OSS-DB Exam Gold
技術解説無料セミナー

@東京 8/6（土）

NTTソフトウェア株式会社

Copyright© NTT Software Corporation. All rights reserved.

アジェンダ

1. 性能改善のために必要な知識
2. インデックスが効かない原因例
   1. 統計情報が正しく取得されていない
   2. 検索条件に式・関数が利用されている
   3. インデックスを利用しても効果がない
   4. 統計情報の精度が足りない
3. 実行時間とネットワーク転送コスト

【別紙】SQLコマンド表

【参考】PostgreSQL 高度技術者育成テキスト 抜粋

Copyright© NTT Software Corporation. All rights reserved.

1. 性能改善のために必要な知識

- 性能改善は「問題個所の特定」「原因の特定」「解決策の立案・実施」の流れで行われる
- PostgreSQLの関連機能や出力内容の読み方（基礎知識）の暗記だけでは「原因の特定」で躓きやすい
- 基礎知識は公開資料で比較的容易に独学が可能
- 原因はある程度はパターン化されているため、「原因の特定」はまず既知のパターンとの照合作業から始まる

NTTソフトウェア
Copyright© NTT Software Corporation. All rights reserved.

1. 性能改善のために必要な知識

- 基礎知識を習得するための参考資料
  - PostgreSQL勉強会の公開資料
    - http://www.postgresql.jp/wg/shikumi/
  - 第30回がお勧め
  - 公式ドキュメント
    - http://www.postgresql.jp/document/current/
  - Googleサイト内検索を利用すると便利
  - 書籍
    - 内部構造から学ぶPostgreSQL 設計・運用計画の鉄則
      - http://gihyo.jp/book/2014/978-4-7741-6709-1
    - LPI-Japan OSS-DB Gold 認定教材 PostgreSQL 高度技術者育成テキスト
      - http://www.amazon.co.jp/dp/B00P4WD4QG
    - Goldの試験範囲は網羅性が高いのでお勧め

「OSS-DB Exam Gold 技術解説無料セミナー @東京 8/6（土）」
NTTソフトウェア
Copyright© NTT Software Corporation. All rights reserved.

2. インデックスが効かない原因例

1. 統計情報が正しく取得されていない

公式ドキュメントには以下のような記載がある

> デフォルトのPostgreSQLの設定では、自動バキュームデーモンが、データが最初にロードされた時や通常の操作を通して変更された時にテーブルの自動解析まで面倒をみます。 もし自動バキュームが無効にしているならばANALYZEは定期的に、もしくは、テーブルの内容に大きな変更がある度に行うことを推奨します。

http://www.postgresql.jp/document/9.5/html/sql-analyze.html の注釈

但し、自動バキュームが有効であったとしても、「テーブルの内容に大きな変更がされた時」は必要に応じて手動でVACUUM ANALYZEを実行すべきであり、

「データが最初にロードされた時」は、直後に確実に手動でVACUUM ANALYZEを実行すべきである

Copyright© NTT Software Corporation. All rights reserved.

2. インデックスが効かない原因例

2. 検索条件に式・関数が利用されている

式・関数インデックスの作成で解決することもある

関数インデックス作成例

```
=> CREATE INDEX ON test (lower(mail_address));
```

但し、式・関数インデックスを利用しない他の方法で解決可能ならば積極的に利用すべきではない

また、利用する関数の変動性分類によってはそもそも関数インデックスを作成できないこともある

【参考】Lt 関数の変動性分類についておさらいしてみる。
- http://www.slideshare.net/toshiharada/lt-41040251

Copyright© NTT Software Corporation. All rights reserved.

2. インデックスが効かない原因例

3. インデックスを利用しても効果がない

例えば、大量レコードを持つテーブルの列1の値が1レコードのみAで、その他が全てBの場合、検索条件としてBを指定しても、インデックスは利用されない

カーディナリティが低い列に対する検索処理がボトルネックとなっている場合は、検索条件等に誤りがないか確認する必要がある（WHERE句を見直す等）

- 複雑なクエリやビューを利用しているケースでは特に見落とされがち

Copyright© NTT Software Corporation. All rights reserved.

2. インデックスが効かない原因例

4. 統計情報の精度が足りない

統計情報が正しく取得されていたとしても、精度が足りない場合はインデックスが利用されない非効率な実行計画が選択される可能性がある

> 【参考】行推定の例
>
> http://www.postgresql.jp/document/9.5/html/row-estimation-examples.html

ANALYZE直後であっても予測行数と実態が乖離している場合はALTER文で列単位での精度の変更を検討する

但し、問題がテスト環境で発生している場合、同様の問題が本番環境でも発生するかを検討する必要がある（テストデータの出来が良くない場合など）

Copyright© NTT Software Corporation. All rights reserved.

3. 実行時間とネットワーク転送コスト

- 実行時間
  - PostgreSQLには、クエリの実行時間を取得する方法が複数あるが、各実行時間の正確な意味を把握できていない場合、ボトルネック箇所を見誤る可能性が高い
  - 以下の実行時間は、全て算出方法が異なる
    - ・log_min_duration_statement で出力されるduration
    - ・EXPLAIN ANALYZEで出力されるExecution time
    - ・psqlの¥timingで出力されるTime
    - ・アプリケーション側のログに出力される実行時間
  - ネットワーク転送コストが実行時間に含まれているかを強く意識すべきである

Copyright© NTT Software Corporation. All rights reserved.

