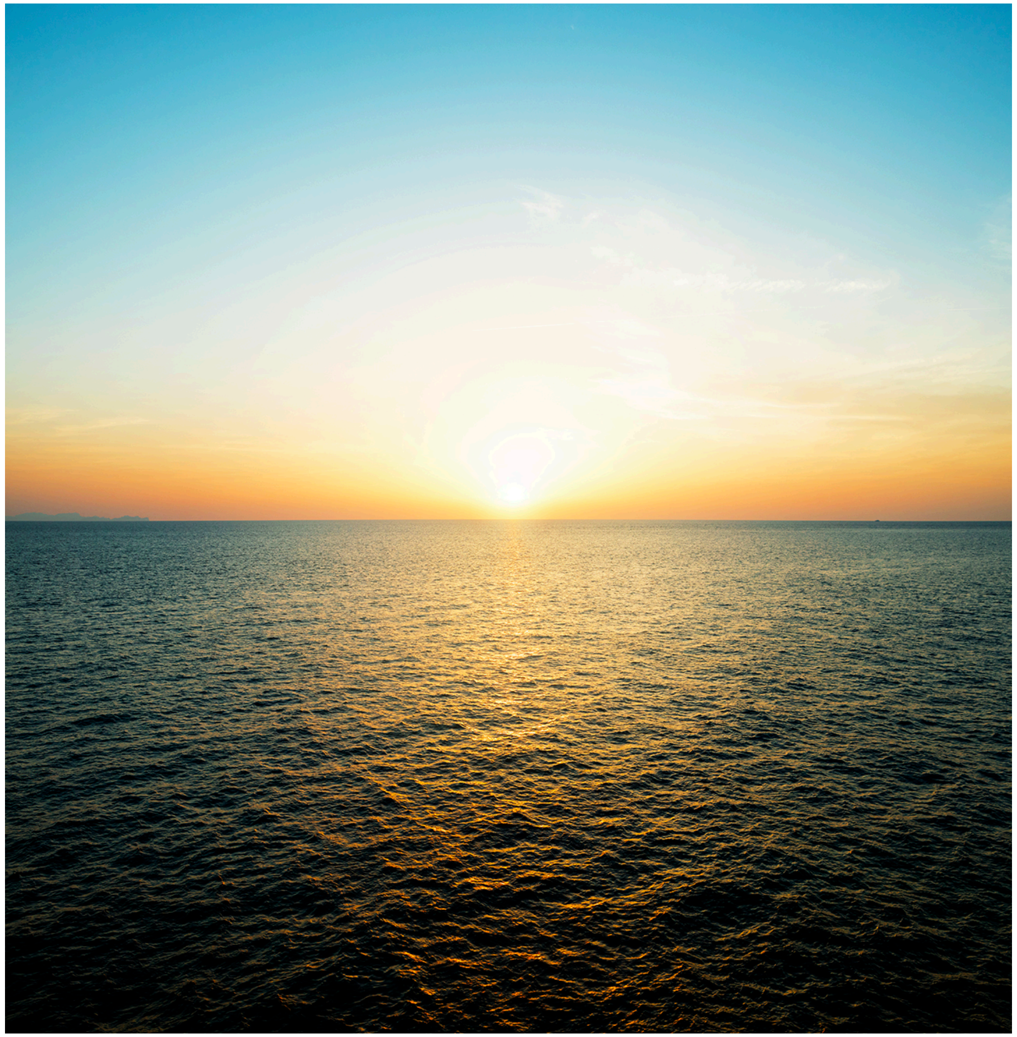
A Deep Dive into Statistics

Louise Leinweber

PGConfEU2024

About me

- Principal software engineer at Crunchy Data
- Working on a managed Postgres service: https://crunchybridge.com/
- Ruby, crystal, SQL, python are my everyday languages
- louisemeta.com
- Used to like climbing but then had a small human who takes time
- Also, used to like sleeping



Today’s agenda

1. Looking at your statistics
2. Statistics gathered by default
3. How the query optimizer uses them
4. Why are single column statistics not enough
5. Extended statistics
6. Configuration
7. How statistics are gathered
8. Conclusion

About the sample database

https://aact.ctti-clinicaltrials.org/

Database that includes all information (protocol, result data, conditions, etc.) for every study registered in ClinicalTrials.gov.

ClinicalTrials.gov:

- Information about studies is gathered by investigator
- Clinical research studies and their results
- New studies added almost every day
- Studies in 50 states and over 200 countries

Looking at your statistics

pg_stats View

schemaname	name
tablename	name
attname	name
inherited	boolean
null_frac	real
avg_width	integer
n_distinct	real
most_common_vals	anyarray
most_common_freqs	real[]
histogram_bounds	anyarray
correlation	real
most_common_elems	anyarray
most_common_elem_freqs	real[]
elem_count_histogram	real[]

@louisemeta

pg_stats View

```
SELECT * FROM pg_stats WHERE tablename = 'outcome_analyses' AND attname = 'ci_percent';
```

schemaname	ctgov
tablename	outcome_analyses
attname	ci_percent
most_common_values	{95.0,90.0,80.0,97.5,60.0,99.0,95.1,98.0,98.75,97.3}
most_common_freqs	{0.6150333,0.06973334,0.0089,0.006733333,0.0013333333,0.0010666667,0.0003,0.0003,0.0003,0.00026666667}
histogram_bounds	{-42.88,0.0,0.0,0.769,0.7763,0.95,0.95,0.95,0.95,0.975,2.0,2.0,5.0,5.0,10.0,20.0,65.0, 85.0,90.46,92.0,92.83,95.001,95.003,95.03,95.47,95.6,95.8,95.8,95.9,95.9,95.9,...}
...	...

@louisemeta

Statistics gathered by default

Statistics gathered

- **Most common values**
- **Histogram**
- **Distinct values:** estimation of the number of distinct values in the table
- **Average datum width:** calculated for types like text, json. Otherwise it's the a constant (int, uuid, etc)
- **Fraction of null values**
- **Correlation:** varies from -1 to 1. Describes the correlation between physical order of your tuples and values order of this column.

