PGECons
PostgreSQL Enterprise Consortium

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

2013 年度 WG3 活動報告書

－走り続ける PostgreSQL システム構築のために－

改訂履歴

版	改訂日	変更内容
1.0	2014/4/17	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 2013 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）

2013 年度に新たに設立された WG3 では、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で必要とされる DBMS の非機能要求に着目し、可用性、バックアップ、監視の観点から PostgreSQL の典型的システム方式の調査および動作検証を行い、PostgreSQL で安定稼働を実現するための技術ノウハウを整理してきました。

■本資料の概要と目的

情報システムによる業務サービスを業務要件とコストのバランスを考えて、可能な限り長く運転し続けることは企業にとって重要な課題となります。本資料は 2013 年度の WG3 における活動成果を報告するもので、業務サービスの継続性をどのようにして高めるか、システムの運用や保守サービスをどこまで実現するか等の要求に対して、一般的に DBMS に求められる要件を PostgreSQL でどのように実現しているのか解説しています。また、より高いレベルの可用性要求を満たすために PostgreSQL で実現可能なクラスタ構成がそれぞれ持つ特徴も解説し、一部構成において実機検証を行った結果を紹介しておりますので、実システムへの PostgreSQL 適用を検討する上で参考にしていただけます。

■本資料の構成

1～2 章 企業システム、DBMS に求められる要件の整理

「PostgreSQL は企業システムで安心して使えるか」を評価するには、企業システムの DBMS に求められる要件が明確でなければなりません。そこで、本資料では最初に企業システムに求められる要件および DBMS に求められる要件について整理します。

1 章では独立行政法人情報処理推進機構（IPA）によって定義された「非機能要求グレード利用ガイド」から、企業システムの非機能要求として一般的に考慮されるべき観点について紹介します。また 2 章では、1 章で整理した要件をもとに可用性、バックアップ、監視という観点で、DBMS に必要な要件を定義します。

3～12 章 PostgreSQL で実現可能なシステム構成の紹介

1～2 章で整理した可用性、バックアップ、監視についての DBMS 要件に対して、PostgreSQL がどのような仕組み、機能を使って実現しているか、何が実現できないかを解説します。

企業システムで実際に PostgreSQL を利用する場合は、シングルサーバから複雑なクラスタ構成まで様々なシステム構成が考えられるため、3 章で PostgreSQL で一般的に利用されるシステム構成の全貌を紹介した後、4 章以降で各システム構成について詳細を解説していきます。

13 章 PostgreSQL を運用する上での技術検証

13 章では、3～12 章で解説した PostgreSQL の運用技術をより詳細に確認するための技術検証結果について報告します。ここでは以下のような技術検証を行っています。

- pgpool-II と PostgreSQL ストリーミングレプリケーションを組み合わせたクラスタでの可用性検証
- 論理バックアップ、リカバリ検証
- 物理バックアップ、Point In Time Recovery でのリカバリ検証
- PostgreSQL ストリーミングレプリケーション環境でのバックアップ、リカバリ検証
- 実際の障害パターンを想定した監視ケーススタディ

別紙（Appendix）
13 章で行った技術検証の環境構築手順、検証シナリオといった情報を掲載しています。

■想定読者

本書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

■謝辞

動作検証用の機器および環境を SRA OSS, Inc. 日本支社、株式会社日立製作所、富士通株式会社よりご提供いただきました。この場を借りて厚く御礼を申し上げます。

目次

1. 企業システムに求められる非機能要求

情報システムは「業務アプリケーション」と「システム基盤」の大きく二つの要素より構成されています。
「業務アプリケーション」はビジネス・業務そのものの機能を実現する仕組みであり、これらに対する要求事項が「機能要求」となります。一方、「システム基盤」は業務アプリケーションを実行するためのインフラであり、これらシステム基盤に対する要求事項が業務機能と区別して「非機能要求」と整理されています。
ここで「機能要求」は情報システムで実現したいビジネスそのものを具現化した機能に対する要求のことであり、そのビジネスの専門家（ユーザ）と IT 技術者（ベンダ）が協力して「業務アプリケーション」の設計に反映していくべき項目となります。
一方、「非機能要求」は情報システムのシステム基盤に対する要求ですが、IT の専門知識が豊富ではないビジネスの専門家（ユーザ）が適切に要求事項の整理を行うことは一般的には難しいと考えられます。また、開発対象の情報システムにおけるビジネスの知識や経験が浅い IT 技術者（ベンダ）にとっても、ユーザに最適な要求条件を適切なタイミングで提示することはきわめて困難であり、「システム基盤」構築にあたっては様々なリスクが生じることが実態です。
このビジネスの専門家（ユーザ）と IT 技術者（ベンダ）間で必要な「非機能要求」に対する共通認識を持つことがとても重要であり、事前に両者で合意しておかなくてはならない項目について、独立行政法人 情報処理推進機構（IPA）にて「非機能要求グレード利用ガイド」として整理が行われています[1]。
この「非機能要求グレード」には、「可用性」「性能・拡張性」「運用・保守性」「移行性」「セキュリティ」「システム環境・エコロジー」の大項目があります。ここで今回（WG3）の目的である、サービスを安定継続したいという要求に応えるためには、「可用性」に着目することが重要であると考えられます。

表 1: 非機能要求とは

大項目	概要	要求例
可用性	システムサービスを継続的に利用可能とするための要求	・運用スケジュール（稼働時間・停止予定など） ・障害、災害時における稼働目標
性能・拡張性	システムの性能および将来のシステム拡張に対する要求	・業務量および今後の増加見積もり ・システム化対象業務の特性（ピーク時、通常時、縮退時）
運用・保守性	システム運用と保守サービスに関する要求	・運用中に求められるシステム稼働レベル ・問題発生時の対応レベル
移行性	現行システム資産の移行に関する要求	・新システムへの移行期間および移行方法 ・移行対象資産の種類および移行量
セキュリティ	情報システムの安全性の確保に関する要求	・利用制限 ・不正アクセスの防止
システム環境・エコロジー	システムの設置環境やエコロジーに関する要求	・耐震/免震、重量/空間、温度/湿度、騒音などシステム環境に関する事項 ・CO2 排出量や消費エネルギーなどエコロジーに関する事項

サービスを安定継続するには？

可用性に着目

1 独立行政法人情報処理推進機構 http://www.ipa.go.jp/sec/softwareengineering/reports/20100416.html

また、後述の「1.2 運用保守性」でも説明がありますが、安定したサービス継続の仕組みを実現するには、システムそのものに対する要求である「可用性」に加えて、通常の運用に対しても配慮が必要な要素があります。
それが「バックアップ」と「運用監視」です。
「バックアップ」はシステム運用の要である「データ」をどのようにして保全するかの要求であり、「可用性」とは切っても切れない関係です。また、「運用監視」はシステムの平常時の健康状態をチェックし、それらの変化を知ることで故障症状を検知し、手遅れになる前に対策を採るための要求であり、これも「可用性」とセットで考慮するべき重要な関係です。

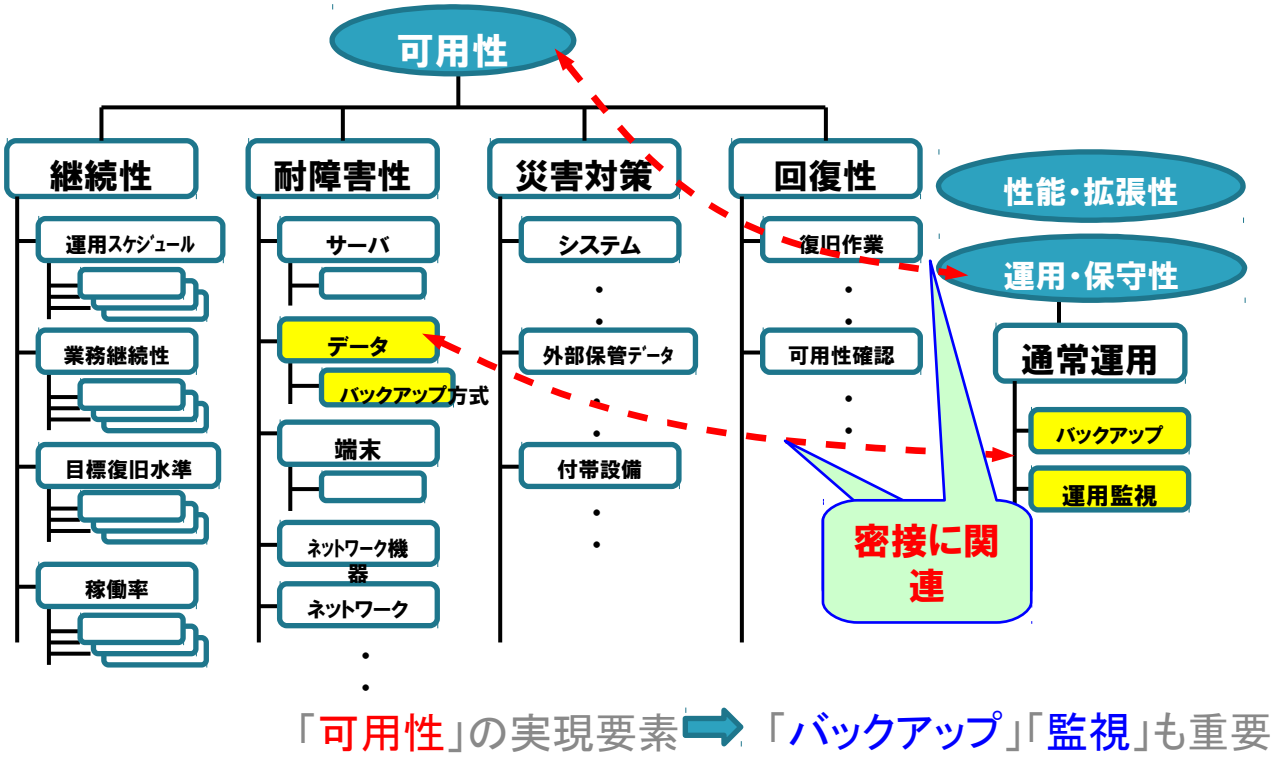


図1　非機能要求における可用性とバックアップ、運用監視の関係

1.1. 可用性

可用性（=アベイラビリティ availability）とは、システムに何かトラブルが発生しても何事もなかったかのようにサービスを継続できるようにするために必要な要求項目となっています。
業務サービス自体を止めないもしくはごく一部のサービス停止に留める、かつサービス品質の低下を極小にするためにどのような工夫や仕組みが必要かをビジネスの専門家（ユーザ）と IT 技術者（ベンダ）間で共通認識としておくことがきわめて重要なこととなります。
可用性に関しては「継続性」「耐障害性」「災害対策」「回復性」の 4 つの項目について、業務システム開発の初期段階、すなわち上流工程にてビジネスの専門家（ユーザ）と IT 技術者（ベンダ）間で議論し合意することが必要です。

1.1.1. 継続性

継続性は「可用性」の議論において最も重要な要求項目であり、システムが正常稼働している状態を表す「運用スケジュール」や「業務継続性：対象業務範囲」、故障発生時の復旧目標である「目標復旧水準」や「業務継続性：サービス切替時間」で整理されます。これらから「稼働率」を導き、業務をどれだけ継続させたいかについて、ユーザとベンダ間での議論と合意が必要とされています。

表 1.1.1: 継続性

要求項目の詳細	解説
運用スケジュール	システムの稼働時間（24 時間無停止、定時内のみ、通常時、特定日など）や停止運用（夜間停止可、朝 1 時間のみなど）、計画停止の有無に関する情報
業務継続性：対象業務範囲	可用性を保証するにあたり、要求される対象業務範囲（内部向け、外部向け、バッチ処理、オンライン処理など）やサービス切替時間（24 時間以内、2 時間、60 秒以内など）、多重故障時の条件（業務停止許容、単一故障は業務継続など）
目標復旧水準（業務停止時）	業務停止を伴う障害が発生した際に何をどこまで、どれ位で復旧させるかの目標定業務のみ、全業務など） RPO：目標復旧地点（5 営業日前のデータ、故障発生時点など） RTO：目標復旧時間（1 営業日以内、2 時間以内など） RLO：目標復旧レベル（特定業務のみ、全業務など）
目標復旧水準（大規模災害時）	大規模災害が発生した際、どれ位で復旧させるかの目標（1 週間以内に再開、1 日以内に再開など）
稼働率	運用スケジュールや目標復旧水準などで明示された利用条件の下で、システムが要求されたサービスを提供できる割合（99.9%≒年間停止時間 8 時間、99.999%≒年間停止時間 5 分など）

1.1.2. 耐障害性

耐障害性は故障への耐性に関してシステムの構成要素の観点から整理した要求事項です。「サーバ」や「端末」、「ネットワーク機器」などシステムを構成する要素で整理されます。
サーバの冗長化は効果的な耐障害性向上の方法となりますが、継続性の観点も加味して過剰な構成とならないように、要求と対策のバランスが取れたシステム構成の合意が必要とされています。
また、冗長化にはサーバ等機器レベルのものとディスク等のコンポーネントレベルのものがあり、単一サーバ構成の場合はコンポーネントレベルの対応が特に重要となります。
さらにハードウェア故障発生時の復旧で時間がかかる場合に確実に業務を再開するためには、万一の際にデータの消失を回避するための「バックアップ」等、冗長化以外の対応も忘れてはなりません。

表 1.1.2: 耐障害性

要求項目の詳細	解説
サーバ	サーバで発生する故障に対する機器の冗長化（特定サーバのみ、 全サーバなど）やコンポーネントに対する冗長化（特定コンポーネント、すべて、など）の要求
端末	端末で発生する故障に対する冗長化（機器やコンポーネント）の要求
ネットワーク機器	ルータやスイッチなどのネットワーク構成機器に対する冗長化の要求
ネットワーク	ネットワークの信頼性を向上させる回線の冗長化、経路の冗長化、さらにはオンライン・バッチ処理などの業務用途や監視・バックアップなどの管理系用途といったセグメント分割の要求
ストレージ	ディスクアレイなどの外部記憶装置で発生する故障に対して装置自体,ディスクの冗長化の要求
データ	データ保護に対する考え方 バックアップ方式：オフラインバックアップ、オンラインバックアップなど データ復旧範囲：一部の重要データのみ、システム内全データなど データ整合性（ データインテグリティ ）：エラーチェックのみ実施、訂正実施など

1.1.3. 災害対策

災害対策は情報システムを設置した施設や地域全体が使用不能になるような大規模災害に対する要求です。
このような自然災害や停電、火災等の場合にはサーバ等の冗長化対応だけでは不十分となります。
この要求に対してはサーバやストレージ等の構成自体にも大きな影響があり、必要性の認識は高いものの実際の対応としては後手に回っている場合が多いことが現状ではないでしょうか。しかし、甚大な災害に対して業務サービスを止めないためには異なるロケーションでの代替システム構築、データ保全を含めた全体設計が必要となります。
開発初期段階の要求条件では配慮が不要な場合でも、将来的な対応を想定して装置のリプレース等の二重投資が生じないような設備の選定が初期段階において重要です。

表 1.1.3: 災害対策

要求項目の詳細	解説
システム	地震、水害、火災、テロなどの大規模災害時の代替機器としてどこに何を配置するかの要求（同一構成を Disaster Recovery（以下 DR）サイトにて構築、限定された構成を DR サイトで構築など）
外部保管データ	地震、水害、火災、テロなどの大規模災害発生により被災した場合に備えてデータ、プログラムを別ロケーションに保管するかの要求（分散保管有無、DR サイトへのリモートバックアップなど）
付帯設備	各種災害に対するシステムの付帯設備（電源、空調等）での要求

1.1.4. 回復性

回復性は故障や災害発生後のシステム機能回復と必要なデータ復旧に対する要求です。
バックアップからのリカバリ作業やシステム停止中の代替機能の実現といった復旧作業と可用性に関する要求事項が実現できているかどうかを確認することが重要となります。

表 1.1.4: 回復性

要求項目の詳細	解説
復旧作業	業務停止を伴う故障が発生した際の復旧作業に必要な労力（バックアップ・リカバリツール活用、自動化有無や代替業務でカバーが可能かなど）に対する要求
可用性確認	可用性として要求された項目をどこまで確認するか（業務停止となる故障の一部のみ確認、すべて確認など）の要求

1.2. 運用保守性

運用保守性とは、必要なバックアップをいつどのように取得するかといったシステムの運用方法や保守サービスに関する要求項目です。導入する機器の構成やミドルウェアの選定、採用する方式決定に対して重大な関係があるため、注意が必要となります。

しかし、業務機能の開発には直接関係がないと誤解されて軽視される傾向があり、後々の下流工程にてトラブルが顕在化する場合があります。

運用保守性に関しては「通常運用」、「保守運用」、「障害時運用」の 3 つの項目について、上流工程にてビジネスの専門家（ユーザ）と IT 技術者（ベンダ）間で議論し合意することが必要です。

1.2.1. 通常運用

通常時の運用パターンに対する要求項目であり、運用時間や時刻同期といった項目に加え、「バックアップ」や「運用監視」といった可用性の実現に重要な項目が整理されています。

バックアップは、可用性のなかの耐障害性の項目でもあり、データ保全という実現すべき重要な要素となります。

運用監視は、可用性の実現のために必要な故障検知を実現する重要な要素であり、明白なエラーや無応答などの検出に加え、必要な監視項目における通常運用（健康）状態の把握や、それらの変化から故障を判断するロジック等についても十分な検討が必要となります。

このように「可用性」の実現のためには「バックアップ」や「運用監視」の項目についても専門家（ユーザ）と IT 技術者（ベンダ）間で十分な議論と合意が必要となります。

表 1.2.1: 通常運用

要求項目の詳細	解説
運用時間	利用者やシステム管理者に対してサービスを提供するためのシステム運用を行う時間（定時内や 24 時間無停止、通常時や特定日など）
バックアップ	システムが利用するデータのバックアップに関するデータ復旧範囲（一部データのみ、全データなど）や外部データの利用可否（一部データ活用可能、利用できないなど）、バックアップ利用範囲（故障発生時のデータ損失防止、ユーザエラー復旧、長期保存など）、バックアップ自動化の範囲（手動、一部手動、すべて自動化など）、バックアップ取得間隔（月次、週次、日次、同期バックアップなど）、バックアップ保存期間（1 年未満、3 年など）、バックアップ方式（オフライン、オンラインなど）の要求
運用監視	システム全体、それを構成するハードウェア、ソフトウェアに対する監視について、監視情報（死活、エラー、リソース、パフォーマンスなど）、監視間隔（定期監視、リアルタイム監視など）、システムレベル監視（一部分、全体など）、プロセスレベル（一部分、全体など）、データベースレベル（一部分、全体など）、ストレージレベル（一部分、全体など）、サーバ（ノード）レベル（一部分、全体など）、端末／ネットワーク機器レベル（一部分、全体など）、ネットワークパケットレベル（一部分、全体など）の要求
時刻同期	システムを構成する機器の時刻同期について、時刻同期設定の範囲（サーバ機器のみ、サーバおよびクライアント機器、ネットワーク機器を含めた全体など）の要求

1.2.2. 保守運用

システムの品質維持に必要なメンテナンスに対する要求項目であり、計画停止、パッチ適用、活性保守等の項目が整理されています。

表 1.2.2: 保守運用

要求項目の詳細	解説
計画停止	点検作業やメンテナンス等のシステム保守作業を目的としたサービス停止について、計画停止の有無やスケジュール変更の可否、計画停止の事前アナウンス（年間計画による、一週間前に通知など）の要求

要求項目の詳細	解説
運用負荷削減	運用保守に関する作業負荷を削減するための設計について、保守作業自動化の範囲（一部作業を自動化、すべての作業を自動化など）、サーバソフトウェア更新作業の自動化（配布機能の実装、自動配布・手動配布、自動更新・手動更新など）、端末ソフトウェア更新作業の自動化（配布機能の実装、自動配布・手動配布、自動更新・手動更新など）の要求
パッチ適用ポリシー	パッチ情報の展開とパッチ適用のポリシーについて、パッチリリース情報の提供（ユーザ要求に応じてベンダがリリース情報を提供、ベンダが定期的にリリース、ベンダがリアルタイムにリリースなど）、パッチ適用方針（推奨パッチのみ適用、すべて適用など）、パッチ適用タイミング（故障発生時、定期保守時、新規パッチリリース毎に、など）、パッチ検証の実施有無（故障パッチのみ検証、故障パッチとセキュリティパッチで検証、など）の要求
活性保守	サービス停止の必要がない活性保守が可能なコンポーネントの範囲について、ハードウェア活性保守の範囲（一部ハードウェアのみ、すべてのハードウェア、など）、ソフトウェア活性保守の範囲（一部ソフトウェアのみ、すべてのソフトウェア、など）の要求
定期保守頻度	システム保全のために必要なハードウェアまたはソフトウェアの定期保守作業について、定期保守頻度（年一回、月一回など）の要求
予防保守レベル	システム構成部材の故障前に予兆を検出し事前交換などの対応について、予防保守レベル（定期保守時の予兆で対応、定期保守とは別に一定期間で予兆検出、リアルタイムに実施など）の要求

1.2.3. 障害時運用

システム障害が発生した際に必要な対応に対する要求項目であり、復旧作業、障害復旧自動化の範囲、システム異常検知時の対応、交換用部材の確保、等の項目が整理されています。

表 1.2.3: 障害時運用

要求項目の詳細	解説
復旧作業	業務停止を伴う故障が発生した際の復旧作業について、復旧作業（手作業、復旧用製品利用、復旧用製品と業務アプリケーション利用など）、代替業務運用の範囲（一部業務について代替運用、すべての業務で代替運用など）の要求
障害復旧自動化の範囲	故障復旧に関するオペレーションを自動化する範囲について、一部の故障復旧作業を自動化、すべてを自動化などの要求
システム異常検知時の対応	システム異常を検知した際のベンダ側対応について、対応可能時期（ベンダの業務時間内、ユーザ指定の時間帯、24 時間対応など）、保守員駆けつけ到着時間（異常検知の翌営業日、数時間以内、保守員常駐など）、SE 到着平均時間（異常検知の翌営業日、数時間以内、SE 常駐など）の要求
交換用部材の確保	故障発生コンポーネントについて、保守部品確保レベル（部品提供ベンダが規定年数部品を確保、保守提供ベンダが当該システム専用に規定年数部品を確保など）、予備機の有無（一部予備機あり、すべて予備機保有など）の要求

1.3. セキュリティ

構築する情報システムの安全性の確保に対する要求であり、法令や情報セキュリティに関する基準、ガイドライン、企業ポリシー等の組織規定である「前提条件・制約条件」や、潜在的脅威の洗い出しや対策範囲の明確化を行う「開発・運用時のセキュリティ管理」、「セキュリティ対策を実現する機能」、これら機能の組み合わせによる「セキュリティ対策パターン」と大きく４つの項目で整理されています。

ここでは DBMS 運用の観点から、「セキュリティ対策を実現する機能」に対する要求項目に着目し、アクセス・利用制限、データの秘匿、不正追跡・監視について整理します。

1.3.1. アクセス・利用制限

開発する情報システムで取り扱う資産（リソース）に対するアクセスおよび利用の制限についての要求項目であり、サーバ、ストレージなど各々に対し検討することが必要です。

表 1.3.1: アクセス・利用制限

要求項目の詳細	解説
認証機能	情報資産を利用する主体（ユーザ、機器）を識別するための認証の実施（ID/パスワード、IC カードなど）、どの程度 実施するかの要求
利用制限	認証された主体（ユーザや機器など）に対して、資産の利用などをソフトウェアやハードウェアで制限するかどうか（USB など I/O デバイスの制限、コマンド実行制限など）の要求
管理方法	認証に必要な情報の追加、更新、削除等のルール策定に関する要求

1.3.2. データの秘匿

開発システム上で流通および蓄積する情報の秘匿についての要求項目となり、暗号化の対象となる情報資産や暗号化の実施箇所について検討、さらに暗号化を行う場合の性能への影響も考慮する必要があります。

表 1.3.2: データの秘匿

要求項目の詳細	解説
データの暗号化	機密性のあるデータを、伝送時や蓄積時に秘匿するための 暗号化（認証情報のみ暗号化、重要情報も暗号化など）を 実施するかの要求

1.3.3. 不正追跡、監視

システム運用後に発生する不正行為の追跡や監視についての要求項目となり、セキュリティログ取得等による性能への影響も合わせて考慮する必要があります。

表 1.3.3: 不正追跡、監視

要求項目の詳細	解説
不正監視	不正行為を検知するために、それらの不正について監視する 範囲や、監視の記録を保存する量や期間を確認するための項目（ログ取得、ログ保管期間、不正監視対象（装置、ネットワーク、侵入者・不正操作者、確認間隔など））
データ検証	情報が正しく処理されて保存されていることを証明可能とし、 情報の改ざんを検知するための仕組みとしてデジタル署名を導入するかの要求

2. DBMS に求められる要件

DBMS に関する故障の原因としては、これまで整理してきたようにストレージ・ディスク故障やサーバ故障、より規模の大きいサイト全体やエリアでのトラブルが考えられます。さらに情報システムの中核的存在である DBMS に特徴的なトラブルとして、データの消失や破壊、論理矛盾についても故障原因のひとつとなるため、DBMS の観点から要件を整理していくことが重要となります。

本章では 1 章で説明した企業システムの非機能要件を元に、DBMS に求められる非機能要求を整理していきます。

2.1. 可用性

そもそもデータベース構築・運用は単一ノード（1つのサーバに1つのデータベース）が基本です。サーバやディスクなどハードウェアが故障した場合には、修理や交換で復旧することが可能です。DBMS などのプログラムの不具合については、プログラム自体の更新頻度は高くないため、再インストールや定期的なバックアップ・リカバリでも対応することができます。

しかし、業務サービスの運用に従って時々刻々と更新されていく DB データそのものについては、最新もしくは任意の時点の状態に DB データを戻すことはそう簡単にはできません。通常は DBMS がクラッシュリカバリやバックアップ・リカバリの仕組みを持っているので、適切な手順に従って正しく復旧することが重要です。

基本的な単一ノードの環境における環境設定や運用手順、運用スキルに問題がある場合は、可用性を高めるための冗長化などの高度な仕組みを構築して切替を行ったとしても、切替前ノードの問題点がそのまま切替後のノードに引き継がれるため、再び問題を起こすリスクが高まります。

DBMS、特に PostgreSQL の非機能要求の継続性については、以下の点が重要となります。

表 2.1.1: 継続性

非機能要求		解説（要件を実現する方法）
継続性	運用作業・メンテナンスのために業務サービスを停止させたくない	・オンラインバックアップ ・データの更新・削除による不要データの増加を押さえる VACUUM ・テーブルデータ・インデックスのオンライン再編成
	参考：基本的に計画停止が必要な場合	・ DBMS（PostgreSQL）自体へのパッチ適用、バージョンアップは計画停止が必要
	故障が発生しても短時間で業務を再開させたい	・DBMS 構成ファイルの冗長化 ・DB や WAL 格納ディスクの冗長化 ・複数 DB サーバの冗長化（共有ディスク、ストリーミングレプリケーション、pgpool-II のクエリベースレプリケーション等）
	故障が発生した直前の状態まで戻したい	・複数世代のバックアップ、差分・増分バックアップ ・任意の時間指定回復（Point In Time Recovery）が可能なこと

次に、耐障害性の観点からは、以下の整理となります。

表 2.1.2: 耐障害性

非機能要求		解説（要件を実現する方法）
耐障害性	故障が発生しても業務サービスを停止させたくない	・特定のサーバで冗長化、全サーバで冗長化 ・特定コンポーネントを冗長化、全コンポーネントを冗長化
	任意の範囲のデータを復元したい 任意の時点のデータを復元したい	・任意の DB、テーブルのみをバックアップ、リカバリ可能なこと ・様々なバックアップ方式を整理 論理バックアップ 物理オフラインバックアップ 物理オンラインバックアップ 非同期レプリケーション 同期レプリケーション ・再開後のサービス縮退（リカバリレベル）と復旧ポイント（リカバリポイント）を意識してデータ復旧範囲を整理 一部テーブルのみ復旧＋低 RPO（日次レベル） 一部テーブルのみ復旧＋高 RPO（災害直前まで） 全テーブルを復旧＋低 RPO（日次レベル） 全テーブルを復旧＋高 RPO（災害直前まで） 他系データベースを含む関連データすべてを復旧 ・OS が検知しないデータエラー、複数サーバ間でトランザクション一貫性が保たれるかどうか（PostgreSQL 単体での ACID 保証が前提）を意識してデータインテグリティを整理 データファイルのチェックサムエラー検出 複数 DB 間でのデータ競合が起きる(人手で回復) 複数サーバ間のデータ同期は、遅延して一貫性担保 複数サーバ間のデータ同期は、トランザクション完了後直ちに一貫性担保

災害対策の観点からは以下のとおりになります。

表 2.1.3: 災害対策

非機能要求		解説（要件を実現する方法）
災害対策	災害時に別の場所で同一のシステムを再稼働させたい	・限定された構成でシステムを再構築 ・同一の構成でシステムを再構築 ・限定された構成を DR サイトで構築 ・同一の構成を DR サイトで構築

以降で「バックアップ」、「監視」の観点から改めて非機能要求を整理・説明していきます。

2.2. バックアップ

DBMS に求められるバックアップの要件とその実現方法について解説していきますが、その前にバックアップがなぜ必要かについて改めて考え、また、DBMS へのバックアップの要件を整理します。

2.2.1. バックアップ・リカバリの必要性について

「データベースシステムのバックアップ・リカバリ」と一言で表しても、システムの概要やプロジェクトの状況に応じていろいろ考え方が存在します。ここではユーザの声の例を挙げてみます。

- 「バックアップは絶対に必要不可欠なもの。リカバリ時の復旧時間を小さくするためやどの時点にも戻せるように日次＋差分で取得している」
- 「バックアップを取りたいのはやまやまだが、24 時間 365 日で高負荷がかかるシステムであり、性能面等の影響を考えてなかなか手がついていない」
- 「今回のシステムでは重要なデータは特にないので、バックアップを取らなくても問題ないと考えている」
- 「データベースのレプリケーションを取得し HA クラスタ構成をとっているので、改めてバックアップは取得していない」
- 「障害からの復旧の為にバックアップが必要と思うが、RAID 構成によるディスクの冗長化はもちろん、システムレベルでも冗長構成をとっているのでバックアップの必要はない」

このように、システムの目的や構築期間、データの重要性、プロジェクトにかけられる費用や人的リソースの問題なども影響してか、全てのシステムで必ずしもバックアップを取得しているわけではありません。
ここで、バックアップ・リカバリはなぜ必要なのか、その理由を挙げていきます。バックアップ・リカバリはハードウェア障害からの復旧を行う際に必要ですが、それだけではないことに注意してください。

1. コンピュータのハードウェア障害
   ストレージ機器の故障や、停電によるシステムの電源断でファイルシステムが不正な状態になることなど、何らかの原因でファイルやデータが読みこめない状態になることを想定しています。データの冗長構成を取っていれば、スタンバイ側を利用すれば復旧可能です。冗長構成をとっていない場合はリカバリが必要になります。

2. ソフトウェアのバグやオペレーションミス
   アプリケーションやミドルウェア、データベース管理システム、OS やファイルシステム等のバグや設計ミスにより、必要なデータを削除・更新が行われた場合を想定しています。加えて、システムを操作する人のオペレーションミスで必要なデータを削除・更新してしまった場合もこのケースに含まれます。
   この場合、ハードウェアの立場からは正しくデータを操作しているだけになります。よって、冗長化構成をとっていても、同様にデータが更新されていますので、リカバリが必要になります。

3. 悪意のある利用者からの攻撃（クラッキング）
   悪意のあるクラッカーにシステムに侵入され、重要なデータを破壊された場合を想定しています。意図的にデータが壊されていますので、データを復旧する為にはリカバリが必要不可欠です。

4. ディザスタリカバリ
   システムを設置している場所（データセンター等）に火災や地震、津波などが発生し、再利用が不能な状態になっている場合を想定しています。当然ですが、同じ設置場所に冗長化しても設置場所のシステムが丸ごと利用できなくなってしまいますので、遠隔地にバックアップデータを退避させ、リカバリができるような状態にしておくことが必要です。

5. 証拠保全・再利用
   大量のデータをデータベースに格納する場合、一般的な方法として一定期間経過したデータはデータベースから削除するという運用が行われます。これらの削除するデータを何かあった場合の為の証拠保全として確保する場合や、分析用途に使うためにはリカバリが必要となります。

以上のとおり、ハードウェアなどの冗長性を確保したシステムであってもバックアップの取得は必要になるケースは存在します。よほど一時的なデータで重要度の低いものでない限り、バックアップ・リカバリはシステムを構築する際に必要な設計項目の一つと考えたほうがよいでしょう。また、リカバリについても、是非リカバリの時間や手順などの確認をバックアップの設計と共に実施することをお勧めいたします。

そもそもバックアップから戻さなければいけない場合は心理的にも緊迫した状況である可能性が高くなります。そのためリカバリについての計画を立てていないと、実際にリカバリができないケースや、予定より時間がかかってしまい、必要な時間までに復旧が行われなかったというケースもあります。

2.2.2. バックアップの要件について

それでは、システムの非機能要求に対応したバックアップの要件を見ていきます。

1章で触れた企業システムに求められる非機能要求からDBMSのバックアップ・リカバリに関連する要件を抜き出し、それぞれを確認していきます。

- （可用性） 運用作業のためにシステムを止めたくない
  ここでは、システム稼働中にオンラインでバックアップを取得出来るのかどうかが要件になります。ただし、オンラインバックアップの場合、バックアップ取得中のシステムに与える影響を考慮することが必要となり、DBMSへの負荷を最小限にとどめたいという要件が加えられます。これらの負荷の影響を考えて、システムの停止時間を確保できる場合はオフラインでのバックアップ取得を行ったほうが効率的な場合もあります。

- （可用性） 障害が発生しても、短い時間で業務再開させたい
  データ破壊を伴う障害が発生した際の目標復旧時間（RTO)を出来るだけ小さくするため、バックアップからのリカバリを短い時間で行うというのが要件に該当します。データベースのバックアップをどの手法（例：論理バックアップ、物理バックアップ）で取得し、どのメディアに取得してどこに確保しておくのかなどで復旧時間の大小に影響が出てきます。

- （可用性） 災害時に別の場所で同一のシステムを再稼働させたい
  バックアップ・リカバリの観点からは遠隔地にデータベースのバックアップデータを取得したいというのが要件になります。バックアップをどの間隔で取得し、遠隔地に確保[2]するのか、また復旧時に再稼働するシステムをどのように用意[3]してリカバリするかなどが検討項目になります。

- （可用性） 障害が発生した際の復旧作業の労力を小さくしたい
  障害発生時にリカバリ作業を省力化したいというのが要件になります。複雑な手順では、復旧時間にも影響を与えますので、バックアップデータを移動・選定する作業等が煩雑にならないよう設計し、例えば出来るだけスクリプト化することなどが必要になるでしょう。

- （運用保守性） 任意の時点（障害発生直前を含む）のデータに復元したい
  問題が発生した場合などで、指定の時刻にデータベースの状態にリカバリを行うというのが要件です。目標復旧地点（RPO)を出来るだけ小さくするためには、出来るだけ問題発生時に近い時点にデータを復旧することが求められます。一般的にデータベースを任意の時点に復旧する為には差分バックアップや、WALによるPoint In Time Recoveryを行うことで指定の時間にデータを復元することが可能になります。また、併せてバックアップ世代管理については別途考慮が必要です。

- （運用保守性） 任意の範囲のデータを復元したい
  データベースインスタンスで保有しているデータすべてではなく、任意のテーブルやデータベースの単位でリカバリが実行できることが要件です。目標復旧レベル（RLO)が特定の業務に限定されているケースでは、特定のデータだけを復元することで、目標復旧時間（RTO)を短くすることにもつながります。

2 例：バックアップを取得しネットワーク経由で送付する、データベース等のレプリケーションで同期しておくなど
3 例：コールド/ホットスタンバイで用意する、事後サーバを用意する、クラウド上のシステムを利用するなど

- （運用保守性） バックアップ運用の労力を小さくしたい
  ここでは、データベースのバックアップ取得を自動実行できることというのが要件になります。バックアップコマンドの自動実行だけでなく、バックアップの世代管理やメディアの操作なども含めて考慮が必要となります。

- リカバリを確実に行いたい
  企業システムの非機能要求項目としては挙がっていませんが、取得したバックアップデータを確実にリカバリしたいといった要件もあります。当然のことですが、取得したバックアップデータが壊れている状態では、バックアップの意味がありません。しかし、例えばシステムの障害が発生した時刻が判断できず、いつまでのデータが信頼できるのか不明瞭な場合もあり、取得したバックアップデータへの信頼性が確認できる方法が必要となってくるケースも出てきます。

- バージョンアップを行いたい
  DBMS のバージョンを上げる際には、一般的にバックアップが必要となります。これも 企業システムの非機能要求項目としては挙がっていませんが、バックアップの手法としてバージョンアップが行えるかどうかというのも、確認すべき項目の一つです。

これらの要件の中で、PostgreSQL のバックアップの手法によって、影響を大きく及ぼす項目を中心に本書では検討を行っていきます。

2.3. 監視

本節では、監視がなぜ重要なのかについて説明するとともに、DBMS の運用要件として求められる監視内容、実現方式について説明します。

2.3.1. 監視の重要性

業務システムを稼働させていくうえで、データベースは中枢を担うことが多くあります。データベースシステムの異常や停止は、業務システム全体に対して多大な影響を与え、場合によっては業務システム全体の停止につながります。これらを未然に防ぐためにも、必要なレベルでデータベースが正常に稼働しているか監視を行い、異常の発生やその兆候が見られる際には早期に手当をすることが重要となります。
「監視」というと、一般的に「死活監視」「エラー監視」「リソース監視」「パフォーマンス監視」といった監視項目が挙げられますが、本書では「死活監視」「エラー監視」「リソース監視」を単純に「死活監視」と、「パフォーマンス監視」を「性能監視」として扱います。

表 2.1: 監視内容と本書での対応付け

非機能要求	監視内容	内容	本書での用語
可用性を高めたい（サービス停止時間を短くしたい）	死活監視を行う	対象のステータスがオンラインの状態にあるかオフラインの状態にあるかを判断する監視のこと	死活監視
	エラー監視を行う	対象が出力するログ等にエラー出力が含まれているかどうかを判断する監視のこと	
	リソース監視を行う	対象が出力するログや別途収集するパフォーマンス情報に基づいて CPU やメモリ、ディスク、ネットワーク帯域といったリソースの使用状況を判断する監視のこと	
運用保守性を高めたい（正常運用を維持したい）	パフォーマンス監視を行う	対象が出力するログや別途収集するパフォーマンス情報に基づいて、業務アプリケーションやディスク I/O、ネットワーク転送等の応答時間やスループットについて判断する監視のこと	性能監視

「死活監視」、「性能監視」のそれぞれが 1 章で述べた「可用性」、「運用保守性」と以下のように関連付けることができます。

【可用性】
- 死活監視を行うことで、DBMS やサーバの異常発生を検知することができるため、サービス停止時間を短くすることができるなどの効果が期待できる

【運用保守性】
- 性能監視を行うことで、性能問題発生を検知することができるため、サービスへの影響を未然に防ぐなどの効果が期待できる

ひとことに「監視」といっても、性能に与えるインパクトや事後の分析を考慮し、何をどの頻度で監視するかを定めることが重要です。本書では「何を監視するか」といった内容を「監視情報」として考えます。また、「どの頻度で監視するか」といった内容を「監視間隔」と考えます。

DBMS 運用要件として監視を行う場合は、対象の DBMS だけでなくそれらを取り巻くリソース(OS や CPU、ネットワークなど)も適切に監視することが重要です。

図 2.1: DBMS 監視の重要性

なお、DBMS（PostgreSQL）の運用において、監視に必要な情報の収集などは能動的に行えますが、「監視」自体は能動的に実施されません。DB 管理者は別途監視ソフトウェア(Zabbix など)を導入し、監視を行います。本書では監視ソフトウェアの設定などについては多く記載はしませんので、必要に応じて対象の監視ソフトウェアの情報を得てください。

以降では、DBMS 監視で行うべき「監視情報」、「監視間隔」の概要について説明します。具体的な監視項目や監視方法などについては、4 章以降で説明します。

2.3.2. 監視情報

「監視情報」はシステム状態をきめ細かく把握するために、どの程度の監視が行えるか（どのような項目に関する監視機能を有しているか)によりレベル分けを行います。DBMS（PostgreSQL）の運用において取得すべき項目は、統計情報、PostgreSQL のログ、sar や vmstat などの OS コマンドから得られる情報、OS のサーバログなどがあります。詳細は 4 章以降を参照ください。

いずれの監視項目についても、ただ監視を行えばよいわけではなく、設計もしくは期待通りに動いている正常状態を把握して、異常発生時にどこが変化しているかを分析できるようにする/なることが重要です。

2.3.3. 監視間隔

「監視間隔」はシステム状態をきめ細かく把握するために、どの程度の頻度で監視すべきかによりレベル分けを行います。監視する目的に応じて、また監視によるオーバヘッドを考慮し、死活監視と性能監視では推奨される頻度は異なります。「目的」は 1 章で述べた「可用性」「運用保守性」のどちらに重きをおくか、ということです。つまり、監視を頻繁に行えば、短時間で異常を検知し即座に対応が可能となりますが、一方で大量の監視情報が蓄積していくことになり解析に手間を要するようになります。

このようにトレードオフの関係にある両者を「目的」に応じて調整した例としては以下のような設定が考えられます。

死活監視のように、監視を行うタイミングでの状態を把握するだけであれば、過去のデータはそこまで重要にはならないため、秒間隔でのリアルタイム監視が可能と考えられます。

一方、性能問題が発生した場合は、何故問題が発生したのか原因を追究したり、今後同様の問題が発生しない

ために改善策を施すなどが必要となります。このように、性能監視については問題発生時点の監視情報だけではなく、ある程度の期間の監視情報が必要になるため、分間隔でのリアルタイム監視を設定します。

以降断りがない限り、本書では「死活監視＝可用性」は秒単位の監視、「性能監視＝運用保守性」は分単位の監視を推奨します。

3. PostgreSQL の代表的な構成

前章でデータベースに求められる要件についてご説明しました。
本章では、PostgreSQL の代表的なシステム構成についてそれぞれの特徴と「データ同期性」「耐障害性」「拡張性」「コスト」「オーバーヘッド」の５つのポイントを整理します。

3.1. 基本構成（シングルサーバ）

シングルサーバとは、1 台のサーバで PostgreSQL データベースを構成します。基本的な構成のため、運用自体もシンプルです。ただし、サーバが単一障害点(SPOF)になり得るため、RAID 構成やバックアップを適正に行い、更新頻度に合わせて各種パラメータやログ格納エリアサイズを余裕を持った値にする、統計情報を日ごろから確認し些細な変化も見逃さないなど、PostgreSQL で運転継続性を高めるためのさまざまな考慮が必要です。
PostgreSQL のバックアップ／リカバリ運用には様々な分類や方式があります。以降の章で企業システムで採用されている方式について詳しく説明します。尚、バックアップ／リカバリはシングル構成前提が基本的な考え方です。また、障害が発生した場合に備え、取得したバックアップからの迅速なリカバリができる仕組み作りが欠かせません。そのためには事前にリカバリ手順の確認や運用担当者の習熟度を高めることも重要です。
監視もシングル構成を前提にデータベースシステムおよびデータベースが稼動するサーバにおいて「死活」と「性能」の２つの観点で情報を収集します。死活監視ではデータベースシステムが利用可能な状態であるか、また性能監視ではデータベースシステムが提供するサービスレベルが低下していないかを監視します。収集した情報をもとに待機サーバへの切替や性能劣化を引き起こすボトルネックの解消など迅速な対応を行います。

本構成の詳細説明は、以下の章に記載しています。
４章　基本構成（シングルサーバ）

表 3.1: 基本構成（シングルサーバ）の特性

ポイント	説明
データ同期性	シングルサーバのため同期はない
耐障害性	クラッシュ時には WAL ファイルからコミット済みデータのリカバリを行う ディスク障害発生時にバックアップからのリストア／リカバリが必要 インスタンス障害時に自動的に再起動される(postmaster 以外の障害の場合)
拡張性	スケールアウトはできないため、CPU やメモリの増設などスケールアップにて対応
コスト	以下の理由から、コストは「低」とする - リソースの利用効率は高い - 最小限のハードウェアで構成可能
オーバーヘッド	シングルサーバのためオーバーヘッドは発生しない

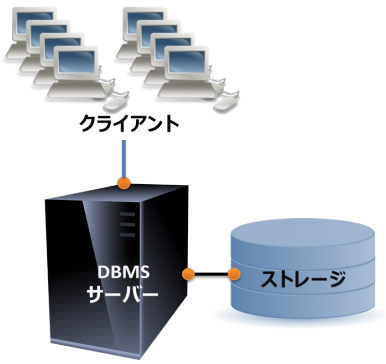


図 3.1: 基本構成（シングルサーバ）

3.2. HA クラスタ構成（共有ストレージ方式）

HA クラスタ構成とは 2 台以上のサーバを利用したアクティブ-スタンバイ構成の PostgreSQL データベースを指します。通常運用時はアクティブサーバで処理を行い、スタンバイサーバは障害発生に備えて待機します。共有ストレージ方式は、データベースのファイル群を共有ストレージ上に配置し、両サーバで共有するシンプルで実績のある方式と言えます。バックアップおよび監視は基本的にシングル構成と同様の考慮が必要です。

本構成の詳細説明は、以下の章に記載しています。
　５章　共有ストレージ

表 3.2 : クラスタ構成（共有ストレージ方式）の特性

ポイント	説明
データ同期性	ストレージを共有するため、同期処理は不要
耐障害性	インスタンス障害には対応しているが、共有ストレージに障害が発生した場合はサービス継続不可
拡張性	スタンバイ側の PostgreSQL は停止しているため、参照処理は不可能 3 台以上の構成も可能
コスト	以下の理由から、コストは「高」とする - 待機系はサービス停止中のため、リソースの利用効率は低い - 共有ストレージについては、冗長構成が取られているなどの信頼性が高いストレージが求められるケースが多く、高価になりがちである
オーバーヘッド	データ同期性が不要であるため、オーバーヘッドは発生しない

