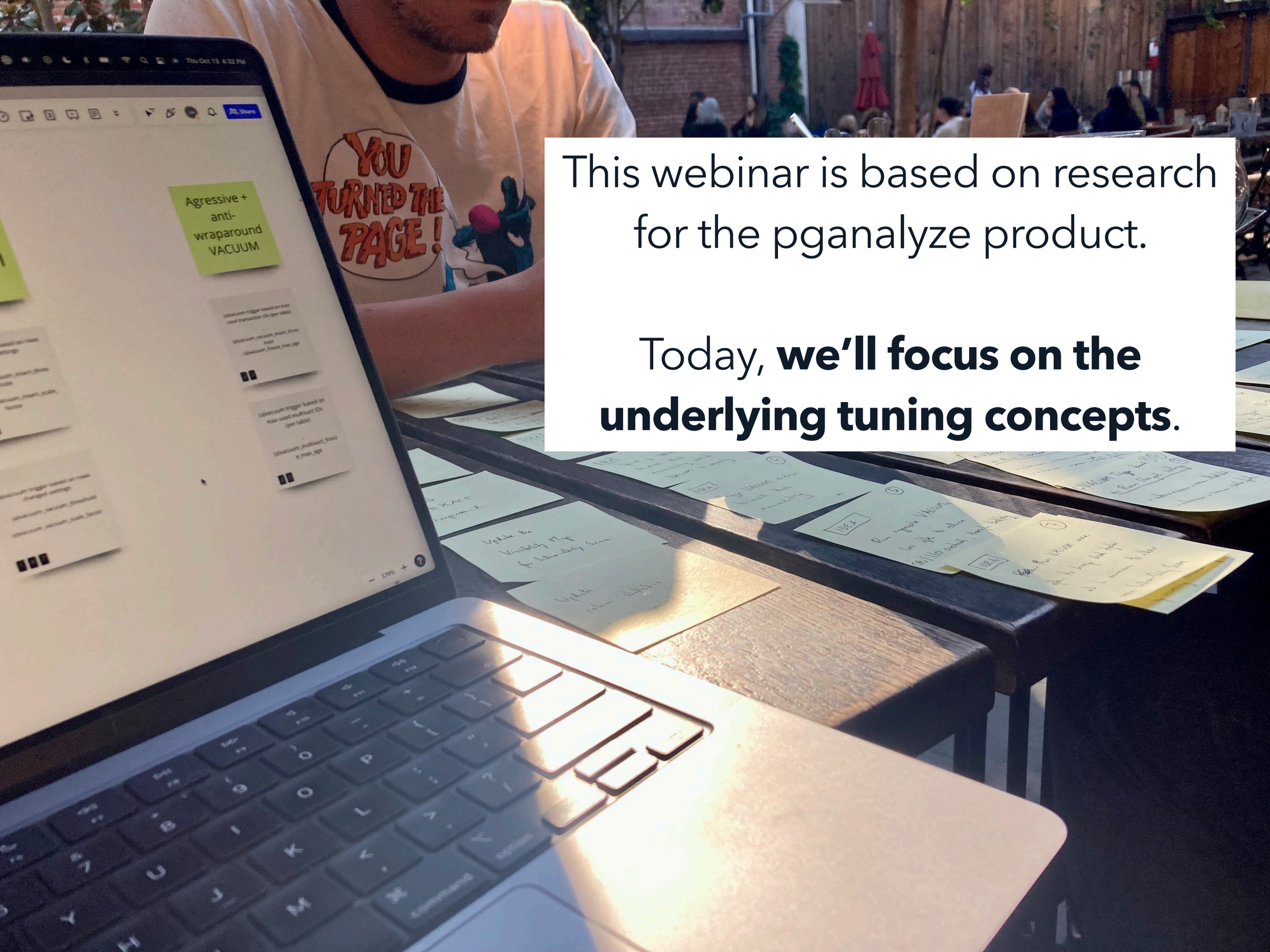
Tuning Postgres autovacuum to Improve Performance and Reduce Bloat

@LukasFittl



This webinar is based on research for the pganalyze product.

Today, **we'll focus on the underlying tuning concepts**.

What We'll Talk About Today

1. On the physicality of tables

2. Tick-tock, it's the Transaction ID clock

3. All-visible pages and slow queries

4. Understanding autovacuum scheduling

5. Cost delay 101

6. Aggressive and Failsafe VACUUMs

7. Dead Tuples Not Yet Removable

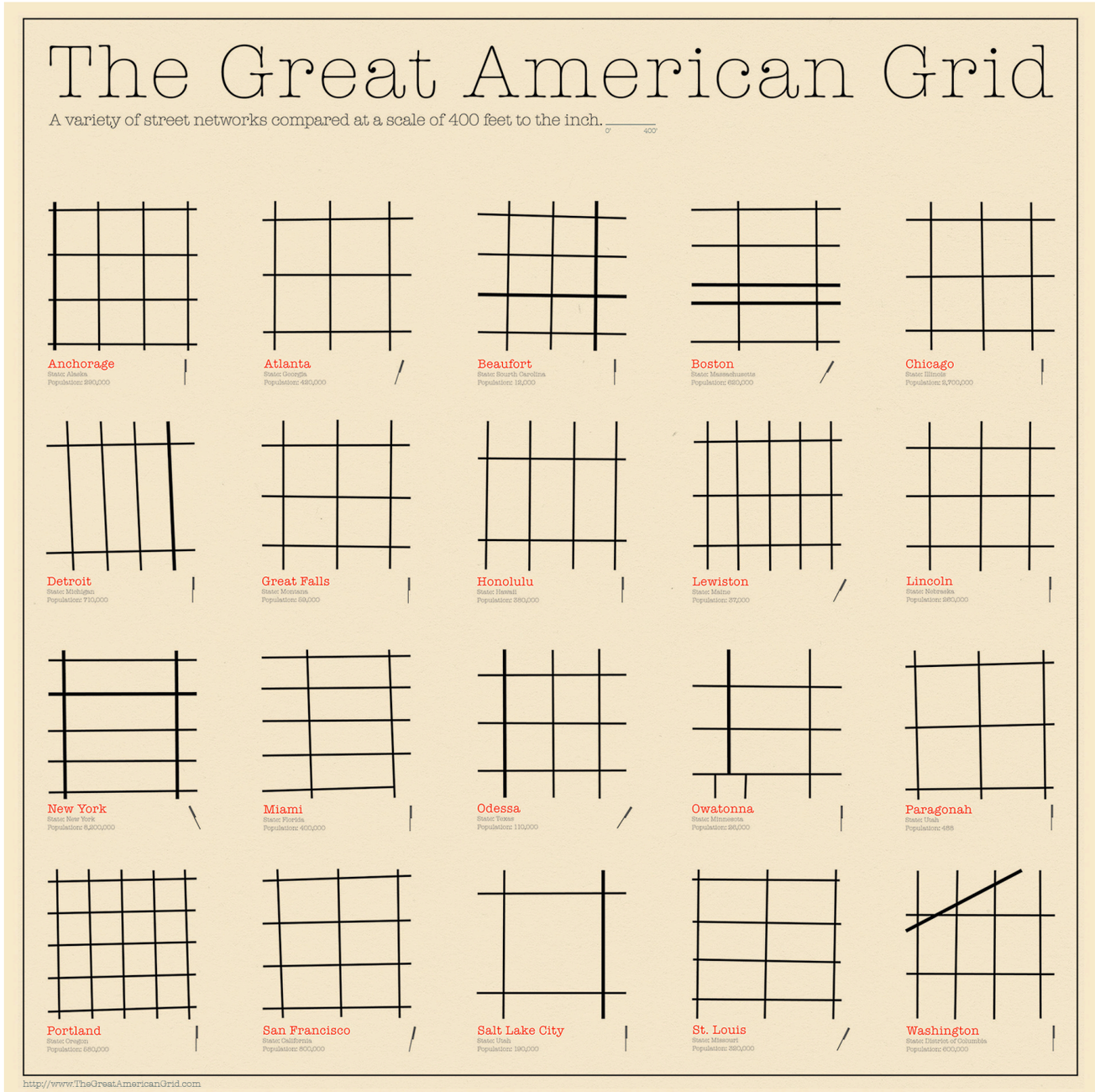
8. VACUUMing in the cloud - RDS, Aurora, Cloud SQL & AlloyDB

9. Estimating and fixing table bloat

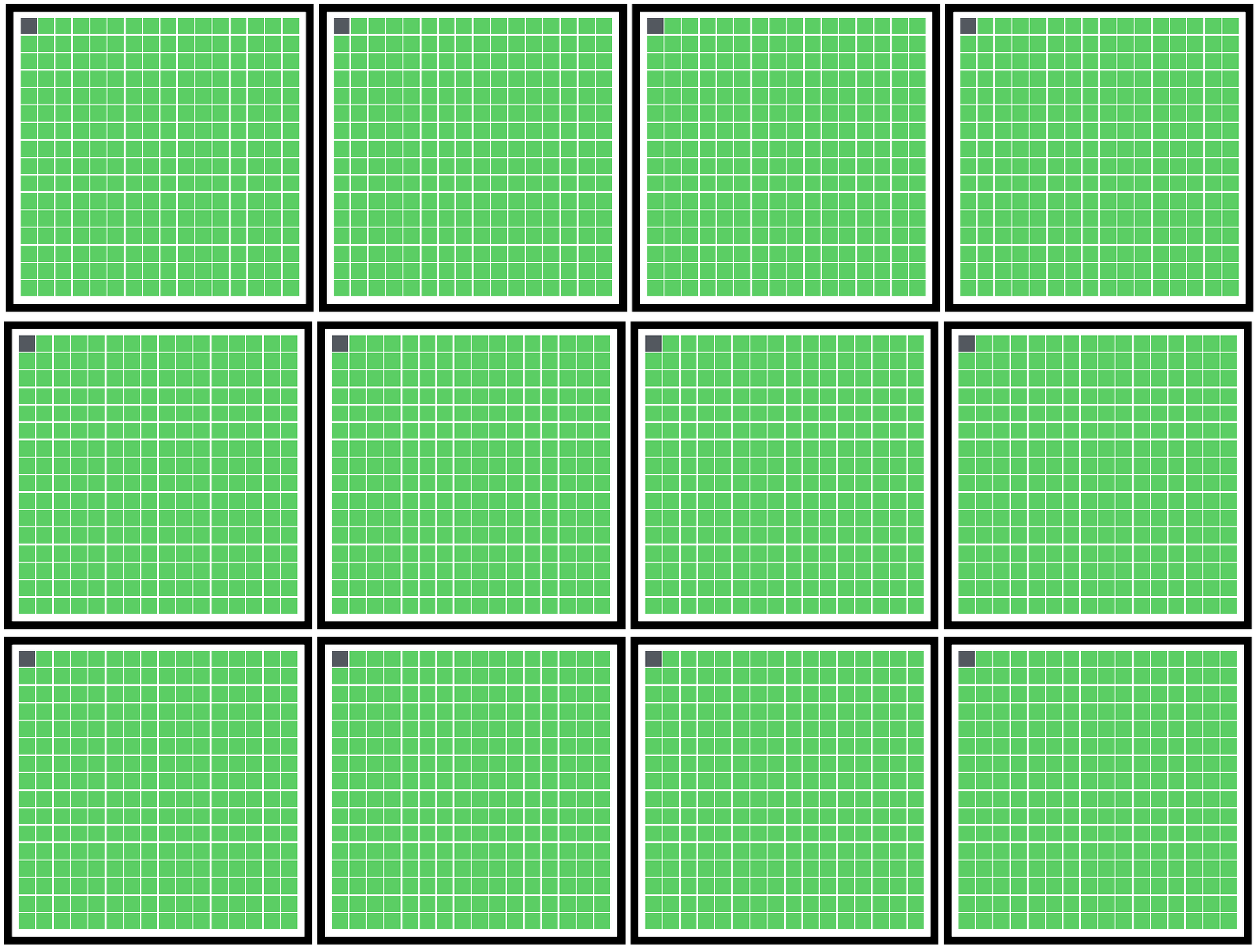
pganalyze

On the physicality of tables

Postgres Tables are a little bit like housing blocks of American Cities



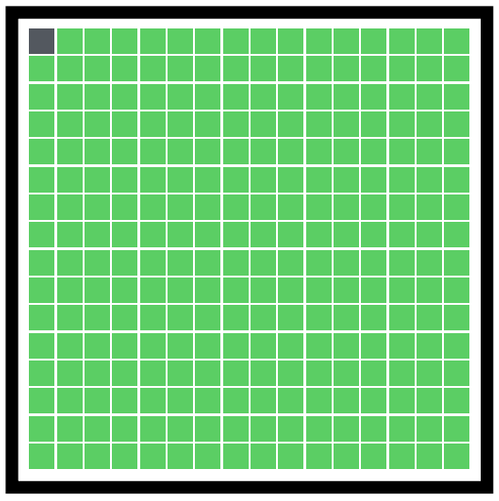
pganalyze

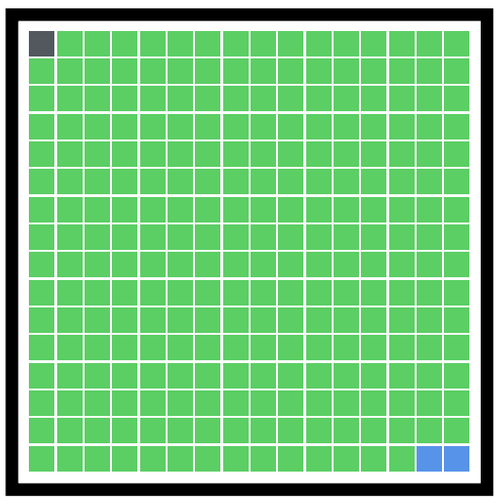


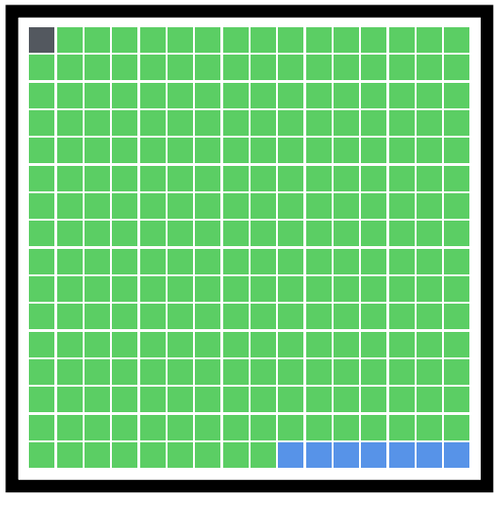
~100kB

Each ▪ is
32 bytes

Each □ is
8kB
(1 page)



