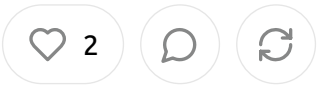
Statistics: How PostgreSQL Counts Without Counting

Part 2: Into the trenches of PostgreSQL statistics

TRADE REPUBLIC ENGINEERING AND SADEQ DOUSTI

FEB 17, 2025



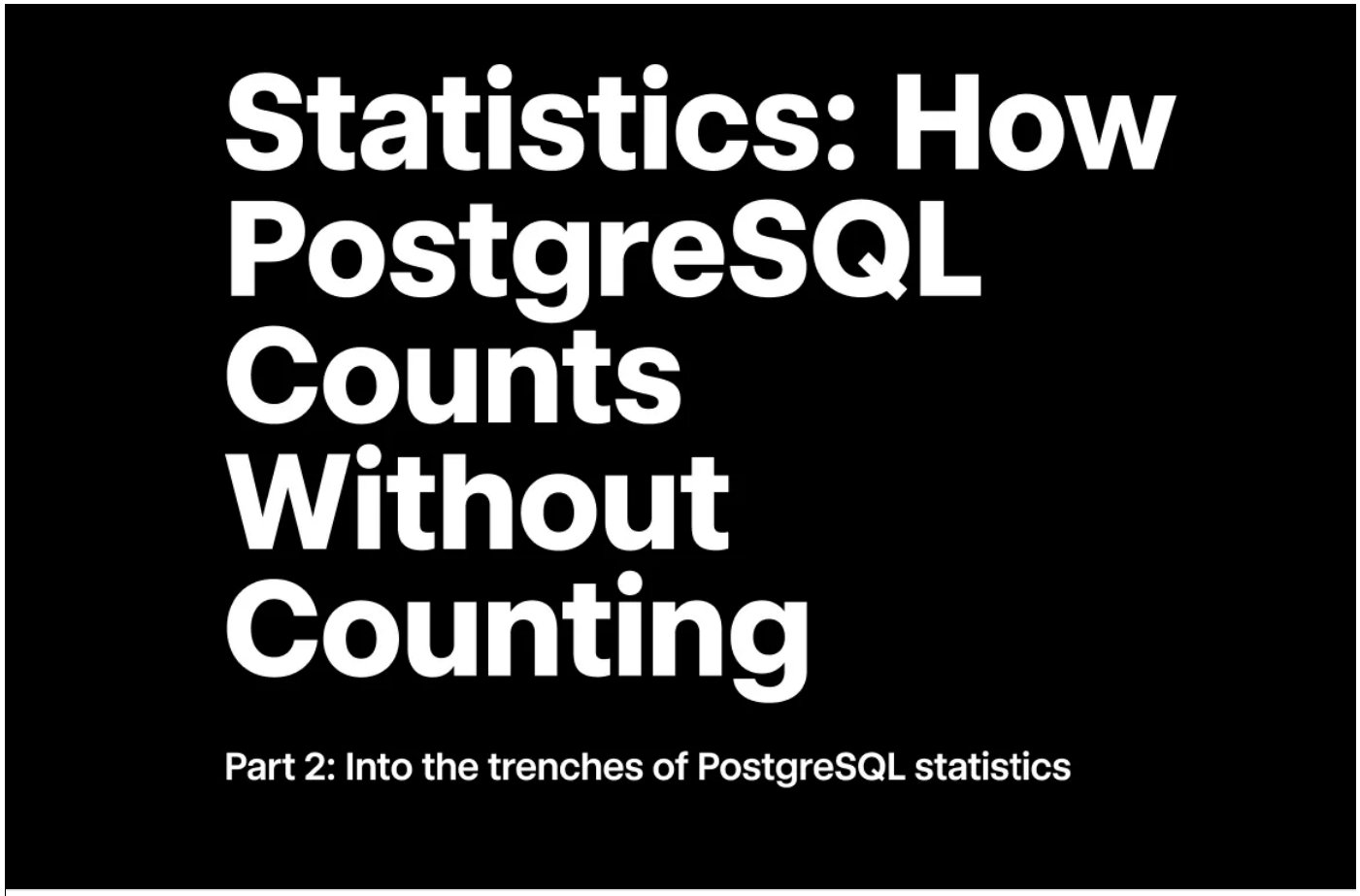


Table of Contents

Abstract

In Part 2 of this two-part article series, we delve deeper into the world of Postgres statistics! We start by looking at how Postgres samples your data, and stores histogram information to be used for query planning. We then look at deleted row and the vacuum and autovacuum processes, which is essentially responsible for keeping the statistics up to date. Speaking of, we see what can happen if the statis are stale, and how it can lead to incidents. Finally, we discuss briefly the notions of multicolumn statistics and functional indexes. The former allows Postgres to gathe statistics on multiple column, when there are some form of dependencies betwee them. The latter allow you to query a function of some column(s), assuming prope statistics is already gathered on the “index” (not just the “table”). We’ll close this series by best practices and additional readings for the curious!

Part 1: https://traderepublic.substack.com/p/statistics-how-postgresql-counts

The source code for queries in this article can be found in this GitHub repository: https://github.com/msdousti/pg-analyze-training.

Review of Part 1

In part 1, we talked about how Postgres planner decides on the cheapest executio path to took, based on a cost estimation for each execution path. We mentioned t this estimation comes from some statistics that Postgres has already gathered abc the data.

