



## Copyright

© Postgres Professional, 2019–2024

Authors Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti summit, Nepal)

## Use of course materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

## Feedback

Please send your feedback, comments and suggestions to:

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

## Disclaimer

In no event shall Postgres Professional company be liable for any damages or loss, including loss of profits, that arise from direct or indirect, special or incidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Basic statistics

Most common values histogram

Statistics for Elements of Composite Values

Leveraging statistics to estimate cardinality and selectivity

Private and Shared Execution Plans

Partial Index and Expression Index

## Table size

rows (`pg_class.reltuples`) and pages (`pg_class.relpages`)

## Collected

DDL operations

vacuum

through analysis

## Configuration

`default_statistics_target = 100`

Base statistics are collected at the table level and the column level.

Table statistics include data on the object's size, such as `reltuples` and `relpages` in the `pg_class` table. Because this statistics is crucial, it is updated by certain DDL operations (`CREATE INDEX`, `CREATE TABLE AS SELECT`) and refined during vacuum and analyze.

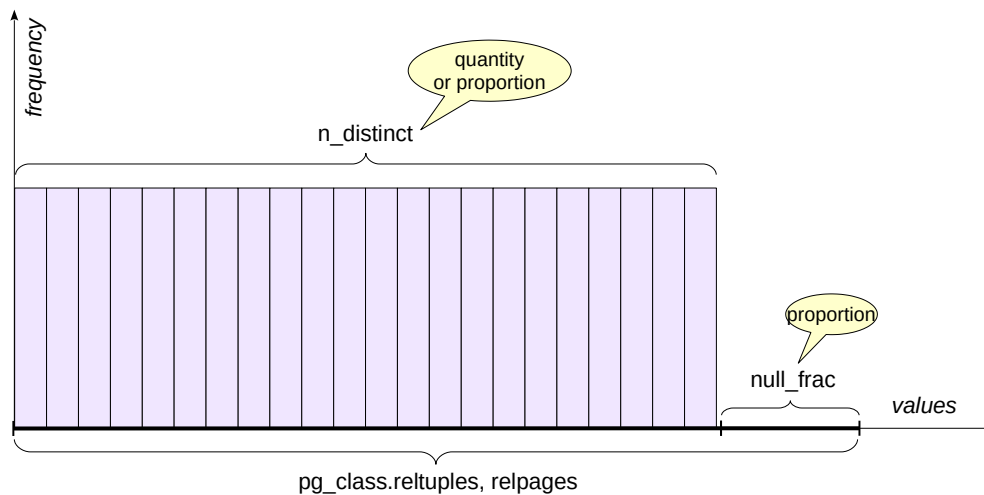
Additionally, the planner adjusts the row count based on the difference between the actual data file size and the `relpages` value.

When analyzing, a random sample of rows is examined. Research has shown that the sample size ensuring accurate estimates is largely independent of the table size. The sample size is based on the statistical target defined by the parameter `default_statistics_target`, multiplied by 300.

Keep in mind that the statistics doesn't have to be perfectly accurate for the planner to select an acceptable plan; often, being in the right ballpark is sufficient.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

pg\_statistic (pg\_stats)



4

During table analysis, all other statistics are collected separately for each column. This is typically handled by autoanalysis, with its configuration discussed in the DBA2 course.

The `pg_statistic` table stores column-level statistics. But the `pg_stats` view is easier to use, as it displays the information in a more convenient format.

The `null_frac` field indicates the proportion of rows with null values in the column (ranging from 0 to 1).

The `n_distinct` field contains the number of distinct values in the column. If `n_distinct` is negative, its absolute value represents the proportion of unique values. For example, -1 indicates that all values are unique (a typical case for a primary key).

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## Row Count

Let's start by estimating cardinality in a simple query without any conditions.

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

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

Exact value:

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

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

The optimizer retrieves the value from pg\_class:

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

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

The parameter controlling the statistics target defaults to 100:

```
=> SHOW default_statistics_target;
```

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

Because the analysis considers  $300 \times \text{default\_statistics\_target}$  rows, estimates for larger tables may not be entirely accurate.

