

# PostgreSQLのクエリプランを Explainする

2016年7月12日

データベースシステム

本日の内容は PostgreSQLのドキュメントをもとに作成しています

# postgresql の explain

- EXPLAIN SELECT \* FROM tenk1;

```
-----  
QUERY PLAN  
-----  
Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)
```

- 表示される内容
  - 初期処理の推定コスト
    - 出力用のスキャンが開始される前に消費される時間
  - 全体の推定コスト
    - 結果の行全体が抽出される場合のコスト
  - この計画ノードが出力する行の推定数
  - この計画ノードが出力する(バイト単位の)推定平均幅

# postgresqlのexplain

- 上位ノードのコストにはすべての子ノードのコストが含まれている
- プランナが関与するコストのみ表示される
  - 結果の行をクライアントに送る時間は考慮されない

# コスト計算方法

- EXPLAIN SELECT \* FROM tenk1;

## QUERY PLAN

Seq Scan on tenk1 (cost=0.00..458.00 rows=10000 width=244)

- Sequential Scanの場合
  - seq\_page\_cost:
    - 1ページを読みだすためのコスト 1.0
  - cpu\_tuple\_cost:
    - cpuで1レコードを処理するためのコスト0.01
  - (seq\_page\_cost) x (アクセスするページ数) + (cpu\_tuple\_cost) x (行数)
    - 上記の例だと $1.0 \times 358 + 0.01 \times 10000 = 458$

# 検索条件が付いたとき

- **EXPLAIN SELECT \* FROM tenk1  
WHERE unique1 < 7000;**

手違いで違うSQLに  
なっていたら修正してくだ  
さいm( \_\_ )m

## QUERY PLAN

Seq Scan on tenk1 (cost=0.00..483.00 rows=7033 width=244)  
Filter: (unique1 < 7000)

- Where句がスキャンのフィルタ条件となっている
- 出力するタプル数が7割になっている
- スキャンのコストはほぼ変わらない
- Filterを適用している分だけ少し遅くなっている
- **cpu\_operator\_cost:**
  - 1行にFilterを適用するためのコスト 0.0025

# 検索条件の選択率が低いとき

- EXPLAIN SELECT \* FROM tenk1  
WHERE unique1 < 3;

QUERY PLAN

Index Scan using tenk1\_unique1 on tenk1 (cost=0.00..10.00 rows=2 width=244)  
Index Cond: (unique1 < 3)

— index scan

- 索引を使ったスキャン

— コストが10になっている

# 検索条件の選択率が高めの時

- EXPLAIN SELECT \* FROM tenk1 WHERE  
unique1 < 100;

QUERY PLAN

-----  
Bitmap Heap Scan on tenk1 (cost=2.37..232.35 rows=106 width=244)

Recheck Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..2.37 rows=106 width=0)

Index Cond: (unique1 < 100)

## － 2段階の計画を使用している

- 1段階目: 索引を使ってスキャンして該当する行の場所を検索し、物理的な順序でソートする
- 2段階目: 該当する行を取得する

# Bitmap Index Scanとは

- インデックスを利用して取得するテーブルのデータ行と対応するビットをオンに切り替えたビットマップを作成
- そのビットマップを利用してテーブルの必要な個所をシーケンシャルに読み取る

Tid	1	2	3	4	5	6	7	8	9
Cond	1	0	1	0	0	1	1	0	0



# 条件を複数書いた場合

- `EXPLAIN SELECT * FROM tenk1  
WHERE unique1 < 3 AND stringu1 = 'xxx';`

QUERY PLAN

-----  
Index Scan using tenk1\_unique1 on tenk1 (cost=0.00..10.01 rows=1 width=244)  
  Index Cond: (unique1 < 3)  
  Filter: (stringu1 = 'xxx'::name)

- stringu1は索引がついていない  
→ unique1<3 のIndex ScanにFilterをつける

# 複数の属性に索引がついている場合

- EXPLAIN SELECT \* FROM tenk1  
WHERE unique1 < 100 AND unique2 > 9000;

## QUERY PLAN

Bitmap Heap Scan on tenk1 (cost=11.27..49.11 rows=11 width=244)

Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))

-> **BitmapAnd** (cost=11.27..11.27 rows=11 width=0)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..2.37 rows=106 width=0)

Index Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique2 (cost=0.00..8.65 rows=1042 width=0)

Index Cond: (unique2 > 9000)

# Bitmap Andとは

Tid	1	2	3	4	5	6	7	8	9
Cond	1	0	1	0	0	1	1	0	0

AND

Tid	1	2	3	4	5	6	7	8	9
Cond	0	1	1	1	0	0	1	0	0



Tid	1	2	3	4	5	6	7	8	9
Cond	0	0	1	0	0	0	1	0	0

# 演習：選択率とアルゴリズム

- 選択率によってpostgresqlはどのアルゴリズムを選択するかを実験的に調べよう

# プランナが使う統計情報

reltuples: タプル数, relpages : ページ数

- SELECT relname, relkind, reltuples, relpages  
FROM **pg\_class** WHERE relname LIKE 'tenk1%';

relname	relkind	reltuples	relpages
tenk1	r	10000	358
tenk1_hundred	i	10000	30
tenk1_thous_tenthous	i	10000	30
tenk1_unique1	i	10000	30
tenk1_unique2	i	10000	30