It was demonstrated that the index does **not** play much of a role here. Rather, calli `analyze` on a table (manually, or as we will see in this part, automatically via a process known as `autovacuum`) causes the statistics to be gathered in the catalog table `pg_statistics`. We used the `pg_stats` view to look at some of the statis such as `most_common_vals` and `most_common_freqs`. We'll continue this part looking at another crucial column called `histogram_bounds`.

Finally, we discussed how `EXPLAIN ANALYZE` can be used to compare the estima rows vs. the actual ones, and introduced a utility function called `c`, the takes a SQL query and extracts useful info from the output of `EXPLAIN ANALYZE` on it. Funct `c` will play an important role in this part, so be sure to create it based on Part 1.

Histograms and Data Distribution

The `pg_stats` view has a column named `histogram_bounds`, which plays an important part in query planning. Let's understand the concept of histograms first

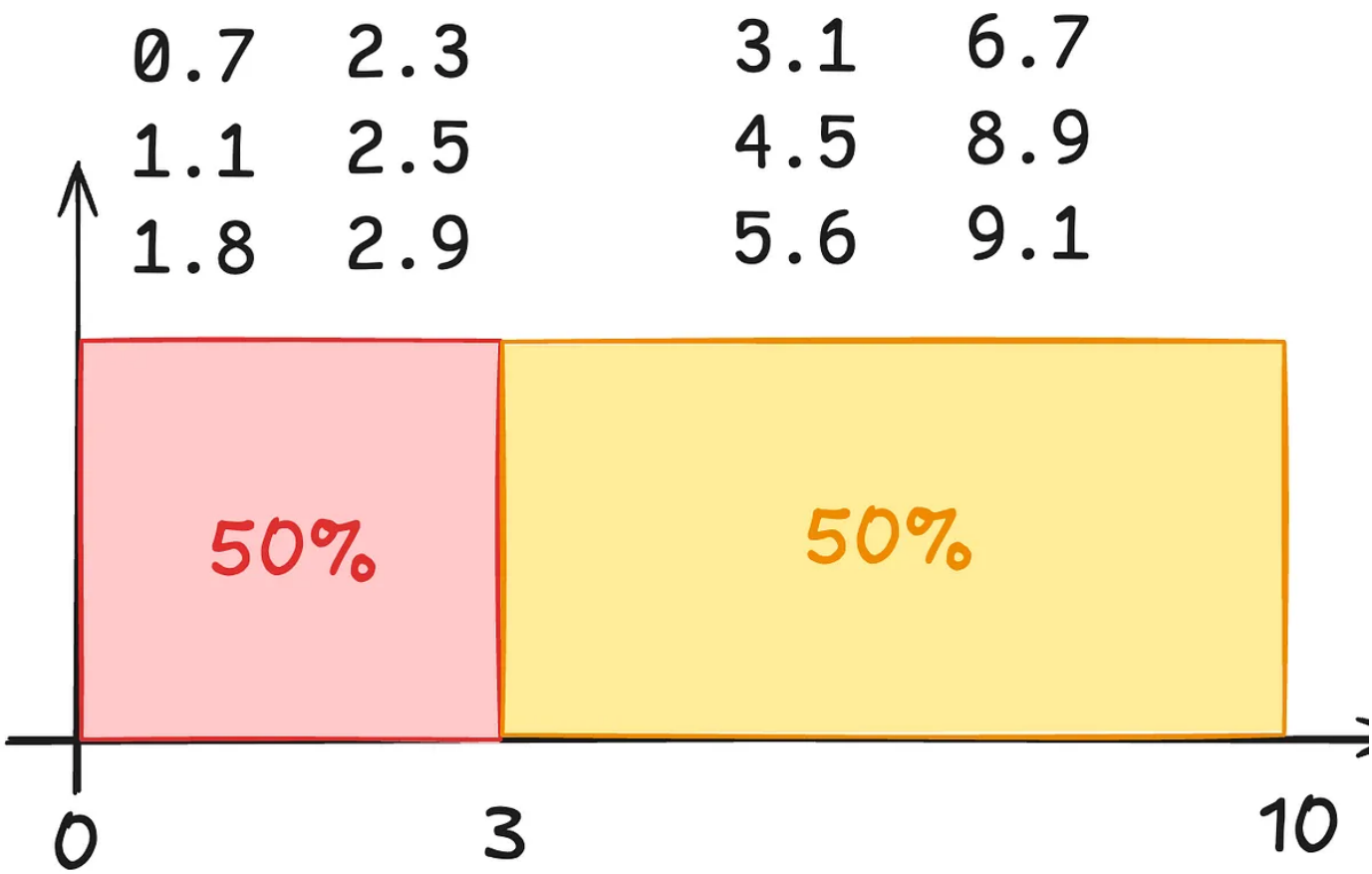
and then take a look at this column.

What are histogram bounds?

Consider the following data:

```
0.7, 1.1, 1.8, 2.3, 2.5, 2.9, 3.1, 4.5, 5.6, 6.7, 8.9, 9.1
```

Half of the data is between 0 and 3, and the other half is between 3 and 10. We ca visually represent this as follows, where "histogram bounds" are `0, 3, 10`.



Histogram bounds for the data, when target = 2 (i.e., two bars)

Histogram bounds represent the distribution of data within a column, allowing PostgreSQL to estimate the number of rows that satisfy specific conditions withou

scanning the entire table. The number of “bars” in a histogram is adjusted by the `default_statistics_target` parameter. By default, it is 100, meaning the histogram has 100 bars, each of which contains 1% of the data.

