PGECons
PostgreSQL Enterprise Consortium

課題検討WG 2018年度活動報告

PostgreSQLをさらに活用するためのヒント
～パーティショニング、Windowsでの運用、性能トラブル対処法～

PostgreSQL エンタープライズコンソーシアム
WG3（課題検討WG）

（株）アシスト 家島 ・ NTTテクノクロス（株） 上原 ・ サイオステクノロジー（株）高岡

アジェンダ

- 2018年度の活動テーマと活動体制

- パーティションニング
  - 宣言的パーティションニング構成の機能、性能を調査
- Windowsでの運用
  - Windows環境での運用に関する調査
- 性能トラブル対処法
  - 性能トラブルの予防・検知・対策に関する調査

2018年度の活動テーマ

- ワーキンググループの活動領域の中から、2018年度は運用性・性能に関するテーマを取り上げた

領域	概要	2018	2017	2015以前
可用性	高可用性クラスタ、BPC		レプリケーション Windows環境(可用性)	可用性, BCP レプリケーション
運用性	バックアップ運用、運用監視	**Windows環境(リソース監視、バックアップ) パブリッククラウド**	Windows環境(ツール、インストレーション)	ツール バックアップ
性能	性能評価手法、性能向上手法等	**性能トラブル パーティションニング**	性能トラブル	ツール
セキュリティ	監査、認証、機密性			セキュリティ
接続性	他ソフトウェアとの連携			異種DB連携
保守性	保守サポート、トレーサビリティ			
互換性	他DBMSとの互換性			

WG3 2018年度活動体制

- パーティションニング班
  - 株式会社アシスト
  - NTTテクノクロス株式会社
- Windows班
  - NTTテクノクロス株式会社
  - 富士通株式会社
- 性能トラブル班
  - サイオステクノロジー株式会社
  - 日本電信電話株式会社
  - 株式会社富士通ソーシアルサイエンスラボラトリ
- パブリッククラウド班
  - 日本電気株式会社
  - 株式会社日立製作所

敬称略・50音順

パーティショニング成果紹介

パーティショニング班の活動目的

- テーマ選択にあたっての背景
  - 昨今の爆発的なデータ増加に伴う、大規模データ対応の重要性の高まり
  - PostgreSQL10から、大規模データに対応するための新たなパーティション機能が追加

- 目的
  - PostgreSQL10から実装された宣言的パーティショニングの概要や管理性、性能について紹介する
  - PostgreSQLのエンタープライズ領域への普及にあたって、パーティショニング機能を活用し、大規模データを効率よく処理するための情報を提供する

成果物について

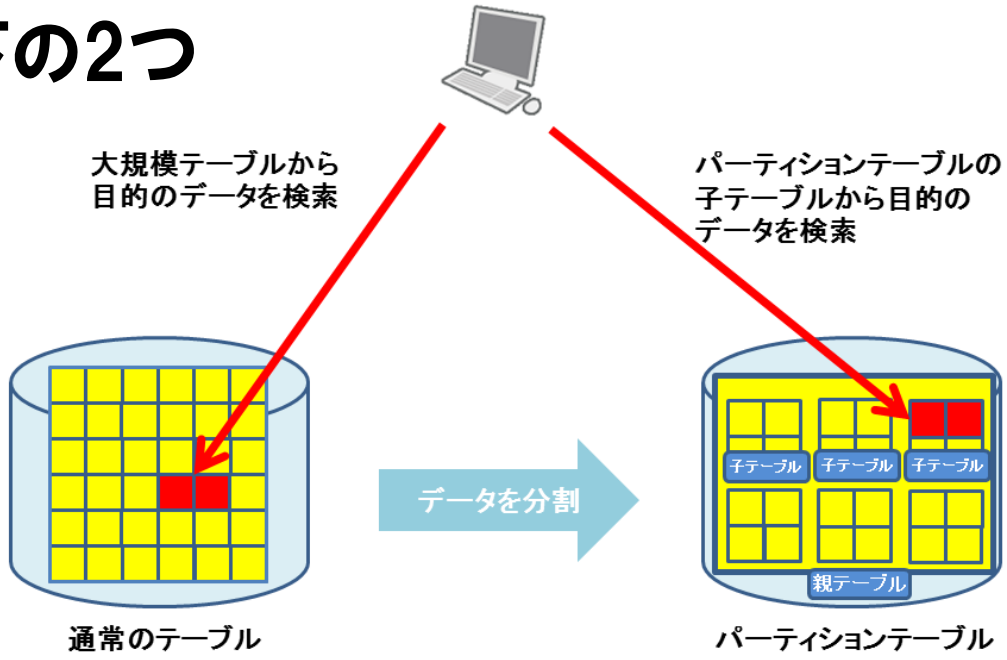
- 参画企業にて、そもそものパーティショニングの目的から最新PostgreSQLバージョン11の機能まで調査・検証した上で、以下の構成で作成
  - PostgreSQLにおけるパーティション
  - パーティションの管理
  - パーティションの性能
  - パーティションのTips

パーティショニング
-PostgreSQLにおけるパーティション-

パーティショニングとは何か

- 論理的には1つの大きなテーブルを物理的に小さなテーブルに分割すること

- 主な目的は、以下の2つ
  - 性能の向上
  - 運用性の向上



パーティショニングの実装方法

- 以下2つのパーティショニングの実装方法がある

	継承を使用したパーティショニング	宣言的パーティショニング
概要	テーブルの「継承」と「CHECK制約」と「トリガー」の機能を使用してデータを分割	CREATE TABLE構文でPARTITION BY句にパーティション方針（範囲／リスト／ハッシュ）とパーティションキーを指定してデータを分割
使用可能バージョン	8.1以降で使用可能	10以降で使用可能
特徴	宣言的パーティショニングより ○　柔軟な構成が行える ×　データ挿入時の性能が劣っている ×　構築手順が煩雑である	継承を使用したパーティショニングより ×　制限が多い ○　データ挿入時の性能が良い ○　構築手順が簡単

PostgreSQL11新機能

- 機能系
  - パーティションを跨る更新
  - ハッシュパーティショニング
  - デフォルトパーティショニングの設定　など

- 性能系
  - パーティション選定の高速化
  - パーティション・ワイズアグリゲート
  - パーティション・ワイズジョイン　など

パーティショニング
-パーティションの管理-

パーティションの管理手法の比較

- テーブル管理SQL
- インデックス管理SQL

Oracle Databaseの知識を有する人にとって
PostgreSQL11の機能でできること/できないことを
イメージしやすいように、一覧にまとめました。

テーブル管理SQL

■ Oracleのテーブル管理SQLと比較

操作	Oracle	PostgreSQL (宣言的パーティション)
パーティション表の作成	1CREATE文で親テーブル、子テーブルを同時に作成できる CREATE TABLE pt1 (列定義) PARTITION BY [RANGE | LIST | HASH] INTERVAL () --インターバルの指定 (RANGEの場合 : PARTITION p1 VALUES LESS THAN ..., LISTの場合 : PARTITION p2 VALUES ..., HASHの場合 : PARTITIONS n (	親テーブルと子テーブルの数だけCREATE文を実行する ・親 : CREATE TABLE pt1 (列定義) **PARTITION** [**RANGE** | **LIST** | **HASH**] (part_key); ・子 : CREATE TABLE p1 **PARTITION OF** pt1 **FOR VALUES** (範囲) RANGEの場合: **FROM** (xxx) **TO** (yyy) LISTの場合 : **IN** (xxx,yyy) HASHの場合 : **WITH** (**MODULUS** xxx, **REMAINDER** yyy) PostgreSQL 11では"DEFAULT"を指定可能
パーティションの追加	ALTER TABLE pt1 ADD PARTITION p1;	CREATE TABLE p1 **PARTITION OF** pt1 **FOR VALUES**...; (上記参照)
パーティションの削除	ALTER TABLE pt1 DROP PARTITION P1;	DROP TABLE p1;
パーティションの データ削除	ALTER TABLE pt1 TRUNCATE PARTITION p1;	TRUCNATE p1;
パーティション名の変更	ALTER TABLE pt1 RENAME PARTITION p1 TO p2;	ALTER TABLE p1 RENAME TO p2;
パーティションの分割	ALTER TABLE pt1 SPLIT PARTITION p1 AT (xx);	なし
パーティションのマージ	ALTER TABLE pt1 MERGE PARTITIONS p1,p2 INTO PARTITION p3;	なし
パーティションの 表領域移動	ALTER TABLE pt1 MOVE PARTITION p1 TABLESPACE ts1;	ALTER TABLE p1 SET TABLESPACE ts1;
パーティションの活性	ALTER TABLE pt1 MODIFY p1 ONLINE;	なし
パーティションの非活性	ALTER TABLE pt1 MODIFY p1 OFFINE;	なし
テーブルをパーティションに変換	ALTER TABLE pt1 ADD PARTITION p1; ALTER TABLE pt1 EXCHANGE PARTITION p1 WITH TABLE t2;	ALTER TABLE pt1 **ATTACH PARTITION** p1 **FOR VALUES** (..);
パーティションをテーブルに変換	CREATE TABLE t2 (..); ALTER TABLE pt1 EXCHANGE PARTITION p1 WITH TABLE t2;	ALTER TABLE pt1 **DETACH PARTITION** p1;
パーティション単位の参照	SELECT * FROM pt1 PARTITION (p1);	SELECT * FROM p1;
リストパーティションの値の追加	ALTER TABLE pt1 MODIFY PARTITION p1 ADD VALUES (val2);	なし
リストパーティションの値の削除	ALTER TABLE pt1 MODIFY PARTITION p1 DROP VALUES (val2);	なし

