実践テクニックをご紹介　新連載

徒然PostgreSQL散策

第1回　火消し的PostgreSQLチューニングテクニック

日本PostgreSQLユーザ会 理事長　石井 達夫
ISHII Tatsuo●ishii@postgresql.jp

はじめに

読者のみなさんこんにちは，『WEB+DB PRESS』では半年以上のご無沙汰になってしまいました，石井達夫と申します．

前回本誌でみなさんにお目にかかったのは「さわって実感 リレーショナルデータベース設計」という，データベースの基礎からアプリケーション構築まで8回に分けてご紹介するという企画でした注1．

前回の連載では「データベースの正規化とは何か」のような，ともすると「難しそう」と敬遠されがちな話題（実際難しいのですが:-)）をできるだけわかりやすく，かつみなさんがあくびをしないで済むように手を動かしつつ体験していただく，というのが一番苦労したポイントでした．おかげさまでこの点に関してはご好評をいただいたようで，ほっとしています．

実は前回の連載を終えたあと担当編集のTさんからは「すぐにでも次の連載を…」というお話もあったのですが，筆者も多忙なのと充電期間が欲しかったので，ちょっとお休みをいただいて，その間に次の企画を考えることにしました．

筆者は血液型がB型のせいもあり:-)，もともと緻密に計画をたてて物事を進めるのが大の苦手です．前回の連載は設計技法とか正規化の話をする都合上，筆者にしては珍しくちゃんと事前に計画を練って記事を書いていったつもりです注2．

でも，そういう進め方は疲れるんですね．やっぱり．そこで今回の連載では，

- いきあたりばったり書く
- おもしろそうなことを思いつくままに書く
- なぜか今まであまり触れられていない話題をあえて書く

ということを基本コンセプトに設定させていただくことにしました．したがって「全6回の連載計画」も提示できません．みなさま申し訳ありません．一応話題はPostgreSQL関係に限定しますが，どんな脱線があるかわかりませんで，その節はお許しください．

第1回目の話題は？

そうは言ってもいきなり連載1回目から与太話では連載が打ち切りになってしまう危険性があるので，まず出だしは「火消し的PostgreSQLチューニングテクニック」，すなわち修羅場で役に立つパフォーマンスチューニングテクニックについてお話しすることにしました．

パフォーマンスチューニングとは

ボトルネックを排除し，データベースの性能を高める作業を一般に「データベースパフォーマンスチューニング」と呼びます．このような作業はデータベースに限らずネットワークやアプリケーションサーバなど，いろいろな適用分野があります．でも実際には「パフ

注1）このときの連載は一部加筆訂正（たとえば対応PostgreSQL，PHPのバージョンを最新にするなど）した上で『PHPxPostgreSQLで作る最強Webシステム』（技術評論社）という書籍の一部に納められています．
注2）「だったらなんで連載予定数を2回もオーバーしたんだ」などという突っ込みは禁止です:-)

ォーマンスチューニング」と言うと，ほとんど「データベースパフォーマンスチューニング」のことを指していると思って間違いないくらい，データベースのパフォーマンスチューニングは話題に上る機会が多いようです．そのくらい使い方で性能が変わるのがデータベースであり，また性能を引き出すのが難しいのがデータベースであるとも言えます．

パフォーマンス設計とは

本来パフォーマンスチューニングをきちんと行うためには，ハードウェアの選定やデータベースの設計，アプリケーションの設計などを事前に綿密に行っておかなければなりません．これを「パフォーマンス設計」と呼びます．

パフォーマンス設計では，少なくとも以下のようなステップが必要になります[注3]（もちろん，システムや機能の設計はある程度終わっているものとします）．

① 業務要件から必要な性能を割り出す．たとえば「レスポンスタイムは3秒以内」など
② システムにかかる負荷を見積もる．たとえば，「1時間に1万回のアクセスがあり，ピーク時にはその3倍のアクセスがある」など
③ アプリケーションがデータベースにアクセスするパターンを調べ，各テーブルへのアクセス頻度を見積もる
④ ②③からPostgreSQLへの性能要件が決まるので，その性能要件を満たすハードウェアを選定する．パラメータとしては，以下のようなものがある．
- CPUのクロック，個数
- メモリの搭載量
- ハードディスクの性能

⑤ できるだけ実際の状態に近いテストプログラムとテストデータを使って，実際にこのハードウェアで性能が満たされているかどうか検証する．満たされていない場合は①に戻り再設計を行う