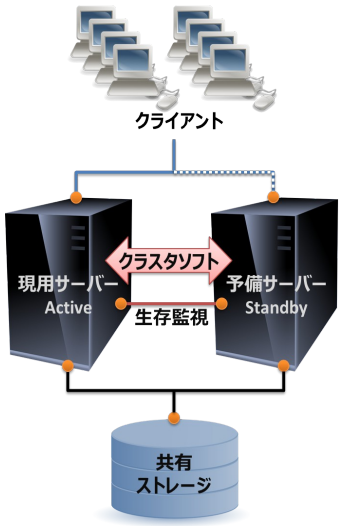


図 3.2: 共有ストレージ方式

3.3. HA クラスタ構成（シェアードナッシング方式）

ストレージレベルでのレプリケーションには、ハードウエアによる実装とソフトウエアによる実装があります。ここではソフトウエアによる実装について説明します。

ソフトウェアを利用したストレージレベルのレプリケーションとして、PostgreSQL ではオープンソースソフトウェアの DRBD を利用した例が多く報告されています。この構成では、データファイルをそれぞれローカルディスクに配置し、DRBD を利用してブロックデバイスレベルでブロックを伝播してデータを同期します。待機系では同期中はレプリケーション領域をマウントできないため、コールドスタンバイとなります。

ただし、死活監視と障害発生時のフェイルオーバは DRBD の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。オープンソースソフトウェアのクラスタソフトとしては、Pacemaker および Heartbeat と組み合わせる事例が多く報告されています。Heartbeat にてクラスタメンバを管理し、Pacemaker にてクラスタリソースを管理します。

バックアップおよび監視は基本的にシングル構成と同様の考慮が必要です。

本構成の詳細説明は、以下の章に DRBD の場合として記載しています。

６章　ストレージレプリケーション(DRBD)

表 3.3: HA クラスタ構成（ストレージレプリケーション方式）の特性

ポイント	説明
データ同期性	ブロック単位の同期処理 同期方式は、非同期モードおよび同期モードから選択（実際はより多くのモードがある）
耐障害性	インスタンス障害、ノード障害、ストレージ障害に対応している 障害によるフェールオーバ実施後、片側だけで更新が進むためデータの再同期が必要
拡張性	スタンバイ側の PostgreSQL は停止しているため、参照処理は不可 3 台以上の構成も可能
コスト	以下の理由から、コストは「中」とする - 待機系はサービス停止中のため、リソースの利用効率は低い - ローカルディスクにデータファイルの配置が可能であるため、費用は抑えられる
オーバーヘッド	データ同期が必要であるため、オーバーヘッドが発生 影響度は同期方式に依存

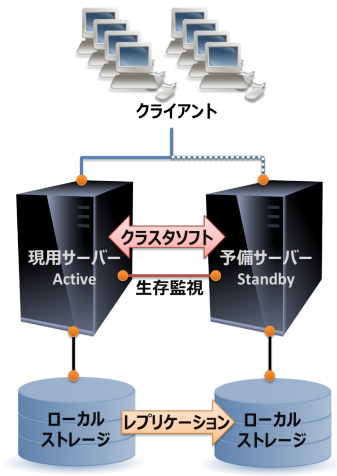


図 3.3: ストレージレプリケーション方式

3.4. ストリーミングレプリケーション

PostgreSQL 9.0 以降で利用可能なレプリケーションです。PostgreSQL の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します。データ転送は非同期、同期モードを選択することができ、複数台のスレーブサーバ構成をとることができます。PostgreSQL 9.2 からはカスケードレプリケーションも可能となり、拡張性が強化されています。またスレーブは参照処理を受け付けることが可能でリソースの利用効率が高い構成です。

ただし、死活監視と障害発生時のフェイルオーバは 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。参照の負荷分散を行う場合は、pgpool-II を使用します。

ストリーミングレプリケーションでは、複製したデータベースをマスタデータベースのバックアップとして運用することができます。ただし、一般的なバックアップと異なる特徴があるため、以降の章で考慮点や一般的なバックアップと組み合わせた運用方法を説明します。

本構成の詳細説明は、以下の章に記載しています。
7 章がストリーミングレプリケーションのみの構成、
8 章,11 章,12 章はクラスタソフトウェアとして pgpool-II を組み合わせた構成です。

7 章　ストリーミングレプリケーション
8 章　pgpool-II（マスタスレーブモード）
11 章　pgpool-II（アクティブスタンバイ）
12 章　pgpool-II（アクティブアクティブ）

表 3.4: ストリーミングレプリケーションの特性

ポイント	説明
データ同期性	操作単位の WAL を非同期、同期モードで伝播
耐障害性	インスタンス障害、ノード障害、ストレージ障害に対応している 複数台のスレーブサーバ構成が可能
拡張性	スレーブサーバの DB は参照可能であるため、参照処理の負荷分散が可能 マルチスレーブサーバ構成とする事が可能（PostgreSQL 9.2 からはカスケード構成も可能）
コスト	以下の理由から、コストは「低」とする - スレーブサーバは参照可能なため、リソースの利用効率は高い - 共有ストレージなど高価なハードウェアは不要
オーバーヘッド	WAL レコード転送によるデータ同期を行うため、オーバーヘッドが発生 影響度は同期方式に依存

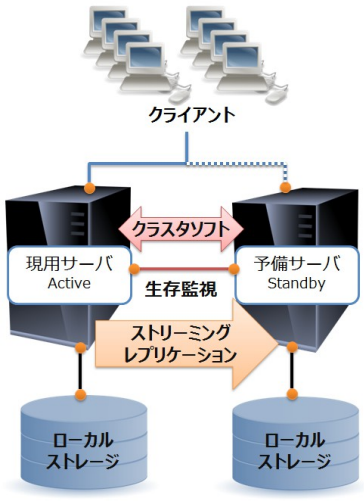


図 3.4: ストリーミングレプリケーション方式

3.5. pgpool-II（レプリケーションモード）

pgpool-II のレプリケーションモードを利用することで複数サーバ間で参照、更新処理を行うことができます。本構成ではアプリケーションから発行された SQL の種類を pgpool-II が判別し、更新処理であればレプリケーションを構成するすべてのサーバに同じ SQL を実行することでデータの同期を行います。本構成もストリーミングレプリケーションと同様に、レプリケートされたデータベースをバックアップとして運用することができます。

本構成の詳細説明は、以下の章に記載しています。
９章　pgpool-II（レプリケーションモード）

表 3.5: pgpool-II（レプリケーションモード）の特性

ポイント	説明
データ同期性	SQL 単位で同期
耐障害性	インスタンス障害、ノード障害、ストレージ障害に対応している 複数サーバ構成が可能
拡張性	全サーバがマスタであり、参照および更新が可能 3台以上の構成が可能
コスト	以下の理由から、コストは「低」とする - 全サーバで参照・更新処理が可能なため、リソース利用効率は極めて高い - 共有ストレージなど高価なハードウェアは不要
オーバーヘッド	各ノードが同時に更新処理を行うためシングル構成と同等に近いが、全ノードのコミット完了を待機するため若干のオーバーヘッドは発生する

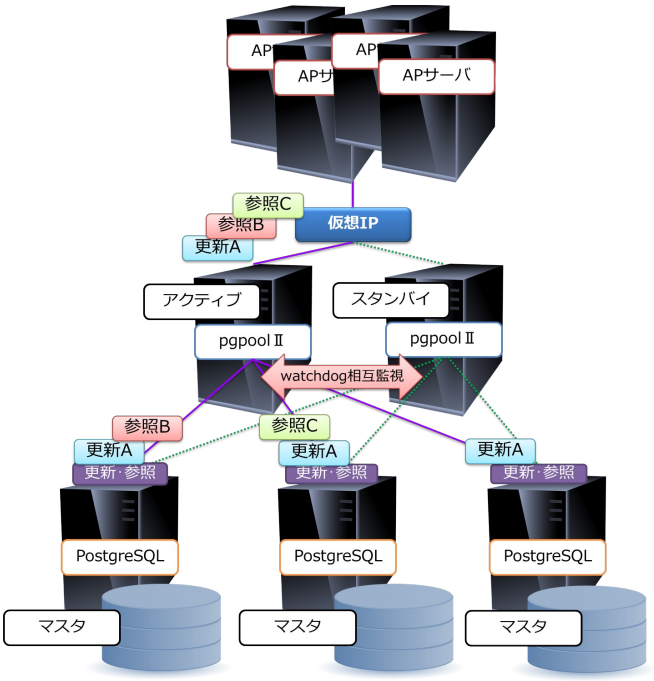


図 3.5: pgpool-II（レプリケーションモード）

3.6.Slony-I （トリガーベース）

Slony-I は PostgreSQL 専用の非同期方式・シングルマスタ・マルチスレーブ型のレプリケーションソフトウェアで、トリガーベースでテーブル単位のデータ複製を行います。構成は、参照/更新がともに可能なマスタサーバ 1 台に対して、参照クエリのみを受け付けるスレーブサーバを複数台持たせることができます。

本構成の詳細説明は、以下の章に記載しています。
10 章　Slony-I

表 3.6: Slony-I(トリガーベース）の特性

ポイント	説明
データ同期性	トリガーベースで非同期
耐障害性	インスタンス障害、ノード障害、ストレージ障害に対応している（全テーブルが複製の対象である場合） スレーブサーバ障害のマスタへの影響なし
拡張性	スレーブサーバの DB は参照が可能であるため、参照処理の負荷分散が可能 マルチスレーブ型であるため、3台以上の構成も可能
コスト	以下の理由から、コストは「低」とする - スレーブサーバは参照可能なため、リソースの利用効率は高い - 共有ストレージなど高価なハードウェアは不要
オーバーヘッド	トリガーにより管理テーブルへの登録処理を行うため、オーバーヘッドが発生

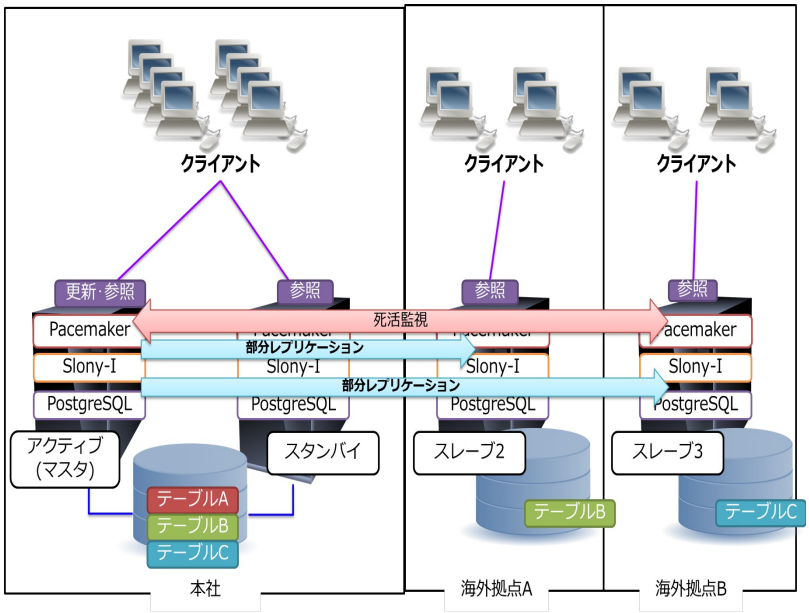


図 3.6: Slony-I

4. 基本構成（シングルサーバ）

前章で整理した PostgreSQL の代表的なシステム構成の特徴を踏まえ、本章以降でそれぞれのシステム構成の技術要素、運用時のポイントといった点を、可用性、バックアップ、監視の観点で解説してきます。
本章では最もシンプルな構成であり、他の構成との比較対象となるシングルサーバ構成について記述します。

4.1. 前提とする構成

シングルサーバは 1 台のサーバで PostgreSQL データベースを構成します。サーバなどの障害に対する可用性は高くありませんが、システム構成としては最も安価に構築することができます。

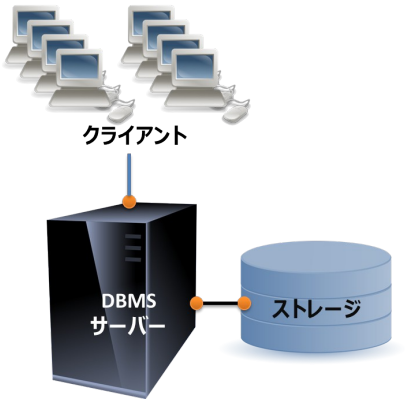


図 4.1: 基本構成（シングルサーバ）

4.2. 可用性

4.2.1. WAL ファイルの冗長化

シングルサーバの PostgreSQL は、データを格納するファイルおよび WAL ファイルや制御ファイルで構成されるデータベースクラスタとその管理を担うプロセス群、そしてメモリ領域のインスタンスで構成されます。
シングルサーバ構成では単一障害点であるハードウェアに障害が発生すると、ハードウェアが復旧するまでサービスを提供できなくなります。また、データベースを構成するファイル群が格納されたストレージに障害が発生すればバックアップから復旧させるまではサービスを提供することができません。
PostgreSQL のバックアップ方式は後述しますが、バックアップ・リカバリに必要不可欠なファイルとして WAL ファイルの存在があります。WAL（Write Ahead Log）とはデータベースに対する変更履歴が記録されるファイルで、インスタンス障害は WAL の自動適用により復旧できますし、ストレージに障害が発生した場合はバックアップからのファイルリストアと WAL ファイルの適用により障害発生直前までリカバリすることができます。
障害からの復旧に必要な WAL ファイルの消失を防ぐためには冗長化が必須ですが、残念ながら PostgreSQL には関連機能は実装されていません。そのため PostgreSQL 以外の機能で実装する必要があります。
対応策としては以下の手法が挙げられます。

- 高信頼性ディスクに配置
  信頼性の高いディスク装置に配置耐障害性を高める RAID1 や、RAID0 との組み合わせである RAID1+0 の構成が広く採用されています。
- ストレージ冗長化
  PostgreSQL では WAL の冗長化機能が実装されていないため、ストレージ冗長化にて代替する方式が考えられます。ストレージ冗長化にはハードウェアによる実装とソフトウェアによる実装があり、いずれの場合も WAL ファイルをレプリケーション領域に配置します。
  ソフトウェアによる実装としては DRBD などがあります。同期方式は同期/非同期など数種類から選択できますが、同期式が推奨されます。安価なストレージで実現できるのが特徴です。

ハードウェアによる実装としてはハードウェアベンダから各種のストレージが提供されており、信頼性が高い事が特徴です。

4.3. バックアップ

PostgreSQL のバックアップ／リカバリの運用には様々な分類や方式があります。ここでは企業システムで採用されているバックアップ／リカバリを中心に、主なバックアップ／リカバリ方式について説明します。PostgreSQL では様々なシステム構成を組むことができますが、シングル構成におけるバックアップ／リカバリ方式が基本となります。

4.3.1. 論理バックアップ／リストア

PostgreSQL の論理バックアップは pg_dump(pg_dumpall)を利用してデータのリストアができる SQL のスクリプト、またはアーカイブ・ファイルを生成するバックアップ方式です。リストアは平文形式に取得したかカスタム形式で取得したかによって、それぞれ psql、pg_restore コマンドを利用します。

論理バックアップとはデータの中身を取得するバックアップ方式のことで pg_dump は内部的に SELECT 文を利用して取得したデータを SQL 文のスクリプトに変換する作業を行ないます。そのため、取得時の一貫性がとれたデータを取得することができます。また、基本的に SELECT なので他のユーザの更新、参照作業を妨げることなくバックアップを取ることができます。ただし、バックアップ中にはデータベースに対する負荷は上がる点には注意が必要です。

論理バックアップの特徴を以下にまとめます。

- 平文形式で取得した場合、内容を人が確認できる形式でバックアップされる。
- クライアントとしてデータを取得(SELECT)するので PostgreSQL の Isolation レベルで一貫性がとれたデータを取得することができる。もし、データが壊れている場合は取得時エラーが発生するのでデータが壊れていることに気づく可能性が高い。
- テーブル単位でバックアップとリストアができる。また、バックアップをとったテーブルと異なるテーブルにデータを戻すことも可能。
- 異なるバージョンの PostgreSQL のバックアップからリストアすることができる。
- 圧縮可能なデータの場合、物理バックアップに比べてバックアップファイルのサイズが小さい。
- 物理バックアップ比べてバックアップ時間が長くデータベースを参照するため、消費するリソースが大きい。
- バックアップした時点以後のデータの復旧はできないのでオペレーティングミスによるデータ損失などの場合、バックアップした時点以後の更新したデータは復旧することはできない。

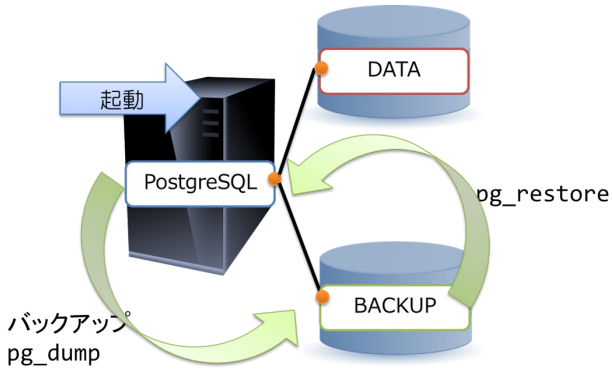


図 4.2: 論理バックアップ／リストア

4.3.2. オフラインバックアップ／リストア(物理バックアップ)

データベース(PostgreSQL)が停止している状態で OS の cp,rsync コマンドなどを利用して、データファイルをコピーするバックアップ方式です。論理バックアップと違ってデータの中身を取得するのではなく、データファイルそのものを取得する物理バックアップ方式です。

リストアはバックアップしたファイルを置き換えるだけなので簡単です。一貫性があるデータを取るためにはデータ

ベースを停止する必要がありますが、rsync コマンドを利用してデータベース起動中にデータをコピーしてからデータベースを停止して差分だけをコピーすることでデータベースの停止時間を抑えることができます。

オフラインバックアップ・リストアの特徴を以下でまとめます。

- バックアップ、リストアの手順が単純。
- 論理バックアップに比べてバックアップ・リストア時間が短い。
- 圧縮可能なデータの場合、論理バックアップに比べてバックアップのデータサイズが大きい。
- データベースを一旦停止する必要がある、ただし rsync を利用すれば停止時間を抑えることが可能。
- バックアップされたデータの内容を直接読むことはできないので、問題が発生したときに原因の把握、解決が難しい。
- バックアップした時点以後のデータの復旧はできないのでオペレーティングミスによるデータ損失などの場合、バックアップした時点以後の更新したデータは復旧することはできない。

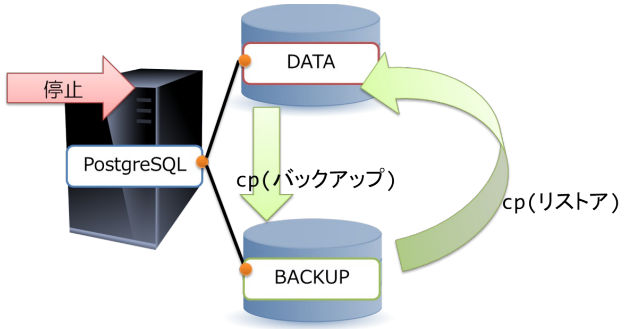


図 4.3: オフラインバックアップ／リカバリ

4.3.3. オンラインバックアップ／リカバリ(物理バックアップ)

データベース(PostgreSQL)が起動している状態で OS の cp コマンドと SQL 関数である pg_start_backup(), pg_stop_backup()、または、これらを自動化させたツールである pg_basebackup(PostgreSQL 9.1 から提供)を利用するバックアップ方式です。さらにアーカイブログ(WAL のアーカイブ)運用することで、特定時点のデータの状態に戻す Point In Time Recovery(以下 PITR)を利用することもできます。

PITR はアーカイブログを利用したロールフォワードで最新、または特定時点のデータにリカバリすることができます。そのため、ベースバックアップ取得以後、障害発生時までにデータの更新があってもデータの損失なく、最新の状態、または指定した時点のデータにリストアすることができるのが特徴です。

オンラインバックアップ・リストアの特徴を以下でまとめます。

- バックアップ取得からデータの更新があっても最新データにリストア可能。
- 特定時間を指定してリストアすることができるのでオペレーティングミスによるデータ削除にも対応可能。
- オフラインバックアップのサイズ+アーカイブログのサイズのディスク空間が必要であるため、かなり大きいディスク空間を必要とする。
- 復旧時間はベースバックアップのコピー時間+ロールフォワード時間であるため、ベースバックアップを取得してからの更新量に大きく依存する。
- 復旧時間、アーカイブログの容量の節約のために定期的に新しいベースバックアップを取得する必要がある。新しいベースバックアップを取得したとき、ベースバックアップ取得以前に確保していたアーカイブログは削除してもいい。(容量の観点から削除することを推奨)
- ベースバックアップ以前の時点に戻すことはできないことに注意。

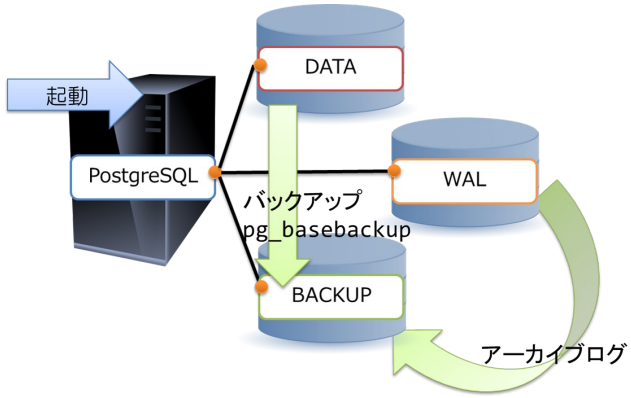


図 4.4:オンラインバックアップ

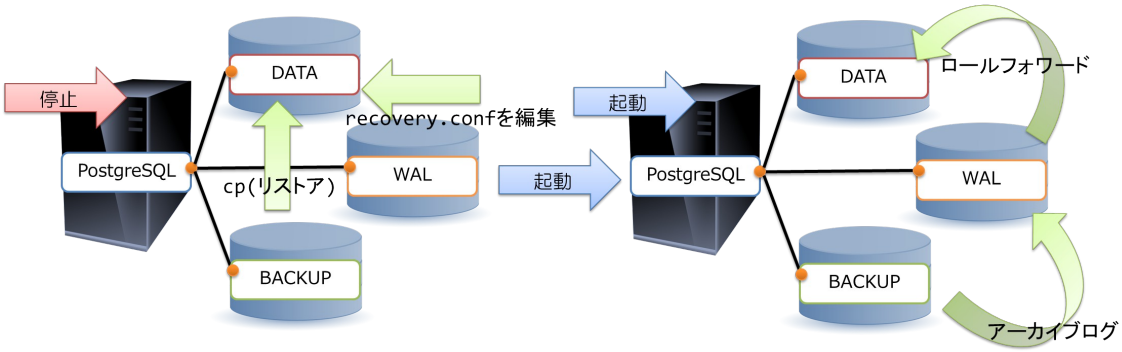


図 4.5: Point in time recovery

4.3.4. ストレージローカルコピー

データベース(PostgreSQL)の物理バックアップや物理リストアの手段に、ストレージ(ハードウェア)が持つローカルコピー機能を用いる方式です。ストレージによるデータコピーは様々な方式や呼称[4]があり、本書では、ストレージローカルコピーと呼んでいます。ファイルシステム/パーティションレベルでのソフトウェアによるミラーとして、DRBDやLVMミラーなどがありますが、この方式はハードウェアの機能を使用したもので、オフラインでもオンラインでも用いられるバックアップ方式です。

4　ミラーリング、ShadowImage、スナップショット、ボリュームレプリケーション、ボリュームコピー、など

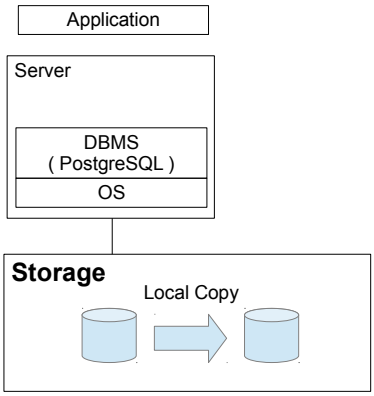


図 4.6: ストレージローカルコピー

DBMS や OS といったソフトウェアの種別やバージョン、サーバの動作状態とは無関係に、接続された外部ストレージ側で独立して物理的にデータをコピーするため、バックアップやリストアに長時間かかるような大規模データを扱う場合でも短時間で処理でき、バックアップ処理がサーバを利用する業務へ大きな影響を与えないことが特徴です。

この方式では、PostgreSQL のオンラインバックアップとして PITR で物理コピーする際に、ミラーされた正ボリューム/副ボリュームをスプリットする(正副ボリュームを切り離してコピーを止める)ことで、ある時点でのバックアップを取得する方法になります。注意点としては、明示的に I/O の静止点を設ける必要があります(付録参照)。

なお、純粋なバックアップとしてだけではなく、共有ディスク構成で共有ディスクが SPOF となることを避けるため、HA クラスタソフトウェアと組み合わせて、副ボリュームを予備サーバ用のストレージとして使用するケースもあります。

4.3.5. バックアップ方式のまとめ

ここまで説明してきたバックアップ方式と、PostgreSQL のレプリケーションによって構成されたスタンバイサーバをバックアップとして使用するケースをまとめると次の表のようになります。なおレプリケーションについての詳細は「7.3 バックアップ」で説明します。

表 4.1: 主なバックアップ／リカバリ方式

		論理バックアップ	オフラインバックアップ	オンラインバックアップ（物理）		レプリケーション	
				OS レベルのファイルコピー	ストレージローカルコピー	非同期	同期
a	バックアップ概要	pg_dump コマンド等を使用	データベースを停止し、cp,rsync などを用いたファイルコピー	pg_start_backup と cp,rsync などの組み合わせ、または pg_basebackup を使用	I/O の静止点でストレージの機能を用いてデータを複製	ストリーミングレプリケーション構成のスタンバイをバックアップとみなす。	
b	業務への影響度	バックアップ中はストレージ負荷が高まる	バックアップ中は業務停止	バックアップ中はストレージ負荷が高まる	大きな影響なし	大きな影響なし	性能面での影響大
c	必要構成	—	—	アーカイブログ運用	左記に加えストレージローカルコピー機能を備えたストレージが必要	スタンバイサーバが必要	スタンバイサーバが必要（※1）
d	バックアップ最小単位	テーブル	データベースクラスタ	データベースクラスタ	データベースクラスタ	データベースクラスタ	
e	バックアップ取得時間	ファイルコピーと比較して長時間（検証では約 2 倍）	ファイルコピー時間と同等	ほぼファイルコピー時間と同等（※2）	短時間（数秒～数分）	—	
f	リストア・リカバリ時間（RTO）	ファイルコピーと比較して長時間（検証では約 5～10 倍）	ファイルコピー時間と同等	ファイルコピー時間＋WAL のロールフォワード時間	短時間（数秒～数分）	—	
g	障害時にどの時点までデータを復旧できるか（RPO）	バックアップ取得時点	バックアップ取得時点（※3）	障害発生時点	障害発生時点	障害発生時点（※4）	障害発生時点
h	オペレーションミスによるデータ障害への対応	基本的に不可（※6）	基本的に不可（※5）	PITR により可	PITR により可	不可	
i	リストア・リカバリに必要なデータ	ダンプファイル	バックアップデータファイル（※7）	・バックアップデータファイル ・WAL ファイル ・アーカイブされた WAL ファイル	・バックアップデータファイル ・WAL ファイル ・アーカイブされた WAL ファイル	—	
j	まとめ・注意点	・比較的小さなデータベースのバックアップに使用する ・テーブル単位、データベース単位にバックアップすることが可能 ・物理バックアップより処理時間が長い	・データベースを停止できる場合に使用する ・ストレージローカルコピーを用いてバックアップすることも可	・可用性（特に継続性）要件のレベルの高いシステムで推奨する方式 ・不要となったアーカイブされた WAL を定期的に削除する必要あり	・バックアップ／リカバリを短時間で実行する要件がある場合に使用する ・I/O の静止点でバックアップを実施する ・アーカイブされた WAL については左記と同様	・他の方式では RTO 要件を満たせない場合に使用する ・非同期レプリケーションでは、マスタ障害時にコミット済データが欠損する可能性がある ・他のバックアップ方式と併用することを推奨	

※1 同期レプリケーション構成の場合、スタンバイサーバに障害が発生するとマスタサーバでトランザクションの実行ができなくなるため、可用性の観点からはスタンバイサーバを複数用意するなどの対応が必要である。
※2 pg_basebackup を使用した場合、検証ではデータベースサーバ上のファイルコピー（cp など）よりは処理時

間がかかるが、リモートコピー（scp）よりは短時間でバックアップ可能であった。
※3 バックアップ取得後、障害発生時点までの WAL ファイルが全て残っている場合、障害発生時点までの復旧が可能である。
※4 マスタサーバに障害が発生し、スタンバイにレプリケーションされていない WAL データがある場合、マスタ上の WAL ファイルを手動でスタンバイにコピーするなどの対応が必要になる。障害でマスタ上の WAL ファイルが失われた場合、レプリケーションされていないデータは復旧できない。
※5 オペレーションミスによるデータ障害がバックアップ取得後かつ、オペレーションミス時点までの WAL ファイルが全て残っている場合、オペレーションミス以前の状態に復旧可能である。
※6 バックアップ取得後、オペレーションミスまでデータが変更されないデータについては、復旧可能である。
※7 ※3 に記載しているようなケースでは、復旧に WAL ファイルやアーカイブされた WAL ファイルも必要になる。

4.4. 監視

本節では PostgreSQL が単体で動作するシステムを前提にし、データベースおよびデータベースを稼働させるサーバに対してどのような監視を行うかについて説明します。なお、監視に際しては OS のコマンドを利用することがあります。本書では、基本的に Linux のコマンドを前提に記載します。

4.4.1. 監視するために収集する情報

PostgreSQL および稼動サーバに対する監視は「死活」と「性能」の2つの観点から行われます。

4.4.1.1. 死活監視するために収集する情報

データベースシステムが利用可能（オンライン)な状態となっているか、監視します。
主な監視対象としては、サーバの応答、ネットワークの異常、PostgreSQL プロセスの停止、SQL の実行可否などがあげられます。通常、監視システムを用いてこれらを数秒程度のリアルタイム間隔にて監視を行い、異常発生時に運用オペレータに通知するよう措置します。
以下の表にサーバ、PostgreSQL それぞれの死活監視で収集すべき主な監視対象をまとめます。

表 4.2: サーバ情報による死活監視

項目	頻度	対象	手段	確認内容
ネットワークの疎通	数秒間隔	ネットワーク疎通の有無	ping コマンド	一定の時間内で応答があること
サーバ OS の応答		サーバ OS の応答	ping コマンド	一定の時間内で応答があること
エラーメッセージ		サーバ上の問題	/var/log/message ファイル	エラーを示すメッセージが無いこと

表 4.3: PostgreSQL 情報による死活監視

項目	頻度	対象	手段	確認内容
PostgreSQL プロセス	数秒間隔	PostgreSQL プロセスの有無	ps コマンド pg_stat_activity ビュー	規定の PostgreSQL プロセスが存在すること
SQL 実行		SQL 実行可否	psql コマンド (SELECT 1 の発行など)	一定時間内に応答があること
エラーメッセージ		PostgreSQL 上の問題	/var/log/message ファイル, pg_log/*のファイル	エラーを示すメッセージが無いこと

4.4.1.2. 性能監視を行うために収集する情報

性能監視では、データベースが提供するサービスレベルが劣化していないか監視します。

データベースを稼働させる OS リソースの不足やネットワークの遅延など、主に基盤リソースの不足によりサービスレベルは悪化します。また長期の稼働期間中に発生したデータの追加／削除等によるストレージのフラグメント、データの配置パターンとマッチしない非効率なインデックスなど、経年的に性能が劣化する要因があります。これらの要因により、データベースの性能が劣化していないか監視します。

以下の表にサーバ、PostgreSQL それぞれの性能監視で収集すべき主な監視対象をまとめます。

表 4.4: サーバ情報による性能監視

項目	頻度	対象	手段	確認内容
CPU	数分間隔	CPU 使用率	sar コマンド	CPU 使用率が規定の範囲内であること
スワップ		スワップ回数	vmstat コマンド	スワップ回数が規定の範囲内であること
I/O 利用状況		I/O ビジー率	sar コマンド, iostat コマンド	I/O ビジー率が規定の範囲内であること
I/O 待ち状況		I/O 待ち個数	sar コマンド, iostat コマンド	I/O 待ち個数が規定の範囲内であること
NW 利用状況		NW 帯域使用率	sar コマンド	NW 帯域の使用量が規定の範囲内であること
ディスクスペース		ディスク使用率	df コマンド, du コマンド	ディスク使用率が規定の範囲内であること

表 4.5: PostgreSQL 情報による性能監視

項目	頻度	対象	手段	確認内容
応答性能	数分間隔	スロークエリ (遅いクエリの有無)	log_min_duration_statement(ログを確認), pg_stat_statements ビュー	log_min_duration_statement の設定値を超えた際に出力されるログ, total_time/calls で実行時間を確認
		接続[ロングトランザクション] (長時間放置されているトランザクション有無)	pg_stat_activity ビュー	state が「idle in transaction」のままで、xact_start が古い(長時間 COMMIT していない)トランザクションの有無
		ロック (ロック待ち時間の長いクエリ有無)	pg_locks ビュー, pg_stat_activity ビュー, pg_class カタログ	granted が false で、xact_start 長い問い合わせがないかを確認 (補足) pg_locks の pid と pg_stat_activity の pid、pg_locks の relation と pg_class の OID を結合するとよい
		キャッシュヒット率	pg_stat_database ビュー, pg_statio_all_tables ビュー,	blks_hit, blks_read, heap_blks_hit, heap_blks_read, idx_blks_hit, idx_blks_read の値から算出した値

項目	頻度	対象	手段	確認内容
			pg_statio_all_indexes ビュー	(補足) 「blks_hit*100/ (blks_hit+blks_read)」で対象のデータベースのキャッシュヒット率が求まる
キャパシティ		同時接続数	pg_stat_activiry ビュー	pg_stat_activity ビューの行数
スループットとエラー		commit/rollback 回数	pg_stat_database ビュー	xact_commit, xact_rollback の回数
DB 性能保全		インデックスのばらつき (ディスクの利用状況)	pg_stats ビュー	correlation（統計的相関）が-1 または　+1 に近いこと
		インデックス利用率	pg_stat_all_tables ビュー	seq_scan, idx_scan の値から算出する値 (補足) idx_scan*100/ (seq_scan+idx_scan)で対象テーブルのインデックス利用率が求まる
		オブジェクトサイズ	pg_database_size 関数, pg_total_relation_size 関数, pg_relation_size 関数	左記関数の結果の増分
		不要領域	pg_stat_all_tables ビュー	n_live_tup、n_dead_tup で分かる有効な行数、無効な行数

4.4.2. 収集した情報の分析／対処

常時稼働を求められることが多いデータベースにとって、障害を早期検出する死活監視は重要な位置を占めます。特に冗長化が行われていないシングルサーバ構成においては、早期に障害原因を切り分け、障害原因を取り除くことが求められます。本節では、監視システムが PostgreSQL の障害を検出した際に、どのように障害原因を切り分け、対応を行うべきかについて説明します。

4.4.2.1. データベース異常時の分析/対処

4.4.1 の項で前述したように、PostgreSQL の死活監視は主に以下の観点で行われます。実際には、原因の切り分けを容易にするためにより細かい単位で監視を行うこともありますが、その場合も障害時に確認すべき点は同様です。

- 対象サーバへのネットワーク経路に異常が無いか (ping による疎通確認等)
- 対象サーバ上で稼働する PostgreSQL に対する SQL 要求に対して応答が返ってくるか
- PostgreSQL から応答が返ってくる場合、正常応答が得られるか

上記の監視結果が正常値でないというアラートが検出された場合、上記の観点から対象サーバの状態を順番に確認して速やかに原因を特定し、復旧作業を行う必要があります。それぞれの観点における障害原因の切り分け手順を以下に示します。

1. 対象サーバへのネットワークに異常が発生した場合

対象サーバからの ping 応答が得られなかった場合、主な原因としては以下が考えられます。

- 接続元サーバから対象サーバへの経路上でネットワーク障害が発生している
- 対象サーバの電源が落ちている、もしくはハードウェアに障害が発生している
- 接続元サーバから対象サーバへの経路上にある Firewall 等の機器で ICMP パケットがフィルタリングされている
- 対象サーバ上の iptables 等のソフトウェアで ICMP パケットがフィルタリングされている

いずれの原因も、直接 PostgreSQL とは関連しない環境依存の要因であると思われます。各環境の設置環境に応じて障害箇所を特定し、対応を行って下さい。

2. 対象サーバ上で稼働する PostgreSQL への SQL 要求に対して応答が返ってこなかった場合

対象サーバからの ping 応答は得られるが、対象サーバ上の PostgreSQL から応答が得られなかった場合、主な原因としては以下が考えられます。

- 接続元サーバから対象サーバへの経路上にある Firewall 等の機器で、PostgreSQL が稼働しているポートへの通信がフィルタリングされている
- 対象サーバ上の iptables 等のソフトウェアで、PostgreSQL が稼働しているポートへの通信がフィルタリングされている
- 対象サーバ上の PostgreSQL のプロセスが稼働していない
- 対象サーバ上の PostgreSQL のプロセスがハングアップしている

前者 2 つについては、それぞれのフィルタリング設定の確認を行います。後者 2 つについては、対象サーバ上で PostgreSQL のプロセスが稼働しており、設定したポート番号でプロセスが LISTEN 状態になっていることを、netstat コマンド等を用いて確認します。

PostgreSQL 9.3 以降には、対象の PostgreSQL に接続可能かを調査するための pg_isready というコマンドが用意されています。PostgreSQL 9.3 以降を利用している場合はこのコマンドを使用すると対象の PostgreSQL に接続可能かどうかを判断することができ、接続できなかった場合は応答が無かったのか、それとも PostgreSQL に接続を拒否されているのかを切り分けることができます。

3. 対象サーバ上で稼働する PostgreSQL に対する SQL 要求に対して正常応答が返ってこなかった場合

対象サーバ上の PostgreSQL から応答は得られるがエラー応答である場合、主な原因としては以下が考えられます。

- 接続元サーバからの接続が拒否されている (pg_hba.conf で許可されていない)
- 指定したユーザの認証に失敗している (ユーザが存在していない、または認証情報が間違っている)
- 指定したデータベースが存在していない
- 指定したユーザに対象のデータベース/テーブルにアクセスする権限が無い
- 同時に接続できるコネクション数の上限に達している
- ディスクの空き容量が残っておらず、書き込みに失敗している
- その他のエラー

上記の切り分けはエラー応答の内容を確認して行います。

a) 接続元サーバからの接続が拒否されている場合

接続元サーバからの接続が拒否されている場合、以下のような応答が返されます。

```
FATAL:  no pg_hba.conf entry for host "127.0.0.1", user "postgres", database "postgres", SSL off
```

この場合、対象サーバ上の pg_hba.conf で接続元サーバからの接続が許可されているかどうかを確認し、必要に応じて修正を行います。

b)　指定したユーザの認証に失敗している場合

指定したユーザの認証に失敗している場合、以下のような応答が返されます。

```
FATAL:  password authentication failed for user "postgres"
```

この場合、初期設定の問題であれば、適切なユーザ・パスワードを用いて監視を行うように修正を行います。設定を変更していないのに特定の時期から通信ができなくなった場合は、意図しない変更が行われていないかを調査します。

c)　指定したデータベースが存在しない場合

指定したデータベースが存在しない場合、以下のような応答が返されます。

```
FATAL:  database "dummy" does not exist
```

この場合、初期設定の問題であれば、監視対象とするデータベースを適切に設定するよう修正を行います。稼働中のシステムで特定の時期からエラーが発生するようになった場合は、意図しない変更が行われていないかを調査します。

d)　指定したユーザに対象のテーブル等のリソースにアクセスする権限が無い場合

指定したユーザに対象のテーブルにアクセスする権限が無い場合、以下のような応答が返されます。

```
ERROR:  permission denied for relation dummy_table
```

この場合、必要であればアクセス権限の付与を行います。

e)　同時に接続できるコネクションの上限に達している場合

同時に接続できるコネクションの上限に達している場合、以下のような応答が返されます。

```
FATAL:  sorry, too many clients already
```

この場合、監視系等も含めて上限内に収まるよう、最大コネクション数の値を調整します。特定の時期からエラーが発生するようになった場合、終了されずに保持され続けているコネクションが滞留していないか確認します。

f)　その他のエラー応答が返ってきている場合

エラー内容やエラーログに出力されている内容を確認して適宜対応を行います。エラーログごとの対応方法については、次項のエラー発生時の原因分析/対処において述べます。

4.4.2.2. エラー発生時の原因分析／対処

1.　概要

シングルサーバ構成のデータベースシステムでエラーが発生した場合、関連するシステム全体へ多大な影響を与えるため、早期にシステムを復旧することが求められます。
PostgreSQL では、エラーの発生と同時にエラーの内容と詳細な情報を含むエラーメッセージがログへ出力

されます。システムでエラーが発生した場合は、エラーメッセージからエラー内容や原因の分析を行い、分析結果に基づいた適切な対処を行うことが重要となります。

本節では、PostgreSQL から出力されるメッセージや、実際に出力されるエラーメッセージを例にエラーの原因分析と対処方法について説明します。

PostgreSQL では日本語表記のメッセージを出力することが出来ますが、本書では英語表記のメッセージを用いて説明します。また、出力メッセージの言語は、**postgresql.conf** ファイル内のパラメータ **lc_messages** を設定することで変更することができます。

2. 出力メッセージについて

本項では、実際に PostgreSQL より出力されるメッセージの分析方法について記載します。
PostgreSQL では、下記フォーマットに従って動作に関する情報やエラーメッセージがログへ出力されます。
[プレフィックス][メッセージレベル]: [メッセージ]

- プレフィックス
  postgresql.conf ファイル内のパラメータ **log_line_prefix** で指定した情報が記録されます。
  含まれる情報にはタイムスタンプなどが挙げられます。
- メッセージレベル
  メッセージの種類や影響範囲に応じたレベルが記録されます。
- メッセージ
  発生した事象についての詳細な情報が記録されます。
  エラーが発生した場合、エラーの原因となったオブジェクトが記録されます。

PostgreSQL がメッセージに割り当てるレベルは下表のとおりです。PostgreSQL でエラーが発生した場合は、エラーの影響範囲に応じて **ERROR、FATAL、PANIC** の 3 つのレベルのいずれかが割当てられます。
そのため、エラーが発生した場合は主に上記 3 つのレベルのメッセージを手がかりに原因を分析します。

表 4.6: メッセージレベル一覧

レベル	説明
DEBUG [1～5]	開発者が主に利用する PostgreSQL の詳細情報
INFO	利用者の操作により暗黙的に要求された情報
NOTICE	利用者に向けた補助的な情報
WARNING	利用者への警告・注意
ERROR	**現在のコマンドが中断される原因となったエラー**
LOG	データベース管理者向けの情報
FATAL	**現在のセッションが中断される原因となったエラー**
PANIC	**全てのセッションが中断される原因となったエラー**

また、PostgreSQL からログファイルへ出力されるメッセージについては、**postgresql.conf** ファイル内のパラメータ **log_min_message** を設定することで変更できます。
デフォルトでは、上表の **WARNING** より下に記載されているレベルについて出力されるように設定されていますが、実際の運用では、監視に必要な情報が出力されるように適切な設定を行うことを考慮する必要があります。上記以外にも HINT、DETAIL、STATEMENT など回避策や詳細、原因となった SQL 文が出力されます。

このように PostgreSQL では、エラー発生時にログに記録されるエラーメッセージが、原因分析と対処方法を計画する上で非常に有用です。
エラーの原因と対処方法についてはケースバイケースの場合が多いため、本書ではケーススタディとしてまとめています。**14.3 監視ケーススタディ**を参照してください。

4.4.2.3. 性能低下時の分析と対処

1. 概要

前述のとおり、データベースは運用とともにデータやユーザの増加などにより性能が低下することが有ります。性能の低下を防止するためには、問題となっている要因を素早く特定して適切な処置を行う事が重要となります。
また要求された性能を達成するだけでなく、運用の見直しを含めた恒久的な対処となっているかということも考慮しなくてはなりません。
本節では取得した情報に対して、想定されるパフォーマンス低下の要因と対応方針について説明します。

2. 性能低下時の問題に対する対応方針

本項では性能低下時の問題に対して行う基本的な対応方針に関して説明します。
原因やデータベースの用途等の要因によって多少の差異は有りますが、大きく分類すると下記のように分けることが出来ます。

1. 運用管理の見直し
2. サーバ/PostgreSQL のパラメータチューニング
3. SQL チューニングやデータベース設計の見直し
4. アプリケーションの修正
5. ハードウェアの増強

基本的に【1】から【5】まで順番に対応を行い、要求されている性能を達成することを目標とします。
アプリケーション改修が不要な【1】【2】は改善効果が薄い場合がありますが、工数やコストが少なく済むという特徴があります。
一方でアプリケーションや機器に改修を加える【3】【4】【5】は高い改善効果を見込むことが出来ますが、工数やコストが高くなるという特徴を持っています。

3. 性能低下時の原因分析と対処

本項では前項で提示した対応方針の詳細を説明し、前項の性能監視時に取得したサーバの情報に対して対応方針を割り当てたものを表 4.7 に、PostgreSQL の情報に対して対応方針を割り当てたものを表 4.8 に記載します。

1. 運用管理の見直し

ログの管理や VACUUM、REINDEX 等の定期的なデータベースのメンテナンス方法を適切な形に見直すことで対応を行います。
主にログファイルや不要領域の増加によってディスク使用率が肥大化している場合や、バックグラウンド処理の頻発によってハードウェアリソースの負荷が高くなっている場合に有効です。
アプリケーションに手を加えることなくデータの肥大化の予防、リソースの効率化が行えます。

2. サーバ/PostgreSQL のパラメータチューニング

CPU 使用率や I/O 利用状況が高い場合は、原因や状況に応じてパラメータの設定を見直すことで対応を行います。
共有メモリの割り当て、チェックポイントなどの設定を見直すことでキャッシュ利用率向上や I/O 負荷の軽減を行うことができ、アプリケーションに手を加えることなく性能を改善します。
また、データベースのトランザクション設定を変更することで、ロックの待ち時間を減少させる事ができる可能性があります。