## Proportion of Null Values

Some flights have not yet departed, so their departure times are not yet determined:

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

```
              QUERY PLAN
-----
Seq Scan on flights (cost=0.00..4772.67 rows=16058 width=63)
  Filter: (actual_departure IS NULL)
(2 rows)
```

Exact value:

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

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

The optimizer's estimate is calculated as the total number of rows multiplied by the proportion of NULL values:

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

```
      ?column?
-----
16057.7268037647
(1 row)
```

## Count of unique values

Let's check the number of aircraft models in the flights table:

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

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

This is accurate:

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

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

What about the airplanes table?

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

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

A value of -1 here indicates that all values are unique. This makes sense since aircraft\_code is the primary key in this table.

## Join Cardinality

Join selectivity refers to the fraction of rows from the Cartesian product of two tables that remains after applying the join condition.

Therefore, to calculate join cardinality, the optimizer estimates the cardinality of the Cartesian product and multiplies it by the join condition's selectivity (and the selectivity of any filter conditions, if applicable).

Consider an example:

```
=> 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)
```

Cardinality value:

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

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

The basic formula for calculating join selectivity (assuming uniform distribution) is the smaller of  $1/nd1$  and  $1/nd2$ , where

- nd1 - the count of distinct join key values in the first set of rows;
  - nd2 - the number of unique join key values in the second row set.
- 

This gives us exactly what's needed:

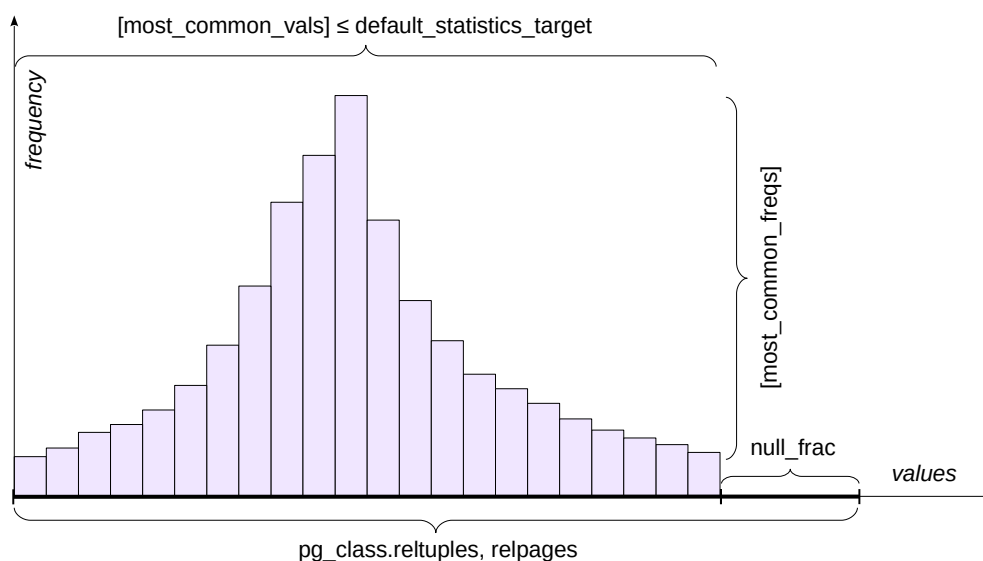
```
=> SELECT round(214867 * 9 * least(1.0/8, 1.0/9));
```

round

-----  
214867

(1 row)

# Most common values



6

Had the data been uniformly distributed—that is, if all values occurred with equal frequency—this information would have been nearly sufficient (minimum and maximum values would still be required).

However, non-uniform distributions are very common in practice. Therefore, the following information is also gathered.

- The most common values array — the `most_common_vals` field;
- The array of frequencies for these values is the `most_common_freqs` column.

The frequencies from these arrays directly serve as a selectivity estimate for querying a specific value.

This works well as long as the number of distinct values isn't too large. The maximum size of each array is constrained by the `default_statistics_target` parameter. This value can be adjusted on a per-column basis; when doing so, the sample size is determined by the table's maximum value.

The tricky part is "large" values. To prevent `pg_statistic` from growing and to avoid overloading the planner with unnecessary work, values exceeding 1 KB are excluded from statistics and analysis. In fact, if such large values are stored in the column, they're likely unique and won't be included in `most_common_vals`.



