PGECons
PostgreSQL Enterprise Consortium

PostgreSQL の標準的ベンチマークツール pgbench について

2018/10/18

SRA OSS, Inc. 日本支社　近藤雄太

アジェンダ

- pgbench とは ― 概要と特性
- 基本的な使い方 ― 前準備からベンチマーク実行まで
- 応用的な使い方 ― 検証条件の多彩な設定
- PGECons での知見 ― その他のツールとの連携
- 質疑応答 ― 皆さんからの質問にお答えします!

pgbench とは

- PostgreSQL に付属する性能検査ツール
  - a simple TPC-B like benchmark program for PostgreSQL

TPC-B
TPCという団体が定めるデータベースのベンチマークにおける各種指標の一つ。
銀行の窓口業務をモデルにしている

- 性能の指標
  - 単位時間あたりの処理能力(スループット)
    - pgbench では動作中のDBに対してSQLシナリオ(トランザクション)を大量に発行して実際に実行されたトランザクション数/秒 [TPS] を計っている

pgbench とは - 性能値の比較対象

- 異なるハードウェア間
- 異なるPostgreSQL設定間
- 異なるバージョンのPostgreSQL間

- ×異なるRDB間
  - PostgreSQL vs MySQL などは不可能

pgbench とは - 何ができる？

- 実際のDBにSQLを実行させた性能実測値を取得
- 40もの豊富なオプション機能
  - ベンチマーク実行の条件を細かく設定できる
  - 実行結果の統計を取得できる

pgbench とは - 何ができない？

- 運用中DBのリアルタイム性能値取得はできない
  - バックエンドで動かし続けて、任意の時刻における性能値を取得するといったことはできない
- 他のRDBとの性能比較はできない
  - pgbench は PostgreSQL専用
- 性能差の原因を直接調査できない

pgbench とは - 歴史

- 1999 年 pgbench 1.0 リリース
  - 弊社 (SRA OSS, Inc. 日本支社) 代表石井が作成
  - PostgreSQL 7.0 から導入

- 2016 年 PostgreSQL 9.5 にて追加提供モジュール扱いから標準コマンドに昇格

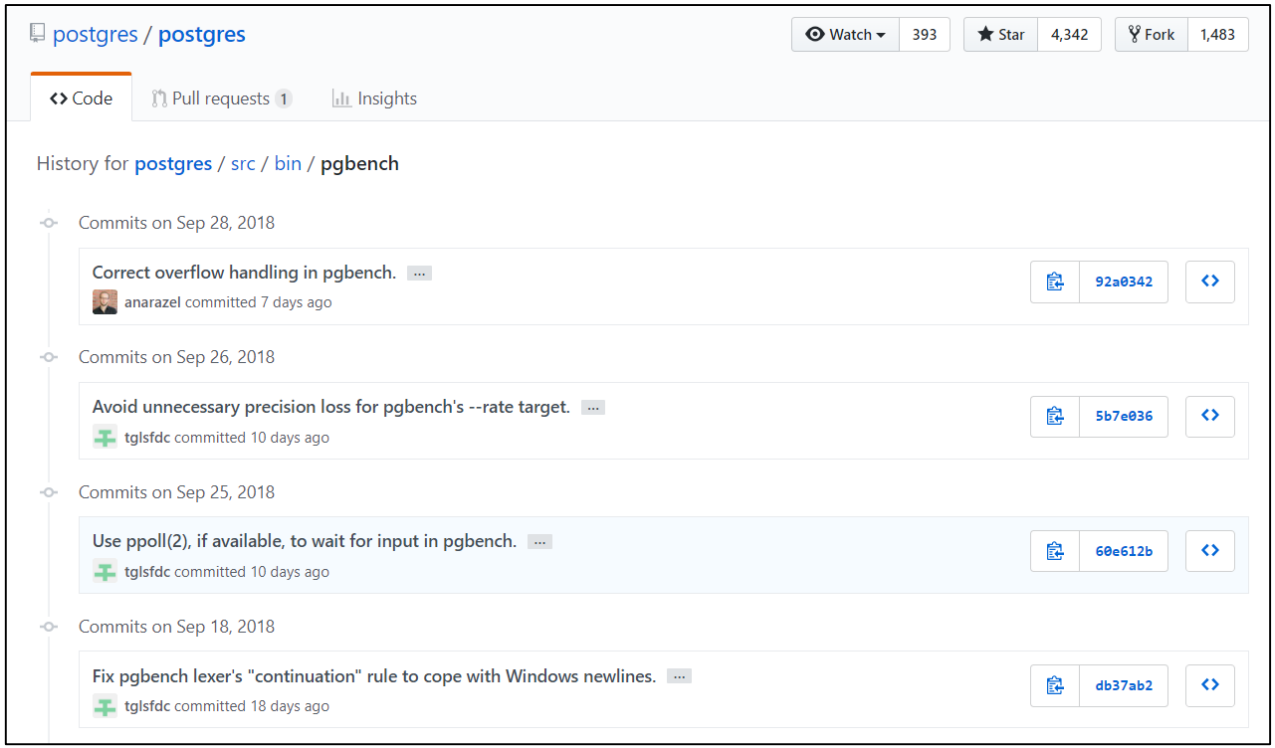
> E.15.3.9.5. pgbench
>
> - Move pgbench from contrib to src/bin (Peter Eisentraut)

