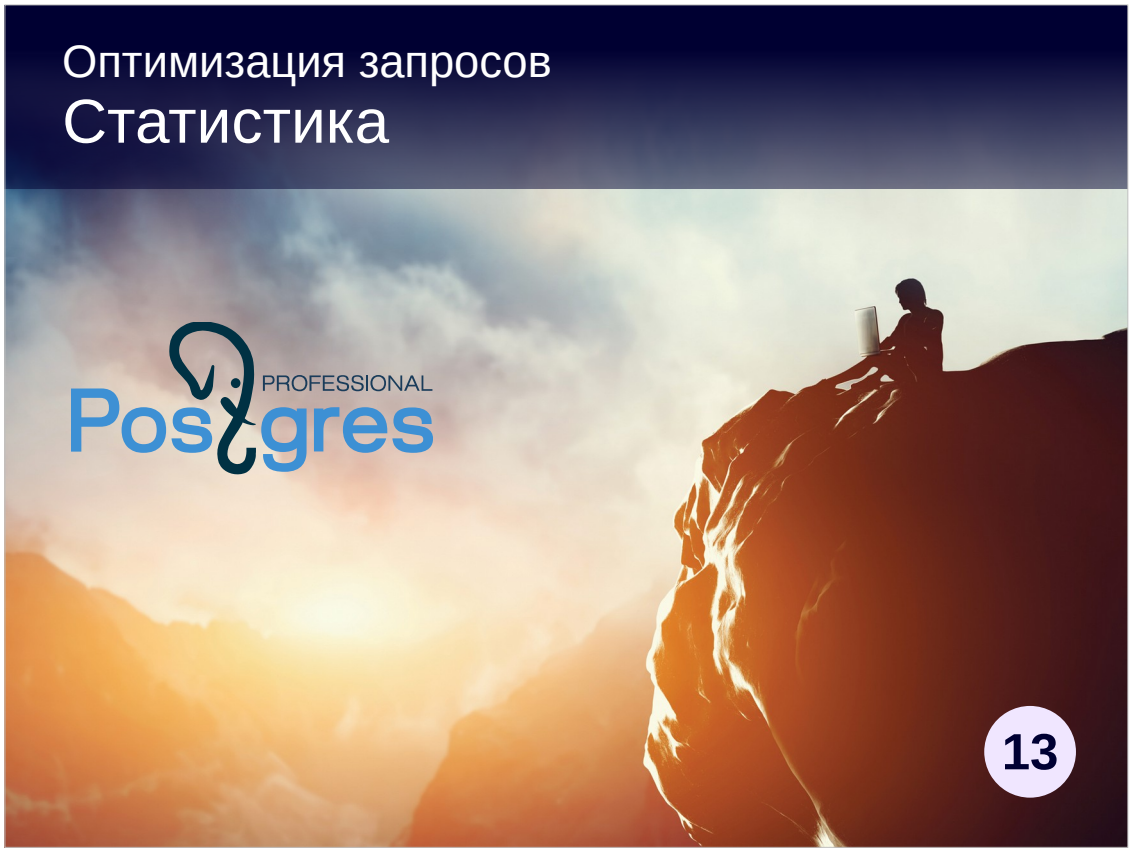


Авторы: Егор Рогов, Павел Лузанов, Павел Толмачев, Илья Баштанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Базовая статистика

Наиболее частые значения и гистограммы

Частные и общие планы выполнения

Расширенная и многовариантная статистика

Статистика по выражениям

Использование статистики для оценки кардинальности и селективности

Базовая статистика

Размер таблицы

- строки (pg_class.reltuples) и страницы (pg_class.relpages)

Собирается

- операциями DDL
- очисткой
- анализом

Настройка

- *default_statistics_target* = 100

Базовая статистика собирается на уровне всей таблицы и на уровне отдельных столбцов.

К статистике таблицы относится информация о размере объекта (reltuples, relpages в таблице pg_class). Поскольку такая статистика крайне важна, она собирается не только при анализе (ANALYZE), но и заполняется некоторыми DDL-операциями (CREATE INDEX, CREATE TABLE AS SELECT), а затем уточняется при очистке.

Кроме того, планировщик масштабирует количество строк в соответствии с отклонением реального размера файла данных от значения relpages.

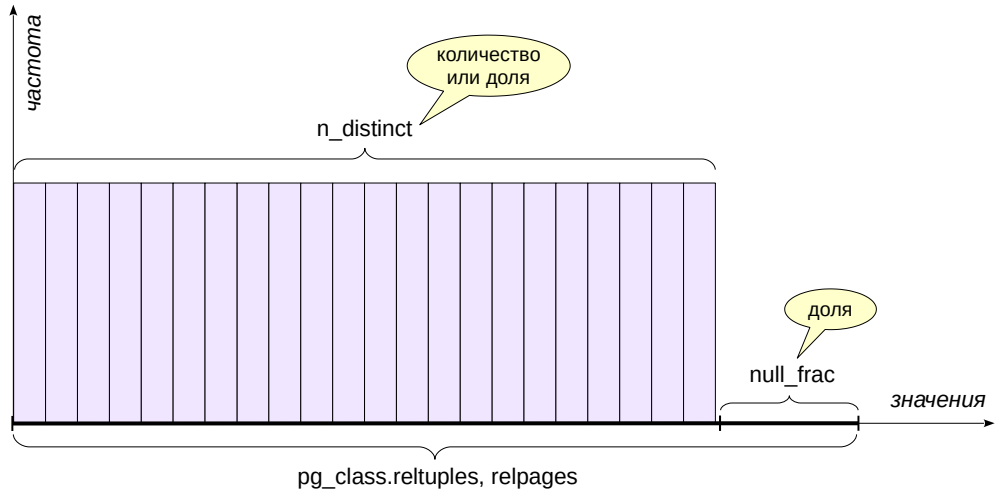
При анализе просматривается случайная выборка строк. Установлено, что размер выборки, обеспечивающий хорошую точность оценок, практически не зависит от размера таблицы. В качестве размера выборки используется ориентир статистики, заданный параметром *default_statistics_target*, умноженный на 300.

