において実施)
$ cat /tmp/gs.sh
#!/bin/sh

for i in `seq 1 200000`
do
    psql -d bdrtest -c "INSERT into gstest(hogehoge) VALUES ('test${i}');"
done
$ sh /tmp/gs.sh
INSERT 0 1
[省略]
```

7. グローバルの状態確認

node1のシーケンスが100,000を利用したのち、node1に割り振られた値を確認しました。

```
(node1のデータベースに接続)
=# SELECT * FROM gstest WHERE 99999 <= id;

   id   | hogehoge
-----+-----
 99999 | test99997
100000 | test99998
100001 | test2
150001 | test99999
150002 | test100000
150003 | test100001
150004 | test100002
150005 | test100003
150006 | test100004
```

-< 100,000を超えると、node1のシーケンスが150,001に割り振られる。

5.5.2.4. 検証結果

グローバルシーケンスを用いることでノード間のシーケンス番号の競合を抑止できることが確認できました。グローバルシーケンスについてはいくつかの制限事項がマニュアルに記載されておりますので、注意して下さい。

- 1 (デフォルト値) の INCREMENT だけがサポートされています。

2. MINVALUEとMAXVALUEはデフォルトでロックされており、変更することはできません。
3. CACHE 指令はサポートされていません。

- <http://bdr-project.org/docs/1.0/global-sequence-limitations.html>

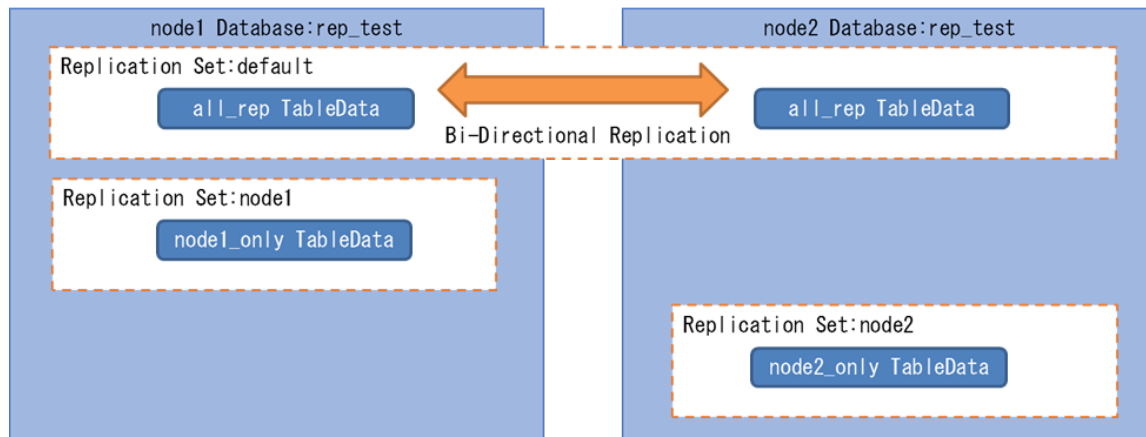
5.5.3. 選択的レプリケーション

選択的レプリケーションについて記載します。

5.5.3.1. 検証目的

テーブル単位での選択的レプリケーションの可否を確認します。選択的レプリケーションの可否は互いのデータベース間での任意のテーブルのデータ状態を元に判断します。また、選択的レプリケーションを実現する際の変更手順を明確にする事を目的とします。

5.5.3.2. 検証内容



- Replication Set : レプリケーション対象とするテーブルの論理集合を表し、ノード毎に設定可能
- <http://bdr-project.org/docs/stable/replication-sets.html>

選択的レプリケーション検証内容

- 双方向レプリケーションの動作確認
- 選択的レプリケーション(node1)の動作確認
- 選択的レプリケーション(node2)の動作確認

5.5.3.3. 検証環境

[こちらの環境](#) で検証を実施しました。

5.5.3.4. 検証手順

1. 選択的レプリケーション環境構築

選択的レプリケーションの動作確認の為に、下記の検証環境を構築します。

(1)データベース作成

テストを実施する為のデータベースを作成します。

```
(node1,node2において実施)
=# CREATE DATABASE rep_test;
```

(2)BDRの有効化

作成したデータベース上でBDR機能の有効化を実施します。

```
(node1,node2のデータベースにおいて実施)
=# CREATE EXTENSION btree_gist;
=# CREATE EXTENSION bdr;
```

(3)BDRグループの作成

BDRノードのクラスタに「[bdr.bdr_group_create](#)」を利用して最初のノードを作成します。

```
(node1のデータベースに接続)
=# SELECT bdr.bdr_group_create(
    local_node_name := 'node1',
    node_external_dsn := 'host=node1 port=5432 dbname=rep_test',
    replication_sets:= ARRAY['default','node1']);           ※ replications_ret
にnode1を追加
```

(4)ノード登録(node2)

既存のBDRノードのクラスタに「[bdr.bdr_group_join](#)」を利用してノードを登録します。これにより、全てのノード間でレプリケーションが開始されます。

```
(node2のデータベースに接続)
=# SELECT bdr.bdr_group_join(
    local_node_name := 'node2',
    node_external_dsn := 'host=node2 port=5432 dbname=rep_test',
    join_using_dsn := 'host=node1 port=5432 dbname=rep_test',
    replication_sets:= ARRAY['default','node2']);           ※ replications_ret
にnode2を追加
```

(5)ノードの状態確認

BDRグループ内のノードのメンバシップを「[bdr.bdr_nodes](#)」を利用して確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=rep_test
node_status        | r                                <-- r (正常) であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=rep_test
node_status        | r                                <-- r (正常) であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=rep_test <-- node1から作成されていることを確認
```

(6)レプリケーションセットの確認

指定したノードのレプリケーションセットを「[bdr.connection_get_replication_sets](#)」を利用して確認します。

```
(node1のデータベースに接続)
=# SELECT bdr.connection_get_replication_sets('node1');
-[ RECORD 1 ]-----+-----
connection_get_replication_sets | {default,node1}                <-- node1が追加されていること

=# SELECT bdr.connection_get_replication_sets('node2');
-[ RECORD 1 ]-----+-----
connection_get_replication_sets | {default,node2}                <-- node2が追加されていること
```

2. 双方向レプリケーション対象テーブルの動作確認

まずは、通常の双方向レプリケーション動作を確認します。片方のテーブルにデータを挿入した時にもう一方のテーブルにもデータが挿入されています。

(1)テーブル(all_rep)作成

テストを実施する為のテーブル(all_rep)を作成します。

```
(node1のデータベースに接続)
=# CREATE TABLE all_rep(id int, PRIMARY KEY(id));
CREATE TABLE
```

(2)レプリケーションセット確認

指定したテーブルのレプリケーションセットを「[bdr.connection_get_replication_sets](#)」を利用して確認します。

```
(node1のデータベースに接続)
=# SELECT bdr.table_get_replication_sets('all_rep');
-[ RECORD 1 ]-----+-----
table_get_replication_sets | {default,all}    <-- defaultが含まれていること
```

(3)データ追加

node1のテーブルにデータを挿入します。

```
(node1のデータベースに接続)
=# INSERT INTO all_rep VALUES (1);
INSERT 0 1
```

(4)レプリケーション確認

node2のテーブルにもデータがレプリケーションされている事を確認します。

```
(node2のデータベースに接続)
=# SELECT id FROM all_rep;
 id
----
  1
(1 row)                                <-- データがレプリケーションされていること
```

3. レプリケーション対象外テーブルの作成(node1のみ)

次に、選択的レプリケーション動作を確認します。指定したテーブルではレプリケーション動作が行われなくなりますので、任意のテーブルのみレプリケーションさせる事が出来ます。

(1)テーブル(node1_only)の作成

テストを実施する為のテーブル(node1_only)を作成します。

```
(node1のデータベースに接続)
=# CREATE TABLE node1_only(id int, PRIMARY KEY(id));
CREATE TABLE
```

(2)レプリケーションセット変更

指定したテーブルのレプリケーションセットを「[bdr.table_set_replication_sets](#)」を利用して'node1'に設定し、レプリケーション対象を'node1'のみにします。設定したレプリケーションセットを確認するには、「[bdr.table_get_replication_sets](#)」を利用します。

```
(node1のデータベースに接続)
=# SELECT bdr.table_set_replication_sets('node1_only', ARRAY['node1']);
-[ RECORD 1 ]-----+
table_set_replication_sets |

=# SELECT bdr.table_get_replication_sets('node1_only');
-[ RECORD 1 ]-----+-----
table_get_replication_sets | {node1,all}      <-- node1が追加されていること
```

(3)データ追加

node1のテーブルにデータを挿入します。

```
(node1のデータベースに接続)
=# INSERT INTO node1_only VALUES (1);
INSERT 0 1
```

(4)レプリケーション確認

node2のテーブルにはデータがレプリケーションされていない事を確認します。

```
(node2のデータベースに接続)
=# SELECT id FROM node1_only;
 id
----
(0 行)                                <-- レプリケーションされていないこと
```

4. レプリケーション対象外テーブルの作成(node2のみ)

