Explaining Explain
～ PostgreSQLの実行計画を読む ～

(2009-03-17 更新 / バージョン8.3対応)

by Robert Treat
(Inspired by Greg Sabino Mullane)

翻訳 : 日本PostgreSQLユーザ会

Explain

- Explainはプランナによって決定された「最良の」実行計画を表示する。
- クエリを実行するための個々のステップを表示する。
- DMLコマンドに対してのみ使用できる。
- レコードセットのカラム数、行数およびコストを表示する。
- 数値は推定されたものであり、実際のコストを見るにはEXPLAIN ANALYZEを使う必要がある。

Explain Planの例

```
=# EXPLAIN SELECT * FROM pg_proc
   ORDER BY proname;

                         QUERY PLAN
----------------------------------------------------
Sort (cost=181.55..185.92 rows=1747 width=322)
  Sort Key: proname
    -> Seq Scan on pg_proc
              (cost=0.00..87.47 rows=1747 width=322)
```

心掛けるべきこと

- 知っておくべき用語
  – プラン、ノード、演算子、フィルタ、入力セット
- テーブルスキャンした結果、あるいはその他の演算子の結果は、すべて上位に渡される。
- すべての演算子や入力セットを受け取り、最上位のノードに辿り付くまで上位の演算子に渡していく。
- 親ノードは子ノードのコストを受け取る。
- InitPlansとSubplansは副問い合わせの際に使われる。

Explaining → Widths

```
=# EXPLAIN SELECT oid FROM pg_proc;
                  QUERY PLAN
------------------------------------------
 Seq Scan on pg_proc
     (cost=0.00..87.47 rows=1747 width=4)
```

- このレベルにおける推定された入力サイズを表示する。
- それほど重要ではない

一般的なデータ型のサイズについて

smallint	2
integer	4
bigint	8
boolean	1
char(n)	n+1
varchar(n)	～
text [n文字]	n+4

Explaining → Rows

```
=# EXPLAIN SELECT oid FROM pg_proc;
                QUERY PLAN
------------------------------------------
 Seq Scan on pg_proc
     (cost=0.00..87.47 rows=1747 width=4)
```

- 推定された行数を表示する
- PostgreSQL 8.0以前では、一度もVACUUM/ANALYZEされていないテーブルについては1000行がデフォルト。
- 実際の値と大きくかけ離れている場合、vacuum あるいはanalyzeをすべきというサインである。

Explaining → Cost

```
=# EXPLAIN SELECT oid FROM pg_proc;
                    QUERY PLAN
-----------------------------------------------
 Seq Scan on pg_proc
      (cost=0.00..87.47 rows=1747 width=4)
```

- コストは、オプティマイザがさまざまなプランの中からある特定のプランを選ぶための指標である
- 2つのコスト: スタートアップコスト (左) とトータルコスト (右)
  - 実行プランの比較で重要なのはトータルコスト。
  - いくつかの演算子にはスタートアップコストがある。無いものもある。
  - コストは推定値に過ぎない。その算出は結構複雑。
- 値はシーケンシャルI/Oで1ページを読み込むコストを 1.0 とした際の相対値で示される。

Explaining → Cost

パラメータ	説明	規定値	相対速度
seq_page_cost	シーケンシャル読み込み1回	1.00	(基準)
random_page_cost	ランダム読み込み1回	4.00	4倍遅い
cpu_tuple_cost	行の処理1回	0.01	100倍速い
cpu_index_tuple_cost	索引の処理1回	0.005	200倍速い
cpu_operator_cost	計算1回	0.0025	400倍速い
effective_cache_size	ページキャッシュサイズ	128MB	N/A

Explaining → Explain Analyze

```
=# EXPLAIN ANALYZE SELECT oid FROM pg_proc;
                QUERY PLAN
------------------------------------------
 Seq Scan on pg_proc
     (cost=0.00..87.47 rows=1747 width=4)
     (actual time=0.077..17.082 rows=1747 loops=1)
 Total runtime: 20.125 ms
```