このようなステップを経て十分な性能を持つシステムが完成し，スムースに運用を開始できるのが理想ですが，筆者の経験では，ほとんどの場合こうなりません．その理由はいろいろあります．

- 納期が短すぎてそんなことをやっている暇がない
- 機能の実現で手一杯で，性能のことを考えている余裕がない
- 予算が少ないのでこのような作業ができるエキスパートが雇えない
- 性能要件を満たすようにアプリケーションやテーブルの設計をやり直す工期と予算の余裕がない
- 政治的な理由から採用するハードウェアがすでに決まっていて動かせない，あるいは予算の都合上あまったハードの流用を強いられる
- メーカが非協力的で，試験機での十分な検証ができない

その結果，カットオーバ間際になってレスポンスが出ないことが発覚して大騒ぎになるのはまだよいほうで，そのままサービスインした結果，過負荷のためにシステムが「メルトダウン」してしまうケースさえ見受けられます．

火事場での火消し的パフォーマンスチューニングテクニックとは

このような状態になると現場は大混乱に陥り，システムの各パート責任者が呼びつけられて事態の収拾に奮闘することになります．中には「データベースをパフォーマンスチューニングすると，アプリケーションを修正しなくても性能が10倍上がるそうじゃないか．今すぐ手を付けてくれ」と無茶なことを言うプロジェクトリーダもいるかもしれません．まことに無体な話ですが，それが現実というものです．

そこで本稿では，不幸にもこういうプロジェクトのデータベース担当者になってしまった読者のために，火事場での火消しに役立つPostgreSQLのパフォーマンスチューニングテクニックを伝授することにしました．

火消しテクニックその１：deadlock_timeoutを見直そう

deadlock_timeoutはPostgreSQLの設定ファイルで

注3）ここでは話を簡単にするために，ネットワークの性能やアプリケーションサーバの性能などの，実際には非常に重要な要件を省いています．

あるpostgresql.confの設定項目の1つです.

デッドロックとは

デッドロックは，複数のトランザクションが直接あるいは間接的にお互いに相手がロックを解放するのを待つために発生します.

例を挙げましょう．2つのテーブルt1とt2があり，2つのトランザクションが以下のような操作を行ったとします．なお，以下でT1やT2はトランザクションを起動するセッションを表します．実験してみるためには，端末ウィンドウを2つ開き，それぞれからpsqlを起動するとよいでしょう.

```
T1: BEGIN;
T2: BEGIN;
T1: LOCK t1;
T2: LOCK t2;
T1: LOCK t2; ⇦ T2がロックを解放するのを待つ
T2: LOCK t1;
ERROR: deadlock detected ⇦ デッドロックが検出される
```

この例では，T1がt2のロック解放を待つ一方で，T2がt1のロック解放を待つ結果，どうにもならなくなってデッドロックが発生しました.

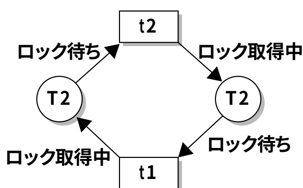
デッドロックの検出

さて，デッドロックがどのようなものかわかったところで，PostgreSQLがデッドロックを検出する方法を説明します.

PostgreSQLはどのトランザクションがどのトランザクションを待っているかを表現するデータ構造を内部に保持しており，これは図1のような有向グラフになります．この有向グラフの中に閉ループが含まれていればデッドロックと判定できます．この例では関与するトランザクションやテーブルが少ないので閉ループの検出は簡単です.

しかし，一般に複雑な有向グラフから閉ループを検出するのは重い処理であるため，PostgreSQLではテーブル（あるいはレコード）へのロックを取得しようとした際に一定時間待ってから，初めてデッドロックの検出処理を起動します．この「一定時間」がすなわちdeadlock_timeoutです．もし首尾よくロックが取得できればデッドロック検出処理の予約（実際にはタイマ）が取り消されるので無駄なデッドロック検出処理は回避される仕掛けになっています.

図1◎トランザクションの遷移とデッドロック



高負荷に注意

この方式は負荷の低いときにはうまく動きますが，負荷の高いときにはロックを獲得するのに時間がかかり，その結果デッドロック検出処理が走りやすくなるという欠点があります．デッドロック処理が走るようになるとますます負荷が高くなるので，事態はさらに悪化していきます．最悪の場合は,

```
FATAL: s_lock ... stuck spin lock. Aborting.
```

のようなメッセージが出て全バックエンドプロセスが強制終了されてしまいます.

