PGECons
PostgreSQL Enterprise Consortium

課題検討WG 2017年度活動報告

意外と知らない? PostgreSQLの動かし方を調べてみた
～レプリケーション・Windows・性能トラブル～

PostgreSQL エンタープライズコンソーシアム
WG3 (課題検討WG)
TIS (株) 中西 ・ 富士通 (株) 山本 ・ NEC (株) 湯村

アジェンダ

- 2017年度の活動テーマと活動体制

- レプリケーション
  - レプリケーション構成の機能、運用方法を調査
- Windows環境
  - Windows環境での運用に関する調査
- 性能トラブル
  - 性能トラブルの予防・検知・対策に関する調査

2017年度の活動テーマ

- ワーキンググループの活動領域の中から、2017年度は可用性・運用性・性能に関するテーマを取り上げた

領域	概要	2017	2016	2015以前
可用性	高可用性クラスタ、BPC	レプリケーション、Windows環境(可用性)	レプリケーション	可用性, BCP
運用性	バックアップ運用、運用監視	Windows環境(ツール、インストレーション)		ツール バックアップ
性能	性能評価手法、性能向上手法等	性能トラブル		ツール
セキュリティ	監査、認証、機密性			セキュリティ
接続性	他ソフトウェアとの連携		異種DB連携	
保守性	保守サポート、トレーサビリティ			
互換性	他DBMSとの互換性			

WG3 2017年度活動体制

- レプリケーション班
  - 株式会社アシスト
  - 株式会社オージス総研
  - TIS株式会社
  - 株式会社富士通ソーシアルサイエンスラボラトリ
- Windows班
  - NTTテクノクロス株式会社
  - 日立製作所
  - 富士通株式会社
- 性能トラブル班
  - 日本電気株式会社
  - 日本電信電話株式会社
  - 株式会社富士通ソーシアルサイエンスラボラトリ

敬称略・50音順

レプリケーション成果紹介

レプリケーション：活動目的

- 設定/監視など運用に必要なノウハウを整理する。また今後も新機能やツールが追加されるため、定点観測として続ける。

今まで

- 可用性向上に活用できるレプリケーション技術の検証
- ストリーミングレプリケーションの既存情報整理と最新機能検証
- マルチマスタレプリケーションシステム「BDR」の検証

2017年度

- ストリーミングレプリケーションの検証範囲拡大
  - 3ノード構成、カスケード構成での障害時運用
- ロジカルレプリケーションの検証
  - PostgreSQL10新機能の実用性

レプリケーション
- ストリーミングレプリケーション -

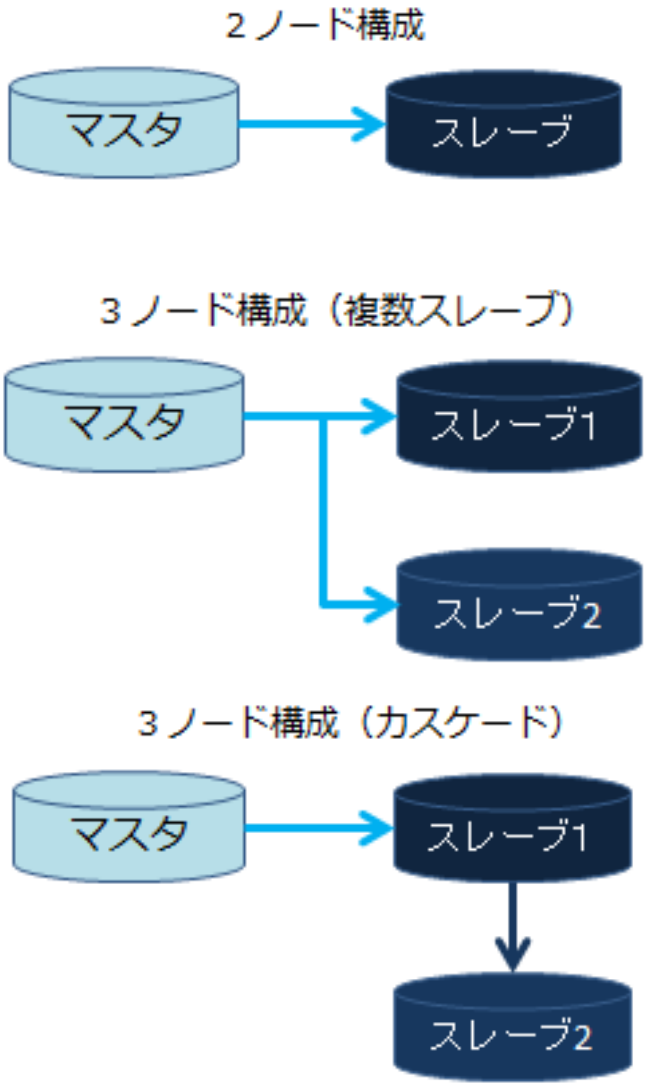
ストリーミングレプリケーションの検証構成

2016年度の構成

- PostgreSQL 9.6
- 2ノード構成（マスタ1、スレーブ1）

2017年度の構成

- PostgreSQL 10
- 3ノード構成（マスタ1、スレーブ2）
  - 複数スレーブ
  - カスケード

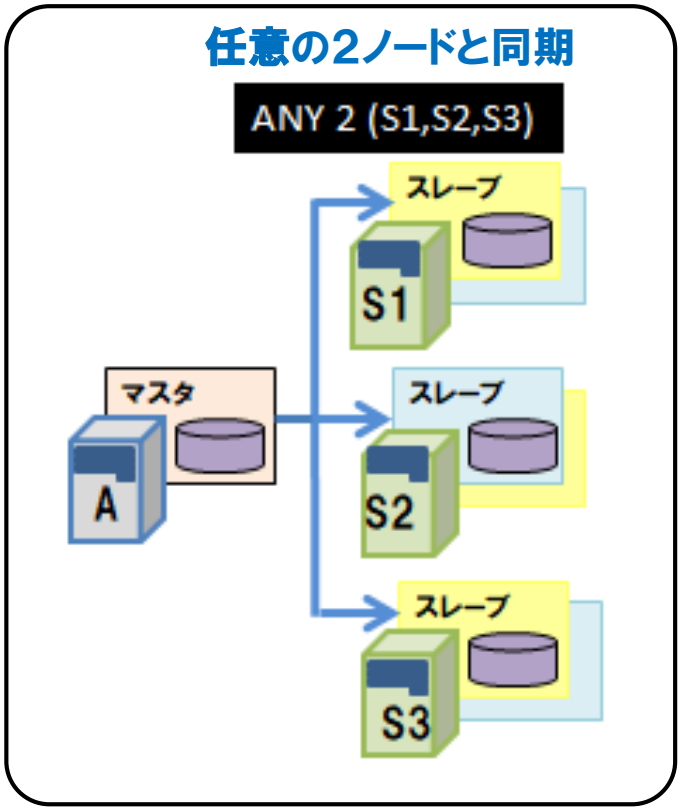


上記構成にて
監視や障害時運用を検証

複数スレーブ構成の特徴

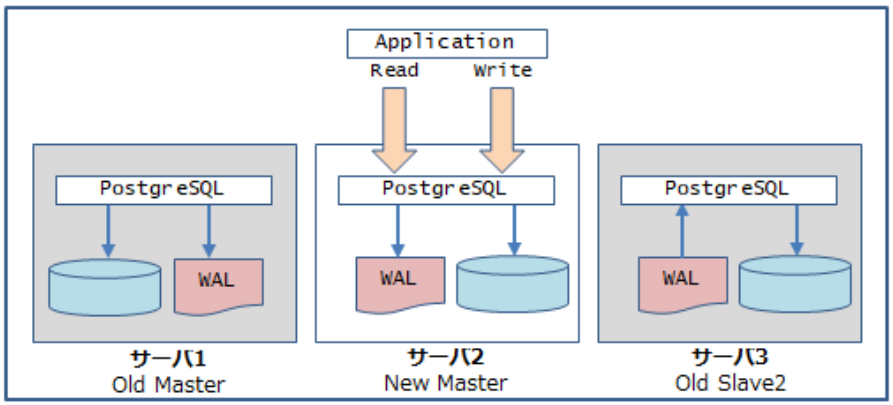
- 特徴
  - 多彩な構成が可能
    - 完全同期/同期/非同期
    - 遅延レプリケーション
  - ◯マスタから全スレーブを監視可能
  - ✖スレーブ数に応じマスタ負荷増大
- 用途
  - バックアップ
  - 参照負荷分散

New！　Quorum-based
同期レプリケーション

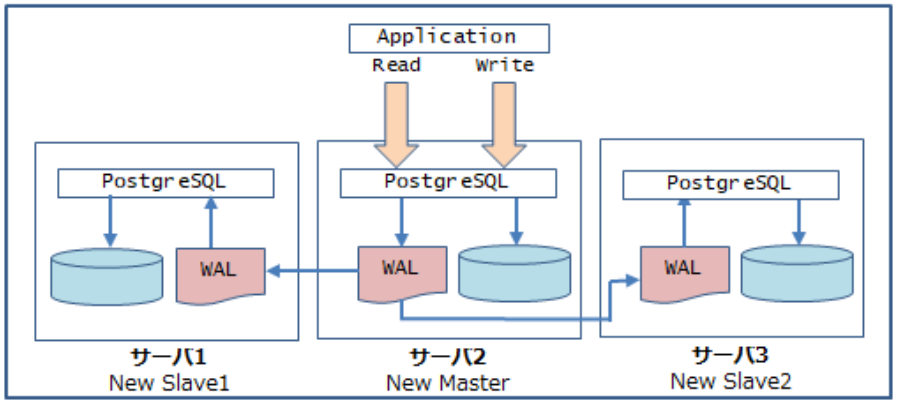


複数スレーブ構成の障害時運用(マスタ障害時)