テーブル管理SQL

■ Oracleのテーブル管理SQLと比較

操作	Oracle	PostgreSQL (宣言的パーティション)
パーティション表の作成	1CREATE文で親テーブル、子テーブルを同時に作成できる CREATE TABLE pt1 (列定義) PARTITION BY [RANGE | LIST | HASH] INTERVAL () --インターバルの指定 (RANGEの場合 : PARTITION p1 VALUES LESS THAN ..., LISTの場合 : PARTITION p2 VALUES ..., HASHの場合 : PARTITIONS n (	親テーブルと子テーブルの数だけCREATE文を実行する ・親 : CREATE TABLE pt1 (列定義) **PARTITION** [**RANGE** | **LIST** | **HASH**] (part_key); ・子 : CREATE TABLE p1 **PARTITION OF** pt1 **FOR VALUES** (範囲) RANGEの場合: **FROM** (xxx) **TO** (yyy) LISTの場合 : **IN** (xxx,yyy) HASHの場合 : **WITH** (**MODULUS** xxx, **REMAINDER** yyy) PostgreSQL 11では"DEFAULT"を指定可能
パーティションの追加	ALTER TABLE pt1 ADD PARTITION p1;	CREATE TABLE p1 **PARTITION OF** pt1 **FOR VALUES**...; (上記参照)
パーティションの削除	ALTER TABLE pt1 DROP PARTITION P1;	DROP TABLE p1;
パーティションの データ削除	ALTER TABLE pt1 TRUNCATE PARTITION p1;	TRUCNATE p1;
パーティション名の変更	ALTER TABLE pt1 RENAME PARTITION p1 TO p2;	ALTER TABLE p1 RENAME TO p2;
パーティションの分割	ALTER TABLE pt1 SPLIT PARTITION p1 AT (xx);	なし
パーティションのマージ	ALTER TABLE pt1 MERGE PARTITIONS p1,p2 INTO PARTITION	なし
パーティション単位の参照	SELECT * FROM pt1 PARTITION (p1);	SELECT * FROM p1;
リストパーティションの値の追加	ALTER TABLE pt1 MODIFY PARTITION p1 ADD VALUES (val2);	なし
リストパーティションの値の削除	ALTER TABLE pt1 MODIFY PARTITION p1 DROP VALUES (val2);	なし

Oracleのテーブル管理SQLとの互換性はないため、
PostgreSQL独自の管理SQLを理解する必要がある

インデックス管理SQL

■ Oracleのインデックス管理SQLと比較

操作	Oracle	PostgreSQL (宣言的パーティション)
ローカル索引	CREATE INDEX idx1 ON pt1 (col1) LOCAL;	CREATE INDEX idx1 ON pt1 (col1); 親に実施した場合、親には索引定義のみ作成、 各子テーブルに索引を作成。 子に実施した場合、特定子テーブルに索引を作成
グローバル・パーティション索引	CREATE INDEX idx1 ON pt1 (col1) GLOBAL PARTITION BY RANGE (col1) (PARTITION p_100 VALUES LESS THAN (value1), PARTITION p_1000 VALUES LESS THAN (value2) ...)	なし
グローバル・非パーティション索引	CREATE INDEX idx1 ON pt1 (col1);	なし
親テーブルの索引定義	なし	CREATE INDEX idx1 ON **ONLY** pt1 (col1); 親の索引に実施、定義のみ作成される
親子の索引の紐づけ	なし	ALTER INDEX idx1 **ATTACH PARTITION** idx1_p1; 親の索引と子の索引を紐付ける
索引パーティションの属性の変更	なし	ALTER INDEX idx1 SET (ATTIBUTE=value); 子の索引に実施
索引パーティションのデフォルト属性の変更	ALTER INDEX idx1 MODIFY DEFAULT ATTRIBUTES	ALTER INDEX idx1 SET (ATTIBUTE=value); 親の索引に実施する事でデフォルト属性の変更に相当 子の既存索引には影響せず、 以後に追加されたパーティション索引作成時に有効
索引パーティションの表領域移動	ALTER INDEX idx1 REBUILD PARTITION pt1 PARAMETERS('TABLESPACE=ts2');	ALTER INDEX idx1 SET TABLESPACE ts2; 親の索引に実施した場合、親の索引の定義のみ変更。 子の既存索引には影響せず、以後に追加されたパーティション索引作成時に有効。 子の索引に実施した場合、特定索引の表領域を移動。
索引パーティション名の変更	ALTER INDEX idx1 RENAME PARTITION idx2;	ALTER INDEX idx1 RENAME TO idx2; 親の索引に実施した場合、 親の索引名のみ変更される。子の既存索引には影響せず、 子に自動作成される索引の命名規則にも影響しない。 <子テーブル>_<列名>_idx 子の索引に実施した場合 、特定索引の名前を変更

※太字部分はパーティションインデックス独自の構文

インデックス管理SQL

- Oracleのインデックス管理SQLと比較

操作	Oracle	PostgreSQL (宣言的パーティション)
ローカル索引	CREATE INDEX idx1 ON pt1 (col1) LOCAL;	CREATE INDEX idx1 ON pt1 (col1); 親に実施した場合、親には索引定義のみ作成、 各子テーブルに索引を作成。 子に実施した場合、特定子テーブルに索引を作成
グローバル・パーティション索引	CREATE INDEX idx1 ON pt1 (col1) GLOBAL PARTITION BY RANGE (col1) (PARTITION p_100 VALUES LESS THAN (value1), PARTITION p_1000 VALUES LESS THAN (value2) ...)	なし
グローバル・非パーティション索引	CREATE INDEX idx1 ON pt1 (col1);	なし
親テーブルの索引定義	なし	CREATE INDEX idx1 ON **ONLY** pt1 (col1); 親の索引に実施、定義のみ作成される
親子の索引の紐づけ	なし	ALTER INDEX idx1 **ATTACH PARTITION** idx1_p1; 親の索引と子の索引を紐付ける
索引パーティションの属性の変更	なし	ALTER INDEX idx1 SET (ATTIBUTE=value); 子の索引に実施
索引パーティションのデフォルト属性の変更	ALTER INDEX idx1 MODIFY DEFAULT ATTRIBUTES	ALTER INDEX idx1 SET (ATTIBUTE=value); 親の索引に実施する事でデフォルト属性の変更に相当 子の既存索引には影響せず、 以後に追加されたパーティション索引作成時に有効

Oracleのインデックス管理SQLとの互換性はないため、
PostgreSQL独自の管理SQLを理解する必要がある

プライマリーキーの作成

■ 概要

PostgreSQL11では全パーティションに跨るプライマリーキーやユニークインデックスを作成することが可能。これによりパーティションテーブルのデータの整合性が高まる。

