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

2015 年度 WG3 活動報告書

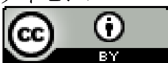
データベースツール編

製作者
担当企業名:
NTT ソフトウェア株式会社
大日本印刷株式会社
TIS 株式会社
株式会社日立ソリューションズ

© 2016 PostgreSQL Enterprise Consortium

改訂履歴

版	改訂日	変更内容
1.0	2016/04/21	新規作成

ライセンス

本作品はCC-BYライセンスによって許諾されています。
ライセンスの内容を知りたい方はhttp://creativecommons.org/licenses/by/2.1/jp/でご確認ください。
文書の内容、表記に関する誤り、ご要望、感想等につきましては、PGEConsのサイトを通じてお寄せいただきますようお願いいたします。
サイトURL https://www.pgecons.org/contact/

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

© 2016 PostgreSQL Enterprise Consortium

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

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

PostgreSQL エンタープライズコンソーシアム 技術部会では PostgreSQL の普及に対する課題を活動テーマとし 3 つのワーキンググループで具体的な活動を行っています。

・WG1（性能ワーキンググループ）
・WG2（移行ワーキンググループ）
・WG3（設計運用ワーキンググループ）

WG3 では PostgreSQL の設計運用に対する課題に対し調査検証を行い、PostgreSQL が広く活用される事を推進しています。

■本資料の概要と目的

本資料は WG3 の 2015 年度の活動として PostgreSQL をエンタープライズ領域で利用するための設計運用に必要となる機能や課題に対し、有効な周辺ツールを整理し一部のツールについて動作確認を行ったものです。

■本資料の構成

1章. はじめに
2章. PostgreSQL の抱える運用面での課題
　PostgreSQL を利用したシステムで抱える設計運用面の課題について記載しています。
3章. PostgreSQL を取り巻くツール群
　PostgreSQL を利用したシステムで抱える設計運用面の課題を解決するための調査したツール群について整理しています。
4章～7章. ツール調査検証
　着目したツール対する調査、検証を行い、設計運用の観点からの考察結果を記載しています。
8章. おわりに

■想定読者

本資料の読者は以下のような知識を有していることを想定しています。

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

© 2016 PostgreSQL Enterprise Consortium

目次

1.はじめに......5
2.PostgreSQL の抱える設計運用面での課題......5
3.PostgreSQL を取り巻くツール群......6
4.性能監視ツール......7
4.1.性能監視ツールに求められる要件......7
4.2.ツール紹介......8
4.3.まとめ......10
5.バックアップツール......17
5.1.バックアップ運用に求められる要件......17
5.2.PostgreSQL のバックアップ機能......17
5.3.pg_basebackup......20
5.4.pg_rman......22
5.5.Barman......27
5.6.OmniPITR......29
5.7.まとめ......32
6.パーティションツール......34
6.1.パーティショニングに求められる要件......34
6.2.パーティショニングツールの紹介......35
6.3.まとめ......47
7.実行計画制御......48
7.1.PostgreSQL の実行計画......49
7.2.ツール紹介......52
7.3.まとめ......56
8.まとめ......57

© 2016 PostgreSQL Enterprise Consortium

1. はじめに

昨今、情報システムは社会活動、企業活動のために不可欠なものとなっています。このような情報システムの社会基盤化に伴い、情報システムの構成要素すなわち適用される技術・製品は、オープン化、ネットワーク化により大規模かつ複雑なものとなっています。このため、情報システムの実現のためには、単に業務をIT化するだけでなく、複雑な構成要素を適切に連携させ、安定的にサービスを提供することが重要となっています。

情報システムはさまざまな業務機能を実現する業務アプリケーションとそれを支えるためのシステム基盤等で構成されます。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、ハードウェア機器やネットワーク機器、OSやミドルウェア、更にはその制御や運用のアプリケーションなどの組み合せで実現されます。これらシステム基盤に対する要求事項を業務機能と区別して「非機能要求」と整理されています。

「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ています。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さまざまなリスクが生じているのが実態です。

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(表 1.1)。

表 1.1: システム基盤の非機能要求

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

2. PostgreSQLの抱える設計運用面での課題

PostgreSQLはオープンソースのRDBMSとして広く利用されるようになっています。情報システムへのPostgreSQLの利用拡大に伴い、情報システム上求められるさまざまな非機能要求に対応していく必要があります。例えばエンタープライズ領域でデータベースを運用する際には、システムのサービスレベルを一定に保つために定常監視や障害時の対応フローなどの運用が必要となってきます。このようなさまざまな非機能要求に対してはPostgreSQLのデフォルト機能では対応できないことがあります。PostgreSQLでは周辺ツールに対するコミュニティの開発も活発に行われており、周辺ツールを活用することでPostgreSQLのデフォルト機能では対応できない非機能要求を満たすことができます。 ただ、PostgreSQLの周辺ツールは英語圏で開発されているものも多く、利用するためのノウハウや情報が少ないという課題がありました。

今年度のWG3では情報システムの非機能要求として主に性能・拡張性や運用・保守性の項目においてPostgreSQLを設計運用する際に必要となる機能や課題に対して有効な周辺ツールを調査し、まとめています。

1 独立行政法人情報処理推進機構
非機能要求グレード http://www.ipa.go.jp/sec/softwareengineering/reports/20100416.html

© 2016 PostgreSQL Enterprise Consortium

3. PostgreSQL を取り巻くツール群

PostgreSQL を利用する情報システムに利用されている代表的な周辺ツールを分類化し表 3.1 に示します。周辺ツールの選定基準は、コミュニティメンバの認知度が高いものを調査対象としました。

表 3.1: 周辺ツール

カテゴリ	ツール名	ツール概要
バックアップ	pg_basebackup	PostgreSQL のベースバックアップを取得する。
	Barman for PostgreSQL	・複数の PostgreSQL サーバのバックアップやリカバリをバックアップサーバで一元管理する。 ・リモートの PostgreSQL サーバから SSH 経由でバックアップが可能
	OmniPITR	・マスタ／スレーブサーバから PITR バックアップ、リカバリ/レプリカの作成、監視
	pg_rman	・バックアップの世代管理、差分バックアップが可能/PostgreSQL サーバと同一のサーバにしかバックアップできない。
運用監視	pg_stat_statements	・SQL の実行回数、総実行時間を蓄積する。情報を専用のビューを介して SQL で取得可能
	pg_statsinfo	・統計情報をスナップショットとして定期的に収集・蓄積する。 ・サーバログの加工 ・監視対象インスタンスの状態監視、アラート機能
	pg_stats_reporter	・pg_statsinfo で取得、蓄積した情報を可視化するレポートを作成する。
	pgBadger	・サーバログを分析したレポートを HTML ファイルで作成する。
	pg_monz（Zabbix）	・Zabbix で PostgreSQL を監視するテンプレート、追加スクリプト群を提供する。
	Hinemos PostgreSQL 性能監視	・Hinemos で PostgreSQL の性能情報の収集・監視を実現する/（Hinemos 4.0 のカスタム監視機能に対するアドオン）
	PgPerf	統計情報をスナップショットとして取得し、専用のスナップショットテーブルに保存する。
	pg_top	PostgreSQL 用の top コマンド
性能維持	pg_dbms_stats	統計情報の管理を行い、間接的に実行計画を制御する。
	pg_hint_plan	ヒント句をクエリに指定して、SQL 文や GUC パラメータを変えずに実行計画を制御する
	pg_prewarm	OS のバッファまたは PostgreSQL のバッファにリレーションデータをロードする
	pg_buffercache	共有バッファ上のブロックの所属(DB,table,block)と状態を確認するビューを提供する
	PgFincore	OS のディスクキャッシュに乗ったテーブルとインデックスのページを管理する関数
	pg_reorg	PostgreSQL のテーブルを再編成するシェルコマンド
	pg_bloat_check	テーブルやインデックスの肥大率、ページサイズ、無駄な領域を表示するレポート
パーティショニング	pg_part	パーティションの作成（＋データの移行）/解除（＋データの移行）/追加/切り離しを簡単に行う関数群を提供する。
	pg_partman	・時間ベースとシリアルベースでテーブルをパーティショニングセットを管理する ・バックグラウンドワーカーでパーティションのメンテナンスを自動化
	pg_shard	・シャーディング拡張機能を提供。水平分散（データを複数のノードに分散）と可用性（同じデータを複数のノードでコピー）を提供する
コネクションプール	pgpool-II	・コネクションプーリング（L7）/SQL の負荷分散
	PgBouncer	・コネクションプーリング（L4）

※PostgreSQL エンタープライズコンソーシアムは表 3.1 に示したツール利用を推奨しているわけではありません。情報システムに対する非機能要求に応じてツール利用を検討ください

それぞれのカテゴリのツールに求められる機能は以下のようなものが挙げられます。

- バックアップ
  バックアップの分野では、可用性ではデータをどこまで保全するかという観点で、運用ではデータをどこまで復旧させるかという観点で機能が必要となります。
- 運用監視
  監視の分野では情報収集を行った結果に応じて適切な宛先に発信する機能が必要となります。どのような情報を発信する必要があるかはシステムの非機能要求によって異なり、エンタープライズ領域ではパフォーマンス監視を行う機能が必要となります。
- 性能維持
  性能要件では具体的な目標値や約束値がある場合、各処理の順守率を規定します。データベースではデータの増減に伴い性能劣化した場合、性能要件を順守するためにチューニングを施す機能が必要となります。
- パーティション
  データベースにおけるパーティショニング機能ではデータを複数に分割して格納することで性能や運用性を向上することができます。反面、パーティショニングの管理を行う機能が必要となってきます。

コネクションプールの機能については情報システムへの適用実績は多数確認できコミュニティメンバの認知度も高いツールが含まれていましたが、本年度は調査対象外としました。

© 2016 PostgreSQL Enterprise Consortium

4. 性能監視ツール

4.1. 性能監視ツールに求められる要件

システムにおいて、性能劣化等の発生は企業ビジネスにおいて致命的な問題になりうるものです。それは、PostgreSQL を使用しているシステムにとっても例外ではなく当てはまります。これらの発生を予防するためには監視が必要となります。監視には障害検知、予防保守、キャパシティ管理等の目的が存在し、ICT サービスを安定的に運用するためにはこれらの目的に対する監視処理が必要になります。

予防保守のひとつである性能監視では、性能情報を取得するだけでは意味がなく取得した情報を分析することにより始めて性能監視を実現できます。また、性能監視はシステムの運用が始まった時点で導入されている必要があります。性能劣化が発生してからでは手遅れとなるためです。性能が劣化した原因の調査およびチューニングするにはシステム稼動時から定期的にデータを取得する必要があります。

PostgreSQL には性能監視ツールがいくつか用意されています。しかし、導入するものの何の情報が取得でき、さらに取得した情報の分析方法といったノウハウが少なく、使いづらいといった声も少なくありませんでした。そこで本章では PostgreSQL の代表的な性能監視ツールである、pg_statsinfo、pg_stats_reporter、pg_monz の使い方について紹介します。

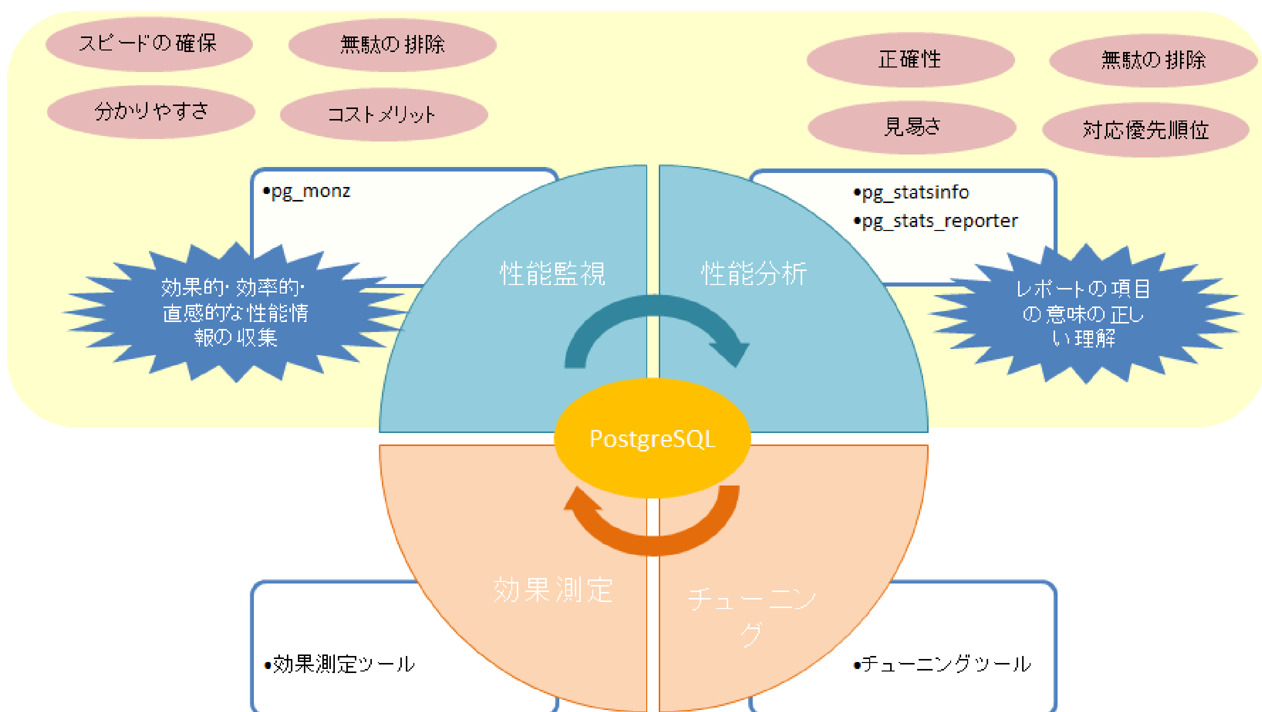


図 4.1: 性能改善のサイクルと周辺ツールの関係

4.1.1. 性能監視の課題

PostgreSQL には、サーバの活動状況に関する情報を収集し統計情報ビューを参照することができます。しかし、定期的に統計情報を取得するツールが標準で用意されていないため、外部ツール用いて実現する必要があります。定期的にデータを取得することができない場合、性能劣化がどのタイミングで発生したか等の原因特定作業が非常に困難なものになります。また取得した情報を分析するには専門的な知識を必要とします。このことから、性能監視に求められる課題は以下のものが挙げられます。

・定期的に統計情報を取得できる。
・過去の統計情報と比較できる。(時系列で確認できる)
・性能分析を容易に実施できる。

© 2016 PostgreSQL Enterprise Consortium

4.1.2. 性能監視ツールの比較

（1）用途比較

本章で取り上げるツールについてですが、ところどころ重複している機能が存在しツールの使い分けが難しい場合があります。そこで、各ツールの主な用途と役割について記載・比較します。各用途に応じて必要なツールを使い分けることにより、無駄なく効果的な性能分析・障害検知を実施できます。

表 4.1: 各ツールの概要と役割

ツール名	用途	役割
pg_statsinfo	性能分析	PostgreSQL の統計情報を定期的に取得し、問題特定・性能分析作業を補助する。
pg_stats_reporter	性能分析	pg_statsinfo の情報をレポートする。 pg_statsinfo で閲覧できるレポートよりも可読性をよくする。
pg_monz	性能監視	Zabbix で PostgreSQL の監視を行うためのテンプレートで死活監視、リソース監視、ストリーミングレプリケーションの冗長構成のステータス、pg-pool Ⅱ の状態監視を実施できる。

（2）リリース情報比較

以下で各ツールのライセンス等の一般的な情報を一覧として記載します。

表 4.2: 公開状況(2016/3/10 調査時点)

ツール名	pg_statsinfo	pg_stats_reporter	pg_monz
最新バージョン	3.0.2	3.0.1	2.0
ツール概要	PostgreSQL や OS のリソース情報、統計情報をスナップショットとして取得するためのツール	pg_stats_info が収集した統計情報を元に、PostgreSQL サーバの利用統計情報を HTML 形式のグラフィカルなレポートで出力するツール	Zabbix で PostgreSQL の各種監視を行うためのテンプレート
ツール種別	エージェント （shared_preload_libraries） コマンドラインツール	Web アプリケーション	スクリプト 設定ファイル
実装方式	C	PHP、java script、SQL	XML、シェルスクリプト
対応バージョン(OS, DB などの要件)	RHEL 5.x, RHEL 6.x, CentOS 5.x, CentOS 6.x	RHEL5.9 RHEL 6.4	Zabbix Server,Zabbix Agent, Zabbix Sender：2.0 以上 PostgreSQL：9.2 以上 pgpool-II：3.4.0 以上
ライセンス形態	BSD	BSD	Apache License Version 2.0
開発元	NTT OSS センタ	NTT OSS センタ	SRA OSS、TIS 株式会社
入手元(URL)	http://pgstatsinfo.sourceforge.net/	http://pgstatsinfo.sourceforge.net/	http://pg-monz.github.io/pg_monz/
有償サポート	有	有	有

4.2. ツール紹介

4.2.1. pg_statsinfo の紹介

(1) 概要

PostgreSQL サーバの利用統計情報を定期的に収集・蓄積することで、DB 設計や PostgreSQL の運用に役立つツールです。システム運用時の性能劣化や問題発生時の原因特定を実施するために有用なツールです。

(2) 取得可能データ

表 4.3 にて pg_statsinfo で取得可能な情報をまとめています。データの取得元等については別途付録に記載させていただきましたので、そちらをご参照ください。

© 2016 PostgreSQL Enterprise Consortium

表 4.3: pg_statsinfo で取得可能なデータの例

機能大項目	機能中項目	機能項目
DB 全体	データベース	容量、キャッシュヒット等
	アーカイブ	アーカイブ取得数、失敗数
	autovacuum	回収ページ数、タプル(行)数、実行時間
	autoanalyze	実行時間
	SQL	遅い query
	lock	デッドロック数、ロック取得クエリ等
	レプリケーション	遅延
	WAL	書き込み量
	トランザクション	トランザクション数、コミット数、ロールバック数、トランザクション時間
	チェックポイント	開始日時、処理時間、書き込み量
DB オブジェクト	テーブル	行数、容量、read 回数、キャッシュヒット、TOAST キャッシュヒット、insert 行数、update 行数、HOT update 行数、delete 行数、最終 vacuum 日時、最終 analyze 日時、無効(vacuum 対象)行数、
	インデックス	容量、index scan 実行回数、キャッシュヒット
	関数	実行回数、総処理時間
	テーブルスペース	使用率
OS リソース	CPU	system 利用、user 利用、idle 利用、iowait 利用
	メモリ	free 量、buffers 量、cached 量、swap 量、dirty 量
	IO	read 容量、read 時間、write 容量、write 時間
	ロードアベレージ	1 分、5 分、15 分平均値

4.2.2. pg_stats_reporter の紹介

(1) 概要

pg_statsinfo で収集した統計情報を元に、PostgreSQL サーバの利用統計情報レポートを HTML のグラフィカルな形式で作成するツールです。レポートの可読性を高めたい場合には有用なツールです。

(2) 取得可能データ

表 4.3 にて pg_stats_reporter で閲覧可能なデータ、及びデータの取得元等については別途付録に記載させていただきましたので、そちらをご参照ください。

4.2.3. pg_monz の紹介

(1) 概要

pg_monz（PostgreSQL monitoring template for Zabbix）は、Zabbix で PostgreSQL の各種監視を行うためのツールです。このツールは Zabbix のテンプレートという形式で実装されています。

このテンプレートを利用することにより、PostgreSQL の死活監視、リソース監視、性能監視などが行えます。また、PostgreSQL 単体で稼働するシングル構成の状態、PostgreSQL のストリーミングレプリケーションを使った冗長構成の状態、pgpool-II を使った負荷分散構成の状態の監視を行うことができ、PostgreSQL を運用する様々な環境の監視を行うことができます。

(2) 取得可能データ

pg_monz で取得可能なデータを以下の表にまとめます。pg_statsinfo で取得できる情報との比較は別途付録に記載させていただきましたので、そちらをご参照ください。

© 2016 PostgreSQL Enterprise Consortium

表 4.4: pg_monz で取得可能なデータの例

機能大項目	機能中項目	機能項目
DB 全体	データベース	容量、キャッシュヒット、read(seq)行数、read(index)行数、insert 行数、update 行数、delete 行数
	lock	デッドロック数
	レプリケーション	遅延
	トランザクション	コミット数、ロールバック数
	チェックポイント	書き込み量
DB オブジェクト	テーブル	行数、容量、read(seq)回数、read(index)回数、キャッシュヒット(seq)、キャッシュヒット(index)、insert 行数、update 行数、HOT update 行数、delete 行数、無効(vacuum 対象)行数