3. SQL チューニングやデータベース設計の見直し

スロークエリや実行負荷の高い SQL が存在する場合は SQL チューニングによって対応を行います。
アクセス情報や I/O 利用率、ログ、統計情報などから性能低下の原因となっている SQL を見つけ出し、実行計画を確認した上でアプリケーションで最低限必要となる適切な結果セットの取得、結合やインデックスの設定を行うことで SQL のレスポンスや I/O 負荷を改善させることができます。
また、必要に応じてデータベースの再設計を行うことも考慮する必要があります。

4. アプリケーションの修正

パラメータ設定や SQL チューニングで問題が改善しきれない場合、アプリケーションの修正によって対応を行います。
コミット処理を適切なタイミングで区切ったり、データベースへの問い合わせを必要最小限に押さえることで、ロック時間やロングトランザクションの削減が出来ます。
またアプリケーションの機能を部分的にデータベースサーバに搭載することで、ネットワーク帯域を確保することも考慮する必要があります。

5. ハードウェアの増強

ハードウェアを増強することで、性能のボトルネックとなっているリソース不足を解消させる対応を行います。
ただし、性能低下の根本的な解決とは言えない場合は一時的な対策にしかならず、将来的なデータの増加やサービスの展開を考えると恒久的な対策とは言い切れないため注意が必要です。そのため、パラメータやアプリケーションを修正し、根本的にハードウェアリソースがボトルネックとなっていると判断した上でこの対応を行うことを推奨します。

表 4.7: サーバから取得した情報に対する対策方針

確認された問題	確認項目	想定される要因	対応方針
CPU 使用率が規定を超えている (user が高い)	sar コマンドの結果	SQL 実行負荷、実行回数に問題	【3】, 【4】, 【5】
CPU 使用率が規定を超えている (iowait と system が高い)	sar コマンドの結果	ディスクアクセスの頻発などから I/O 待ちが発生している	【4】, 【5】
	sar コマンドの結果	大きい VACUUM 処理が発生している	【1】
CPU 使用率が規定を超えている (iowait が高く user, system が低い)	sar コマンドの結果	メモリ不足でスワップが発生している	【2】, 【5】
	sar コマンドの結果	ディスクの性能に問題がある	【5】
スワップ回数が規定を超えている	vmstat コマンドの結果	メモリが不足している	【2】, 【5】
I/O ビジー率、もしくは待ち個数が規定を超えている	sar コマンドの結果, iostat コマンドの結果	スロークエリが存在する	【3】
	sar コマンドの結果, iostat コマンドの結果	I/O の処理に問題がある	【3】, 【4】
	sar コマンドの結果, iostat コマンドの結果	CPU の性能がボトルネックになっている	【5】
ネットワーク帯域使用率が規定を超えている	sar コマンドの結果	クライアントと通信するデータ量が大きい	【3】, 【4】
	sar コマンドの結果	クライアントとのラウンドトリップ数が多い	【3】, 【4】
ディスク使用率が規定を超えている	df コマンドの結果, du コマンドの結果	不要領域が肥大化している	【1】, 【2】
	df コマンドの結果, du コマンドの結果	ログファイルが肥大化している	【1】, 【2】
	df コマンドの結果, du コマンドの結果	想定よりも必要とするディスク容量が多い	【5】

表 4.8: PostgreSQL から取得した情報の分析

確認された問題	確認項目	想定される要因	対応方針
スロークエリが存在する クエリ応答時間が規定を超えている	・log_min_duration_statement(ログ) ・pg_stat_statements の結果	SQL チューニングの必要がある インデックスが機能していない	【3】
ロングトランザクションが存在	・pg_stat_activity ビューの内容 ・プロセスのタイムアウト時間設定	DB の不要なプロセスが残存している	【2】, 【4】
	・pg_stat_activity ビューの内容 ・pg_locks ビューの内容	デッドロックが発生している	【4】
ロック待ち時間が長いクエリが存在する	・トランザクションレベル ・pg_stat_activity ビューの内容 ・pg_locks ビューの内容	デッドロックが発生している トランザクションの分離レベルが適切ではない	【2】
キャッシュヒット率が閾値を下回る	・pg_stat_database ビューの内容 ・pg_statio_all_tables ビューの内容 ・pg_statio_all_indexes ビューの内容 ・shared_buffers の設定値・SQL 毎の実行時間	メモリ/キャッシュサイズが不足している SQL の実行計画が適切な参照を行っていない	【2】, 【3】
max_connections に接続数が近い	・max_connections の設定値と接続あたりのリソース設定	想定よりも同時接続数が多い	【2】, 【5】
インデックス利用状況にばらつきがある	・pg_stats ビューの correlation の値 ・pg_stat_all_indexes ビューの内容	SQL チューニングの必要がある	【3】
オブジェクトサイズが肥大化している	・pg_database_size 関数の結果 ・pg_total_relation_size 関数の結果 ・VACUUM 実施間隔 ・アプリケーションのログ機能と その実行頻度や出力内容	データの挿入、更新が頻繁に行われているため データや不要領域が肥大化している	【1】, 【2】, 【4】
	・ログファイルのローテーション設定	ログファイルの管理が適切に行われていない	【1】, 【2】

4. まとめ

PostgreSQL の性能低下に対する対応は問題の原因やコスト、期限などの要因によりケースバイケースとなります。そのため本項の内容や 14.3 監視ケーススタディを参照し、状況に応じて適切な対処を行うことが重要になります。

4.4.3. シングル構成での監視のまとめ

本節では業務システムにて PostgreSQL を稼働させる際にとられる一般的な監視について説明を行いました。本節に記載した内容を要約すると以下の表のようになります。

表 4.9: シングル構成での監視

監視	頻度	切り分け時に確認すべきポイント	対処方針検討時のポイント
サーバ死活監視	数秒間隔	OS のコマンドを利用した正常動作を確認する	OS やスイッチなどの設定も含めた対処を検討する
PostgreSQL 死活監視		PostgreSQL のコマンドやシステムビューによる正常動作を確認する	PostgreSQL のログの内容も確認し、問題を整理して検討する
サーバ性能監視	数分間隔	OS のコマンドを利用した性能情報を分析する	運用やアプリケーションの特徴も含めた対処を検討する
PostgreSQL 性能監視		PostgreSQL の関数やシステムビューによる性能情報を分析する	

5. 共有ストレージ

HA クラスタには、共有ストレージ方式とストレージレプリケーション方式（ミラーリング）の 2 つの方式があります。本章では、共有ストレージ方式における可用性、バックアップ、監視について記述します。

5.1. 前提とする構成

共有ストレージ方式は、アクティブサーバとスタンバイサーバの 2 台の PostgreSQL サーバと、SAN や NAS、DAS などで接続された共有ストレージで構成されます。このことで、PostgreSQL の可用性を高めることができます。一般的な構成を図 5.1 に示します。

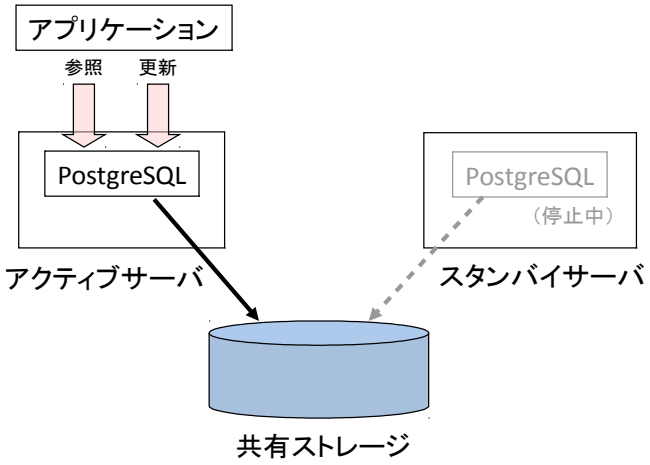


図 5.1: 共有ストレージ構成

共有ストレージ方式とは、その名のとおり、2 台の PostgreSQL サーバ間でデータを共有するものです。
通常運用時は、共有ストレージはアクティブサーバ側にマウントし、アクティブサーバで DB サービスを提供します。一方、スタンバイサーバはスタンバイ状態で待機しています。つまり、アクティブサーバが稼動している間、スタンバイサーバは共有ストレージにアクセスしてはいけないため、使用不可になります。
データの格納領域は唯一なので、データの整合性を気にする必要がありません。また、他の可用性構成に見られるオーバーヘッドを回避でき、データの同期性は高められます。その反面、共有ストレージに障害が起きた場合、ストレージの種類によっては単一障害点(SPOF)となりうるため、シングル構成と同様に、バックアップを取得しておくことや、コンポーネントが全て冗長化された単一障害点(SPOF)とならないストレージを利用するといったデータ保全の仕組みを備えておくことが必要です。
本構成は、代表的な可用性構成の中では、昔からある一般的なシステム構成であるといえます。

5.2. 可用性

障害が発生したとき、障害の場所によって復旧の仕方が異なります。共有ストレージに故障が発生した場合は、代替するストレージが準備されていないため、サービスを停止したうえで、ディスクの交換やデータの復旧などを実施して復旧する必要があります。一方、PostgreSQL サーバに障害が発生した場合は、次節より記述します。

5.2.1. 障害時のサービス継続性

本構成で障害が発生した場合のサービス継続性は、障害が発生する場所が、スタンバイサーバかアクティブサーバによって、障害時の復旧方法が異なります。なお、障害時において、コミット済みのデータは PostgreSQL の持つ耐障害性（WAL によるクラッシュリカバリ）によりデータは保全されますが、コミットされていないトランザクションは失われます。
まず、スタンバイサーバに障害が発生した場合は、図 5.2 のようにアクティブサーバに影響を及ぼさないため、提供しているサービスには影響はありません。

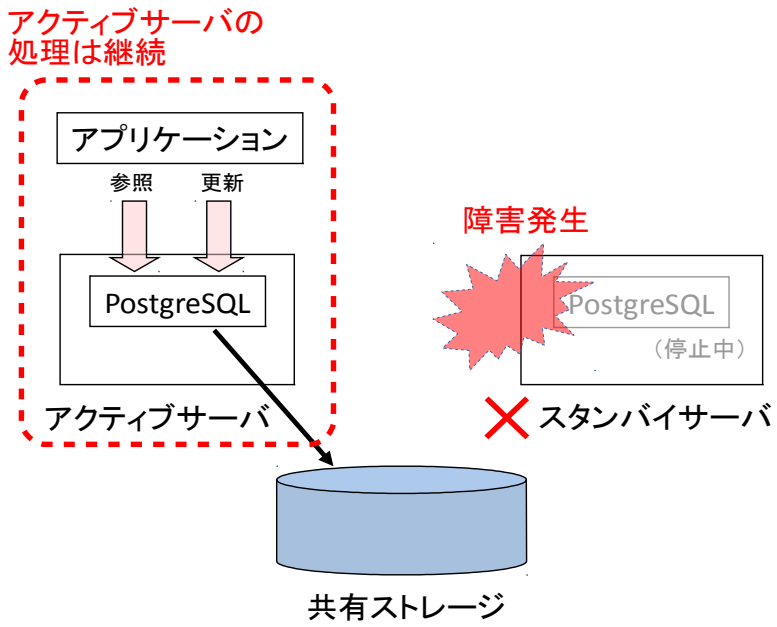


図 5.2: スタンバイサーバ障害時の挙動

次に、アクティブサーバに障害が発生した場合は、まずアクティブサーバを切り離し、スタンバイサーバに共有ストレージをマウントし、スタンバイサーバを新アクティブサーバに昇格させサービス継続させます。昇格には通常の PostgreSQL サーバ起動コマンドで行います。（図 5.3）
なお、HA クラスタソフトウェアなどの各種ツールを使用しない場合は、サーバやサービスの障害検知やフェイルオーバといった切り替えを全て手動で行う必要があるため、サービスの停止が長期化します。

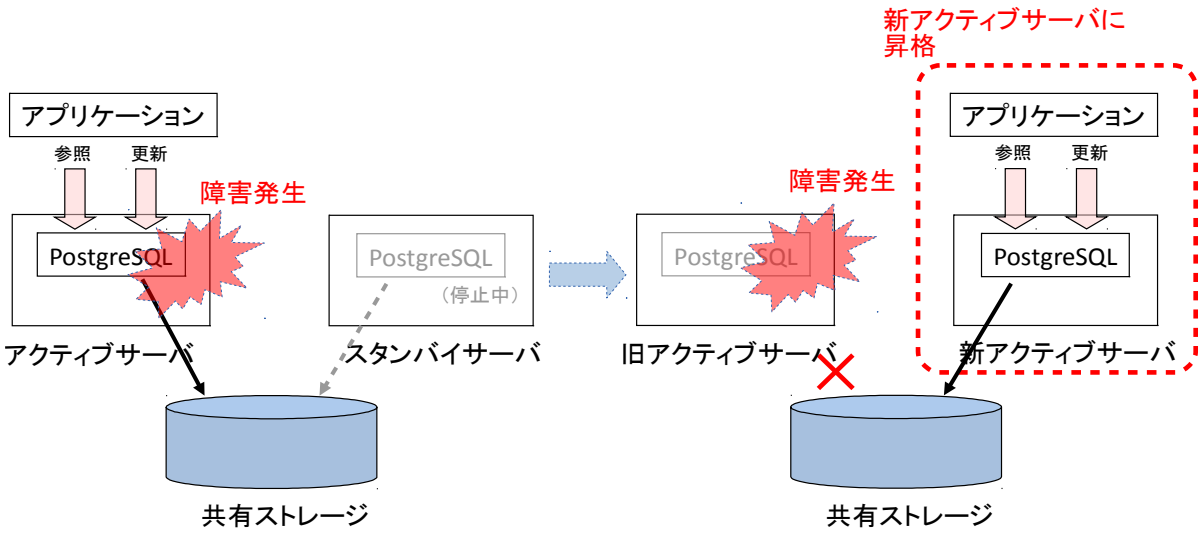


図 5.3: アクティブサーバ障害時の挙動

基幹システム等、可用性の高いサービスレベルを要求するシステムの場合は、自動化を行う必要が出てきます。 自動化を行うには、一般的に HA クラスタソフトウェアを使用します。代表的な OSS の HA クラスタソフトウェアとして、Heartbeat、Corosync（クラスタ制御）、Pacemaker（リソース制御）などがあげられます。クラスタ制御とリソース制御のソフトウェアは組み合わせて使用します。各々、次のような機能を有します。

・クラスタ制御は、ノード間で定期的に送受信するハートビート信号を使って、ノードの状態を把握、管理します
・リソース制御は、クラスタ上で動作するリソース(サービス、IP アドレス、データへのアクセスなど)の状態を監視、また、配置を調整して最適化します
・仮想 IP を利用できるようになるため、上位サービス側に影響を及ぼしにくくなります。ただ、切り替え時間がかかるため、セッションは切断される可能性はあります
これらを組み合わせた構成で障害が発生すると、図 5.4 のようにフェイルオーバします。
各ソフトウェアの詳細は、マニュアルをご確認ください。

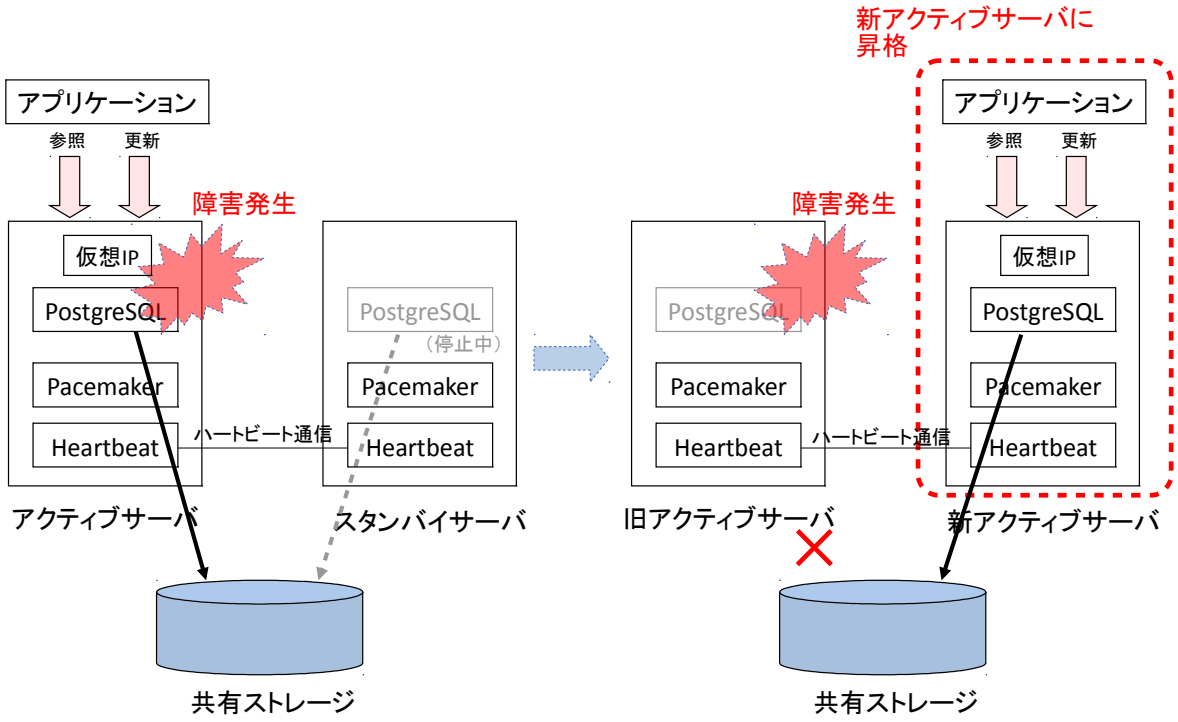


図 5.4: アクティブサーバ障害時の自動切換え

本構成におけるフェイルオーバに関する注意点として、共有ストレージが NFS サーバの場合、NFS サーバへの遅延（非同期）書きこみが信頼性に関する問題を引き起こす可能性があるため、NFS ファイルシステムを（キャッシュ無しで）同期マウントし、これを防止してください。

また、HA クラスタソフトウェアを使用する場合には、相互ノードの死活監視をネットワークを介して行っているため、このネットワークが切断されると各ノードが相手に障害があったと判断してしまい、複数のノードがアクティブサーバになってしまうことがあります。このことをスプリットブレインといいます。
このスプリットブレインを発生させないために、死活監視を行うハートビート通信用のネットワークを専用線にしたり、多重化するなどして、信頼性を高めておく必要があります。また、実際発生してしまった場合にフェンシングする機能として STONITH があります。STONITH を動かすためには、対応するハードウェア制御ボードが必要になります。

5.2.2. 障害復旧時の運用性

元の状態に戻す（フェイルバック）には、障害が発生する場所が、スタンバイサーバかアクティブサーバによって復旧方法が異なります。

スタンバイサーバに障害が発生した場合の復旧は、障害を取り除いた後、スタンバイサーバとしてシステムを起動しておきます（スタンバイ状態）。HA クラスタソフトウェアでクラスタ構成を組んでいる場合は、クラスタサービスを再起動することで、HA クラスタソフトウェアのノード情報が読み込まれスタンバイサーバとして組み込まれます。（図 5.1）
このとき、提供しているサービスには何ら影響を与えません。

アクティブサーバに障害が発生した場合の復旧は、まず障害を取り除いた後、旧アクティブサーバを新スタンバイサーバとしてシステムを起動します。続いて、再度手動で新スタンバイサーバをアクティブサーバにフェイルオーバするこ

とで本来の状態に復旧します。(図 5.5)

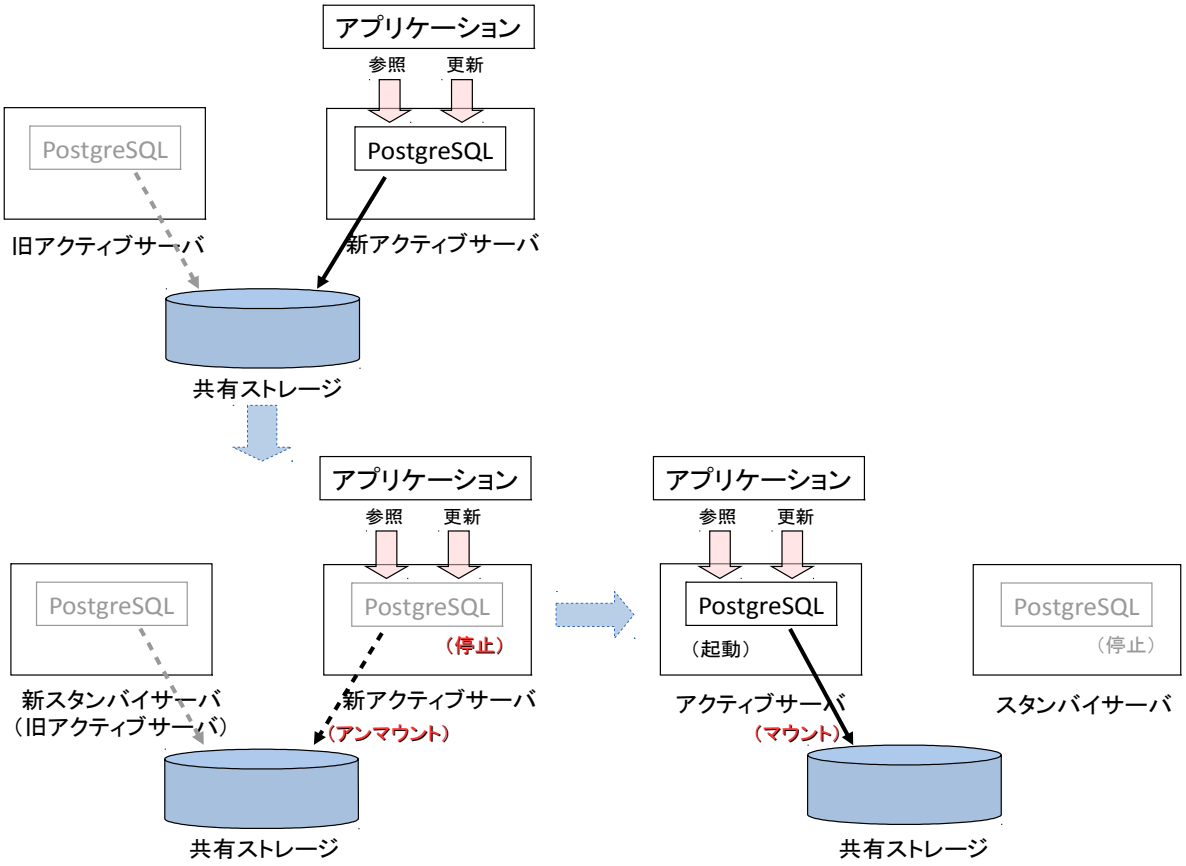


図 5.5: 手動フェイルバック

HA クラスタソフトウェアでクラスタ構成を組んでいる場合は、同じく障害を取り除いた後、旧アクティブサーバのクラスタサービスを起動して新スタンバイサーバとしてクラスタに組み込みます。続いて、リソースの手動移動(Pacemaker の場合は crm コマンド)を行うことで、本来の状態に復旧できます。(図 5.6)

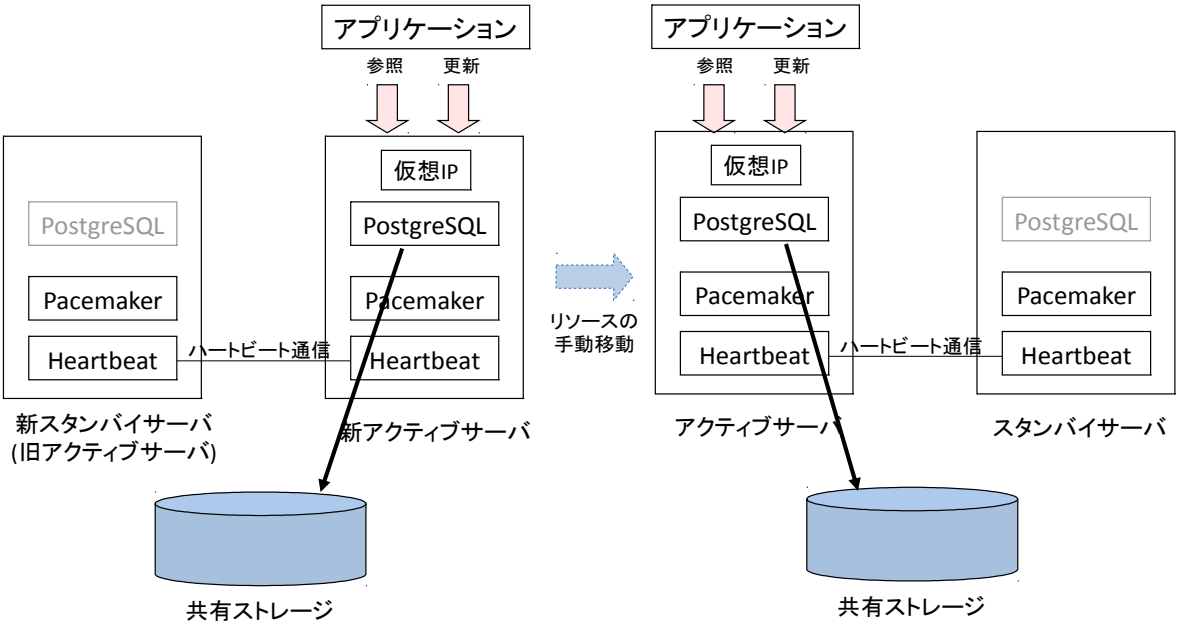


図 5.6: 自動フェイルバック

アクティブサーバの障害復旧において、元の構成に戻すための時間はサービスが停止するため、注意が必要です。

5.3. バックアップ

バックアップ／リカバリの観点からは、本構成は基本的にはシングル構成と同等とみなすことができます。そのため、バックアップ／リカバリ方式もシングル構成における各種方式と同じ方式を用いることができます。

共有ストレージ構成の特徴としては、可用性、信頼性を高めるために単一障害点(SPOF)のない高信頼性、高機能のストレージが用いられるケースが多いことです。そのような高機能なストレージを使用した場合、ストレージローカルコピーを利用してバックアップを取得することができます。

5.4. 監視

5.4.1. 共有ストレージに対する監視内容の変更

本章の構成において監視を行うにあたっては、追加したスタンバイサーバを監視する必要があります。

5.4.2. 方式の違いによる監視内容の変更点

本章の構成による監視内容は既に記載したサーバの監視から変更点はありません。シングル構成の内容をご確認ください。

監視の観点からは、本構成は以下のようにみなすことができます。

- アクティブサーバ・・・シングル構成と同等
- スタンバイサーバ・・・シングル構成と同等（PostgreSQL 自身の監視を除く）

このため、基本的にはシングル構成における監視方式を適用できます。なおスタンバイサーバについてはデータベースサービスを稼動させていないため、PostgreSQL に対する監視設定は除くなどの考慮が必要となります。また、HA クラスタソフトウェアを使用している場合は、HA クラスタソフトウェアにてデータベースサービスの状態監視が行われます。このため、監視設定のうち死活監視の部分については HA クラスタソフトウェア側で行うことができます。

5.4.3. 共有ストレージでの監視のまとめ

本章の構成では新たに追加したスタンバイサーバを監視する必要があります。
データベースサービスの状態監視は HA クラスタソフトウェア側で行うようにします。

6. ストレージレプリケーション（DRBD）

本章では、HA クラスタのうち、ストレージレプリケーション方式（ミラーリング）における可用性、バックアップ、監視について記述します。

6.1. 前提とする構成

ストレージレプリケーション方式は、5 章の共有ストレージを複数ストレージ間のレプリケーションによるデータ同期に置き換えた方式です。ストレージレプリケーションはハードウェア（ストレージ装置）の機能で実現する場合と、ソフトウェアで実現する場合があります。ソフトウェアによるストレージレプリケーションは、共有ストレージよりも安価に実現できることが多く、各社から商用製品が出ています。OSS では DRBD がよく用いられます。

ここでは DRBD[5]を利用することを前提とし、一般的な構成を図 6.1 に示します。

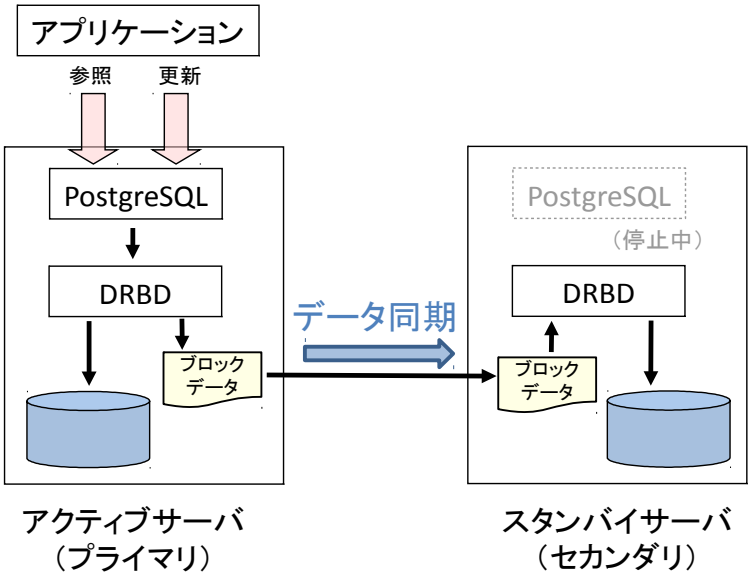


図 6.1: ストレージレプリケーション(DRBD)構成

DRBD は論理ディスクレベルでレプリケーション（同期）します。

通常の運用において、参照/更新ともアクティブサーバのみでサービスを提供します。例外として、排他制御のあるファイルシステム（ex. GFS、OCFS2 など）を使うと、2 台ともアクティブサーバとして稼動させることが可能となり、負荷分散を行うことができます。

アクティブサーバは、受け取ったデータをアクティブサーバ上のディスクへの書き込み、および、スタンバイサーバに更新されたブロックデータを送信します。一方、スタンバイサーバは、アクティブサーバから送られたデータを書き込みます。スタンバイサーバ上のディスクは、前述以外の外部からのアクセスは一切拒否されます。

DRBD は同期の種類として、完全同期モードと非同期モードの 2 種類あります。

①完全同期モード

アクティブサーバ、スタンバイサーバ両方の書き込みを保証するモードで、完全同期モードがデフォルト設定となります。通常、この完全同期モードを使うことが推奨されています。両サーバの書き込みを保証しているため、非同期モードと比較するとオーバヘッドが大きくなります。

②非同期モード

アクティブサーバへの書き込みのみを保証しているモードです。非同期モードは、BCP 対策などスタンバイサーバを遠隔地に置いている場合に使われることが多いです。完全同期モードと比較して高速ですが、アクティブサーバ障害

5 http://www.drbd.org/ja/

時にデータ消失の可能性があるため、最新データの保障が必要のないシステムにおいて利用してください。
ここでは、完全同期モードの場合について記述します。

本構成の注意点として、DRBD のデータ同期の速度を設定しないと、通常のアプリケーション処理に影響が出てくる可能性があります。具体的には、ネットワークに余裕があっても想定した処理速度が出ないことや、バックグラウンド同期でネットワークを切迫させてアプリケーションの処理速度が低下してしまうことが挙げられます。

6.2. 可用性

6.2.1. 障害時のサービス継続性

本構成で障害が発生した場合のサービス継続性は 、障害が発生する場所が、スタンバイサーバかアクティブサーバによって、障害時の復旧方法が異なります。なお、障害時において、コミット済みのデータは PostgreSQL の持つ耐障害性（WAL によるクラッシュリカバリ）によりデータは保全されますが、コミットされていないトランザクションは失われます。
まず、スタンバイサーバに障害が発生した場合は、DRBD によるスタンバイサーバとのコネクションが切断され、データ同期を停止します。その際、提供しているサービスは数秒接続ができなくなる可能性がありますが、自動的に復帰します。（図 6.2）

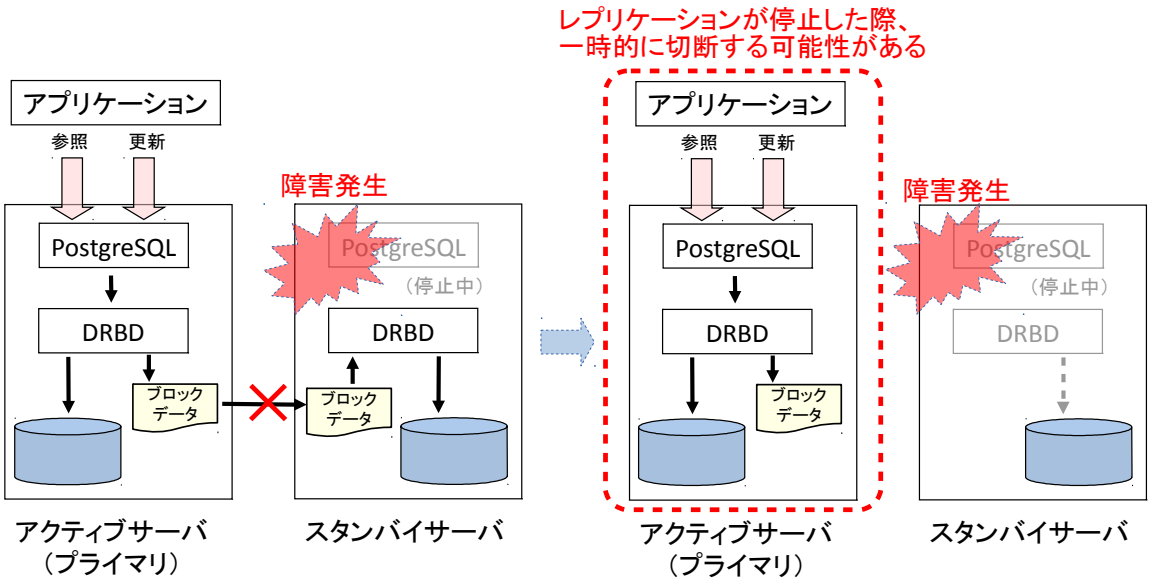


図 6.2: スタンバイサーバ障害時の挙動

次に、アクティブサーバに障害が発生した場合は、スタンバイサーバの障害と同様に、DRBD によるスタンバイサーバとのコネクションが切れ、データ同期を停止します。アクティブサーバを切り離した後、スタンバイサーバを新アクティブサーバに昇格させサービスを継続させます。昇格には、新アクティブサーバの DRBD を primary に切り替えて、DRBD 領域をマウントした後、PostgreSQL サーバを起動します。起動には通常の PostgreSQL サーバ起動コマンドで行います。（図 6.3）
なお、HA クラスタソフトウェアなどの各種ツールを使用しない場合は、サーバやサービスの障害検知やフェイルオーバといった切り替えを全て手動で行う必要があるため、サービスの停止は長期化します。

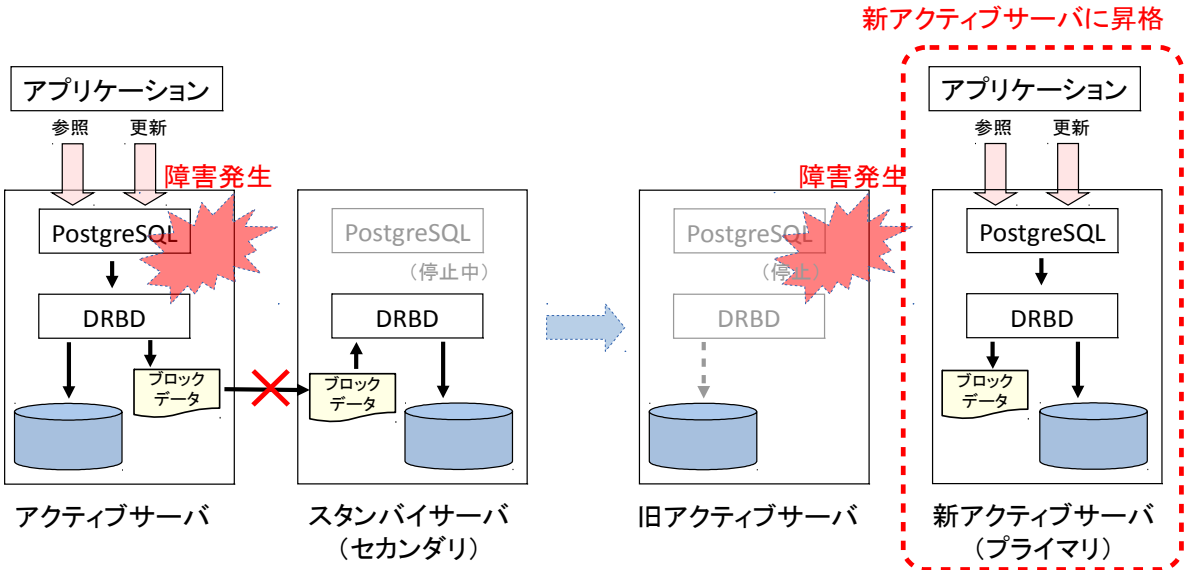


図 6.3: アクティブサーバ障害時の挙動

要求されるサービスレベルによって自動化が必要になりますが、自動化を行う場合の HA クラスタソフトウェアの説明は、5 章で行ったため割愛します。HA クラスタソフトウェアと組み合わせた構成で障害が発生すると、図 6.4 のように自動的にフェイルオーバします。

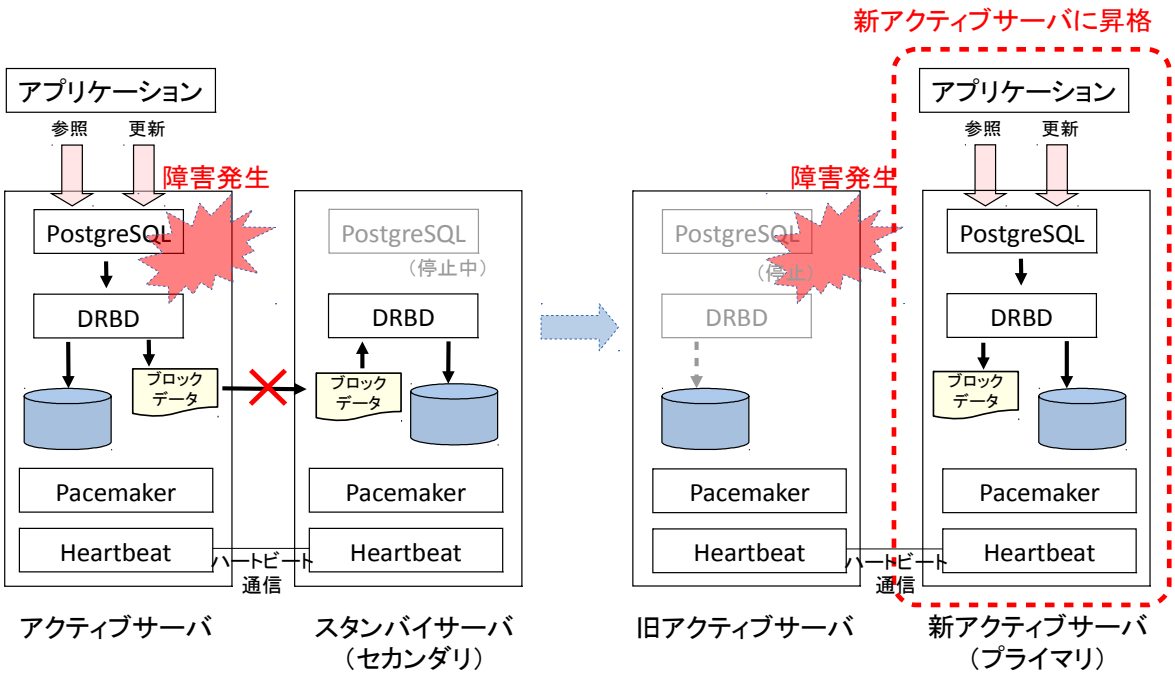


図 6.4: アクティブサーバ障害時の自動切換え

本構成におけるフェイルオーバの注意点として、5 章と同様に、HA クラスタソフトウェアを使用する場合には、スプリットブレインを発生させないために、死活監視を行うハートビート通信用のネットワークを多重化するなどして、信頼性を高めておく必要があります。

6.2.2. 障害復旧時の運用性

元の状態に戻す（フェイルバック）には、スタンバイサーバ障害、アクティブサーバ障害のいずれの場合も、発生した障害を取り除き、DRBD プロセスを起動します。プロセス起動すると、自動的にノード間の接続が再確立され、停止中に更新された変更内容が全て同期されます。（図 6.5、図 6.6）

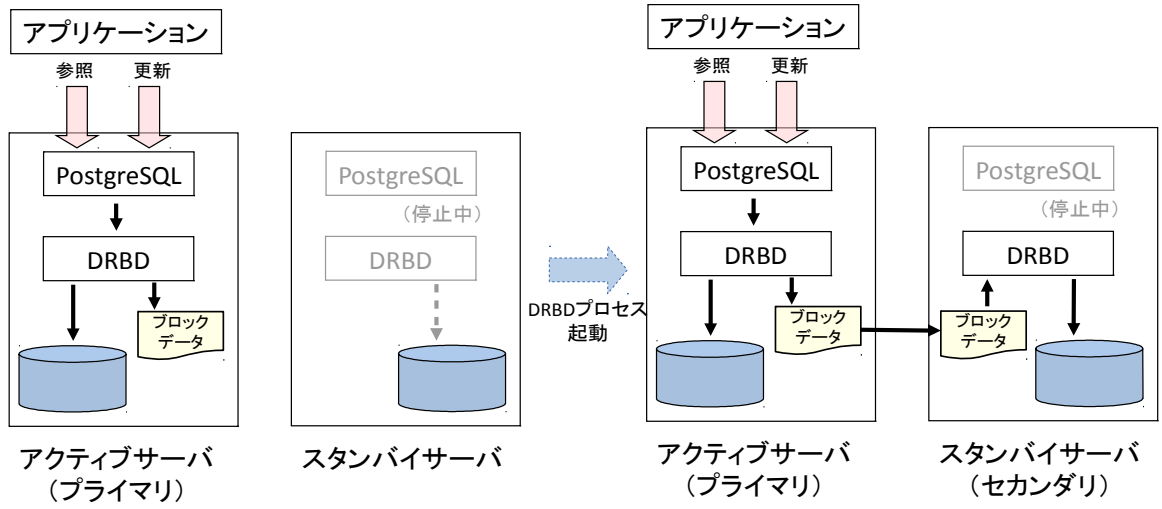


図 6.5: スタンバイサーバ障害時の復旧

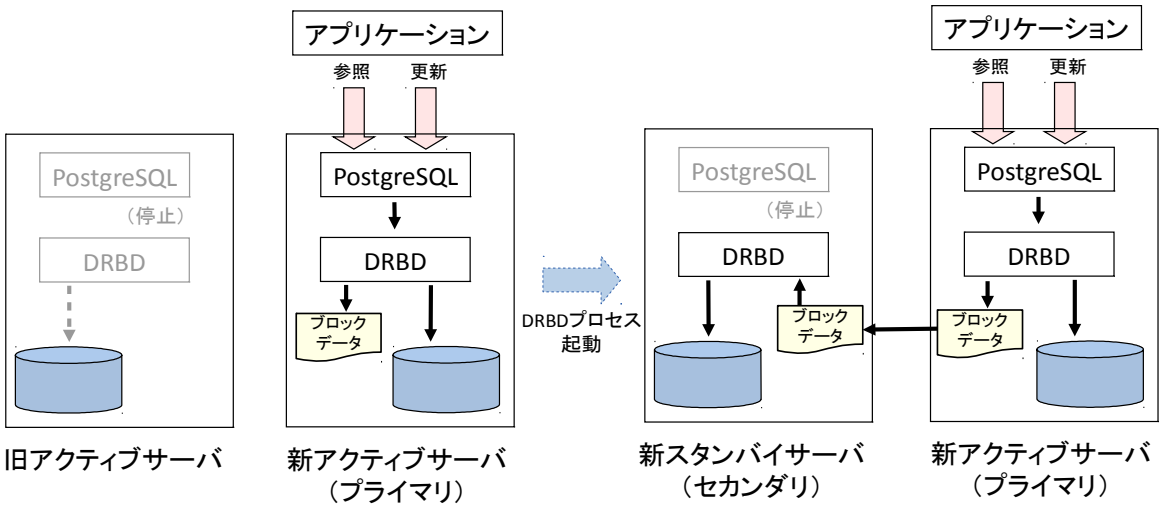


図 6.6: アクティブサーバ障害時の復旧

アクティブサーバ障害の復旧の場合は、図 6.6 にあるようにアクティブサーバとスタンバイサーバが本来の状態と逆になっているため、本来の状態に戻さないといけません。そのために、新アクティブサーバを drbdadm secondary でスタンバイサーバに降格させた後、新スタンバイサーバを drbdadm primary でアクティブサーバとして昇格させます。これらの手続きをすることで本来の状態に復旧できます。（図 6.7）

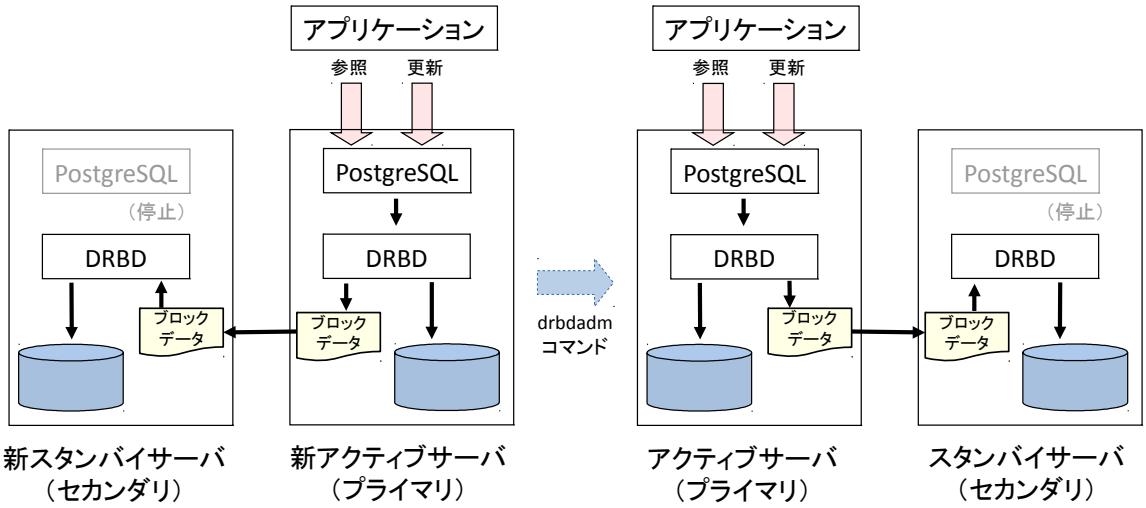


図 6.7: アクティブサーバ障害時の復旧で本来の状態にフェイルバック

障害復旧するに当たり、元の構成に戻すための時間はサービスが停止するため、注意が必要です。
本構成におけるフェイルバックに関する注意点として、ハードディスクを交換した場合には、再度 drbdadm create-md コマンドでメタデータを作成した後、再接続する必要があります。

