株式会社メトロシステムズ

LPI-JAPAN
Linux Professional Institute Japan

OSS-DB Exam Gold 技術解説無料セミナー

2016/3/27

株式会社メトロシステムズ
佐藤　千佳

自己紹介

■氏名
- 佐藤　千佳（さとう　ちか）

■所属
- 株式会社メトロシステムズ

■略歴
- 2006年に株式会社メトロシステムズ入社
  - Oracleを用いた業務システムの保守・管理を担当
- 2007年からオープンソースデータベースを担当する部署に所属
  - PostgreSQL機能調査、周辺ツールを含めた性能評価を担当
  - PostgreSQLと他DBMSとの機能比較、性能比較評価なども担当

本日の流れ

- 試験概要
- 運用管理
  - 運用管理コマンド全般
    - バックアップとリカバリ、リストア
    - 日常的なメンテナンス
    - サーバログ監視
  - ホット・スタンバイ運用
- アクセス統計情報
- クエリ実行計画

試験概要

OSS-DB技術者認定資格

オープンソースデータベース（OSS-DB）に関する技術と知識を認定するIT技術者認定

OSS-DB /Silver

データベースシステムの設計・開発・導入・運用ができる技術者

OSS-DB /Gold

大規模データベースシステムの
改善・運用管理・コンサルティングができる技術者

OSS-DB技術者認定資格の必要性

商用/OSSを問わず様々なRDBMSの知識を持ち、データベースの構築、運用ができる、または顧客に最適なデータベースを提案できる技術者が求められている

OSS-DB Gold試験の出題範囲

■運用管理（30%）

- データベースサーバ構築
- 運用管理用コマンド全般
- データベースの構造
- ホット・スタンバイ運用

■性能監視（30%）

- アクセス統計情報
- テーブル / カラム統計情報
- クエリ実行計画
- その他の性能監視

■パフォーマンスチューニング（20%）

- 性能に関係するパラメータ
- チューニングの実施

■障害対応（20%）

- 起こりうる障害のパターン
- 破損クラスタ復旧
- ホット・スタンバイ復旧

試験時間　：90分※
試験方式　：CBT
問題数　：30問
合格点　：70点

※アンケート時間等を含む

OSS-DB試験の注意事項

■最新の試験範囲はWebで確認！
- http://www.oss-db.jp/outline/examarea.shtml

■試験対象のPostgreSQLバージョンは「9.0以上」
- 試験はPostgreSQL9.2に対応
- 2016年3月時点の対応バージョンは「9.2.15」

■OSに依存しない内容だが、表記はLinuxベース
- シェルのコマンドプロンプトは「$」
- 「フォルダ」でなく「ディレクトリ」
- ディレクトリ区切り文字は「￥」や「＼」でなく「/」

運用管理

運用
管理

- バックアップとリストア、リカバリ
- 日常的なメンテナンス
  - VACUUM、ANALYZE
- サーバログ監視

バックアップとリストア、リカバリ

■バックアップとは

- 障害が発生してもデータを紛失しないための仕組み

■障害の種類

- データベースで想定すべき障害はさまざま
  - オペレーションミスなどのヒューマンエラー
  - OSダウンや電源断等のインスタンス障害
  - ネットワーク障害やH/Wの故障
  - 規模の大きなものではディザスタ障害

用語の整理

バックアップとリストア、リカバリで使う用語

- データベースクラスタ
  - データベースを構成するファイル群
- WAL
  - データベースに対する変更履歴ログ（トランザクションログ）
- アーカイブログ
  - 変更履歴ログ（WAL)のアーカイブ
- 物理バックアップ
  - データベースクラスタを構成するファイル群を物理的に別の媒体へコピー（バックアップ）すること
- 論理バックアップ
  - 表の定義や格納されているデータなどをファイルに出力すること
    - SQLファイルやバイナリファイル

用語の整理

- コールドバックアップ（オフラインバックアップ）
  - データベースを停止した状態で取得するバックアップ
- ホットバックアップ（オンラインバックアップ）
  - データベースを稼働させた状態で取得するバックアップ
- リストア
  - バックアップファイルを再ロード、または再配置すること
    - リストアだけではデータベースはバックアップ取得時点の内容
- リカバリ
  - バックアップ取得以降にデータベースに行われた変更を反映し、任意の時点までデータベースの内容を戻すこと

運用管理

PostgreSQLのバックアップとリカバリ

バックアップ種類	コールドバックアップ	エクスポート	ホットバックアップ
リカバリできる範囲	バックアップ取得時点	バックアップ取得開始時点	ベースバックアップ取得以降の任意の時点（PITR）
メジャーバージョン移行	不可	可	不可
バックアップ単位	DBクラスタ全体	テーブル データベース 全データベース	DBクラスタ全体
設定ファイル	含む	含まない	含む

PostgreSQLのバックアップ種類

	論理バックアップ	物理バックアップ
オフライン	なし	**コールドバックアップ** • cp • rsync • tar • ストレージスナップショット ...等
オンライン	**エクスポート** pg_dump Pg_dumpall	**ホットバックアップ** • pg_start_backup() / pg_stop_backup() + アーカイブログ • pg_basebackup + アーカイブログ

コールドバックアップの特徴

コールドバックアップ

- インスタンス停止状態でデータベースクラスタ全体をコピー
  - rsyncコマンドやtarコマンドを使うのが一般的
  - ストレージのスナップショット機能も利用可能
- バックアップした時点にのみリカバリ可能
- 設定ファイルなどもまとめてバックアップされるので手順がシンプル
- リストア時間は他の手法と比べて短い
- バックアップを類似構成の別マシンにコピーすることも可能
- メジャーバージョンが一致していること

コールドバックアップ実行方法

運用
管理

■バックアップ方法

1. データベースインスタンスを停止させる
   - 稼働中にバックアップすると一貫性がなくリカバリできない
2. データベースクラスタ全体をコピーする
   - cp -rp $PGDATA $BACKUP_DIR/
   - tar zcf $PGDATA $BACKUP_DIR/pgdata.tar.gz
   - ストレージのスナップショット機能でボリューム全体をコピー

■リストア方法

1. 古いデータベースクラスタを退避または削除する
2. 元のデータベースクラスタの位置にバックアップを展開する
   - 展開方法はバックアップ方法に応じる
   - 展開先は別の場所でもOK（ただしPGDATA環境変数に注意）
3. データベースインスタンスを起動する

ホットバックアップの特徴

運用
管理

ホットバックアップ

- インスタンス稼働状態で論理バックアップを取得
- データはバックアップ開始時点のもので、一貫性あり
  - バックアップした時点にのみリカバリ可能
- pd_dumpコマンドまたはpg_dumpallコマンドで実施
  - pg_dump　　：データベース単位
  - pg_dumpall　　：ユーザ情報を含むデータベースクラスタ全体
    - --dbnameオプション指定でDB単位で取得可能
- 設定ファイルは別途バックアップしておく必要がある
- バックアップ取得元とリカバリ先でメジャーバージョンが異なっていてもリストア可能

pg_dump

■バックアップ方法

- 基本構文
  - pg_dump [接続オプション] [オプション] [データベース名]
- オプション
  - -f|--filename=出力ファイル名（省略すると標準出力にダンプ）
  - -F|--format＝出力フォーマット
  - p/plain　　：テキスト形式（デフォルト）
  - c/custom　　：バイナリ形式・自動的に圧縮
  - t/tar　　：tar形式