https://www.postgresql.org/docs/9.5/static/release-9-5.html

PostgreSQL 標準ベンチマークツールとして認められた

pgbench とは - 歴史

- 2018 年現在も開発継続中

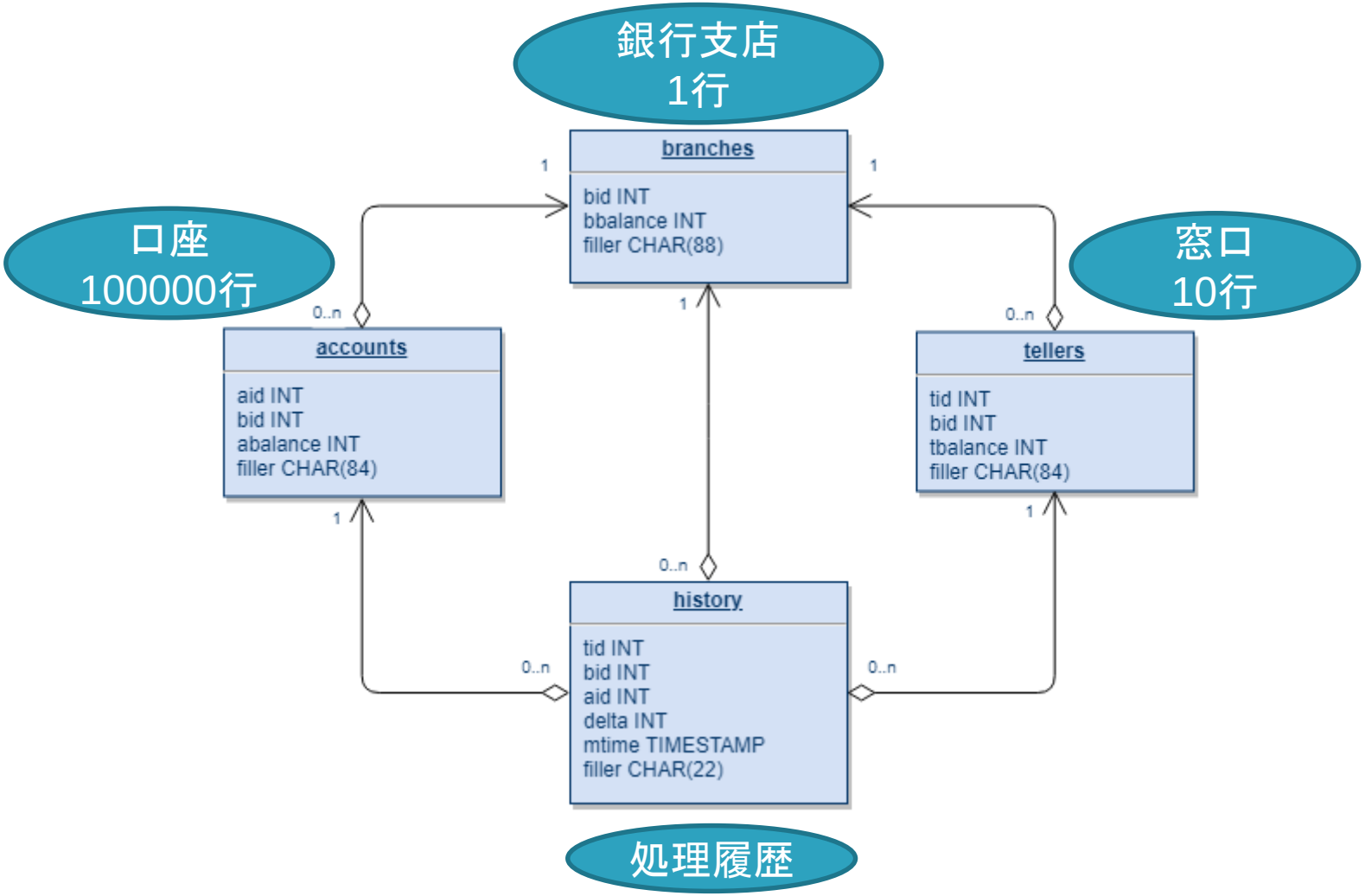


基本的な使い方 - pgbench 利用の流れ

- SQLシナリオの作成
  - ビルトインシナリオを使用するのであれば不要
- ベンチマークの操作対象となるテーブルの準備
  - ビルトインシナリオを使用するのであれば pgbench -i で標準テーブルを用意してくれる
- ベンチマーク実行
  - オプションに条件を指定して pgbench 実行
    - pgbench 用の設定ファイルはない

基本的な使い方 – 4つの標準テーブル

- ビルトインシナリオに利用される



基本的な使い方 – ビルトインシナリオ

- ビルトインシナリオは 3 つ
  - tpcb-like ※デフォルト
  - simple-update
  - select-only

- -b list でビルトインシナリオ一覧を取得可能

```
$ pgbench -b list
Available builtin scripts:
        tpcb-like
        simple-update
        select-only
```