- フェイルオーバ
  - マスタ障害時に全スレーブへの伝搬が停止
  - 任意のスレーブをマスタへ昇格(pg_ctl promote)
- フェイルバック
  - サーバ2を新マスタとして、サーバ1,3をスレーブ追加
  - サーバ1は新マスタからデータを再取得する(負荷影響に注意)



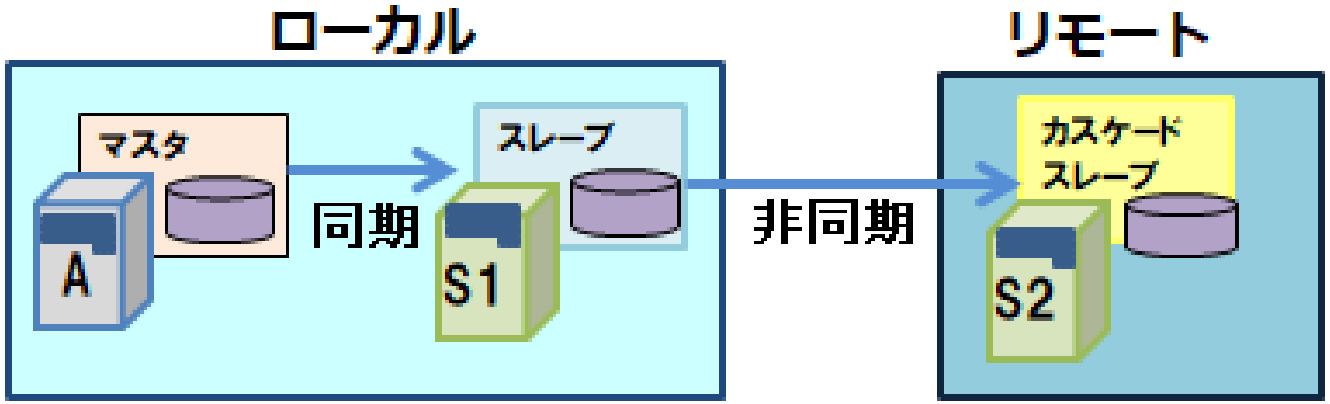
FailOver後



FailBack後

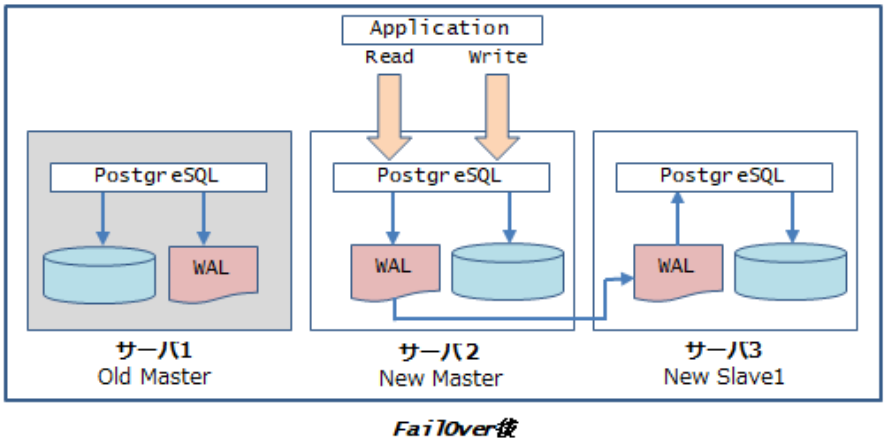
カスケード構成の特徴

- 特徴
  - ◯ マスタの負荷は一定
  - ✕ マスタから全スレーブを監視できない
- 用途
  - マスタ負荷の軽減
  - ディザスタリカバリ

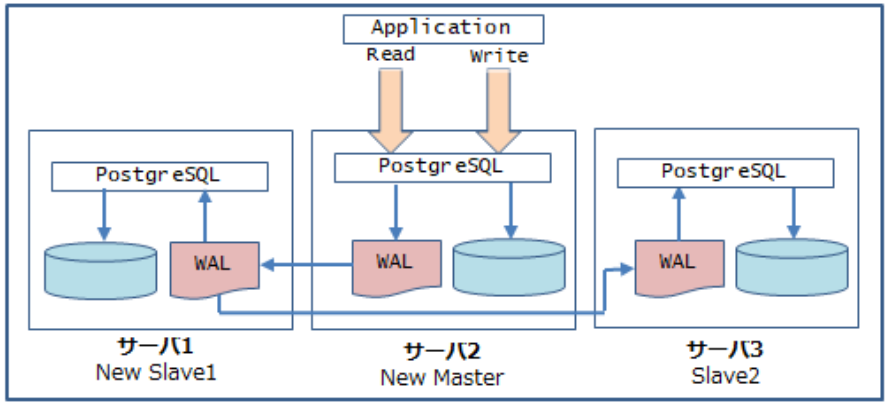


カスケード構成の障害時運用(マスタ障害時)

- フェイルオーバ
  - サーバ2をマスタへ昇格(pg_ctl promote)
  - マスタ昇格後もサーバ2→サーバ3のレプリケーションは維持
- フェイルバック(サーバ3を末端にしたい場合)
  1. サーバ2 (サーバ3の切り離し)
  2. サーバ2 → サーバ1 (サーバ1をスレーブとして追加)
  3. サーバ2 → サーバ1 → サーバ3 (サーバ3をスレーブとして追加)



FailOver後



FailBack後

ストリーミングレプリケーションのまとめ

- PostgreSQL10では多彩な構成が可能
  - 複数スレーブ、カスケード
  - 完全同期/同期(Quorum-based)/非同期
  - 遅延レプリケーション

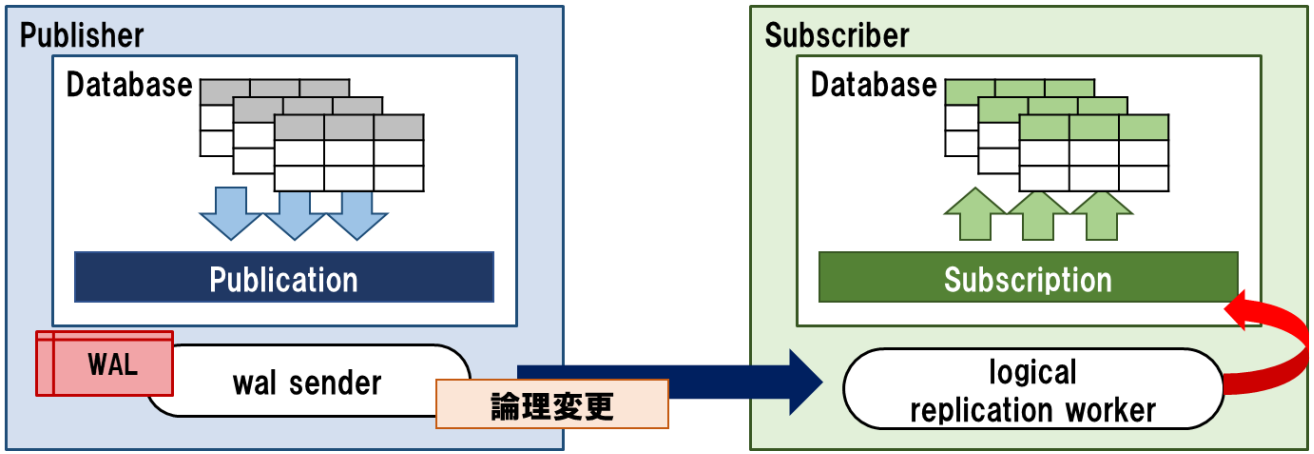
- 複数スレーブ構成とカスケード構成を検証して
  - マスタ障害時、復旧時にやることは大差ない
  - 複数スレーブが運用はしやすい（監視、クラスタウェア）
  - まずは複数スレーブで構成し、スレーブ台数の増加や用途（DR等）に応じてカスケード構成を採用すればよい

- 成果物にはより多くの障害パターンについて記載

レプリケーション
- ロジカルレプリケーション -

ロジカルレプリケーションの概要

- 概要
  - テーブルデータの論理的な変更内容の送受信を
    複製元（Publisher）/複製先（Subscriber）間で行いデータ複製を実現
- 特徴
  - 任意のテーブルのみをレプリケーション可能
  - 特定の処理（例えばINSERTのみ）のみをレプリケーション可能
  - 異なるPostgreSQLバージョン間でのレプリケーション可能
  - レプリケーション先でテーブルデータの更新やインデックス定義が可能



ロジカルレプリケーションの概要

- 制限事項
  - ロジカルレプリケーションの制限事項は以下のとおり（PostgreSQL10時点）

項番	レプリケーション対象外	対象方法
1	データベーススキーマ およびDDLコマンド	データベーススキーマは、pg_dump -schema-onlyを 利用して移行可能
2	シーケンス	シーケンスはレプリケーションされないが、 シーケンスによって裏付けされたSERIAL型や識別列のデータは、テーブルデータの一部としてレプリケーション可能
3	TRUNCATEコマンド	DELETEコマンドで回避することは可能
4	ラージオブジェクト	通常のテーブルにデータを格納する以外の回避方法なし
5	テーブル以外のオブジェクト	ビュー、マテリアライズドビュー、パーティションのルートテーブル（親テーブル）、外部テーブルはレプリケーションしようとするとエラーになる