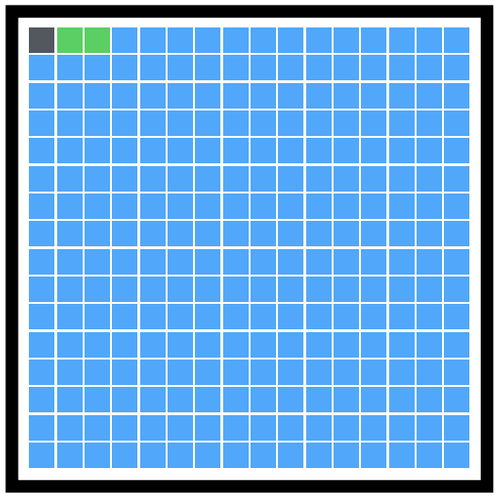




CREATE TABLE test(id SERIAL, data text);

INSERT INTO test VALUES ('short');

INSERT INTO test VALUES (‘longlonglonglonglong longlonglonglonglonglonglonglonglonglonglong longlonglonglonglonglonglonglonglonglonglong longlonglonglonglong');



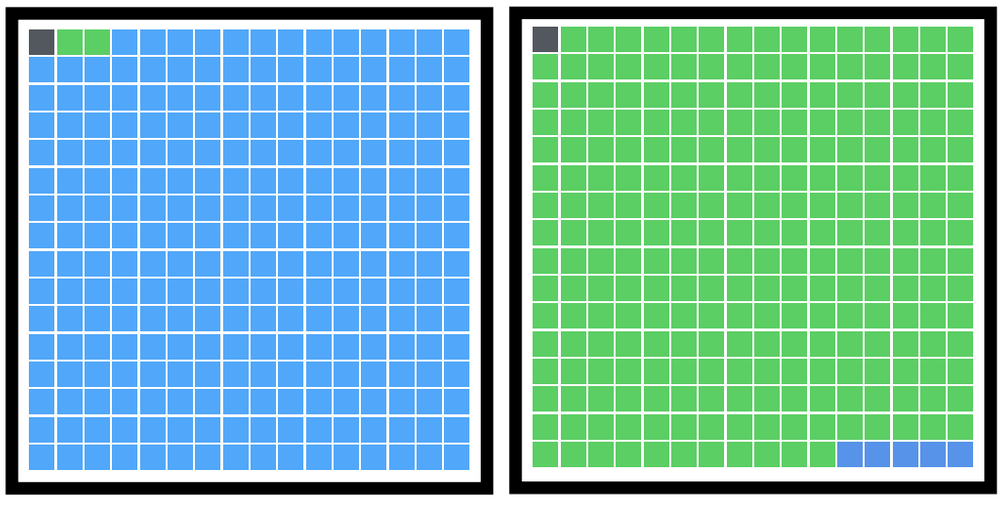
INSERT INTO test VALUES ('longlonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglong');

48 more times

We now have 50 live tuples...

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 8192
tuple_count        | 50
tuple_len          | 7874
tuple_percent      | 96.12
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 84
free_percent       | 1.03
```

live tuples / page density = 50

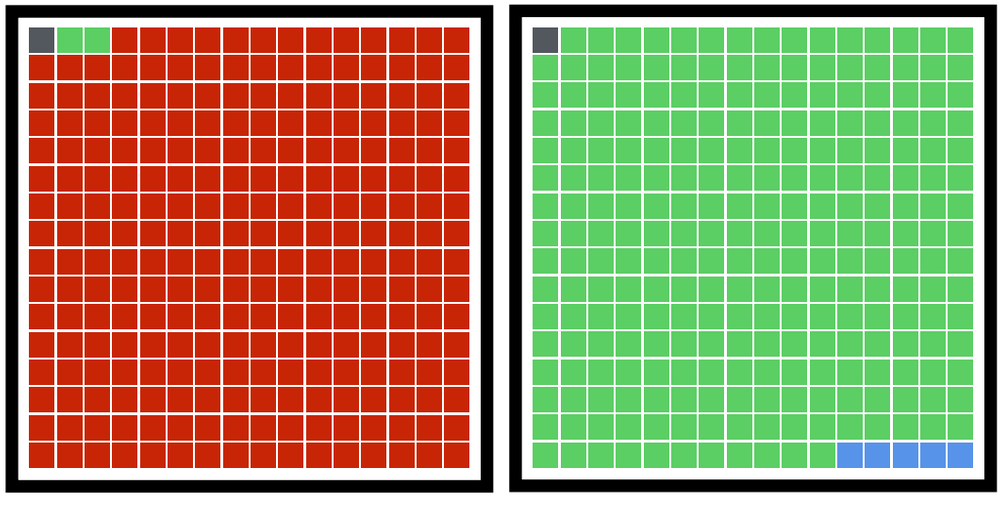


INSERT INTO test VALUES ('longlonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglonglonglonglonglonglong longlonglonglonglong');

We now have 51 live tuples.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 51
tuple_len          | 8034
tuple_percent      | 49.04
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 8084
free_percent       | 49.34
```

live tuples / page density = 25.5

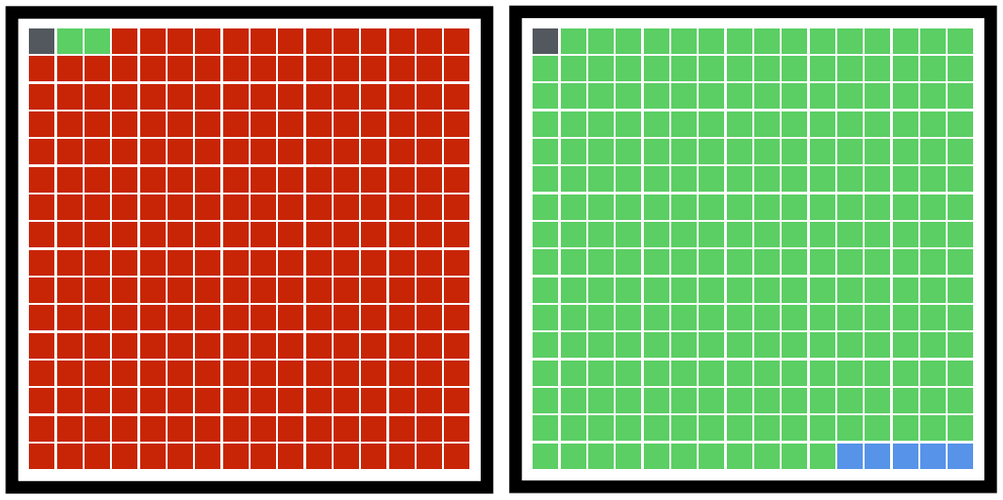


DELETE FROM test WHERE id < 51

We now have 1 live tuple and 50 dead tuples.

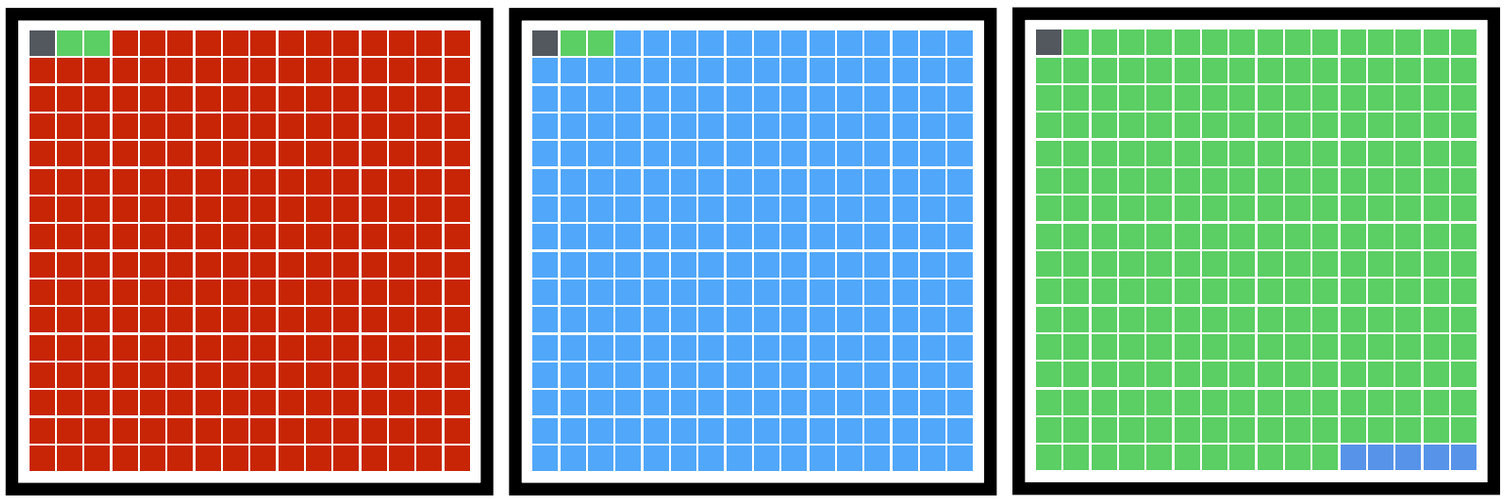
```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 50
dead_tuple_len     | 7874
dead_tuple_percent | 48.06
free_space         | 8084
free_percent       | 49.34
```

live tuples / page density = 0.5

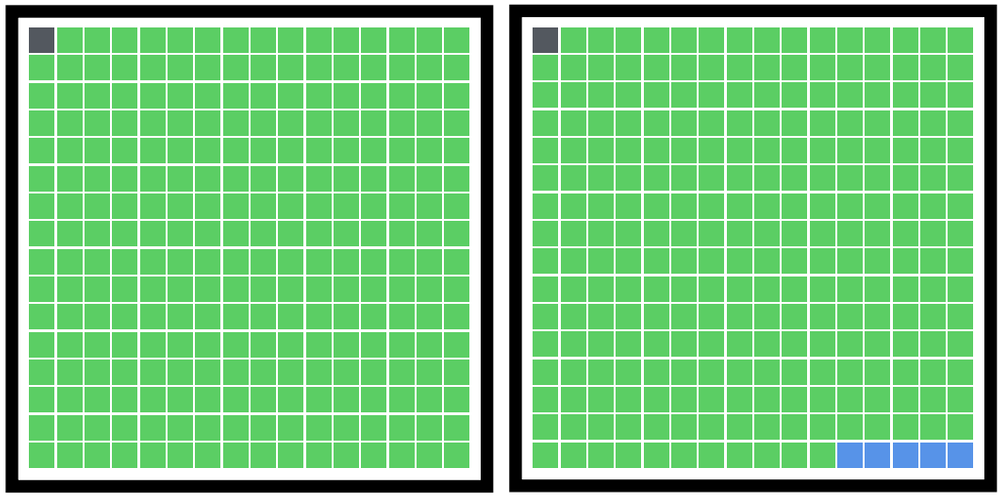


Do we get VACUUMed before the next 50 inserts?

If not, we keep growing the table...



...

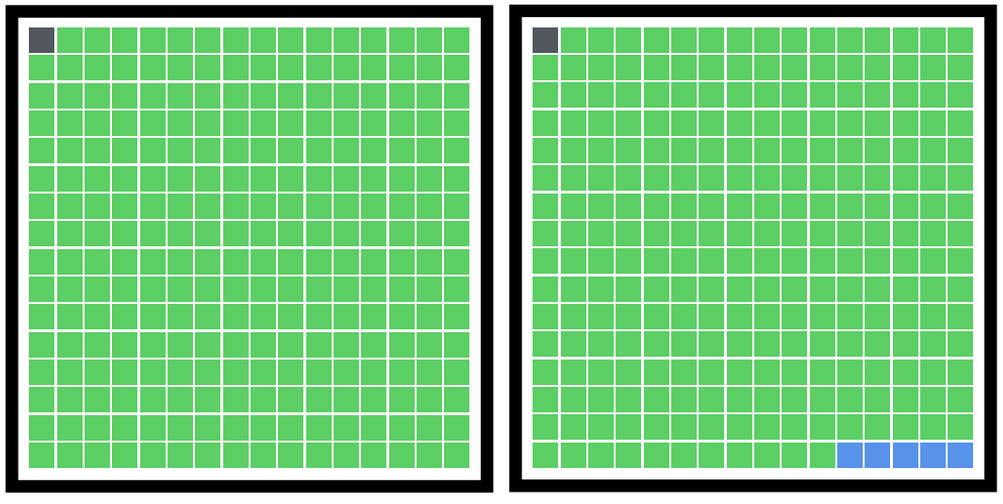


VACUUM test;

We now have 1 live tuple.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 16160
free_percent       | 98.63
```

live tuples / page density = 0.5



VACUUM test;

We now have 1 live tuple.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 16160
free_percent       | 98.63
```

live tuples / page density = 0.5

Bloat!

VACUUM's first job:

Keeping (excessive) bloat away.

(by turning dead tuples into free space,

before we need that free space)

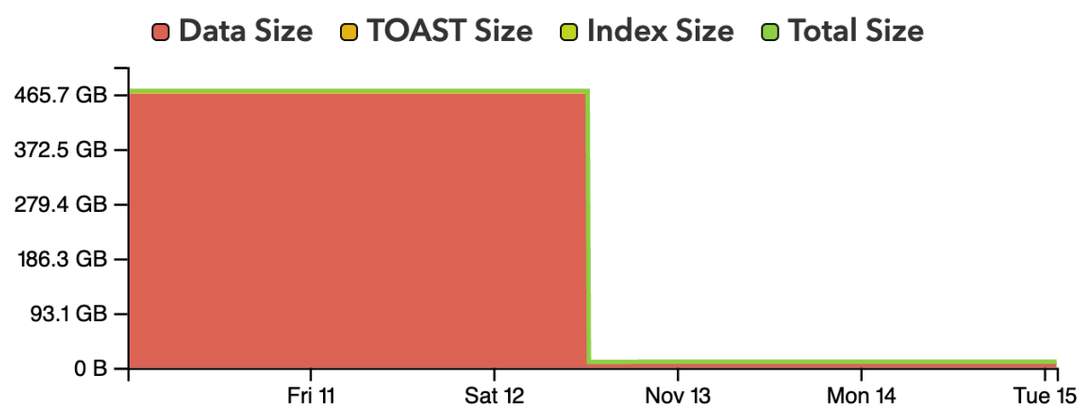
When VACUUM didn't do its job,

you probably need pg_repack to fix it (more on that later)

```
-[ RECORD 1 ]------+-------------
table_len          | 501153767424
tuple_count        | 22117518
tuple_len          | 8623158256
tuple_percent      | 1.72
dead_tuple_count   | 676
dead_tuple_len     | 262123
dead_tuple_percent | 0
free_space         | 476997629164
free_percent       | 95.18
```

→

```
-[ RECORD 1 ]------+-----------
table_len          | 9559777280
tuple_count        | 22629063
tuple_len          | 9047624099
tuple_percent      | 94.64
dead_tuple_count   | 166446
dead_tuple_len     | 132483269
dead_tuple_percent | 1.39
free_space         | 183796600
free_percent       | 1.92
```



pganalyze

pganalyze

Tick-tock, it's the Transaction ID clock

"It is necessary to vacuum every table in every database at least once **every two billion transactions**."

```
typedef uint32 TransactionId;

TransactionId xid;

/* advance a transaction ID variable, handling wraparound correctly */
#define TransactionIdAdvance(dest)  \
    do { \
        (dest)++; \
        if ((dest) < FirstNormalTransactionId) \
            (dest) = FirstNormalTransactionId; \
    } while(0)
```

Transaction IDs are stored as 32-bit (~ 4 billion unique values)

What happens if we increment max(uint32) by 1?