基本的な使い方 – ビルトインシナリオ

- tpcb-like
  - デフォルトシナリオ

1トランザクションのたび変数は以下の通りセットされる
¥set aid random(1, 100000 * :scale)
¥set bid random(1, 1 * :scale)
¥set tid random(1, 10 * :scale)
¥set delta random(-5000, 5000)

(1) BEGIN;

(2) UPDATE accounts SET abalance = abalance + :delta WHERE aid = :aid;
- 口座に預け入れ(もしく引き出し)
(3) SELECT abalance FROM accounts WHERE aid = :aid;
- 口座の残高照会
(4) UPDATE tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
- 窓口の預かり金額更新
(5) UPDATE branches SET bbalance = bbalance + :delta WHERE bid = :bid;
- 支店の預かり金額更新
(6) INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
- 処理した口座、窓口、支店、金額、時刻を履歴に残す
(7) END;

基本的な使い方 – ビルトインシナリオ

- simple-update
  - デフォルトシナリオの UPDATE を 1 回だけにしたシナリオ
  - -b simple-update, --builtin=simple-update,
    -N, --skip-some-updates のいずれかで指定

```
(1) BEGIN;

(2) UPDATE accounts SET abalance = abalance + :delta WHERE aid = :aid;

(3) SELECT abalance FROM accounts WHERE aid = :aid;

(4) INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES
(:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);

(5) END;
```

基本的な使い方 – ビルトインシナリオ

- select-only
  - デフォルトシナリオのUPDATE, INSERTを取り除いたシナリオ
  - -b select-only, --builtin=select-only,
    -S, --select-only のいずれかで指定

```
(1) BEGIN;

(2) SELECT abalance FROM accounts WHERE aid = :aid;

(3) END;
```

基本的な使い方 – デモ

- pgbench -i, pgbench 実行

```
[postgres@localhost ~]$ createdb pgbench
[postgres@localhost ~]$ pgbench -i pgbench
dropping old tables...
NOTICE:  table "pgbench_accounts" does not exist, skipping
NOTICE:  table "pgbench_branches" does not exist, skipping
NOTICE:  table "pgbench_history" does not exist, skipping
NOTICE:  table "pgbench_tellers" does not exist, skipping
creating tables...
generating data...
100000 of 100000 tuples (100%) done (elapsed 0.19 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.
[postgres@localhost ~]$ pgbench pgbench
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
[postgres@localhost ~]$
```

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- transaction type
  - 実行したSQLシナリオの名前
  - カスタムSQLシナリオを指定していればそのファイル名
  - デフォルトは“<builtin: TPC-B (sort of)>“

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- scaling factor
  - 標準テーブルのサイズ係数
  - -s,--scale で指定可能。デフォルトは1
  - 1 だと 15MB, 1000 で 15GB 程度

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- query mode
  - サーバへ問い合わせを送信するために使用するプロトコル
    - 簡易問い合わせ、拡張問い合わせ、プリペアードステートメントを伴う拡張問い合わせの3種類がある
  - -M, --protocol で指定可能。

    デフォルトは“simple”（簡易問い合わせ）

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- number of clients
  - 同時接続クライアント数
  - -c, --client で指定可能。デフォルトは 1

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- number of threads
  - pgbench の同時実行スレッド数
  - -j, --jobs で指定可能。デフォルトは 1

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- number of transactions per client
  - 1クライアントあたりの実行トランザクション数

    -t, --transactions で指定可能。デフォルトは 10
- number of transactions actually processed
  - （実際に実行されたトランザクション数）

    /（予定されたトランザクション数）

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- latency average
  - 1トランザクションの平均実行時間
  - 単位は ms
  - 9.6 から実装された。9.5以前では表示されないので注意

基本的な使い方 – 結果の読み方

```
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 10
number of transactions actually processed: 10/10
latency average = 4.433 ms
tps = 225.580586 (including connections establishing)
tps = 241.413390 (excluding connections establishing)
```

- tps
  - including connections establishing
    - 接続確立にかかった時間を考慮した実行トランザクション数/秒
  - excluding connections establishing
    - 接続確立にかかった時間を考慮しない実行トランザクション数/秒