4.3. まとめ

4.3.1. pg_statsinfo および pg_stats_reporter の使い時

pg_statsinfo 及び pg_stats_reporter の使い時は、PostgreSQL の性能に関して『詳細な分析をしたいとき』です。これは特に、データ量が膨大であったり、複雑なデータ構造が必要であったり、性能に関する問題が生じやすいシステムにあてはまります。

以下でより詳細な使い時を挙げます。

(1) 詳細な分析をしたいとき

pg_statsinfo は極めて多岐にわたる性能情報を取得します。pg_monz も十分な種類の性能情報を取得しますが、それと比較しても、pg_statsinfo のみ取得している性能情報がかなりあります(詳細は付録をご参照ください)。情報の幅は分析の幅に繋がります。詳細な分析をしたいとき、この幅が大きな武器になるはずです。以下に pg_statsinfo の情報の幅が有効になる例を挙げます。

≪SQL 文単位の性能分析≫

pg_statsinfo は pg_stat_statements(SQL 文の統計情報を取得するモジュール)と連携して SQL 文単位の分析が可能です。pg_stat_statements 単体では他の統計情報と同様に、SQL 文の実行時間や実行回数などの累積値のみが記録されます。この累積値を pg_statsinfo 等で定期に記録することで、例えば以下のような分析が可能になります。

SQL 文単位の性能分析実施例

例) ○月×日 △時□分～▲時■分に、性能上の問題が発生していたので、以下を確認したい ・1回あたりの実行時間が長かった SQL 文のランキング ・実行回数の多かった SQL 文のランキング ・時間のかかった、実行計画単位のランキング(※) (※) pg_stat_statements 以外にも、外部モジュール(pg_store_plans)が必要

また、システム運用について、「アプリケーション担当」と「インフラ担当」を分ける現場も少なくないと思います。そのような体制のとき、DB の性能問題に対して SQL 文が両担当の共通言語になります。両担当の協力関係という観点でも、SQL 文単位の性能分析は重要です。

≪インデックス単位の性能分析≫

pg_statsinfo は全てのインデックスについて、インデックス単位の統計情報を記録します。性能劣化の原因がインデックスにあることが分かった場合、必ずどのインデックスに対処をするか決める必要があります。pg_statsinfo を使うと、「どのインデックス」というスコープまで絞って分析可能です。具体的には以下のような分析が可能になります。

© 2016 PostgreSQL Enterprise Consortium

インデックス単位の性能分析実施例

例) ○月×日 △時□分以降で DB の使い方が変化したので、
　インデックスの見直しを目的に、以下に関して変化以後のインデックスの状況を確認したい

・インデックスごとの容量の増加量
・インデックスごとの読み込み量
・インデックスごとのキャッシュヒット率

※ pgstattuple 等との連携機能はないので、インデックスのタプルレベルの分析はできないので、注意

ここまで、pg_statsinfo が取得している情報の幅を活用した分析例を挙げました。分析の幅だけでなく、レポートの出力方法についても、詳細な分析を助けるポイントがあります。

性能に関する問題が発生した時は、問題発生期間についてしばしば複数の要素を横断で確認する必要があります。一方で pg_statsinfo の簡易レポートや pg_stats_reporter によるレポート機能は、期間を指定して DB の各項目横断でレポートしてくれる機能です。そのため、このレポートの形式は性能問題発生時の詳細な分析に非常にマッチしたものと言えます。

また、性能問題発生時以外にも、定期的に DB の状況をレビューするような業務フローを設けている運用の現場もあると思います。このような定期のレビュー業務でもレポート機能は効力を発揮するはずです。

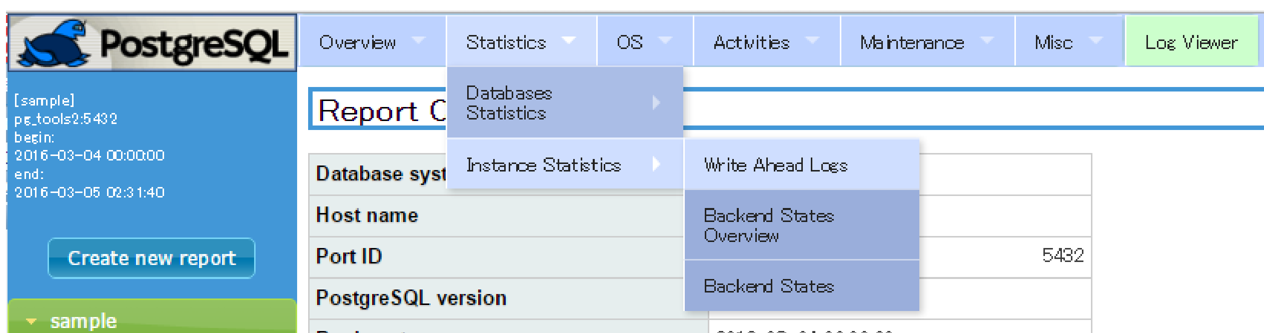


図 4.2: 項目横断でレポートできる pg_stats_reporter

pg_statsinfo の簡易レポートや pg_stats_reporter は上述のようによく纏まっています。このレポートを使えば、殆どのケースで十分な分析を実施可能でしょう。しかしこれは、pg_statsinfo が取得している性能情報全てをカバーできているわけではありません。レポートではカバーできていない分析を実施したい場合、直接リポジトリデータベースに SQL 文を実行する必要があります。

直接 SQL 文を実行する分析はレポートによる分析と比べると、かなり敷居が高いです。リポジトリデータベースのデータ構造を把握する必要がありますし、自分が使っている pg_statsinfo のバージョンに合ったデータ構造に関するドキュメントが見つからず、結果手さぐりで構造を確認する必要があるかもしれません。ただ、pg_statsinfo は PostgreSQL 専用の分析ツールとして設計されているため、統計情報ビュー(pg_stat*)に近い構造になっています。普段からこれらを見ている DBA ならば、比較的構造の把握はしやすいはずです。

以下に SQL 文による分析例を挙げます。

© 2016 PostgreSQL Enterprise Consortium

pg_statsinfo リポジトリへの SQL 文による分析例

例) customer テーブルのキャッシュヒット率の時間による遷移を知りたいが、
レポートでは機能提供していないので、SQL 文実行で代替する

```
statsinfo=# select
statsinfo-#     s.time,
statsinfo-#     statsrepo.div(
statsinfo(#         statsrepo.sub(
statsinfo(#             t.heap_blks_hit,
statsinfo(#             lag(t.heap_blks_hit, 1) over (order by t.snapid asc)
statsinfo(#             ),
statsinfo(#         statsrepo.sub(
statsinfo(#             t.heap_blks_read,
statsinfo(#             lag(t.heap_blks_read, 1) over (order by t.snapid asc)
statsinfo(#             ) +
statsinfo(#         statsrepo.sub(
statsinfo(#             t.heap_blks_hit,
statsinfo(#             lag(t.heap_blks_hit, 1) over (order by t.snapid asc)
statsinfo(#             )
statsinfo(#         ) "キャッシュヒット率(seq)"
statsinfo-# from
statsinfo-#     statsrepo.tables t LEFT JOIN statsrepo.snapshot s
statsinfo-#         on t.snapid = s.snapid
statsinfo-# where
statsinfo-#     t."table" = 'customer' and
statsinfo-#     t.database = 'tpcc1' and
statsinfo-#     t.schema = 'public'  and
statsinfo-#     s.time between '2015-09-14 00:00:00'::timestamp and '2015-09-14
19:00:00'::timestamp
statsinfo-# ;

          time           | キャッシュヒット率(seq)
-------------------------------+-------------------------
 2015-09-14 00:00:00.011973+09 |           0.999
 2015-09-14 00:30:00.016698+09 |           1.000
 2015-09-14 01:00:00.145519+09 |           1.000
 2015-09-14 01:30:00.150055+09 |           1.000
・・・
```

(2) 監視ツールを自由に選択したいとき

pg_statsinfo は性能分析ツールであり、監視に関しては関与しません。監視が不要なシステムは少ないので、実際には pg_statsinfo の環境とは別に監視ツールの導入が必要で、その分手間がかかります。しかし、裏を返すと監視ツールを自由に選択できるとも言えます。運用を担当する組織内で Zabbix 以外に使い慣れた監視ツールがあるならば、無理に pg_monz を導入して Zabbix の扱いに四苦八苦するより寧ろ運用は楽になるかもしれません。

(3) ログ出力を細かくコントロールしたいとき

性能監視そのものの機能ではありませんが、pg_statsinfo にはログ管理機能があります。この機能を使うと、エラーの種類に応じてログを分配したり、syslog に渡すタイミングでレベルを変更したりと、柔軟なログ運用が可能になります。

PostgreSQL のログは様々な情報が混ざっています。その分閲覧や監視にフィルタが頻繁に必要となるので、それほど取扱いやすいとは言えません。ログの運用に苦慮しているのであるならば、この機能が pg_statsinfo 採用の最後の一押しになるかもしれません。

© 2016 PostgreSQL Enterprise Consortium

4.3.2. pg_monz の使い時

pg_monz の使い時は性能監視内容を細かくコントロールしたいなど、『拡張性が求められるとき』です。また、元々システム監視ツールとして Zabbix を採用する予定であり、『監視全般を Zabbix にて一括で提供したいとき』も使い時です。Zabbix にて、pg_monz による PostgreSQL の性能監視だけでなく、OS やサービス、PostgreSQL の死活監視機能なども一括で提供すると、システムの構成をシンプルにできるというメリットがあります。

以下でより詳細な使い時を挙げます。

(1) 拡張性が求められるとき

pg_monz は拡張性に優れ、監視項目の追加が必要になっても独自でそれを実施できます。pg_monz は Zabbix 上で実装されています。Zabbix はユーザが自由に監視項目や発報条件、監視内容のグラフ表示方法などを設定できます。また、それらの設定項目をまとめて「テンプレート」として取り扱うこともできます。pg_monz はこの「テンプレート」として纏めた形で公開されています。つまり pg_monz を導入した時、追加の監視要件が発生しても、テンプレートの変更などでそれを追加して対応が可能です。

以下で pg_monz の拡張性について、もう少し踏み込んで紹介します。

≪監視項目に対する拡張性≫

pg_monz はユーザが望んだ監視項目の追加ができます。実際に PostgreSQL を運用していると、監視項目の追加がしばしば必要になります。例えば、長い期間運用されたシステムには、様々なトラブルが発生します。しかしそのトラブルに対して根本解決を図る変更を即時に施せるケースは必ずしも多くありません。その様なケースでは、まずはそのトラブルに関連する項目を監視して様子を見る、というのが現実的な対応になるでしょう。しかし、トラブル対応で追加した監視項目は「担当者しか分からない/知らない」内容になりがちです。pg_monz を導入し、Zabbix で監視項目を一元管理できると比較的そのリスクを軽減できるかもしれません。特に、アプリケーションの追加/拡張を繰り返すようなシステムの場合は、構築当時には想定していなかった監視要件が発生しやすいです。その結果、pg_monz の拡張性に頼るケースが多くなるかもしれません。

pg_monz は監視項目の追加ができるものの、追加にはある程度の手間がかかります。上述のように、pg_monz を運用する際、監視項目に対する拡張性は大きな武器となります。利用ケースによってはこの拡張性に期待して、pg_statsinfo で取得している性能情報など、様々な情報を追加で取得したくなるかもしれません。しかし、項目の追加は必ずしも簡単ではないので、その点は考慮しておく必要があります。

pg_monz への監視項目追加に手間がかかるのは、PostgreSQL への問い合わせスクリプトを自前で用意する必要があるためです。Zabbix は「OS のディスク使用量」や「CPU 使用率」など、様々な情報を取得する機能を標準で有しており、Zabbix の設定上でこれを指定すればすぐに監視項目として追加できるようになっています。しかし、Zabbix には「PostgreSQL のキャッシュヒット率」など PostgreSQL の性能情報を取得する機能は有していません。そのため、Zabbix で追加の性能情報を取得したい場合、以下を自前で実装する必要があります。

- 性能情報を取得するスクリプト
  - 多くは pg_stat*ビューを psql などで select するもの
- 上記スクリプトで得られた情報を Zabbix へ登録するスクリプト/仕組み
- 上記に対する Zabbix 側の設定

また、「全インデックスについて、網羅的かつ動的に情報取得したい」ケースなどはこれに加えて Zabbix の「ローレベルディスカバリ」という機能を用いて、以下のようなスクリプトも実装する必要があります。

- 取得したい対象を動的に通知するスクリプト
  - 上記インデックスの例だと、pg_indexes を select し、全ての名前を Zabbix に通知するもの

監視項目を追加するために、このような機能を自前で用意する場合、pg_monz の実装方法が参考になりますし、これに合わせておくと管理もしやすいでしょう。しかし pg_monz の実装部分は以下の様にやや複雑になっています。特にバージョン 2.0 以降で取得項目のグルーピングやその管理を容易にする便利な機能を追加したことにより、実装は高度になっています。もしも実装の仕組が複雑で手に余ると感じたならば、pg_monz とは独立した簡易的な構造で実装することを検討しても良いでしょう。

© 2016 PostgreSQL Enterprise Consortium

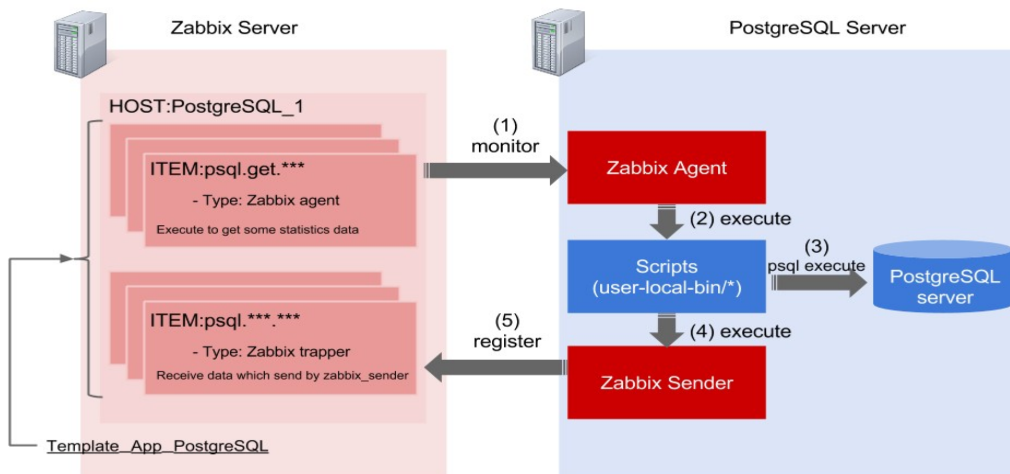


図 4.3: pg_monz の監視データの流れ

※ pg_monz 公式サイトのマニュアル[2]より

pg_monz にとって監視項目に関する拡張性は大きな利点です。しかし、上述の通り項目の追加はある程度手間がかかります。そのため、実際の運用ではこの追加の手間も念頭に入れ、項目追加で得られる価値と実装のコストを比較し、バランスの良い監視設定を心掛けると良いでしょう。

≪監視情報の出力方法 に対する拡張性≫

ここまで、pg_monz の監視項目に関する拡張性について述べてきました。しかし、前述のとおり Zabbix は監視項目だけでなく、グラフ表示方法などもカスタマイズが可能です。これらの機能を使うことで、「監視項目」だけでなく「監視情報の出力方法」についても拡張性を享受できます。

この出力方法に対する拡張性は、性能監視のグラフを実際の業務フローに載せる際に大きな力を発揮します。例えば、マルチテナント型のシステムなどで、1つの PostgreSQL データベースクラスタ上で複数のデータベースを用意し、データベースごとにアプリケーション担当者が別、という運用体制で回っているシステムがあったとします。このような体制でデータベースのレビューなどを実施する時、全データベース横断のグラフを用いるよりは、担当しているデータベースに絞ったグラフを用意できたほうが、業務は恐らくスムーズに回るでしょう。pg_monz ならば、このような細かい要求に沿ったグラフを用意できます。加えて、カスタムグラフ作成は Zabbix の GUI 操作だけで完結する簡単な作業なため、頻繁な監視要件の変更にも十分対応できます。業務に合わせたグラフを自前でカスタマイズできることは pg_monz の大きな利点と言えるでしょう。

2 http://pg-monz.github.io/pg_monz/

© 2016 PostgreSQL Enterprise Consortium

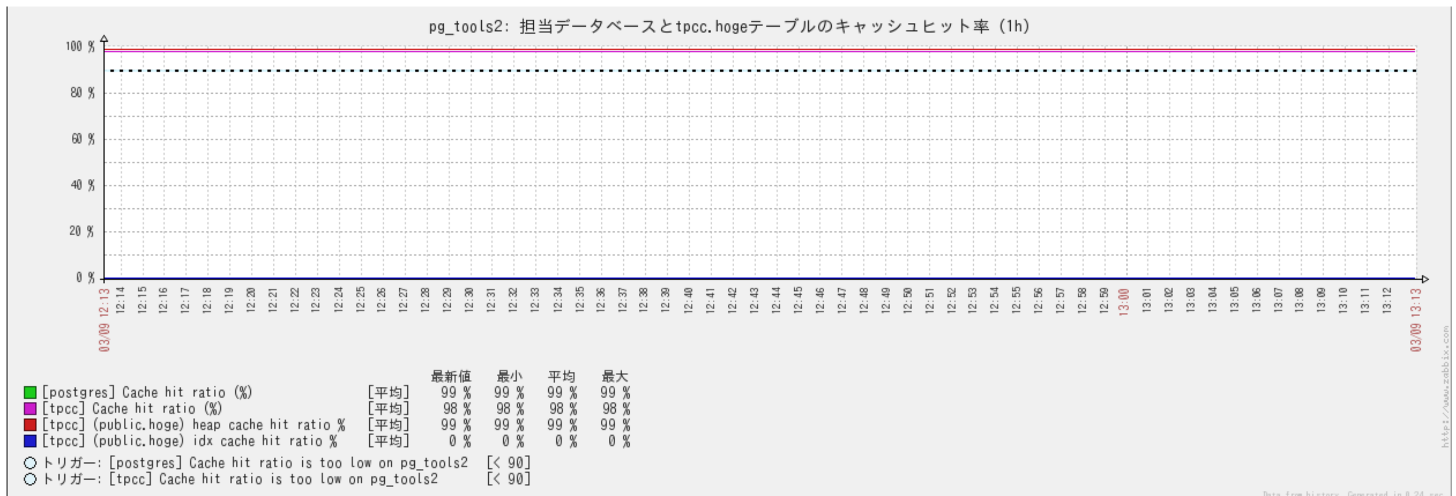


図 4.4: 興味のあるデータベースとテーブルに絞ったキャッシュヒット率のカスタムグラフ

また、Zabbix には WebAPI 機能が標準で用意されており、これも強力な拡張性を提供します。WebAPI を用いると http 通信にて、監視情報を画像形式ではなく、生の数値データが入った JSON 形式にて取得できます。WebAPI の機能を活用することで自前で用意したシステムと連携できるようになります。例えば、自前でポータル機能用アプリなどを用意している組織ならば、これに Zabbix API をアクセスするコードを追加することで、ポータル画面から pg_monz 側のコントロールや pg_monz 関連情報を表示する機能を実装できます。また、監視情報の履歴をアーカイブする運用をしている組織ならば、WebAPI で情報の取得と選別をして保存することで、スムーズなアーカイブ運用が可能になるかもしれません。また、WebAPI を活用することで、Zabbix 単体では実現しにくい機能を外部から用意出来るケースもあります。Zabbix 単体で実装しにくい機能の例として、ランキング機能がありますが、WebAPI にて一括取得したデータを自前でソートするコードを追加すれば、これも実現可能です。
WebAPI については、付録にも情報がありますので、興味を持たれた方はそちらもご覧ください。

WebAPI による拡張は有用ですが、直接リポジトリのデータベースで SQL 文を実行したほうが便利なケースもあります。それは特に、WebAPI で可能な問い合わせに表現力の不足を感じるケースです。例えば、分析の一環として特定スキーマ上にあるテーブルのうち、キャッシュヒット率の低いテーブルのランキングを取得したいとします。Zabbix のリポジトリデータベース上では、どのような監視項目か(どのスキーマのテーブルか)とその実際のデータ部分を別テーブルに記録しています。SQL 文ならば両テーブルを結合すればよいだけですが、WebAPI では両者を結合する形で問い合わせすることができません。結果、両テーブルの情報を個別に取得して自前で結合する処理を実装する必要があります。これは簡単な例なので WebAPI を利用した実装も少々手間なだけですが、性能について複雑な分析を実施する必要がある場合、それが可能な環境なら SQL 文での問い合わせをしたほうが良いケースもあるでしょう。

Zabbix のリポジトリ DB に直接 SQL 文を実行して分析する場合、データ構造には注意が必要です。Zabbix はユーザが後から登録した監視項目にも対応するために、全監視項目のデータをまとめて1種類のテーブル(正確には、記録するデータ型の応じて整数・浮動小数点・文字列・テキスト・ログの 5 つのテーブル)に格納する、というテーブル設計になっています。これはエンティティ・アトリビュート・バリューと呼ばれる構造に近く、SQL 文が複雑になりがちです。もしもこのような分析が必要となりそうな場合は、予め Zabbix のデータ構造をチェックしておくと、スムーズな運用が可能になるでしょう。