=> We get 0 (InvalidTransactionID)

=> When Postgres does that, it advances the TXID to 3 (FirstNormalTransactionID)

```
typedef struct HeapTupleFields
{
    TransactionId t_xmin;       /* inserting xact ID */
    TransactionId t_xmax;       /* deleting or locking xact ID */
…
```

In each tuple (row version), the header's t_xmin/t_xmax determines which connections can see the tuple.

Each active backend (or connection) has an "xmin", which is that connection's view of the world, consistent as of that TXID

xmin = 5

X t_xmin=3, t_xmax=

X t_xmin=4, t_xmax=7

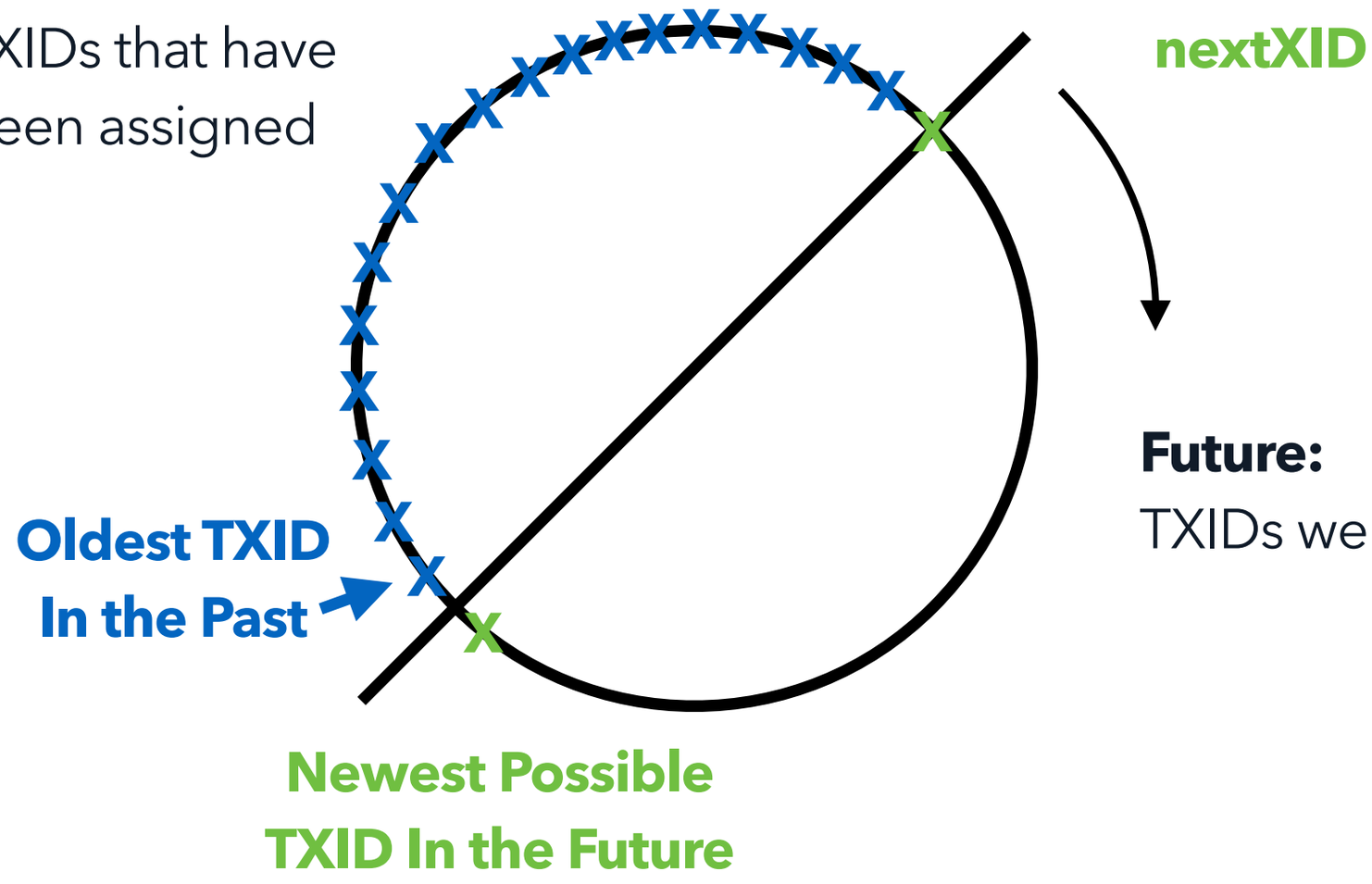
X t_xmin=5, t_xmax=

X t_xmin=6, t_xmax=

```
=# SELECT backend_xmin FROM pg_stat_activity WHERE pid = pg_backend_pid();

-[ RECORD 1 ]+--
backend_xmin | 5
```

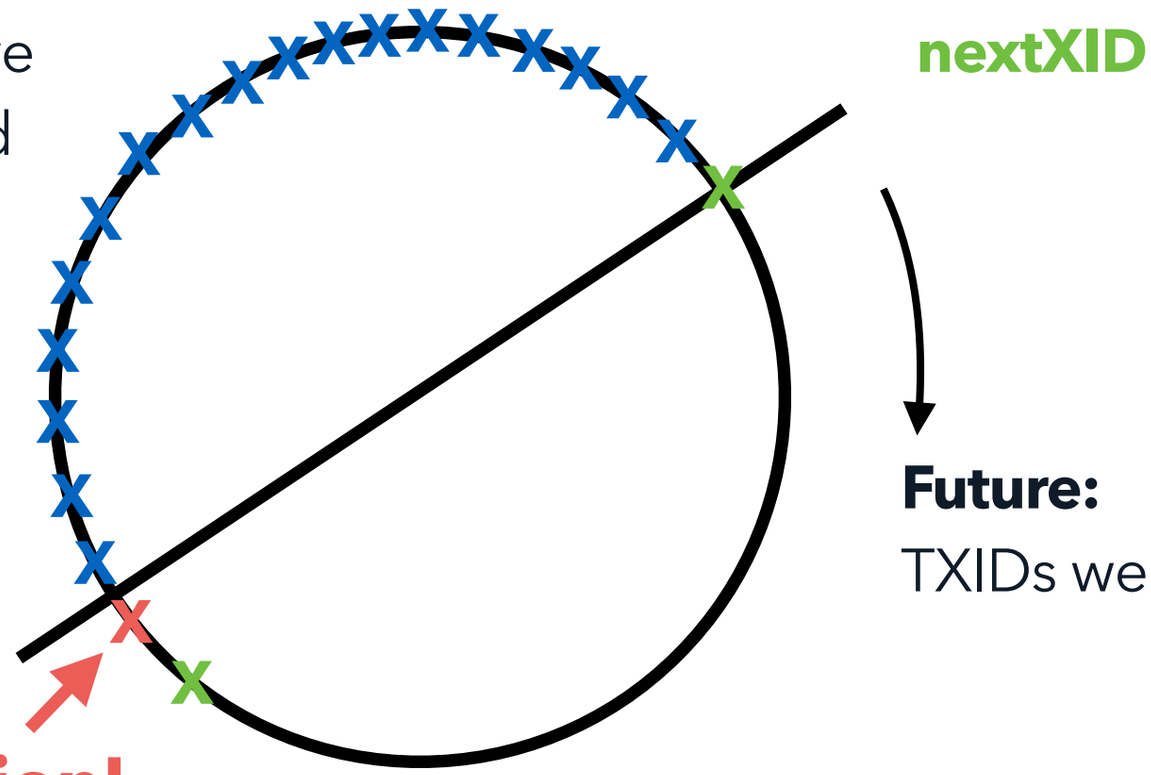
Past:
TXIDs that have
been assigned



Future:
TXIDs we can assign

pganalyze

Past:
TXIDs that have
been assigned

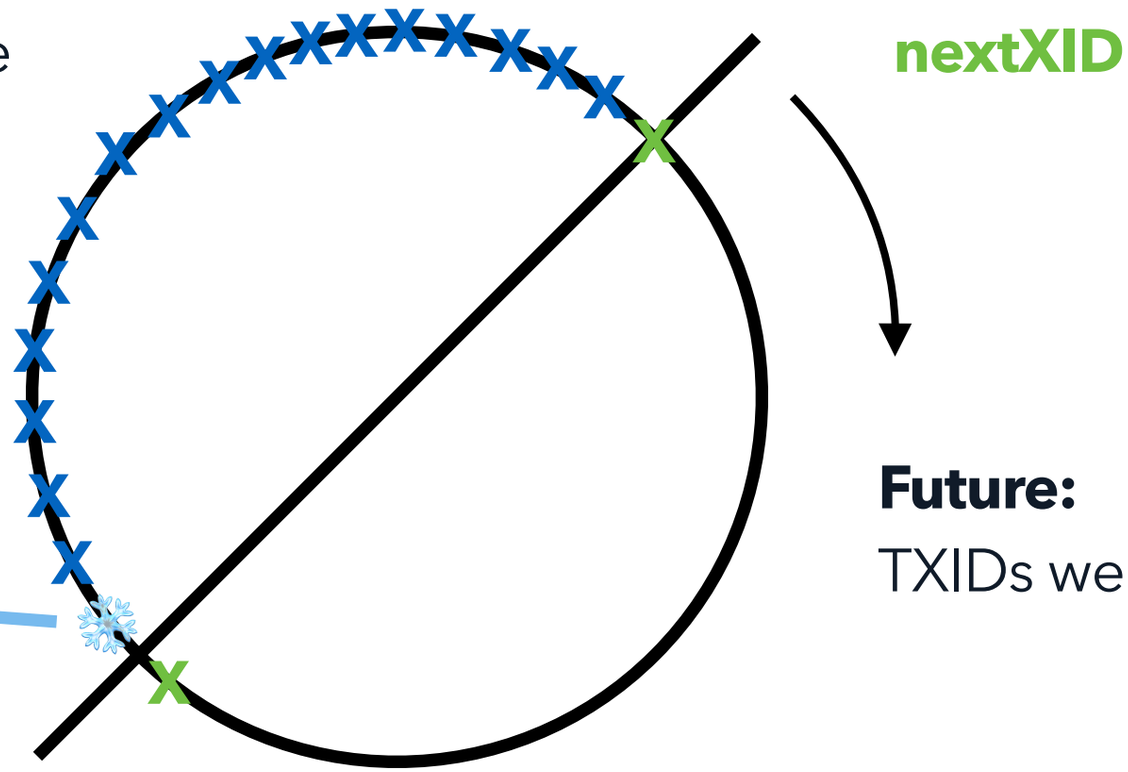


Future:
TXIDs we can assign

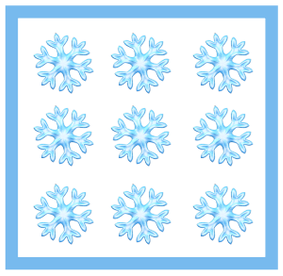
Data Corruption!
The row suddenly disappears,
as it appears to be in the future

pganalyze

Past:
TXIDs that have
been assigned



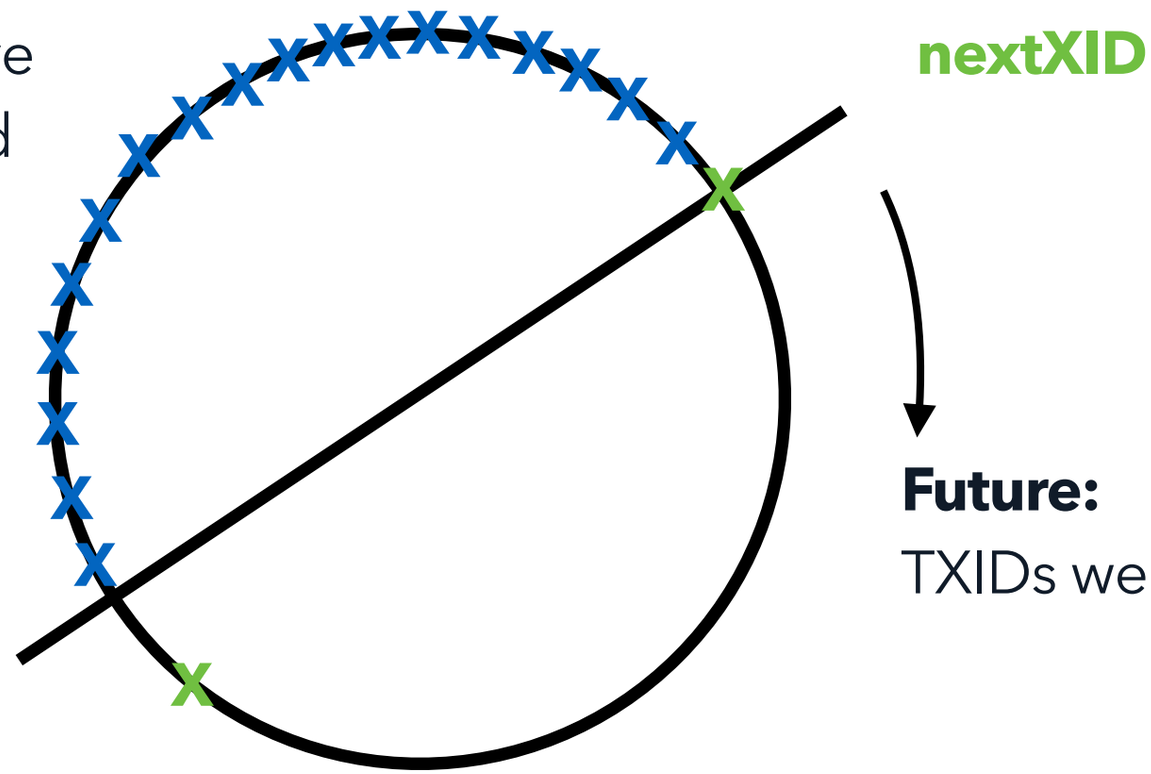
Future:
TXIDs we can assign



Frozen XID
= Special XID that means
the page is visible to everyone,
needs no more VACUUM visits,
and its outside the “Transaction ID clock”