Most common values

Most common values are gathered from the table along with their distribution

schemaname	ctgov
tablename	conditions
attname	name
most_common_vals	{Healthy,"Breast Cancer",Obesity,"Prostate Cancer",Depression,Hypertension,"HIV Infections",Stroke,Pain,"Coronary Artery Disease",Asthma,Cancer,"Heart Failure","Diabetes Mellitus, Type 2","Colorectal Cancer","Atrial Fibrillation”,...}
most_common_freqs	{0.0111,0.009666666,0.008366667,0.0048,0.0046666665,0.0046,0.0045333332,0.0044333334,0.0042666667,0.0042333333,0.0041,0.004,0.0037666666,0.0036333334,0.0034666667,0.0033666666,0.0033666666,0.0033333334,0.0031666667,0.0030333332,0.003,0.0029333334,0.0029,0.0026,0.0026,0.0025333334,0.0025,0.0023666667,0.0023333333,0.0023333333,0.0023,0.0022,0.0021666666,0.0020666667,0.0019666667,0.0019333333,0.0019333333,0.0019,0.0019,...}

@louisemeta

Most common values

```
SELECT * FROM pg_stats WHERE tablename = 'studies' AND attname = 'study_type';
```

schemaname	ctgov
tablename	studies
attname	study_type
...	
n_distinct	3
most_common_vals	{INTERVENTIONAL,OBSERVATIONAL,EXPANDED_ACCESS}
most_common_freqs	{0.76283336,0.23366667,0.0017666667}

Most common values

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM conditions WHERE name = 'Healthy';
                               QUERY PLAN
-------------------------------------------------------------------------
 Aggregate  (cost=20794.70..20794.71 rows=1 width=8) (actual
time=57.273..57.274 rows=1 loops=1)
   ->  Seq Scan on conditions  (cost=0.00..20768.99 rows=10283
width=0) (actual time=0.013..57.020 rows=10040 loops=1)
         Filter: ((name)::text = 'Healthy'::text)
         Rows Removed by Filter: 871319
 Planning Time: 0.030 ms
 Execution Time: 57.287 ms
(6 rows)
```

@louisemeta

Distribution histogram

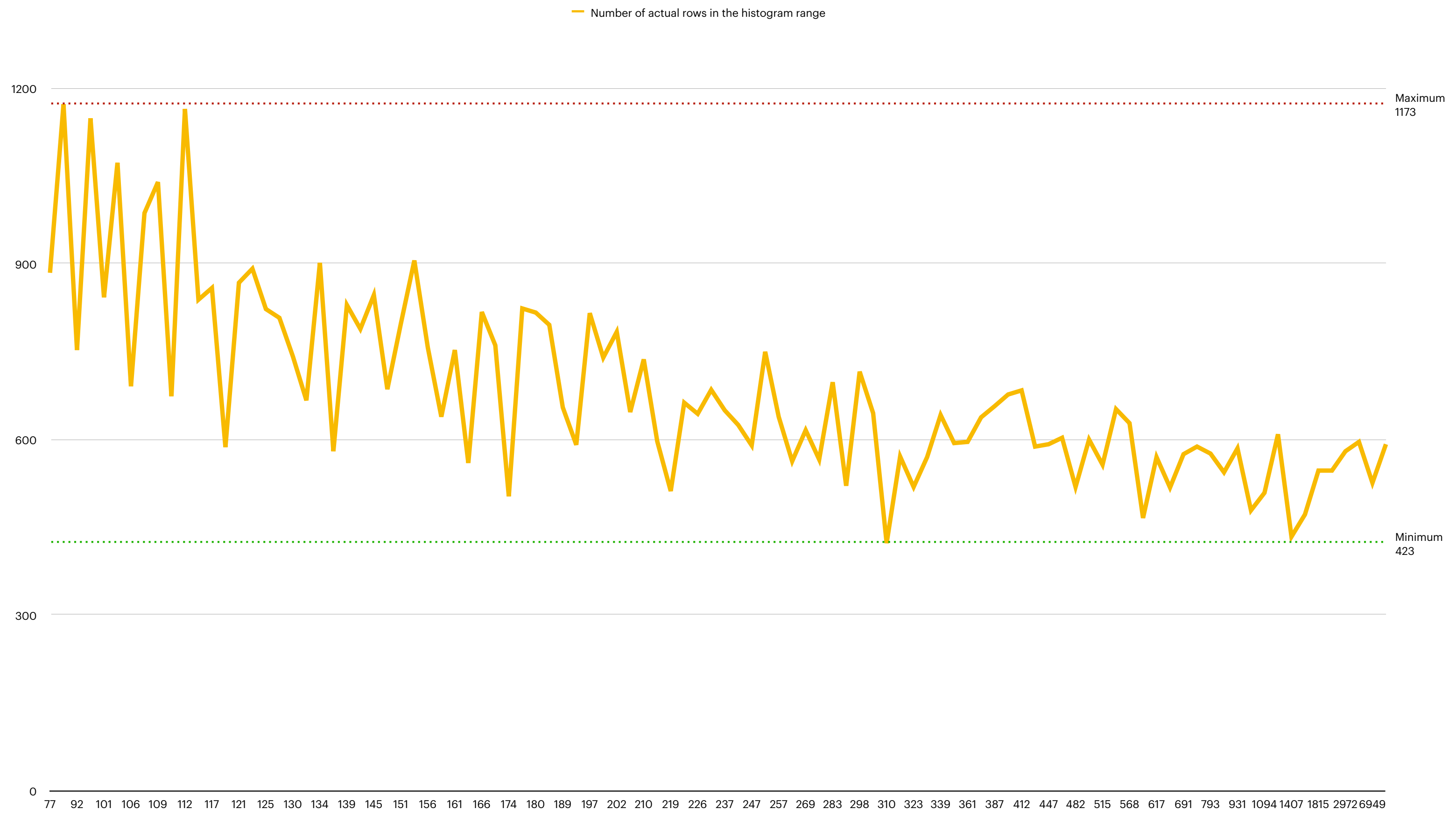
- Histogram describes the data distribution outside of the most common values
- Evenly distributed buckets