同様に、もう一方からの選択的レプリケーション動作も確認します。

(1)テーブル(node2_only)の作成

テストを実施する為のテーブル(node2_only)を作成します。

```
(node2のデータベースに接続)
=# CREATE TABLE node2_only(id int, PRIMARY KEY(id));
CREATE TABLE
```

(2)レプリケーションセット変更

指定したテーブルのレプリケーションセットを「[bdr.table_set_replication_sets](#)」を利用して'node2'に設定し、レプリケーション対象を'node2'のみにします。設定したレプリケーションセットを確認するには、「[bdr.table_get_replication_sets](#)」を利用します。

```
(node2のデータベースに接続)
=# SELECT bdr.table_set_replication_sets('node2_only', ARRAY['node2']);
-[ RECORD 1 ]-----+
table_set_replication_sets |

=# SELECT bdr.table_get_replication_sets('node2_only');
-[ RECORD 1 ]-----+-----
table_get_replication_sets | {node2,all}      <-- node2が追加されていること
```

(3)データ追加

node2のテーブルにデータを挿入します。

```
(node2のデータベースに接続)
=# INSERT INTO node2_only VALUES (1);
INSERT 0 1
```

(4)レプリケーション確認

node1のテーブルにはデータがレプリケーションされていない事を確認します。

```
(node1のデータベースに接続)
=# SELECT id FROM node2_only;
 id
----
(0 行)
```

<-- レプリケーションされていないこと

5.5.3.5. 検証結果

今回の検証結果では、レプリケーションセットに任意のノードを指定する事で選択的レプリケーションが実現される事が確認出来ました。また、選択的レプリケーション実現の為の設定変更手順についても確認出来ました。

ただし、レプリケーションセットにテーブルを追加する場合は、過去のデータまで反映しないので、手動で同期が必要です。

選択的レプリケーションの主な注意事項は下記になります。

- <http://bdr-project.org/docs/1.0/replication-sets-concepts.html>
 - <http://bdr-project.org/docs/1.0/replication-sets-tables.html>
1. DDLはレプリケーションセットの設定に関係なく常にすべてのノードに影響します。
 2. TRUNCATEは常にレプリケートされます。望ましくない場合は、DELETEを利用する必要があります。
 3. レプリケーションセットの設定は、初期ノード追加(結合時)にはテーブルデータのコピーに影響を与えません。
 4. レプリケーションセットにテーブルを追加した場合も以前のデータ内容はノードに同期されません。通常、管理者は、テーブルをレプリケーションセットに追加した後、手動で同期する必要があります。

5.5.4. 更新処理競合時の動作

5.5.4.1. 検証目的

BDRはマルチマスタ構成する各ノードに対して、参照処理と更新処理を実行することが可能です。複数ノードに対して同時に更新処理が実施された場合、各ノードに対して実行された更新処理が競合する事象が発生する場合があります。

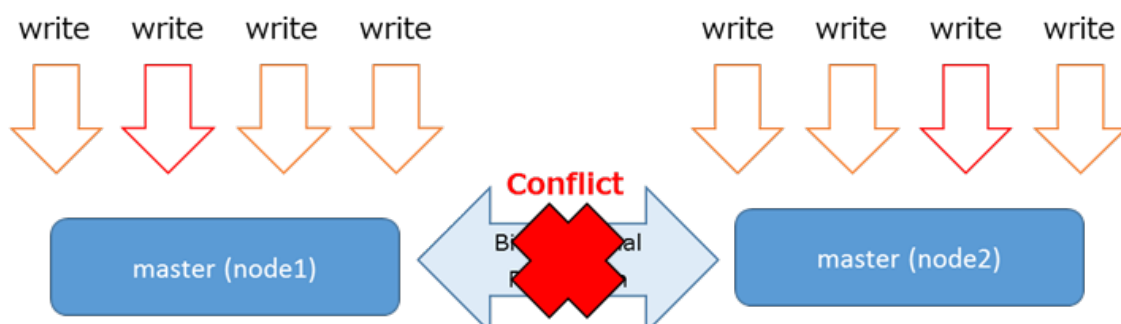
BDR競合が発生した場合、最後の更新処理が適用されます(last_update_wins)。また、競合結果はテーブル「[bdr.bdr_conflict_history](#)」で確認可能です。

- 「bdr.bdr_conflict_history」に格納される情報については下記をご参照下さい。
- <http://bdr-project.org/docs/stable/conflicts-types.html>

下記を明らかにするため検証を実施しました。

1. 更新処理が競合した場合の挙動
2. 競合発生時にシステムカタログに記録される情報

5.5.4.2. 検証内容



BDR動作検証概要図

下表の競合発生時の動作を検証しました。

表 5.7 検証した競合のパターン

項番	分類	説明
1	PRIMARY KEYまたはUNIQUE制約	2つの操作が同じUNIQUE KEYを持つ行に影響を及ぼす行の競合を検証します。
2	外部キー制約	外部キー制約が定義されたテーブルにおいて、制約に違反するデータ削除によって引き起こされる競合を検証します。
3	排他制約	BDRでは排他制約をサポートしていないために、排他制約において競合が発生した場合を検証します。
4	グローバルなデータ	ノードのグローバル(PostgreSQLシステム全体)のデータ(ロールなど)が異なる場合での競合を検証します。
5	ロックの競合とデッドロックの中断	BDR適用プロセスとロックの競合について検証します。
6	その他	自動的に解決出来ないデータの相違が発生した場合に手動で調整を行う方法を検証します。

5.5.4.3. 検証環境

[こちらの環境](#) で検証を実施しました。

5.5.4.4. 検証手順

1. 競合ログオプションの有効化確認

競合発生時にテーブル「[bdr.bdr_conflict_history](#)」に情報を記録するため、bdr.log_conflicts_to_tableパラメータがonに設定されていることを確認します。

※ offになっていたら、postgresql.confに「bdr.log_conflicts_to_table=on」を追記し、再読み込みを実行して下さい。

node1/node2のデータベースに接続し、パラメータを確認するために下記コマンドを実行します。

```
(node1にて実施)
=# SHOW bdr.log_conflicts_to_table;
bdr.log_conflicts_to_table
-----
on
(1 行)
```

```
(node2にて実施)
=# SHOW bdr.log_conflicts_to_table;
bdr.log_conflicts_to_table
-----
on
(1 行)
```

2. レプリケーション反映時間の設定

ネットワーク遅延などの実際の環境を想定して検証を行うため、bdr.default_apply_delayパラメータを設定します。このパラメータを設定する事でレプリケーションの反映を設定した時間(ミリ秒)遅らせる事が出来ます。

※ 今回の検証では、競合の発生を確認しやすくするために"2s"を設定しています。

node1/node2のデータベースに接続し、パラメータを確認するために下記コマンドを実行します。

```
(node1にて実施)
=# SHOW bdr.default_apply_delay;
bdr.default_apply_delay
-----
2s
(1 行)
```

```
(node2にて実施)
=# SHOW bdr.default_apply_delay;
bdr.default_apply_delay
-----
2s
(1 行)
```

3. BDRの各ノード状態確認

BDRの各ノード間のレプリケーションが正常に動作していることを「[bdr.bdr_nodes](#)」を利用して確認します。

```

(node1にて実施)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;

-[ RECORD 1 ]-----+-----
node_name      | node1
node_local_dsn | host=node1 port=5432 dbname=bdrtest
node_status    | r                                     <-- r (正常)であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name      | node2
node_local_dsn | host=node2 port=5432 dbname=bdrtest
node_status    | r                                     <-- r (正常)であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest <-- node1から作成されていることを確認

```

4. 検証用テーブルとデータの作成

pgbenchを利用して、検証時に利用するテーブルとデータを作成します。

```

(node1にて実施)
$ pgbench -i -s 10 bdrtest
$ psql -h node1 -d bdrtest

=# INSERT INTO pgbench_accounts (aid, bid, abalance) VALUES (1000001, 1, 0);

```

5.5.4.4.1. PRIMARY KEYまたは UNIQUE制約

1. INSERT vs INSERT

■ 競合概要

最も一般的な競合として、2つの異なるノードのINSERTが同じPRIMARY KEYの値(または、単一のUNIQUE制約の値)を持つデータを挿入するケースを検証しました。

■ 検証結果

タイムスタンプに従い、最後にデータが作成された方が保持されました。

■ 競合発生手順

各ノードからそれぞれINSERTを実行し、INSERT vs INSERT の競合を発生させます。

```

(node1のデータベースへSQLを投入)
=# INSERT INTO pgbench_accounts (aid, bid, abalance) VALUES (1000002, 2, 0);

(node2のデータベースへSQLを投入)
=# INSERT INTO pgbench_accounts (aid, bid, abalance) VALUES (1000002, 3, 0);

```

競合ログを「[bdr.bdr_conflict_history](#)」を利用して確認し、競合の発生を確認します。

※ 競合履歴テーブルでは競合ログが出力されている側のサーバをローカル、反対側をリモートとして表示します。よって、それぞれのサーバで実行されたトランザクションをローカルトランザクション、リモートトランザクションとして説明します。