При этом следует понимать, что статистика не должна быть абсолютно точной, чтобы планировщик мог выбрать приемлемый план; часто достаточно попадания в порядок.

https://postgrespro.ru/docs/postgresql/13/row-estimation-examples

Базовая статистика

pg_statistic (pg_stats)



Вся остальная статистика собирается отдельно для каждого столбца при анализе таблицы. Обычно этим занимается автоанализ (его настройка рассматривается в курсе DBA2).

Статистика на уровне столбцов хранится в таблице pg_statistic. Но смотреть проще в представление pg_stats, которое показывает информацию в более удобном виде.

Поле null_frac содержит долю строк с неопределенными значениями в столбце (от 0 до 1).

Поле n_distinct хранит число уникальных значений в столбце. Если значение n_distinct отрицательно, то модуль этого числа показывает *долю* уникальных значений. Например, -1 означает, что все значения уникальны (типичный случай для первичного ключа).

https://postgrespro.ru/docs/postgresql/13/planner-stats#id-1.5.13.5.3

Число строк

Начнем с оценки кардинальности в простом случае запроса без предикатов.

```
=> EXPLAIN SELECT * FROM flights;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=63)
(1 row)
```

Точное значение:

```
=> SELECT count(*) FROM flights;
```

```
 count
--------
 214867
(1 row)
```

Оптимизатор получает значение из pg_class:

```
=> SELECT reltuples, relpages FROM pg_class WHERE relname = 'flights';
```

```
 reltuples | relpages
-----------+----------
    214867 |     2624
(1 row)
```

Значение параметра, управляющего ориентиром статистики, по умолчанию равно 100:

```
=> SHOW default_statistics_target;
```

```
 default_statistics_target
---------------------------
 100
(1 row)
```

Поскольку при анализе таблицы учитывается 300*default_statistics_target строк, то оценки для относительно крупных таблиц могут не быть абсолютно точными.

Доля неопределенных значений

Часть рейсов еще не отправились, поэтому время вылета для них не определено:

```
=> EXPLAIN SELECT * FROM flights WHERE actual_departure IS NULL;
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..4772.67 rows=16000 width=63)
   Filter: (actual_departure IS NULL)
(2 rows)
```

Точное значение:

```
=> SELECT count(*) FROM flights WHERE actual_departure IS NULL;
```

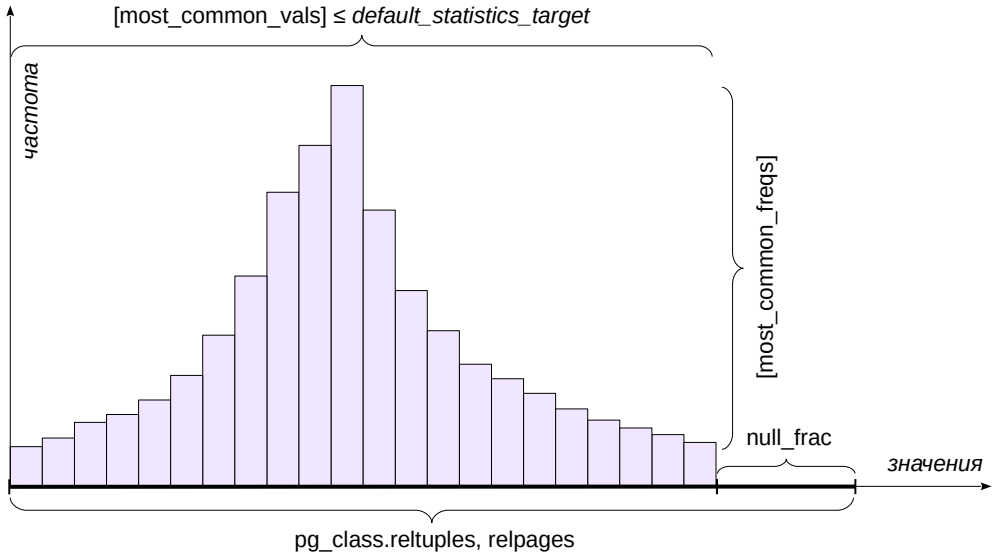
```
 count
-------
 16348
(1 row)
```

Оценка оптимизатора получена как общее число строк, умноженное на долю NULL-значений:

```
=> SELECT 214867 * null_frac FROM pg_stats
WHERE tablename = 'flights' AND attname = 'actual_departure';
```

```
      ?column?
--------------------
 16000.429568059742
(1 row)
```

Наиболее частые значения



Если бы все данные были всегда распределены равномерно, то есть все значения встречались бы с одинаковой частотой, этой информации было бы почти достаточно (нужен еще минимум и максимум).

Но в реальности неравномерные распределения встречаются очень часто. Поэтому собирается еще и следующая информация:

- массив наиболее частых значений — поле most_common_vals;
- массив частот этих значений — поле most_common_freqs.

Частота из этих массивов непосредственно служит оценкой селективности для поиска конкретного значения.

Все это прекрасно работает, пока число различных значений не очень велико. Максимальный размер каждого из массивов ограничен параметром *default_statistics_target*. Это значение можно переопределять на уровне отдельного столбца; в этом случае размер анализируемой выборки будет определяться по максимальному значению для таблицы.

Тонкий момент представляют «большие» значения. Чтобы не увеличивать размер pg_statistic и не нагружать планировщик бесполезной работой, значения, превышающие 1 Кбайт, исключаются из статистики и анализа. В самом деле, если в поле хранятся такие большие значения, скорее всего, они уникальны и не имеют шансов попасть в most_common_vals.