- 実際にクエリを実行し、実際の情報を表示する。
- 時間はミリ秒で表示される。「コスト」とは無関係。
- 全体の実行時間も表示される。
- 「loops」は処理の繰り返し回数。実行時間(time)は繰り返し全体の時間を表す。

Explaining → プラン演算子

演算子	関連処理	始動コスト
Seq Scan	表スキャン	無
Index Scan	索引スキャン	無
Bitmap Index Scan		有
Bitmap Heap Scan		有
Subquery Scan	副問合せ	無
Tid Scan	ctid = ...	無
Function Scan	関数スキャン	無
Nested Loop	結合	無
Merge Join	結合	有
Hash Join	結合	有
Sort	ORDER BY	有
Hash		有

演算子	関連処理	始動コスト
Result	関数スキャン	無
Unique	DISTINCT UNION	有
Limit	LIMIT OFFSET	有
Aggregate	count, sum, avg, stddev	有
Group	GROUP BY	有
Append	UNION	無
Materialize	副問合せ	有
SetOp	INTERCECT EXCEPT	有

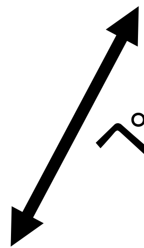
Seq Scan 演算子：例題

```
=# EXPLAIN SELECT oid FROM pg_proc;
               QUERY PLAN
------------------------------------------
 Seq Scan on pg_proc
     (cost=0.00..87.47 rows=1747 width=4)
```

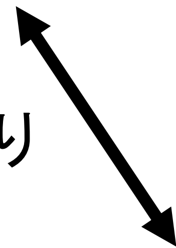
- 最も基本。単に表を最初から最後へとスキャンする
- 条件にかかわらず各行をチェックする
- 大きなテーブルはインデックススキャンの方が良い
- コスト(開始コスト無し), 行(タプル), 幅(oid)
- トータルコストは 87.47

Seq Scan 演算子：説明

```
=# SELECT relpages, reltuples FROM
        pg_class WHERE relname = 'pg_proc';
 relpages | reltuples
----------+-----------
       70 |      1747
```



ページの読み取り



行ごとの計算

$(70 \times 1.0) + (1747 * 0.01)$

= **87.47**

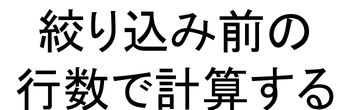
WHERE句のコスト

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE oid > 0;

                QUERY PLAN
---------------------------------------
Seq Scan on pg_proc
    (cost=0.00..91.84 rows=583 width=4)
  Filter: (oid > 0::oid)
```

87.47 + (1747 * 0.0025)

= **91.84**



Sort 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc ORDER BY oid;
                  QUERY PLAN
---------------------------------------------
Sort (cost=181.55..185.92 rows=1747 width=4)
   Sort Key: oid
-> Seq Scan on pg_proc
    (cost=0.00..87.47 rows=1747 width=4)
```

- 明示的なソート：ORDER BY句
- 暗黙的なソート：Unique, Sort-Merge Join など
- 開始コストを持っている: 最初の値はすぐには返却されない

Index Scan 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE oid=1;
                    QUERY PLAN
----------------------------------------------------
 Index Scan using pg_proc_oid_index on pg_proc
     (cost=0.00..5.99 rows=1 width=4)
   Index Cond: (oid = 1::oid)
```

- 特に大きなテーブルではコストが低くなるので選ばれる可能性が高い
- Index Condが無い場合は、ソートの代わりとして使われるインデックス順のフルスキャンを表す

Bitmap Scan 演算子

```
test=# EXPLAIN SELECT * FROM q3c,q3c as q3cs
  WHERE (q3c.ipix>=q3cs.ipix-3 AND q3c.ipix<=q3cs.ipix+3)
     OR (q3c.ipix>=q3cs.ipix-1000 AND q3c.ipix<=q3cs.ipix-993);
                    QUERY PLAN
------------------------------------------------
Nested Loop
  -> Seq Scan on q3c q3cs
     -> Bitmap Heap Scan on q3c
        -> BitmapOr
             -> Bitmap Index Scan on ipix_idx
             -> Bitmap Index Scan on ipix_idx
```

- 8.1で追加された
- BitmapOr, BitmapAnd で複数のビットマップを合体
- リレーションの”ビットマップ“をメモリ内で作成する

Result 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE 1+1=3;
                  QUERY PLAN
------------------------------------------------
Result (cost=0.00..87.47 rows=1747 width=4)
  One-Time Filter: false
  -> Seq Scan on pg_proc
        (cost=0.00..87.47 rows=1747 width=4)
```