3. 実行時間とネットワーク転送コスト

- ネットワーク転送コスト
  - クラウド環境においては、ネットワーク帯域幅だけでなくネットワークレイテンシ（遅延）もボトルネックとなるケースが多い
  - PostgreSQLではクライアントサーバ間の通信プロトコルにTCP/IPが用いられるため、TCP/IPの影響をそのまま受けることになる
    - ・スリーウェイハンドシェイクでの接続確立
    - ・TCPスロースタート …etc

Copyright© NTT Software Corporation. All rights reserved.

3. 実行時間とネットワーク転送コスト

- log_min_duration_statementで出力されるdurationには、ネットワーク転送コストが含まれない

クライアント側の¥timingでの実行時間

```
 $ psql -h 遠隔地サーバのIP
=> ¥timing
タイミングは on です。
=> SELECT now();
              now
-------------------------------
 2015-06-12 12:00:59.19005+00
(1 行)
時間: 84.298 ms
```

サーバ側での実行時間は1ms未満だが、クライアント側からみたクエリを発行してから結果を取得するまでの時間は80ms以上もかかっている。

これはクライアントサーバ間に伝播遅延（レイテンシ）が発生しているためである。

※つまり、ネットワーク転送コストがボトルネックとなっても、サーバ側のログからは問題個所の特定が困難となる

サーバ側のログに出力される実行時間

```
LOG:  duration: 0.113 ms  statement: SELECT now();
```


NTTソフトウェア Copyright© NTT Software Corporation. All rights reserved.

3. 実行時間とネットワーク転送コスト

- ネットワーク転送コストの削減方法
  - コネクションプーリングを利用する
    - ・コネクション確立時のコストを低減する
  - クエリの発行回数を抑える
    - ・ストアドプロシージャを利用する
    - ・1度のクエリで送受信するデータ量を増やす
  - 通信の圧縮を有効にする
    - ・sslcompression
  - TCPのチューニングを行う
    - ・各種ウィンドウサイズを大きくする
    - ・tcp_slow_start_after_idleをoffにする　…etc
  - そもそも、ネットワーク転送コストが問題となるような構成にしない

「OSS-DB Exam Gold 技術解説無料セミナー @東京 8/6（土）」

NTTソフトウェア Copyright© NTT Software Corporation. All rights reserved.

1.統計情報が正しく取得されていない

```
1  1.統計情報が正しく取得されていない
2  test=> CREATE TABLE test1 AS SELECT generate_series(1,50) col1;
3  test=> CREATE TABLE test2 AS SELECT generate_series(1,10000) col1;
4  test=> CREATE INDEX ON test2(col1);
5  test=> SELECT relname,reltuples FROM pg_class WHERE relname LIKE '%test%';
6      relname     | reltuples
7  ----------------+-----------
8   test1          |         0
9   test2          |     10000
10  test2_col1_idx |     10000
11 (3 行)
12
13 test=> SELECT count(*) FROM pg_stats WHERE tablename LIKE '%test%';
14  count
15 -------
16      0
17 (1 行)
18
19 test=> EXPLAIN ANALYZE SELECT * FROM test1 NATURAL JOIN test2;
20                                            QUERY PLAN
21 ----------------------------------------------------------------------------------------------
22  Merge Join  (cost=180.07..2567.57 rows=127500 width=4)
23              (actual time=0.220..0.347 rows=50 loops=1)
24    Merge Cond: (test2.col1 = test1.col1)
25    ->  Index Only Scan using test2_col1_idx on test2  (cost=0.29..450.29 rows=10000 width=4)
26                                                       (actual time=0.052..0.082 rows=51 loops=1)
27          Heap Fetches: 51
28    ->  Sort  (cost=179.78..186.16 rows=2550 width=4)
29              (actual time=0.159..0.178 rows=50 loops=1)
30          Sort Key: test1.col1
31          Sort Method: quicksort  Memory: 27kB
32          ->  Seq Scan on test1  (cost=0.00..35.50 rows=2550 width=4)
33                                 (actual time=0.086..0.108 rows=50 loops=1)
34 Planning time: 0.336 ms
35 Execution time: 0.409 ms
36 (10 行)
37
38 test=> SELECT count(*) FROM pg_stats WHERE tablename LIKE '%test%';
39  count
40 -------
41      1
42 (1 行)
```