■リストア方法

1. 事前にデータベースクラスタを作成してインスタンスを起動
2. リストア先データベースを作成（別データベースでも可）
3. psqlコマンドまたはpg_restoreコマンドでリストア
   - psql -f ダンプファイル名 データベース名
   - pg_restore -d データベース名 ダンプファイル名

pg_dumpall

■バックアップ方法

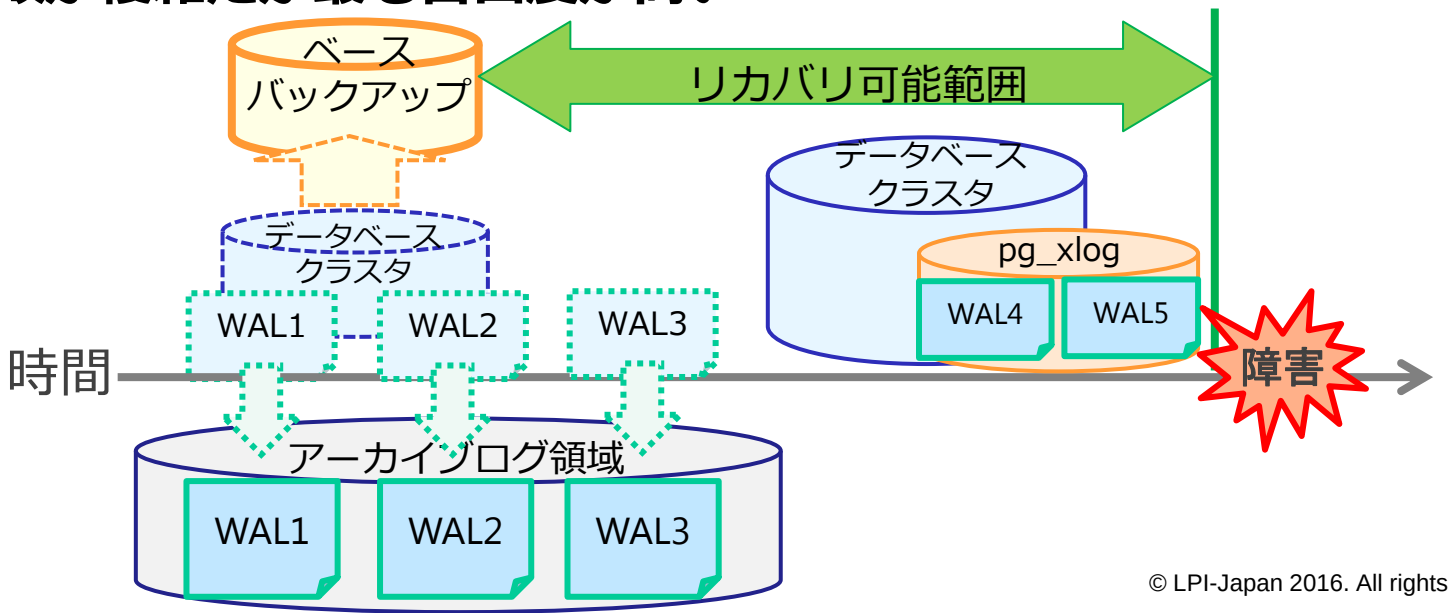
- 基本構文
  - pg_dumpall [接続オプション] [オプション]
- オプション
  - -f|--filename=出力ファイル名（省略すると標準出力にダンプ）
  - -g|--globals-only : グローバルオブジェクトのみバックアップ

■リストア方法

1. 事前にデータベースクラスタを作成
2. テキスト形式なのでpsqlコマンドでリストア
   - psql -f ダンプファイル名 postgres

PITR（Point In Time Recovery）

- 運用中に出たWALをアーカイブしておき、定期的にインスタンス稼働中のデータベースクラスタ全体をコピー（ベースバックアップ）
- ベースバックアップ取得前後にインスタンスにバックアップ開始・終了を通知する必要がある
- ベースバックアップ取得から障害発生までの任意の時点の状態にリカバリできる
  - オペレーションミスなどへの対応に便利
- 手順が複雑だが最も自由度が高い



PITRの事前準備

- GUCパラメータpostgresql.confにて、普段からWALを安全な場所にアーカイブしておく
  - wal_level=archive|hot_standby
  - archive_mode=on
  - archive_command='cp -i %p /path/to/archive/%f'

デフォルト設定ではアーカイブされない

Point In Time Recovery

運用
管理

■バックアップ方法

- 低レベルAPI
  1. SELECT pg_start_backup('バックアップラベル');
  2. データベースクラスタ全体をコピー
     - コピー開始後のファイル変更をエラーとしないように注意
  3. SELECT pg_stop_backup();
     - ベースバックアップが取れたらそれ以前のWALは破棄可能
- pg_basebackup
  - バックアップ開始・終了の通知関数が不要
  - リモートからでも取得可能（レプリケーションプロトコル使用）
  - スーパーユーザまたはREPLICATION権限のあるユーザで実行
  - GUCパラメータmax_wal_sendersを1~2増やす

```
$ pg_basebackup -D /home/postgres/backup/basebackup -U repli -h localhost --xlog --
checkpoint=fast --progress
93665/93665 kB (100%), 1/1 tablespace
```

Point In Time Recovery

運用
管理

リストア方法

1. インスタンスが稼働している場合は停止
2. $PGDATA/pg_xlogの中身（WAL)を任意の場所に退避
   - 障害直前に変更内容はここにしかないので忘れずに！
3. データベースクラスタを退避または削除
4. ベースバックアップを$PGDATAに再配置
5. $PGDATA/pg_xlogの中を全て削除し(２)で退避したWALをコピー
6. recovery.confを作成し$PGDATA配下に配置
   - restore_command = 'cp /path/to/archive/%f %p'
   - アーカイブログの格納先を指定

Point In Time Recovery

運用
管理

リカバリ方法

1. pg_hba.confを編集して一般ユーザの接続を拒否
2. インスタンスを起動し、自動的にリカバリを開始させる
   - リカバリが完了するとrecovery.confがrecovery.doneにリネーム
   - サーバーログに「LOG: archive recovery complete」と出力される
3. データベースの内容を確認
4. pg_hba.confを元の設定に戻して設定をリロード

便利な外部ツール

pg_rman

- PostgreSQLのオンライン・バックアップ、リストアツール
- オンランイン・バックアップやPITRを簡単操作で実現
  - メンテナンスコマンドも充実
  - バックアップの世代管理も可能
    https://sourceforge.net/projects/pg-rman/

Barman

- PostgreSQLのオンライン・バックアップ、リストア、リカバリツール
- pg_rman同様簡単操作でバックアップ、リカバリを実現
- バックアップ時のI/O、ネットワーク帯域の上限指定が可能

  https://sourceforge.net/projects/pgbarman/

- バックアップとリストア、リカバリ
- **日常的なメンテナンス**
  - **VACUUM、ANALYZE**
- サーバログ監視

日常的なメンテナンス

運用
管理

なぜメンテナンスが必要なのか

- データベースが安定して稼働するには日常的なメンテナンスが重要
  - メンテナンスを怠ると不要領域（後述）の増大を招く
  - データ量は増えていないのに、データの追加、更新、削除を繰り返すことでディスクの利用効率が悪くなる
  - 性能劣化の原因に！

PostgreSQLの主なメンテナンスコマンド

運用
管理

- テーブル、インデックスの不要領域回収
  - VACUMM
- インデックスの再作成（未使用領域のないインデックスの作成）
  - REINDEX
- インデックス順にテーブルデータを再編成（並べ替える）
  - CLUSTER
- データの並び順や物理的な配置などのデータ分布を管理している統計情報を最新の状態にする
  - ANALYZE

不要領の回収

■不要領域とは

- **PostgreSQLは追記型アーキテクチャを採用**
  - **更新処理によってデータファイル内に更新前データが蓄積されていく**
  - **複数のユーザからの同時処理を実現するためにMVCCを採用**
    - 更新や削除が実行されると対象データに更新済みマークをつけ、更新後データは別の場所に保存する
    - 更新したトランザクション以外が更新前のデータをロック競合なしで参照できる
- **運用を進めていくと、更新前のデータが残りデータファイルのサイズが実データ量よりも大きくなっていく**
  - **ファイルサイズが大きくなると、ストレージ容量の不足やパフォーマンス悪化といった問題が発生**

「VACUUM」処理により不要領域を回収

VACUUM

VACUUMとは

- 不要領域を回収してデータファイル肥大化を防ぐ
  - 更新されて不要となった行（不要領域）を削除または再利用しデータの肥大化を抑える
- プランナ用の統計情報更新
  - 非効率的な実行計画を防ぐように統計情報を更新する
- Visibility Mapの更新
  - 効率よくVACUUM処理を実施する
  - Index Only Scan機能を使いやすくなる
- トランザクションIDの周回エラー防止
  - トランザクションIDが最大数（約42億（32bitの限界））を超過することを防ぐ

■データファイルの内部構造イメージ

所属	名前	削除
総務	田中	
経理	佐藤	
営業	鈴木	

UPDATE

所属	名前	削除
総務	**田中**	×
経理	佐藤	
営業	鈴木	
経理	**田中**	

UPDATE

所属	名前	削除
総務	田中	×
経理	**佐藤**	×
営業	鈴木	
経理	田中	
総務	**佐藤**	

VACUUM

所属	名前	削除
広報	田中	
経理	**橋本**	
営業	鈴木	
経理	田中	×
総務	佐藤	

INSERT

所属	名前	削除
広報	**田中**	
営業	鈴木	
経理	**田中**	×
総務	佐藤	

UPDATE

所属	名前	削除
営業	鈴木	
経理	田中	
総務	佐藤	

VACUUM実行方法

VACUUM

- SQL文にてVACUUMを実施する手法
- 基本構文
  - VACUUM [オプション] [テーブル名];
- オプション
  - FULL：不要領域回収後、データファイルを切り詰める

vacuumdbコマンド

- OSコマンドにてVACUUMを実施する手法
- 基本構文
  - vacuumdb [接続オプション] [オプション] [データベース名]
- オプション
  - -f|--full　　：不要領域回収後、データファイルを切り詰める
  - -z|--analyze　：ANALYZEを併せて実行する