デフォルトでは各クライアントは一度だけ接続を確立する。
トランザクションごとに新しい接続を確立するには -C, --connect オプションを用いる

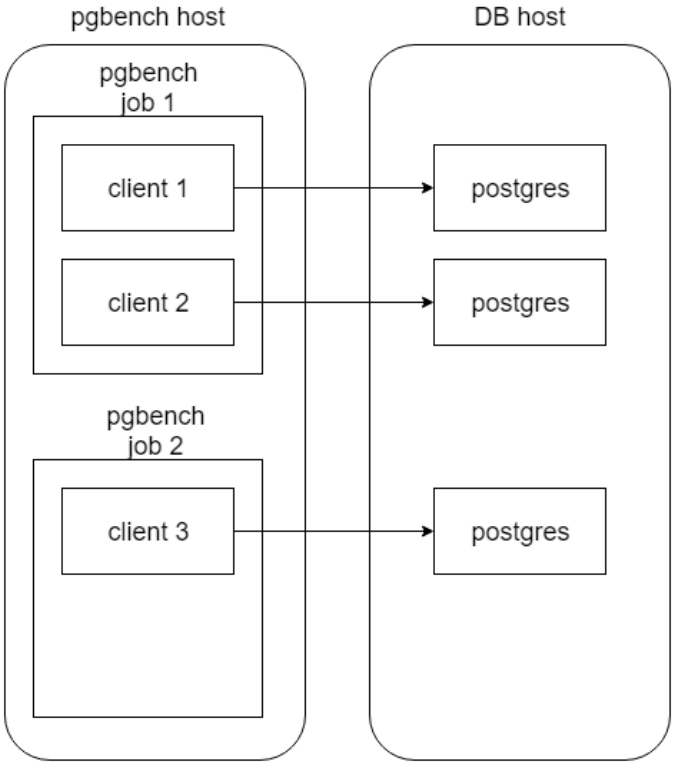
基本的な使い方 – 重要なオプション

- 実行時間（-T）
  - 実行時間を5分以上で指定しましょう
    - デフォルトではトランザクションが10回しか実行されずTPSが不正確な値となりやすい
  - 実行時間指定が厳密に適用されるようになったのは9.6から
    - 9.5以前は指定時間内に実行が開始されたトランザクションが完了するまでベンチマークの完了も待たされていた

基本的な使い方 – 重要なオプション

- 同時接続クライアント数(-c)、ジョブ数(-j)
  - 複数ユーザの同時接続をシミュレートするために必須
  - クライアントとジョブの関係

クライアント数 3, ジョブ数 2 のとき



- ジョブは pgbench が生成するPOSIXスレッド
- ジョブ数は pgbench ホストの並列実行数
- クライアント数は DB ホストの並列実行数

- クライアントは各ジョブ上で実行される

そのため、ジョブ数はクライアント数の約数にすること

  - さもなくば各ジョブのクライアント数が異なることになる
  - 9.5 より前は約数にしなければそもそも動作しなかった

- 理想はジョブ数=クライアント数
  - CPUコア数が足りずジョブ数を減らすことがほとんど

基本的な使い方 – 重要なオプション

- カスタムシナリオ（カスタムクエリ）(-f)
  - 任意のSQLシナリオを実行することが可能
  - SQL以外にも pgbench が利用できるコマンド
    - ¥set 変数に値を代入
    - ¥sleep 指定した時間だけ待機。us, ms, s 単位で指定可能
  - 注意
    - ベンチマークに標準テーブルを用いないのであれば、pgbench -i の代わりの初期化操作が必要になる
    - 同一シナリオを実行していなければ性能比較はできない

基本的な使い方 – (ごく基本の) チューニング

- 検索キーに関するインデックスの作成
  - 性能が1万倍変わることも
- postgresql.conf のパラメータを変更する
  - shared_buffers を増やす
    - Linux では搭載メモリの 1/4 を推奨
    - Windows では効果が薄い
- ハードウェアの増強 (CPU, メモリ等)
  - ただし、リソースの値が倍になってもDB性能が倍になることは基本的にない

基本的な使い方 – 使用時の注意点

- なるべく新しい pgbench を用いる
  - 9.6 より前は -T による実行時間の指定が厳密でない
  - 9.5 より前は tps (excluding connections establishing) の計算が不正確

※ただし、比較対象が古いpgbenchを利用している場合はそちらに合わせる

基本的な使い方 - 使用時の注意点

- ベンチマーク実行対象テーブルの状態を把握しておく
  - 更新系のシナリオを実行すると実行前と異なる状態になる
    - 事前もしくは実行中のバキュームの有無は意図しておく
    - 事前バキュームを実行させない場合は -n を指定
    - 実行中に自動バキュームを実行されないようにするなら autovacuum = off としておく

- pgbench クライアントと PostgreSQL サーバは分離
  - pgbench プロセスの存在が結果に影響してしまう場合がある

基本的な使い方 - 使用時の注意点(うっかり編)

- 初期化(pgbench -i)せずに pgbench 実行
- -d はデータベースの指定オプションではない
  - -d はデバッグ用出力。データベースの指定は通常引数
- postgres データベースに初期化操作
  - そのまま利用してもOKだけど
- 短い statement_timeout 設定
  - 初期化が失敗したり、ベンチマーク結果が意図しないものとなる
- log_statement = all
  - 大量のログが吐かれる上、結果にも影響が出る

応用的な使い方 – 様々なオプション

- マルチシナリオ実行
  - 複数シナリオを実行可能
  - 各シナリオが実行される割合は @ で示した重みで決定

```
$ pgbench -b tpcb-like@4 -f custom.sql@1 pgbench -t 1000

transaction type: multiple scripts
..
tps = 513.959420 (including connections establishing)
tps = 514.900247 (excluding connections establishing)
SQL script 1: <builtin: TPC-B (sort of)>
 - weight: 4 (targets 80.0% of total)
 - 811 transactions (81.1% of total, tps = 416.821090)
 - latency average = 2.288 ms
 - latency stddev = 0.690 ms
SQL script 2: custom.sql
 - weight: 1 (targets 20.0% of total)
 - 189 transactions (18.9% of total, tps = 97.138330)
 - latency average = 0.458 ms
 - latency stddev = 0.382 ms
```