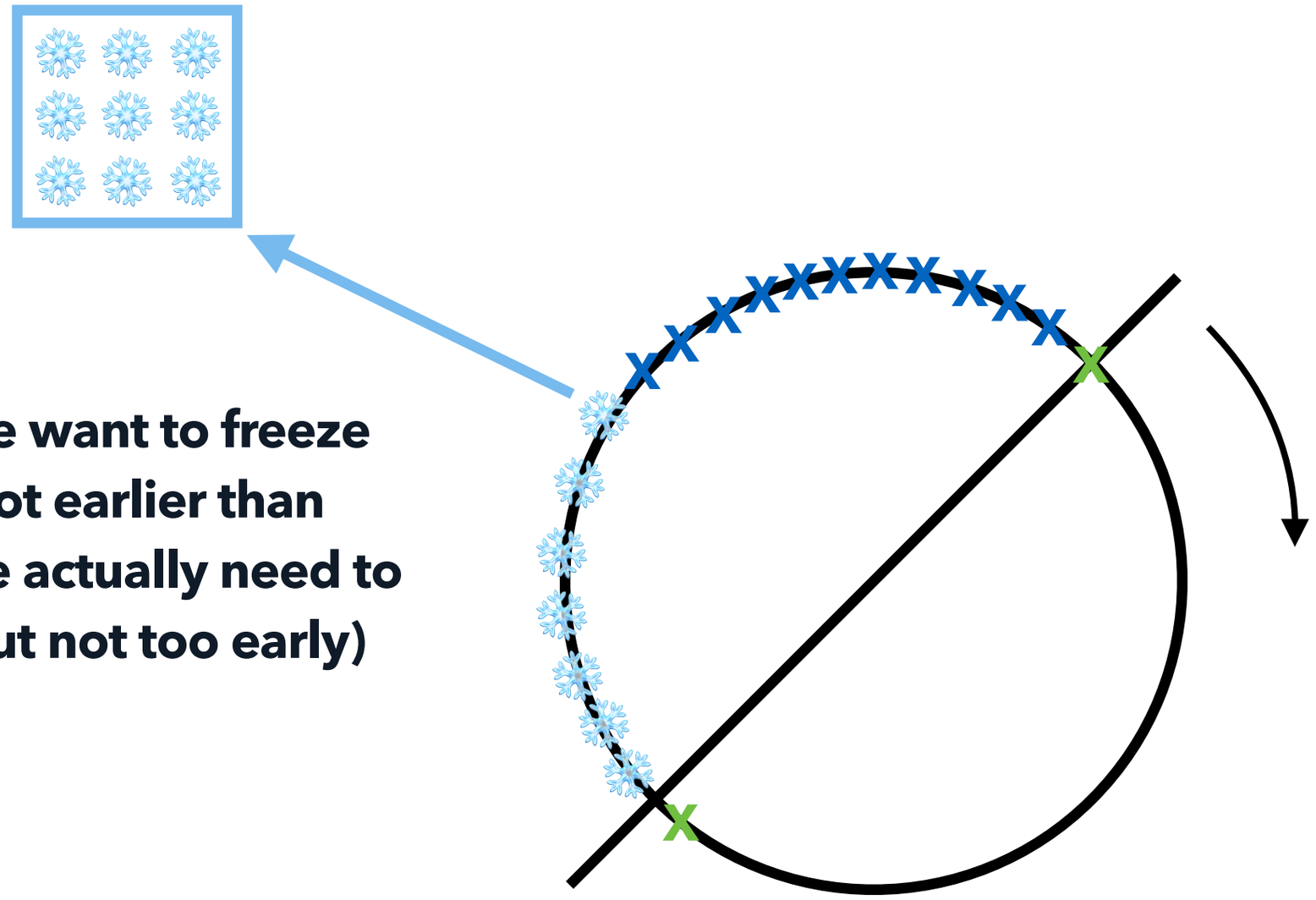
pganalyze

Past:
TXIDs that have
been assigned



Future:
TXIDs we can assign

pganalyze



We want to freeze a lot earlier than we actually need to (but not too early)

pganalyze

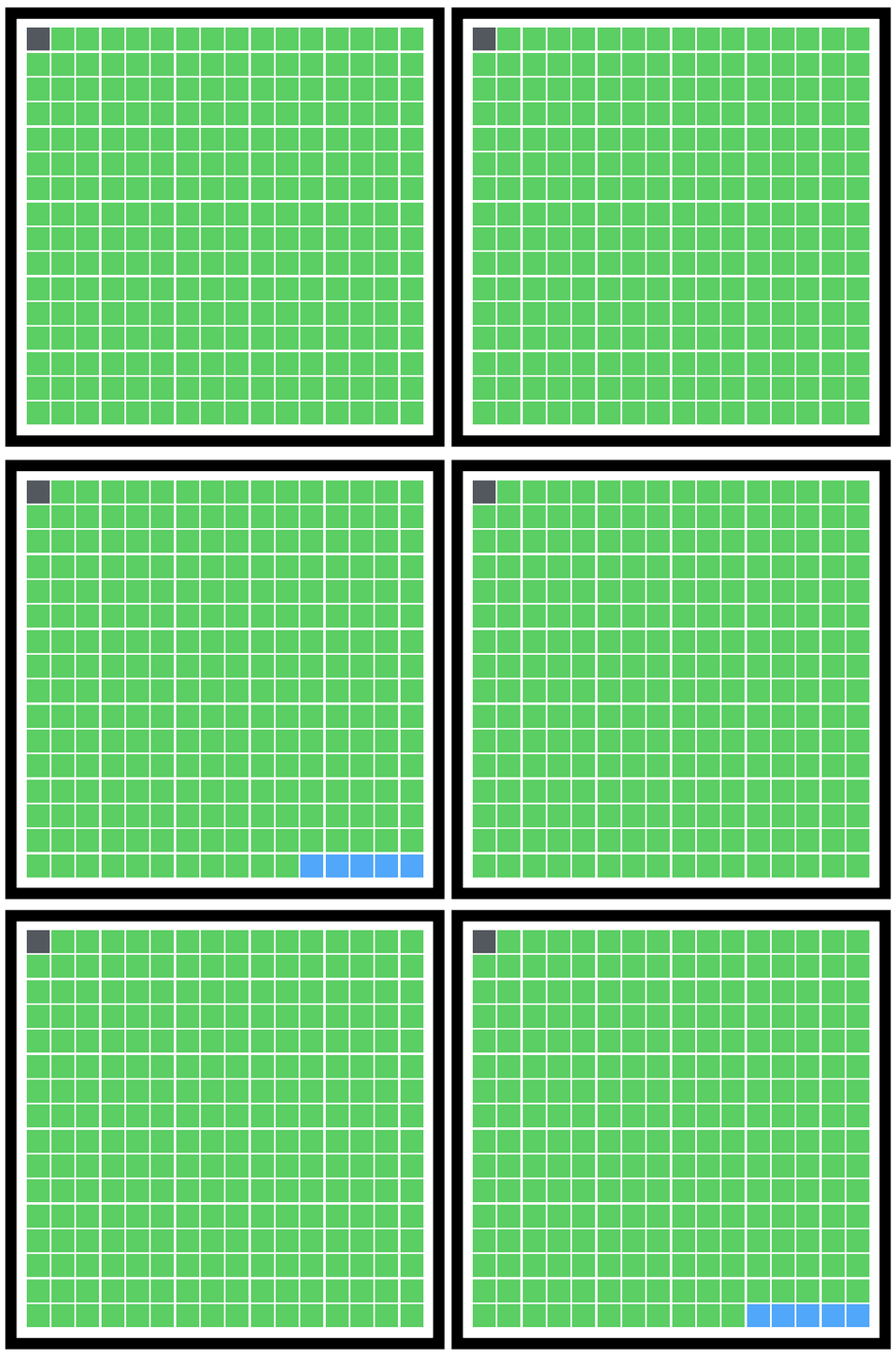
VACUUM's second job:

Prevent Postgres from running out of Transaction IDs it can safely assign.

(by freezing old row versions / tuples **ahead of time**)

pganalyze

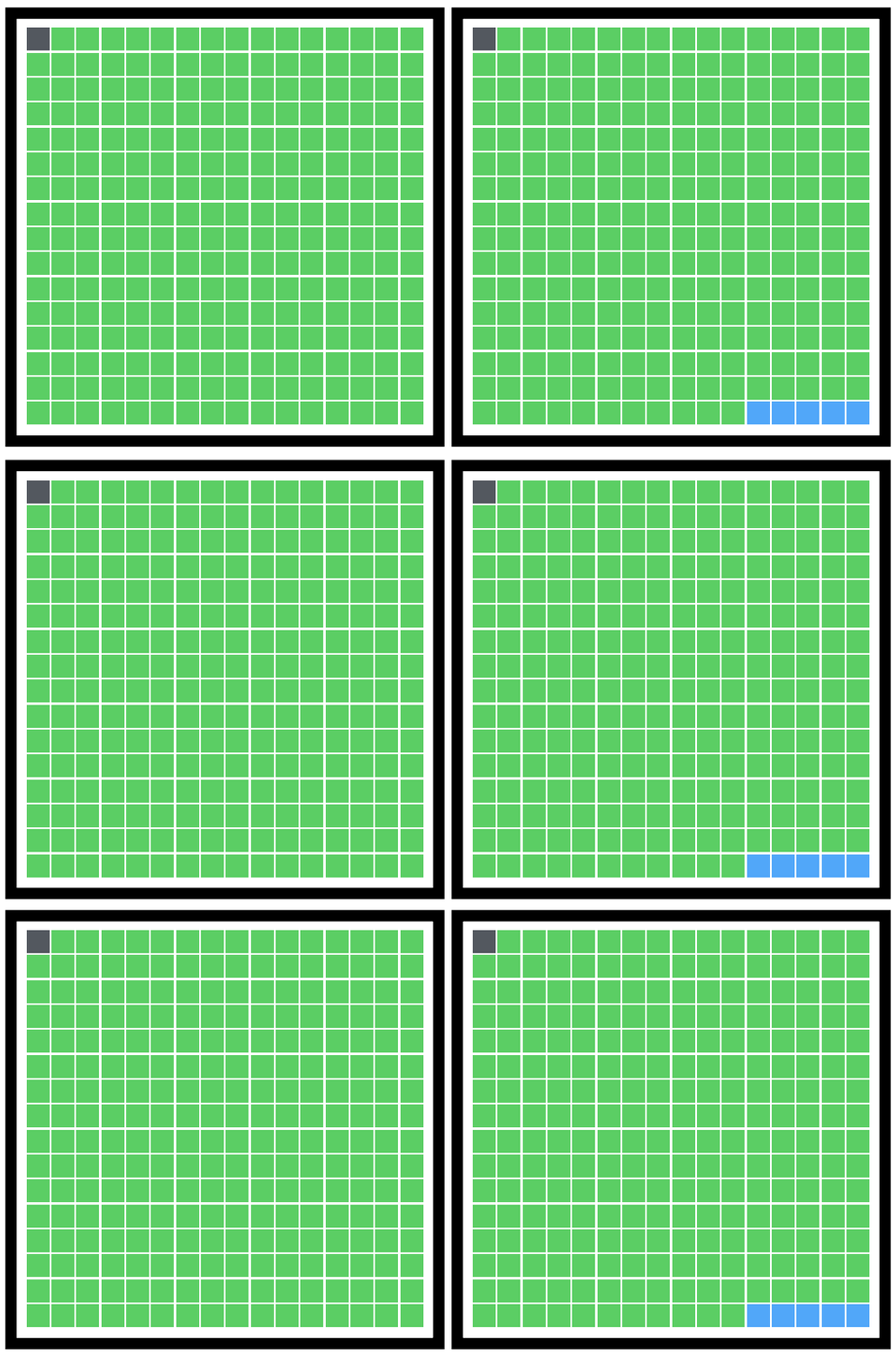
All-visible pages and slow queries



=# **SELECT COUNT(*) FROM test**
2

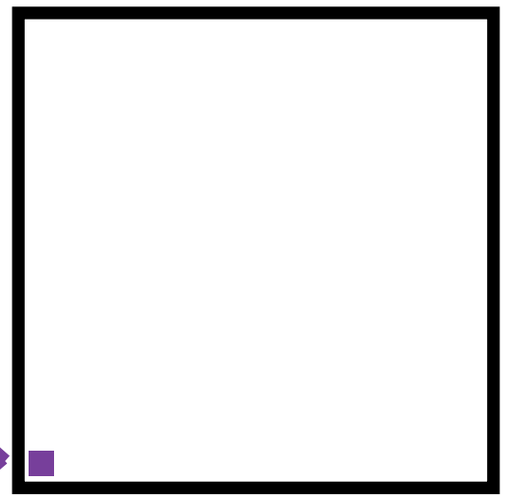
=> Has to load 6 pages (8kb each)
to find two rows with ~160 bytes each.

6x more bytes read than needed,
compared to when both rows
were on a single page.

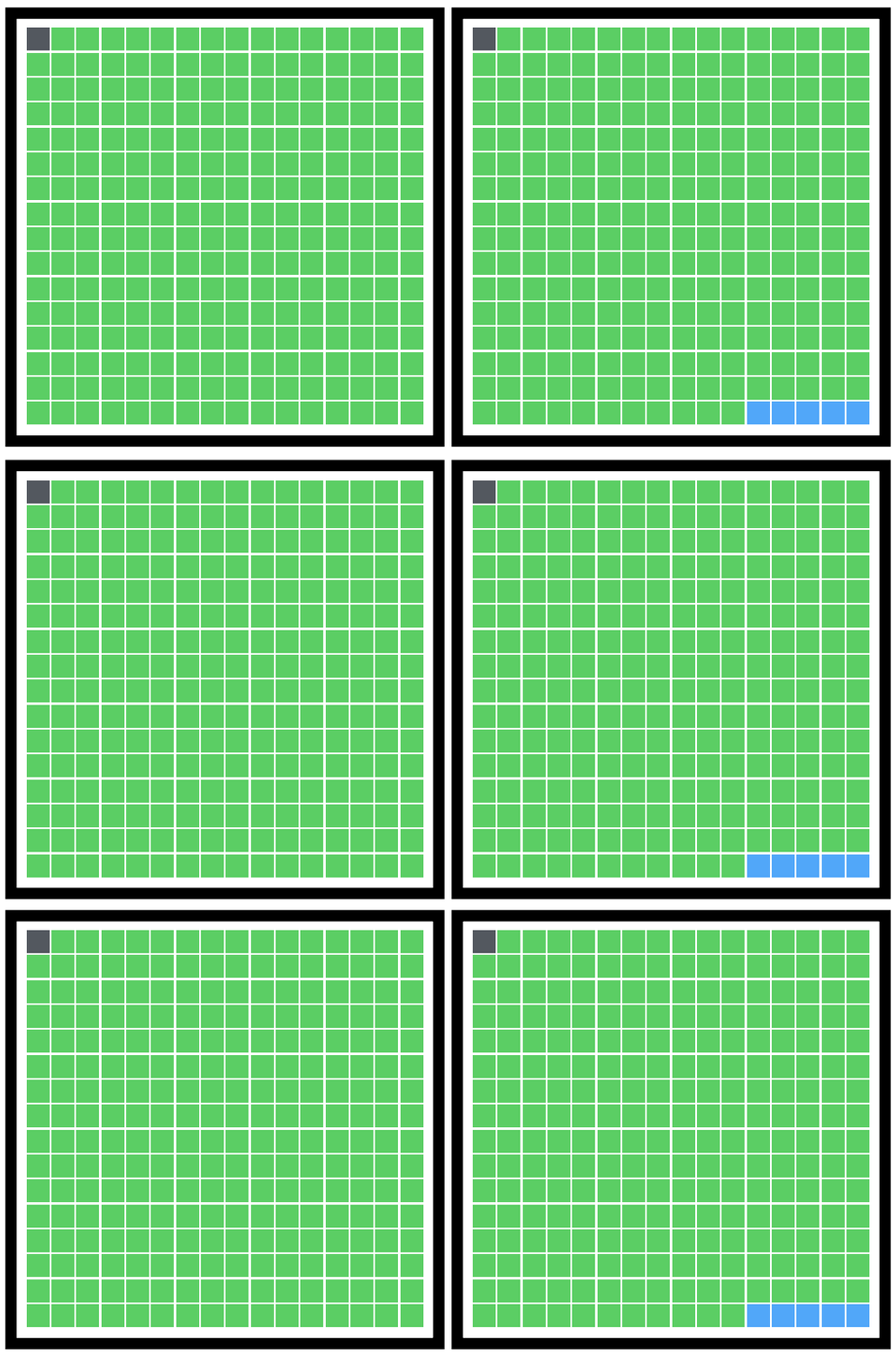


```
=# SELECT id FROM test
    WHERE data = “short";
```