この問題はdeadlock_timeoutの値を増やすことによって回避できます[注4]．特にWebサーバとDBサーバが1台のマシンに同居するようなケースではシステムの負荷が高くなりがちなので，deadlock_timeoutの値を増やしておくと良いでしょう.

同時接続ユーザが多いときも注意

このデッドロック検出処理は各バックエンドプロセスが独自に行うため，同時接続ユーザ数が増えてくると，デッドロック検出処理が起動される確率も増えてきます．たとえば，deadlock_timeoutのデフォルト値は1000msec＝1秒ですが，同時接続ユーザ数が100になると，システム全体では平均して1000／100＝

注4）deadlock_timeoutを増やすとデッドロックの検出が遅れるようになりますが，そもそもデッドロックが発生する可能性があるようでは業務システムとして論外なので，実用上は特に問題にならないでしょう.

10msec＝0.01秒ごとにデッドロック検出処理が走る可能性がでてきます．これではシステムはたまりません．この場合もdeadlock_timeoutを増やしましょう．とりあえず，

デフォルト値（1000）×同時接続ユーザ数

くらいにし，状況を見て増やすようにしましょう．

火消しテクニックその2：同時接続ユーザ数を減らす

増えてしまいがちな同時接続数

Webシステムのバックエンドにデータベースを使っている場合，1つのApacheプロセスから1つ以上のデータベースへの接続が発生することがあります．Apacheプロセスの数が足りないとWebサイトへアクセスしたときにエラーになってしまうので，Apacheプロセスの数は減らせません．また，人気の高いサイトではWebサーバを複数台設置して負荷分散することも珍しくないので，PostgreSQLへの接続要求はますます増えてしまいます．

●同時接続数が増えると？

そのため，DBサーバのハードウェアを増強するなどしてPostgreSQLが扱うことのできる同時接続数を増やす努力をするわけですが，これにも限界があります．なによりも問題なのは，データベースの性能を最大限に発揮できないことです．

一般にデータベースへの同時接続数が増えるにつれてデータベースのスループットは上昇していきますが，ある程度以上接続数が増えると頭打ちになり，その後は徐々に低下していきます（図2）．接続数が非常に多い状態でPostgreSQLを使うということは，わざわざ性能の悪いところでデータベースを運用することになり，ひいてはシステム全体の性能に悪影響をもたらします．

コネクションプールの利用

Javaなどを使ったシステムでは，「コネクションプール」という機能を使ってこの問題に対応できます．コネクションプールはデータベースへの接続をキャッシュしてデータベースへの接続オーバヘッドを軽減するとともに，データベースへの接続数を一定数に制限することができます．

●コネクションプールが使えなくても…

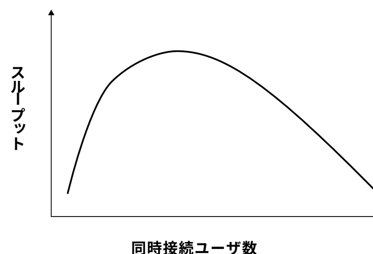
残念ながらPHPではコネクションプールが使えませんし，同時接続ユーザ数を制限することもできません．そこでちょっと工夫して「疑似コネクションプール」を作ってみましょう．

まずPostgreSQLがもっとも性能を発揮できる同時接続ユーザ数に，PostgreSQLの設定値であるmax_connectionsを制限してしまいます．こうすると当然のことながら，max_connectionsを超えてPHPが接続しようとするとエラーになってしまいますが，そのときにすぐにあきらめずに時間をおいて再接続を試みるようにします．

これは一種のビジーループなので，効率はよくありません．と言うか，はっきり言って非常に悪いです．こんなものでも果たして使いものになるかどうか，実験してみましょう．

リスト1がそのサンプルプログラムです．①のpg_connect()がデータベースへの接続に失敗したときは，最初は0.1秒，その後は0.2秒，その後は0.4秒…待ってから再度接続を試みます[注5]．②が実際のデータベース処理で，ここではPostgreSQLに付属するpgbenchというベンチマークプログラム用の10万件のテーブルからランダムに1件検索するようにしています．

