

Статистика

Базовая статистика

Postgres PROFESSIONAL

16

Авторские права

© Postgres Professional, 2019–2024

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

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

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

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

Статистика элементов составных значений

Использование статистики для оценки кардинальности
и селективности

Частные и общие планы выполнения

Частичный индекс и индекс по выражению

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` и `relpages` в таблице `pg_class`). Поскольку такая статистика крайне важна, она обновляется некоторыми DDL-операциями (`CREATE INDEX`, `CREATE TABLE AS SELECT`) и уточняется при очистке и анализе.

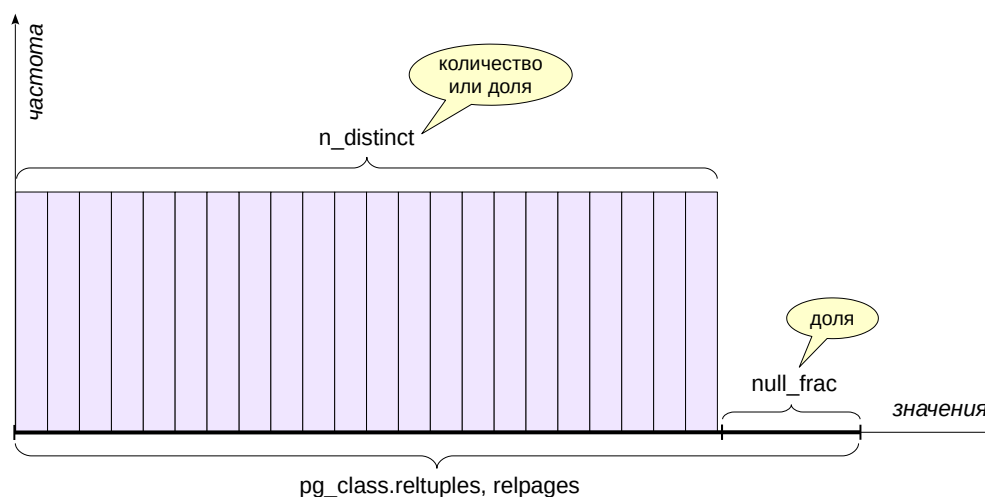
Кроме того, планировщик масштабирует количество строк в соответствии с отклонением реального размера файла данных от значения `relpages`.

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/16/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/16/planner-stats#PLANNER-STATS-SINGLE-COLUMN>

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=16444 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?
-----
16444.4875472039
(1 row)
```

Число уникальных значений

Проверим количество моделей самолетов в таблице рейсов:

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'flights' AND attname = 'aircraft_code';
```

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

Это соответствует действительности:

```
=> SELECT count(DISTINCT aircraft_code) FROM flights;
```

```
 count
-----
      8
(1 row)
```

А в таблице самих самолетов?

```
=> SELECT n_distinct FROM pg_stats
WHERE tablename = 'aircrafts_data' AND attname = 'aircraft_code';
```

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

Здесь значение -1 говорит о том, что каждое значение является уникальным. Это неудивительно, ведь aircraft_code является первичным ключом в этой таблице.

Кардинальность соединения

Селективность соединения — доля строк от декартова произведения двух таблиц, которая остается после применения условия соединения.

Поэтому для расчета кардинальности соединения оптимизатор оценивает кардинальность декартова произведения и умножает его на селективность условия соединения (и на селективность условий фильтров, если они есть).