■ 注意点

インデックスキーにはパーティションキーを含める必要がある。

インデックス作成のカスケード

■ 概要

PostgreSQL11の宣言的パーティショニングでは、
親テーブルにインデックスを作成することで、
子テーブルにも自動的にインデックスが作成される。

■ 注意点

インデックスの属性情報も引き継ぐため、同一のテーブルスペースに作成される。

※親テーブルに作成後にALTER INDEXでローカルインデックスを指定してテーブルスペースを変更することは可能。

パーティショニング
-パーティションの性能-

検証項目

No.	検証項目	検証目的
①	COPY FROM および INSERTの改善 **継承パーティションニング vs 宣言的パーティションニング**	継承を使用したパーティショニングの課題である挿入処理が、宣言的パーティショニングで改善されていることを確認すること
②	非パーティションテーブルと宣言的パーティショニングの特性比較 **インデックス走査 vs パーティションプルーニング**	非パーティションテーブル/パーティションテーブルがそれぞれ優位となるケースを確認すること
③	宣言的パーティショニングの集計処理の改善 **PostgreSQL 10 vs PostgreSQL 11**	集計性能がPostgreSQL11の新機能であるパーティション・ワイズジョインにて改善されていることを確認すること

検証に関する情報 （1/2）

■ 検証環境

項目	仕様
CPU	Intel Xeon CPU E5-2640 v2 @ 2.00GHz * 32 Core
搭載メモリ	64 GB
内蔵ストレージ	約 2 TB
OS	Red Hat Enterprise Linux Server release 6.5
PostgreSQL	PostgreSQL 10.1 PostgreSQL 11.1

■ 検証データ

- StarSchemaBenchmarkのlineorderテーブルを使用
- データはスケール4を指定して作成（件数：2,400万件、サイズ：2.3GB）
- データは各パーティションテーブルでほぼ均等に分布

検証に関する情報 (2/2)

■ 使用するパーティションテーブル

- □ lineorderテーブルを使用
- □ すべてレンジパーティションによる時系列のパーティション表

■ 検証方法

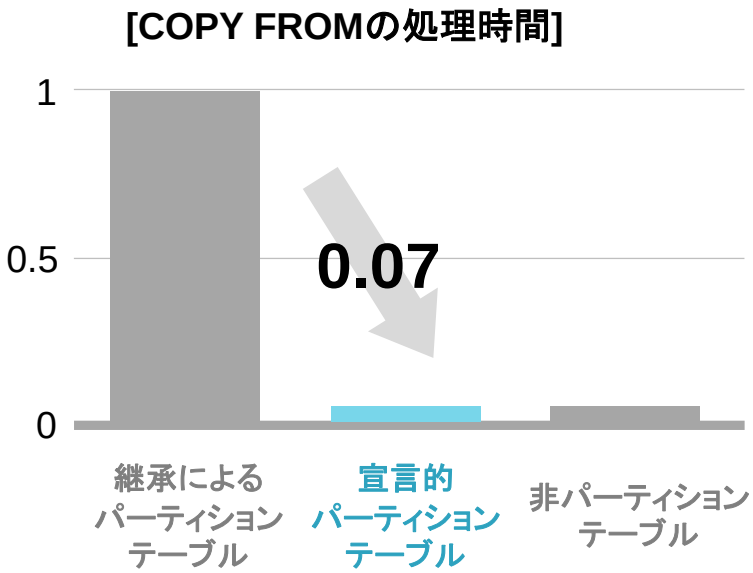
- □ 以下の構成のテーブルを使用
- □ テーブルのデータは、pg_prewarmで共有メモリ上に載せる
- □ Index Only Scanが効くように検証前にはVACUUMを実施

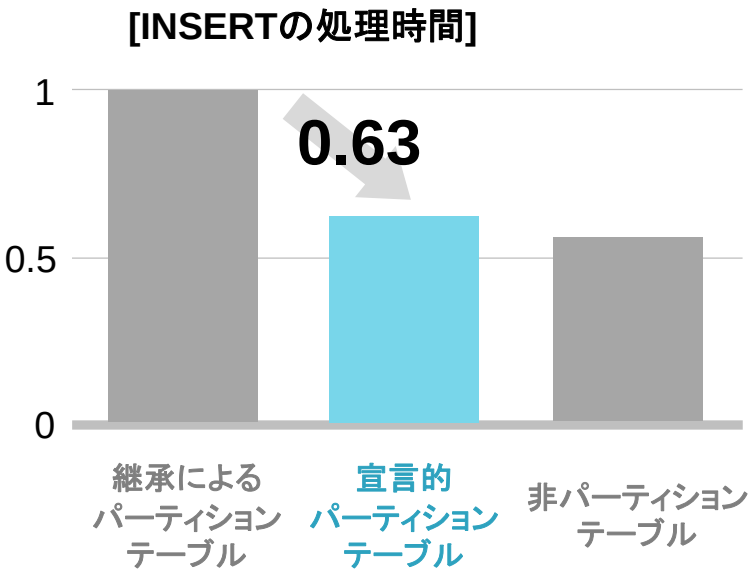
構成	パーティション数
非パーティションテーブル	0
宣言的パーティションテーブル(年)	7
継承によるパーティションテーブル(年)	7

検証結果① ～挿入処理（継承 vs 宣言的）～

- 継承によるパーティショニングの課題であった挿入処理が宣言的パーティショニングで大幅に改善
- 特にCOPY FROMでは宣言的パーティショニングが圧倒

継承によるパーティションテーブルの処理時間を1.00とした相対値



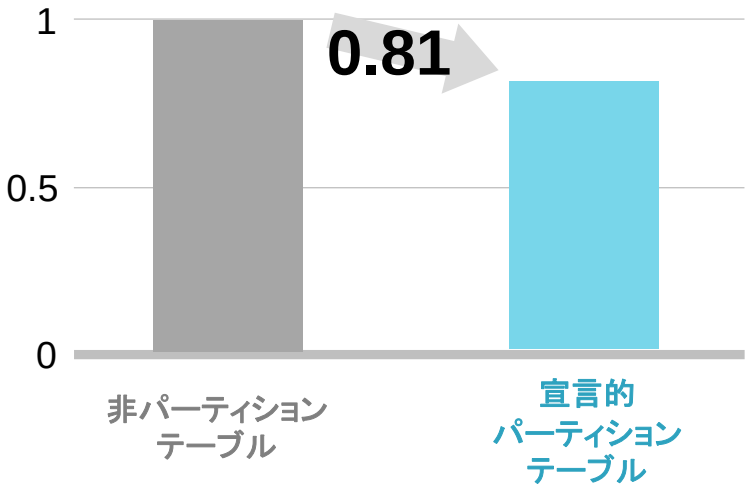


検証結果② ～範囲検索（インデックス vs パーティション）～

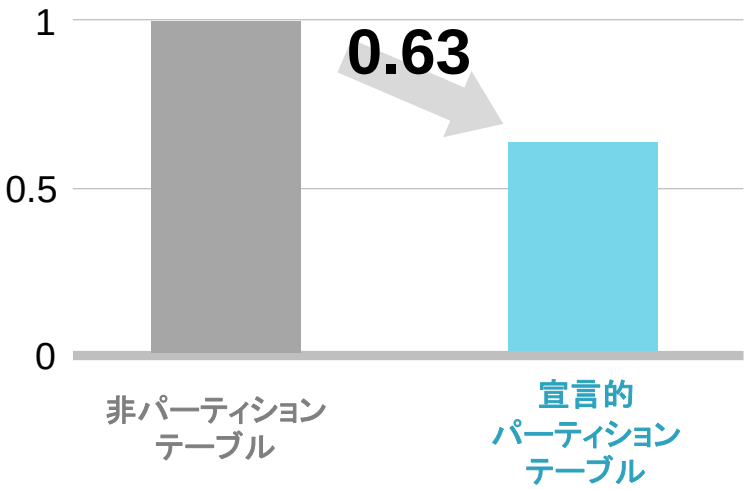
- 検索範囲が狭い場合、大きな差異はない
  - インデックス走査が有効であり、パーティションプルーニングに対抗可能
- 検索範囲が広い場合、パーティションが優位
  - インデックス走査は全表走査に切り替る
  - パーティションプルーニングは検索範囲に依らず安定動作

非パーティションテーブルの処理時間を1.00とした相対値

[索引あり－検索範囲3ヶ月の処理時間]



[索引あり－検索範囲3年の処理時間]

