PGECons
PostgreSQL Enterprise Consortium

PostgreSQL エンタープライズ・コンソーシアム 技術部会 WG#3
設計運用ワーキンググループ（WG3）

2014 年度 WG3 活動報告書

－ 可用性編 －

改訂履歴

版	改訂日	変更内容
1.0	2015/04/23	新規作成

はじめに

■PostgreSQL エンタープライズコンソーシアムと WG3 について

エンタープライズ領域における PostgreSQL の普及を目的として設立された PostgreSQL エンタープライズコンソーシアム（以降 PGECons）では、技術部会における PostgreSQL の普及に対する課題の検討を通じて 2014 年度の活動テーマを挙げ、その中から 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（性能ワーキンググループ）
- WG2（移行ワーキンググループ）
- WG3（設計運用ワーキンググループ）

WG3 では、2013 年度の活動成果であるエンタープライズ領域での PostgreSQL の典型的システム方式の調査と動作検証に続き、2014 年度はより広い視野の可用性、セキュリティの観点からシステム方式の調査と動作検証を行い、技術ノウハウを整理してきました。

■本資料の概要と目的

本資料は 2014 年度の WG3 における活動として、PostgreSQL におけるサイト障害に対応するシステム方式について整理し、一部の構成について動作確認を行ったものです。

■本資料の構成

1章. 事業継続と IT サービス
　IT サービス継続の考え方と重要な指標値である復旧目標について記載しています。
2章. DR 要件を実現する PostgreSQL の代表的なシステム構成
　サイト障害に対応する PostgreSQL の典型的なシステム方式について整理しています。
3章. 運用技術検証
　着目したシステム構成に対する運用手順を確認し、レプリケーションの視点から動作検証を行います。
4章. おわりに

■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

目次

1. 事業継続と IT サービス

現代の社会経済は、情報技術（以下 IT）やネットワーク技術の発達・低コスト化により、さまざまな恩恵を享受しており、企業においても本格的な活用が進んでいます。

1.1. 経済活動を支える IT システム

現代の社会において IT を活用することは不可欠となっており、もはや IT を活用したシステム（以下 IT システム）は企業の事業や地域の経済活動を支える基盤といえます。さらに、ソフトウェアやネットワーク技術の発達・低コスト化により、IT システムの大規模化、複雑化が進んでおり、1 つの IT システムが複数の企業や地域の経済活動を支えるようになってきています。

そのため、故障や災害により IT システムが停止することは、複数の企業や広い地域の経済活動に大きな影響を及ぼす恐れがあります。

1.2. IT サービス継続

組織もしくは企業の業務遂行のために必要なサービスは IT システムとそれに関連する体制を組合せて実現されます。ここではそれを IT サービスと呼びます。

IT サービス継続とは、IT サービスが停止・中断したり、機能停止や性能低下が事業継続に与える影響を軽減する取り組みで、要求されるサービスレベルに応じて最適な方策を選択する必要があります。

IT サービス継続を実現するには IT に関する中長期的投資計画や体制整備が必要であり、結果として IT サービス継続は IT 戦略の一部と位置づけられるものとなります。

1.3. ディザスタリカバリの重要性

ディザスタリカバリは自然災害などで被害を受けたシステムを復旧・修復すること、また、そのための備えとなる機器やシステム、体制のことを指し、Disaster Recovery（以下、DR）と称されます。

システムを災害から守ることももちろん重要ですが、災害や故障に起因するトラブルは「必ず起こりうる」ことを想定し、いざというときに効率よくかつ迅速に IT システムを復旧するという観点から対策を練ることも重要です。

ここで大事なのは、システムの規模や特性に応じて、IT システムを復旧するには、それ相応のコストが掛かることを理解しておくことです。しかしながら、まだ起こっていない、いつ起きるかどうかわからない災害や故障に対して、どこにどの程度の投資を行うかを平常時に判断するのはとても難しい問題です。

事前にしっかりと対応手順や対策を講じておくことで、実際に災害や故障が発生した場合の事業継続リスクと復旧コストを低く抑えることが可能になります。逆に、事前の対策を採らずに事後の対応となった場合、取り得る対策の選択肢が限定され、かつ復旧もしくは事業継続を断念せざるを得ない程の高額なコストが必要となる場合も考えられます。

1.4. DR を検討する指標

災害などのトラブルにより、エンタープライズ級の IT システムが停止した場合を想定して事業や経営への影響を評価する際にDRの検討指標となる、「復旧目標」の考え方について簡単に整理します。

復旧目標には以下の3つの指標があります。

復旧時間目標：RTO（Recovery Time Objective）

復旧作業にあたり、故障もしくは災害発生後、何時間後もしくは何日後までにシステムを再稼働するかを目標とする指標値です。言い換えれば、システムの停止時間（ダウンタイム）と同じ意味となります。ただし、単一システムの運用の場合は単純ですが、複数のシステムが NW を介して連携する複雑なシステムを構成している場合は、実質的な業務の停止時間はさらに延びる可能性があります。

このように、システム停止もしくは機能低下が、許容される時間の限界より短くなるよう目標設定されるべきです。

復旧レベル目標:RLO(Recovery Level Objective)

故障もしくは災害発生前のシステム機能や性能が、復旧後にどのくらい低下するのか、もしくは以前のまま100%のレベルを保つようにするのかを目標とする指標値です。システム復旧時にすべての機能を復旧させるのか、一部の機能は停止させるのか、それとも性能(処理能力)は災害発生前と同じなのか、低コストの機器やNWを活用して復旧するため、処理能力は若干犠牲にするのか、を目標として設定することになります。
このように、機能や性能低下のレベルに対し、許容限界を上回るよう目標設定されるべきです。

復旧時点目標:RPO(Recovery Point Objective)

復旧作業にあたり、故障もしくは災害発生前のどの時点(ポイント)にシステムもしくはデータを戻すかを目標とする指標値です。システムが復旧した際に、データが災害発生直前に戻るのか、それとも1日前のデータに戻るのか、その後の業務運用にとって極めて重要な目標設定となります。
このように、平常時の情報データの喪失を許容する期間が、許容できる限界を上回るよう目標設定されるべきです。

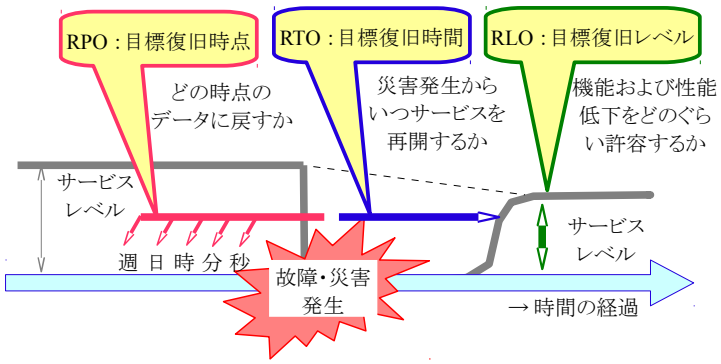


図 1.1 復旧目標の指標値

RPOが「0(ゼロ)」に近く短いほど、損失となるデータは少なく、RTOが短いほどダウンタイムが短時間で済み、RLOが高いほど災害前の機能と性能に近いことになります。このようにRPOとRTOを短くし、RLOを高くするには、災害に備えるためのソリューションや機器は相応に高価なものとなるため、費用対効果を考えて十分な事前設計が必要となります。これらの「目標」は高く設定することが理想ですが、必要コストが増大しないように、現実解を踏まえ関係者全体でバランスよく設定することが重要となります。
故障や災害発生時におけるITサービス継続は極めて重要なテーマとなります。従って、復旧すべきサービスに対する復旧目標を想定することはもちろんですが、その際に復旧対象サービスの優先度をきちんと見極め、さらにはその対応に要するコストとのバランスを十分に検討した上で、事前に対応方法を意思決定しておくことが必要となります。

また、さらに広い事業継続の視点でみた場合、ITシステム単体もしくはITサービスが復旧しても、それに関連する他のITシステムが正常に稼働して必要なデータの送受信ができるか否かや、通信や電力などのインフラが復旧できていない可能性もあり、現実的な対応方法を選択することに注意が必要となります。
なお、通常故障およびローカルサイト障害に対応するITシステム単体としての可用性や非機能要求に関する詳細については、昨年度2013年度のWG3成果「設計運用(走り続けるPostgreSQLシステム構築のために)」[1]にて整理および検証を行っておりますので、参照いただければ幸いです。
また、ITシステムの信頼性向上に関しては、「情報システムの信頼性向上に関するガイドライン第2版(経済産業省)」[2]にて整理されています。さらに、ITサービス継続に関しては、「ITサービス継続ガイドライン(経済産業省)」[3]にて整理されていますので、必要に応じてご参照願います。

1 https://www.pgecons.org/download/works_2013/
2 http://www.meti.go.jp/committee/materials2/downloadfiles/g90722a07j.pdf
3http://www.meti.go.jp/policy/netsecurity/docs/secgov/2011_IoformationSecurityServiceManagementGuidelineKaiteiban.pdf

2. DR要件を実現するPostgreSQLの代表的なシステム構成

本章では災害等に起因する広範囲にわたるITシステム障害に対して、PostgreSQLの「データ保全」または「サービス継続」を行うための代表的なシステム構成について紹介します。

・構成パターン：PostgreSQLが単独で稼働する「シングル」構成、PostgreSQLの稼働（Active）中サーバに加えて待機（Standby）サーバを設けることでトラブル発生時の信頼性を向上する「Active-Standby」構成、待機（Standby）中サーバ上で別の参照系クエリを実行することで信頼性に加えて稼働効率をアップする「Active-Active」構成で整理します。
・復旧目標：1章で解説したRTO、RPO、RLOについて整理します。
・コスト：初期費用（ハードウェア）や、システムの設定・構築費用、運用費用等、「費用」の視点から整理します。
・運用性：通常もしくはトラブル発生時の運用について、オペレータ（ユーザ）に対する「運用の複雑さや負担」の視点で整理しています。
・構築期間：システム構築の難易度に対応する調達や設計、構築、試験工程等の「期間」の視点から整理しています。
・DRサイト活用：DRサイト側のリソースが、非常時に備えた待機のみの状態であるか、何らかのクエリ処理を行うことでサービス処理に貢献しているかの視点で整理します。

なお、本資料ではDRを念頭においたRLO視点の考慮を中心に捉えており、新規構築時に配慮が必要な性能やセキュリティ等の面は基本的にスコープ対象外としています（3.2節にてレプリケーション方式による性能検証結果を紹介）。
以下、表2.1にそれぞれのシステム構成の特長について俯瞰し、各構成の詳細については後述します。

表 2.1 DR要件を実現するPostgreSQLの代表的なシステム構成

構成		データ保全				サービス継続				
		構成1	構成2	構成3	構成4	構成5	構成6	構成7	構成8	構成9
		フルバックアップ＆データ保管のみ	差分バックアップ＆データ保管のみ	H/W（ストレージ）でレプリケーション	S/W（DRBD）でレプリケーション	フルバックアップ&事前リストア	差分バックアップ＆事前リストア	マスタ⇒遠隔地DB非同期レプリケーション	マスタ⇒遠隔地DB部分レプリケーション	マスタ⇒メインサイト側スタンバイ⇒遠隔地DBカスケードレプリケーション
構成パターン（※1）		シングル	シングル	Active-Standby	Active-Standby	Active-Standby	Active-Standby	Active-Active	Active-Active	Active-Active
復旧目標	RTO 目標時間	長い	長い	長い	中	長い	長い	短い	短い	短い
	RPO 復旧ポイント	長い（バックアップ時点）	中（バックアップ時点）	短い（方式次第）	短い	長い（バックアップ時点）	中（バックアップ時点）	短い	短い	短い
	RLO 復旧レベル	（方式次第）	（方式次第）	（方式次第）	（方式次第）	（方式次第）	（方式次第）	（方式次第）	低	（方式次第）
コスト		低	低	高	中	低	低	中	中	中
運用性		楽	楽	楽	中	楽	楽	中	中	難
構築期間		短い	短い	短い	長い	短い	短い	長い	長い	長い
DRサイト活用（※2）		×	×	×	×	△	△	○	○	○
実機検証対象		—	—	—	—	—	—	○	—	○

※1：典型的なPostgreSQLのローカルActive-Standby構成やActive-Active構成は含めておりません。2013年度の成果報告をご参照願います。
※2：○→DRサイトを参照系クエリ処理に活用できる、△→参照クエリ処理に活用できるもデータ同期レベルが落ちるもの

2.1. データ保全に対応するシステム構成

ここではサイト障害等、広範囲に影響を及ぼす問題の対処方法として、必要な「データ」を確保する「データ保全」の方法についていくつか紹介していきます。

2.1.1. 構成 1：フルバックアップ＆データ保管のみ

この「構成1：フルバックアップ＆データ保管のみ」は、サイト障害等広範囲にわたるトラブル時を見据えて、リモートサイト（遠隔地）に事業に必要なデータを残す、データ保全を目的としています。
定期的（例えば毎週末、毎夜など）なフルバックアップをベースとし、信頼性が高く、サイト障害の影響を受けないリモートサイトにデータを転送して保管します。
メインサイトのデータ更新頻度が低めで、かつリアルタイムの更新データを保存する必要性が少ないか、要件として割り切れる場合が主な検討の条件となります。
復旧目標は必要最小限、すなわち RPO はフルバックアップ取得時点となり長く、また RTO はバックアップからリモートサイトで起動した PostgreSQL にリストア後、周辺ミドルや関係する IT サービスを含めて再起動を行うため、サービス開始までの時間は相応に長くなります。場合によっては、サービス再開を考慮する必要がない場合においても、一定のデータ保全を実施可能です。
サイト障害対応のコストは、DR サイト側のデータ保管が中心であり、各種クラウドサービスの活用により IT リソースの費用負担を最小レベルで抑えることが可能となります。
運用性についてはリモート保管は手順的にもシンプルで容易に実現でき、さらに複雑な作り込みが必要ないことから構築期間面では最短レベルであり、既存の小規模システムにおけるサイト障害対策の1つとして採用しやすい構成となります。ただし、DR サイト側はデータの保管のみであり、サービス面におけるメリットはありません。

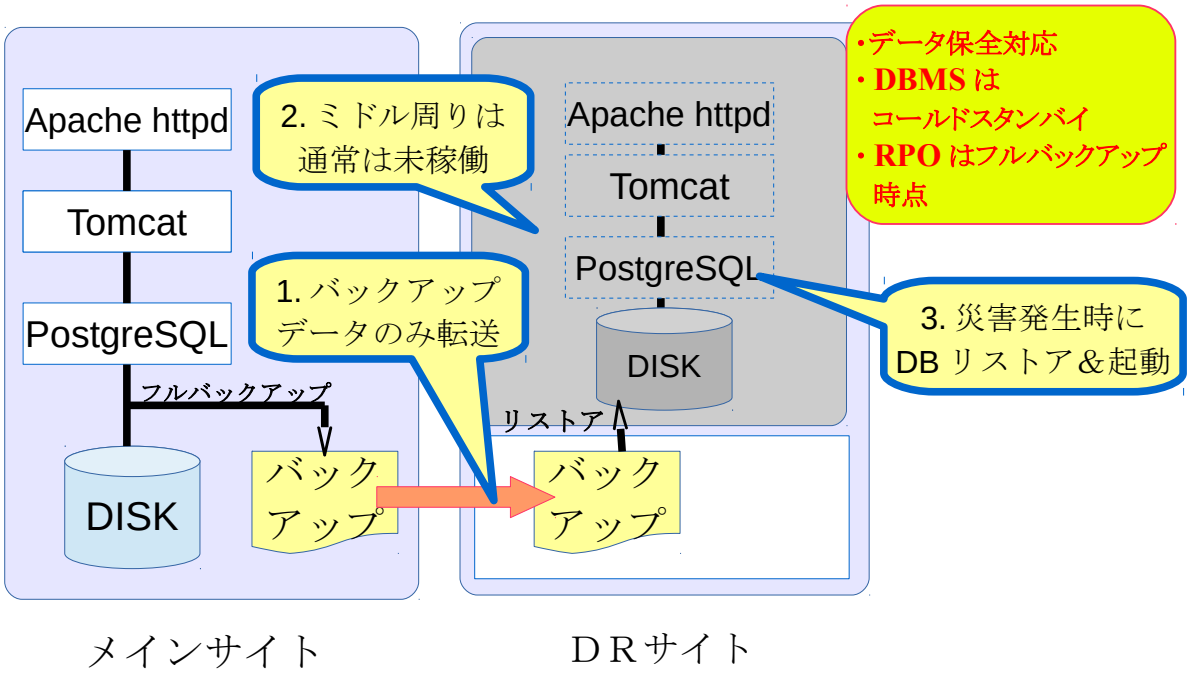


図 2.1: 構成1:フルバックアップ＆データ保管のみ（例）

なお、バックアップの取得方法として
pg_rman、pg_basebackup、pg_start_backup/pg_stop_backup、pg_dump 等、どのツールを利用するか、データ転送を効率化のためにデータの圧縮を行うかは、検討が必要となります。

2.1.2. 構成2：差分バックアップ＆データ保管のみ

この「構成2：差分バックアップ＆データ保管のみ」も、サイト障害時を見据えて、リモートサイト（遠隔地）に事業に必要なデータを残す、データ保全に特化した方法の 1 つとなります。

構成1との違いは、定期的なフルバックアップに加えて差分バックアップを取得することで RPO の向上が図れることにあります。

メインサイトのデータ更新頻度が低めで、かつリアルタイムの更新データを保存する必要性が少ないか、要件として割り切れる場合が主な条件となることは同様ですが、差分バックアップの取得でバックアップの取得頻度が向上し、より広い範囲のデータを救うことができるようになります。

復旧目標の中で、RPO は差分バックアップ取得のタイミングであり構成1に比べて向上、RTO については構成1と比べて差分を反映する分だけ若干長くなります。こちらも、データ保管のみを行って急ぎのサービス再開は考慮する必要がない場合の選択肢の1つとなります。

構成1と同様、サイト障害対応のコストは、DR サイト側のデータ保管が中心であり、差分バックアップ分のストレージ容量が追加で必要となりますが各種クラウドサービスの活用により大きな問題とはならないと思われます。

運用性についてはリモート保管の手順はシンプルで容易、構築期間的に機能構築が単純であることから最短レベルであり、既存の小規模システムにおけるサイト障害対策として採用しやすい構成となります。ただし、DR サイト側はデータ保管のみであり、サービス面におけるメリットはありません。

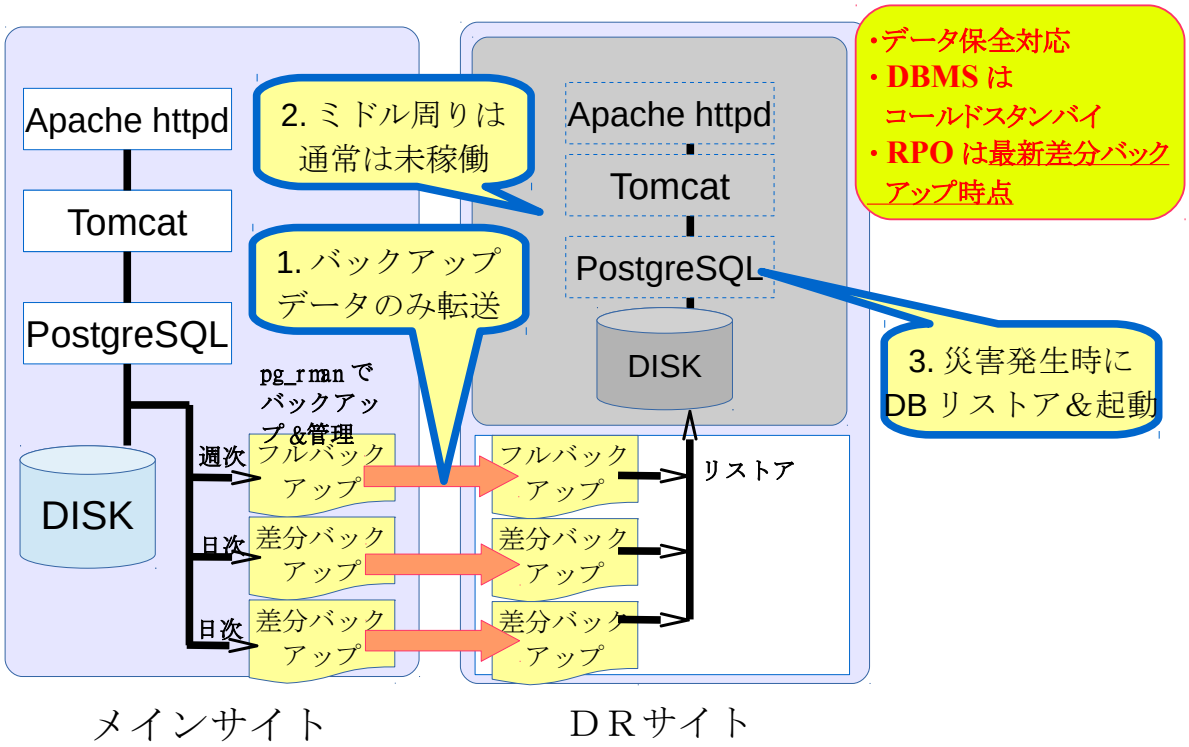


図 2.2: 構成2:差分バックアップ＆データ保管のみ（例）

差分バックアップの取得方法としては pg_rman（物理差分バックアップ）の活用が選択肢の 1 つであり、データ転送の効率化のためにデータの圧縮を行うか否かも、検討が必要となります。

2.1.3. 構成3:H/W(ストレージ)でレプリケーション

この「構成3:H/W(ストレージ)でレプリケーション」は高機能なハードウェア・ストレージを活用し、それにバックアップやリモートコピーを任せることにより、運用がシンプルになり、かつメインサイト側のサーバ負荷を軽くできる構成です。代表的な機能として、「スナップショット」と「(ストレージ)レプリケーション」があります。

「スナップショット」は、ある時点(瞬間)のデータがどこにあるかを記録したポインタ相当の(コピーとは異なる)イメージとなります。スナップショットされた別の領域に存在するデータを、その時点(瞬間)のデータとして見せることで、複数世代のデータを効率的に保持することができます。