ロジカルレプリケーションの応用

以下のケースについて検証しました。

- PK未定義のテーブルに対するレプリケーション
  - 変更したレコードを特定するREPLICA IDENTITYを設定する必要あり
  - PK以外にUNIQE INDEXやFULL(注:非効率)を指定可能
- 同期レプリケーション
  - SRと同様の同期レプリケーション設定が可能
- 複数サブスクリプション
  - 1つの複製元(Publication)から複数の複製先(Subscription)を設定可能
- カスケード構成
  - 複製先のテーブルを新たな複製元に設定する構成が可能
  - 同一テーブルで複製元/先をループさせるとエラー(例: サーバ A -> B -> A)
- パーティショニングとの組み合わせ
  - パーティションの子テーブルへのレプリケーションが可能
  - 複数サーバのテーブルを1台のパーティションテーブルに集約することも可能

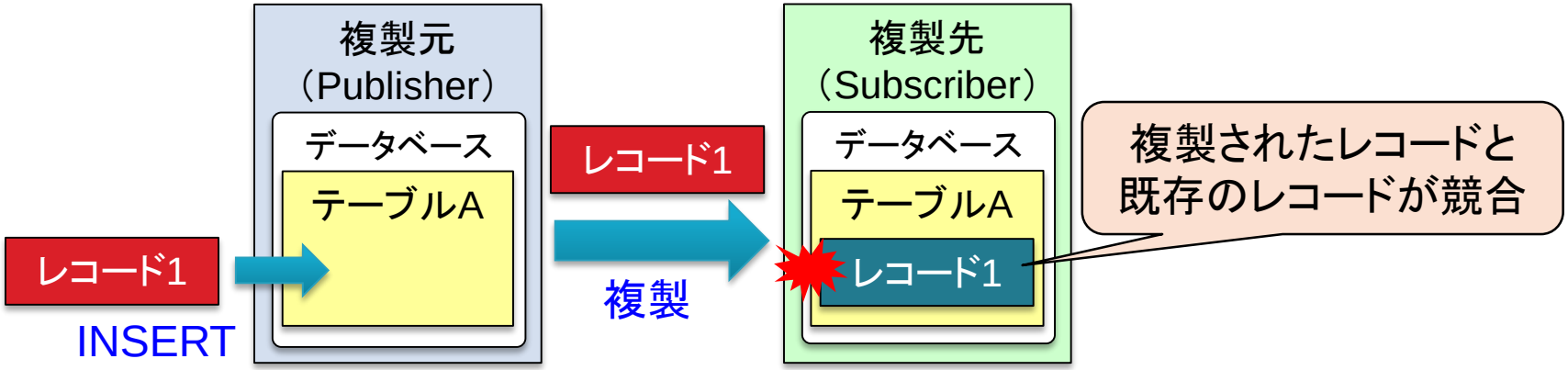
ロジカルレプリケーションの運用

以下のケースについて検証しました。

- ロジカルレプリケーションの監視
- 障害発生時におけるロジカルレプリケーションの挙動
- レプリケーション対象テーブルの追加、定義変更の手順
- 複製されない操作、オブジェクトへの対処
  - TRUNCATE, シーケンス
- 更新競合時の挙動と対処方法
- ストリーミングレプリケーションとの併用

更新競合時の動作

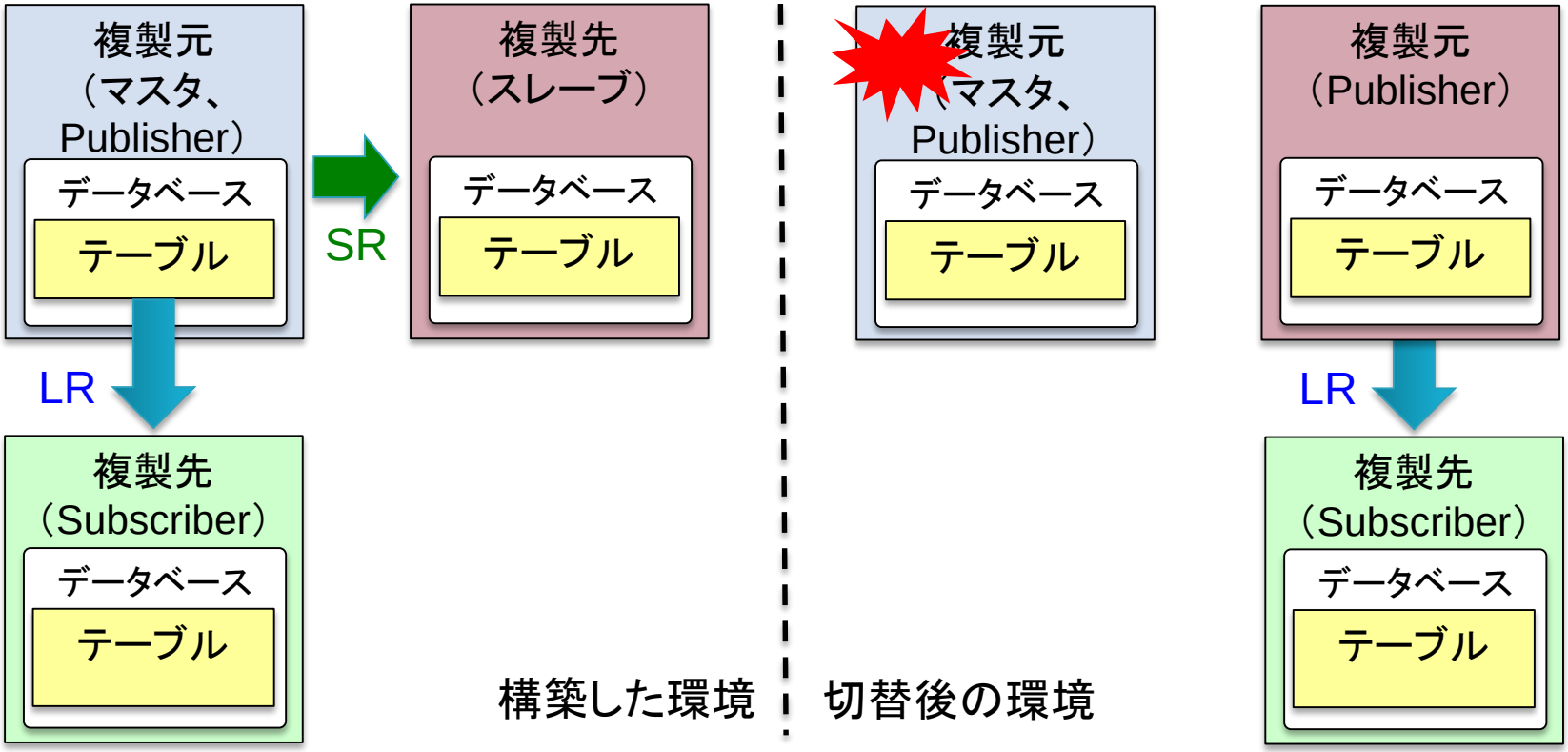
- ロジカルレプリケーションは複製先 (Subscriber) を更新できるが、複製元 (Publisher) への更新が複製先のデータと競合しうる。



- 競合が発生する9パターンの挙動と競合の解消方法を調査
  - 主キー違反、更新データ未存在、データ型不一致、更新レコードのロック待ち etc
  - レプリケーションの停止、更新内容の無視、レプリケーションの待機
- PostgreSQLでは競合を手動で解消する必要がある。
  - 複製先から競合の原因 (重複レコード etc) を取り除く
  - pg_replication_origin_advance関数で競合するWALをスキップさせる

ストリーミングレプリケーションとの併用

- それぞれの長所を組み合わせた構成
  - ストリーミング：完全一致な複製 ⇒ マスタの冗長化による可用性向上
  - ロジカル: テーブル単位で複製、複製先のみ更新可 ⇒ 参照負荷の分散
- 併用環境の構築手順と障害発生時の切替手順を検証



ストリーミングレプリケーションとの併用

- 構築手順
  - 新規でストリーミングレプリケーションを稼働させる手順と変わらない。

- 障害発生時の切替手順
  - ストリーミングレプリケーションのマスタはpg_ctl promoteコマンドで切替
  - ロジカルレプリケーションの切替にはいくつかの手順が必要
  - 0. 複製元のPublicationを再定義する必要なし
  - 1. 複製先に存在するSubscriptionとレプリケーションスロットの対応付けを解除
  - 2. 複製先に存在するSubscriptionを削除
  - 3. 新たな複製元に対するSubscriptionを作成

障害発生時の切替を自動化するにはハードルがある

ロジカルレプリケーションのまとめ

- ストリーミングレプリケーションとは違う特徴を持つ
  - データベース単位、テーブル単位、更新操作単位で柔軟に設定
  - マイグレーション、分析用データベース、参照負荷分散に応用
- 現状の実装では運用上気をつけるべき点がある
  - 複製されない操作、オブジェクト
  - 更新競合は自力で解消する必要あり
    ⇒ 競合が発生しないように使いこなす
  - 障害発生時の切替手順が煩雑

PostgreSQLのレプリケーション

- 成果物では3種類のレプリケーション技術を調査
  - ストリーミングレプリケーション（SR）
  - ロジカルレプリケーション（LR）
  - Bi-Directional Replication（BDR）

- それぞれの特徴に応じて使い分けましょう
  - 可用性向上 ⇒ マスタを完全複製できるSR
  - 参照負荷分散 ⇒ 検索用途に特化したスレーブを作れるLR
  - マルチマスタ ⇒ 双方向レプリケーションが可能なBDR

Windows環境
成果紹介

テーマ選択の背景と現状認識

- PostgreSQL @ Windows環境に関連する情報量の不足
  - 感覚的にも検索等で得られる情報はLinux環境が中心
  - 成果報告会のアンケート結果(2015年度)