図2◎同時接続数とスループット



注5）一般にリトライ処理では，このように徐々にインターバルを長くし，リトライ処理が輻輳（ふくそう）しないようにするのが普通です．

▼リスト1　擬似コネクションプール

```
<?php
 ini_set("track_errors", "1");
 define_syslog_variables();

 $sleep_microsec = 100000;
 $retry = 200;
 do {
   @$con = pg_connect("dbname= test user=nobody");───①
   if ($con !== FALSE) {
     break;
   }
   usleep($sleep_microsec);
   $sleep_microsec *= 2;
   $retry--;
 } while ($retry > 0);
 if ($con === FALSE) {
   syslog(LOG_ERR, "could not connect $php_errormsg");
   trigger_error("Could not connect to DB", E_USER_ERROR);
   exit;
 }

 $aid = rand(1,10000);
 pg_query(
   $con, "SELECT * FROM accounts WHERE aid = $aid");───②
 pg_close($con);
?>
```

●擬似コネクションプールで一定の効果

このスクリプトを適当なURLに登録し，Apacheに付属するabというApache用のベンチマークプログラムで擬似的に負荷をかけてみます．abの起動方法は，

```
ab -v 3 -c 100 -n 100 'http://localhost/~t-ishii/bench.php'
```
（実際は1行）

としました．すなわち，同時アクセスユーザが100で，トータル100個のリクエストを処理するわけです．PostgreSQLのmax_connectionsを4，8，16，32，64と変化させてabを実行し，その結果1秒間に処理できたリクエスト数をグラフにしたのが図3です．

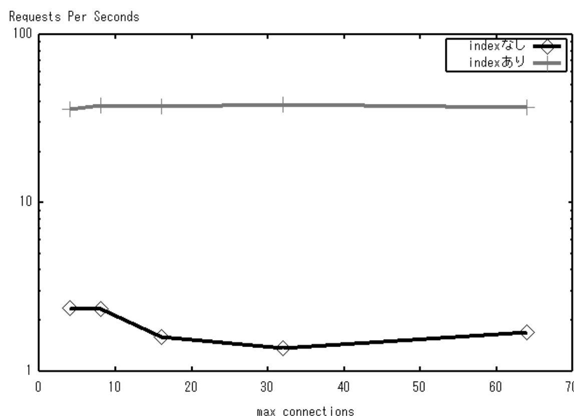
10万件のデータにインデックスが張られていない状態，すなわちDB処理が重いときは（この環境に限って言えば注6）同時接続ユーザ数を4か8くらいに制限したほうがトータルの性能が良好であることがわかります．実際には同時接続ユーザ数が少ないときはPostgreSQLに接続しようとして無駄にリトライが繰り返されているはずですが，その分の負荷を差し引いてもおつりがきたということでしょう．

参考までにインデックスを張っているときの状態もグラフにしてあります．この場合，DB処理が非常に軽いので，PostgreSQLは常に余裕をもって動作しており，ほとんどの場合，処理要求待ち状態になっています．その結果PHP側は接続順番待ちしなくても順調にリクエストをこなすことができるので，グラフ上でも同時接続ユーザ数にかかわらずほぼ一定の性能を示しています．

●使うべきかどうか

以上から，DB処理がある程度重い場合は擬似コネクションプールもそれなりに効果があることがわかりました．しかし，ここで説明した擬似コネクションプールは所詮間に合わせです．長々と説明しておきながら恐縮ですが，本稿は，なんらかの方法でDBへの接続要求を制限することが有効であることを説明するの

図3◎負荷測定の結果



注6）テストに使ったの手元にあるノートPCで，Vine Linux 2.6CR（kernel 2.4.20），CPUはMobile Pentium III 866MHz，メモリ512Mバイトです．

が主旨で，このようなある意味「野蛮」な方法をみなさんにお勧めしているわけではないのです．

ではどうするか

それではPHP使ったシステムでは，大規模なWebシステムを構築するのはあきらめるしかないのでしょうか．対策としては，2つ考えられます．

① もっとまともなコネクションプールシステムを作る
② できるだけDB処理を軽くする

本稿で説明した方法は「接続できなかったら再度チャレンジを試みる」という野蛮な方法です．PHP側でもっときちんとDBへの接続要求を管理できれば，このようなことをせずに済むはずです．