具体的には、データ更新の際に更新前データを別の領域に待避し、ポインタはその待避先を指し示す仕組みとなっています。このため、ポインタの指し示す元のデータ領域に問題が生じた場合、スナップショットは利用できなくなります。従って、スナップショットをバックアップの代替策として活用することは推奨されません。さらに、取得済みスナップショットをバックアップ元に指定することで、メイン(稼働系)サーバからの最新データへのアクセス競合の軽減が期待できますが、一方でストレージ自体の負荷は下がらないことに注意が必要です。

次に「(ストレージ)レプリケーション」は、データの複製をストレージ内部で行う「ローカルレプリケーション」と、NW経由で遠隔地に行う「リモートレプリケーション」の方法があり、レプリケーション直前の完全なデータの複製が可能となります。なお、レプリケーションでは元のデータ量と同じストレージ容量が必要となりますが、通常のバックアップと異なり、リストア作業の必要がなく、複製されたデータ領域を再マウントするだけで、運用の再開が可能になり、効率が良いことが期待されます。

しかし、ストレージレプリケーション単体では、ある特定時点のデータ内容に戻すことは不可能であり、スナップショットとストレージレプリケーションでお互いの欠点を補うこと、もしくは PostgreSQL のバックアップと PITR を併用することでこれらの問題を補完することができます。

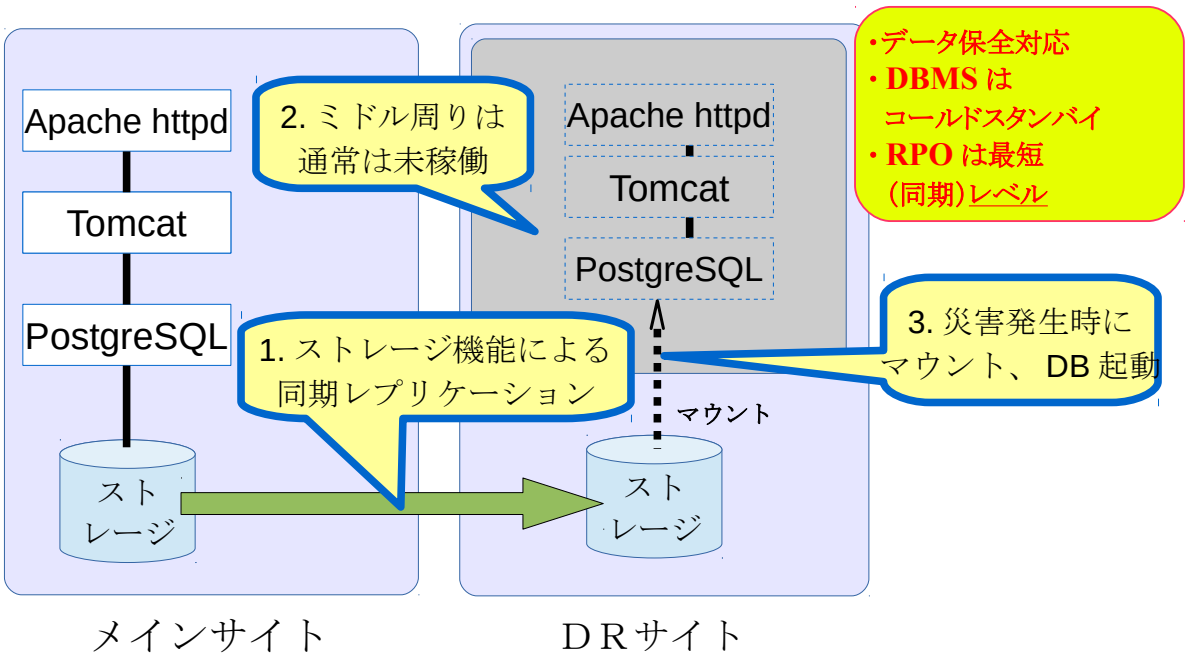


*図 2.3: 構成3:*H/W(ストレージ)でレプリケーション(例)

復旧目標はストレージの機能やNW性能に依存しますが、短い RPO を期待することができ、RTO はリモートサイトで起動した PostgreSQL にマウントすることで対応が可能となるため、前述のデータのみ保管の構成1, 2と比較するとリカバリに要する時間が不要な分、サービス開始までに要する時間は短くなることが想定されます。

なお、高機能なハードウェアが必要になるため、IT リソースのコストは相応にアップすることになり、この構成の採用には、メインサーバへの影響(DB 性能の改善やストレージ集約による運用改善等のメリット)、データの重要性、システム予算(デメリット)など、複数の観点での十分な事前検討による意思決定が必要となります。

運用性についてはストレージの機能により複製が自動的に行われるためオペレータの運用手順は単純で、構築期間的にも複雑な設定や機能構築は不要であることから最短レベルとなることが想定されます。また、DR サイトの

活用については期待できません。

2.1.4. 構成4：S/W（ブロックレベルレプリケーション機能）でレプリケーション

この「構成4：S/W（ブロックレベルレプリケーション機能）でレプリケーション」におけるツールの1例として、DRBD（Distributed Replicated Block Device）あります。これは特定パーティションへのデータ更新に対して、NW を経由してリモートサイトにデータをレプリケーションする機能を持つソフトウェア（OSS）です。2台のサーバ間で NW 経由の RAID1 相当のレプリケーション機能を実現します。

なお DRBD におけるレプリケーション対象は、ブロックデバイス（/dev/sda1 等）単位で指定する必要があり、ディレクトリやファイル単位でレプリケーションを取得することはできません。

アーキテクチャとしては、ファイルシステムとディスクドライバの中間である低いレイヤーで動作し、レプリケーションの対象として設定したブロックデバイスへの変更に対し、ローカルディスクへの書込みと同時にリモートサイトへの書込みが自動的に行われます。このため、通常の運用時には利用者が特に意識することはありませんが、DRBD のインストール後に、ストレージや NW、各種リソースの設定および初期同期をきちんと実施しておくことが大切です。DRBD の同期方向を間違えた場合、元のデータを消失する場合がありますので、特に慎重な対応が必要となります。

復旧目標は H/W（ストレージ）でレプリケーションと同様に短い RPO を期待することができ、RTO もリモートサイトで PostgreSQL を起動すれば対応が可能となるため、データベースのサービス開始までに要する時間は短くなることが想定されます。

DRBD の導入コストはそれほど高くありませんが、しっかりした設計に基づく構築が必要となります。さらに、ブロック単位での転送を NW 経由で行うため、相応の NW リソースの消費を考慮すべきです。

運用性については NW 等の性能面を含めた設計および日常のオペレーションに注意を払うことが必要です。また、慎重な設計が必要なことから、余裕を持った構築期間を確保することが重要です。また、DR サイトの活用面でのメリットはありません。

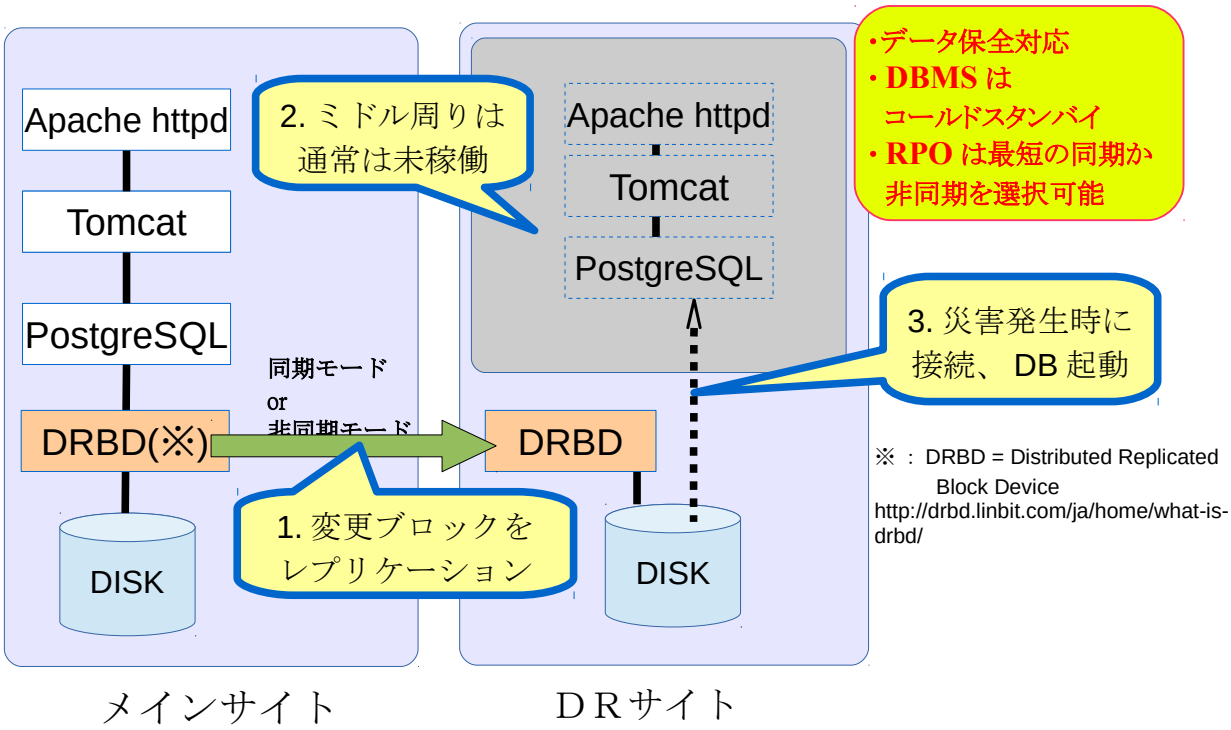


*図 2.4: 構成4：*S/W（DRBD)でレプリケーション（例）

2.2. サービス継続に対応するシステム構成

前節では、必要な「データ」を確保する「データ保全」を実現する構成を紹介して参りました。ここでは、データベース（PostgreSQL）のサービス継続を意識（RTO を向上）するシステム構成について紹介していきます。

2.2.1. 構成5：フルバックアップ＆事前リストア

この「構成5：フルバックアップ＆事前リストア」は、サイト障害時にリモートサイト（遠隔地）に事業に必要なデータをしっかり残し、サービス再開までの時間（RTO）短縮のため、リモートサイトの PostgreSQL にバックアップをリストアし、常時稼働させて待機する方法となります。定期的（例えば毎週末、毎夜など）なフルバックアップをベースとし、信頼性が高く、サイト障害の影響を受けないリモートサイトにデータを転送し、稼働中の PostgreSQL に転送済みフルバックアップデータをリストアして、データベースサービスをスタンバイします。

メインサイトのデータ更新頻度が低めで、かつリアルタイムの更新データを保存する必要性が少ないか、要件として割り切れる場合が検討の条件となりますが、「構成1」と比べ、PostgreSQL が起動状態で待機しているため、より短時間でのサービス再開が期待できます。

復旧目標は、RPO はフルバックアップのタイミングとなり長くなりますが、データベースに対する RTO は PostgreSQL が稼働状態で待機しているため、短い時間で済むことが期待できます。

サイト障害対応のコストは、DR サイト側 PostgreSQL が常時稼働している環境を維持する必要があり、構成1や2と比べれば IT リソースの面からランニングコストはアップします。

運用性についてはリストア手順の確立が必要となるものの、実績のある手順であり機能構築が単純であることから構築期間的には最短レベルであり、既存の小規模システムにおけるサイト障害対策として選択肢の 1 つとなります。また、メインサイト側とのデータ同期のズレを許容できることを条件として、参照系クエリやオンライン分析（OLAP）処理等を DR サイト側で実施することが可能です。

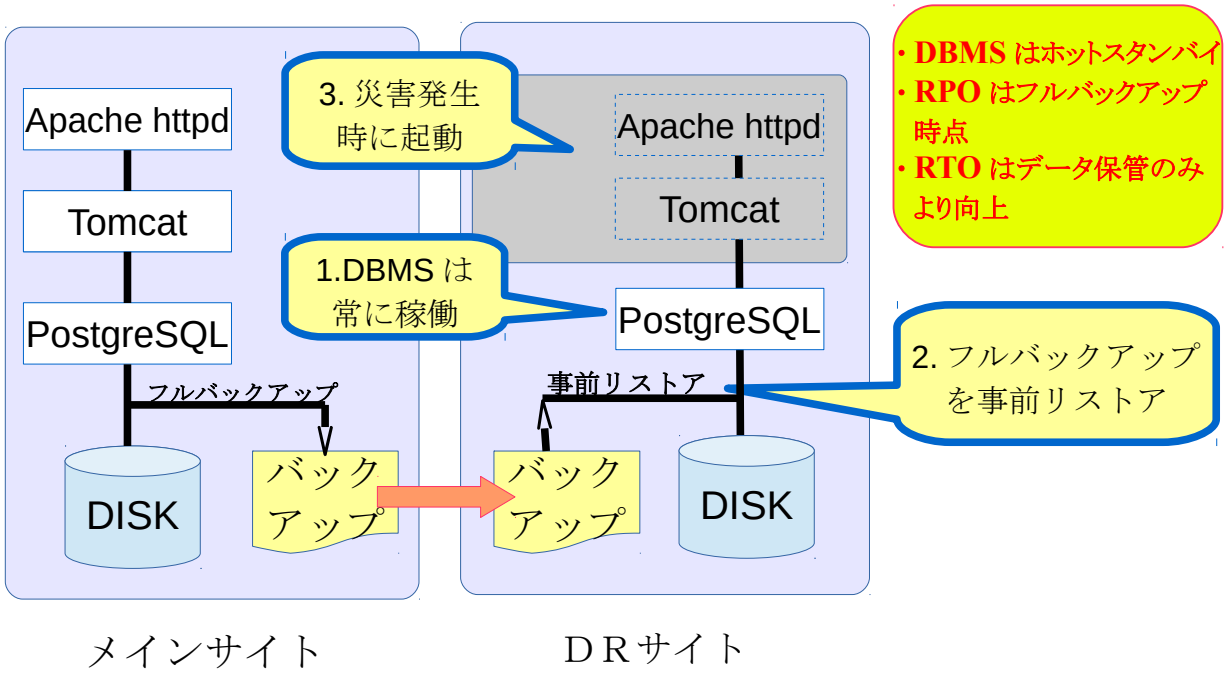


図 2.5: 構成5：フルバックアップ＆事前リストア（例）

この構成においても、バックアップの取得方法として pg_rman、pg_basebackup、pg_start_backup/pg_stop_backup、pg_dump 等、どのツールを利用するか、データ転送を効率化のためにデータの圧縮を行うかは、同様に検討が必要となります。

2.2.2. 構成6：差分バックアップ＆事前リストア

この「構成6：差分バックアップ＆事前リストア」も、サイト障害時にリモートサイト（遠隔地）に事業に必要なデータをしっかり残し、サービス再開までの時間（RTO）短縮のため、リモートサイトで常時稼働させた PostgreSQL にリストアしておく方法となります。

構成5との違いは、定期的なフルバックアップに加えて差分バックアップを取得することで RPO の向上が図れることにあります。

メインサイトのデータ更新頻度が低めで、かつリアルタイムの更新データを保存する必要性が少ないか、要件として割り切れることが条件となりますが、構成2のデータ保管の場合と比べ、PostgreSQL が起動状態で待機しているため、より短時間でのサービス再開が期待できます。

復旧目標において、RPO は差分バックアップの取得タイミングであり構成5に比べて向上し、RTO については構成5と比べて差分を反映する分だけ若干長くなります。

サイト障害対応のコストは、DR サイト側 PostgreSQL が常時稼働している環境を維持する必要があり、構成1や2と比べればランニングコストはアップします。

運用性についてはリストア手順の確立が必要となるものの、実績のある手順であり機能構築が単純であることから構築期間的には最短レベルであり、既存の小規模システムにおけるサイト障害対策として選択肢の 1 つとなります。また、メインサイト側とのデータ同期のズレを許容できる統計情報処理等を DR サイト側で実施することが可能です。

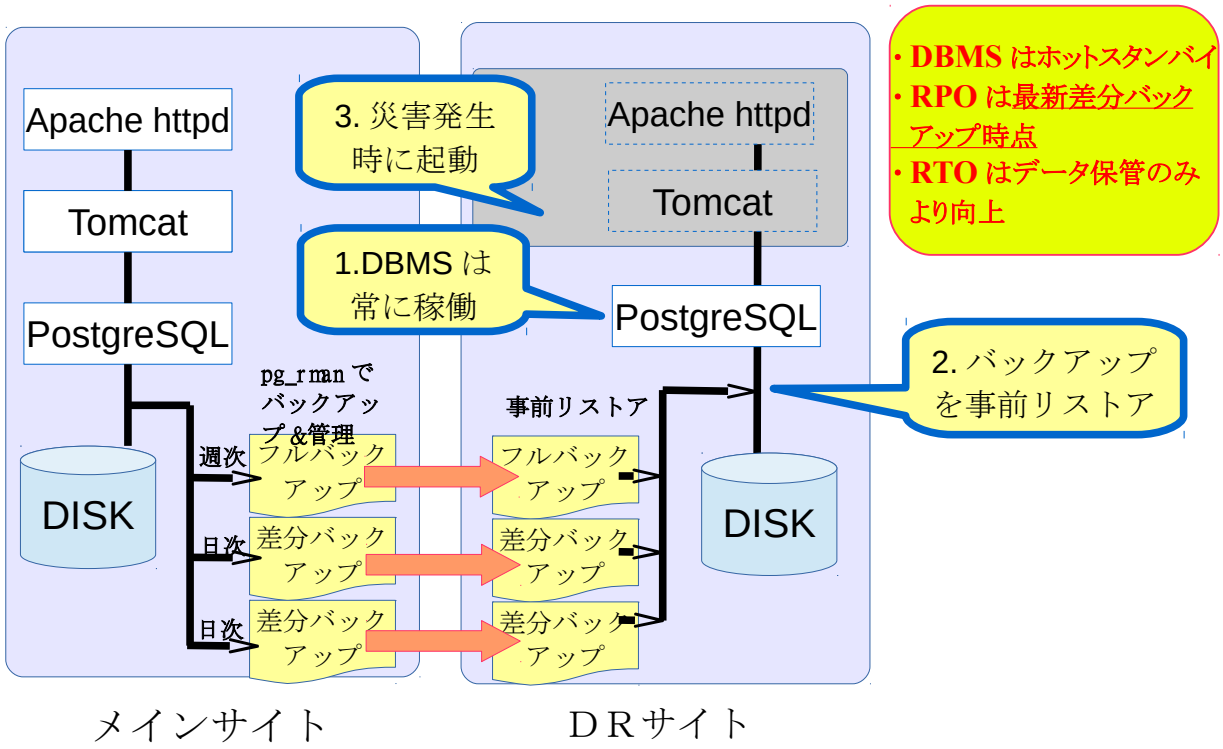


図 2.6: 構成6：差分バックアップ＆事前リストア（例）

また、差分バックアップの取得方法としては pg_rman の活用が選択肢の 1 つであり、データ転送を効率化のためにデータの圧縮を行うか否かは、検討が必要となります。

2.2.3. 構成7:マスタ⇒遠隔地 DB 非同期レプリケーション

PostgreSQL では、version9.0 以上よりトランザクションログによるデータベースのレプリケーション機能(ストリーミングレプリケーション)が搭載されており、非同期レプリケーション、同期レプリケーション、カスケードレプリケーションと順次その機能が拡張されてきています。その仕組みは、マスタからトランザクションログ(ログ先行書込み WAL:Write Ahead Log)をスタンバイ側に転送し、スタンバイ側の PostgreSQL にてログをリカバリしながら、データベース全体を複製していきます(部分的なデータベースの複製には対応していないことに注意が必要)。

レプリケーションの活用により、スタンバイ側にて参照系クエリ(select)の処理が可能となり、参照負荷の分散や統計処理等の参照系バッチ処理のマスタ側負荷を低減することが可能になります。なお、更新系クエリ(insert、update、delete 等)やメンテナンス系コマンド(vacuum、analyze 等)は、マスタ側のみで実行が可能です。

この「構成7:マスタ→遠隔地 DB・非同期レプリケーション」構成は、ローカルとリモートサイトにそれぞれスタンバイデータベースを配置したマルチスタンバイ構成となります。例として、メイン側スタンバイには同期レプリケーションを行い通常故障に備えた高可用構成を取り、DR 側スタンバイにはローカルシステムの性能面を考慮して非同期レプリケーションによるデータベースの複製を考慮した方式を採ることも可能です。

また、NW の遅延等の原因により、トランザクションログの転送に一定の遅延が生じた場合、レプリケーションが継続できず、停止してしまう場合も考慮しておく必要があります。この対応策として、アーカイブログを平行してリモートサイトに転送し、万一の際にリカバリによる復元ができるような安全対策の配慮も大切です。

さらにレプリケーションにおいては、転送されたトランザクションログをスタンバイ側バッファ上に書き込んだ時点でローカル側でコミットしたり、スタンバイ側バッファの内容をディスクにフラッシュ(書込み)するまで待つ等の選択が可能(※注)であり、よりきめ細かな設定を行えます。いずれにせよ、リモートサイトの RTO・RPO は共に短いレベルを維持しています。

コスト面では PostgreSQL のレプリケーション機能活用の設定と、DR サイト側で PostgreSQL が常時稼働するためのリソースに対する配慮が必要になります。さらに、設計面とレプリケーションとリカバリが継続できているかの運用監視、レプリケーション停止時の対応手順の構築など、意識しておく必要があります。

運用面では、監視、運用手順に従った対応が必要となり、オペレータへの負荷は若干重くなります。また、構築期間的にも設計面や運用手順の確認、トラブル発生時の対応手順の構築と確認が複雑になるため、構築、運用手順の作成には余裕を持って取り組みことが重要となります。