- 非テーブル問い合わせ
- テーブルを参照せずに結果が得られる場合

Unique 演算子

```
=# EXPLAIN SELECT distinct oid FROM pg_proc;
                    QUERY PLAN
--------------------------------------------------
Unique (cost=181.55..190.29 rows=1747 width=4)
  -> Sort (cost=181.55..185.92 rows=1747 width=4)
       Sort Key: oid
  -> Seq Scan on pg_proc
       (cost=0.00..87.47 rows=1747 width=4)
```

- 入力セットから重複する値を削除
- 行の並べ替えはせず、単に重複する行を取り除く
- 入力セットは予めソート済み (Sort演算子の後に行う)
- タプルコストごとに「CPU演算」×2
- DISTINCT と UNION で使用される

Limit 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc LIMIT 5;
               QUERY PLAN
------------------------------------------
Limit (cost=0.00..0.25 rows=5 width=4)
  -> Seq Scan on pg_proc
      (cost=0.00..87.47 rows=1747 width=4)
```

- 行は指定された数に等しい
- 最初の行を即時に返す
- 少量の開始コスト追加でオフセットの扱いも可

```
=# EXPLAIN SELECT oid FROM pg_proc LIMIT 5 OFFSET 5;
               QUERY PLAN
------------------------------------------
Limit (cost=0.25..0.50 rows=5 width=4)
  -> Seq Scan on pg_proc
      (cost=0.00..87.47 rows=1747 width=4)
```

Aggregate 演算子

```
=# EXPLAIN SELECT count(*) FROM pg_proc;
                          QUERY PLAN
--------------------------------------------------------------
Aggregate (cost=91.84..91.84 rows=1 width=0)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=0)
```

- count, sum, min, max, avg, sttdev, varianceを使用
- GROUP BY 使用の場合差異が認められることがあり

```
=# EXPLAIN SELECT count(oid), oid FROM pg_proc GROUP BY oid;
                           QUERY PLAN
-------------------------------------------------------------
HashAggregate (cost=96.20..100.57 rows=1747 width=4)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

GroupAggregate 演算子

```
=# EXPLAIN SELECT count(*) FROM pg_foo GROUP BY oid;
                           QUERY PLAN
-----------------------------------------------------------------
GroupAggregate (cost=37442.53..39789.07 rows=234654 width=4)
  -> Sort (cost=37442.53..38029.16 rows=234654 width=4)
     Sort Key: oid
       -> Seq Scan on pg_foo (cost=0.00..13520.54 rows=234654 width=4)
```

- GROUP BYを使用し、より大きな結果セット上に集約を行う

Append 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc
   UNION ALL SELECT oid ORDER BY pg_proc;
                                   QUERY PLAN
------------------------------------------------------------------
Append (cost=0.00..209.88 rows=3494 width=4)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
  -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747 width=4)
```

- UNION (ALL) によるトリガー, 継承
- 開始コスト無し
- コストは単に全ての入力の合計

Nested Loop 演算子

```
=# SELECT * FROM pg_foo JOIN pg_namespace
        ON (pg_foo.pronamespace=pg_namespace.oid);
                                  QUERY PLAN
----------------------------------------------------------------------
Nested Loop (cost=1.05..39920.17 rows=5867 width=68)
  Join Filter: ("outer".pronamespace = "inner".oid)
    -> Seq Scan on pg_foo (cost=0.00..13520.54 rows=234654 width=68)
         -> Materialize (cost=1.05..1.10 rows=5 width=4)
         -> Seq Scan on pg_namespace (cost=0.00..1.05 rows=5 width=4)
```

- ２つのテ=ブルの結合(2つの入力セット)
- INNER JOIN とLEFT OUTER JOIN の使用
- 「外部」テーブルをスキャンし、「内部」テーブルにマッチするものの発見
- 開始コスト無し
- インデックスが無い場合遅い問い合わせになる可能性、特にselect句に関数がある場合

Merge Join 演算子