Histogram Bounds in PostgreSQL

To demonstrate histogram bounds in PostgreSQL, let’s create a table with a very “skewed” distribution. We insert these numbers:

> 1
> 1,2
> 1,2,3
> 1,2,3,4
> ...
> 1,2,3,4,...,1000

As you can see, number 1 is inserted 1000 times. Number 2 is inserted 999 times, a so on, until number 1000 is inserted only once. Here’s the SQL script to do this:

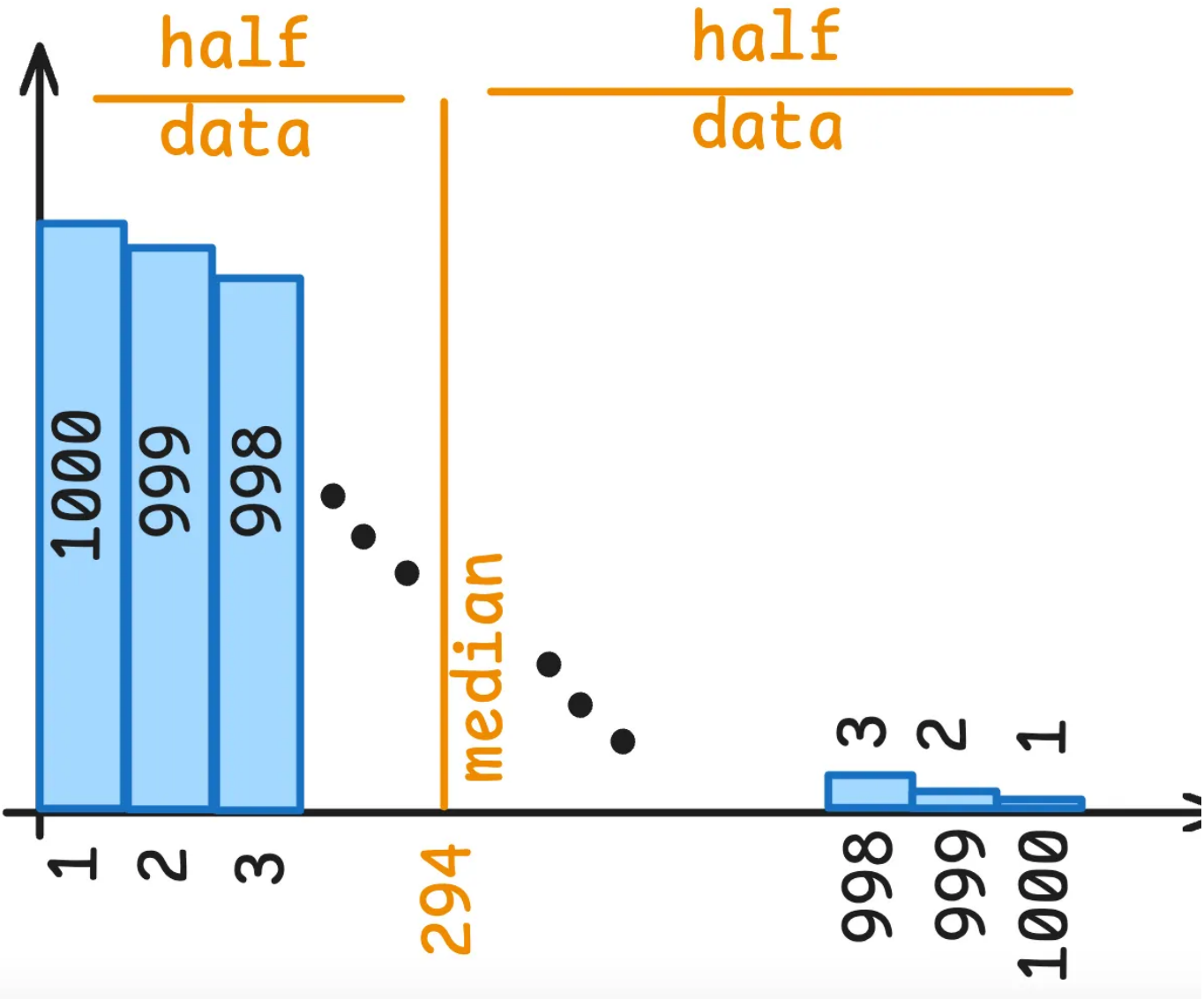
```
drop table if exists t;


create table t(n)
  with (autovacuum_enabled = off)
  as select generate_series(1, i)
     from generate_series(1, 1000) as i;
```

The data is skewed: Half of the data is in the first 1/3 of the numbers. You can chec this by asking Postgres to compute the median:

```
select
  percentile_disc(0.5) within group (order by n) as median
from t;
```

```
┌────────┐
│ median │
├────────┤
│    294 │
└────────┘
```



Distribution of inserted data.

Just to be sure that 294 is indeed the median of the data, let's actually check that 50% of the data is less than or equal to 294:

```
select
  count(*) filter (where n <= 294)
  /
  count(*)::numeric
  as ratio
from t;
```

```
┌──────────────────────────┐
│          ratio           │
├──────────────────────────┤
│ 0.50135664335664335664 │
└──────────────────────────┘
```

Let's do `ANALYZE VERBOSE t;` the keyword `VERBOSE` provides some useful information, which I present below with a little formatting:

```
scanned 2269 of 2269 pages,

containing 500500 live rows and 0 dead rows;

30000 rows in sample,

500500 estimated total rows
```

Interesting facts in the output of `ANALYZE VERBOSE`:

- **Scanned 2269 of 2269 pages:** PostgreSQL stores table data in pages on disk. Each page is by default 8KB. This means our table takes `2269 * 8KB = 18 M` of disk space.
- **Containing 500500 live rows and 0 dead rows:** There are 500,500 rows in the table, all of them **live** none is **dead**. This means there are no deleted rows. We come back to this point later.

