Использование статистики в PostgreSQL для оптимизации производительности

Алексей Ермаков
alexey.ermakov@postgresql-consulting.com

PostgreSQL-Consulting.com

О чем сегодня будем говорить

- Как планировщик оценивает число строк в таблице
- Как определяется селективность условий в запросах
- Какая собирается статистика
- Полезные приемы
- Грабли и способы их обхода
- Мониторинг производительности и диагностика проблем

Как выполняется запрос?

- Connection
- Parser
- Rewrite system
- Planner/Optimizer
- Executor

Как выполняется запрос?

- Генерируется множество планов выполнения
- Для каждой элементарной операции оценивается число строк и время выполнения
- Больше таблиц в запросе $\Rightarrow$ дольше время планирования

Preparing test case

```
pgday=# create table posts (
id serial primary key,
category_id integer,
content text,
rating integer not null);

pgday=# create index concurrently posts_category_id on posts using btree(category_id);
```

Preparing test case

```
pgday=# \d+ posts
                                        Table "public.posts"
   Column    |  Type   |           Modifiers            | Storage  | Stats target | Description
-------------+---------+--------------------------------+----------+--------------+-------------
 id          | integer | not null default nextval(...) | plain    |              |
 category_id | integer |                                | plain    |              |
 content     | text    |                                | extended |              |
 rating      | integer | not null                       | plain    |              |
Indexes:
    "posts_pkey" PRIMARY KEY, btree (id)
    "posts_category_id" btree (category_id)
Has OIDs: no
```

Preparing test case

```
pgday=# insert into posts (category_id, content, rating)
select floor(100*random()), -- равномерное распределение на [0..99]
'hello world ' || id,
normal_rand(1, 50, 10) -- нормальное распределение с mean = 50, stddev = 10
from generate_series(1, 10000) gs(id);
INSERT 0 10000
```

Preparing test case

```
pgday=# select * from posts order by id limit 5;
 id | category_id |    content    | rating
----+-------------+---------------+--------
  1 |          28 | hello world 1 |     40
  2 |          83 | hello world 2 |     39
  3 |          16 | hello world 3 |     52
  4 |          60 | hello world 4 |     53
  5 |          26 | hello world 5 |     49
(5 rows)
```

cardinality

```
pgday=# explain select count(*) from posts;
                            QUERY PLAN
------------------------------------------------------------------
 Aggregate  (cost=198.00..198.01 rows=1 width=0)
   ->  Seq Scan on posts  (cost=0.00..173.00 rows=10000 width=0)
(2 rows)
```

cardinality

```
pgday=# select reltuples, relpages from pg_class where relname = 'posts';
 reltuples | relpages
-----------+----------
     10000 |       74
```

$$\text{rows} \approx \frac{reltuples}{relpages} * \text{current\_relpages}$$

```
select n_tup_ins, n_live_tup, last_autoanalyze, autoanalyze_count
from pg_stat_user_tables where relname = 'posts';
 n_tup_ins | n_live_tup |       last_autoanalyze        | autoanalyze_count
-----------+------------+-------------------------------+-------------------
     10000 |      10000 | 2015-07-04 02:22:04.806939+07 |                 1
```

autoanalyze

- inserted + updated + deleted > threshold ⇒ run autoanalyze
- threshold = autovacuum_analyze_threshold + reltuples*autovacuum_analyze_scale_factor
- autovacuum_analyze_scale_factor (default = 0.1)
- autovacuum_analyze_threshold (default = 50)
- default_statistics_target (default = 100 since 8.4)
- rows in sample = 300 * stats_target

pg_stats

```
pgday=# \d+ pg_stats
         Column          |   Type    | Modifiers | Storage  | Description
-------------------------+-----------+-----------+----------+-------------
 tablename               | name      |           | plain    |
 attname                 | name      |           | plain    |
 null_frac               | real      |           | plain    |
 avg_width               | integer   |           | plain    |
 n_distinct              | real      |           | plain    |
 most_common_vals        | anyarray  |           | extended |
 most_common_freqs       | real[]    |           | extended |
 histogram_bounds        | anyarray  |           | extended |
 correlation             | real      |           | plain    |
 most_common_elems       | anyarray  |           | extended |
 most_common_elem_freqs  | real[]    |           | extended |
 elem_count_histogram    | real[]    |           | extended |
```