VACUUMとVACUUM FULL

VACUUM

所属	名前	削除
総務	田中	×
経理	**佐藤**	×
営業	鈴木	
経理	田中	
総務	**佐藤**	

所属	名前	削除
営業	鈴木	
経理	田中	
総務	佐藤	

UPDATE

所属	名前	削除
広報	**田中**	
営業	鈴木	
経理	**田中**	×
総務	佐藤	

VACUUM FULL

所属	名前	削除
営業	鈴木	
経理	田中	
総務	佐藤	

UPDATE

所属	名前	削除
営業	鈴木	
経理	**田中**	×
総務	佐藤	
広報	**田中**	

自動VACUUM

自動VACUUM（AUTOVACUUM）とは

- VACUUM実行専用のプロセス
  - 一定以上の割合で更新されたテーブルを自動的にVACUUM
- 通常のクエリにあまり影響を与えないように、ある程度処理したらしばらくスリープする、というサイクルで実行

自動VACUUM

■自動VACUUM（AUTOVACUUM）対象となるテーブル

- VACUUM
  - レコードの更新、削除が閾値を超えたらVACUUM対象になる
  - 閾値の設定はGUCパラメータ（postgresql.conf）にて行う

autovacuum_vacuum_threshold +
autovacuum_vacuum_scale_factor × レコード数

（例）
autovacuum_vacuum_threshold=50(def.)
autovacuum_vacuum_scale_factor=0.2(def.)

レコード数=100の場合
テーブル内の70件が更新、削除された場合にVACUUM対象となる

自動VACUUM

- **ANALYZE**
  - **レコードの追加、更新、削除が閾値を超えたらANALYZE対象になる**
  - **閾値の設定はGUCパラメータ（postgresql.conf）にて行う**

    autovacuum_analyze_threshold +
    autovacuum_analyze_scale_factor × レコード数

  - **テーブルの追加、更新、削除されたタプル数の確認方法**
    - pg_stat_user_tablesのn_tup_*情報を確認

- バックアップとリストア、リカバリ
- 日常的なメンテナンス
  - VACUUM、ANALYZE
- **サーバログ監視**

なぜ監視が必要なのか

運用
管理

なぜ監視するのか

- データベースを健全な状態に保つ
- 「いつの間にか止まっていた」とならないように
- 「いつの間にか遅くなっていた」とならないように

監視に求められること

- 必要な情報を収集し現状を把握する
- 今後予想される状況を推測する

運用が始まってから監視を始めるのでは遅い！

何のために、何を監視するのかを明確にすることが重要

- 「監視するだけ」にならないようなフィードバックサイクルを作る
- データベースのチューニング
- ハードウェアの増強
- 運用開始前からどのような項目を監視するか決めておく
- 監視によるオーバーヘッドを見込んでサイジングする

何を監視するのか

運用管理

- OSレベルの監視
  - CPU、メモリ、ネットワーク、ディスク、プロセス
    - sar、dstat、vmstat、mpstat、free、top、netstat、iostat、ps
- データベースレベルの監視
  - SQLパフォーマンス監視
    - セッション数
    - クエリ所要時間
    - テーブルやインデックスへのアクセス数
    - キャッシュヒット率
    - WAL書き込み（ディスクI/O）
  - ディスク領域監視
    - データ領域（データベース、テーブルスペース、オブジェクト）
    - トランザクションログ領域
    - アーカイブログ領域
  - サーバログ監視
    - FATALログ、ERRORログ、WARNINGログ、LOGログ

ログの取得

運用管理

- ログは監視の第一歩
  - ログから何が分かるのか
  - 必要な情報を取得できるように適切に設定する
- PostgreSQLのログで分かること
  - エラーなどの異常処理
  - 接続、切断情報
  - スロークエリ
  - 実行クエリ
    - ユーザ毎に出力の有無を設定可能
  - AUTOVACUUM、CHECKPOINTなどの処理情報
- ログの設定
  - デフォルトの設定ではログ解析に必要な情報は付与されない
  - 以下のパラメータについては必ず設定
    - log_line_prefix
    - log_destination
    - logging_collector

問題が起こってからでは遅い

ログの取得

log_line_prefix

- ログ行の先頭に付加する文字列パターンを指定する
  - %t：タイムスタンプ（年月日時分秒までの精度）
  - %u：ユーザ名
  - %d：データベース名
  - %p：プロセスID
  - %c：セッションID
  - %x：トランザクションID（トランザクション外の場合は「0」）
  - %e：SQLSTATEエラーコード
  - %a：アプリケーション名
  - %%：「%」そのもの

データ型	文字列
デフォルト値	''（何も付加しない）
変換可能タイミング	リロード時

※ ログ本文と連結して読みにくくなるため、設定値の最後に空白を入れる

ログの取得

log_destination

- サーバ側ログの出力先を指定する
  - stderr：テキスト形式で標準エラーに出力
  - csvlog：CSV形式で標準エラーに出力（要logging_collector=on）
  - syslog：テキスト形式でsyslogに出力
  - eventlog：Windowsのイベントログに出力（Windows環境のみ）
    - 文字化けするという情報あり
- 複数指定する場合はカンマで区切る

データ型	文字列
デフォルト値	'stderr'
変換可能タイミング	リロード時

ログの取得

logging_collector

- サーバプロセスの標準エラー出力をファイルにリダイレクトするかを指定する
- log_destination=csvlogを使うときは「on」にする必要がある
- 外部ツールでログファイルを切り替える場合は「off」にする

データ型	論理値
デフォルト値	off
変換可能タイミング	インスタンス起動時

この設定を「on」にすると、「logger process」という役割のpostgresプロセスが起動する

スロークエリの監視

スロークエリの監視方法

1. GUCパラメータで設定する
   - log_min_duration_statementパラメータ
2. contribモジュールを使い、ログにSQLの実行計画を出力する
   - auto_explainモジュール

log_min_duration_statement

- 指定時間以上かかったSQL文を処理時間と共にログに出力する
- 「0」を指定すると実行された全てのSQLをログに出力
- 値の設定には配慮が必要
  - 多くのログを出力すると出力自身がオーバーヘッドになる

データ型	数値
デフォルト値	-1（無効）
変換可能タイミング	リロード時

スロークエリの監視

log_min_duration_statement 出力例

```
$ psql -c "SHOW log_min_duration_statement" postgres
 log_min_duration_statement
----------------------------
 3ms
(1 行)

$ less /home/postgres/pgdata/pg_log/postgresql-2016-03-03_211959.log
...
2016-03-03 21:20:59 JST postgres satock 25402 100897 LOG:  duration: 5.059 ms
statement: SELECT abalance FROM pgbench_accounts WHERE aid = 99573;
2016-03-03 21:20:59 JST postgres satock 25402 100938 LOG:  duration: 8.678 ms
statement: UPDATE pgbench_accounts SET abalance = abalance + 2006 WHERE aid = 50075;
...
```

処理に3ミリ秒以上
要したSQLが出力

スロークエリの監視

auto_explain

- 処理時間が設定値を超過したSQLについて、実行計画をロギングしてくれるcontribモジュール

事前準備

- インストール
  - ソースからビルドしてインストール
    postgresql-9.2.15/contrib/auto_explainをビルド&インストール
  - パッケージ管理システムからインストール
    postgresql92-contribパッケージをインストール

GUCパラメータの設定

- shared_preload_librariesパラメータにauto_explainモジュールの共有ライブラリを読み込むように設定
  - 設定の反映にはデータベースの再起動が必要
- auto_explain.log_*パラメータの設定

スロークエリの監視

auto_explain GUCパラメータの設定例

```
$ less /home/postgres/pgdata/postgresql.conf
…
shared_preload_libraries = 'auto_explain'
auto_explain.log_min_duration = '3ms'  # 3ミリ秒以上要したSQLの実行計画をログ出力
auto_explain.log_verbose = on          # verbose情報を出力
…

$ pg_ctl restart -D /home/postgres/pgdata
サーバ停止処理の完了を待っています.....完了
…
```