> Q. PostgreSQLをより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
> 採用するための課題は何だとお考えですか？
>
> ⇒ 高可用構成がLinuxを中心とした構成が多い。Windows構成での
> 事例が少なく、提案しにくい

- PostgreSQLの利用者の傾向の変化
  - PGConf.ASIA 2017(2017年12月)参加登録者へのアンケート結果より

> PostgreSQL初心者を中心にWindows上での利用者の割合が高く(4割)
> 既存ユーザ含めた全体の傾向も、昨年度と比較してやや増加傾向

中長期的にLinux環境と同等の運用ノウハウ整理が必要

今年度の活動のねらい

運用方式設計、環境構築の際に有用な情報を発信する

- 活動を通して発信したい情報
  - Windows環境での基本的な運用ノウハウ
  - PostgreSQL@Linux環境、@Windows環境の差異の明確化
- 2017年度は以下の観点で調査・検証
  - インストレーション
    - インストレーション時のカスタマイズ可否とLinux環境との差異の整理
    - contribや関連OSS製品のWindows環境での対応状況整理
  - ユーティリティコマンド
    - 基本的な運用手順の確認とLinux環境との差異
  - 高可用性
    - Pgpool-II(Linux環境)でWindows上のPostgreSQLを管理する場合の構築手順
    - 障害を想定した動作確認(3台構成)

Windows環境
-インストレーション-

インストレーション時のカスタマイズ可否

- PostgreSQL Documentに則りインストレーションを実施
  ※Chapter 17. Installation from Source Code on Windows
- カスタマイズ可否の確認
  - Linuxのconfigure相当の内容を比較
  - Windowsでは設定ファイル(Config.pl)に記述しカスタマイズ

検証結果　カスタマイズ可能/不可の範囲を明確化

- カスタマイズ可能な例:WALセグメントのサイズ 他
- カスタマイズ不可な例:readline関連 他(コマンドライン編集&履歴)

■configureオプションの比較
- Windowsでソースからのインストレーションを実施する場合、config.plファイルを編集してオプションを記述します。
- config.pl上で設定できるパラメタと記述内容を[Windows - config.plへの記述方法]列に記載しています。

インストーラでの設定有無

UNIX		Windows			
ソースからのインストレーション					
configureオプションの記述方法	デフォルト値	configure.plへの記述方法	デフォルト値	インストーラでのインストレーションで設定されている内容	PostgreSQL Documentでの説明
		icu	(設定なし)	○	Build with support for the ICU library. This requires the ICU4C package to be installed. The minimum required version of ICU4C is currently 4.2.
		openssl	(設定なし)	○	SSL(暗号化)接続のサポートを有効にして構築します。これには、OpenSSLパッケージがインストールされていなければなりません。configureは、処理を進める前にOpenSSLのインストールを確認するために、必要なヘッダファイルとライブラリを検索します。

対応するオプション有無
あれば記述方法

contribモジュールの対応状況

- contrib配下のモジュールのWindows対応状況を整理
- 機能の利用方法の差異も検証

検証結果　大部分のcontribモジュールは対応

- インストール先ディレクトリに実行ファイルやDLLファイルが生成される
- 使用方法は以下の通り
  - 例）pgbench：インストール先¥bin 配下に生成されたコマンド実行
  - 例）pg_stat_statements：shared_preload_librariesに設定

■contribのWindowsでの対応状況
・ビルド時にvcxprojファイルが生成され、インストール時にライブラリがlib配下に配置される場合に○としています。

拡張機能名	マニュアル記載箇所 (PostgreSQL 10)	Windows対応状況	拡張機能概要 (Windows対応状況が「×」の場合のみ)	備考
adminpack/	F.1	○		
amcheck/	F.2	○		
auth_delay/	F.3	○		
auto_explain/	F.4	○		
bloom/	F.5	○		

周辺ツールの対応状況

- 2015年度WG3成果物「2015年度 WG3活動報告書 データベースツール編」の記載のツールについて机上調査

検証結果　Windows対応しているツールは一部

- 記載されている24種類のツールの内、5種類
  - 動作不可例：Pgpool-II（ただし後述の検証の通り、Windows上で動作するPostgreSQLに対する管理は一部を除き可能）

■周辺ツールのWindowsでの対応状況
周辺ツールとしては、2015年の成果物の「2015 年度 WG3 活動報告書 データベースツール編」より選定。

カテゴリ	ツール名	Windows対応状況	拡張
バックアップ	pg_basebackup	○	本体
バックアップ	Barman for PostgreSQL	×	P 要は 作、
バックアップ	OmniPITR	△	Perl れば

○：動作可の明記あり
△：ツールの性質上動作の可能性あり
×：動作不可の明記あり

検証結果のまとめ

- インストレーション手順の試行と、以下の検証を実施

インストレーション時のカスタマイズ可否

- 業務パッケージへの組込用途等必要な場合は、
  カスタマイズ可否と要件の突合せし、回避策の検討が必要

Contribモジュールの対応状況

- 大部分のモジュールは利用可能
- インストレーション時にモジュールがインストールされる

周辺ツールの対応状況

- Windowsで使用可能なものは限定的。
- その場合は要件を満たす代替手段の検討が必要

今後：対応できない場合の代替手段含めた情報整理

Windows環境
-ユーティリティコマンド-

データベースクラスタの初期化と制御

- initdbコマンドによるデータベースクラスタ作成手順の整理
  - WALの書き出し先を変更する場合の注意点の整理
- pg_ctlコマンドを利用した起動/終了/状態確認

検証結果 WAL書き出し先の変更は特に注意点なく可能

- initdb
  - WAL書き出し先を変更するコマンドオプションはLinux同様
    initdbコマンドの-Xオプションを利用
  - 実装方式は異なる(OSの機能での実装)
    Linux:シンボリックリンクにより、WALの書き出し先を変更
    Windows:JUNCTION機能を使ってWALの書き出し先を変更
- pg_ctl
  - 起動/終了/状態確認が可能
  - Windows固有のオプションとして「(un)register」がある

サーバログ出力

- Linuxのsyslog同様のサーバログ出力が可能か整理
- サーバログが出力先の明確化

検証結果 指定内容、出力先共に異なる点に注意

- 指定内容
  - log_destinationの指定は、'syslog'ではなく、'eventlog'
- 出力先
  - ‘eventlog’ 指定時の出力先はWindowsのイベントビューアを使って サーバログが確認可能
- その他注意点
  - イベントソース名(通常は'PostgreSQL') を変更する場合は レジストリの変更が必要

バックアップ/リストア

- pg_dumpによる論理バックアップ/リストア手順の整理
- pg_basebackupによるオンラインバックアップ/リストア/リカバリ手順の整理

検証結果 Linux/Windowsで大きな差異はなし

- pg_dump (論理バックアップ/リストア)
  - Linux/Windowsで手順の差異はなし
- pg_basebackup (オンラインバックアップ/リストア/リカバリ)
  - 下記の注意点を除いてはLinux/Windowsで手順の差はなし
- その他注意点
  - archive_commandの指定値 (Linux:cp → Windows:COPY)
  - ベースバックアップの展開先へのアクセス権限付与が必要になる場合がある

検証結果のまとめ

- ユーティリティコマンドの試行と運用の差異の有無を検証

データベースクラスタの初期化と制御

- initdb、pg_ctlコマンドは問題なく実行可能
- WALの書き出し先変更もinitdbの-Xオプションで可能

サーバログ出力

- syslogに相当するサーバログの出力先として
  Windowsのイベントビューアを指定可能

バックアップ/リストア

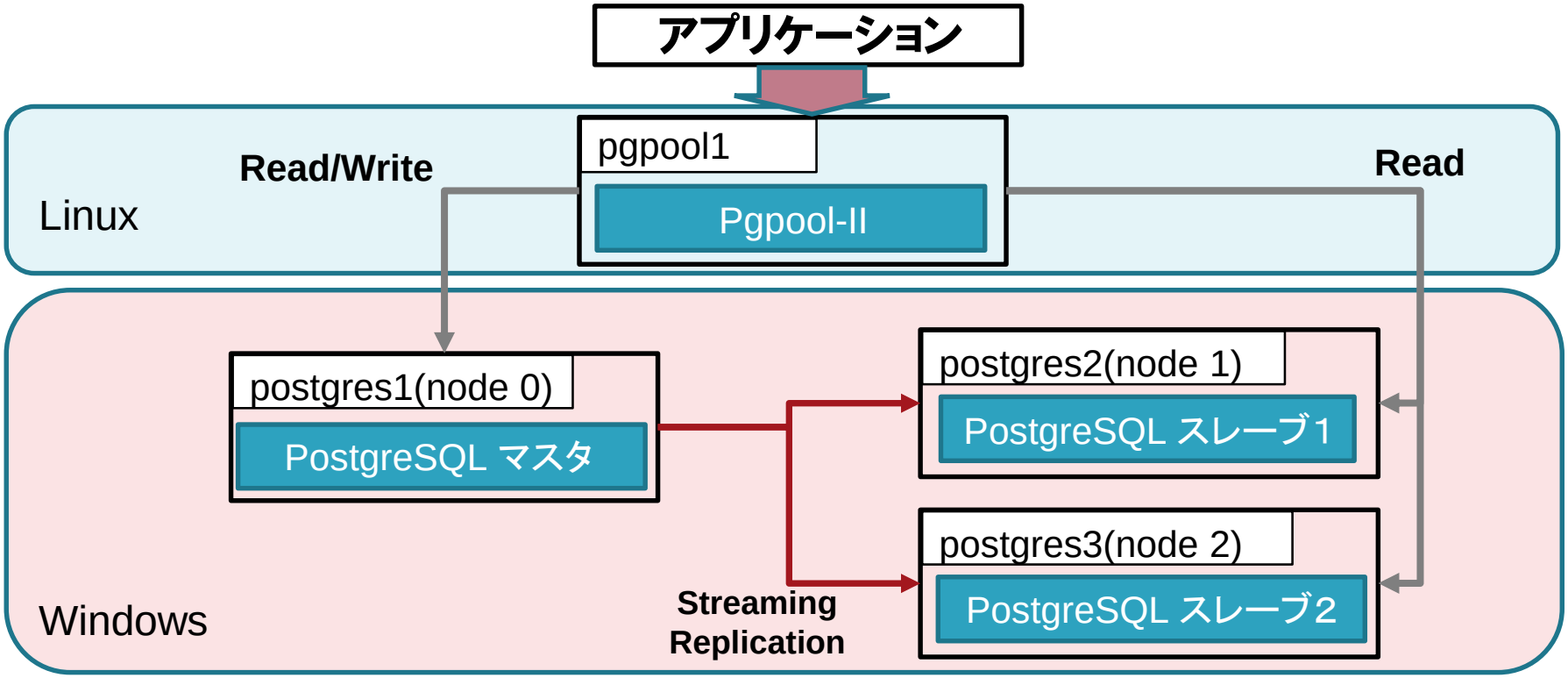
- 基本的な手順はLinuxと同様
- ベースバックアップの展開先へのアクセス権限には注意