pg_stats

```
pgday=# \x
Expanded display is on.
pgday=# select * from pg_stats where tablename = 'posts' and attname = 'id';
schemaname             | public
tablename              | posts
attname                | id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | -1
most_common_vals       |
most_common_freqs      |
histogram_bounds       | {1,100,200,300,400,500,600,700, ... ,9400,9500,9600,9700,9800,9900,10000}
correlation            | 1
```

selectivity estimate

```
pgday=# explain select count(*) from posts where id < 250;
                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=14.29..14.29 rows=1 width=0)
   ->  Index Only Scan using posts_pkey on posts  (cost=0.29..13.66 rows=250 width=0)
         Index Cond: (id < 250)
```

```
histogram_bounds       | {1,100,200,300,400,500,600,700, ... ,9400,9500,9600,9700,9800,9900,10000}
```

$$\text{selectivity} = \frac{2 + \frac{250-200}{300-200}}{100} = 0.025$$

rows $\approx$ selectivity * cardinality = 0.025 * 10000 = 250

pg_stats

```
pgday=# select * from pg_stats where tablename = 'posts' and attname = 'category_id';
schemaname             | public
tablename              | posts
attname                | category_id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | 100
most_common_vals       | {98,22,20,99,32,6,23,92,7,18,65,67,14,26,28,76,77,84,...}
most_common_freqs      | {0.0121,0.012,0.0118,0.0117,0.0116,0.0115,0.0115,0.0115,0.0114,...}
histogram_bounds       |
correlation            | 0.0194019
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

selectivity estimate

```
pgday=# explain select count(*) from posts where category_id = 98;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=83.78..83.79 rows=1 width=0)
   ->  Bitmap Heap Scan on posts  (cost=5.22..83.48 rows=121 width=0)
         Recheck Cond: (category_id = 98)
         ->  Bitmap Index Scan on posts_category_id  (cost=0.00..5.19 rows=121 width=0)
               Index Cond: (category_id = 98)
```

```
most_common_vals       | {98,22,20,99,32,6,23,92,7,18,65,67,14,26,28,76,77,84,...}
most_common_freqs      | {0.0121,0.012,0.0118,0.0117,0.0116,0.0115,0.0115,0.0115,0.0114,...}
```

selectivity = 0.0121

rows ≈ selectivity * cardinality = 0.0121 * 10000 = 121

pg_stats

```
pgday=# alter table posts alter column category_id set statistics 10;
ALTER TABLE

pgday=# analyze posts;
ANALYZE

pgday=# \d+ posts
                                        Table "public.posts"
   Column    |  Type   |           Modifiers            | Storage  | Stats target | Description
-------------+---------+--------------------------------+----------+--------------+-------------
 category_id | integer |                                | plain    | 10           |
...
```

pg_stats

```
pgday=# select * from pg_stats where tablename = 'posts' and attname = 'category_id';
schemaname             | public
tablename              | posts
attname                | category_id
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | 100
most_common_vals       |
most_common_freqs      |
histogram_bounds       | {0,9,20,29,39,50,60,70,80,90,99}
correlation            | 0.0194019
most_common_elems      |
most_common_elem_freqs |
elem_count_histogram   |
```

selectivity estimate

```
pgday=# explain select count(*) from posts where category_id = 98;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=84.48..84.49 rows=1 width=0)
   ->  Bitmap Heap Scan on posts  (cost=5.06..84.23 rows=100 width=0)
         Recheck Cond: (category_id = 98)
         ->  Bitmap Index Scan on posts_category_id  (cost=0.00..5.04 rows=100 width=0)
               Index Cond: (category_id = 98)
```

$$\text{selectivity} = \frac{1 - null\_frac - sumcommon}{n\_distinct - distinctcommon} = \frac{\text{p(row in histogram bounds)}}{\text{number of distinct values in histogram bounds}} = \frac{1-0-0}{100-0} = 0.01$$

rows $\approx$ selectivity * cardinality = 0.01 * 10000 = 100

Statistical independence