```
=# EXPLAIN SELECT relname,nspname FROM pg_class left join
        pg_namespace ON (pg_class.relnamespace = pg_namespace.oid);
                                QUERY PLAN
------------------------------------------------------------------------
Merge Right Join (cost=14.98..17.79 rows=186 width=128)
  Merge Cond: ("outer".oid = "inner".relnamespace)
  -> Sort (cost=1.11..1.12 rows=5 width=68)
       Sort Key: pg_namespace.oid
       -> Seq Scan on pg_namespace (cost=0.00..1.05 rows=5 width=68)
       -> Sort (cost=13.87..14.34 rows=186 width=68)
            Sort Key: pg_class.relnamespace
            -> Seq Scan on pg_class (cost=0.00..6.86 rows=186 width=68)
```

- 二つのデータセットをJOINする：outerとinner
- Merge Right JoinとMerge In Joinがある
- データセットはあらかじめソートされていなければならず、また両方同時に走査される。

Hash & Hash Join 演算子

```
=# EXPLAIN SELECT relname, nspname FROM pg_class JOIN
        pg_namespace ON (pg_class.relnamespace=pg_namespace.oid);
                          QUERY PLAN
------------------------------------------------------------------------
Hash Join (cost=1.06..10.71 rows=186 width=128)
  Hash Cond: ("outer".relnamespace = "inner".oid)
    -> Seq Scan on pg_class (cost=0.00..6.86 rows=186 width=68)
    -> Hash (cost=1.05..1.05 rows=5 width=68)
         -> Seq Scan on pg_namespace (cost=0.00..1.05 rows=5 width=68)
```

- Hashは、異なるHash Join演算子で使用されるハッシュテーブルを作成する
- 一方の入力からハッシュテーブルを作成し、二つの入力を比較する
- INNER JOIN、OUTER JOINと同時に使われる
- ハッシュの作成にはスタートアップコストが伴う

Tid Scan 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc WHERE ctid = '(0,1)';
                      QUERY PLAN
------------------------------------------------------
Tid Scan on pg_proc (cost=0.00..4.01 rows=1 width=4)
  Filter: (ctid = '(0,1)'::tid)
```

- カラムタプルID
- “ctid=”がクエリに指定された場合のみ使われる
- 滅多に使わない、非常に速い

Function Scan 演算子

```
=# CREATE FUNCTION foo(integer) RETURNS SETOF integer AS
   $$
       select $1;
   $$
   LANGUAGE sql;

=# EXPLAIN SELECT * FROM foo(12);
                         QUERY PLAN
------------------------------------------------------------
 Function Scan on foo (cost=0.00..12.50 rows=1000 width=4)
```

- 関数がデータをgatherするときに出てくる
- トラブルシューティングの観点からは若干ミステリアス
- 関数の中で使われているクエリについてexplainを走らせるべき

SetOp 演算子

```
=# EXPLAIN SELECT oid FROM pg_proc INTERSECT SELECT oid FROM pg_proc;
                              QUERY PLAN
------------------------------------------------------------------------
SetOp Intersect (cost=415.51..432.98 rows=349 width=4)
  -> Sort (cost=415.51..424.25 rows=3494 width=4)
       Sort Key: oid
         -> Append (cost=0.00..209.88 rows=3494 width=4)
              -> Subquery Scan "*SELECT* 1" (cost=0.00..104.94 rows=1747)
                   -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747)
              -> Subquery Scan "*SELECT* 2" (cost=0.00..104.94 rows=1747)
                   -> Seq Scan on pg_proc (cost=0.00..87.47 rows=1747)
```

- INTERSECT, INTERSECT ALL, EXCEPT, EXCEPT ALL句のために使用される
  - SetOp Intersect, Intersect All, Except, Except All