応用的な使い方 – 様々なオプション

- ステートメントごとのレイテンシ出力（-r, --report-latencies）
  - トランザクション内で遅いクエリを特定できる

```
tps = 618.764555 (including connections establishing)
tps = 622.977698 (excluding connections establishing)
script statistics:
 - statement latencies in milliseconds:
        0.002  ¥set aid random(1, 100000 * :scale)
        0.005  ¥set bid random(1, 1 * :scale)
        0.002  ¥set tid random(1, 10 * :scale)
        0.001  ¥set delta random(-5000, 5000)
        0.326  BEGIN;
        0.603  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
        0.454  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
        5.528  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
        7.335  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
        0.371  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIMESTAMP);
        1.212  END;
```

応用的な使い方 – 様々なオプション

- トランザクションごとのログ出力（-l（エル），--log）
  - 生データを取るならこのオプション
  - 出力される値の意味（左から）
    - クライアントID
    - トランザクションID
    - トランザクション処理時間（マイクロ秒）
    - スクリプトファイル番号
    - トランザクション完了エポック時刻（秒）
    - トランザクション完了エポック時刻（マイクロ秒オフセット）

```
$ cat pgbench_log.32317
0 1 30423 0 1539319168 31146
0 2 2226 0 1539319168 33368
…
```

応用的な使い方 - PG11 からの新オプション

- ¥if
  - カスタムシナリオに if 文が利用できるようになった
- -I (アイ), --init-steps
  - -i よりも詳細に初期化操作を指定できるようになった
  - 指定可能な操作
    - d (Drop)
    - t (create Tables)
    - g (Generate data)
    - v (Vacuum)
    - p (create Primary keys)
    - f (create Foreign keys)

    デフォルトは --init-steps=dtgvp (-i と同じ操作)

PGEConsでの知見 - PGECons について

- PostgreSQLエンタープライズコンソーシアム
（略称 PGECons）

  - 団体の目的

    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、エンタープライズ領域へのPostgreSQLの普及を推進すること

- 活動の一つとして、各ワーキンググループによるPostgreSQL検証を行なっています。

PGEConsでの知見 - ワーキンググループについて

- 現在 3 つのワーキンググループにて活動中
  - WG1(新技術検証ワーキンググループ)
    - 新バージョンの性能や新技術の検証を通じて有用性を明確化
    - スケールアップ検証、新機能における性能特性調査
  - WG2(移行ワーキンググループ)
    - 異種DBMSからの移行をテーマに活動
    - 商用DBMSからの移行プロセスに伴う技術調査や検証を実施
  - WG3(課題検討ワーキンググループ)
    - データベース管理者やアプリケーション開発者が抱える現場の課題や困り事に対するテーマを設定
    - 可用性・運用性・保守性・セキュリティ・接続性が主な課題領域

- 年に 1 回、成果報告会を開いています

PGEConsでの知見 - WG1での pgbench 利用

- PGECons の新技術検証グループ(WG1)では多くの検証で pgbench を用いています
  - 定点観測検証（スケール性能検証）
  - Windows 検証
  - パラレルクエリ検証
  - Pgpool-II 検証

ここからは、定点観測検証を例に挙げて説明します

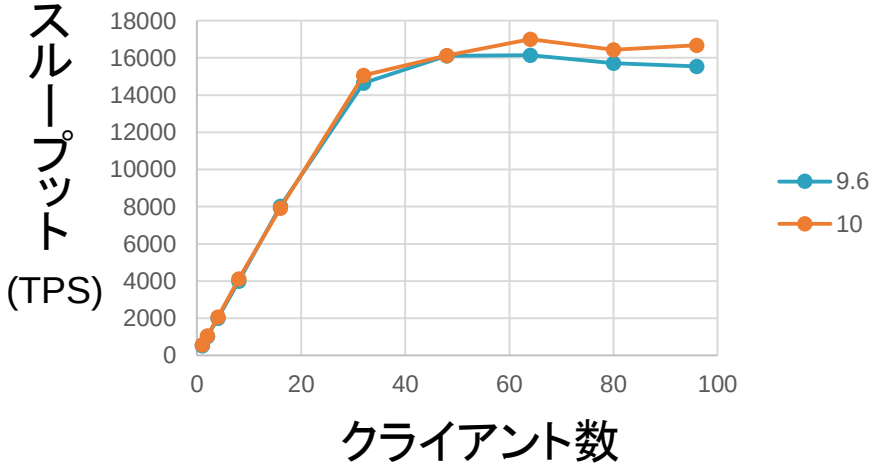
PGEConsでの知見 – 定点観測検証

- 目的
  - 新バージョンの PostgreSQLの性能改善の傾向を知る
    - PGECons発足当初（2012年度、PostgreSQL 9.2）から継続的に実施（定点観測）
    - 直前のバージョンとの性能比較