(2) 各種監視機能も一括で提供したいとき

pg_monz を導入する際、Zabbix が必要ですが、この Zabbix を用いて PostgreSQL 以外の監視や PostgreSQL の死活監視なども提供できます。監視など管理系のサーバは台数削減の対象になりやすいです。また、監視系はシステム横断でノウハウを共有しやすく、システム間で近い構成になりがちです。結果シンプルな構成が好まれます。そのため、pg_monz にて、PostgreSQL の性能監視と併せて、各種死活監視などを一括で提供できることは大きな利点となります。また、Streaming Replication のクラスタ状態の監視など、実装が難しい機能も pg_monz 側で用意されているので、監視実装の手間を大幅に減らせる点もポイントです。

また、Zabbix は監視ツールなので、問題発生時にメールなどを発報する機能も有しています。「性能管理ツール」として考えたとき、サービス障害など死活監視の発報ほど迅速な対応が必要なケースはそれほど多くないかもし

© 2016 PostgreSQL Enterprise Consortium

れません。しかし、性能劣化の兆候を見逃してしまうことを防ぐ目的などでこの発報機能を活用することもできるでしょう。また、Zabbix は PostgreSQL 専用ツールではなく、専用のレポーティング機能もないので、性能に関する各種情報を横断した確認が若干やりにくいです。そのため、相対的に発報機能による通知が重要になるという側面もあります。

結果、pg_monz の「性能管理ツール」という側面においても発報機能は重要な役割となるでしょう。

上記のような各種監視機能や発報機能など、システムに必須な機能を一括で提供できることは pg_monz の大きな魅力の一つです。

(3) 性能情報の長期保管が必要など、データ量に懸念があるとき

pg_monz は pg_statsinfo と比べると取得している情報の種類がやや少ないです。しかしこれは裏を返すと、必要なストレージ量が少なく、メンテナンスも実施しやすいというアドバンテージにもなります。また、監視項目ごとにデータ保存期間を調整できる機能も有しています。

上記機能のため、長期間のトレンドを確認する用途にも比較的向いています。

pg_statsinfo との具体的なデータ量の比較については、付録の検証結果をご覧ください。

© 2016 PostgreSQL Enterprise Consortium

5. バックアップツール

5.1. バックアップ運用に求められる要件

企業活動に纏わるあらゆるデータがデータベースに格納されている昨今、データベースのデータの消失があっては企業活動の停止にもつながります。そのため不慮の事故でもデータを保持し続けることができるよう、定期的にバックアップを取得しておくことが、堅牢なシステム運営のために必要になります。

また、複数世代分のバックアップを保持しておくというのも重要です。なぜなら一世代しかバックアップを保持していない場合、その唯一のバックアップが万が一にも破損していたり、誤って削除された場合には、データのリストアができなくなってしまうからです。

一般的にバックアップ運用に求められる要件として、表 5.1 のような項目が挙げられます。

表 5.1: バックアップ運用に求めれられる要件

要件	目的
オンラインバックアップ	サーバを停止させない
バックアップの自動実行	運用コストの削減
バックアップの世代管理	バックアップ管理
バックアップファイルの圧縮	リソース節約
増分バックアップの取得	リソース節約
バックアップファイルを遠隔地で保存する	災害対策

またバックアップの自動メンテナンス機能としては、例として表 5.2 のような項目があります。

表 5.2: バックアップの自動メンテナンス機能

機能	目的
設定数分だけバックアップを保持する	冗長性
保存期限が過ぎたバックアップの削除	リソース節約

本章では代表的な PostgreSQL のバックアップツールを紹介し、それぞれのツールが前述の要件をどこまで満たしているのかを説明します。

5.2. PostgreSQL のバックアップ機能

PostgreSQL は標準機能として論理バックアップを取得するための pg_dump コマンド、物理バックアップを取得するための pg_basebackup コマンドが用意されていますが、複雑なバックアップ運用を実現したい場合はユーザ側でスクリプト等の作り込みが必要で手間がかかるものでした。

本章では、PostgreSQL の物理バックアップに焦点を絞って、標準機能である pg_basebackup と、代表的なサードパーティツールである pg_rman、Barman、OmniPITR の 4 つのツールを取り上げます。

5.2.1. バックアップツールの比較

それぞれのバックアップツールの公開情報は表 5.3 の通りです。

© 2016 PostgreSQL Enterprise Consortium

表 5.3: PostgreSQL の代表的なバックアップツール

ツール名	pg_basebackup	pg_rman	Barman	OmniPITR
最新バージョン	9.5	1.3.1	1.5.1	1.3.3
ツール概要	データベースクラスタの物理バックアップ実行ツール	物理バックアップ・リストア実行ツール	DR(disaster recovery)管理ツール	WAL ファイル運用の補助や、データベースクラスタの物理バックアップを取得するスクリプト
実装方式	C	C	Python	Perl
対応バージョン(OS、DB 等の要件)	OS：Linux/Unix、Windows DB：PostgreSQL9.1～	OS：RHEL 5/6/7 、Ubuntu 12.04LTS DB：PostgreSQL8.4～	OS：Linux/Unix 系 DB：PostgreSQL8.4～	OS：Linux、Solaris DB：PostgreSQL8.2～
ライセンス形態	PostgreSQL License	BSD	GNU GPL3	PostgreSQL License
開発元	PGDG	NTT OSS センタ	2ndQuadrant	omniTI
入手元(URL)	http://www.postgresql.org/	https://github.com/ossc-db/pg_rman/releases	http://www.pgbarman.org/	https://github.com/omniti-labs/omnipitr
有償サポート	日本の企業で複数社あり、PostgreSQL 標準機能であるためサポートレベルは高い	有(国内企業複数社)	有(2ndQuadrant 社)	無(メーリングリストはあり)

ライセンス体系は異なりますがいずれも OSS 公開されており、無償で利用することができます。
また、国内企業がサポートしており、導入が比較的容易なツールもあります。

5.2.2. バックアップ機能の評価観点

5.1 節で紹介したバックアップに求められる要件を基に、次節以降では 4 つのバックアップツールを表 5.4 のような項目を観点として評価します。

表 5.4:評価観点

項目		**評価観点**
大項目	**小項目**	
バックアップ取得方法	オンラインバックアップの実行	DB を停止させないでバックアップを取得可能か
	スタンバイサーバでバックアップ取得	レプリケーション構成のスタンバイサーバからバックアップを取得可能か
	リモートサイトからのバックアップ取得	DB とは別サーバから接続してバックアップを取得可能か
バックアップの管理	バックアップファイルの圧縮	バックアップファイルを圧縮形式で取得可能か
	バックアップの世代管理	取得したバックアップの世代管理が可能か
	リストア機能の有無	ツールにバックアップのリストア機能があるか (リカバリには場合によってはバックアップ取得後の情報も必要となるため、あくまでリストアのみを評価対象とする)
	不要なバックアップの削除	不要となったバックアップを自動で削除可能か
バックアップの対象範囲	サーバログの取得	PostgreSQL のサーバログもバックアップ対象にすることが可能か
	WAL の取得	WAL もバックアップ対象にすることが可能か
	設定ファイルの取得	postgresql.conf、pg_hba.conf、recovery.conf の設定ファイルもバックアップ対象とすることが可能か
	テーブルスペース対応	個別に作成したテーブルスペースもバックアップ対象とすることが可能か
	増分バックアップの取得	データの増分バックアップが取得可能か

© 2016 PostgreSQL Enterprise Consortium

なお、バックアップのスケジュール実行については、例えばOSのスケジューラや、ジョブ実行管理ツールなどの手段で実現可能であり必須の要件ではないとみなし、本章では比較の対象として扱いません。

また、バックアップを取得するにはバックアップ対象のサーバやDBに接続する必要があります。どのような権限を持ったユーザでバックアップが可能なのか、またどの程度セキュリティ要件を緩める必要があるのかも、バックアップ運用では考慮したい点です。次節以降では、各ツールのセキュリティ要件を以下の項目で評価します。

表 5.5: セキュリティ要件の評価項目

項目	評価観点
特定ポート解放の必要性	ツールを使用するにあたってサーバ側で特定のポートを解放する必要があるか
root ユーザである必要性	ツールのコマンド実行ユーザが OS の root 権限ユーザである必要があるか
認証なしの SSH 接続である必要性	リモートサイトから接続してバックアップを取得する場合に、サーバ接続が認証なしの SSH 接続である必要があるか
DB の superuser 権限の必要性	DB に接続するユーザが superuser 権限を有している必要があるか
pg_hba.conf の trust 認証の必要性	DB に接続するユーザが pg_hba.conf で接続認証方法が trust に設定されている必要があるか

5.2.3. バックアップ機能の評価

5.2.2 項で紹介した評価観点を基に実際にツールを評価した結果が表 5.6 になります。
評価についての詳細や注意点は、次節以降の各ツール紹介の中で記述しています。

表 5.6: バックアップ機能の評価結果

項目		ツール			
大項目	小項目	pg_basebackup	pg_rman	Barman	OmniPITR
バックアップ取得方法	オンラインバックアップの実行	○	○	○	○
	スタンバイサーバでバックアップ取得	○	○	○	○
	リモートサイトからのバックアップ取得	○	×	○	×
バックアップの管理	バックアップファイルの圧縮	○	○	○	○
	バックアップの世代管理	×	○	○	×
	リストア機能の有無	−(※)	○	○	×
	不要なバックアップの削除	×	○	○	○
バックアップの対象範囲	サーバログの取得	×	○	×	×
	WAL の取得	○	○	○	○
	設定ファイルの取得	○	○	○	○
	テーブルスペース対応	○	○	○	○
	増分バックアップの取得	×	○	○	×

(※)pg_basebackup はデータベースクラスタファイルをコピーするため、リストア機能は必要ありません。

評価は以下のように行いました。

・○…ツールにコマンド等が用意されており、そのツール単体で完全に実現可能
・△…ツールである程度補うことはできるが、完全に実現するには他ツールと組み合わせる必要がある
・×…ツールでは実現不可能
・ー…評価対象にしない

© 2016 PostgreSQL Enterprise Consortium

セキュリティ要件を評価した結果は表 5.7 になります。
それぞれの評価についての詳細や、注意点などは、次節以降の各ツールについての紹介の中で記述しています。

表 5.7: セキュリティ要件の評価結果

項目	ツール			
	pg_basebackup	pg_rman	Barman	OmniPITR
特定ポート解放の必要性	不要	–	必要	不要
root ユーザである必要性	不要	不要	不要	不要
認証なしの SSH 接続である必要性	不要	不要	必要	不要
DB の superuser 権限の必要性	不要	–	必要	必要
pg_hba.conf の trust 認証の必要性	不要	不要	不要	不要

評価は以下のように行いました。
・必要…ツールを使用するにあたって設定・制約が必要となる
・不要…ツールを使用するにあたって設定・制約は特に不要
・ー…ツールの使用に関係がないため評価対象にしない

5.3. pg_basebackup

pg_basebackup はデータベースクラスタの物理バックアップを取得するコマンドで、一般的には PITR(Point In Time Recovery)のためのベースバックアップや、ストリーミングレプリケーションのスタンバイサーバ構築時に使用されます。
pg_basebackup はローカルでもリモートからでも対象サーバのバックアップを取得することができるため、例えば図 5.1 のような環境で使用することができます。

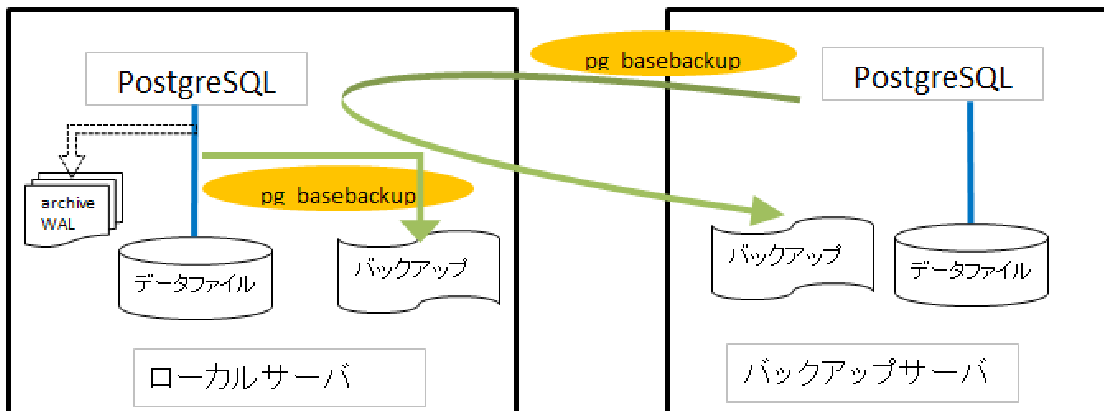


図 5.1: pg_basebackup の使用環境例

pg_basebackup の基本的な操作については、5 章付録_backup ツール内で紹介します。
次項以降では pg_basebackup の基本的な機能と、ツール使用時のセキュリティ要件を紹介します。

5.3.1. pg_basebackup の機能紹介

pg_basebackup の機能は表 5.8 のようになります。
各小項目において、特に pg_basebackup の特徴と言えるものについては後述しています。

© 2016 PostgreSQL Enterprise Consortium

表 5.8: pg_basebackup で実現できる機能

項目		取得可否	備考
大項目	**小項目**		
バックアップ取得方法	オンラインバックアップの実行	○	
	スタンバイサーバ でバックアップ取得	○	
	リモートサイトからのバックアップ取得	○	
バックアップの管理	バックアップファイルの圧縮	○	gzip 形式が可能
	バックアップの世代管理	×	
	リストア機能の有無	○	完全な PITR にはバックアップ取得時以降に作成された WAL を recovery.conf 内の restore_command で適用する必要がある
	不要なバックアップの削除	×	
バックアップの対象範囲	サーバログの取得	×	データベースクラスタ配下にある場合は取得される
	WAL の取得	○	--xlog オプションで可能
	設定ファイルの取得	○	
	テーブルスペース対応	○	--format=plain オプションで可能
	増分バックアップの取得	×	

≪オンラインバックアップの取得≫
pg_basebackup コマンドはバックアップのためのサーバ停止を考慮する必要はなく、またリモートのサーバ(但し PostgreSQL がインストールされている場合)から接続してバックアップを取得することも可能です。

≪スタンバイサーバでバックアップ取得≫
ストリーミングレプリケーション構成を組んでいる場合はスタンバイサーバからバックアップを取得することも可能です。

≪バックアップファイルの圧縮≫
バックアップファイルの圧縮機能として gzip 形式を選択することができます。ただしこの場合はコマンドのオプションでバックアップのフォーマットを tar ファイルにする必要があります。

≪バックアップの世代管理≫
バックアップの世代管理機能は用意されていないため、複数世代のバックアップを保持する場合は、コマンド実行時に、-D/--pgdata オプションで指定するデータベースクラスタファイルの保存先をその都度変更するなど工夫が必要になります。

≪テーブルスペース対応≫
個別に作成したテーブルスペースもバックアップの対象になります。但し、バックアップ元と同じ絶対パス上にテーブルスペースファイルのバックアップを設置するため、同一サーバ内でバックアップを取るときは既存のテーブルスペースが上書きされないよう注意が必要です。なお、バックアップで取得したテーブルスペースには既にシンボリックリンクが貼られている状態ですので、再度リンクを貼り直す必要はありません。

5.3.2. pg_basebackup とセキュリティ要件

pg_baebackup を使用するときのセキュリティ要件は表 5.9 のようになります。

© 2016 PostgreSQL Enterprise Consortium

表 5.9: pg_basebackup 使用時のセキュリティ要件

項目	設定	備考
特定ポート解放の必要性	不要	
root ユーザである必要性	不要	
認証なしの SSH 接続である必要性	不要	
DB の superuser 権限の必要性	不要	replication 権限のあるユーザでよい
pg_hba.conf の trust 認証の必要性	不要	

≪特定ポートの解放≫
pg_basebackup は PostgreSQL サーバが一般のユーザからの接続を待ち受けるポート(デフォルト 5432)を使うため、特定ポートの解放は必要ありません。

≪DB の superuser 権限の必要性≫
バックアップは PostgreSQL のレプリケーションプロトコルで接続されますので、DB 接続ユーザは replication 権限が与えられているユーザであれば実行可能です。

≪pg_hba.conf の trust 認証の必要性≫
DB 接続時の認証方法は pg_hba.conf で選択できる認証方法の全てを選択することができます。

このようにリモートサイトからの接続であってもバックアップ取得のために既存の PostgreSQL のセキュリティ要件を弱めなくても済むというポイントは pg_basebackup の大きな利点と言えるでしょう。

5.4. pg_rman

pg_rman(PostgreSQL Recovery Manger released)は NTT OSS センタが開発している PostgreSQL 専用の物理バックアップ・リストアツールです。
pg_rman はローカルサーバ内の PostgreSQL のバックアップしか取得できないため、PostgreSQL と同一サーバ内に pg_rman をインストールする必要があります。

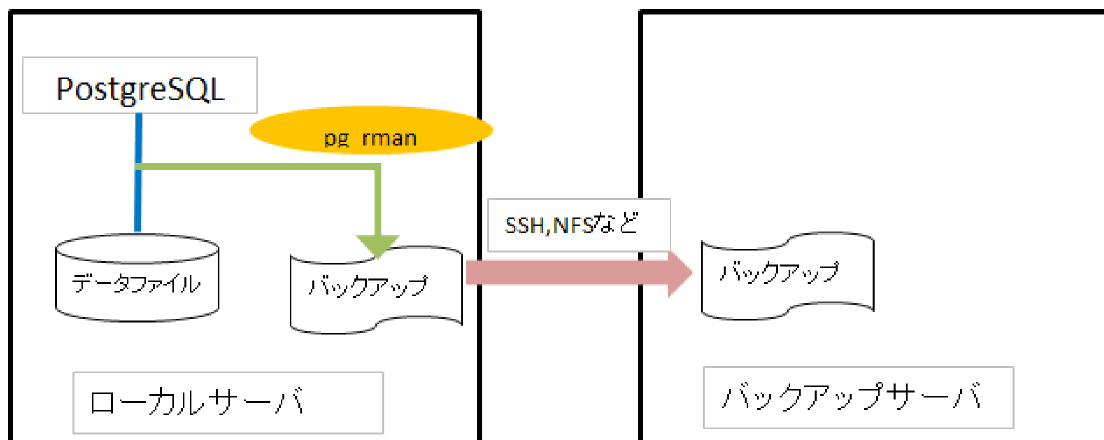


図 5.2: pg_rman の使用環境例

pg_rman の導入手順、基本的な操作は、5 章付録_backup ツール内で紹介します。
次項以降では pg_rman の基本的な機能と、ツール使用時のセキュリティ要件を紹介します。

© 2016 PostgreSQL Enterprise Consortium

5.4.1. pg_rman の機能紹介

pg_rman は取得したバックアップをバックアップカタログという領域に保存します。そのため pg_rman をインストール直後はこのバックアップカタログを初期化する必要がありますが、この時 pg_rman.ini ファイルが作成されます。この pg_rman.ini にはサーバーログのパス、アーカイブ WAL のパス、リテンションポリシーなどを設定します。pg_rman は特に指定がない限りは pg_rman.ini に記載した設定をデフォルト設定としてバックアップを実行します。

```
≪pg_rman.ini ファイルに記載されている内容≫
[postgres@postgres01 pg_rman]$ cat pg_rman.ini
ARCLOG_PATH='/usr/local/pgsql/archive/'
SRVLOG_PATH='/usr/local/pgsql/data//pg_log'
BACKUP_MODE = full
COMPRESS_DATA = YES
KEEP_ARCLOG_FILES = 10
KEEP_ARCLOG_DAYS = 10
KEEP_DATA_GENERATIONS = 3
KEEP_DATA_DAYS = 120
KEEP_SRVLOG_FILES = 10
KEEP_SRVLOG_DAYS = 10
```

pg_rman.ini に指定できる項目と、バックアップ実行時にオプションとして指定する項目を踏まえると、pg_rman の機能は表 5.10 のようになります。
各小項目において、特に pg_rman の特徴と言えるものについては後述しています。

表 5.10: pg_rman で実現できる機能

項目		取得可否	備考
大項目	**小項目**		
バックアップ取得方法	オンラインバックアップの実行	○	
	スタンバイサーバでバックアップ取得	○	
	リモートサーバからのバックアップ取得	×	
バックアップの管理	バックアップファイルの圧縮	○	gzip 形式が可能
	バックアップの世代管理	○	
	リストア機能の有無	○	完全な PITR にはバックアップ取得時以降の WAL を別途適用する必要がある
	不要なバックアップの削除	○	pg_rman delete の場合、管理情報は残る pg_rman purge で管理情報の削除が可能
バックアップの対象範囲	サーバログの取得	バックアップモードによって異なる 詳しくは表 5.11 を参照	
	WAL の取得		
	設定ファイルの取得		
	テーブルスペース対応	○	
	増分バックアップの取得	○	