CREATE INDEX USING btree ON test(data);

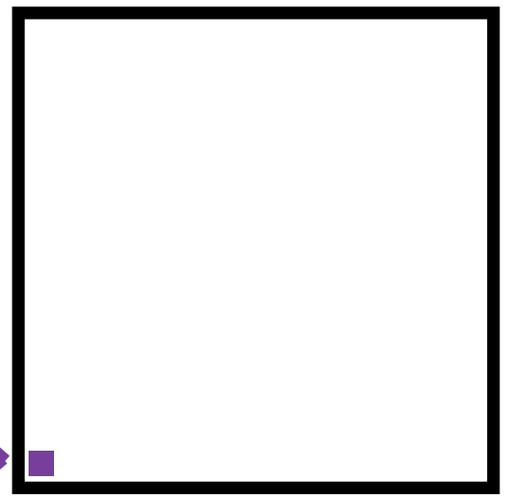


leaf index entry
data = “short”



=# SELECT COUNT(*) FROM test
WHERE data = “short";

CREATE INDEX USING btree ON test(data);



leaf index entry
data = “short”

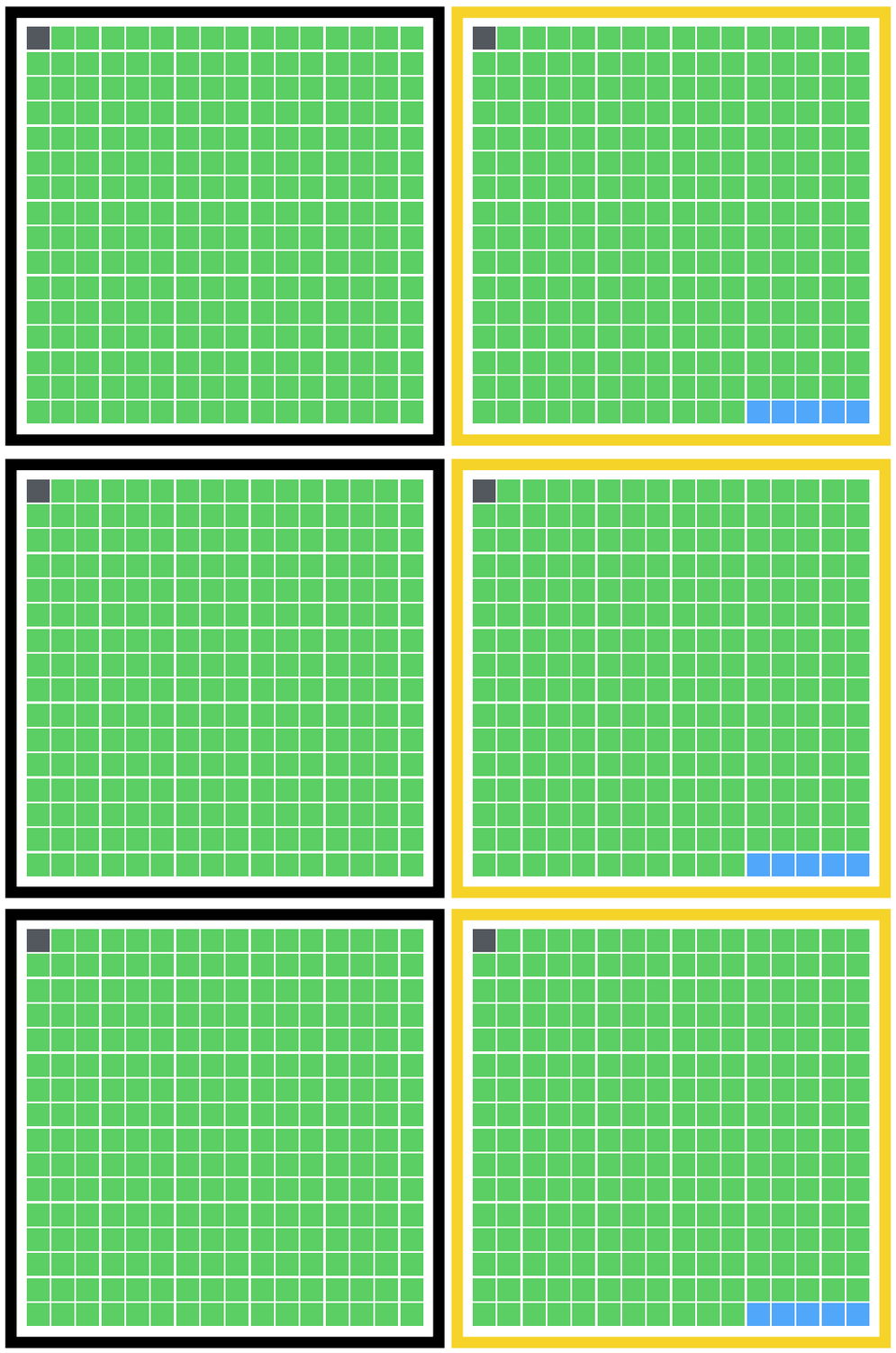
Do we need to
access the table?

"PostgreSQL tracks, for each page in a table's heap, **whether all rows stored in that page are old enough to be visible to all current and future transactions.**

This information is stored in a bit in the table's **visibility map.**

An index-only scan, after finding a candidate index entry, **checks the visibility map** bit for the corresponding heap page**. If it's set, the row is known visible and so the data can be returned with no further work."**

Postgres documentation: 11.9. Index-Only Scans and Covering Indexes



VACUUM test;

□ = page is all visible

Any modification
(e.g. UPDATE/DELETE)
causes the page to loose
its all-visible bit.

Unless there are modifications,
the next regular VACUUM
will skip all-visible pages.

```
=# SELECT COUNT(*) FROM test
   WHERE data = “short";
```

CREATE INDEX USING btree ON test(data);

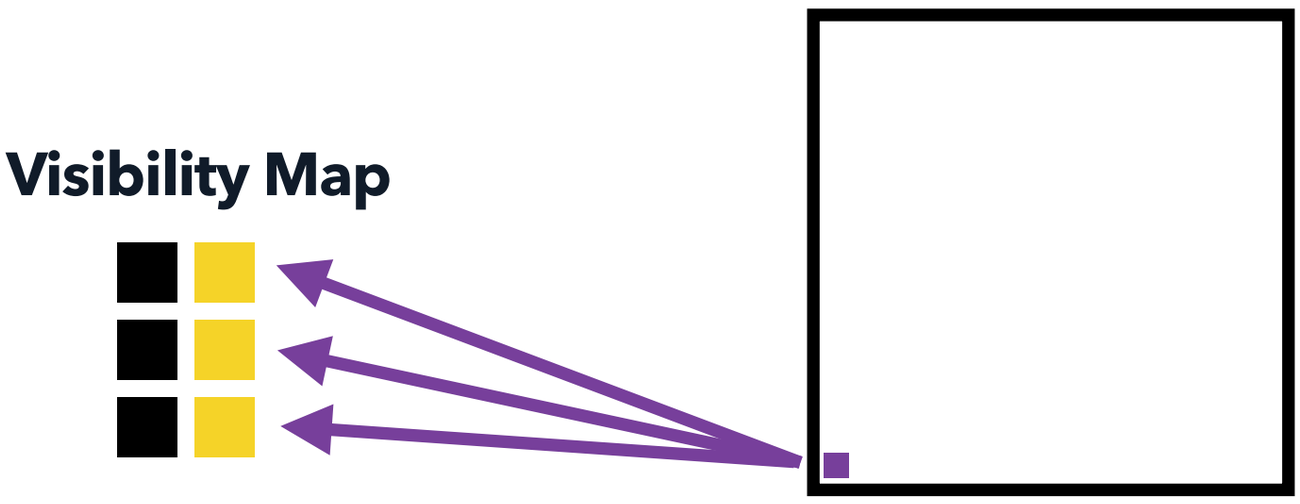


Table (Heap) Access Is Not Necessary, Because Pages Are All-Visible

25 kB (3 table pages)

VS

12 bit (visibility map)

This is called “Heap Fetches” in EXPLAIN for Index Only Scans!

```
=# EXPLAIN (ANALYZE, BUFFERS) SELECT COUNT(*) FROM query_analyses WHERE query_id = 43852814;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
---------------
 Aggregate  (cost=8.30..8.31 rows=1 width=8) (actual time=0.098..0.123 rows=1 loops=1)
   Buffers: shared hit=4
   ->  Index Only Scan using index_query_analyses_on_query_id on query_analyses  (cost=0.28..8.30 rows=1 width=0) (actual time=0.069..0.082
rows=1 loops=1)
         Index Cond: (query_id = 43852814)
         Heap Fetches: 1
         Buffers: shared hit=4
 Planning Time: 0.070 ms
 Execution Time: 0.202 ms
(8 rows)

=# VACUUM query_analyses;
VACUUM

=# EXPLAIN (ANALYZE, BUFFERS) SELECT COUNT(*) FROM query_analyses WHERE query_id = 43852814;
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
---------------
 Aggregate  (cost=4.30..4.31 rows=1 width=8) (actual time=0.051..0.076 rows=1 loops=1)
   Buffers: shared hit=3
   ->  Index Only Scan using index_query_analyses_on_query_id on query_analyses  (cost=0.28..4.29 rows=1 width=0) (actual time=0.022..0.035
rows=1 loops=1)
         Index Cond: (query_id = 43852814)
         Heap Fetches: 0
         Buffers: shared hit=3
 Planning:
   Buffers: shared hit=39
 Planning Time: 0.185 ms
 Execution Time: 0.154 ms
(10 rows)
```

The Three Main Goals of VACUUM

1. **Reducing table bloat**, by making allocated disk space used by dead row versions usable for new rows
2. **Prevent TXID/MultiXact Wraparound**, by freezing rows to allow the oldest TXID / MultiXact ID to advance
3. **Mark pages as all-visible**, to avoid repeated VACUUM work and allow Index Only Scans to skip heap fetches for visibility checks

pganalyze

Understanding autovacuum scheduling

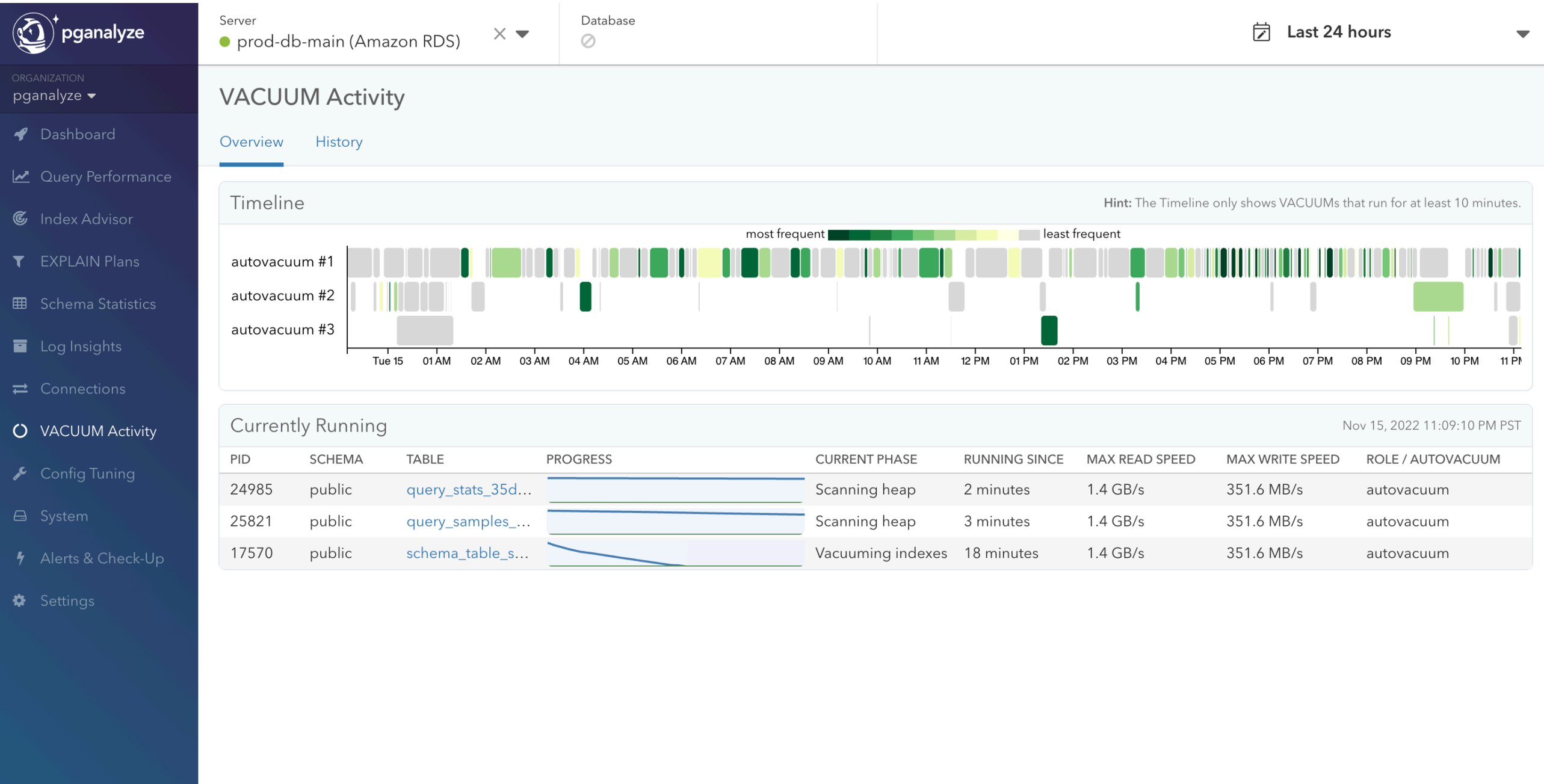
Your Postgres server has a

maximum number of autovacuum workers (autovacuum_max_workers)

1 worker handles 1 database at a time,

but multiple workers can process the same database.