(node1またはnode2の競合履歴テーブルに競合ログが出力)

```
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;

-[ RECORD 1 ]-----+-----
conflict_id          | 48
local_node_sysid     | 6371561413984311673
local_conflict_xid    | 3944663
local_conflict_lsn    | 3/765949B8
local_conflict_time   | 2017-01-25 10:36:03.787698+09
object_schema        | public
object_name           | pgbench_accounts
remote_node_sysid     | 6371553825031594764
remote_txid           | 3960113
remote_commit_time    | 2017-01-25 10:32:48.772254+09
remote_commit_lsn     | 2/CA1AD950
conflict_type         | insert_insert
conflict_resolution   | last_update_wins_keep_local
local_tuple           | {"aid":1000002,"bid":3,"abalance":0,"filler":null}
remote_tuple          | {"aid":1000002,"bid":2,"abalance":0,"filler":null}
local_tuple_xmin      | 3944662
local_tuple_origin_sysid |
error_message         |
error_sqlstate        |
error_querystring     |
error_cursorpos       |
error_detail          |
error_hint            |
error_context         |
error_columnname      |
error_typename        |
error_constraintname  |
error_filename        |
error_lineno          |
error_funcname        |
```

それぞれのテーブルの状態を確認し、競合発生時の処理を確認します。

※ 後にデータが作成されたローカルトランザクションが保持されています。

(node1のデータベースを確認)

```
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid  | bid | abalance | filler
-----+-----+-----+-----
1000001 | 1 | 0 |
1000002 | 3 | 0 |
(2 行)
```

(node2のデータベースを確認)

```
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid  | bid | abalance | filler
-----+-----+-----+-----
1000001 | 1 | 0 |
1000002 | 3 | 0 |
(2 行)
```

※ last_update_wins_keep_local : 最新のタイムスタンプであるローカル側の更新が適用

2. INSERT vs UPDATE

■ 競合概要

一つのノードでINSERTしたデータともう片方のノードでUPDATEしたデータが同じPRIMARY KEYの値を持つケースを検証しました。

■ 検証結果

INSERT/UPDATEの競合が発生した場合、競合を解消するためオペレータ側でのデータ操作が必要となり、注意が必要です。

■ 競合発生手順

下記テーブルを初期状態とします。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
1000001 |   1 |         0 |
(1 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
1000001 |   1 |         0 |
(1 行)
```

各ノードからそれぞれINSERT/UPDATEを実行し、INSERT vs UPDATE の競合を発生させます。

```
(node1のデータベースへSQLを投入)
=# INSERT INTO pgbench_accounts (aid, bid, abalance) VALUES (1000002, 4, 0);

(node2のデータベースへSQLを投入)
=# UPDATE pgbench_accounts SET aid = 1000002 WHERE aid = 1000001;
```

競合ログを「[bdr.bdr_conflict_history](#)」を利用して確認し、競合の発生を確認します。

```
(node1またはnode2の競合履歴テーブルに競合ログが出力)
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;

-[ RECORD 1 ]-----+-----
conflict_id          | 49
local_node_sysid     | 6371561413984311673
local_conflict_xid    | 3944715
local_conflict_lsn    | 3/81295CF8
local_conflict_time   | 2017-01-25 10:36:46.780822+09
object_schema        | public
object_name          | pgbench_accounts
remote_node_sysid     | 6371553825031594764
remote_txid           | 3960133
remote_commit_time    | 2017-01-25 10:33:31.640126+09
remote_commit_lsn     | 2/D1CE6F00
conflict_type         | insert_insert
conflict_resolution   | last_update_wins_keep_local
local_tuple           | {"aid":1000002,"bid":1,"abalance":0,"filler":null}
remote_tuple          | {"aid":1000002,"bid":4,"abalance":0,"filler":null}
local_tuple_xmin      | 3944714
local_tuple_origin_sysid |
error_message         |
error_sqlstate        |
error_querystring     |
error_cursorpos       |
error_detail          |
error_hint            |
error_context         |
error_columnname      |
error_typename        |
error_constraintname  |
error_filename        |
error_lineno          |
error_funcname        |
```

それぞれのテーブルの状態を確認し、競合発生時の処理を確認します。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
1000001 |   1 |         0 |
1000002 |   4 |         0 |
(2 行)

(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
1000002 |   1 |         0 |
(1 行)
```

PRIMARY KEYのUPDATEで発生した競合でリモート側では一意制約違反のために適用プロセスがデータ反映できず、ローカル側では論理レプリケーション中に接続が切断され、データの整合性がとれなくなる状態が確認されました。

(node1のデータベースログ)

```
LOG:  starting background worker process "bdr (6371553825031594764,1,43905,)->bdr
(6371561413984311673,2,"
ERROR:  duplicate key value violates unique constraint "pgbench_accounts_pkey"
DETAIL:  Key (aid)=(1000002) already exists.
CONTEXT:  apply UPDATE from remote relation public.pgbench_accounts in commit
1/2F279190, xid 3941360 committed at 2017-01-10 11:00:22.736985+09 (action #2) from node
(6371561413984311673,2,43865)
LOG:  worker process: bdr (6371553825031594764,1,43905,)->bdr (6371561413984311673,2,
(PID 21287) exited with exit code 1
```

(node2のデータベースログ)

```
bdr (6371553825031594764,1,43905,):receive LOG:  starting logical decoding for slot
"bdr_43865_6371553825031594764_1_43905__"
bdr (6371553825031594764,1,43905,):receive DETAIL:  streaming transactions committing
after 1/2F279190, reading WAL from 1/2F279068
bdr (6371553825031594764,1,43905,):receive LOG:  logical decoding found consistent point
at 1/2F279068
bdr (6371553825031594764,1,43905,):receive DETAIL:  There are no running transactions.
bdr (6371553825031594764,1,43905,):receive LOG:  could not receive data from client:
Connection reset by peer
bdr (6371553825031594764,1,43905,):receive LOG:  unexpected EOF on standby connection
```

※ 競合するタブルをローカル側から手動で削除するか、新しいリモートタブルと競合しなくなるようにUPDATEする必要がある

上記状態では、ローカル側からのレプリケーションが実施されない状態が続くため、解消するために、ローカル側のデータをUPDATEします。

(node1のデータベースへSQLを投入)

```
=# UPDATE pgbench_accounts SET aid = 1000003 WHERE aid = 1000002;
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
 1000003 |   4 |         0 |
(2 行)
```

(node2のデータベースを確認)

```
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000002 |   1 |         0 |
(1 行)
```

※ 競合は解消され、次のデータからレプリケーションが始まるがデータの整合性で問題あり

3. UPDATE vs DELETE

■ 競合概要

一つのノードでUPDATEしたデータともう片方のノードでDELETEしたデータが同じPRIMARY KEYの値を持つケースを検証しました。

■ 検証結果

UPDATE/DELETEが競合した場合、UPDATEが破棄されました。

■ 競合発生手順

下記テーブルを初期状態とします。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
(1 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
(1 行)
```

各ノードからそれぞれUPDATE/DELETEを実行し、UPDATE vs DELETE の競合を発生させます。

```
(node1のデータベースにSQLを投入)
=# UPDATE pgbench_accounts SET bid = 4 WHERE aid = 1000001;
```

```
(node2のデータベースにSQLを投入)
=# DELETE FROM pgbench_accounts WHERE aid = 1000001;
```

競合ログを「[bdr.bdr_conflict_history](#)」を利用して確認し、競合の発生を確認します。

```
(node1またはnode2の競合履歴テーブルに競合ログが出力)
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;
```

```
-[ RECORD 1 ]-----+-----
conflict_id          | 51
local_node_sysid     | 6371561413984311673
local_conflict_xid    | 0
local_conflict_lsn    | 3/8BF7BA38
local_conflict_time   | 2017-01-25 10:37:45.561678+09
object_schema        | public
object_name           | pgbench_accounts
remote_node_sysid     | 6371553825031594764
remote_txid           | 3960157
remote_commit_time    | 2017-01-25 10:34:30.564936+09
remote_commit_lsn     | 2/D9831CC0
conflict_type         | update_delete
conflict_resolution   | skip_change
local_tuple           |
remote_tuple          | {"aid":1000001,"bid":4,"abalance":0,"filler":null}
local_tuple_xmin      |
local_tuple_origin_sysid |
error_message         |
error_sqlstate        |
error_querystring     |
error_cursorpos       |
error_detail          |
error_hint            |
error_context         |
error_columnname      |
error_type_name       |
error_constraintname  |
error_filename        |
error_lineno          |
error_funcname        |
```

それぞれのテーブルの状態を確認し、競合発生時の処理を確認します。

※ DELETE後のUPDATEが破棄されています。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid | bid | abalance | filler
-----+-----+-----+-----
(0 行)

(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid | bid | abalance | filler
-----+-----+-----+-----
(0 行)

※ skip_change : 変更を無視し、破棄された
```

4. INSERT vs DELETE

■ 競合概要

一つのノードでINSERTしたデータがもう片方のノードでDELETEされたデータと同じPRIMARY KEYの値を持つケースを検証しました。

■ 検証結果

DELETEの処理が破棄されました。競合発生の状態についてはシステムカタログから確認することができませんでした。

■ 競合発生手順

下記テーブルを初期状態とします。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid | bid | abalance | filler
-----+-----+-----+-----
1000001 | 1 | 0 |
(1 行)

(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid | bid | abalance | filler
-----+-----+-----+-----
1000001 | 1 | 0 |
(1 行)
```

各ノードからそれぞれINSERT/DELETEを実行し、INSERT vs DELETE の競合を発生させます。