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 кардинальности:

```
=> 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.

В данном случае получаем ровно то, что требуется:

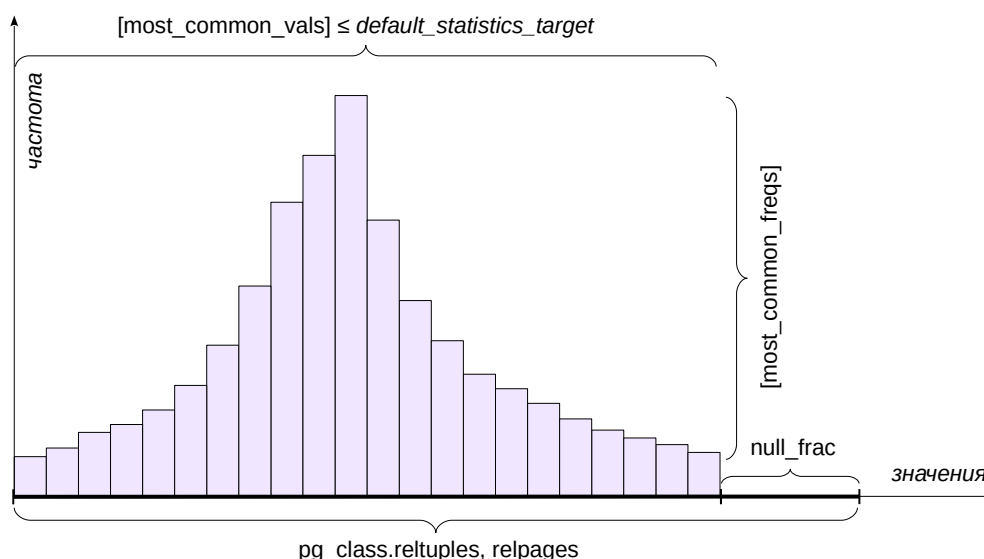
```
=> SELECT round(214867 * 9 * least(1.0/8, 1.0/9));
```

```
round
```

```
-----
```

```
214867
```

```
(1 row)
```



Если бы данные всегда были 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 таком 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=18973 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,AER,ROV,BZK}
most_common_freqs |
{0.096433334,0.0883,0.0581,0.0536,0.0326,0.022333333,0.018466666,0.0183,0.018033333,0.0177
}
```

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?
-----
18972.755487821996
(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`.

Если же указанного значения нет в списке наиболее частых, то селективность 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=1317 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.attnname = 'arrival_airport';
```

```
sum
-----
0.4238666
(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:

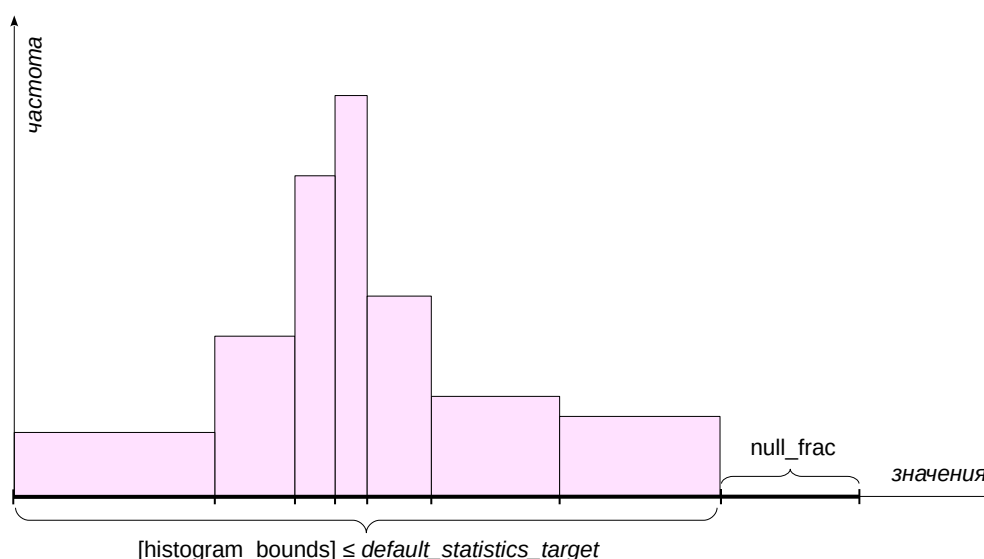
```
=> SELECT n_distinct
FROM pg_stats s
WHERE s.tablename = 'flights' AND s.attnname = 'arrival_airport';
```