≪スタンバイサーバでバックアップ取得≫
pg_rman でもストリーミングレプリケーションしているスタンバイサーバからバックアップを取得することが可能ですが、設定するオプションに少々注意が必要になります。
具体的には、-D/--pgdata オプションでスタンバイサイトのデータベースクラスタを指定し、その他の接続オプション(-d/--dbname、-h/--host、-p/--port など)でマスターサイトの設定情報を指定する必要があります。また、スタンバイ接続オプション(--standby-host、--standby-port)でスタンバイサイトの設定情報を指定する必要があります。

© 2016 PostgreSQL Enterprise Consortium

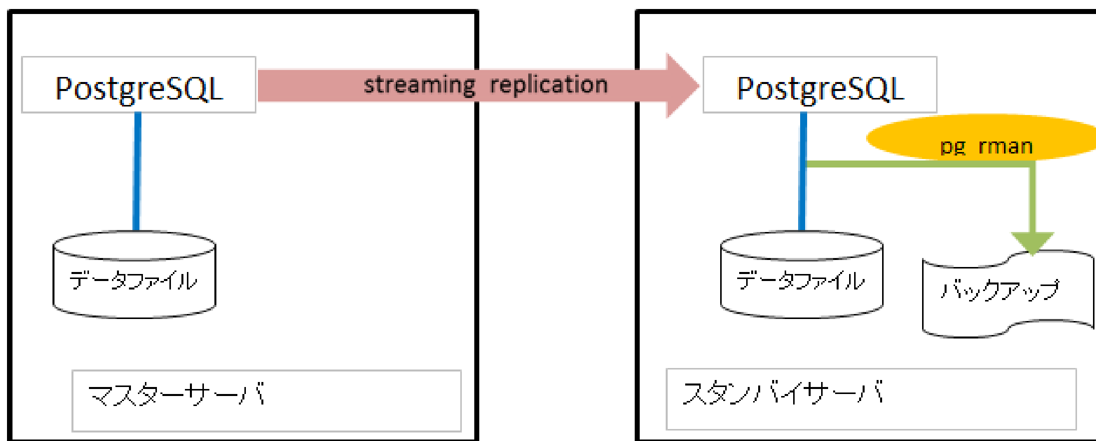


図 5.3: スタンバイサーバでバックアップを取得する例

コマンド実行例

```
$pg_rman backup --pgdata=/home/postgres/stdby_pgdata --backup-mode=full --host=master
--standby-host=localhost --standby-port=5432
```

≪リモートサーバからのバックアップ取得≫
pg_rman はリモートから DB に接続してバックアップを取得することができません。また、取得したバックアップをリモートサーバに保存する機能もありません。したがって、バックアップを遠隔地に保存したい場合は、SSH や NFS 等を使用して、一度ローカル上で取得したバックアップを別途転送する方法が挙げられます。

≪バックアップの世代管理≫
pg_rman show コマンドでバックアップの取得時間、データ量、WAL やログのサイズ、バックアップの種類(フル or 増分)等が確認可能です。

```
$ pg_rman show
==========================================================
 StartTime           Mode  Duration    Size   TLI  Status
==========================================================
2016-02-25 11:14:18  FULL        0m  4470kB     1  OK
2016-02-24 17:40:42  INCR        0m      0B     1  ERROR
2016-02-24 17:38:31  FULL        0m  4362kB     1  OK
2016-02-24 16:59:56  FULL        0m  4307kB     1  OK
```

pg_rman show detail コマンドでは取得したバックアップのタイムライン ID を確認することができるため、古いバックアップの削除運用の目安になります。

```
$ pg_rman show detail
============================================================================================================
 StartTime           Mode  Duration    Data  ArcLog  SrvLog   Total  Compressed  CurTLI  ParentTLI  Status
============================================================================================================
2016-02-25 11:14:18  FULL        0m    21MB   134MB    ----  4470kB        true       1          0  OK
2016-02-24 17:40:42  INCR        0m      0B    ----    ----      0B        true       1          0  ERROR
2016-02-24 17:38:31  FULL        0m    21MB    67MB    ----  4362kB        true       1          0  OK
2016-02-24 16:59:56  FULL        0m    21MB    33MB    ----  4307kB        true       1          0  OK
```

また、pg_rman show コマンドの StartTime の値を指定することで、個別のバックアップについての詳細な情報を確認することができます。

```
$ pg_rman show '2016-02-24 16:59:56'
# configuration
BACKUP_MODE=FULL
FULL_BACKUP_ON_ERROR=false
WITH_SERVERLOG=false
COMPRESS_DATA=true
# result
```


© 2016 PostgreSQL Enterprise Consortium

```
TIMELINEID=1
START_LSN=0/02000028
STOP_LSN=0/020000b8
START_TIME='2016-02-24 16:59:56'
END_TIME='2016-02-24 17:00:22'
RECOVERY_XID=1810
RECOVERY_TIME='2016-02-24 17:00:19'
TOTAL_DATA_BYTES=21172045
READ_DATA_BYTES=21172045
READ_ARCLOG_BYTES=33554748
WRITE_BYTES=4307891
BLOCK_SIZE=8192
XLOG_BLOCK_SIZE=8192
STATUS=DONE
```

≪不要なバックアップの削除≫
古いバックアップの削除は pg_rman delete コマンドで可能です。但し、delete コマンドで消去した場合データ自体はファイルシステムから削除されますが、「このバックアップは削除された」というバックアップの管理情報は残ります。
(pg_rman show コマンドから確認可能です)
管理情報も含め完全に削除するには、pg_rman purge コマンドを実行する必要があります。

≪バックアップの範囲≫
pg_rman は取得したバックアップファイルや管理情報をバックアップカタログという領域に保存します。
バックアップカタログ内には表 5.11 にあるディレクトリ・ファイル一覧が保存されますが、バックアップのモードによっては取得されないものもあります。

表 5.11: バックアップカタログ内のファイルとバックアップモード

バックアップカタログ内のディレクトリ・ファイル名	**ディレクトリ・ファイルの内容**	**バックアップモード**		
		フルバックアップ(full)	**増分バックアップ(incremental)**	**アーカイブ WAL バックアップ(archive)**
arclog	アーカイブされた WAL ファイル	○	○	○
backup.ini	バックアップの詳細情報	○	○	○
database	データベースクラスタ内のファイル	○	△(※1)	△(※1)(※2)
file_arclog.txt	アーカイブ WAL のバックアップ時刻	○	○	○
file_database.txt	データベースクラスタ内ファイルのバックアップ時刻	○	○	×
mkdirs.sh	バックアップファイル内にデータベースクラスタ以下のファイルを再作成するためのスクリプト	○	○	×
srvlog	サーバログ	○(※3)	○(※3)	○(※3)

(※1)設定ファイル(*.conf)は取得されません。
(※2)バックアップカタログ内にファイルは存在しますが中見は空です。
(※3)バックアップ取得時に--with-serverlog オプションをつけることで取得可能です。

フルバックアップとその他のモードとの大きな違いとして、*.conf といった設定ファイルはフルバックアップモードのときのみ取得されることがあげられます。
また、増分バックアップモードの時はデータベースクラスタ配下にあるファイルの一部もバックアップされますが、アーカイブ WAL モードの時はデータベースクラスタ配下のファイルは取得されず中身が空の database ファイルのみが作成されます。したがってデータベースクラスタ配下のファイルのバックアップ情報が記入される file_database.txt ファイルと、バックアップカタログ内にそれらのファイルを作成するための mkdir.sh もアーカイブ WAL モードの時は作成されません。

pg_rman backup コマンド実行時には、PostgreSQL 内部では pg_start_backup 関数、pg_stop_backup 関数が実行されます。これらの関数は実行時に WAL のスイッチが行われ PostgreSQL 内部では checkpoint 処理が走るため、バックアップに含まれる WAL ファイルは自動的にアーカイブされた状態になります。

© 2016 PostgreSQL Enterprise Consortium

尚、増分バックアップを取得する場合、下記の 2 点に注意する必要があります。
(1)現状のタイムライン内で最低一度はフルバックアップを取得する必要があります。
フルバックアップの取得実績がない場合、以下のようなエラーメッセージが出力されます。

```
INFO: database backup start
ERROR: There is no validated full backup with current timeline.Please take a full backup and validate it before doing an incremental backup.
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
```

(2)増分バックアップ前に取得したバックアップは必ず pg_rman validate コマンドを実行し、バックアップを検証する必要があります。
なぜなら未検証のバックアップは新規に増分バックアップを取得する際の基準ポイントとして見なされないからです。validate されたバックアップがない場合、新規に増分バックアップを取得しようとしても以下のような ERROR メッセージが出力されバックアップは失敗します。

```
$ pg_rman backup --backup-mode=incremental
INFO: copying database files
ERROR: cannot take an incremental backup
DETAIL: There is no validated full backup with current timeline.
HINT: Please take a full backup and validate it before doing an incremental backup. Or use with --full-backup-on-error command line option.
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
```

≪テーブルスペース対応≫
pg_rman は個別に作成したテーブルスペース領域もバックアップ対象とします。pg_rman restore コマンドでバックアップのリストアが可能ですがこの時テーブルスペースも自動で作成され、シンボリックリンクも貼られますので、リストア後ユーザが特に意識してシンボリックリンクを再作成する必要はありません。

5.4.2. pg_rman とセキュリティ要件

pg_rman を使用するときのセキュリティ要件は表 5.12 のようになります。

表 5.12: pg_rman 使用時のセキュリティ要件

項目	設定	備考
特定ポート解放の必要性	-	リモートからの取得は不可のため評価しない
root ユーザである必要性	不要	
認証なしの SSH 接続である必要性	-	リモートからの取得は不可のため評価しない
DB 接続での superuser 権限の必要性	不要	superuser 権限もしくは replication 権限のどちらかがあればよい
pg_hba.conf の trust 認証の必要性	不要	

≪root ユーザである必要性≫
pg_rman コマンド実行ユーザは OS のユーザである必要はなく、データベースクラスタ・アーカイブ WAL 格納先・バックアップ先のファイルへアクセスできる権限があるユーザで実行可能です。

≪DB 接続での superuser 権限の必要性≫
pg_rman backup コマンド実行時に指定する DB 接続ユーザは superuser 権限もしくは replication 権限を付与されていれば DB に接続してバックアップを取得することができます。DB 接続ユーザにどちらの権限も付与されていない場合は、以下のようにエラーメッセージが出力され、バックアップは実行されせん。

```
$ pg_rman -U test backup
INFO: copying database files
ERROR: query failed: ERROR:  must be superuser or replication role to run a backup query was:
SELECT * from pg_xlogfile_name_offset(pg_start_backup($1, $2))
```


© 2016 PostgreSQL Enterprise Consortium

5.5. Barman

Barman(backup and recovery manager)は 2ndQuadrant 社が開発した PostgreSQL の DR(disaster recovery)管理ツールです。

Barman はリモートから PostgreSQL に接続してバックアップを取得することが可能なため、Barman と同一サーバ内に PostgreSQL がインストールされている必要はありません。また Barman の方でバックアップ対象を設定するため、設定次第では一つの Barman で複数台の PostgreSQL のバックアップを取得するが可能です。

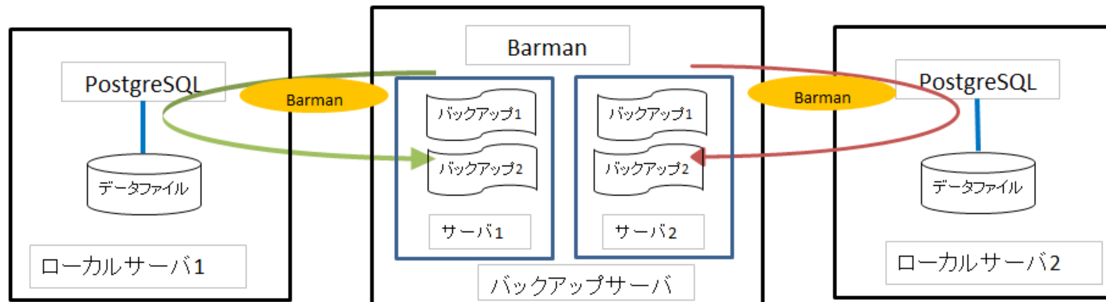


図 5.4:Barman の使用環境例

Barman の導入手順～基本的な操作は 5 章付録_backup ツール内で紹介します。
次項以降では Barman の基本的な機能・使用時のセキュリティ要件を紹介します。

5.5.1. Barman の機能紹介

Barman には設定ファイルとして barman.conf が用意されています。この barman.conf には Barman の設定、バックアップのリテンションポリシー、バックアップ対象サーバへの接続情報等を設定します。この設定情報は複数台分を記述できますので、バックアップ対象サーバが複数台ある場合でも対応可能です。

Barman で実現できるバックアップの機能は表 5.13 のようになります。
各小項目において、特に Barman の特徴と言えるものについては後述しています。

表 5.13: Barman で実現できる機能

項目		取得可否	備考
大項目	小項目		
バックアップ取得方法	オンラインバックアップの実行	○	
	スタンバイサーバでバックアップ取得	○	pgespresso のインストールが必要 rsync のインストールが必要
	リモートサイトからのバックアップ取得	○	SSH 接続で取得可能
バックアップの管理	バックアップファイルの圧縮	○	gzip、bzip2、custom 形式が選択可能
	バックアップの世代管理	○	
	リストア機能の有無	○	
	不要なバックアップの削除	○	削除コマンドはあるが自動実行機能はない
バックアップの対象範囲	サーバログの取得	×	バックアップモードによらない
	WAL の取得	○	バックアップモードによらない
	設定ファイルの取得	○	バックアップモードによらない
	テーブルスペース対応	○	
	増分バックアップの取得	○	llink と copy の 2 モードが選択可能

© 2016 PostgreSQL Enterprise Consortium

≪スタンバイサーバからのバックアップ取得≫
Barman はストリーミングレプリケーション構成のスタンバイサーバからもバックアップを取得することが可能ですが、その場合 pgespresso という extension ツール及び rsync のインストールも必要になります。pgespresso の導入方法は付録_OmniPITR の利用例で紹介いたします。

≪リモートサイトからのバックアップ取得≫
Barman はリモートサイトから SSH で PostgreSQL に接続してバックアップを取得することが可能ですが、この時 B 両サーバ間で認証なしの SSH 接続できるようあらかじめ設定しておく必要があります。SSH 接続時のホスト名及びユーザ名、DB 接続時のホスト名及びユーザ名は全て barmna.conf という設定ファイル内で記述します。

≪バックアップファイルの圧縮≫
Barman は barman.conf の compression パラメータで指定した圧縮形式でバックアップを取得することが可能です。圧縮形式は gzip、bzip2、custom が選択できます。

≪バックアップの世代管理≫
barman list-backup {サーバ名}で該当サーバのバックアップ一覧を確認することができます。
個別に作成したテーブルスペースがある場合、下記の実行結果のように(tablespace:)としてテーブルスペース情報が追加されます。{サーバ名}には all を指定することもでき、その場合は全サーバで取得したバックアップ一覧が表示されます。

```
$ barman list-backup postgres01
postgres01 20151021T141647 - Wed Oct 21 14:17:02 2015 - Size: 35.3 MiB - WAL Size: 0 B
(tablespaces: tablespace:/usr/local/pgsql/tablespace)
postgres01 20151021T114736 - STARTED
```

≪増分バックアップの取得≫
Barman には前回のバックアップからの増分を取得する増分バックアップモードとして link と copy の 2 種類のモードが選択できます。

link モードでは、前回のバックアップ取得時点から変更のないファイルに対して、そのファイルへのハードリンクを作成します。これによりバックアップファイルのサイズ縮小、およびバックアップ取得にかかる時間を短縮します。

link モードに設定したときのバックアップ実行状況

```
$ barman backup postgres03
Starting backup for server postgres03 in /var/lib/barman/postgres03/base/20160113T114402
Backup start at xlog location: 0/66B47358 (000000030000000000000066, 00B47358)
Copying files.
Copy done.
Asking PostgreSQL server to finalize the backup.
BActuackup size: 41.9 MiB. al size on disk: 15.8 MiB (-62.35% deduplication ratio).
Backup end at xlog location: 0/68000000 (000000030000000000000067, 00000000)
Backup completed
```

copy モードでは、前回のバックアップ取得時点から変更のないファイルについては、そのファイルを取得した時点のバックアップ内からファイルをコピーしてきます。これによりバックアップ取得にかかる時間を短縮しますが、ファイルコピーにはある程度の時間が必要になりますので link モード時に比べると多少バックアップ取得に時間がかかり、またバックアップファイルのサイズも大きくなります。

copy モードに設定したときのバックアップ実行状況

```
$ barman backup postgres03
Starting backup for server postgres03 in /var/lib/barman/postgres03/base/20160113T114704
Backup start at xlog location: 0/677D2FA0 (000000030000000000000067, 007D2FA0)
Copying files.
Copy done.
Asking PostgreSQL server to finalize the backup.
Backup size: 41.9 MiB
Backup end at xlog location: 0/69000000 (000000030000000000000068, 00000000)
Backup completed
```


© 2016 PostgreSQL Enterprise Consortium

5.5.2. Barman とセキュリティ要件

Barman 使用時のセキュリティ要件は表 5.14 のようになります。

表 5.14: Barman 使用時のセキュリティ要件

項目	Barman	
	設定	備考
特定ポート解放の必要性	必要	22 番ポートの解放が必須
root ユーザである必要性	不要	
認証なしの SSH 接続である必要性	必要	
DB の superuser 権限の必要性	必要	
pg_hba.conf の trust 認証の必要性	不要	

≪特定ポート解放の必要性≫
Barman はリモートサイトから PostgreSQL サーバに SSH で接続しますので、22 ポートを解放する必要があります。

≪認証なしの SSH 接続である必要性≫
Barman と PostgreSQL サーバ間の SSH 接続は認証なしの必要がありますので、事前に両サーバ間が公開鍵認証できるよう設定しておく必要があります。

≪DB の superuser 権限の必要性≫
DB への接続ユーザ、パスワードは barman.conf 内の conninfo パラメータで設定可能ですが、接続ユーザは superuser 権限が与えられている必要があります。パスワード情報は conninfo パラメータに記述する以外にも、パスワードファイル(~/.pgpass)に記述しても構いません。

5.6. OmniPITR

OmniPITR はストリーミングレプリケーションにおける WAL の運用や、マスターサーバ、スタンバイサーバからのバックアップを取得するためのスクリプト群です。postgresql.conf、recovery.conf の設定パラメータにコマンドの代わりに記述する、もしくは CLI 上で直接コマンドとして実行するという形で使用します。
なお、OmniPITR には多数のオプションが用意されており、それぞれのスクリプトを実行するにあたって必須とされているものもいくつかありますので、オプションの種類、設定などに気を付けて使用してください。

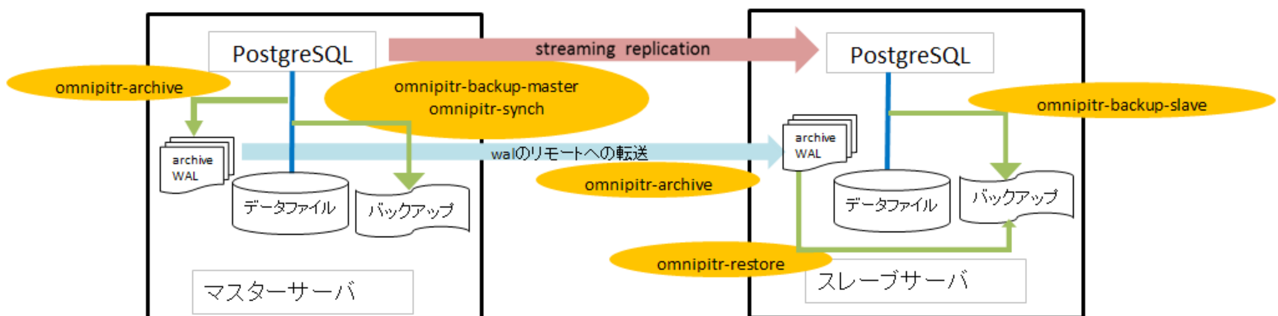


図 5.5:OmniPITR の使用環境例

バージョン 1.3.3 では下記 8 つのスクリプトが用意されています。

© 2016 PostgreSQL Enterprise Consortium

表 5.15: OmniPITR のスクリプト一覧

No	スクリプト名	スクリプト実行サーバ	解説
1	omnipitr-archive	マスターサーバ	WAL のアーカイブ手法を設定できる(保存先、圧縮方法など) postgresql.conf の archive_command に設定する
2	omnipitr-restore	スタンバイサーバ	リストアのために必要な WAL を取得しバックアップに適用させる recovery.conf の restore_command に設定する
3	omnipitr-backup-master	マスターサーバ	マスターサーバからバックアップを取得する 実行するにあたりいくつか設定必須となるオプションがある
4	omnipitr-backup-slave	スタンバイサーバ	スタンバイサーバからバックアップを取得する 実行するにあたりいくつか設定必須となるオプションがある
5	omnipitr-monitor	マスターサーバ	レプリケーションのラグ、直近のバックアップの時刻などのデータを Nagios,Cacti で使用できるよう整形する
6	omnipitr-cleanup	WAL がアーカイブされているサーバ	ストリーミングレプリケーションに不要となった WAL を削除する recovery.conf の restore_command に設定する
7	omnipitr-synch	マスターサーバ スタンバイサーバ	データベースクラスタのコピーを取得する 同時に複数個所で実行することができる テーブルスペースのファイルもコピー対象となる
8	omnipitr-backup-cleanup	バックアップが保存されているサーバ	不要なバックアップを削除する cron などで自動実行されるよう使用する