pgbench を PostgreSQL バージョン間の性能変化を観察するために用いている

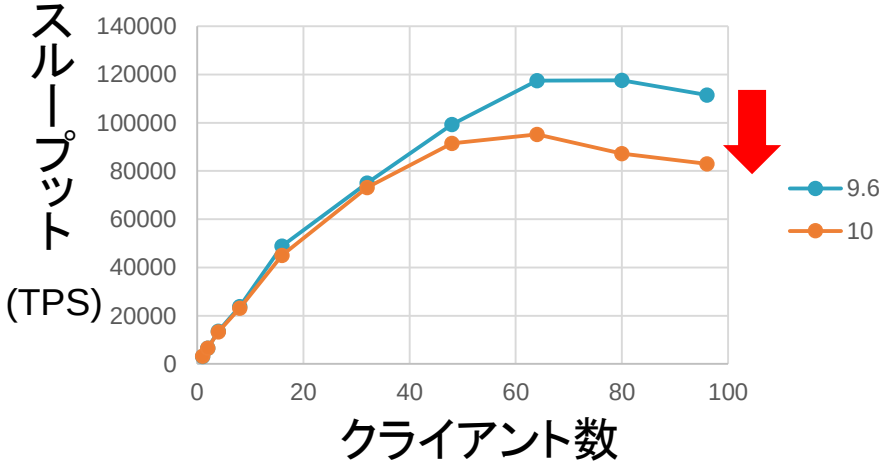
PGEConsでの知見 - 定点観測検証

- 条件（2017年度）
  - 参照性能と更新性能のそれぞれを検証
  - DBデータをオンメモリで走行（ストレージIOによるブレを最小限にする意図）
    - 30GB（スケールファクタ 2000 のデータ）/384GB（搭載メモリ）
    - 事前に pg_prewarm（DBデータをメモリに読み込むツール）を実行
  - クライアント数を変化させスループットを測定（スケーラビリティを観察）
    - PG 9.6 vs 10［参照性能］



PGEConsでの知見 - PG 9.6 vs 10 測定結果

- 例: PG 9.6 vs 10 [更新性能]
  - 9.6に比べて10は明らかな更新性能低下が見られた



更新性能劣化の原因は何？

PGEConsでの知見 - 定点観測検証

- （おさらい） pgbench ではできないこと
  - 性能差の原因を直接調査できない

- バージョン間性能差の原因を明らかにするには
他の性能解析ツールとの連携が必要

  今回は以下の2つをご紹介
  - sar（リソースの状態取得ツール）
  - strace（システムコールトレーサ）

PGEConsでの知見 - sar

- システムリソースの状態情報を取得
  - CPU（使用率、ロード）
  - メモリ（使用量）
  - ディスク（I/O, スワップ）
  - ネットワーク（トラフィック）

同様の情報を取得する他ツール
: top, vmstat
: top, vmstat, free
: iostat
: tcpdump

- リアルタイムに状態を見るよりも、後から過去の状態を確認する場合に用いる事が多い
  - 定期実行されレポートが作成されている
    - sar (sysstat) がインストールされている環境
    - cron スクリプト /etc/cron.d/sysstat
    - レポートデータ /var/log/sa/saDD（DD は2桁の日付）

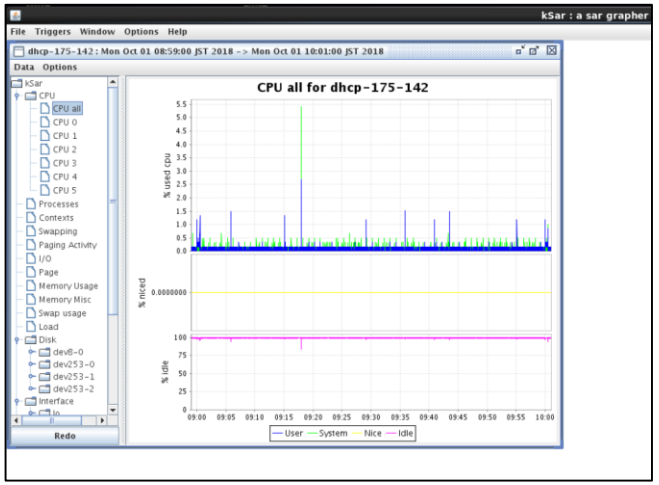
PGEConsでの知見 - sar

CPUに関する情報の取得例

```
# sar 1 5 # 1秒間のデータを5回分取得
Linux 3.10.0-514.2.2.el7.x86_64 (localhost.localdomain)       2018年10月11日  _x86_64_       (2 CPU)

14時56分51秒     CPU     %user     %nice   %system   %iowait    %steal     %idle
14時56分52秒     all      0.00      0.00      0.00      0.00      0.00    100.00
14時56分53秒     all      0.00      0.00      0.48      0.00      0.00     99.52
14時56分54秒     all      0.00      0.00      0.00      0.00      0.00    100.00
14時56分55秒     all      0.00      0.00      0.50      0.00      0.00     99.50
14時56分56秒     all      0.50      0.00      0.50      0.00      0.00     99.00
平均値:      all      0.10      0.00      0.30      0.00      0.00     99.61
```

- グラフ化ツール
  - kSar



PGECons での知見 - sar

- sar のレポートから PostgreSQL の処理負荷がサーバのリソース観点で比較観察できる
  - どこかの項目に大きな差があれば性能差の原因調査のヒントになりうる