CREATE INDEX により test2 の reltaples は更新されるが、test1 は更新されない

AUTO VACUUM による自動解析が行われるまで、統計情報は取得されない

1 度も ANALYZE が行われていないため、test1 の推定レコード数が実態と乖離することになる

test2 は自動解析されるが、test1 は更新レコード数が autovacuum_analyze_threshold を上回らないため自動解析の対象とならない

```
43   test=> EXPLAIN ANALYZE SELECT * FROM test1 NATURAL JOIN test2;
44                                             QUERY PLAN
45   ------------------------------------------------------------
46   Hash Join  (cost=270.00..340.56 rows=2550 width=4)
47              (actual time=26.443..26.583 rows=50 loops=1)
48     Hash Cond: (test1.col1 = test2.col1)
49     ->  Seq Scan on test1  (cost=0.00..35.50 rows=2550 width=4)
50                            (actual time=0.016..0.052 rows=50 loops=1)
51     ->  Hash  (cost=145.00..145.00 rows=10000 width=4)
52               (actual time=26.373..26.373 rows=10000 loops=1)
53           Buckets: 16384  Batches: 1  Memory Usage: 480kB
54           ->  Seq Scan on test2  (cost=0.00..145.00 rows=10000 width=4)
55                                  (actual time=0.014..10.627 rows=10000 loops=1)
56    Planning time: 0.291 ms
57    Execution time: 26.670 ms
58   (8 行)
59
60
61   test=> VACUUM ANALYZE test1;
62   test=> VACUUM ANALYZE test2;
63
64   test=> EXPLAIN ANALYZE SELECT * FROM test1 NATURAL JOIN test2;
65                                             QUERY PLAN
66   -----------------------------------------------------------------------------------------------
67    Merge Join  (cost=3.20..5.39 rows=50 width=4)
68                (actual time=0.074..0.238 rows=50 loops=1)
69      Merge Cond: (test2.col1 = test1.col1)
70      ->  Index Only Scan using test2_col1_idx on test2  (cost=0.29..270.29 rows=10000 width=4)
71                                                         (actual time=0.015..0.042 rows=51 loops=1)
72            Heap Fetches: 0
73      ->  Sort  (cost=2.91..3.04 rows=50 width=4)
74                (actual time=0.053..0.071 rows=50 loops=1)
75            Sort Key: test1.col1
76            Sort Method: quicksort  Memory: 27kB
77            ->  Seq Scan on test1  (cost=0.00..1.50 rows=50 width=4)
78                                   (actual time=0.003..0.024 rows=50 loops=1)
79   Planning time: 0.260 ms
80   Execution time: 0.287 ms
81   (10 行)
```

test1 の統計情報は変化しないが、test2 の統計情報が更新された結果 Hash Join のコストが下がり、選択される実行計画が変化する

ハッシュ作成に時間がかかり、統計情報更新前より実行時間がかかるようになる

VACUUM ANALYZE は対象を省略すれば全てのテーブルに対して実行されるため、初回ロード時は対象を指定する必要はない

統計情報を正しく更新すれば、適切な実行計画が選択される。
また、VACUUM により Visibility Map が更新されるため、
IndexOnlyScan も高速化する

1 **2.検索条件に式・関数が利用されている**

2

```
3  test=> CREATE TABLE test3 AS
4  test->   SELECT DISTINCT mail_address FROM
5  test->     (   SELECT
6  test(>             string_agg (localpart, '') || '@example.com' mail_address
7  test(>         FROM
8  test(>             (   SELECT
9  test(>                     chr (64 + (random () * 58) ::int) AS localpart,gcol
10 test(>                 FROM generate_series (1, 10) length,
11 test(>                      generate_series (1, 10000) gcol
12 test(>             ) foo1
13 test(>         WHERE localpart ~ '[a-zA-Z]' GROUP BY gcol
14 test(>     ) foo2 ;
15
16 test=> SELECT * FROM test3 LIMIT 3;
17      mail_address
18 -----------------------
19  QvZznaaD@example.com
20  OFaVhSFPJ@example.com
21  AIXpAsonc@example.com
22 (3 行)
```

テストデータとして、大文字小文字混交の
メールアドレスデータを作成する

```
23
24 test=> CREATE INDEX ON test3 (mail_address);
25 test=> VACUUM ANALYZE test3;
26
27 test=> EXPLAIN ANALYZE SELECT * FROM test3 WHERE mail_address = 'qvzznaad@example.com';
28                                                QUERY PLAN
29 ------------------------------------------------------------------------------------------------
30  Index Only Scan using test3_mail_address_idx on test3  (cost=0.29..4.30 rows=1 width=21)
31                                         (actual time=0.083..0.085 rows=0 loops=1)
32    Index Cond: (mail_address = 'qvzznaad@example.com'::text)
33    Heap Fetches: 0
34 Planning time: 0.211 ms
35 Execution time: 0.199 ms
36 (5 行)
37
38
39
40
41
42
```

大文字小文字が一致していないため、
インデックスは機能するが
取得件数は0件となる

```
43  test=> EXPLAIN ANALYZE SELECT * FROM test3 WHERE
44  test->    lower(mail_address) = lower('QvZznaaD@example.com');
45                                          QUERY PLAN
46  -----------------------------------------------------------------------------------------------
47   Seq Scan on test3  (cost=0.00..214.00 rows=50 width=21) (actual time=0.023..5.131 rows=1 loops=1)
48     Filter: (lower(mail_address) = 'qvzznaad@example.com'::text)
49     Rows Removed by Filter: 9999
50  Planning time: 0.216 ms
51  Execution time: 8.900 ms
52  (5 行)
53
54  test=> CREATE UNIQUE INDEX ON test3 (lower(mail_address));
55  test=> VACUUM ANALYZE test3;
56
57  test=> EXPLAIN ANALYZE SELECT * FROM test3 WHERE
58  test->    lower(mail_address) = lower('QvZznaaD@example.com');
59                                          QUERY PLAN
60  -----------------------------------------------------------------------------------------------
61   Index Scan using test3_lower_idx on test3  (cost=0.29..8.30 rows=1 width=21)
62                                              (actual time=0.044..0.045 rows=1 loops=1)
63     Index Cond: (lower(mail_address) = 'qvzznaad@example.com'::text)
64  Planning time: 0.363 ms
65  Execution time: 0.079 ms
66  (3 行)
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
```