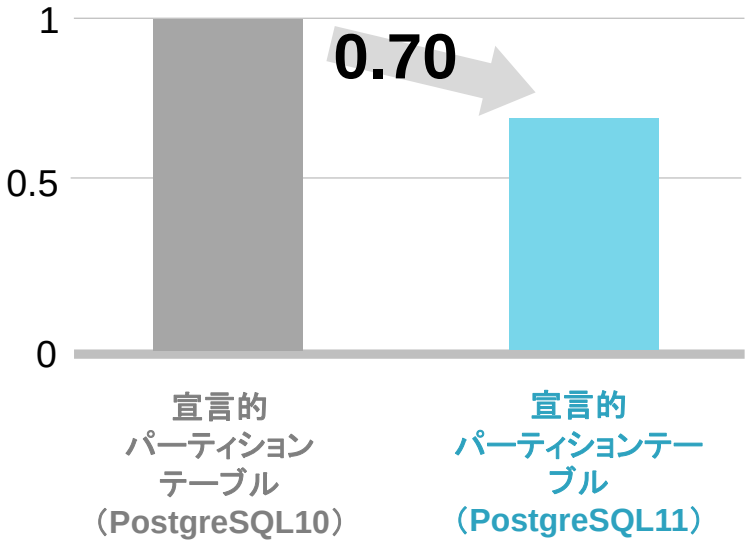


検証結果③ ～集計処理（Postgres10 vs Postgres11）～

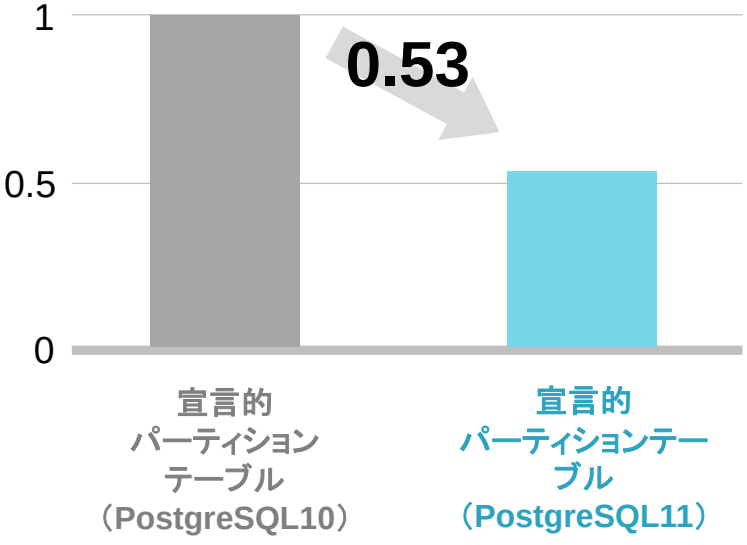
- 総じてPostgreSQL11の方が性能が良い
- データ量が増加するにつれて性能差は顕著に

PostgreSQL10の集計処理時間を1.00とした相対値

[対象期間3日のCOUNT(*)の集計処理]



[対象期間2,500日のCOUNT(*)の集計処理]



パーティショニング
-パーティションのTips-

パーティションが変更になるUPDATE

- PostgreSQL10までは、DELETEおよびINSERTが必要
- PostgreSQL11では、パーティションが変更になるUPDATEが可能

■PostgreSQL10ではエラーが発生

```
=# UPDATE  lineorder_yyyymmdd
    SET        lo_orderdate = '19980102'
    WHERE   lo_orderdate = '19920101';
ERROR: new row for relation "lineorder_yyyymmdd_19920101" violates
partition constraint
```

■PostgreSQL11では正常終了

```
=# UPDATE  lineorder_yyyymmdd
    SET        lo_orderdate = '19980102'
    WHERE    lo_orderdate = '19920101';
UPDATE 10228
```

プルーニング適用条件

■ パーティションプルーニングとは

対象のデータが格納されるパーティションテーブルだけにアクセスし、不要なパーティションを読み飛ばす機能。

■ 適用条件

- □ WHERE句の条件でパーティションキーを指定
- □ 以下のWHERE句の条件に合致

指定方法	指定例
完全一致	partition-key = 20190101
範囲指定	partition-key **BETWEEN** 20190101 **AND** 20190201 partition-key **IN** (20190101, 20190102) partition-key > 20190101 AND partition-key < 20190131

※WHERE句の指定方法が、前方一致 or 中間一致 or 後方一致 or 関数指定の場合、
パーティションプルーニングは効かない。

まとめ

- 宣言的パーティションにより性能・管理性が向上し、パーティション導入のしきいは下がっている
- PostgreSQL11を使うことで、さらに性能・管理性の向上が期待できる
- 成果物にはより詳細な内容を記載

Windows環境調査成果紹介

もくじ

- 導入
  - テーマ選択の背景
  - 活動方針
- 周辺ツールの調査結果
  - リソース監視
    - Prometheus
    - typeperf
  - バックアップ運用
    - pg_rmanとの比較

テーマ選択の背景と現状認識

- 成果報告会のアンケート（2015年度）

Q. PostgreSQLをよりミッションクリティカル性の高いエンタープライズ領域で採用するための課題は何だとお考えですか？

⇒ 高可用構成がLinuxを中心とした構成が多い。
Windows構成での事例が少なく、提案しにくい。

- PGConf.ASIA 2017のアンケート

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

Windows環境の運用ノウハウが必要と考えた

昨年度からの活動方針

- 2017年度：Windows環境での運用手順を整理
  - 運用上の大きな懸念点はない
  - 周辺ツールには「Windows非対応」のものがある

https://pgecons-sec-tech.github.io/tech-report/html_wg3_windows/wg3_windows.html

- 2018年度：周辺ツールの調査
  - よく利用される周辺ツールを対象
    - リソース監視
    - バックアップ運用

OS/PostgreSQL層のリソース監視

リソース監視方法の調査方針

- リソース監視はOS/PostgreSQL層を対象
- pg_statsinfoはWindowsでは利用できないため、Windowsで利用可能な「Prometheus」、「typeperf」の2種類を調査する
  - pg_statsinfoと比較し、取得可能リソースを整理する

- それぞれの選定理由は以下のとおり
  - Prometheus
    - OS層のリソース監視ができるエージェントが存在するため
    - 体系的なリソース監視を提供できるため
  - typeperf
    - Windowsに標準でインストールされているため

Prometheusとは

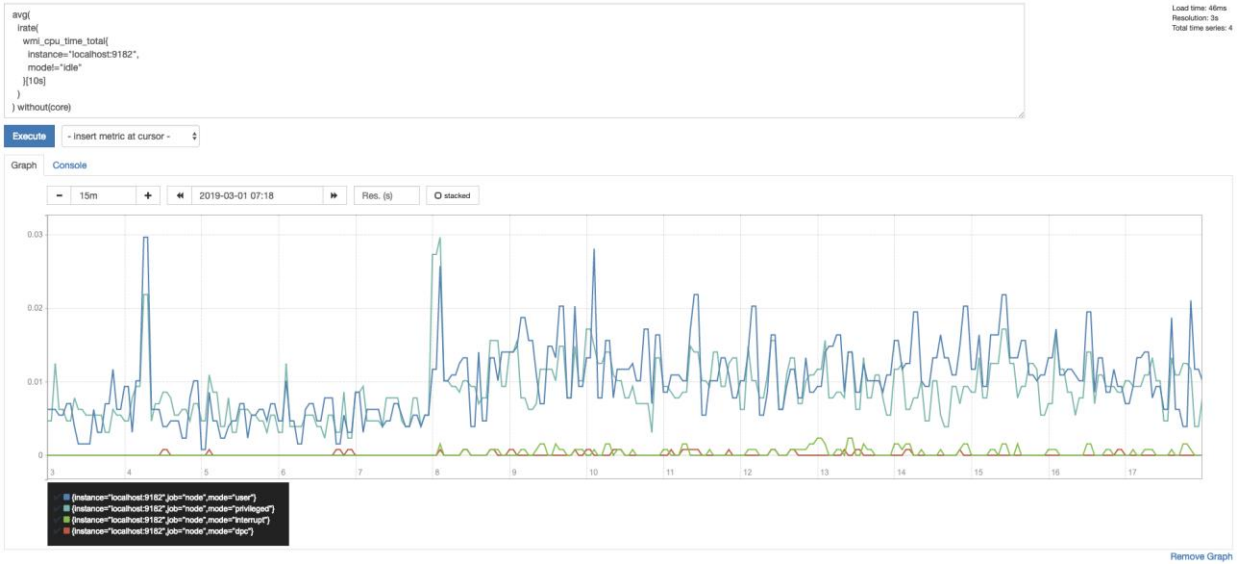
- リソース情報を収集するミドルウェア
  - https://prometheus.io/
  - Pull型のアーキテクチャを採用
  - CFCNでGraduatedステータス取得済み
    - CFCN = Cloud Native Computing Foundation
    - Graduatedはコミュニティが十分成熟している
  - golang製でどのOSでも動作
  - ライセンスはApache License 2.0を採用