スロークエリの監視

auto_explain 出力例

```
$ less /home/postgres/pgdata/pg_log/postgresql-2016-03-04_105927.log
...
2016-03-04 11:11:01 JST postgres postgres 18369 200856 LOG:  duration: 4.444 ms  plan:
        Query Text: UPDATE pgbench_accounts SET abalance = abalance + -1404 WHERE aid =
8006902;
        Update on public.pgbench_accounts  (cost=0.56..8.58 rows=1 width=103)
          ->  Index Scan using pgbench_accounts_pkey on public.pgbench_accounts
(cost=0.56..8.58 rows=1 width=103)
                Output: aid, bid, (abalance + (-1404)), filler, ctid
                Index Cond: (pgbench_accounts.aid = 8006902)
2016-03-04 11:11:01 JST postgres postgres 18369 200856 LOG:  duration: 4.886 ms
statement: UPDATE pgbench_accounts SET abalance = abalance + -1404 WHERE aid = 8006902;
...
```

VACUUMの監視

VACUUMの監視方法

- GUCパラメータで設定する
  - log_autovacuum_min_durationパラメータ

監視ポイント

- VACUUM実行頻度
  - 頻繁に動いていないか
- VACUUM実行時間帯
  - 負荷の高い時間帯に動いていないか
- VACUUM所要時間
  - VACUUMに異常な時間を要していないか
- ガベージの回収が阻害されていないか
  - ロングトランザクションの可能性あり

log_autovacuum_min_duration

- 指定時間以上かかったAUTOCACUUM処理をログに出力する
- 「0」を指定すると全てのAUTOVACUUM処理をログに出力する

データ型	数値
デフォルト値	-1（無効）
変換可能タイミング	リロード時

設定例

```
$ less /home/postgres/pgdata/postgresql.conf
…
log_autovacuum_min_duration = 1min
…

$ pg_ctl reload -D /home/postgres/pgdata
サーバにシグナルを送信しました
…
```

VACUUMの監視

運用
管理

log_autovacuum_min_duration 出力例

```
$ less /home/postgres/pgdata/pg_log/postgresql-2016-03-04_105927.log
...
2016-03-04 11:09:51 JST   18324 0 LOG:  automatic vacuum of table
"postgres.public.pgbench_tellers": index scans: 0
    pages: 0 removed, 220 remain
    tuples: 1074 removed, 6836 remain
    buffer usage: 506 hits, 0 misses, 0 dirtied
    avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
    system usage: CPU 0.00s/0.01u sec elapsed 10.24 sec
...
```

ANALYZEの監視

ANALYZEの監視方法

- GUCパラメータで設定
  - log_autovacuum_min_durationパラメータ

監視ポイント

- ANALYZE実行頻度
  - 頻繁に動いていないか
- ANALYZE実行時間帯
  - 負荷の高い時間帯に動いていないか
- ANALYZE所要時間
  - ANALYZEに異常な時間を要していないか

ANALYZEの監視

log_autovacuum_min_duration

- 「VACUUMの監視」と同じ
  - 指定時間以上かかったAUTOANALYZE処理をログに出力する
  - 「0」を指定すると全てのAUTOANALYZE処理をログに出力する

出力例

```
$ less /home/postgres/pgdata/pg_log/postgresql-2016-03-04_105927.log
...
2016-03-04 11:08:51 JST   18297 196254 LOG:  automatic analyze of table
"postgres.public.pgbench_tellers" system usage: CPU 0.08s/0.05u sec elapsed 28.05 sec
...
```

CHECKPOINTの監視

CHECKPOINTの監視方法

- GUCパラメータで設定する
  - log_checkpointsパラメータ

監視ポイント

- CHECKPOINT実行間隔
  - 適切な間隔で動いているか：頻発するとパフォーマンスに影響
  - 所要時間は意図した通りか
  - バックグランドライタやバックエンドによる書き出しが多く発生していないか ⇒ pg_stat_bgwriterビュー

```
my_db=# SELECT * FROM pg_stat_bgwriter;
-[ RECORD 1 ]---------+------------------------------
checkpoints_timed     | 20
checkpoints_req       | 1
checkpoint_write_time | 5308471
checkpoint_sync_time  | 9859
buffers_checkpoint    | 60340
buffers_clean         | 0
maxwritten_clean      | 0
buffers_backend       | 227643
buffers_backend_fsync | 0
buffers_alloc         | 143517
stats_reset           | 2016-03-07 12:23:50.405225+09
```

CHECKPOINTの監視

log_checkpoints

- チェックポイントの開始と終了、書き出したバッファ数等の統計情報をログに書き出す

データ型	論理値
デフォルト値	off
変換可能タイミング	リロード時

出力例

```
$ less /home/postgres/pgdata/pg_log/postgresql-2016-03-04_105927.log
…
2016-03-04 11:09:23 JST   18009 0 LOG:  checkpoint starting: time
…
2016-03-04 11:10:53 JST   18009 0 LOG:  checkpoint complete: wrote 294737 buffers
(35.1%); 0 transaction log file(s) added, 0 removed, 553 recycled; write=345.759 s,
sync=51.833 s, total=399.209 s; sync files=92, longest=4.678 s, average=0.563 s
…
```

ディスク領域監視

運用
管理

ディスク使用量の監視方法

- データ領域
  - OSコマンドやPostgreSQLの関数を用いて確認
- トランザクションログ領域
  - WALファイルの1つのサイズは16MB
  - 基本的には循環的に使用されるため最大容量はGUCパラメータで決まる
    16MB ×（checkpoint_segments × 3 + 1）
- アーカイブログ領域
  - OSコマンドを用いて確認

ディスク領域監視

■監視する場所

- データ領域
  - $PGDATA/base
  - $PGDATA/base_pgsql_tmp
  - $PGDATA/pg_log
    - log_directoryパラメータで指定
  - テーブルスペース
- トランザクションログ領域
  - $PGDATA/pg_xlog
- アーカイブログ領域
  - アーカイブログディレクトリ
    - archive_commandパラメータで指定

ディスク領域監視

■データ領域

- **$PGDATA/baseとテーブルスペース以外はlsやduなどOSコマンドで確認**
- **オブジェクトサイズ（$PGDATA/baseとテーブルスペース）は関数を使って確認**

オブジェクト	関数名
データベース	pg_database_size()
テーブル	pg_relation_size()
インデックス	pg_relation_size()
テーブル（TOAST含む）	pg_table_size()
テーブル内全インデックス	pg_indexes_size()
テーブル（TOAST含む）＋インデックス	pg_total_relation_size()
テーブルスペース	pg_tablespace_size()

■オブジェクトサイズの確認 出力例

```
# データベースのサイズ
postgres=# SELECT pg_database_size('postgres');
 pg_database_size
------------------
        167946424
(1 行)

# テーブルのサイズ
postgres=# SELECT pg_relation_size('pgbench_accounts');
 pg_relation_size
------------------
        136593408
(1 行)

# インデックスのサイズ
postgres=# SELECT pg_relation_size('pgbench_accounts_pkey');
 pg_relation_size
------------------
         22487040
(1 行)
```

```
# テーブル（TOAST含む）のサイズ
postgres=# SELECT pg_table_size('pgbench_accounts');
 pg_table_size
---------------
     136658944
(1 行)

# テーブルに付与されている全てのインデックスのサイズ
postgres=# SELECT pg_indexes_size('pgbench_accounts');
 pg_indexes_size
-----------------
        22487040
(1 行)

# テーブル（TOAST含む）＋インデックスのサイズ
postgres=# SELECT pg_total_relation_size('pgbench_accounts');
 pg_total_relation_size
------------------------
              159145984
(1 行)
```

ディスク領域監視

■アーカイブログ領域

- lsやduなどOSコマンドで確認

■トランザクションログ領域

- lsやduなどOSコマンドで確認

■注意点

- VACUUMの実施
  - トランザクションログが大量に出る
  - アーカイブログ領域に注意
- REINDEXやALTER TABLEの実施
  - 内部的には、新規作成して古いものと切り替える
    - 実施前には最低限、対象テーブルの1.5倍の空き領域があることを確認