上記 8 つのうち、バックアップ運用が主目的のスクリプトは No 3,4,7,8 です。WAL の運用が主目的のスクリプトは No1,2 です。
本節ではこれら 6 つのスクリプトついて調査・検証した結果を紹介いたします。

5.6.1. OmniPITR の機能紹介

OmniPITR で実現できる機能は表 5.16 のようになります。
各項目において、特に OmniPITR の特徴と言えるものについては後述しています。

表 5.16: OmniPITR で実現できる機能

項目		**omniPITR**		
		取得結果	**対象スクリプト**	**備考**
バックアップ取得方法	オンラインバックアップ	○	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	
	スタンバイサーバからのバックアップ取得	○	omnipitr-backup-slave omnipitr-synch	
	リモートからのバックアップ取得	×		※リモートへの転送機能オプションにより、遠隔地での保存は可能(-dr/--dst-remote)
バックアップの管理	バックアップファイルの圧縮	○	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	gzip,bzip2,lzma,lz4,xz 形式が選択可能
	バックアップの世代管理	×		バックアップ取得時にファイル名を手動で設定することで対応可能
	リストア機能の有無	×		
	不要なバックアップの削除	○	omnipitr-backup-cleanup	
バックアップの対象範囲	サーバログの取得	×		初期設定(pg_log)の場所にあれば取得される
	WAL の取得	○	omnipitr-archive omnipitr-restore omnipitr-backup-master omnipitr-backup-slave	

			omnipitr-synch	
	設定ファイルの取得	○	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	
	テーブルスペース対応	○	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	-m/--map オプションでテーブルスペースの保存先を指定することが可能
	増分バックアップ	×		

≪オンラインバックアップ≫
OmniPITR ではバックアップ取得のため omnipitr-backup-master、omnipitr-backup-slave、omnipitr-synch の 3 種類のスクリプトが用意されています。omnipitr-backup-master は PostgreSQL のマスターサーバで実行することで、マスターのバックアップを取得します。omnipitr-backup-slave は PostgreSQL のスタンバイサーバで実行することで、スタンバイのバックアップを取得します。omnipitr-synch は標準機能の pg_basebackup と同一の機能を持ち、データベースクラスタの物理バックアップを取得します。一点 pg_basebackup と違うところは、omnipitr-synch はテーブルスペースの再現先を-m/--map オプションで指定することができますので、同一サーバ内でバックアップを取得する場合でも、既存のテーブルスペースに影響を与えることなくバックアップを取得することができます。

スクリプトは基本的にはサーバのローカルで実行しますが、実行時に--dr/--dst-remote オプションを付けることでバックアップファイルをリモートに転送することも可能です。

≪バックアップファイルの圧縮≫
スクリプト実行時のオプションでバックアップファイルを圧縮することができます。gzip、bzip2、lzma、lz4、xz の 5 つの圧縮形式をとることができます。

≪リストア機能の有無≫
OmniPITR は取得したバックアップファイルをリストアする機能はありません。
リストアと名の付くもので omnipitr-restore というスクリプトは存在しますが、omnipitr-restore はアーカイブされた WAL ファイルをローカル・リモートどちらでも指定した箇所へ転送するスクリプトで、主にストリーミングレプリケーションのスタンバイサーバで使用するものと想定されます。

≪不要なバックアップの削除≫
omnipitr-backup-cleanup で不要なバックアップの削除が可能です。ただし、このスクリプトを自動実行して定期的にバックアップ削除運用するには、例えば OS のスケジューラや、ジョブ実行管理ツールなどを使用する必要があります。

≪WAL の取得≫
OmniPITR はバックアップ取得時に pg_xlog 配下の WAL についても取得しますが、これらに追加して omnipitr-archive、omnipitr-restore を使用することで、より堅牢な WAL の保存やリストアといった運用を実現することができます。

≪テーブルスペース対応≫
テーブルスペースディレクトリはバックアップの tar ファイル内に保存されます。ファイル解凍時には、バックアップ取得時と同一の絶対パス上にテーブルスペースが再現されますので、同一サーバ内でリカバリ作業をする場合は注意が必要です。

5.6.2. OmniPITR のセキュリティ要件

OmniPITR のスクリプト群を使用する時のセキュリティ要件は表 5.17 のようになります。
なお、omnipitr-archive,omnipitr-restore は CLI 上での直接の使用でなく、設定ファイル内で定義するコマンドとして使用したため今回は評価対象としていません。

© 2016 PostgreSQL Enterprise Consortium

表 5.17: OmniPITR 使用時のセキュリティ要件

項目	OmniPITR		
	設定	対象スクリプト	備考
特定ポート解放の必要性	不要	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	
root ユーザである必要性	不要	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	
認証なしの SSH 接続である必要性	不要	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	
DB の superuser 権限の必要性	必要	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	
pg_hba.conf の trust 認証の必要性	不要	omnipitr-backup-master omnipitr-backup-slave omnipitr-synch	

≪DB の superuser 権限の必要性≫
DB 接続ユーザは superuser 権限が付与されている必要があります。superuser 権限がない場合は以下のようなメッセージが出力され、バックアップは実行されません。

```
$ /home/postgres/omnipitr-master/bin/omnipitr-synch -o postgres@10.1.12.20:/tmp/backup5 -U
barman
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL : Running [show data_directory]
via psql failed: $VAR1 = {
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL :          'stderr' => 'ERROR:
must be superuser to examine "data_directory"
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL : ',
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL :          'status' => 256,
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL :          'stdout' => '',
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL :          'error_code' => 1
2016-03-10 11:02:05.403967 +0900 : 14235 : omnipitr-synch : FATAL :        };
```

5.7. まとめ

5.2 節～5.6 節で紹介してきた内容を基に、以下でそれぞれのツールの使いどころを挙げます。

5.7.1. pg_basebackup でよいケース

pg_basebackup のメリットは PostgreSQL の標準機能であることと言えます。

そのため DB 接続などのセキュリティ要件も標準 PostgreSQL に準拠した形で使用することができ、またツールの情報収集やサポートが日本語で受けられます。

ただし、バックアップの世代管理は標準機能として備わっていないため、複数世代のバックアップを保持したい場合はユーザ側での工夫が必要となります。また、フルバックアップのみに対応しているため、データサイズが膨大な場合はバックアップ取得に時間がかかってしまいます。

以上より、pg_bacebakcp の適用ケースとして例えば以下のような要件の時が挙げられます。

・バックアップ運用のためにセキュリティ要件を緩めたくない
・フルバックアップ 1 世代のみをバックアップとして保持する
・データ量が小さく増分バックアップの必要性があまりない

© 2016 PostgreSQL Enterprise Consortium

5.7.2. pg_rman でよいケース

pg_rman のメリットは、標準機能の pg_basebackup では実現できない増分バックアップやバックアップの世代管理が実現できることです。また、日本の企業が開発しているため日本語の情報が豊富であり、また開発元以外にもツールをサポートしている企業が国内に複数社あるため日本語のサポートを得やすいところもポイントです。

ただし、pg_rman はバックアップ対象の PostgreSQL と同一サーバ内でなければならなく、バックアップファイルも同サーバ内で保持されるためサーバリソースには注意が必要になります。また万が一の DISK の故障に備えておく必要もあります。

以上より、pg_rman の適用ケースとして例えば以下のような要件の時が挙げられます。

・フルバックアップ、増分バックアップを組み合わせた運用をする
・複数世代のバックアップファイルを保持する
・PostgreSQL のバックアップ専用ツールを日本語のサポートがある形で使用する

5.7.3. Barman でよいケース

Barman も pg_rman と同じく増分バックアップやバックアップ世代の管理が可能ですが、Barman としての一番のメリットは、1 つの Barman に対しバックアップ対象 PostgreSQL を複数台設定できることです。また、基本的にリモートから接続してバックアップを取得するため、遠隔地にデータを保存しておくという災害対策の要件を満たすこともできます。

ただし、Barman は開発元が海外の企業であるため取得できる情報のほとんどが英語であり、日本語に訳された情報は圧倒的に少ないです。また、リモートからの接続方法は現時点では認証なしの SSH 接続のみですので、導入する際はこの点のセキュリティも考慮する必要があると考えられます。

以上より、Barman の適用ケースとして例えば以下のような要件の時が挙げられます。

・バックアップ対象が複数台ある
・バックアップファイルを遠隔地に保存する
・複数世代のバックアップファイルを保持する

5.7.4. OmniPITR でよいケース

OmniPITR はバックアップツールではなく、本来は WAL 運用ツールです。OmniPITR のメリットは、WAL のアーカイブ、WAL のリモートへの転送、リストアに不要な WAL の削除といった WAL 運用作業をより多機能に実現できることです。WAL の運用は postgresql.conf や recovery.conf でコマンドを設定しますが、OS の cp コマンドよりも OmniPITR の方がファイルの圧縮、複数個所への転送などより自由度高く運用できます。

ただし OmniPITR は今回紹介した 4 ツールの中で一番情報が少なく、また日本語の情報は皆無に近いです。また増分バックアップやバックアップファイルの世代管理はできません。

以上より、OmniPITR の適用ケースとして例えば以下のような要件の時が挙げられます。

・PITR に備え WAL のアーカイブ運用を行う
・archive_command、restore_command パラメータを OS コマンドで実装するよりも多機能に実装する
・WAL ファイルやバックアップファイルを複数個所に保存する(ローカルとリモート、リモート 1 とリモート 2 など)

© 2016 PostgreSQL Enterprise Consortium

6. パーティションツール

本章では、パーティショニングツールを使うにあたっての要件と課題を整理し、ピックアップしたパーティショニングツールを紹介します。

6.1. パーティショニングに求められる要件

パーティションテーブルを検討する場合の多くは、1つのテーブルが肥大化してしまい、性能劣化や運用性の低下が無視できなくなってきたときがきっかけになると思われます。つまり、パーティショニングの機能を使ってテーブルを分割することで、次のような性能の改善や管理の効率化が要件になると考えます。

性能・拡張性

・テーブルを分割し、対象を局所化することで性能向上を図りたい
・データ量が多くなったとき、容易に領域を拡張したい
・アプリケーション側からパーティションテーブルであることを意識させないようにしたい
・アクセスを分散して、ディスク負荷を低減したい

運用保守・移行性

・壊れるデータを最小限にしたい(影響を局所化する)
・1テーブルのサイズが小さくなることによるバックアップ、リカバリの労力や時間短縮を図りたい

6.1.1. パーティショニングの課題

PostgreSQL ではパーティショニングをテーブルの継承によりサポートしています。それぞれのパーティションは 1 つの親テーブルの子テーブルとして作成されなくてはいけません。子テーブルはそれぞれのパーティションでのキー値を定義するためにテーブル制約が定義されています。

パーティショニングを導入することで 6.1 章の要件に対応することができますが、代わりにパーティショニング特有の課題やデメリットが発生してしまうことも指摘されています。図 6.1 に PostgreSQL の標準機能でパーティショニングを行う場合のパーティション表作成フローを示します。

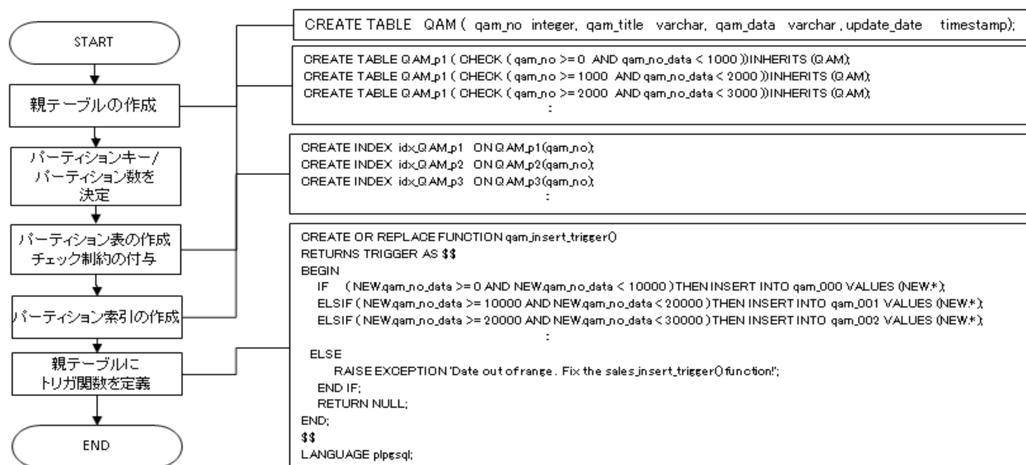


図 6.1: パーティション表作成フロー

PostgreSQL の標準機能でパーティショニングを行う場合、図 6.1 で示したようにパーティション毎にチェック制約の SQL 文が異なることや構造の変化を把握した上で親テーブルトリガーを修正する必要があり、パーティションの管理を行うための作業コストが高いことが課題となっています。

また、PostgreSQL でパーティショニングを使用する際に留意すべき点はマニュアル上では下記のように記載されています。

・全ての CHECK 制約が相互に排他であるかどうか自動で確認する方法はない
・他のパーティションに移動するような UPDATE 文は CHECK 制約により失敗する
・手動 VACUUM, ANALYZE を行っている場合は、それぞれのパーティションで個別に実行が必要
・CURRENT_TIMESTAMP のような非 immutable 関数は最適化できない
・パーティション制約が複雑になると、プランナがパーティションの除外を計画できなくなる
・パーティショニングの分割数は最大 100 までにしないと、プランナのコストがかなり増加する

© 2016 PostgreSQL Enterprise Consortium

これらの課題の中で、WG3 の活動テーマであるエンタープライズ領域で使うにあたっての運用管理に着目すると、手順書としてまとめておけばよいものと、課題を評価して対策を検討する必要があるものが考えられます。
ここでは、後者の観点を中心に、具体的な調査内容は各ツールごとになりますが、大まかには、性能と運用性や保守性のトレードオフのバランスが必要となってくるので、テーブル分割における性能への影響具合と、実際使うにあたっての管理運用面を調査内容としてまとめています。

6.1.2. パーティショニングツールの比較

6.1.1 章で挙がった課題を解決するために、さまざまなパーティショニングツールが存在します。本資料においては、pg_part, pg_partman, pg_shard の 3 つをパーティショニングツールとしてピックアップしました。各ツールの概要について、以下の表 6.1 にまとめます。

表 6.1: パーティショニングツール概要

ツール名	pg_part	pg_partman	pg_shard
最新バージョン	0.1.0	2.2.3	1.2.3
ツール概要	パーティションの作成（＋データの移行）/解除（＋データの移行）/追加/切り離しを簡単に行う関数群を提供する	時系列と連番によるパーティショニングテーブルの作成、管理を行う拡張モジュール	テーブルのシャーディング（パーティショニング）とレプリケーションを同時に実現するツール
ツール種別	SQL ファンクション	EXTENSION モジュール	EXTENSION モジュール
実装方式	SQL	C, SQL, python	C, SQL
対応バージョン(OS, DB などの要件)	－	PostgreSQL 9.4 or greater	OS：Linux, OS X DB： ・PostgreSQL 9.3.4～ ・PostgreSQL 9.4.0～ ・CitusDB 3.2～ GCC：4.6～
ライセンス形態	GPLv2	PostgreSQL License	LGPLv3
開発元	Uptime Technologies	keithf4	Citus Data, Inc.
入手元(URL)	https://github.com/uptimejp/pg_part	https://github.com/keithf4/pg_partman	https://github.com/citusdata/pg_shard
有償サポート	－	－	有(開発元の Citus Data 社、$450/月(年間契約))

6.2. パーティショニングツールの紹介

それでは、6.1.2 章でピックアップしたパーティショニングツールの機能紹介をします。

6.2.1. pg_part

pg_part は、PostgreSQL のパーティショニングの操作をいくつかの SQL 関数として提供しており、PostgrSQL のパーティショニングで必要となる手順を簡略化することができます。図 6.2 に PostgreSQL のパーティション表作成プロセスと pg_part の対応範囲を示します。

© 2016 PostgreSQL Enterprise Consortium

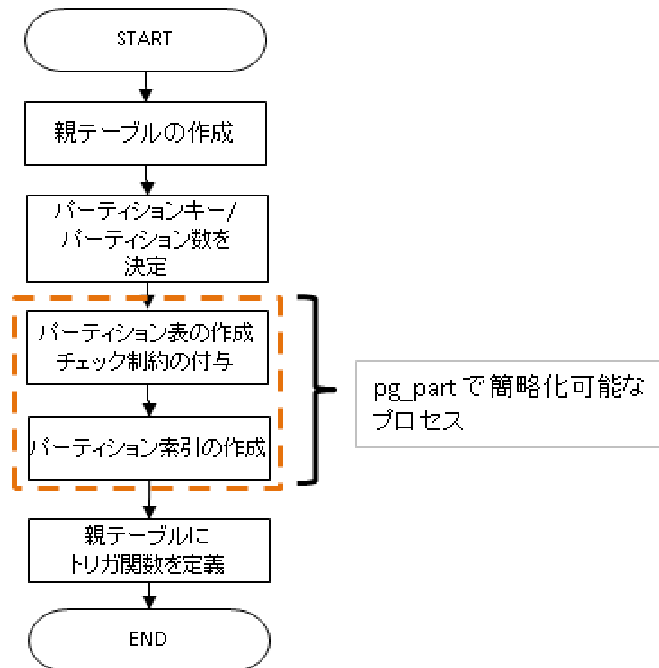


図 6.2: パーティション表作成プロセスとpg_part の対応範囲

pg_part 機能:

pg_part で提供されている 5 つの SQL 関数を表 6.2 に示します。それぞれの関数は複数の SQL 文をまとめて実行する関数となっており、例えばパーティションの作成を行う pg_part.add_partiton() では SQL 関数一つで 図 6.3 のような処理を実行することができます。

表 6.2: pg_part で提供されている関数

関数名	機能
add_partition	パーティションの作成（＋データの移行）
merge_partition	パーティションの解除（＋データの移行）
attach_partition	パーティションの追加（アタッチ）
detach_partition	パーティションの切り離し（デタッチ）
show_partition	パーティションの表示

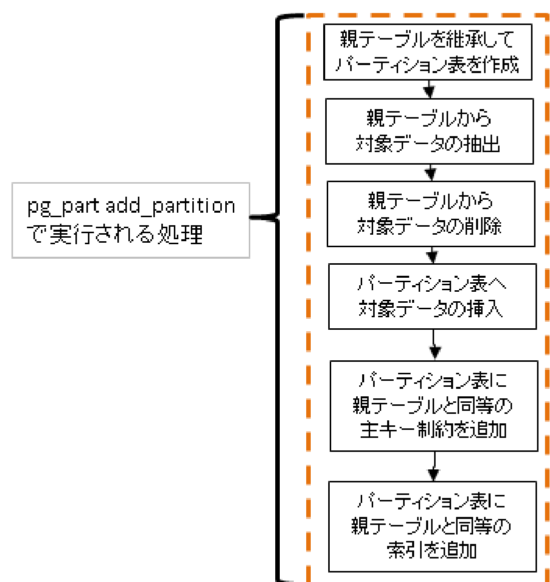


図 6.3: pg_part.add_partition で実行される処理

PostgreSQL でのパーティショニングは、パーティションの作成時には親テーブルと同等の制約、索引を構成する必要があったり、メンテナンス作業として行われるパーティションの追加、削除の処理がパーティションごとに異なる SQL となるなどデータベース管理者のコストが高い作業となります。pg_part により提供されている関数を利用することでデータベース管理者の作業コストを低減することができます。

© 2016 PostgreSQL Enterprise Consortium

6.2.2. pg_partman

pg_partman は、時間ベースと一意な数字ベースでのパーティションの作成と管理が行える拡張モジュールです。また、デフォルトではサポートされていないサブパーティションについてもサポートされています。図 6.4 に PostgreSQL のパーティション表作成プロセスと pg_partman の対応範囲を示します。