Or, when visualizing this as a timeline graph over time:



Each worker process will check each table within its database and execute VACUUM and/or ANALYZE as needed.

If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers might become occupied with vacuuming those for a long period.

This would result in
other tables and databases not being vacuumed
until a worker becomes available.

Postgres Documentation - 25.1.6. The Autovacuum Daemon

We got an autovacuum traffic jam!

Let’s back up for a moment.

(don’t do this on the freeway)

How do databases and tables become eligible for autovacuum in the first place?

"We pick the database that was **least recently auto-vacuumed,** or one that needs **vacuuming to prevent Xid wraparound-related data loss.**

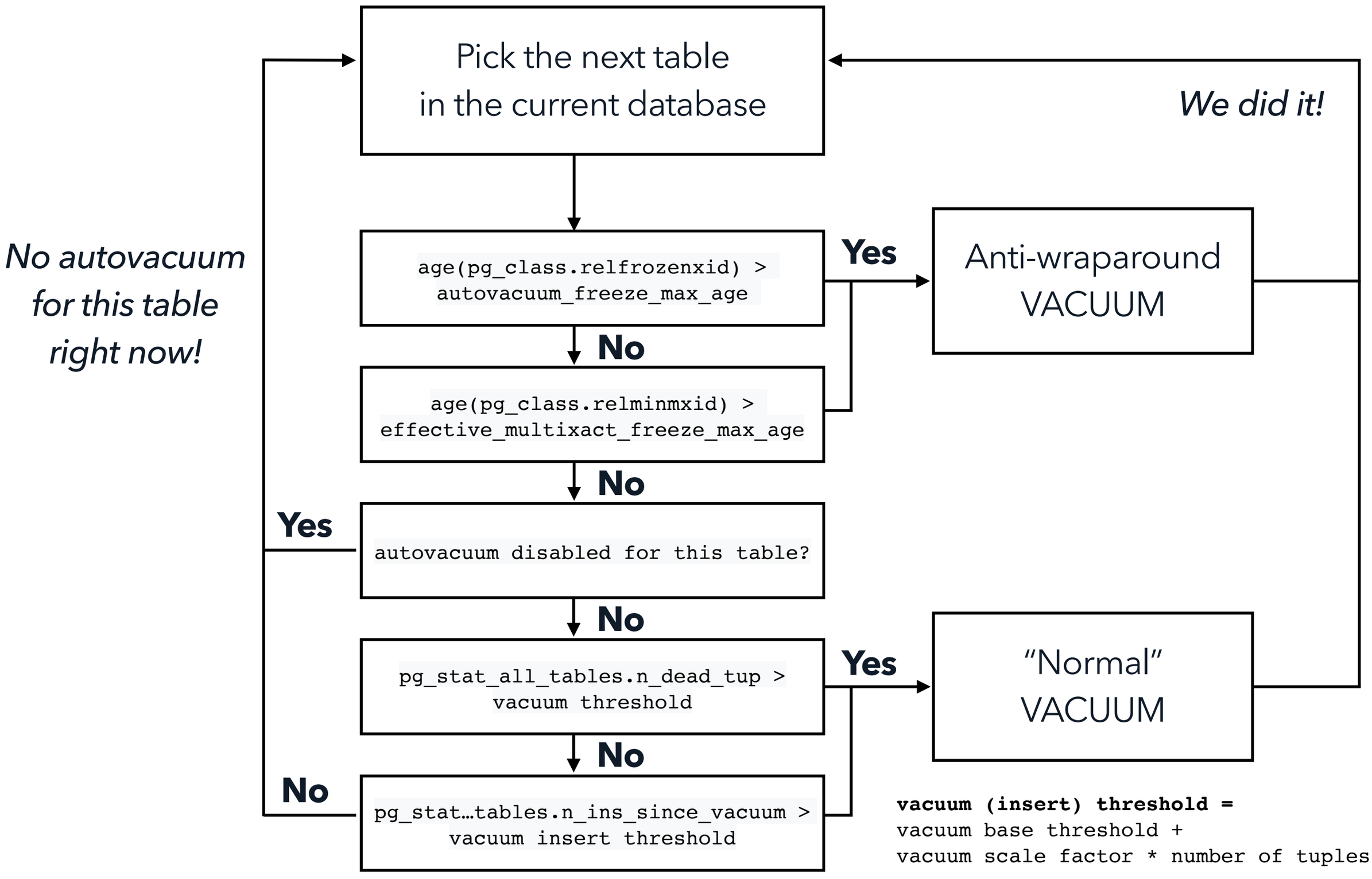
If any db at risk of Xid wraparound is found, **we pick the one with oldest datfrozenxid, independently of autovacuum times;** similarly we pick the one with the oldest datminmxid if any is in MultiXactId wraparound. "

- Postgres Source - autovacuum.c

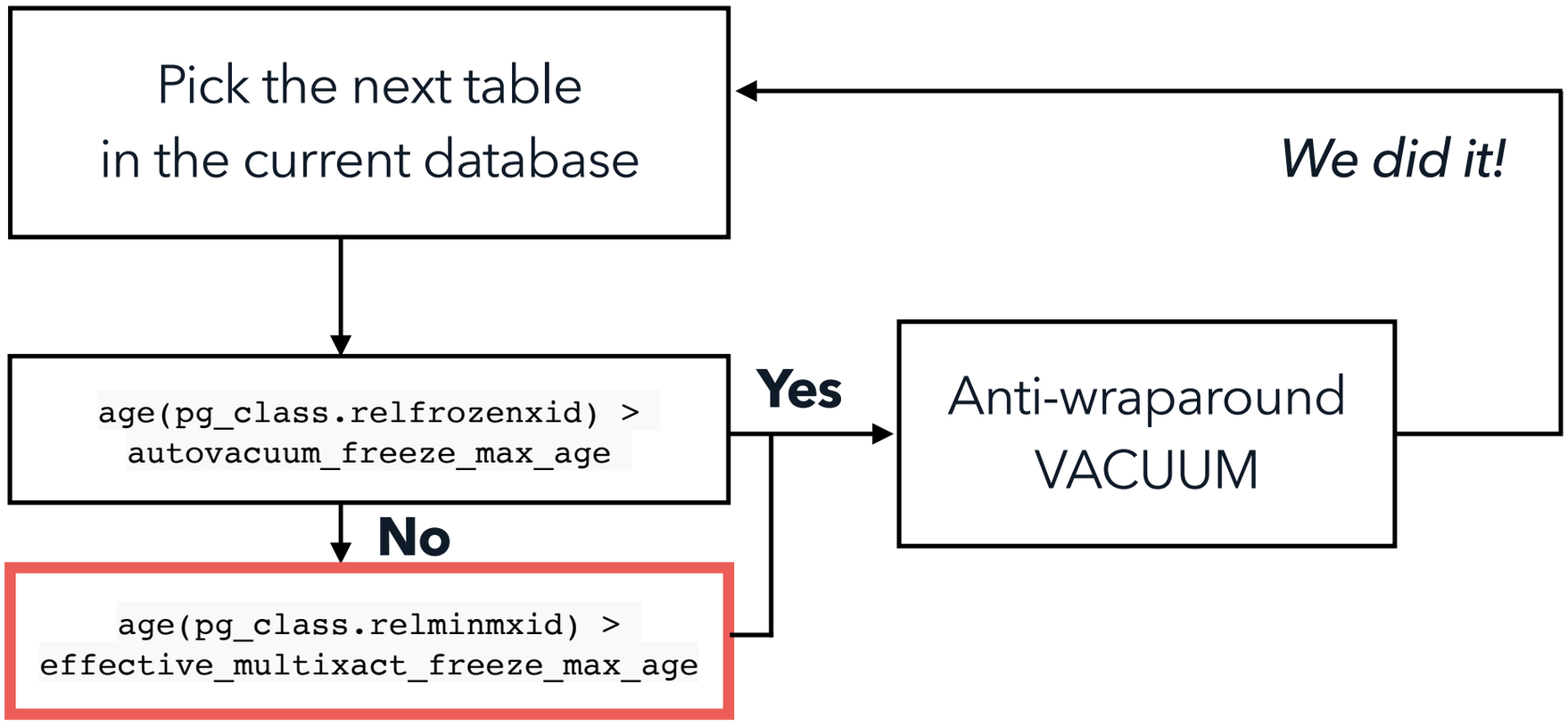
Tuning Tip:

If you have tens or hundreds of databases, **raise autovacuum_max_workers**.

Otherwise its very likely you'll end up with bloat, since anti-wraparound vacuums will take priority.



vacuum (insert) threshold =
vacuum base threshold +
vacuum scale factor * number of tuples



MultiXact IDs are special 32-bit integer IDs
that are used for **concurrent row-level locks by different transactions.**

They are rarely something you will have to deal with, except:

1) **You have a lot of them, and its causing MultiXact anti-wraparound vacuums**
2) **You have a lot of them, and the multixacts members folder exceeds 2GB in size**

The solution is to either fix your workload, or (sometimes) tuning the multixact limits.

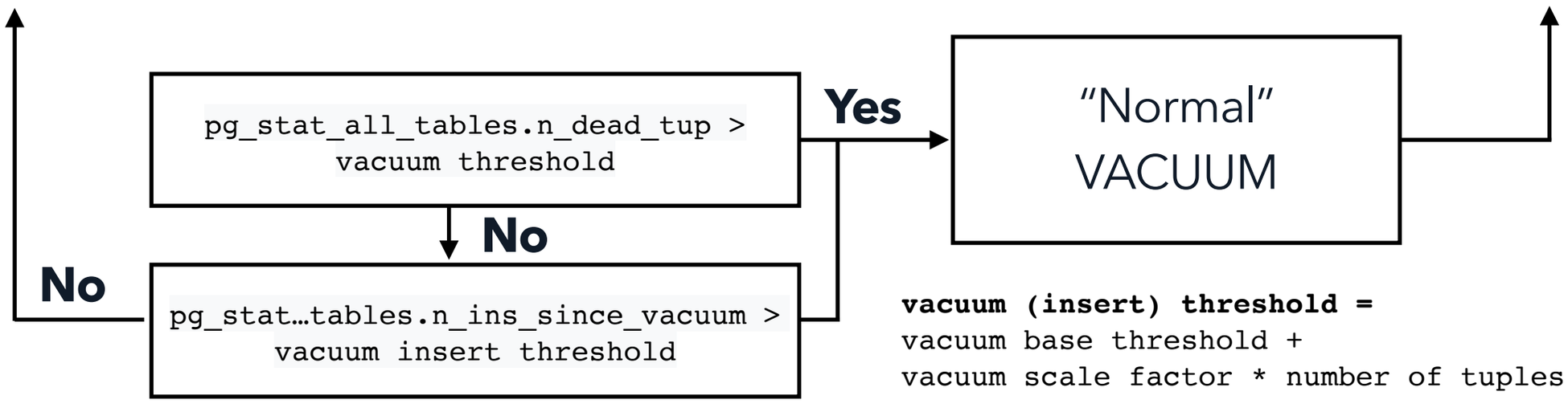
Tuning Tip:

Consider raising
autovacuum_freeze_max_age to reduce anti-wraparound vacuum frequency

(default of 200 million TXIDs is very conservative, something like 400 million is commonly recommended)

Autovacuum Scheduling

Thresholds



vacuum (insert) threshold =
vacuum base threshold +
vacuum scale factor * number of tuples

Understanding autovacuum scheduling

pganalyze

Server: prod-db-main (Amazon RDS) | Database: pgaweb | Last 7 days

ORGANIZATION: pganalyze

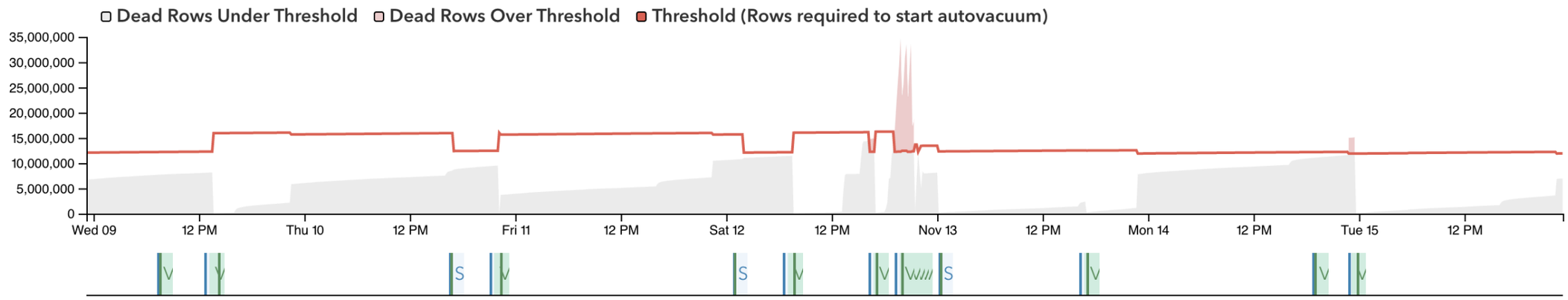
- Dashboard
- Query Performance
- Index Advisor
- EXPLAIN Plans
- Schema Statistics
- Log Insights
- Connections
- VACUUM Activity
- Config Tuning
- System
- Alerts & Check-Up
- Settings

Table: public.queries

Statistics | Partitions | Queries | Columns | Indexes | Constraints | VACUUM/ANALYZE Activity

VACUUM Activity

Avg. VACUUM Duration	39 minutes	**Number of VACUUMs / day**	2
Autovacuum Enabled	Yes	**Autovacuum Freeze Max Age**	200,000,000 / 400,000,000
Autovacuum Cost Delay	2 ms	**Autovacuum Cost Limit**	1,800
Autovacuum Threshold	50	**Autovacuum Scale Factor**	10%