Наиболее частые значения

Для эксперимента ограничим размер списка наиболее частых значений (который по умолчанию определяется параметром default_statistics_target) на уровне столбца:

```
=> ALTER TABLE flights ALTER COLUMN arrival_airport
SET STATISTICS 10;
```

```
ALTER TABLE
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

Если значение попало в список наиболее частых, селективность можно узнать непосредственно из статистики. Пример (Шереметьево):

```
=> EXPLAIN SELECT * FROM flights WHERE arrival_airport = 'SVO';
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=19696 width=63)
   Filter: (arrival_airport = 'SVO'::bpchar)
(2 rows)
```

Точное значение:

```
=> SELECT count(*) FROM flights WHERE arrival_airport = 'SVO';
```

```
 count
-------
 19348
(1 row)
```

Вот как выглядит список наиболее частых значений и частота их встречаемости:

```
=> SELECT most_common_vals, most_common_freqs
FROM pg_stats
WHERE tablename = 'flights' AND attname = 'arrival_airport' \gx
```

```
-[ RECORD 1 ]-----+-----------------------------------------------------------------------------------------------------
most_common_vals  | {DME,SVO,LED,VKO,OVB,KJA,SVX,BZK,PEE,AER}
most_common_freqs | {0.10073333,0.09166667,0.0564,0.0525,0.030033333,0.0215,0.020733334,0.019266667,0.0189,0.0179}
```

Кардинальность вычисляется как число строк, умноженное на частоту значения:

```
=> SELECT 214867 * s.most_common_freqs[array_position((s.most_common_vals::text::text[]),'SVO')]
FROM pg_stats s
WHERE s.tablename = 'flights' AND s.attname = 'arrival_airport';
```

```
     ?column?
------------------
 19696.14209356904
(1 row)
```

Список наиболее частых значений может использоваться и для оценки селективности неравенств. Для этого в most_common_vals надо найти все значения, удовлетворяющие неравенству, и просуммировать частоты соответствующих элементов из most_common_freqs.

Число уникальных значений

Если же указанного значения нет в списке наиболее частых, то оно вычисляется исходя из предположения, что все данные (кроме наиболее частых) распределены равномерно.

Например, в списке частых значений нет Владивостока.

```
=> EXPLAIN SELECT * FROM flights WHERE arrival_airport = 'VVO';
```

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=1304 width=63)
   Filter: (arrival_airport = 'VVO'::bpchar)
(2 rows)
```

Точное значение:

```
=> SELECT count(*) FROM flights WHERE arrival_airport = 'VVO';
```

```
 count
-------
  1188
(1 row)
```

Для получения оценки вычислим сумму частот наиболее частых значений:

```
=> SELECT sum(f) FROM pg_stats s, unnest(s.most_common_freqs) f
  WHERE s.tablename = 'flights' AND s.attname = 'arrival_airport';
```

```
    sum
------------
 0.42963332
(1 row)
```

На менее частые значения приходятся оставшиеся строки. Поскольку мы исходим из предположения о равномерности распределения менее частых значений, селективность будет равна 1/nd, где nd — число уникальных значений:

```
=> SELECT n_distinct
FROM pg_stats s
WHERE s.tablename = 'flights' AND s.attname = 'arrival_airport';
```

```
 n_distinct
------------
        104
(1 row)
```

Учитывая, что из этих значений 10 входят в список наиболее частых, и нет неопределенных значений, получаем следующую оценку:

```
=> SELECT 214867 * (1 - 0.42963332) / (104 - 10);
```

```
       ?column?
-----------------------
 1303.7550790591489362
(1 row)
```

Частные и общие планы

Неравномерные распределения значений приводят к тому, что запросы, отличающиеся константами или значениями параметров, могут иметь разные планы выполнения. Например, подготовим следующий запрос:

```
=> PREPARE f(text) AS SELECT * FROM flights WHERE status = $1;
```

```
PREPARE
```

Поиск отмененных рейсов будет использовать индекс, поскольку статистика говорит о том, что таких рейсов мало:

```
=> CREATE INDEX ON flights(status);
```

```
CREATE INDEX
```

```
=> EXPLAIN EXECUTE f('Cancelled');
```

```
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Index Scan using flights_status_idx on flights  (cost=0.29..404.38 rows=387 width=63)
   Index Cond: ((status)::text = 'Cancelled'::text)
(2 rows)
```

А поиск прибывших рейсов — нет, поскольку их много:

```
=> EXPLAIN EXECUTE f('Arrived');
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=198322 width=63)
   Filter: ((status)::text = 'Arrived'::text)
(2 rows)
```

Такие планы называются частными, поскольку они построены с учетом конкретных значений параметров.

Пять первых планирований всегда используют частные планы. Затем может оказаться, что стоимость общего плана (построенного без учета конкретного значения, в предположении равномерного распределения) не превышает среднюю стоимость уже построенных частных планов. Тогда планировщик запомнит общий план и будет использовать его, не выполняя планирование каждый раз.

Построим план еще несколько раз:

```
=> EXPLAIN EXECUTE f('Arrived');
```

```
                           QUERY PLAN
----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=198322 width=63)
   Filter: ((status)::text = 'Arrived'::text)
(2 rows)
```

```
=> EXPLAIN EXECUTE f('Arrived');
```

```
                           QUERY PLAN
----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=198322 width=63)
   Filter: ((status)::text = 'Arrived'::text)
(2 rows)
```

```
=> EXPLAIN EXECUTE f('Arrived');
```

```
                           QUERY PLAN
----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=198322 width=63)
   Filter: ((status)::text = 'Arrived'::text)
(2 rows)
```

В следующий раз планировщик переключится на общий план. Вместо конкретного значения в плане будет указан номер параметра:

```
=> EXPLAIN EXECUTE f('Arrived');
```

```
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=401.83..3473.47 rows=35811 width=63)
   Recheck Cond: ((status)::text = $1)
   ->  Bitmap Index Scan on flights_status_idx  (cost=0.00..392.88 rows=35811 width=0)
         Index Cond: ((status)::text = $1)
(4 rows)
```

При неравномерном распределении это может вызывать проблемы. Параметр plan_cache_mode позволяет отключить использование частных планов (или наоборот, с самого начала использовать общий план):

```
=> SHOW plan_cache_mode;
```

```
 plan_cache_mode
-----------------
 auto
(1 row)
```

```
=> SET plan_cache_mode = 'force_custom_plan';
```

```
SET
```