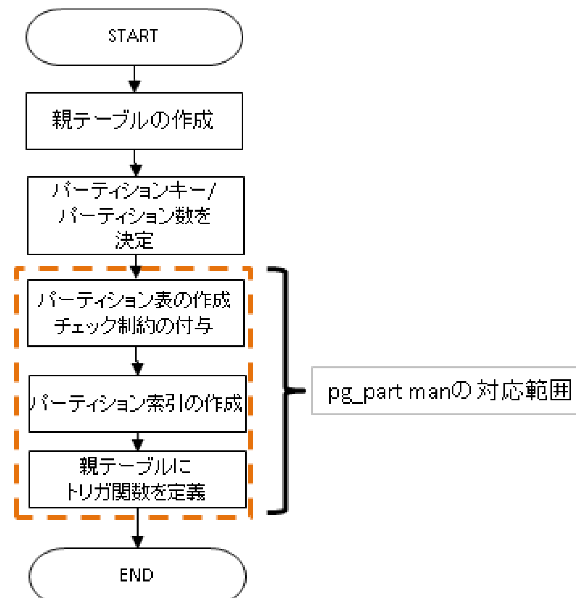


図 6.4: パーティション表作成プロセスと pg_partman の対応範囲

pg_partman ではパーティショニングで必要となる子テーブルの作成や親テーブルへのトリガー生成も自動的に行われ、PostgreSQL のパーティション表で必要となる処理がほぼすべて自動化されています。

<u>パーティションの交換</u>

パーティションの交換は、パーティションを非パーティション表に非パーティション表をパーティション表のパーティションに変換する作業です。

pg_partman で非パーティション表をパーティション表のパーティションに変換する場合は以下の手順で行います。

1.create parent()関数を実行し、指定した表をパーティション表として管理する

2.partition_data_id()関数または partition_data_time()関数を実行し、親テーブルに存在するデータをパーティション表へ移動(該当するパーティション表がない場合は自動的にパーティションが作成)

3.check_parent()関数を実行し、親テーブルにデータが残っていないことを確認する

また、pg_partman でパーティションを非パーティション表に変換する場合は undo_partition_id()関数または undo_partition_time()関数を実行します。

<u>パーティションの追加</u>

パーティションの追加は、パーティション表に新しいパーティションを追加する作業です。

pg_partman では run_maintenance() 関数を定期的に実行することでパーティション表を必要に応じて自動的に追加しています。なお、現在のパーティションセットに含まれないデータが挿入された場合は、データは親テーブルに格納される形となりますが partition_data_id()関数または partition_data_time()関数を実行し該当パーティション表の追加およびデータの移動を行うことになります。

<u>パーティションの削除</u>

パーティションの削除は、パーティション表から特定のパーティションを削除する作業です。

pg_partman では run_maintenance() 関数を定期的に実行することでパーティション表を必要に応じて自動的に削除することができます。パーティションの削除の自動化は part_config 表の retention 列に値を入れることで有効になります。

© 2016 PostgreSQL Enterprise Consortium

6.2.3. pg_shard

pg_shard は、PostgreSQL 用のシャーディングで、水平分散と可用性を実現するツールだが、パーティショニングツールとも言えます。データの分割方法は、PostgreSQL 標準のパーティショニング(レンジパーティショニング、リストパーティショニング)と異なり、ハッシュパーティショニングとレンジパーティショニングによる分割方法です。これらは、コマンドで自動的に設定されるため、利用者が分割の範囲を検討することなく、また、設定漏れや不備、キーの複雑化なども発生しないメリットがあります。さらに、透過的にテーブル走査するためのファンクションやトリガの設定も不要なため、間違いを発生させることなく容易に構築することができることは想像がつくと思います。

(1) 構築面の検証

図 6.5 に pg_shard の構築プロセスを示します。

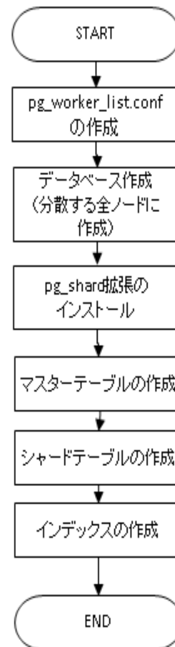


図 6.5: pg_shard 構築プロセス

詳しい環境構築の手順、設定内容については、別紙をご確認ください。次に、環境構築を行った結果を図 6.6 に示します。

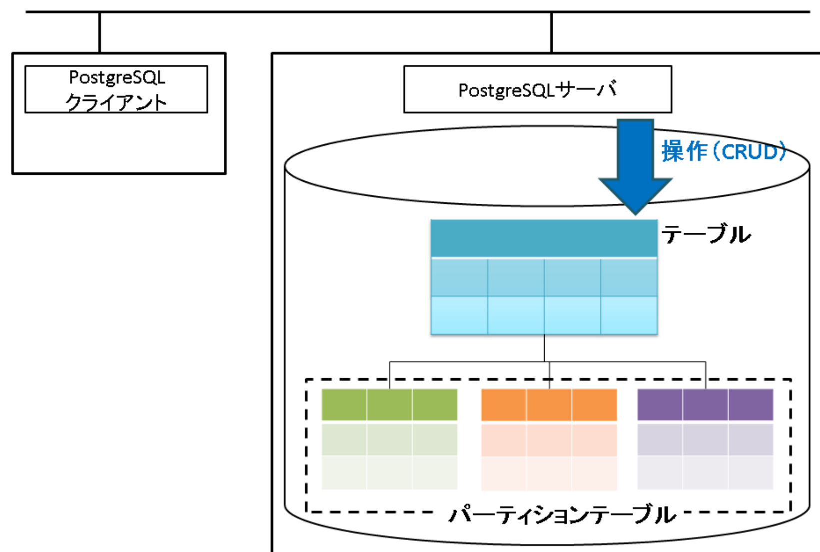


図 6.6: pg_shard イメージ図

© 2016 PostgreSQL Enterprise Consortium

図 6.6 のシングル構成の場合は、メタデータを格納するマスターノードのみで構成されます。複数 DB 構成の場合は、マスターノードとそれ以外のワーカーノードで構成されます。

一方、pg_shard 独自の制約事項があります。環境構築後に動作確認を行った結果、特に劣化が気になる点を以下に列挙しましたが、これらの制約は実運用にあたって厳しい点になると思われます。
・対象が一つに絞り込めないと、更新（UPDATE 文）や削除（DELETE 文）はできません。空化（TRUNCATE 文）もできません。また、更新対象が分割キーで他のパーティションに移動するような UPDATE 文もエラーとなり処理されません。
・EXPLAIN 文が使えません（auto_explain を使用する）
・COPY 文が使えません（代わりにスクリプト「copy_to_distributed_table」が提供されています）
・親テーブルの変更（ALTER TABLE 文）しても、子テーブルに処理が伝播せず相違が出ます
・親テーブルを削除（DROP TABLE 文）しても、子テーブルは削除されません
・「INSERT INTO … SELECT ～」のようなクエリはサポートされません

(2) 運用面の検証
(a) 性能検証
6.1.1 章の課題で挙げたように、パーティションの分割数が性能に影響があると言われています。pg_shard の場合もパーティション分割数がどの程度の影響が出るのか、実際に性能検証を実施しました。
検証環境のハードウェア環境を表 6.3 に、ソフトウェア環境を表 6.4 に示します。なお、pg_shard の gcc のバージョンの必須要件により、OS は CentOS バージョン 7 を使用しました。

表 6.3: ハードウェア環境

仮想環境	Microsoft Azure
インスタンス	Standard D3
CPU	Intel(R) Xeon(R) CPU E5-2660 2.20GHz @ 4core
RAM	14GB
Disk	SSD 200GB

表 6.4: ソフトウェア環境

OS	CentOS 7.2.1511
PostgreSQL サーバ	PostgreSQL 9.4.5(PGDG)
pg_shard	1.2.3
gcc	4.8.5

PostgreSQL の設定は、メモリまわりと、特に書き込み性能の向上のため、checkpoint まわりと synchronous_commit パラメータを設定しています（pg_shard クイックスタートガイドより）（図 6.7）。

```
synchronous_commit = off
checkpoint_segments = 64
checkpoint_timeout = 30min
checkpoint_completion_target = 0.8
checkpoint_warning = 30min
```

図 6.7: postgresql.conf の設定（抜粋）

その他のパラメータの設定内容は、別紙をご確認ください。なお、synchronous_commit の影響については、検証内容 2 にて検証を行っています。

検証を実施するにあたり、表 6.5 のテーブルを作成し、pg_shard によるパーティションテーブルとしました（primary key の設定は検証内容 3 の検索性能の検証時に設定）。なお、このとき使用したクエリは、別紙をご確認ください。

© 2016 PostgreSQL Enterprise Consortium

表 6.5: pgbench_accounts

aid	bigint not null	primary key, pg_shard キー
bid	int	
abalance	int	
filler	char(84)	

検証内容 1:
pg_shard では、一括挿入として copy_to_distributed_table が準備されています。マニュアルでは、データロードのパラレル化により性能向上が図られるといった記述があるため、パラレル化による影響を検証しました。ここでは、パーティション分割数の影響を避けるため、分割数を 1 に設定しています。
実行するスクリプトを図 6.8 に示します。データロードするファイルは accounts.csv で、中身は'|'区切りのデータが 1,000 万件入っています。

```
#!/bin/sh
mkdir chunks
split -n l/ パラレル数 accounts.csv chunks/
find chunks/ -type f | xargs -n 1 -P パラレル数 sh -c 'echo $0 `copy_to_distributed_table
        -C -c “utf8” -d “|” -n NULL $0 pgbench_accounts`' （改行なし）
rm -rf chunks
```

図 6.8: 検証スクリプト 1

検証結果 1:
パラレル化によって、挿入性能はリニアに向上したので、copy_to_distributed_table はパラレル化に対して有効に機能していることがわかります。また、CPU 数以上は性能はほぼ一定だったので、CPU 数でパラレル化するのが有効であることが確認できました。

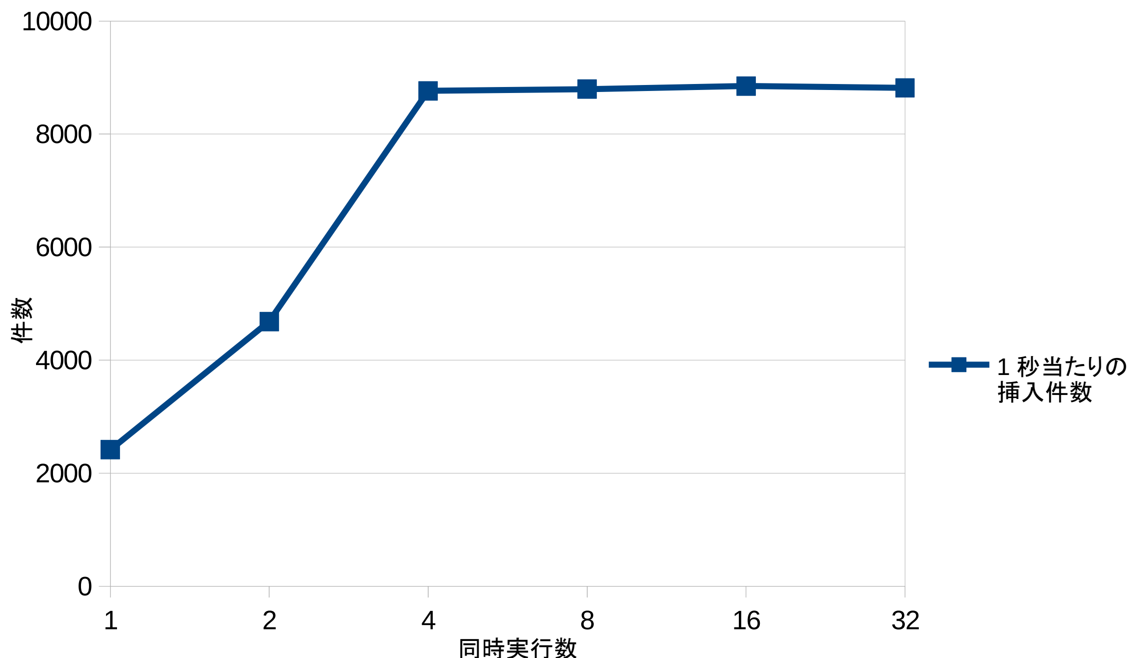


図 6.9: copy_to_distributed_table による挿入性能

パラレル化によりデータの挿入順は不定になることが注意点ですが、本来 ORDER BY を指定しない場合は取り出し順序は不定となっているので、問題はないと思われます。

© 2016 PostgreSQL Enterprise Consortium

検証内容 2:

次に、パーティション分割数による挿入性能への影響を検証するため、copy_to_distributed_table による 1,000 万件の挿入性能を検証しました。併せて、synchronous_commit パラメータによる影響も検証しています。

実行するスクリプトを図 6.10 に示します。データロードするファイルは accounts.csv で、中身は'|'区切りのデータが 1,000 万件入っています。

```
#!/bin/sh
mkdir chunks
split -n l/4 accounts.csv chunks/
find chunks/ -type f | xargs -n 1 -P 4 sh -c 'echo $0 `copy_to_distributed_table
        -C -c "utf8" -d "|" -n NULL $0 pgbench_accounts`' (改行なし)
rm -rf chunks
```

図 6.10: 検証スクリプト 2

検証結果 2:

pg_shard においても、図 6.11 からわかるように、分割数が増えるにつれてリニアに性能劣化が見られました（分割数 100 で 4 割程度劣化）。ただし、分割数 100 を超えた時点で極端に性能が悪化するといった現象は確認できませんでしたので、分割数 100 にこだわることなく、なるべく分割数の少ない環境を構築すべきです。

もうひとつの検証ポイントであった synchronous_commit パラメータの影響は、明らかにありました。COPY や insert の多い更新系システムで使用する場合には、synchronous_commit の設定を off にすることを検討すべきです。

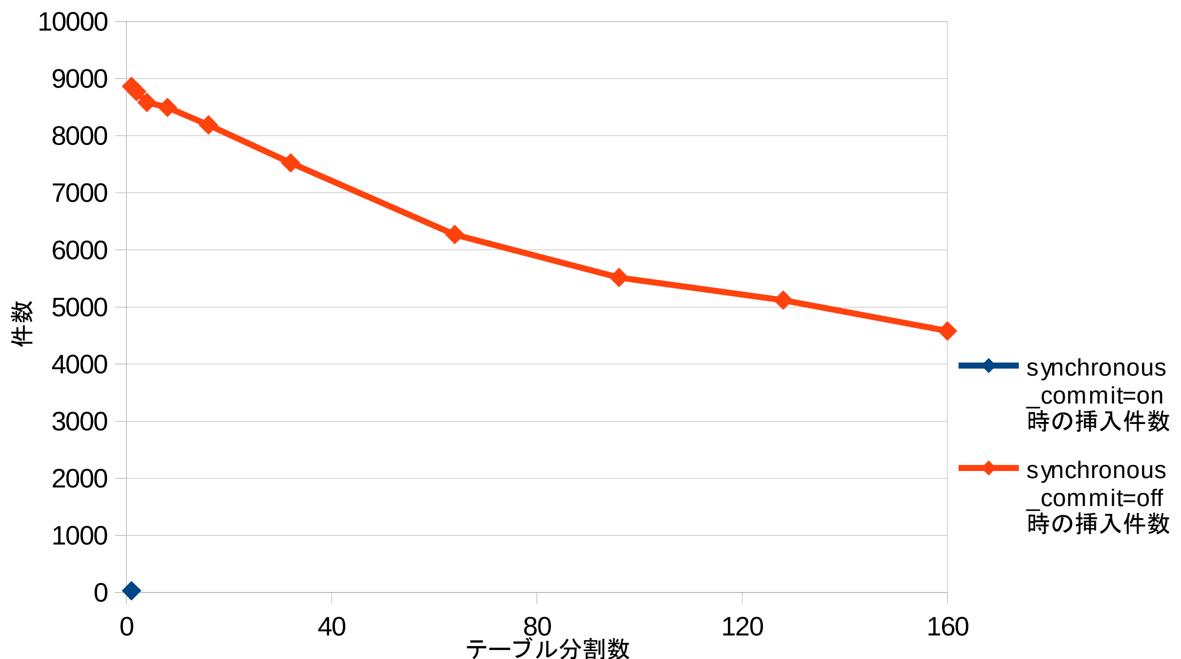


図 6.11: パーティションの分割数の影響（挿入性能）

© 2016 PostgreSQL Enterprise Consortium

検証内容 3:

今度は、パーティション分割数による検索性能への影響を検証する為、pgbench による検索性能を測定しました。pgbench のスケールファクタは 100 に設定し、以下のコマンドを実行しました。

```
$ pgbench -h 10.0.0.1 -p 5432 -c 32 -j 16 -T 60 -s 100 -n -f custom.sql pgbench
```

ベンチマークシナリオは、図 6.12 のカスタムシナリオを実行していますが、内容は'-S'を付与したデフォルトのシナリオと同等の内容です。

```
\set naccounts 100000 * :scale
\setrandom aid  1 :naccounts

SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

図 6.12: pgbench によるベンチマークシナリオ(custom.sql)

※なお、検証内容 3 を実施する前に、マスタテーブルと全てのパーティションテーブルに対して、vacuum と primary key 付与を実施しています。

検証結果 3:

検索性能も図 6.13 からわかるように、分割数が増えるにつれてリニアに性能劣化が見られました(分割数 100 で 35%程度劣化)。検証結果 2 と同様に、分割数 100 を超えた時点で極端に性能が悪化するといった現象は確認できませんでしたので、なるべく分割数の少ない環境を構築すべきです。

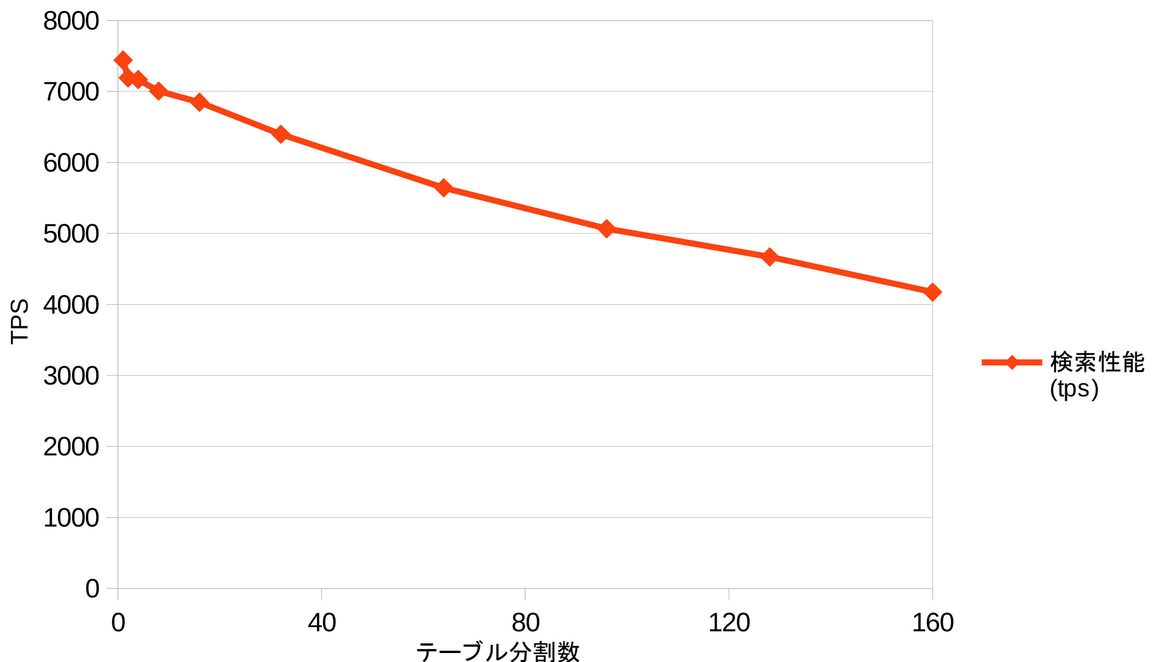


図 6.13: パーティションの分割数の影響(検索性能)

© 2016 PostgreSQL Enterprise Consortium

(b) 管理運用検討

6.1.1 章で挙げたもうひとつの観点としては、実際に使うにあたっての管理運用面がエンタープライズ領域で利用するポイントになります。パーティショニングの管理にあたって、検討が必要な点をまとめました。なお、ここで実施する内容は、更新、参照がないメンテナンス時間に行ってください。

(i) 障害対応について

・**マスターノード障害**（シングル構成ではマスターノードのみ）

マスターノードは、pg_shard のメタデータを格納しているので、可用性要求に応じてどのような構成にするか検討が必要です。

高い可用性が要求される場合は、ストリーミングレプリケーションなどを使用して、レプリケーション環境を構築します。要求レベルがそこまでではない場合は、pg_dump やクラウドサービスのボリュームデータのスナップショットでバックアップを定期的に取得しておき、障害が発生した際にはリカバリできるようにしておきます（pg_dump の仕方については、後述します）。

・**ワーカーノード障害**

ワーカーノードは、パーティションデータを冗長で持ち、あるパーティションに障害が発生した場合でも、自動的に障害を検出し、別のパーティションを見に行くことで問い合わせを可能にします。

その冗長数は、初期設定の master_create_worker_shards で指定するノード数になりますので、設計時にレプリカをいくつ取得する必要があるか検討が必要です。

・**パーティション修復**

パーティションが何らかの形でエラーになりデータの更新が止まった場合、そのパーティションの修復が必要になります。その場合、master_copy_shard_placement でパーティションを修復することができます。

(ii) パーティション管理について

・**ノード数の追加、削除**

処理の分散のために、ノードの追加、または、削除したいといった要求があるかもしれません。そのような場合、次の方法で追加、削除をすることが可能です。

追加方法：

1．pgs_distribution_metadata.shard_placement_id_sequence からパーティション数分の次のシーケンス値を取得します。