実行プランの強制

- SET enable_演算子 = off;
  - プランナーがある演算子を使おうとするのを「強く思いとどまらせる」ことができる
  - SETを行ったセッションのみに影響する
- Planner Method Configuration (on/off)
  - enable_bitmapscan
  - enable_hashagg
  - enable_hashjoin
  - enable_indexscan
  - enable_mergejoin
  - enable_nestloop
  - enable_seqscan
  - enable_sort
  - enable_tidscan

Seq Scan の強制

```
=# EXPLAIN SELECT * FROM pg_class;
                               QUERY PLAN
-------------------------------------------------------------------------
Seq Scan on pg_class
   (cost=100000000.00..100000006.86 rows=186 width=164)
```

- 始動コストに 100000000.0 を足すだけ
  - /src/backend/optimizer/path/costsize.c

スキャン強制, プランを変える

```
=# EXPLAIN ANALYZE SELECT * FROM pg_class WHERE oid > 2112;
                    QUERY PLAN
------------------------------------------------
Seq Scan on pg_class
    (cost=0.00..7.33 rows=62 width=164)
    (actual time=0.087..1.700 rows=174 loops=1)
  Filter: (oid > 2112::oid)
Total runtime: 2.413 ms

=# SET enable_seqscan = off;
=# EXPLAIN ANALYZE SELECT * ORDER BY pg_class WHERE oid > 2112;
                    QUERY PLAN
------------------------------------------------
Index Scan using pg_class_oid_index on pg_class
    (cost=0.00..22.84 rows=62 width=164)
    (actual time=0.144..1.802 rows=174 loops=1)
  Index Cond: (oid > 2112::oid)
Total runtime: 2.653 ms
```

心掛けるべきこと

- プランの強制は開発時にはよいが、製品には不適
  - やむを得ず使う場合は SET LOCAL で設定すること。トランザクショ完了時に元の設定に戻すように。
- （Tom Laneでもない限り）人はプランナーより賢くない
- 他方では、プランナーは推測しかしない
  - 統計情報を正しい状態に保つため定期的なANALYZEを。autovacuum に任せるのが一番確実。
  - 環境に合わせてコスト変数 (Planner Cost Constants) を適切に設定することが重要
- 可能なときには、explain analyzeを使いなさい

現実のデバッグ

- 例1. ANALYZEをしよう
- 例2. とにかくANALYZEをしよう
- 例3. テーブルの肥大化に気をつけよう
- 例4. 結合, IN, EXISTS を使い分けよう

実際のデバッグ(例1) : ANALYZE前

```
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                            QUERY PLAN
--------------------------------------------------------------------
Nested Loop (cost=0.00..2654.65 rows=199 width=8)
            (actual time=151.16..538.45 rows=124 loops=1)
  -> Seq Scan on exception_notice_map
         (cost=0.00..250.50 rows=399 width=4)
         (actual time=0.10..101.61 rows=15181 loops=1)
       Filter: (notice_id = 3)
  -> Index Scan using exception_pkey on exception
         (cost=0.00..6.01 rows=1 width=4)
         (actual time=0.03..0.03 rows=0 loops=15181)
       Index Cond: (exception.exception_id = "outer".exception_id)
       Filter: (complete IS FALSE)
Total runtime: 538.76 msec
```

exception表に“WHERE complete IS False”という条件の部分インデックスがあり、条件を満たす行は251行だけなのに使ってくれない

実際のデバッグ(例1)：ANALYZE後

```
=# ANALYZE exception;
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                          QUERY PLAN
--------------------------------------------------------------
Hash Join (cost=28.48..280.98 rows=1 width=8)
          (actual time=31.45..97.78 rows=124 loops=1)
  Hash Cond: ("outer".exception_id = "inner".exception_id)
    -> Seq Scan on exception_notice_map
          (cost=0.00..250.50 rows=399 width=4)
          (actual time=0.12..77.12 rows=15181 loops=1)
         Filter: (notice_id = 3)
    -> Hash (cost=26.31..26.31 rows=251 width=4)
            (actual time=2.96..2.96 rows=0 loops=1)
         -> Index Scan using active_exceptions on exception
                (cost=0.00..26.31 rows=251 width=4)
                (actual time=0.24..2.55 rows=251 loops=1)
              Filter: (complete IS FALSE)
Total runtime: 97.99 msec
```