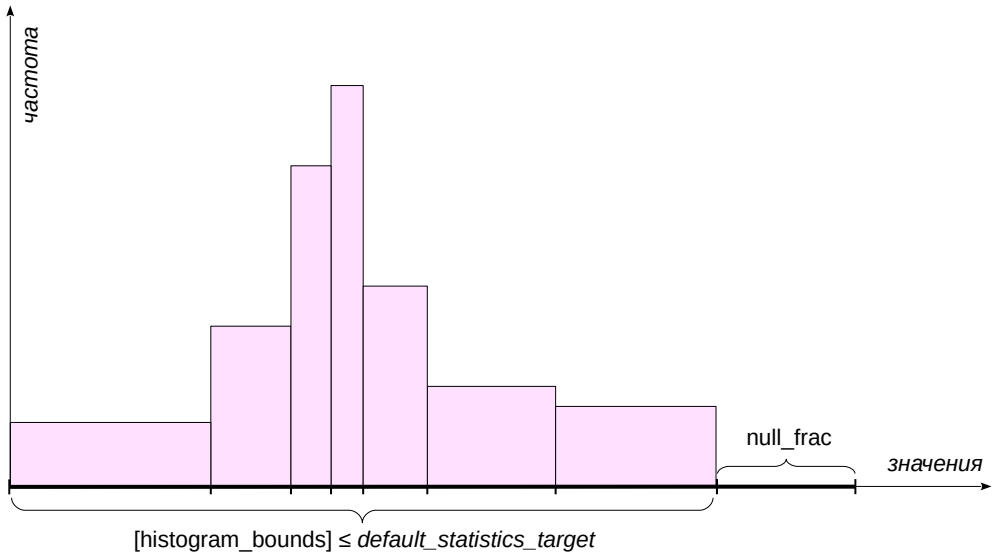
```
=> EXPLAIN EXECUTE f('Arrived');
```

```
                           QUERY PLAN
----------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=198322 width=63)
   Filter: ((status)::text = 'Arrived'::text)
(2 rows)
```

```
=> RESET plan_cache_mode;
```

```
RESET
```

Гистограмма



Если число различных значений слишком велико, чтобы записать их в массив, на помощь приходит гистограмма. Гистограмма состоит из нескольких корзин, в которые помещаются значения. Количество корзин ограничено тем же параметром *default_statistics_target*.

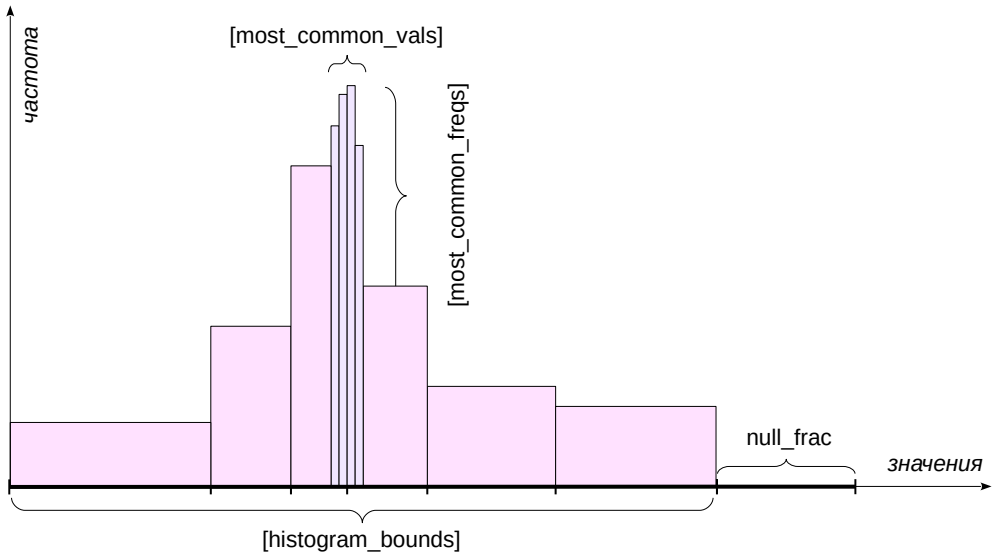
Ширина корзин выбирается так, чтобы в каждую попало примерно одинаковое число значений (на рисунке это выражается в одинаковой площади прямоугольников).

При таком построении достаточно хранить только массив крайних значений каждой корзины — поле histogram_bounds. Частота одной корзины равна 1/(число корзин).

Оценить селективность условия *поле* < *значение* можно как N/(общее число корзин), где N — число корзин, лежащих слева от *значения*. Оценку можно улучшить, добавив часть корзины, в которую попадает само *значение*.

Если же надо оценить селективность условия *поле* = *значение*, то гистограмма в этом не может помочь, и приходится довольствоваться предположением о равномерном распределении и брать в качестве оценки 1/n_distinct.

Комбинация методов



Но обычно два подхода объединяются: строится список наиболее частых значений, а все остальные значения покрываются гистограммой.

При этом гистограмма строится так, что в ней не учитываются значения, попавшие в список. Это позволяет улучшить оценки.

Гистограмма

При условиях «больше» и «меньше» для оценки будет использоваться список наиболее частых значений, или гистограмма, или оба способа вместе. Гистограмма строится так, чтобы не включать наиболее частые значения и NULL:

```
=> SELECT histogram_bounds
FROM pg_stats s
WHERE s.tablename = 'flights' AND s.attname = 'arrival_airport';

               histogram_bounds
-----------------------------------------------
 {AAQ,CSY,HMA,KHV,MCX,NNM,OVS,ROV,TJM,ULY,YKS}
(1 row)
```

Число корзин гистограммы определяется параметром default_statistics_target, а границы выбираются так, чтобы в каждой корзине находилось примерно одинаковое количество значений.

Рассмотрим пример:

```
=> EXPLAIN SELECT * FROM flights WHERE arrival_airport <= 'HMA';

                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5309.84 rows=54141 width=63)
   Filter: (arrival_airport <= 'HMA'::bpchar)
(2 rows)
```

Точное значение:

```
=> SELECT count(*) FROM flights WHERE arrival_airport <= 'HMA';

 count
-------
 54311
(1 row)
```

Как получена оценка?

Учтем частоту наиболее частых значений, попадающих в указанный интервал:

```
=> SELECT sum( s.most_common_freqs[array_position((s.most_common_vals::text::text[]),v)] )
FROM pg_stats s, unnest(s.most_common_vals::text::text[]) v
WHERE s.tablename = 'flights' AND s.attname = 'arrival_airport' AND v <= 'HMA';

  sum
--------
 0.1379
(1 row)
```

Указанный интервал занимает ровно 2 корзины гистограммы из 10, а неопределенных значений в данном столбце нет, получаем следующую оценку:

```
=> SELECT 214867 * (1 - 0.42963332) * (2.0 / 10.0) + 214867 * 0.1379;

              ?column?
-------------------------------------
 54140.754786312000000000000000000
(1 row)
```

В общем случае учитываются и не полностью занятые корзины (с помощью линейной аппроксимации).