検索条件に関数を利用すれば
大文字小文字に関係なく
結果を取得できるが、通常の
インデックスは機能しない

UNIQUE インデックスとする
ことで、大文字小文字違いの
重複登録を防げる

関数インデックスを作成すれば
大文字小文字に関係なく結果を取得でき、
かつインデックスが有効に機能する

```
85   test=> CREATE TABLE test4 AS SELECT
86   test-> generate_series('2010-01-01 00:00:00'::timestamptz,'2015-01-01 00:00:00','1 hour') col1;
87
88   test=> VACUUM ANALYZE test4;
89   test=> SELECT * FROM test4 LIMIT 3;
90             col1
91   ------------------------
92    2010-01-01 00:00:00+09
93    2010-01-01 01:00:00+09
94    2010-01-01 02:00:00+09
95   (3 行)
96
97   test=> EXPLAIN ANALYZE SELECT * FROM test4 WHERE cast(col1 AS date) = '2010-01-01';
98                                        QUERY PLAN
99   ----------------------------------------------------------------------------------------------
100   Seq Scan on test4  (cost=0.00..851.38 rows=219 width=8)
101                      (actual time=0.051..22.132 rows=24 loops=1)
102     Filter: ((col1)::date = '2010-01-01'::date)
103     Rows Removed by Filter: 43801
104  Planning time: 0.292 ms
105  Execution time: 42.606 ms
106  (5 行)
107
108  test=> CREATE INDEX ON test4 (cast(col1 AS date));
109  ERROR:  functions in index expression must be marked IMMUTABLE
110
111  test=> CREATE FUNCTION my_date_cast(timestamptz) RETURNS date AS
112  test->  'SELECT cast($1 AS date)' LANGUAGE SQL IMMUTABLE;
113
114  test=> CREATE INDEX ON test4 (my_date_cast(col1));
115  test=> VACUUM ANALYZE test4;
116
117  test=> EXPLAIN ANALYZE SELECT * FROM test4 WHERE my_date_cast(col1) = '2010-01-01';
118                                        QUERY PLAN
119  ----------------------------------------------------------------------------------------------
120   Index Scan using test4_my_date_cast_idx on test4  (cost=0.54..8.96 rows=24 width=8)
121                                                     (actual time=0.024..0.049 rows=24 loops=1)
122     Index Cond: (my_date_cast(col1) = '2010-01-01'::date)
123   Planning time: 0.352 ms
124   Execution time: 0.250 ms
125  (4 行)
126
```

但し、全ての関数をインデックスに利用できるわけではない。
テストデータとして、timestamptz 型のデータを作成する

条件を年月日とし、該当日の件数を取得する。方法は複数あるが、今回は CAST 関数を用いて date 型に変換して取得するものとする

変動性分類が IMMUTABLE でない関数は利用できない

自作関数 my_date_cast を作成し、cast 関数を強制的に IMMUTABLE にする

関数インデックスが有効に機能しているように見えるが…

```
127  test=> INSERT INTO test4 VALUES ('2016-08-06'::date);
128
129  test=> SHOW timezone;
130   TimeZone
131  ----------
132   Japan
133  (1 行)
134
135  test=> SET timezone TO 'GMT+12';
136
137  test=> SELECT * FROM test4 WHERE my_date_cast(col1) = '2016-08-06'::date;
138            col1
139  ------------------------
140   2016-08-05 03:00:00-12
141  (1 行)
142
143  test=> SET enable_indexscan  TO OFF;
144  test=> SET enable_indexonlyscan TO OFF;
145  test=> SET enable_bitmapscan TO OFF;
146
147  test=> SELECT * FROM test4 WHERE my_date_cast(col1) = '2016-08-06'::date;
148   col1
149  ------
150  (0 行)
```

関数インデックスの場合、データ更新時に関数が実行される。
つまり、timezone が Japan の状態でインデックスが生成される

関数インデックスの場合、選択時に関数は実行されない。
つまり、現在の timezone は GMT+12 なのに、timezone が Japan であることが前提のインデックスが利用される

インデックスを無効にすると、同様の検索条件でも検索結果が 0 件となる。
インデックスが利用されない場合は timezone が GMT+12 の状態で col1 に対して関数が実行される。
つまり、選択される実行計画により、検索結果が変わってしまうことになる