しかし、定点観測検証では sar データだけでは足りない

- PostgreSQLバージョン間の性能差の原因を追うのであればプロセスの具体的な処理を直接見る必要がある

→ 例えば strace

PGEConsでの知見 - strace

- プロセスが実行したシステムコールの検知ツール
  - プロセスが呼び出したシステムコールを出力

```
$ strace -p《任意のプロセスID》
Process 4738 attached
select(6, [3 4 5], NULL, NULL, {53, 752537}) = ? ERESTARTNOHAND (To be
restarted if no handler)
…(省略)…
```

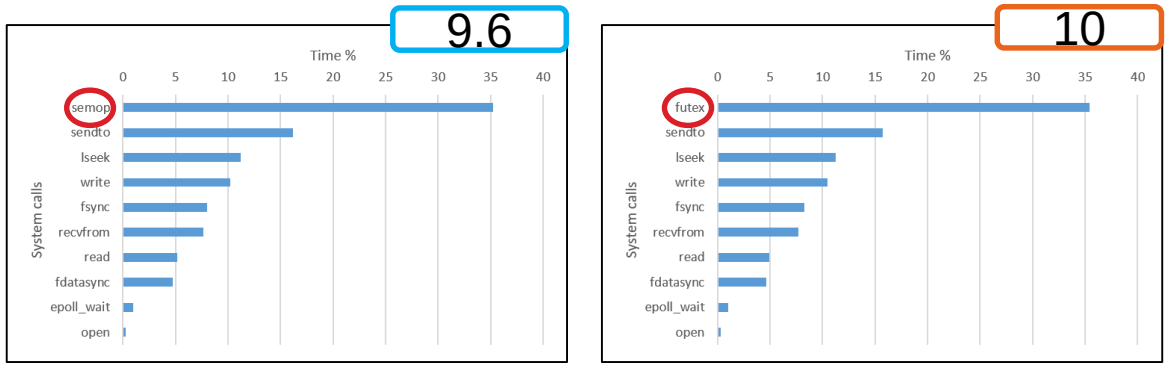
  - プロセスが呼び出した各システムコールの全体の実行時間割合、実行回数など出力

```
$ strace -p《任意のプロセスID》 -c
Process 4738 attached
^C
$ Process 4738 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 75.71    0.000187         187         1           clone
…(省略)…
  0.40    0.000001           1         2         2 rt_sigreturn
------ ----------- ----------- --------- --------- ----------------
100.00    0.000247                    17         6 total
```

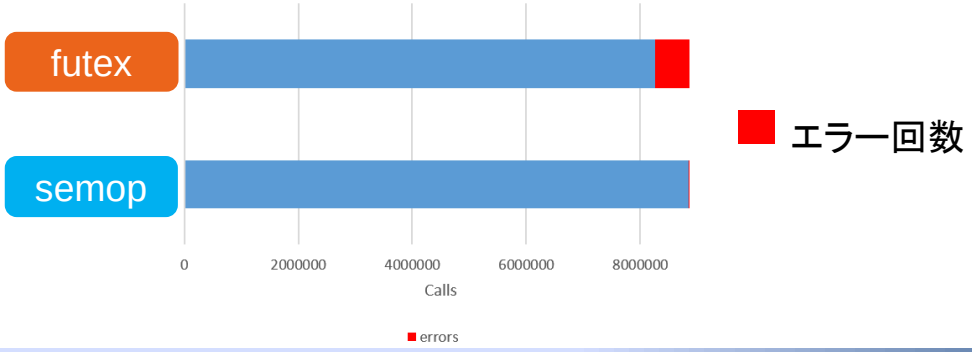
PGEConsでの知見 - strace

- PG 9.6 vs 10
  - postmaster プロセスを指定(-p)
  - 子プロセス以下も対象 (-f)
  - 各システムコールの実行時間割合を出力 (-c)
  - 結果をファイルに出力(-o)

```
$ cat ${PGDATA}/postmaster.pid | head -n1 | xargs -I@ strace -p @ -f -c -o《出力先ファイル》
```



実行時間割合の大きいシステムコールが異なる



エラー回数

実行時間割合の大きいシステムコールのエラー率が異なる

PGEConsでの知見 - strace

- PG 9.6 vs 10（もう一段階 strace で検証）
  - システムコール名を指定して出力を絞り込み（-e）
  - システムコールを実行したソースをトレース（-k）

    ※ -k オプションは多くのOSで実装されておらず、利用するためには strace のソースコードを手動でビルドする必要がある

$ strace … -k -e semop　　9.6

```
semop(5472262, [{3, -1, 0}], 1)         = 0
 > /usr/lib64/libc-2.17.so(semop+0x7) [0xf8f37]
 > /usr/local/pgsql/9.6.6/bin/postgres(PGSemaphoreLock+0x41) [0x24f441]
 >…

semop(5505031, [{1, 1, 0}], 1)          = 0
 > /usr/lib64/libc-2.17.so(semop+0x7) [0xf8f37]
 > /usr/local/pgsql/9.6.6/bin/postgres(PGSemaphoreUnlock+0x41) [0x24f4c1]
 >…
```