- **30000 rows in sample:** Statistics is all about sampling. PostgreSQL generates a random sample of the table each time you run ANALYZE, and gathers statistics on the sample. This is because gathering statistics on a huge table would take enormous amount of time, which would impair other PostgreSQL tasks.
- 500500 **estimated total rows:** Based on the sample, PostgreSQL has a rough estimate of the total number of rows. In this case, it's an exact match, but it's not always the case. Under the hood, it uses a variant of reservoir sampling:

> Reservoir sampling is a family of randomized algorithms for choosing a simple random sample, without replacement, of k items from a population of unknown size n in a single pass over the items. The size of the population n is not known to the algorithm and is typically too large for all n items to fit into main memory.

`default_statistics_target` corresponds to the parameter *k* in the above definition.

Let's now take a look at query plans based on the gathered statistics:

```
select * from c('select * from t where n <= 294');
```

```
┌──────────┬──────────┬────────┐
│   scan   │ estimate │ actual │
├──────────┼──────────┼────────┤
│ Seq Scan │ 252548   │ 250929 │
└──────────┴──────────┴────────┘
```

The estimate is pretty accurate, right? If we run ANALYZE again, it gathers statistics based on another sample, and the estimate changes a bit:

```
analyze t;
```

```
select * from c('select * from t where n <= 294');
```

scan	estimate	actual
Seq Scan	249674	250929

If we look at the `histogram_bounds`, we see that 101 values are gathered, corresponding to 100 (= `default_statistics_target`) histogram bars. Each consequtive range is supposed to hold 1% of the data:

```
select
histogram_bounds::text::int[] as histogram_bounds
from pg_stats
where tablename = 't';
```

histogram_bounds
5,18,32,40,54,68,80,100,110,119,125,136,144,149,154,160,165,171,178,182
189,195,202,211,216,222,227,234,239,245,250,256,261,268,273,278,283,289,295,30
307,314,320,325,332,339,346,352,358,364,370,376,383,389,396,402,409,415,422,42
436,443,450,458,466,473,481,489,496,503,511,519,527,536,545,555,564,573,583,59
603,613,623,633,645,657,668,680,693,706,721,734,749,765,783,802,824,846,873,91
994

Histogram bounds when default_statistics_target = 100. There are 101 values collected, corresponding to 100 bars in the histogram. Each bar is supposed to contain 1% of the data. Since the data is skewed, the initial bounds are closed to each other that the final ones.

Adjusting Samples and Histograms

Adjusting the `default_statistics_target` parameter influences not only the amount of samples taken from the data, but also the granularity of these histogra

- **Higher Values**: More detailed histograms with better estimates but increased overhead (slower statistics gathering).

- **Lower Values**: Less detailed histograms, faster analysis but potentially less accurate estimates in the query plan.

For example, setting `default_statistics_target` to a low value like 2 results fewer histogram bounds, leading to less precise estimates.

```
show default_statistics_target;
┌─────────────────────────────┐
│ default_statistics_target │
├─────────────────────────────┤
│ 100                         │
└─────────────────────────────┘

analyze verbose t;

scanned 2269 of 2269 pages,
containing 500500 live rows and 0 dead rows;
30000 rows in sample, 500500 estimated total rows


set default_statistics_target = 2;


analyze verbose t;

scanned 600 of 2269 pages,
containing 131984 live rows and 0 dead rows;
600 rows in sample, 499119 estimated total rows
```

The `ANALYZE VERBOSE` shows that only 600 rows were sampled now (as opposed 30,000 rows previously). The estimated number of rows in the table is now 499,11 a little bit less accurate.

Looking at the three histogram bounds now, they are the `min`, `median`, and `max` o the sampled data, supposedly an accurate measure of the `min`, `median`, and `max` c

the actual data:

```
select
  histogram_bounds
from pg_stats
  where tablename = 't';
```

```
┌──────────────────┐
│ histogram_bounds │
├──────────────────┤
│ {1,285,968}      │
└──────────────────┘
```

```
select min(n),
  percentile_disc(0.5)
    within group (order by n) as median,
  max(n)
  from t;
```

```
┌─────┬────────┬──────┐
│ min │ median │ max  │
├─────┼────────┼──────┤
│   1 │    294 │ 1000 │
└─────┴────────┴──────┘
```

If we check the estimates now, they are a little bit less accurate than when `default_statistics_target` was 100, especially when we get far from the median, since the data is skewed:

```
select * from c('select * from t where n <= 294');
```

```
┌──────────┬──────────┬────────┐
│   scan   │ estimate │ actual │
├──────────┼──────────┼────────┤
```

```
| Seq Scan | 252848   | 250929 |
```

```
select * from c('select * from t where n < 700');
```

```
|   scan   | estimate | actual |
| Seq Scan | 400607   | 455049 |
```