## Most Common Values

To experiment, limit the number of most frequent values (which is determined by the `default_statistics_target` parameter) at the column level:

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

```
ALTER TABLE
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

If a value is among the most common values, its selectivity can be found directly in the statistics. Example (Sheremetyevo):

```
=> EXPLAIN
SELECT * FROM flights WHERE arrival_airport = 'SV0';
```

QUERY PLAN

```
-----
Seq Scan on flights (cost=0.00..5309.84 rows=19195 width=63)
  Filter: (arrival_airport = 'SV0'::bpchar)
(2 rows)
```

Exact value:

```
=> SELECT count(*) FROM flights WHERE arrival_airport = 'SV0';
```

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

Here's what the list of most common values and their frequencies looks like:

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

```
-[ RECORD 1
]------+-----
most_common_vals | {DME,SV0,LED,VKO,OVB,KJA,SVX,PEE,R0V,BZK}
most_common_freqs |
{0.09926666,0.08933333,0.057733335,0.054366667,0.0338,0.0203,0.02,0.0194,0.019233333,0.018
833334}
```

Cardinality is calculated by multiplying the number of rows by the value's frequency:

```
=> SELECT 214867 * s.most_common_freqs[array_position((s.most_common_vals::text::text[]), 'SV0')]
FROM pg_stats s
WHERE s.tablename = 'flights' AND s.attname = 'arrival_airport';
```

```
?column?
-----
19194.785277836025
(1 row)
```

The Most Common Values list can also be used to estimate the selectivity of inequalities. This involves identifying all values in `most_common_vals` that satisfy the inequality and summing the frequencies of the corresponding entries from `most_common_freqs`.

If the specified value is not among the most frequent values, selectivity is calculated under the assumption that all other data (excluding the most frequent ones) are uniformly distributed.

For example, Vladivostok is not among the most common values.

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

## QUERY PLAN

```
Seq Scan on flights (cost=0.00..5309.84 rows=1298 width=63)
  Filter: (arrival_airport = 'VVO'::bpchar)
(2 rows)
```

Exact value:

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

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

To estimate, sum the frequencies of the most common values:

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

```
sum
-----
0.4322667
(1 row)
```

The remaining rows account for less frequent values. Since we assume a uniform distribution of these values, the selectivity is calculated as  $1/nd$ , where  $nd$  represents the number of unique values.

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

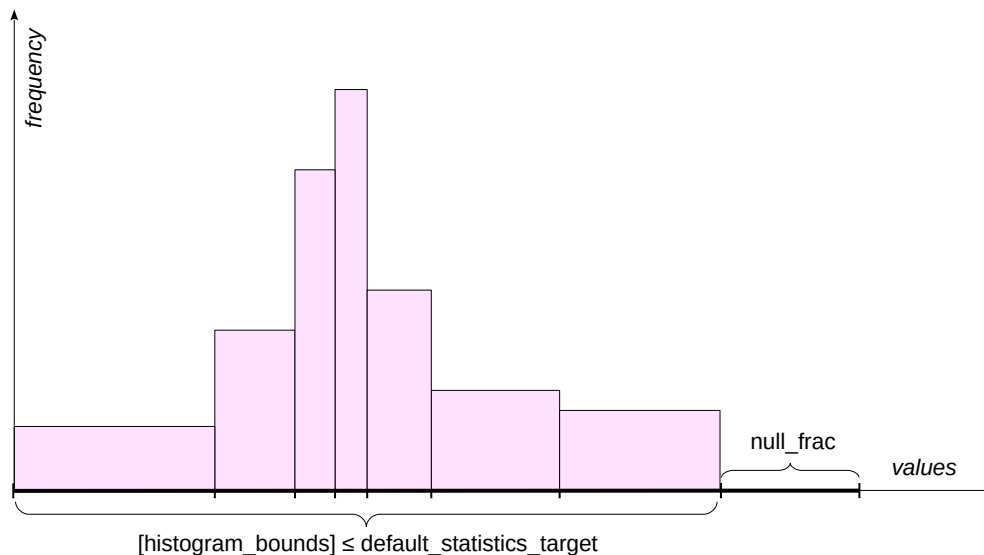
```
n_distinct
-----
104
(1 row)
```

Given that 10 of these values are among the most frequent and no null values are present, the following estimate is obtained:

```
=> SELECT 214867 * (1 - 0.4322667) / (104 - 10);
```

```
?column?
-----
1297.7356486287234043
(1 row)
```

# Histogram



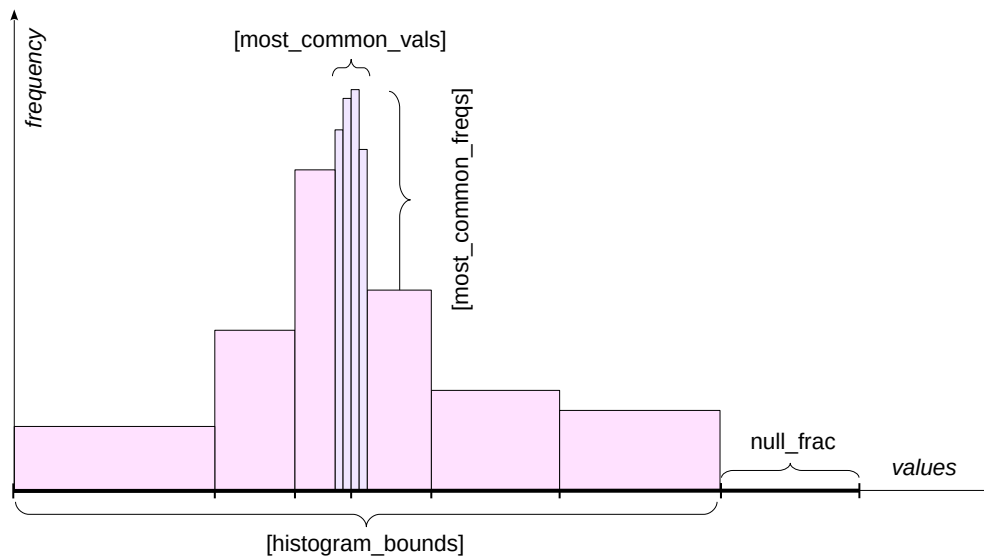
When the number of distinct values grows too large to store them all in an array, the system applies the histogram representation. A histogram employs several buckets to store values in. The number of bins is limited by the same parameter, `default_statistics_target`.

The width of the bins is set to ensure each contains roughly the same number of values (as shown by equal rectangle areas in the figure).

With this setup, only the array of extreme values for each bin needs to be stored — the `histogram_bounds` field. The frequency of a bin is 1 divided by the number of bins.

To estimate the selectivity of the condition field  $< \text{value}$ , calculate  $N$  divided by the total number of bins, where  $N$  represents the number of bins located to the left of value. The estimate can be refined by including a portion of the bin that contains the value itself.

However, when estimating the selectivity of the condition field  $= \text{value}$ , the histogram cannot help, and you must rely on the assumption of a uniform distribution, taking  $1/n_{\text{distinct}}$  as the estimate.



However, the two methods are typically integrated: a list of the most common values is created, with all remaining values represented in a histogram.

The histogram is constructed to exclude values that are already in the list. This helps improve the estimates.

## Histogram

Under "greater than" and "less than" conditions, the estimation process may utilize the most frequent values list, a histogram, or both. The histogram is constructed to exclude the most frequent values and NULLs:

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

```
          histogram_bounds
-----
{AAQ,CEK,G0J,KGP,KZN,NBC,OMS,REN,TJM,ULY,YKS}
(1 row)
```

The number of histogram bins is set by the `default_statistics_target` parameter, with boundaries chosen to ensure each bin contains roughly the same number of values.

Consider the following example:

```
=> EXPLAIN
SELECT * FROM flights WHERE arrival_airport <= 'G0J';
```

```
          QUERY PLAN
-----
Seq Scan on flights (cost=0.00..5309.84 rows=49773 width=63)
  Filter: (arrival_airport <= 'G0J'::bpchar)
(2 rows)
```

Exact value:

```
=> SELECT count(*) FROM flights WHERE arrival_airport <= 'G0J';
```

```
count
-----
50520
(1 row)
```

How is the estimate calculated?

Consider the frequency of the most common values within the given range:

```
=> 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 <= 'G0J';
```

```
sum
-----
0.118099995
(1 row)
```

The specified interval covers exactly 2 out of 10 histogram bins, with no null values present in this column, leading to the following estimate:

```
=> SELECT 214867 * (
          0.118099995 + (1 - 0.4322667) * (2.0 / 10.0)
);
```

```
          ?column?
-----
49773.221819885000000000000000000000
(1 row)
```

In general, even partially filled bins are considered using linear approximation.

## Join Cardinality

In the case of uneven data distribution in the join keys, the basic formula for calculating join selectivity (taking the minimum of  $1/nd_1$  and  $1/nd_2$ ) produces an incorrect result. For example, flights use different aircraft models with varying capacities, and joining flights with seats would result in:

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

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

However, the exact value is half:

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

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

---

However, the planner can account for lists of the most common values and histograms, leading to a nearly accurate estimate:

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

QUERY PLAN

```
-----
Hash Join (cost=38.13..278910.40 rows=16559177 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)
```

---

Unfortunately, the situation gets worse when joining multiple tables. For instance, adding the airplanes table to the previous query won't affect the total number of rows in the sample:

```
=> 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)
```

---

However, the planner now makes an error:

```
=> 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..8414625.77 rows=31967435 width=118)
  Hash Cond: (f.aircraft_code = ml.aircraft_code)
    -> Hash Join (cost=38.13..278910.40 rows=16559177 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)
```

The problem is that, when joining the first two tables, the planner doesn't have detailed statistics on the resulting row set. In many cases, this is the main cause of poor estimates.

## Ordering (Use a bit map?)

`pg_stats.correlation` (1 = ascending, 0 = chaotic, -1 = descending)

## Visibility (Use index-only scan?)

`pg_class.relallvisible`

## Average value size in bytes (memory estimate)

`pg_stats.avg_width`

The server maintains additional statistical metrics.

The `pg_stats.correlation` field records the physical ordering of the column's values. If the values are stored in strictly ascending order, the value will be close to 1; if in descending order, close to -1. The more randomly the data is arranged on disk, the closer the value gets to zero. The optimizer uses this field when deciding between a bitmap scan and a regular index scan.

The `pg_class.relallvisible` field tracks the number of table pages containing only the latest tuple versions (this data is updated in conjunction with the visibility map). If the count is insufficient, the planner may prefer a bitmap scan over an index-only scan.

The `pg_stats.avg_width` field stores the average size of values in bytes for this column to estimate the memory needed for the operation.

<https://postgrespro.com/docs/postgresql/16/sql-copy>

## Most Common Elements

`pg_stats.most_common_elems`  
`pg_stats.most_common_elem_freqs`

## Element Count Histogram

`pg_stats.elem_count_histogram`

For composite types like arrays or tsvector, `pg_stats` stores not just the distribution of the values themselves, but also their elements:

- The `most_common_elems` and `most_common_elem_freqs` columns store the most frequent elements and their frequencies;
- `elem_count_histogram` contains an element count histogram within the value (such as, for an array, a histogram of array lengths)

This enables more accurate query planning for fields not in the first normal form. In particular, this information is crucial for GIN index method operator classes, as it enables distinguishing between frequent values (those appearing in many documents, representing low-selectivity conditions) and rare values (high-selectivity conditions).



## Composite Field Elements

As an example, let's look at the `pg_constraint` table, which contains integrity constraints applied to tables. It also includes a `conkey` column that holds an array of column numbers forming the constraint:

```
=> 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}

(4 rows)

For this field, we can retrieve the most common values and their frequencies. To make viewing the statistics easier, we'll collect them with a lower target value:

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

SET

```
=> ANALYZE pg_constraint;
```

ANALYZE

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

```
-[ 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}
```

In this case, the column contains arrays of elements. Such statistics cannot accurately estimate cardinality, especially for conditions checking if an element is present in the array. However, the planner's estimation remains accurate:

```
=> 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)
```

For such conditions, per-element statistics are used:

```
=> 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}
```

At the end of the array, additional information (minimum, maximum, average) is available for frequencies and histograms, causing the number of values to exceed the target setting.