$ strace … -k -e futex　　10

```
futex(0x7f7f17ddfc98, FUTEX_WAIT, 0, NULL) = 0
 > /usr/lib64/libpthread-2.17.so(do_futex_wait.constprop.1+0x2b) [0xd79b]
 > /usr/lib64/libpthread-2.17.so(__new_sem_wait_slow.constprop.0+0x4f) [0xd82f]
 > /usr/lib64/libpthread-2.17.so(sem_wait+0x2b) [0xd8cb]
 > /usr/local/pgsql/10.1/bin/postgres(PGSemaphoreLock+0x22) [0x284e92]
 >…
futex(0x7f7f17ddfe58, FUTEX_WAKE, 1)    = 1
 > /usr/lib64/libpthread-2.17.so(sem_post+0x3d) [0xdb7d]
 > /usr/local/pgsql/10.1/bin/postgres(PGSemaphoreUnlock+0x22) [0x284ef2]
 >…
```

セマフォに関する関数で用いられているシステムコールの変更が
更新性能に影響していることを示唆

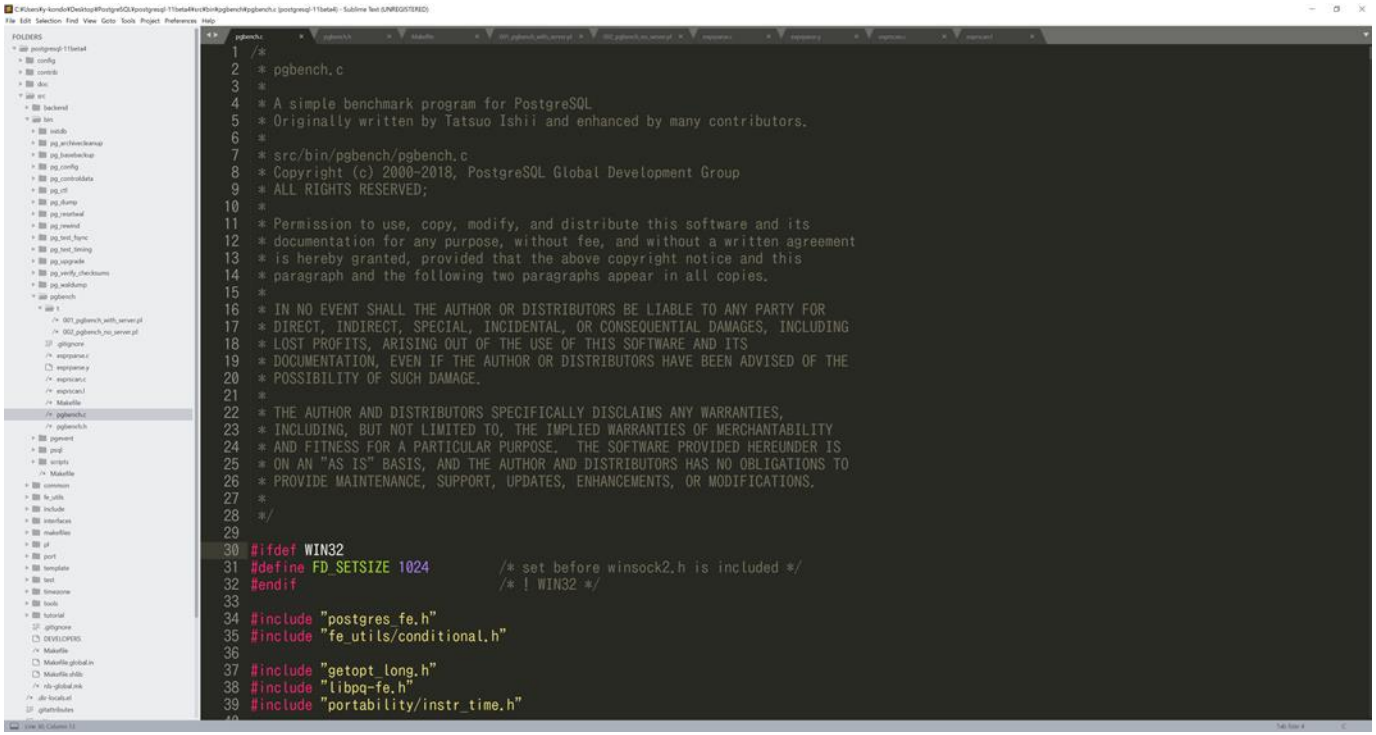
おわりに - pgbench関連リンク

- pgbenchの使いこなし（Let's Postgres）
  - https://lets.postgresql.jp/documents/technical/contrib/pgbench
  - pgbench 作者が解説
  - PostgreSQL 8.4 時代なので少々内容が古い
- PostgreSQL 10 文書
  - https://www.postgresql.jp/document/10/html/pgbench.html
- PostgreSQL 11 文書（beta, 英語）
  - https://www.postgresql.org/docs/11/static/pgbench.html

おわりに

- pgbench をもっと知りたい！という方は

ぜひソースコード src/bin/pgbench 以下を読んでください



ご清聴ありがとうございました

質疑応答