- Histogram are not computed when all values are listed in the MCV

tablename	baseline_counts
attname	count
histogram_bounds	{77,85,92,93,101,103,106,107,109,111,112,115,117,119,121,123,125,128,130,132,134,137,139,142,145,148,151,153,156,159,161,164,166,170,174,176,180,184,189,193,197,200,202,206,210,215,219,222,226,231,237,242,247,251,257,263,269,276,283,292,298,303,310,315,323,330,339,351,361,373,387,400,412,431,447,463,482,499,515,537,568,600,617,653,691,737,793,854,931,1012,1094,1214,1407,1571,1815,2235,2972,4107,6949,15044,985424}

Distribution histogram

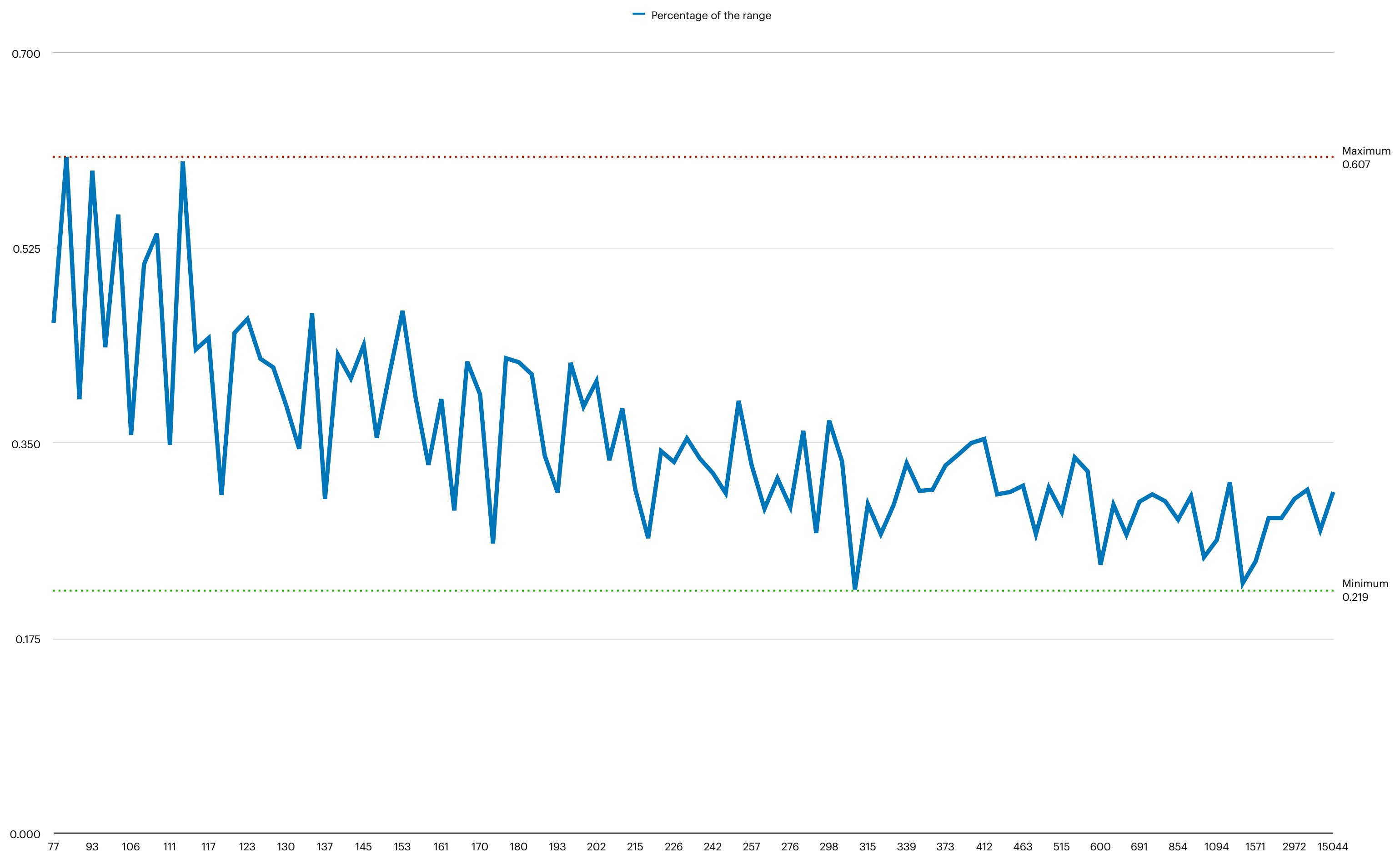
```
WITH
histogram AS (
   SELECT
      most_common_vals::text::int[],
      unnest(histogram_bounds::text::int[]) as value,
      generate_series(1, array_length(histogram_bounds, 1)) as id
   FROM pg_stats
   WHERE
   attname = 'count' AND tablename = 'baseline_counts'
),
bounds AS (
   SELECT
      h1.most_common_vals,
      h1.value as min,
      h2.value as max
   FROM histogram h1
   LEFT OUTER JOIN histogram h2 ON (h2.id = h1.id +1)
)
SELECT min, max, count(bc.*) as nb_rows, count(bc.*)/193212 AS percent
FROM bounds
LEFT JOIN baseline_counts bc ON bc.count BETWEEN min AND COALESCE(max, 2738162)
WHERE NOT (bc.count = ANY (most_common_vals))
GROUP BY(min, max)
ORDER BY min;
```

Distribution histogram



@louisemeta

Distribution histogram



@louisemeta

Distribution histogram

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM baseline_counts WHERE count = 78;
                              QUERY PLAN
----------------------------------------------------------------------
 Aggregate  (cost=4542.72..4542.73 rows=1 width=8) (actual
time=33.057..33.058 rows=1 loops=1)
   ->  Seq Scan on baseline_counts  (cost=0.00..4541.55 rows=468
width=0) (actual time=2.109..33.011 rows=504 loops=1)
         Filter: (count = 78)
         Rows Removed by Filter: 197060
 Planning Time: 0.153 ms
 Execution Time: 33.098 ms
(6 rows)
```