ユーティリティコマンドによる運用に大きな差異・懸念点はなし

Windows環境
-高可用性-

高可用構成の構築

- Pgpool-II(Linux)を使って、Windows上のPostgreSQLを管理するための構築手順を整理
- Pgpool-IIのコマンド実行用にMicrosoft版OpenSSHの構築手順を整理



高可用構成の構築

- Pgpool-II(Linux)を使って、Windows上のPostgreSQLを管理するための構築手順を整理
- Pgpool-IIのコマンド実行用にMicrosoft版OpenSSHの構築手順を整理

検証結果　Linux同様に管理できることを確認

- Windows版固有の注意点を検出
  - スレーブ構築の際、ベースバックアップの取得先ディレクトリのアクセス権限の変更が必要
  - データベースクラスタにスペースを含むディレクトリを指定した場合に注意(例:Program Files 等)
  - Microsoft版OpenSSHは「This is a pre-release (non-production ready)」とあり、使用には注意が必要

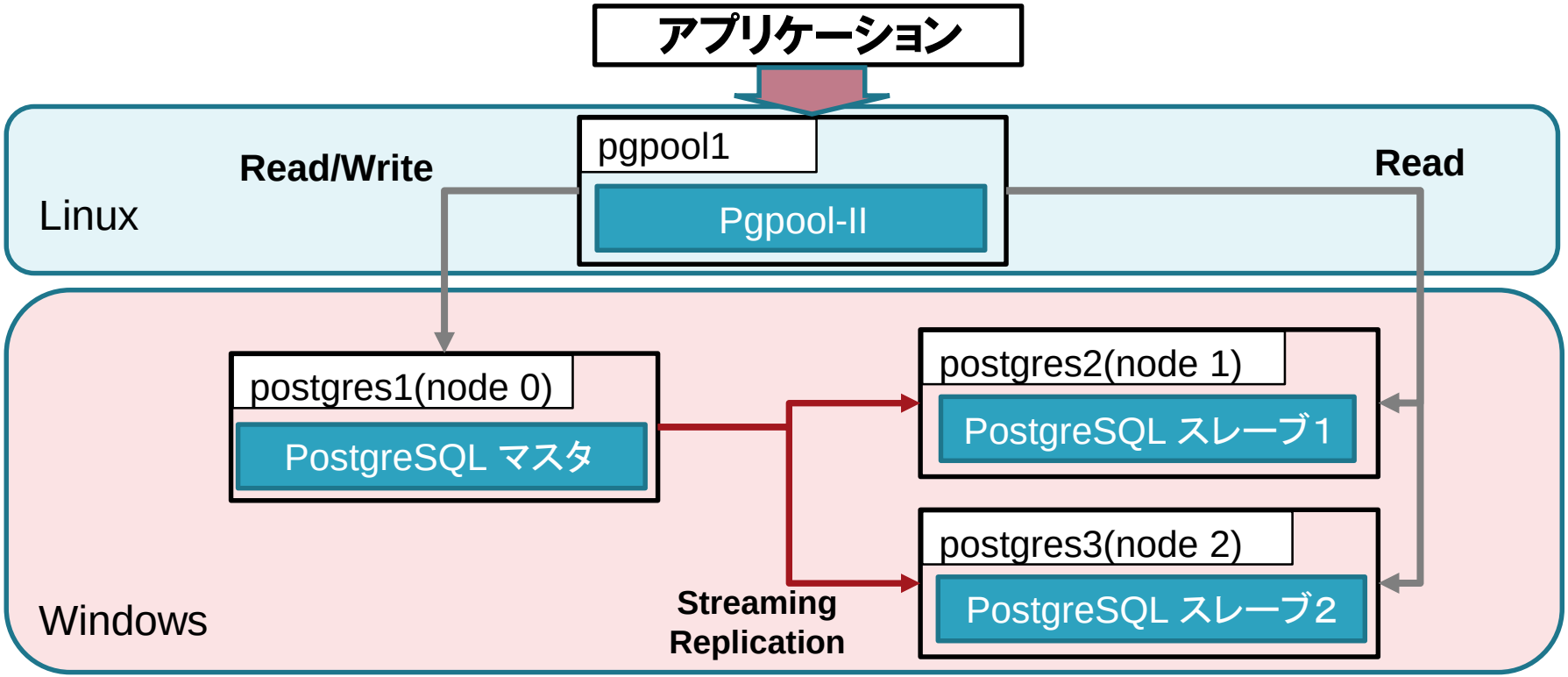
運用管理を行うコマンドの動作確認

- PCPコマンド(Pgpool-IIの管理コマンド)の利用可否を確認
  - Pgpool-II間で死活監視するwatchdogはWindows検証と関係がないため除外

コマンド名	利用可否	補足
pcp_node_count	○	
pcp_node_info	○	
pcp_proc_count	○	
pcp_proc_info	○	
pcp_pool_status	○	
pcp_detach_node	○	
pcp_attach_node	○	
pcp_promote_node	○	
pcp_stop_pgpool	○	
pcp_recovery_node	×	前提のpgpool-recoveryがWindowsビルド非対応

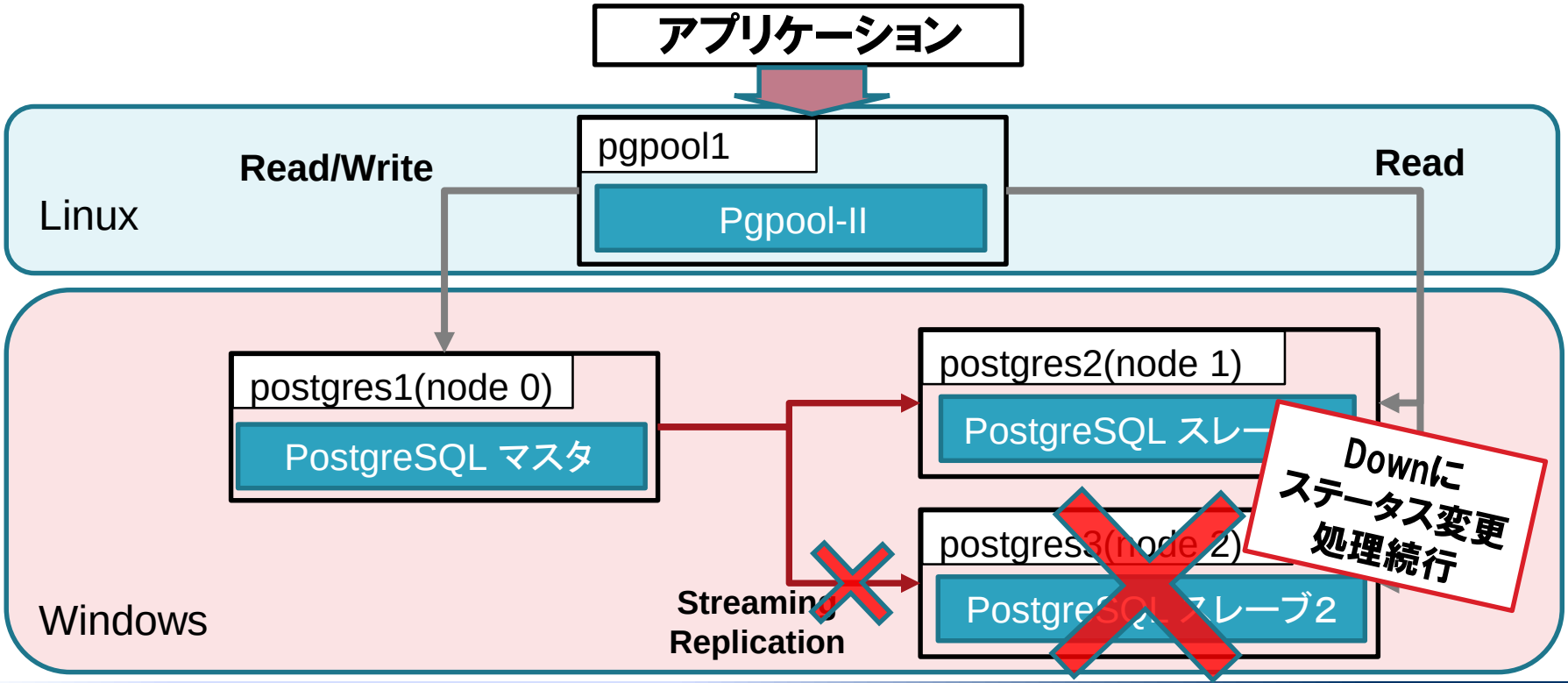
障害時の動作検証

- PostgreSQLサーバでの障害検知・切り替えの動作検証
  - パターン1：スレーブ2がダウンした場合
  - パターン2：マスタがダウンした場合
  - パターン3：パターン2の後、スレーブ1（新マスタ）もダウンした場合