一方で、DR サイト側にて参照系クエリ処理が可能となり、リソースの活用効率を高めることが可能です。

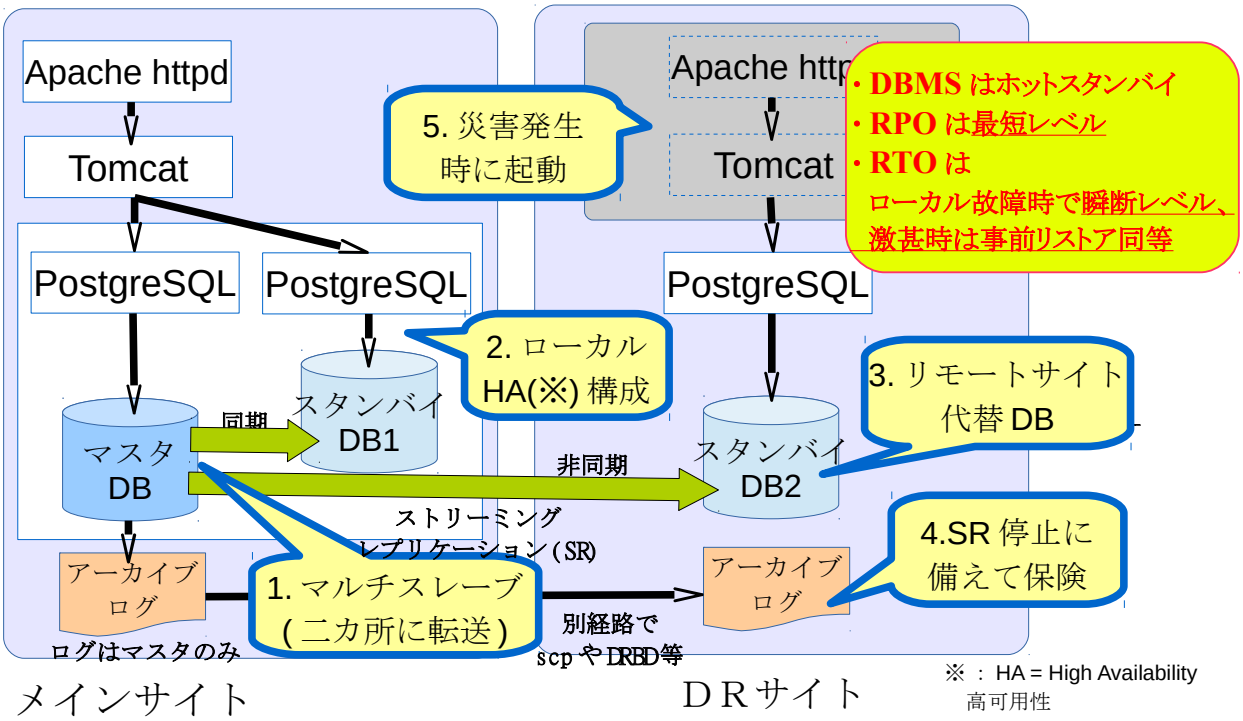


*図 2.7: 構成7:*マスタ→遠隔地 DB・非同期レプリケーション(例)

（※注）同期モードの指定
postgresql.conf ファイルの synchronous_commit()で設定が可能です。
on：マスタとスタンバイのディスクへのログ同期書込みまで待つ（同期モード）
remote_write：マスタ側ディスクへのログ同期書込みとスタンバイ側メモリへの書込みまで待つ
local：マスタ側ディスクへのログ同期書込みまで待ち、スタンバイに関しては待たない
off：マスタとスタンバイに対してログ書込みを待たない(非同期モード)

なお、図中のアーカイブログの転送については、レプリケーション方式上、必須ではありませんが、NW その他のトラブルによりレプリケーションが不測の停止状態となることを配慮して、検討しておくことが安全です。

2.2.4. 構成8：マスタ⇒遠隔地 DB 部分レプリケーション

この「構成8：マスタ⇒遠隔地 DB・部分レプリケーション」は、メインサイトのマスタ DB に対して、事業継続に最低限必要なテーブルのみを DR サイトに転送することで、効率的な災害対策を図る構成となります。この部分レプリケーションの実現手段の例として、Slony-I や xDB Replication（PostgreSQL ベースの商用製品 Postgres Plus Enterprise Edition（PPEE）（※注））、または PostgreSQL9.4 でリリースされたロジカルレプリケーション等の機能を活用することが選択肢として考えられます。
この部分レプリケーションを実施した場合、転送するデータ量を小さく絞り込むこととなり、DR サイト側サーバのリソースや NW の転送コストを小さく抑えることができます。その反面、DR サイトに構築するデータベースのテーブルを絞り込むことから、災害復旧時におけるサービスには制約が生じることとなり、転送対象とするテーブルの選定を慎重に実施する必要があります。言い換えれば RLO についてはある程度妥協することとなります。最終的には災害発生前の状態に戻すことを考慮すると、緊急対応的な構成に近いものと考えることができます。

（※注）xDB Replication では、マスタ DB が 1 つで複数のスタンバイを構成する場合と、複数のマスタ DB で構成する場合の2つのパターンに対応することが可能であり、レプリケーション設定されたテーブルについて、insert、update、delete にトリガを設定し、クエリベースの複製を実施するアーキテクチャとなっています。

コスト面では、DR サイト側で PostgreSQL が常時稼働するためのリソースが必要になることと、転送対象の絞り込みを含む設計や運用手順や試験の配慮等が必要となります。また、最終的な完全復旧に向けた対応手順も検討しておく必要があります。
運用性については、DB がサブセットとなることに起因して従来と異なる運用となることが想定され、オペレータへの負荷は重くなることが想定されます。また、構築期間的にも設計面や運用手順の確認、トラブル発生時の対応手順が複雑になるため、構築、運用手順の作成には余裕を持って取り組むことが重要となります。

一方で、DR サイト側にて参照系クエリ処理が可能となり、リソースの活用効率を高めることが可能です。

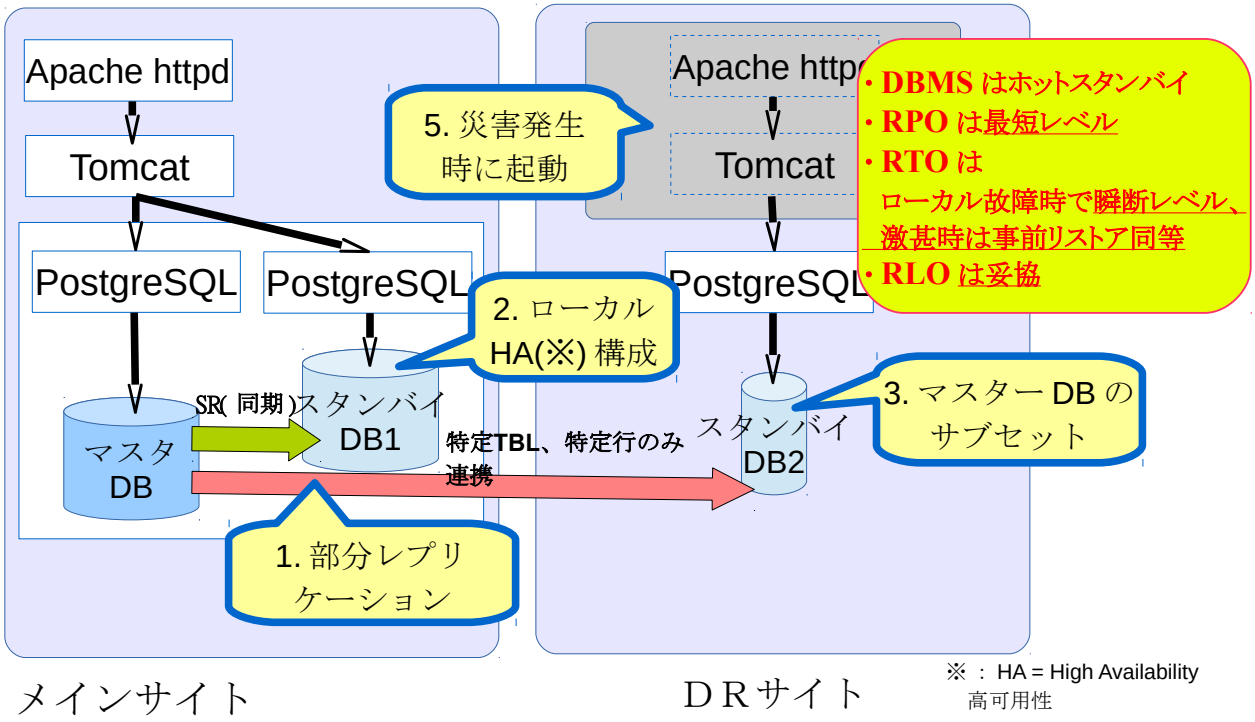


*図 2.8: 構成8:*マスタ→遠隔地 DB・部分レプリケーション（例）

2.2.5. 構成9：マスタ⇒メインサイト側スタンバイ⇒遠隔地 DB カスケードレプリケーション

「構成9：マスタ⇒メインサイト側スタンバイ⇒遠隔地 DB・カスケードレプリケーション」構成は、マスタ DB→メイン側スタンバイ DB→DR 側スタンバイ DB のようにマスタ 1 台と複数のスタンバイによるカスケード構成となります。例として、スタンバイ 1 台目には同期レプリケーションを行い通常故障に備えた高可用構成を採り、リモートのスタンバイ 2 台目には非同期レプリケーションによるデータベース複製を考慮した方法を採ることも可能です。

この構成では、マスタ DB と接続するスタンバイ DB が複数存在する構成7のマルチスタンバイ構成に比べて、マスタ DB に直接つながるスタンバイが 1 台となるため、マスタ DB におけるレプリケーション処理の負担は軽くなることが想定されます。

また、NW の遅延等の原因により、トランザクションログの転送に一定の遅延が生じた場合、レプリケーションが継続できず、停止してしまう場合も考慮しておく必要があります。この対応策として、アーカイブログを平行してリモートサイトに転送し、万一の際にリカバリによる復元ができるような安全対策の配慮も大切です

コスト面では PostgreSQL のレプリケーション機能により実現するため初期投資の必要はありません。DR サイト側で PostgreSQL が常時稼働するためのリソースが必要になることと、設計での配慮とレプリケーションとリカバリが継続できているかの運用監視、レプリケーション停止時の対応手順など、コストアップを意識しておく必要があります。

運用性については監視、運用手順が必要となり、オペレータへの負荷は若干重くなります。また、構築期間的にも設計面や運用手順の確認、トラブル発生時の対応手順が複雑になるため、構築、運用手順の作成には余裕を持って取り組むことが重要となります。

一方で、DR サイト側にて参照系クエリ処理が可能となり、リソースの活用効率を高めることが可能です。

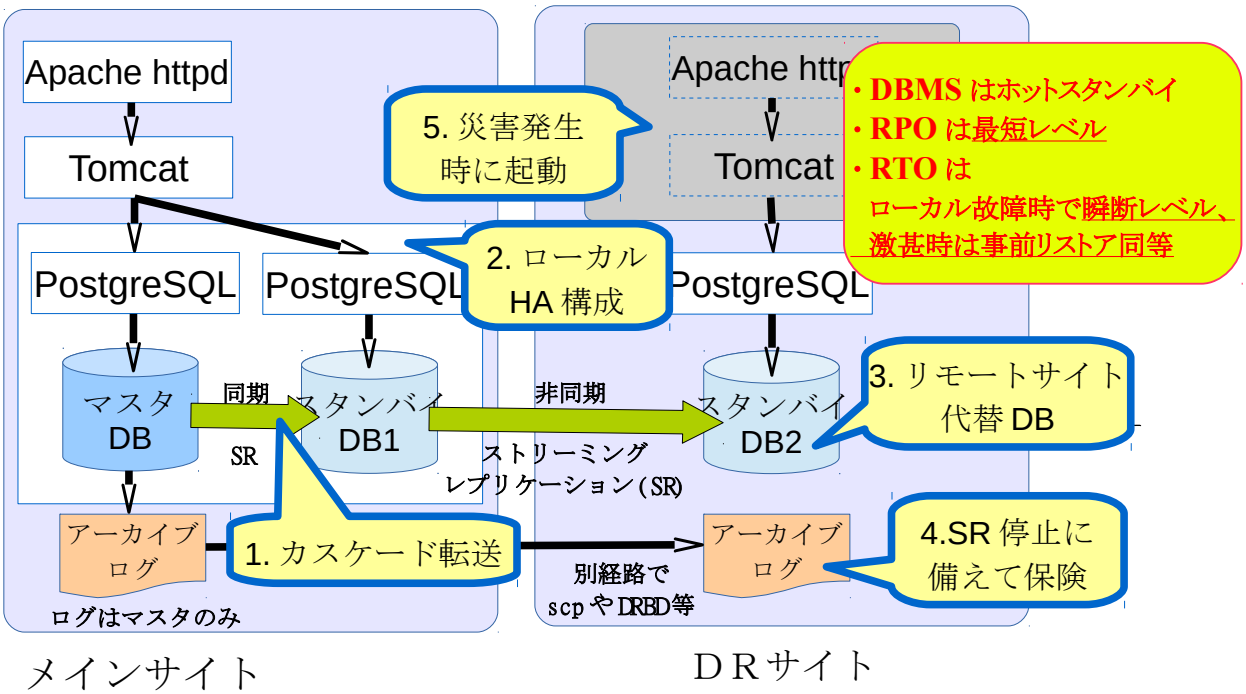


*図 2.9: 構成9:*マスタ→メインサイト側スタンバイ→遠隔地 DB・カスケードレプリケーション（例）

なお、図中のアーカイブログの転送については、レプリケーション方式上、必須ではありませんが、NW その他のトラブルによりレプリケーションが不測の停止状態となることを配慮して、検討しておくことが安全です。

3. 運用技術検証

本章では第２章「表 2.1 DR 要件を実現する PostgreSQL の代表的なシステム構成」で記載した構成の内、サービス継続性を重視した下記２つの構成に対し運用手順および性能の観点から検証を行った結果について報告します。
構成 7：マスタ⇒遠隔地 DB 非同期レプリケーション(以降、マルチスタンバイ構成と略す)
構成 9：マスタ⇒メインサイト側スタンバイ⇒遠隔地 DB カスケードレプリケーション(以降、カスケード構成と略す)

3.1. 運用手順

本節では DR サイトの運用手順として検証環境の構築手順の確立、および復旧手順について整理します。図 3.1 にマルチスタンバイ構成、図 3.2 にカスケード構成の概要図を示します。

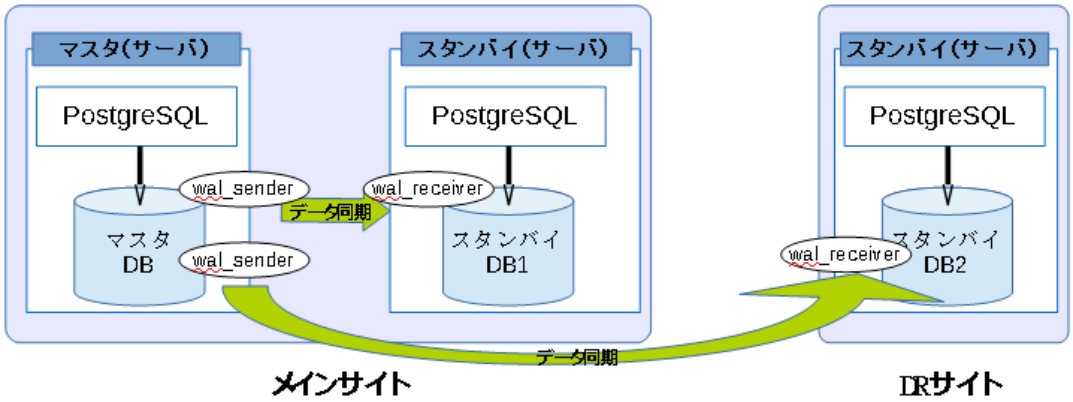


図 3.1: マルチスタンバイ構成

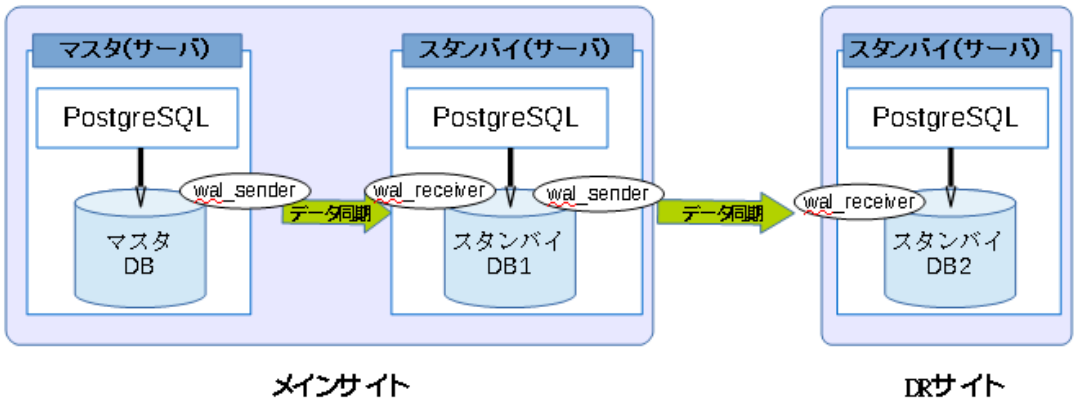


図 3.2: カスケード構成

次項より PostgreSQL における DR サイトの運用手順として以下の検証を実施した結果を報告します。

- 構築手順
  2013 年度の WG3 活動で可用性を担保する構成の基礎検証を実施しましたが、大規模障害に対応できる DR 構成を構築する場合、可用性構成の構築と同様でよいかという懸念がありました。
  そこで、本年度では、ピックアップした２つの DR 構成について、実際に検証環境の構築を行い、構築手順を整理します。

- 復旧手順
  DR 構成を利用するようなミッションクリティカルなシステムにおいては、障害が発生した場合でも手順に沿ってスムーズに復旧できることが重要になります。

そこで、本検証ではメインサイトに障害が発生した場合の復旧手順を作成し、実機で検証、挙動確認を行います。メインサイトに対する障害の種類は、被害度が大きい以下のパターンを想定しています。

case1.マスタサーバ障害時の挙動

マスタサーバ障害イメージを図 3.3 に示します。

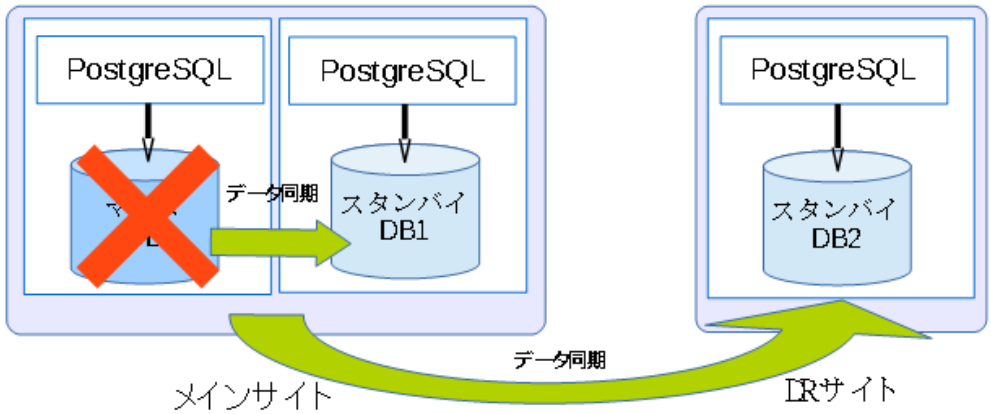


図 3.3: マスタサーバ障害イメージ

case2.メイン側スタンバイサーバ障害時の挙動

メイン側スタンバイサーバ障害イメージを図 3.4 に示します。

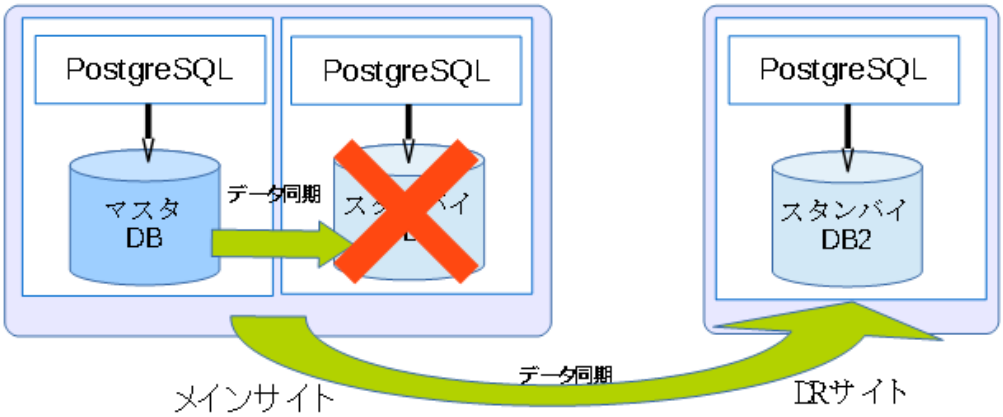


図 3.4: メイン側スタンバイサーバ障害イメージ

case3.メインサイト全体障害時の挙動

メインサイト全体障害イメージを図 3.5 に示します。

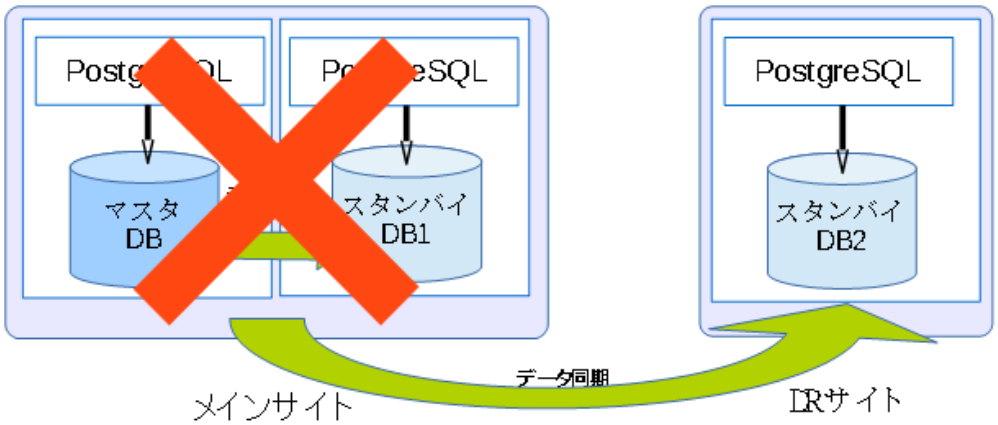


図 3.5: メインサイト全体障害イメージ

なお、DR サイトの障害は、構築手順と同等の対応となる点、至急に復旧すべき対象にはなりにくい点から、今回の検証からは除外しています。

3.1.1. マルチスタンバイ構成

本項では PostgreSQL のストリーミングレプリケーション機能を用いたマルチスタンバイ構成に対する検証を行います。

3.1.1.1.検証環境

動作検証に用いる DR 構成の構築は、メインサイトにマスタサーバおよびメイン側スタンバイサーバを構築し、DR サイトに DR 側スタンバイサーバを構築します。マルチスタンバイ構成では各スタンバイサーバに対しマスタサーバから WAL が転送されて同期処理が行われます。データ同期は、手順の確立を目的としており、パフォーマンス測定を目的としていないため、レプリケーションは全て非同期設定にします。マルチスタンバイ構成として今回使用した検証環境を図 3.6 に示します。