@louisemeta

Distribution histogram

A weird one

```sql
SELECT * FROM pg_stats WHERE tablename = 'outcome_analyses' AND attname = 'ci_percent';
```

schemaname	ctgov
tablename	outcome_analyses
attname	ci_percent
...	**...**
histogram_bounds	{**-42.88,**0.0,0.0,0.769,0.7763,0.95,0.95,0.95,0.95,0.975,2.0,2.0,5.0,5.0,10.0,20.0,65.0,85.0,90.46,92.0,92.83,95.001,95.003,95.03,95.47,95.6,95.8,95.8,95.9,95.9,95.9,...}

Distribution histogram

A confidence of -43% is not good.

Here's the study ClinicalTrials.gov ID that it's linked to: NCT01078675.

An Efficacy and 2-Year Safety Study of Open-label Rosuvastatin in Children and Adolescents (Aged From 6 to Less Than 18 Years) With Familial Hypercholesterolaemia

Conclusion: humans make mistakes.

How the query optimizer uses statistics

What are statistics for?

When you run a query, postgres generates different paths to execute the query.
It will pick the best one based on cost, and use it for the query plan.

Based on statistics, we estimate:

- Number of rows returned
- Size of the data returned
- Number of pages to scan

About selectivity

- Selectivity = % of rows returned after applying a filter.
- Estimated based on MCV, histogram, null fraction, etc.
- Helps choose a query plan.
  - If a where clause is filtering most rows, an index scan makes more sense
  - If a where clause is returning most rows, a sequential scan is preferred

Algorithms we'll look at today

`src/backend/utils/adt/selfuncs.c:`

- Calculating the selectivity of a WHERE column = constant
- Calculating the selectivity of a WHERE column (<,>,<=,>=) constant

`src/backend/optimizer/path/clausesel.c`

- Combining clauses with AND

`src/backend/optimizer/path/costsize.c`

- From selectivity to rows

The = clause

var_eq_const

- The value is in the MCV: we have the exact selectivity in stat numbers.
- The value isn't in the MCV
  - The selectivity is initialized with: 1 - (sum of MCV frequencies) - nullfrac
  - Calculate number of distinct values outside of the MCV
  - The selectivity then is: selectivity / other distincts.

This assumes the values outside of the MCV, are evenly distributed.

The = clause

Examples: value in MCV

```
SELECT * FROM facilities WHERE city = ‘Boston’;
```

Here are the stats on this column:

```
most_common_vals         | {"New York”,Seoul,Houston,Boston,…}
most_common_freqs        | {0.009933333,0.0088,0.008366667,0.008333334,…}
```

The selectivity will be 0.008333334

The = clause

var_eq_const

- The value is in the MCV: we have the exact selectivity in stat numbers.
- The value isn’t in the MCV
  - The selectivity is initialized with: 1 - (sum of MCV frequencies) - nullfrac
  - Calculate number of distinct values outside of the MCV
  - The selectivity then is: selectivity / other distincts.

This assumes the values outside of the MCV, are evenly distributed.

The = clause

Examples: value not in MCV

```
SELECT * FROM facilities
WHERE city = ‘Grenoble’;
```

Relevant statistics

```
null_frac                  | 6.666667e-05
n_distinct                 | 6655
```

Extra data we need to calculate selectivity:

```
SELECT
array_length(most_common_vals, 1),
(SELECT SUM(freqs) FROM
UNNEST(most_common_freqs) freqs )
FROM  pg_stats WHERE attname =
'city';
```

```
array_length | 100
sum          | 0.3193
```

selectivity = 1 - null_frac - (sum of MCV frequencies)

selectivity = 1 - 6.666667e-05 - 0.319 = **0.681**

other distincts = n_distinct - (number of MCV)

other_distincts = 6655 - 100 = **6555**

selectivity = selectivity/other_distincts

selectivity = 0.681/6555 = 0.0001038

(So something like **0.01%**)

The <, >, <=, >= operators

scalarineqsel

To be able to evaluate the selectivity of a where clause with one of these operators, we're going to look both at the MCV and at the histogram.

1. We retrieve the mcv_selectivity
2. We retrieve the histogram_selectivity
3. We combine these to get the selectivity

The <, >, <=, >= operators

mcv_selectivity

1. Initialize the selectivity to 0
2. Loop through mcv values:
    1. Try to apply the operator to the current value, if it matches, add it to the selectivity

The <, >, <=, >= operators

mcv_selectivity

```
SELECT * FROM outcome_analyses WHERE ci_percent >= 95;
```

Relevant statistics:

most_common_values	{95.0,90.0,80.0,97.5,60.0,99.0,95.1,98.0,98.75,97.3}
most_common_freqs	{0.6150333,0.06973334,0.0089,0.006733333,0.0013333333,0.0010666667,0.0003,0.0003,0.0003,0.00026666667}

Selectivity = 0.6150333 + 0.006733333 + 0.0010666667 + 0.0003 + 0.0003 + 0.0003 + 0.00026666667 = 0.62399996637 (around 62%)

The <, >, <=, >= operators

Histogram selectivity

To calculate the selectivity of a histogram, we look for the buckets matching the clause.

1. Initialize the number of match to 0
2. Loop through the histogram values

   Try to apply the operator to the value, if it matches, increment the match by 1
3. Return the selectivity: match/number of buckets in the histogram

The <, >, <=, >= operators

histogram_selectivity

```
SELECT * FROM outcome_analyses WHERE ci_percent >= 95;
```

Relevant statistics:

histogram_bounds	{-42.88,0.0,0.0,0.769,0.7763,0.95,0.95,0.95,0.95,0.975,2.0,2.0,5.0,5.0,10.0,20.0,65.0,85.0,90.46,92.0,92.83,95.001,95.003,95.03,95.47,95.6,95.8,95.8,95.9,95.9,95.9,96.0,96.0,96.39,96.7,97.0,97.0,97.4,97.47,97.47,97.51,97.6,98.25,98.25,98.3,98.3,98.33,98.33,98.34,98.4,98.4,98.6,98.77,99.1,99.1,99.2,99.4,99.5,99.6,99.7,99.8,99.8,99.8,99.875,985.0}

match = 44

number of buckets = 64

selectivity = 44/64 = 0.6875

The <, >, <=, >= operators

Calculating the selectivity

We have

- The sum of MCV selectivities (sumcommon)
- The fraction of null values (nullfrac)
- The MCV selectivity
- The histogram selectivity

The <, >, <=, >= operators

Calculating the selectivity

We're going to use this to figure out what's the selectivity, overall, for our clause.

1. Initialize selectivity:

   selec = 1.0 - nullfrac - sumcommon:

2. Merge the histogram selectivity:

   selec *= hist_selec

3. Merge the MCV selectivity:

   selec += mcv_select

   Reminder: we removed completely the MCV selectivities in the initialization, which is why we add it back here.

The <, >, <=, >= operators

Calculating the selectivity

```
SELECT * FROM outcome_analyses WHERE ci_percent >= 95;
```

- nullfrac = 0.292
- sumcommon = 0.704
- mcv_select = 0.624
- histogram_select = 0.6875

1. Initialize selectivity:

   selec = 1.0 - nullfrac - sumcommon = 0.0037

   Merge the histogram selectivity:

   selec *= hist_selec = 0.0037 * 0.6875 = 0.00254

   Merge the MCV selectivity:

   selec += mcv_select = 0.00254 + 0.624 = 0.627

@louisemeta

Handling several clauses
(Queries with AND)

`Function clauselist_selectivity in src/backend/optimizer/path/clausesel.c`

- We start with a selectivity of 1 (all rows would be returned) (s1)
- Loop through clauses
    - Compute the selectivity of each clause in isolation (s2)
    - Merge it to our original selectivity s1 = s1 * s2

Handling several clauses

Queries with AND

Merging by multiplying means that out of the 8.5% of studies in phase 1, 2% have diabetes in their title.

```
EXPLAIN ANALYZE SELECT * FROM studies WHERE phase = 'PHASE1' AND brief_title
ILIKE '%diabetes%';
                                    QUERY PLAN
--------------------------------------------------------------------
 Seq Scan on studies  (cost=0.00..43801.63 rows=434 width=1632) (actual
time=1.564..422.335 rows=616 loops=1)
   Filter: ((brief_title ~~* '%diabetes%'::text) AND ((phase)::text =
'PHASE1'::text))
   Rows Removed by Filter: 505441
 Planning Time: 0.150 ms
 Execution Time: 422.371 ms
(5 rows)
```