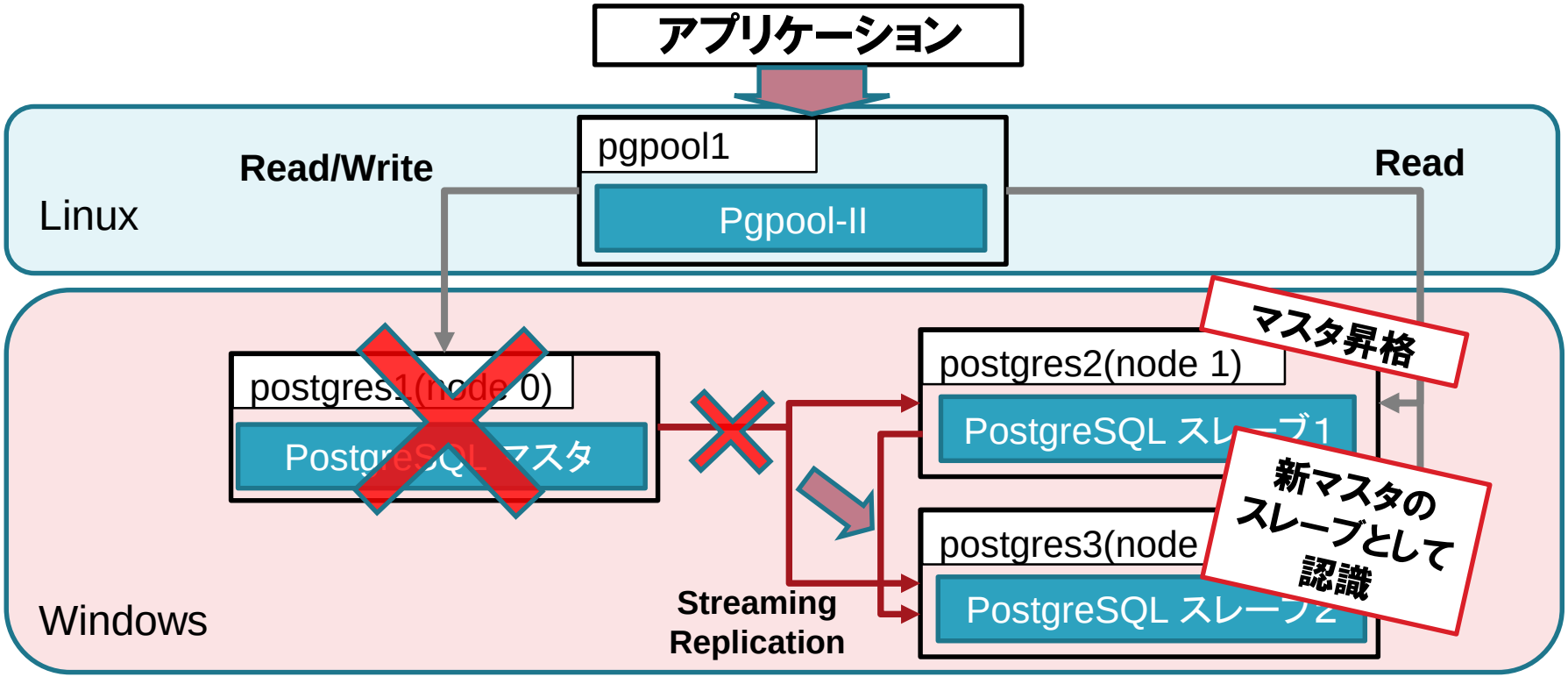
障害時の動作検証

- PostgreSQLサーバでの障害検知・切り替えの動作検証
  - パターン1：スレーブ2がダウンした場合
  - パターン2：マスタがダウンした場合
  - パターン3：パターン2の後、スレーブ1（新マスタ）もダウンした場合



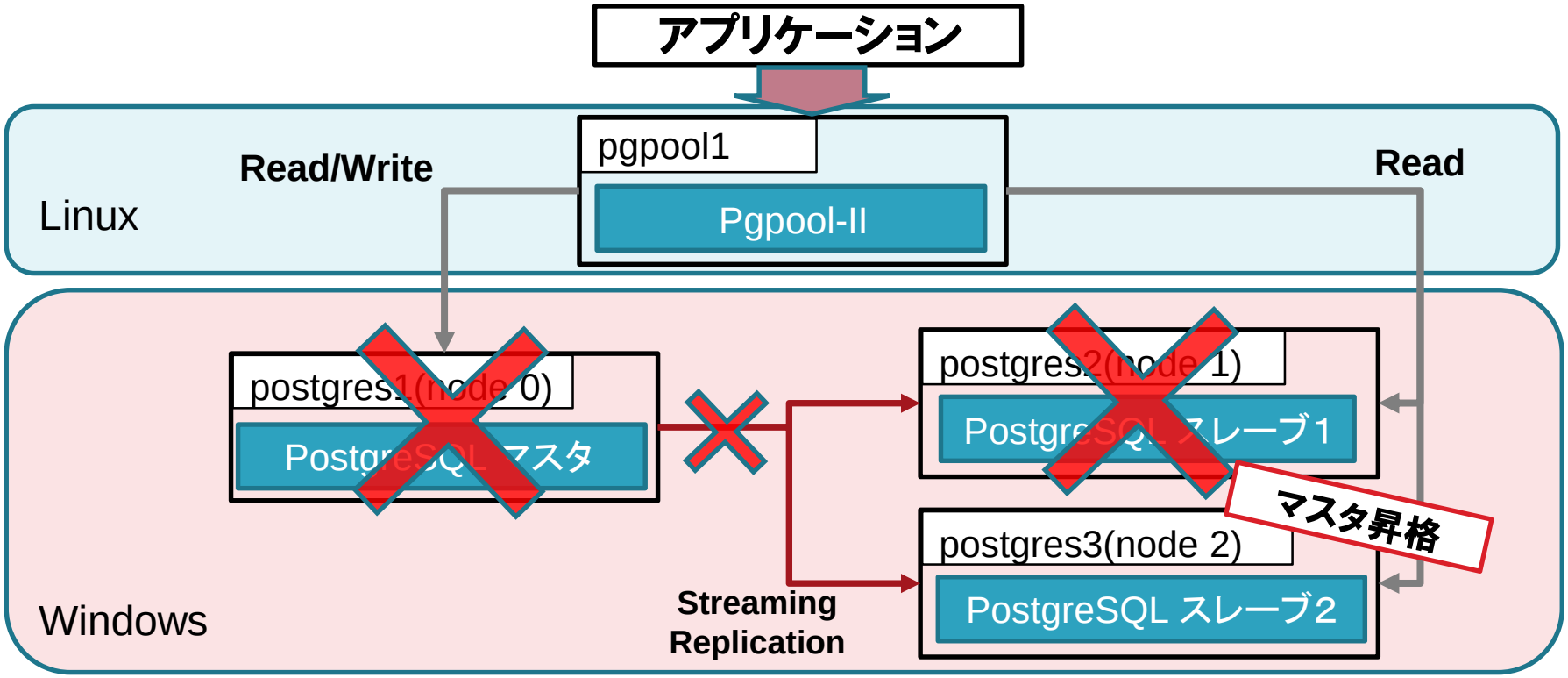
障害時の動作検証

- PostgreSQLサーバでの障害検知・切り替えの動作検証
  - パターン1：スレーブ2がダウンした場合
  - パターン2：マスタがダウンした場合
  - パターン3：パターン2の後、スレーブ1（新マスタ）もダウンした場合



障害時の動作検証

- PostgreSQLサーバでの障害検知・切り替えの動作検証
  - パターン1：スレーブ2がダウンした場合
  - パターン2：マスタがダウンした場合
  - **パターン3：パターン2の後、スレーブ1（新マスタ）もダウンした場合**



障害時の動作検証

- PostgreSQLサーバでの障害検知・切り替えの動作検証
  - パターン1：スレーブ2がダウンした場合
  - パターン2：マスタがダウンした場合
  - パターン3：パターン2の後、スレーブ1（新マスタ）もダウンした場合

検証結果

- 一通りのHA検証を行ったが、問題になる点はなかった

検証結果のまとめ

- 高可用性構成の構築と運用、障害パターンの実機検証を実施

高可用性構成の構築

- Linuxと同様に、Windows環境のPostgreSQLを管理可能
- Windows固有の注意点あり（権限、ディレクトリパス 等）

PCPコマンドの動作確認

- Windows上でのビルドが前提のコマンドを除き動作可能

障害時の動作検証

- 3台構成で障害パターンを検証
- Linuxと同様に障害検知および切り替えが可能

Pgpool-IIを利用したHAクラスタ構成に大きな懸念点はない

Windows環境調査2017年度活動のまとめ

PostgreSQL@Windowsで基本的な運用手順を整理

インストレーション

- インストレーションのカスタマイズ可否とツール類について Linuxとの差異を明確化

ユーティリティコマンド

- Linux同様にユーティリティコマンドを実行可能 Windows固有（OS固有）の動作について明確化

高可用性

- Linux版Pgpool-IIからWindows版PostgreSQLを管理可能
- 障害パターンを実機検証し、障害検知・切替動作を確認

性能トラブル成果紹介

性能トラブル班の活動目的

- テーマ選択にあたっての背景
  - PostgreSQLの普及/利用者拡大に伴い、
    様々なケースの性能トラブルが発生
  - 性能トラブルを未然に防ぐ方法や早期に検知するための
    情報を整理することをWG3で取り組むべき課題として選択

