



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

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.

Functional dependency

Most frequent value combinations

Count of unique value combinations

Expression Statistics

Contains

- Multivariate statistics (across multiple columns)
- Expression Statistics

A database object Database object created manually

- resides in `pg_statistic_ext` and `pg_statistic_ext_data`
- the `pg_stats_ext` and `pg_stats_ext_exprs` views

Once created, the statistics are automatically collected.

The base statistics automatically collected may not be sufficient for accurate estimates of cardinality and selectivity.

PostgreSQL allows the database administrator to manually determine which additional, extended statistics are required. You can collect statistics that cover multiple columns (multivariate statistics) or statistics for arbitrary expressions.

Keep in mind that base statistics are automatically collected for tables and their columns, but not for indexes—except for expression indexes.

Therefore, an index built on multiple columns does not automatically result in the generation of multivariate statistics.

Extended statistics can be created using the `CREATE STATISTICS` command. Once the object is created, the corresponding statistics are automatically collected in the background or via the `ANALYZE` command.

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

The collected information is stored in the tables `pg_statistic_ext` and `pg_statistic_ext_data`; the statistics accessible to users are displayed in the views `pg_stats_ext` and `pg_stats_ext_exprs`.

Functional Dependency (Dependencies)

The value of one column defines the value of another column.
Statistics improves the estimation of condition selectivity.

city	index
Samara	443000
Kazan	420000
Nizhny Novgorod	603000
Veliky Novgorod	173000

There are several types of multivariate statistics (i.e., statistics for multiple table columns) that can be specified when creating an extended statistics object.

Functional dependency between columns shows how much the data in one column is determined by the value of another column.

In the example on the slide, the postal code clearly defines the city, so the selectivity of the condition `city = 'Samara'` and `index = '443000'` is determined by the selectivity of the `index = '443000'` predicate. Such predicates, whose selectivity cannot be calculated independently of each other, are called correlated.

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

Functional Dependencies

Let's look at a query with two conditions

```
=> SELECT count(*)
FROM flights
WHERE flight_no = 'PG0007' AND departure_airport = 'VK0';

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

The estimate proves to be significantly underestimated:

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

                                QUERY PLAN

-----
Bitmap Heap Scan on flights  (cost=12.13..1264.75 rows=26 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)
```

The issue arises because the planner assumes predicate independence and calculates the final selectivity as the product of individual predicate selectivities. For example, in the provided plan, the estimate in the Bitmap Index Scan node (flight_no condition) is accurate, but after filtering in the Bitmap Heap Scan node (departure_airport condition), it's significantly underestimated.

However, we know that the flight number uniquely identifies the departure airport: essentially, the second condition is redundant (assuming the airport is correctly specified).

This can be explained to the planner using functional dependency statistics:

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

CREATE STATISTICS

=> ANALYZE flights;

ANALYZE
```

The collected statistics are stored in the following format:

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

dependencies
-----
{"2 => 5": 1.000000, "5 => 2": 0.010700}
(1 row)
```

The attributes' ordinal numbers come first, with the dependency coefficient following the colon.

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

QUERY PLAN

```

-----
Bitmap Heap Scan on flights (cost=12.09..1225.67 rows=473 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..11.97
rows=473 width=0)
    Index Cond: (flight_no = 'PG0007'::bpchar)
(5 rows)

```

The estimate has now improved.

The \d command displays the extended statistics objects for a specific table:

=> \d flights

```

Table "bookings.flights"
  Column          |          Type          | Collation | Nullable |
Default
-----+-----+-----+-----+
flight_id         | integer                |           | not null |
nextval('flights_flight_id_seq'::regclass)
flight_no         | character(6)           |           | not null |
scheduled_departure | timestamp with time zone |           | not null |
scheduled_arrival  | timestamp with time zone |           | not null |
departure_airport  | character(3)           |           | not null |
arrival_airport    | character(3)           |           | not null |
status            | character varying(20)   |           | not null |
aircraft_code      | character(3)           |           | not null |
actual_departure   | timestamp with time zone |           |          |
actual_arrival     | timestamp with time zone |           |          |
Indexes:
    "flights_pkey" PRIMARY KEY, btree (flight_id)
    "flights_flight_no_scheduled_departure_key" UNIQUE CONSTRAINT, btree (flight_no,
scheduled_departure)
Check constraints:
    "flights_check" CHECK (scheduled_arrival > scheduled_departure)
    "flights_check1" CHECK (actual_arrival IS NULL OR actual_departure IS NOT NULL AND
actual_arrival IS NOT NULL AND actual_arrival > actual_departure)
    "flights_status_check" CHECK (status::text = ANY (ARRAY['On Time'::character
varying::text, 'Delayed'::character varying::text, 'Departed'::character varying::text,
'Arrived'::character varying::text, 'Scheduled'::character varying::text,
'Cancelled'::character varying::text]))
Foreign-key constraints:
    "flights_aircraft_code_fkey" FOREIGN KEY (aircraft_code) REFERENCES
aircrafts_data(aircraft_code)
    "flights_arrival_airport_fkey" FOREIGN KEY (arrival_airport) REFERENCES
airports_data(airport_code)
    "flights_departure_airport_fkey" FOREIGN KEY (departure_airport) REFERENCES
airports_data(airport_code)
Referenced by:
    TABLE "ticket_flights" CONSTRAINT "ticket_flights_flight_id_fkey" FOREIGN KEY
(flight_id) REFERENCES flights(flight_id)
Statistics objects:
    "bookings.flights_flight_no_departure_airport_stat" (dependencies) ON flight_no,
departure_airport FROM flights