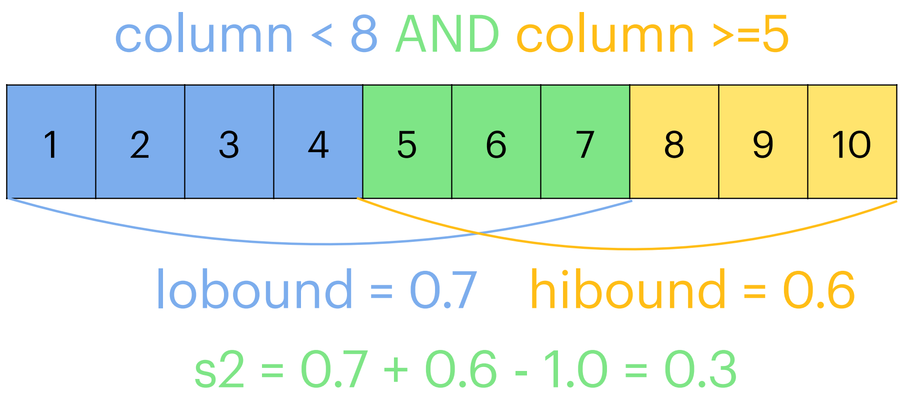
@louisemeta

Handling several clauses (Queries with AND)

Looking at function clauselist_selectivity in src/backend/optimizer/path/clausesel.c

If your query has a range with a low and high bound, we need to calculate the selectivity of the **overlap**

s2 = rqlist->hibound + rqlist->lobound - (1.0 - nullstats)



Handling several clauses
Example

```sql
SELECT nct_id,
official_title,
is_fda_regulated_drug
FROM studies
WHERE
completion_date > '2023-12-31'
AND completion_date < '2025-01-01'
AND study_type = 'INTERVENTIONAL';
```

Selectivity for:

study_type = ‘INTERVENTIONAL’: **0.76283336**

completion_date > ‘2023-12-31’:

selec = (1.0 - 0.031366665 - 0.32790005) = 0.640733285

histogram_selectivity = 0.23

mcv_select = 0.0849668

select = 0.640733285 * 0.23 + 0.0849668 = **0.23233545555**

completion_date< ‘2025-01-01':

selec = (1.0 - 0.031366665 - 0.32790005) = 0.640733285

histogram_selectivity = 0.85

mcv_select = 0.2787

select = 0.640733285 * 0.85 + 0.2787 = **0.82348989225**

Handling several clauses
Example

```
SELECT nct_id,
official_title,
is_fda_regulated_drug
FROM studies
WHERE
completion_date > '2023-12-31'
AND completion_date < '2025-01-01'
AND study_type = 'INTERVENTIONAL';
```

study_type = ‘INTERVENTIONAL’: **0.76283336**

completion_date > ‘2023-12-31’: **0.23233545555**

completion_date < ‘2025-01-01': **0.82348989225**

Nullfrac: **0.031366665**

Query selectivity = 0.76283336 * (0.23233545555 + 0.82348989225 - (1 - 0.031366665)) **= 0.06651297609**

Calculation of cost

Number of rows

```
EXPLAIN SELECT nct_id, official_title, is_fda_regulated_drug
FROM studies
WHERE
completion_date > '2023-12-31'
AND completion_date < '2025-01-01'
AND study_type = 'INTERVENTIONAL';

                                        QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..44260.65 rows=33672 width=151)
   Workers Planned: 2
   ->  Parallel Seq Scan on studies  (cost=0.00..39899.35 rows=14005 width=151)
         Filter: ((completion_date > '2023-12-31'::date) AND (completion_date <
'2025-01-01'::date) AND ((study_type)::text = 'INTERVENTIONAL'::text))
(4 rows)
```

Calculation of cost

Number of rows

rows=33672

```
SELECT reltuples AS estimate FROM pg_class WHERE relname =
'studies';
estimate
----------
506242
```

rows = selectivity * reltuples = **0.06651297609 * 506242 = 33671.6620417538**

@louisemeta

Calculation of cost

Number of rows

Query plan for EXPLAIN ANALYZE:

```
                                          QUERY PLAN
------------------------------------------------------------------------------------
 Gather  (cost=1000.00..44260.65 rows=33672 width=151) (actual time=1.688..351.477
rows=32686 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on studies  (cost=0.00..39899.35 rows=14005 width=151) (actual
time=0.486..344.455 rows=10895 loops=3)
         Filter: ((completion_date > '2023-12-31'::date) AND (completion_date <
'2025-01-01'::date) AND ((study_type)::text = 'INTERVENTIONAL'::text))
         Rows Removed by Filter: 157790
 Planning Time: 0.055 ms
 Execution Time: 352.224 ms
(8 rows)
```

Calculation of cost

If you want to know more about costs:

`src/backend/optimizer/path/costsize.c`

Why are single column statistics not enough

Combination of selectivity

```
EXPLAIN SELECT nct_id, name FROM facilities WHERE city = 'Lyon' AND country =
'France';
                                   QUERY PLAN
--------------------------------------------------------------------------------
 Index Scan using index_facilities_on_city on facilities  (cost=0.43..24535.07
rows=297 width=47)
   Index Cond: ((city)::text = 'Lyon'::text)
   Filter: ((country)::text = 'France'::text)
```

Combination of selectivity

rows=297

Selectivity France: 0.060533334 (6%)
Selectivity Lyon: 0.0015666666 (0.16%)
Reltuples: 3132540

The selectivity is combined by assuming that out of the 0.16% of cities named Lyon, only 6% will be in France.