6.3. バックアップ

本構成のバックアップ／リカバリ方式には、シングル構成における各種方式と同じ方式を用いることができます。

6.4. 監視

6.4.1. ストレージレプリケーション（DRBD）に対する監視内容の変更

本章の構成において監視を行うにあたっては、追加した DRBD が稼働するサーバと DRBD のプロセス、ログを監視する必要があります。

6.4.2. 方式の違いによる監視内容の変更点

本章の構成による監視内容は既に記載したサーバの監視に加え、DRBD の機能が正しく動作しているかどうかを確認する死活監視の項目をあわせて追加する必要があります。性能監視についてはそれぞれのサーバは基本的にシングル構成と違いはありません。
DRBD の死活監視は以下の観点からの監視が必要です。実際の監視には DRBD が提供する drbdadm コマンドを使用することができます。

・レプリケーションの接続に問題がないか？
・レプリケーションされたストレージの状態に問題がないか？

(a) DRBD の接続状態の監視

drbdadm cstate コマンドにて接続状態を監視します。Connected となっているのが正常です。

```
# drbdadm cstate <resource>
Connected
```

(b) レプリケーションされたストレージの状態監視

drbdadm dstate コマンドにて接続状態を監視します。
自サーバ/相手サーバともに UpToDate となっているかを確認します。

```
# drbdadm dstate <resource>
UpToDate/UpToDate
```

6.4.3. ストレージレプリケーション（DRBD）での監視のまとめ

本章の構成では新たに DRBD のプロセス、ログを監視する必要があります。
DRBD の死活監視は DRBD が提供する drbdadm コマンドを使用することができます。

7. ストリーミングレプリケーション

本章では、PostgreSQL のストリーミングレプリケーション機能を使ったクラスタ構成における可用性、バックアップ、監視について記述します。

7.1. 前提とする構成

ストリーミングレプリケーションは PostgreSQL 9.0 から搭載された組み込みのデータ複製機能です。マスタサーバへの更新処理が記録された WAL レコードをスレーブサーバへ転送することでデータベースが複製されます。スレーブサーバに WAL レコードが転送され、更新結果が反映されるまでには若干の遅延があるものの、マスタサーバへの負荷影響も小さいレプリケーション方式です。

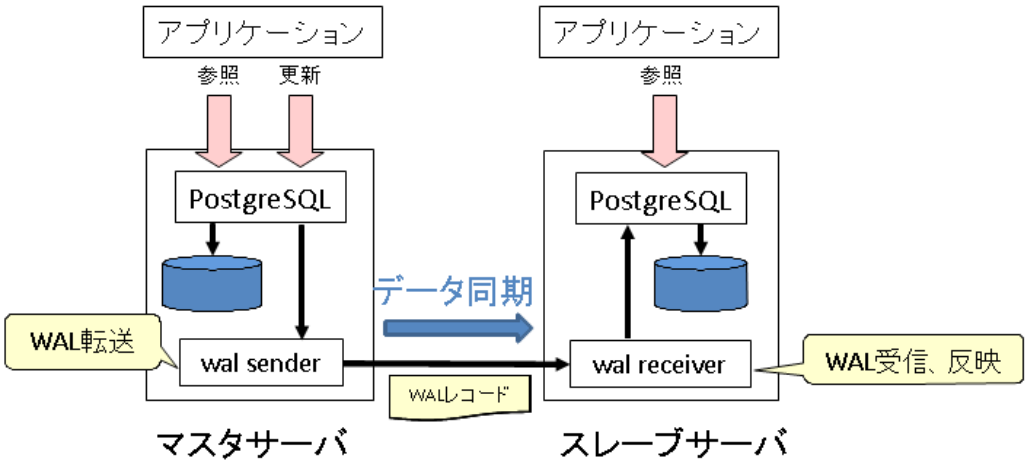


図 7.1: ストリーミングレプリケーション

PostgreSQL のストリーミングレプリケーションでは、参照/更新ともに可能なマスタサーバの役割は 1 台のみしか担えませんが、参照処理のみ可能なスレーブサーバは複数台構成することができます。そのため、参照処理の負荷をスレーブサーバに分散することで、システム全体の参照性能を向上させることができます。

ただし、本章の構成では発行する SQL を振り分けるミドルウェアが存在しないため、「更新処理ならマスタサーバに SQL を発行する」「参照処理ならスレーブサーバのどれか 1 台に SQL を発行する」という振り分けをアプリケーション側で制御する作り込みが必要となります。そこで、システムで発行される全ての SQL を対象に負荷分散するよりは、大量データを参照してデータ集計や分析を行うような高負荷バッチ処理をスレーブサーバで処理し、マスタサーバの負荷を軽減するようなアプローチが現実的と考えられます。

7.2. 可用性

7.2.1. 同期レプリケーションによる信頼性向上

PostgreSQL 9.1 以降ではストリーミングレプリケーションの設定で「非同期レプリケーション」と「同期レプリケーション」を選択できます。9.0 から搭載されている非同期レプリケーションは、WAL レコードがスレーブサーバに届いたことを確認せずにトランザクションのコミット成功をクライアントに返却するため、マスタサーバの障害時にコミット済のデータを消失する可能性がありました。同期レプリケーションは WAL レコードがスレーブサーバのディスク[6]に書き込まれたことを確認してからコミット成功をクライアントに返却するため、マスタサーバでコミット済のデータはスレーブサーバに到達していることが保証され、高い信頼性を求める用途への適用も可能です。

一方で、同期レプリケーションでは更新処理をコミットする度にスレーブサーバのディスクへの書き込み完了を待つオーバーヘッドが発生するため、非同期レプリケーションと比較して性能が劣化します。また、PostgreSQL の同期レプリケーションはスレーブサーバへの WAL 書き込みをタイムアウトなしで待つ仕様となっており、スレーブサー

6 PostgreSQL 9.2 から synchronous_commit パラメータを remote_write に設定することで、WAL レコードがスレーブサーバのメモリに書き込まれるまでを保証する同期モードも選択可能です。

バが停止した場合は再びスレーブサーバがレプリケーション可能な状態になるまでマスタサーバへの更新処理が停止してしまいます。そのため、サービス全体の継続性を損なう懸念があります。

このように PostgreSQL が提供する同期レプリケーションは「信頼性」と「性能」「サービス継続性」という面で一長一短の特性を持っているため、システム要件に応じてレプリケーション方式を選択する必要があります。

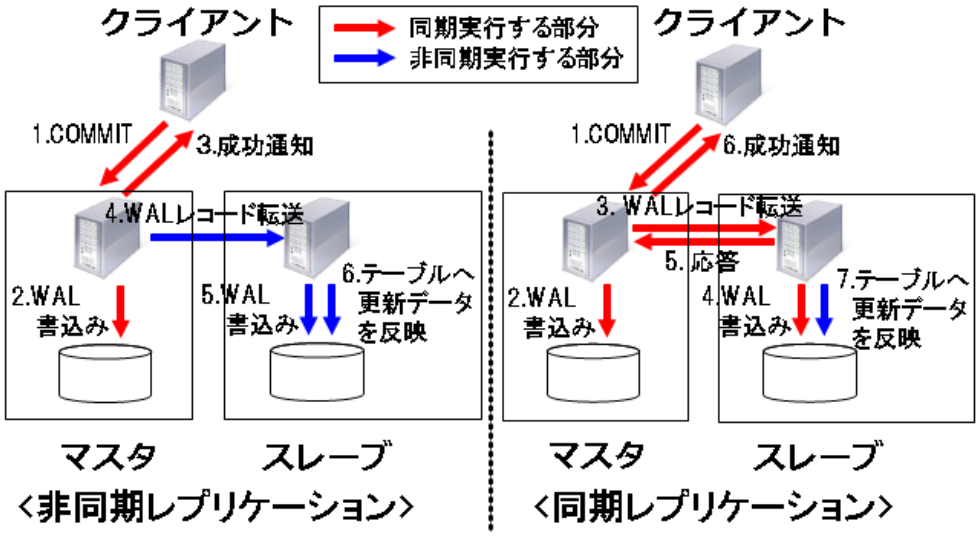


図 7.2: 同期レプリケーションと非同期レプリケーション

7.2.2. 障害時のサービス継続性

PostgreSQL のストリーミングレプリケーションを行っている環境で障害が発生した場合のサービス継続性は、障害が発生したサーバがマスタサーバか、同期レプリケーションのスレーブサーバか、非同期レプリケーションのスレーブサーバか、で異なります。

非同期レプリケーションのスレーブサーバに障害が発生した場合は、スレーブサーバへのデータ複製は停止するものの、マスタサーバに与える影響はなく、データベースサービスをそのまま継続することができます。

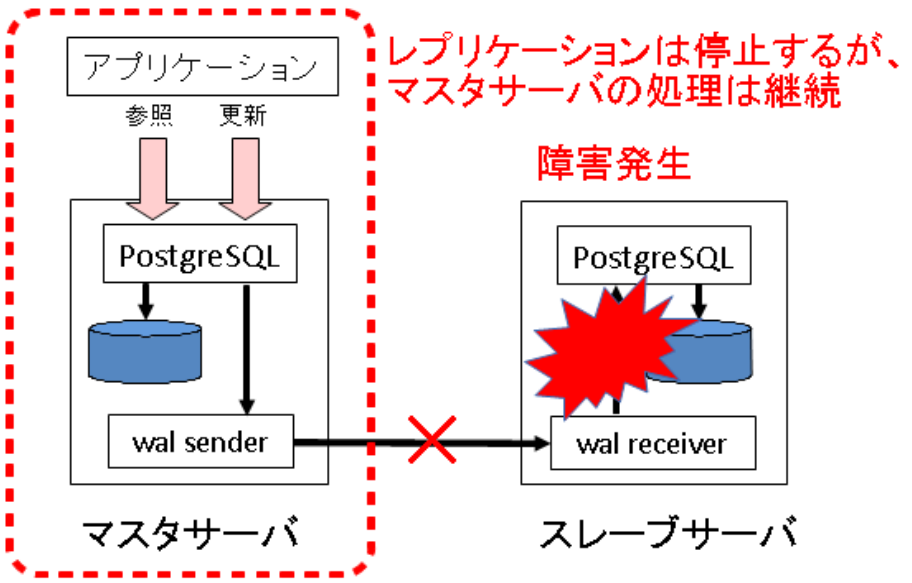


図 7.3:非同期レプリケーションのスレーブサーバ障害

同期レプリケーションのスレーブサーバに障害が発生した場合は、マスタサーバの更新処理が停止してしまうため、データベースサービスの一部が継続できない状態になります。このケースでは、マスタサーバ 1 台、スレーブサーバ

1 台で運用している環境ならマスタサーバの postgresql.conf の synchronous_standby_names に記載されている同期レプリケーションの設定を削除し、マスタサーバ 1 台での運用に切り替えると更新処理を受け付ける状態にできます。

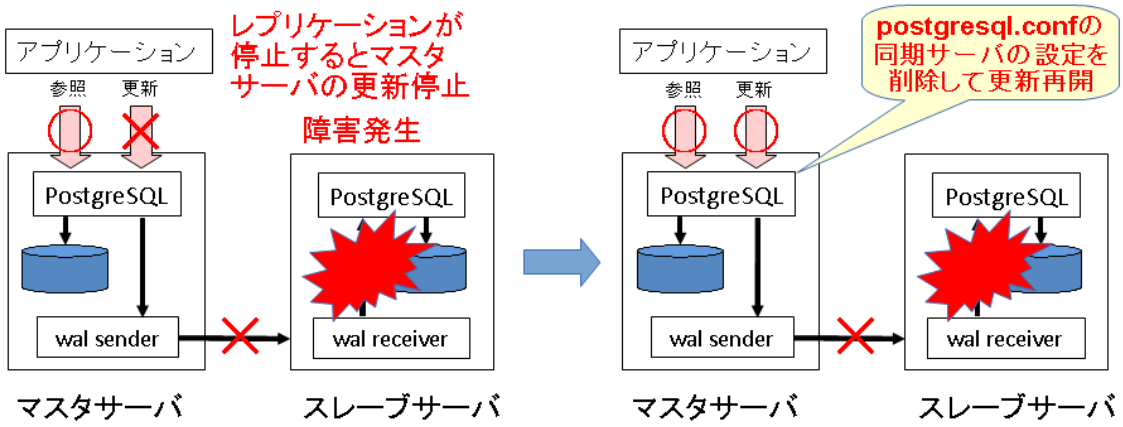


図 7.4: 同期レプリケーションのスレーブサーバ障害（スレーブ1台）

また、マスタサーバ 1 台、スレーブサーバ複数台で運用している環境ならマスタサーバの postgresql.conf の synchronous_standby_names にあらかじめ複数のスレーブサーバを列挙しておくことで PostgreSQL サーバが自動的に同期サーバを切り替えるため、特に手動で対処することなくサービスが継続できます。

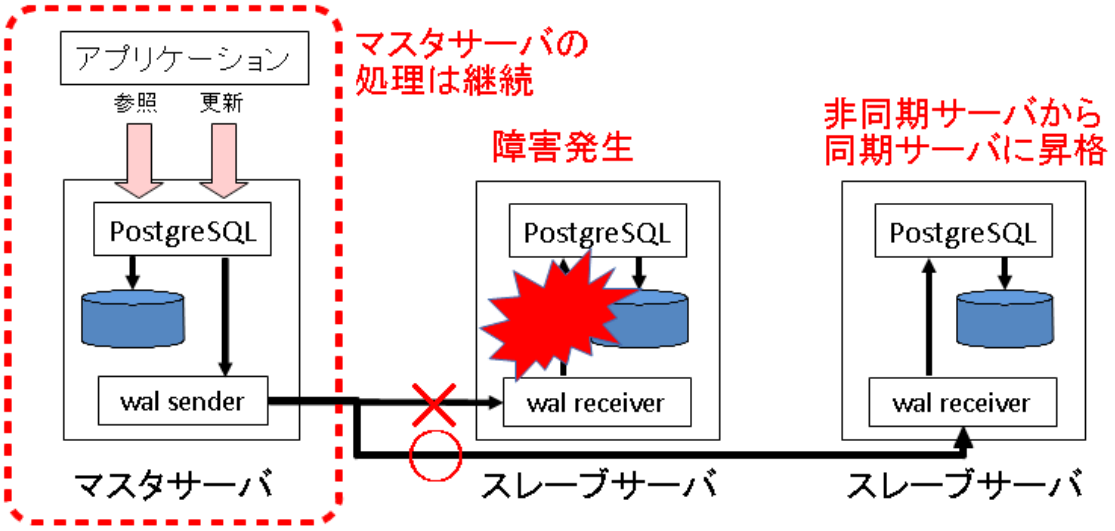


図 7.5: 同期レプリケーションのスレーブサーバ障害（スレーブ複数台）

マスタサーバに障害が発生した場合はスレーブサーバの1つを新しいマスタサーバに昇格させてマスタサーバの処理を引き継ぐフェイルオーバ操作を行うことで、データベースシステムのサービス継続性を向上させることができます。PostgreSQL 9.1 以降では pg_ctl promote という専用コマンドを使ってスレーブサーバをマスタサーバに昇格させることができます[7]。

7 PostgreSQL 9.0 では recovery.conf で設定した場所にトリガファイルを作成することで昇格させます。

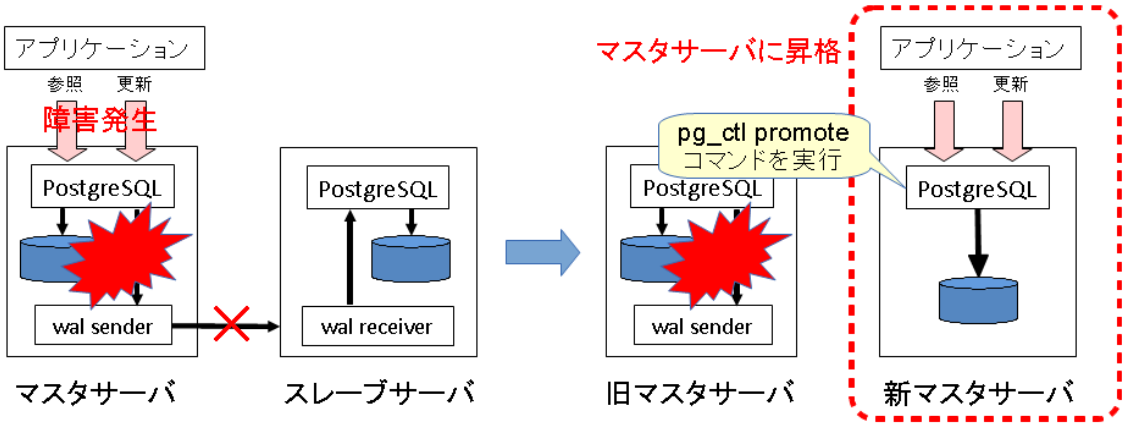


図 7.6: マスタサーバ障害時のフェイルオーバ

ただし、PostgreSQL にはマスタサーバの障害を検知して自動でフェイルオーバ操作を行う機能は搭載されていません。フェイルオーバ操作を手動で行うこともできますが、サービス停止時間の短縮が求められるシステムであれば Pacemaker 等の HA クラスタリングソフトウェアや pgpool-II といった専用ツールを組合せてフェイルオーバ操作を自動化することが推奨されます。

例えば Pacemaker と組み合わせると下図のように Pacemaker が PostgreSQL サーバの稼働状況を監視し、マスタサーバの異常停止を検知した際は pg_ctl promote コマンドでスレーブサーバを昇格させるとともにアプリケーションからアクセスに使用する仮想 IP を付け替えるフェイルオーバ操作を自動的に行うことができます。

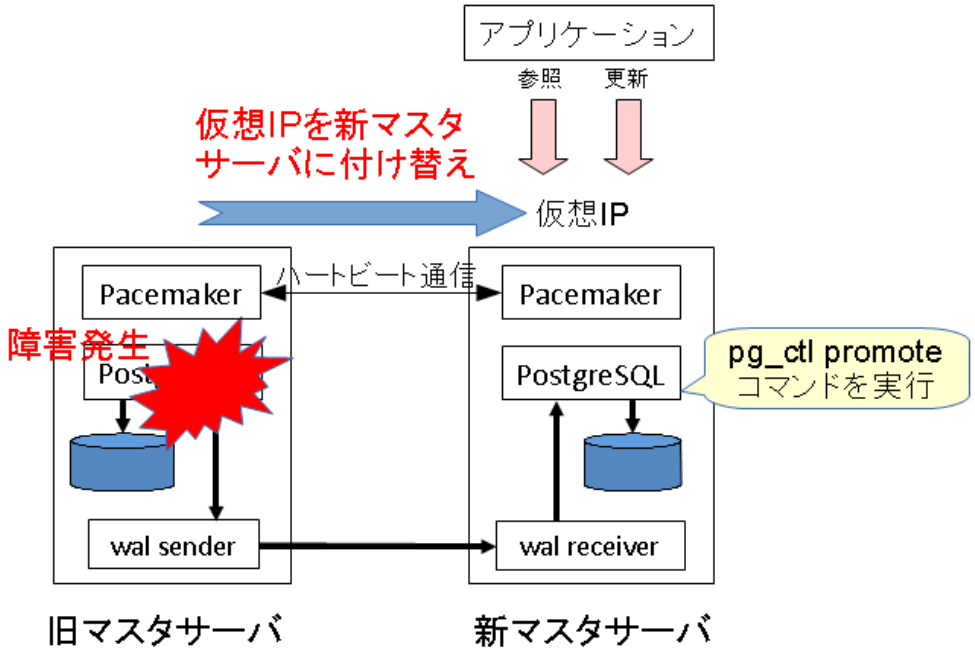


図 7.7: Pacemaker によるフェイルオーバ操作の自動化

7.2.3. 障害復旧時の運用性

PostgreSQL のストリーミングレプリケーションを行っている環境で障害発生すると、前章で述べたフェイルオーバ操作を行うことで、ひとまず停止したサーバを切り離した縮退状態でのサービス継続を図りますが、停止したサーバを復旧し、元の状態に戻す操作（フェイルバック）も障害が発生したサーバがマスタサーバか、スレーブサーバかで異なります。

障害が発生したスレーブサーバを復旧する場合、レプリケーション対象の WAL がマスタサーバ側に残っていれば、

そのまま再起動するだけでレプリケーションが再開します。一方、ディスク障害等でデータを失っている場合はマスタサーバから pg_basebackup 等でベースバックアップを取得して、スレーブサーバを再構築する作業が必要となります。なお、いずれのケースでもマスタサーバを停止することなく復旧作業が可能です。

マスタサーバに障害が発生した場合は、新しく昇格したマスタサーバに対するスレーブサーバとして復旧してからマスタサーバとスレーブサーバを入れ替える方法が一般的ですが、入れ替えを行うタイミングで全てのサーバを停止させる必要があることに注意が必要です。また、運用の作業自体も複雑なため、復旧作業は綿密な計画を立てて実施することが望まれます。

7.3. バックアップ

ストリーミングレプリケーションの利用方法の1つに、複製したデータベースをマスタデータベースのバックアップとして使用するという用途があります。本節では複製したデータベースをバックアップとして利用する際に考慮すべき点や、ストリーミングレプリケーションと一般的なバックアップを組み合わせた運用方法などについて記載します。

7.3.1. レプリケーションをバックアップとして利用する際の注意点

ストリーミングレプリケーションによって複製されたデータベース（スレーブデータベース）は、マスタデータベースと全く同じデータを保持しています。その特徴により、スレーブデータベースをマスタデータベースのバックアップとみなして運用することができます。ただし、ストリーミングレプリケーションには一般的なバックアップと異なる特徴があります。そのため、データの保全性を高めるためには考慮しなければならない点があります。

ストリーミングレプリケーションを利用したバックアップが一般的なバックアップと異なる大きな点は、バックアップデータであるスレーブデータベースが常に更新され続けているという点です。このため、例えばオペレーションミスにより誤ってマスタデータベース上のデータを削除してしまった場合、スレーブデータベース上のデータも削除されてしまいます。このようなケースでデータを復旧するためには、スレーブデータベースとは別にバックアップを取得しておく必要があります。

ストリーミングレプリケーション構成においては、マスタデータベースのバックアップを取得することも、スレーブデータベースのバックアップを取得することもできます。バージョン 9.2 からはスレーブデータベースでも pg_basebackup コマンドにより、オンラインでベースバックアップの取得が可能となりました。スレーブデータベースのオンラインバックアップについては次項で説明します。

7.3.2. レプリケーション（スレーブ）側のオンラインバックアップ

前述のとおり、PostgreSQL 9.2 からスレーブデータベースでも pg_basebackup によるオンラインバックアップが可能となりました。マスタデータベースの障害時に、スレーブデータベースから取得したバックアップを使用してマスタデータベースを復旧（リストアとリカバリ）することもできます。

スレーブデータベースにおいて pg_basebackup によりバックアップを取得するための設定例は以下のとおりです。

(1) postgresql.conf ファイル

```
listen_addresses = '*'
wal_level = hot_standby
archive_mode = on
archive_command = 'test ! -f /disk3/archive/%f && cp %p /disk3/archive/%f'
max_wal_senders = 2       # スレーブ DB の数 + 1
```

その他に
logging_collector = on
shared_buffers
checkpoint_segments
checkpoint_timeout
wal_keep_segments
などを、データベースサイズなどの要件に合わせて適宜変更します。

(2) pg_hba.conf ファイル

```
host    replication     repuser        172.16.3.101/32    md5   <= マスタサーバの IP アドレス。フェイルオーバー時に必要
host    replication     repuser        172.16.3.102/32    md5   <= スレーブサーバの IP アドレス。
host    replication     repuser        127.0.0.1/32       md5
```

上記は repuser というロールでレプリケーションやバックアップを行う設定になります。なお、repuser ロールには PostgreSQL の REPLICATION 権限が必要です。

pg_basebackup の実行例は以下のとおりです。

```
# su - postgres
$ pg_basebackup -h 127.0.0.1 -U repuser -D /backup/data --xlog
```

なお、本年度の活動として、ストリーミングレプリケーション構成においてマスタデータベースに障害が発生し、スレーブデータベースのバックアップを用いて復旧する手順を検証しました。その検証については 13.運用技術検証の「13.2.2 ストリーミングレプリケーション構成におけるバックアップ／リカバリ」に記載していますので、詳細についてはそちらをご参照ください。

7.4. 監視

7.4.1. ストリーミングレプリケーションに対する監視内容の変更

本章の構成において監視を行うにあたっては、追加した PostgreSQL が稼働するスレーブサーバとスレーブサーバの PostgreSQL のプロセス、ログを監視する必要があります。

7.4.2. 方式の違いによる監視内容の変更点

本章の構成による監視内容はシングルサーバ構成と比較し、ストリーミングレプリケーションによるレプリケーション機能に対する監視が追加されます。
死活監視でレプリケーション機能が正しく動作しているかを確認します。尚、性能監視についてはそれぞれのサーバで確認する内容は基本的にシングル構成と違いはありません。
レプリケーションの死活監視の観点は次の 3 つです。
・レプリケーションの接続に問題がないか？
・レプリケーションプロセスに異常がないか？
・レプリケーション状態に問題がないか？

上記 3 点は PostgreSQL が提供する pg_stat_replication ビューで確認することができます。マスタサーバ、スレーブサーバの間で適切なプロセスが起動してレプリケーションを行えていると、pg_stat_replication ビューから情報を取得することができます。下記のように、state が「streaming」となっていると正常なレプリケーション状態と判断できます。

```
> psql -x -p 5432 -c “SELECT * FROM pg_stat_replication” | grep state
state           | streaming
```

7.4.3. ストリーミングレプリケーションでの監視のまとめ

本章の構成では新たに追加した PostgreSQL が稼働するスレーブサーバとスレーブサーバの PostgreSQL のプロセス、ログを監視する必要があります。
レプリケーション状態の監視は PostgreSQL が提供する pg_stat_replication ビューで確認することができます。

8. pgpool-II（マスタスレーブモード）

本章では、pgpool-II のマスタスレーブモードを使ったクラスタ構成における可用性、バックアップ、監視について記述します。

8.1. 前提とする構成

pgpool-II（マスタスレーブモード）は、Slony-I や PostgreSQL のストリーミングレプリケーションといったマスタ/スレーブ型のレプリケーションソフトにレプリケーションをまかせる pgpool-II の動作モードです。PostgreSQL のストリーミングレプリケーションでは参照処理はマスタサーバ、スレーブサーバともに可能ですが、更新処理はマスタサーバでのみ可能ですので、通常はアプリケーション側で SQL の種類に応じた振り分け先を制御する必要があります。

pgpool-II を PostgreSQL のストリーミングレプリケーションと組合せてマスタスレーブモードで動作させる場合、アプリケーションから発行された SQL を pgpool-II が中継する際に SQL の種類を判別し、更新処理ならマスタ、参照処理ならスレーブへといった振り分けを行うロードバランサの役割を担うため、アプリケーション側で制御する必要がなくなります。

また、PostgreSQL のスレーブサーバが複数存在する構成では、参照処理の振り分け先を分散することも可能ですので、スレーブサーバを追加していくことで参照処理を負荷分散することができ、PostgreSQL サーバの台数をアプリケーション側で意識することなくシステム全体の参照性能を向上させることができます。一方で更新処理はマスタサーバ1台でしか実行できないため、更新性能を向上させることはできません。

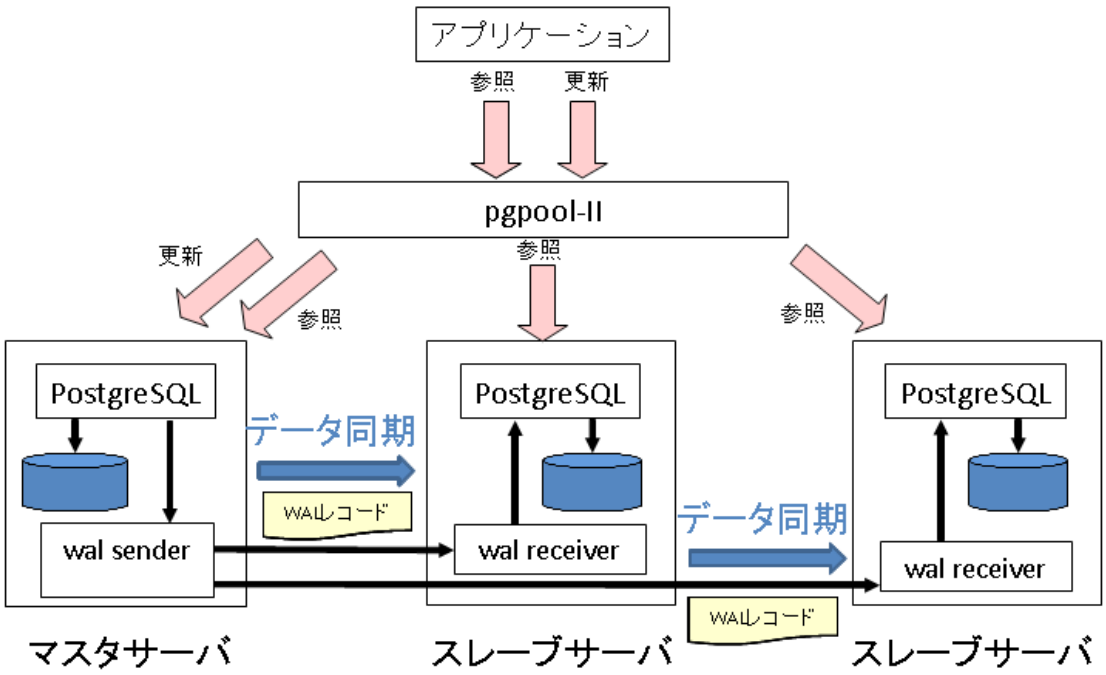


図 8.1: pgpool-II（マスタスレーブモード）

8.2. 可用性

8.2.1. 障害時のサービス継続性

本章の構成で障害が発生した場合のサービス継続性は、障害が発生したサーバがマスタサーバか、スレーブサーバか、pgpool-II が稼働するサーバかで異なります。

スレーブサーバに障害が発生した場合、マスタサーバから障害が発生したスレーブサーバへのデータ複製が停止するものの、データベースサービスを継続できる点は前章で解説したとおりですが、本章の構成では pgpool-II がスレーブサーバの障害を検知し、このサーバへの参照処理の振り分けを停止します。そのため、アプリケーションからは特に意識することなく障害サーバの切り離しが可能となります。

pgpool-II では PostgreSQL サーバの障害を即時に検知するわけではなく、PostgreSQL に対して通信を行った

タイミングか、ヘルスチェックと呼ばれる定期的チェックにより検知しています。そのため、PostgreSQL サーバの障害を pgpool-II が検知するまで一定の時間が必要[8]であり、その間 pgpool-II は停止している PostgreSQL サーバにも参照処理を振り分けてしまいます。つまり、スレーブサーバ障害時のサービス停止時間は、pgpool-II が PostgreSQL サーバの障害を検知する時間に依存する、と言えます。

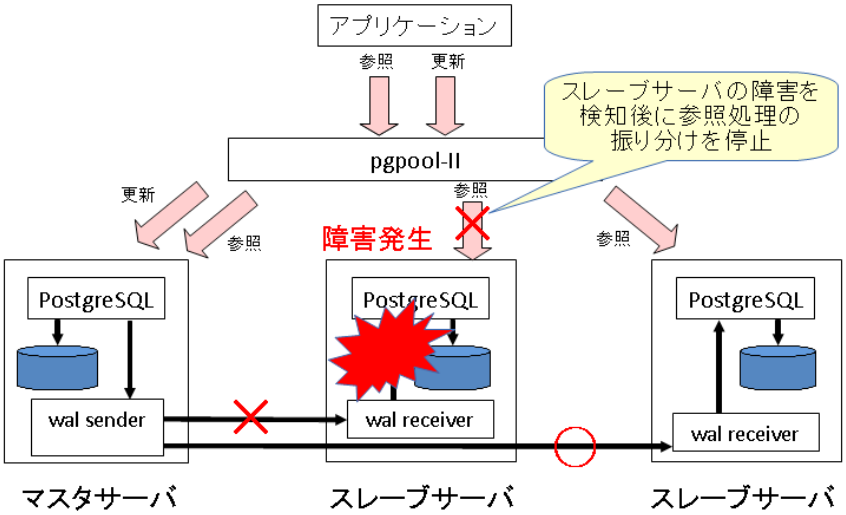


図 8.2: スレーブサーバ障害時の挙動

マスタサーバに障害が発生した場合は前章で解説したとおり、スレーブサーバの1つを新しいマスタサーバに昇格させてマスタサーバの処理を引き継ぐフェイルオーバ操作を行うことで、データベースシステムのサービス継続性を向上させることができます。また、本章の構成では pgpool-II によってフェイルオーバ操作を自動化できます。

pgpool-II はマスタサーバの障害を検知した場合、障害サーバへの SQL 振り分けを停止するとともに、スレーブサーバの1つに対して pg_ctl promote コマンド等を実行するフェイルオーバ用のスクリプト[9]を実行し、スレーブサーバをマスタサーバとして昇格させます。また、残りのスレーブサーバを昇格したマスタサーバへ再接続するスクリプト[10]を実行することでレプリケーションを再開した上で、昇格したマスタサーバへ更新処理の振り分けを開始してフェイルオーバ操作を完了します。

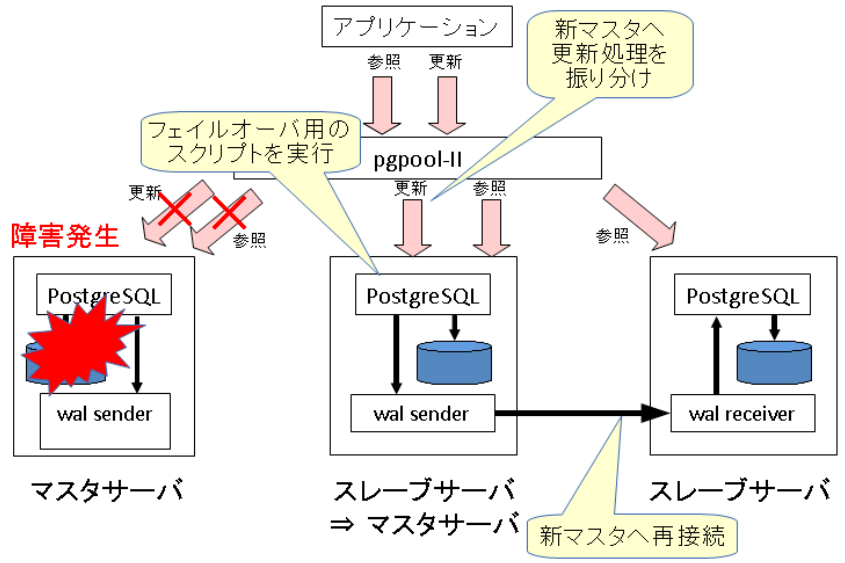


図 8.3: マスタサーバ障害時の挙動

8 pgpool-II 設定ファイル（pgpool.conf）において health_check_timeout 等のパラメータで設定します。
9 pgpool-II 設定ファイル（pgpool.conf）において failover_command のパラメータで設定します。
10 pgpool-II 設定ファイル（pgpool.conf）において follow_master_commnad のパラメータで設定します。

なお、マスタサーバの障害を pgpool-II が検知するまで一定の時間が必要であることに加え、スレーブサーバの昇格、新マスタへのレプリケーション再接続にも一定の時間がかかることから、マスタサーバ障害時のサービス停止時間は、スレーブサーバ障害時よりも長くなります。

pgpool-II が稼働するサーバに障害が発生した場合は、データベースシステムのサービス全体が停止した状態になることに注意が必要です。これは、本章の構成においてアプリケーションから発行した全ての SQL を中継する pgpool-II が単一障害点となるためです。pgpool-II が単一障害点となることを回避するには pgpool-II を冗長化する方式を採るのが一般的であり、これについては後述の章で解説します。

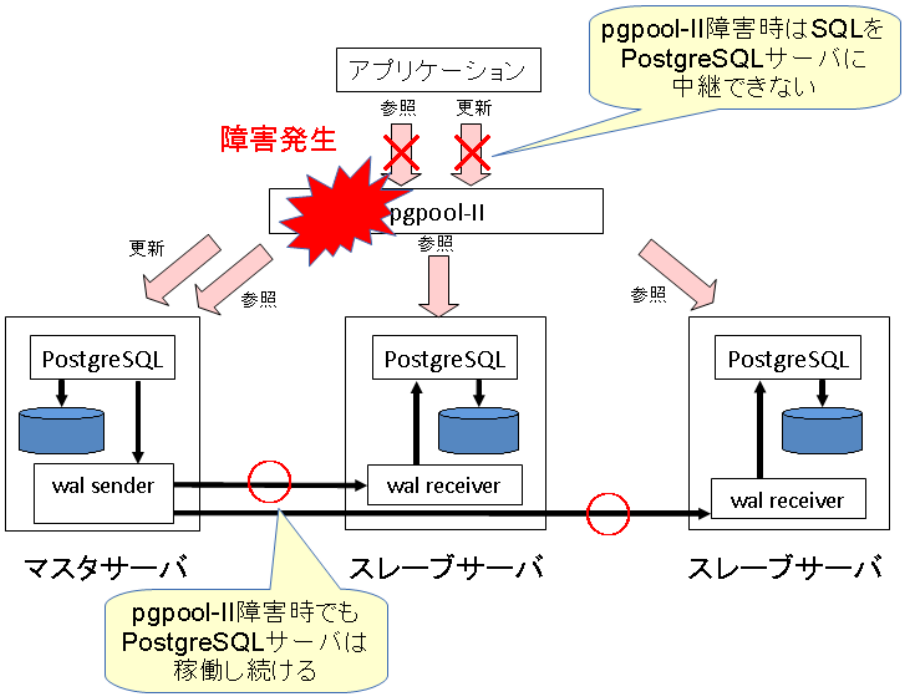


図 8.4: pgpool-II 障害時の挙動

8.2.2. 障害復旧時の運用性

本章の構成で障害が発生した PostgreSQL サーバを復旧する場合は、前章で解説したように PostgreSQL サーバをスレーブサーバとして復旧する手順と、pgpool-II のバックエンドサーバとして追加する手順が必要です。PostgreSQL サーバを復旧する手順は前章と同様で、pgpool-II のバックエンドサーバとして追加するには pgpool-II の PCP コマンド[11]である pcp_attach_node を利用します。また、pgpool-II には上記の手順を1つの操作で実行するオンラインリカバリ機能が用意されており[12]、PCP コマンドである pcp_recovery_node もしくは pgpool-II の管理ツールである pgpoolAdmin のリカバリボタンから呼び出すことができます。なお、上記のいずれの操作もデータベースサービスを停止することなく実行することができます。

一方、障害が発生した pgpool-II 稼働サーバは pgpool-II を再起動することで復旧できますが、プロセス障害等で異常終了した場合は pgpool.pid といった不要ファイルが残存していることがあるので注意が必要です。

8.3. バックアップ

本構成のバックアップ／リカバリ方式には、ストリーミングレプリケーション構成における各種方式と同じ方式を用いることができます。

なお本書では pgpool-II の設定ファイルなどのバックアップについては扱いません。

11 http://www.pgpool.net/docs/latest/pgpool-ja.html#pcp_command
12 http://www.pgpool.net/docs/latest/pgpool-ja.html#stream

8.4. 監視

8.4.1. pgpool-II（マスタスレーブモード）に対する監視内容の変更

本構成はシングルサーバ構成と比較すると、複数のデータベースサーバ間のデータ同期、個々のサーバの障害検知、障害発生時のフェイルオーバなどの機能が追加されています。そのため、それぞれの機能が正しく動作しているかどうかを確認する死活監視の項目もあわせて追加する必要があります。また性能監視については監視対象のサーバが増えた以外は、基本的にシングル構成と違いはありません。

8.4.2. 方式の違いによる監視内容の変更点

本構成では、データ同期は PostgreSQL 自身が持つストリーミングレプリケーションの機能によって行い、負荷分散、障害検知、障害発生時のフェイルオーバを pgpool-II の機能で行います。ストリーミングレプリケーションの監視については、前章で述べた内容と違いはありません。pgpool-II については、以下の点について監視内容の変更、または監視項目の追加を行う必要があります。

a) データベースサービスの死活監視

pgpool-II を利用する場合、アプリケーションは直接 PostgreSQL にアクセスするのではなく、pgpool-II を介してデータベースにアクセスすることになります。そのため、データベースサービスが正常に稼動しているか確認するための SQL 監視も、アクセス先を pgpool-II のエンドポイントに変更しておく必要があります。

b) pgpool-II 自身の死活監視

pgpool-II 自身の死活監視も、PostgreSQL の死活監視と同様に、以下の観点での監視を行います。

- pgpool-II が稼動しているサーバから ping 応答が得られるか
- pgpool-II のプロセスが稼動しているか
- pgpool-II のプロセスから正常応答が得られるか
- pgpool-II のログにエラーが出力されていないか

監視方法は、ログ監視以外は PostgreSQL を直接監視する場合と違いはありません。ログ監視については、pgpool-II ではフェイルオーバなどのイベントのログが 'LOG' のレベルで出力される点に注意が必要です。

c) PostgreSQL サーバの障害発生の検知

本構成では PostgreSQL サーバに障害が発生しても pgpool-II が自動的に障害サーバを切り離し、マスタサーバが不在になればスレーブサーバをマスタサーバに昇格させることでデータベースサービスの提供を継続します。しかしその後速やかに障害が発生したサーバを復旧させて冗長構成を回復させるためには、障害が発生したことを検知して通知を行うことが望ましいです。

障害の有無は個々の PostgreSQL サーバにクエリを発行することでも確認できますが、pgpool-II では pcp_node_info コマンドを使用することで、pgpool-II の管理下にある PostgreSQL サーバの状態を取得可能です。pcp_node_info コマンドを実行すると以下のような出力が得られます。

```
# pcp_node_info 10 127.0.0.1 9898 postgres password 0
192.168.0.1 5432 2 0.333333
```

出力は“<ホスト名> <ポート番号> <ステータス> <ロードバランスウェイト>”の形式で出力されます。ステータスの数値は、1 がノード稼働中で接続無し、2 がノード稼働中で接続有り、3 がノードダウンを表しています。

また、pgpool-II 経由で PostgreSQL に接続し、'show pool_nodes;' のクエリを発行することでも同様の情報を得ることができます。この場合の出力は以下のとおりです。こちらの方法では管理下のサーバを一度に確認でき、pcp_node_info で得られた情報に加え、各 PostgreSQL サーバの role の情報を得ることもできます。ただし、接続

可能な PostgreSQL サーバが残っていない場合にはセッション自体を確立できない点には注意が必要です。

```
# show pool_nodes;
 node_id |  hostname   | port | status | lb_weight |  role
---------+-------------+------+--------+-----------+---------
 0       | 192.168.0.1 | 5432 | 2      | 0.333333  | primary
 1       | 192.168.0.2 | 5432 | 2      | 0.333333  | standby
 2       | 192.168.0.3 | 5432 | 2      | 0.333333  | standby
```

d) PostgreSQL サーバの障害発生時のフェイルオーバの成否

pgpool-II は管理下の PostgreSQL に障害が発生した場合、自動的に事前に設定されたフェイルオーバ用のコマンドを実行します。フェイルオーバが成功すれば自動的にデータベースサービスが復旧されますが、障害原因によっては失敗することも考えられます。参照処理を負荷分散している場合、データベースの死活監視を参照系の SQL で行っているとプライマリサーバが存在しなくても参照クエリは正常に応答が返ってくるため、別途フェイルオーバの成否も監視しておくことが望ましいです。

pgpool-II の場合、フェイルオーバが実行されると、以下のようなログが出力されます。

```
2014-02-24 10:56:40 ERROR: pid 10890: connect_inet_domain_socket: getsockopt() detected error:
Connection refused
2014-02-24 10:56:40 ERROR: pid 10890: make_persistent_db_connection: connection to 192.168.0.1(5432)
failed
2014-02-24 10:56:40 ERROR: pid 10890: health check failed. 0 th host 192.168.0.1 at port 5432 is down
2014-02-24 10:56:40 LOG:   pid 10890: set 0 th backend down status
2014-02-24 10:56:40 LOG:   pid 10890: starting degeneration. shutdown host 192.168.0.1(5432)
2014-02-24 10:56:40 LOG:   pid 10890: Restart all children
2014-02-24 10:56:40 LOG:   pid 10890: execute command: /usr/local/bin/failover.sh 0 192.168.0.1 5432
/var/lib/pgsql/9.3/data 1 0 192.168.0.2 0 5432 /var/lib/pgsql/9.3/data
2014-02-24 10:56:40 LOG:   pid 10890: find_primary_node_repeatedly: waiting for finding a primary node
2014-02-24 10:56:41 LOG:   pid 10890: find_primary_node: primary node id is 1
2014-02-24 10:56:41 LOG:   pid 10890: failover: set new primary node: 1
2014-02-24 10:56:41 LOG:   pid 10890: failover: set new master node: 1
2014-02-24 10:56:41 LOG:   pid 10925: worker process received restart request
2014-02-24 10:56:41 LOG:   pid 10890: failover done. shutdown host 192.168.0.1(5432)
2014-02-24 10:56:42 LOG:   pid 10924: pcp child process received restart request
2014-02-24 10:56:42 LOG:   pid 10890: PCP child 10924 exits with status 256 in failover()
2014-02-24 10:56:42 LOG:   pid 10890: fork a new PCP child pid 11061 in failover()
2014-02-24 10:56:42 LOG:   pid 10890: worker child 10925 exits with status 256
2014-02-24 10:56:42 LOG:   pid 10890: fork a new worker child pid 11062
```

フェイルオーバが失敗しプライマリサーバが存在しなくなった場合は、以下のように、find_primary_node: の行が出力されず、failover: new primary node: の値が -1 となります。

```
2014-02-21 16:38:15 LOG:   pid 30707: execute command: /usr/local/bin/failover.sh 0 192.168.166.165 5432
/var/lib/pgsql/9.3/data 1 0 192.168.166.39 0 5432 /var/lib/pgsql/9.3/data
sh: /usr/local/bin/failover.sh: Permission denied
2014-02-21 16:38:15 LOG:   pid 30707: find_primary_node_repeatedly: waiting for finding a primary node
2014-02-21 16:38:25 LOG:   pid 30707: failover: set new primary node: -1
2014-02-21 16:38:25 LOG:   pid 30707: failover: set new master node: 1
```