3.インデックスを利用しても効果がない

```
test=> CREATE TABLE test5 AS SELECT 'B'::text col1 FROM generate_series(1,10000);
test=> INSERT INTO test5 VALUES ('A');
test=> CREATE INDEX ON test5(col1);
test=> VACUUM ANALYZE test5;

test=> EXPLAIN ANALYZE SELECT * FROM test5 WHERE col1 = 'A';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Only Scan using test5_col1_idx on test5  (cost=0.29..4.30 rows=1 width=2)
                                           (actual time=0.056..0.059 rows=1 loops=1)
   Index Cond: (col1 = 'A'::text)
   Heap Fetches: 0
Planning time: 1.786 ms
Execution time: 0.182 ms
(5 行)

test=> EXPLAIN ANALYZE SELECT * FROM test5 WHERE col1 = 'B';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on test5  (cost=0.00..170.01 rows=10000 width=2)
                    (actual time=0.057..4.710 rows=10000 loops=1)
   Filter: (col1 = 'B'::text)
   Rows Removed by Filter: 1
Planning time: 0.184 ms
Execution time: 19.632 ms
(5 行)

test=> SELECT tablename,attname,n_distinct,most_common_vals,most_common_freqs FROM pg_stats
test->   WHERE tablename = 'test5';

 tablename | attname | n_distinct | most_common_vals | most_common_freqs
-----------+---------+------------+------------------+-------------------
 test5     | col1    |          2 | {B}              | {0.9999}
(1 行)
```

テストデータとして、文字列Bが1万行、文字列Aが1行のテーブルを作成する

検索条件をAとした場合はインデックスが有効に動作するが、Bの場合ではインデックスは利用されない

統計情報はpg_statsで確認可能

```
43   test=> CREATE INDEX ON test5(col1) WHERE col1 = 'A';
44
45   test=> SELECT pg_size_pretty(pg_relation_size('test5_col1_idx')) 通常,
46   test->   pg_size_pretty(pg_relation_size('test5_col1_idx1')) 部分;
47     通常  | 部分
48   --------+-------
49    240 kB | 16 kB
50   (1 行)
```

インデックスの大半が利用されない場合は、
部分インデックスの作成を検討すべき

4.統計情報の精度が足りない

```
test=> CREATE TABLE test6 AS WITH r AS
test->   (SELECT md5(random()::text) col1 FROM generate_series(1,101))
test->   SELECT col1 from r,generate_series(1,10000);

test=> INSERT INTO test6 SELECT md5(random()::text) FROM generate_series(1,1000000);

test=> VACUUM ANALYZE test6;

test=> SELECT col1,count(col1) FROM test6 WHERE col1 NOT IN(
test(>   SELECT unnest(most_common_vals::text::text[]) FROM pg_stats
test(>   WHERE tablename = 'test6' AND attname='col1')
test(>   GROUP BY col1 ORDER BY count(col1) DESC LIMIT 1;
               col1               | count
----------------------------------+-------
 1fc520e04c8c5a7076a98f3dae52fa1a | 10000
(1 行)
```

most_common_vals は特殊な配列で格納されているため、一度テキスト型にキャストをしなければならない

2 行目~は 101 種のランダム文字列が 1 万行ずつ格納されるテストテーブルを作成している。
6 行目はランダムな文字列が格納されているレコードを 100 万行追加している。
10 行目~は pg_stats の most_common_vals から外れた値で、最も件数の多い文字列を確認している。

```
test=> EXPLAIN ANALYZE SELECT * FROM test6 WHERE col1 = '1fc520e04c8c5a7076a98f3dae52fa1a';
                                            QUERY PLAN
-------------------------------------------------------------------------------------------
 Seq Scan on test6  (cost=0.00..41875.00 rows=34 width=33)
                    (actual time=25.497..558.873 rows=10000 loops=1)
   Filter: (col1 = '1fc520e04c8c5a7076a98f3dae52fa1a'::text)
   Rows Removed by Filter: 2000000
Planning time: 0.112 ms
Execution time: 274.794 ms
(5 行)
```

VACUUM ANALYZE 直後なのに、予測行数と実際の取得行数が乖離している

```
43  test=> SHOW default_statistics_target ;
44   default_statistics_target
45  ---------------------------
46   100
47  (1 行)
48
49  test=> ALTER TABLE test6 ALTER COLUMN col1 SET STATISTICS 101;
50  test=> VACUUM ANALYZE test6;
51
52  test=> SELECT col1,count(col1) FROM test6 WHERE col1 NOT IN(
53  test(>   SELECT unnest(most_common_vals::text::text[]) FROM pg_stats
54  test(>   WHERE tablename = 'test6' AND attname='col1')
55  test(>   GROUP BY col1 ORDER BY count(col1) DESC LIMIT 1;
56                col1              | count
57  ----------------------------------+-------
58   0f5934b78e846f15b5c02bd78086f77d |     2
59  (1 行)
60
61
62  test=> EXPLAIN ANALYZE SELECT * FROM test6 WHERE col1 = '1fc520e04c8c5a7076a98f3dae52fa1a';
63                                                   QUERY PLAN
64  -----------------------------------------------------------------------------------------------
65   Seq Scan on test6  (cost=0.00..41875.00 rows=10614 width=33)
66                      (actual time=25.497..558.873 rows=10000 loops=1)
67     Filter: (col1 = '1fc520e04c8c5a7076a98f3dae52fa1a'::text)
68     Rows Removed by Filter: 2000000
69  Planning time: 0.107 ms
70  Execution time: 275.934 ms
71  (5 行)
```

統計情報の精度はテーブルの列単位に設定可能

2 行目のクエリで作成された 101 種の値が全て most_common_vals に登録される

2 行目のクエリで作成された 101 種の値が検索条件の場合は、どの値でも適切な予測行数が算出されるようになる

- ４章　性能監視とパフォーマンス -

テーブル/カラム統計情報

- 概要
  - テーブルのデータ状況に関する集計情報
  - プランナが実行計画の作成時に参照する
    - テーブル統計情報はpg_classシステムカタログに格納され、VACUUM、ANALYZEおよびCREATE INDEXなど一部のDDLコマンドで更新される
    - カラム統計情報はpg_statisticシステムカタログに格納され、ANALYZEコマンドで更新される