部分インデックスを
使ってくれた

実際のデバッグ(例2)：ANALYZE前

```
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                                QUERY PLAN
---------------------------------------------------------------
Hash Join (cost=22.51..45.04 rows=2 width=8)
          (actual time=9961.14..11385.11 rows=105 loops=1)
  Hash Cond: ("outer".exception_id = "inner".exception_id)
    -> Seq Scan on exception
           (cost=0.00..20.00 rows=500 width=4)
           (actual time=365.12..10659.11 rows=228 loops=1)
         Filter: (complete IS FALSE)
    -> Hash (cost=22.50..22.50 rows=5 width=4)
            (actual time=723.39..723.39 rows=0 loops=1)
         -> Seq Scan on exception_notice_map
                (cost=0.00..22.50 rows=5 width=4)
                (actual time=10.12..694.57 rows=15271 loops=1)
              Filter: (notice_id = 3)
Total runtime: 11513.78 msec
```

推定値と結果 (actual) の
行数 (rows) の違いに注目。
まずはANALYZEしてみる。

実際のデバッグ(例2)：ANALYZE 1回目

```
=# ANALYZE exception_notice_map;
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                            QUERY PLAN
-------------------------------------------------------------
Merge Join (cost=42.41..802.93 rows=390 width=8)
           (actual time=10268.79..10898.29 rows=105 loops=1)
  Merge Cond: ("outer".exception_id = "inner".exception_id)
    -> Index Scan using exception_id on exception_notice_map
          (cost=0.00..714.22 rows=15562 width=4)
          (actual time=50.80..1063.05 rows=15271 loops=1)
         Filter: (notice_id = 3)
-> Sort (cost=42.41..43.66 rows=500 width=4)
        (actual time=9800.32..9800.65 rows=222 loops=1)
     Sort Key: exception.exception_id
       -> Seq Scan on exception
             (cost=0.00..20.00 rows=500 width=4)
             (actual time=357.18..9799.63 rows=228 loops=1)
           Filter: (complete IS FALSE)
Total runtime: 10898.57 msec
```

行数の推定は正しくなった

妙にキリが良い数値を疑う

実際のデバッグ(例2)：ANALYZE 2回目

```
=# ANALYZE exception;
=# EXPLAIN ANALYZE SELECT exception_id FROM exception
      JOIN exception_notice_map USING (exception_id)
     WHERE complete IS FALSE AND notice_id = 3;
                              QUERY PLAN
-----------------------------------------------------------
Merge Join (cost=0.00..796.57 rows=31 width=8)
           (actual time=425.41..971.81 rows=105 loops=1)
  Merge Cond: ("outer".exception_id = "inner".exception_id)
    -> Index Scan using active_exceptions on exception
           (cost=0.00..41.86 rows=651 width=4)
           (actual time=54.04..84.22 rows=222 loops=1)
         Filter: (complete IS FALSE)
    -> Index Scan using exception_id on exception_notice_map
           (cost=0.00..714.22 rows=15562 width=4)
           (actual time=34.42..843.10 rows=15271 loops=1)
         Filter: (notice_id = 3)
Total runtime: 972.05 msec
```

キリが良い数値が
無くなり速度が改善。
ただし、見積の誤差が
増加した理由は謎...