Prometheusとは

- ノードに配置するエージェントはexporterと呼ばれる
  - exporterごとに収集するリソースが違う
    - wmi_exporter
    - postgres_exporter
  - exporterはコミュニティから数多く配布されている
    - https://prometheus.io/docs/instrumenting/exporters/
  - 自作も可能
    - API(/metrics)を実装する

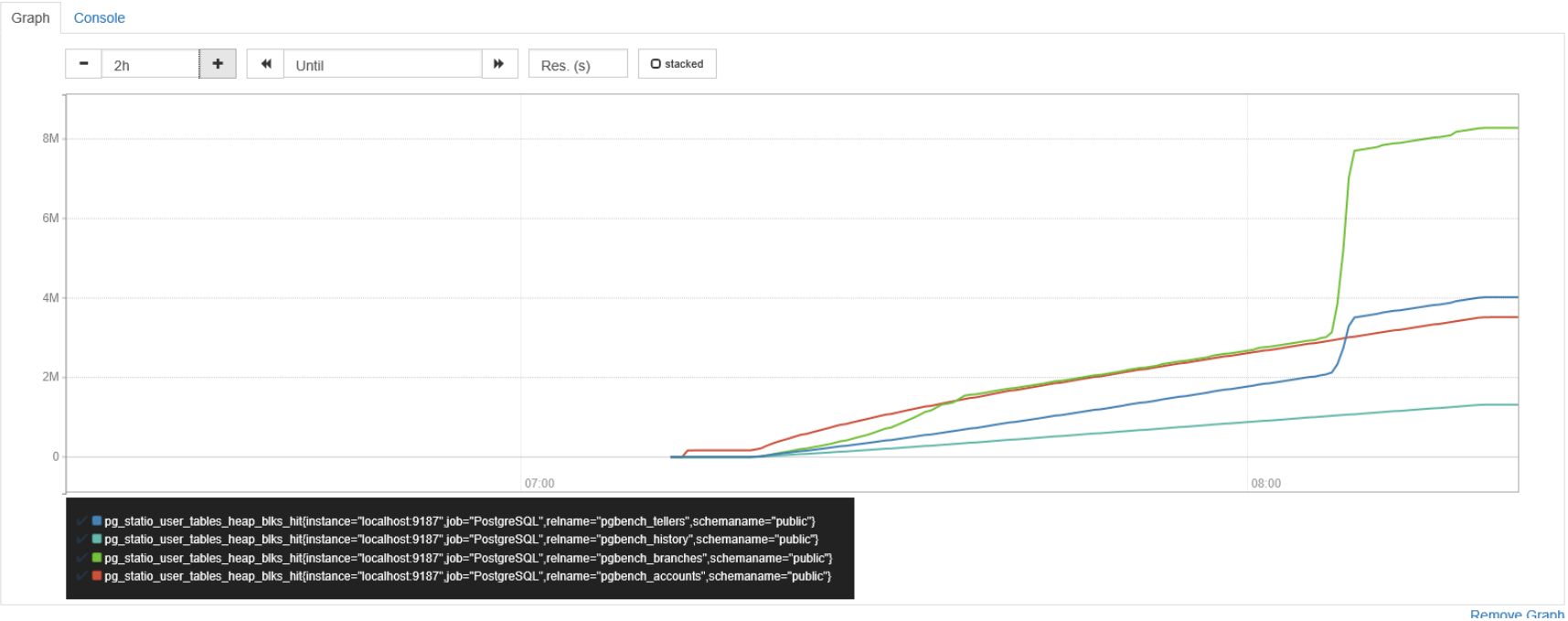
wmi_exporterの収集内容

- OS層のリソース情報を収集できる
  - CPU使用率
  - メモリ使用率
  - 各ディスクの書き込み量 etc…
- 画像はCPU使用率をGUIで表示したもの



postgres_exporterの収集内容

- PostgreSQL層のリソースを収集できる
  - データベース容量
  - キャッシュヒット率 etc…
- 以下はキャッシュヒット率のグラフ（pgbench実行時）



typeperfとは

- Windowsに標準で提供されているツール
  - LinuxのvmstatやsarのコマンドLinux相当のリソースを取得可能
    - コマンドの詳細は成果物を参照

https://pgecons-sec-tech.github.io/tech-report/html_wg3_windows_2018/wg3_windows_2018.html - id2

typeperfとは

- コマンドの発行例
  - -cf 取得リソースファイルを指定
  - -si 取得間隔 [sec]
  - -sc 回数出力 [回]
  - -o 出力ファイルを指定

```
PS > typeperf.exe -cf typeperf_list.txt
     -si 5 -sc 3 -o typep.csv
```

- 出力されるファイルは以下
  - https://pgecons-sec-tech.github.io/tech-report/html_wg3_windows_2018/_downloads/03_typep.csv
    - CPU使用率
    - メモリ使用率
    - 各ディスクの書き込み量 etc…

検証結果

実現ツール	メリット	デメリット
Prometheus	• exporterを追加して取得リソースを増やせすことが可能 • GUIでリソースを参照できる • 他の機能(アラートなど)を利用できる • exporterが自作できる	• 運用するミドルウェアが増える
typeperf	• Windowsの標準機能 • より多くの項目のリソース情報が取得できる	• GUIがなく、リアルタイムでの確認が困難

リソース監視のまとめ

- Prometheusやtypeperfを利用することで各種リソースを取得できるが、pg_statsinfoと比較すると、その項目を全てカバーはできていない
  - カバーできる
    - デッドロック回数
    - CPU (user) 使用率 etc…
  - カバーできない
    - チェックポイント傾向
    - CPU (system) 使用率 etc…
- 取得リソース項目の比較結果は以下
  - https://pgecons-sec-tech.github.io/tech-report/html_wg3_windows_2018/_downloads/01_comparison_with_pg_statsinfo.pdf

Windows環境
バックアップ運用

バックアップの調査方針

- バックアップ運用をWindowsで実現する方法を調査
  - 昨年調査したツールは全てWindowsに対応していなかった
    - pg_rman
    - Barman for PostgreSQL
    - OmniPITR
- 本体機能で実現することを方針とした
  - バックアップ運用はpg_rmanの機能を基準として実現性を整理
    - https://ossc-db.github.io/pg_rman/index-ja.html
    - pg_rmanを選定した理由はGoogle Trendや検索のヒット件数が最も多かったため

検証結果

- 本体機能で代用可能
  - ○ DBクラスタ全体のバックアップ/リストア
    - pg_basebackupを利用
  - ○ WALのバックアップ
    - PostgreSQLのarchive機能を利用

- 代用不可能
  - × バックアップの世代管理
  - × バックアップファイルの検証
  - × 増分バックアップ

https://pgecons-sec-tech.github.io/tech-report/html_wg3_windows_2018/_downloads/02_comparison_with_PostgreSQL_on_Windows.pdf

Windows環境調査2018年度活動のまとめ

リソース監視、バックアップ運用調査を深掘

リソース監視

調査ツールの組み合わせでリソース監視は実現可能

- Prometheus
  - 拡張性が高く、導入のハードルも低い
  - pg_statsinfoに比べて監視項目は見劣りする
- typeperf
  - OS標準のコマンドを利用可能

バックアップアップ運用

- PostgreSQLの機能でバックアップ/リストアの運用をする
- バックアップユーティリティ的な運用をするには課題が多い

性能トラブル班
成果紹介

性能トラブル班の活動目的

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

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

成果物について

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

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

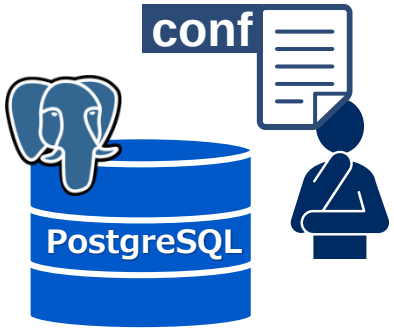
の3つの観点で調査/整理したノウハウを紹介

成果物の構成（昨年度）

予防

3章 性能トラブルを予防するために考慮すべきデータベース設計のポイント

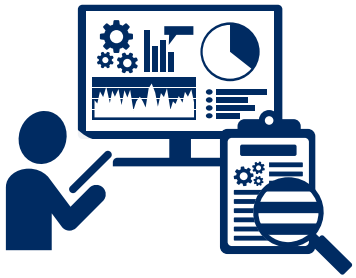
- 性能トラブルを予防するために、設計段階で考慮するポイントを再確認
- 詳細はリンク先の技術情報、講演資料で深掘り