3132540 * 0.060533334 * 0.0015666666 = 297

EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT nct_id, name FROM facilities WHERE city =
'Lyon' AND country = 'France';

                               QUERY PLAN
--------------------------------------------------------------------
 Index Scan using index_facilities_on_city on facilities
(cost=0.43..24535.07 rows=297 width=47) (actual
time=0.072..6.770 rows=5605 loops=1)
   Index Cond: ((city)::text = 'Lyon'::text)
   Filter: ((country)::text = 'France'::text)
 Planning Time: 0.206 ms
 Execution Time: 7.090 ms
```

EXPLAIN ANALYZE

The expected and actual number of rows are widely different

```
(cost=0.43..1172.06 rows=297 width=47) (actual time=2.248..84.619
rows=5605 loops=1)
```

The issue is that, potentially, it's choosing the wrong query plan!

EXPLAIN ANALYZE

```
CREATE STATISTICS (dependencies) ON country, city FROM facilities;
ANALYZE facilities;

EXPLAIN ANALYZE SELECT nct_id, name FROM facilities WHERE city = 'Lyon' AND
country = 'France';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using index_facilities_on_city on facilities  (cost=0.43..24146.29
rows=5747 width=47) (actual time=0.050..5.021 rows=5605 loops=1)
   Index Cond: ((city)::text = 'Lyon'::text)
   Filter: ((country)::text = 'France'::text)
 Planning Time: 0.151 ms
```

@louisemeta

Types of multicolumn statistics

Spooky query

```
create or replace function mycountingthing(tname text, colname1 text, val1 text, colname2 text, val2 text) returns integer AS $$
DECLARE
        c int;
begin
EXECUTE 'SELECT COUNT(*) FROM '|| tname || ' WHERE ' || colname1 ||'::text = $1 AND '|| colname2 ||'::text = $2'
INTO c
USING val1, val2;
RETURN c;
end;
$$ language plpgsql;

WITH fields AS (
SELECT
tablename,
attname,
unnest(most_common_vals::text::text[]) as val,
unnest(most_common_freqs::text::float[]) as freq,
null_frac,
reltuples,
(SELECT SUM(freqs) FROM UNNEST(most_common_freqs) freqs ) total_freqs
FROM pg_stats
INNER JOIN pg_class ON (relname = tablename)
WHERE schemaname  = 'ctgov'
AND most_common_vals IS NOT NULL
AND reltuples > 100000
),
eligible_fields AS (
        SELECT * FROM fields WHERE total_freqs > 0.8
),
counts_combine AS (
SELECT
f.tablename,
f.attname att1,
f.val,
e.attname att2,
e.val,
f.freq * e.freq * e.reltuples as expected_rows,
mycountingthing(f.tablename, f.attname, f.val, e.attname, e.val) as actual_rows
FROM eligible_fields e
INNER JOIN fields f ON (f.tablename = e.tablename AND f.attname <> e.attname)
WHERE f.freq * e.freq * e.reltuples > 100
AND mycountingthing(f.tablename, f.attname, f.val, e.attname, e.val) > 0
)

SELECT DISTINCT(tablename, att1, att2)
FROM counts_combine
WHERE actual_rows > 0 AND expected_rows < actual_rows * 0.2;
```

CREATE STATISTICS

The goal of CREATE STATISTICS is to mitigate the case we just described.

You can manually force postgres to link two columns. There are three types of statistics you can create:

- Functional dependencies
- Multivariate N-Distinct Count
- Multivariate MCV lists

Dependencies

Describes a dependency between two columns:

- Country and city that I just showed
- The values of two columns vary together (column a = column b + 1)

```
CREATE STATISTICS (dependencies) on country, city FROM facilities;
```

Multivariate ndistinct count

For each column, you have an ndistinct.

During GROUP BY, calculating the number of distinct groups can be wrong when the columns are linked.

To improve that you can do:

```
CREATE STATISTICS (ndistinct) on category, title FROM
baseline_measurements;
```

NDistinct

In baseline_measurements, the category and the title have a dependency.

Here are examples of the values that might make you understand why:

```
("Sex: Female, Male",Female)                                             | 183232
("Sex: Female, Male",Male)                                               | 183229
("Race (NIH/OMB)",Asian)                                                 |  69469
("Race (NIH/OMB)",White)                                                 |  69465
("Race (NIH/OMB)","American Indian or Alaska Native")                    |  69460
("Race (NIH/OMB)","Black or African American")                           |  69448
("Race (NIH/OMB)","More than one race")                                  |  69445
("Race (NIH/OMB)","Unknown or Not Reported")                             |  69442
("Race (NIH/OMB)","Native Hawaiian or Other Pacific Islander")           |  69436
```

NDistinct

```sql
SELECT category, title, SUM(number_analyzed)
FROM baseline_measurements
WHERE category IS NOT NULL
GROUP BY category, title
ORDER BY 3 DESC
LIMIT 10;
```

Before CREATE STATISTICS

Time: 872.956 ms

After CREATE STATISTICS

Time: 335.421 ms

NDistinct

Before CREATE STATISTICS

```
                                              QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Limit  (cost=211176.28..211176.31 rows=10 width=44)
   ->  Sort  (cost=211176.28..211753.87 rows=231037 width=44)
         Sort Key: (sum(number_analyzed)) DESC
         ->  Finalize GroupAggregate  (cost=138745.01..206183.66 rows=231037 width=44)
               Group Key: category, title
               ->  Gather Merge  (cost=138745.01..200407.73 rows=462074 width=44)
                     Workers Planned: 2
                     ->  Partial GroupAggregate  (cost=137744.98..146072.90 rows=231037 width=44)
                           Group Key: category, title
                           ->  Sort  (cost=137744.98..139249.37 rows=601755 width=40)
                                 Sort Key: category, title
                                 ->  Parallel Seq Scan on baseline_measurements  (cost=0.00..63523.06
rows=601755 width=40)
                                       Filter: (category IS NOT NULL)
(13 rows)
```

NDistinct

After CREATE STATISTICS

```
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Limit  (cost=71144.62..71144.64 rows=10 width=44)
   ->  Sort  (cost=71144.62..71159.58 rows=5985 width=44)
         Sort Key: (sum(number_analyzed)) DESC
         ->  Finalize GroupAggregate  (cost=69469.06..71015.28 rows=5985 width=44)
               Group Key: category, title
               ->  Gather Merge  (cost=69469.06..70865.66 rows=11970 width=44)
                     Workers Planned: 2
                     ->  Sort  (cost=68469.04..68484.00 rows=5985 width=44)
                           Sort Key: category, title
                           ->  Partial HashAggregate  (cost=68033.71..68093.56 rows=5985 width=44)
                                 Group Key: category, title
                                 ->  Parallel Seq Scan on baseline_measurements
(cost=0.00..63517.75 rows=602129 width=40)
                                       Filter: (category IS NOT NULL)
(13 rows)
```

@louisemeta

NDistinct

Comparing Query Plans

```
->  Gather Merge
(cost=138745.01..200407.73
rows=462074 width=44)
Workers Planned: 2
  ->  Partial GroupAggregate
  (cost=137744.98..146072.90
  rows=231037 width=44)
  Group Key: category, title
    ->  Sort
    (cost=137744.98..139249.37
    rows=601755 width=40)
    Sort Key: category, title
```

```
->  Gather Merge
(cost=69469.06..70865.66
rows=11970 width=44)
Workers Planned: 2
  ->  Sort
  (cost=68469.04..68484.00
  rows=5985 width=44)
  Sort Key: category, title
      ->  Partial HashAggregate
      (cost=68033.71..68093.56
      rows=5985 width=44)
      Group Key: category, title
```

Multivariate MCV list

This is the same idea than MCVs, but for more than one column.

It will aggregate the frequency of combined columns.

The difference with functional dependencies is that MCV list support other operators like <,>,<=,>=.