```
(node1のデータベースにSQLを投入)
=# INSERT INTO pgbench_accounts (aid, bid, abalance) VALUES (1000002, 5, 0);

(node2のデータベースにSQLを投入)
=# DELETE FROM pgbench_accounts WHERE aid = 1000002;
```

競合ログを「[bdr.bdr_conflict_history](#)」を利用して確認し、競合の発生を確認します。

※ 競合は確認出来ませんでした。

```
(node1またはnode2の競合履歴テーブルに競合ログが出力されない)
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;
```

それぞれのテーブルの状態を確認します。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
 1000002 |   5 |         0 |
(2 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
 1000002 |   5 |         0 |
(2 行)
```

5. DELETE vs DELETE

■ 競合概要

2つの異なるノードで実行されたDELETEが同じPRIMARY KEYの値を持つデータを削除するケースを検証しました。

■ 検証結果

DELETE/DELETEが競合した場合、片方のDELETEが無視され処理を完了しました。

■ 競合発生手順

下記テーブルを初期状態とします。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
(1 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
   aid   | bid | abalance | filler
-----+-----+-----+-----
 1000001 |   1 |         0 |
(1 行)
```

各ノードからそれぞれDELETEを実行し、DELETE vs DELETE の競合を発生させます。

```
(node1のデータベースにSQLを投入)
=# DELETE FROM pgbench_accounts WHERE aid = 1000001;

(node2のデータベースにSQLを投入)
=# DELETE FROM pgbench_accounts WHERE aid = 1000001;
```

競合ログを「[bdr.bdr_conflict_history](#)」を利用して確認し、競合の発生を確認します。

```
(node1またはnode2の競合履歴テーブルに競合ログが出力)
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;

-[ RECORD 1 ]-----+-----
conflict_id          | 52
local_node_sysid     | 6371561413984311673
local_conflict_xid    | 0
local_conflict_lsn    | 3/A193CA88
local_conflict_time   | 2017-01-25 10:39:07.794699+09
object_schema        | public
object_name           | pgbench_accounts
remote_node_sysid     | 6371553825031594764
remote_txid           | 3960197
remote_commit_time    | 2017-01-25 10:35:52.78243+09
remote_commit_lsn     | 2/E8E9A388
conflict_type         | delete_delete
conflict_resolution   | skip_change
local_tuple           |
remote_tuple          | {"aid":1000001,"bid":null,"abalance":null,"filler":null}
local_tuple_xmin      |
local_tuple_origin_sysid |
error_message         |
error_sqlstate        |
error_querystring     |
error_cursorpos       |
error_detail          |
error_hint            |
error_context         |
error_columnname      |
error_typename        |
error_constraintname  |
error_filename        |
error_lineno          |
error_funcname        |
```

それぞれのテーブルの状態を確認し、競合発生時の処理を確認します。

※ ローカル側のノードからのDELETEが無視されています。

```
(node1のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid | bid | abalance | filler
-----+-----+-----+-----
(0 行)

(node2のデータベースを確認)
=# SELECT * FROM pgbench_accounts WHERE aid >= 1000001;
 aid | bid | abalance | filler
-----+-----+-----+-----
(0 行)

※ skip_change : 変更を無視し、破棄された
```

5.5.4.4.2. 外部キー制約

1. 外部キー制約の競合

■ 競合概要

1つのノードで外部キー制約が定義されたテーブルに対してデータを挿入し、上記データ挿入がもう片方のノードに反映される前に、もう片方のノードにおいて外部キーの参照先であるテーブルのデータを削除することで、外部キー制約が定義されたテーブルへのデータ挿入と、外部キー参照先テーブルに対するデータ削除を競合させます。

■ 検証結果

外部キー制約に違反する処理が実施された場合でもBDRではエラーにならず、データが外部キー制約違反の状態になりました。

■ 競合発生手順

下記の親子関係を持つテーブルを作成します。

```
(node1のデータベースに接続)
=# CREATE TABLE parent(id integer primary key);
=# CREATE TABLE child(id integer primary key, parent_id integer not null references
parent(id));
=# INSERT INTO parent(id) VALUES (1), (2);
=# INSERT INTO child(id, parent_id) VALUES (11, 1), (12, 2);
```

各ノードからそれぞれINSERT/DELETEを実行し、外部キー制約の競合を発生させます。

※ node1からのINSERTはnode2では、子テーブルで参照されている親が存在せず、node2のDELETEはnode1で参照している親データを削除しようとしているので、それぞれ適用出来なくなります。

```
(node1のデータベースにSQLを投入)
=# INSERT INTO child(id, parent_id) VALUES (21, 2);

(node2のデータベースにSQLを投入)
=# DELETE FROM child WHERE parent_id = 2;
=# DELETE FROM parent WHERE id = 2;
```

それぞれのテーブルの状態を確認し、競合発生時の処理を確認します。

```
(node1のデータベースを確認)
=# SELECT * FROM child;
id | parent_id
----+-----
11 |         1
21 |         2
(2 行)

=# SELECT * FROM parent;
id
----
1
(1 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM child;
id | parent_id
----+-----
11 |         1
21 |         2
(2 行)

=# SELECT * FROM parent;
id
----
1
(1 行)

※ 外部キー制約違反のデータがchildテーブルに存在する。
```

5.5.4.4.3. 排他制約

1. 排他制約の競合

■ 競合概要

排他な関係である2つのデータを異なるノードから同時にテーブルにINSERTし、ノード間のデータに競合が発生するケースを検証します。

■ 検証結果

一つのノードでINSERTされたデータをもう片方のノードに反映させる際に排他制約違反が発生し、ノード間の接続が切断されます。競合を解消するためにはオペレータでのデータ操作が必要となるため注意が必要です。

■ 競合発生手順

下記の排他制約を持つテーブルを作成します。

```
(node1のデータベースにSQLを投入)
=# CREATE TABLE sample (aid integer primary key, range daterange, price integer);
=# ALTER TABLE sample ADD EXCLUDE USING gist (price WITH =, range WITH &&);
```

各ノードからそれぞれINSERTを実行し、排他制約の競合を発生させます。

※ [2012-04-18 ~ 2012-04/20]の期間が競合しているために排他制約違反を起こしています。

```
(node1のデータベースにSQLを投入)
=# INSERT INTO sample VALUES(1, '[2012-04-16, 2012-04-20]', 10000);

(node2のデータベースにSQLを投入)
=# INSERT INTO sample VALUES(2, '[2012-04-18, 2012-04-23]', 10000);
```

```
(node1またはnode2の競合履歴テーブルに競合ログが出力されない)
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;
```

それぞれのテーブルの状態を確認し、競合発生時の処理を確認します。

※ 他のノードで実行されたデータ更新を反映する際に、制約違反が発生し、BDRによるノード間の接続が切断されます。

```
(node1のデータベースを確認)
=# SELECT * FROM sample;
 aid |          range          | price
-----+-----+-----
   1 | [2012-04-16,2012-04-21) | 10000
(1 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM sample;
 aid |          range          | price
-----+-----+-----
   2 | [2012-04-18,2012-04-24) | 10000
(1 行)
```

```
(node1のデータベースログ)
bdr (6371561413984311673,2,42879,):receive LOG: starting logical decoding for slot
"bdr_42892_6371561413984311673_2_42879_"
bdr (6371561413984311673,2,42879,):receive DETAIL: streaming transactions committing
after 1/15D41640, reading WAL from 1/15D413B0
bdr (6371561413984311673,2,42879,):receive LOG: logical decoding found consistent point
at 1/15D413B0
bdr (6371561413984311673,2,42879,):receive DETAIL: There are no running transactions.
bdr (6371561413984311673,2,42879,):receive LOG: unexpected EOF on standby connection
LOG: starting background worker process "bdr (6371553825031594764,1,42892,)->bdr
(6371561413984311673,2,"
ERROR: conflicting key value violates exclusion constraint "sample_price_range_excl"
DETAIL: Key (price, range)=(10000, [2012-04-18,2012-04-24)) conflicts with existing key
(price, range)=(10000, [2012-04-16,2012-04-21)).
CONTEXT: apply INSERT from remote relation public.sample in commit 1/194FBC20, xid
3940666 committed at 2017-01-06 16:12:28.275376+09 (action #2) from node
(6371561413984311673,2,42879)
LOG: worker process: bdr (6371553825031594764,1,42892,)->bdr (6371561413984311673,2,
(PID 27843) exited with exit code 1
```