実際のデバッグ(例3) : Seq Scanが遅い

```
=# EXPLAIN ANALYZE SELECT s.site_id, s.name, i.image_name FROM images i
   JOIN host h USING (host_id) JOIN site s USING (site_id) WHERE images_id > 2112;
                              QUERY PLAN
----------------------------------------------------------------------------------
Hash Join (cost=113.88..253.51 rows=534 width=53)
          (actual time=610.52..627.11 rows=534 loops=1)
  Hash Cond: ("outer".site_id = "inner".site_id)
    -> Seq Scan on site s (cost=0.00..73.74 rows=1974 width=34)
                          (actual time=5.25..17.43 rows=1974 loops=1)
    -> Hash (cost=112.54..112.54 rows=534 width=19)
            (actual time=605.15..605.15 rows=0 loops=1)
         -> Hash Join (cost=15.01..112.54 rows=534 width=19)
                      (actual time=590.89..604.06 rows=534 loops=1)
              Hash Cond: ("outer".host_id = "inner".host_id)
                -> Seq Scan on host h (cost=0.00..77.24 rows=2724 width=8)
                                      (actual time=567.99..581.30 rows=2724 loops=1)
         -> Hash (cost=13.68..13.68 rows=534 width=11)
                 (actual time=17.30..17.30 rows=0 loops=1)
              -> Seq Scan on images i (cost=0.00..13.68 rows=534 width=11)
                                      (actual time=14.55..16.47 rows=534 loops=1)
                   Filter: (images_id > 2112)
```

host表のSeq Scan時間が他表と比べて長すぎる

実際のデバッグ(例3)：肥大化の回復

```
=# VACUUM FULL VERBOSE host;
INFO:  --Relation public.host--
INFO:  Pages 4785: Changed 0, reaped 4761, Empty 0, New 0; Tup 2724: Vac 0,
       Keep/VTL 0/0, UnUsed 267553, MinLen 100, MaxLen 229;
       Re-using: Free/Avail. Space 37629760/37627880;
       EndEmpty/Avail. Pages 0/4751.
       CPU 0.30s/0.03u sec elapsed 0.32 sec.
INFO:  Index host_pkey: Pages 1214; Tuples 2724: Deleted 0.
       CPU 0.07s/0.01u sec elapsed 0.08 sec.
INFO:  Rel host: Pages: 4785 --> 50; Tuple(s) moved: 2724.
       CPU 0.52s/1.09u sec elapsed 1.66 sec.
INFO:  Index host_pkey: Pages 1214; Tuples 2724: Deleted 2724.
       CPU 0.14s/0.00u sec elapsed 0.14 sec.
INFO:  --Relation pg_toast.pg_toast_2124348104--
INFO:  Pages 0: Changed 0, reaped 0, Empty 0, New 0; Tup 0: Vac 0, Keep/VTL 0/0,
       UnUsed 0, MinLen 0, MaxLen 0; Re-using: Free/Avail. Space 0/0;
       EndEmpty/Avail. Pages 0/0.
       CPU 0.00s/0.00u sec elapsed 0.00 sec.
INFO:  Index pg_toast_2124348104_index: Pages 1; Tuples 0.
       CPU 0.00s/0.00u sec elapsed 0.00 sec.
VACUUM
```

VACUUM FULL で
肥大化から回復させる。
多くのUnUsedを回収し
ページ数が大幅に減少。

実際のデバッグ(例4)：結合

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact
   JOIN advertiser USING (advertiser_id) WHERE type=1;
                           QUERY PLAN
------------------------------------------------------------------
Aggregate (cost=1.87..1.87 rows=1 width=0)
          (actual time=8.790..8.791 rows=1 loops=1)
  -> Merge Join (cost=1.03..1.86 rows=2 width=0)
                (actual time=8.752..8.766 rows=2 loops=1)
     Merge Cond: ("outer".advertiser_id = "inner".advertiser_id)
       -> Index Scan using advertiser_id_pkey on advertiser
            (cost=0.00..2.11 rows=8 width=4)
            (actual time=8.627..8.650 rows=4 loops=1)
          Filter: ("type" = 1)
            -> Sort (cost=1.03..1.03 rows=2 width=4)
                    (actual time=0.073..0.075 rows=2 loops=1)
               Sort Key: advertiser_contact.advertiser_id
                 -> Seq Scan on advertiser_contact
                      (cost=0.00..1.02 rows=2 width=4)
                      (actual time=0.021..0.027 rows=2 loops=1)
Total runtime: 8.978 ms
```

単純に結合を使うと
8.978 ms。
もっと速くできないか？

実際のデバッグ(例4)：IN

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact WHERE advertiser_id
   IN (SELECT advertiser_id FROM advertiser WHERE type = 1);
                                    QUERY PLAN
----------------------------------------------------------------------------------
Aggregate (cost=2.23..2.23 rows=1 width=0)
          (actual time=0.261..0.261 rows=1 loops=1)
  -> Hash Join (cost=2.15..2.23 rows=2 width=0)
               (actual time=0.231..0.246 rows=2 loops=1)
     Hash Cond: ("outer".advertiser_id = "inner".advertiser_id)
       -> HashAggregate (cost=1.12..1.12 rows=8 width=4)
                        (actual time=0.091..0.112 rows=8 loops=1)
            -> Seq Scan on advertiser (cost=0.00..1.10 rows=8 width=4)
                                      (actual time=0.051..0.068 rows=8 loops=1)
               Filter: ("type" = 1)
       -> Hash (cost=1.02..1.02 rows=2 width=4)
               (actual time=0.101..0.101 rows=0 loops=1)
            -> Seq Scan on advertiser_contact
                 (cost=0.00..1.02 rows=2 width=4)
                 (actual time=0.088..0.094 rows=2 loops=1)
Total runtime: 0.422 ms
```