Дополнительные поля

Упорядоченность (использовать ли битовую карту?)

pg_stats.correlation
(1 — по возрастанию, 0 — хаотично, –1 — по убыванию)

Видимость (использовать ли сканирование только индекса?)

pg_class.relallvisible

Средний размер значения в байтах (оценка памяти)

pg_stats.avg_width

Информация об элементах массивов, tsvector и т. п.

pg_stats.most_common_elems
pg_stats.most_common_elem_freqs
pg_stats.elem_count_histogram

Есть еще несколько значений статистики.

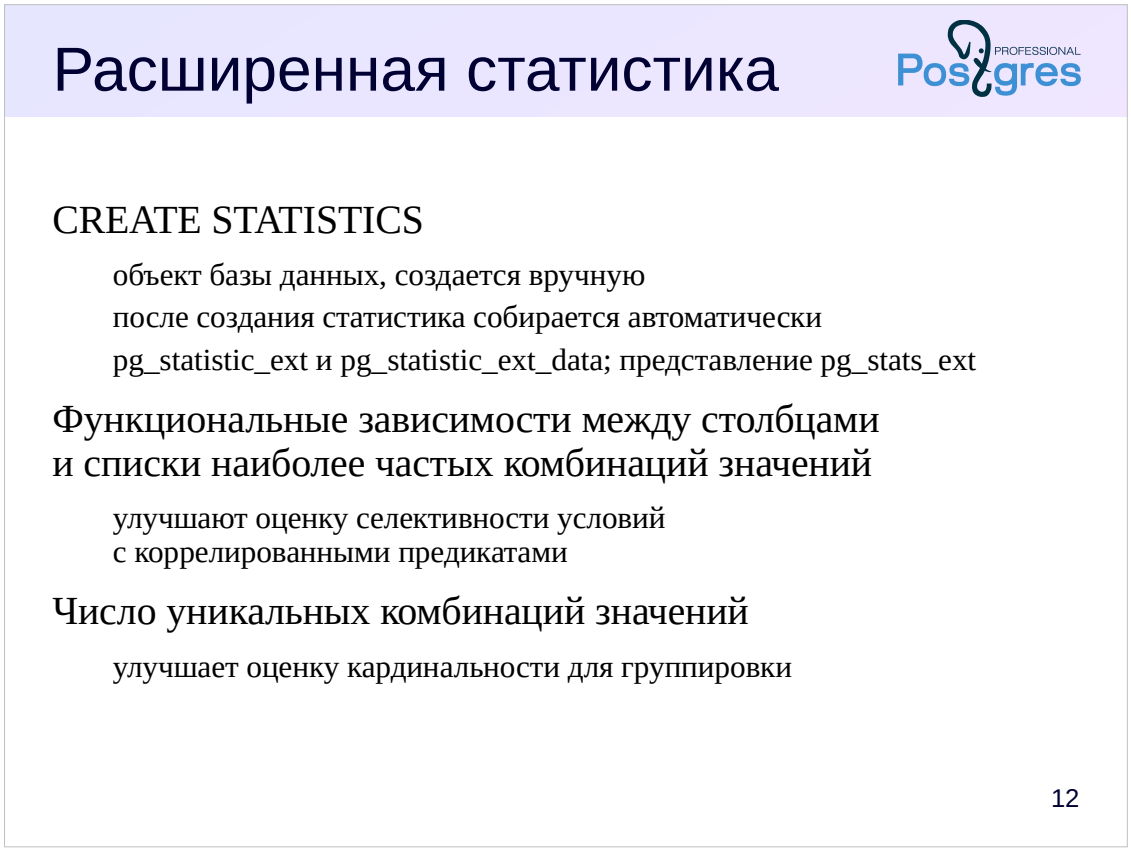
В поле correlation записывается показатель упорядоченности значений на диске. Если значения хранятся строго по возрастанию, показатель будет близок к единице; если по убыванию — к минус единице. Чем более хаотично расположены данные на диске, тем ближе значение показателя к нулю. Именно это поле использует оптимизатор, когда выбирает между сканированием битовой карты и обычным индексным сканированием.

Поле pg_class.relallvisible хранит количество страниц таблицы, которые содержат только актуальные версии строк (эта информация обновляется вместе с картой видимости). Если количество недостаточно велико, планировщик может отказаться от сканирования только индекса в пользу сканирования по битовой карте.

В поле avg_width сохраняется средний размер значений в данном столбце в байтах для расчета необходимого для операции объема памяти.

В полях most_common_elems, most_common_elem_freqs и elem_count_histogram для таких составных типов, как массивы или tsvector, хранится распределение не самих значений, а их элементов. Это позволяет более точно планировать запросы с участием полей *не в первой нормальной форме.*

https://postgrespro.ru/docs/postgresql/13/view-pg-stats



Начиная с PostgreSQL 10, можно создавать специальный объект для *расширенной статистики* командой CREATE STATISTICS. После того, как объект создан, соответствующая статистика будет собираться автоматически.

Существует три вида *многовариантной статистики* (то есть статистики по нескольким столбцам таблицы), которые можно указать при создании объекта расширенной статистики.

Функциональные зависимости между столбцами. Такая статистика показывает, насколько данные в одном столбце определяются значением другого столбца. Она помогает улучшить оценку в случае коррелированных предикатов.

Число уникальных комбинаций значений в столбцах. Такая информация позволяет улучшить оценку кардинальности группировки по нескольким столбцам.

Список наиболее частых комбинаций значений. Статистика помогает улучшить оценку условий, в которых проверяются значения нескольких столбцов.

При создании расширенной статистики можно указать любую комбинацию статистик и столбцов.