8.4.3. pgpool-II（マスタスレーブモード）での監視のまとめ

本章の構成では新たに複数のデータベースサーバ間のデータ同期、個々のサーバの障害検知、障害発生時のフェイルオーバなどの機能が正しく動作しているか確認する死活監視もあわせて行う必要があります。

データ同期は PostgreSQL 自身が持つストリーミングレプリケーションの機能によって行い、負荷分散、障害検知、障害発生時のフェイルオーバは pgpool-II の機能で行います。ストリーミングレプリケーションの監視については、前章で述べた内容と違いはありません。pgpool-II については、以下の点について監視内容の変更、または監視項目の追加を行う必要があります。

a) データベースサービスの死活監視
b) pgpool-II 自身の死活監視
c) PostgreSQL サーバの障害発生の検知
d) PostgreSQL サーバの障害発生時のフェイルオーバの成否

9. pgpool-II（レプリケーションモード）

本章では、pgpool-II のレプリケーションモードを使ったクラスタ構成における可用性、バックアップ、監視について記述します。

9.1. 前提とする構成

pgpool-II（レプリケーションモード）は、pgpool-II が複数の PostgreSQL サーバ間のデータ同期を担う pgpool-II の動作モードです。pgpool-II をレプリケーションモードで動作させる場合、アプリケーションから発行された SQL を pgpool-II が中継する際に SQL の種類を判別し、更新処理なら全ての PostgreSQL サーバに同じ SQL を発行することでデータの同期を図ります。また、参照処理の場合はどれか1つの PostgreSQL サーバのみに SQL を発行します。

このモードではアプリケーション側で SQL 振り分けの制御をする必要がなく、スレーブサーバを追加していくことで参照処理を負荷分散することができるため、PostgreSQL サーバの台数を増やすことでシステム全体の参照性能を向上させることができます。

また、このモードはストリーミングレプリケーションとは違い、更新データをリアルタイムに同期しているため、どの PostgreSQL サーバで参照 SQL を実行しても同じ結果が得られるというメリットがある反面、OID やトランザクション ID の様な PostgreSQL サーバ毎に異なる値を使う更新 SQL[13]を扱えない制約がある点や PostgreSQL サーバの台数を増加させると更新処理の性能が低下していくことに注意が必要です。

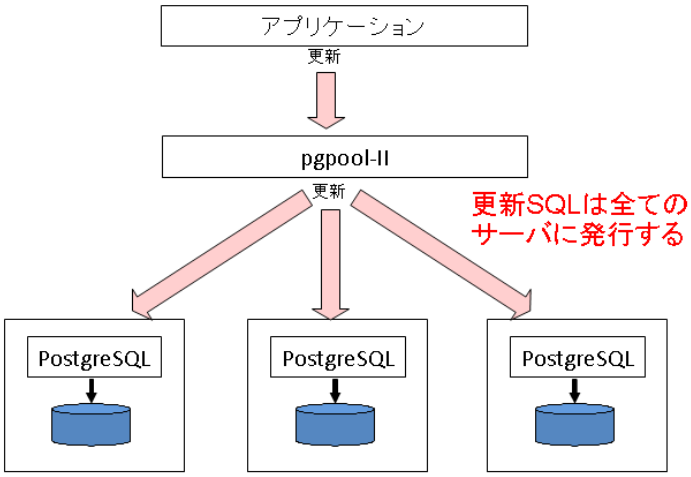


図 9.1: pgpool-II（レプリケーションモード）における更新系処理

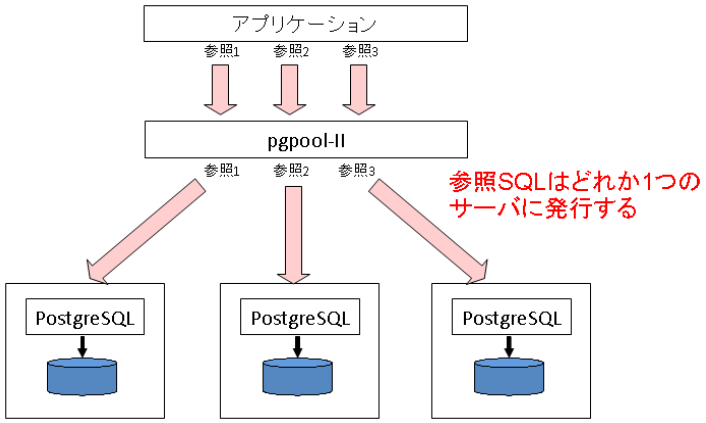


図 9.2: pgpool-II（レプリケーションモード）における参照系処理

13　例えば以下の様な SQL は実行できません。
SELECT oid FROM ...;　← OID を取得
UPDATE t1 SET ... WHERE oidcol = [上の SELECT 文で取ってきた OID 値]

9.2. 可用性

9.2.1. 障害時のサービス継続性

本章の構成では、マスタスレーブモードのようにバックエンドの PostgreSQL サーバにマスタサーバ、スレーブサーバという役割の区別がないため、どの PostgreSQL サーバに障害が発生した場合でも障害が発生した PostgreSQL サーバを pgpool-II が切り離す挙動となります。そのため、PostgreSQL サーバに障害が発生した場合のサービス停止時間は、pgpool-II が PostgreSQL サーバの障害を検知する時間に依存します。

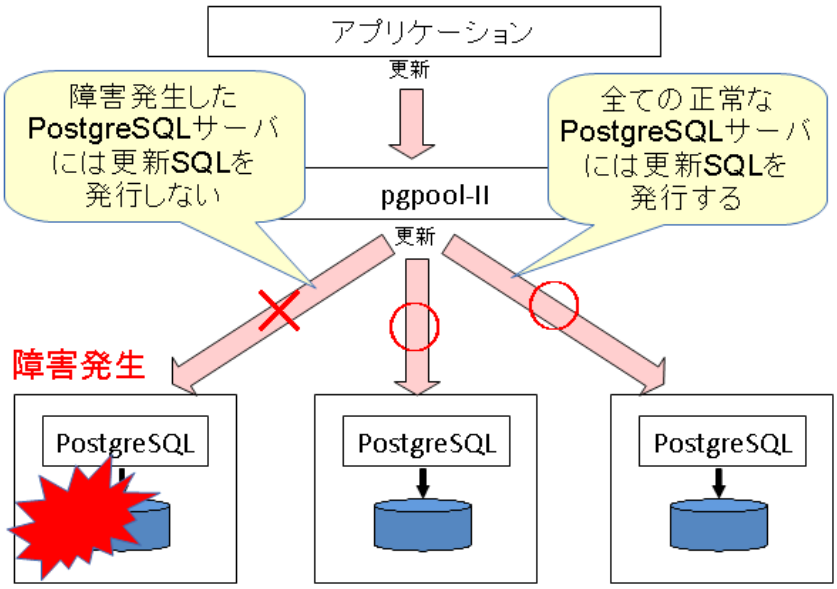


図 9.3: 障害時の更新系処理

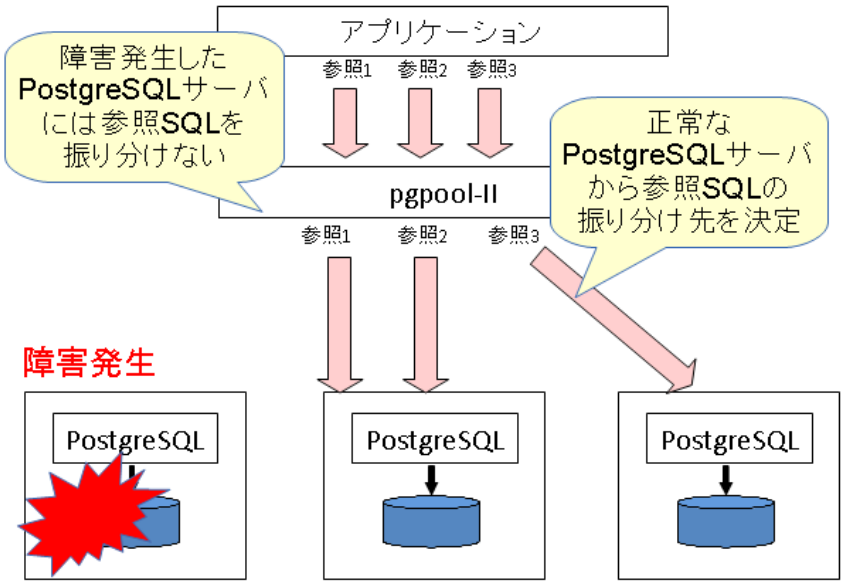


図 9.4: 障害時の参照系処理

なお、本章の構成ではアプリケーションから発行した全ての SQL を中継する pgpool-II が単一障害点となるため、pgpool-II が稼働するサーバに障害が発生した場合は、前章の構成と同様にデータベースシステムのサービス全体が停止した状態になります。pgpool-II が単一障害点となることを回避するには pgpool-II を冗長化する方式を採るのが一般的であり、これについては後述の章で解説します。

9.2.2. 障害復旧時の運用性

本章の構成で障害が発生した PostgreSQL サーバを復旧する場合は、pgpoo-II のオンラインリカバリ機能[14]を使い、障害で停止していたサーバのデータを再同期した上で復旧させる操作を行います。

pgpool-II のオンラインリカバリは、ファーストステージとセカンドステージと呼ばれる2段階で実行されます。ファーストステージはデータベースクラスタをコピーする操作が行われ、この間はデータベースサービスの運用を続けることが可能です。一方でセカンドステージはファーストステージの間に変更された部分を反映する処理ですが、データベースの整合性を保つために他の処理をブロックする必要があります。具体的には pgpool-II はセカンドステージに入る前に接続中のクライアントが全て接続を終了するまで待ち、新たな接続リクエストは全てブロックします。

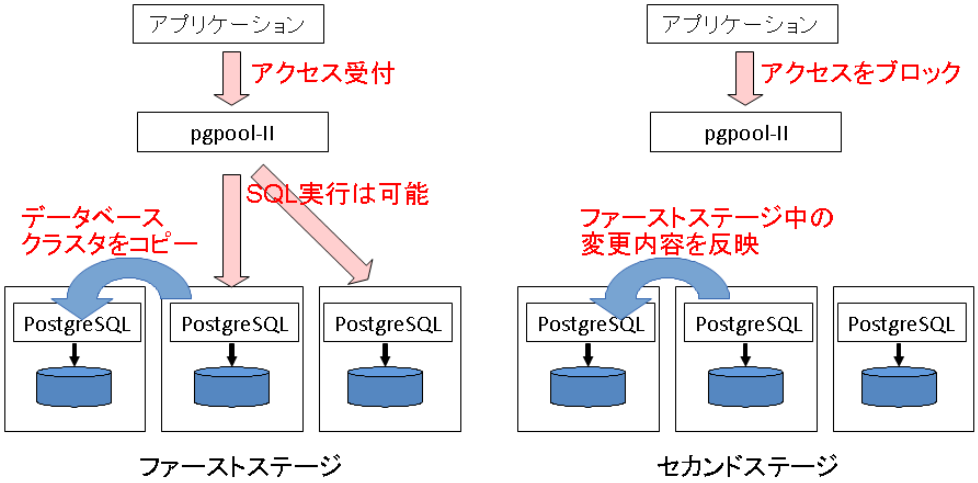


図 9.5: pgpool-II のオンラインリカバリ

つまり、本章の構成では障害が発生した PostgreSQL サーバを復旧する際にデータベースサービスの停止が発生します。また、サービス停止を伴うセカンドステージを早急に終了させるには他にクライアントからの接続がない状態にする必要があるため、復旧作業はトランザクションがほとんど発生しない時間帯に行う等、運用上の工夫が必要であると言えます。

一方、障害が発生した pgpool-II 稼働サーバは前章の構成と同様に pgpool-II を再起動することで復旧できます。プロセス障害等で異常終了した場合は pgpool.pid といった不要ファイルが残存していることがあるので注意が必要です。

9.3. バックアップ

本構成ではストリーミングレプリケーション構成と同様に、レプリケーションされたデータベースをバックアップとみなして運用することができます。ただし、やはりオペレーションミスによるデータ破壊への対策は必要です。個々のデータベースはシングル構成のデータベースと同じようにバックアップすることができますので、システムに適した方式でバックアップを取得することが推奨です。

なお本書では pgpool-II の設定ファイルなどのバックアップについては扱いません。

14 http://www.pgpool.net/docs/latest/pgpool-ja.html#online-recovery

9.4. 監視

9.4.1. pgpool-II（レプリケーションモード）に対する監視内容の変更

本構成はシングルサーバ構成と比較すると、複数のデータベースサーバ間のデータ同期、個々のサーバの障害検知、障害発生時のフェイルオーバなどの機能が追加されています。そのため、それぞれの機能が正しく動作しているかどうかを確認する死活監視の項目もあわせて追加する必要があります。また性能監視については監視対象のサーバが増えた以外は、基本的にシングル構成と違いはありません。

9.4.2. 方式の違いによる監視内容の変更点

本構成では、データ同期、負荷分散、障害検知、障害発生時のフェイルオーバを全て pgpool-II の機能で行います。上記のうち、負荷分散、障害検知については前章で述べた内容と違いはありません。
障害発生時のフェイルオーバについては、本方式ではマスタ/スレーブの区別が無いため、failover_command を指定する必要が無く、コマンドの成否を監視する必要が無いこと以外は前章と違いはありません。
データ同期に関しては注意が必要です。pgpool-II のレプリケーション機能を用いた場合、PostgreSQL サーバ間で連携してデータの同期を行うわけではなく、更新を伴う SQL を全てのサーバ上で等しく実行することで同じ状態を保っています。そのため、以下のようなケースではデータの不一致が生じる可能性があります。

- pgpool-II の制限事項に反する項目(乱数やトランザクション ID など)を含む SQL が発行された場合
- pgpool-II を介さずに直接個別のサーバに更新を伴う SQL が発行された場合
- データの同期が取れていないサーバを pgpool-II の管理下に追加/復帰した場合
- 一部のサーバで更新を伴う SQL の処理に失敗し、かつ pgpool-II による切り離しが行われなかった場合

このような状態が発生した場合の影響を最小限に抑えるために、pgpool-II が持つ以下の機能を活用することができます。

- replication_stop_on_mismatch
  - PostgreSQL サーバからの応答の種類が不一致になった場合に少数派となったサーバ側を切り離す
- failover_if_affected_tuples_mismatch
  - INSERT/UPDATE/DELETE に対する PostgreSQL サーバからの応答の結果行数が不一致になった場合に少数派となったサーバ側を切り離す

これらの機能を有効にした場合、何らかの理由で応答が一致しなくなったサーバは pgpool-II から切り離されて参照されなくなり、また前章で前述した pcp_node_info 等の結果を監視しておくことで異常を検知することが出来ます。

9.4.3. pgpool-II（レプリケーションモード）での監視のまとめ

本章の構成ではデータ同期、負荷分散、障害検知、障害発生時のフェイルオーバを全て pgpool-II の機能で行います。上記のうち、負荷分散、障害検知については前章で述べた内容と違いはありません。
障害発生時のフェイルオーバについては、コマンドの成否を監視する必要が無いこと以外は前章と違いはありません。
なお、データの不一致により適切な監視を行えなくならないよう、pgpool-II が持つ以下の機能を活用して、影響を最小限に抑えることができます。

- replication_stop_on_mismatch
- failover_if_affected_tuples_mismatch

10. Slony-I

本章では、Slony-I を用いた構成の可用性、バックアップ、監視について記述します。

10.1. 構成の概要

Slony-I[15]は PostgreSQL 専用の非同期方式・シングルマスタ・マルチスレーブ型のレプリケーションソフトウェアです。参照/更新がともに可能なマスタサーバが 1 台に対して、参照クエリのみを受け付けるスレーブサーバを複数台構成することができます。参照処理は全てのスレーブで受け付けることができるので、負荷分散によりシステム全体の参照性能を向上させることができます。

ただし、Slony-I 自体はクエリ振り分けの機能を備えてはいません。「更新クエリはマスタで処理し、参照クエリはスレーブで負荷分散する」という振り分け処理はアプリケーション側で作り込むか、あるいは pgpool-II などのツールと組み合わせることで実現する必要があります。

レプリケーションの仕組みは PostgreSQL のトリガ機能を利用しています。レプリケーション対象のテーブルにはトリガが作成されており、マスタサーバで更新が発生するとその内容がトリガ関数により Slony-I の管理テーブルに記録されます。この情報が定期的に Slony-I のデーモンプロセス slon によってスレーブサーバへと転送され、データベースに適用されることによりレプリケーションを実現しています(図 10.1)。なお、この適用はトランザクション単位で行われます。

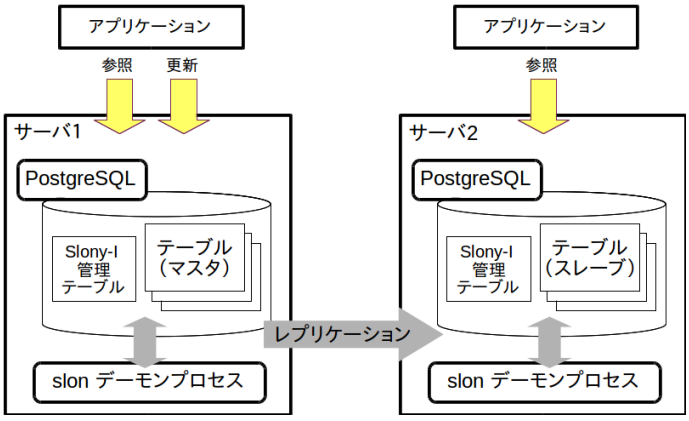


図 10.1: Slony-I 概要

10.1.1. 柔軟なレプリケーション構成

Slony-I はデータベースに含まれる全テーブルを無条件に複製するのではなく、任意のテーブルとシーケンスの集合をレプリケーションの対象にすることができます。この集合のことを「セット」と呼びます。また、レプリケーションに参加する PostgreSQL インスタンスの集合を「クラスタ」[16]、個々のデータベースを「ノード」[17]と呼びます。

1つのクラスタの中にセットは複数作成することが可能で、セット毎にレプリケーション元となるマスタノードと、その複製であるスレーブノードを指定することができます。更新が可能なのはマスタのみであり、スレーブは参照のみが可能です。セットに含まれない複製対象外のテーブルでは更新/参照ともに可能です。

このように、PostgreSQL 組み込みの機能であるストリーミングレプリケーションと比較して、より柔軟なレプリケーション構成を組めることが Slony-I の利点です。

この特徴を利用してテーブル単位の部分レプリケーション構成を組むことが可能です。図 10.2 にその例を示します。テーブル 1、テーブル 2 からなるセット 1 と、テーブル 3 からなるセット 2 があり、両セットともサーバ 1 がマスタノードです。部分レプリケーションはネットワーク負荷を抑える目的やセキュリティの観点から必要最低限のテーブルのみを複製したい場合に有用な構成です。マスタからスレーブへのレプリケーションだけではなく、スレーブからスレーブへのカスケーディングレプリケーションが可能です。これにより、マスタにかかる負荷をより低減させることができます。また、サーバ毎に異なるテーブル構成をとることができるので、マスタサーバ側には存在しない集計用の

15 http://slony.info/
16 PostgreSQL におけるデータベースクラスタとは区別されます。
17 データベースサーバマシンやデータベースクラスタとは無関係であることに注意してください。

テーブルをスレーブサーバ側で持つといった応用も考えられます。
また、マスタへの更新情報はバイナリではなくテキストの形式で記録しているため、バージョンの異なるPostgreSQL 間や、ハードウェアや OS の異なるシステム間でレプリケーションを行うことも可能です。

10.1.2. 制約および注意点

レプリケーションにトリガ機能を使用している関係から、Slony-I が対応しているクエリには制約があります。DDL コマンドによる変更、ラージオブジェクトの変更、およびユーザ/ロールの変更に関しては自動レプリケーションさせることはできません。これらのクエリは専用のコマンドを用いて実行するか、あるいは psql などを用いて手動で全ノードに対して実行する必要があります。
また、レプリケーション対象に含めるテーブルは全て主キーを持っている必要があります。もしレプリケーション対象としたいテーブルに主キーが存在しない場合には、事前にテーブルスキーマを変更して作成しておく必要があります。

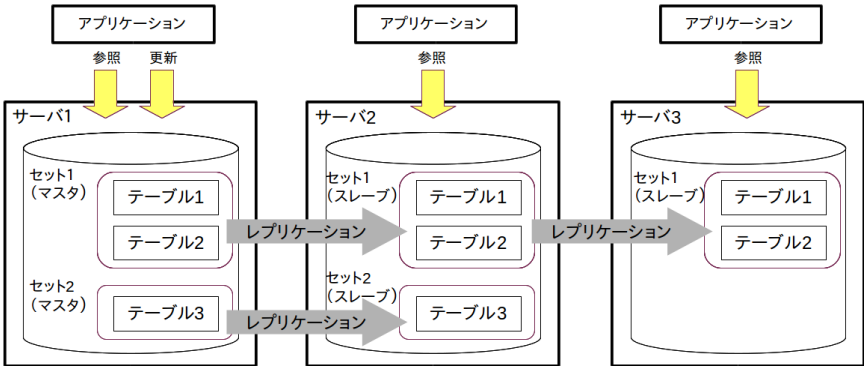


図 10.2: テーブル毎の部分レプリケーション構成

これらの制約の他にも Slony-I には、更新負荷が高い場合は PostgreSQL 組み込みのストリーミングレプリケーションに比べて更新遅延が大きい、設定や保守には専用のコマンドを実行する必要があり手間が多いという注意点があります。
以上の点を踏まえると、Slony-I が推奨されるのは、9.0 より前の古いバージョンの PostgreSQL を用いる場合、バージョン違いの PostgreSQL 間や、OS またはアーキテクチャが異なるサーバ間でレプリケーションを行う場合、特定のテーブルのみの部分レプリケーショ ン構成を組む場合などであり、それ以外では PostgreSQL 組み込みのストリーミングレプリケーション機能を用いることを推奨します。

10.2. 可用性

10.2.1. 同期性

Slony-I のレプリケーションは非同期型のためマスタとスレーブの間には遅延が発生し、データベースの内容は必ずしも一致しない場合があります。
マスタへの更新情報はトリガ関数により Slony-I の管理テーブルに記録されます。この書き込みは元の更新と同じトランザクションの中で行われため、更新が成功して更新情報の書き込みだけが失敗するということはありません。このため、元のテーブルのコミット済みの更新履歴と管理テーブル内の更新情報は完全に一致しており、仮にマスタサーバがクラッシュした場合でも、その後のリカバリが可能であれば、マスタに対して行われた全ての更新をスレーブに適用することができます。ただし非同期であるので、マスタサーバのデータが失われた場合にはデータ消失の可能性があります。
また、10.1.2 項で延べたとおり、特殊な事情がない限りは全テーブルを同期させる用途には向いていません。

10.2.2. 障害時のサービス継続性

前述のとおり、スレーブに障害が発生した場合においても、マスタへの更新・参照は問題なく行えます。その間の更

新情報は管理テーブルに記録されるため、スレーブをクラスタに復帰させるとレプリケーションを継続することができます。

マスタに障害が発生するとレプリケーション対象のテーブルを更新することが出来なくなります。ここでフェイルオーバを実行すると、指定したスレーブをマスタへと昇格させることができます。フェイルオーバは Slony-I の failover コマンドを用いて行います。このコマンドを実行すると、今まで旧マスタからレプリケーションを行っていた全てのノードの複製元が新マスタへと切り替えられ、レプリケーションが再開されます。図 10.3 に例を示します。サーバ 2 が障害が発生したサーバ 1 の役割を引き継ぎ、セット 1、セット 2 のマスタへと昇格した様子を表しています。ただし、Slony-I はマスタの障害を検出して自動的にフェイルオーバする機能は備えていません。これを実現するためには、Pacemaker や pgpool-II などの他のクラスタリングツールと併用する必要があります。

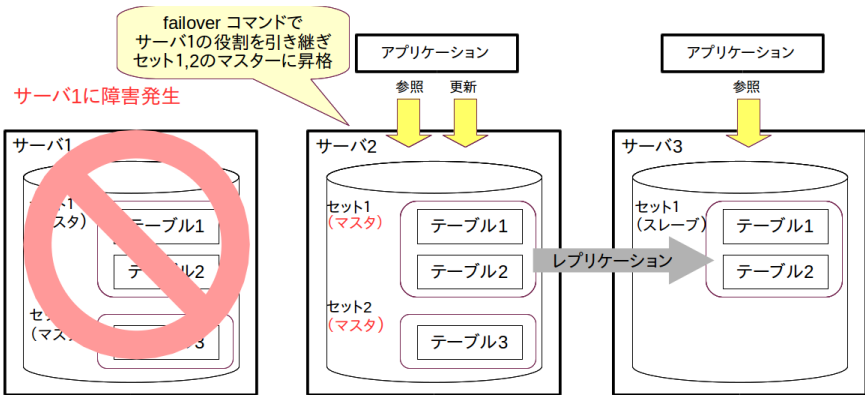


図 10.3: マスタ障害時のフェイルオーバ

10.2.3. 障害復旧時の運用性

障害が発生したノードはクラスタから取り除き、障害の原因解明と対応を行います。また可能ならば必要なデータの退避などを行います。例えば、マスタにはコミット済みだが他のサーバへは適用されなかった更新データが旧マスタの Slony-I の管理テーブルに残っている可能性があります。

一度障害が発生し切り離されたノードは他のノードとの不整合が発生している可能性があるため、クラスタに復帰させる前にはデータベースおよびテーブルスキーマを手動で再作成しておく必要があります。その後ノードを復帰させ、スレーブとして再設定するとレプリケーションが開始されます。前項の例で切り離されたマスタをスレーブとしてクラスタに復帰させた様子を図 10.4 に示します。

この状態から障害発生前の図 10.2 の状態に戻すには、マスタとスレーブを入れ替えるスイッチオーバ操作を行う必要があります。スイッチオーバは move set というコマンドで行います。スイッチオーバはレプリケーション対象のセット毎に必要であり、例えば図 10.4 から図 10.2 の状態に移行する場合にはセット 1 とセット 2 それぞれに対しスイッチオーバを実行します。スイッチオーバはデータの消失を伴わずに実行可能であり、マスタノードの障害復旧時以外にも、通常メンテナンスのための計画停止の場合に使用されます。

スイッチオーバはその直前に排他ロックを取得するため、マスタで進行中のトランザクションが存在する場合にはその完了を待って実行されます。また、この操作により接続中のセッションが切断されるようなことはありません。

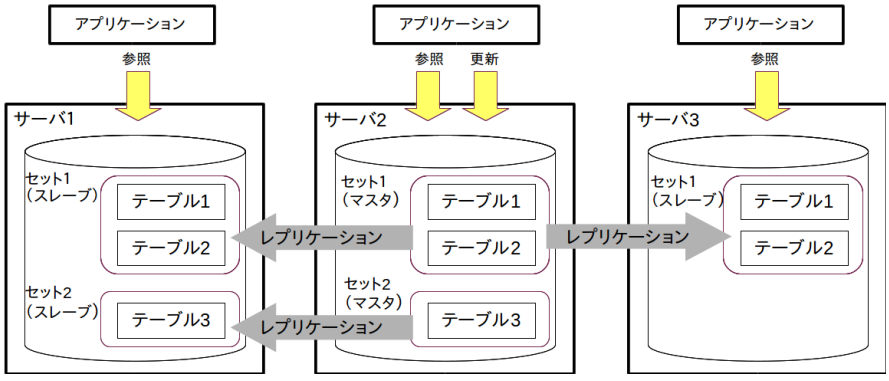


図 10.4: 障害が発生したサーバ 1 をスレーブとしてクラスタに戻した状態

10.3. バックアップ

Slony-I はテーブル単位でレプリケーションを行います。レプリケーション対象となっているテーブルは、レプリカをバックアップとみなすこともできますが、データベース全体のバックアップは別途必要になります。また、レプリケーションしているテーブルも、オペレーションミスによるデータ破壊への対策は必要です。以上より、本構成では基本的にはシングル構成と同じバックアップ方式での運用が必要になります。

なお本書では Slony-I の設定ファイルなどのバックアップについては扱いません。

10.4. 監視

10.4.1. Slony-I を用いた構成に対する監視内容の変更

本構成はシングルサーバ構成と比較すると、マスタサーバと複数のスレーブサーバ間のデータ同期、個々のサーバの障害検知、障害発生時のフェイルオーバなどの機能が追加されています。そのため、それぞれの機能が正しく動作しているかどうかを確認する死活監視の項目もあわせて追加する必要があります。また性能監視については監視対象のサーバが増えた以外は、基本的にシングル構成と違いはありません。

10.4.2. 方式の違いによる監視内容の変更点

本構成では、データ同期、負荷分散、障害発生時のフェイルオーバは Slony-I の機能で行いますが、障害を自動で検出する機能や、障害検出時にフェイルオーバを実施する処理は別途スクリプトを用意する必要があります。

10.4.3. Slony-I を用いた構成での監視のまとめ

Slony-I の特徴は非同期であることと、更新のみが行える 1 つのマスタに複数のスレーブが付随することです。そのためサーバ単位での監視はシングル構成と同様ですが、マスタからスレーブに対して正常にデータの更新が行われていることを確認するスクリプト、及びマスタとスレーブそれぞれに対して障害を自動で検出するスクリプトを追加する必要があります。

11. pgpool-II（アクティブスタンバイ）

本章では、pgpool-II を冗長化したクラスタ構成における可用性、バックアップ、監視について記述します。

11.1. 前提とする構成

pgpool-II（アクティブスタンバイ）は、単一障害点となる pgpool-II をアクティブスタンバイ型で冗長化することでシステム全体の可用性を向上させた構成です。この構成ではアプリケーションからアクセスする際に利用する仮想 IP をアクティブな pgpool-II が稼働するサーバに割り当てるのが一般的です。

この構成には、pgpool-II 間での死活監視や障害時の切替えの仕組みを用意する必要があり、Heartbeat, Pacemaker といった HA クラスタソフトウェアを組み合わせる方法や、バージョン 3.2 以降 の pgpool-II に搭載された watchdog 機能[18]を利用する方法が一般的です。

なお、この構成における性能拡張性は前述の「pgpool-II（マスタスレーブモード）」と同一です。ただし、pgpool-II を複数台運用することになるため、バックエンドの PostgreSQL サーバを増減させた場合は、アクティブ、スタンバイ両方の pgpool-II に対して同じ変更を反映させておく必要があることに注意が必要です。

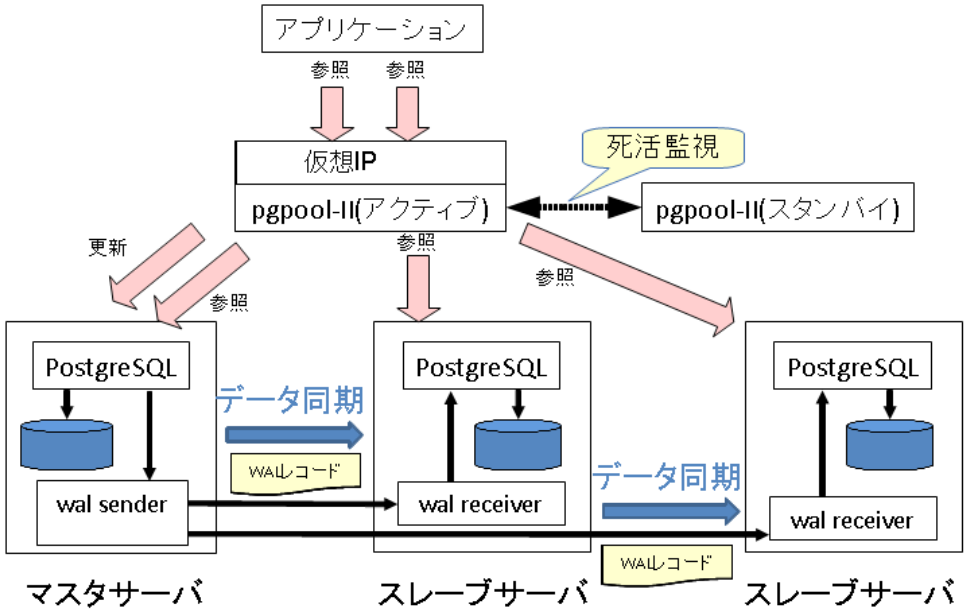


図 11.1: pgpool-II（アクティブスタンバイ）

11.2. 可用性

11.2.1. 障害時のサービス継続性

本章の構成で、PostgreSQL サーバに障害が発生した場合の挙動は、前述の「pgpool-II（マスタスレーブモード）」の場合の挙動と同一であるため解説は割愛します。

pgpool-II が稼働するサーバに障害が発生した場合は、アプリケーションからのアクセスに利用する仮想 IP をアクティブな pgpool-II からスタンバイの pgpool-II へと割り当てを変更することで、アプリケーションからアクセスする先の pgpool-II を切り替えます。なお、アクティブな pgpool-II の障害は即時に検知するわけではなく、定期的に死活監視を行うことで検知するため、検知するまでに一定の時間がかかります。つまり、pgpool-II 障害時のサービス停止時間は、pgpool-II サーバの障害を検知する時間[19]に依存することになります。

18 http://www.pgpool.net/docs/latest/pgpool-ja.html#watchdog
19 pgpool-II の watchdog を使う場合、設定ファイル（pgpool.conf）の wd_interval 等のパラメータで設定します。

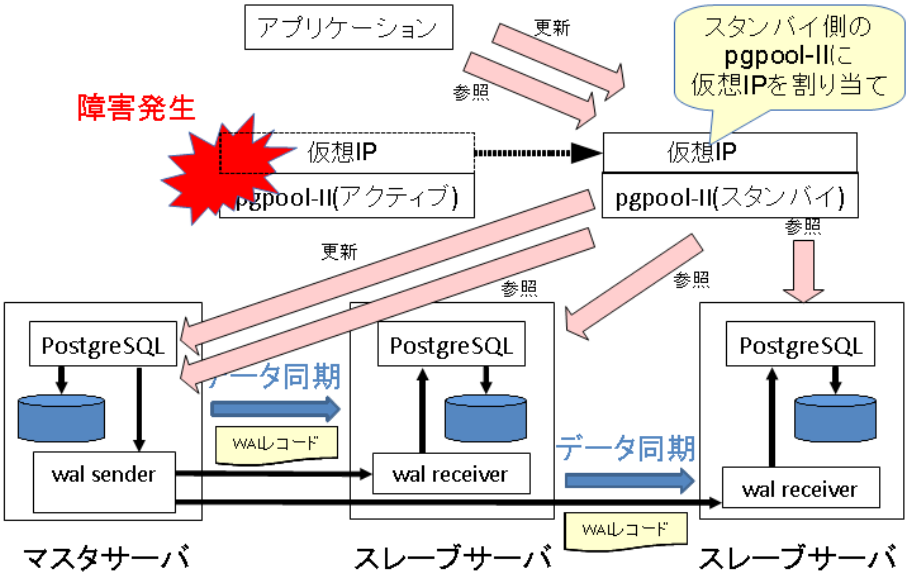


図 11.2: pgpool-II 稼働サーバ障害時の挙動

11.2.2. 障害復旧時の運用性

本章の構成で PostgreSQL サーバに障害が発生した場合の復旧は、前述の「pgpool-II（マスタスレーブモード）」の場合の挙動と同一です。また、pgpool-II が稼働するサーバに障害が発生した場合の復旧手順は watchdog 機能を利用する場合でも Pacemaker といった HA クラスタソフトウェアを利用する場合でも、pgpool-II を再起動するのみで、データベースサービスを停止することなく実行することができます。

11.3. バックアップ

本構成のバックアップ／リカバリ方式には、シングル構成における各種方式と同じ方式を用いることができます。

11.4. 監視

pgpool-II 自身の死活監視は watchdog で行うことを前提とした監視内容の変更について確認します。

11.4.1. pgpool-II(アクティブスタンバイ)に対する監視内容の変更

本章の構成において監視を行うにあたっては、追加した pgpool-II が稼働するサーバと pgpool-II のプロセス、ログを監視する必要があります。

11.4.2. 方式の違いによる監視内容の変更点

本章の構成による監視内容は既に記載したサーバの監視、pgpool-II の監視から変更点はありません。シングル構成や pgpool-II を利用した構成の内容をご確認ください。

11.4.3. pgpool-II(アクティブスタンバイ)での監視のまとめ

本章の構成で新たにまとめるべき項目はありません。

12. pgpool-II（アクティブアクティブ）

本章では、pgpool-II を冗長化したクラスタ構成における可用性、バックアップ、監視について記述します。

12.1. 前提とする構成

pgpool-II（アクティブアクティブ）は、単一障害点となる pgpool-II を冗長化することでシステム全体の可用性を向上させた構成です。また、pgpool-II（アクティブスタンバイ）と違い、この構成では全ての pgpool-II がアクティブでありアプリケーションから利用可能です。

この構成は、Apache Web Server 等の Web サーバや JBoss、Tomcat 等のアプリケーションサーバが稼働しているサーバに同居させることで Web サーバ、アプリケーションサーバと pgpool-II 間がローカルマシン内の高速なソケット通信を利用できます。また、複数の Web サーバ／アプリケーションサーバによって自然と単一障害点を回避できるというメリットもあります。

また、この構成における性能拡張性は前述の「pgpool-II（マスタスレーブモード）」と同一です。ただし、pgpool-II を複数台運用することになるため、バックエンドの PostgreSQL サーバを増減させた場合は、全ての pgpool-II に対して同じ変更を反映させておく必要があることに注意が必要です。

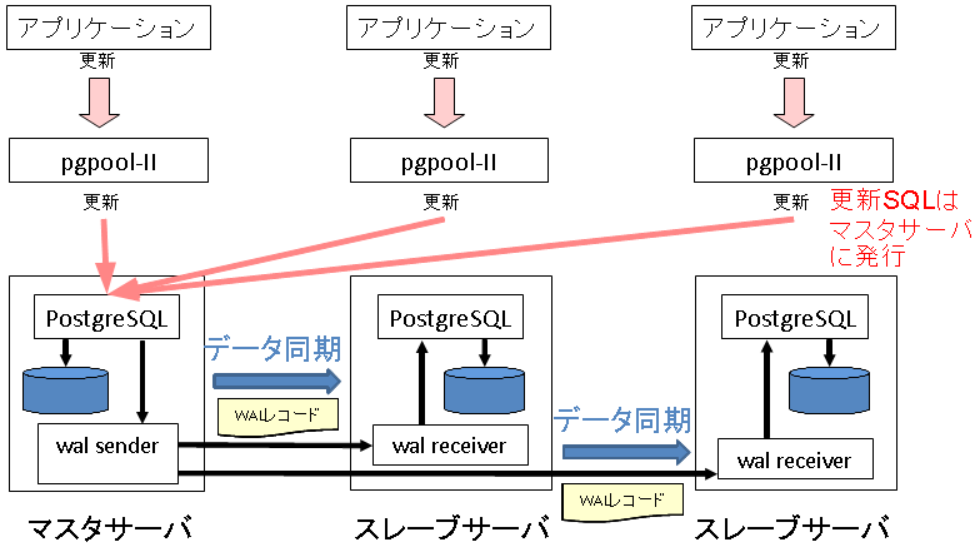


図 12.1: pgpool-II（アクティブアクティブ）における更新系処理

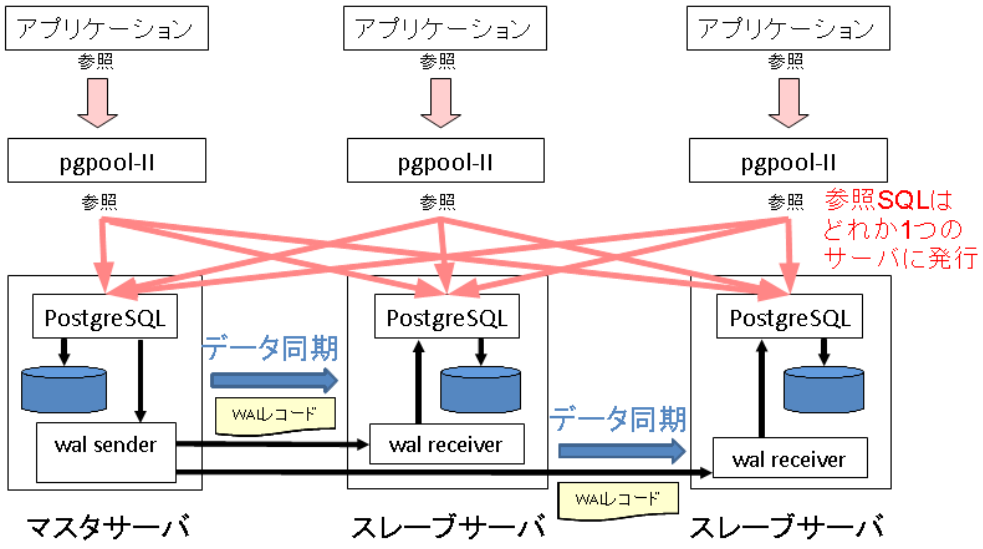


図 12.2: pgpool-II（アクティブアクティブ）における参照系処理

12.2. 可用性

12.2.1. 障害時のサービス継続性

本章の構成で PostgreSQL サーバに障害が発生した場合の挙動は、前述の「pgpool-II（マスタスレーブモード）」の場合の挙動とほぼ同一ですが、複数の pgpool-II は独立してバックエンドの PostgreSQL サーバを監視しているため、マスタサーバに障害が発生した際にフェイルオーバ操作が重複して実行されてしまいます。

pgpool-II バージョン 3.3 以降では watchdog 機能で各 pgpool-II, PostgreSQL サーバの状態を共有するようになり、フェイルオーバ操作の排他制御が可能となっていますが、この機能が使えない環境では排他制御を行う仕組みを別途作りこむ必要があります。

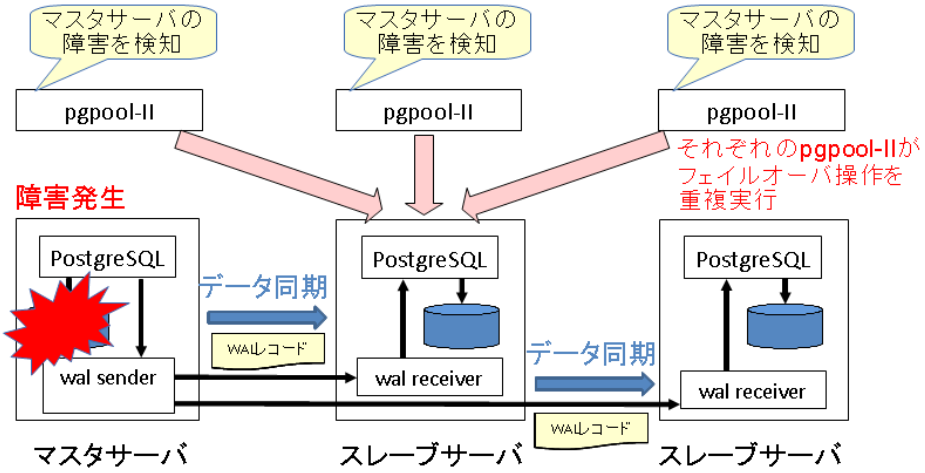


図 12.3: マスタサーバ障害時の挙動

アプリケーションサーバと pgpool-II が同一サーバに同居する構成で pgpool-II が稼働するサーバに障害が発生した場合は、下位サーバの稼働状況に応じて処理を振り分けるロードバランサが上位に配置されていれば、障害が発生したアプリケーションサーバが切り離された縮退状態になりますが、データベースサービスは継続することができます。

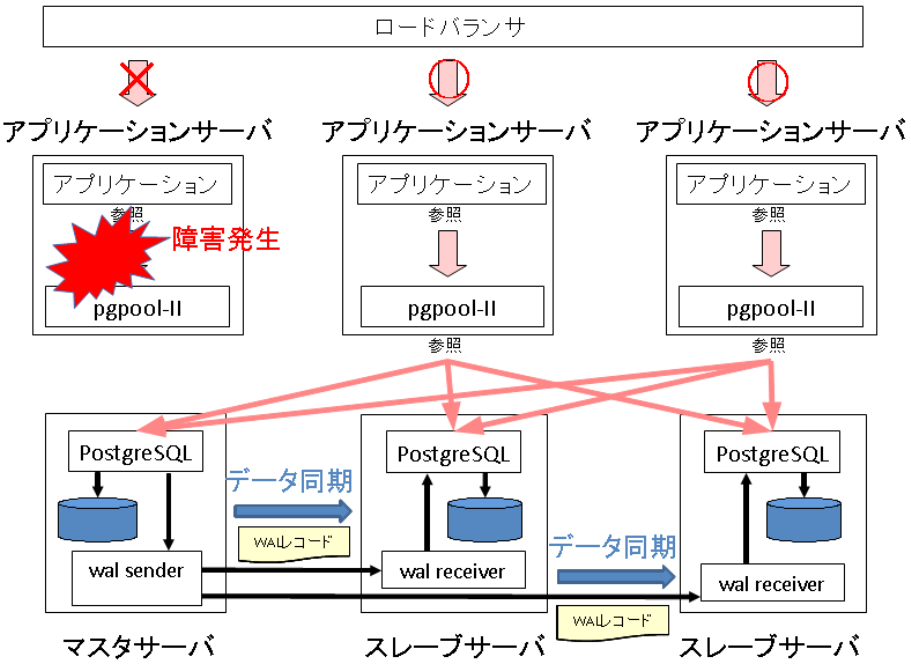


図 12.4: pgpool-II 稼働サーバ障害時の挙動

12.2.2. 障害復旧時の運用性

本章の構成で PostgreSQL サーバに障害が発生した場合の復旧方法は、前述の「pgpool-II（マスタスレーブモード）」の場合の挙動とほぼ同一ですが、フェイルオーバ操作と同様に排他制御の考慮が必要となります。

12.3. バックアップ

本構成のバックアップ／リカバリ方式には、シングル構成における各種方式と同じ方式を用いることができます。

12.4. 監視

pgpool-II 自身の死活監視は watchdog で行うことを前提とした監視内容の変更について確認します。

12.4.1. pgpool-II(アクティブアクティブ)に対する監視内容の変更

本章の構成において監視を行うにあたっては、追加した pgpool-II が稼働するサーバと pgpool-II のプロセス、ログを監視する必要があります。

12.4.2. 方式の違いによる監視内容の変更点

本章の構成による監視内容は既に記載したサーバの監視、pgpool-II の監視から変更点はありません。シングル構成や pgpool-II を利用した構成の内容をご確認ください。