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

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

```
=> SELECT 214867 * (1 - 0.4238666) / (104 - 10);
```

```
?column?
-----
1316.9367580617021277
(1 row)
```



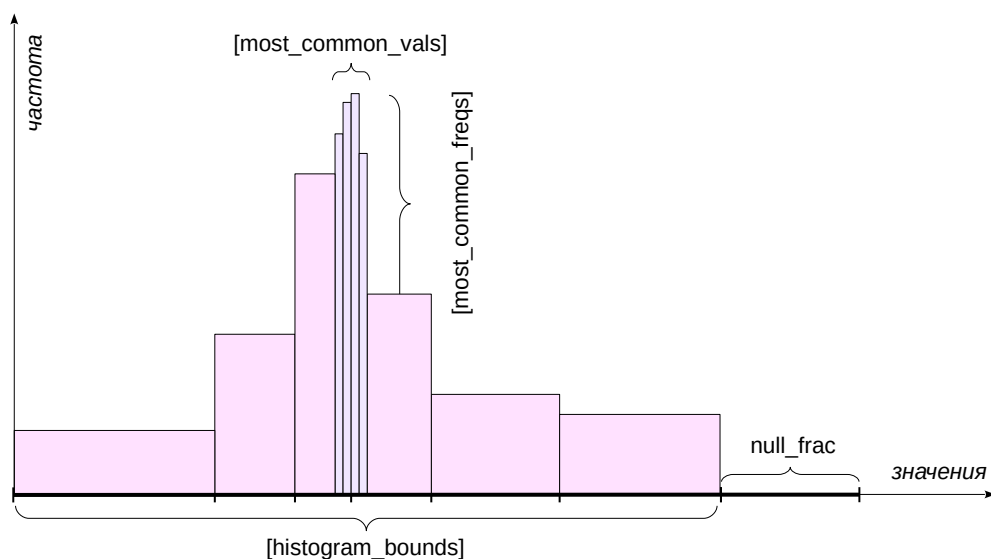
Если число различных значений слишком велико, чтобы записать их в массив, на помощь приходит гистограмма. Гистограмма состоит из нескольких корзин, в которые помещаются значения. Количество корзин ограничено тем же параметром *default_statistics_target*.

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,CSY,HMA,KHV,MJZ,NOJ,OVS,REN,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=53214 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
-----
0.13243334
(1 row)
```

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

```
=> SELECT 214867 * (
          0.13243334 + (1 - 0.4238666) * (2.0 / 10.0)
);

?column?
-----
53213.965517340000000000000000000000
(1 row)
```

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

Кардинальность соединения

В случае неравномерного распределения данных в ключах соединения базовая формула расчета селективности соединения (минимальное из значений $1/nd_1$ и $1/nd_2$) дает неправильный результат. Например, рейсы совершают разные модели самолетов с разной вместимостью, и для соединения рейсов с местами мы получили бы:

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

```
round
-----
35963364
(1 row)
```

При этом точное значение в два раза меньше:

```
=> 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..278932.82 rows=16561419 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..8414656.85 rows=31967435 width=118)
  Hash Cond: (f.aircraft_code = ml.aircraft_code)
  -> Hash Join (cost=38.13..278932.82 rows=16561419 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)
```

Дело в том, что, соединив первые две таблицы, планировщик не имеет детальной статистики о результирующем наборе строк. Во многих случаях именно это является основной причиной плохих оценок.

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`

Сервер хранит еще несколько показателей статистики.

В поле `pg_stats.correlation` записывается показатель физической упорядоченности значений столбца. 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 `pg_stats.avg_width` сохраняется средний размер значений в данном столбце в байтах для расчета необходимого для операции объема памяти.

<https://postgrespro.ru/docs/postgresql/16/view-pg-stats>

Наиболее частые элементы

`pg_stats.most_common_elems`

`pg_stats.most_common_elem_freqs`

Гистограмма количества элементов

`pg_stats.elem_count_histogram`

Для таких составных типов, как массивы или `tsvector`, в `pg_stats` хранится распределение не только самих значений, но и их элементов:

- `most_common_elems` и `most_common_elem_freqs` содержат наиболее частые элементы и их частоты;
- `elem_count_histogram` содержит гистограмму количества элементов в значении (например, в случае массива — гистограмму длин массивов).

Это позволяет более точно планировать запросы с участием полей *не в первой нормальной форме*. В частности, эта информация важна для классов операторов индексного метода GIN, поскольку позволяет отделить частые значения (то есть встречающиеся во многих документах, условие с низкой селективностью) от редких (условие с высокой селективностью).

Элементы составных полей

Для примера рассмотрим таблицу `pg_constraint` — в ней хранятся ограничения целостности, определенные для таблиц. И в ней есть поле `conkey` с массивом номеров столбцов, образующих ограничение:

```
=> SELECT conname, conkey
FROM pg_constraint
WHERE conname LIKE 'boarding%';
```

conname	conkey
boarding_passes_flight_id_boarding_no_key	{2,3}
boarding_passes_flight_id_seat_no_key	{2,4}
boarding_passes_pkey	{1,2}
boarding_passes_ticket_no_fkey	{1,2}

Для этого поля можно получить наиболее частые значения и их частоты. Чтобы было удобнее просматривать статистику, соберем ее с уменьшенным значением ориентира:

```
=> SET default_statistics_target = 7;
```

SET

```
=> ANALYZE pq constraint;
```

ANALYZE

```
=> SELECT most_common_vals, most_common_freqs
FROM pg_stats
WHERE tablename = 'pg_constraint' AND attname = 'conkey' \ax
```

```
-[ RECORD 1
]-+-----
most_common_vals | {"{1}", "{2}", "{2,3}", "{1,2}", "{1,2,3}", "{2,3,4}", "{2,4}" }
most_common_freqs | {0.4057971, 0.10869565, 0.10144927, 0.06521739, 0.036231883, 0.028985508, 0.028985508}
```

В данном случае значения в столбце — это массивы элементов. Такая статистика не позволит оценить кардинальность, например, для условия вхождения элемента в массив. Однако планировщик не ошибается в оценке:

```
=> SELECT count(*)
FROM pg_constraint
WHERE conkey @> ARRAY[2::smallint];
```

```
count
-----
      65
(1 row)
```

```
=> EXPLAIN SELECT *
FROM pg_constraint
WHERE conkey @> ARRAY[2::smallint];
```

```

              QUERY PLAN
-----
Seq Scan on pg_constraint (cost=0.00..5.72 rows=65 width=1073)
  Filter: (conkey @> '{2}':::smallint[])
(2 rows)

```

Для таких условий используется статистика по отдельным элементам:

```
=> SELECT most_common_elems, most_common_elem_freqs, elem_count_histogram
FROM pg_stats
WHERE tablename = 'pg_constraint' AND attname = 'conkey' \gx
```

```

-[ RECORD 1
]-+-----+-----
-----
most_common_elems      | {1,2,3,4,5,6,7,8,9,10,20}
most_common_elem_freqs |
{0.5514706,0.4779412,0.30882353,0.11029412,0.036764707,0.022058824,0.022058824,0.022058824
,0.022058824,0.007352941,0.007352941,0.007352941,0.5514706,0}
elem_count_histogram   | {1,1,1,1,2,2,4,1.5882353}

```

Для частот и гистограмм в конце массива содержится дополнительная информация (минимум, максимум, среднее), поэтому количество значений превышает установленный ориентир.

Планировщик получает оценку, используя частоту элемента 2:

```

=> SELECT 0.4779412 * reltuples rows
FROM pg_class
WHERE relname = 'pg_constraint';

```