検知

4章 性能状態を把握するための監視

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



対処

5章 ケーススタディ

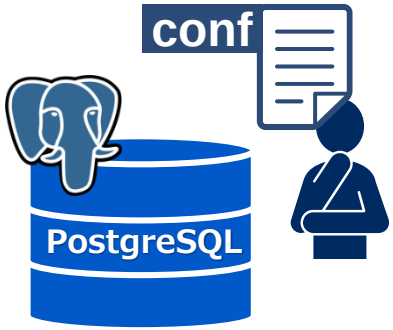
- 性能トラブル発生から解決までのサンプル事例3件

成果物の構成(今年度追記分)

予防

4章 性能トラブルを予防するために考慮すべきデータベース設計のポイント

- 性能関連パラメータ設計を支援するソフトウェアPGTuneの紹介
- autovacuumの処理時間とパラメータチューニング
- pg_basebackup中の性能影響
- インデックス再作成中の性能影響



検知

5章 性能状態を把握するための監視

- 待機イベントの監視方法と確認観点
  pg_wait_sampling

対処

6章 ケーススタディ

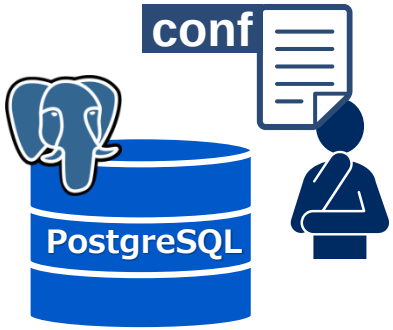
- 性能トラブル発生から解決までのサンプル事例4件
  - 複合インデックスの定義誤りによる性能トラブル
    他、3件

予防・検知・対処の追記分を抜粋してご紹介

予防

4章 性能トラブルを予防するために考慮すべきデータベース設計のポイント

- 性能関連パラメータ設計を支援するソフトウェアPGTuneの紹介
- autovacuumの処理時間とパラメータチューニング
- pg_basebackup中の性能影響
- インデックス再作成中の性能影響



検知

5章 性能状態を把握するための監視

- 待機イベントの監視方法と確認観点
  pg_wait_sampling

対処

6章 ケーススタディ

- 性能トラブル発生から解決までのサンプル事例4件
  - 複合インデックスの定義誤りによる性能トラブル
    他、3件

autovacuum処理時間とパラメータチューニング-①

- autovacuumを利用する上での課題と検証目的

  - 課題
    - 不要領域の発生速度がautovacuumの回収速度を上回っている場合にテーブル肥大化
    - テーブル肥大化防止のためのパラメータ設計に関する情報が不足

  - 検証目的
    - テーブル肥大化防止のためのパラメータ設計の方法を明確化

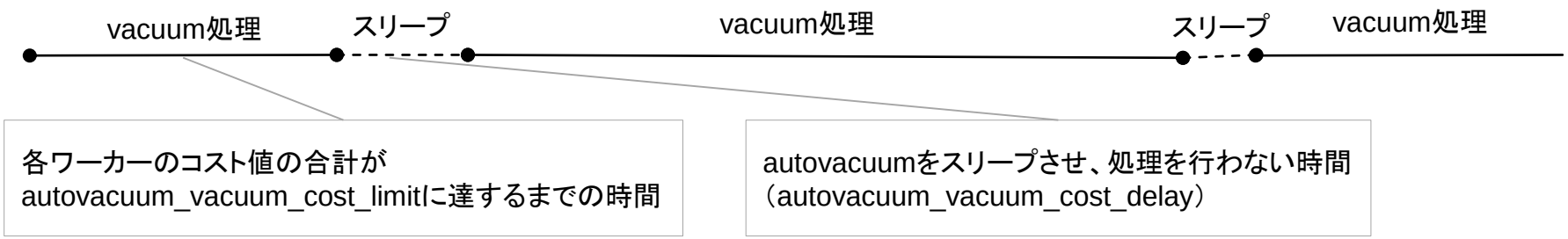
autovacuum処理時間とパラメータチューニング-②

■ パラメータの説明

autovacuumの速度を制御するパラメータ

パラメータ	パラメータ説明(概要)
autovacuum_vacuum_cost_limit	autovacuumに使用されるコスト限界値を指定 デフォルト値は200
autovacuum_vacuum_cost_delay	autovacuumに使用されるコスト遅延値を指定 デフォルト値は20(ミリ秒)

autovacuumのデフォルト設定では、autovacuumによるI/O負荷を軽減するために、処理とスリープを繰り返しながら回収を行う。



※コスト値:シーケンシャルI/Oで1ページを読み込むコストを1.0とした際の相対値

autovacuum処理時間とパラメータチューニング-③

■ 検証1. 不要領域回収速度の測定

□ 測定方法

- 1000万件データに対して更新を行い、25%(250万件)の不要領域を作成
- autovacuumによる不要領域の回収速度を確認

autovacuum処理時間とパラメータチューニング-④

■ 検証1の測定結果

各パラメータ値を変更して1秒間にautovacuumが処理できる平均件数を算出し、不要領域の回収速度とした。
() 内はデフォルト設定を1とした場合との性能比率

	autovacuum_vacuum_cost_limit					
autovacuum_vacuum_cost_delay	設定値	200 (デフォルト)	400	600	800	1000
	20 (デフォルト)	1645(件/秒) (1.00)	3016(件/秒) (1.83)	4046(件/秒) (2.46)	5051(件/秒) (3.07)	6113(件/秒) (3.72)
	15	2213(件/秒) (1.35)	3842(件/秒) (2.34)	5152(件/秒) (3.13)	5961(件/秒) (3.62)	6856(件/秒) (4.17)
	10	3087(件/秒) (1.88)	5092(件/秒) (3.10)	6281(件/秒) (3.82)	7347(件/秒) (4.47)	8013(件/秒) (4.87)
	5	5183(件/秒) (3.15)	7399(件/秒) (4.50)	8588(件/秒) (5.22)	9222(件/秒) (5.61)	10054(件/秒) (6.11)

autovacuum処理時間とパラメータチューニング-⑤

■ 検証2. チューニング効果測定

□ 測定方法

- 1000万件のデータに対して更新を行い、25%(250万件)の不要領域を作成
- 1分毎にテーブルの3.125%(31.25万件)を更新
- 不要行数とテーブルサイズの推移を測定