It is noteworthy that PostgreSQL allows you to fine tune the sampling size on a **pe column** basis. You do not need to tamper with the global `default_statistics_target` parameter.

```
alter table t alter column n
    set statistics 10;
```

```
analyze verbose t;

scanned 2269 of 2269 pages,
containing 500500 live rows and 0 dead rows;
3000 rows in sample,
500500 estimated total rows
```

```
select array_length(histogram_bounds, 1) from pg_stats
  where tablename = 't';
```

```
| array_length |
|           11 |
```

```
select * from c('select * from t where n < 700');
```

```
┌──────────┬──────────┬────────┐
│   scan   │ estimate │ actual │
├──────────┼──────────┼────────┤
│ Seq Scan │ 453720   │ 455049 │
└──────────┴──────────┴────────┘
```

To reset everything back to defaults:

```
RESET default_statistics_target;
```

```
show default_statistics_target;
```

```
┌─────────────────────────────┐
│ default_statistics_target   │
├─────────────────────────────┤
│ 100                         │
└─────────────────────────────┘
```

```
alter table t alter column n
    set statistics -1;
```

Starting with PostgreSQL 17, you can use

`alter table ... alter column ... set statistics default;`

rather than

`alter table ... alter column ... set statistics -1;`.

Vacuum vs. Analyze

This article is about statistics and gathering it with `ANALYZE`, but I'd like to briefly address a confusion: The difference between `VACUUM` and `ANALYZE`.

Understanding Vacuum

After deleting (or updating) rows, PostgreSQL retains the old rows, but marks ther as "dead". This is similar to putting deleted files in Recycle Bin or Trash Can. If the transaction that deleted or updated the row commits, and no other transaction re to that row, the row becomes eligible for deleting forever. The `VACUUM` process cleans up these dead rows, freeing up space and maintaining database performanc

> `VACUUM` by itself does not return the space used by dead rows to the OS. It free the space so that later, PostgreSQL can insert new rows on top of the old rows. Running `VACUUM FULL` or `CLUSTER` returns the space to the OS, but at a very h cost: These command take an exclusive lock on the table, preventing any querry from executing on the table until they finish. They then create a new table, copy the data from the old table to the new, and then switch the tables and drop the old one. The process needs a lot of disk space and IO activity, and is not recommended on production databases, unless you know exactly what you're doing!

It's crucial to distinguish between `VACUUM` and `ANALYZE`:

- **`VACUUM`**: Performs 4 essential tasks: (1) Clean up dead rows, (2) Update free space maps, (3) Update visibility maps, (4) Freeze rows to prevent transaction wrap-around. (See Laurenz Albe's excellent article on this topic).
- **`ANALYZE`**: Gathers statistics about the data distribution in tables to aid the qu planner.

I just want to give you an idea of how `VACUUM` cleans up dead rows:

```
delete from t where n <= 294;


analyze verbose t;

scanned 2269 of 2269 pages,
containing 249571 live rows and 250929 dead rows;
30000 rows in sample,
249571 estimated total rows
```

The important part is that after deleting rows, ANALYZE reports that it has encountered “dead” rows during statistics gathering:

```
containing 249571 live rows and 250929 dead rows;
```

Running VACUUM will remove those dead rows (I used VACUUM VERBOSE to show more info):

```
vacuum verbose t;

pages: 0 removed, 2269 remain, 2269 scanned (100.00% of total)
tuples: 250929 removed, 249571 remain, 0 are dead but not yet removable
removable cutoff: 766, which was 0 XIDs old when operation ended
frozen: 0 pages from table (0.00% of total) had 0 tuples frozen
index scan not needed: 0 pages from table (0.00% of total) had 0 dead
item identifiers removed
avg read rate: 0.000 MB/s, avg write rate: 232.528 MB/s
buffer usage: 4599 hits, 0 misses, 535 dirtied
WAL usage: 3900 records, 1 full page images, 731913 bytes
system usage: CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.01 s
```

The important part is here:

```
tuples: 250929 removed, 249571 remain, 0 are dead but not yet removable
```

After running `VACUUM`, a total of 249,571 live rows remain, and there are 0 dead rc that are not removable (dead rows are removable if they are not visible to any transaction currently running—That's why long-running transactions are a bad ide

Re-running `ANALYZE`, we see that dead rows are gone:

```
analyze verbose t;
```

```
scanned 2269 of 2269 pages,
containing 249571 live rows and 0 dead rows;
30000 rows in sample,
249571 estimated total rows
```

Combining Vacuum and Analyze

It is possible to combine Vacuum and Analyze in a single `VACUUM ANALZE` commar The advantage is that it passes once over the table, and does all the required steps do both tasks. If you need to throw in more options, you can do so by combining th in a comma-separated list as follows:

```
vacuum (analyze, verbose) t;
```

Dangers of Stale Statistics

Stale statistics can lead to inefficient query plans. Let's demonstrate. We first crea table with `n` as primary key and disabled `autovacuum`, insert 3 rows, and then run `ANALYZE` to gather statistics. Then, we insert more values. At this point, the previously gathered statistics are **stale**:

```
drop table if exists t;


create table t(n int primary key)
  with (autovacuum_enabled = off);


insert into t values (0), (1), (2);


analyze t;


insert into t select generate_series(3, 999);
```

Next, we check the query plan, before and after updating statistics:

```
select scan from c('select * from t where n<5');
┌─────────────┐
│    scan     │
├─────────────┤
│ Seq Scan    │
└─────────────┘
```

```
analyze t;
```

```
select scan from c('select * from t where n<5');
┌──────────────────┐
│       scan       │
├──────────────────┤
│ Index Only Scan  │
└──────────────────┘
```

Before updating the statistics, the planner uses `Seq Scan`. This will sequentially s the whole table, even though we have an index (PostgreSQL primary keys are alwa indexed). After updating statistics, the planner picks the right plan: `Index Only Scan`. This means that the index has sufficient data, and the planner does not ever have to look at the table!