サーバログ監視

■ログレベルの意味

レベル	syslog	eventlog	意味
PANIC	CRIT	ERROR	データベースインスタンス全体に影響する問題 全ての接続が切断されPostgreSQLが停止する
FATAL	ERR	ERROR	セッションレベルに影響する問題
ERROR	WARNING	ERROR	SQLレベルに影響する問題
LOG	INFO	INFORMATION	パフォーマンスや内部処理のエラーに関する情報
WARNING	NOTICE	WARNING	使い方についての警告
NOTICE	NOTICE	INFORMATION	ユーザ補助となる情報
INFO	INFO	INFORMATION	ユーザによって明示的に出力を指定された情報
DEBUG	DEBUG	INFORMATION	開発者向けの詳細な情報を出力 レベル1～5まで指定可能

監視に使えるツール

運用管理

Zabbix＋pg_monz

- Zabbix
  - Zabbixはオープンソースの監視ツール
  - サーバとエージェントの組み合わせで動作
- pg_monz
  - TIS株式会社とSRA OSS, Inc. 日本支社が共同開発した、Zabbix向けのPostgreSQL監視用テンプレート
  - PostgreSQL 9.2以上に対応
  - 一時配布元はhttp://pg-monz.github.io/pg_monz/
- 監視項目
  - サーバ死活
  - サーバログ（PANIC/FATAL/ERROR）
  - データベースサイズ（グラフあり）
  - プロセス数、状態別接続数（グラフあり）
  - チェックポイント回数（グラフあり）
  - キャッシュヒット率（グラフあり）
  - デッドロック発生回数（グラフあり）
  - COMMIT/ROLLBACK回数（グラフあり）
  - 一時ファイル書き込み量（グラフあり）
  - 状態別の滞留バックエンド数

監視に使えるツール

- pg_statsinfo＋pg_stats_reporter
  - pg_statsinfo
    - NTTが開発したPostgreSQL専用の性能監視ツール
    - ある時点の統計情報を「スナップショット」として定期的に保存
    - サーバログの解析やアラート機能
    - スナップショットの保存先として「リポジトリDB」が必要
    - PostgreSQL8.3～対応、OSはRed hat系のLinuxに対応
    - 一時配布元は http://pgstatsinfo.projects.pgfoundry.org/index_ja.html
  - pg_stats_reporter
    - pg_statsinfoで収集した情報をビジュアルに表示するツール

監視に使えるツール

運用管理

pg_statsinfo＋pg_stats_reporter

- **監視項目**
  - **データベースシステム**
    - データベースサイズ、リカバリコンフリクト、WAL統計
    - インスタンス処理率、インスタンスCPU使用率
  - **OS**
    - CPU使用率、ロードアベレージ、ディスクI/O、メモリ使用量
    - テーブルスペース・テーブルごとのディスク使用量
  - **SQL**
    - 更新/参照の多いテーブル、断片化したテーブル
    - 多く実行されている関数/SQL、実行時間の長いSQL
    - ロック競合
  - **運用管理**
    - チェックポイント活動、自動VACUUM活動
    - レプリケーション活動、レプリケーション遅延
  - **情報**
    - テーブル/インデックスのサイズやアクセス状況
    - パラメータ設定

ホット・スタンバイ運用

レプリケーション

株式会社メトロシステムズ OSS-DB

ホット スタン バイ

■レプリケーションとは

- **複数のサーバにデータベースのレプリカ（複製）を作成する仕組み**

■レプリケーションの目的

- **高可用性**
  - **データベースサーバを冗長化することでシステムの稼働率を高める**
    - 同じ内容のデータベースが複数存在するので、いずれかのサーバが故障しても別のサーバが処理を引き継ぐ
- **負荷分散**
  - **負荷を分散することで処理性能を向上させる**
    - 同じ内容のデータベースが複数存在するため更新、参照などの処理を分散できる

用語の整理

ホット スタン バイ

レプリケーションで用いられる用語

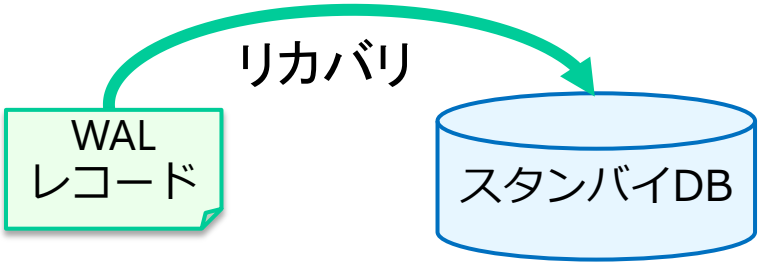
- マスタサーバ
  - レプリカの元となるメインのデータベースサーバ
- スタンバイサーバ
  - マスタサーバのレプリカを配置するデータベースサーバ
- フェイルオーバ
  - 運用中のマスタサーバに故障が発生した際、メインのデータベースをマスタからスタンバイに切り替えること
- フェイルバック
  - 修理の完了した旧マスタを再びシステムに組み込み、レプリケーション構成を再開させること

ホット
スタン
バイ

PostgreSQL本体のレプリケーション機能

- ストリーミング・レプリケーション
  - マスタサーバで生成されたWAL（トランザクションログ）をレコード単位（データベースへの操作単位）でスタンバイサーバに転送する