- 目的
  - 性能トラブルを未然に防ぐためのノウハウや
    トラブルを検知するための情報を整理する
  - PostgreSQLのエンタープライズ領域への普及にあたって、
    性能トラブル対策の一助となる情報を提供する

成果物について

- 参画企業が直面した性能トラブルの事例と原因に関する情報を共有し、以下の観点で分析・議論

- 性能トラブルを防ぐために
  - 設計段階での**予防**
  - トラブルの早期**検知**
  - いざ発生した時の**対処**

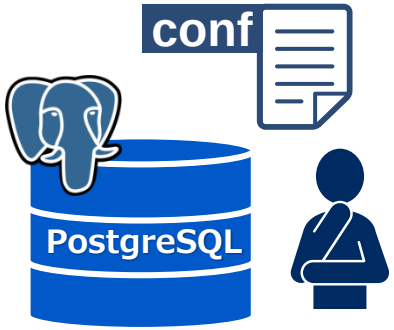
  の3つの観点で調査/整理したノウハウを紹介

成果物の構成

予防

3章 性能トラブルを予防するために考慮すべきデータベース設計のポイント

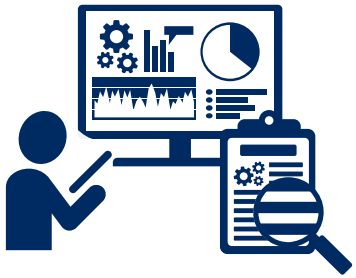
- 性能トラブルを予防するために、設計段階で考慮するポイントを再確認
- 詳細はリンク先の技術情報、講演資料で深掘り



検知

4章 性能状態を把握するための監視

- 性能トラブルを早期に発見するための監視項目とその監視方法を確認
- どのような情報を取れるか、どんな観点でチェックすればよいかを記載



対処

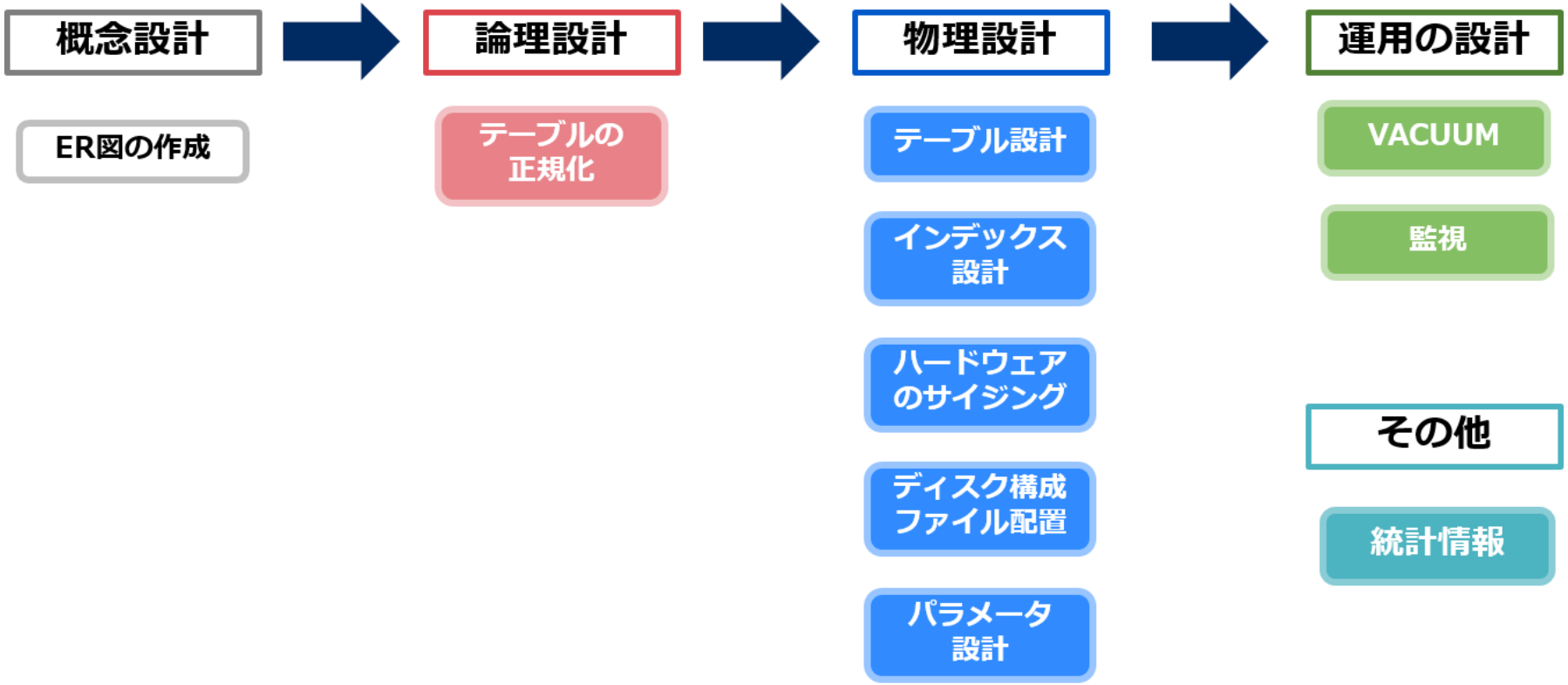
5章 ケーススタディ

- 性能トラブル発生から解決までのサンプル事例（参考情報）

性能トラブル
-予防-

予防としての設計のポイントについて

- データベースの設計を4段階に分け、その中で9種類の項目を重要ポイントとしてピックアップ
- 各要素について、過去の勉強会や講演でとりあげられた知見/ノウハウを改めてとりあげてまとめている



ポイントごとに設定の方針と関連するトラブルを簡潔に説明

3.3.2. インデックス設計¶

テーブルのデータ型やパーティショニングを検討・設計した後、想定される検索や結合の条件を精査し、対象となる列にインデックスを定義していきます。

考慮すべきポイント

闇雲なインデックスの付与は更新処理の性能低下にもつながるため、定義すべき列の選択について以下の観点で精査していきます。

- 検索SQLのうち以下の対象となる列に作成する
    1. WHERE句の条件
    2. テーブル結合の条件
    3. ORDER BYやGROUP BYに指定される列
- カーディナリティの高い列に作成する

その他、B-Treeインデックスにおいては更に性能を向上させられる応用的な使い方もあり [9] 、例として部分
ースもあります [10]

また、定義したインデックスが実際にはあまり使われていないと更新性能の足かせとなってしまいます。実運用
を参照し不要なインデックスを削除することも全
す [9]

考慮漏れによる起こりうる性能トラブルの例

意外と現場で起こりうるのがレコード数の多いテーブルに対するインデックスの定義漏れであり、seq scanが実
繋がることがあります。一定規模のテーブルに対するインデックスの定義漏れ、検討漏れに注意が必要です。

参考情報

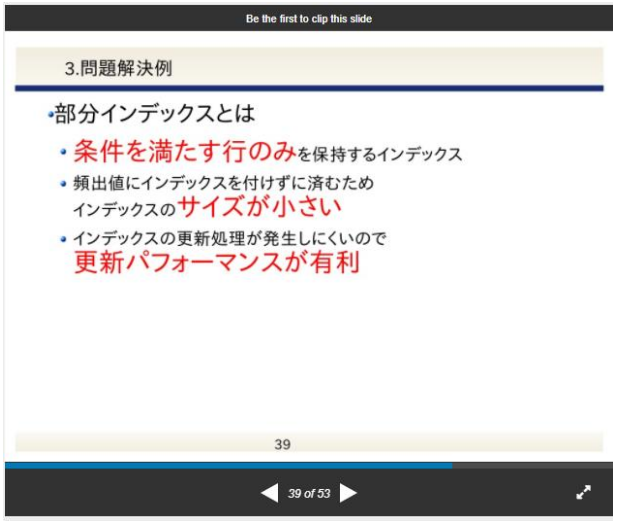
注釈	タイトル	URL	
[9] (1, 2)	PostgreSQLのインデックス・チューニング by Tomonari Katsumata	https://www.slideshare.net/InsightTechnology/dbts-osaka-2014-b23-postgresql-tomonari-katsumata	インデックスの有効活用の仕組みや、利用状況の確認の仕方など
[10]	PostgreSQL SQLチューニング入門 実践編	https://www.slideshare.net/satoshiyamada71697/postgresql-sql	部分インデックスを用いて性能改善させる実践例

ポイント

起こりうるトラブル

詳細情報はリンク先へ

既存の公開資料



報告書の記載例：テーブル設計のポイント

- テーブル設計/定義にあたって性能面に影響するポイント
  - 必要な情報を表現する最小のデータ型を選択する
    - numeric型は必要なケースのみ使用する。表現すべきデータの範囲に合わせてint型やreal型の使用を検討。 numeric型は精度は高いが、計算処理では明らかな性能差が出る
  - HOTを有効に活用できるようFILFACTORの値を検討する
    - デフォルトでは100だが、これを80～90にしておくと、UPDATEが多いケースでは大きな効果が出ることも