```
(node2のデータベースログ)
LOG: starting background worker process "bdr (6371561413984311673,2,42879,)->bdr
(6371553825031594764,1,"
ERROR: conflicting key value violates exclusion constraint "sample_price_range_excl"
DETAIL: Key (price, range)=(10000, [2012-04-16,2012-04-21)) conflicts with existing key
(price, range)=(10000, [2012-04-18,2012-04-24)).
CONTEXT: apply INSERT from remote relation public.sample in commit 1/15D41640, xid
3957701 committed at 2017-01-06 16:09:10.299493+09 (action #2) from node
(6371553825031594764,1,42892)
LOG: worker process: bdr (6371561413984311673,2,42879,)->bdr (6371553825031594764,1,
(PID 29311) exited with exit code 1
bdr (6371553825031594764,1,42892,):receive LOG: starting logical decoding for slot
"bdr_42879_6371553825031594764_1_42892_"
bdr (6371553825031594764,1,42892,):receive DETAIL: streaming transactions committing
after 1/194FB9D0, reading WAL from 1/194FB700
bdr (6371553825031594764,1,42892,):receive LOG: logical decoding found consistent point
at 1/194FB700
bdr (6371553825031594764,1,42892,):receive DETAIL: There are no running transactions.
bdr (6371553825031594764,1,42892,):receive LOG: unexpected EOF on standby connection
```

※ 各ノードで制約違反が発生し、BDRによる接続が切断されてしまう。解消のため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する

制約違反の状態を解消するために、両側のデータをUPDATEします。

```
(node1のデータベースにSQLを投入)
=# UPDATE sample SET range = '[2012-04-11,2012-04-16]' WHERE aid = 1;
(node2のデータベースにSQLを投入)
=# UPDATE sample SET range = '[2012-04-23,2012-04-29]' WHERE aid = 2;
```

```
(node1のデータベースを確認)
=# SELECT * FROM sample;
 aid |          range          | price
-----+-----+-----
   1 | [2012-04-11,2012-04-17) | 10000
   2 | [2012-04-23,2012-04-30) | 10000
(2 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM sample;
 aid |          range          | price
-----+-----+-----
   2 | [2012-04-23,2012-04-30) | 10000
   1 | [2012-04-11,2012-04-17) | 10000
(2 行)
```

5.5.4.4.4. グローバルなデータ

1. グローバルなデータの競合

■ 競合概要

ロール(グローバルデータ)の情報がノード間で異なる状態で、他のノードに存在しないロールを利用した場合に発生する競合のケースを検証しました。

■ 検証結果

レプリケーション先のノードに同名のロールが存在しない場合、エラーになります。エラーを解消するには、オペレータ側での操作が必要となるため注意が必要です。

■ 競合発生手順

新規に作成したロールでテーブルを作成します。

```
(node1のデータベースにSQLを投入)
=# CREATE ROLE testuser SUPERUSER LOGIN;
=# \c testuser
=# CREATE TABLE test01 (id integer primary key);
=# \d
```

スキーマ	名前	型	所有者
public	test01	テーブル	testuser

(1 行)

```
(node2のデータベースにSQLを投入)
=# \d
リレーションがありません。

※ ロールなどはレプリケーション対象とならないので、そのロールで作成されたテーブルもレプリケーション対象とならない
```

データベースログを確認し、ERRORが継続して出力されていれば競合が発生しています。

(node2のデータベースログ)

```
LOG:  starting background worker process "bdr (6371561413984311673,2,42879,)->bdr
(6371553825031594764,1,"
ERROR:  role "testuser" does not exist
CONTEXT:  during DDL replay of ddl statement: CREATE TABLE public.test01 (id
pg_catalog.int4 , CONSTRAINT test01_pkey PRIMARY KEY (id) ) WITH (oids=OFF)
        apply QUEUED_DDL in commit 1/15DD2278, xid 3958054 committed at 2017-01-06
17:37:58.608995+09 (action #2) from node (6371553825031594764,1,42892)
LOG:  worker process: bdr (6371561413984311673,2,42879,)->bdr (6371553825031594764,1,
(PID 30136) exited with exit code 1
```

※ 権限は関係なく、同名のロールが作成されれば、競合は解決する

レプリケーションを再開させるためにロールなどのグローバルデータを作成します。

(node2のデータベースにSQLを投入)

```
=# CREATE ROLE testuser LOGIN;
=# \d
```

リレーションの一覧

スキーマ	名前	型	所有者
public	test01	テーブル	testuser

(1 行)

5.5.4.4.5. ロックの競合とデッドロックの中断

1. ロックの競合とデッドロックの中断

■ 競合概要

一つのノードで取得したロックともう片方のノードで取得したロックが、デッドロック状態となるケースを検証しました。

■ 検証結果

異なるノードで実行されたトランザクションでデッドロック状態が発生した場合、ロックが解除されるまでロック待ちが発生します。PostgreSQLにおいてデッドロックが発生した場合、デッドロックを引き起こすトランザクションが自動でロールバックされますが、異なるノード間で発生したデッドロックについては検出できないため、ロック待ち状態が続きます。

■ 競合発生手順

下記テーブルを初期状態とします。

(node1のデータベースにSQLを投入)

```
=# CREATE TABLE sample (aid integer primary key, range daterange, price integer);
=# ALTER TABLE sample ADD EXCLUDE USING gist (price WITH =, range WITH &&);
=# INSERT INTO sample VALUES(1, '[2012-04-16, 2012-04-20]', 10000);
=# INSERT INTO sample VALUES(2, '[2012-04-21, 2012-04-30]', 12000);
```

```
(node1のデータベースを確認)
=# SELECT * FROM sample;
 aid |          range          | price
-----+-----+-----
   1 | [2012-04-16,2012-04-21) | 10000
   2 | [2012-04-21,2012-05-01) | 12000
(2 行)
```

```
(node2のデータベースを確認)
=# SELECT * FROM sample;
 aid |          range          | price
-----+-----+-----
   1 | [2012-04-16,2012-04-21) | 10000
   2 | [2012-04-21,2012-05-01) | 12000
(2 行)
```

1つのノードからテーブルをロックし、もう片方のノードからはUPDATEを実行し、ロック待ちの影響を受ける事を確認します。ロックが解除されるまで、BDR適用プロセスはロック待ちが発生します。

```
(node1のデータベースにSQLを投入)
=# BEGIN;
=# LOCK TABLE sample IN ACCESS EXCLUSIVE MODE;
=# SELECT pg_sleep(10);
=# END;
```

```
(node2のデータベースにSQLを投入)
=# UPDATE sample SET price = 14000 WHERE aid = 2;
```

```
(ロック待ちなので競合ログが出力されない)
=# \x
=# SELECT * FROM bdr.bdr_conflict_history ORDER BY conflict_id DESC LIMIT 1;
```

5.5.4.4.6. その他

自動的には解決出来ないデータの相違が発生した場合は、以下設定を使用して手動で調整する必要があります。

※ レプリケーション環境を破壊することが可能であるため使用する際には注意が必要です。

項番	パラメータ	説明
1	bdr.do_not_replicate(boolean)	このパラメータセットを持つトランザクションで行われた変更は、他ノードへのレプリケーションのためにキューに格納されない。

```
=# CREATE TABLE test (id int primary key, name text);
=# \q
```

```

(node1に対してのみレコード追加を実行)
$ export PGOPTIONS='-c bdr.do_not_replicate=on'
$ psql -h node1 bdrtest
=# INSERT INTO test values(1, 'test');
=# SELECT * from test;
   id | name
-----+-----
    1 | test
(1 行)

(node2には反映されていないことを確認)
=# SELECT * from test;
   id | name
-----+-----
(0 行)

```

項番	パラメータ	説明
2	bdr.skip_ddl_replication(boolean)	DDLによる変更のレプリケーションをスキップする。 一部のノードのみにDDLを実行したい場合に有効。 スーパーユーザのみが設定可能。

```

(node1にのみテーブルを作成)
=# BEGIN;
=# SET LOCAL bdr.skip_ddl_replication = true;
=# CREATE TABLE skip (id int primary key, name text);
=# END;
=# \d
          リレーションの一覧
 スキーマ | 名前 | 型 | 所有者
-----+-----+-----+-----
 public  | sample | テーブル | postgres
 public  | skip | テーブル | postgres
 public  | test | テーブル | postgres
(3 行)

(node2には反映されていないことを確認)
=# \d
          リレーションの一覧
 スキーマ | 名前 | 型 | 所有者
-----+-----+-----+-----
 public  | sample | テーブル | postgres
 public  | test | テーブル | postgres
(2 行)

```

項番	パラメータ	説明
3	bdr.permit_unsafe_ddl_commands(boolean)	安全にレプリケーションできないスキーマの変更を許可する。 スーパーユーザのみが設定可能。 以下のようなデフォルトでは実行が許可されていないウエリを実行可能。 * CREATE TABLE AS * ALTER TABLE ... ADD COLUMN ... DEFAULT * CREATE MATERIALIZED VIEW * REFRESH MATERIALIZED VIEW

```
(デフォルトではエラーとなる)
=# CREATE TABLE test2 AS SELECT * from test;
ERROR: CREATE TABLE AS is not supported when bdr is active