```
CREATE STATISTICS (mcv) on organ_system, adverse_event_term FROM
reported_events;
```

Multivariate MCV list

```
                            organ_system                             | adverse_event_term
-------------------------------------------------------------------------------------------
 Gastrointestinal disorders                                          | Nausea
 Nervous system disorders                                            | Headache
 Gastrointestinal disorders                                          | Vomiting
 General disorders                                                   | Fatigue
 Gastrointestinal disorders                                          | Diarrhoea
 Nervous system disorders                                            | Dizziness
 Gastrointestinal disorders                                          | Constipation
 General disorders                                                   | Pyrexia
 Gastrointestinal disorders                                          | Abdominal pain
 Musculoskeletal and connective tissue disorders                     | Back pain
 Respiratory, thoracic and mediastinal disorders                     | Cough
```

Multivariate MCV List

Before and After CREATE STATISTICS

```
EXPLAIN ANALYZE
SELECT nct_id, organ_system, adverse_event_term,
frequency_threshold
FROM reported_events
WHERE organ_system = 'Respiratory, thoracic and mediastinal
disorders'
AND adverse_event_term = 'Hypoxia'
ORDER BY frequency_threshold DESC
LIMIT 10;
```

Multivariate MCV List

Before and after CREATE STATISTICS

```
Before

Index Scan using reported_events_organ_system_adverse_event_term_idx on reported_events
(cost=0.56..4944.66 rows=1232 width=63) (actual time=0.042..14.498 rows=17057
loops=1
)


After

Index Scan using reported_events_organ_system_adverse_event_term_idx on reported_events
(cost=0.56..57259.44 rows=14453 width=63) (actual time=0.112..16.302
rows=17057 loops
=1)
```

@louisemeta

Extended Statistics Limitations

Because histograms aren’t supported in extended statistics, they are only accurate:

- For MCVs
- If the rest of your dataset is evenly distributed

Extended Statistics Limitations

The same query that I just ran, with a different adverse event, will have, again, a inaccurate selectivity:

```
EXPLAIN ANALYZE SELECT nct_id, organ_system, adverse_event_term, frequency_threshold FROM
reported_events WHERE organ_system = 'Respiratory, thoracic and mediastinal disorders' AND
adverse_event_term = 'Asthma' ORDER BY frequency_threshold DESC LIMIT 10;
                                             QUERY PLAN
----------------------------------------------------------------------------------------------------
 Limit  (cost=230.33..230.35 rows=10 width=63) (actual time=83.550..83.553 rows=10 loops=1)
   ->  Sort  (cost=230.33..230.47 rows=56 width=63) (actual time=83.546..83.547 rows=10 loops=1)
         Sort Key: frequency_threshold DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Index Scan using reported_events_organ_system_adverse_event_term_idx on reported_events
(cost=0.56..229.12 rows=56 width=63) (actual time=1.252..81.977 rows=11889 loops=1)
               Index Cond: (((organ_system)::text = 'Respiratory, thoracic and mediastinal
disorders'::text) AND ((adverse_event_term)::text = 'Asthma'::text))
 Planning Time: 0.433 ms
 Execution Time: 83.625 ms
```

Configuration

Configurations

- `default_statistics_target`: default is 100, can go from 1 to 10000.
- `ALTER TABLE reported_events ALTER COLUMN organ_system SET STATISTICS 1000`;
- `ALTER TABLE foo SET (n_distinct = value)`:
  - Positive value = exact nb of distinct
  - Negative value = percentage of the overall rows: -1 = each value is unique, -0.5: each value appears twice
- Filtering columns: `ANALYZE foo (bar);`

Configurations

```
EXPLAIN ANALYZE SELECT nct_id, organ_system, adverse_event_term, frequency_threshold FROM
reported_events WHERE organ_system = 'Respiratory, thoracic and mediastinal disorders' AND
adverse_event_term = 'Asthma' ORDER BY frequency_threshold DESC LIMIT 10;
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------
 Limit  (cost=52487.38..52487.41 rows=10 width=63) (actual time=16.687..16.690 rows=10 loops=1)
   ->  Sort  (cost=52487.38..52520.29 rows=13161 width=63) (actual time=16.685..16.687 rows=10
loops=1)
         Sort Key: frequency_threshold DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Index Scan using reported_events_organ_system_adverse_event_term_idx on
reported_events  (cost=0.56..52202.98 rows=13161 width=63) (actual time=0.093..14.569
rows=11889 loops
=1)
               Index Cond: (((organ_system)::text = 'Respiratory, thoracic and mediastinal
disorders'::text) AND ((adverse_event_term)::text = 'Asthma'::text))
 Planning Time: 3.293 ms
 Execution Time: 16.725 ms
```

@louisemeta

Configurations