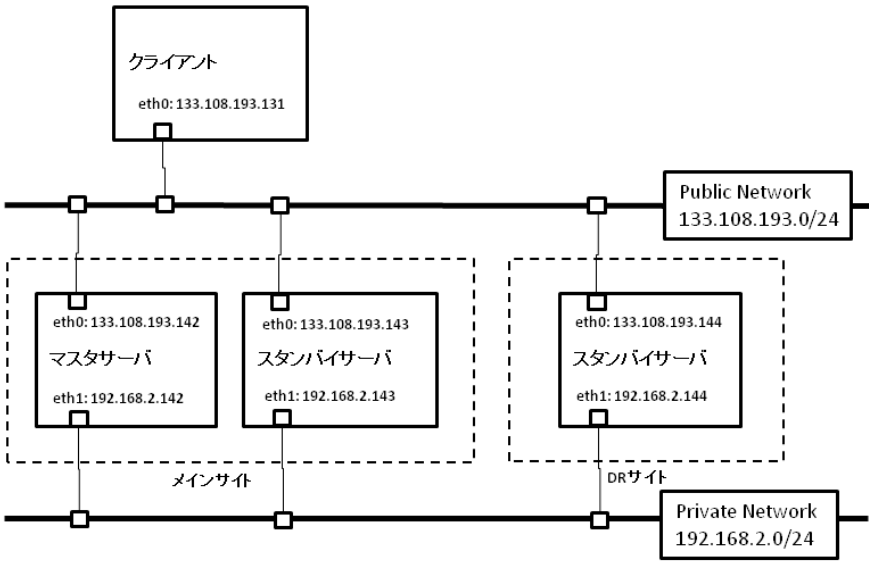


図 3.6: マルチスタンバイ構成概要

(1) システム構成

表 3.1: 検証環境のシステム構成

構成	構成情報	備考
OS	RHEL/CentOS 6.4 x86_64	
ハードウェア・アーキテクチャ	x86_64	PostgreSQL サーバはマスタとスタンバイ 2 台で計 3 台用意
PostgreSQL	9.4.0	
PostgreSQL スーパユーザ名	postgres	
前提 AP	openssh-clients	restore_command で scp を使用するため

(2) ネットワーク構成

表 3.2: 検証環境のネットワーク構成

構成サーバ名	構成情報	備考
マスタサーバ	eth0:133.108.193.142	Public Network
	eth1:192.168.2.142	Private Network
メイン側スタンバイサーバ	eth0:133.108.193.143	Public Network
	eth1:192.168.2.143	Private Network
DR 側スタンバイサーバ	eth0:133.108.193.144	Public Network
	eth1:192.168.2.144	Private Network
クライアント	eth0: 133.108.193.131	Public Network

(3) パラメータ構成

1. postgresql.conf

表 3.3: postgresql.conf (全サーバ)

パラメータ	補足

# date control 	
timezone = 'Japan'
	日付書式関連パラメータ

# network control 	
listen_addresses = '*'
	ネットワーク関連パラメータ

#Performance control
max_connections = 100
shared_buffers = 200MB
wal_buffers = 1MB
checkpoint_segments = 10
	パフォーマンス関連パラメータ

#log control
logging_collector = on
log_connections = on
log_line_prefix = '%t %d [%p-%l]'
log_timezone = 'Japan'
log_filename = 'postgresql-%Y-%m-%d.log'
	ログ出力関連パラメータ

#Streaming Replication Primary
wal_level = hot_standby
archive_mode = on
archive_command = 'test ! -f /usr/local/src/pgsql/9.4.0/data/pgarc/%f
&& cp %p /usr/local/src/pgsql/9.4.0/data/pgarc/%f'
max_wal_senders = 4
hot_standby = on
	ストリーミングレプリケーション関連パラメータ

2. pg_hba.conf

表 3.4: pg_hba.conf (全サーバ)

パラメータ

```
host   all           postgres      133.108.193.131/32      md5
host   postgres    postgres      133.108.193.142/32      md5
host   postgres    postgres      133.108.193.143/32      md5
host   postgres    postgres      133.108.193.144/32      md5
host   replication  postgres      192.168.2.142/32        md5
host   replication  postgres      192.168.2.143/32        md5
host   replication  postgres      192.168.2.144/32        md5
```

3 recovery.conf

表 3.5: recovery.conf(メイン側スタンバイサーバ)

パラメータ
standby_mode = 'on' primary_conninfo = 'host=**192.168.2.142** port=5432 user=postgres' restore_command = 'scp **192.168.2.142**:/usr/local/src/pgsql/9.4.0/data/pgarc/%f %p' recovery_target_timeline = 'latest''

表 3.6: recovery.conf(DR 側スタンバイサーバ)

パラメータ
standby_mode = 'on' primary_conninfo = 'host=**192.168.2.142** port=5432 user=postgres' restore_command = 'scp **192.168.2.142**:/usr/local/src/pgsql/9.4.0/data/pgarc/%f %p' recovery_target_timeline = 'latest'

- primary_conninfo は、WAL ファイル提供元であるサーバへの接続情報を定義します。
- restore_command は、リカバリ時に使用する WAL ファイルの場所を定義します。

3.1.1.2.検証ケース

(1) 構築手順

実際に検証環境に構築を行い、構築手順の確立と構築するにあたっての懸念点がないか検証します。

(2) 復旧手順

メインサイトに障害が発生した場合の挙動を検証します。

それぞれ、クライアントからのセッションの状態は、SQL 実行スクリプトを作成して、PostgreSQL に定期的に以下のクエリを発行することで、状態を確認しました。

```
# SQL 実行スクリプト
export PGPASSWORD=postgres

PSQL=/usr/bin/psql
HOST=133.108.193.142   # 接続先はマスタサーバ
DATABASE=test01
COUNT=1

while true
do
 QUERY="insert into test01 values ($COUNT,'key01','value');"

 val1=$(date)
 val2=$(PGCONNECT_TIMEOUT=1 $PSQL -h $HOST -p 5432 -U postgres -d $DATABASE -q -t -c "$QUERY")
 if [ $? -ne 0 ]; then
  val2="ERROR"
 else
  val2="SUCCESS"
 fi
 echo "$COUNT,${val1},${val2}"

 COUNT=$(($COUNT+1))
 sleep 1
done
```

図 3.7: SQL 実行スクリプト

case1. マスタサーバ障害時の挙動
　マスタサーバに障害が発生した場合の挙動を検証します。検証ケースは、サーバ電源が落ちた場合の挙動と、その後のサービス復旧を想定しました。マスタサーバに障害が発生すると、クライアントからのセッションは切断され、復帰までエラーを返します。サービスの復旧は、手作業による切り替えを行い、切り替えが完了後に再び接続できるようになります。ここでは、マスタサーバがダウンしたことの確認と、サービスの復旧手順の確認を実施します。

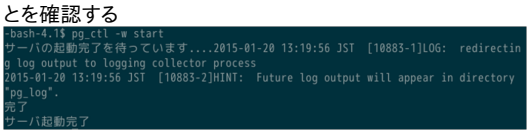
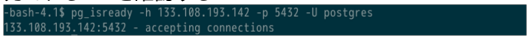
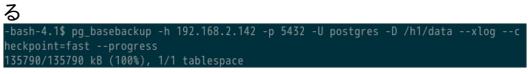
case2. メイン側スタンバイサーバ障害時の挙動
　メイン側スタンバイサーバに障害が発生した場合の挙動を検証します。検証ケースは、case1 と同様にサーバ電源が落ちた場合の挙動を想定しました。メイン側スタンバイサーバがダウンするためレプリケーションは途切れますが、マスタサーバはダウンしていないため、サービスへの影響はありません。また、**DR 側スタンバイサーバは生きているため、データ転送の遅延は発生しますがデータ保全性は保持されています。**ここでは、メイン側スタンバイサーバがダウンしたことの確認と、レプリケーション状態の確認を実施します。

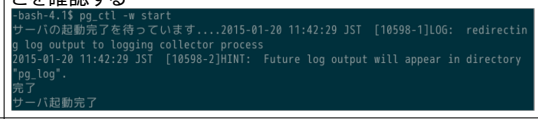
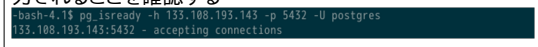
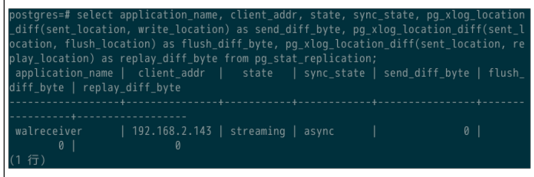
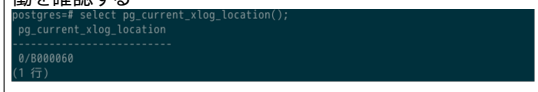
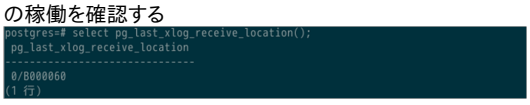
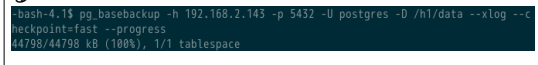
case3. メインサイト全体障害時の挙動
　メインサイト全体の電源が落ちた場合の挙動と、その後のサービス復旧を想定しました。メインサイトには、マスタサーバとメイン側スタンバイサーバが設置してあることを想定しています。まず、メインサイトに障害が発生すると、マスタサーバに接続していたクライアントからのセッションは切断され、復帰までエラーを返します。一方、サービスの復旧は、PostgreSQL の機能ではサービスの切り替えを自動的に行えないため、手作業による切り替えを行う必要があり、切り替えが完了後に再び接続できるようになります。ここでは、メインサイトがダウンしたことの確認と、サービスの復旧手順の確認を実施します。なお、障害前の構成まで復旧する手順の確認は、今回の検証からは除外しました。

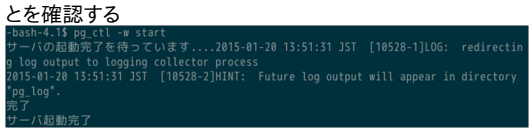
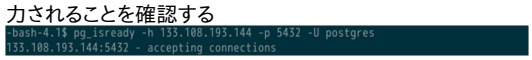
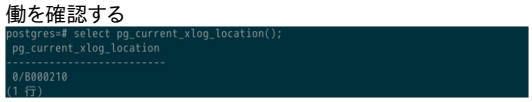
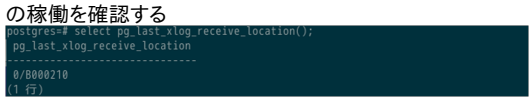
3.1.1.3.検証結果

(1) 構築手順

前提として PostgreSQL のインストールは完了しているものとします。

分類	No	作業内容	手順	確認
マスタサーバの構築	1	DB クラスタ作成	クラスタ DB を初期化する (コマンド例) initdb --encoding=UTF8 --no-locale --data-checksums -k -A md5 -W	initdb コマンドが正常に終了することを確認する 
	2	マスタサーバのパラメータを設定する	環境構成シートに記載のパラメータファイルを配置する ・postgresql.conf ・pg_hba.conf	設定が正しいことを 3.1.1.1 を参考に確認する
	3	PostgreSQL マスタサーバを起動する	PostgreSQL サーバを起動する (コマンド例) pg_ctl -w start	実行したコンソール上に「server started」と出力されることを確認する 
	4	正常稼働チェック	PostgreSQL サーバに対し pg_isready を実行し正常に起動していることを確認する (コマンド例) pg_isready -h ${TARGETHOSTNAME} -p ${TARGETPORT} -U ${TARGETUNAME} -d ${TARGETDBNAME}	実行したコンソール上に「accepting connections」と出力されることを確認する 
メイン側スタンバイサーバの構築	1	データをコピーする	マスタサーバからデータをコピーする (コマンド例) pg_basebackup -h ${HOSTNAME_Primary_pri} -p ${PORT_Primary} -U postgres -D ${PGDATA_Secondary} --xlog --checkpoint=fast --progress ※${HOSTNAME_Primary_pri}の設定値には、レプリケーション用 IP アドレスを指定	pg_basebackup コマンドが正常に終了することを確認する 
	2	メイン側スタンバイサーバのパラメータを設定する	環境構成シートに記載のパラメータファイルを配置する ・postgresql.conf	設定が正しいことを 3.1.1.1 を参考に確認する

分類	No	作業内容	手順	確認
			・pg_hba.conf ・recovery.conf	
	3	PostgreSQL メイン側スタンバイサーバを起動する	PostgreSQL サーバを起動する (コマンド例) pg_ctl -w start	実行したコンソール上に「server started」と出力されることを確認する 
	4	正常稼働チェック	PostgreSQL サーバに対し pg_isready を実行し正常に起動していることを確認する (コマンド例) pg_isready -h ${TARGETHOSTNAME} -p ${TARGETPORT} -U ${TARGETUNAME} -d ${TARGETDBNAME}	実行したコンソール上に「accepting connections」と出力されることを確認する 
	5	レプリケーション稼働チェック	レプリケーション元 PostgreSQL サーバに対し pg_stat_replication ビューを参照し正常に起動していることを確認する (コマンド例) SELECT application_name ,client_addr ,state ,sync_state, pg_xlog_location_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_location, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, replay_location) as replay_diff_byte FROM pg_stat_replication;	レプリケーション元で pg_stat_replication の client_addr でクライアント接続元、state で接続状態を確認する。 
			1) レプリケーション元 稼働チェック 　レプリケーション元 PostgreSQL サーバに対し pg_current_xlog_location を参照し正常に結果を返すことを確認する。 2) レプリケーション先 稼働チェック 　レプリケーション先 PostgreSQL サーバに対し pg_last_xlog_receive_location を参照し正常に結果を返すことを確認する。 1)と 2)の値が一致していることを確認する。	1) pg_current_xlog_location でレプリケーション元の稼働を確認する  2) pg_last_xlog_receive_location でレプリケーション先の稼働を確認する 
DR側スタンバイサーバの構築	1	データをコピーする	メイン側スタンバイサーバからデータをコピーする (コマンド例) pg_basebackup -h ${HOSTNAME_Secondary_pri} -p ${PORT_Secondary} -U postgres -D ${PGDATA_Tertiary} --xlog --checkpoint=fast --progress ※${HOSTNAME_Secondary_pri}の設定値には、レプリケーション用 IP アドレスを指定	pg_basebackup コマンドが正常に終了することを確認する 
	2	DR 側スタンバイサーバのパラメータを設定する	環境構成シートに記載のパラメータファイルを配置する ・postgresql.conf ・pg_hba.conf	設定が正しいことを 3.1.1.1 を参考に確認する

分類	No	作業内容	手順	確認
			・recovery.conf	
	3	PostgreSQL DR 側スタンバイサーバを起動する	PostgreSQL サーバを起動する (コマンド例) pg_ctl -w start	実行したコンソール上に「server started」と出力されることを確認する 
	4	正常稼働チェック	PostgreSQL サーバに対し pg_isready を実行し正常に起動していることを確認する (コマンド例) pg_isready -h ${TARGETHOSTNAME} -p ${TARGETPORT} -U ${TARGETUNAME} -d ${TARGETDBNAME}	実行したコンソール上に「accepting connections」と出力されることを確認する 
	5	レプリケーション稼働チェック	レプリケーション元 PostgreSQL サーバに対し pg_stat_replication ビューを参照し正常に起動していることを確認する (コマンド例) SELECT application_name ,client_addr ,state ,sync_state, pg_xlog_location_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_location, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, replay_location) as replay_diff_byte FROM pg_stat_replication;	レプリケーション元で pg_stat_replication の client_addr でクライアント接続元、state で接続状態を確認する。 
			1) レプリケーション元 稼働チェック レプリケーション元 PostgreSQL サーバに対し pg_current_xlog_location を参照し正常に結果を返すことを確認する。 2) レプリケーション先 稼働チェック レプリケーション先 PostgreSQL サーバに対し pg_last_xlog_receive_location を参照し正常に結果を返すことを確認する。 1) と 2) の値が一致していることを確認する。	1) pg_current_xlog_location でレプリケーション元の稼働を確認する  2) pg_last_xlog_receive_location でレプリケーション先の稼働を確認する 

(2) 復旧手順

メインサイトに対する障害の種類として以下のパターンの障害の検証手順を記載します。

case1. マスタサーバ障害時の挙動

<u>マスタサーバの電源断</u>

サービス提供中にマスタサーバに障害が発生したことをシミュレートするため、クライアントで SQL 実行スクリプトを実行した状態で、マスタサーバの電源ボタン長押しで電源断しました。

SQL 実行スクリプトでクライアントから接続中のセッションが切断され、エラーが返ることを確認できました。

```
1,2015年  1月 21日 水曜日 09:20:13 JST,SUCCESS
2,2015年  1月 21日 水曜日 09:20:16 JST,SUCCESS
3,2015年  1月 21日 水曜日 09:20:19 JST,SUCCESS
psql: タイムアウト期間が過ぎました
4,2015年  1月 21日 水曜日 09:20:22 JST,ERROR
psql: タイムアウト期間が過ぎました
5,2015年  1月 21日 水曜日 09:20:27 JST,ERROR
```

図 3.8: マスタサーバ障害時のセッションの挙動

メイン側スタンバイサーバ、DR 側スタンバイサーバがレプリケーション状態のままであることを、pg_controldata の「データベースクラスタの状態」が「アーカイブリカバリ中」（もしくは、「Database cluster state」が「in archive recovery」）で確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106235617320065236
データベースクラスタの状態:              アーカイブリカバリ中
```

図 3.9: マスタサーバ障害時のメイン側スタンバイサーバの状態

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106235617320065236
データベースクラスタの状態:              アーカイブリカバリ中
```

図 3.10: マスタサーバ障害時の DR 側スタンバイサーバの状態

メイン側スタンバイサーバをマスタサーバとして復旧

次に、メイン側スタンバイサーバをマスタサーバに昇格するため、promote コマンドを発行します。

```
-bash-4.1$ pg_ctl promote
サーバを昇進中です。
```

図 3.11: メイン側スタンバイサーバを昇格

マスタサーバに昇格したことが、pg_controldata の「データベースクラスタの状態」が「運用中」（もしくは、「Database cluster state」が「in production」）に変更、および、クライアントで実行していた SQL 実行スクリプトのセッションが回復したことにより確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106235617320065236
データベースクラスタの状態:              運用中
```

図 3.12: メイン側スタンバイサーバ→マスタサーバ昇格時の状態

```
ERROR:  cannot execute INSERT in a read-only transaction
1,2015年  1月 21日 水曜日 09:30:04 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
2,2015年  1月 21日 水曜日 09:30:07 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
3,2015年  1月 21日 水曜日 09:30:10 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
4,2015年  1月 21日 水曜日 09:30:13 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
5,2015年  1月 21日 水曜日 09:30:16 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
6,2015年  1月 21日 水曜日 09:30:19 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
7,2015年  1月 21日 水曜日 09:30:22 JST,ERROR
8,2015年  1月 21日 水曜日 09:30:25 JST,SUCCESS
9,2015年  1月 21日 水曜日 09:30:28 JST,SUCCESS
```

図 3.13: メイン側スタンバイサーバ→マスタサーバ昇格時のセッションの挙動

レプリケーション状態に復旧
DR 側スタンバイサーバのレプリケーションは、マスタサーバを切り替えたため、未接続状態になっています(図 3.14)。

```
postgres=# SELECT application_name,client_addr, state, sync_state, pg_xlog_location_
diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_lo
cation, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, rep
lay_location) as replay_diff_byte FROM pg_stat_replication;
 application_name | client_addr | state | sync_state | send_diff_byte | flush_diff_b
yte | replay_diff_byte
-----------------+-------------+-------+------------+----------------+-------------
----+-----------------
(0 行)
```

図 3.14: レプリケーション状態

よって、マスタサーバに障害が発生した場合はレプリケーションが維持できないため、レプリケーション復旧手順を明確にしておく必要があることが確認できます。

レプリケーションを復旧にするには、DR 側スタンバイサーバの接続先を切り替えた先のマスタサーバを再設定する必要があります。具体的には、recovery.conf の primary_conninfo と restore_command のパラメータを新マスタサーバに再設定し直し(図 3.15)、PostgreSQL サーバを再起動(図 3.16)します。その後、レプリケーション元でレプリケーション状態を確認すると、復旧したことを確認できました(図 3.17)。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.2.143 port=5432 user=postgres'
restore_command = 'scp 192.168.2.143:/usr/local/src/pgsql/9.4.0/data/pgarc/%f %p'
recovery_target_timeline = 'latest'
```

図 3.15: recovery.conf の再設定

```
-bash-4.1$ pg_ctl restart
サーバ停止処理の完了を待っています....完了
サーバは停止しました
サーバは起動中です。
-bash-4.1$ 2015-03-05 11:38:25 JST  [1430-1]LOG:  redirecting log output to logging
collector process
2015-03-05 11:38:25 JST  [1430-2]HINT:  Future log output will appear in directory "
pg_log".
```