□ 次のパターンで測定

- デフォルト設定
- 毎分発生する不要領域の発生速度を上回るパラメータ値を選択

autovacuum処理時間とパラメータチューニング-⑥

■ デフォルト設定の回収速度

デフォルト設定では、31.25万(件/分)の不要領域発生速度を下回る autovacuum回収速度である。
31.25万(件/分)は約5200(件/秒)である。

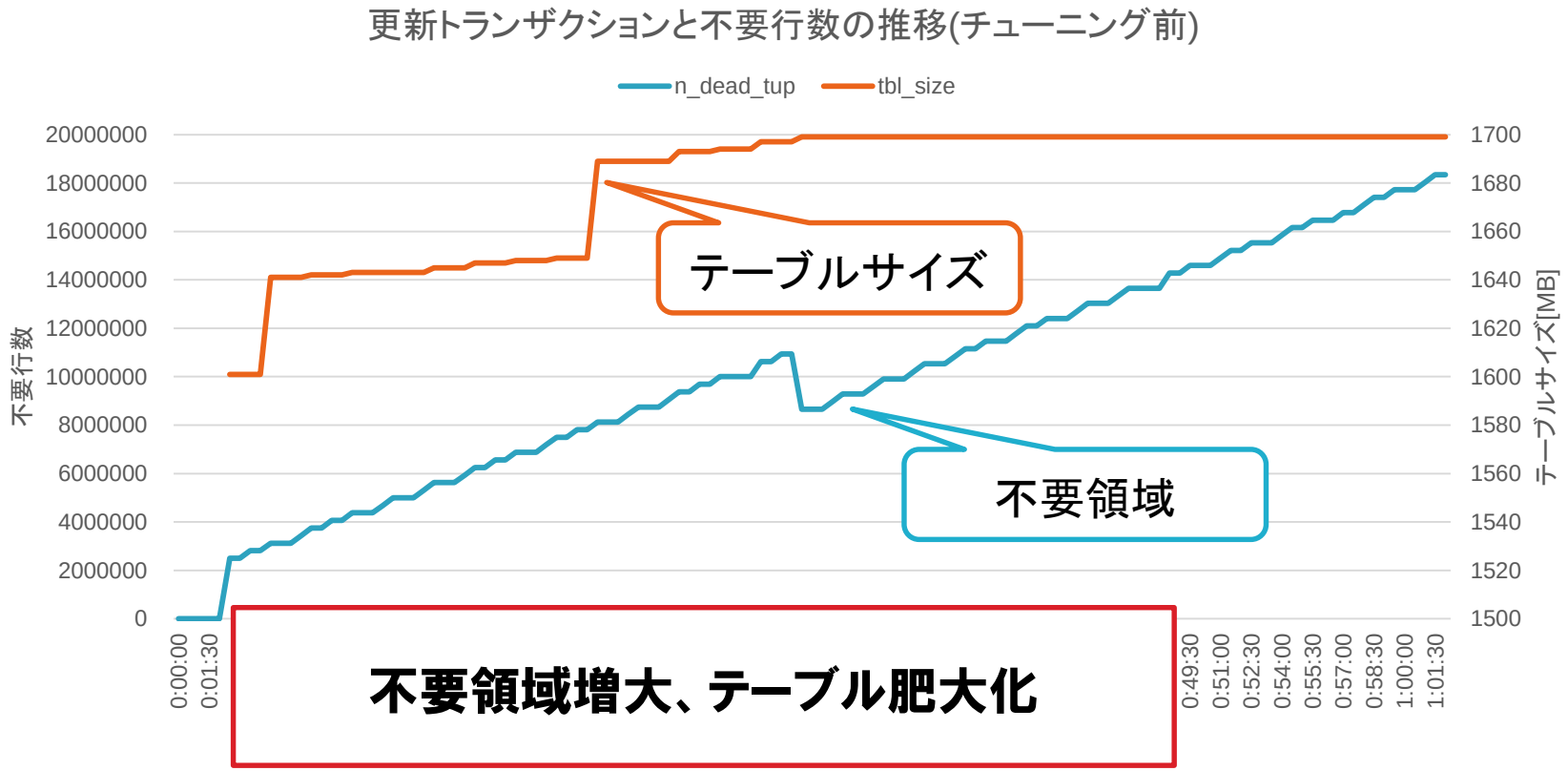
	autovacuum_vacuum_cost_limit					
autovacuum_vacuum_cost_delay	設定値	200 (デフォルト)	400	600	800	1000
	20 (デフォルト)	1645(件/秒) (1.00)	3016(件/秒) (1.83)	4046(件/秒) (2.46)	5051(件/秒) (3.07)	6113(件/秒) (3.72)
	15	2213(件/秒) (1.35)	3842(件/秒) (2.34)	5152(件/秒) (3.13)	5961(件/秒) (3.62)	6856(件/秒) (4.17)
	10	3087(件/秒) (1.88)	5092(件/秒) (3.10)	6281(件/秒) (3.82)	7347(件/秒) (4.47)	8013(件/秒) (4.87)
	5	5183(件/秒) (3.15)	7399(件/秒) (4.50)	8588(件/秒) (5.22)	9222(件/秒) (5.61)	10054(件/秒) (6.11)

autovacuum処理時間とパラメータチューニング-⑦

- **検証2. パラメータ値をデフォルトに設定した場合**

  autovacuum_vacuum_cost_delay=20

  autovacuum_vacuum_cost_limit=200



autovacuum処理時間とパラメータチューニング-⑧

■ パラメータ値の検討

テーブル肥大化を抑止するためには、31.25万(件/分)(約5200(件/秒))の不要領域発生速度を上回るautovacuum回収速度を実現するパラメータ値を設定する必要がある。

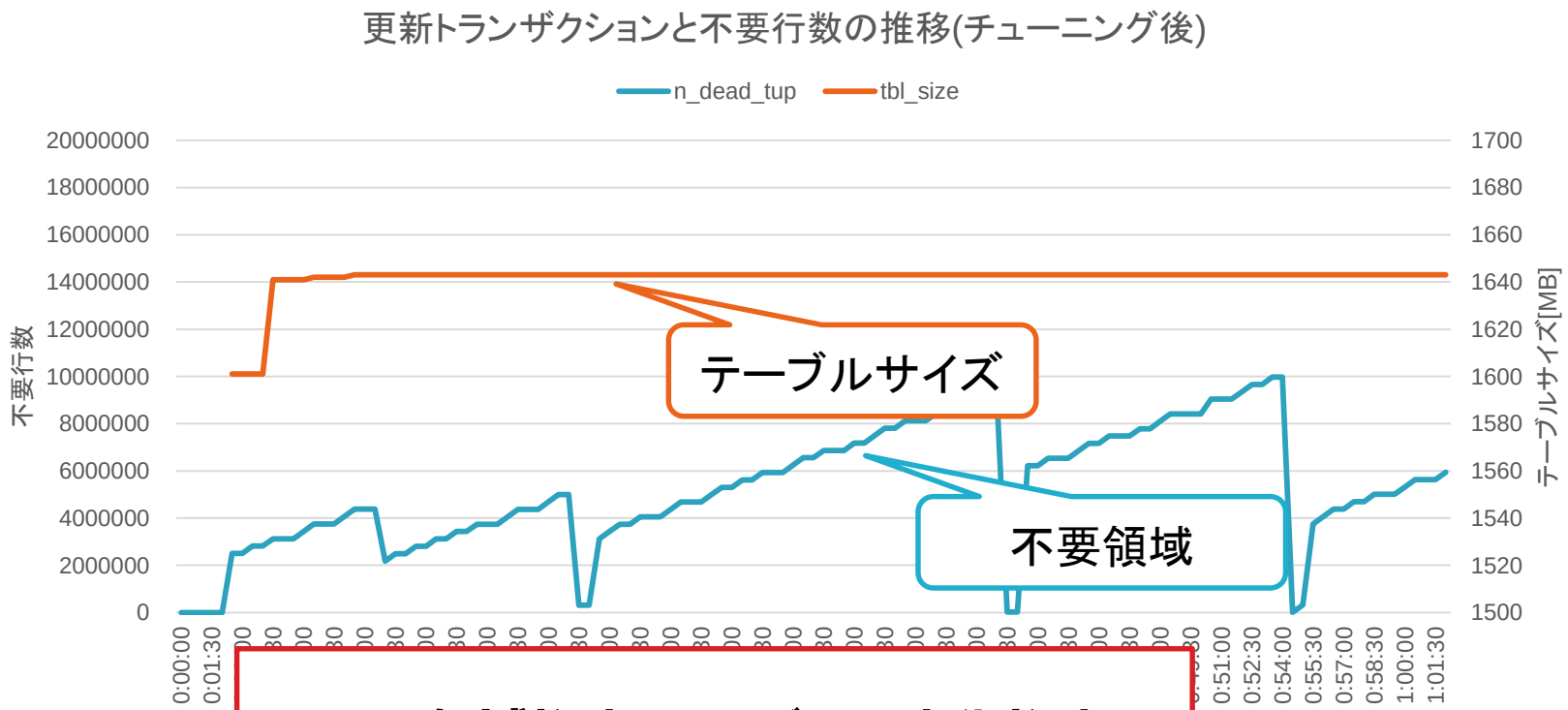
	autovacuum_vacuum_cost_limit					
autovacuum_vacuum_cost_delay	設定値	200 (デフォルト)	400	600	800	1000
	20 (デフォルト)	1645(件/秒) (1.00)	3016(件/秒) (1.83)	4046(件/秒) (2.46)	5051(件/秒) (3.07)	6113(件/秒) (3.72)
	15	2213(件/秒) (1.35)	3842(件/秒) (2.34)	5152(件/秒) (3.13)	5961(件/秒) (3.62)	6856(件/秒) (4.17)
	10	3087(件/秒) (1.88)	5092(件/秒) (3.10)	6281(件/秒) (3.82)	7347(件/秒) (4.47)	8013(件/秒) (4.87)
	5	5183(件/秒) (3.15)	7399(件/秒) (4.50)	8588(件/秒) (5.22)	9222(件/秒) (5.61)	10054(件/秒) (6.11)