Собранная информация хранится в таблицах pg_statistic_ext и pg_statistic_ext_data; доступная пользователю статистика отображается в представлении pg_stats_ext.

https://postgrespro.ru/docs/postgresql/13/planner-stats#PLANNER-STATS-EXTENDED

Функциональные зависимости

Рассмотрим запрос с двумя условиями:

```
=> SELECT count(*)
FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
```

```
 count
-------
   396
(1 row)
```

Оценка оказывается сильно заниженной:

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
```

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=11.92..1212.34 rows=24 width=63)
   Recheck Cond: (flight_no = 'PG0007'::bpchar)
   Filter: (departure_airport = 'VKO'::bpchar)
   ->  Bitmap Index Scan on flights_flight_no_scheduled_departure_key  (cost=0.00..11.91 rows=466 width=0)
         Index Cond: (flight_no = 'PG0007'::bpchar)
(5 rows)
```

Причина в том, что планировщик полагается на то, что предикаты не коррелированы, и считает общую селективность как произведение селективностей условий, объединенных логическим «и». Это хорошо видно в приведенном плане: оценка в узле Bitmap Index Scan (условие на flight_no) одна, а после фильтрации в узле Bitmap Heap Scan (условие на departure_airport) — другая.

Однако мы понимаем, что номер рейса однозначно определяет аэропорт отправления: фактически, второе условие избыточно (конечно, считая, что аэропорт указан правильно).

Начиная с версии PostgreSQL 10, это можно объяснить и планировщику с помощью статистики по функциональной зависимости:

```
=> CREATE STATISTICS flights_dep(dependencies)
ON flight_no, departure_airport FROM flights;
```

```
CREATE STATISTICS
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

Собранная статистика хранится в следующем виде:

```
=> SELECT dependencies
FROM pg_stats_ext WHERE statistics_name = 'flights_dep';
```

```
              dependencies
-----------------------------------------
 {"2 => 5": 1.000000, "5 => 2": 0.011067}
(1 row)
```

Сначала идут порядковые номера атрибутов, а после двоеточия — коэффициент зависимости.

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VKO';
```

```
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=10.56..816.91 rows=276 width=63)
   Recheck Cond: (flight_no = 'PG0007'::bpchar)
   Filter: (departure_airport = 'VKO'::bpchar)
   ->  Bitmap Index Scan on flights_flight_no_scheduled_departure_key  (cost=0.00..10.49 rows=276 width=0)
         Index Cond: (flight_no = 'PG0007'::bpchar)
(5 rows)
```

Теперь оценка улучшилась.

Наиболее частые комбинации значений

Не всегда между значениями разных столбцов есть явная функциональная зависимость. Выполним такой запрос:

```
=> EXPLAIN (analyze, timing off, summary off) SELECT *
FROM flights
WHERE departure_airport = 'LED' AND aircraft_code = '321';

                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..5589.59 rows=697 width=63) (actual rows=5148 loops=1)
   Workers Planned: 1
   Workers Launched: 1
   ->  Parallel Seq Scan on flights  (cost=0.00..4519.89 rows=410 width=63) (actual rows=2574 loops=2)
         Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
         Rows Removed by Filter: 104860
(6 rows)
```

Планировщик ошибается в несколько раз. Учет функциональной зависимости недостаточно исправит ситуацию:

```
=> CREATE STATISTICS flights_dep2(dependencies)
ON departure_airport, aircraft_code FROM flights;

CREATE STATISTICS

=> ANALYZE flights;

ANALYZE

=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'LED' AND aircraft_code = '321';

                                          QUERY PLAN
-----------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..5693.69 rows=1738 width=63)
   Workers Planned: 1
   ->  Parallel Seq Scan on flights  (cost=0.00..4519.89 rows=1022 width=63)
         Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
(4 rows)
```

Начиная с версии PostgreSQL 12 можно строить расширенную статистику по частым комбинациям значений нескольких столбцов и использовать ее в запросах не только равенства, но и неравенства:

```
=> DROP STATISTICS flights_dep2;

DROP STATISTICS

=> CREATE STATISTICS flights_mcv(mcv)
ON departure_airport, aircraft_code FROM flights;

CREATE STATISTICS

=> ANALYZE flights;

ANALYZE
```

Теперь оценка улучшилась:

```
=> EXPLAIN SELECT * FROM flights
WHERE departure_airport = 'LED' AND aircraft_code = '321';

                                       QUERY PLAN
-----------------------------------------------------------------------------------------
 Seq Scan on flights  (cost=0.00..5847.00 rows=4906 width=63)
   Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
(2 rows)
```

Статистику по частым комбинациям можно посмотреть так:

```
=> SELECT m.*
FROM pg_statistic_ext
  JOIN pg_statistic_ext_data ON oid = stxoid,
  pg_mcv_list_items(stxdmcv) m
WHERE stxname = 'flights_mcv'
LIMIT 10;

 index |  values   | nulls |      frequency       |    base_frequency
-------+-----------+-------+----------------------+----------------------
     0 | {DME,SU9} | {f,f} |               0.0304 | 0.024656880000000003
     1 | {SVO,SU9} | {f,f} |               0.0303 | 0.022782479999999997
     2 | {DME,CR2} | {f,f} |               0.0246 | 0.026817733333333336
     3 | {LED,321} | {f,f} | 0.022833333333333334 |           0.00325248
     4 | {VKO,CR2} | {f,f} |               0.0214 |            0.0142614
     5 | {BZK,SU9} | {f,f} | 0.019366666666666667 |           0.00495012
     6 | {SVO,CR2} | {f,f} | 0.018933333333333333 |  0.02477906666666667
     7 | {KJA,CN1} | {f,f} |               0.0148 |           0.00601272
     8 | {VKO,SU9} | {f,f} | 0.013833333333333333 | 0.013112279999999999
     9 | {DME,321} | {f,f} |               0.0127 |           0.00555648
(10 rows)
```

Число уникальных комбинаций значений

Другая ситуация, в которой планировщик ошибается с оценкой, связана с группировкой. Количество пар аэропортов, связанных прямыми рейсами, ограничено:

```
=> SELECT count(*) FROM (
  SELECT DISTINCT departure_airport, arrival_airport FROM flights
) t;
```

```
 count
-------
   618
(1 row)
```

Но планировщик не знает об этом:

```
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport FROM flights;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=5847.01..5955.16 rows=10816 width=8)
   Group Key: departure_airport, arrival_airport
   ->  Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

Расширенная статистика позволяет исправить и эту оценку:

```
=> CREATE STATISTICS flights_nd(ndistinct)
  ON departure_airport, arrival_airport FROM flights;
```

