

# Оптимизация запросов Статистика



## Авторские права

© Postgres Professional, 2019–2022

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

## Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

## Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

[edu@postgrespro.ru](mailto:edu@postgrespro.ru)

## Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

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

## 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.

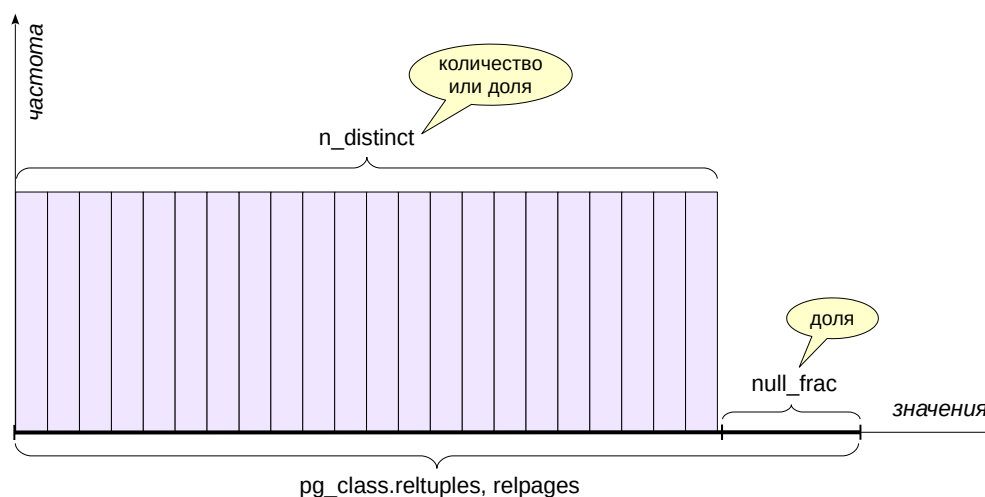
Кроме того, планировщик масштабирует количество строк в соответствии с отклонением реального размера файла данных от значения `relpages`.

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>

pg\_statistic (pg\_stats)



4

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  $300 \times \text{default\_statistics\_target}$  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;
```

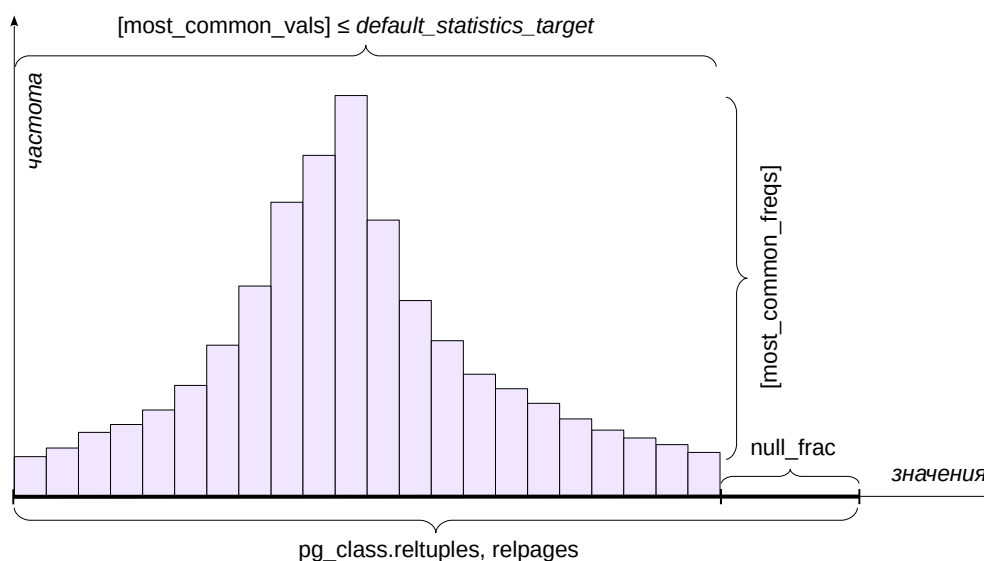
```
count
-----
16348
(1 row)
```

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)



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=19044 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,PEE,ROV,SGC}
most_common_freqs | {0.099366665,0.088633336,0.056366667,0.05226667,0.031633332,0.021466667,0.020533333,0.0198,0.0195,0.0178}
```