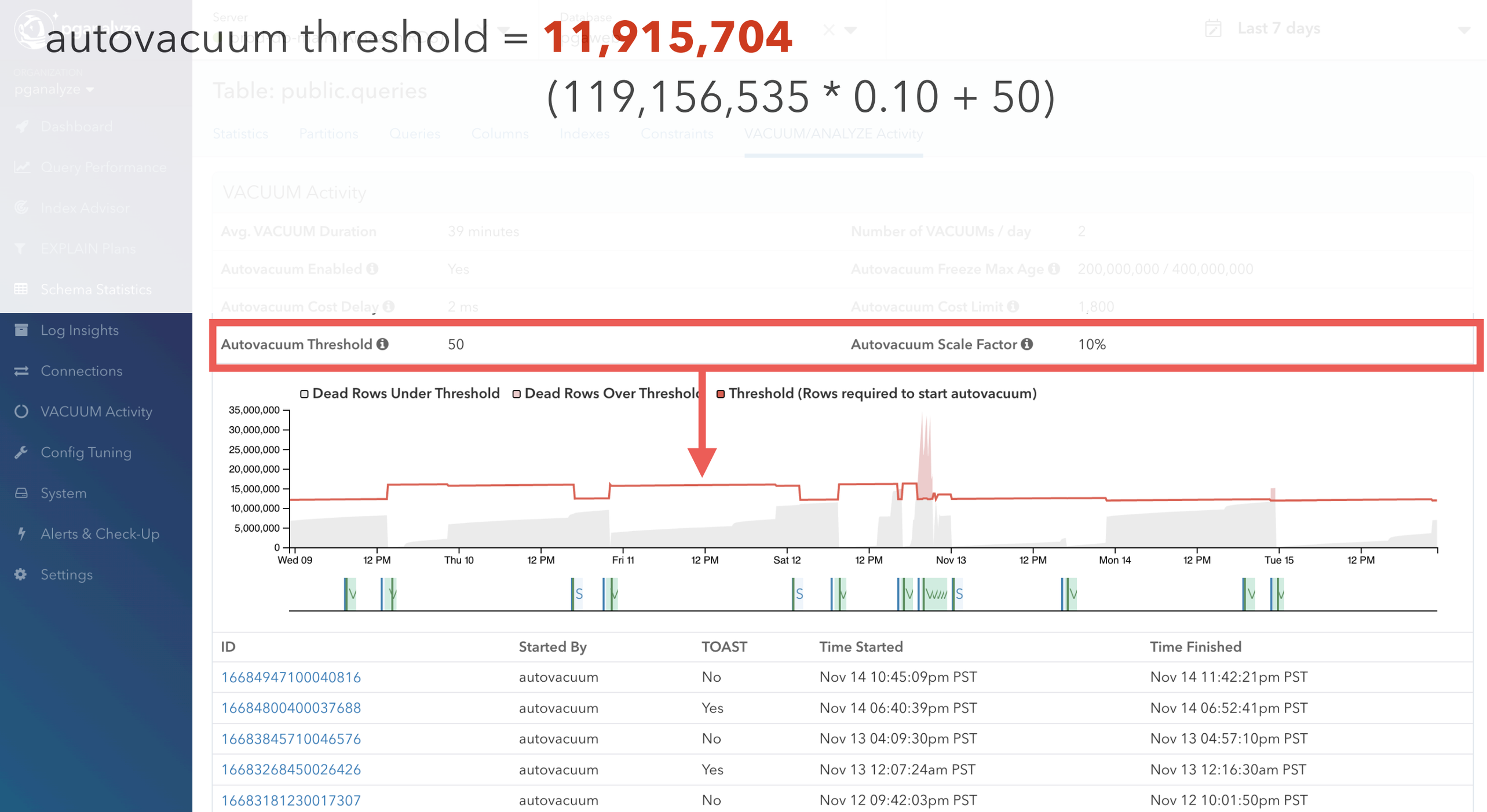


ID	Started By	TOAST	Time Started	Time Finished
16684947100040816	autovacuum	No	Nov 14 10:45:09pm PST	Nov 14 11:42:21pm PST
16684800400037688	autovacuum	Yes	Nov 14 06:40:39pm PST	Nov 14 06:52:41pm PST
16683845710046576	autovacuum	No	Nov 13 04:09:30pm PST	Nov 13 04:57:10pm PST
16683268450026426	autovacuum	Yes	Nov 13 12:07:24am PST	Nov 13 12:16:30am PST
16683181230017307	autovacuum	No	Nov 12 09:42:03pm PST	Nov 12 10:01:50pm PST

119,156,535 live tuples

autovacuum threshold = **11,915,704**

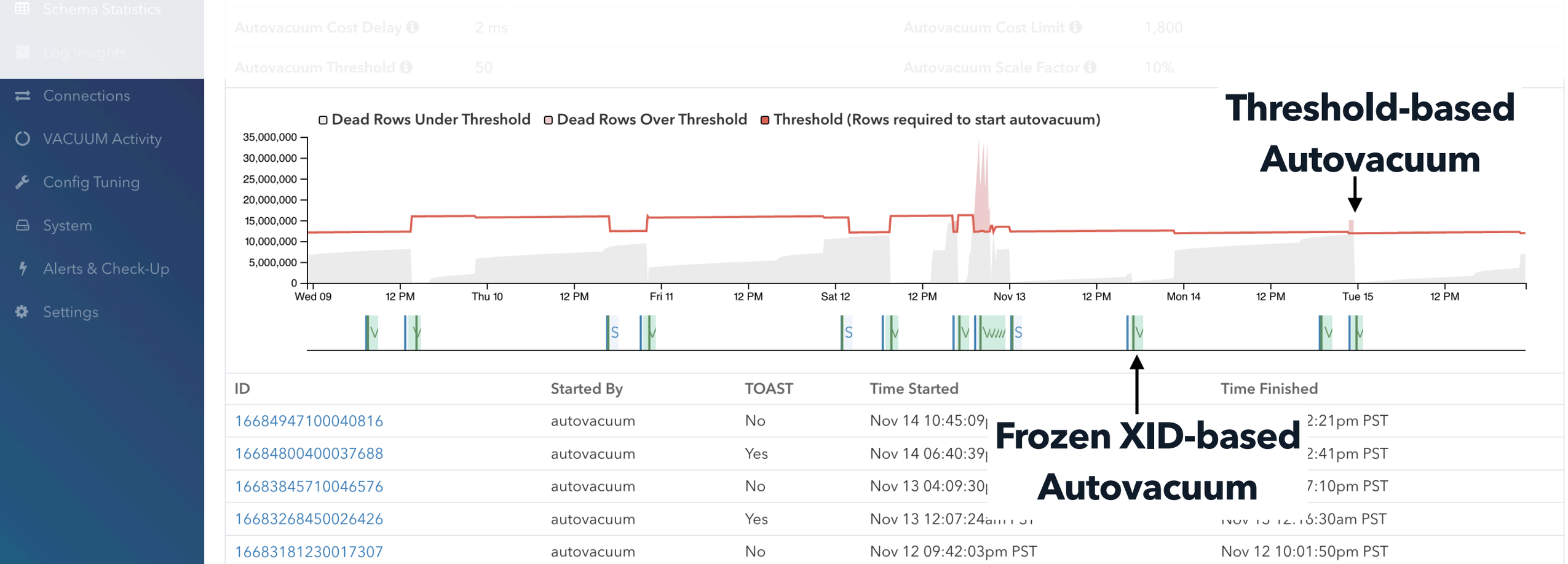
(119,156,535 * 0.10 + 50)



ID	Started By	TOAST	Time Started	Time Finished
16684947100040816	autovacuum	No	Nov 14 10:45:09pm PST	Nov 14 11:42:21pm PST
16684800400037688	autovacuum	Yes	Nov 14 06:40:39pm PST	Nov 14 06:52:41pm PST
16683845710046576	autovacuum	No	Nov 13 04:09:30pm PST	Nov 13 04:57:10pm PST
16683268450026426	autovacuum	Yes	Nov 13 12:07:24am PST	Nov 13 12:16:30am PST
16683181230017307	autovacuum	No	Nov 12 09:42:03pm PST	Nov 12 10:01:50pm PST

Threshold-based Autovacuum Triggers On The Threshold

Frozen XID-based Autovacuum Follows its Own Schedule (The Transaction ID Clock)



You can tune autovacuum thresholds

on a per-table basis

```
ALTER TABLE test SET
(autovacuum_vacuum_scale_factor = 0.05);
```

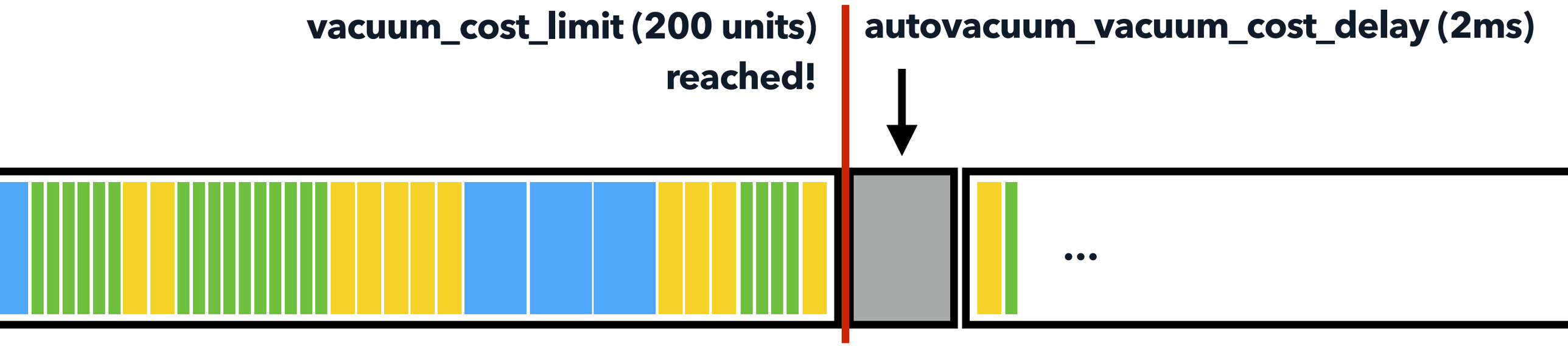
Tuning Tip:

If you have a **large table with a small active portion**, turn off the % based scale factor, and only use the threshold setting.

```
ALTER TABLE test SET
(autovacuum_vacuum_scale_factor = 0,
autovacuum_vacuum_threshold = 10000);
```

pganalyze

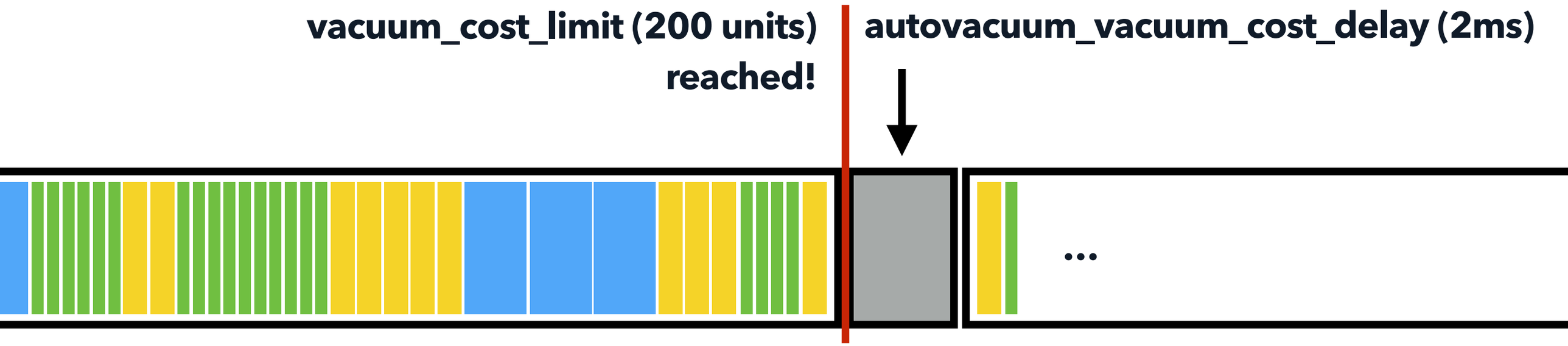
Cost delay 101



Cost delay exists to slow down autovacuum

(and VACUUM, if enabled)

to reduce impact on other activity



vacuum_cost_page_hit (1 unit)
Page was in shared buffer cache

vacuum_cost_page_read (2 units)
Page loaded from disk / OS page cache

vacuum_cost_page_dirty (20 units)
Page dirtied that will need to be written out

Server-wide autovacuum cost limit is
split between autovacuum workers

**But, if you override it for a specific table
it gets reserved fully for the worker!**

Tuning Tip:

If you are on a Postgres release older than PG12 and on hardware from the last 10 years:

Set your server-wide
autovacuum_vacuum_cost_delay to **2ms**
(New default since Postgres 12, also check any customizations to cost limit if you change this)

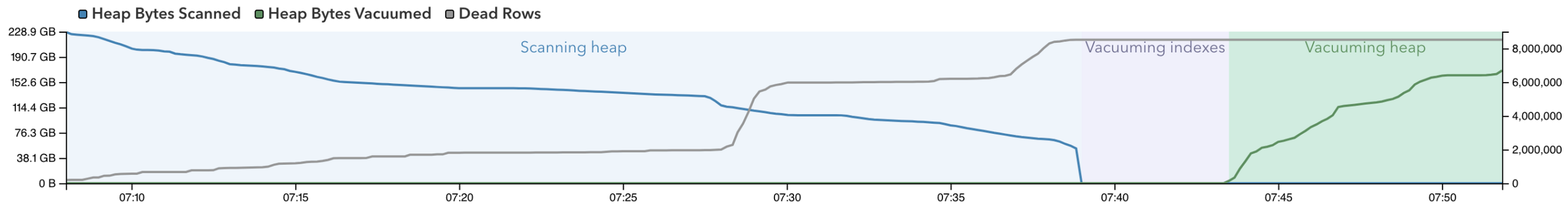
To monitor autovacuum and VACUUM progress, look at pg_stat_progress_vacuum

VACUUM Run 16683088770038640

Information

Table Name	public.queries	**Start Time**	Nov 12, 2022 7:07:57 PM PST
Duration	an hour	**End Time**	Nov 12, 2022 7:53:00 PM PST
Autovacuum	Yes	**Postgres Role**	n/a
Heap Blocks Total	231.4 GB · 30,326,029 blocks	**Max Dead Tuples / Phase**	178,956,969

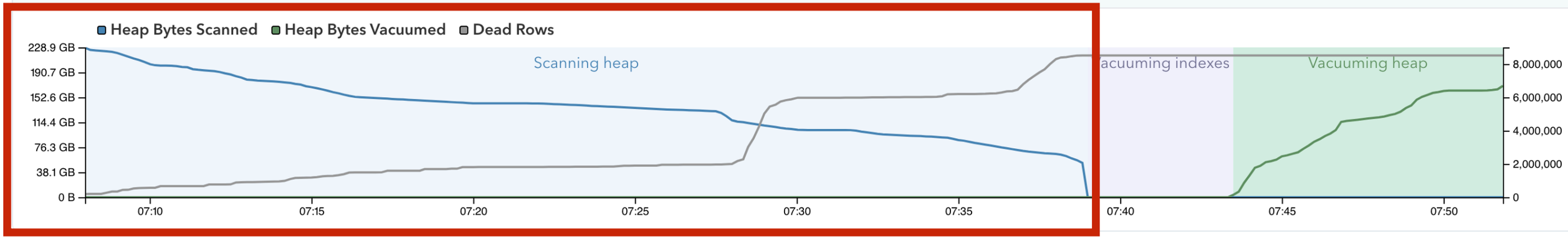
Progress



What parts of our table does autovacuum look at?