> **Why is that?**
>
> When the statistics are stale, the planner thinks that there are only 3 values in t table, so it does not bother to even traverse the index. Sequential scan against a huge table is a very dangerous query plan, and can take minutes or even hours t execute.

This is an admittedly contrived example, but if I had a nickel for every time PostgreSQL chooses the wrong query plan due to stale statistics. As mentioned earlier, I've seen my fair share of incidents because of this. Perplexed developers a Our service was quite fast until yesterday, but it suddenly took 2 hours in the midd of the night! Looking at dashboards, PostgreSQL logs, trying to create indices, … none works 😖. But simply running `ANALYZE` on the the affected tables does wonders!

> One other issue I faced is the lack of permission: To run `ANALYZE`, you should either be a superuser, or the table owner. My go-to solution is to have a "securit definer" function:

```
CREATE OR REPLACE FUNCTION analyze_table(table_name text)
RETURNS void
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
    -- Ensure the table name is properly quoted to prevent SQL injectio
    EXECUTE format('ANALYZE %I', table_name);
END;
$$;

-- Optional: Restrict the search path for additional security
ALTER FUNCTION analyze_table(text)
    SET search_path = public;


-- Only allow a specific role 'admin' to run this function
REVOKE ALL ON FUNCTION analyze_table(text) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION analyze_table(text) TO admin;
```

Preventing Stale Statistics

To prevent statistics from going stale, it's essential to ensure that `ANALYZE` runs regularly, especially after significant data modifications like bulk inserts or deletes This can be done manually by scheduling regular `ANALYZE`, or by exploiting PostgreSQL's `autovacuum`, which is a process run by PostgreSQL in the backgrour to regularly perform both `VACUUM` and `ANALYZE` on the database. If you list the process tree for the "oldest"`postgres` process (i.e., the parent) via the following shell command, you'll see `autovacuum launcher` listed as well:

```
pstree $(pgrep -xo postgres)


-+= 00001 postgres postgres -c max_connections=200
```

```
 |--= 00127 postgres postgres: postgres postgres [local] idle
 |--= 00069 postgres postgres: postgres postgres [local] idle
 |--= 00066 postgres postgres: logical replication launcher
 |--= 00065 postgres postgres: autovacuum launcher
 |--= 00064 postgres postgres: walwriter
 |--= 00062 postgres postgres: background writer
 \--= 00061 postgres postgres: checkpointer
```

It's important to know how `autovacuum` works. It gets up occasionally (default: ev 60 seconds), checks whether any table is eligible for vacuum/analyze, performs it i that case, and then goes back to taking a nap. `autovacuum_naptime`: Controls h often `autovacuum` runs.

```
select * from pg_settings where name = 'autovacuum_naptime';
```

```
┌─[ RECORD 1 ]───┬────────────────────────────────────────────┐
│ name           │ autovacuum_naptime                         │
│ setting        │ 60                                         │
│ unit           │ s                                          │
│ category       │ Autovacuum                                 │
│ short_desc     │ Time to sleep between autovacuum runs.     │
│ extra_desc     │ ∅                                          │
│ context        │ sighup                                     │
│ vartype        │ integer                                    │
│ source         │ default                                    │
│ min_val        │ 1                                          │
│ max_val        │ 2147483                                    │
│ enumvals       │ ∅                                          │
│ boot_val       │ 60                                         │
│ reset_val      │ 60                                         │
│ sourcefile     │ ∅                                          │
│ sourceline     │ ∅                                          │
│ pending_restart │ f                                         │
└─────────────────┴────────────────────────────────────────────┘
```

As shown above, the default is `60 seconds`, and it requires a `sighup` (Signal Han Up) for new values to take effect. This means you need to do an `alter system`, followed by `pg_reload_conf`, in order for the new value to be applied:

```
alter system set autovacuum_naptime = '100s';

select pg_reload_conf();

show autovacuum_naptime;
```

```
┌──────────────────────┐
│ autovacuum_naptime │
├──────────────────────┤
│ 100s                 │
└──────────────────────┘
```

Autovacuum offers a plethora of options, that can be set per table or system-wide

```
autovacuum_analyze_scale_factor
autovacuum_analyze_threshold
autovacuum_enabled
autovacuum_freeze_max_age
autovacuum_freeze_min_age
autovacuum_freeze_table_age
autovacuum_multixact_freeze_max_age
autovacuum_multixact_freeze_min_age
autovacuum_multixact_freeze_table_age
autovacuum_vacuum_cost_delay
autovacuum_vacuum_cost_limit
autovacuum_vacuum_insert_scale_factor
autovacuum_vacuum_insert_threshold
autovacuum_vacuum_scale_factor
autovacuum_vacuum_threshold
```

Among these, `autovacuum_enabled` enables or disables autovacuum (we've see this a lot before). If autovacuum is enabled, it determines eligibility of a table base on the amount of changes applied to it. For performing the **vacuum**, configure the parameters:

```
autovacuum_vacuum_insert_scale_factor
autovacuum_vacuum_insert_threshold
autovacuum_vacuum_scale_factor
autovacuum_vacuum_threshold
```

For performing the **analyze** task, the following parameters must be configured:

```
autovacuum_analyze_scale_factor
autovacuum_analyze_threshold
```

Since this article is about gathering statistics, we focus on the second set of configurations.