```
EXPLAIN ANALYZE SELECT nct_id, organ_system, adverse_event_term, frequency_threshold FROM
reported_events WHERE organ_system = 'Respiratory, thoracic and mediastinal disorders' AND
adverse_event_term = 'Hypercapnia' ORDER BY frequency_threshold DESC LIMIT 10;
                                        QUERY PLAN
---------------------------------------------------------------------------------------------
 Limit  (cost=36.77..36.79 rows=8 width=63) (actual time=3.175..3.177 rows=10 loops=1)
   ->  Sort  (cost=36.77..36.79 rows=8 width=63) (actual time=3.174..3.175 rows=10 loops=1)
         Sort Key: frequency_threshold DESC
         Sort Method: top-N heapsort  Memory: 26kB
         ->  Index Scan using reported_events_organ_system_adverse_event_term_idx on
reported_events  (cost=0.56..36.65 rows=8 width=63) (actual time=0.354..3.011
rows=718 loops=1)
               Index Cond: (((organ_system)::text = 'Respiratory, thoracic and mediastinal
disorders'::text) AND ((adverse_event_term)::text = 'Hypercapnia'::text))
 Planning Time: 1.528 ms
 Execution Time: 3.218 ms
```

@louisemeta

Configurations

- `default_statistics_target`: default is 100, can go from 1 to 10000.
- `ALTER TABLE reported_events ALTER COLUMN organ_system SET STATISTICS 1000;`
- `ALTER TABLE foo SET (n_distinct = value)`:
  - Positive value = exact nb of distinct
  - Negative value = percentage of the overall rows: -1 = each value is unique, -0.5: each value appears twice
- Filtering columns: `ANALYZE foo (bar);`

How statistics are gathered

Overview of the process

`src/backend/commands/analyze.c`

While analyzing a table, postgres will gather statistics on that table. Here are the steps:

1. Gathering the sample rows
2. Computing statistics on those sample rows according to the data type
3. Inserting/Updating into the pg_statistics table

Gathering the sample rows

`src/backend/utils/misc/sampling.c`

The number of rows is based on the number on how many values we want to compute (default_statistics_target parameter).

Number of rows by default: 300*100.

Postgres will go through the table and uses a reservoir sampling algorithm (Vitter)

1. Initialize list of sample rows
2. Scan rows until the list is full
3. Each new row has (to extremely simplify) a probability of 1/number of rows to be selected.
4. If row is selected, replace a random row in the existing list

Gathering the sample rows

`src/backend/utils/misc/sampling.c`

The number of rows is based on the number on how many values we want to compute (default_statistics_target parameter).

Number of rows by default: 300*100.

Postgres will go through the table and uses a reservoir sampling algorithm (Vitter)

1. Initialize list of sample rows
2. Scan rows until the list is full
3. Each new row has (to extremely simplify) a probability of 1/number of rows to be selected.
4. If row is selected, replace a random row in the existing list

Computing statistics

Trivial, distinct of scalar ?

`src/backend/commands/analyze.c`

	Fraction of non-null rows	Average datum width	MCV	Number of distinct values	Histogram	Correlation	Operators
compute_trivial_stats	X	X					No =
compute_distinct_stats	X	X	X	X			Only =
compute_scalar_stats	X	X	X	X	X	X	<,>,<=,>=,=

Computing statistics

Most Common Values

1. Initialize track, a list of previously seen values and their counter
2. Loop through the sample rows
    1. If value hasn't been seen before: insert it after the last item that has a count greater than 1, the items after (that all had also 1 as a value), are bumped down the track list, potentially dropping off older values.
    2. Otherwise we increment the counter, and potentially bump it up the list

Value						
Counter						

Computing statistics

Most Common Values

1. Initialize track, a list of previously seen values and their counter
2. Loop through the sample rows
    1. If value hasn’t been seen before: insert it after the last item that has a count greater than 1, the items after (that all had also 1 as a value), are bumped down the track list, potentially dropping off older values.
    2. Otherwise we increment the counter, and potentially bump it up the list

Value	1	5	8	3	4	6
Counter	4	3	3	2	1	1

New value 2

@louisemeta

Computing statistics

Most Common Values

1. Initialize track, a list of previously seen values and their counter
2. Loop through the sample rows
    1. If value hasn't been seen before: insert it after the last item that has a count greater than 1, the items after (that all had also 1 as a value), are bumped down the track list, potentially dropping off older values.
    2. Otherwise we increment the counter, and potentially bump it up the list

Value	1	5	8	3	2	4
Counter	4	3	3	2	1	1

Computing statistics

Most Common Values

1. Initialize track, a list of previously seen values and their counter
2. Loop through the sample rows
   1. If value hasn't been seen before: insert it after the last item that has a count greater than 1, the items after (that all had also 1 as a value), are bumped down the track list, potentially dropping off older values.
   2. Otherwise we increment the counter, and potentially bump it up the list

Value	1	5	8	3	2	4
Counter	4	3	3	2	1	1

$\uparrow$ Found another 8

@louisemeta

Computing statistics

Most Common Values

1. Initialize track, a list of previously seen values and their counter
2. Loop through the sample rows
    1. If value hasn't been seen before: insert it after the last item that has a count greater than 1, the items after (that all had also 1 as a value), are bumped down the track list, potentially dropping off older values.
    2. Otherwise we increment the counter, and potentially bump it up the list

Value	1	8	5	3	2	4
Counter	4	4	3	2	1	1

Computing statistics

Histogram

Compute scalar stats algorithm:

1. Compute a track list of all values in the sample rows
2. The values that are significantly more common go into MCV
3. The leftover values are ordered by value and split into buckets
    1. Nb buckets = number of distincts values - number of MCV
    2. Pick values from the list every X items for the histogram bounds:

       X = number of left items in the track list / number of buckets

Computing statistics

Histogram

Value	1	8	5	3	2	4	100	7	12	10	13	6	9	11
Count	50	49	48	45	20	18	17	2	1	1	1	1	1	1

MCV = (1, 8, 5, 3)

Nb distinct = 30

Nb buckets = 26

Items left = 71

Buckets size = 71/26 = 2.7.

2	4	6	7	9	10	11	12	13	100
20	18	1	2	1	1	1	1	1	17

Histogram: (2, 2, … 4, 4, **9, 11, 13, 100,** 100…)

Computing statistics

Number of distinct values

Limited number of distinct values:

If you track list includes every value from our sample rows, n_distinct = nb of value seen in the sample

Otherwise, postgres is using an estimator proposed by Haas and Stokes.

$$\underbrace{n}_{\text{Number of non null rows in the sample}} * \underbrace{d}_{\text{Number of values seen in sample}} / (n - f1 + \underbrace{f1}_{\text{Number of distinct values in sample}} * n / \underbrace{N}_{\text{Estimated number of non null rows}})$$

Computing statistics

Number of distinct values

Let’s say I had 100 sample rows: 20 were nulls

Let’s say I had 15 distinct values, and 10 multiple ones in my track list

n = 80 non null

d = 25 values

f1 = 15 distinct values

N = 1200 * (1 - 0.2) = 1000

Value	1	8	5	3	2	4	28	12	13	23
Count	9	8	7	6	5	5	3	2	2	2

$$n*d / (n - f1 + f1*n/N)$$

$$80*25/(80-15+15*80/1000) = 30.21111$$

@louisemeta

Conclusion

@louisemeta

What we learned today

- Postgres gathers single column statistics during ANALYZE
- The query optimizer uses those statistics to choose a query plan
- Selectivity is the % of rows that a clause would return
- Postgres merges selectivities by assuming that columns are unrelated which is not always true
- Histogram and MCVs are computed by brute force, your ANALYZE is impacted by choosing a higher target value
- A higher target value also means a slower query optimizer as it has to loop through more elements in the MCV and histogram
- BUT a higher value might lead to a better query plan, therefor a faster query execution
- CREATE STATISTICS can help the query planner to choose a better plan if your columns have a relationship between them.
- But multivariate statistics also have limitations

Questions ?

(Come and see me, there's no way I finish this talk with enough time for questions)

@louisemeta