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?
-----
19044.37903355807
(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=1309 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
-----
0.42736667
(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 - 0.42736667) / (104 - 10);

?column?
-----
1308.9362310330851064
(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..515.72 rows=516 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=198451 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=198451 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=198451 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=198451 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');
```

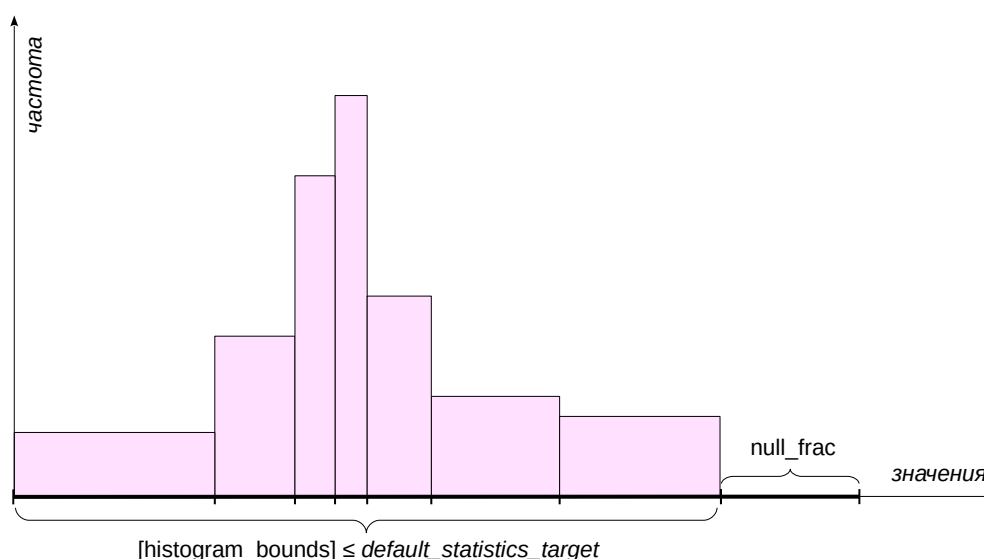
```

          QUERY PLAN
-----
Seq Scan on flights (cost=0.00..5309.84 rows=198451 width=63)
  Filter: ((status)::text = 'Arrived'::text)
(2 rows)

```

```
=> RESET plan_cache_mode;
```

```
RESET
```



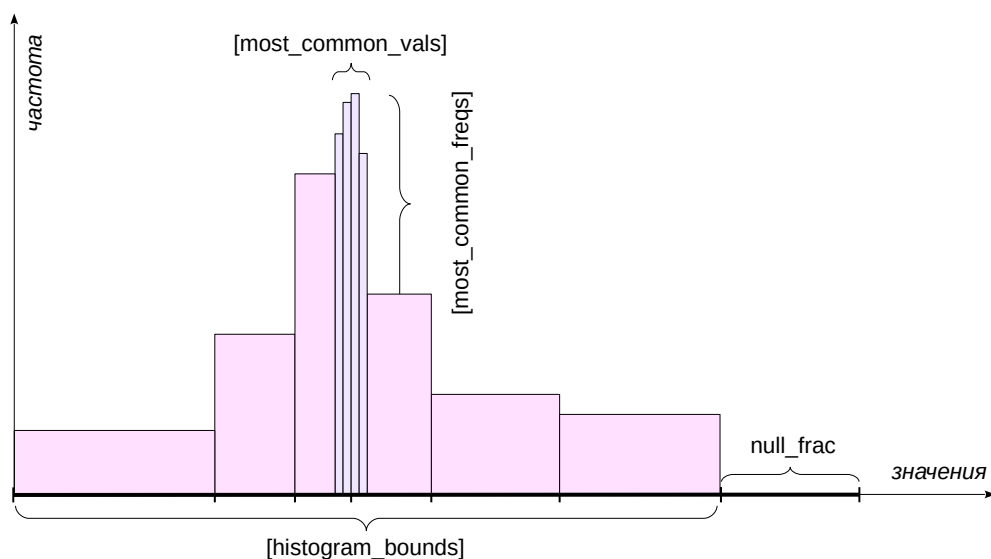
Если число различных значений слишком велико, чтобы записать их в массив, на помощь приходит гистограмма. Гистограмма состоит из нескольких корзин, в которые помещаются значения. Количество корзин ограничено тем же параметром *default\_statistics\_target*.

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  $1/(\text{число корзин})$ .

Оценить селективность условия *поле < значение* можно как  $N/(\text{общее число корзин})$ , 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  $1/n\_distinct$ .



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,BZK,ESL,JOK,KVX,MRV,NUX,PEZ,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 <= 'ESL';
```

```
          QUERY PLAN
-----
Seq Scan on flights (cost=0.00..5309.84 rows=45959 width=63)
  Filter: (arrival_airport <= 'ESL'::bpchar)
(2 rows)
```