12.4.3. pgpool-II(アクティブアクティブ)での監視のまとめ

本章の構成で新たにまとめるべき項目はありません。

13. 運用技術検証

本章では、3～12 章で解説した PostgreSQL の運用技術をより詳細に確認するための技術検証結果について報告します。可用性・バックアップ、監視について、それぞれ以下の検証結果を報告します。

- PostgreSQL ストリーミングレプリケーションと pgpool-II を組み合わせた高可用性検証
- シングル構成とストリーミングレプリケーション構成でのバックアップ、リカバリ検証
- 実際の障害パターンを想定した監視ケーススタディ

13.1. pgpool-II+PostgreSQL の構成検証

13.1.1. 可用性を担保する機能の基礎検証

本項では第 11 章で解説した pgpool-II と PostgreSQL を用いた高可用性構成について実機検証を行った結果を報告します。

検証対象は 2 台の PostgreSQL によるストリーミングレプリケーションと pgpool-II の自動フェイルオーバ機能を組み合わせた PostgreSQL 冗長化構成です。また、pgpool-II も watchdog 機能を利用して冗長化したアクティブスタンバイ構成となっています。この構成の特徴について詳しくは第 11 章を参照してください。

本検証の目的は、実際の運用で起きうる障害発生を実機環境でシミュレーションすることで、本構成の高可用性性能を評価することにあります。検証する機能の概要は以下のとおりです。

(1) PostgreSQL の障害検出と自動フェイルオーバ（図 13.1）

pgpool-II はバックエンドに登録されている PostgreSQL の状態を定期的に監視しています（ヘルスチェック機能）。この定期監視に失敗した場合、またはセッション中に接続エラーを検出した場合には、pgpool-II はこのバックエンドを切り離しクエリの送信対象から除外します。また、ストリーミングレプリケーションのマスタサーバに障害が発生した場合にはスレーブサーバを新マスタサーバに昇格させる自動フェイルオーバ機能を持ちます。

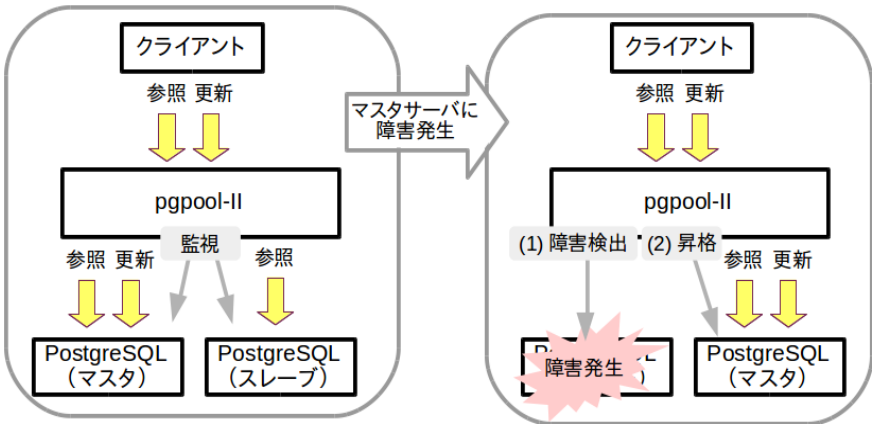


図 13.1: PostgreSQL の障害検出と自動フェイルオーバ

(2) pgpool-II の障害検出と仮想 IP アドレスの付け替え（図 13.2）

watchdog 機能を用いた 2 台の pgpool-II のアクティブスタンバイ構成であり、片方の pgpool-II は仮想 IP アドレスを保持してクライアントの接続を受け付け、もう片方は障害に備えて待機しています。仮想 IP アドレスを保持しているアクティブ pgpool-II に障害が発生した場合には、待機しているスタンバイ pgpool-II が自動的に仮想 IP アドレスを引き継ぎ、新アクティブとなります。これにより pgpool-II が単一障害点となることを回避します。

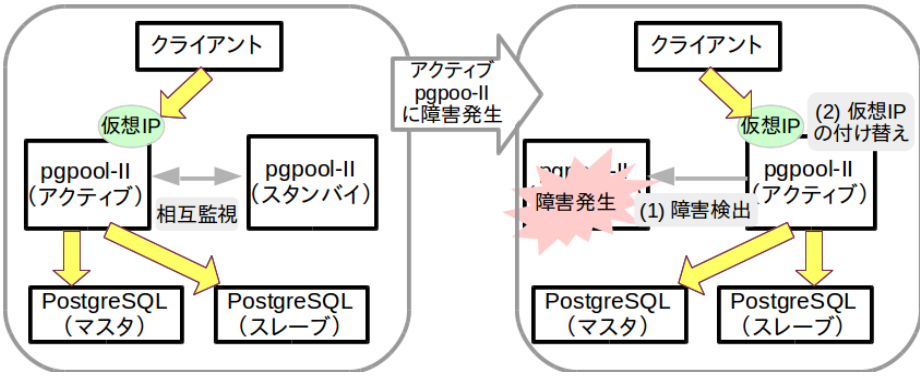


図 13.2: pgpool-II の障害検出と仮想 IP アドレスの付け替え

(3) バックエンドのオンラインリカバリ（図 13.3）

障害により切り離された PostgreSQL を pgpool-II のバックエンドにスレーブサーバとして復帰させることが可能です。この操作は接続中のセッションに影響を与えずに行うことができます。

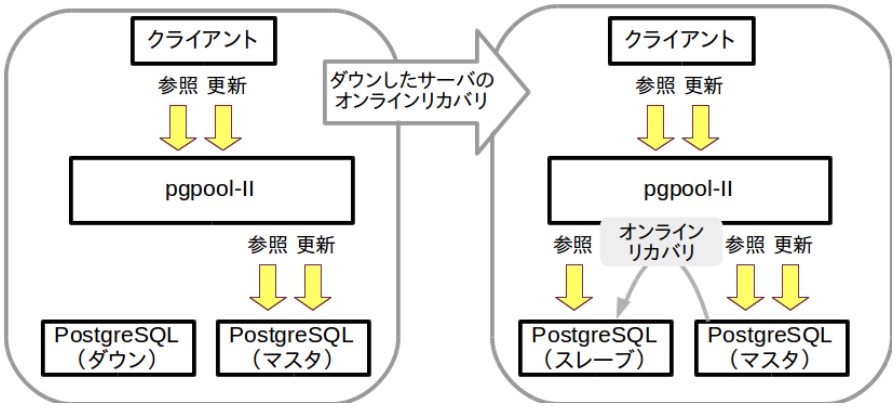


図 13.3: バックエンドのオンラインリカバリ

(4) スレーブサーバの動的追加による参照負荷分散性能のスケールアウト（図 13.4）

接続中のセッションに影響を与えることなく、スレーブサーバを動的に追加 し、参照負荷分散性能をスケールアウトすることが可能です。

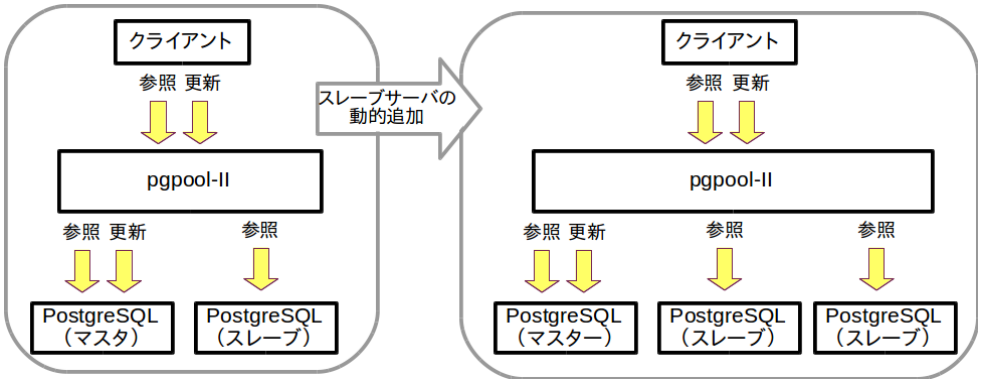


図 13.4: スレーブサーバの動的追加による参照負荷分散性能のスケールアウト

検証環境

(1)ソフトウェア構成

検証には pgpool-II 3.3.2 と PostgreSQL 9.3.1 を用いました。これらは共に本検証実施時(2013 年 12 月 11 日)の最新バージョンです。

(2)マシン構成

検証に用いた環境のネットワーク構成を図 13.5 に示します。pgpool-II 同士の死活監視を行うハートビート信号用の LAN を二重に冗長化し、その他の通信は全て共通の LAN で行うネットワーク構成としました。

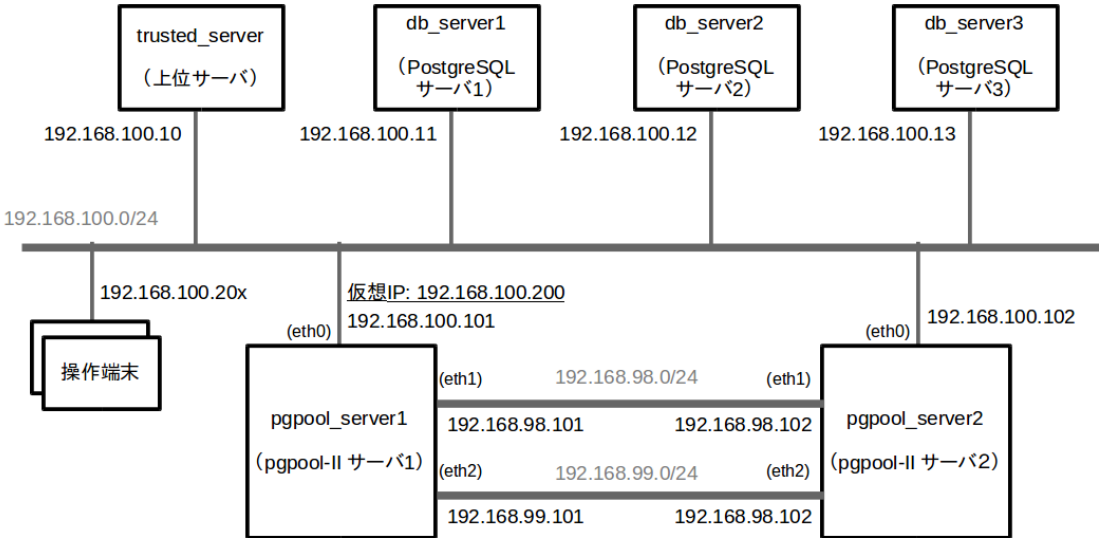


図 13.5: ネットワーク構成

環境を構成するマシンは、pgpool-II サーバ2台、PostgreSQL サーバ3台、上位サーバ 1 台の、合計6台です。各マシンのホスト名、役割、および使用した IP アドレスの一覧を表 13.1 に、ハードウェアおよびソフトウェアのスペックの一覧を表 13.2 に示します。

表 13.1: マシン一覧

ホスト名	役割	IP アドレス
pgpool_server1	pgpool-II サーバ	192.168.100.101, 192.168.98.101, 192.168.99.101
pgpool_server2	pgpool-II サーバ	192.168.100.102, 192.168.98.102, 192.168.99.102
	(pgpoo-II 用仮想 IP アドレス)	192.168.100.200
trusted_server	上位サーバ、クライアント	192.168.100.10
db_server1	PostgreSQL サーバ	192.168.100.11
db_server2	PostgreSQL サーバ	192.168.100.12
db_server3	PostgreSQL サーバ	192.168.100.13

表 13.2: サーバスペック一覧

マシン	項目	内容
pgpool-II サーバ	CPU	Intel Pentium プロセッサ g2020T 2.50GHz
	メモリ	10GB
	NIC 個数	3（eth0, eth1, eth2 ）
	OS	CentOS 6.4 64bit
上位サーバ、PostgreSQL サーバ	CPU	Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz
	メモリ	2GB
	NIC 個数	1（eth0）
	OS	CentOS 6.4 64bit

検証は各 pgpool-II のバックエンドとして db_server1, db_server2 の2台の PostgreSQL サーバが登録されている状態から開始しました。もう 1 台の PostgreSQL サーバである db_server3 はバックエンドの追加および削除の検証で用いました。

表 13.1 の役割にある「上位サーバ」とは pgpool-II のクライアントと同じ LAN 上にあるサーバであり、pgpool-II は自己診断にこのサーバを用います。pgpool-II は定期的にこのサーバへ ping による接続を試みることで、自身がクライアントに対してサービスを提供可能であるかどうかチェックします。上位サーバへの ping が失敗した場合には自身に障害が発生したと判断します。本来は上位サーバには複数台数のサーバを指定するのが望ましいですが、今回は 1 台としました。

また本検証では、上位サーバは pgpool-II に接続するクライアントの役割も兼ねています。具体的には、以下のスクリプトを上位サーバ上で実行することで、pgpool-II に対し定期的に SQL クエリを発行しました。このスクリプトを用いてクエリ実行の成否を確認することで、サービス継続性（検証中の各操作がクライアントに与える影響）の検証としました。

```
# SQL 実行スクリプト:
# 定期的にクエリを実行し、成功すれば SUCCESS、失敗すれば ERROR と出力する。

HOST=192.168.100.200
while true
do
 val1=$(date)
 val2=$(PGCONNECT_TIMEOUT=1 psql -h $HOST -p 9999 -U postgres -q -t -c "SELECT NOW();")
 if [ $? -ne 0 ]; then
  val2="ERROR"
 else
  val2="SUCCESS"
 fi
 echo "${val1}, ${val2}"
 sleep 1
done
```

表 13.1 に記載したマシンおよび IP アドレスの情報は各 pgpool-II の設定ファイルである pgpool.conf に記載しておきます。以下にその抜粋を示します。

```
# バックエンドのサーバの設定
backend_hostname0 = '192.168.100.11'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/var/lib/pgsql/data'
```

```
#backend_flag0 = 'ALLOW_TO_FAILOVER'

backend_hostname1 = '192.168.100.12'
backend_port1 = 5432
backend_weight1 = 1
backend_data_directory1 = '/var/lib/pgsql/data'
#backend_flag1 = 'ALLOW_TO_FAILOVER'

# 上位サーバの設定
trusted_servers = '192.168.100.10'

# 仮想 IP アドレスの設定
delegate_IP = '192.168.100.200'

# 自ホストの設定（pgpool_server1 の場合）
wd_hostname = '192.168.100.101'

# 相手ホストの設定（pgpool_server1 の場合）
other_pgpool_hostname0 = '192.168.100.102'
other_pgpool_port0 = 9999
other_wd_port0 = 9000

# ハートビート信号の送り先（pgpool_server1 の場合）
heartbeat_destination0 = '192.168.98.102'
heartbeat_destination_port0 = 9694
heartbeat_device0 = ''

heartbeat_destination1 = '192.168.99.102'
heartbeat_destination_port1 = 9694
heartbeat_device1 = ''
```

（3）環境構築

PostgreSQL はコミュニティのレポジトリ[20]から RPM を用いてインストールしました。各 PostgreSQL サーバ（db_server1, db_server2, db_server3）の他、各 pgpool-II サーバ（pgpool_server1, pgpool_server2）にも PostgreSQL をインストールしておく必要があります。これは後述する pgpool-II インストーラの前提条件となっているためです。

pgpool-II の公式サイトでは対話式のインストーラが公開されており、これを用いると 2 台のサーバへの pgpool-II のインストールと設定を簡単に行うことができます。ただし、このインストーラで構築できる環境は各サーバに pgpool-II と PostgreSQL が同居する構成のみであり、本検証で用いる pgpool-II と PostgreSQL を別々のサーバにインストールするような構成には対応していません。そのため本検証環境は、まず公式のインストーラを用いて pgpool_server1, pgpool_server2 の 2 台を設定した後に、さらに手動で追加の設定を行うことで構築しました。

追加で行った設定の概要は以下のとおりです。

- pgpool.conf にバックエンドの情報、ハートビート信号の送り先、上位サーバを設定
- pgpool-II 実行ユーザ、PostgreSQL 実行ユーザから他の PostgreSQL 実行ユーザへのパスワードなしの SSH が可能になるよう設定（ssh-copy-id を用いて~/.ssh/authorized_keys に公開鍵を登録）
- PostgreSQL サーバに pgpool-II で必要なライブラリ関数をインストール
- PostgreSQL に pgpool-II で必要な EXTENTION を作成
- 各サーバにホスト名を設定（/etc/hosts, /etc/sysconfig/network を編集）
- PostgreSQL サーバにスクリプトファイルを配置、ディレクトリを作成、など

20 http://yum.postgresql.org/repopackages.php

これ以外の設定は全てデフォルト値を用いました。環境構築手順の詳細は「別紙：検証環境の構築手順（13.1.1章）」を参照してください。

(4) pgpoolAdmin

インストーラにより pgpool-II サーバには pgpoolAdmin という GUI 管理ツールがインストールされます。pgpoolAdmin は PHP で作成された Web アプリケーションであり、ブラウザから pgpool-II の操作やステータスの確認などが可能です(図 13.6)。本検証においても pgpool-II の操作は pgpoolAdmin を用いて行いました。

pgpoolAdmin は pgpool_server1, pgpool_server2 の両サーバにインストールされます。ブラウザの URL 欄に「http://192.168.100.101/pgpoolAdmin/」のよう pgpool-II サーバの IP アドレスを入力することで、それぞれの pgpool-II に対応する pgpoolAdmin を起動することができます。

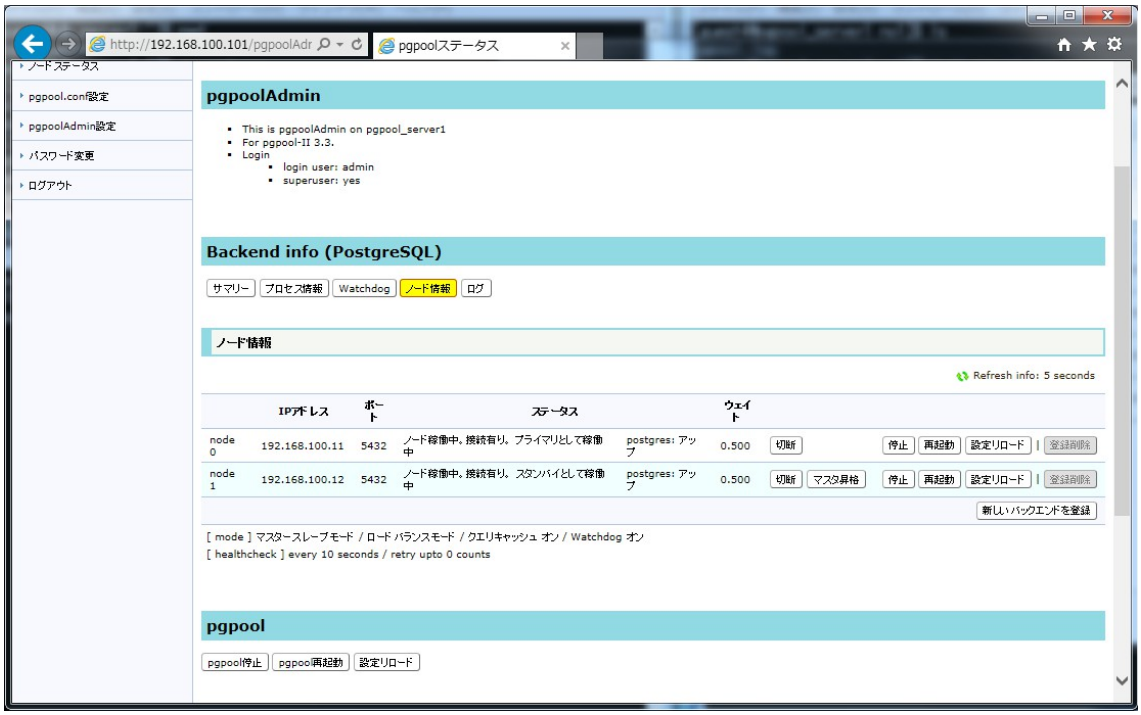


図 13.6: pgpoolAdmin

検証ケース

本検証で用いるシナリオの概要を説明します。

(1) pgpool-II 起動時および停止時の挙動

2台の pgpool-II を順に起動および停止した際の挙動を検証します。

pgpool-II は起動時に「指定された上位サーバが存在するか」、「指定された仮想 IP アドレスが既にネットワーク上に存在していないか」のチェックを行い、問題がない場合にのみ起動に成功します。最初に起動された pgpool-II は仮想 IP アドレスを立ち上げ、クライアントからの接続を受け付けるアクティブ pgpool-II となります。

2番目に起動された pgpool-II は既に起動済みの pgpool-II に対してクラスタ参加のリクエストを発行します。ここで、2台の pgpool-II 間で仮想 IP アドレスの指定が異なる場合には、後から起動した pgpool-II は起動済みの pgpool-II から拒否され起動することができません。これにより、同じクラスタを構成する全ての pgpool-II が同じ仮想 IP アドレス設定を共有することが保証されます。後から起動した pgpool-II は、仮想 IP アドレスを保持せずに待機系として振る舞うスタンバイ pgpool-II となります。

アクティブ pgpool-II が正常停止する際、停止の直前に他の pgpool-II へ自分がダウンすることを通知します。それを受けたスタンバイ pgpool-II は新しいアクティブ pgpool-II に昇格して仮想 IP アドレスを引き継ぎます。なお、その後に旧アクティブ pgpool-II を再起動させた場合は、スタンバイ pgpool-II としてクラスタに参加します。

(2) DB 障害時の挙動

PostgreSQL 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、(a) PostgreSQL サーバの電源が落ちた場合、(b) PostgreSQL プロセスが停止した場合、(c) pgpool-II と PostgreSQL の間のネットワークが遮断された場合、の3通りを想定しました。
いずれの場合も障害が検出されると pgpool-II はバックエンドからその PostgreSQL を切り離し、クエリの送信対象から外します。その時には pgpool-II に接続していたセッションは一旦切断され、クライアントには一時的にエラーが返されます。切り離し終了後には再び接続できるようになります。
障害が発生した PostgreSQL がマスタサーバであった場合には、pgpool-II の自動フェイルオーバ機能により、スレーブサーバが新マスタサーバに昇格します。この際にもマスタサーバの切り離しを伴うため、pgpool-II に接続していたセッションは一旦切断され、フェイルオーバ完了後に再び接続できるようになります。
障害発生により PostgreSQL が pgpool-II から切り離された後に、障害の発生した PostgreSQL の再起動、あるいは遮断されたネットワークの再接続が行われたとします。しかし、それだけでは一度切り離された PostgreSQL が pgpool-II のバックエンドに復旧することはありません。これは、問題を残した PostgreSQL が意図せずバックエンドに戻ってきてしまうのを防ぐためです。一度障害により切り離された PostgreSQL は、障害の原因解明や対応がなされた後に、pgpool-II のオンラインバックアップ機能を用いてスレーブサーバとして復帰させます。この操作は接続中のセッションに影響を与えずに行うことができます。
マスタサーバがダウンした場合、これを再びマスタサーバとして復帰させるためには 3 段階の手順が必要です。(1)旧マスタサーバをオンラインリカバリにより新スレーブサーバとして復帰させます。(2)現マスタ(旧スレーブ)サーバを停止してフェイルオーバを発生させ、現スレーブ(旧マスタ)サーバを再びマスタサーバに昇格させます。この時には接続中のセッションに切断が生じます。(3)直前に停止させた旧スレーブサーバをオンラインリカバリにより再びスレーブサーバとして復帰させると障害発生前の状態に戻すことができます。

(3) pgpool-II プロセス障害時の挙動

pgpool-II のプロセスが異常停止した場合の挙動を検証します。
pgpool-II は親であるメインプロセスと、そこから fork した複数の子プロセスから構成されています。子プロセスが異常停止した場合には、メインプロセスがそれを検出し自動的に子プロセスの再生成を行います。pgpool-II の監視を行っている watchdog 機能も子プロセス群により実現されていますので、仮に watchdog 関連のプロセスが異常停止した場合にも、これらは自動的に復旧されます。
一方で、watchdog のプロセスからもメインプロセスを監視しており、もしメインプロセスが異常停止した場合にはこれを検出し、pgpool-II をダウン状態へと移行させます。アクティブ pgpool-II がダウン状態に移行した場合には、通知を受けたスタンバイ pgpool-II が新しい アクティブに昇格し、仮想 IP アドレスの付け替えが行われます。なお、一度ダウン状態になった pgpool-II はアクティブにもスタンバイにもなることができません。これを復旧させるためにはダウン状態の pgpool-II を再起動させる必要があります。

(4) pgpool-II ネットワーク障害時の挙動

pgpool-II が用いるネットワークに障害が発生した場合の挙動を検証します。
pgpool-II は上位サーバに接続できなかったときは、サービス LAN に障害が発生して自身がサービス継続不能になったとみなしダウン状態へと移行します。それがアクティブ pgpool-II であった場合には、通知を受けたスタンバイ pgpool-II が新しいアクティブに昇格し仮想 IP アドレスの付け替えが行われます。ダウン状態から復旧させるためには、pgpool-II を再起動する必要があります。
pgpool-II 同士の相互監視に用いているハートビート LAN は本検証では二重化しているため、片方が遮断されただけでは問題はありません。ただし、両方の LAN に一度に遮断された場合にはハートビート信号の交換が完全に途絶えてしまい、pgpool-II は互いに相手がダウンしたと解釈してしまいます。そのため、両方の pgpool-II がアクティブとなり、同じ仮想 IP アドレスが LAN 上に複数存在する状態となってしまいます。この状態から復旧するためには、片方の pgpool-II を再起動させ、スタンバイ pgpool-II としてクラスタに再参加させる必要があります。

(5) バックエンドの追加と削除

pgpool-II のバックエンドへの PostgreSQL サーバを新規追加、および既存のサーバの削除ができることを検証します。

pgpool-II バックエンドへの PostgreSQL サーバの追加は前述のオンラインリカバリ機能を利用します。まず、全ての pgpool-II に新しいサーバの情報を登録します。具体的には、サーバのホスト名、PostgreSQL ポート番号などを設定ファイル pgpool.conf に追加で書き込んだ後に、pgpool-II をリロードして設定を読み込ませます。その後、オンラインリカバリを実行すると、新しいサーバがスレーブサーバとして追加されます。この操作では接続中のセッションには影響を与えません。

pgpool-II からバックエンドサーバを削除する場合は、まず当該サーバ上の PostgreSQL を停止させます。pgpool-II がこれを検出することにより、サーバはバックエンドから切り離されます。その後 pgpool.conf からサーバの情報を取り除き、pgpool-II を再起動します。この操作ではバックエンドの切り離しを伴うので、pgpool-II に接続中のセッションは一旦切断されてしまいます。

検証結果

前項のシナリオに従って検証を行った結果、実際に得られた挙動を説明します。検証開始前の状態では、db_server1,db_server2 上の PostgreSQL はそれぞれマスタサーバ、スレーブサーバとして稼働済みであり、pgpool-II は 2 台とも停止された状態でした。

(1) pgpool-II 起動時と停止時の挙動

本検証の全体を通して、pgpool-II の操作は pgpoolAdmin から行いました。pgpool-II の起動は pgpoolAdmin の「pgpool 起動」ボタンから行うことができます。また pgpool-II のログは各 pgpool-II サーバの /var/log/pgpool/pgpool.log に出力されます。

pgpool-II の起動（1 台目）

まず、上位サーバの LAN ケーブルを抜いた状態で pgpool_server1 の pgpool-II を起動すると、ログに以下のようなエラーが出力され、上位サーバにアクセスできない状況では想定どおり起動に失敗することが確認されました。

```
2013-12-11 11:15:42 ERROR: pid 5838: wd_init: failed to connect trusted server
2013-12-11 11:15:42 ERROR: pid 5838: watchdog: wd_init failed
2013-12-11 11:15:42 ERROR: pid 5838: wd_main error
```

次に上位サーバの LAN ケーブルを戻し、仮想 IP アドレスに上位サーバと同じ IP アドレス指定します。具体的には pgpool.conf の仮想 IP アドレスの設定を以下のように書き換えました。

```
delegate_IP = '192.168.100.10'
#delegate_IP = '192.168.100.200'
```

その状態で pgpool-II を起動すると、ログに以下のようなエラーが出力されました。想定どおり、ネットワーク上に既に存在する IP アドレスを仮想 IP アドレス指定した場合には起動に失敗することが確認されました。

```
2013-12-11 11:23:44 ERROR: pid 6018: wd_init: delegate_IP 192.168.100.10 already exists
2013-12-11 11:23:44 ERROR: pid 6018: watchdog: wd_init failed
2013-12-11 11:23:44 ERROR: pid 6018: wd_main error
```

pgpool.conf の設定を元に戻し、pgpool_server1 の pgpool-II を起動するとログに以下のように出力され正常に起動できたことが確認されました。

```
2013-12-11 11:27:29 LOG:   pid 6061: wd_escalation: escalated to master pgpool successfully
2013-12-11 11:27:29 LOG:   pid 6061: wd_init: start watchdog
2013-12-11 11:27:29 LOG:   pid 6061: pgpool-II successfully started. version 3.3.2 (tokakiboshi)
```

また、ifconfig コマンドにより pgpool_server1 に指定した仮想 IP アドレス 192.168.100.200 が立ち上がっていることが確認されました。

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.101  Bcast:192.168.100.255  Mask:255.255.255.0
...
eth0:0    Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.200  Bcast:192.168.100.255  Mask:255.255.255.0
...
```

pgpool-II の起動後、pgpoolAdmin の「ノード情報」の画面で 2 台の PostgreSQL がそれぞれマスタサーバ、スレーブサーバとして稼働していることが確認されました(図 13.7)。

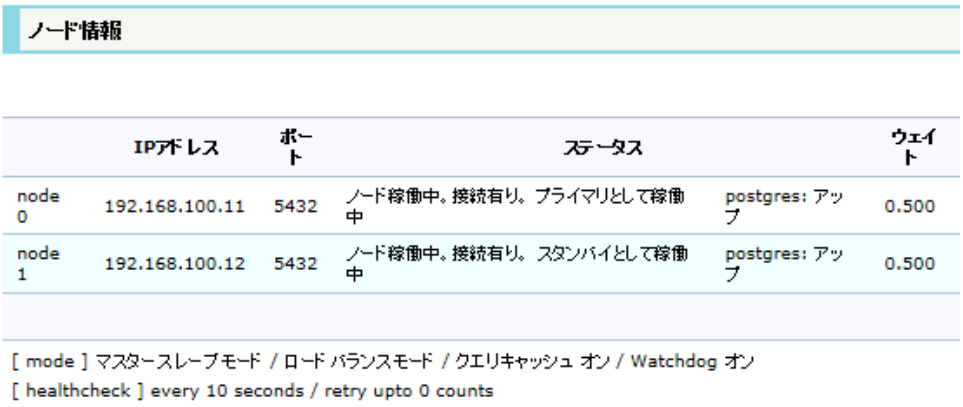


図 13.7: pgpoolAdmin ノード情報（初期状態）

また、pgpoolAdmin の watchdog 情報の画面では pgpool_server1 の pgpool-II がアクティブとして起動したことが確認されました（図 13.8）。

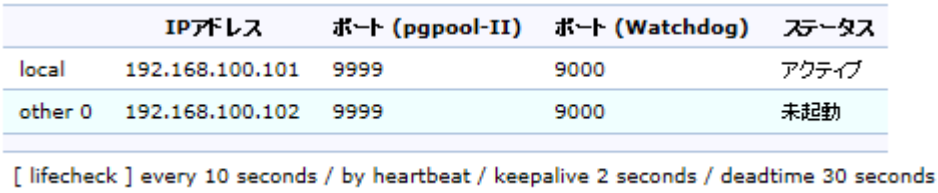


図 13.8: pgpool_server1 の pgpool-II がアクティブとして起動

pgpool-II の起動（2台目）

次に pgpool_server2 上で pgpool-II を起動します。まず以下のように pgpool_server2 の pgpool.conf を編集し、pgpool_server1 と異なる仮想 IP アドレスを設定してみます。

```
delegate_IP = '192.168.100.222'
#delegate_IP = '192.168.100.200'
```

その状態で pgpool_server2 の pgpool-II を起動すると、ログに以下のメッセージが出力され起動に失敗することが確認されました。

```
2013-12-11 11:38:33 ERROR: pid 2874: wd_init: failed to start watchdog
2013-12-11 11:38:33 ERROR: pid 2874: watchdog: wd_init failed
2013-12-11 11:38:33 ERROR: pid 2874: wd_main error
```

pgpool_server1 のログには以下のように出力されており、仮想 IP アドレスの設定が異なる pgpool-II の参加が拒否されていることが確認されました。

```
2013-12-11 11:38:33 ERROR: pid 6072: wd_set_wd_list: delegate IP mismatch error
2013-12-11 11:38:33 LOG:   pid 6072: wd_send_response: receive add request from 192.168.100.102:9999 and reject it
```

次に、pgpool.conf の仮想 IP アドレスの設定を元に戻し pgpool_server2 の pgpool-II を起動すると、ログに以下のように出力され正常に起動したことが確認されました。

```
2013-12-11 11:42:03 LOG:   pid 2918: wd_init: start watchdog
2013-12-11 11:42:03 LOG:   pid 2918: pgpool-II successfully started. version 3.3.2 (tokakiboshi)
```

pgpool_server1 のログには以下のように出力され、今度は pgpool-II の起動が受理されたことが確認されました。

```
2013-12-11 11:42:03 LOG:   pid 6072: wd_send_response: receive add request from 192.168.100.102:9999 and accept it
```

しばらく経過すると、両 pgpool-II のログに以下のメッセージが出力され、相互の死活監視が開始されたことが確認されました。

```
2013-12-11 11:42:30 LOG:   pid 6077: watchdog: lifecheck started
```

ifcoonfig コマンドにより pgpool_server2 では仮想 IP アドレスは立ち上がっていないことを確認しました。

pgpoolAdmin の watchdog 情報の画面でも pgpool_server2 の pgpool-II がスタンバイとして起動したことが確認されました（図 13.9、図 13.10）。

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.101	9999	9000	アクティブ
other 0	192.168.100.102	9999	9000	スタンバイ

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.9: pgpool_server2 の pgpool-II がスタンバイとして起動（pgpool_server1 から見た場合）

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.102	9999	9000	スタンバイ
other 0	192.168.100.101	9999	9000	アクティブ

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.10: pgpool_server2 の pgpool-II がスタンバイとして起動（pgpool_server2 から見た場合）

アクティブ pgpool-II の停止

次に pgpool_server1 の pgpool-II を停止します。事前に上位サーバから SQL 実行スクリプトを実行し、pgpool-II の仮想 IP アドレスに対して定期的にクエリを発行されている状態にしておきます。その後、pgpool_server1 の pgpoolAdmin 画面で「pgpool 停止」ボタンから pgpool-II を停止しました。
ログには以下のように出力され、pgpool_server1 の pgpool-II が停止し、pgpool_server2 の pgpool-II がアクティブに昇格したことが確認されました。（ログメッセージの中の”master pgpool”とはアクティブ pgpool-II のことを意味しています。）

```
(pgpool_server1)
2013-12-11 12:59:23 LOG:   pid 6061: received fast shutdown request
2013-12-11 12:59:26 LOG:   pid 6077: wd_IP_down: ifconfig down succeeded
```

```
(pgpool_server2)
2013-12-11 12:59:26 LOG:   pid 2923: wd_escalation: escalating to master pgpool
...
2013-12-11 12:59:28 LOG:   pid 2923: wd_escalation: escalated to master pgpool successfully
```

SQL 実行スクリプトの結果からは、仮想 IP アドレス切り替わりの間はクエリ実行が失敗することが確認されました。

```
2013 年 12 月 11 日 水曜日 12:59:21 JST, SUCCESS
2013 年 12 月 11 日 水曜日 12:59:22 JST, SUCCESS
2013 年 12 月 11 日 水曜日 12:59:23 JST, ERROR
2013 年 12 月 11 日 水曜日 12:59:26 JST, SUCCESS
2013 年 12 月 11 日 水曜日 12:59:28 JST, SUCCESS
```

ifconfig コマンドにより仮想 IP アドレスが pgpool_server1 では停止し、代わりに pgpool_server2 で立ち上がったことが確認されました。

```
(pgpool_server1)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.101  Bcast:192.168.100.255  Mask:255.255.255.0
...
eth1      Link encap:Ethernet  HWaddr 10:60:4B:92:38:28
          inet addr:192.168.98.101  Bcast:192.168.98.255  Mask:255.255.255.0
...
```

```
(pgpool_server2)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:B6:FA
          inet addr:192.168.100.102  Bcast:192.168.100.255  Mask:255.255.255.0
...
```

```
eth0:0    Link encap:Ethernet  HWaddr 2C:27:D7:15:B6:FA
          inet addr:192.168.100.200  Bcast:192.168.100.255  Mask:255.255.255.0
...
```

pgpoolAdmin からも、pgpool_server2 の pgpool-II がアクティブになったことが確認されました（図 13.11）。

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.102	9999	9000	アクティブ
other 0	192.168.100.101	9999	9000	ダウン

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.11: pgpool_server1 の pgpool-II を停止。pgpool_server2 の pgpool-II がアクティブに昇格する。（pgpool_server2 からみた場合）

pgpool-II の再起動

先ほど停止した pgpool_server1 の pgpool-II を再起動させます。ログには以下のように出力され正常起動が確認されました。

```
(pgpool_server1)
2013-12-11 13:05:53 LOG:   pid 15608: wd_init: start watchdog
2013-12-11 13:05:53 LOG:   pid 15608: pgpool-II successfully started. version 3.3.2 (tokakiboshi)
```

```
(pgpool_server2)
2013-12-11 13:05:53 LOG:   pid 2923: wd_send_response: receive add request from 192.168.100.101:9999 and accept it
```

ifconfig コマンドの結果から仮想 IP アドレスの所在には変化がないことが確認されました。また pgpoolAdmin の画面から pgpool_server1 の pgpool-II が今度はスタンバイとして起動したことが確認されました（図 13.12、図 13.13）。

SQL 実行スクリプトの出力には異常はなく、クライアントへの影響は確認されませんでした。

```
2013 年 12 月 11 日 水曜日 13:05:51 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:05:52 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:05:53 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:05:54 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:05:55 JST, SUCCESS
```

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.101	9999	9000	スタンバイ
other 0	192.168.100.102	9999	9000	アクティブ

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.12: pgpool_server1 の pgpool-II を再起動（pgpool_server1 からみた場合）

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.102	9999	9000	アクティブ
other 0	192.168.100.101	9999	9000	スタンバイ

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.13: pgpool_server1 の pgpool-II を再起動（pgpool_server2 からみた場合）

ここまでの検証で、db1_server, db2_server の PostgreSQL はそれぞれマスタサーバ、スレーブサーバとして、pgpool_server1, pgpool_server2 の pgpool-II はそれぞれスタンバイ, アクティブとして稼働している状態になっています。

(2a) DB ノード障害

スレーブ DB サーバの電源断

サービス提供中にスレーブサーバが異常停止した状況をシミュレートするため、上位サーバで SQL 実行スクリプトを実行し pgpool-II にクエリが送信されている状態で db2_server の電源をシャットダウンしました。シャットダウンはサーバの電源ボタンを長押しすることで行いました。サーバのシャットダウン後しばらくすると、pgpool-II が DB のダウンを検出し、db2_server がバックエンドから切り離されたことがログにより確認されました。

```
(pgpool_server1)
2013-12-11 13:15:56 ERROR: pid 15608: health check failed. 1 th host 192.168.100.12 at port 5432 is down
2013-12-11 13:15:56 LOG:   pid 15608: set 1 th backend down status
…
2013-12-11 13:15:57 LOG:   pid 15608: starting degeneration. shutdown host 192.168.100.12(5432)
…
2013-12-11 13:15:57 LOG:   pid 15608: execute command: /etc/pgpool-II/failover.sh 1 192.168.100.12
5432 /var/lib/pgsql/data 0 0 192.168.100.11 0 5432 /var/lib/pgsql/data
Node 1 which is not the primary dies. This script doesn't anything.
…
2013-12-11 13:15:58 LOG:   pid 15608: failover done. shutdown host 192.168.100.12(5432)
```

```
(pgpool_server2)
2013-12-11 13:15:56 LOG:   pid 2918: starting degeneration. shutdown host 192.168.100.12(5432)
…
2013-12-11 13:15:58 LOG:   pid 2918: failover done. shutdown host 192.168.100.12(5432)
```

pgpoolAdmin の「ノード情報」画面からは db2_server のステータスが「ノードダウン」という表記になったことが確認されました（図 13.14）。

ノード情報

	IPアドレス	ポート	ステータス		ウェイト	
node 0	192.168.100.11	5432	ノード稼働中。接続有り。プライマリとして稼働中	postgres: アップ	0.500	切断
node 1	192.168.100.12	5432	ノードダウン	postgres: ダウン	0.500	リカバリ

[mode] マスタースレーブモード / ロード バランスモード / クエリキャッシュ オン / Watchdog オン
[healthcheck] every 10 seconds / retry upto 0 counts

図 13.14: db_server2 の PostgreSQL がダウンした状態

接続中のセッションは一度切断され、切り離し終了後に再度接続が可能になることが確認されました。

```
2013 年 12 月 11 日 水曜日 13:15:50 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:15:51 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:15:52 JST, ERROR
2013 年 12 月 11 日 水曜日 13:15:55 JST, ERROR
2013 年 12 月 11 日 水曜日 13:15:57 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:15:59 JST, SUCCESS
```

停止したスレーブサーバの再起動

db2_server の電源投入後に PostgreSQL を起動しても pgpool-II からの切り離しは維持され、pgpoolAdmin の画面には「ノードダウン」と表示されたままでした（図 13.15）。

図 13.15: ダウンしたスレーブサーバを再起動

スレーブサーバのフェイルバック

次に、オンラインリカバリにより切り離された旧スレーブサーバをクラスタに復帰させます。db2_server の PostgreSQL を停止した状態（図 13.14 の状態）で、pgpoolAdmin の「リカバリ」ボタンを押下するとオンラインリカバリが開始されます。これはどちらの pgpoolAdmin から操作しても構いませんが、ここでは現アクティブである pgpool_server2 の pgpoolAdmin で実行しました。

pgpool_server2 のログに以下のように出力され、オンラインリカバリが実行されたことが確認されました。

```
2013-12-11 13:31:17 LOG:   pid 10011: starting recovering node 1
…
2013-12-11 13:31:44 LOG:   pid 10011: send_failback_request: fail back 1 th node request from pid 10011
…
2013-12-11 13:31:45 LOG:   pid 2918: starting fail back. reconnect host 192.168.100.12(5432)
…
2013-12-11 13:31:46 LOG:   pid 2918: failback done. reconnect host 192.168.100.12(5432)
2013-12-11 13:31:46 LOG:   pid 10011: recovery done
```

pgpool_server1 のログには以下のように出力され、アクティブ pgpool-II だけではなくスタンバイ pgpool-II にも PostgreSQL がフェイルバックされたことが確認されました。

```
2013-12-11 13:31:44 LOG:   pid 15617: send_failback_request: fail back 1 th node request from pid 15617
…
2013-12-11 13:31:45 LOG:   pid 15608: starting fail back. reconnect host 192.168.100.12(5432)
…
2013-12-11 13:31:46 LOG:   pid 15608: failback done. reconnect host 192.168.100.12(5432)
```

pgpoolAdmin の画面でも db2_server の PostgreSQL がスレーブサーバとして復帰したことが確認されました（図 13.7 と同画面）。

この操作による接続中のセッションには影響はみられませんでした。

```
2013 年 12 月 11 日 水曜日 13:31:10 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:31:11 JST, SUCCESS
...
2013 年 12 月 11 日 水曜日 13:31:45 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:31:46 JST, SUCCESS
```

マスタサーバの電源断

次に現マスタサーバである db_server1 の電源をシャットダウンしました。シャットダウンの方法は先ほどと同じです。しばらくすると pgpool-II がマスタサーバがダウンしたことを検出しフェイルオーバが開始されました。旧マスタサーバが両 pgpool-II のバックエンドから切り離され、代わりにスレーブサーバが新マスタサーバに昇格したことがログおよび pgpoolAdmin の画面（図 13.16）により確認されました。

```
(pgpool_server1)
2013-12-11 13:36:55 LOG:   pid 15608: starting degeneration. shutdown host 192.168.100.11(5432)
...
2013-12-11 13:36:56 LOG:   pid 15608: find_primary_node_repeatedly: waiting for finding a primary node
2013-12-11 13:36:57 LOG:   pid 15608: find_primary_node: primary node id is 1
...
2013-12-11 13:36:57 LOG:   pid 15608: failover done. shutdown host 192.168.100.11(5432)
```

```
(pgpool_server2)
2013-12-11 13:36:54 ERROR: pid 2918: health check failed. 0 th host 192.168.100.11 at port 5432 is down
2013-12-11 13:36:54 LOG:   pid 2918: set 0 th backend down status
...
2013-12-11 13:36:55 LOG:   pid 2918: starting degeneration. shutdown host 192.168.100.11(5432)
...
2013-12-11 13:36:55 LOG:   pid 2918: execute command: /etc/pgpool-II/failover.sh 0 192.168.100.11 5432
/var/lib/pgsql/data 1 0 192.168.100.12 0 5432 /var/lib/pgsql/data
2013-12-11 13:36:56 LOG:   pid 2918: find_primary_node_repeatedly: waiting for finding a primary node
2013-12-11 13:36:57 LOG:   pid 2918: find_primary_node: primary node id is 1
...
2013-12-11 13:36:58 LOG:   pid 2918: failover done. shutdown host 192.168.100.11(5432)
```

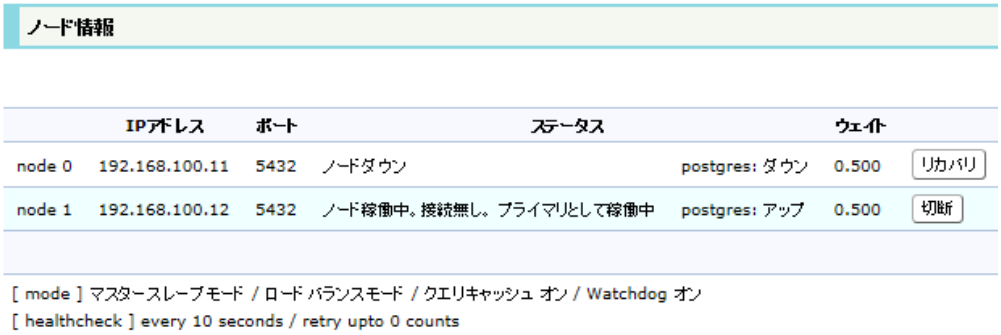


図 13.16: db_server1 の PostgreSQL がダウンした状態：db_server2 の PostgreSQL がマスタサーバに昇格する

バックエンド切り離しを伴うため接続中のセッションは一度切断されましたが、切り離し終了後には接続が可能になりました。