```
CREATE STATISTICS
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

```
=> EXPLAIN SELECT DISTINCT departure_airport, arrival_airport FROM flights;
```

```
                            QUERY PLAN
------------------------------------------------------------------
 HashAggregate  (cost=5847.01..5853.19 rows=618 width=8)
   Group Key: departure_airport, arrival_airport
   ->  Seq Scan on flights  (cost=0.00..4772.67 rows=214867 width=8)
(3 rows)
```

Статистику по уникальным комбинациям можно увидеть так:

```
=> SELECT n_distinct
FROM pg_stats_ext WHERE statistics_name = 'flights_nd';
```

```
  n_distinct
---------------
 {"5, 6": 618}
(1 row)
```

Статистика по выражению

Если в условиях используются обращения к функциям, планировщик не учитывает множество значений. Например, рейсов, совершенных в январе, будет примерно 1/12 от общего количества:

```
=> SELECT count(*) FROM flights
  WHERE extract(month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow') = 1;
```

```
 count
-------
 16831
(1 row)
```

Однако планировщик не понимает смысла функции extract и использует фиксированную селективность 0,5%:

```
=> EXPLAIN SELECT * FROM flights
  WHERE extract(month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow') = 1;
```

```
                                                                                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..5943.27 rows=1074 width=63)
   Workers Planned: 1
   ->  Parallel Seq Scan on flights  (cost=0.00..4835.87 rows=632 width=63)
         Filter: (date_part('month'::text, timezone('Europe/Moscow'::text, scheduled_departure)) = '1'::double precision)
(4 rows)
```

```
=> SELECT 214867 * 0.005;

 ?column?
----------
 1074.335
(1 row)
```

Ситуацию можно исправить, построив индекс по выражению, так как для таких индексов собирается собственная статистика. В общем случае функция extract имеет класс изменчивости STABLE, поскольку зависит от часового пояса, и поэтому не может участвовать в выражении индекса. Но с явным указанием часового пояса AT TIME ZONE функция постоянна, так что мы напишем обертку с классом изменчивости IMMUTABLE, указав тем самым, что функция гарантированно возвращает одно и то же значение при одних и тех же значениях параметров:

```
=> CREATE FUNCTION get_month(t timestamptz) RETURNS integer
AS $$
  SELECT extract(month FROM t AT TIME ZONE 'Europe/Moscow')::integer
$$ IMMUTABLE LANGUAGE sql;

CREATE FUNCTION

=> CREATE INDEX ON flights(get_month(scheduled_departure));

CREATE INDEX

=> ANALYZE flights;

ANALYZE
```

```
=> EXPLAIN SELECT * FROM flights
  WHERE get_month(scheduled_departure) = 1;

                                                                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on flights  (cost=191.68..3154.74 rows=16953 width=63)
   Recheck Cond: ((date_part('month'::text, timezone('Europe/Moscow'::text, scheduled_departure)))::integer = 1)
   ->  Bitmap Index Scan on flights_get_month_idx  (cost=0.00..187.44 rows=16953 width=0)
         Index Cond: ((date_part('month'::text, timezone('Europe/Moscow'::text, scheduled_departure)))::integer = 1)
(4 rows)
```

Оценка исправилась.

Статистика для индексов по выражению хранится вместе со статистикой для таблиц:

```
=> SELECT n_distinct FROM pg_stats WHERE tablename = 'flights_get_month_idx';

 n_distinct
------------
         12
(1 row)
```

Соединения

Селективность соединения — доля строк от декартового произведения двух таблиц. Рассмотрим пример:

```
=> EXPLAIN SELECT *
  FROM flights f JOIN aircrafts a ON a.aircraft_code = f.aircraft_code;

                                  QUERY PLAN
------------------------------------------------------------------------------
 Hash Join  (cost=1.20..59857.41 rows=214867 width=103)
   Hash Cond: (f.aircraft_code = ml.aircraft_code)
   ->  Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 width=63)
   ->  Hash  (cost=1.09..1.09 rows=9 width=72)
         ->  Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 width=72)
(5 rows)
```

Точное значение:

```
=> SELECT count(*)
  FROM flights f JOIN aircrafts a ON a.aircraft_code = f.aircraft_code;

 count
--------
 214867
(1 row)
```

Базовая формула для расчета селективности соединения (в предположении равномерного распределения) — минимальное из значений 1/nd1 и 1/nd2, где

- nd1 — число уникальных значений ключа соединения в первом наборе строк;
- nd2 — число уникальных значений ключа соединения во втором наборе строк.

```
=> SELECT s1.n_distinct, s2.n_distinct
FROM pg_stats s1, pg_stats s2
WHERE s1.tablename = 'flights'        AND s1.attname = 'aircraft_code'
  AND s2.tablename = 'aircrafts_data' AND s2.attname = 'aircraft_code';

 n_distinct | n_distinct
------------+------------
          8 |         -1
(1 row)
```

В данном случае получаем:

```
=> SELECT 214867 * 9 * least(1.0/8, 1.0/9);

          ?column?
-----------------------------
 214866.99999999999999785133
(1 row)
```

В более сложных случаях приведенная формула дала бы неправильный результат. Например, рейсы совершают разные модели самолетов с разной вместимостью, и для соединения рейсов с местами получили бы:

```
=> SELECT 214867 * 1339 * least(1.0/8, 1.0/8);

           ?column?
-------------------------------
 35963364.12500000000000000000
(1 row)
```

При этом точное значение:

```
=> SELECT count(*)
  FROM flights f JOIN seats s ON f.aircraft_code = s.aircraft_code;

  count
----------
 16518865
(1 row)
```

Однако планировщик умеет учитывать списки наиболее частых значений и гистограммы, и получает практически точную оценку:

```
=> EXPLAIN SELECT *
  FROM flights f JOIN seats s ON f.aircraft_code = s.aircraft_code;

                              QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=38.13..278610.66 rows=16529203 width=78)
   Hash Cond: (f.aircraft_code = s.aircraft_code)
   ->  Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 width=63)
   ->  Hash  (cost=21.39..21.39 rows=1339 width=15)
         ->  Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=15)
(5 rows)
```

К сожалению, ситуация ухудшается, когда соединяются несколько таблиц. Например, добавим в предыдущий запрос таблицу самолетов — это никак не повлияет на общее количество строк в выборке:

```
=> SELECT count(*)
  FROM flights f
    JOIN aircrafts a ON a.aircraft_code = f.aircraft_code
    JOIN seats s ON a.aircraft_code = s.aircraft_code;

  count
----------
 16518865
(1 row)
```