- **Scale factor** is a percentage of the table that should change, before it's eligibl for `ANALYZE`. The default is 10%.
- **Threshold** is the minimum number of rows in a table that has to change, befo it's eligible for `ANALYZE`. The default is 50 rows.

These two conditions are ANDed together, meaning at least 10% of the rows must change, AND the count of the changed rows must be at least 50. For massive table 10% is too much: A 100GB table needs a 10GB change before `AUTOANALYZE` kicks This can cause delays in statistics gathering, and lead to stale statistics. In this case setting `autovacuum_vacuum_scale_factor` to `0` and adjusting the `autovacuum_vacuum_threshold` to a fixed but large number of rows can make `autovacuum` more effective.

```
alter table t set (
    autovacuum_enabled = on, -- default
    autovacuum_analyze_scale_factor = 0,
    autovacuum_analyze_threshold = 100000
);
```

To monitor the last time manual/auto analyze or vacuum is run on a table, use the following query:

```
select
    last_analyze,
    last_autoanalyze,
    last_vacuum,
    last_autovacuum
from pg_stat_user_tables
where relname = 't';
```

Final Note: Autovacuum is a non-intrusive process; it is mindful that other queries running and it should try its best to impose little performance penalty. This is done **cost-based vacuum delay feature**. In contrast, manual vacuum/analyze is as intrus as possible. The idea is to finish the job as soon as possible. You can adjust both behaviors by configuring the following parameters (see "How does the VACUUM c model work" for more information).

```
select name, setting, unit
from pg_settings
where name ilike '%vacuum_cost%';
```

```
┌─────────────────────────────────┬─────────┬──────┐
│              name               │ setting │ unit │
├─────────────────────────────────┼─────────┼──────┤
```

```
| autovacuum_vacuum_cost_delay | 2       | ms   |
| autovacuum_vacuum_cost_limit | -1      | ø    |
| vacuum_cost_delay            | 0       | ms   |
| vacuum_cost_limit            | 200     | ø    |
| vacuum_cost_page_dirty       | 20      | ø    |
| vacuum_cost_page_hit         | 1       | ø    |
| vacuum_cost_page_miss        | 2       | ø    |
└──────────────────────────────┴─────────┴──────┘
```

Multicolumn Statistics

When dealing with multiple correlated columns, PostgreSQL's default single-colum statistics may not suffice. By creating **extended statistics** on combinations of columns, PostgreSQL can better understand the dependencies between them, leading to more accurate query plans. For instance, let's create a table with correlated columns `n` and `m`:

```
drop table if exists t;



create table t(n int, m int)
  with (autovacuum_enabled = off);



insert into t
  select mod(n,2), mod(n,4) from generate_series(1,1000) as n;
```

`ANALYZE` the table, and then query each column individually, and then in combinat to each other. You see that the individual column queries result in 100% correct estimate, while the combination `n=1 AND m=1` results in an estimate that is only 50% of the actual value:

```
analyze t;
```

```
select * from c('select * from t where n=1');
```

scan	estimate	actual
Seq Scan	500	500

```
select * from c('select * from t where m=1');
```

scan	estimate	actual
Seq Scan	250	250

```
select * from c('select * from t where n=1 and m=1');
```

scan	estimate	actual
Seq Scan	125	250

The reason is that PostgreSQL gathers statistics for individual columns only:

```
select
    attname,
    most_common_vals,
```

```
    most_common_freqs
from pg_stats
where tablename = 't';
```

attname	most_common_vals	most_common_freqs
n	{0,1}	{0.5,0.5}
m	{0,1,2,3}	{0.25,0.25,0.25,0.25}

When performing `AND`, it assumes the two columns are independently distributed. 50% of the rows have `n=1`, and 25% of the rows have `m=1`. Assuming independenc only `0.5 * 0.25 = 0.125` of the rows have both conditions satisfied, which amounts to the wrong estimate of 125 rows.

Note that a composite index does not help in a better estimate:

```
create index on t(n,m);
```

```
select * from c('select * from t where n=1 and m=1');
```

scan	estimate	actual
Bitmap Heap Scan	125	250

With extended statistics, queries involving both `n` and `m` can be optimized more effectively, as PostgreSQL now recognizes their correlation:

```
create statistics t_m_n on n,m from t;
```

```
analyze t;


select * from c('select * from t where n=1 and m=1');
┌──────────────────────┬──────────┬────────┐
│         scan         │ estimate │ actual │
├──────────────────────┼──────────┼────────┤
│ Bitmap Heap Scan     │ 250      │ 250    │
└──────────────────────┴──────────┴────────┘
```

Extended statistics are stored in the table `pg_statistic_ext`. The view `pg_stats_ext` provides a user-friendly representation of this table:

```
select * from pg_stats_ext where statistics_name = 't_m_n';
┌─[ RECORD 1 ]──────────┬──────────────────────────────────┐
│ schemaname            │ public                           │
│ tablename             │ t                                │
│ statistics_schemaname │ public                           │
│ statistics_name       │ t_m_n                            │
│ statistics_owner      │ sadeq                            │
│ attnames              │ {n,m}                            │
│ exprs                 │ ∅                                │
│ kinds                 │ {d,f,m}                          │
│ inherited             │ f                                │
│ n_distinct            │ {"1, 2": 4}                      │
│ dependencies          │ {"2 => 1": 1.000000}             │
│ most_common_vals      │ {{0,0},{0,2},{1,1},{1,3}}        │
│ most_common_val_nulls │ {{f,f},{f,f},{f,f},{f,f}}        │
│ most_common_freqs     │ {0.25,0.25,0.25,0.25}            │
│ most_common_base_freqs │ {0.125,0.125,0.125,0.125}       │
└──────────────────────┴──────────────────────────────────┘
```

The `kinds` column is interesting:

- `f`: **Extended statistics of kind** `functional dependencies`**.** The simplest ki of extended statistics tracks *functional dependencies*, a concept used in

definitions of database normal forms. We say that column b is functionally dependent on column a if knowledge of the value of a is sufficient to determi the value of b, that is there are no two rows having the same value of a but different values of b.