```
pgday=# explain analyze select count(*) from posts where category_id = 98 and id < 250;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=14.29..14.30 rows=1 width=0) (actual time=0.132..0.132 rows=1 loops=1)
   ->  Index Scan using posts_pkey on posts  (cost=0.29..14.29 rows=2 width=0)
                                             (actual time=0.081..0.129 rows=3 loops=1)
         Index Cond: (id < 250)
         Filter: (category_id = 98)
```

selectivity = selectivity1 * selectivity2 = 0.025 * 0.01 = 0.00025

rows ≈ selectivity * cardinality = 0.00025 * 10000 = 2.5

pg_stats

```
pgday=# select * from pg_stats where tablename = 'posts' and attname = 'rating';

schemaname             | public
tablename              | posts
attname                | rating
inherited              | f
null_frac              | 0
avg_width              | 4
n_distinct             | 72
most_common_vals       | {50,51,48,54,49,55,56,53,52,47,46,45,57,44,43,42,59,58,41,60,39,40,61,...}
most_common_freqs      | {0.0411,0.0407,0.0399,0.0389,0.0384,0.038,0.0374,0.0369,0.0365,0.0355,...}
histogram_bounds       | {12,23,25,26,27,28,29,29,30,30,31,31,31,32,32,32,33,33,33,33,67,67,67,...}
correlation            | 0.0412645
most_common_elems      |
most_common_elem_freqs |
```

pg_stats: distribution of functional indexes

```
pgday=# create index concurrently posts_expr_idx on posts using btree((rating^2));

select * from pg_stats where tablename = 'posts_expr_idx';--tablename? no, index name
tablename              | posts_expr_idx
attname                | expr
avg_width              | 8
n_distinct             | 72
most_common_vals       | {2500,2601,2304,2916,2401,3025,3136,2809,2704,2209,2116,2025,3249,1936...}
most_common_freqs      | {0.0411041,0.0407041,0.039904,0.0389039,0.0384038,0.0380038,0.0374037...}
histogram_bounds       | {144,529,625,676,729,784,841,841,900,900,961,961,961,1024,1024,1089...}

pgday=# alter index posts_expr_idx alter column expr set statistics 1000;--no documentation!
```

default estimators

```
/src/include/utils/selfuncs.h

/* default selectivity estimate for equalities such as "A = b" */
#define DEFAULT_EQ_SEL 0.005
/* default selectivity estimate for inequalities such as "A < b" */
#define DEFAULT_INEQ_SEL 0.3333333333333333
/* default selectivity estimate for range inequalities "A > b AND A < c" */
#define DEFAULT_RANGE_INEQ_SEL 0.005
/* default selectivity estimate for pattern-match operators such as LIKE */
#define DEFAULT_MATCH_SEL 0.005
```

default estimators

```
pgday=# explain analyze select count(*) from posts where id < (select 100);
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Aggregate  (cost=134.95..134.96 rows=1 width=0) (actual time=0.083..0.083 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Result  (cost=0.00..0.01 rows=1 width=0) (actual time=0.001..0.001 rows=1 loops=1)
   ->  Index Only Scan using posts_pkey on posts  (cost=0.29..126.61 rows=3333 width=0)
                                                  (actual time=0.031..0.069 rows=99 loops=1)
         Index Cond: (id < $0)
         Heap Fetches: 99
```

selectivity = 0.3333333333333333

Оценка распределения в небольшой таблице

```
pgday=# select count(*), count(distinct category_id) as ndistinct from posts;
 count | ndistinct
-------+-----------
 10000 |       100
```

Оценка распределения в небольшой таблице

```
pgday=# select category_id, count(*),
count(*) * 100/(sum(count(*)) over ())::float as count_percent
from posts group by 1 order by 2 desc limit 5;

 category_id | count | count_percent
-------------+-------+---------------
           4 |   124 |          1.24
          97 |   123 |          1.23
          24 |   121 |          1.21
          20 |   118 |          1.18
           0 |   117 |          1.17
```

Для больших таблиц не работает

- Очень медленно
- А если нужно посмотреть несколько распределений?
- pg_stats содержит больше информации и гораздо удобней

Промежуточные выводы

- pg_stats содержит много полезной информации, важно уметь оттуда ее читать
- stats_target можно менять, причем per column
- Некоторые настройки autovacuum/autoanalyze стоит менять, причем можно менять per table
- Предполагается статистическая независимость условий

Зачем это все нужно?

Создание частичных индексов

Не всегда нужно индексировать все значения