```

The Statistics objects section lists the names, columns, and non-standard target values of the statistics objects.

Most Frequent Combinations

Most Common Value Combinations (MCV)

Similar to `pg_stats.most_common_vals/freqs`, but for multiple columns.
Statistics improves the estimation of condition selectivity.

city	river
Samara	Volga
Kazan	Volga
Nizhny Novgorod	Oka
Nizhny Novgorod	Volga
Veliky Novgorod	Volkhov

The list of the most common value combinations enables the storage of multiple value combinations and their frequencies.

The slide shows possible city-river pairs. It's clear that the predicates with the city and river columns are correlated, but unlike the previous example, neither column determines the other — there's a many-to-many relationship between cities and rivers. In this case, functional dependency statistics won't help improve the estimates.

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

Common Combinations

Columns may be correlated, but a direct functional dependency doesn't always exist. Let's run this query:

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

QUERY PLAN

```
-----
Gather  (cost=1000.00..5591.39 rows=715 width=63) (actual rows=5148 loops=1)
  Workers Planned: 1
  Workers Launched: 1
  -> Parallel Seq Scan on flights  (cost=0.00..4519.89 rows=421 width=63) (actual
rows=2574 loops=2)
    Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
    Rows Removed by Filter: 104860
(6 rows)
```

The planner's estimates are off by several times. Accounting for functional dependencies isn't enough to fix the situation:

```
=> CREATE STATISTICS (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..5696.09 rows=1762 width=63)
  Workers Planned: 1
  -> Parallel Seq Scan on flights  (cost=0.00..4519.89 rows=1036 width=63)
    Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
(4 rows)
```

In this case, you can add extended statistics for common combinations of values from multiple columns:

```
=> DROP STATISTICS flights_departure_airport_aircraft_code_stat;
```

DROP STATISTICS

```
=> CREATE STATISTICS (mcv)
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

```
-----
Seq Scan on flights  (cost=0.00..5847.00 rows=5429 width=63)
  Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
(2 rows)
```

The estimate has now improved.

The `default_statistics_target` parameter controls the number of most frequent values collected. You can configure it for individual statistics:


```
=> ALTER STATISTICS flights_departure_airport_aircraft_code_stat
SET STATISTICS 300;
```

```
ALTER STATISTICS
```

```
=> ANALYZE flights;
```

```
ANALYZE
```

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

```

              QUERY PLAN
-----
Seq Scan on flights  (cost=0.00..5847.00 rows=5088 width=63) (actual rows=5148 loops=1)
  Filter: ((departure_airport = 'LED'::bpchar) AND (aircraft_code = '321'::bpchar))
  Rows Removed by Filter: 209719
(3 rows)
```

The cardinality estimate has improved slightly.

However, this did not result in further plan improvements, so increasing the statistics target in this case is likely not justified.

You can view the statistics for the most frequent combinations as follows:

```
=> SELECT values, frequency
FROM pg_statistic_ext
  JOIN pg_statistic_ext_data ON oid = stxoid,
  pg_mcv_list_items(stxdmccv) m
WHERE stxname = 'flights_departure_airport_aircraft_code_stat'
LIMIT 10;
```

values	frequency
{SV0,SU9}	0.031422222222222222
{DME,SU9}	0.030533333333333332
{DME,CR2}	0.024888888888888887
{LED,321}	0.023677777777777777
{VK0,CR2}	0.021144444444444445
{BZK,SU9}	0.018966666666666666
{SV0,CR2}	0.018588888888888887
{KJA,CN1}	0.014477777777777777
{VK0,SU9}	0.013922222222222223
{OVB,CN1}	0.0131

(10 rows)

Unique Value Combinations

Count of Unique Value Combinations (ndistinct)

Similar to `pg_stats.ndistinct`, but for multiple columns
Statistics improves the estimation of cardinality for grouping.

city	region	index
Samara	Samara Region	443000
Kazan	Republic Tatarstan	420000
Nizhny Novgorod	Nizhny Novgorod Region	603000
Veliky Novgorod	Novgorod Region	173000

The number of unique value combinations enables more accurate cardinality estimates when grouping by multiple columns.

In the example on the slide, the number of possible combinations of all fields cannot be determined by simply multiplying the unique counts for each column.

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

Unique Combinations

Another scenario where the planner miscalculates the estimate arises with grouping. The number of airport pairs connected by direct flights is limited:

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

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

But the planner isn't aware of this:

```
=> 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)
```

Extended statistics can improve this estimate (if the type of statistics is not specified, all supported types will be included in the created object):

```
=> CREATE STATISTICS
ON departure_airport, arrival_airport FROM flights;

CREATE STATISTICS

=> ANALYZE flights;

ANALYZE

=> EXPLAIN
SELECT DISTINCT departure_airport, arrival_airport FROM flights;

               QUERY PLAN
```

```
-----
-----
Unique  (cost=5616.51..5621.15 rows=618 width=8)
  -> Sort  (cost=5616.51..5618.06 rows=618 width=8)
      Sort Key: departure_airport, arrival_airport
      -> Gather  (cost=5519.88..5587.86 rows=618 width=8)
          Workers Planned: 1
          -> HashAggregate  (cost=4519.88..4526.06 rows=618 width=8)
              Group Key: departure_airport, arrival_airport
              -> Parallel Seq Scan on flights (cost=0.00..3887.92 rows=126392
width=8)
(8 rows)
```

You can view the statistics for unique combinations as follows:

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

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

You can see the list of all extended statistics objects with the \dX command:

```
=> \x \dX \x
```

Expanded display is on.
List of extended statistics

-[RECORD 1]+	
Schema	bookings
Name	flights_departure_airport_aircraft_code_stat
Definition	departure_airport, aircraft_code FROM flights
Ndistinct	
Dependencies	
MCV	defined
-[RECORD 2]+	
Schema	bookings
Name	flights_departure_airport_arrival_airport_stat
Definition	departure_airport, arrival_airport FROM flights
Ndistinct	defined
Dependencies	defined
MCV	defined
-[RECORD 3]+	
Schema	bookings
Name	flights_flight_no_departure_airport_stat
Definition	flight_no, departure_airport FROM flights
Ndistinct	
Dependencies	defined
MCV	

Expanded display is off.

Statistics by type (Ndistinct, Dependencies, MCV) are displayed, with the actual values available in the pg_statistic_ext_data table.

Extended expression statistics

as if a generated column existed in the table

Statistics enhances the estimation of selectivity for conditions using expressions.

city	index	address
Samara	443000	443000, Samara
Kazan	420000	420000, Kazan
Nizhny Novgorod	603000	603000, Nizhny Novgorod
Veliky Novgorod	173000	173000, Veliky Novgorod City

Extended Statistics on Expressions extended Statistics on Expression enables the collection of all basic statistics that would be gathered if a column computed using this expression existed in the table.

If the predicate uses an expression instead of a column name on either side of the operator, the planner uses a fixed selectivity estimate. Using expression-based statistics, this issue can be resolved.

Note that expressions can also be used instead of column names in any type of multivariate statistics.

Expression Statistics

As discussed in the "Basic Statistics" section, if an expression is present in the condition, the planner, without selectivity information, defaults to a constant and frequently errs:

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

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

```
=> 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)
```

In the "Basic Statistics" section, we improved the situation by building an expression index, but this isn't always feasible. Additionally, indexes require storage space and ongoing maintenance. Another approach is to add expression statistics:

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

CREATE STATISTICS

```
=> ANALYZE flights;
```

ANALYZE

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

QUERY PLAN

```
-----
Seq Scan on flights  (cost=0.00..6384.17 rows=17046 width=63)
  Filter: (EXTRACT(month FROM (scheduled_departure AT TIME ZONE 'Europe/Moscow'::text))
= '1'::numeric)
(2 rows)
```

The evaluation is now correct.

Extended expression statistics are stored separately. Here are some columns:

```
=> SELECT statistics_name, expr, n_distinct, most_common_vals
FROM pg_stats_ext_exprs \gx
```

```
-[ RECORD 1
]-+-----
statistics_name | flights_expr_stat
expr            | EXTRACT(month FROM (scheduled_departure AT TIME ZONE
'Europe/Moscow'::text))
n_distinct      | 12
most_common_vals | {8,9,1,10,7,12,5,4,3,11,6,2}
```

Extended statistics assist in complex scenarios
Manually created and automatically maintained
May increase the costs of analysis and planning

1. Using the commands from the demo, create statistics such as dependencies, mcvc, and ndistinct for the flights table. Measure the execution time of the ANALYZE command. Remove the extended statistics, measure the execution time of the ANALYZE command once more, and compare it to the previous result.
2. Write a query to select all business class flights priced over 100,000 rubles. Will extended statistics help improve the cardinality estimate for the result? If so, which type of statistics is better to use?

1. 1. Measure the execution time multiple times and average the results to smooth out irregularities.

1. Cost of Analysis

Let's create extended statistics in the same way as demonstrated.

Functional Dependency Statistics

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

CREATE STATISTICS

Most frequent value combination lists:

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

CREATE STATISTICS

Statistics on Unique Value Combinations:

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

CREATE STATISTICS

Let's measure the analysis run time.