```
pgbench=# select nextval('pgs_distribution_metadata.shard_placement_id_sequence');
 nextval
---------------
        3
```

2．pgs_distribution_metadata.shard_placement テーブルに新しいノードのパーティションのエントリを追加します。id は取得したシーケンス値を使用します。

```
pgbench=# insert into pgs_distribution_metadata.shard_placement values (3, 10000, 1, '10.0.0.2', 5432);
pgbench=# insert into pgs_distribution_metadata.shard_placement values (4, 10001, 1, '10.0.0.2', 5432);
```

```
pgbench=# select * from pgs_distribution_metadata.shard_placement;
 id  | shard_id | shard_state | node_name  | node_port
--------------------------------------------------------------------------------
  1 |    10000  |              1  | 10.0.0.1     |        5432
  2 |    10001  |              1  | 10.0.0.1     |        5432
  3 |    10000  |              1  | 10.0.0.2     |        5432
  4 |    10001  |              1  | 10.0.0.2     |        5432      ) 追加分
```

3．パーティションテーブルのデータを取得します

既存のパーティションテーブルの論理バックアップを取得します。pg_dump コマンドでバックアップする場合、以下

© 2016 PostgreSQL Enterprise Consortium

のようにします。

```
$ pg_dump pgbench -t pgbench_accounts_* -f output.sql
```

4. 追加するノードに同名のデータベースを作成します。

```
$ createdb pgbench
```

5. psql コマンドでデータ挿入を行います。必要な場合は、SQL ファイルの中身を修正してください。

```
$ psql pgbench -f output.sql
```

削除方法:
1. pgs_distribution_metadata.shard_placement テーブルから、削除したいノードのエントリを削除します。

```
pgbench=# delete from pgs_distribution_metadata.shard_placement where node_name = '10.0.0.2';
```

```
pgbench=# select * from pgs_distribution_metadata.shard_placement;
 id | shard_id | shard_state | node_name | node_port
--------------------------------------------------------------------------------
  1 |  10000 |            1 | 10.0.0.1    |      5432
  2 |  10001 |            1 | 10.0.0.1    |      5432
```

2. 削除ノードのテーブルを削除します。

```
pgbench=# drop table pgbench_accounts_10000;
pgbench=# drop table pgbench_accounts_10001;
```

・パーティションのアタッチ、デタッチ

パーティションテーブルのサイズの偏りなどで性能への影響が発生した場合には、パーティションを分割、もしくは、統合したいといった要求があるかもしれません。その場合、次の方法でアタッチ、デタッチすることが可能です。

アタッチ方法:
1. pgs_distribution_metadata.shard_id_sequence テーブルから次のシーケンス値を取得します。

```
pgbench=# select nextval('pgs_distribution_metadata.shard_id_sequence');
 nextval
---------------
10002
```

2. pgs_distribution_metadata.shard テーブルに新しいパーティションのエントリを追加します。
id は取得したシーケンス値を使用します。また、ハッシュ値の範囲を−2147483648～2147483647 の範囲で抜け漏れがないように設定します。
以下の例は、id=10001 のパーティションの範囲を−1～1000000000 に変更し、追加した id=10002 のパーティション

```
pgbench=# update pgs_distribution_metadata.shard set max_value = 1000000000 where id = 10001;
pgbench=# insert into pgs_distribution_metadata.shard values (10002, 16542, 't', 1000000001, 2147483647);
```

```
pgbench=# select * from pgs_distribution_metadata.shard;
  id    | relation_id | storage | min_value      | max_value
--------------------------------------------------------------------------------
10000 |        16542 | t          | -2147483648 | -2
10001 |        16542 | t          | -1               | 1000000000
10002 |        16542 | t          | 1000000001  | 2147483647
```

3. pgs_distribution_metadata.shard_placement_id_sequence から次のシーケンス値を取得します。

© 2016 PostgreSQL Enterprise Consortium

```
pgbench=# select nextval('pgs_distribution_metadata.shard_placement_id_sequence');
 nextval
---------------
        3
```

４. pgs_distribution_metadata.shard_placement テーブルに新しいパーティションのエントリを追加します。id は取得したシーケンス値を使用します。

```
pgbench=# insert into pgs_distribution_metadata.shard_placement values (3, 10002, 1, '10.0.0.1', 5432);
```

```
pgbench=# select * from pgs_distribution_metadata.shard_placement;
  id  |  shard_id | shard_state | node_name  | node_port
-----------------------------------------------------------------------------------
   1 |    10000  |              1  | 10.0.0.1       |        5432
   2 |    10001  |              1  | 10.0.0.1       |        5432
   3 |    10002  |              1  | 10.0.0.1       |        5432
```

５. テーブルを作成します

テーブル名は「マスターテーブル_*取得したシーケンス値*」、テーブル構造はマスターテーブルと同じにします（表 6.6）。

表 6.6: pgbench_accounts_10002

aid	bigint not null	primary key
bid	int	
abalance	int	
filler	char(84)	

６. 分割するパーティションテーブルのデータを取得します

パーティションテーブルの論理バックアップを取得します。copy_to_distributed_table でデータ挿入を行うので、今回は、psql で pgbench_accounts_10001 のデータを'|'区切りで取得する場合の例を以下に示します。この時、取得したデータに不要な行があるので削除しておきます。

```
$ psql pgbench -c"select * from pgbench_accounts_10001" -A -F'|' > dump.sql
```

７. 分割するパーティションテーブルのデータを削除します。

```
pgbech=# delete from pgbench_accounts_10001;
```

８. copy_to_distributed_table でマスターテーブルに対して、データ挿入します。

```
$ copy_to_distributed_table -C -d '|' -n NULL dump.sql pgbench_accounts
```

デタッチの方法：

１. pgs_distribution_metadata.shard_placement テーブルにパーティションのエントリを削除します。

```
pgbench=# delete from pgs_distribution_metadata.shard_placement where id = 2;
```

2. pgs_distribution_metadata.shard テーブルでパーティションのエントリを削除します。

ハッシュ値の範囲を-2147483648～2147483647 の範囲で抜け漏れ無いように設定します。

```
pgbench=# delete from pgs_distribution_metadata.shard where id = 10001;
```


© 2016 PostgreSQL Enterprise Consortium

```
pgbench=# select * from pgs_distribution_metadata.shard;
  id    | relation_id | storage | min_value     | max_value
--------------------------------------------------------------------------------
10000 |        16542 | t          | -2147483648 | -2
10002 |        16542 | t          | -1                  | 2147483647
```

3. 削除するパーティションテーブルのデータを取得します
 パーティションテーブルの論理バックアップを取得します。copy_to_distributed_table でデータ挿入を行うので、今回は psql で pgbench_accounts_10001 のデータを'|'区切りで取得する場合の例を示します。この時、取得したデータに不要な行があるので削除しておきます。

```
$ psql pgbench -c"select * from pgbench_accounts_10001" -A -F'|' > dump.sql
```

4. copy_to_distributed_table でマスターテーブルに対して、データ挿入します

```
$ copy_to_distributed_table -C -d '|' -n NULL dump.sql pgbench_accounts
```

(iii) データの移行について

移行方法として、物理バックアップを使用する方法と論理バックアップを使用する方法がありますので、それぞれ紹介します。

・物理バックアップ
pg_basebackup を使った方法を紹介します。

(a) バックアップ
 pg_basebackup で、マスターノードとワーカーノードの物理バックアップを取得します。なお、ワーカーノードのデータは、マスターノードから復旧可能なので、最低限マスターノードのデータをバックアップします。

```
$ pg_basebakup -h 10.0.0.1 -p 5432 -D /tmp/backup --xlog --verbose
```

(b) リカバリ
全く同じ場所、同じ構成でリカバリするのであれば、変更する必要はありません。違う場合は、pgs_distribution_metadata.shard_placement テーブルを修正してください。

```
pgbench=# select * from pgs_distribution_metadata.shard_placement;
  id  | shard_id | shard_state | node_name  | node_port
--------------------------------------------------------------------------------
   1 |    10000  |               1  | 10.0.0.2       |         5432
   3 |    10002  |               1  | 10.0.0.2       |         5432
```

・論理バックアップ
pg_dump を使った方法を紹介します。
移行先のパーティション分割数を変更するかどうかで、移行方法が 2 パターンあります。

・<u>パーティション分割数が同じ場合</u>
(a) バックアップ
pg_dump コマンドでバックアップする場合は、パーティションテーブルのデータのみを論理バックアップで取得します。

```
$ pg_dump pgbench --data-only -t pgbench_accounts_* -f output.sql
```

この時のハッシュ値の範囲を確認しておきます。

© 2016 PostgreSQL Enterprise Consortium

```
pgbench=# select * from pgs_distribution_metadata.shard;
  id    | relation_id | storage | min_value     | max_value
---------------------------------------------------------------------------------
10000 |        16542 | t         | -2147483648 | -2
10001 |        16542 | t         | -1                 | 2147483647
```

(b) リカバリ

移行先でパーティションテーブルを再作成したのち、ハッシュ値の範囲が同じ範囲であることを確認します。もし、違う場合は編集してください。

その後、psql コマンドでデータ挿入を行います。パーティションテーブルの名称が異なる場合は、SQL ファイルの中身を修正してください。

```
$ psql pgbench -f output.sql
```

・<u>パーティション分割数を変更したい場合</u>

ここでのパーティション分割数の変更は、アタッチ／デタッチのレベルではなく、新しく構築し直したい場合です。

(a) バックアップ

パーティション分割数と同じ場合と同様に、論理バックアップを取得します。copy_to_distributed_table でリカバリを行うので、今回は、psql で pgbench_accounts のデータを'|'区切りで取得する場合の例を示します。この時、取得したデータに不要な行があるので削除しておきます。

```
$ psql pgbench -c”select * from pgbench_accounts” -A -F'|' > output.sql
```

(b) リカバリ

移行先でパーティションテーブルを再作成したのち、copy_to_distributed_table でマスターテーブルに対して、データ挿入します。

```
$ copy_to_distributed_table -C -d '|' -n NULL output.sql pgbench_accounts
```

6.3. まとめ

各パーティショニングツールの使いどきをまとめます。

6.3.1. pg_part の使いどき

PostgreSQL はパーティショニング専用の組み込み機能を持たず、テーブル継承や CHECK 制約などを組み合わせて実現しているため、パーティショニング操作を行うためにはそれぞれの親テーブルおよび継承された子テーブルの構造を把握した形でメンテナンスを行う必要があります。パーティションメンテナンス作業が頻繁に行われるようなシステムではパーティション操作を pg_part 関数を用いることで運用コストを低減することができます。

6.3.2. pg_partman の使いどき

pg_partman ではパーティショニング操作をほぼ自動化することができます。このためパーティショニングが前提となるような大規模表やメンテナンスそのものも自動化する必要があるシステムで利用する価値があります。

デフォルトの機能では実装されていないサブパーティションも構成することができるため、DWH や BI のような大規模テーブルへの参照が必要な環境での利用も考えられます。

6.3.3. pg_shard の使いどき

pg_shard は複数 DB の分散パーティション環境の構築が容易に実現でき、CHECK 制約の排他確認、パーティション制約の複雑性などの留意点も払拭されるので、作業コストの低減が図れます。

ただ、更新系システムの場合は、PostgreSQL 標準機能のパーティショニングと比べて制約や性能劣化の影響が大きく、パーティション構成変更といった運用コストも大きくなりがちなので、一度作ったらそのまま運用するような DWH や BI といった参照系のシステムで利用するのが適している、といった印象を受けました。また、分割数の性能への影響が大きいので、最大 100 で考えるのではなく 1 つでも少ない分割数で構築するように検討すべきです。

なお、PostgreSQL の標準機能である FDW（Foreign Data Wrapper）の機能が強化されており、バージョン 9.5 より分散パーティション環境の構築ができるようになりました。この部分は、現在も改善、拡張が進んでいる部分なので、併せて今後の拡張を期待していきたいと考えています。

© 2016 PostgreSQL Enterprise Consortium

7. 実行計画制御

本章では、PostgreSQLの実行計画を制御する運用ツールについて説明します。

アプリケーションで発行したSQLはPostgreSQL内部で、パーサによる構文解析→リライタによるルール書き換え→プランナによる実行計画作成→オプティマイザによる最適化→エグゼキュータによる処理という順序で実行されます。

また、PostgreSQLではテーブルの行数・値の分布などを統計情報として管理しており、プランナでは、この統計情報とSQLにもとづいた実行計画を作成します。

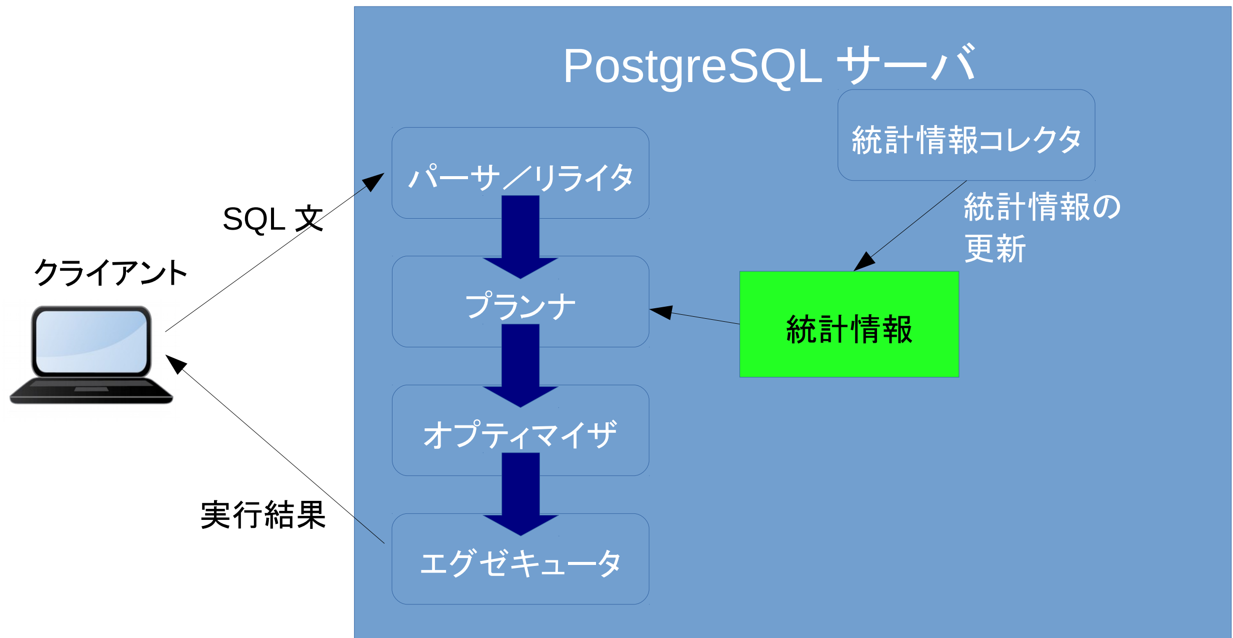


図 7.1: PostgreSQLのSQL文処理の流れ

実行計画制御の運用ツールは、上記のフェーズのうち、実行計画作成/最適化の処理フェーズに介入して、PostgreSQLの実行計画を制御します。

© 2016 PostgreSQL Enterprise Consortium

7.1. PostgreSQL の実行計画

PostgreSQL では、テーブル内に格納された情報をサンプリングした統計情報を元に、発行した SQL 文の実行計画を自動的に作成します。そして、作成した実行計画に従った処理を行います。

1 つの SQL 文を実行するときに、PostgreSQL の内部では様々な組み合わせの実行計画が作成されます。

たとえば、1 つのテーブルに対して検索を行う場合、インデックス使用の要否、あるいは複数のインデックスが存在した場合、どのインデックスを使って作成するのか、といった選択肢が存在します。

さらに複数のテーブルを結合して検索を行う場合には、結合の方式(入れ子ループ結合、マージ結合、ハッシュ結合)の選択肢が発生します。さらに 3 つ以上のテーブルを検索する場合には、結合の順序の選択も必要になります。

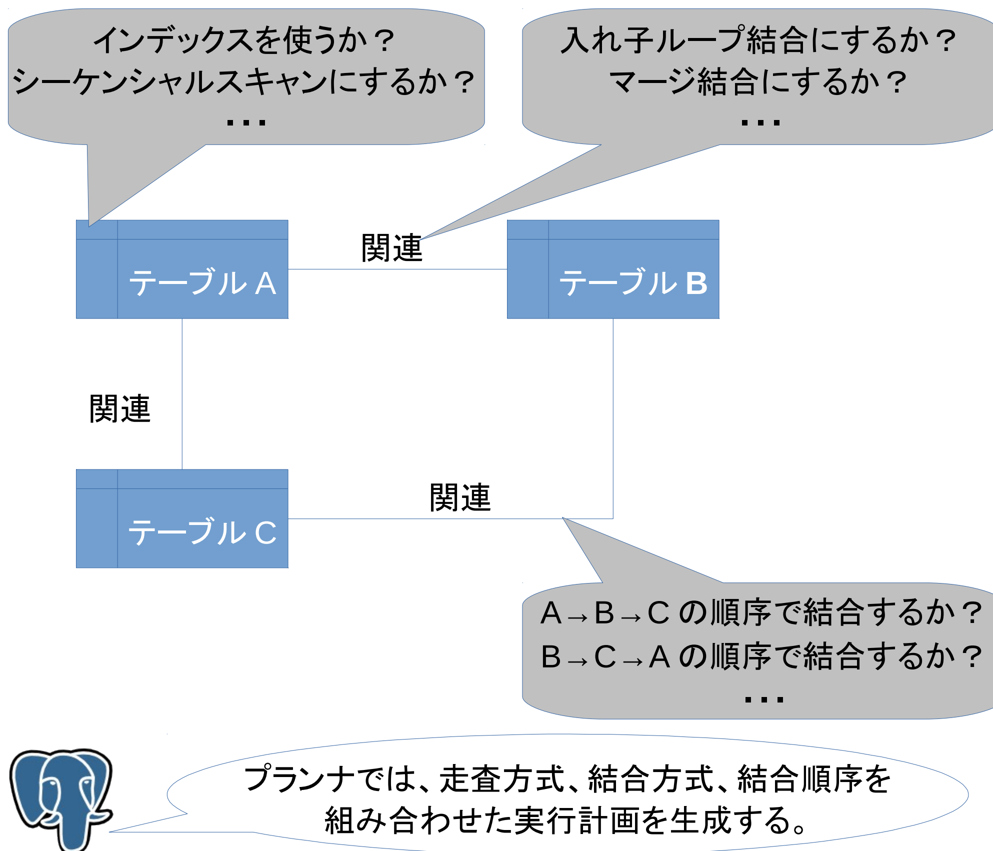


図 7.2: 実行計画の組み合わせ

PostgreSQL ではこうした実行計画として考えられる組み合わせを全て作成した上で、各実行計画の「コスト」を算出します。そして、もっとも「コスト」が低いものを、実行計画として選択します。[3]

3　SQL 文で使用されるテーブル数が非常に多い場合には、「遺伝的問い合わせ最適化(GEQO)」アルゴリズム(http://www.postgresql.jp/document/9.4/html/geqo.html)が使用されることがあります。

© 2016 PostgreSQL Enterprise Consortium

7.1.1. 実行計画制御が必要となるケース

通常の用途では、実行計画の制御はプランナ/オプティマイザに任せて基本的には問題ありません。しかし、要件によっては、PostgreSQL が自動的に選択する実行計画ではなく、別の実行計画を強制的に選択させて実行したほうが良いケースもありえます。

例えば、非常に複雑な SQL 文では、プランナ/オプティマイザが常に最適な実行計画を選択しないケースがありえます。また、環境によっては、プランナ/オプティマイザが選択された結合方式よりも、別の結合方式が適している場合(たとえば、ハッシュ結合よりマージ結合のほうがコストは低いが、実際に実行するとハッシュ結合を選択したほうが処理時間が短い場合など)もありえます。

実行計画制御のツールはこうしたケースで有効となるものです。

7.1.2. PostgreSQL 自体が持つ実行計画制御機能

PostgreSQL 自体も実行計画を制御する機能を持っています。
PostgreSQL における実行計画の制御は、PostgreSQL の設定パラメータを使って行います。
実行計画に関連する PostgreSQL のパラメータには大別して 2 種類あります。

・プランナメソッド設定
・プランナコスト定数

プランナメソッド設定は、特定のプランナメソッド(インデックス検索、結合方式)を有効/無効にするパラメータです。

表 7.1 プランナメソッド設定(PostgreSQL 9.4)

パラメータ名	対象となるプランナメソッド	備考
enable_bitmapscan	ビットマップスキャン	
enable_hashagg	ハッシュ集約	
enable_hashjoin	ハッシュ結合	
enable_indexscan	インデックススキャン	
enable_indexonlyscan	インデックスオンリースキャン	
enable_material	具体化	完全には禁止できない。
enable_mergejoin	マージ結合	
enable_nestloop	入れ子ループ結合	完全に禁止はできない。
enable_seqscan	シーケンシャルスキャン	完全に禁止はできない。
enable_sort	明示的並び替え手順	完全に禁止はできない。
enable_tidscan	TID スキャン	

デフォルトは全てのプランナメソッドは有効になっており、実行計画を制御する場合には、特定のプランナメソッドを無効にします。

プランナメソッド設定のパラメータは大雑把にプランを制御することはできますが、特定のテーブルにアクセスするときのプランナメソッドの無効化などの細かい制御はできません。また、インデックスの設定や SQL 文の内容によっては、無効にしたプランナメソッドでも実行計画として使われるケースもあります。

また、PostgreSQL 設定ファイル上でプランナメソッド設定パラメータを変更すると、その効果は PostgreSQL データベースクラスタ全体に波及します。このため、特定の SQL 文のみプランナメソッド設定パラメータを変更する場合には、セッション内で SET 文を使って変更することが推奨されます。

© 2016 PostgreSQL Enterprise Consortium

プランナコスト定数は、プランナ/オプティマイザ内で行われる、コストの算出の係数となるパラメータです。プランナメソッド設定とは異なり、直接的にプランを変更するものではなく、実行計画の作成や選択はプランナ/オプティマイザに委ねられます。

表 7.2: プランナコスト定数(PostgreSQL 9.4)

パラメータ名	説明	備考
seq_page_cost	シーケンシャルスキャンにおけるディスクページ取り出し時の推定コスト。	他のプランナコスト定数のパラメータは、このパラメータに対する相対的な処理コストを示す。 特定のテーブル空間に対して設定可能。
random_page_cost	非シーケンシャルスキャンにおけるディスクページ取り出し時の相対的な推定コスト。	特定のテーブル空間に対して設定可能。
cpu_tuple_cost	問いあわせ時のそれぞれの行の処理に対する相対的な推定コスト。	
cpu_index_tuple_cost	インデックススキャン間にそれぞれのインデックス行の処理に対する相対的な推定コスト。	
cpu_operator_cost	問い合わせ時に実行される各演算子や関数の処理に対する相対的な推定コスト。	
effective_cache_size	インデックスを使用するコスト推定値の要素。この値を高くすればインデックススキャンが、低くすればシーケンシャルスキャンが選択されやすくなる。	

プランナメソッド設定の変更より、プランナコスト定数の変更を行うことを、PostgreSQL 書では推奨しています。
しかし、プランナコスト定数の変更には、実行計画の作成に対する高度な知識が要求され、またプランナコスト定数を調整するためのツールもないため、試行錯誤しながらプランナコスト定数を調整していく必要があります。
このため、PostgreSQL 内部の挙動に熟知したデータベース管理者であっても、このプランナコスト定数を使ったチューニングを行うことは現状困難です。

本章で紹介する実行計画制御ツールは、この実行計画の制御を比較的簡単にサポートする位置づけのツールです。

© 2016 PostgreSQL Enterprise Consortium

7.2. ツール紹介

本節では、実行計画を制御する「pg_hint_plan」とプランナが使用する統計情報を固定化する「pg_dbms_stats」の 2 つのツールを紹介します。

7.2.1. pg_hint_plan

pg_hint_plan は実行計画を示すヒントを SQL 文に指定することで、SQL 文や GUC パラメータを変えずに実行計画を制御することができるツールです。

具体的には、実行計画を制御するためのヒントを SQL 文のコメントとして記述することで、プランナに介入し、実行計画作成時に特定の実行計画を選択させることができます。

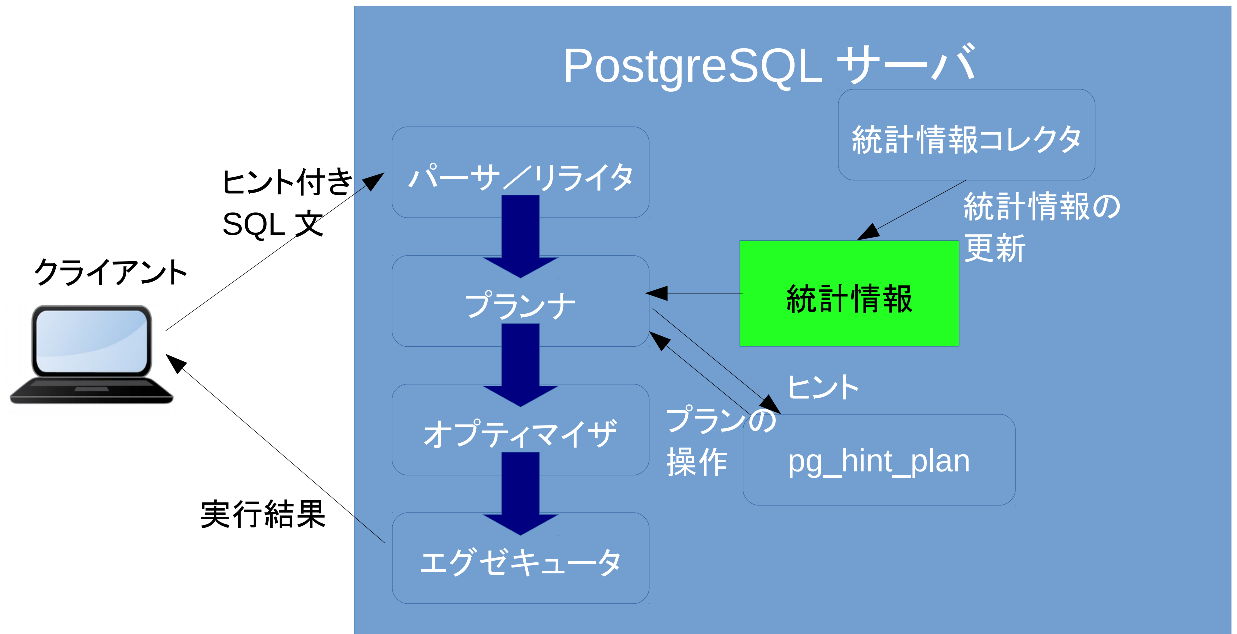


図 7.3: pg_hint_plan の概要

pg_hint_plan では以下のヒントを SQL 文の実行時に与えることができます。
・スキャン方式
・結合方式
・結合順序/結合方向
・見積もり件数
・SQL 文実行時のみ変更する PostgreSQL パラメータ

また、ヒント用のテーブルに事前にヒントを登録することもできます。
詳細については、pg_hint_plan のドキュメントを参照してください[4]。

4 http://pghintplan.osdn.jp/pg_hint_plan-ja.html

© 2016 PostgreSQL Enterprise Consortium

pg_hint_plan は、SQL 文コメントの中に書かれた特定の記法のヒント情報を解釈します。
ヒントを記述した SQL 文の例を以下に示します。