- d**: Extended statistics of kind** `ndistinct` **(Number of distinct values).** Sing column statistics store the number of distinct values in each column. Estimate the number of distinct values when combining more than one column (for example, for `GROUP BY a, b`) are frequently wrong when the planner only h single-column statistical data, causing it to select bad plans.
- `m`**: Extended statistics of kind** `mcv` **(Most Common Values).** This allows very accurate estimates for individual columns, but may result in significant misestimates for queries with conditions on multiple columns.

Depending on the scenario, you may be interested to gather only a specific extend statistics. This can be done by specifying the kind in paranthesis:

```
create statistics (dependencies) on n,m from t;

create statistics (ndistinct) on n,m from t;

create statistics (mcv) on n,m from t;
```

For more information, see the official documentation on extended statistics.

Indexes on Expressions (Functional Indexes)

From Postgres docs,

> An index column need not be just a column of the underlying table, but can be a function or scalar expression computed from one or more columns of the table

> This feature is useful to obtain fast access to tables based on the results of computations.

Running ANALYZE on the table is mandatory after creation of a functional index. Here's an example, where we run ANALYZE before, and don't get the desired plan.

So, create the table, insert some data, and run ANALYZE:

```
drop table t;

create table t(i int)
  with (autovacuum_enabled = off);

insert into t
  select generate_series(1, 10000);

analyze t;
```

Then, create a functional index, and see the query plan:

```
create index sqrt_idx on t((sqrt(i)::int));

select * from c('select * from t where sqrt(i)::int = 4');
┌────────────────────┬──────────┬────────┐
│        scan        │ estimate │ actual │
├────────────────────┼──────────┼────────┤
│ Bitmap Heap Scan   │ 50       │ 8      │
└────────────────────┴──────────┴────────┘
```

Hm. `Bitmap Heap Scan`! And the estimate is almost an order of magnitude off. Let's run ANALYZE again, and see how it affects the query plan:

```
analyze t;
```

```
select * from c('select * from t where sqrt(i)::int = 4');
┌────────────┬──────────┬────────┐
│    scan    │ estimate │ actual │
├────────────┼──────────┼────────┤
│ Index Scan │ 8        │ 8      │
└────────────┴──────────┴────────┘
```

Excellent! We now have `Index Scan` with an accurate estimate.

To see what has changed under the hood, look at `pg_stats`. And if you get shock by using the index name to filter for `tablename`, you're not alone! This is probably one of those use cases where something is retrofitted into Postgres 😉. And I shou point out that in Postgres, indexes and tables are stored on disk and in RAM in a ve similar manner.

```
select * from pg_stats where tablename = 'sqrt_idx';
```

Best Practices for Maintaining Statistics

1. **Regularly Run** ANALYZE: Especially after significant data changes, to ensure PostgreSQL has accurate statistics for query planning.
2. **Leverage Autovacuum**: Configure `autovacuum` settings to balance performance and overhead, ensuring timely maintenance without disrupting operations.
3. **Use Extended Statistics**: For tables with correlated columns, create extended statistics to improve query planning accuracy.
4. **Monitor Performance Metrics**: Set up alerts based on query response times t detect and address issues caused by stale statistics.

5. **Post-Migration Analysis**: After major version upgrades or failovers, run ANAL
to ensure statistics are accurate and up-to-date.

Additional Reading

- *Louise Grandjonc—A Deep Dive into Postgres Statistics (PGConf.EU 2024)*
- *Franck Pachot— pg_hint_plan Series*
- *Franck Pachot— Out of Range statistics with PostgreSQL & YugabyteDB*
- *How we used Postgres extended statistics to achieve a 3000x speedup*

Conclusions

Statistics are the backbone of PostgreSQL's query planner, enabling it to make intelligent decisions without the need for exhaustive counting. By understanding a effectively managing these statistics, you can ensure that your PostgreSQL databa remain performant and responsive, even under heavy and complex workloads.

Key Takeaways:

- Statistics are super important in getting a good query plan
- Autovacuum does 5 different tasks, one of which is gathering statistics
- Adjust autovacuum configurations on your tables
- Disable autovacuum before big ELT, enable afterwards & run analyze
- Run analyze always after (1) **major version upgrade**, (2) creating index on **expressions**, and (3) failover for **logical** replicas
- Consider running vacuum analyze as a nightly task using `pg_cron`
- Monitor `pg_stat_user_tables` for the last time (auto)vacuum/analyze executed

- Consider using extended statistics

A guest post by

Sadeq Dousti

I am a staff software engineer and tech lead with over 14 years of experience in various domains, including banking, retail, telecom, and logistics. I maintain and contribute to several open-source libraries, such as Logbook and Riptide.

Discussion about this post