Точное значение:

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

```
count
-----
46900
(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 <= 'ESL';
```

```
sum
-----
0.099366665
(1 row)
```

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

```
=> SELECT 214867 * (1 - 0.42736667) * (2.0 / 10.0) + 214867 * 0.099366665;
```

```
          ?column?
-----
45958.618351977000000000000000000000
(1 row)
```

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

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`

11

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 строки 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>

## CREATE STATISTICS

объект базы данных, создается вручную  
после создания статистика собирается автоматически  
pg\_statistic\_ext и pg\_statistic\_ext\_data; представление pg\_stats\_ext

### Функциональные зависимости между столбцами и списки наиболее частых комбинаций значений

улучшают оценку селективности условий  
с коррелированными предикатами

### Число уникальных комбинаций значений

улучшает оценку кардинальности для группировки

12

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 = 'VK0';
```

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

Оценка оказывается сильно заниженной:

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VK0';
```

### QUERY PLAN

```
Bitmap Heap Scan on flights (cost=10.49..816.84 rows=14 width=63)
  Recheck Cond: (flight_no = 'PG0007'::bpchar)
  Filter: (departure_airport = 'VK0'::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)
```

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": 0.011167}
(1 row)
```

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

```
=> EXPLAIN SELECT * FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VK0';
```

### QUERY PLAN

```
Bitmap Heap Scan on flights (cost=12.25..1264.87 rows=494 width=63)
  Recheck Cond: (flight_no = 'PG0007'::bpchar)
  Filter: (departure_airport = 'VK0'::bpchar)
  -> Bitmap Index Scan on flights_flight_no_scheduled_departure_key (cost=0.00..12.12 rows=494 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..5591.59 rows=717 width=63) (actual rows=5148 loops=1)
  Workers Planned: 1
  Workers Launched: 1
  -> Parallel Seq Scan on flights  (cost=0.00..4519.89 rows=422 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..5687.59 rows=1677 width=63)
  Workers Planned: 1
  -> Parallel Seq Scan on flights  (cost=0.00..4519.89 rows=986 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=5465 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	{SV0,SU9}	{f,f}	0.0314	0.023825753333333335
1	{DME,SU9}	{f,f}	0.0289	0.024210453333333333
2	{LED,321}	{f,f}	0.025433333333333332	0.0034996266666666667
3	{DME,CR2}	{f,f}	0.024466666666666668	0.025708266666666663
4	{VK0,CR2}	{f,f}	0.021866666666666666	0.014161333333333331
5	{SV0,CR2}	{f,f}	0.019366666666666667	0.025299766666666664
6	{BZK,SU9}	{f,f}	0.018633333333333333	0.004778828888888888
7	{KJA,CN1}	{f,f}	0.014966666666666666	0.005873611111111111
8	{OVb,CN1}	{f,f}	0.013433333333333334	0.008899695555555556
9	{VK0,SU9}	{f,f}	0.013266666666666666	0.013336266666666666

(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(nddistinct)
    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=182.96..3133.84 rows=16344 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..178.87 rows=16344 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)
```

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

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.