```
2013 年 12 月 11 日 水曜日 13:36:50 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:36:51 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:36:52 JST, ERROR
2013 年 12 月 11 日 水曜日 13:36:55 JST, ERROR
2013 年 12 月 11 日 水曜日 13:36:56 JST, ERROR
2013 年 12 月 11 日 水曜日 13:36:59 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:37:00 JST, SUCCESS
```

停止したマスタサーバの再起動

db1_server の電源を入れた後に PostgreSQL を起動しましたが、pgpool-II からの切り離しは維持され、pgpoolAdmin には「ノードダウン」と表示されたままでした（図 13.17）。これにより pgpool-II のバックエンドにマスタサーバが 2 台存在する状況が回避されています。

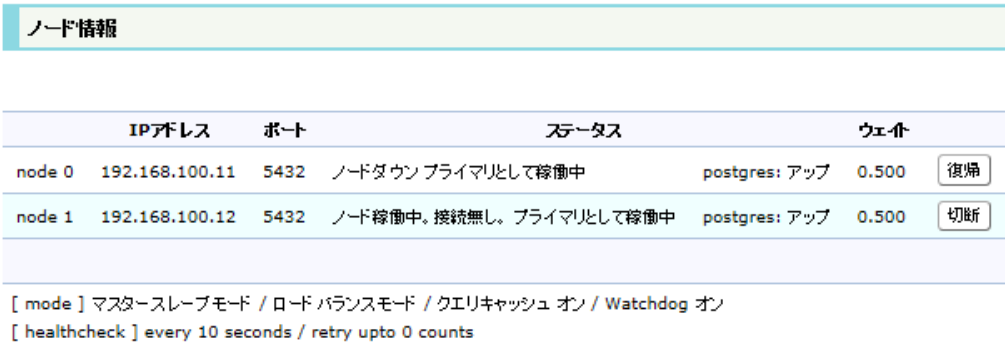


図 13.17: ダウンしたマスタサーバを再起動

マスタサーバのフェイルバック

ここから障害発生前状態（db_server1 がマスタサーバ、db_server2 がスレーブサーバ：図 13.7）の状態に戻すため、まず旧マスタサーバをオンラインリカバリしスレーブサーバとして復帰させます。db1_server の PostgreSQL を停止した状態（図 13.16 の状態）で、pgpoolAdmin の「リカバリ」ボタンを押下します。どちらの pgpoolAdmin から操作しても構いませんが、今回は現アクティブである pgpool_server2 の pgpoolAdmin から実行しました。リカバリ終了後、db1_server の PostgreSQL がスレーブサーバとして復帰したことが確認されました（図 13.18）。

図 13.18: 初期状態からマスタサーバとスレーブサーバが逆転した状態

その後、現マスタサーバである db_server2 の PostgreSQL を停止すると、フェイオーバが発生し db_server1 の PostgreSQL が再びマスタサーバに昇格し図 13.14 と同じ状態になりました[21]。PostgreSQL の停止は pgpoolAdmin のノード情報画面にある「停止」ボタンから行いました（図 13.6 参照）。停止モードのオプションを選ぶことが可能で、ここでは「-i immediately」で停止させました。

バックエンドの切り離しが発生し、接続中のセッションは一度切断されました。

```
2013 年 12 月 11 日 水曜日 13:51:47 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:51:48 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:51:49 JST, ERROR
2013 年 12 月 11 日 水曜日 13:51:50 JST, ERROR
2013 年 12 月 11 日 水曜日 13:51:53 JST, ERROR
2013 年 12 月 11 日 水曜日 13:51:56 JST, SUCCESS
2013 年 12 月 11 日 水曜日 13:51:57 JST, SUCCESS
```

21 図 13.18 の pgpoolAdmin 画面に「マスタ昇格」というボタンがありますが、これは pgpool-II から見たステータスを変化させるものであり、実際にバックエンドの PostgreSQL のスイッチオーバを行うものではありません。

最後に db_server2 の PostgreSQL をオンラインリカバリでスレーブサーバとして復帰させました（pgpoolAdmin で「リカバリ」ボタンを押下）。これにより、PostgreSQL の構成状態を障害発生前と同じ状態（図 13.7 の状態）に戻すことができました。

(2b) PostgreSQL プロセス障害

スレーブサーバの停止

サービス提供中に PostgreSQL プロセスが異常停止した状況をシミュレートするため、上位サーバで SQL 実行スクリプトを実行中に db2_server の PostgreSQL プロセスを強制停止します。PostgreSQL の停止は pgpoolAdmin のノード情報画面の「停止」ボタンにより行いました。停止のオプションは「-i immediately」を用いました。

pgpool-II が PostgreSQL のダウンを検出しバックエンドから切り離されたことがログ出力及び pgpoolAdmin の画面（図 13.14 と同じ状態）で確認されました。

```
(pgpool_server1)
2013-12-11 14:19:19 LOG:   pid 15608: starting degeneration. shutdown host 192.168.100.12(5432)
…
2013-12-11 14:19:20 LOG:   pid 15608: failover done. shutdown host 192.168.100.12(5432)
```

```
(pgpool_server2)
2013-12-11 14:19:18 ERROR: pid 18660: connection to 192.168.100.12(5432) failed
…
2013-12-11 14:19:19 LOG:   pid 2918: starting degeneration. shutdown host 192.168.100.12(5432)
…
2013-12-11 14:19:19 LOG:   pid 2918: execute command: /etc/pgpool-II/failover.sh 1 192.168.100.12
5432 /var/lib/pgsql/data 0 0 192.168.100.11 0 5432 /var/lib/pgsql/data
Node 1 which is not the primary dies. This script doesnt't anything.
…
2013-12-11 14:19:20 LOG:   pid 2918: failover done. shutdown host 192.168.100.12(5432)
```

バックエンド切り離しにより接続中のセッションは一度切断されますが、切り離し終了後に接続が可能になります。

```
2013 年 12 月 11 日 水曜日 14:19:16 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:19:17 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:19:18 JST, ERROR
2013 年 12 月 11 日 水曜日 14:19:19 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:19:21 JST, SUCCESS
```

停止したスレーブサーバの再起動

その後 db2_server の PostgreSQL を起動しても、pgpool-II からの切り離しは維持されたままであることが確認されました。（図 13.15 と同じ状態）。

スレーブサーバのフェイルバック

障害発生前の状態に戻すため、切り離された旧スレーブサーバをクラスタに復帰させました。db2_server の PostgreSQL を停止させた状態で、pgpoolAdmin の「リカバリ」ボタンを押下しオンラインリカバリを行いました。

マスタサーバの停止

次に現マスタサーバである db_server1 の PostgreSQL を停止します。pgpool-II がマスタサーバのダウンを検出しフェイルオーバが発生し、スレーブサーバが新マスタサーバに昇格され、旧マスタサーバがバックエンドから切り離されたことが以下のログおよび pgpoolAdmin の画面（図 13.16 と同じ状態）により確認されました。

```
(pgpool_server1)
2013-12-11 14:30:43 LOG:   pid 15608: starting degeneration. shutdown host 192.168.100.11(5432)
…
2013-12-11 14:30:44 LOG:   pid 15608: find_primary_node_repeatedly: waiting for finding a primary node
…
2013-12-11 14:30:49 LOG:   pid 15608: find_primary_node: primary node id is 1
…
2013-12-11 14:30:49 LOG:   pid 15608: failover done. shutdown host 192.168.100.11(5432)
```

```
(pgpool_server2)
2013-12-11 14:30:43 ERROR: pid 24281: connection to 192.168.100.11(5432) failed
2013-12-11 14:30:43 ERROR: pid 24281: new_connection: create_cp() failed
…
2013-12-11 14:30:43 LOG:   pid 2918: starting degeneration. shutdown host 192.168.100.11(5432)
…
2013-12-11 14:30:43 LOG:   pid 2918: execute command: /etc/pgpool-II/failover.sh 0 192.168.100.11
5432 /var/lib/pgsql/data 1 0 192.168.100.12 0 5432 /var/lib/pgsql/data
2013-12-11 14:30:43 LOG:   pid 2918: find_primary_node_repeatedly: waiting for finding a primary node
2013-12-11 14:30:48 LOG:   pid 2918: find_primary_node: primary node id is 1
…
2013-12-11 14:30:49 LOG:   pid 2918: failover done. shutdown host 192.168.100.11(5432)
```

バックエンド切り離しにより接続中のセッションは一度切断されますが、切り離し終了後に接続可能になりました。

```
2013 年 12 月 11 日 水曜日 14:30:41 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:30:42 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:30:43 JST, ERROR
2013 年 12 月 11 日 水曜日 14:30:44 JST, ERROR
2013 年 12 月 11 日 水曜日 14:30:47 JST, ERROR
2013 年 12 月 11 日 水曜日 14:30:50 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:30:51 JST, SUCCESS
```

停止したマスタサーバの再起動

その後 db1_server の PostgreSQL を起動しても pgpool-II からの切り離しは維持されることが確認されました（図 13.17 と同じ状態）。

マスタサーバのフェイルバック

(2a)の最後に記載したのと同じ手順で障害発生前の状態（db_server1 がマスタサーバ、db_server2 がスレーブサーバ）に戻しておきます。

(2c) PostgreSQL ネットワーク障害

スレーブサーバのネットワーク切断

サービス中に PostgreSQL サーバにネットワーク障害が発生した状況をシミュレートするため、上位サーバで SQL 実行スクリプトを実行中に db2_server の LAN ケーブルを抜きました。しばらくすると pgpool-II が PostgreSQL がダウンしたことを検出し、バックエンドからの切り離されたことが以下のログおよび pgpoolAdmin

の画面（図 13.14 と同じ状態）から確認されました。

```
(pgpool_server1)
2013-12-11 14:45:52 ERROR: pid 15608: health check failed. 1 th host 192.168.100.12 at port 5432 is down
2013-12-11 14:45:52 LOG:   pid 15608: set 1 th backend down status
…
2013-12-11 14:45:53 LOG:   pid 15608: starting degeneration. shutdown host 192.168.100.12(5432)
…
2013-12-11 14:45:53 LOG:   pid 15608: execute command: /etc/pgpool-II/failover.sh 1 192.168.100.12
5432 /var/lib/pgsql/data 0 0 192.168.100.11 0 5432 /var/lib/pgsql/data
Node 1 which is not the primary dies. This script doesnt't anything.
…
2013-12-11 14:45:54 LOG:   pid 15608: failover done. shutdown host 192.168.100.12(5432)
```

```
(pgpool_server2)
2013-12-11 14:45:52 ERROR: pid 2918: health check failed. 1 th host 192.168.100.12 at port 5432 is down
…
2013-12-11 14:45:53 LOG:   pid 2918: starting degeneration. shutdown host 192.168.100.12(5432)
…
2013-12-11 14:45:54 LOG:   pid 2918: failover done. shutdown host 192.168.100.12(5432)
```

バックエンド切り離しにより接続中のセッションは一度切断され、切り離し終了後に接続が可能になりました。

```
2013 年 12 月 11 日 水曜日 14:45:43 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:45:44 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:45:45 JST, ERROR
2013 年 12 月 11 日 水曜日 14:45:48 JST, ERROR
2013 年 12 月 11 日 水曜日 14:45:51 JST, ERROR
2013 年 12 月 11 日 水曜日 14:45:54 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:45:55 JST, SUCCESS
```

スレーブサーバのネットワーク再接続

その後、db2_server の LAN ケーブルを再接続しても、pgpool-II からの切り離しは維持されることが確認されました（図 13.15 と同じ状態）。

スレーブサーバのフェイルバック

次の検証の準備として、切り離された旧スレーブサーバをオンラインリカバリでクラスタに復帰させました。db2_server の PostgreSQL を停止させた後に、pgpoolAdmin の「リカバリ」ボタンを押下しました。

マスタサーバのネットワーク切断

次に現マスタサーバである db_server1 の LAN ケーブルを抜きます。pgpool-II がマスタサーバのダウンを検出しフェイルオーバが発生し、スレーブサーバが新マスタサーバに昇格し、旧スレーブサーバはバックエンドから切り離されたことが以下のログおよび pgpoolAdmin の画面（図 13.16 と同じ状態）で確認されました。

```
(pgpool_server1)
2013-12-11 14:57:36 LOG:   pid 15608: starting degeneration. shutdown host 192.168.100.11(5432)
…
2013-12-11 14:57:37 LOG:   pid 15608: find_primary_node_repeatedly: waiting for finding a primary node
2013-12-11 14:57:38 LOG:   pid 15608: find_primary_node: primary node id is 1
…
2013-12-11 14:57:38 LOG:   pid 15608: failover done. shutdown host 192.168.100.11(5432)
```

```
(pgpool_server2)
2013-12-11 14:57:35 ERROR: pid 2918: health check failed. 0 th host 192.168.100.11 at port 5432 is down
2013-12-11 14:57:35 LOG:   pid 2918: set 0 th backend down status
…
2013-12-11 14:57:36 LOG:   pid 2918: starting degeneration. shutdown host 192.168.100.11(5432)
2013-12-11 14:57:36 LOG:   pid 2918: Restart all children
2013-12-11 14:57:36 LOG:   pid 2918: execute command: /etc/pgpool-II/failover.sh 0 192.168.100.11
5432 /var/lib/pgsql/data 1 0 192.168.100.12 0 5432 /var/lib/pgsql/data
2013-12-11 14:57:37 LOG:   pid 2918: find_primary_node_repeatedly: waiting for finding a primary node
2013-12-11 14:57:38 LOG:   pid 2918: find_primary_node: primary node id is 1
…
2013-12-11 14:57:39 LOG:   pid 2918: failover done. shutdown host 192.168.100.11(5432)
```

バックエンド切り離しにより接続中のセッションは一度切断され、切り離し終了後に接続が可能になりました。

```
2013 年 12 月 11 日 水曜日 14:57:32 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:57:33 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:57:34 JST, ERROR
2013 年 12 月 11 日 水曜日 14:57:37 JST, ERROR
2013 年 12 月 11 日 水曜日 14:57:40 JST, SUCCESS
2013 年 12 月 11 日 水曜日 14:57:41 JST, SUCCESS
```

マスタサーバのネットワーク再接続

また、db1_server1 の LAN ケーブルを再接続しても、pgpool-II からの切り離しは維持されたままであることが確認されました（図 13.17 と同じ状態）。

マスタサーバのフェイルバック

(2a)の最後と同様の手順で障害発生前の状態（db_server1 がマスタサーバ、db_server2 がスレーブサーバである状態）に戻しておきます。

(3) pgpool-II プロセス障害時の挙動

この段階では pgpool_server1 の pgpool-II はスタンバイ、pgpool_server2 の pgpool-II はアクティブとして稼働している状態となっています（図 13.12、図 13.13 の状態）。

watchdog プロセスの強制終了

pgpool-II の watchdog のプロセス障害を kill コマンドを用いてシミュレートしました。この検証は pgpool_server1 にて行いました。まず ps コマンドで watchdog プロセスの PID を特定しました。「pgpool: watachdog」という名前のプロセスが該当します。

```
$ ps | grep watchdog
apache   15608  0.0  0.5 290876 54272 ?       S    13:05   0:00 /usr/bin/pgpool -f /etc/pgpool-II/pgpool.conf -F
/etc/pgpool-II/pcp.conf -n

apache   15617  0.0  0.0 290876  1208 ?       S    13:05   0:00 pgpool: watchdog
…
```

これを kill -9 コマンドで強制終了しました。

```
# kill -9 15617
```

その直後に再び ps コマンドを実行すると、watchdog プロセスが新しい PID で再起動されていることが確認されました。

```
$ ps | grep watchdog
apache   15608  0.0  0.5 290876 54272 ?       S    13:05   0:00 /usr/bin/pgpool -f /etc/pgpool-II/pgpool.conf -F
/etc/pgpool-II/pcp.conf -n
...
apache   37088  0.0  0.0 290876   964 ?       S    15:17   0:00 pgpool: watchdog
```

ログには以下のように出力され、新しいプロセスが fork されたのが確認できました。

```
2013-12-11 15:17:40 LOG:   pid 15608: fork a new watchdog child pid 37088
```

接続中のセッションには影響はありませんでした。

```
2013 年 12 月 11 日 水曜日 15:17:38 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:17:39 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:17:40 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:17:41 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:17:42 JST, SUCCESS
```

スタンバイ pgpool-II 親プロセスの強制停止

次にスタンバイ pgpool-II のプロセス障害を kill コマンドでシミュレートしました。ps コマンドで pgpool_server1 上の pgpool-II 親プロセスの PID を特定します。

```
$ ps | grep pgpool
apache   15608  0.0  0.5 290876 54272 ?       S    13:05   0:00 /usr/bin/pgpool -f /etc/pgpool-II/pgpool.conf -F
/etc/pgpool-II/pcp.conf -n
...
```

これを kill -9 コマンドで強制終了しました。

```
# kill -9 15608
```

しばらくすると、pgpool_server1 の pgpool-II は親プロセスの異常を検出しダウン状態に移行しました（図 13.19）。その通知を受けて pgpool_server2 の pgpool-II でもステータスが更新されたことが確認されました（図 13.20）。

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.101	9999	9000	ダウン
other 0	192.168.100.102	9999	9000	アクティブ

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.19: pgpool_server1 の pgpool-II がダウンした状態（pgpool_server1 からみた場合）

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.102	9999	9000	アクティブ
other 0	192.168.100.101	9999	9000	ダウン

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.20: pgpool_server2 の pgpool-II がダウンした状態（pgpool_server2 からみた場合）

ログには以下のように出力されました。

```
(pgpool_server1)
2013-12-11 15:23:12 LOG:   pid 15622: check_pgpool_status_by_hb: lifecheck failed. pgpool 0
(192.168.100.101:9999) seems not to be working
2013-12-11 15:23:22 ERROR: pid 15622: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
2013-12-11 15:23:32 ERROR: pid 15622: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
…
```

```
(pgpool_server2)
…
2013-12-11 15:23:23 LOG:   pid 2928: check_pgpool_status_by_hb: pgpool 1 (192.168.100.101:9999) is in
down status
2013-12-11 15:23:35 LOG:   pid 2928: check_pgpool_status_by_hb: pgpool 1 (192.168.100.101:9999) is in
down status
…
```

接続中のセッションには影響はありませんでした。

```
2013 年 12 月 11 日 水曜日 15:22:12 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:22:13 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:22:14 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:22:15 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:22:16 JST, SUCCESS
```

一度ダウン状態となった pgpool-II を復旧させるには再起動が必要です。今回は kill コマンドにより強制的に終了させたので、停止前に適切に処理されず残ってしまった不要なファイルが残っています。これらを削除した後に pgpool-II の再起動を行いました。

```
# rm /var/run/pgpool.pid
# rm /tmp/.sPGSQL.9*
```

pgpoolAdmin から pgpool_server1 の pgpool-II を再起動させるとスタンバイとして起動することを確認しました（図 13.12、図 13.13 の状態）。

アクティブ pgpool-II 親プロセスの強制終了

次に、アクティブ pgpool-II のプロセス障害を kill コマンドでシミュレートします。ps コマンドで pgpool_server2 上の pgpool-II 親プロセスの PID を特定します。

```
$ ps | grep pgpool
apache   2918  0.0  0.5 290876 54280 ?        S    11:42   0:00 /usr/bin/pgpool -f /etc/pgpool-II/pgpool.conf
-F /etc/pgpool-II/pcp.conf -n
…
```

これを kill -9 コマンドで強制終了させました。

```
$ kill -9 2918
```

pgpool_server2 の pgpool-II が親プロセスの異常を検出しダウン状態に移行し、それを受けた pgpool_server1 の pgpool-II がアクティブ pgpool-II に昇格しました（図 13.21、図 13.22）。

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.101	9999	9000	アクティブ
other 0	192.168.100.102	9999	9000	ダウン

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.21: pgpool_server2 の pgpool-II がダウンし、スタンバイがアクティブに昇格（pgpool_server1 からみた場合）

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.102	9999	9000	ダウン
other 0	192.168.100.101	9999	9000	スタンバイ

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.22: pgpool_server2 の pgpool-II がダウンし、スタンバイがアクティブに昇格（pgpool_server1 からみた場合）

ログには以下のように出力されました。

```
(pgpool_server1)
2013-12-11 15:38:26 LOG:   pid 39325: wd_escalation: escalating to master pgpool
...
2013-12-11 15:38:28 LOG:   pid 39325: wd_escalation: escalated to master pgpool successfully
2013-12-11 15:38:29 LOG:   pid 39330: check_pgpool_status_by_hb: pgpool 1 (192.168.100.102:9999) is in down status
2013-12-11 15:38:41 LOG:   pid 39330: check_pgpool_status_by_hb: pgpool 1 (192.168.100.102:9999) is in down status
```

```
(pgpool_server2)
2013-12-11 15:38:23 LOG:   pid 2928: check_pgpool_status_by_hb: lifecheck failed. pgpool 0 (192.168.100.102:9999) seems not to be working
2013-12-11 15:38:26 LOG:   pid 2928: wd_IP_down: ifconfig down succeeded
2013-12-11 15:38:38 ERROR: pid 2928: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
2013-12-11 15:38:48 ERROR: pid 2928: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
```

pgpool_server2 で仮想 IP アドレスが停止され、pgpool_server1 で仮想 IP アドレスが立ち上がったことが ifconfig コマンドにより確認されました。

```
(pgpool-server1)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.101  Bcast:192.168.100.255  Mask:255.255.255.0
 ...

eth0:0    Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.200  Bcast:192.168.100.255  Mask:255.255.255.0
```

```
(pgpool-server2)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:B6:FA
          inet addr:192.168.100.102  Bcast:192.168.100.255  Mask:255.255.255.0
...

eth1      Link encap:Ethernet  HWaddr 10:60:4B:92:3C:DC
          inet addr:192.168.98.102  Bcast:192.168.98.255  Mask:255.255.255.0
```

仮想 IP アドレスの切り替わりのため、接続中のセッションは一時切断されました。

```
2013 年 12 月 11 日 水曜日 15:38:22 JST, SUCCESS
```

```
2013 年 12 月 11 日 水曜日 15:38:23 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:38:24 JST, ERROR
2013 年 12 月 11 日 水曜日 15:38:27 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:38:28 JST, SUCCESS
```

pgpool-II のステータスを障害発生前の状態に戻すため、前述のように不要なファイルを削除してから pgpool_server2 の pgpool-II を起動しました。これにより pgpool_server2 の pgpool-II はスタンバイとして起動しました。次に pgpool_server1 の pgpool-II を再起動させると、pgpool_server2 がアクティブ、pgpool_server1 がスタンバイの状態に戻りました。

(4) pgpool-II ネットワーク障害時の挙動

スタンバイ pgpool-II のネットワーク切断

スタンバイ pgpool-II ネットワーク障害をシミュレートするため、pgpool_server1 のサービス LAN 側のケーブルを抜きました。しばらくすると、pgpool_server1 の pgpool-II は上位サーバにアクセスできないことを検出しダウン状態に移行しました（図 13.19、図 13.20 と同じ状態）。

ログには以下のように出力されました。

```
(pgpool_server1)
2013-12-11 15:52:13 ERROR: pid 43301: wd_lifecheck: failed to connect to any trusted servers
…
2013-12-11 15:52:23 ERROR: pid 43301: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
…
2013-12-11 15:52:33 ERROR: pid 43301: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
```

```
(pgpool_server2)
2013-12-11 15:52:45 LOG:   pid 46681: check_pgpool_status_by_hb: lifecheck failed. pgpool 1
(192.168.100.101:9999) seems not to be working
2013-12-11 15:52:45 LOG:   pid 46681: pgpool_down: 192.168.100.101:9999 is going down
2013-12-11 15:52:57 LOG:   pid 46681: check_pgpool_status_by_hb: pgpool 1 (192.168.100.101:9999) is in
down status
2013-12-11 15:53:09 LOG:   pid 46681: check_pgpool_status_by_hb: pgpool 1 (192.168.100.101:9999) is in
down status
```

接続中のセッションには影響はありませんでした。

```
2013 年 12 月 11 日 水曜日 15:52:10 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:52:11 JST, SUCCESS
…
2013 年 12 月 11 日 水曜日 15:52:59 JST, SUCCESS
2013 年 12 月 11 日 水曜日 15:53:00 JST, SUCCESS
```

ダウン状態の pgpool_server1 の pgpool-II を再起動すると、スタンバイ pgpool-II として復帰しました。接続中のセッションには影響はありませんでした。

アクティブ pgpool-II のネットワーク切断

次にアクティブ pgpool-II ネットワーク障害をシミュレートするため、pgpool_server2 のサービス LAN 側のケーブルを抜きました。pgpool_server2 の pgpool-II は上位サーバにアクセスできないことを検出してダウン状態に移行しました。その後、pgpool_server1 の pgpool-II はそれを検出し、アクティブ pgpool-II に昇格しました（図 13.21、図 13.22 と同じ状態）。

ログには以下のように出力されました。

```
(pgpool_server1)
2013-12-11 16:11:36 LOG:   pid 46226: check_pgpool_status_by_hb: lifecheck failed. pgpool 1
(192.168.100.102:9999) seems not to be working
2013-12-11 16:11:36 LOG:   pid 46226: pgpool_down: 192.168.100.102:9999 is going down
2013-12-11 16:11:36 LOG:   pid 46226: pgpool_down: I'm oldest so standing for master
2013-12-11 16:11:36 LOG:   pid 46226: wd_escalation: escalating to master pgpool
...
2013-12-11 16:11:38 LOG:   pid 46226: wd_escalation: escalated to master pgpool successfully
2013-12-11 16:11:50 LOG:   pid 46226: check_pgpool_status_by_hb: pgpool 1 (192.168.100.102:9999) is in
down status
2013-12-11 16:12:02 LOG:   pid 46226: check_pgpool_status_by_hb: pgpool 1 (192.168.100.102:9999) is in
down status
```

```
(pgpool_server2)
2013-12-11 16:11:01 ERROR: pid 52028: wd_lifecheck: failed to connect to any trusted servers
2013-12-11 16:11:04 LOG:   pid 52028: wd_IP_down: ifconfig down succeeded
...
2013-12-11 16:11:17 ERROR: pid 52028: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
...
2013-12-11 16:11:27 ERROR: pid 52028: wd_lifecheck: watchdog status is DOWN. You need to restart pgpool
```

ifconfig コマンドにより、pgpool_server2 では仮想 IP アドレスが停止され、pgpool_server1 では仮想 IP アドレスが立ち上がったことが確認されました。

```
(pgpool-server1)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.101  Bcast:192.168.100.255  Mask:255.255.255.0
 ...

eth0:0    Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.200  Bcast:192.168.100.255  Mask:255.255.255.0
```

```
(pgpool-server2)]
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:B6:FA
          inet addr:192.168.100.102  Bcast:192.168.100.255  Mask:255.255.255.0
...

eth1      Link encap:Ethernet  HWaddr 10:60:4B:92:3C:DC
          inet addr:192.168.98.102  Bcast:192.168.98.255  Mask:255.255.255.0
```

仮想 IP アドレスの切り替わりにより、接続中のセッションは一時切断されたことが確認されました。

```
2013 年 12 月 11 日 水曜日 16:10:38 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:10:39 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:10:40 JST, ERROR
2013 年 12 月 11 日 水曜日 16:10:43 JST, ERROR
...
2013 年 12 月 11 日 水曜日 16:11:31 JST, ERROR
2013 年 12 月 11 日 水曜日 16:11:34 JST, ERROR
```

```
2013 年 12 月 11 日 水曜日 16:11:37 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:11:38 JST, SUCCESS
```

pgpool_server2 の pgpool-II を再起動すると、スタンバイ pgpool-II として復帰しました（図 13.9、図 13.10 の状態）。この時には接続中のセッションには影響はありませんでした。

ここで障害発生前の pgpool_server2 をアクティブ、pgpool_server1 をスタンバイの状態に戻すため、pgpool_server1 の pgpool-II を再起動しました（図 13.12、図 13.13 の状態に戻る）。

ハートビート LAN の切断

最後にハートビート LAN に障害が発生した場合の挙動を検証します。

まずハートビート通信の2本の LAN の内、片方のみを抜きしばらく放置しましたが、何も影響はなく、動作は正常のままでした。ログにこのことを表すメッセージは出力されませんでした。また、抜いた LAN ケーブルを再度接続しなおしても影響はありませんでした。

次にハートビート LAN の2本両方を抜きしばらく放置すると、2台の pgpool-II それぞれが相手をダウン状態とみなし自分をアクティブとする状態になりました（図 13.23、図 13.24）。

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.101	9999	9000	アクティブ
other 0	192.168.100.102	9999	9000	ダウン

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.23: 相手をダウンとみなし自分がアクティブとなった状態（pgpool_server1 からみた場合）

	IPアドレス	ポート (pgpool-II)	ポート (Watchdog)	ステータス
local	192.168.100.102	9999	9000	アクティブ
other 0	192.168.100.101	9999	9000	ダウン

[lifecheck] every 10 seconds / by heartbeat / keepalive 2 seconds / deadtime 30 seconds

図 13.24: 相手をダウンとみなし、自分がアクティブとなった状態（pgpool_server2 からみた場合）

ログには以下のように出力され、互いに相手をダウンしたとみなしていること、および pgpool_server1 がアクティブに昇格したことが確認されました。

```
(pgpool_server1)
2013-12-11 16:25:23 LOG:   pid 48883: check_pgpool_status_by_hb: lifecheck failed. pgpool 1
(192.168.100.102:9999) seems not to be working
2013-12-11 16:25:23 LOG:   pid 48883: pgpool_down: 192.168.100.102:9999 is going down
2013-12-11 16:25:23 LOG:   pid 48883: pgpool_down: I'm oldest so standing for master
2013-12-11 16:25:23 LOG:   pid 48883: wd_escalation: escalating to master pgpool
...
2013-12-11 16:25:25 LOG:   pid 48883: wd_escalation: escalated to master pgpool successfully
2013-12-11 16:25:37 LOG:   pid 48883: check_pgpool_status_by_hb: pgpool 1 (192.168.100.102:9999) is in
down status
```

```
2013-12-11 16:25:49 LOG:   pid 48883: check_pgpool_status_by_hb: pgpool 1 (192.168.100.102:9999) is in down status
```

```
(pgpool_server2)
2013-12-11 16:25:18 LOG:   pid 54257: check_pgpool_status_by_hb: lifecheck failed. pgpool 1 (192.168.100.101:9999) seems not to be working
2013-12-11 16:25:18 LOG:   pid 54257: pgpool_down: 192.168.100.101:9999 is going down
2013-12-11 16:25:30 LOG:   pid 54257: check_pgpool_status_by_hb: pgpool 1 (192.168.100.101:9999) is in down status
2013-12-11 16:25:42 LOG:   pid 54257: check_pgpool_status_by_hb: pgpool 1 (192.168.100.101:9999) is in down status
```

ifconfig コマンドにより仮想 IP アドレスが両サーバで立ち上がっていることが確認されました。

```
(pgpool_server1)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.101  Bcast:192.168.100.255  Mask:255.255.255.0
...

eth0:0    Link encap:Ethernet  HWaddr 2C:27:D7:15:C2:58
          inet addr:192.168.100.200  Bcast:192.168.100.255  Mask:255.255.255.0
```

```
(pgpool_server2)
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 2C:27:D7:15:B6:FA
          inet addr:192.168.100.102  Bcast:192.168.100.255  Mask:255.255.255.0
...

eth0:0    Link encap:Ethernet  HWaddr 2C:27:D7:15:B6:FA
          inet addr:192.168.100.200  Bcast:192.168.100.255  Mask:255.255.255.0
```

SQL 実行スクリプトの結果からは接続中のセッションには影響は見られませんでした。しかし、同じネットワーク内に同じ仮想 IP アドレスが2つ存在する状態となっているため、新しいクライアントからの接続があった場合にどちらの pgpool-II に接続されるかは不定となることが予想されます。

```
2013 年 12 月 11 日 水曜日 16:25:02 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:25:03 JST, SUCCESS
...
2013 年 12 月 11 日 水曜日 16:25:44 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:25:45 JST, SUCCESS
```

その後、2本の LAN を接続しなおしても、2台の pgpool-II のステータスに変化はありませんでした。障害発生前の pgpool_server2 をアクティブ、pgpool_server1 をスタンバイの状態に戻すため、pgpool_server1 の pgpool-II を再起動しました。

(5) バックエンドの追加と削除

バックエンドサーバの追加

pgpool-II の新しいバックエンドとして db_server3 の PostgreSQL を追加する操作を検証しました。db_server3 の PostgreSQL は事前に停止した状態にしておきます。pgpool_server1, pgpool_server2 の両方の pgpool.conf を以下のように編集し、db_server3 の情報を追加しました。

```
backend_hostname2 = '192.168.100.13'
backend_port2 = 5432
backend_weight2 = 1
backend_data_directory2 = '/var/lib/pgsql/data'
#backend_flag1 = 'ALLOW_TO_FAILOVER'
```

その後、両サーバで pgpool-II の pgpool-II 設定のリロードを行いました。この操作は pgpoolAdmin の画面下部にある「設定リロード」ボタンから実行しました。リロード後、db_server3 の情報が pgpoolAdmin に表示されました（図 13.25）。この時点ではノードダウンというステータスとなっていることが確認されました。

図 13.25: pgpoolAdmin に db_server3 の情報が表示される

次に、pgpool_server2 の pgpoolAdmin よりオンラインリカバリを行うと、db_server3 の PostgreSQL がバックエンドに追加されたことが確認されました（図 13.26）。

図 13.26: db_server3 の PostgreSQL がスレーブサーバとして追加される

ログには以下のように出力されました。

```
(pgpool_server1)
2013-12-11 16:49:57 LOG:   pid 53491: reload config files.
2013-12-11 16:49:57 LOG:   pid 53491: Backend weight for backend2 changed from 0.000000 to 1.000000.
This will take effect from next client session.
2013-12-11 16:49:57 LOG:   pid 53539: reload config files.
```

```
2013-12-11 16:55:15 LOG:   pid 53500: send_failback_request: fail back 2 th node request from pid 53500
...
2013-12-11 16:55:15 LOG:   pid 53491: starting fail back. reconnect host 192.168.100.13(5432)
...
2013-12-11 16:55:16 LOG:   pid 53491: failback done. reconnect host 192.168.100.13(5432)
```

```
(pgpool_server2)
2013-12-11 16:50:02 LOG:   pid 54236: reload config files.
2013-12-11 16:50:02 LOG:   pid 54236: Backend weight for backend2 changed from 0.000000 to 1.000000.
This will take effect from next client session.
2013-12-11 16:50:02 LOG:   pid 54291: reload config files.
2013-12-11 16:54:21 LOG:   pid 54290: starting recovering node 2
...
2013-12-11 16:55:15 LOG:   pid 54290: send_failback_request: fail back 2 th node request from pid 54290
...
2013-12-11 16:55:15 LOG:   pid 54236: starting fail back. reconnect host 192.168.100.13(5432)
...
2013-12-11 16:55:16 LOG:   pid 54236: failback done. reconnect host 192.168.100.13(5432)
```

接続中のセッションには影響はありませんでした。

```
2013 年 12 月 11 日 水曜日 16:54:20 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:54:21 JST, SUCCESS
...
2013 年 12 月 11 日 水曜日 16:55:19 JST, SUCCESS
2013 年 12 月 11 日 水曜日 16:55:20 JST, SUCCESS
```

バックエンドサーバの削除

次に、db_server3 の PostgreSQL を pgpool-II のバックエンドから削除する検証を行いました。まず、db_server3 の PostgreSQL を停止させました。するとこれを検出した pgpool-II が PostgreSQL の切り離しを実行しました（図 13.25 の状態となる）。この操作では接続中のセッションは一旦切断されます。

```
2013 年 12 月 11 日 水曜日 17:00:38 JST, SUCCESS
2013 年 12 月 11 日 水曜日 17:00:39 JST, SUCCESS
2013 年 12 月 11 日 水曜日 17:00:40 JST, ERROR
2013 年 12 月 11 日 水曜日 17:00:42 JST, SUCCESS
2013 年 12 月 11 日 水曜日 17:00:43 JST, SUCCESS
```

その後、両 pgpool-II の pgpool.conf を編集して db_server3 の情報を削除しました。この設定の変更を反映させるため、全ての pgpool-II の再起動が必要です。まずスタンバイの pgpool_server1 を再起動しました。この操作では接続中のセッションは影響されません。最後にアクティブの pgpool_server2 を再起動しました。この操作では仮想 IP アドレスの切り替わりが発生するため、接続中のセッションは一時切断されます。

```
2013 年 12 月 11 日 水曜日 17:06:29 JST, SUCCESS
2013 年 12 月 11 日 水曜日 17:06:30 JST, SUCCESS
2013 年 12 月 11 日 水曜日 17:06:31 JST, ERROR
2013 年 12 月 11 日 水曜日 17:06:34 JST, SUCCESS
2013 年 12 月 11 日 水曜日 17:06:35 JST, SUCCESS
```

以上の操作で、db_server3 の情報は pgpool-II から削除され、pgpoolAdmin の画面に表示されなくなりました（図 13.7 の状態に戻る）。

まとめ

PostgreSQL のストリーミングレプリケーション機能、および pgpool-II の自動フェイルオーバと watchdog 機能を用いた高可用性構成を実機環境を用いて検証しました。その結果、想定される多くの PostgreSQL 障害ケースに際しては pgpool-II の自動フェイルオーバ機能により速やかに縮退運転へと移行できること、pgpool-II 障害に際しても watchdog 機能を用いた仮想 IP アドレスの自動切り替えによりサービスを継続提供できることが確認されました。また障害発生後の PostgreSQL サーバの復旧および、サーバの追加・削除などの運用が容易に行えることも確認されました。これらから、PostgreSQL サーバを複数台用いた高可用性システムを構築・運用する手段として本構成の有用性が高いことが示されました。

また、この構成では、接続中のセッションに影響を与えることなく、スレーブサーバを動的に追加できることが確認できました。この機能により、システム全体の運用を止めることなく読み取り負荷分散性能をスケールアウトすることが可能と言えます。

ストリーミングレプリケーションは PostgreSQL の組み込みの機能であり、watchdog 機能は pgpool-II の組み込みの機能です。そのため本構成には PostgreSQL と pgpool-II の他の外部ソフトウェアは必要なく、そのため設定が楽であるという利点があります。

ただし pgpool-II のコミュニティで公開しているインストーラでは本検証のような PostgreSQL と pgpool-II が別々のサーバにインストールされる構成には対応していなかったため、手動での追加設定が必要となりました。インストーラが本構成に対応することにより、導入の労力をより削減することができるでしょう。

13.1.2. 高負荷下での可用性を担保する機能の検証

ミッション・クリティカルを含むエンタープライズ領域での運用では、大規模データ処理による高負荷状態でも正常に運用できることが重要です。
そこで 13.1.1 項で実施した可用性を担保する機能の基礎検証に加えて、本項では CPU 負荷が高い状況での実機検証を行い、運用性を確認します。

検証対象は 13.1.1 項と同様に、2 台の PostgreSQL によるストリーミングレプリケーションと、pgpool-II の自動フェイルオーバ機能と watchdog 機能を組み合わせた PostgreSQL 冗長化構成です。
本検証の目的は、この構成で 2 台の pgpool-II に CPU 負荷をかけた状況での、自動フェイルオーバ機能と watchdog 機能の運用性を評価することにあります。

検証環境

(1)マシン構成
検証に用いた環境のネットワーク構成を図 13.27 に示します。

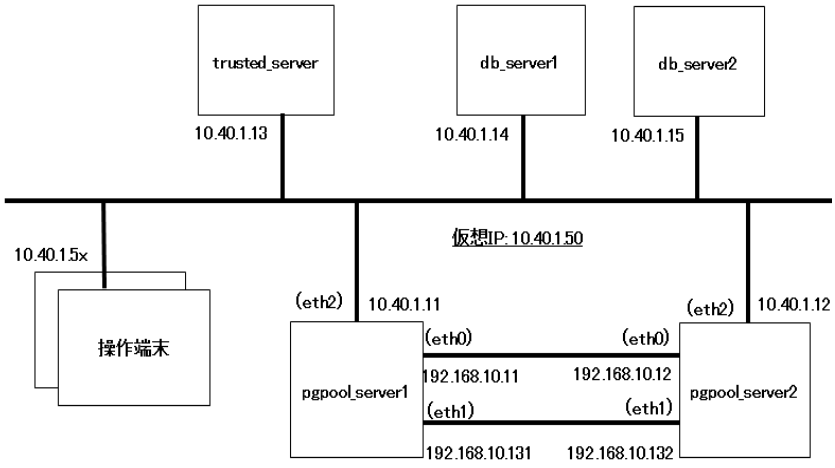


図 13.27: ネットワーク構成

また図 13.27 に示す各マシンのホスト名、役割、および使用した IP アドレスの一覧を表表 13.3 に、スペック一覧を表 13.4 に示します。

表 13.3: マシン一覧

ホスト名	役割	IP アドレス
pgpool_server1	pgpool-II サーバ	192.168.10.11, 192.168.10.131, 10.40.1.11
pgpool_server2	pgpool-II サーバ	192.168.10.12, 192.168.10.132, 10.40.1.12
	(pgpool-II 用仮想 IP アドレス)	10.40.1.50
trusted_server	上位サーバ	10.40.1.13
db_server1	PostgreSQL サーバ	10.40.1.14
db_server2	PostgreSQL サーバ	10.40.1.15

表 13.4: サーバスペック一覧

検証環境項目		内容
pgpool-II サーバ PostgreSQL サーバ	ハードウェア	CPU ：Intel® Xeon® E5-2690(2.90GHz 8core) 2 Processor / 合計 16Core Memory：4GB × 2 内臓 HDD：600GB × 2 ネットワークカード：DualPortLAN カード(10GBASE)
	OS	Redhat Enterprise Linux 6.4 (for Intel64)
	PostgreSQL バージョン	PostgreSQL 9.3.1
	pgpool-II バージョン	pgpool-II3.3.2
上位サーバ	ハードウェア	CPU ：Intel® Xeon® E5-2690(2.90GHz 8core) 2 Processor / 合計 16Core Memory：4GB × 2 内臓 HDD：600GB × 2 ネットワークカード：DualPortLAN カード(10GBASE)
	OS	Redhat Enterprise Linux 6.4 (for Intel64)
	PostgreSQL バージョン	PostgreSQL 9.3.1

表 13.4 の役割にある「上位サーバ」とは pgpool-II のクライアントと同じ LAN 上にあるサーバを指します。また本検証でも、13.1.1 項の検証と同様に、上位サーバは pgpool-II に接続するクライアントの役割も兼ねています。具体的には以下のスクリプトを上位サーバ上で実行し定期的に更新系 SQL クエリ（INSERT 文）を発行することで、pgpool-II への接続エラーの有無をチェックしました。

```
QUERY="INSERT INTO pgbench_history VALUES (1,2,3,4,NOW(),'hoge')"
PSQL=/usr/local/pgsql/bin/psql
HOST=10.40.1.50

while true
do
  val1=$(date)
  val2=$(PGCONNECT_TIMEOUT=1 $PSQL -h $HOST -p 9999 -U postgres -d postgres -q -t -c "$QUERY")
  if [ $? -ne 0 ]; then
    val2="ERROR"
  else
    val2="SUCCESS"
  fi
  echo "${val1},${val2}"
  sleep 1
done
```

（2）環境構築

PostgreSQL 環境は以下のとおりです。

```
# PostgreSQL 起動ユーザ：postgres
# データディレクトリ：/var/lib/pgsql/data
# アーカイブファイルディレクトリ：/var/lib/pgsql/archivedir
# pgpool ログディレクトリ：/var/log/pgpool
```

pgpool-II の設定ファイルである pgpool.conf の記載内容を抜粋します。

```
# バックエンドのサーバの設定
backend_hostname0 = '10.40.1.14'
backend_port0 = 5432
backend_weight0 = 1
backend_data_directory0 = '/var/lib/pgsql/data'
#backend_flag0 = 'ALLOW_TO_FAILOVER'

backend_hostname1 = '10.40.1.15'
backend_port1 = 5432
backend_weight1 = 1
backend_data_directory1 = '/var/lib/pgsql/data'
#backend_flag1 = 'ALLOW_TO_FAILOVER'

# 上位サーバの設定
trusted_servers = '10.40.1.13'

# 仮想 IP アドレスの設定
delegate_IP = '10.40.1.50'

# 自ホストの設定（pgpool_server1 の場合）
wd_hostname = '10.40.1.11'

# 相手ホストの設定（pgpool_server1 の場合）
other_pgpool_hostname0 = '10.40.1.12'
other_pgpool_port0 = 9999
other_wd_port0 = 9000

# ハートビート信号の送り先（pgpool_server1 の場合）
heartbeat_destination0 = '192.168.10.132'
heartbeat_destination_port0 = 9694
heartbeat_device0 = ''

heartbeat_destination1 = '192.168.10.12'
heartbeat_destination_port1 = 9694
heartbeat_device1 = ''

# ハートビート信号の送り先（pgpool_server2 の場合）
heartbeat_destination0 = '192.168.10.131'
heartbeat_destination_port0 = 9694
heartbeat_device0 = ''

heartbeat_destination1 = '192.168.10.11'
heartbeat_destination_port1 = 9694
heartbeat_device1 = ''
```

PostgreSQL は pgbench で初期化します。

```
$ pgbench -i
```

検証ケース

本項では検証ケースの概要を説明します。
なお本検証では、PostgreSQL サーバ、pgpool-II サーバのアクティブ・スタンバイともに CPU 負荷をかけた状況で検証するため、マシン単体に直接負荷を掛けられる stress ツールを使用しました。本検証で使用したマシン OS は Redhat Enterprise Linux 6.4 のため、以下の rpm パッケージをインストールしました。

```
$ rpm -ivh stress-1.0.2-1.el6.rf.i686.rpm
```

各検証で 4 台とも以下の stress コマンドを 10 分間実行し、CPU 負荷を掛けている状態としています。

```
$ stress –cpu 33 –timeout 3600s
```

sar コマンドで CPU 利用状況を確認すると、PostgreSQL サーバ、pgpool-II サーバともに利用状況(%user 値)はほぼ 100%近くを指しています。結果の一部を抜粋します。