図 3.16: PostgreSQL サーバ再起動

```
postgres=# SELECT application_name,client_addr, state, sync_state, pg_xlog_location_
diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_lo
cation, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, rep
lay_location) as replay_diff_byte FROM pg_stat_replication;
 application_name |  client_addr  |   state   | sync_state | send_diff_byte | flush_
diff_byte | replay_diff_byte
------------------+---------------+-----------+------------+----------------+-------
----------+------------------
 walreceiver      | 192.168.2.144 | streaming | async      |              0 |
        0 |                0
(1 行)
```

図 3.17: レプリケーション稼働確認 (pg_stat_replication ビュー)

case2. メイン側スタンバイサーバ障害時の挙動

メイン側スタンバイサーバの電源断
サービス提供中にメイン側スタンバイサーバに障害が発生したことをシミュレートするため、クライアントで SQL 実行スクリプトを実行した状態で、メイン側スタンバイサーバの電源ボタン長押しで電源断しました。
接続中のセッションに影響がないことが確認できました。

```
1,2015年  1月 20日 火曜日 17:28:33 JST,SUCCESS
2,2015年  1月 20日 火曜日 17:28:36 JST,SUCCESS
3,2015年  1月 20日 火曜日 17:28:39 JST,SUCCESS
4,2015年  1月 20日 火曜日 17:28:42 JST,SUCCESS
5,2015年  1月 20日 火曜日 17:28:45 JST,SUCCESS
6,2015年  1月 20日 火曜日 17:28:48 JST,SUCCESS
```

図 3.18: メイン側スタンバイサーバ障害時のセッションの挙動

また、レプリケーションの状態を pg_stat_replication ビューで確認すると、マスタサーバと DR 側スタンバイサーバはレプリケーションを維持できていることが確認できました。

```
postgres=# select application_name, client_addr, state, sync_state, pg_xlog_location
_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_l
ocation, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, re
play_location) as replay_diff_byte from pg_stat_replication;
 application_name |  client_addr  |   state   | sync_state | send_diff_byte | flush_
diff_byte | replay_diff_byte
------------------+---------------+-----------+------------+----------------+-------
----------+------------------
 walreceiver      | 192.168.2.144 | streaming | async      |              0 |
        0 |                0
(1 行)
```

図 3.19: メイン側スタンバイサーバ障害時のレプリケーション状態の確認

さらに、マスタサーバと DR 側スタンバイサーバのレプリケーションの進捗位置を確認するため、トランザクションログの位置を取得すると、一致していることが確認できました。

```
postgres=# select pg_current_xlog_location();
 pg_current_xlog_location
--------------------------
 0/12000E78
(1 行)
```

図 3.20: トランザクションログの書き込み位置を取得

```
postgres=# select pg_last_xlog_receive_location();
 pg_last_xlog_receive_location
-------------------------------
 0/12000E78
(1 行)
```

図 3.21: ストリーミングレプリケーションで受信したトランザクションログの最後の位置を取得

case3. メインサイト全体障害時の挙動

メインサイトの電源断

サービス提供中にメインサイトに障害が発生したことをシミュレートするため、クライアントで SQL 実行スクリプトを実行した状態で、マスタサーバとメイン側スタンバイサーバの電源ボタン長押しで電源断しました。

SQL 実行スクリプトでクライアントから接続中のセッションが切断され、復旧するまでエラーを返すことを確認できました。

```
1,2015年  1月 20日 火曜日 14:49:27 JST,SUCCESS
2,2015年  1月 20日 火曜日 14:49:29 JST,SUCCESS
psql: タイムアウト期間が過ぎました
3,2015年  1月 20日 火曜日 14:49:31 JST,ERROR
psql: タイムアウト期間が過ぎました
4,2015年  1月 20日 火曜日 14:49:35 JST,ERROR
```

図 3.22: メインサイト障害時のセッションの挙動

DR 側スタンバイサーバがレプリケーション状態のままであることを、pg_controldata の「データベースクラスタの状態」が「アーカイブリカバリ中」（もしくは「Database cluster state」が「in archive recovery」）で確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106235617320065236
データベースクラスタの状態:              アーカイブリカバリ中
```

図 3.23: メインサイト障害時の DR 側スタンバイサーバの挙動

DR 側スタンバイサーバをマスタサーバとして復旧

次に、DR 側スタンバイサーバをマスタサーバに昇格するため、promote コマンドを発行します。

```
-bash-4.1$ pg_ctl promote
サーバを昇進中です。
```

図 3.24: DR 側スタンバイサーバをマスタサーバ昇格

マスタサーバに昇格したことが、pg_controldata の「データベースクラスタの状態」が「運用中」(もしくは、「Database cluster state」が「in production」)に変更、および、クライアントで実行していた SQL 実行スクリプトのセッションが回復したことにより確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106235617320065236
データベースクラスタの状態:              運用中
```

図 3.25: DR 側スタンバイサーバ→マスタサーバ昇格時の状態

```
ERROR:  cannot execute INSERT in a read-only transaction
1,2015年  1月 20日 火曜日 14:52:49 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
2,2015年  1月 20日 火曜日 14:52:51 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
3,2015年  1月 20日 火曜日 14:52:53 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
4,2015年  1月 20日 火曜日 14:52:55 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
5,2015年  1月 20日 火曜日 14:52:57 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
6,2015年  1月 20日 火曜日 14:52:59 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
7,2015年  1月 20日 火曜日 14:53:01 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
8,2015年  1月 20日 火曜日 14:53:03 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
9,2015年  1月 20日 火曜日 14:53:05 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
10,2015年  1月 20日 火曜日 14:53:07 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
11,2015年  1月 20日 火曜日 14:53:09 JST,ERROR
12,2015年  1月 20日 火曜日 14:53:11 JST,SUCCESS
13,2015年  1月 20日 火曜日 14:53:13 JST,SUCCESS
14,2015年  1月 20日 火曜日 14:53:15 JST,SUCCESS
15,2015年  1月 20日 火曜日 14:53:17 JST,SUCCESS
```

図 3.26: DR 側スタンバイサーバ→マスタサーバ昇格時のセッションの挙動

3.1.1.4.考察

(1) 構築容易性

マルチスタンバイ構成では、各スタンバイサーバの構成は同じになるので、スタンバイサーバの複製は容易に実行できることが確認できました。このことから、各スタンバイサーバが同等の管理方法となるため、単純にスタンバイサーバを増減することで参照パフォーマンスの負荷分散を動的に実施することができます。

(2) 障害影響度

マルチスタンバイ構成でマスタサーバに障害が発生した場合には、**レプリケーション接続元にマスタサーバを設定している全てのサーバのレプリケーションが停止してしまいます。**レプリケーションを復旧するには特定のスタンバイサーバをマスタサーバに昇格、並びに、その他スタンバイサーバの設定変更が必要になります。さらに、昇格したマスタサーバからの WAL を受け取るまではレプリケーションを再開できないことが確認できました。

このため、メインサイト内の障害復旧処理を自動化している場合は特に注意して、DR サイトとのレプリケーションの設定や復旧手順等を一緒に整備しておく必要があります。

3.1.2. カスケード構成

本項では、PostgreSQL のストリーミングレプリケーション機能を用いたカスケード構成に対する検証を行います。

3.1.2.1.検証環境

動作検証に用いる DR 構成の構築は、メインサイトにマスタサーバおよびメイン側スタンバイサーバを構築し、DR サイトに DR 側スタンバイサーバを構築します。カスケード構成ではマスタサーバからメイン側スタンバイサーバに対し WAL が転送されて同期処理が行われ、メイン側スタンバイサーバから DR サイト側スタンバイサーバに対し WAL が転送されて同期処理が行われます。データ同期は、手順の確立を目的としており、パフォーマンス測定を目的としていないため、レプリケーションは全て非同期設定にします。カスケード構成として今回使用した検証環境を図 3.27 に示します。

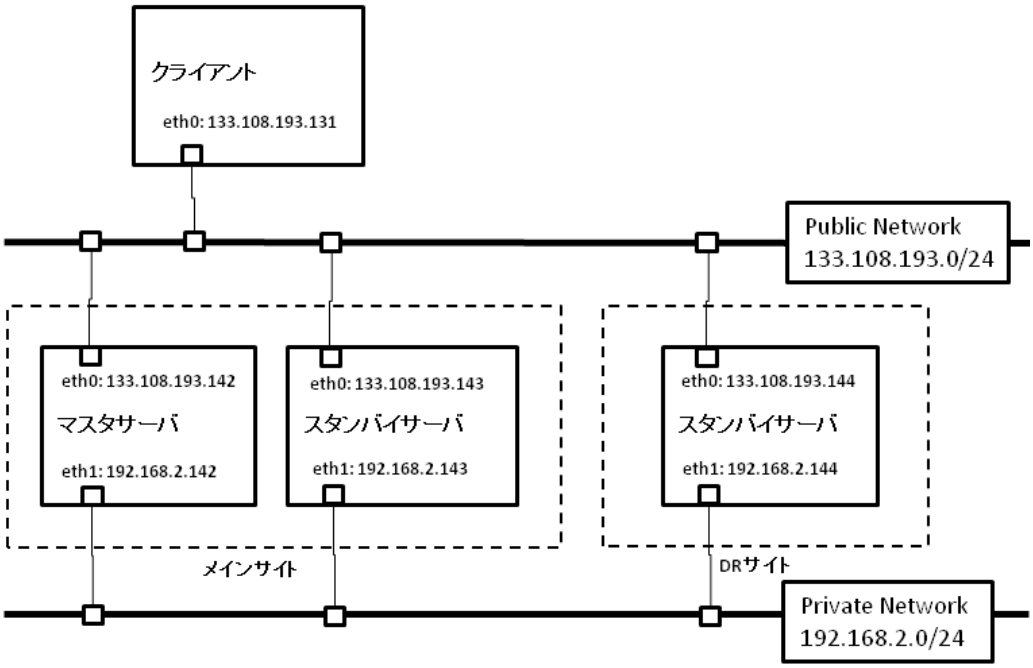


図 3.27: カスケード構成概要

(1).システム構成

表 3.7: 検証環境のシステム構成

構成情報	設定内容	備考
OS	RHEL/CentOS 6.4 x86_64	
ハードウェア・アーキテクチャ	x86_64	PostgreSQL サーバはマスタとスタンバイ 2 台で計 3 台用意
PostgreSQL	9.4.0	
PostgreSQL スーパユーザ名	postgres	
前提 AP	openssh-clients	restore_command で scp を使用するため

(2).ネットワーク構成

表 3.8: 検証環境のネットワーク構成

構成サーバ名	構成情報	備考
マスタサーバ	eth0:133.108.193.142	Public Network
	eth1:192.168.2.142	Private Network
メイン側スタンバイサーバ	eth0:133.108.193.143	Public Network
	eth1:192.168.2.143	Private Network
DR 側スタンバイサーバ	eth0:133.108.193.144	Public Network
	eth1:192.168.2.144	Private Network
クライアント	eth0: 133.108.193.131	Public Network

(3).パラメータ構成

1. postgresql.conf

表 3.9: postgresql.conf (全サーバ)

パラメータ	補足

# date control 	
timezone = 'Japan'
	日付書式関連パラメータ

# network control 	
listen_addresses = '*'
	ネットワーク関連パラメータ

#Performance control
max_connections = 100
shared_buffers = 200MB
wal_buffers = 1MB
checkpoint_segments = 10
	パフォーマンス関連パラメータ

#log control
logging_collector = on
log_connections = on
log_line_prefix = '%t %d [%p-%l]'
log_timezone = 'Japan'
log_filename = 'postgresql-%Y-%m-%d.log'
	ログ出力関連パラメータ

#Streaming Replication Primary
wal_level = hot_standby
archive_mode = on
archive_command = 'test ! -f /usr/local/src/pgsql/9.4.0/data/pgarc/%f
&& cp %p /usr/local/src/pgsql/9.4.0/data/pgarc/%f'
max_wal_senders = 4
hot_standby = on
	ストリーミングレプリケーション関連パラメータ

2. pg_hba.conf

表 3.10: pg_hba.conf (全サーバ)

パラメータ
host all postgres 133.108.193.131/32 md5 host postgres postgres 133.108.193.142/32 md5 host postgres postgres 133.108.193.143/32 md5 host postgres postgres 133.108.193.144/32 md5

```
host   replication  postgres       192.168.2.142/32        md5
host   replication  postgres       192.168.2.143/32        md5
host   replication  postgres       192.168.2.144/32        md5
```

3. recovery.conf

表 3.11: recovery.conf(メイン側スタンバイサーバ)

パラメータ
standby_mode = 'on' primary_conninfo = 'host=**192.168.2.142** port=5432 user=postgres restore_command = 'scp **192.168.2.142**:/usr/local/src/pgsql/9.4.0/data/pgarc/%f %p' recovery_target_timeline = 'latest'

表 3.12: recovery.conf(DR 側スタンバイサーバ)

パラメータ
standby_mode = 'on' primary_conninfo = 'host=**192.168.2.143** port=5432 user=postgres restore_command = 'scp **192.168.2.143**:/usr/local/src/pgsql/9.4.0/data/pgarc/%f %p' recovery_target_timeline = 'latest'

- primary_conninfo は、WAL ファイル提供元であるサーバへの接続情報を定義します。
- restore_command は、リカバリ時に使用する WAL ファイルの場所を定義します。

3.1.2.2.検証ケース

(1) 構築手順

実際に検証環境に構築を行い、構築手順の確立と構築するにあたっての懸念点がないか検証します。

(2) 復旧手順

メインサイトに障害が発生した場合の挙動を検証します。

それぞれ、クライアントからのセッションの状態は、SQL 実行スクリプトを作成して、PostgreSQL に定期的に以下のクエリを発行することで、状態を確認しました。

```
# SQL 実行スクリプト
export PGPASSWORD=postgres

PSQL=/usr/bin/psql
HOST=133.108.193.142   # 接続先はマスタサーバ
DATABASE=test01
COUNT=1

while true
do
 QUERY="insert into test01 values ($COUNT,'key01','value');"

 val1=$(date)
 val2=$(PGCONNECT_TIMEOUT=1 $PSQL -h $HOST -p 5432 -U postgres -d $DATABASE -q -t -c "$QUERY")
 if [ $? -ne 0 ]; then
  val2="ERROR"
 else
  val2="SUCCESS"
 fi
 echo "$COUNT,${val1},${val2}"

 COUNT=$(($COUNT+1))
 sleep 1
done
```

図 3.28: SQL 実行スクリプト

case1. マスタサーバ障害時の挙動

マスタサーバに障害が発生した場合の挙動を検証します。検証ケースは、サーバ電源が落ちた場合の挙動と、その後のサービス復旧を想定しました。マスタサーバに障害が発生すると、クライアントからのセッションは切断され、復帰までエラーを返します。サービスの復旧は、手作業による切り替えを行い、切り替えが完了後に再び接続できるようになります。ここでは、マスタサーバがダウンしたことの確認と、サービスの復旧手順の確認を実施します。

case2. メイン側スタンバイサーバ障害時の挙動

メイン側スタンバイサーバに障害が発生した場合の挙動を検証します。検証ケースは、case1 と同様にサーバ電源が落ちた場合の挙動を想定しました。メイン側スタンバイサーバがダウンするためレプリケーションは途切れますが、マスタサーバはダウンしていないため、サービスへの影響はありません。**但し、DR 側スタンバイサーバのレプリケーション元はメイン側スタンバイサーバのため、レプリケーションは途切れてしまいます。**ここでは、メイン側スタンバイサーバがダウンしたことの確認と、レプリケーションの復旧手順を実施します。

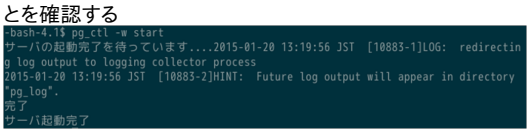
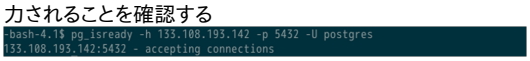
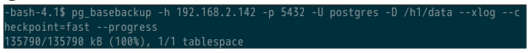
case3. メインサイト全体障害時の挙動

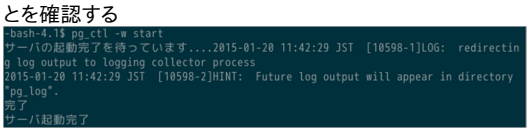
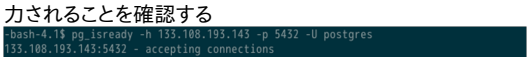
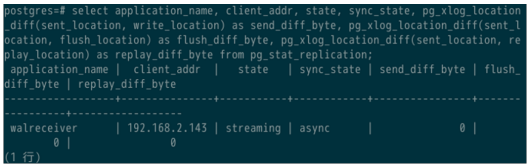
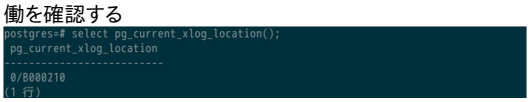
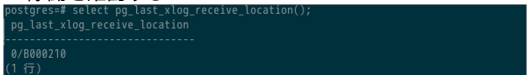
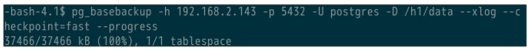
メインサイト全体の電源が落ちた場合の挙動と、その後のサービス復旧を想定しました。メインサイトには、マスタサーバとメイン側スタンバイサーバが設置してあることを想定しています。まず、メインサイトに障害が発生すると、マスタサーバに接続していたクライアントからのセッションは切断され、復帰までエラーを返します。一方、サービスの復旧は、PostgreSQL の機能ではサービスの切り替えを自動的に行えないため、手作業による切り替えを行う必要があり、切り替えが完了後に再び接続できるようになります。ここでは、メインサイトがダウンしたことの確認と、サービスの復旧手順の確認を実施します。なお、障害前の構成まで復旧する手順の確認は、今回の検証からは除外しました。

3.1.2.3.検証結果

(1) 構築手順

前提として PostgreSQL のインストールは完了しているものとします。

分類	No	作業内容	手順	確認
マスタサーバの構築	1	DB クラスタ作成	クラスタ DB を初期化する (コマンド例) initdb --encoding=UTF8 --no-locale --data-checksums -k -A md5 -W	initdb コマンドが正常に終了することを確認する 
	2	マスタサーバのパラメータを設定する	環境構成シートに記載のパラメータファイルを配置する ・postgresql.conf ・pg_hba.conf	設定が正しいことを 3.1.2.1 を参考に確認する
	3	PostgreSQL マスタサーバを起動する	PostgreSQL サーバを起動する (コマンド例) pg_ctl -w start	実行したコンソール上に「server started」と出力されることを確認する 
	4	正常稼働チェック	PostgreSQL サーバに対し pg_isready を実行し正常に起動していることを確認する (コマンド例) pg_isready -h ${TARGETHOSTNAME} -p ${TARGETPORT} -U ${TARGETUNAME} -d ${TARGETDBNAME}	実行したコンソール上に「accepting connections」と出力されることを確認する 
メイン側スタンバイサーバの構築	1	データをコピーする	マスタサーバからデータをコピーする (コマンド例) pg_basebackup -h ${HOSTNAME_Primary_pri} -p ${PORT_Primary} -U postgres -D ${PGDATA_Secondary} --xlog --checkpoint=fast --progress ※${HOSTNAME_Primary_pri}の設定値には、レプリケーション用 IP アドレスを指定	pg_basebackup コマンドが正常に終了することを確認する 
	2	メイン側スタンバイサーバのパラメータを設定する	環境構成シートに記載のパラメータファイルを配置する ・postgresql.conf ・pg_hba.conf ・recovery.conf	設定が正しいことを 3.1.2.1 を参考に確認する

分類	No	作業内容	手順	確認
	3	PostgreSQL メイン側スタンバイサーバを起動する	PostgreSQL サーバを起動する (コマンド例) pg_ctl -w start	実行したコンソール上に「server started」と出力されることを確認する 
	4	正常稼働チェック	PostgreSQL サーバに対し pg_isready を実行し正常に起動していることを確認する (コマンド例) pg_isready -h ${TARGETHOSTNAME} -p ${TARGETPORT} -U ${TARGETUNAME} -d ${TARGETDBNAME}	実行したコンソール上に「accepting connections」と出力されることを確認する 
	5	レプリケーション稼働チェック	レプリケーション元 PostgreSQL サーバに対し pg_stat_replication ビューを参照し正常に起動していることを確認する (コマンド例) SELECT application_name ,client_addr ,state ,sync_state, pg_xlog_location_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_location, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, replay_location) as replay_diff_byte FROM pg_stat_replication;	レプリケーション元で pg_stat_replication の client_addr でクライアント接続元、state で接続状態を確認する。 
			1) レプリケーション元 稼働チェック レプリケーション元 PostgreSQL サーバに対し pg_current_xlog_location を参照し正常に結果を返すことを確認する。 2) レプリケーション先 稼働チェック レプリケーション先 PostgreSQL サーバに対し pg_last_xlog_receive_location を参照し正常に結果を返すことを確認する。 1)と 2)の値が一致していることを確認する。	1) pg_current_xlog_location でレプリケーション元の稼働を確認する  2) pg_last_xlog_receive_location でレプリケーション先の稼働を確認する 
DR側スタンバイサーバの構築	1	データをコピーする	メイン側スタンバイサーバからデータをコピーする (コマンド例) pg_basebackup -h ${HOSTNAME_Secondary_pri} -p ${PORT_Secondary} -U postgres -D ${PGDATA_Tertiary} --xlog --checkpoint=fast --progress ※${HOSTNAME_Secondary_pri}の設定値には、レプリケーション用 IP アドレスを指定	pg_basebackup コマンドが正常に終了することを確認する 
	2	DR 側スタンバイサーバのパラメータを設定する	環境構成シートに記載のパラメータファイルを配置する ・postgresql.conf ・pg_hba.conf ・recovery.conf	設定が正しいことを 3.1.2.1 を参考に確認する
	3	PostgreSQL DR 側ス	PostgreSQL サーバを起動する	実行したコンソール上に「server started」と出力されるこ

分類	No	作業内容	手順	確認

		タンバイサーバを起動する	(コマンド例)
			pg_ctl -w start
				とを確認する
				`-bash-4.1$ pg_ctl -w start`
				`サーバの起動完了を待っています....2015-01-20 13:51:31 JST [10528-1]LOG: redirecting log output to logging collector process`
				`2015-01-20 13:51:31 JST [10528-2]HINT: Future log output will appear in directory "pg_log".`
				`完了`
				`サーバ起動完了`