---

The query planner uses the frequency of element 2 to generate an estimate:

```
=> 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

## Private Plans

May be beneficial in cases of uneven distribution, but the query is re-planned each time it's executed.

Seq Scan on tasks  
Filter: status = 'done'

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

is built  
with values  
parameters

## Common Plan

Best suited for uniform distribution

Cached within the session for prepared statements

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

is built  
without considering the values  
parameters

With the simple query protocol (refer to the "Planning and Execution" section), each query is re-planned, taking into account the parameter values.

The extended protocol allows for the preparation of statements, which can have parameters. Preparation always involves parsing and query rewriting, with the parse tree stored in the backend process's local memory.

When executing a prepared statement, there are options. The query is typically re-planned each time, taking into account the parameter values. Such plans are called custom and (custom)

It makes sense when there's an uneven distribution, as optimal plans may vary depending on the values. For example, index access is more effective for highly selective conditions, while a sequential scan is preferable when the value is common.

But the query can be planned without considering parameter values. This allows not only saving the parse tree but also the plan itself, avoiding the need for re-planning. Such a plan is referred to as generic (generic).

A generic plan performs well in cases of uniform distribution, where the condition's selectivity is not dependent on the specific value. But in the case of uneven distribution, a generic plan may perform well for some values but poorly for others.

Let's see how the planner decides which option to use.

## Private and General Plans

Let's prepare the query and create an index:

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

PREPARE

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

CREATE INDEX

The query for canceled flights will use an index because the statistics show there are few such flights:

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

QUERY PLAN

```
-----  
Index Scan using flights_status_idx on flights (cost=0.29..433.99 rows=415 width=63)  
  Index Cond: ((status)::text = 'Cancelled'::text)  
(2 rows)
```

However, the query for arriving flights does not use the index, as there are many of them.

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

QUERY PLAN

```
-----  
Seq Scan on flights (cost=0.00..5309.84 rows=198294 width=63)  
  Filter: ((status)::text = 'Arrived'::text)  
(2 rows)
```

These plans are called private because they are generated based on specific parameter values.

However, the planner also generates a general plan without considering specific parameter values. If at any point the cost of the general plan does not exceed the average cost of previously generated private plans, the planner switches to using the general plan, ceasing further planning. However, the first five times, private plans are used to accumulate statistics.

Let's run the statement three more times:

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

QUERY PLAN

```
-----  
Seq Scan on flights (cost=0.00..5309.84 rows=198294 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=198294 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=198294 width=63)  
  Filter: ((status)::text = 'Arrived'::text)  
(2 rows)
```

The number of executions for general and private plans can be found in the pg\_prepared\_statements view:

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

name	generic_plans	custom_plans
f	0	5

(1 row)

Next time, the planner will switch to a general plan. Instead of a specific value, the parameter number will be shown in the plan:

```
=> 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)

If you have a query with parameters displayed as numbers (e.g., from the server log), you can view the general execution plan for that query like this:

```
=> 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)

```

Switching to a general plan can be problematic in cases of uneven value distribution. The `plan_cache_mode` parameter allows you to disable private plans (or alternatively, use a general plan from the beginning):

```
=> 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=198294 width=63)
  Filter: ((status)::text = 'Arrived'::text)
(2 rows)

```

```
=> RESET plan_cache_mode;
```

```
RESET
```

## Partial index Partial Index

The search for already arrived flights will be performed via a sequential scan of the flights table, so the index entries with the Arrived key are never used. However, they consume space and require resources to stay in sync with table row changes.

We'll check the size of the full index on the status column:

```
=> 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

(1 row)

The index can only include row pointers that meet high-selectivity criteria.

Replace the full index with a partial one:

```
=> DROP INDEX flights_status_idx;
```

```
DROP INDEX
```

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

```
CREATE INDEX
```

The partial index is much smaller than the full index:

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

indexname	size
flights_status_idx	16 kB

(1 row)

A partial index is used when it's advantageous (for rare values):

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

QUERY PLAN

```
-----
Index Scan using flights_status_idx on flights (cost=0.15..437.85 rows=415 width=63)
  Index Cond: ((status)::text = 'Cancelled'::text)
(2 rows)
```

...and for frequent values — sequential scan:

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

QUERY PLAN