(エラーが出ないことを確認)
=# BEGIN;
=# SET LOCAL bdr.permit_unsafe_ddl_commands = true;
=# CREATE TABLE test2 AS SELECT * from test;
=# END;
```

5.5.4.5. 検証結果

一部の競合パターンにて、意図しない動作が発生するため、現段階では競合が発生しないパターンで利用すべきと考えます。

5.5.5. ノード障害と復旧

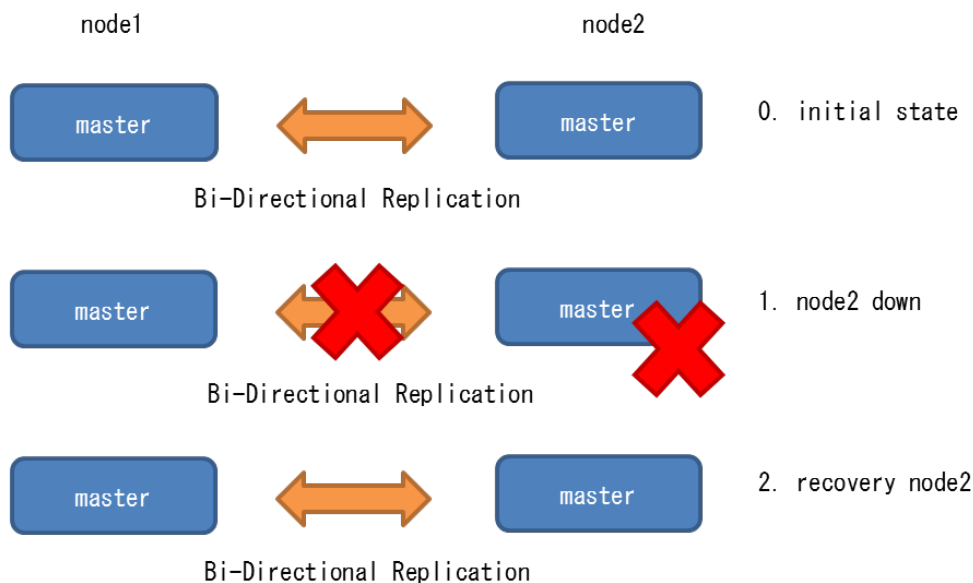
5.5.5.1. 検証の目的

複数ノードで構成されるクラスタ環境内の1ノードに障害が発生した場合でも、他ノードで継続利用可能か否かを確認します。また、障害が発生したノードをクラスタ環境に復旧させる手順を確認します。

5.5.5.2. 検証内容

本検証では2台で構成されたBDRクラスタに対して、下記を実施しました。

1. ノード障害
2. ノード復旧



5.5.5.3. 検証環境

[こちらの環境](#) で検証を実施しました。

5.5.5.4. 検証手順

5.5.5.4.1. ノード障害

1. BDRの各ノード状態確認

BDRの各ノード間のレプリケーションが正常に動作していることを「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1にて実施)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name      | node1
node_local_dsn | host=node1 port=5432 dbname=bdrtest
node_status    | r
<-- r(正常)であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name      | node2
node_local_dsn | host=node2 port=5432 dbname=bdrtest
node_status    | r
<-- r(正常)であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest
<-- node1から作成されていることを確認
```

2. 検証用テーブルとデータの作成

pgbenchを利用して、検証時に利用するテーブルとデータを作成します。

```
(node1にて実施)
$ pgbench -i -s 10 bdrtest
```

3. トランザクション実行

pgbenchを利用して、ノード1にトランザクションを継続的に実行します。ノード2に障害が発生した場合に、ノード1に対して実行したトランザクションにエラーが発生するか否かを確認します。

```
$ pgbench -h node1 -c 10 -t 100000 bdrtest
starting vacuum...end.
```

4. ノード障害

ノード2で動作するPostgreSQLを停止させます。

```
(node2にて実施)
$ pg_ctl stop -m i
waiting for server to shut down..... done
server stopped
```

5. ノード1のログ確認

ノード2障害時に出力されるログメッセージを確認します。下記メッセージは出力されますが、pgbenchのトランザクションは継続して実行可能です。

```
$ tail -f [PostgreSQLのログファイル]
LOG:  starting background worker process "bdr (6367633348875313343,1,34478,)->bdr
(6369931070716042622,2,"
ERROR:  establish BDR: could not connect to server: Connection refused
Is the server running on host "node2" (192.168.1.3) and accepting
TCP/IP connections on port 5432?
```

6. BDRの各ノード状態確認

ノード2障害後、ノード状態が変化するか否かを「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | r               <-- r (正常) の状態のまま
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest
※ 状態変化は確認できませんでした。
```

7. ノードの切り離し

障害が発生したノード2を切り離しするため、「[bdr.bdr_part_by_node_names](#)」関数を実行します。

```
(node1のデータベースに接続)
=# SELECT bdr.bdr_part_by_node_names (ARRAY['node2']);
bdr_part_by_node_names
-----
(1 row)
```

8. ノードの切り離し結果確認

ノードの切り離し結果を「[bdr.bdr_nodes](#)」テーブルを用いて確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn , node_status, node_init_from_dsn FROM
bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | k               <-- k (削除) であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest
```

9. システムカタログの残データ削除

本手順は推奨される手順ではありませんが、削除状態のノードが「[bdr.bdr_nodes](#)」テーブルに残っているとノード復旧が実行できないため暫定的な対処です。BDRのIssuesで「[bdr.bdr_connections](#)」のデータも削除する事が提案されていたので、こちらも暫定的な手順ですが実行します。詳細は下記をご参照下さい。

- <https://github.com/2ndQuadrant/bdr/issues/97>
- <https://github.com/2ndQuadrant/bdr/issues/126>

```
(node1のデータベースに接続)
=# DELETE FROM bdr.bdr_connections USING bdr.bdr_nodes WHERE node_status = 'k' AND"
(node_sysid, node_timeline, node_dboid) = (conn_sysid, conn_timeline, conn_dboid);
=# DELETE FROM bdr.bdr_nodes where node_status = 'k';
```

10. トランザクション状態実行状態確認

手順1で実行したpgbenchにエラーが発生していないことを確認します。本検証ではエラーは発生しませんでした。

5.5.5.4.2. ノード復旧

BDRノードの復旧方法(ノード追加)する場合、既存ノードのデータベースと復旧させるノードのデータベースのスキーマおよびデータを同期させる必要があります。ノード間のデータコピーには、論理コピーと物理コピーの2つの手法があります。

- <http://bdr-project.org/docs/stable/functions-node-mgmt.html#FUNCTION-BDR-GROUP-JOIN>

表 5.8 コピー取得方法

項番	コピー取得	説明	備考
1	bdr.bdr_group_join 関数実行	ユーザが指定したノード内データベースのスキーマとデータダンプを取得	pg_dumpコマンドに相当
2	bdr_init_copyコマンド	ユーザが指定したノード上の全てのデータベースのコピーを取得	pg_basebackupコマンドに相当

■ 論理コピーによる復旧

1. トランザクション実行

pgbenchを利用して、ノード1にトランザクションを継続的に実行します。ノード2に復旧する際に、ノード1に対して実行したトランザクションの停止が必要か否かを確認します。

```
$ pgbench -h node1 -c 10 -t 100000 bdrtest
starting vacuum...end.
```

2. ノード2のPostgreSQL起動

ノード2上で動作するPostgreSQLを起動させます。

```
$ pg_ctl start
```

3. BDRの無効化

ノード2のデータベースからBDRを削除し、BDR拡張機能を削除します。

```
(node2のデータベースに接続)
=# SELECT bdr.remove_bdr_from_local_node(true);
WARNING:  forcing deletion of possibly active BDR node
NOTICE:   removing BDR from node
NOTICE:   BDR removed from this node. You can now DROP EXTENSION bdr and,
if this is the last BDR node on this PostgreSQL instance, remove bdr from
shared_preload_libraries.
remove_bdr_from_local_node
-----

=# DROP EXTENSION bdr;
DROP EXTENSION
```

4. データベース削除

BDRで利用したデータベースを削除します。

```
(node2のデータベースに接続)
=# DROP DATABASE bdrtest ;
DROP DATABASE
```

※ 接続が残っており、削除できない場合はPostgreSQLを再起動

5. データベースの再作成

BDRで利用するデータベースを再度作成します。

```
(node2のデータベースに接続)
=# CREATE DATABASE bdrtest;
```

6. BDR有効化

無効化したBDRを再度有効化します。

```
(node2のデータベースに接続)
=# CREATE EXTENSION btree_gist;
CREATE EXTENSION
=# CREATE EXTENSION bdr;
CREATE EXTENSION
```

7. ノードの追加

ノードを追加(復旧)させるため、「[bdr.bdr_group_join](#)」関数を実行します。

```
(node2のデータベースに接続)
=# SELECT bdr.bdr_group_join(
    local_node_name := 'node2',
    node_external_dsn := 'host=node2 port=5432 dbname=bdrtest',
    join_using_dsn := 'host=node1 port=5432 dbname=bdrtest'
);
bdr_group_join
-----
(1 行)
```

8. ノード追加の確認待ち

ノードが追加されたことを確認するため、「[bdr.bdr_node_join_wait_for_ready](#)」関数を実行します。

```
(node2のデータベースに接続)
=# SELECT bdr.bdr_node_join_wait_for_ready();
※ トランザクションが実行中の場合、上記関数の結果が戻りません。
```

node1のPostgreSQLログファイルに下記メッセージが出力された後、復旧の処理が開始されません。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found initial starting point at 0/BB399BF0
DETAIL:  10 transactions need to finish.
```

pgbenchコマンドで実行中のトランザクションを停止すると、下記ログメッセージが出力され、復旧処理が開始されます。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found consistent point at 0/B2895648
DETAIL:  There are no running transactions.
LOG:  exported logical decoding snapshot: "00046FE0-1" with 0 transaction IDs
```