テーブル/カラム統計情報は、テーブルのデータ状況に関する集計情報です。

アクセス統計情報と名称が似ていますが、テーブル/カラム統計情報はプランナが実行計画の作成時に参照する情報となります。

一般的に「統計情報」とのみ呼称された場合は、テーブル/カラム統計情報を指しているケースが多いです。

テーブル/カラム統計情報

- pg_class
  - テーブル単位の統計情報が格納される
    - ・relpages列にテーブルのページ数
    - ・reltuples列にテーブルの行数
- pg_statistic
  - 列単位に統計情報が格納される
  - 実データの一部が格納されるため、一般ユーザは参照できない
  - 一般ユーザが参照するためのビューとしてpg_statsが用意されている

テーブルや列の情報などのスキーマメタデータと内部的な情報を格納する場所のことをシステムカタログと呼び、それ自体もPostgreSQLのテーブルの形式で管理されています。

pg_classはシステムカタログで、テーブルをはじめとして、ビューやインデックスなどの情報を扱っています。

名称・所有者・テーブル空間といった多くの列を保有しますが、その中でもプランナが統計情報を作成する際に特に参照する列はrelpages列とreltuples列となります。

これらの統計情報はプランナが使用する推測値で、VACUUMやANALYZEおよびCREATE INDEX等により更新されます。そのため、常に最新の情報が格納されているわけではありません。

pg_statisticシステムカタログでは、カラムに関する統計情報を扱っています。例えば、あるカラムに登録されている値とその分布、といった情報が格納されており、問い合わせ時にプランナがこの情報を参照しています。

pg_statisticには実データの一部が格納されることになるため、誰でも参照できるようになっていると機密保持上問題があります。

そのため、pg_statisticは一般ユーザは参照できません。代わりに、一般ユーザが参照しても問題ない範囲に限定したシステムビューとしてpg_statsが用意されています。

実行計画

- 概要
  - 与えられたSQL文に対し、プランナが統計情報を参照して作成する
  - SQL文が参照するテーブルをスキャンする方法やテーブルを結合するアルゴリズム等を示す
  - SQL文の前にEXPLAINを付与して実行する
    - EXPLAINは実行計画を表示するのみで実際にSQLは実行されないがANALYZEオプションを付与すると実際にSQLが実行され、実行結果に基づく情報も併せて表示される

```
=>   EXPLAIN SELECT 1;
                      QUERY PLAN
     ------------------------------------------
      Result  (cost=0.00..0.01 rows=1 width=0)
```

実行計画は、SQL文が参照するテーブルをスキャンする方法や、テーブルを結合するアルゴリズム等を示します。

実行計画は、その時点の統計情報や各種パラメータを参照してプランナが生成します。

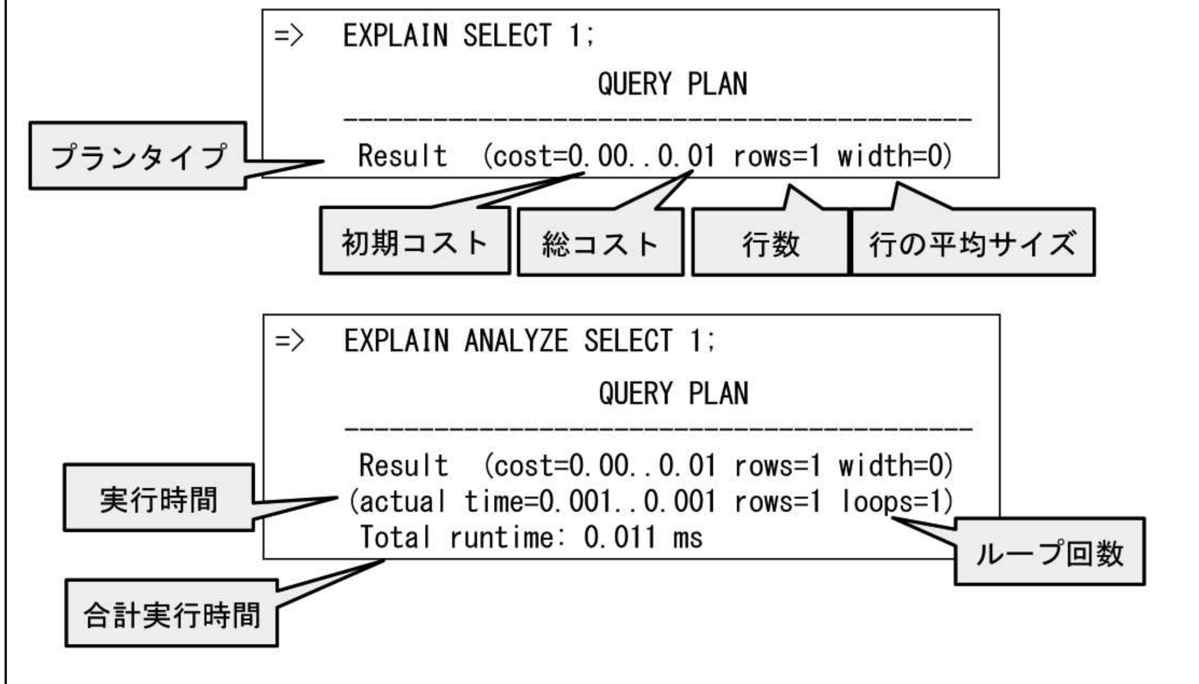
内部的には、一つのSQL文に対して一つの実行計画のみが生成されるわけではありません。同一の結果を取得するための方法は一つとは限らず、複雑なSQL文になるほど、実行計画のパターンは増加することになります。その複数の実行計画の中から、最も効率的と思われるものが最終的には選択されることになります。