さらに理想的なのは，独立したコネクションプールサーバを作ることでしょう．実際，JavaではWebサーバとは独立したサーブレットコンテナや，アプリケーションサーバがこの役割を果たしています．

①はすぐには実現できそうもないので，「火消しテクニック」には役立ちません．そこで②を実現する具体的な方法を考えましょう．

火消しテクニックその３：重いＤＢ処理を見つける

修羅場と化している現場では，最短の時間で最大の効果を上げることが求められます．そのためには，PostgreSQLの適切なログを取ることが先決です．適切なログさえあれば，複数の人間で分担してすばやくシステムの状況を解析することができます．ここでは，PostgreSQL自身を使って重いDB処理をさらに効率よく見つける方法を説明します．

例題はpgbench

例題として，pgbenchによってベンチマークのために実行された問い合わせの中から重いDB処理を見つけることにします．そのままではどこが重い処理なのかがわかりにくいので，一番大きなテーブルからインデックスを外し，わざと処理を重くしておきます．

```
$ pgbench -i test ⇦ ベンチマークデータベースの初期化
$ psql -c "ALTER TABLE accounts DROP
CONSTRAINT accounts_pkey" test     （実際は1行）
 ⇦ accountsテーブルからインデックスを外す
```

なお，ここではPostgreSQL 7.3以降を使用することを前提にします．

実行時間ログの取り方

PostgreSQLでは，postgresql.confの設定で個々のSQLの実行時間をログに取ることができます．そのための設定項目は以下です．

```
log_duration = true
log_pid = true
log_statement = true
log_timestamp = true
```

この設定をすると多量のログが出力されるため，syslogをログ出力にすると負荷かがかかりすぎます．普通のファイルにログを取るようにしてpostmasterを起動します．

```
$ postmaster >& postmaster.log&
```

ここでpgbenchを実行します（図4）．これにより，postmaster.logに図5のようなログが取れます．

“query”が実行された問い合わせ，“duration”が実行時間です．注意しておかなければならないのは，durationの該当行が必ずしも問い合わせの直後に表示されないことです．[5991]のような数字はバックエン

▼図4　pgbenchの実行

```
$ pgbench -c 10 -t 1 test
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1
number of clients: 10
number of transactions per client: 1
number of transactions actually processed: 10/10
tps = 1.560105 (including connections establishing)
tps = 1.587846 (excluding connections establishing)
```

▼図5　ログの一部

```
2003-05-05 17:22:20 [5991]   LOG:  query: update branches set bbalance = bbalance + 531 where bid = 1
2003-05-05 17:22:20 [5991]   LOG:  duration: 0.002057 sec
```

ドプロセスのプロセスIDを表しており，query:のプロセスIDと同じプロセスIDを持つ，最初のdurationの行が実行時間になります．

実行時間ログの解析

durationが長い問い合わせがすなわち重いDB処理ですから，durationに注目してpostmaster.logを目で追えば重いDB処理がわかります．けれども，それはなかなか大変です．

そこでログの解析にPostgreSQLを使うことにします．PostgreSQLを使ってPostgreSQLを解析するのもなかなかおつなものです:-)

まずpostmaster.logのデータを格納する「log」というテーブルを用意します（リスト2）．そしてリスト3のスクリプトを使ってpostmaster.logのデータを格納します．

重いDB処理の「パターン」を発見する

準備ができたらいよいよlogテーブルから重いDB処理を探します．ここで重要なのは問い合わせの「パターン」の発見です．一般に個々の問い合わせは文字列としてみると少しずつ異なります（図6）．

これらをすべて異なる問い合わせとしてしまうと，重いDB処理の「パターン」を見つけ出すことができません．理想的には，同じアプリケーションの同じ個所で生成された問い合わせは同じパターンとして扱いたいものです．そうすれば，少ないプログラムやテーブルの修正で，問題点を解決することができます．

パターンを見つけ出す凝った方法はいろいろ考えられますが，ここではとりあえず「先頭から10文字が同じなら同じパターンであると見なす」という非常にシンプルな方法を採ります．

10%の法則

「何事も支配的な部分は全体のわずか10%であり，全体の振る舞いはその10%に左右される」という話

▼リスト2　logテーブル

```
CREATE TABLE log (
  qsec INTERVAL, -- 実行時間
  qdate TIMESTAMP WITHOUT TIME ZONE, -- 実行時刻
  pid INTEGER,   -- プロセスID
  query TEXT     -- 問い合わせ
);
```