9. BDRの各ノード状態確認

ノード2が追加されたことを「[bdr.bdr_nodes](#)」テーブルの情報から確認します。

```
(node1のデータベースに接続)
=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn     | host=node1 port=5432 dbname=bdrtest
node_status        | r               <-- r (正常)であることを確認
node_init_from_dsn |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn     | host=node2 port=5432 dbname=bdrtest
node_status        | r               <-- r (正常)であることを確認
node_init_from_dsn | host=node1 port=5432 dbname=bdrtest      <-- node1から作成されていることを確認
```

■ 物理コピーによる復旧

1. トランザクション実行

pgbenchを利用して、ノード1にトランザクションを継続的に実行します。ノード2に復旧する際に、ノード1に対して実行したトランザクションの停止が必要か否かを確認します。

```
$ pgbench -h node1 -c 10 -t 100000 bdrtest
starting vacuum...end.
```

2. ノード2のPostgreSQL停止確認

ノード2上で動作するPostgreSQLが停止していることを確認します。

```
(node2にて実施)
$ pg_ctl status
pg_ctl: no server running
```

3. 物理コピーの取得

ノード2上で「bdr_init_copy」コマンドを実行し、node1上のコピーを取得します。

```
(node2にて実施)
$ rm -rf $PGDATA/*
$ bdr_init_copy -D $PGDATA -n node2 -h node1 -p 5432 -d bdrtest
--local-host=node2 --local-port=5432 --local-dbname=bdrtest
bdr_init_copy: starting ...
Getting remote server identification ...
Detected 1 BDR database(s) on remote server
Updating BDR configuration on the remote node:
bdrtest: creating replication slot ...
bdrtest: creating node entry for local node ...
Creating base backup of the remote node...
50357/50357 kB (100%), 1/1 tablespace
Creating restore point on remote node ...
Bringing local node to the restore point ...
トランザクションログをリセットします。
Initializing BDR on the local node:
bdrtest: adding the database to BDR cluster ...
All done
```

node1のPostgreSQLログファイルに下記メッセージが出力された後、復旧の処理が開始されません。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found initial starting point at 0/BB399BF0
DETAIL:  10 transactions need to finish.
```

pgbenchコマンドで実行中のトランザクションを停止すると、下記ログメッセージが出力され、復旧処理が開始されます。

```
(node1のログメッセージ抜粋)
LOG:  logical decoding found consistent point at 0/B2895648
DETAIL:  There are no running transactions.
STATEMENT:  SELECT
pg_create_logical_replication_slot('bdr_25434_6369931070716042622_2_25434__', 'bdr');
```

4. BDRの各ノード状態確認

ノード2が追加されたことを「[bdr.bdr_nodes](#)」テーブルの情報から確認します。

```
(node1のデータベースに接続)
bdrtest=# SELECT node_name, node_local_dsn, node_status, node_init_from_dsn FROM
bdr.bdr_nodes;
-[ RECORD 1 ]-----+-----
node_name          | node1
node_local_dsn      | host=node1 port=5432 dbname=bdrtest
node_status         | r              <-- r(正常)であることを確認
node_init_from_dsn  |
-[ RECORD 2 ]-----+-----
node_name          | node2
node_local_dsn      | host=node2 port=5432 dbname=bdrtest
node_status         | r              <-- r(正常)であることを確認
node_init_from_dsn  | host=node1 port=5432 dbname=bdrtest          <-- node1から作
成されていることを確認
```

5.5.5.5. 検証結果

ノード障害が発生した際にも、他のノードではトランザクションを継続実行することが可能でした。障害ノードを復旧させるためにはデータ操作が禁止されているシステムカタログのデータ削除が必要でした。

5.5.5.6. 備考

ノード障害後および復旧中にDDLを実行すると下記のエラーが出力されます。

```
=# CREATE TABLE test2(id int);
ERROR:  No peer nodes or peer node count unknown, cannot acquire global lock
HINT:   BDR is probably still starting up, wait a while
```

5.6. BDR性能検証

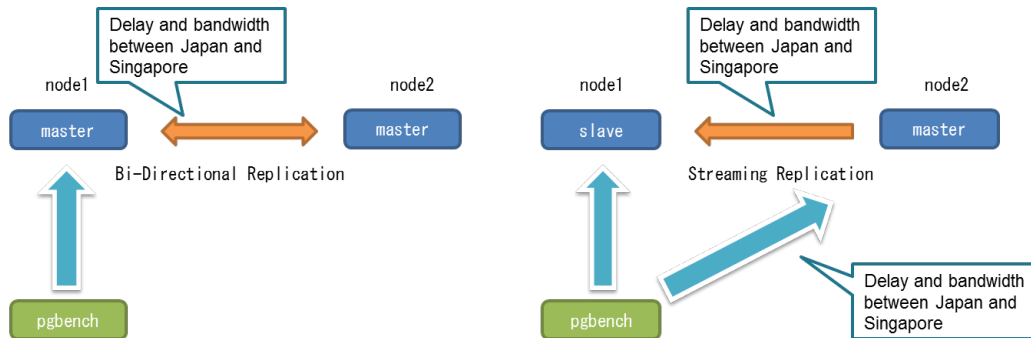
5.6.1. 更新性能検証結果

5.6.1.1. 検証目的

本試験はBDRの本来のユースケースである、高レイテンシ環境との双方向レプリケーション環境における更新クエリのレスポンス改善とそれに伴う性能改善を確認したものです。

5.6.1.1.1. 検証内容

今回は [2014年度検証報告書（可用性編）](#) に記載された東京-シンガポール間の回線情報（応答速度76.95ms、帯域幅0.16Gbits/s）をもとに、以下のような環境を構築しpgbenchを実行しました。



通常のストリーミングレプリケーションであれば、更新クエリを実行したい場合はシンガポールのマスタノードに更新要求を実施する必要がありますが、BDRでは最寄りのノードに対して更新要求を実施することが可能です。

5.6.1.1.2. 検証環境

[こちらの環境](#) で検証を実施しました。

5.6.1.1.3. 検証手順

1. 環境構築

pgbenchを実行する準備を行います。(BDR, SR環境両方)

```
# pgbench -i test -U postgres -s 10
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
creating tables...
100000 of 100000 tuples (100%) done (elapsed 0.25 s, remaining 0.00 s).
vacuum...
set primary keys...
done.
```

node1, node2ともにテーブルが作成されていることを確認します。

```
# psql -U postgres test -c "\d"
リレーションの一覧
 スキーマ |      名前      | 型   | 所有者
-----+-----+-----+-----
 public   | pgbench_accounts | テーブル | postgres
 public   | pgbench_branches | テーブル | postgres
 public   | pgbench_history  | テーブル | postgres
 public   | pgbench_tellers  | テーブル | postgres
(4 行)
```

```
# /usr/pgsql-9.4/bin/psql -U postgres -h node2 test -c "\d"
リレーションの一覧
 スキーマ |      名前      | 型   | 所有者
-----+-----+-----+-----
 public   | pgbench_accounts | テーブル | postgres
 public   | pgbench_branches | テーブル | postgres
 public   | pgbench_history  | テーブル | postgres
 public   | pgbench_tellers  | テーブル | postgres
(4 行)
```

帯域制限及び遅延設定を行います。

```
(clientはnode2向け通信のみ制限)
tc qdisc add dev eth0 root handle 1:0 htb
tc class add dev eth0 parent 1:0 classid 1:10 htb rate 160Mbit
tc filter add dev eth0 protocol ip parent 1:0 prio 1 u32 match ip dst 192.168.0.12/32
flowid 1:10
tc qdisc add dev eth0 parent 1:10 handle 10:1 netem delay 38ms 1ms

(node1はnode2向け通信のみ制限)
tc qdisc add dev enp0s25 root handle 1:0 htb
tc class add dev enp0s25 parent 1:0 classid 1:10 htb rate 160Mbit
tc filter add dev enp0s25 protocol ip parent 1:0 prio 1 u32 match ip dst 192.168.0.12/32
flowid 1:10
tc qdisc add dev enp0s25 parent 1:10 handle 10:1 netem delay 38ms 1ms

(node2は両ノード向け通信とも制限)
tc qdisc add dev enp0s25 root handle 1:0 htb
tc class add dev enp0s25 parent 1:0 classid 1:0 htb rate 160Mbit
tc qdisc add dev enp0s25 parent 1:0 handle 10:1 netem delay 38ms 1ms
```

帯域制限の結果以下のような環境になります。

表 5.9 帯域制限後のネットワーク情報

Client/Server	client	node1	node2
client	—	遅延:0.5 ms 帯域幅:934 Mbits/s	遅延:76ms 帯域幅:160 Mbits/s
node1	遅延:0.5 ms 帯域幅:934 Mbits/s	—	遅延:76ms 帯域幅:160 Mbits/s
node2	遅延:76ms 帯域幅:160 Mbits/s	遅延:76ms 帯域幅:160 Mbits/s	—

2. レスポンスタイム確認

更新クエリの応答速度を確認します。