	4	正常稼働チェック	PostgreSQL サーバに対し pg_isready を実行し正常に起動していることを確認する
			(コマンド例)
			pg_isready -h ${TARGETHOSTNAME} -p ${TARGETPORT} -U ${TARGETUNAME} -d ${TARGETDBNAME}
				実行したコンソール上に「accepting connections」と出力されることを確認する
				`-bash-4.1$ pg_isready -h 133.108.193.144 -p 5432 -U postgres`
				`133.108.193.144:5432 - accepting connections`

	5	レプリケーション稼働チェック	レプリケーション元 PostgreSQL サーバに対し pg_stat_replication ビューを参照し正常に起動していることを確認する (コマンド例) SELECT application_name ,client_addr ,state ,sync_state, pg_xlog_location_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_location, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, replay_location) as replay_diff_byte FROM pg_stat_replication;	レプリケーション元で pg_stat_replication の client_addr でクライアント接続元、state で接続状態を確認する。 `postgres=# select application_name, client_addr, state, sync_state, pg_xlog_location_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_location, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, replay_location) as replay_diff_byte from pg_stat_replication;` `application_name | client_addr | state | sync_state | send_diff_byte | flush_diff_byte | replay_diff_byte` `walreceiver | 192.168.2.144 | streaming | async | 0 | 0 | 0` `(1 行)`

			1) レプリケーション元 稼働チェック
			レプリケーション元 PostgreSQL サーバに対し pg_current_xlog_location を参照し正常に結果を返すことを確認する。
			
			2) レプリケーション先 稼働チェック
			レプリケーション先 PostgreSQL サーバに対し pg_last_xlog_receive_location を参照し正常に結果を返すことを確認する。
			
			1) と 2) の値が一致していることを確認する。
				1) pg_current_xlog_location でレプリケーション元の稼働を確認する
				`postgres=# select pg_last_xlog_receive_location();`
				`pg_last_xlog_receive_location`
				`0/3000570`
				`(1 行)`
				2) pg_last_xlog_receive_location でレプリケーション先の稼働を確認する
				`postgres=# select pg_last_xlog_receive_location();`
				`pg_last_xlog_receive_location`
				`0/3000570`
				`(1 行)`

(2) 復旧手順

メインサイトに対する障害の種類として以下のパターンの障害の検証手順を記載します。

case1. マスタサーバ障害時の挙動

マスタサーバの電源断

サービス提供中にマスタサーバに障害が発生したことをシミュレートするため、クライアントで SQL 実行スクリプトを実行した状態で、マスタサーバの電源ボタン長押しで電源断しました。

SQL 実行スクリプトでクライアントから接続中のセッションが切断され、エラーを返すことを確認できました。

```
[inagaki@cob2 zzz]$ ./psql.sh
1,2015年  1月 21日 水曜日 15:34:59 JST,SUCCESS
2,2015年  1月 21日 水曜日 15:35:01 JST,SUCCESS
3,2015年  1月 21日 水曜日 15:35:03 JST,SUCCESS
psql: タイムアウト期間が過ぎました
4,2015年  1月 21日 水曜日 15:35:05 JST,ERROR
psql: タイムアウト期間が過ぎました
5,2015年  1月 21日 水曜日 15:35:09 JST,ERROR
```

図 3.29: マスタサーバ障害時のセッションの挙動

メイン側スタンバイサーバ、DR 側スタンバイサーバがレプリケーション状態のままであることを、pg_controldata の「データベースクラスタの状態」が「アーカイブリカバリ中」（もしくは、「Database cluster state」が「in archive recovery」）で確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106603130502636760
データベースクラスタの状態:              アーカイブリカバリ中
```

図 3.30: マスタサーバ障害時のメイン側スタンバイサーバの状態

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106603130502636760
データベースクラスタの状態:              アーカイブリカバリ中
```

図 3.31: マスタサーバ障害時の DR 側スタンバイサーバの状態

メイン側スタンバイサーバをマスタサーバとして復旧

次に、メイン側スタンバイサーバをマスタサーバに昇格するため、promote コマンドを発行します。

```
-bash-4.1$ pg_ctl promote
サーバを昇進中です。
```

図 3.32: メイン側スタンバイサーバを昇格

マスタサーバに昇格したことが、pg_controldata の「データベースクラスタの状態」が「運用中」（もしくは、「Database cluster state」が「in production」）に変更、および、クライアントで実行していた SQL 実行スクリプトのセッションが回復したことにより確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106603130502636760
データベースクラスタの状態:              運用中
```

図 3.33: メイン側スタンバイサーバ→マスタサーバ昇格時の状態

```
[inagaki@cob2 zzz]$ ./psql.sh
ERROR:  cannot execute INSERT in a read-only transaction
1,2015年  1月 21日 水曜日 15:46:49 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
2,2015年  1月 21日 水曜日 15:46:51 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
3,2015年  1月 21日 水曜日 15:46:53 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
4,2015年  1月 21日 水曜日 15:46:55 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
5,2015年  1月 21日 水曜日 15:46:57 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
6,2015年  1月 21日 水曜日 15:46:59 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
7,2015年  1月 21日 水曜日 15:47:01 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
8,2015年  1月 21日 水曜日 15:47:03 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
9,2015年  1月 21日 水曜日 15:47:05 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
10,2015年  1月 21日 水曜日 15:47:07 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
11,2015年  1月 21日 水曜日 15:47:09 JST,ERROR
12,2015年  1月 21日 水曜日 15:47:11 JST,SUCCESS
13,2015年  1月 21日 水曜日 15:47:13 JST,SUCCESS
14,2015年  1月 21日 水曜日 15:47:15 JST,SUCCESS
```

図 3.34: メイン側スタンバイサーバ→マスタサーバ昇格時のセッションの挙動

カスケード構成ではマスタサーバ障害時にメイン側スタンバイサーバがマスタサーバに昇格した場合、レプリケーションは維持されます。DR 側スタンバイサーバは新マスタサーバをレプリケーション元として動作し続けます。(図 3.35)

```
postgres=# select application_name, client_addr, state, sync_state, pg_xlog_location
_diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_l
ocation, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, re
play_location) as replay_diff_byte from pg_stat_replication;
 application_name |  client_addr  |   state   | sync_state | send_diff_byte | flush_
diff_byte | replay_diff_byte
------------------+---------------+-----------+------------+----------------+-------
----------+------------------
 walreceiver      | 192.168.2.144 | streaming | async      |              0 |
        0 |                0
(1 行)
```

図 3.35: レプリケーション稼働確認 (pg_stat_replication ビュー)

case2. メイン側スタンバイサーバ障害時の挙動

メイン側スタンバイサーバの電源断

サービス提供中にメイン側スタンバイサーバに障害が発生したことをシミュレートするため、クライアントで SQL 実行スクリプトを実行した状態で、メイン側スタンバイサーバの電源ボタン長押しで電源断しました。

接続中のセッションに影響がないことが確認できました。

```
1,2015年  1月 20日 火曜日 17:28:33 JST,SUCCESS
2,2015年  1月 20日 火曜日 17:28:36 JST,SUCCESS
3,2015年  1月 20日 火曜日 17:28:39 JST,SUCCESS
4,2015年  1月 20日 火曜日 17:28:42 JST,SUCCESS
5,2015年  1月 20日 火曜日 17:28:45 JST,SUCCESS
6,2015年  1月 20日 火曜日 17:28:48 JST,SUCCESS
```

図 3.36: メイン側スタンバイサーバ障害時のセッションの挙動

レプリケーション状態に復旧

レプリケーションの状態を pg_stat_replication ビューで確認すると、マスタサーバはレプリケーションを維持できていないことが確認できました。

```
postgres=# SELECT application_name,client_addr, state, sync_state, pg_xlog_location_
diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_lo
cation, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, rep
lay_location) as replay_diff_byte FROM pg_stat_replication;
 application_name | client_addr | state | sync_state | send_diff_byte | flush_diff_b
yte | replay_diff_byte
-----------------+-------------+-------+------------+----------------+-------------
----+-----------------
(0 行)
```

図 3.37: メイン側スタンバイサーバ障害時のレプリケーション状態の確認

よって、メイン側スタンバイサーバに障害が発生した場合はレプリケーションが維持できないため、レプリケーション復旧手順を明確にしておく必要があることが確認できます。

レプリケーションを復旧にするには、DR 側スタンバイサーバの接続先をメイン側スタンバイサーバからマスタサーバに再設定する必要があります。具体的には、recovery.conf の primary_conninfo と restore_command のパラメータを新マスタサーバに再設定し直し(図 3.38)、PostgreSQL サーバを再起動(図 3.39)します。その後、レプリケーション元でレプリケーション状態を確認すると、復旧したことを確認できました(図 3.40)。

```
standby_mode = 'on'
primary_conninfo = 'host=192.168.2.142 port=5432 user=postgres'
restore_command = 'scp 192.168.2.142:/usr/local/src/pgsql/9.4.0/data/pgarc/%f %p'
recovery_target_timeline = 'latest'
```

図 3.38: recovery.conf の再設定

```
-bash-4.1$ pg_ctl restart
サーバ停止処理の完了を待っています....完了
サーバは停止しました
サーバは起動中です。
-bash-4.1$ 2015-03-05 11:38:25 JST  [1430-1]LOG:  redirecting log output to logging
collector process
2015-03-05 11:38:25 JST  [1430-2]HINT:  Future log output will appear in directory "
pg_log".
```

図 3.39: PostgreSQL サーバ再起動

```
postgres=# SELECT application_name,client_addr, state, sync_state, pg_xlog_location_
diff(sent_location, write_location) as send_diff_byte, pg_xlog_location_diff(sent_lo
cation, flush_location) as flush_diff_byte, pg_xlog_location_diff(sent_location, rep
lay_location) as replay_diff_byte FROM pg_stat_replication;
 application_name |  client_addr  |   state   | sync_state | send_diff_byte | flush_
diff_byte | replay_diff_byte
------------------+---------------+-----------+------------+----------------+-------
----------+------------------
 walreceiver      | 192.168.2.144 | streaming | async      |              0 |
        0 |                0
(1 行)
```

図 3.40: レプリケーション稼働確認 (pg_stat_replication ビュー)

case3. メインサイト全体障害時の挙動

メインサイトの電源断

サービス提供中にメインサイトに障害が発生したことをシミュレートするため、クライアントで SQL 実行スクリプトを実行した状態で、マスタサーバとメイン側スタンバイサーバの電源ボタン長押しで電源断しました。

SQL 実行スクリプトでクライアントから接続中のセッションが切断され、復旧するまでエラーを返すことを確認できました。

```
[inagaki@cob2 zzz]$ ./psql.sh
1,2015年  1月 21日 水曜日 14:15:23 JST,SUCCESS
2,2015年  1月 21日 水曜日 14:15:25 JST,SUCCESS
3,2015年  1月 21日 水曜日 14:15:27 JST,SUCCESS
psql: タイムアウト期間が過ぎました
4,2015年  1月 21日 水曜日 14:15:29 JST,ERROR
psql: タイムアウト期間が過ぎました
5,2015年  1月 21日 水曜日 14:15:33 JST,ERROR
```

図 3.41: メインサイト障害時のセッションの挙動

DR 側スタンバイサーバがレプリケーション状態のままであることを、pg_controldata の「データベースクラスタの状態」が「アーカイブリカバリ中」（もしくは、「Database cluster state」が「in archive recovery」）で確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106603130502636760
データベースクラスタの状態:              アーカイブリカバリ中
```

図 3.42: メインサイト障害時の DR 側スタンバイサーバの挙動

DR 側スタンバイサーバをマスタサーバとして復旧

次に、DR 側スタンバイサーバをマスタサーバに昇格するため、promote コマンドを発行します。

```
-bash-4.1$ pg_ctl promote
サーバを昇進中です。
```

図 3.43: DR 側スタンバイサーバをマスタサーバ昇格

マスタサーバに昇格したことが、pg_controldata の「データベースクラスタの状態」が「運用中」（もしくは、「Database cluster state」が「in production」）に変更、および、クライアントで実行していた SQL 実行スクリプトのセッションが回復したことにより確認できました。

```
-bash-4.1$ pg_controldata
pg_controlバージョン番号:                942
カタログバージョン番号:                  201409291
データベースシステム識別子:              6106603130502636760
データベースクラスタの状態:              運用中
```

図 3.44: DR 側スタンバイサーバ→マスタサーバ昇格時の状態

```
[inagaki@cob2 zzz]$ ./psql.sh
ERROR:  cannot execute INSERT in a read-only transaction
1,2015年  1月 21日 水曜日 14:26:26 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
2,2015年  1月 21日 水曜日 14:26:28 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
3,2015年  1月 21日 水曜日 14:26:30 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
4,2015年  1月 21日 水曜日 14:26:32 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
5,2015年  1月 21日 水曜日 14:26:34 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
6,2015年  1月 21日 水曜日 14:26:36 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
7,2015年  1月 21日 水曜日 14:26:38 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
8,2015年  1月 21日 水曜日 14:26:40 JST,ERROR
ERROR:  cannot execute INSERT in a read-only transaction
9,2015年  1月 21日 水曜日 14:26:42 JST,ERROR
10,2015年  1月 21日 水曜日 14:26:44 JST,SUCCESS
11,2015年  1月 21日 水曜日 14:26:46 JST,SUCCESS
```

図 3.45: DR 側スタンバイサーバ→マスタサーバ昇格時のセッションの挙動

3.1.2.4.考察

(1) 構築容易性

カスケード構成では、各サイトのスタンバイサーバの構成が異なるため、スタンバイサーバの複製はレプリケーション元、レプリケーション先を考慮することが必要となります。このことから、各スタンバイサーバは各々の役割に合わせた運用が必要となります。

(2) 障害影響度

カスケード構成でマスタサーバの障害が発生した場合、メイン側スタンバイサーバと DR 側スタンバイサーバは設定の変更なく、レプリケーション状態を維持しており、メイン側スタンバイサーバをマスタサーバに昇格すれば、昇格したマスタサーバから継続して WAL を受け取ることができました。

しかし、メイン側スタンバイサーバに障害が発生した場合は、**レプリケーション接続元にメイン側スタンバイサーバを設定している全てのサーバのレプリケーションが停止してしまいます。**これらのサーバのレプリケーションが復旧するには、接続先の設定変更を行って、さらに、WAL を受け取るまではレプリケーションは再開できないことが確認できました。

このため、カスケード構成ではシステム全体のレプリケーションが停止しないようにメイン側スタンバイサーバに対し障害監視を行うことや、メイン側スタンバイサーバを多重化するなどの検討が必要となります。

3.2. 性能検証

本章では PostgreSQL のストリーミングレプリケーション時のマスタやスタンバイの性能を詳細に確認するための検証結果について報告します。

3.2.1. 検証の観点と手法

検証は以下の 3 つの観点で行いました。

レプリケーション方式による性能特性の違い

例えば、2.2.3 章で紹介したマスタ⇒遠隔地 DB 非同期レプリケーション(以下マルチスタンバイレプリケーション構成)はマスタから直結しているスタンバイが 2 台あるため、直結しているスタンバイが 1 台しかない 2.2.5 章で紹介したマスタ⇒メインサイト側スタンバイ⇒遠隔地 DB カスケードレプリケーション(以下カスケードレプリケーション構成)に比べて負荷が高くなることが予想されます。
そこで、本検証ではトランザクションが大量に発行された際のマスタの性能をレプリケーションの構成によって比較します。

レプリケーション遅延の実態

PostgreSQL のストリーミングレプリケーションは、マスタが WAL レコード(以下 WAL)をスタンバイに転送し、スタンバイが順次それを適用することで実現しています。しかしマスタとスタンバイの拠点間のネットワーク性能等によって WAL の転送する速度は変わるため、スタンバイが WAL を適用するのに時間差が生じる場合があります。
そこで本検証ではスタンバイが WAL を適用する時間差を、マスタと比べスタンバイがまだ適用できていない WAL の量(byte)という観点から調査します。

レプリケーション方式特有のリスクと対策方法

遅延しているレプリケーションでは、スタンバイがまだ適用していない WAL をマスタが削除する可能性があります。そうなればスタンバイはレプリケーションを継続することができなくなります。
本検証では、2.2.3 章のマルチスタンバイ構成でレプリケーションが途切れた状態から、レプリケーションを復旧させる方法を確認します。

3.2.2. 検証環境

本検証で利用した検証環境は以下の通りです。

(1) ソフトウェア
検証には、PostgreSQL9.4.0 を用いました。これは本検証実施時(2015 年 2 月 20 日)の最新バージョンです。

(2) ハードウェア
メインサイトと DR のための遠隔地サイトはそれぞれ Amazon Web Service(以下 AWS)で構築しました。
サーバのスペックは表 3.13 の通りです。

表 3.13: サーバスペック一覧

マシン(ホスト名)	項目	AWS のインスタンスタイプとスペック
master	インスタンスタイプ	t2.medium
	CPU	Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
	メモリ	4GB
	NIC 個数	1(eth0)
	OS	Amazon Linux AMI 2014.09.2
	DISK	32GB(EBS で追加)
slave. singapore-slave	インスタンスタイプ	t2.micro
	CPU	Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
	メモリ	1GB
	NIC 個数	1(eth0)
	OS	Amazon Linux AMI 2014.09.2
	DISK	8GB

(3)サーバ構成
検証に用いた環境のサーバ構成を図 3.46 図 3.47 に示します。検証はメインサイトは国内に、DR サイトは海外にあることを想定して行うため、両サイトとも AWS で構築しました。

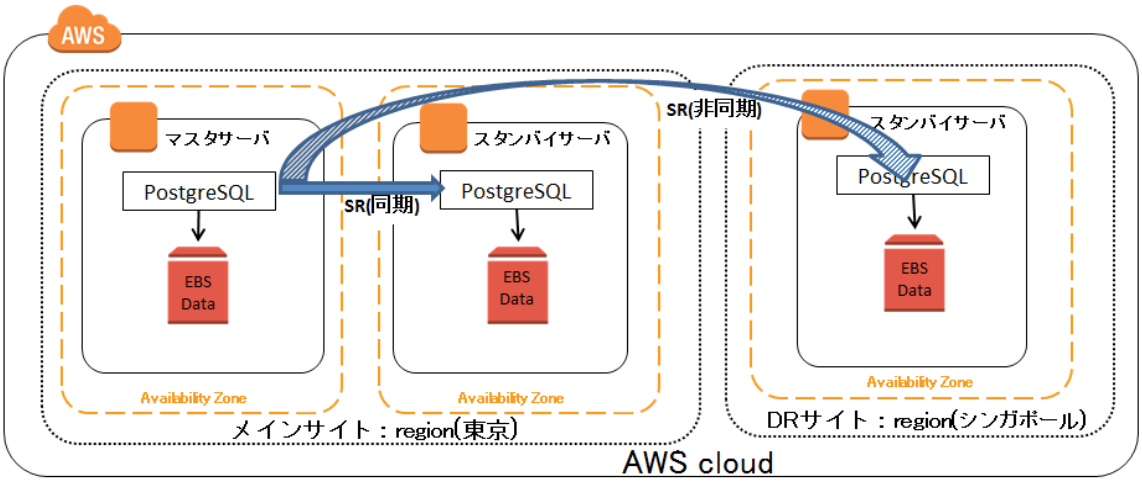


図 3.46: マルチスタンバイレプリケーション構成

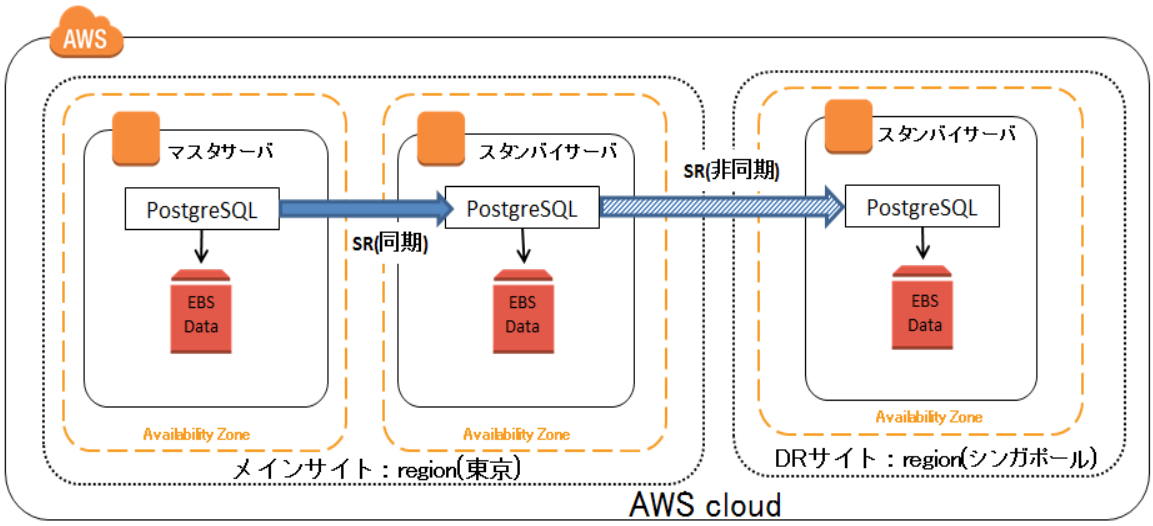


図 3.47: カスケードレプリケーション構成

表 3.14: 検証に用いたサーバー覧

ホスト名	サーバの設置場所	役割	IP アドレス	AWS のインスタンスタイプ
master	東京	レプリケーションのマスタ	54.65.232.0	t2.medium
slave	東京	レプリケーションの同期スタンバイ	54.92.66.190	t2.micro
singapore-slave	シンガポール	レプリケーションの非同期スタンバイ	54.251.130.204	t2.micro

(4) 検証環境における postgresql.conf の設定

基本的には 3.1.1 章の表 3.3 と 3.1.2 章の表 3.9 で紹介した postgresql.conf ファイルの設定を用いて検証しましたが、一部変更があります。