```
/*+
  SeqScan(a)
  HashJoin(a b)
  Set(random_page_cost 2.0)
 */
SELECT * FROM pgbench_accounts a JOIN pgbench_branches b ON a.bid = b.bid
 ORDER BY a.aid LIMIT 10;
```

この例は、a というテーブル(またはその別名)に対して、シーケンシャルスキャンを行わせ、かつ、a と b というテーブル(またはその別名)に対して、ハッシュ結合を行うプランを選択させます。
また、この SQL 文の実行中の PostgreSQL のプランナコスト定数 random_page_cost に 2.0 という値を設定させます。

ただし、現状、pg_hint_plan では誤ったコメントを記述した場合、その誤りに対するエラー報告を行わず、そのヒントを無視して PostgreSQL の実行計画作成に委ねてしまいます。
また、元々作成されない実行計画を選択させることはできません。例えば、インデックスを設定しない列に対して検索を行う SQL 文では、インデックススキャンを行うヒントを記述してもインデックス検索は行われません。この場合も、特に検索時にエラーにはならず、そのヒントは無視されます。

pg_hint_plan を使う場合には、記述したヒントによって意図した実行計画になっているか、十分な検証を行う必要があります。

© 2016 PostgreSQL Enterprise Consortium

7.2.2. pg_dbms_stats

pg_dbms_stats は、統計情報を制御することにより、間接的に実行計画を制御するツールです。
pg_dbms_stats では、事前に利用者が最適と判断した実行計画を作成する統計情報を保存しておき、データの挿入や更新によって件数や値の分布が変動した場合でも、保存された統計情報を保護することで、一定の実行計画を作成させることができます。

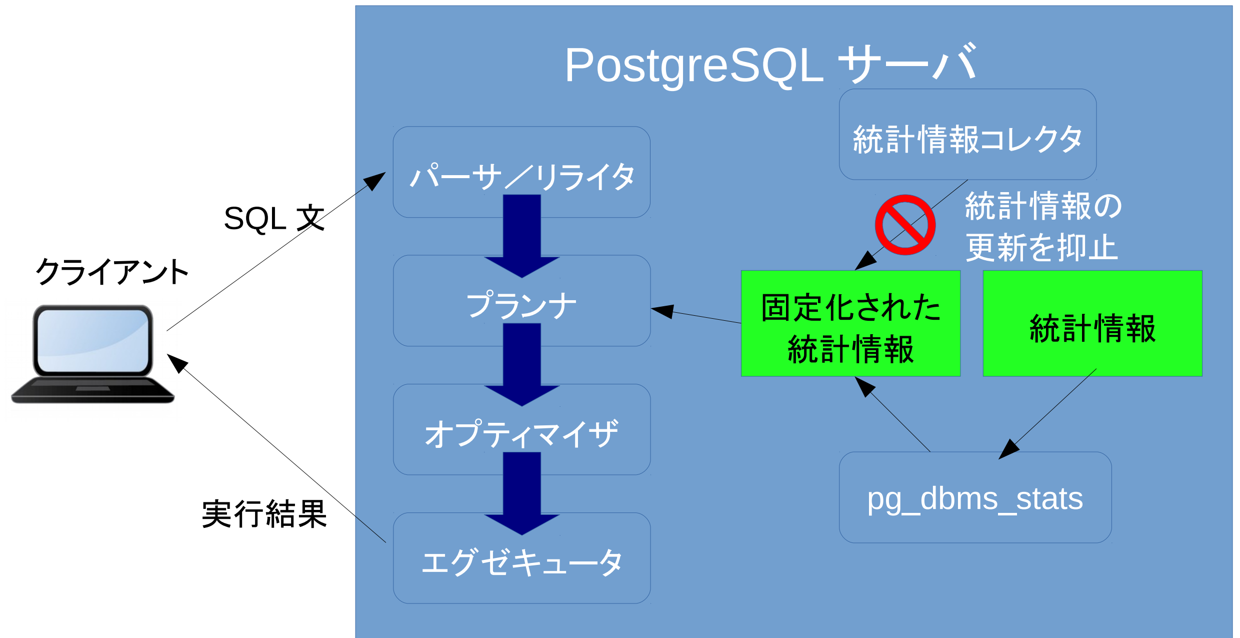


図 7.4: pg_dbms_stats の概要

pg_dbms_stats では、統計情報を管理するために、以下の機能を持っています。

・統計情報のバックアップ/リストア
・不要な統計情報のパージ
・統計情報のロック/ロック解除
・統計情報状態のクリーンアップ
・統計情報のエクスポート/インポート

また、pg_dbms_stats では、通常のテーブル、インデックス、外部表、マテリアライズド・ビューで使われる統計情報を扱います。

詳細については、pg_dbms_stats のドキュメントを参照してください[5]。

5 http://pgdbmsstats.osdn.jp/pg_dbms_stats-ja.html

© 2016 PostgreSQL Enterprise Consortium

PostgreSQL を使うアプリケーションでは、常に最新の統計情報を元にプランナ/オプティマイザが作成する実行計画を使うことで十分なケースが多いです。
しかし、要件によっては最適な実行計画ではない可能性はあるが、一定のレスポンスを維持することが必要になるケースがあります。特に大規模データを扱う場合、実行計画の変動により、大きく性能が変わる可能性があります。このようなリスクを抑えたい場合、本ツールの適用が有効となります。
例えば、運用中にデータの件数や、データの値の分布が大きく変わるようなケースでは、最新の統計情報により、以前よりも良いレスポンスとなる実行計画に変化する場合があります。
しかし、逆に最新の統計情報により、レスポンスが悪化して、システムとして許容できるレスポンス時間を超過するケースもありえます(図 7.5)。

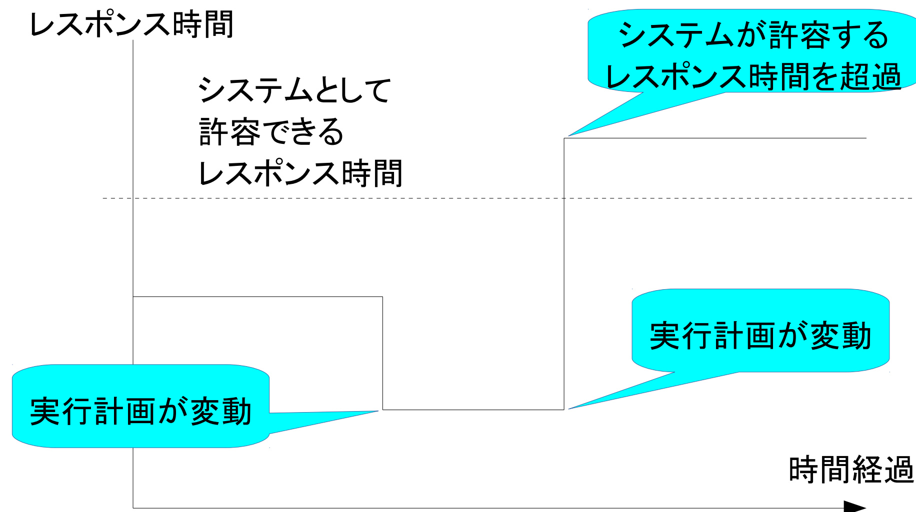


図 7.5: 統計情報の変化による弊害

このような場合は、「最速ではないが、安定したレスポンスとなる」実行計画を常に作成させるために、統計情報を固定化します(図 7.6)。

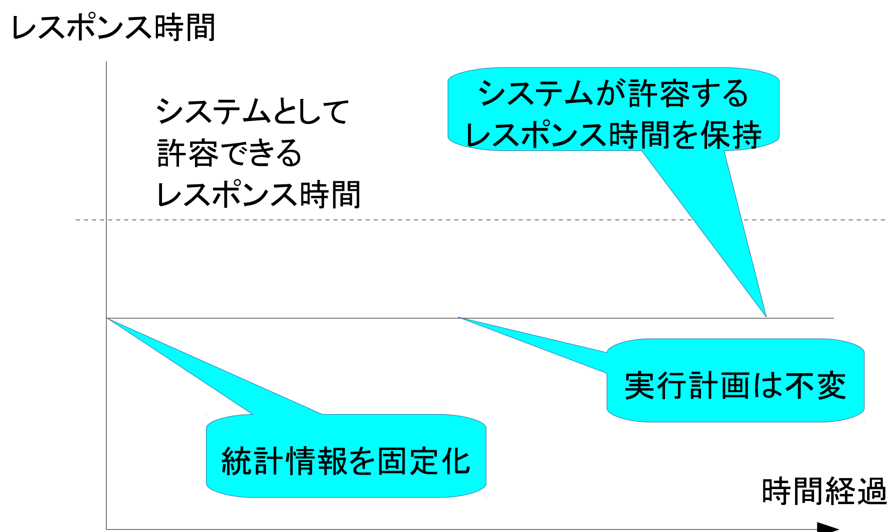


図 7.6: 統計情報の固定化による効果

© 2016 PostgreSQL Enterprise Consortium

7.3. まとめ

実行計画制御ツールを使うべきケースを最後にまとめます。まず、これらのツールは最初から適用を考えるのではなく、基本的には、PostgreSQL の統計情報の管理や、実行計画の作成を PostgreSQL に任せることを推奨します。

以下のケースに該当する場合に、実行計画制御ツールの利用を検討してください。

7.3.1. pg_hint_plan の使いどき

pg_hint_plan の適用を検討すべきなのは、以下のケースです。

・実行する SQL 文を大きく変更することなく、システムに必要な性能を満たす実行計画を作成したいケース。

例えば、非常に複雑な SQL 文を発行するケースがありえます。この場合、PostgreSQL では最適でない実行計画を作成することがありえるため、実行計画の確認も含めた性能検証をしながら適用します。

・実行計画(cost)が最適であっても、実際の処理時間(actual time)が劣化するケース。

例えば、環境によってはコスト上は有利な結合方式であっても、ディスク性能の問題で別の結合方式を選択したほうが良いケースもありえます。

どのようなケースであれ、実行計画の確認を含めた性能検証を実施しながら、適用の要否を検討する必要があります。

また、PostgreSQL 自体のバージョンアップにより、pg_hint_plan を使わなくても性能上問題のない実行計画を作成する可能性があります。pg_hint_plan を導入したシステムで、PostgreSQL のバージョンアップを行う場合には、その契機で、pg_hint_plan をバージョンアップ後にも適用すべきか検証を行うべきです。

7.3.2. pg_dbms_stats の使いどき

pg_dbms_stats の適用を検討すべきなのは、以下のケースです。

・実行計画が運用中に変更されることを防止したいケース。

・最適な実行計画・処理時間でなくても良いが、安定した処理時間を要求されるケース。

こうした場合、システムが要求する条件を満たす状態の統計情報を固定することで、常に安定した実行計画を作成し、処理時間を担保することが可能になります。

© 2016 PostgreSQL Enterprise Consortium

8. まとめ

本報告書では、エンタープライズ用途での PostgreSQL の運用を支援するツール類の紹介と使いどころについて説明しました。

PostgreSQL 自体は高度な機能を備えたデータベースシステムですが、PostgreSQL 単体では、構築時または運用時に足りない機能や使い勝手の悪い面もあります。本報告書では、構築や運用の問題として、性能監視、バックアップ、パーティション化、実行計画の制御を取り上げ、PostgreSQL 単体では何が問題か、そしてその問題をサポートするツールを紹介しました。

これらのツールをうまく活用することで、PostgreSQL をより活用できるようになるでしょう。本報告書がその一助となれば幸いです。

本報告書で紹介した構築や運用を支援する PostgreSQL のツールは、一部でしかありません。

また、今回紹介したツールも PostgreSQL のバージョンに追随しつつ、ツール自体の機能拡張もされているため、今後も調査を継続する必要があります。

こうした運用を支援するツールの紹介や使いどころのノウハウがニーズとして高ければ、次年度もこうした調査を継続していきたいと考えています。もし、何かのツールのノウハウを得たい、あるいは検証してみたいツールがある、という方は、次年度の活動に参加していただければと思います。

© 2016 PostgreSQL Enterprise Consortium

著者

版	所属企業・団体名	部署名	氏名
2015 年度 WG3 活動報告書 第 1 版	NTT ソフトウェア株式会社	クラウド＆セキュリティ事業部	原田 登志
	大日本印刷株式会社	情報イノベーション事業部 C&Iセンター マーケティング・決済プラットフォーム本部	亀山 潤一
	大日本印刷株式会社	情報イノベーション事業部 C&Iセンター マーケティング・決済プラットフォーム本部	田中 良幸
	大日本印刷株式会社	情報イノベーション事業部 C&Iセンター システム開発運用推進本部 ITサービスマネジメント部	望月 慎吾
	TIS 株式会社	IT基盤技術本部　OSS 推進室	中西 剛紀
	TIS 株式会社	IT基盤技術本部　OSS 推進室	下雅意 美紀
	株式会社日立ソリューションズ	技術開発本部　研究開発部	稲垣 毅
	株式会社日立ソリューションズ	サービスビジネス第2部	池田 尊敏

© 2016 PostgreSQL Enterprise Consortium