Однако теперь планировщик ошибается:

```
=> EXPLAIN SELECT *
  FROM flights f
    JOIN aircrafts a ON a.aircraft_code = f.aircraft_code
    JOIN seats s ON a.aircraft_code = s.aircraft_code;
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=39.33..8414210.29 rows=31967435 width=118)
   Hash Cond: (f.aircraft_code = ml.aircraft_code)
   ->  Hash Join  (cost=38.13..278610.66 rows=16529203 width=78)
         Hash Cond: (f.aircraft_code = s.aircraft_code)
         ->  Seq Scan on flights f  (cost=0.00..4772.67 rows=214867 width=63)
         ->  Hash  (cost=21.39..21.39 rows=1339 width=15)
               ->  Seq Scan on seats s  (cost=0.00..21.39 rows=1339 width=15)
   ->  Hash  (cost=1.09..1.09 rows=9 width=72)
         ->  Seq Scan on aircrafts_data ml  (cost=0.00..1.09 rows=9 width=72)
(9 rows)
```

Причина в том, что, соединив первые две таблицы, планировщик не имеет детальной статистики о результирующем наборе строк.

Итоги

Характеристики данных собираются в виде статистики

Статистика нужна для оценки кардинальности

Кардинальность используется для оценки стоимости

Стоимость позволяет выбрать оптимальный план

Основа успеха —
адекватная статистика и корректная кардинальность

Практика

1. Создайте индекс на таблице билетов (tickets) по имени пассажира (passenger_name).
2. Какая статистика имеется для этой таблицы?
3. Объясните оценку кардинальности и выбор плана выполнения следующих запросов:
   а) выборка всех билетов,
   б) выборка билетов на имя ALEKSANDR IVANOV,
   в) выборка билетов на имя ANNA VASILEVA,
   г) выборка билета с идентификатором 0005432000284.

1. Индекс

```
=> CREATE INDEX ON tickets(passenger_name);
```

```
CREATE INDEX
```

2. Наличие статистики

Некоторые основные значения:

```
=> SELECT reltuples, relpages FROM pg_class WHERE relname = 'tickets';
```

```
  reltuples   | relpages
--------------+----------
 2.949857e+06 |    49415
(1 row)
```

```
=> SELECT
    attname,
    null_frac nul,
    n_distinct,
    left(most_common_vals::text,20) mcv,
    cardinality(most_common_vals) mc,
    left(histogram_bounds::text,20) histogram,
    cardinality(histogram_bounds) hist,
    correlation
  FROM pg_stats WHERE tablename = 'tickets';
```

```
    attname     | nul | n_distinct  |         mcv          | mc  |      histogram       | hist |  correlation
----------------+-----+-------------+----------------------+-----+----------------------+------+---------------
 ticket_no      |   0 |          -1 |                      |     | {0005432000401,00054 |  101 |             1
 book_ref       |   0 | -0.49600527 |                      |     | {00008F,028252,05006 |  101 |   0.005945341
 passenger_id   |   0 |          -1 |                      |     | {"0000 126752","0099 |  101 | 2.6059448e-05
 passenger_name |   0 |       10320 | {"ALEKSANDR IVANOV", | 100 | {"ADELINA BOGDANOVA" |  101 |  0.0022156097
 contact_data   |   0 |          -1 |                      |     | {"{\"phone\": \"+700 |  101 | 1.8587435e-06
(5 rows)
```

- Ни один столбец не содержит неопределенных значений.
- Уникальных номеров бронирования примерно в два раза меньше, чем строк в таблице (то есть на каждое бронирование в среднем приходится два билета). Имеется около 10000 разных имен. Все остальные столбцы содержат уникальные значения.
- Размеры массивов наиболее частых значений и гистограмм соответствуют значению параметра default_statistics_target (100).
- Для имен пассажиров есть наиболее частые значения. Для других столбцов они не имеют смысла, так как максимальное количество билетов (5) встречается в 194 бронированиях, а остальные столбцы уникальны.
- Гистограммы есть для всех столбцов, они нужны для оценки предикатов с условиями неравенства.
- Строки таблицы физически упорядочены по номеру билета. Данные в других столбцах расположены более или менее хаотично.

3. Планы запросов

```
=> EXPLAIN SELECT * FROM tickets;
```

```
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on tickets  (cost=0.00..78913.57 rows=2949857 width=104)
(1 row)
```

Кардинальность равна числу строк в таблице; выбрано полное сканирование.

```
=> EXPLAIN SELECT * FROM tickets WHERE passenger_name = 'ALEKSANDR IVANOV';
```

```
                                      QUERY PLAN
---------------------------------------------------------------------------------------
 Bitmap Heap Scan on tickets  (cost=85.59..19956.82 rows=7375 width=104)
   Recheck Cond: (passenger_name = 'ALEKSANDR IVANOV'::text)
   ->  Bitmap Index Scan on tickets_passenger_name_idx  (cost=0.00..83.74 rows=7375 width=0)
         Index Cond: (passenger_name = 'ALEKSANDR IVANOV'::text)
(4 rows)
```

Селективность оценена по списку наиболее частых значений; выбрано сканирование по битовой карте.

```
=> EXPLAIN SELECT * FROM tickets WHERE passenger_name = 'ANNA VASILEVA';

                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tickets  (cost=6.48..1007.89 rows=264 width=104)
   Recheck Cond: (passenger_name = 'ANNA VASILEVA'::text)
   ->  Bitmap Index Scan on tickets_passenger_name_idx  (cost=0.00..6.41 rows=264 width=0)
         Index Cond: (passenger_name = 'ANNA VASILEVA'::text)
(4 rows)
```

Селективность оценена исходя из равномерного распределения; выбрано сканирование по битовой карте.

```
=> EXPLAIN SELECT * FROM tickets WHERE ticket_no = '0005432000284';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tickets_pkey on tickets  (cost=0.43..8.45 rows=1 width=104)
   Index Cond: (ticket_no = '0005432000284'::bpchar)
(2 rows)
```

Кардинальность равна 1, так как значения этого столбца уникальны; выбрано индексное сканирование.