表 3.15: パラメータ変更箇所

パラメータ	設定値	パラメータの説明	変更理由
max_connections	105	PostgreSQL サーバに同時に接続するクライアントの最大数	検証 3.2.3 で pgbench のクライアント数を増やして測定するため
wal_keep_segments	0	pg_xlog 以下に保持する WAL ファイルの数	検証 3.2.4 でレプリケーションが途切れる事象を発生させやすくするため

(5) ネットワークの基礎性能

検証の前にネットワークの基礎性能を知るため、以下①～③の応答速度、バンド幅を調べました。(表 3.16)
表番号は以下のサーバ間を表しています。
①マスタ(東京) - 同期スタンバイ(東京)
②マスタ(東京) - 非同期スタンバイ(シンガポール)
③同期スタンバイ(東京) - 非同期スタンバイ(シンガポール)

表 3.16:ネットワークの応答速度とバンド幅

番号	応答速度(ms)	バンド幅(Gbits/sec)
①	1.33	1.11
②	76.95	0.16
③	76.25	0.16

【取得例:応答速度】

ping コマンドより、サーバ間の応答速度を取得しました。[4]

①マスタ(東京) - 同期スタンバイ(東京)

```
[postgres@master ~]$ ping 54.92.66.190 -c 10
PING 54.92.66.190 (54.92.66.190) 56(84) bytes of data.
64 bytes from 54.92.66.190: icmp_seq=1 ttl=63 time=1.28 ms
64 bytes from 54.92.66.190: icmp_seq=2 ttl=63 time=2.06 ms
64 bytes from 54.92.66.190: icmp_seq=3 ttl=63 time=1.43 ms
64 bytes from 54.92.66.190: icmp_seq=4 ttl=63 time=1.30 ms
64 bytes from 54.92.66.190: icmp_seq=5 ttl=63 time=1.44 ms
64 bytes from 54.92.66.190: icmp_seq=6 ttl=63 time=1.03 ms
64 bytes from 54.92.66.190: icmp_seq=7 ttl=63 time=1.29 ms
64 bytes from 54.92.66.190: icmp_seq=8 ttl=63 time=0.903 ms
64 bytes from 54.92.66.190: icmp_seq=9 ttl=63 time=1.23 ms
64 bytes from 54.92.66.190: icmp_seq=10 ttl=63 time=1.36 ms

--- 54.92.66.190 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9012ms
rtt min/avg/max/mdev = 0.903/1.336/2.064/0.292 ms
```

以上より平均応答速度は 1.33ms でした。

【取得例:バンド幅】

iperf コマンドよりサーバ間のバンド幅を取得しました。[5]

①マスタ(東京) - 同期スタンバイ(東京)

```
[postgres@slave ~]$ iperf -c 54.65.232.0
------------------------------------------------------------
Client connecting to 54.65.232.0, TCP port 5001
TCP window size:  325 KByte (default)
------------------------------------------------------------
[  3] local 172.30.1.13 port 58923 connected with 54.65.232.0 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.30 GBytes  1.11 Gbits/sec
```

以上よりバンド幅は 1.11Gbits/sec でした。

4ping コマンドは、相手のサーバにパケットを送り、それに対する返事が返ってくるかどうかを調べるコマンドです。

5iperf コマンドはサーバ間のネットワークのバンド幅やネットワーク転送性能を測定します。iperf は Linux の標準ツールではないため、ソースファイル一式をダウンロードする必要があります。(http://sourceforge.net/projects/iperf/)

3.2.3. レプリケーション方式による性能特性の違い

本検証ではレプリケーションの構成を変えて、トランザクションが大量に発行された際のマスタの負荷や性能を比較します。

3.2.3.1 検証手順

(1) 負荷をかける手段

マスタに大量のトランザクションを発行するため、PostgreSQL 標準のベンチマークツールである pgbench を使用しました。ベンチマークシナリオはデフォルトのまま使用しています。

```
1. BEGIN;
2. UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
3. SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
4. UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
5. UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
6. INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid,
:aid, :delta, CURRENT_TIMESTAMP);
7. END;
```

図 3.48:pgbench のデフォルトのシナリオで実行される SQL

(2) 実行方法

pgbench はスケールファクタは 50 に設定し以下のコマンドで実行しました。
pgbench 出力結果にある tps(Transactions Per Second)は一秒間に実行するトランザクション数を示します。

```
[postgres@master ~]$ pgbench -v -c 60 -t 100 -N
starting vacuum...end.
starting vacuum pgbench_accounts...end.
transaction type: TPC-B (sort of)
scaling factor: 50
query mode: simple
number of clients: 60
number of threads: 1
number of transactions per client: 100
number of transactions actually processed: 6000/6000
latency average: 0.000 ms
tps = 992.476039 (including connections establishing)
tps = 1007.419646 (excluding connections establishing)
```

図 3.49:pgbench 実行結果

(3) 使用したオプション

-c オプション
pgbench 実行時に接続するクライアント数を設定します。この数値を決定するため、クライアント数を 10~100 まで 10 ずつ増やしてそれぞれで pgbench を実行しました。これを合計 3 回繰り返し、得られた各クライアント数での TPS(tansactions per second)の平均値を比較しました。(表 3.17)

表 3.17 pgbench のクライアント数と TPS

クライアント数	マルチスタンバイ構成の TPS 中央値	カスケード構成の TPS 中央値
10	860	978
20	1002	1018
30	968	1062
40	1043	1033
50	1009	945
60	1065	1168
70	837	996
80	805	1039
90	572	620
100	504	570

以上よりクライアント数は両構成ともに、今回の検証環境で最も良い性能を引き出せる 60 に固定しました。

-N オプション
件数の少ないテーブルでのロック競合を防ぐため、そのようなテーブルの更新は行わないことにします。
このオプションをつけるとデフォルトのベンチマークシナリオの 4 と 5 の処理を省略します。

-v オプション
試験前に 4 つの標準テーブル全てを VACUUM します。pgbench では非常に大量の更新系処理が走るため、一度の pgbench で生成される不要領域も大量となります。この大量の不要領域を残したままにすると PostgreSQL の性能低下につながる危険があるため VACUUM をすることにしました。

-t オプション
各クライアントが実行するトランザクション数を設定します。この数値に”-c　クライアント数”を掛けた数のトランザクションが一度の pgbench で実行されます。今回の検証では 100 に設定していたため、一度の pgbench で実行されたトランザクション数は、100×60=6000 となります。

(4) 結果測定方法

上記した pgbench コマンドをレプリケーションの各構成ごとに 10 回実行し、PostgreSQL の接続時間も含めた tps(including connections establishing)を取得します。取得したデータの中央値の平均値と標準偏差を比較し、それぞれの構成時のマスタのスループットを調べます。

3.2.3.2.検証結果

表 3.18: pgbench による TPS 測定結果

測定回数	マルチスタンバイ構成時の TPS	カスケード構成時の TPS
1	1407	1022
2	916	1019
3	664	877
4	1283	714
5	709	878
6	798	950
7	748	838
8	1310	1330
9	994	1124
10	670	1064
中央値の平均値	929	972
中央値の標準偏差	194	72

以上より、中央値の平均値を見るとカスケード構成時の方がマルチスタンバイ構成時に比べてスループットが約4.63%大きくなりました。

3.2.3.3.考察

中央値の平均値を見ると両構成で多少の差はあるように見えますが、中央値の標準偏差の値は両構成で全く異なっています。標準偏差は平均値からの値のばらつき具合を示すものですが、今回のように比較するもの同士で標準偏差が異なる場合は双方のデータのばらつきが違うということですので、これらを一概に比較することはできなくなります。

また、測定全てにおいてカスケード構成時の方がマルチスタンバイ構成時に比べスループットが大きかったというわけではありません。したがってこの 4.63%という数字は決して有意なものではありません。

このことから、マスタから直結しているスタンバイの数が 1 つ程度しか変わらない場合では、PostgreSQL のスループットに大きな影響はないということが言えます。

3.2.4. レプリケーション遅延の実態

本検証では大量のレコードを更新するバッチ処理が走ったことを想定し、2.2.3 章のマルチスタンバイ構成のメインサイトのスタンバイと DR サイトのスタンバイがそれぞれどの範囲まで WAL を適用できるのかを時系列で測定します。

また 2.2.3 章ではメインサイトのスタンバイは同期スタンバイですが、DR サイトのスタンバイは非同期スタンバイです。同期スタンバイと非同期スタンバイの設定の違いによる WAL 適用の時間差が、レプリケーションの遅延にどの程度影響を与えているのかを知るため、両サイトのスタンバイとも非同期スタンバイの場合も同様に測定します。それぞれの検証に下記のように番号をつけました。

(1) 検証 3.2.4-1：同期スタンバイ×1 台、非同期スタンバイ×1 台の場合
(2) 検証 3.2.4-2：非同期スタンバイ×2 台の場合

3.2.4.1 検証手順(検証 3.2.4-1、検証 3.2.4-2 で共通)

(1) 負荷をかける手段

pgbench で作成される pgbench_accounts テーブル(本検証では 500 万レコードに設定)を全件一括 update します。一度の update では生成される WAL の量では、今回の検証の観点である大きな WAL 適用の時間差を引き起こすのに不十分だったため、update を複数回繰り返すために以下のようなスクリプトで実行しました。

```
HOST=54.65.232.0

i=0
while [ $i -le 4 ];
do  i=`expr $i + 1`
val1=$(psql -h $HOST -p 5432 -U postgres -q -t -c "select current_time;")
val2=$(psql -h $HOST -p 5432 -U postgres -q -t -c "update pgbench_accounts set bid = random();")
val3=$(psql -h $HOST -p 5432 -U postgres -q -t -c "select current_time;")

echo "${val1},${val3}"

done > /usr/local/pgsql/result/update.sh
```

(2) スタンバイの WAL 適用量の算出方法

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ました。

この差分を 20 秒間ごとに取得するため以下のようなスクリプトを実行しました。

```
HOST=54.65.232.0

i=0
while [ $i -le 199 ];
do
 i=`expr $i + 1`
 val1=$(psql -h $HOST -p 5432 -U postgres -q -t -c "select current_time;")
 val2=$(psql -h $HOST -p 5432 -U postgres -q -t -c "select
client_addr,pg_xlog_location_diff(master,replay_location)as replaydiff from (select
pg_current_xlog_insert_location() master)as m,pg_stat_replication;")
echo "${val1},${val2}"
 sleep 20
```

```
done > /usr/local/pgsql/result/lsndiff.test
```

3.2.4.2.検証結果

図 3.50 は検証 3.2.4 の結果を表しています。また事象発生時のマスタとスタンバイのサーバログ(図 3.51 図 3.52)と、レプリケーションの状態を確認するビュー(図 3.53)も取得しました。

(1)検証結果 3.2.4-1：同期スタンバイ×1 台、非同期スタンバイ×1 台の場合

横軸は時刻(hh:mm:ss 形式)を表しています。縦軸はスタンバイ側ではまだ適用できていない WAL 量(byte)、つまりレプリケーションの遅延量を表しています。
グラフ内に示されているサーバログ①～②、ビュー①～②は、図 3.52 と図 3.53 でそれぞれ該当時刻のものを提示しています。

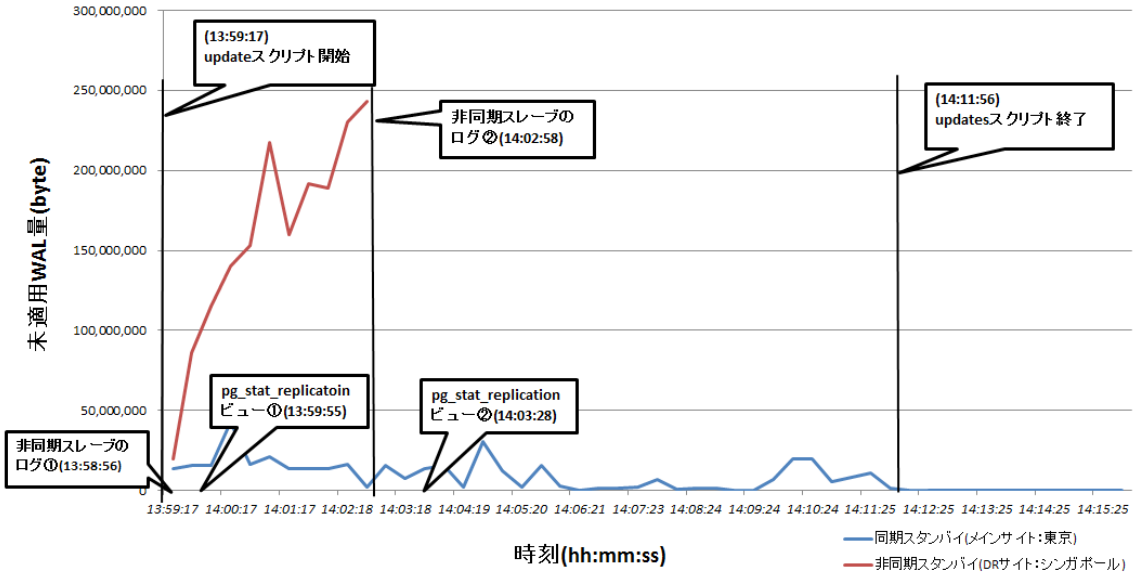


図 3.50: 検証 3.2.4-1 の各スタンバイの未適用 WAL 量

```
2015-02-28 14:02:50 JST [3709-41]LOG: checkpoints are occurring too frequently (8 seconds apart)
2015-02-28 14:02:50 JST [3709-42]HINT: Consider increasing the configuration parameter
"checkpoint_segments".
2015-02-28 14:02:57 JST [unknown][3725-3]ERROR: requested WAL segment
000000010000000070000004E has already been removed
2015-02-28 14:02:58 JST [unknown][4109-1]LOG: connection received: host=54.65.232.0 port=34333
2015-02-28 14:02:58 JST postgres[4109-2]LOG: connection authorized: user=postgres database=postgres
2015-02-28 14:02:58 JST [unknown][4111-1]LOG: connection received: host=54.65.232.0 port=34334
2015-02-28 14:02:58 JST postgres[4111-2]LOG: connection authorized: user=postgres database=postgres
```

図 3.51 検証 3.2.4-1 のマスタサーバのログ

ログ①レプリケーションの開始
2015-02-28 13:58:36 JST [2290-1]LOG: started streaming WAL from primary at 6/7D000000 on timeline 1
ログ②マスタで未適用のスタンバイが削除されたためストリーミングレプリケーション停止
2015-02-28 14:02:58 JST [2290-2]FATAL: could not receive data from WAL stream: ERROR: requested WAL segment 000000010000000070000004E has already been removed

図 3.52: 検証 3.2.4-1 の非同期スタンバイサーバのログ

ビュー①ストリーミングレプリケーションしているスタンバイは 2 台
postgres=# select current_time as timetz, client_addr, sync_state from pg_stat_replication;
-[RECORD 1]----------
timetz | 13:59:55.232788+09
client_addr | 54.251.130.204
sync_state | async
-[RECORD 2]----------
timetz | 13:59:55.232788+09
client_addr | 54.92.66.190
sync_state | sync

ビュー②ストリーミングレプリケーションしているスタンバイは 1 台(DR サイト非同期スタンバイはレプリケーション停止)
postgres=# select current_time as timetz, client_addr, sync_state from pg_stat_replication;
-[RECORD 1]----------
timetz | 14:03:28.110256+09
client_addr | 54.92.66.190
sync_state | sync

図 3.53 検証 3.2.4-1 の pg_stat_replication ビュー

(2)検証結果 3.2.4-2:非同期スタンバイ×2 台の場合

こちらも(1)検証結果 3.2.4-1 と同様に結果をグラフ化しました。(図 3.54)
マスタ、スタンバイそれぞれのサーバログ、レプリケーションの状態確認のビューは(2)検証 3.2.4-1 と同様なので省略しました。

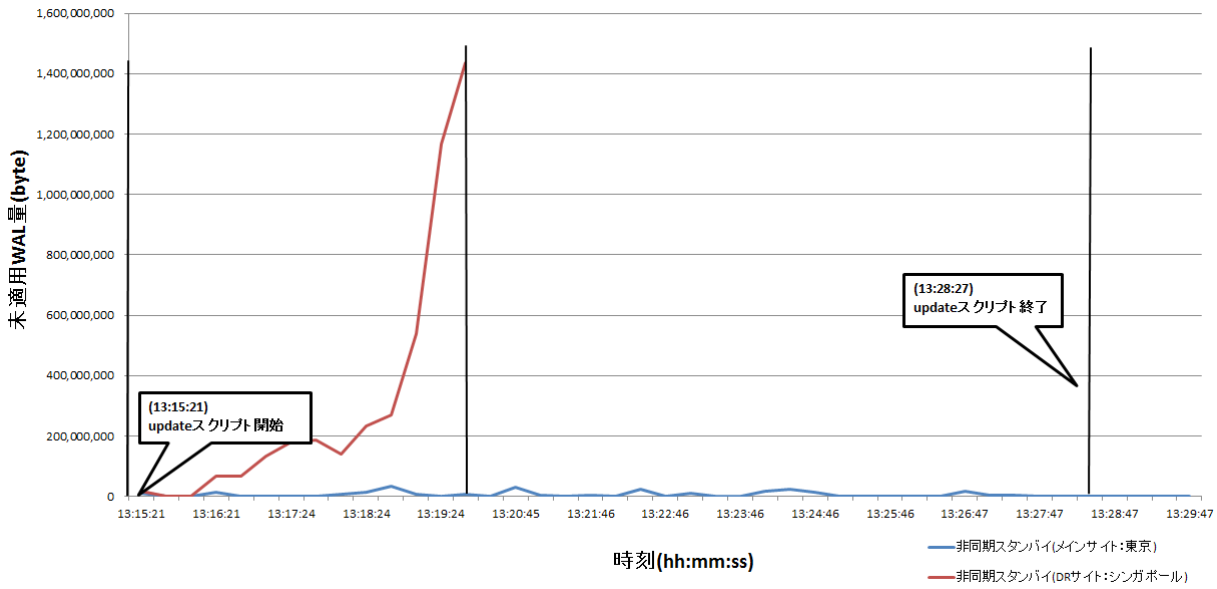


図 3.54: 検証 2.3.4-2 の各スタンバイの未適用 WAL 量

3.2.4.3.考察

(1)検証 3.2.4-1 と(2)検証 3.2.4-2 の二つの結果より、DR サイトにあるスタンバイの WAL 未適用量はストリーミングレプリケーションの経過時間とともに多くなっていくことが分かりました。また、しばらくするとスタンバイが未適用の WAL がマスタから削除されてしまったため、スタンバイはストリーミングレプリケーションを停止してしまいました。

このことを図示すると以下のようになります。(図 3.55)

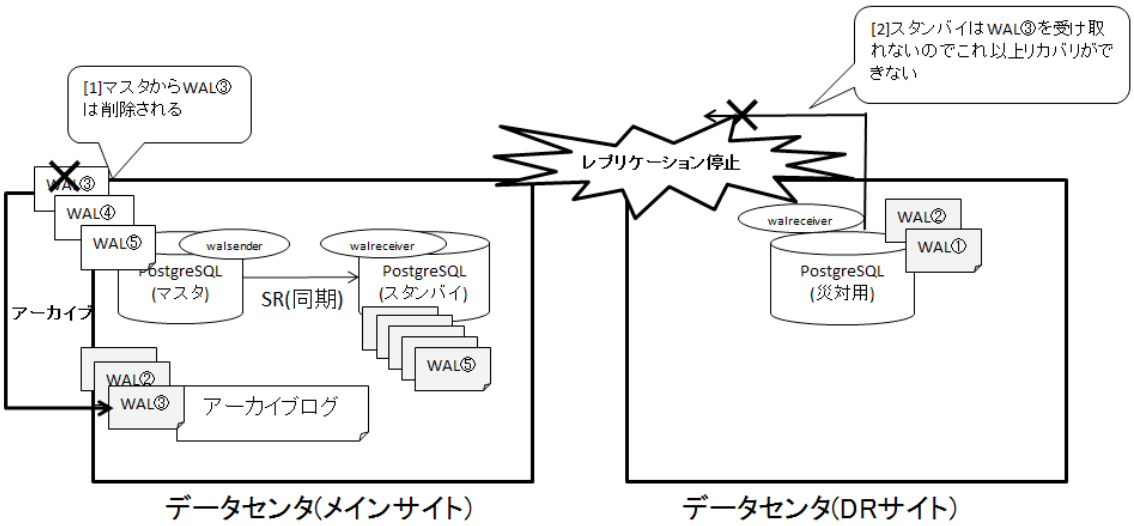


図 3.55: ストリーミングレプリケーションの停止

図 3.55 では[1]マスタが WAL③ をアーカイブして、WAL③ をデータベースクラスタ以下(デフォルト pg_xlog/)から削除したため、[2]スタンバイが WAL③ を取得することができません。整合性保持のために WAL は順次適用されなければなりませんので、要求する WAL が得られない場合スタンバイはストリーミングレプリケーションを停止してしまいます。

(1)検証 2.3.4-1 と(2)検証 2.3.4-2 とで、それぞれの DR サイトのスタンバイのストリーミングレプリケーションが途切れる直前の WAL 未適用量を比較すると、同期スタンバイがない(2)検証 2.3.4-2 では、同期スタンバイのある(1)検証 2.3.4-1 に比べ未適用 WAL 量が約 5.9 倍となりました。

しかし、いずれにせよどちらも DR サイトのスタンバイはストリーミングレプリケーションを停止させてしまいましたので、ストリーミングレプリケーションを継続させるには、スタンバイが同期か非同期かの設定よりも、サーバ間の距離、すなわちネットワークの転送量の方に注意する必要があります。

3.2.5. レプリケーション方式特有のリスクと対策方法

本検証では、検証 3.2.4 で途切れたストリーミングレプリケーションを復活させる手法を検証しました。

3.2.5.1.検証手順

(1) 実施方法

スタンバイの recovery.conf ファイルの restore_command パラメータに scp コマンドを設定し、「マスタの IP アドレス:マスタのアーカイブログディレクトリまでの絶対パス」を記述しました。なお、このように記述する場合はマスタとスタンバイが OpenSSH により postgres ユーザーで公開鍵認証接続が可能となっている状態にしておく必要があります。
上記のように restore_command パラメータを設定をすることで、スタンバイのストリーミングレプリケーションが途切れた際に、スタンバイはマスタから WAL ファイルを補い、必要とする WAL を適用をし続けるのか確認しました。