```
-----
Seq Scan on flights (cost=0.00..5309.84 rows=198294 width=63)
  Filter: ((status)::text = 'Arrived'::text)
(2 rows)
```

This way, you can save disk space and reduce the costs of maintaining the full index.

## Expression Index Expression-Based Index

If the conditions involve functions instead of columns, the planner doesn't consider the number of distinct values. For instance, flights in January would make up roughly 1/12 of the total:

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

count
16831

(1 row)

However, the planner doesn't recognize the purpose of the extract function and uses a fixed selectivity of 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)

```

This situation can be addressed by building an expression index, since these expressions have their own statistics collected.

```

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

```

CREATE INDEX

=> **ANALYZE flights;**

ANALYZE

The estimate is now correct.

```

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

```

## QUERY PLAN

```

-----
Bitmap Heap Scan on flights  (cost=312.70..3226.36 rows=16552 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..308.56 rows=16552 width=0)
        Index Cond: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE
'Europe/Moscow'::text)) = '1'::numeric)
(4 rows)

```

Expression index statistics are stored alongside the table's basic column statistics:

```

=> 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 | {8,9,10,3,5,6,12,1,7,4,11,2}
most_common_freqs |
{0.121133335,0.113866664,0.08033333,0.07833333,0.0782,0.0777,0.0776,0.07703333,0.07643333,
0.07603333,0.0732,0.070133336}

```

An index can only be created using a deterministic expression, meaning that for the same column values, it must consistently return the same result. See the "Functions" section for more details.

# Takeaways

Data characteristics are collected as statistics.

Statistics are used to estimate cardinality

Cardinality is used to estimate cost.

Cost is used to select the optimal plan

The key to success lies in accurate statistics and proper cardinality.



1. Create an index on the tickets table (tickets) by the passenger name (passenger\_name).
2. What statistics are available for this table?
3. Explain how cardinality is estimated and how the execution plan is selected for the following queries: a) selecting all tickets, b) selecting tickets by the name ALEKSANDR IVANOV, c) selecting tickets by the name ANNA VASILEVA, d) selecting a ticket by the identifier 0005432000284.

## Index

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

CREATE INDEX

## 2. Availability of Statistics

Some key values:

```
=> 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
-----+-----+-----+-----+-----+-----+
book_ref | 0 | -0.48178762 | | | {000184,02852C,0533D |
101 | -0.0059772413
contact_data | 0 | -1 | | | {"{"phone": "+700 |
101 | -0.010117838
ticket_no | 0 | -1 | | | {0005432000297,00054 |
101 | 0.9999999
passenger_id | 0 | -1 | | | {"0000 376168","0098 |
101 | 0.0040331036
passenger_name | 0 | 10324 | {"ALEKSANDR KUZNECOV | 100 | {"ADELINA ZHUKOVA"," |
101 | 0.0002467275
(5 rows)
```

- No column contains null values.
- There are approximately half as many unique booking IDs as there are rows in the table (meaning each booking has on average two tickets). There are approximately 10,000 distinct names. All other columns have unique values.
- The arrays of most common values and histograms are sized to match the default\_statistics\_target parameter (100).
- Passenger names have most common values. For other columns, these values are not applicable because the maximum number of tickets (5) is found in 194 bookings, and the remaining columns are unique.
- Histograms exist for all columns and are used to evaluate inequality predicates.
- The table rows are physically sorted by ticket number. Data in other columns is arranged in a somewhat random order.

## 3. Query Execution Plans

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

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

Cardinality is equal to the number of rows in the table; a full scan was selected.

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

#### QUERY PLAN

```
-----  
---  
Bitmap Heap Scan on tickets (cost=66.34..15690.14 rows=5408 width=104)  
  Recheck Cond: (passenger_name = 'ALEKSANDR IVANOV'::text)  
    -> Bitmap Index Scan on tickets_passenger_name_idx (cost=0.00..64.99 rows=5408  
width=0)  
      Index Cond: (passenger_name = 'ALEKSANDR IVANOV'::text)  
(4 rows)
```

Selectivity was determined using the most common values list; a null bitmap scan was chosen.

=> **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)
```

Selectivity was estimated using a uniform distribution; a bitmap scan was chosen.

=> **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)
```

Cardinality is 1 because the column's values are unique; an index scan was selected.