  - スタンバイサーバは転送されたWALを適用（リカバリ）することでデータベースをレプリカ（複製）する

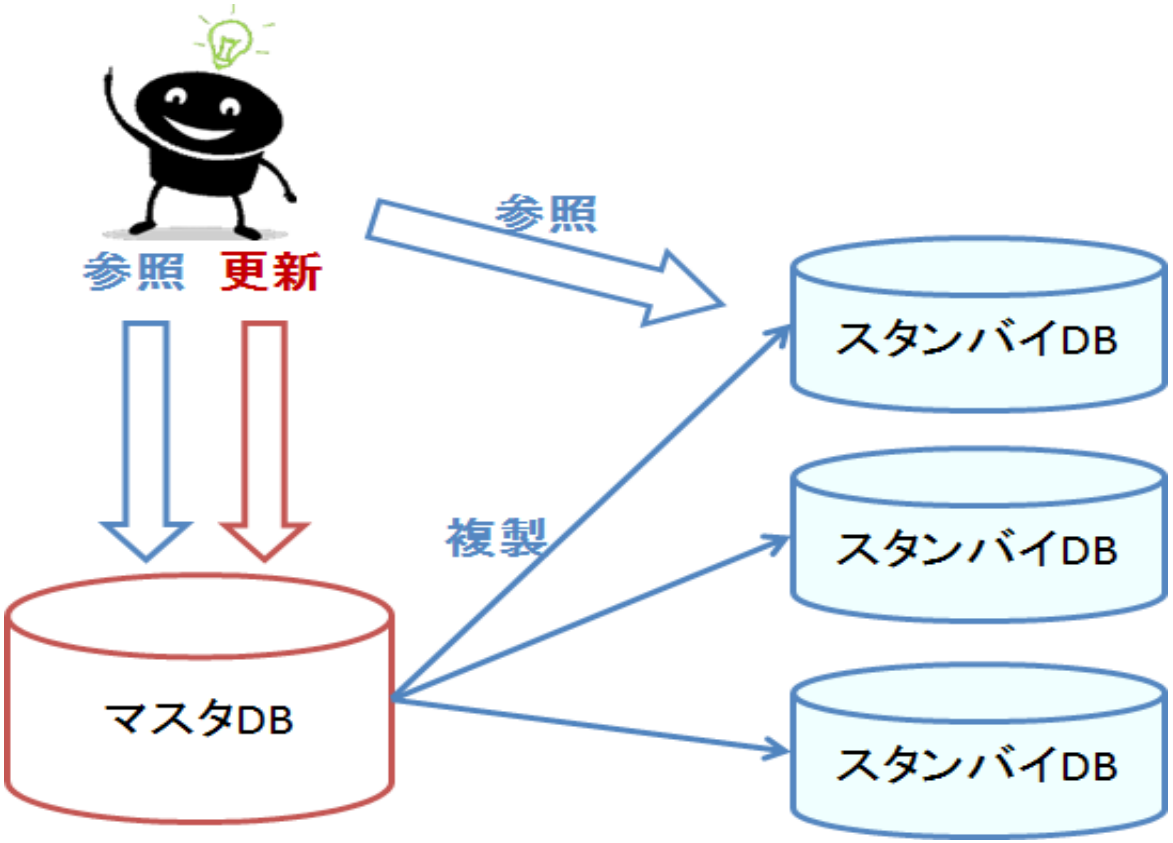


  - ホット・スタンバイ機能と組み合わせることで、スタンバイサーバを参照用サーバとして活用できる

ホット
スタン
バイ

■構成できるレプリケーションは2種類

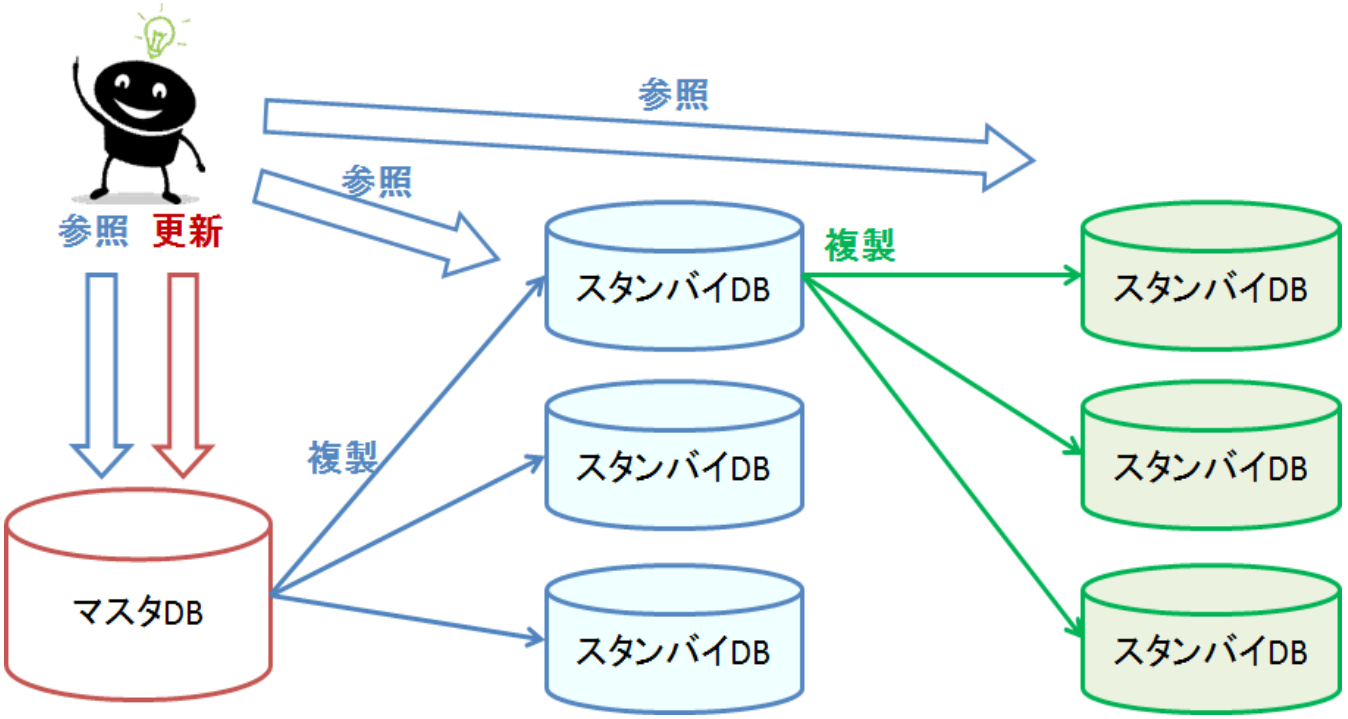
1. シングルマスタ・マルチスタンバイ構成



ホット
スタン
バイ

2. カスケード・レプリケーション構成

- PostgreSQL9.2以上
- スタンバイサーバに更にスタンバイをつなげることが可能
- カスケードする際のスタンバイサーバへの接続は「非同期モード」のみ



レプリケーションの種類

ホット スタン バイ

■レプリケーションのモードは3種類

1. 非同期モード：PostgreSQL9.1～
   - マスタサーバのディスクへのWAL書き込みを保障する
   - スタンバイサーバのディスクへのWAL書き込みを保障しない

2. 同期モード：PostgreSQL9.1～
   - マスタサーバのディスクへのWAL書き込みを保障する
   - スタンバイサーバのディスクへのWAL書き込みを保証する

3. 非同期と同期の中間モード：PostgreSQL9.2～
   - マスタサーバのディスクへのWAL書き込みを保障する
   - スタンバイサーバのWAL書き込みを保障する
     - ディスクへのWAL書き込みは保障しない

バックアップとレプリケーションの違い

ホット スタン バイ

■バックアップとは

- 不測の事態に備えてデータベースを別の記憶媒体に保存
  - ハードウェア故障
  - ソフトウェア故障
  - ヒューマンエラー
    - 操作ミスでテーブルを削除
    - アプリケーションバグによる論理破壊

■レプリケーション構成で回避できること

- ハードウェア故障
- ソフトウェア故障

■レプリケーション構成では回避できないこと

- ヒューマンエラー
  - データベースへの論理的変更はスタンバイサーバに伝搬

レプリケーション環境の構築

ホットスタンバイ

ストリーミング・レプリケーション環境を構築する手順

1. パラメータファイル（マスタ）を設定
   - postgresql.conf
   - pg_hba.conf

2. スタンバイサーバでマスタサーバのベースバックアップを取得

3. パラメータファイル（スタンバイ）を設定
   - postgresql.conf
   - recovery.conf

4. スタンバイサーバの起動

ホット
スタン
バイ

設定ファイル（マスタ）

- postgresql.conf
- pg_hba.conf

ストリーミング・レプリケーションに関連するパラメータ

- postgresql.conf
  - wal_level
  - synchronous_commit
  - synchronous_standby_names
  - vacuum_defer_cleanup_age
  - wal_sender_delay（～PostgreSQL9.1）
  - max_wal_senders
  - wal_keep_segments
  - wal_sender_timeout（PostgreSQL9.3～）
    - replication_timeout （PostgreSQL9.2）
  - max_replication_slots（PostgreSQL9.4～）

マスタサーバで設定必須のパラメータ

ホット スタン バイ

マスタ側で変更する必要のあるパラメータ

1. wal_level
   - WALの出力レベルを指定
   - “archive” または “hot_standby” を指定
2. max_wal_senders
   - WALを転送するスレーブサーバの最大数を指定
   - max_connections以上の値は指定できない
3. wal_keep_segments
   - pg_xlog配下に保持する最小WALセグメント数を指定
   - レプリケーション遅延が発生した場合、WAL転送前にマスタ側でWALが削除される可能性がある

4. synchronous_standby_names
   - 同期レプリケーション時に設定
   - カンマ区切りで複数サーバを指定可能
   - リスト内で先頭に書かれているサーバが同期レプリケーション対象

マスタサーバで設定必須のパラメータ

ホットスタンバイ

- 必要に応じて以下のパラメータも設定
  - listen_addresses
  - archive_mode
  - archive_command

ホット
スタン
バイ

設定ファイル（スタンバイ）

- postgresql.conf
- recovery.conf

ストリーミング・レプリケーションに関連するパラメータ

- postgresql.conf
  - hot_standby
  - max_standby_archive_delay
  - max_standby_streaming_delay
  - hot_standby_feedback
  - wal_receiver_timeout
  - wal_receiver_status_interval
  - synchronous_standby_names
- recovery.conf
  - restore_command
  - archive_cleanup_command
  - standby_mode
  - primary_conninfo
  - primary_slot_name
  - trigger_file
  - recovery_min_apply_delay

ホット
スタン
バイ

- **postgresql.conf**
  1. **hot_standby**
     - スタンバイで問い合わせ処理を受け付け可能にするか設定
     - “on”に設定することで負荷分散が可能
  2. **synchronous_standby_names**
     - ベースバックアップ取得の際にマスタ用の設定のままコピーされるのでスタンバイ用の設定に変更
     - スタンバイサーバのリストを削除（このパラメータを無効にする）

- **recovery.conf**
  1. **standby_mode**
     - ”on”に設定
  2. **primary_conninfo**
     - スタンバイからマスタへの接続情報を記載

レプリケーションの開始と確認方法

ホット スタンバイ

■レプリケーションの開始

- パラメータ設定後、スタンバイを起動することでレプリケーションが開始

```
standby-$ pg_ctl start -D /home/postgres/pgdata
```

■レプリケーションの状態確認

- マスタでpg_stat_replicationビューを参照

```
postgres=# SELECT * FROM pg_stat_replication;
-[ RECORD 1 ]----+------------------------------
pid              | 17696
usesysid         | 16516
usename          | rep_user
application_name | standby
client_addr      | 192.168.1.10
client_hostname  |
client_port      | 28557
backend_start    | 2016-03-15 17:14:40.426885+09
backend_xmin     |
state            | streaming
sent_location    | 0/16000278
write_location   | 0/16000278
flush_location   | 0/16000278
replay_location  | 0/16000278
sync_priority    | 1
sync_state       | sync
```