統計情報が正しくない状態等では、最適な実行計画が選択されないケースがあるため、各種チューニングが必要となります。

実行計画は、発行するSQL文の文頭にEXPLAINを付与することで内容を確認できます。この場合、SQLそのものは実際には発行されず、表示される情報も推測値のみです。

EXPLAIN ANALYZEを付与した場合は、実際にSQLが実行され、実行結果に基づく情報も併せて表示されます。SELECT文であれば実データに影響を及ぼしませんが、INSERT、UPDATE、DELETE、CREATE TABLE AS、EXECUTE文に対して、実データに影響を与えないようにEXPLAIN ANALYZEを実行したい場合は、以下の方法を使用してください。

```
BEGIN;
EXPLAIN ANALYZE ...;
ROLLBACK;
```



プランタイプ	エグゼキュータが実際にデータ処理するための具体的な手順を指します。 テーブルスキャンやテーブルの結合等、複数の種類があります。 詳細は後述します。
初期コスト （costの左側）	最初の行を取得する前までのコストです。 コストはプランナコスト定数と統計情報を基に算出された想定値で、相対的な値となります。（実行時間ではありません） ソート実行時やハッシュ作成時等に初期コストが発生します。
総コスト （costの右側）	全ての行を取得するまでのコストです。
行数 （rows）	統計情報を基に推測される取得行数です。ANALYZEオプションを付与した場合は実際の取得行数も併せて表示されるため、統計情報が正しく更新されているかの判断材料とすることができます。
行の平均サイズ （width）	統計情報を基に推測される1行あたりのバイト単位でのサイズです。
実行時間 （actual time）	該当処理の1回当たりの平均時間です。（単位はミリ秒）
ループ回数 (loops)	処理によっては複数回実行されることがあり、その回数がループ回数として表示されます。
合計実行時間 (Total runtime)	エグゼキュータの起動・停止時間等を含めた合計時間です。 データをクライアントに転送する時間は含まれないことに注意が必要です。

実行計画

- EXPLAINおよびEXPLAIN ANALYZEの実行結果

```
=>  EXPLAIN ANALYZE SELECT 1 UNION SELECT 2;
                                 QUERY PLAN
    -----------------------------------------------------------------------------
     Unique  (cost=0.05..0.06 rows=2 width=0)
             (actual time=0.004..0.005 rows=2 loops=1)
       ->  Sort  (cost=0.05..0.06 rows=2 width=0)
                 (actual time=0.004..0.004 rows=2 loops=1)
             Sort Key: (1)
             Sort Method:  quicksort  Memory: 25kB
             ->  Append  (cost=0.00..0.04 rows=2 width=0)
                         (actual time=0.001..0.002 rows=2 loops=1)
                   ->  Result  (cost=0.00..0.01 rows=1 width=0)
                               (actual time=0.001..0.001 rows=1 loops=1)
                   ->  Result  (cost=0.00..0.01 rows=1 width=0)
                               (actual time=0.000..0.001 rows=1 loops=1)
     Total runtime: 0.015 ms
```

　実行計画は計画ノードのツリーで構成されます。

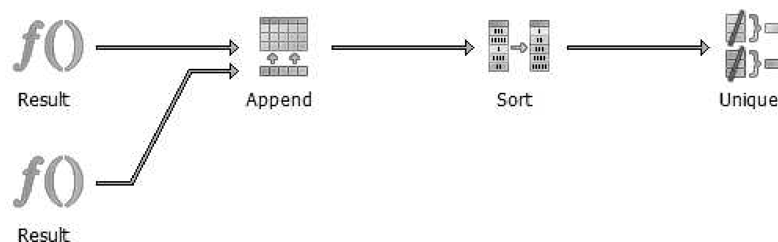
　取得結果に対して結合、集約、ソート等の操作が必要な場合、計画ノードの上位にノードが追加されます。計画ノードの階層関係はインデントと -> によって表現されます。

　実際の処理は子ノードから実行されます。親ノードのコストは子ノードのコストを含んだ形で表示されることになります。よって、最初の1行目（最上位ノード）には、計画全体の実行コスト推定値が含まれます。プランナはこの値が最小になるように動作します。

　コストは実行時間ではない相対的な値であるため、実行時間と単純に比較をしても意味がありません。「各子ノードのコスト」が「計画全体のコスト」に占める割合と、「各子ノードの実行時間」が「合計実行時間」に占める割合を比較し、乖離している箇所がないか確認をします。**この値が乖離している場合、プランナコスト定数が正しく設定されていないか、統計情報が正しく集計されていない可能性があります。**