```
\dt+ foo
                        Список отношений
 Схема  |     Имя      |   Тип   | Владелец | Размер | Описание
--------+--------------+---------+----------+--------+----------
 public |     foo      | таблица | postgres | 73 GB  |

select reltuples::int from pg_class where relname = 'foo';
 reltuples
-----------
  73251096
```

Создание частичных индексов

```
select * from pg_stats where tablename = 'foo' and attname = 'bar_id';
null_frac                | 0.00739433
avg_width                | 4
n_distinct               | 50
most_common_vals         | {20,31,73,26,3,235,38,37,183,167,110,27,147,165,...}
most_common_freqs        | {0.555908,0.117836,0.10815,0.100445,0.0505153,0.017418,0.0101523,...}

SELECT f.*
   FROM foo f
   WHERE f.bar_id = 183
ORDER BY f.id DESC OFFSET 0 LIMIT 20
```

Напрашивается индекс на (bar_id, id), но...

PGDAY'15 RUSSIA

Создание частичных индексов

```
select * from pg_stats where tablename = 'foo' and attname = 'bar_id';
null_frac              | 0.00739433
avg_width              | 4
n_distinct             | 50
most_common_vals       | {20,31,73,26,3,235,38,37,183,167,110,27,147,165,...}
most_common_freqs      | {0.555908,0.117836,0.10815,0.100445,0.0505153,0.017418,0.0101523,...}
```

88% записей приходится на 4 значения

Создание частичных индексов

Поэтому достаточно частичного индекса, который раз в 10 меньше полного:

```
create index concurrently foo_bar_id_id_partial on foo
using btree(bar_id, id) where bar_id not in (20,26,31,73);

\di+ foo_bar_id_id_partial

 Схема  |              Имя               |  Тип   | Владелец |   Таблица    | Размер | Описание
--------+--------------------------------+--------+----------+--------------+--------+----------
 public | foo_bar_id_id_partial          | индекс | postgres |      foo     | 758 MB |
```

Для запросов с bar_id из списка будет эффективно использоваться индекс по id

Создание оптимального индекса

```
select ... from table where a = ? and b = ?
```

Какой индекс создать?

- a
- a, b
- (a, b)
- (b, a)
- a where b = smth
- a может вообще не нужен индекс?

Создание оптимального индекса

- Смотрим типичные параметры в запросе в логах и соответствующние планы их выполнения
- Смотрим распределения
- Выбираем условия с минимальным selectivity
- Стараемся на них составить индекс и поставить их в начало
- На условия с большим selectivity скорей всего индекс не нужен

PGDAY'15 RUSSIA

Грабли

www.saveelephant.org

Грабли

Отсутствие cross columns статистики (многомерных распределений)

```
pgday=# explain analyze select count(*) from posts where content < 'hello world 250';
------------------------------------------------------------------------------------
 Aggregate  (cost=203.24..203.25 rows=1 width=0) (actual time=3.317..3.317 rows=1 loops=1)
   ->  Seq Scan on posts  (cost=0.00..199.00 rows=1697 width=0)
                          (actual time=0.016..3.111 rows=1669 loops=1)

pgday=# explain analyze select count(*) from posts where content < 'hello world 250' and id < 250;
-------------------------------------------------------------------------------------------
 Aggregate  (cost=14.39..14.40 rows=1 width=0) (actual time=0.183..0.184 rows=1 loops=1)
   ->  Index Scan using posts_pkey on posts  (cost=0.29..14.29 rows=42 width=0)
                                             (actual time=0.034..0.155 rows=168 loops=1)
```

1697*0.025 = 42.425

Грабли

Отсутствие cross columns статистики (многомерных распределений)

```
pgday=# explain analyze select count(*) from posts where id < 250 and
        (content < 'hello world 250' or abs(id) < 0);
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Aggregate  (cost=15.81..15.82 rows=1 width=0) (actual time=0.210..0.210 rows=1 loops=1)
   ->  Index Scan using posts_pkey on posts  (cost=0.29..15.54 rows=112 width=0)
                                             (actual time=0.032..0.185 rows=168 loops=1)
         Index Cond: (id < 250)
         Filter: ((content < 'hello world 250'::text) OR (abs(id) < 0))
```

Грабли

Отсутствие статистики по json полям

- в pg_stats вообще нет записей по json полям
- а значит, что и нет null_frac, n_distinct и прочего
- например, если много null в этом поле и есть условие на not null, то план может выбраться неоптимальный
- по jsonb статистика есть

Грабли

Неиспользование статистики у intarray операторов