レプリケーションの遅延状態確認

ホット
スタン
バイ

WAL位置の差分情報から遅延状態を確認する

- マスタ
  - pg_current_xlog_location()　　：WALの書き出し位置を確認
- スタンバイ
  - pg_last_xlog_receive_location()　：WALの受信位置を確認
  - pg_last_xlog_replay_location()　：WALのリカバリ位置を確認

- pg_stat_replicationビュー
  - *_locationの情報から、スタンバイがマスタからどれだけWALの受け取りや再生が遅れているか確認

pg_xlog_location_diff()を用いることで、16進数の差分をバイト数として確認できる

アクセス統計情報

システム情報

統計情報

pg_statistic テーブル

- データベースに関する統計情報が保存されている
  - NULL値である列の割合や、実際の値とその分布など
- ANALYZEによって作成・更新され、実行計画の作成に使用される
- データの性質上、一般ユーザは見ることが出来ない
- stats_collectorプロセスが情報を受け取り記録している

pg_stats ビュー

- pg_statistic のビューで、一般ユーザが読み取り可能
- 最頻出値や、物理的な並び順の整列度合いなどが確認できる

定期的に取得したい統計情報

pg_stat_database

- データベース単位の統計情報を表示
  - コミット、ロールバック情報
  - 現在の接続数
  - 更新、参照件数

累積情報なので定期的に取得し、その差分情報を確認する

```
my_db=# SELECT * FROM pg_stat_database WHERE datname = 'my_db';
-[ RECORD 1 ]--+------------------------------
datid          | 16384
datname        | my_db
numbackends    | 1
xact_commit    | 156089
xact_rollback  | 30
blks_read      | 447893
blks_hit       | 2902866
tup_returned   | 35494446
tup_fetched    | 340071
tup_inserted   | 5154291
tup_updated    | 459693
tup_deleted    | 555
conflicts      | 0
temp_files     | 17
temp_bytes     | 300400640
deadlocks      | 0
blk_read_time  | 0
blk_write_time | 0
stats_reset    | 2016-03-07 12:23:57.477089+09
```

定期的に取得したい統計情報

pg_stat_user_tables、pg_stat_user_indexes

- テーブル、インデックス単位の統計情報
  - シーケンシャルスキャン、インデックススキャンの回数と件数
  - 更新件数
  - (auto)vacuum/analyze 回数と最後に実施した時間

```
my_db=# SELECT * FROM pg_stat_user_tables WHERE relname = 'pgbench_accounts';
-[ RECORD 1 ]-----+------------------------------
relid             | 32978
schemaname        | public
relname           | pgbench_accounts
seq_scan          | 6
seq_tup_read      | 6000000
idx_scan          | 47806
idx_tup_fetch     | 47806
n_tup_ins         | 1000000
n_tup_upd         | 38835
n_tup_del         | 0
n_tup_hot_upd     | 23821
n_live_tup        | 1000000
n_dead_tup        | 25398
last_vacuum       | 2016-03-08 07:52:14.509574+09
last_autovacuum   |
last_analyze      | 2016-03-08 07:52:15.1885+09
last_autoanalyze  |
vacuum_count      | 1
autovacuum_count  | 0
analyze_count     | 1
autoanalyze_count | 0
```

累積情報なので定期的に取得し、その差分情報を確認する

定期的に取得したい統計情報

pg_stat_activity

- 情報取得時点でのデータベースに接続している各クライアントの処理内容を表示
  - クエリ内容
  - 接続開始、トランザクション開始、クエリ開始時間
  - ロック待ちかどうか
  - ステータス
    - 問い合わせ中
    - コマンド待機中　　…等
- 揮発情報なので取得タイミングを意識

定期的に取得したい統計情報

- pg_stat_activity出力例

```
my_db=# SELECT * FROM pg_stat_activity;
-[ RECORD 1 ]----+--------------------------------
datid            | 16384
datname          | my_db
pid              | 32513
usesysid         | 10
usename          | postgres
application_name | psql
client_addr      |
client_hostname  |
client_port      | -1
backend_start    | 2016-03-08 09:35:17.770864+09
xact_start       | 2016-03-08 10:26:25.207591+09
query_start      | 2016-03-08 10:26:25.207591+09
state_change     | 2016-03-08 10:26:25.207616+09
waiting          | f
state            | active
query            | SELECT * FROM pg_stat_activity;
```

定期的に取得したい統計情報

その他、活用したい統計情報

- pg_stat_statements
- pg_statio_user_tables
- pg_statio_user_indexes
- pg_locks

定期的に取得したい統計情報

pg_stat_statements

- サーバで実行された全てのSQL実行回数や実行時間などの統計情報を収集するcontribモジュール
- 累積情報なので定期的に取得し、その差分情報を確認する

事前準備

- インストール
  - ソースからビルドしてインストール
    postgresql-9.2.15/contrib/pg_stat_statementsをビルド&インストール
- パッケージ管理システムからインストール
  - postgresql92-contribパッケージをインストール

GUCパラメータの設定と関数の作成

- shared_preload_librariesパラメータにpg_stat_statementsモジュールの共有ライブラリを読み込むように設定
  - 設定の反映にはデータベースの再起動が必要
- 関数とビューを作成

定期的に取得したい統計情報

pg_stat_statements GUCパラメータと関数の実行例

```
$ less /home/postgres/pgdata/postgresql.conf
…
shared_preload_libraries = 'pg_stat_statements'
…

$ pg_ctl restart -D /home/postgres/pgdata
サーバ停止処理の完了を待っています.....完了
…

$ psql -c "SHOW shared_preload_libraries" postgres
 shared_preload_libraries
--------------------------
 pg_stat_statements
(1 行)

$ psql -c "CREATE EXTENSION pg_stat_statements" postgres
CREATE EXTENSION
```

pg_stat_statements 出力例

```
postgres=# SELECT * FROM pg_stat_statements ORDER BY total_time DESC LIMIT 1;
-[ RECORD 1 ]-------+----------------------------------------------------------------
userid              | 10
dbid                | 12896
query               | UPDATE pgbench_accounts SET abalance = abalance + ? WHERE aid = ?;
calls               | 132696
total_time          | 1180248.32300001
rows                | 132696
shared_blks_hit     | 1364188
shared_blks_read    | 254148
shared_blks_dirtied | 248486
shared_blks_written | 0
local_blks_hit      | 0
local_blks_read     | 0
local_blks_dirtied  | 0
local_blks_written  | 0
temp_blks_read      | 0
temp_blks_written   | 0
blk_read_time       | 0
blk_write_time      | 0

postgres=#
```

定期的に取得したい統計情報

統計情報

- pg_statio_user_tables、pg_statio_user_indexes
  - 各テーブルへ実施されたブロックアクセス状況を表示
    - テーブル、インデックス、TOASTについて以下の情報を提供
      - ブロックの読み込み数
      - キャッシュヒットした読み込み数
  - 累積情報なので定期的に取得し、その差分情報を確認する
  - この情報からキャッシュヒット率を確認できる

キャッシュヒット率算出例

```
postgres=# SELECT relname,
postgres-# round(heap_blks_hit*100/(heap_blks_hit+heap_blks_read), 2) AS cache_hit_ratio
postgres-# FROM pg_statio_user_tables
postgres-# WHERE heap_blks_read > 0 ORDER BY cache_hit_ratio;
     relname      | cache_hit_ratio
------------------+-----------------
 pgbench_accounts |           67.00
 pgbench_history  |           98.00
 pgbench_tellers  |           99.00
 pgbench_branches |           99.00
(4 行)
```

定期的に取得したい統計情報

pg_locks

- ロックの状態を一覧で表示
  - ロック種別
  - ロック対象のオブジェクト、テーブル情報　　…等
- pg_stat_activityと組み合わせるとロックを保持している処理などが把握できる
- 累積情報なので定期的に取得し、その差分情報を確認する

統計情報注意点

- クラッシュリカバリした後は統計情報コレクタを修復する
  - PostgreSQLはクラッシュした場合、統計情報コレクタをクリア
  - autovacuum処理のトリガにもなる情報なので早い段階で修復
    - 手動でanalyzeを実施
    - 有効行数pg_stat_all_tables.n_live_tupと不要行数pg_stat_all_tables.n_dead_tupが最新化される