　複雑な実行計画を解析する際には、pgAdminⅢ等のツールで実行計画を図画することで効率的な解析を望めます。

pgAdminⅢでの表示例



- 4章　性能監視とパフォーマンス -

スロークエリの検出

- log_min_duration_statement
  - 設定した時間以上の処理時間がかかった問い合わせの処理時間をログに出力する
    - ・-1で無効化（デフォルト）
    - ・0に設定すれば全問い合わせの実行時間が出力される
- auto_explain
  - 設定した時間以上の処理時間がかかった問い合わせの処理時間および実行計画をログに出力する
    - ・モジュールを追加インストールしなければならない

log_min_duration_statementでは、クエリの実行に指定したミリ秒数以上かかった場合に、それぞれのクエリの実行に要した時間をログに記録します。0に設定すれば、すべての文の実行時間が出力されます。

auto_explainモジュールは、手動でEXPLAINの実行を必要とせず、自動的に遅い文の実行計画をログに出力する手段を提供します。大きなアプリケーションにおける最適化されていない問い合わせを追跡するのに特に有用です。

このモジュールはSQLでアクセスできる関数を提供しません。使用するには、LOAD文を用いてセッションごとに個別に読み込む必要があります。

実行するためにはスーパーユーザでなければならず、また、最低限、auto_explain.log_min_durationを設定しなければ動作しません。

```
LOAD 'auto_explain';
SET auto_explain.log_min_duration = '5ms';
```

また、postgresql.confのshared_preload_librariesにauto_explainを含めることで、全てのセッションで事前にモジュールをロードすることができます

```
$ grep auto_explain postgresql.conf
  shared_preload_libraries = 'auto_explain'
  auto_explain.log_min_duration = '3s'
```

オーバヘッドは当然発生しますが、想定外に低速なクエリが発生した場合であっても、自動的に該当クエリの実行計画がログに出力されるようになります。

スロークエリの検出

- pg_stat_statements
  - 実行された全てのSQL文の実行時の統計情報を記録する
    - 実行回数やSQLの累計の処理時間を確認できるため、スロークエリの解析に利用できる
  - postgresql.confでパラメータを設定し、モジュールを追加インストールしなければならない
  - Version9.1までは、クエリパラメータの値が異なると別のクエリとして集計されていたが、Version9.2で正規化しての集計が可能となった

pg_stat_statementsモジュールは、サーバで実行されたすべてのSQL文の実行時の統計情報を記録する手段を提供します。

このモジュールは追加の共有メモリを必要とするため、postgresql.confのshared_preload_librariesにpg_stat_statementsを追加してモジュールをロードしなければなりません。Version9.1以降であれば、以下の例のようにCREATE EXTENSION文で導入可能です。このモジュールによって収集された統計情報は、pg_stat_statementsというシステムビューを通して利用することができます。

total_time列では、該当のクエリの累計時間を取得できます。各回の実行時間は長くはなくとも、頻繁に呼び出されることでシステム全体の遅延に影響を及ぼしているようなクエリを見つけ出すことができます。（各回の実行時間が短いクエリの場合、log_min_duration_statementによる取得は困難です）

ビューの統計情報の値のリセットには、 pg_stat_statements_reset()関数を呼び出します。

```
=>   SHOW shared_preload_libraries ;
     -[ RECORD 1 ]------------+-------------------
     shared_preload_libraries | pg_stat_statements

=>   CREATE EXTENSION pg_stat_statements ;
=>   ¥d
      スキーマ |        名前         |   型   |  所有者
     ----------+---------------------+--------+----------
      public   | pg_stat_statements | ビュー | postgres
```

pg_stats

列名	概要
schemaname	スキーマ名
tablename	テーブル名
attname	列名
inherited	真の場合、この行には指定されたテーブルの値だけではなく、 継承関係の子の列が含まれる
null_frac	NULLとなっている列項目の割合
avg_width	列項目のバイト単位による平均幅
n_distinct	ゼロより大きい値は列内の個別値の推定数
most_common_vals	列の中の最も共通した値のリスト
most_common_freqs	最も一般的な値、もしくは要素の出現頻度のリスト （つまり行の総数で出現数を割算した数字）
histogram_bounds	列の値を満遍なく似たような数でグループに分配した値のリスト
correlation	ディスク上の物理的な行の並び順と論理的な値の並び順の相関率
most_common_elems	列の値の中で最もよく出現する非NULLの要素値のリスト
most_common_elem_freqs	最も一般的な要素値の出現頻度のリストで、与えられた値の少なくとも1つのインスタンスを含む行の断片
elem_count_histogram	列の値でNULLではない要素値の個別数のヒストグラム

```
=>  SELECT * FROM pg_stats WHERE tablename = 'pgbench_tellers'
    AND attname = 'tid';
    -[ RECORD 1 ]----------+------------------------------------------------
    schemaname             | public
    tablename              | pgbench_tellers
    attname                | tid
    inherited              | f
    null_frac              | 0
    avg_width              | 4
    n_distinct             | -1
    most_common_vals       |
    most_common_freqs      |
    histogram_bounds       | {1,2,3,4,5,6,7,8,9,10}
    correlation            | 0.345455
    most_common_elems      |
    most_common_elem_freqs |
    elem_count_histogram   |
```