INにしたら速くなった！
8.978 → 0.422 ms

実際のデバッグ(例4)：EXISTS

```
=# EXPLAIN ANALYZE SELECT count(*) FROM advertiser_contact WHERE
    EXISTS (SELECT 1 FROM advertiser
            WHERE advertiser_id=advertiser_contact.advertiser_id AND type = 1);
                         QUERY PLAN
------------------------------------------------------------
Aggregate (cost=3.27..3.27 rows=1 width=0)
          (actual time=0.200..0.201 rows=1 loops=1)
  -> Seq Scan on advertiser_contact
       (cost=0.00..3.26 rows=1 width=0)
       (actual time=0.162..0.179 rows=2 loops=1)
     Filter: (subplan)
     SubPlan
       -> Seq Scan on advertiser
            (cost=0.00..1.12 rows=1 width=0)
            (actual time=0.034..0.034 rows=1 loops=2)
          Filter: ((advertiser_id = $0) AND ("type" = 1))
Total runtime: 0.333 ms
```

EXISTSはさらに速い！
0.422 → 0.333ms

- 1つのクエリに対して何通りものアプローチがある
- 実際のデータシナリオに対してもテストすること

気を付けておくこと

- まず最初に、テーブルがバキュームとアナライズされていることを確かめる
- クエリを1つのプランに対し2回以上実行すること(キャッシュの影響があるため)
- 下方から上方に向かって、不正確な行数の推定を探す
- EXPLAINの出力を確認する
  - 本当の行数 count(*) と 推定行数 rows は一致しているか?
- インデックスを試してみる
- 実際のデータを使う (Slonyでデータを抜いてくる)
- PostgreSQL をアップグレードする / 最新版を使う
  - オプティマイザも新しいバージョンほど賢くなっているので

ヘルプを求める場合は

- まず自分でデバッグしてみる
- PostgreSQLのバージョンを書く
- VACUUMとANALYZEを正確に実行してあること
- EXPLAIN ANALYZEの結果を必ず書く
- クエリ、テーブル、データもできれば含める

pgsql-performance@postgresql.org (英語)
pgsql-jp@ml.postgresql.jp (日本語)

ありがとうございました

- Greg Sabino Mullane
- AndrewSN@#postgresql
- Magnifikus@#postgresql
- Bryan Encina

外部リンク

- オリジナルのスライド資料
  - http://redivi.com/~bob/oscon2005_pgsql_pdf/OSCON_Explaining_Explain_Public.pdf
- PostgreSQL文書
  - EXPLAINの利用
    - http://www.postgresql.jp/document/current/html/using-explain.html
  - 行推定の例
    - http://www.postgresql.jp/document/current/html/row-estimation-examples.html
- Reading PgAdmin Graphical Explain Plans
  - http://www.postgresonline.com/journal/index.php?/archives/27-Reading-PgAdmin-Graphical-Explain-Plans.html