```
# time psql -h node1 -x -c "UPDATE pgbench_branches SET bbalance = bbalance + 100 WHERE
bid = 1" test
UPDATE 1

real    0m0.025s
user    0m0.001s
sys     0m0.001s

# time psql -h node2 -x -c "UPDATE pgbench_branches SET bbalance = bbalance + 100 WHERE
bid = 1" test
UPDATE 1

real    0m0.259s
user    0m0.001s
sys     0m0.000s
(ネットワーク遅延に伴いレスポンスが低下していることを確認)
```

3. SR環境の性能試験

非同期SR構成のマスタに対してpgbenchを実施します。

```
# pgbench -U postgres -p 5433 -h node2 -s 10 -c 10 test -T 180
scale option ignored, using count from pgbench_branches table (10)
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
duration: 180 s
number of transactions actually processed: 3078
latency average: 584.795 ms
tps = 17.042442 (including connections establishing)
tps = 17.057455 (excluding connections establishing)
```

4. BDR環境の性能試験

競合が発生しているか確認するために、「[bdr.bdr_conflict_history](#)」を参照し競合履歴数を確認します。

```
# psql -h node1 -x -c "SELECT count(*) FROM bdr.bdr_conflict_history;" test
-[ RECORD 1 ]
count | 52265

# psql -h node2 -x -c "SELECT count(*) FROM bdr.bdr_conflict_history;" test
-[ RECORD 1 ]-
count | 132935
```

BDR環境のノードにpgbenchを実施します。

```
# pgbench -U postgres -h node1 -s 10 -c 10 test -T 180
scale option ignored, using count from pgbench_branches table (10)
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 10
query mode: simple
number of clients: 10
number of threads: 1
duration: 180 s
number of transactions actually processed: 16677
latency average: 107.933 ms
tps = 92.598796 (including connections establishing)
tps = 92.602290 (excluding connections establishing)
```

競合が発生していないことを確認します。

```
# psql -h node1 -x -c "SELECT count(*) FROM bdr.bdr_conflict_history;" test
-[ RECORD 1 ]
count | 52265

# psql -h node2 -x -c "SELECT count(*) FROM bdr.bdr_conflict_history;" test
-[ RECORD 1 ]-
count | 132935
```

5.6.1.1.4. 検証結果

本検証では以下を確認することが出来ました。

- レスポンスタイムの低減

低レイテンシのサーバに更新クエリを実行可能であるため、レスポンスタイムを低減することが出来ました。

- TPSの向上

レスポンスタイムが低減されたため、TPSの向上が見られました。
 "17.057455" -> "92.602290" (約5.4倍)

ただし、本検証では片系のみに変更を実施し競合が発生しないようにした試験だったため、このような結果になったものと思われます。

本検証で使用しているpgbench (TCP-Bライク) のような、現在の値に対して加算していくような処理の場合、結果整合で競合を解決することができないため、絶対に競合が発生しない構成が必要です。

例) 更新するテーブルを拠点ごとに分ける等

5.7. まとめ

5.7.1. BDR検証まとめ

本検証では、BDRの機能や特徴および主なユースケースを机上調査を実施した上で、BDRの動作検証および性能検証を実施しました。本検証で実施したBDRの動作検証の結果は下表の通りです。

表 5.10 結果評価基準

記号	意味
○	問題なし。
△	利用時に問題になるケースがある。
×	対応していない。事実上使えない。

表 5.11 BDR検証結果

項番	検証概要	結果	補足
1	ノード追加/削除	△	ノード削除はオンラインで実行可能。ノード追加時にトランザクションの停止が必要。また、削除したノードを追加する場合にはシステムカタログの操作が必要。
2	グローバルシーケンス	○	シーケンスの競合を防ぐことが可能。ただし、マニュアルに記載された制限事項については確認が必要。
3	選択的レプリケーション	○	BDRを利用して任意のテーブルのデータ集約等を実現可能
4	更新処理競合時の動作	△	更新が競合するパターンで意図しない動作が発生し、競合解消のため手動での操作が必要。
5	ノード障害と復旧	△	ノード復旧時にはトランザクションの停止が必要。また、障害が発生したノードを復旧させる際にシステムカタログの操作が必要。

本検証で利用したバージョン(1.0.2)では更新が競合するパターンで意図しない動作が発生するため、BDRを適用する場合、競合が発生しないようなアプリケーション設計やテーブル設計が必要になります。

例) 更新するテーブルを拠点ごとに分ける等

ノード障害時の運用においても、一般的でないシステムカタログの操作を必要とするといった今後の改善が必要と思われる結果が確認されました。

また、BDRの選択的レプリケーションを用いることで、PostgreSQLのストリーミング・レプリケーションでは実現できない、テーブル単位のレプリケーションが実現可能なことを確認できました。上記機能を用いて、システム間のデータ連携(データ集約等)を柔軟に実現できると考えております。

性能検証の結果より、ユースケースで想定している「遠距離拠点間で双方向に更新する」場合に、レスポンスタイムの低減と処理向上につながる事が確認できました。

机上で調査した通り、BDRを適用することで「遠距離拠点間のトランザクション性能改善」や選択的レプリケーション機能を利用した「柔軟なデータ連携」を実現できると考えます。

5.8. 参考文献

[BDR] Postgres-BDR ドキュメント <http://bdr-project.org/docs/stable/>

5.8.1. 関数一覧

表 5.12 関数一覧

項番	関数	参照ドキュメント
1	bdr.bdr_group_create	functions-node-mgmt.html
2	bdr.bdr_group_join	functions-node-mgmt.html
3	bdr.bdr_part_by_node_names	functions-node-mgmt.html
4	bdr.remove_bdr_from_local_node	functions-node-mgmt.html
5	bdr.bdr_node_join_wait_for_ready	functions-node-mgmt.html
6	bdr.table_set_replication_sets	functions-replication-sets.html
7	bdr.table_get_replication_sets	functions-replication-sets.html
8	bdr.connection_get_replication_sets	functions-replication-sets.html

5.8.2. システムカタログ一覧

表 5.13 システムカタログ一覧

項番	システムカタログ	参照ドキュメント
1	bdr.bdr_conflict_history	catalog-bdr-conflict-history.html
2	bdr.bdr_nodes	catalog-bdr-nodes.html
3	bdr.bdr_connections	catalog-bdr-connections.html

6. まとめ

本文書では、PostgreSQLにおけるレプリケーション技術として、標準機能として組み込まれているストリーミングレプリケーションと論理レプリケーション機能を活用してマルチマスタ構成に対応したBi-Directional Replication(BDR)を取り上げました。

ストリーミングレプリケーションは、2010年9月にリリースされたPostgreSQL 9.0で標準機能として組み込まれ、徐々に実運用での利用される機会が増えるとともに、実運用で必要となる機能改善がその後のバージョンアップで継続的に行われています。

今回の検証では、ストリーミングレプリケーション構成を構築する基本的な手順を確認した上で、レプリケーションスロット、遅延レプリケーション、WAL圧縮といった新しい機能に対しても、設定から実際の動作を確認するまでの具体的な手順を整理することができました。また、障害発生時の運用についてもPostgreSQL 9.5から提供されるようになったpg_rewindコマンドを使うことで、これまで必要とされていたフルバックアップ取得を不要とする運用が可能になっていることを確認できました。

BDRは、現状のストリーミングレプリケーションでは不可能なマルチマスタ構成に対応することから、シングルマスタ構成に起因する課題解消が期待されます。今回の検証では、BDRの特徴、ユースケース、メカニズムを類似機能、製品と机上比較した上で、更新処理競合時の動作、ノード障害時の動作、更新性能についてそれぞれ検証しました。更新性能の検証結果より、ユースケースで想定している「遠距離拠点間で双方向に更新する」場合に、レスポンスタイムの低減とTPS向上につながる事がわかりました。

一方で、現時点のバージョンでは更新が競合するパターンで意図しない動作が発生するため、現段階では競合が発生しないパターンで利用すべきと考えます。そこで、BDRの利用時は競合が発生しにくいテーブル設計にすることが望ましいです。また、ノード障害時の運用においても、一般的でないシステムカタログの操作を必要とするといった今後の改善が必要と思われる結果が確認されました。

これらの結果から、BDRについてはユースケースと実際の利用用途が合致しているかを見極めた上で利用するかどうかを判断する必要があるといえます。

今回の検証結果がストリーミングレプリケーション構成を実際に利用している方々の運用改善につながる事、またBDRを利用すべきかどうかを判断する際の参考情報として活用いただけることを期待しています。

7. 著者

(企業・団体名順)

版	所属企業・団体名	部署名	氏名
第1.0版 (2016年度WG3)	株式会社アシスト	データベース技術本部	竹内 尚也
	株式会社アシスト	データベース技術本部	柘植 丈彦
	株式会社オージス総研	プラットフォームサービス本部 IT基盤技術部	大西 斉
	TIS株式会社	IT基盤技術本部 IT基盤技術推進部	中西 剛紀
	日本電信電話株式会社	オープンソースソフトウェアセンタ	坂田 哲夫
	株式会社富士通ソーシャルサイエンスラボラトリ	プラットフォームインテグレーション本部 第四システム部	小山田 政紀
	株式会社富士通ソーシャルサイエンスラボラトリ	プラットフォームインテグレーション本部 第四システム部	高橋 勝平
	株式会社富士通ソーシャルサイエンスラボラトリ	プラットフォームインテグレーション本部 第四システム部	香田 紗希