詳細なノウハウはリンクで紹介

- 考慮漏れで起こりうる性能トラブル
  - NUMBER型からの移行でnumeric型を選択し、SQL内の計算処理で性能遅延。要件は桁数であったためbigint型を選択し性能向上
  - UPDATEが多いシステムにおいてFILFACTORをデフォルトである100のまま運用し、データの規模が大きくなるにつれてある時UPDATEが大幅に遅延

性能トラブル
-検知-

トラブルの検知と監視

- 検知の基本は、『正常な値』を把握し、『変動』を捉えること
- その比較のために通常時の情報を定期取得し、蓄積しておく
- 監視対象の項目と取得方法、分析の観点などを紹介

項番	項目の種類	取得できる情報	検知できる予兆
1	SQL処理数、 実行時間	トランザクション数、 SQL数、遅延SQL	処理数の低下、 実行時間の長いSQLの検出など
2	特定の時刻における 処理状況	実行中のトランザクション、 実行中のVACUUM処理進捗状況	COMMIT漏れなどの ロングトランザクション、 VACUUMが阻害されているなど
3	HWのリソース情報	sarコマンドなどの取得結果	CPUやメモリ、I/O負荷など
4	各種統計情報	オブジェクト（テーブル、インデックス、ビュー）の各種情報やロックの状態 （次頁に紹介）	オブジェクトの肥大化、 不要領域の比率、 ロック待ちやデッドロックなど

各種統計情報で取得できる情報

表 4.7 データベース性能に影響を与える要素¶

項番	性能情報	種別	概要
1	ロングトランザクション	瞬間値	ロングトランザクション
3	オブジェクトサイズ	瞬間値	データベースサイズ テーブルサイズ インデックスサイズ
4	オブジェクトスキャン	累計値	テーブルスキャン時の読み込み行数 インデックススキャンの割合
5	オブジェクトの状態	累計値	テーブル毎の不要領域確認 テーブル断片化 インデックス断片化
6	キャッシュヒット率	累計値	データベース毎のキャッシュヒット率 テーブル毎のキャッシュヒット率 インデックス毎のキャッシュヒット率
7	共有メモリ状況	瞬間値	共有メモリ状況
8	接続数	瞬間値	データベース毎の同時接続数
9	ロック状態	瞬間値	ロック待ち状態 ロック待ちSQL
10	デッドロック	累計値	デッドロック回数 デッドロックSQL
11	ディスクソート	累計値	SQLによるディスクソートの処理回数(work_memの不足回数) ログメッセージの確認(ディスクソート)

スライドで
取得例を紹介

報告書の記載例：テーブル毎の不要領域確認－①

- pg_stat_user_tablesビューでは、推定値となるが有効行と不要行の現在値が収集されている
- 以下のようなSQLを用いることで、全体の行数の内と不要行の割合から不要領域を取得可能

```
SELECT
    relname,
    n_live_tup,
    n_dead_tup,
    CASE n_dead_tup
        WHEN 0 THEN 0
        ELSE
round(n_dead_tup*100/(n_live_tup+n_dead_tup),2)
    END AS ratio
FROM
    pg_stat_user_tables;
```

n_dead_tup(不要行)の割合を計算し、「ratio」で表示

報告書の記載例:テーブル毎の不要領域確認-②

- 先述のSQLにより以下のような結果が表示される

```
-[ RECORD 1 ]---------------
relname    | pgbench_history
n_live_tup | 41819
n_dead_tup | 0
ratio      | 0
-[ RECORD 2 ]---------------
relname    | pgbench_accounts
n_live_tup | 100000
n_dead_tup | 1813
ratio      | 1.00
-[ RECORD 3 ]---------------
relname    | pgbench_tellers
n_live_tup | 10
n_dead_tup | 22
ratio      | 68.00
           :
```

relname : テーブル名
n_live_tup : 有効行の推定値
n_dead_tup : 不要行の推定値
ratio : 不要領域の割合

報告書の記載例：テーブル毎の不要領域確認－③

- 不要領域の割合であるratioが長時間にわたって autovacuum_vacuum_scale_factorパラメータ[1]以上の場合、以下のようなトラブルが考えられる
  1. autovacuumが該当テーブルで実行されていない
  2. autovacuumが実行されているが、不要領域の回収を阻害する要因がある

```
-[ RECORD 3 ]---------------
relname    | pgbench_tellers
n_live_tup | 10
n_dead_tup | 22
ratio      | 68.00
```

[1] autovacuumを実行する契機となる、不要領域の割合の設定値。
デフォルトでは20%であり、
これを超えたテーブルに実行される

性能トラブル
-対処-

対処　ケーススタディについて

- トラブルの検知から解決までの方法を事例として紹介
  1. トラブルの内容
  2. 調査と分析の進め方
  3. 改善手法の紹介

- 報告書では以下の3つの事例を紹介
  - ロングトランザクションによる性能トラブル
  - ディスク性能の考慮漏れによる性能トラブル
  - 適切でない実行計画が選択されてしまうことによる性能トラブル

次ページ以降で紹介

ロングトランザクションによる性能トラブル-①

- **トラブルの内容**

  データベースを運用していたところ、しばらくは問題なく動作していたが
  いつ頃からか処理遅延が頻発するように。
  具体的に以下のような処理遅延を確認

1. **日中のトランザクション処理数が低下**

```
（正常期間）  tps = 773.07803
（遅延発生時）  tps = 400.656880
```

2. **夜間のバッチ処理内のSQLに遅延が発生**

   10秒以上を遅延とし、「log_min_duration_statement = 10s」を
   設定していたところ以下のようなログ出力を確認

```
[20XX-XX-XX XX:XX:XX.XXX JST][<ユーザ名>][<データベース名>][[<ホスト
名]][<PID>][<セッションID>-<各セッションのログ行の番号>][0][00000]
psql LOG:  duration: 17374.466 ms  statement: SELECT bid ,count(*) FROM
pgbench_accounts GROUP BY bid;
```

ロングトランザクションによる性能トラブル-②

- **調査と分析の進め方**

1. **テーブルサイズの確認**
   今回の事例ではデータベースの処理が全体的に遅延しており
   データベースのテーブルサイズをまず確認する(pg_classカタログを参照)

正常時

```
     relname      | reltuples | tablesize
------------------+-----------+-----------
 pgbench_history  |    256897 | 15 MB
 pgbench_tellers  |      1000 | 128 kB
 pgbench_branches |       100 | 96 kB
 pgbench_accounts |     1e+07 | 1299 MB
```

遅延発生時

```
     relname      | reltuples | tablesize
------------------+-----------+-----------
 pgbench_history  |    975506 | 51 MB
 pgbench_tellers  |      1000 | 42 MB      -レコード数は変化していないが、テーブルサイズが顕著に増加
 pgbench_branches |       100 | 34 MB      -レコード数は変化していないが、テーブルサイズが顕著に増加
 pgbench_accounts |     1e+07 | 1407 MB
```

テーブルサイズが顕著に増加！！
レコード数が増えた？ ⇒ 変化なし
VACUUMに問題ありか？

ロングトランザクションによる性能トラブル－③

- **調査と分析の進め方**

2. **不要領域の確認**
   VACUUM処理による不要領域の回収状況を確認（pg_stat_user_tables）

遅延発生時

relname	n_live_tup	n_dead_tup	ratio
pgbench_branches	100	984767	99.00
pgbench_history	998339	0	0
pgbench_accounts	10000006	999295	9.00
pgbench_tellers	1000	984779	99.00

不要領域が99%！！
autovaccumに問題がある
(通常、20%溜まると実行)

VACUUM処理が阻害されている。
ロングトランザクションが発生しているのでは？

ロングトランザクションによる性能トラブル－④

- **調査と分析の進め方**

3. **稼動統計情報の確認**
   稼働統計情報のうち、pg_stat_activityでトランザクションの状態を確認

遅延発生時

```
      :
-[ RECORD 7 ]---+-----------------------------------------------------------------------------
-------------------------------------------------------------------------------------------
-------------------------------------------------------------
pid             | 6763
wait_event_type | Client
wait_event      | ClientRead
state           | idle in transaction
duration        | 00:30:10.235
query           | INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (111111111,
1, 1, 0, CURRENT_TIMESTAMP);
      :
```

トランザクションが長時間
閉じられていない！

ロングトランザクションが発生している事と、該当のクエリまで特定できた！

ロングトランザクションによる性能トラブル－⑤

- 改善手法の紹介
  - 一般的にロングトランザクション化する要因として、
    アプリケーション側のコミット漏れが多い。
    アプリケーション側に問題がある場合、改修と改善は容易ではない
  - 本件の場合、一時的な回避策として以下のような方法がある

1. PostgreSQLの設定ファイルで
   「idle_in_transaction_session_timeout（PostgreSQL 9.6～）」
   を任意の時間に設定し、設定ファイルの変更を反映（pg_ctl reload）
2. 対象のプロセスをpg_terminate_backendコマンドで終了

上記はあくまで暫定対策であり、アプリケーション側でトランザクション
を閉じるなど、根本的な改修を検討すべき

まとめと所感

- トラブルの早期検知、発生時の原因究明時
  ともに『正常時の値』との比較が重要
  - 取得情報および方法、変動の要因について整理
  - 性能監視/収集ツールなどを活用し、情報収集や蓄積を行うことも検討
    - 2015 年度 WG3 活動報告書 (4.性能監視ツール)

- 性能トラブルを予防するためのノウハウについて調査を行う上で
  既存の技術資料が大変参考になった
  - いくつかの技術情報へのリンクを資料内に記載

- 性能トラブルについては成果物で紹介できないものも多く
  紹介可能な事例を探すことに苦労した

**紹介できる(面白い)性能トラブル事例を
お持ちの方は一緒に活動しませんか？**

PGECons

PostgreSQL Enterprise Consortium