```
14 時 28 分 38 秒    CPU    %user    %nice  %system   %iowait   %steal    %idle
14 時 28 分 43 秒    all     99.99     0.00     0.01      0.00     0.00     0.00
14 時 28 分 48 秒    all     99.99     0.00     0.01      0.00     0.00     0.00
14 時 28 分 53 秒    all     99.99     0.00     0.01      0.00     0.00     0.00
14 時 28 分 58 秒    all     99.99     0.00     0.01      0.00     0.00     0.00
14 時 29 分 03 秒    all    100.00     0.00     0.00      0.00     0.00     0.00
14 時 29 分 08 秒    all     99.99     0.00     0.01      0.00     0.00     0.00
14 時 29 分 13 秒    all    100.00     0.00     0.00      0.00     0.00     0.00
```

検証は、以下 5 つのケースを実施しました。

(1)　高負荷状況での pgpool-II の挙動

更新系 SQL 文(INSERT)を実行し、数分間、定期的に pgpool-II の状態を監視します。

(2)　高負荷状況での PostgreSQL プロセス障害時の挙動

PostgreSQL のプロセスが異常停止した場合の挙動を検証します。

(3)　高負荷状況での PostgreSQL プロセス復帰時の挙動

異常停止した PostgreSQL のプロセスが復帰した場合の挙動を検証します。

(4)　高負荷状況での pgpool-II プロセス障害時の挙動

pgpool-II のプロセスが異常停止した場合の挙動を検証します。

(5)　高負荷状況での pgpool-II サーバ障害時の挙動

pgpool-II サーバの電源が落ちた場合の挙動を検証します。

検証結果

本検証では、db1_server, db2_server の PostgreSQL はそれぞれマスタサーバ、スレーブサーバとして稼働しており、pgpool_server1, pgpool_server2 の pgpool-II はそれぞれアクティブ、スタンバイで起動している状態になっています。

(1)　高負荷状況での pgpool-II の挙動

まず、pgpool-II が現在の PostgreSQL の状態をどのように認識しているかを確認します。

```
$ show pool_nodes;
 node_id |  hostname  | port | status | lb_weight |  role
---------+------------+------+--------+-----------
 0       | 10.40.1.14 | 5432 | 2      | 0.500000  | primary
 1       | 10.40.1.15 | 5432 | 2      | 0.500000  | standby
(2 行)
```

”show pool_nodes”は、PostgreSQL の状態を表示します。”status”の値が 1 または 2 の場合問題なし、3 の場合ダウン状態となります。また”role”の値から、db1_server はマスタサーバ、db2_server はスレーブサーバとして問題なく稼働していることが確認できます。

また pgpool-II の稼働状況は、pcp_watchdog_info コマンドで確認することができます。pcp_watchdog_info コマンドの出力結果は順に、pgpool-II のホスト名、pgpool-II ポート番号、watchdog ポート番号、watchdog ステータスを表しています。また、ステータスの値は、1: 初期状態（指定された pgpool-II が未起動の場合に表示される） 2: スタンバイ 3: アクティブ 4: ダウン を表します。

```
$ pcp_watchdog_info 0 pgpool_server1 9898 postgres postgres
10.40.1.11 9999 9000 3
$ pcp_watchdog_info 0 pgpool_server2 9898  postgres postgres
10.40.1.12 9999 9000 2
```

pgpool_server1 がアクティブ、pgpool_server2 がスタンバイで稼動していることが確認できます。
CPU 負荷を掛けている間、pgpool-II の出力ログに問題は確認されず、接続中のセッションにも影響はみられませんでした。

```
2013 年 12 月 20 日 金曜日 13:47:30 JST,SUCCESS
2013 年 12 月 20 日 金曜日 13:47:31 JST,SUCCESS
…
2013 年 12 月 20 日 金曜日 13:53:30 JST,SUCCESS
2013 年 12 月 20 日 金曜日 13:53:31 JST,SUCCESS
```

(2)　高負荷状況での PostgreSQL プロセス障害時の挙動

show コマンドの出力結果から、db1_server はマスタサーバ、db2_server はスレーブサーバとして稼働していることが確認できます。

```
$ show pool_nodes;
 node_id |  hostname  | port | status | lb_weight |  role
---------+------------+------+--------+-----------
 0       | 10.40.1.14 | 5432 | 2      | 0.500000  | primary
 1       | 10.40.1.15 | 5432 | 2      | 0.500000  | standby
(2 行)
```

サービス中に PostgreSQL プロセスが異常停止した状況をシミュレートするため、上位サーバで SQL 実行スクリプトを実行中に db1_server の PostgreSQL プロセスを停止します。PostgreSQL の停止は pg_ctl コマンドを実施しました。

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

pgpool-II が PostgreSQL のダウンを検出し、フェイルオーバがされることをログ出力で確認されました。

```
2013-12-20 14:13:43 LOG:   pid 5809: degenerate_backend_set: 0 fail over request from pid 5809
…
2013-12-20 14:13:43 LOG:   pid 30179: starting degeneration. shutdown host 10.40.1.14(5432)
2013-12-20 14:13:43 LOG:   pid 30179: Restart all children
2013-12-20 14:13:43 LOG:   pid 30179: execute command: /usr/local/etc/failover.sh 0 10.40.1.14 5432
/var/lib/pgsql/data 1 0 10.40.1.15 0 5432 /var/lib/pgsql/data
…
2013-12-20 14:13:50 LOG:   pid 30179: failover done. shutdown host 10.40.1.14(5432)
```

また show コマンドで db1_server の”status”の値が 3 を示しており、ダウン状態であることが確認されました。

```
$ show pool_nodes;
 node_id |  hostname  | port | status | lb_weight |  role
---------+------------+------+--------+-----------
 0       | 10.40.1.14 | 5432 | 3      | 0.500000  | standby
 1       | 10.40.1.15 | 5432 | 2      | 0.500000  | primary
(2 行)
```

接続中のセッションは一度切断されますが、切り離し終了後に接続が可能になります。

```
2013 年 12 月 20 日 金曜日 14:13:42 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:13:43 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:13:44 JST,ERROR
2013 年 12 月 20 日 金曜日 14:13:47 JST,ERROR
2013 年 12 月 20 日 金曜日 14:13:50 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:13:51 JST,SUCCESS
```

(3)　高負荷状況での PostgreSQL プロセス復帰時の挙動

検証(2)で切り離された旧マスタサーバを、オンラインリカバリでクラスタに復帰させます。db1_server の PostgreSQL プロセスを停止した状態で、pcp_recovery コマンドを実行します。
pgpool_server のログに以下のように出力され、オンラインリカバリが実行されたことが確認されました。

```
2013-12-20 14:30:16 LOG:   pid 6620: starting recovering node 0
2013-12-20 14:30:16 LOG:   pid 6620: starting recovery command: "SELECT pgpool_recovery('basebackup-
stream.sh', '10.40.1.14', '/var/lib/pgsql/data')"
…
2013-12-20 14:30:33 LOG:   pid 6620: send_failback_request: fail back 0 th node request from pid 6620
…
2013-12-20 14:30:34 LOG:   pid 30179: starting fail back. reconnect host 10.40.1.14(5432)
…
2013-12-20 14:30:35 LOG:   pid 30179: failback done. reconnect host 10.40.1.14(5432)
2013-12-20 14:30:35 LOG:   pid 6621: worker process received restart request
2013-12-20 14:30:35 LOG:   pid 6620: recovery done
```

また show コマンドで db1_server の”status”の値が 2 を示しており、スレーブサーバとして復帰したことが確認されました。

```
$ show pool_nodes;
 node_id |  hostname  | port | status | lb_weight |  role
---------+------------+------+--------+-----------
 0       | 10.40.1.14 | 5432 | 2      | 0.500000  | standby
 1       | 10.40.1.15 | 5432 | 2      | 0.500000  | primary
(2 行)
```

この操作による接続中のセッションには影響はみられませんでした。

```
2013 年 12 月 20 日 金曜日 14:26:36 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:26:37 JST,SUCCESS
…
2013 年 12 月 20 日 金曜日 14:32:03 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:32:04 JST,SUCCESS
```

(4)　高負荷状況での pgpool-II プロセス障害時の挙動

この段階では db1_server はスレーブサーバ、db2_server はマスタサーバとして稼動しています。
また pgpool_server1 でアクティブ pgpool-II、pgpool_server2 でスタンバイ pgpool-II が稼働している状態となっています。

```
$ pcp_watchdog_info 0 pgpool_server1 9898 postgres postgres
10.40.1.11 9999 9000 3
$ pcp_watchdog_info 0 pgpool_server2 9898  postgres postgres
10.40.1.12 9999 9000 2
```

pgpool-II のプロセス障害を killall コマンドでシミュレートしました。

```
$ killall -9 pgpool
```

すると、pgpool_server1 の pgpool-II が親プロセスの異常を検出しダウン状態に移行し、pgpool_server2 の pgpool-II がそれを検出しアクティブ pgpool-II に昇格しました。

```
$ pcp_watchdog_info 0 pgpool_server1 9898 postgres postgres
10.40.1.11 9999 9000 4
$ pcp_watchdog_info 0 pgpool_server2 9898  postgres postgres
10.40.1.12 9999 9000 3
```

ログには以下のように出力されます。

```
2013-12-20 14:59:16 LOG:   pid 28558: wd_escalation: escalating to master pgpool
2013-12-20 14:59:16 LOG:   pid 28558: wd_escalation: clear all the query cache on shared memory
2013-12-20 14:59:16 LOG:   pid 28558: pool_shared_memory_cache_size: number of blocks: 64
WARNING: interface is ignored: Operation not permitted
…
2013-12-20 14:59:19 LOG:   pid 28558: wd_escalation: escalated to master pgpool successfully
…
2013-12-20 14:59:31 LOG:   pid 28558: check_pgpool_status_by_hb: pgpool 1 (10.40.1.11:9999) is in down
status
```

しかし、セッションが復帰できませんでした。これは、pgpool-II を停止させる際に killall コマンドで全プロセスを終了したので、仮想 IP アドレスを停止させる処理が起動しなかったことが原因と考えられます。

```
2013 年 12 月 20 日 金曜日 14:58:58 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:58:59 JST,SUCCESS
2013 年 12 月 20 日 金曜日 14:59:00 JST,ERROR
2013 年 12 月 20 日 金曜日 14:59:01 JST,ERROR
2013 年 12 月 20 日 金曜日 14:59:02 JST,ERROR
2013 年 12 月 20 日 金曜日 14:59:03 JST,ERROR
2013 年 12 月 20 日 金曜日 14:59:04 JST,ERROR
2013 年 12 月 20 日 金曜日 14:59:05 JST,ERROR
…
2013 年 12 月 20 日 金曜日 15:02:36 JST,ERROR
```

ifconfig コマンドにより、pgpool_server1 では仮想 IP アドレスが停止されず、両 pgpool-II サーバで仮想 IP アドレスが立ち上がっていることを確認しました。

```
(pgpool_server1)
eth2      Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.11  Bcast:10.40.1.255  Mask:255.255.255.0

eth2:0    Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.50  Bcast:10.40.1.255  Mask:255.255.255.0

(pgpool_server2)
eth2      Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.11  Bcast:10.40.1.255  Mask:255.255.255.0

eth2:0    Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.50  Bcast:10.40.1.255  Mask:255.255.255.0
```

全プロセスが一度にダウンすることは、業務システムの運用上考えにくいことです。プロセスを終了させる場合には、pgpool-II の仕様に従った運用(コマンド例 $pgpool -m fast stop)を行うよう、注意する必要があります。

(5)　高負荷状況での pgpool-II サーバ障害時の挙動

pgpool_server1 でスタンバイ pgpool-II、pgpool_server2 でアクティブ pgpool-II が稼働している状態となっています。

```
$ pcp_watchdog_info 0 pgpool_server1 9898 postgres postgres
10.40.1.11 9999 9000 2
$ pcp_watchdog_info 0 pgpool_server2 9898  postgres postgres
10.40.1.12 9999 9000 3
```

アクティブ pgpool-II サーバの電源を落とすと、pgpool_server2 の pgpool-II が親プロセスの異常を検出しダウン状態に移行し、pgpool_server1 の pgpool-II がそれを検出しアクティブ pgpool-II に昇格しました。

```
$ pcp_watchdog_info 0 pgpool_server1 9898 postgres postgres
10.40.1.11 9999 9000 3
$ pcp_watchdog_info 0 pgpool_server2 9898  postgres postgres
10.40.1.12 9999 9000 4
```

ログには以下のように出力されます。

```
2013-12-20 15:41:18 LOG:   pid 10582: wd_escalation: escalating to master pgpool
2013-12-20 15:41:18 LOG:   pid 10582: wd_escalation: clear all the query cache on shared memory
2013-12-20 15:41:18 LOG:   pid 10582: pool_shared_memory_cache_size: number of blocks: 64
WARNING: interface is ignored: Operation not permitted
…
2013-12-20 15:41:20 LOG:   pid 10582: wd_escalation: escalated to master pgpool successfully
…
2013-12-20 15:41:32 LOG:   pid 10582: check_pgpool_status_by_hb: pgpool 1 (10.40.1.12:9999) is in down
status
```

pgpool_server2 では仮想 IP アドレスが停止され、pgpool_server1 では仮想 IP アドレスが立ち上がりました。

```
(pgpool_server1)
eth2      Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.11  Bcast:10.40.1.255  Mask:255.255.255.0

eth2:0    Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.50  Bcast:10.40.1.255  Mask:255.255.255.0

(pgpool_server2)
eth1      Link encap:Ethernet  HWaddr 00:19:99:C2:E3:09
          inet addr:192.168.10.131  Bcast:192.168.10.255  Mask:255.255.255.128

eth2      Link encap:Ethernet  HWaddr 00:19:99:A1:3E:12
          inet addr:10.40.1.11  Bcast:10.40.1.255  Mask:255.255.255.0
```

仮想 IP アドレスの切り替わりにより、接続中のセッションは一時切断されました。

```
2013 年 12 月 20 日 金曜日 15:40:44 JST,SUCCESS
2013 年 12 月 20 日 金曜日 15:40:45 JST,SUCCESS
2013 年 12 月 20 日 金曜日 15:40:46 JST,ERROR
2013 年 12 月 20 日 金曜日 15:40:49 JST,ERROR
2013 年 12 月 20 日 金曜日 15:40:52 JST,ERROR
2013 年 12 月 20 日 金曜日 15:40:55 JST,ERROR
2013 年 12 月 20 日 金曜日 15:40:58 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:01 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:04 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:07 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:10 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:13 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:16 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:19 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:22 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:25 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:28 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:31 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:34 JST,ERROR
2013 年 12 月 20 日 金曜日 15:41:37 JST,SUCCESS
```

検証(5)では、セッションの回復に約 50 秒掛かりました。この理由として、障害検出に関する設定が関係していると考えられます。

本検証環境における pgpool.conf の設定は、死活監視を 10 秒間隔で行い（wd_interval ）、障害が発生したとみなす閾値が 30 秒（wd_heartbeat_deadtime）であるため、障害検出に最大で 40 秒程度の時間が掛かる計算となります。

```
wd_interval = 10
wd_heartbeat_deadtime = 30
```

まとめ

本項では、CPU 負荷をかけた状況での、pgpool-II の自動フェイルオーバ機能と watchdog 機能の運用性を検証しました。

その結果、CPU 負荷をかけた状況でも、PostgreSQL サーバと pgpool-II サーバを強制停止させた際に、自動フェイルオーバ機能により速やかに縮退運転へと移行できること、また watchdog 機能を用いた仮想 IP アドレスの自動切り替えによりサービスを継続提供できることが確認できました。またフェイルオーバにかかる時間は、pgpool.conf の watchdog のハートビート受信に関するパラメータ値によるものが大きいので、目的に合った設定を行う必要があります。

最後に本検証では、killall コマンドで全プロセスを一度に終了させたため、セッション復帰ができないという現象となりました。プロセスを終了させる場合には、pgpool-II の仕様に従った運用を行うよう注意する必要があります。

13.2. バックアップ／リカバリ検証

バックアップに関する検証は、運用シナリオにもとづいた検証をシングル構成とストリーミングレプリケーション構成の2つの構成で実施しました。ここではどのような検証を行ったかという内容の紹介を行います。検証内容の詳細や手順については別冊の『バックアップ検証（シングルサーバ編）』、『バックアップ検証（SR 編）』をご参照ください。

13.2.1. シングル構成

シングル構成の検証は以下の 3 つのシナリオで行いました。

1 つ目のシナリオでは は更新中のサーバに電源断がおこり、再起動したときに WAL からの自動ロールフォワードで復旧することを確認します。

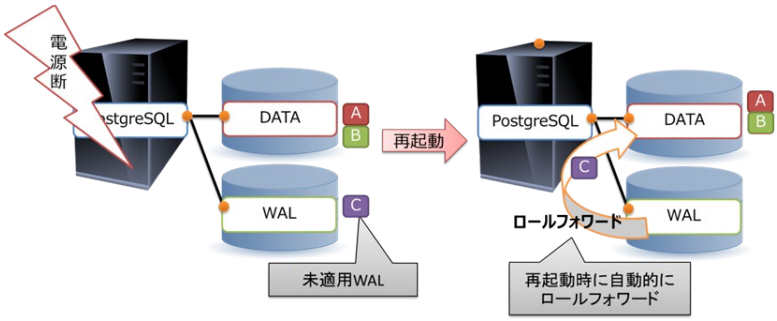


図 13.28: クラッシュリカバリシナリオ

2 つ目のシナリオでは pg_dump でシステムバックアップをとったシステムのデータに異常が発生したとき、pg_restore を利用してデータの復旧を行います。

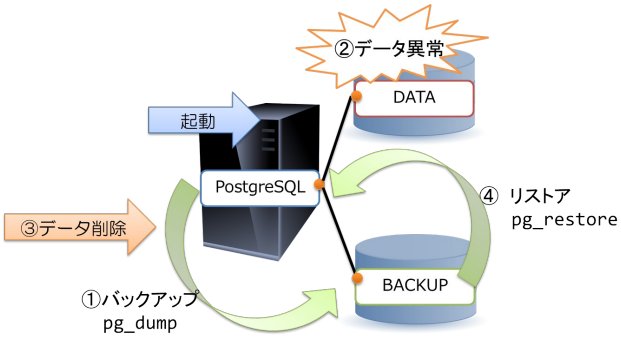


図 13.29: 論理バックアップからのリストアシナリオ

３つ目はアーカイブログを運用しているシステムでオペレーションミスによりデータを削除されて、Point in time recovery(PITR)を利用してデータを復旧します。

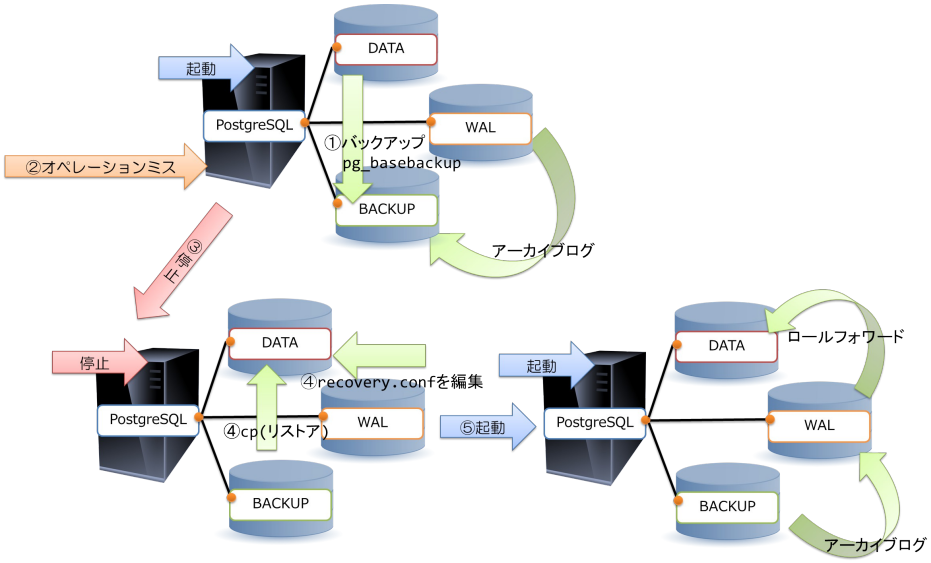


図 13.30: Point in time recovery シナリオ

以上についての手順や注意点を『バックアップ検証（シングルサーバ編）』に記載していますので、ご参照ください。

13.2.2. ストリーミングレプリケーション構成におけるバックアップ／リカバリ

ストリーミングレプリケーション構成のバックアップ／リカバリ検証は、以下のような環境で行いました。検証環境の構成は、2 台構成のストリーミングレプリケーション構成で、スレーブサーバ側でバックアップを取得するという運用を想定しています。

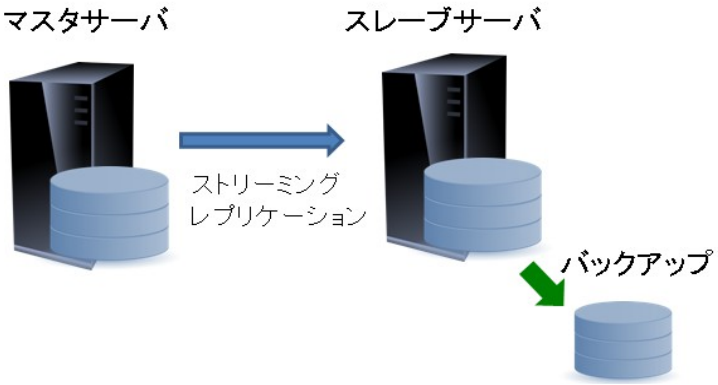


図 13.31　検証構成イメージ

この環境でマスタサーバで障害が発生した状況からの復旧シナリオについて、2 つのケースの検証を行いました。
1 つ目は、マスタサーバでハードウェア障害が発生し、スレーブサーバをマスタに昇格させて業務を復旧させ、その後ストリーミングレプリケーション構成を再構成するというシナリオです。ストリーミングレプリケーションを再構成する際に業務をほぼ止めずに行うようにするため、次の図のような流れで復旧を行いました（図中の SR はストリーミングレプリケーションの略）。

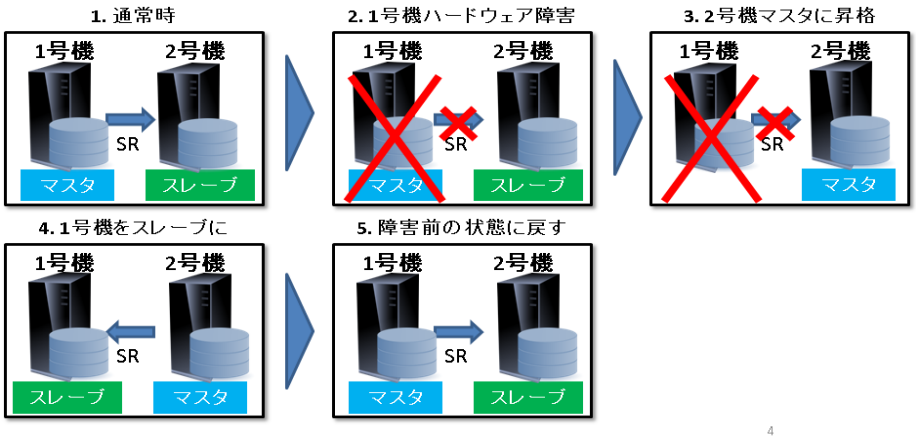


図 13.32　障害復旧シナリオイメージ

２つ目は、オペレーションミスによりデータを消去してしまったため、スレーブサーバで取得していたバックアップを使用してマスタサーバをデータ消去直前までリカバリし、その後ストリーミングレプリケーション構成を再構成するというシナリオです。

以上についての手順や注意点を『バックアップ検証（SR 編）』に記載していますので、ご参照ください。

13.3. 監視ケーススタディ

本節では、監視ケーススタディとして、問題発見から収集した情報やエラーメッセージの分析、対処方針案の策定および効果の確認を行う流れをまとめます。

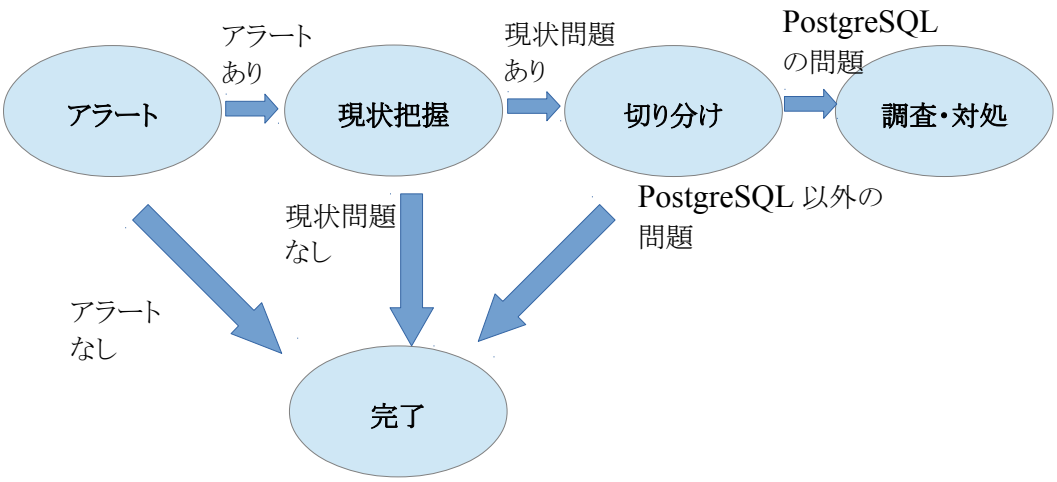
13.3.1. 全体の流れ

本ケーススタディでは、大きく以下の４つの作業を想定し、各作業でどのような点を念頭に置くとよいかについて記述します。

表 13.5: 問題発見後の作業項目

作業項目	作業内容
アラート	異常を検知することです。本ケーススタディではアラートの検知は、何かしらの監視ツールを用いている前提で記述します
現状把握	現在、どのような事象が発生しているのか、どの程度の影響を受けているのかを確認し、早急な対処が必要かどうかを判断します
切り分け	現状に何かしらの問題が残っている場合、どの部分で問題が生じているのかを切り分けていきます
調査・対処	切り分けの結果、問題の原因が PostgreSQL にあると判断した場合、さらに PostgreSQL のどの部分で問題が発生しているのかを調査し、調査結果に応じた対処を行っていきます

作業のフローは以下のようになります。



13.3.2. 状況に応じた事例

(1)「ディスク容量不足」のケース

前項で記述したフローに合わせて、ディスク容量が不足した場合の作業の流れをまとめます。

【アラート】
ディスク使用率が超過したことを示すアラートがあがった。

【現状把握】
アラートを受け、現状把握として現在のディスク使用率を df コマンドを用いて実測した。

```
$ df -h
```

【切り分け】
現状把握の結果、あるデバイスの使用率が閾値を超えていることが確認できた。
そのデバイス上にはデータベースクラスタが配置されていたため、下記 SQL 関数等を用いてデータベースやテーブルのサイズを確認した。

```
# SELECT pg_database_size('mydb');
```

```
# SELECT pg_relation_size('mytbl');
```

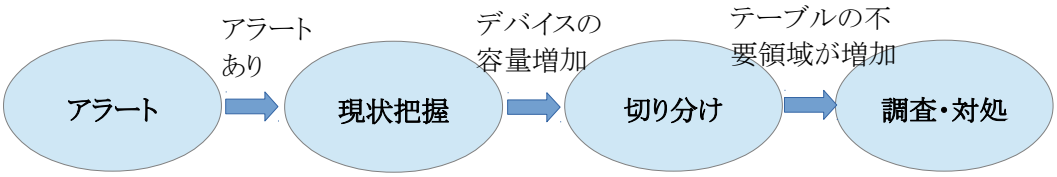
【調査・対処】
切り分けの結果、mytbl が必要以上に肥大化していることが確認できた。さらに下記 SQL 関数で肥大化している原因を調査した。この結果、不要な領域が大半を占めていることが確認できた。

```
# SELECT n_dead_tup FROM pg_stat_user_tables WHERE relname =
'mytbl';
```

このことから、何故不要な領域が増加したのかを調査することとなるが、このままの運用を続けるとサービス全体への影響が大きくなると考えられた。このため、根本原因への対処は別途行うこととし、暫定対処としてサービスを一時的に停止し、対象のテーブルの論理バックアップを異なるデバイス上に取得し、リストアを行い、不要な領域を削除することとした。

```
# pg_dump -Fc -t mytbl mydb > /dev2/mytbl.dmp
# psql mydb -c “DROP TABLE mytbl”
# pg_restore -d mydb /dev2/mytbl.dmp
```

暫定対処の後、現状把握を行い適切な容量となっていることが確認できた。



(2)「コネクションの確立に失敗する」ケース１

【アラート】
SQL による死活監視に失敗したことを示すアラートがあがった。また、ログに「 FATAL: remaining connection slots are reserved for non-replication superuser connections」というエラーが出力されたことを示すアラートもあわせてあがった。

【現状把握】
ログ監視のアラート内容から、コネクション数の上限に達したためコネクションの確立に失敗している可能性が高い。アラートを受けて、手動で SQL 監視を再実行した結果、ログ監視の結果と同じエラー応答が得られた。

```
$ psql -h 127.0.0.1 -p 5432 -U monitor -d test -c 'SELECT 1;'
psql: FATAL:  remaining connection slots are reserved for non-replication superuser
connections
```

【切り分け】
エラー内容から superuser_reserved_connections の枠は残っていることが予想されたため、postgres ユーザでデータベースへの接続を試みたところ、コネクションの確立に成功した。その後、下記の SQL を実行して現在接続中のコネクションの状態を確認した。

```
postgres=# SELECT usename, count(*) FROM pg_stat_activity GROUP BY usename;
 usename     | count
-------------+-------
 postgres    |     1
 application |    97
```

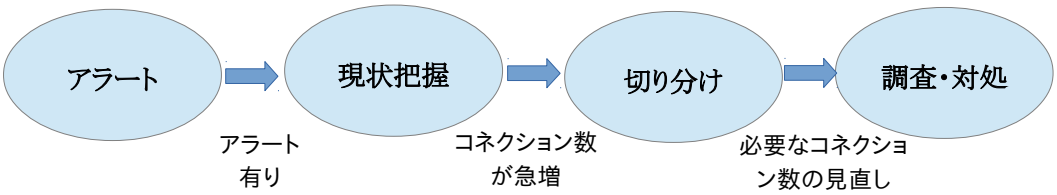
```
postgres=# SELECT * FROM pg_stat_activity;
-[ RECORD 1 ]----+--------------------------------
datid            | 12896
datname          | postgres
pid              | 6520
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      | 127.0.0.1
client_hostname  |
client_port      | 54377
backend_start    | 2014-02-25 10:43:41.334028+09
xact_start       | 2014-02-25 13:38:16.549726+09
query_start      | 2014-02-25 13:38:16.549726+09
state_change     | 2014-02-25 13:38:16.54973+09
waiting          | f
state            | active
query            | SELECT * from pg_stat_activity;
-[ RECORD 2 ]----+--------------------------------
( 後略 )
```

また以下の SQL を実行し、現在適用されている設定内容を確認した。

```
postgres=# show max_connections; show superuser_reserved_connections;
-[ RECORD 1 ]---+----
max_connections | 100

-[ RECORD 1 ]------------------+--
superuser_reserved_connections | 3
```

【調査・対処】
切り分けの結果、コネクション数が上限に達していることが確認された。さらにコネクションの内容を確認した所、state が 'idle in transaction' となったまま長時間経過しているロングトランザクションや、waiting が 't' となっているロック開放待ちのコネクションは見受けられなかった。client_addr が想定していない接続元となっているコネクションも見受けられなかった。そのため、何らかの理由で必要なコネクション数が max_connections を超えてしまったと考えられる。
上記の SQL の結果を保存した後、暫定対応として max_connections の値をより大きな値に設定して PostgreSQL の再起動を行い、新たな接続が可能になったことを確認した。また後日、使用される可能性のあるコネクション数を再調査し、設定内容を見直した。



(3)「コネクションの確立に失敗する」ケース２

【アラート】
psql からの接続に失敗する事象が発生した。その際のメッセージは以下のようであった。

```
psql: could not connect to server: No such file or directory
        Is the server running locally and accepting
        connections on Unix domain socket "/tmp/.s.PGSQL.5432"?
```

【現状把握】
現状把握のため、複数のクライアント機から同様の接続を試みたが、すべて上記と同じ結果となった。
早急な対処が必要なため、問題の切り分け、対処を実施した。

【切り分け】
まずは、ping コマンドで、サーバおよびクライアント/サーバ間の NW の状態を確認したところ、問題はなかった。

```
$ ping db-server
```

次に、ps コマンドでサーバ上の PostgreSQL プロセスの存在有無を確認したところ、プロセスが存在していないことが分かった。

```
$ ps -ef | grep postgres
```

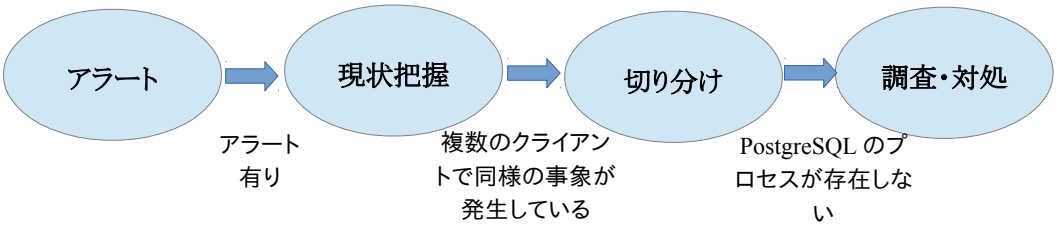
上記より、PostgreSQL が何らかの理由で起動できなかったことが問題である、と考えた。

【調査・対処】
切り分けの結果、PostgreSQL の起動に問題があったと考えられるため、PostgreSQL のログファイルを確認し、調査を行った。ログには以下のメッセージが記録されていた。

```
< 2013-12-23 10:25:15.138 JST >LOG:  redirecting log output to logging collector process
< 2013-12-23 10:25:15.138 JST >HINT:  Future log output will appear in directory
"pg_log".
< 2013-12-24 14:30:27.904 JST >FATAL:  data directory "/var/lib/pgsql/data" has group
or world access
< 2013-12-24 14:30:27.904 JST >DETAIL:  Permissions should be u=rwx (0700).
```

上記、FATAL、DETAIL のメッセージよりデータベースクラスタの権限が 700 以外に設定されているために PostgreSQL を起動できなかったことが原因と判断した。
調査結果から、データベースクラスタの権限を 700 に設定後、PostgreSQL を起動し対処とした。

```
$ chmod -R 700 /var/lib/pgsql/data
$ pg_ctl -D /var/lib/pgsql/data start
```



(4)「キャッシュヒット率が低い」ケース

【アラート】
テーブルのキャッシュヒット率が低いというアラートがあがった。

【現状把握】
アラートを受け、現状把握としてテーブルのキャッシュヒット率を確認する。

```
SELECT relname,
       round(heap_blks_hit*100/(heap_blks_hit+heap_blks_read), 2) AS cache_hit_ratio
FROM pg_statio_user_tables
WHERE heap_blks_read > 0
ORDER BY cache_hit_ratio;
```

上記 SQL を実施したところ、下記のような結果が得られた。

```
       relname       | cache_hit_ratio
---------------------+-----------------
 tbl_order           |           52.00
 tbl_sales           |           65.00
 tbl_stock           |           96.00
 mst_user            |           97.00
 mst_customer        |           98.00
 mst_shop            |           99.00
 mst_warehouse       |           99.00
 ...
```

【切り分け】
現状把握の結果、いくつかのテーブルに対してキャッシュが上手く機能していないことがわかった。
まずは postgresql.conf のパラメータを確認し、適切な値が設定されているかを確認する。

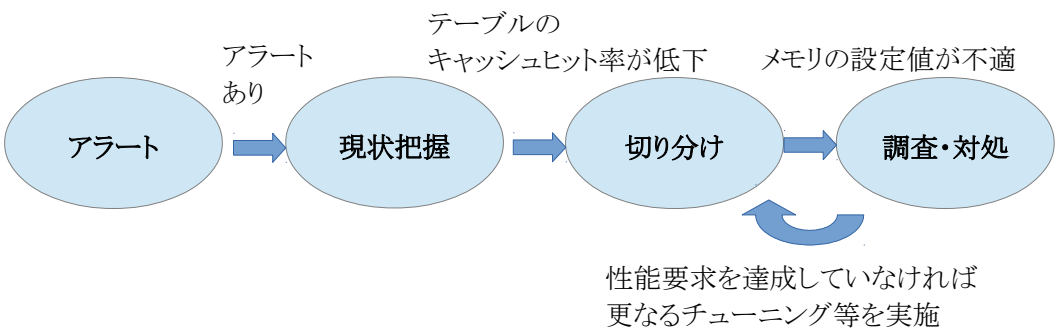
```
shared_buffers = 32MB
```

```
work_mem = 1MB
```

【調査・対処】
切り分けの結果、shared_buffers と work_mem の設定値が初期値のままであることが判明した。そのため SQL やアプリケーションのチューニングを行う前に、一般的とされる値を用いてパラメータの設定を行う。shared_buffers には実メモリの 20～25%を設定する。次に work_mem には 8MB を初期値として与える。その後 PostgreSQL を再起動し、postgresql.conf の設定変更を反映させる。

```
pg_ctl restart
```

この後アラート要因となったテーブルを参照する SQL を一定期間動作させ、キャッシュヒット率が閾値を達成していることが確認できた。今回は簡潔なパラメータチューニングのみで対処を行ったが、性能要求が達成できていない場合は詳細なパラメータチューニングや SQL チューニング、アプリケーションの改修が必要となる場合がある。



(5) 「HA クラスタの更新性能が低下した」ケース

【アラート】
昨晩実行されたデータベースへの更新を行うバッチの処理時間が、想定よりも長かったという報告を受けた。

【現状把握】
実際にログに記録された実行時間を確認したところ、確かに通常時よりも有意に処理時間が増加していた。バッチ処理の内容は特に変更されておらず、処理するデータ量も通常時と大きな違いは無かった。

【切り分け】
本環境では、ストリーミングレプリケーションを用いた冗長構成を取っていた。そのためデータベースへの更新処理はマスタサーバ上で実行されることになる。そこでまずバッチ処理実行時点でどのサーバがマスタサーバであったかを確認することにした。
PostgreSQL には現在、直接対象のサーバがマスタ/スレーブのどちらであるかを返す関数やコマンドは用意されていない。だが以下に挙げるいくつかの関数や SQL 等の結果からストリーミングレプリケーションの状態を知ることができる。

1. pg_is_in_recovery 関数の戻り値
ストリーミングレプリケーションを実行中のスレーブサーバでは、常時リカバリ処理を行っている状態となる。また正常に稼動しているマスタサーバはリカバリ中になる事は無い。そのため、ストリーミングレプリケーションを行っているサーバ上での pg_is_in_recovery 関数の戻り値が 't' であれば、スレーブサーバであると推測できる。

```
# スレーブサーバ上での実行結果
postgres=# select pg_is_in_recovery();
 pg_is_in_recovery
-------------------
 t
```

2. transaction_read_only の値
ホットスタンバイが有効な場合、スレーブサーバは読み取り専用でアクセス可能な状態となる。そのため transaction_read_only の値を確認し、off であればマスタサーバ、on であればスレーブサーバであると推測することができる。

```
# スレーブサーバ上での実行結果
postgres=# show transaction_read_only;
 transaction_read_only
-----------------------
 on
```

3. pg_stat_replication ビューの情報
pg_stat_replication ビューには、現在のストリーミングレプリケーションの状態を表す情報が格納されている。このビューにはマスタサーバ側にしかレコードは存在しないため、このビューを参照してレコードが 1 件以上返ってきた場合、そのサーバはマスタサーバであると判断できる。ただし全てのスレーブサーバが同期に失敗している場合、マスタサーバ側でもレコードは 0 件となるため、0 件であるからスレーブサーバであるとは言い切れない。

```
# マスタサーバ上での実行結果
postgres=# select * from pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 7267
usesysid         | 16434
usename          | replication
application_name | walreceiver
client_addr      | 192.168.0.2
client_hostname  |
client_port      | 57018
backend_start    | 2014-03-12 11:52:05.134241+09
state            | streaming
sent_location    | 0/5034080
write_location   | 0/5034080
flush_location   | 0/5034080
replay_location  | 0/5034080
sync_priority    | 1
sync_state       | sync
```

4. $PGDATA/recovery.conf ファイルの有無

$PGDATA/recovery.conf が存在している場合、PostgreSQL ではスレーブサーバとしてサービスを起動しようとし、マスタサーバとしては動作しない。そのため、recovery.conf が存在していればおそらくスレーブサーバであると推測できる。ただし通常は無いと思われるが、サービス起動後に recovery.conf ファイルを作成することは可能なので、確実にスレーブとして動作しているとは言いきれない。

5. PostgreSQL のログ出力

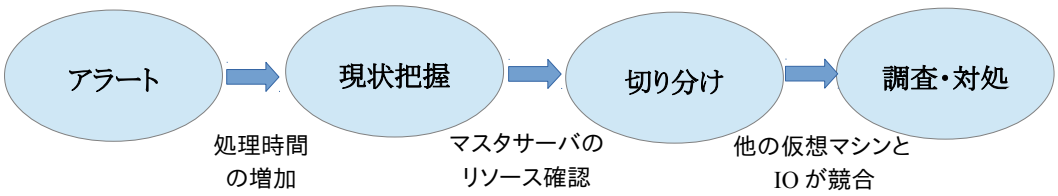
PostgreSQL ではスレーブとして起動した場合、サービスの起動時に「entering standby mode」のログが記録される。このログが記録されてからシャットダウンのログが記録されるまでの間は、スレーブとして稼動していると判断できる。

```
# マスタサーバ上のログ
< 2014-03-12 17:48:08.131 JST >LOG:  database system is ready to accept connections
< 2014-03-12 17:48:08.131 JST >LOG:  autovacuum launcher started
# スレーブサーバ上のログ
< 2014-03-12 17:28:42.265 JST >LOG:  entering standby mode
< 2014-03-12 17:28:42.272 JST >LOG:  consistent recovery state reached at 0/F000698
< 2014-03-12 17:28:42.272 JST >LOG:  record with zero length at 0/F000698
< 2014-03-12 17:28:42.275 JST >LOG:  database system is ready to accept read only connections
< 2014-03-12 17:28:42.287 JST >LOG:  started streaming WAL from primary at 0/F000000 on timeline 1
```

今回はバッチ実行時点でのマスタノードを特定したかったため、時刻の記録が残っているログをもとにマスタノードを確認した。

次にバッチが実行された時間帯のマスタノードのリソース使用率を性能監視の情報から確認したところ、CPU 使用率の iowait の割合が高い値となっていることが確認された。対象サーバは仮想化環境上の仮想マシンであり、他の仮想マシンと物理ディスクを共用していたためさらに調査を行った所、同時刻に物理ディスクを共用する別の仮想マシン上で、大量のディスクアクセスを伴う処理が行われていたことがわかった。

【調査・対処】
　今後も同様の事象が発生することが予想されたため、ディスクアクセスが競合していた別の仮想マシンを別の仮想環境へマイグレーションすることで対応を行った。



13.3.3. ケーススタディのまとめ

　本節では、監視後の作業として「アラート」、「現状把握」、「切り分け」、「調査・対処」の４つの項目を挙げ、いくつかの状況においてそれぞれ具体的にどのような作業を行うとよいかという点をケーススタディとしてまとめました。
　どのような問題が発生するかはケースバイケースですが、大きな方針として、これら４つを意識していくとよいと思います。

14. おわりに

本報告書では、2012 年 4 月に 10 社が集まって発足した PostgreSQL エンタープライズコンソーシアム(PGECons)において、2013 年度新たに設立された WG3 の初年度活動として実施した、PostgreSQL の典型的システム方式の調査および 動作検証の結果について説明しました。WG3 は初年度にも関わらず 14 社という多くの参加企業にご協力をいただき、 調査、検証作業の中から得られた様々な知見や技術ノウハウが詰まった報告書に仕上がっているものと感じています。

今年度の活動で特に良かったのは、可用性、バックアップ、監視といったまだまだ情報が少ない領域に対する検証を、日常的に業務として PostgreSQL に携わっている技術力の高いメンバーが集まって出来たことです。日頃は競合することもある各企業のメンバーが、今回オープンソースでのコミュニティ活動として PostgreSQL のエンタープライズ活用を促進させる共通の目的で一緒に検証を実施することで、本報告書での成果発表とともに技術者同士の交流を深めることが出来て、PGECons の設立趣旨に合致したかと感じています。

今年度の WG3 の活動は初年度ということもあり、どのように活動していくかという方向性から議論を積み重ねて、ワーキンググループの形を作るところから始まりましたが、参加企業がそれぞれ積極的に意見を出し合うことで、なんとか活動を軌道に乗せることができました。

その反面、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、次年度以降チャレンジ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
2013 年度 WG3 活動報告書 第 1 版	株式会社アイ・アイ・エム	CCP	住友 邦男
	株式会社アイ・アイ・エム	技術本部　プロダクトセンター	岩田 知範
	株式会社アシスト	データベース技術本部 技術開発部	高瀬 洋子
	株式会社アシスト	データベース技術本部 技術開発部	柘植 丈彦
	株式会社インフォメーションクリエーティブ	テクニカル運用本部 第 3 部	松下 祐太郎
	SRA OSS, Inc. 日本支社	取締役支社長	石井 達夫
	SRA OSS, Inc. 日本支社	マーケティング部 OSS 技術グループ	長田 悠吾
	NTT ソフトウェア株式会社	クラウド事業部	勝俣 智成
	クオリカ株式会社	IT サービス事業本部 流通サービス事業部 リテールソリューション部	角 昂
	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	高橋 和也
	日本電気株式会社	システムプラットフォームビジネスユニット システムソフトウェア事業部	川畠 輝聖
	日本電気株式会社	システムプラットフォームビジネスユニット システムソフトウェア事業部	鄭 東紀
	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
	日本ヒューレット・パッカード株式会社	テクノロジーコンサルティング事業統括	北山 貴広
	日本ヒューレット・パッカード株式会社	テクノロジーコンサルティング事業統括	高橋 智雄
	株式会社日立製作所	情報・通信システム社 ITプラットフォーム事業本部	福岡 博
	株式会社日立ソリューションズ	技術開発本部 オープンソース技術開発センタ	稲垣 毅
	富士通株式会社	データマネジメント・ミドルウェア事業部	山本 明範
	富士通株式会社	データマネジメント・ミドルウェア事業部	榎本 友理枝