```
pgday=# create table test as select array[100]::integer[] as f1 from
generate_series(1,10000);
SELECT 10000
pgday=# analyze test;
ANALYZE
pgday=# explain analyze select * from test where f1 && array[100];
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..532.40 rows=10000 width=25)
                   (actual time=0.048..6.207 rows=10000 loops=1)
   Filter: (f1 && '100'::integer[])
```

Грабли

Неиспользование статистики у intarray операторов

```
pgday=# create extension intarray;
CREATE EXTENSION
pgday=# explain analyze select * from test where f1 && array[100];
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..199.00 rows=10 width=25)
                   (actual time=0.051..6.493 rows=10000 loops=1)
   Filter: (f1 && '100'::integer[])
```

Грабли

Неиспользование статистики у intarray операторов

```
pgday=# explain analyze select * from test where f1 OPERATOR(pg_catalog.&&) array[100];
                              QUERY PLAN
-----------------------------------------------------------------------
 Seq Scan on test  (cost=0.00..199.00 rows=10000 width=25)
                   (actual time=0.021..5.686 rows=10000 loops=1)
   Filter: (f1 OPERATOR(pg_catalog.&&) '100'::integer[])
```

Грабли

Недостаточный statistics_target

- Например, поиск несуществующего (редкого) значения в очень большой таблице по полю с небольшим n_distinct
- selectivity = $\frac{1-null\_frac-sumcommon}{n\_distinct-distinctcommon} = \frac{\text{p(row in histogram bounds)}}{\text{number of distinct values in histogram bounds}}$
- Чем больше statistics_target $\Rightarrow$ тем больше sumcommon (сумма most common freqs)
- Оценка может отличаться на несколько порядков
- Выкручиваем stats_target до 1000-10000
- Analyze может быть медленным

Мониторинг

Statistics Collector Views

- pg_stat_user_tables
- pg_stat_user_indexes
- pg_stat_user_functions
- pg_stat_database
- pg_stat_activity
- pg_statio_user_tables
- pg_statio_user_indexes

Мониторинг

- pg_stat_reset() - сброс всей “мониторинговой“ статистики в текущей базе
- track_io_timing
- track_functions
- stats_temp_directory - RAM disk
- track_activity_query_size

pg_stat_statements

```
pgday=# select * from (select unnest(proargnames) from pg_proc where proname = 'pg_stat_statements')
       unnest
---------------------
 userid
 dbid
 query
 calls
 total_time
 rows
 ...
 blk_read_time
 blk_write_time
```

pg_stat_statements

```
total time: 50:49:48 (IO: 0.64%)
total queries: 301,163,398 (unique: 9,206)
report for all databases, version 0.9.3 @ PostgreSQL 9.2.13
tracking top 10000 queries, logging 100ms+ queries

================================================================================================================
pos:1  total time: 14:39:43 (28.8%, CPU: 28.8%, IO: 36.8%) calls: 4,895,890 (1.63%)
avg_time: 10.78ms (IO: 0.8%)
user: bravo  db: echo  rows: 4,895,890  query:
SELECT sum(o.golf) as golf, sum(o.romeo) as romeo, sum(o.whiskey) as whiskey,
               sum(o.hotel) as hotel FROM oscar AS o LEFT JOIN uniform AS u ON u.kilo = o.kilo JOIN
```

pg_stat_statements

- pg_stat_statements.max
- pg_stat_statements.track
- pg_stat_statements.track_utility

Заключение

- В PostgreSQL собирается 2 вида статистики: по распределениям данных (собирается autoanalyze) и различные системные счетчики (собирается stats collector)
- С их помощью можно выявлять проблемные места и устранять них
- Планировщик иногда может ошибаться
- pg_stat_statements стоит использовать

Полезные ссылки

- PostgreSQL Manual 61.1. Row Estimation Examples
- PostgreSQL Manual 27.2. The Statistics Collector
- depesz: Explaining the unexplainable
- `https://github.com/PostgreSQL-Consulting/pg-utils`
- `http://blog.postgresql-consulting.com/`

Вопросы?

alexey.ermakov@postgresql-consulting.com