表 3.19: restore_command パラメータの設定

```
restore_command = 'scp 54.65.232.0:/usr/local/pgsql/arc/%f %p'
```

また restore_command パラメータの設定によりコピーされたマスタのアーカイブログから、スタンバイ側が未適用分 WAL を適用しきった後に、ストリーミングレプリケーションは自動で復旧するのかも確認しました。

3.2.5.2.検証結果

検証した結果をグラフにしました。(図 3.56)
グラフの軸は検証 3.2.4 と同様に、横軸は時刻、縦軸はスタンバイが未適用の WAL 量を表しています。

DR サイトのスタンバイの 14:03:01 時点のサーバログ(図 3.57)から、DR サイトのスタンバイはストリーミングレプリケーションが途切れている間、restore_command パラメータに設定したマスタのアーカイブログから WAL ファイルをコピーし、WAL を適用し続けることが分かりました。
また、同じく DR サイトのスタンバイの 14:29:10 時点のサーバログから、未適用の WAL が十分適用できた後は自動でストリーミングレプリケーションを再開することも確認できました。

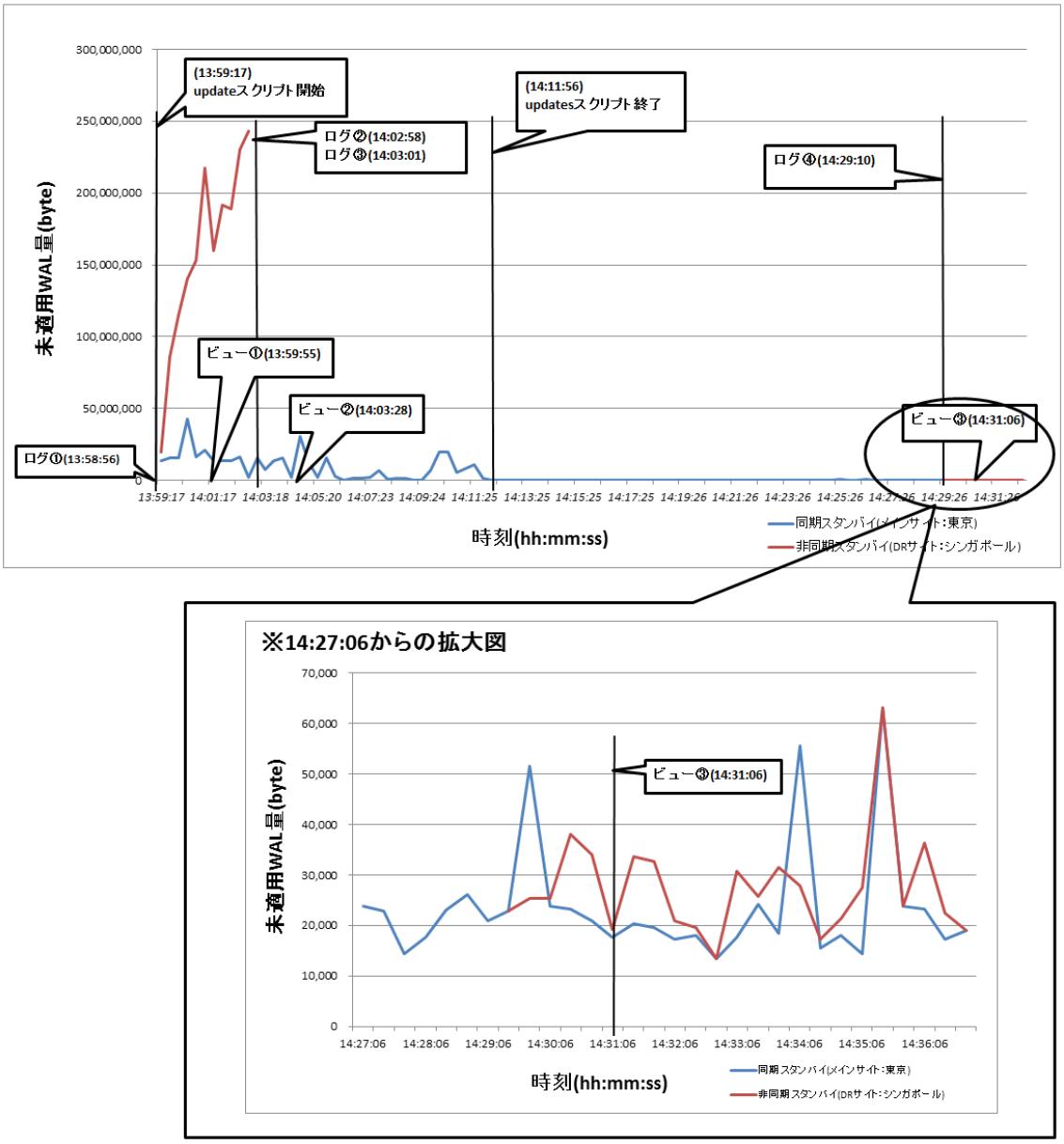


図 3.56: 検証 3.2.4-1 のストリーミングレプリケーション継続

```
2015-02-28 14:02:50 JST [3709-41]LOG: checkpoints are occurring too frequently (8 seconds apart)
2015-02-28 14:02:50 JST [3709-42]HINT: Consider increasing the configuration parameter
"checkpoint_segments".
2015-02-28 14:02:57 JST [unknown][3725-3]ERROR: requested WAL segment
000000010000000070000004E has already been removed
2015-02-28 14:02:58 JST [unknown][4109-1]LOG: connection received: host=54.65.232.0 port=34333
2015-02-28 14:02:58 JST postgres[4109-2]LOG: connection authorized: user=postgres database=postgres
2015-02-28 14:02:58 JST [unknown][4111-1]LOG: connection received: host=54.65.232.0 port=34334
2015-02-28 14:02:58 JST postgres[4111-2]LOG: connection authorized: user=postgres database=postgres

2015-02-28 14:29:06 JST [unknown][10671-1]LOG:  connection received: host=54.65.232.0 port=34501
2015-02-28 14:29:06 JST postgres[10671-2]LOG:  connection authorized: user=postgres database=postgres
2015-02-28 14:29:10 JST [unknown][10686-1]LOG:  connection received: host=54.251.130.204
port=36452
2015-02-28 14:29:10 JST [unknown][10686-2]LOG:  replication connection authorized: user=postgres
2015-02-28 14:29:26 JST [unknown][10689-1]LOG:  connection received: host=54.65.232.0 port=34502
2015-02-28 14:29:26 JST postgres[10689-2]LOG:  connection authorized: user=postgres database=postgres
```

図 3.57 検証 3.2.4-1 のマスタサーバのログ

```
ログ①ストリーミングレプリケーションの開始
2015-02-28 13:58:36 JST [2290-1]LOG: started streaming WAL from primary at 6/7D000000 on timeline 1

ログ②マスタで未適用のスタンバイが削除されたためストリーミングレプリケーション停止
2015-02-28 14:02:58 JST [2290-2]FATAL: could not receive data from WAL stream: ERROR: requested WAL
segment 000000010000000070000004E has already been removed

ログ③アーカイブログから WAL をリストアする
2015-02-28 14:03:01 JST [2277-8]LOG: restored log file "000000010000000070000004E" from archive
2015-02-28 14:03:04 JST [2277-9]LOG: restored log file "000000010000000070000004F" from archive
～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～
2015-02-28 14:16:55 JST [2277-219]LOG: restored log file "000000010000000800000021" from archive
2015-02-28 14:16:57 JST [2277-220]LOG: restored log file "000000010000000800000022" from archive
～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～～
2015-02-28 14:29:02 JST [2277-466]LOG: restored log file "000000010000000900000018" from archive
2015-02-28 14:29:05 JST [2277-467]LOG: restored log file "000000010000000900000019" from archive
2015-02-28 14:29:08 JST [2277-468]LOG: restored log file "00000001000000090000001A" from archive
scp: /usr/local/pgsql/arc/00000001000000090000001B: No such file or directory

ログ④ストリーミングレプリケーションの開始
2015-02-28 14:29:10 JST [2277-469]LOG: unexpected pageaddr 8/D5000000 in log segment
00000001000000090000001B, offset 0
 2015-02-28 14:29:10 JST [3258-1]LOG: started streaming WAL from primary at 9/1A000000 on timeline 1
```

図 3.58 検証 3.2.4-1 の DR サイトのスタンバイサーバのログ

また、統計情報からも、ストリーミングレプリケーションが復旧したことが読み取れます。(図 3.59)

ビュー①ストリーミングレプリケーションしているスタンバイは 2 台
```
postgres=# select current_time as timetz, client_addr, sync_state from pg_stat_replication;
-[ RECORD 1 ]----------------
timetz      | 13:59:55.232788+09
client_addr | 54.251.130.204
sync_state | async
-[ RECORD 2 ]----------------
timetz      | 13:59:55.232788+09
client_addr | 54.92.66.190
sync_state | sync
```

ビュー②ストリーミングレプリケーションしているスタンバイは 1 台(DR サイトのスタンバイはレプリケーション停止)
```
postgres=# select current_time as timetz, client_addr, sync_state from pg_stat_replication;
-[ RECORD 1 ]----------------
timetz      | 14:03:28.110256+09
client_addr | 54.92.66.190
sync_state | sync
```

ビュー③ストリーミングレプリケーションしているスタンバイは 2 台(DR サイトのレプリケーションが復旧)
```
postgres=# select current_time as timetz, client_addr, sync_state from pg_stat_replication;
-[ RECORD 1 ]----------------
timetz      | 14:31:06.69948+09
client_addr | 54.251.130.204
sync_state | async
-[ RECORD 2 ]----------------
timetz      | 14:31:06.69948+09
client_addr | 54.92.66.190
sync_state | sync
```

図 3.59 検証 3.2.4-1 の pg_stat_replication ビュー

3.2.5.3.考察

今回 recovery.conf ファイルの restore_command パラメータに「マスタの IP アドレス:マスタのアーカイブログディレクトリまでの絶対パス」と設定したことで、実際にスタンバイはマスタのアーカイブログから WAL ファイルをコピーし、WAL を適用することがわかりました。しかしこれの動作はスタンバイのストリーミングレプリケーションが途切れてから行われるので、ストリーミングレプリケーションの復旧までにはどうしてもタイムラグが生じてしまいます。現に本検証ではストリーミングレプリケーション停止時間は約 30 分間もありました。

もしスタンバイのストリーミングレプリケーションの停止時間を少しでも縮めたいのならば、スタンバイのストリーミグレプリケーションが途切れる前に、別の手段で定期的にマスタのアーカイブログをスタンバイに送信するといった方法が有効と考えられます。例えば scp コマンドや rsync コマンドを cron で定期的に実行するといった方法が考えられます。

また、どのような方法にせよマスタのアーカイブログをコピーしスタンバイ側で保持する場合は、スタンバイのディスク容量保持のため、不要になったアーカイブログを削除する設定が必要となります。この設定は recovery.conf ファイルの archive_cleanup_command パラメータに pg_archivecleanup コマンドを設定することで実現できるので運用時に設定しておくとよいでしょう。

3.2.6　まとめ

ストリーミングレプリケーション運用継続について

PostgreSQL の WAL ファイルはデフォルトで pg_xlog 以下に保持されます。またその保持数は postgresql.conf ファイルの wal_keep_segments パラメータの設定によって決まり、もし WAL ファイルが設定した保持数を超え生成されると、保持している WAL ファイルの中で一番古いものから削除していき、新たな WAL ファイルを保持するという仕組みになっています。

前述してきたように PostgreSQL のストリーミングレプリケーションは WAL の転送で実現されていますので、もし消去された WAL ファイルの中に、ストリーミングレプリケーションしているスタンバイがまだ適応していない WAL が含まれていた場合、スタンバイはそれ以上ストリーミングレプリケーションすることが出来なくなります。

この状態を再現するため、検証 3.2.4 ではマスタがより早く WAL ファイルを pg_xlog 以下から削除するために(スタンバイがマスタに追いつきにくくするため)、意図的に PostgreSQL ファイルの wal_keep_segments パラメータをデフォルトの 0 に設定しました。また実際に検証 3.2.4 はスタンバイのストリーミングレプリケーションが停止したことが確認できました。

以上より、反対にもしこの wal_keep_segments パラメータが有効になっており大きめの値が設定されていれば(例：16 や 32 等)、マスタが WAL ファイルを削除する周期を広めることができるので、ストリーミングレプリケーションの途切れ対策になるとも考えらえれます。

実際に、3.2.4 の検証の条件から wal_keep_segments パラメータだけを有効に変更し、同条件で負荷を掛け検証して、各スタンバイの WAL 適用量と、ストリーミングレプリケーションの継続を調べました。以下がその検証結果のグラフです。

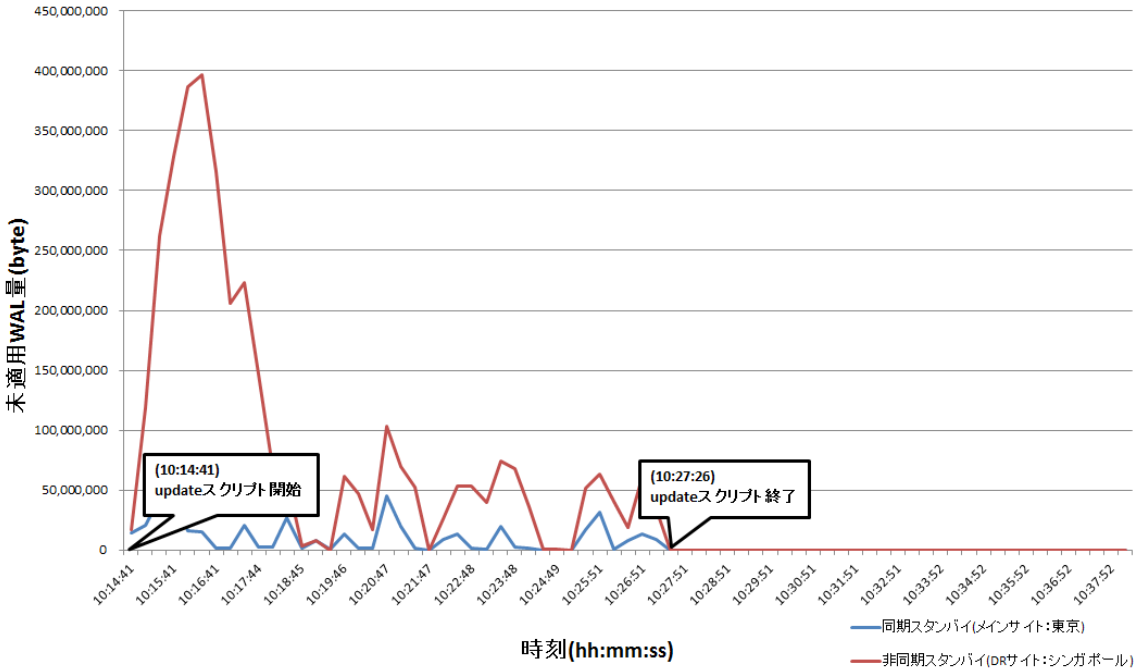


図 3.60:wal_keep_segments=32 の時の マルチスタンバイ構成時の未適用 WAL 量

以上より、DR サイトのスタンバイもストリーミングレプリケーションを継続し続けたので、ストリーミングレプリケーションを途切れさせないための対策として、wal_keep_segments パラメータを大きめに設定することは有効であることが分かりました。

ただし、wal_keep_segments パラメータを大きめに設定し WAL ファイルを保持することはその分マスタの DISK を使用するということなので DISK の容量には十分気をつける必要があります。

スタンバイの未適用 WAL 量取得時の注意点

今回の検証を通じて、スタンバイの未適用 WAL 量の確認の際に気を付けるべきポイントがあることが分かりました。

ストリーミングレプリケーションの情報を格納している統計情報の pg_stat_replication ビューの中に、スタンバイの WAL の位置(LSN)を示す以下の情報があります。

表 3.20: pg_stat_replication の LSN 項目

項目	説明
sent_location	接続上に送信された最後の WAL の位置
write_location	スタンバイによってディスクに書き出された最後の WAL の位置
flush_location	スタンバイによってディスクに吐き出された最後の WAL の位置
replay_location	スタンバイのデータベースに再生された最後の WAL の位置

この表 3.20 の sent_location が示す値とは、「マスタが各スタンバイに送った WAL の LSN」であり、マスタそのものの WAL の LSN を示すものではないことが分かりました。なぜならスタンバイがマスタに要求する WAL は、そのスタンバイの WAL 適用状態に依存するからです。したがってスタンバイの WAL 適用量を正確に測定するには、マスタの WAL の位置はスタンバイの状態に依存しない関数、例えば pg_current_xlog_insert_location()のような関数から取得しなければならないことが分かりました。(今回の検証では pg_current_xlog_insert_location()関数を使用しました。)
また、このような関数を使用する場合、スタンバイの未適用 WAL 量は次のように計算して取得しなければなりません。

未適用 WAL(LSN) ＝ 関数から取得した LSN － replay_location(LSN)…(1)

式(1)より算出した未適用 WAL は単位が LSN であり、これはデータ量を表す単位ではないため単位を byte に変換する必要があります。これには pg_xlog_location_diff 関数を使用する必要があります。
このように、スタンバイの未適用 WAL 量の情報を得るには、複数の関数を使用したり、単位に注意する必要があります。スタンバイの未適用 WAL 量の情報は、ストリーミングレプリケーションを運用する上でかなりの重要度で注視されるであろう情報と思われますので、pg_stat_replication ビューからスタンバイの未適用 WAL 量、もしくはマスタの LSN が手軽に取得できない点は PostgrerSQL が改善すべき課題一つではないかと思われます。

また、上記のような関数は WAL control functions という種類の関数のため、リカバリモードのサーバからは実行することができません。したがってカスケードレプリケーションのようにスタンバイが直接マスタから WAL を参照しない場合(スタンバイにとっての親もまたサーバ構成全体でみるとスタンバイの場合)は、親としているスタンバイで関数を実行することができなくなるため、単純に式(1)の方法で未適用 WAL 量を計算することはできなくなります。
本検証でもどうにか未適用 WAL 量を取得できないかと、「マスタの LSN 取得 SQL 文」、「末端のスタンバイの LSN 取得 SQL 文」、「それぞれの取得 LSN の計算 SQL 文」、と複数に SQL 文を分けての未適用 WAL 量の計算を試みましたが、関数で取得したデータは型がエスケープされないため、「取得 LSN の計算 SQL 文」の実行時に syntax error を起こし、未適用 WAL 量が計算できないという結果になりました。
このような結果からも、pg_stat_replication ビューの項目の一つとしてマスタの LSN が追加されたらと思います。

レプリケーションスロットについて

PostgreSQL9.4 から新機能としてレプリケーションスロットという機能が追加されました。これはマスタ側でスタンバイのストリーミングレプリケーション状態や付帯情報を管理する枠組みです。

レプリケーションスロットがあれば「このスタンバイはどの WAL ファイルを必要としているか」という情報が保持されるため、たとえストリーミングレプリケーションのみの構成であっても整合性に必要な WAL ファイルを消すことがなく、ストリーミングレプリケーションの継続や、スタンバイへの問い合わせのコンフリクトの防止を実現することができます。

また、このレプリケーションスロットは、マスタからスペックが異なる複数台のスタンバイがあるために最適な wal_keep_segments パラメータの数値を決めづらい場合や、wal_keep_segments パラメータの設定が不適切だった場合の備えとして、WAL ファイルを消去せずに保持しておく保険としても使用することができます。

しかしこの機能も裏を返せば、マスタの DISK 容量がひっ迫していないか監視する必要があると考えることができます。今回の検証ではこの機能は使用しませんでしたが、今後はストリーミングレプリケーションの運用時に使用してみようと思います。

4. おわりに

本報告書では、PostgreSQL で DR サイトを実現するシステム構成の調査およびストリーミングレプリケーションの機能を使った環境を使った動作検証結果について説明しました。

今回ストリーミングレプリケーションを使って実際に DR サイトを構築し検証を行ったところ、DR サイトは思ったより簡単に構築することができました。一方で実際に運用することを想定した場合、スレーブサーバのレプリケーション遅延の状況を把握することが困難だったり、遅延幅が大きくなった時にレプリケーションが停止する可能性があったり、あるいはカスケードレプリケーション構成で中間に位置するサーバが停止した場合の運用が複雑になったりと、いくつか考慮すべき点も見つかりました。

ストリーミングレプリケーションは PostgreSQL 9.0 より標準機能として取り込まれ、この機能を使って可用性の高いクラスタシステムを構築するケースも一般的になりつつありますが、DR サイトを構築するケースはまだ一般的ではなく、ノウハウが確立しているとは言えません。今回実運用に向けた留意点やノウハウが得られたことは、今年度の WG 活動の成果と言えるでしょう。また、DR サイトを実現する手法としてストリーミングレプリケーションを採用する事例が今後増えることで、より多くのノウハウが利用者間で共有され、運用手法が確立されていくことを期待したいところです。

昨年度に引き続き、今年度の活動でも PostgreSQL を安定運用する上で必要な可用性を確保する手法について調査を行えましたが、様々な実運用シーンを想定すると PostgreSQL を安定稼働に導くというテーマではまだまだ詳細な調査や技術検証を要するものがありますので、次年度以降も WG3 で調査や検証を続けていこうと考えています。

PostgreSQL がエンタープライズで利用されるようなきっかけとなるよう今後も活動を継続していきますので、何かやってみたいと思われた方は次の活動を一緒に実施していきましょう。

著者

版	所属企業・団体名	部署名	氏名
2014 年度 WG3 活動報告書 第 1 版	TIS 株式会社	コーポレート本部戦略技術センター	中西 剛紀
	TIS 株式会社	コーポレート本部戦略技術センター	下雅意 美紀
	日本電信電話株式会社	OSS センタ 基盤技術ユニット	黒岩 淳一
	株式会社日立ソリューションズ	技術開発本部 研究開発部	稲垣 毅
	株式会社日立ソリューションズ	サービスビジネス第2部	池田 尊敏