(5 rows)

# ヒストグラムから選択率を求める

- `SELECT histogram_bounds FROM pg_stats WHERE tablename='tenk1' AND attname='unique1';`

```
          histogram_bounds
-----
{0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
             = (1 + (1000 - 993)/(1997 - 993))/10
             = 0.100697
```

```
rows = rel_cardinality * selectivity = 10000 * 0.100697 = 1007 (rounding off)
```

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;
```

```
      QUERY PLAN
```

```
-----
Bitmap Heap Scan on tenk1 (cost=24.06..394.64 rows=1007 width=244)
```

```
  Recheck Cond: (unique1 < 1000)
```

```
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..23.80 rows=1007 width=0)
```

```
        Index Cond: (unique1 < 1000)
```

# 結合をするSQLのクエリプラン

- EXPLAIN SELECT \*  
FROM tenk1 t1, tenk2 t2  
WHERE t1.unique1 < 10  
AND t1.unique2 = t2.unique2;

## QUERY PLAN

Nested Loop (cost=2.37..553.11 rows=106 width=488)

-> **Bitmap Heap Scan** on tenk1 t1 (cost=2.37..232.35 rows=106 width=244)

Recheck Cond: (unique1 < 100)

-> **Bitmap Index Scan** on tenk1\_unique1 (cost=0.00..2.37 rows=106 width=0)

Index Cond: (unique1 < 100)

-> **Index Scan** using tenk2\_unique2 on tenk2 t2 (cost=0.00..3.01 rows=1 width=244)

Index Cond: (t2.unique2 = t1.unique2)

# 演習：前頁のクエリプランを 図式化してみよう

- また、最終コストがなぜこのような値になるか、  
計算式を考えてみよう



# 選択率を変えた場合のプラン

- EXPLAIN SELECT \*  
FROM tenk1 t1, tenk2 t2  
WHERE t1.unique1 < 300 AND t1.unique2 = t2.unique2;

## QUERY PLAN

Hash Join (cost=230.43..713.94 rows=101 width=488)

Hash Cond: (t2.unique2 = t1.unique2)

-> Seq Scan on tenk2 t2 (cost=0.00..445.00 rows=10000 width=244)

-> Hash (cost=229.17..229.17 rows=101 width=244)

-> Bitmap Heap Scan on tenk1 t1 (cost=5.03..229.17 rows=101 width=244)

Recheck Cond: (unique1 < 100)

-> Bitmap Index Scan on tenk1\_unique1 (cost=0.00..5.01 rows=101 width=0)

Index Cond: (unique1 < 100)

# ソートマージを採用するプラン

- EXPLAIN SELECT \*  
FROM tenk1 t1, onek t2  
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

## QUERY PLAN

Merge Join (cost=197.83..267.93 rows=10 width=488)

Merge Cond: (t1.unique2 = t2.unique2)

-> Index Scan using tenk1\_unique2 on tenk1 t1 (cost=0.00..656.25 rows=101 width=244)

Filter: (unique1 < 100)

-> Sort (cost=197.83..200.33 rows=1000 width=244)

Sort Key: t2.unique2

-> Seq Scan on onek t2 (cost=0.00..148.00 rows=1000 width=244)

# 特定のアルゴリズムを 選ばないようにする方法

- SET enable\_<アルゴリズム> = [ON/OFF]
  - 例) 入れ子ループを使わない
    - SET enable\_nestloop = OFF;
- アルゴリズム

bitmapscan	hashagg
hashjoin	indexscan
mergejoin	nestloop
seqscan	sort
tidscan	

# 演習2: 入れ子ループを選んだ理由

- EXPLAIN SELECT \*

FROM tenk1 t1, tenk2 t2

WHERE t1.unique1 < 10

AND t1.unique2 = t2.unique2;

が入れ子ループを選んだ理由を  
コスト計算の面で調査をして考察しよう

7月1日 13:00

提出場所: 図書室 ボックス

# ANALYZE

- analyzeをつけると実際に実行してコストの精度を点検することができる
  - EXPLAIN (ANALYZE, BUFFERS)  
SELECT \* FROM tenk1 t1, tenk2 t2  
WHERE t1.unique1 < 100  
AND t1.unique2 = t2.unique2;

## QUERY PLAN

```
Nested Loop (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..12.700 rows=100 loops=1)
-> Bitmap Heap Scan on tenk1 t1 (cost=2.37..232.35 rows=106 width=244) (actual time=0.878..2.367 rows=100 loops=1)
    Recheck Cond: (unique1 < 100)
    -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..2.37 rows=106 width=0) (actual time=0.546..0.546 rows=100 loops=1)
        Index Cond: (unique1 < 100)
    -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.00..3.01 rows=1 width=244) (actual time=0.067..0.078 rows=1 loops=100)
        Index Cond: (t2.unique2 = t1.unique2)
Total runtime: 14.452 ms
```