autovacuum処理時間とパラメータチューニング-⑨

- **検証2. パラメータ変更後の効果測定**

  autovacuum_vacuum_cost_delay=10

  autovacuum_vacuum_cost_limit=600



不要領域抑止、テーブル肥大化抑止

検知

待機イベントの監視方法と監視観点の紹介－①

- 発生回数の多い待機イベントの紹介と対処例

  - 待機イベント

WALWriteLock,WALWrite（WAL書き込みにて発生する待機イベント）

  - 待機内容

WALのディスクへの書き込みで待ちが発生しています。

  - 対処例

wal_buffersパラメータの値を増加させる、もしくはWALの出力先ディスクをより高速なディスクに変更するなどの対処を検討します。

※報告書では他4件の待機イベントと対処例をご紹介

検知

待機イベントの監視方法と監視観点の紹介－②

待機イベントを自動収集できる外部ツール「pg_wait_sampling」を利用した待機イベントの発生回数の確認方法と確認観点の紹介

サンプルSQL

```
=#
SELECT
    event_type,
    event,
    SUM(count)
FROM
    pg_wait_sampling_profile
WHERE
    event IS NOT NULL
GROUP BY
    event_type, event
ORDER BY
    sum(count) DESC;
```

実行結果

```
-[ RECORD 1 ]-------------------
event_type | LWLock
event | WALWriteLock
sum | 984
-[ RECORD 2 ]-------------------
event_type | Lock
event | transactionid
sum | 682
```

対処

ケーススタディについて

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

- 報告書では以下の4つの事例を紹介
  - インデックスの定義漏れによる性能トラブル
  - インデックスの肥大化・断片化による性能トラブル
  - 複合インデックスの定義順誤りによる性能トラブル
  - ロケール“C”以外でのインデックスの演算子クラスの定義漏れによる性能トラブル

次ページ以降で紹介

対処

複合インデックスの定義順誤りによる性能トラブル－①

■ テーブル、インデックス、SQLの概要

□ テーブル、インデックスの概要

全国にある事業所を、都道府県、市町村区分毎に管理

・・・・・	郵便番号	都道府県	市区町村	事業所名	・・・・・
・・・・・・	001-0000	北海道	札幌市	○○○○株式会社	・・・・・・
・・・・・・	201-0000	東京都	港区	株式会社△△△△	・・・・・・・
・・・・・・	401-0000	山梨県	甲府市	学校法人□□□□	
・・・・・・	601-0000	京都府	京都市	×××株式会社	

郵便番号、都道府県、事業所名の順番で複合インデックスを定義

□ 検索SQLの概要

都道府県が「東京都、神奈川県、埼玉県、千葉県」のいづれかに該当し
事業所名が「学校法人」で始まる行を検索

```
SELECT …  FROM “全国住所”
WHER “都道府県” IN (・・・) AND “事業所名” LIKE “学校法人%”
```

対処

複合インデックスの定義順誤りによる性能トラブル－②

■ トラブルの内容

データベースを運用していたところ、データの増加に伴い、参照SQLの処理遅延が発生し、ログファイルに遅延SQLとして出力された。

3秒以上を遅延とし、「log_min_duration_statement = 3s」を設定していたところ、以下のようなログ出力を確認した。

約5秒

```
LOG: duration: 5335.203 ms statement: SELECT "郵便番号", "都道府県", "市区町村", "事業所名"
FROM "全国住所" WHERE "都道府県" IN ('東京都', '神奈川県', '埼玉県', '千葉県') AND "事業所名" LIKE
'学校法人%';
```

対処

複合インデックスの定義順誤りによる性能トラブル－③

■ 調査と分析の進め方

1. 実行計画中の負荷の高いステップを確認
2. インデックスが効果的に使用できているかを確認

```
Index Scan using comb_idx on "全国住所" (cost=0.56..163403.17 rows=888 width=67)
(actual time=22.246..12028.735 rows=4230 loops=1)
  Index Cond: ((("事業所名")::text >= '学校法人'::text) AND (("事業所名")::text < '学校法?'::text))
  Filter: ((("事業所名")::text ~~ '学校法人%'::text) AND (("都道府県")::text = ANY ('{東京都,神奈川県,埼玉県,千葉県}'::text[])))
  Rows Removed by Filter: 4110
Planning Time: 0.264 ms
Execution Time: 12029.023 ms
(6 行)
```

Indexが事業所名の検索のみに使用されている

インデックススキャン(IndexScan)のcost値が高い

事業所名と都道府県を絞り込むFilterによって4110件除外している。

対象のSQLに対して複合インデックスが効率的に使用できていないのでは？

対処

複合インデックスの定義順誤りによる性能トラブル－④

■ 改善の前に

複合インデックスが効率的に使用される・されないケースの確認

- 複合インデックスが効率よく使用**される**ケース
  - 先頭列（左側）で定義した列から検索結果の絞り込みを行う検索

- 複合インデックスが効率よく使用**されない**ケース
  - 先頭列で定義した列で検索結果の絞り込みを行わない検索
  - 検索条件に含まれない列を複合インデックスで定義

その結果。。。。

実行計画で複合インデックスが使用されていても効率の悪い検索となる。
複合インデックスは使用されずシーケンシャルスキャンとなる場合が多い。

対処

複合インデックスの定義順誤りによる性能トラブル－⑤

■ 改善手法の紹介

“都道府県”, “事業所名”をインデックスの先頭列に定義する。
他のSQLへの影響を考慮して、複合インデックス修正か追加を判断する。

- 複合インデックスを修正

他のSQLへの影響が無い場合に有効

- 複合インデックスを追加

他のSQLでは効率的に使用されているのでインデックスを修正できない場合に有効

対処

複合インデックスの定義順誤りによる性能トラブル－⑥

■ 改善結果の確認

改善前：約5秒
改善後：約5ミリ秒

複合インデックスを修正した場合の例

```
Index Scan using comb_idx on "全国住所" (cost=0.56..22.29 rows=933 width=67)
(actualtime=0.052..5.335 rows=4230 loops=1)
    Index Cond: (("都道府県")::text = ANY ('{東京都,神奈川県,埼玉県,千葉
県}'::text[])) AND(("事業所名")::text >= '学校法人'::text) AND (("事業所名")::text < '
学校法?'::text))
    Filter: (("事業所名")::text ~~ '学校法人%'::text)
Planning Time: 0.151 ms
Execution Time: 5.585 ms
(5 行)
```

インデックスアクセスのみで都道府県と事業所の絞り込みを実施

インデックススキャン(IndexScan)のcost値が低下

テーブルアクセスによるフィルタ処理が無くなった

まとめと所感

予防

- パラメータ設定の指針、運用に関する情報を追加した。
- 特に、autovacuumパラメータ調整等の性能影響調査は、運用設計で考慮すべき事項であるにも関わらず現状では情報が不足していたため、実機検証に基づいた設定例を紹介できたことは大きな成果と考えている。

検知

- 待機イベントの検知と対処例を追加した。監視設計の参考にして欲しい。

対処

- 昨年同様、公開可能なケーススタディの洗い出しと事象の再現に苦労した。

性能トラブル対処に関するノウハウの収集を継続します！
ノウハウが不足する分野などあれば、アンケートにご記入下さい。

PGECons

PostgreSQL Enterprise Consortium

付録

coffee break: WMIとは

- Windows Management Instrumentationの略称
- Windowsシステムの構成要素の情報収集や各種通知をするインターフェイス
  - https://docs.microsoft.com/en-us/windows/desktop/wmisdk/wmi-start-page

Prometheus server/exporterの起動

- GitHubからバイナリを取得してコマンドラインで起動するのみ
  - Prometheus: https://github.com/prometheus/prometheus/releases
  - wmi_exporter: https://github.com/martinlindhe/wmi_exporter/releases
  - postgres_exporter: https://github.com/wrouesnel/postgres_exporter/releases
- 実際のコマンドや起動時の設定は成果物を参照
  - 成果物参照リンク