クエリ実行計画

クエリチューニングの考え方

クエリチューニングの流れ

- 現状のままクエリを実行し実行計画を取得
- 特に時間のかかっている処理を特定
- 遅い箇所を改善するアイデア
  - 同等の他の処理に置き換えられないか検討
  - 同等の別のSQLに書き換えられないか検討
- 様々なパターンで試行

クエリチューニングのメリット・デメリット

- メリット
  - 特定のクエリを狙い撃ちで性能改善できる
  - 状況によっては数百倍程度に性能が改善されることもある
- デメリット
  - アプリケーションの改修が必要になることが多く、テストなどのコストがかかる
  - 実際の環境でないと試行が難しい

■プラン（実行計画）の取得

- クエリチューニングの基本は、EXPLAINコマンドでのプランの取得

実行計画の例

```
pgbench=> EXPLAIN SELECT * FROM pgbench_branches b
pgbench-> JOIN pgbench_accounts a ON a.bid = b.bid;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=1.11..20073.11 rows=500000 width=461)
   Hash Cond: (a.bid = b.bid)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..13197.00 rows=500000 width=97)
   ->  Hash  (cost=1.05..1.05 rows=5 width=364)
         ->  Seq Scan on pgbench_branches b  (cost=0.00..1.05 rows=5 width=364)
(5 rows)
```

EXPLAIN結果の読み方

```
pgbench=> EXPLAIN SELECT * FROM pgbench_branches b
pgbench-> JOIN pgbench_accounts a ON a.bid = b.bid;
                                  QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=1.11..20073.11 rows=500000 width=461)
   Hash Cond: (a.bid = b.bid)
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..13197.00 rows=500000 width=97)
   ->  Hash  (cost=1.05..1.05 rows=5 width=364)
         ->  Seq Scan on pgbench_branches b  (cost=0.00..1.05 rows=5 width=364)
(5 rows)
```

- EXPLAINの出力内容
- プランノード名
  - 処理の種類を表す名前
- 処理対象リレーション名
  - スキャン対象リレーション（テーブルやインデックス）の名前
- 見積情報
  - ANALYZEで取得した統計情報から見積もった情報

■プランノードの例

- スキャン
  - **Seq Scan：先頭ブロックから順にヒープ全体をスキャン**
    - 大きいテーブルでは致命的にディスクI/Oが出る
  - **Index Scan：インデックスに基づいてヒープをスキャン**
    - ランダムアクセスになる
    - 大量にヒットする場合は遅くなる
  - **Index Only Scan：インデックスのみを用いてスキャン**
    - こまめにVACUUMをしないと選ばれない
  - **Bitmap Index Scan：複数のインデックスから作成したビットマップに基づいてヒープをスキャン**
    - ビットマップがメモリに収まれば高速

実行計画

プランノードの例

- **結合**
  - **Nest Loop：Outer1行につきInner全体をスキャンし結合**
    - Inner側の件数が多いと極端に遅い
  - **Merge Join：結合キーでソートされた結果同士を結合**
    - 事前にソートする必要があるが、大きい結果通しでもある程度高速
  - **Hash Join：Innerの結合キーでハッシュテーブルを作成し、それに基づいてOuterをスキャン**
    - ハッシュテーブルがメモリに収まればかなり高速
- **その他**
  - **Sort：結果を並べ替え**
    - 結果が多いと一時ファイルを使い始める
    - 大量データのソートが必要なケースではwork_memを上げる
  - **Limit：結果の行方向の部分集合を取得**
    - 最終的に必要な件数が少なくて済む場合があり結果セットごとに性能が変わりやすい
  - **Materialize：結果セットを一時領域に保存**
    - 結果セットがwork_mem以上なら一時ファイルに書き出す

EXPLAINオプション

- ANALYZE（Boolean）
  - Trueであれば実際にクエリを実行して各ノードの処理時間を計測
- VERBOSE（Boolean）
  - Trueであれば各種追加情報を出力
- BUFFERS（Boolean）
  - Trueであれば共有バッファの使用状況を出力
- FORMAT（TEXT、XML、JSON、YAMLのいずれか）
  - EXPLAIN結果の出力フォーマットを指定
  - デフォルトはTEXT形式

実行計画

EXPLAIN ANALYZE結果の読み方

```
pgbench=> EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM pgbench_branches b
JOIN pgbench_accounts a ON a.bid = b.bid;
                QUERY PLAN
-------------------------------------------------------------------------------------------
 Hash Join  (cost=1.11..20073.11 rows=500000 width=461) (actual time=0.032..609.519
rows=500000 loops=1)
   Hash Cond: (a.bid = b.bid)
   Buffers: shared hit=2273 read=5925
   ->  Seq Scan on pgbench_accounts a  (cost=0.00..13197.00 rows=500000 width=97)
(actual time=0.005..173.159 rows=500000 loops=1)
         Buffers: shared hit=2272 read=5925
   ->  Hash  (cost=1.05..1.05 rows=5 width=364) (actual time=0.009..0.009 rows=5
loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 1kB
         Buffers: shared hit=1
         ->  Seq Scan on pgbench_branches b  (cost=0.00..1.05 rows=5 width=364)
(actual time=0.001..0.001 rows=5 loops=1)
               Buffers: shared hit=1
 Total runtime: 710.760 ms
(11 rows)
```

時間情報
- time ：所要時間（スタートアップ..全体）
- rows ：実際のヒット件数
- loops：繰り返し回数

EXPLAINの注意点

■見積り件数

- 見積り件数と実際の件数に乖離がある場合は、最適でない実行計画が選ばれがち

■統計情報は最新に

- 定期的にANALYZEを実行して統計情報を更新
- 自動VACUUMで十分か確認する
- 大量更新バッチの後は手動ANALYZEをする

■動作中のアプリの実行計画はauto_explainで確認

- サーバーログに実行されたSQLの実行計画を出力する
  - 使用方法は「運用管理」の項を参照
- 負荷が高くなるので、経過時間設定を on/off の切り替えなどで出力対象を限定的にする

EXPLAINの注意点

■見積行数と実際の行数の乖離

- プランナーは見積行数に基づいてプランを作成
  - 実際の行数と乖離があると不適切なプランになることが多い
  - 乖離している場合はANALYZEを実行して統計情報を更新
- 検索条件が複数ある場合に見積が過少になりやすい
  - 列同士の相関関係は考慮しない

■ソート処理とマテリアライズ

- sortノードの「Sort Method」項目が「external sort」になった場合はメモリ上でソートが完結していない
  - 「Disk」項目の値に応じてwork_memを増やすことを検討
  - work_memはプランノードごとの制限値なので増やしすぎないように注意

EXPLAINの注意点

共有バッファヒット率

- あるノードで表示されるBuffers情報はそのサブツリーの合計値
- 下位ノードの合計と一致していればそのノードでは共有バッファは参照していない

EXPLAIN ANALYZEでのloops

- loopsは、そのノード全体が繰り返された回数なので、繰り返し回数が増える場合はボトルネックになりやすい
- Nest LoopのInner側などで1よりも大きくなる

実行計画の変え方

enable_*系GUCパラメータを変更

- プランノードの使用可否を制御するパラメータを変更する
  - enable_seqscan、enable_indexscanなど
  - enable_nestloop、enable_mergejoinなど
  - enable_material、enable_sortなど
- SET文でセッション中に変更可能
  - SQL文ごとに切り替えることも可能
    ※ ただし、SET文のオーバーヘッドがある
- ユーザ別GUCパラメータを使用すると、接続時に設定が切り替わる

コストファクタ系GUCパラメータを変更

- コスト値算出に使われるパラメータを変更する
  - seq_page_cost、random_page_cost
  - cpu_tuple_cost、cpu_index_tuple_cost
  - cpu_operator_cost
  - effective_cache_size

実行計画の変え方

クエリを書き換える

- クエリの構成を書き換える
  - クエリの一部をCTEに分離
  - EXISTSやNOT INに書き換え … 等
- クエリの結果が変わらないように注意！

便利な外部ツール

実行計画

pg_dbms_stats

- PostgreSQLの持つ統計情報を管理
  - 統計情報を固定して実行計画の変化を抑止
  - 本番環境の統計情報を検証環境に移植してチューニング
    https://osdn.jp/projects/pgdbmsstats/scm/git/pg_dbms_stats/

pg_hint_plan

- PostgreSQLでヒント句を利用可能にする
  https://osdn.jp/projects/pghintplan/scm/git/pg_hint_plan/

pgAdmin-Ⅲ

- 実行計画をグラフィカルに表示
- クエリ書き換えのトライ&エラー
  http://www.pgadmin.org/

ご清聴ありがとうございました。