```

      rows
-----
 65.9558856
(1 row)

```

```

=> RESET default_statistics_target;

```

```

RESET

```

```

=> ANALYZE pg_constraint;

```

```

ANALYZE

```

Частные планы

может быть полезен при неравномерном распределении,
но запрос планируется заново при каждом вызове

Seq Scan on tasks
Filter: status = 'done'

Index Scan on tasks
Index Cond: status = 'todo'

строится
с учетом значений
параметров

Общий план

оптимален для равномерного распределения
кешируется в случае подготовленных операторов

Index Scan on tasks
Index Cond: id = \$1

строится
без учета значений
параметров

При простом протоколе запросов (вспомните тему «Планирование и выполнение») каждый запрос планируется заново с учетом значений параметров.

Расширенный протокол позволяет подготавливать операторы, причем операторы могут иметь параметры. Подготовка всегда включает разбор и переписывание запроса, и дерево разбора сохраняется в локальной памяти обслуживающего процесса.

При выполнении подготовленного оператора есть варианты. Запрос может, как обычно, планироваться каждый раз заново с учетом значений параметров. Такие планы называются *частными* (custom).

Это имеет смысл при неравномерном распределении, ведь для разных значений оптимальные планы могут отличаться. Например, для высокоселективных условий может лучше работать индексный доступ, а если значение встречается часто — последовательный.

Но запрос можно спланировать и без учета параметров. Это позволяет сохранить не только дерево разбора, но и сам план, и не тратить время на повторное планирование. Такой план называется *общим* (generic).

Общий план отлично работает в случае равномерного распределения, когда селективность условия не зависит от конкретного значения. Но в случае неравномерного распределения общий план может хорошо работать для одних значений и плохо — для других.

Разберемся, как планировщик решает, какой вариант использовать.

Частные и общие планы

Подготовим запрос и создадим индекс:

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

PREPARE

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

CREATE INDEX

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

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

QUERY PLAN

```
-----  
Index Scan using flights_status_idx on flights (cost=0.29..326.79 rows=308 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=198344 width=63)  
  Filter: ((status)::text = 'Arrived'::text)  
(2 rows)
```

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

Однако планировщик строит и общий план без учета конкретных значений параметров. Если в какой-то момент оказывается, что стоимость общего плана не превышает среднюю стоимость уже построенных ранее частных планов, планировщик начинает пользоваться общим планом, больше не выполняя планирование. Но первые пять раз используются частные планы, чтобы накопить статистику.

Выполним оператор еще три раза:

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

QUERY PLAN

```
-----  
Seq Scan on flights (cost=0.00..5309.84 rows=198344 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=198344 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=198344 width=63)  
  Filter: ((status)::text = 'Arrived'::text)  
(2 rows)
```

Количество выполнений общего и частных планов можно посмотреть в представлении pg_prepared_statements:

```
=> SELECT name, generic_plans, custom_plans  
FROM pg_prepared_statements;
```

name	generic_plans	custom_plans
f	0	5

(1 row)

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)

```

```
=> SELECT name, generic_plans, custom_plans
FROM pg_prepared_statements;
```

name	generic_plans	custom_plans
f	1	5

(1 row)

Имея текст запроса с параметрами, представленными номерами (например, из журнала сообщений сервера), можно посмотреть общий план такого запроса следующим образом:

```
=> EXPLAIN (generic_plan)
SELECT * FROM flights WHERE status = $1;
```

```

                                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)

```

Переход на общий план может быть нежелателен в случае неравномерного распределения значений. 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=198344 width=63)
  Filter: ((status)::text = 'Arrived'::text)
(2 rows)

```

```
=> RESET plan_cache_mode;
```

```
RESET
```

Частичный индекс

Поиск уже прибывших рейсов будет выполняться последовательным сканированием таблицы `flights`, поэтому индексные записи с ключом `Arrived` никогда не используются. При этом они занимают место и требуют ресурсов для синхронизации с изменениями строк таблицы.

Можно поместить в индекс только ссылки на строки, удовлетворяющие условиям с высокой селективностью:

```
=> CREATE INDEX on flights(status)
WHERE status IN ('Delayed', 'Departed', 'Cancelled');
```

CREATE INDEX

Размер частичного индекса существенно меньше, чем полного:

```
=> SELECT indexname, pg_size_pretty(pg_relation_size(indexname::text)) size
FROM pg_indexes
WHERE indexname LIKE 'flights_status%';
```

indexname	size
flights_status_idx	1472 kB
flights_status_idx1	16 kB

(2 rows)

А поиск отмененных рейсов будет теперь использовать частичный индекс и станет выполняться быстрее:

```
=> EXPLAIN SELECT *
FROM flights WHERE status = 'Cancelled';
```

QUERY PLAN

```
-----
Index Scan using flights_status_idx on flights (cost=0.29..326.79 rows=308 width=63)
  Index Cond: ((status)::text = 'Cancelled'::text)
(2 rows)
```

Индекс по выражению

Если в условиях используются не столбцы, а, например, 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: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE
'Europe/Moscow'::text)) = '1'::numeric)
(4 rows)
```

```
=> SELECT 214867 * 0.005;
```

?column?
1074.335

(1 row)

Ситуацию можно исправить, построив индекс по выражению, так как для таких выражений собирается собственная статистика.

```
=> CREATE INDEX ON flights(
  extract(month FROM scheduled_departure AT TIME ZONE 'Europe/Moscow')
);
```

CREATE INDEX

```
=> ANALYZE flights;
```

ANALYZE

Теперь оценка исправилась:

=> EXPLAIN

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

QUERY PLAN

```
-----
Bitmap Heap Scan on flights (cost=306.64..3215.67 rows=16287 width=63)
  Recheck Cond: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE
'Europe/Moscow'::text)) = '1'::numeric)
    -> Bitmap Index Scan on flights_extract_idx (cost=0.00..302.57 rows=16287 width=0)
          Index Cond: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE
'Europe/Moscow'::text)) = '1'::numeric)
(4 rows)
```

Статистика для индексов по выражению хранится вместе с базовой статистикой по столбцам таблиц:

```
=> SELECT n_distinct, most_common_vals, most_common_freqs
FROM pg_stats
WHERE tablename = 'flights_extract_idx' \gx
```

```
-[ RECORD 1
]-+-----
n_distinct      | 12
most_common_vals | {8,9,7,10,5,11,3,4,12,1,6,2}
most_common_freqs |
{0.12116667,0.111,0.08003333,0.07863333,0.0783,0.077766664,0.07773333,0.07723334,0.0761666
7,0.0758,0.0734,0.07276667}
```

Индекс можно построить только по детерминированному выражению, то есть для одинаковых значений параметров (столбцов) оно всегда должно выдавать одинаковый результат. Подробнее об этом рассказано в теме «Функции».

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 |                      |   | {0005432000330,00054 |
101 |  0.9999998
book_ref     |  0  |    -0.48490152 |                      |   | {0001CF,028E70,0525F |
101 |  0.0060454095
passenger_id |  0  |         -1 |                      |   | {"0000 102302","0103 |
101 |  0.0088426955
contact_data |  0  |         -1 |                      |   | {"{"phone\": \"+700 |
101 |  0.0029053022
passenger_name |  0  |        10366 | {"ALEKSANDR IVANOV", | 100 | {"ADELINA IVANOVA", |
101 | -0.00157338
(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=92.63..20749.30 rows=7768 width=104)  
  Recheck Cond: (passenger_name = 'ALEKSANDR IVANOV'::text)  
    -> Bitmap Index Scan on tickets_passenger_name_idx (cost=0.00..90.69 rows=7768  
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=6.46..1000.50 rows=262 width=104)  
  Recheck Cond: (passenger_name = 'ANNA VASILEVA'::text)  
    -> Bitmap Index Scan on tickets_passenger_name_idx (cost=0.00..6.39 rows=262 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.