```
=> \timing on
```

Timing is on.

```
=> ANALYZE flights;
```

ANALYZE

Time: 890.684 ms

The initial analysis run may take significantly longer than usual.

```
=> ANALYZE flights;
```

ANALYZE

Time: 560.744 ms

```
=> ANALYZE flights;
```

ANALYZE

Time: 548.848 ms

```
=> \timing off
```

Timing is off.

Remove the created extended statistics:

```
=> DROP STATISTICS flights_dep;
```

DROP STATISTICS

```
=> DROP STATISTICS flights_mcv;
```

DROP STATISTICS

```
=> DROP STATISTICS flights_nd;
```

DROP STATISTICS

Let's measure the run time again:

```
=> \timing on
```

Timing is on.

```
=> ANALYZE flights;
```

ANALYZE

Time: 351.717 ms

```
=> ANALYZE flights;
```

ANALYZE

Time: 372.846 ms

```
=> \timing off
```

Timing is off.

Only three extended statistics were generated, yet the analysis time before and after deletion varied significantly. As the number of collected extended statistics increases, the analysis time also increases, which can result in higher server load.

Therefore, extended statistics should be used judiciously and only when needed.

2. Using Extended Statistics

Business class flights priced over 100,000 RUB:

```
=> EXPLAIN (analyze, timing off, summary off)
SELECT *
FROM ticket_flights
WHERE fare_conditions = 'Business' and amount > 100_000;
```

QUERY PLAN

```
-----
Gather  (cost=1000.00..124675.15 rows=12924 width=32) (actual rows=111203 loops=1)
  Workers Planned: 2
  Workers Launched: 2
    -> Parallel Seq Scan on ticket_flights  (cost=0.00..122382.75 rows=5385 width=32)
(actual rows=37068 loops=3)
      Filter: ((amount > '100000'::numeric) AND ((fare_conditions)::text =
'Business'::text))
      Rows Removed by Filter: 2760216
(6 rows)
```

The optimizer is off by an order of magnitude. The reason is that ticket price and service class are correlated, so extended statistics should improve the estimate. Since there's no direct functional dependency between the columns, we'll add statistics for the most frequent values:

```
=> CREATE STATISTICS (mcv) ON fare_conditions, amount FROM ticket_flights;
```

CREATE STATISTICS

```
=> ANALYZE ticket_flights;
```

ANALYZE

```
=> EXPLAIN (timing off, summary off)
SELECT *
FROM ticket_flights
WHERE fare_conditions = 'Business' and amount > 100_000;
```

QUERY PLAN

```
-----
Gather  (cost=1000.00..124587.55 rows=12048 width=32)
  Workers Planned: 2
    -> Parallel Seq Scan on ticket_flights  (cost=0.00..122382.75 rows=5020 width=32)
      Filter: ((amount > '100000'::numeric) AND ((fare_conditions)::text =
'Business'::text))
(4 rows)
```

The estimate has improved slightly, but it still differs significantly from the exact value. You can see that the proportion of rows meeting the condition is just over 1%:

```
=> SELECT count(*) FILTER (WHERE fare_conditions = 'Business' and amount > 100_000)
/ count(*)::float
FROM ticket_flights;
```

?column?

```
-----
0.01325130614791586
(1 row)
```

Because the default setting stores 100 most frequent pairs, it's likely that only 1-2 of them will meet the condition, making the optimizer's estimate inaccurate. To improve accuracy, let's increase the statistics volume:

```
=> ALTER STATISTICS ticket_flights_fare_conditions_amount_stat SET STATISTICS 500;
```

ALTER STATISTICS

```
=> ANALYZE ticket_flights;
```

ANALYZE

```
=> EXPLAIN (timing off, summary off)
SELECT *
FROM ticket_flights
WHERE fare_conditions = 'Business' and amount > 100_000;
```

QUERY PLAN

```
-----
-----
Gather  (cost=1000.00..133885.78 rows=105037 width=32)
  Workers Planned: 2
    -> Parallel Seq Scan on ticket_flights (cost=0.00..122382.08 rows=43765 width=32)
        Filter: ((amount > '100000'::numeric) AND ((fare_conditions)::text =
'Business'::text))
(4 rows)
```

The estimate is now nearly accurate.