▼リスト3　ログ解析用スクリプト

```
#! /bin/sh
cat postmaster.log|
sed -e 's/\[//' \
    -e 's/\]//' |
awk '{
    pid = $3
    if ($5 == "query:") {
        s1 = sprintf("%s\ %s\t%s\t",$1,$2,$3)
        gsub("^.*query: ","")
        gsub("\t"," ")
        s2 = $0
        QUERY[pid] = s1 s2
    } else if ($5 == "duration:") {
        printf("%s\t%s\n",$6,QUERY[pid])
    }
}'|
psql -c '\copy log from stdin' test
```

▼図6　問い合わせのログ

```
2003-05-05 17:22:14 [5988]   LOG:  query: update accounts set abalance = abalance + 91 where aid =
22038
2003-05-05 17:22:14 [5989]   LOG:  query: update accounts set abalance = abalance + 163 where aid =
43662
```

▼図7　実行結果のサンプル

```
-[ RECORD 1 ]-----------------------------------------------------------
count   | 10
max     | 00:00:03.517026
min     | 00:00:02.935153
avg     | 00:00:03.129523
pattern | update acc
sample  | update accounts set abalance = abalance + 91 where aid = 22038
-[ RECORD 2 ]-----------------------------------------------------------
count   | 10
max     | 00:00:02.989063
min     | 00:00:02.52415
avg     | 00:00:02.807223
pattern | select aba
sample  | select abalance from accounts where aid = 95197
```

を筆者は聞いたことがあります．このことを今回の件に適用すると，足を引っ張っている10％の支配的なパターンを発見する，ということになります．

そこで，出現頻度が高く，平均実行時間が1秒以上かかっている問い合わせのワースト10に注目して効率よく重いDB処理を見つけ出すことにします．これを求める問い合わせは次のようになります．

```
SELECT count(qsec),max(qsec),min(qsec),avg(qsec),
 substring(query from 1 for 10) AS pattern,
 max(query) AS sample
FROM log GROUP BY pattern
HAVING avg(qsec) > INTERVAL '1 sec'
ORDER BY avg(qsec) DESC LIMIT 10;
```

実行結果のサンプルは図7です（見やすくするために，psqlで\xと\psetfooterオプションを使っています）．この例では，実行時間が平均1秒以上かかっているパターンは2つだけでしたが，いずれも実行回数が10回，すなわちpgbenchで毎回実行される問い合わせなので，影響が大きそうです．

重いDB処理を軽くする

まあ，わかっていたことではありますが:-)今回見つけた重いDB処理はどちらもaccountsテーブルに絡む処理でした．これを軽くするためには，aid列にインデックスを追加すれば良いわけです．具体的には，

```
ALTER TABLE accounts ADD PRIMARY KEY(aid);
```

を実行します．

結果の確認

念のために，explain analyzeコマンドを使ってインデックスが使われていることと，検索が高速化されたことを確認します（図8）．単独実行なので負荷が軽いせいもありますが，かなり高速化されていることがわかります．

pgbenchの結果はどうでしょうか．図9のようになります．どうやら30倍くらい速くなったようです．これでくだんのプロジェクトリーダも満足してくれることでしょう．

最後に

最近はPostgreSQLのパフォーマンスチューニングに関しての情報もある程度雑誌に流れるようになりましたが，実際現場でどのようにしたら性能改善ができるのかということについてはあまり語られていないようなので，少々泥臭い話を交えて実践的なテクニックをお話ししました．

次回のネタはまだ決めてませんが，今回とは逆に「実践的ではないが，ディープかつ（筆者にとって）楽しい話題」にしたいと思っています．

▼図8　explain analyzeの結果

```
test=# explain analyze select abalance from accounts where aid = 95197;
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------
 Index Scan using accounts_pkey on accounts  (cost=0.00..3.66 rows=1 width=4) (actual time=0.08..0.09 
rows=1 loops=1)
   Index Cond: (aid = 95197)
 Total runtime: 0.20 msec
(3 rows)
```

▼図9　pgbenchの結果

```
$ pgbench -c 10 -t 1 test
starting vacuum...end.
transaction type: TPC-B (sort of)
scaling factor: 1
number of clients: 10
number of transactions per client: 1
number of transactions actually processed: 10/10
tps = 31.568546 (including connections establishing)
tps = 48.780250 (excluding connections establishing)
```