Heap Blocks Total	231.4 GB · 30,326,029 blocks	**Max Dead Tuples / Phase**	178,956,969

Progress



pganalyze

Aggressive and Failsafe VACUUMs

“Normal” autovacuum

Visits all pages that are not yet all-visible

“Aggressive” autovacuum

Visits all pages that are not yet all-frozen
(including all-visible but not yet frozen)

“Normal” autovacuum

SELECT * FROM table

WHERE NOT pg_visibility(…).**all_visible**

“Aggressive” autovacuum

SELECT * FROM table

WHERE NOT pg_visibility(…).**all_frozen**

Is this VACUUM aggressive?

age(pg_class.relfrozenxid) >

vacuum_freeze_table_age

(defaults to 150 million TXIDs)

Anti-wraparound VACUUMs
are aggressive 99.9% of the time.

Threshold-based VACUUMs are
sometimes aggressive, based on TXID age.

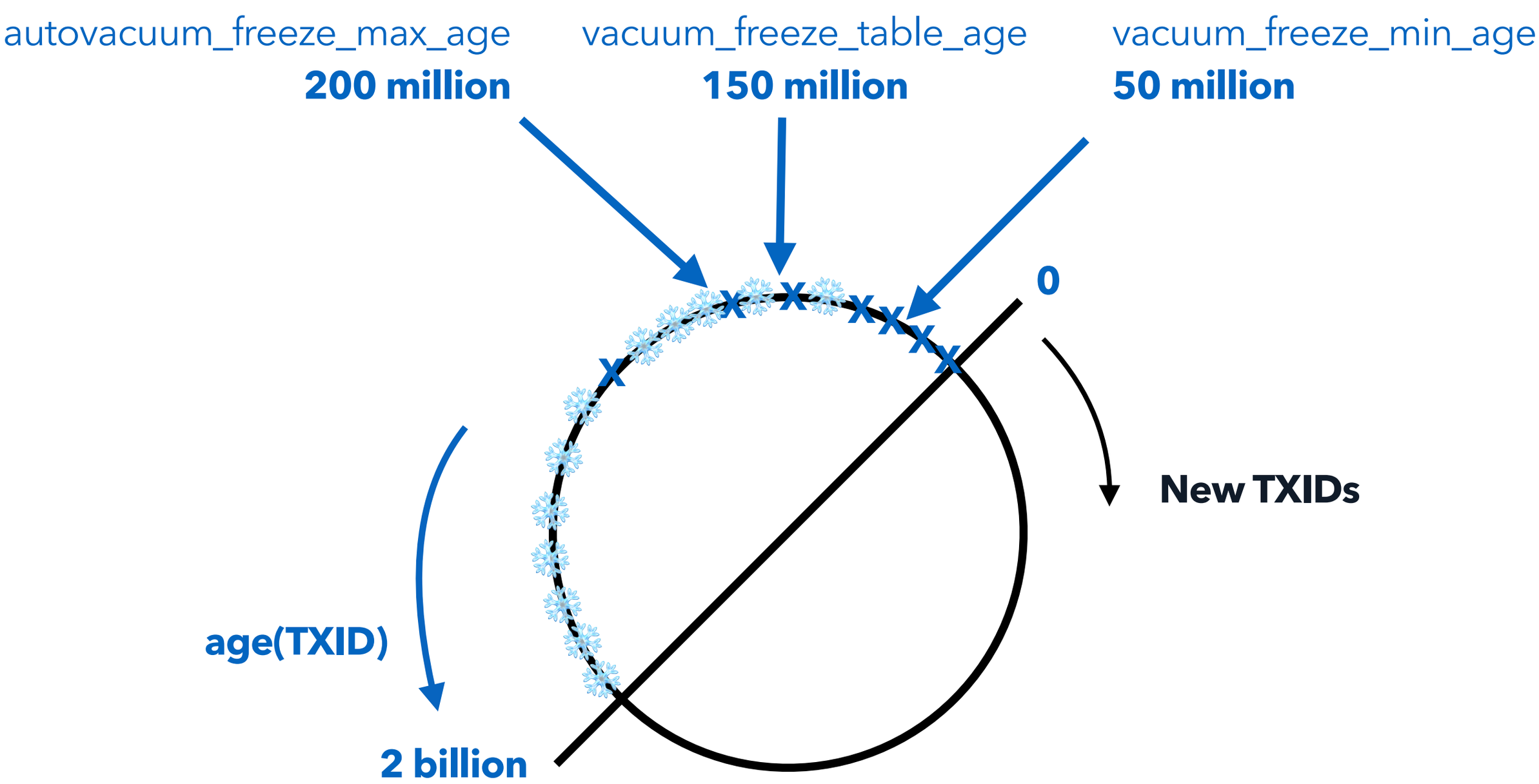
Both normal and aggressive VACUUMs will freeze pages

(but for aggressive VACUUMs its their main purpose)

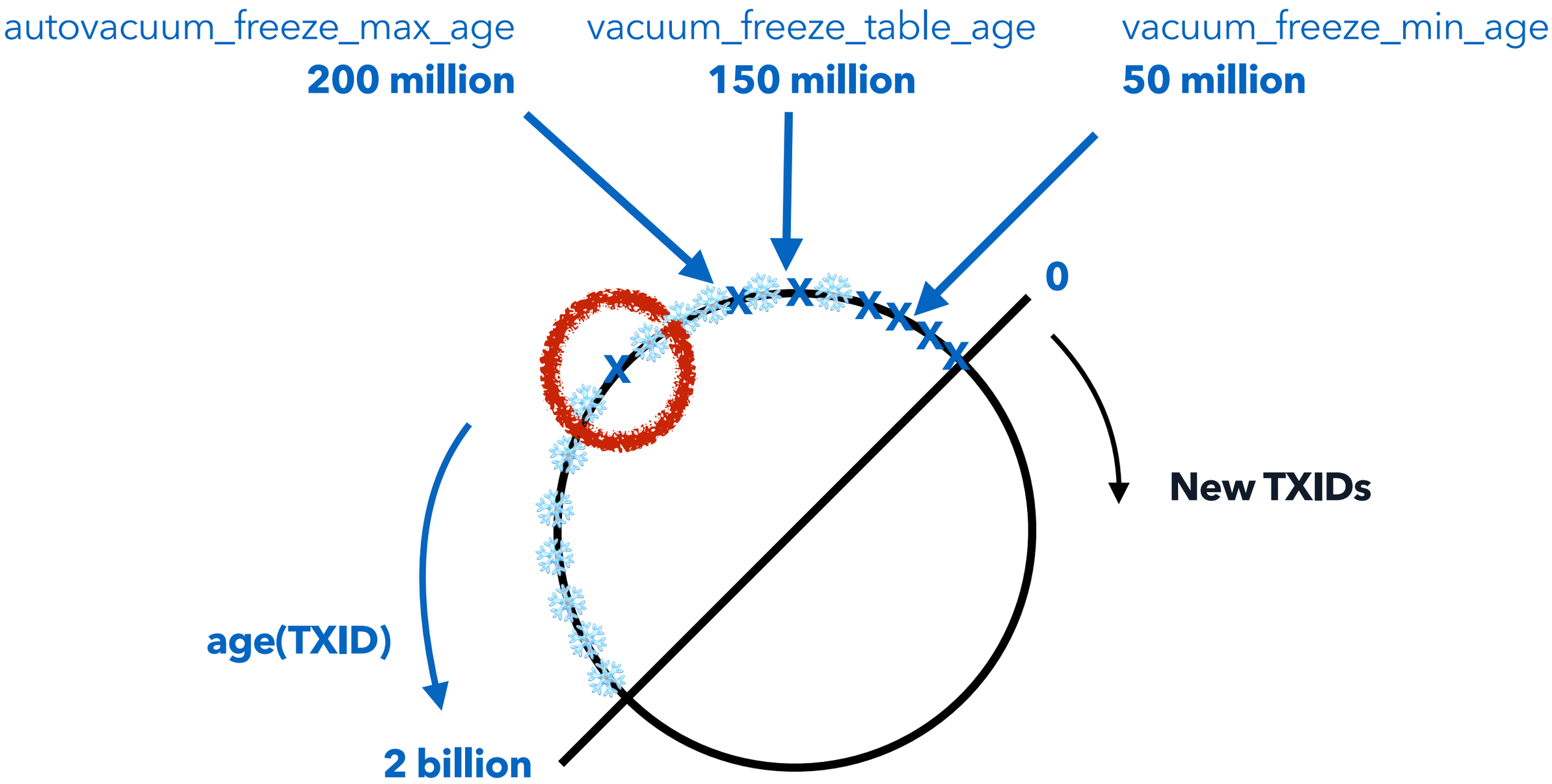
Which pages get frozen by VACUUM?

If we freeze everything, it causes a lot of write I/O -
and thats not good if we change the page soon again

VACUUM only freezes TXIDs whose age
is at least **vacuum_freeze_min_age** or higher
(defaults to 50 million)

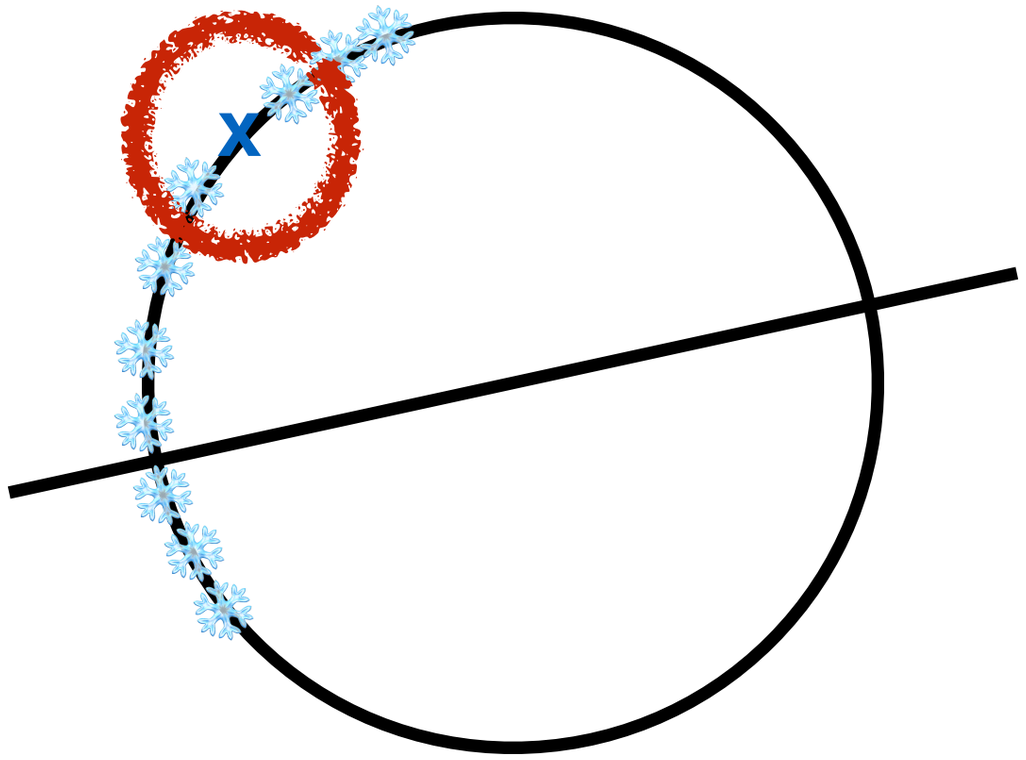

autovacuum_freeze_max_age
200 million
vacuum_freeze_table_age
150 million
vacuum_freeze_min_age
50 million
0
New TXIDs
age(TXID)
2 billion

pganalyze


autovacuum_freeze_max_age
200 million
vacuum_freeze_table_age
150 million
vacuum_freeze_min_age
50 million
0
New TXIDs
age(TXID)
2 billion

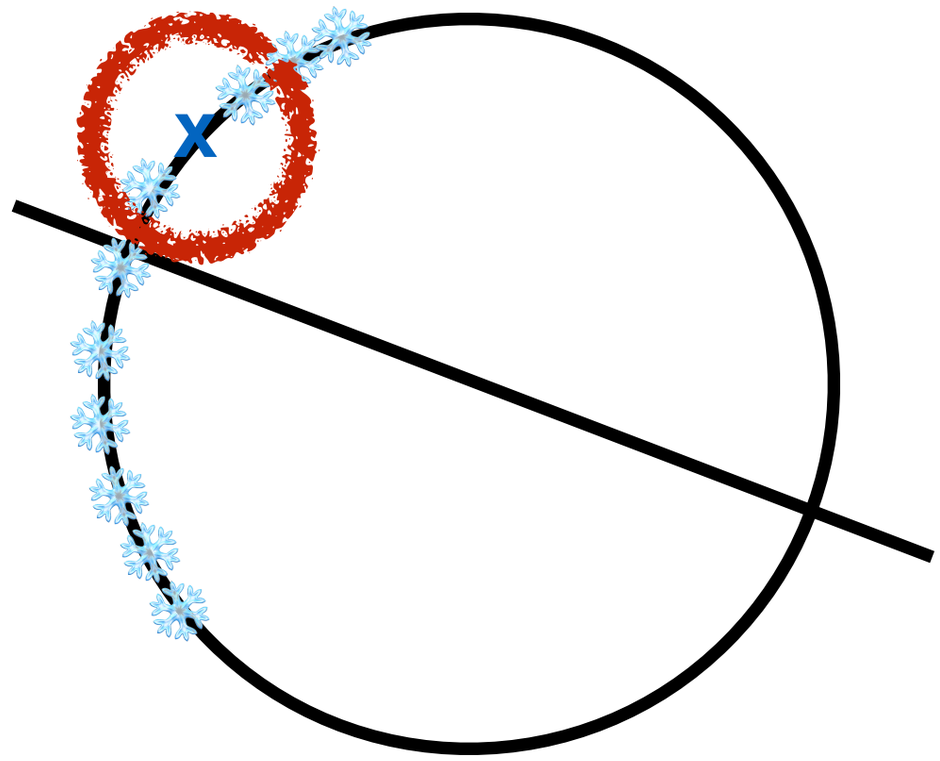
pganalyze

VACUUM is running,
but it hasn't finished :(



pganalyze

Database shut down and
we have to manually VACUUM it



pganalyze

Anti-wraparound VACUUMs

do not run at full speed

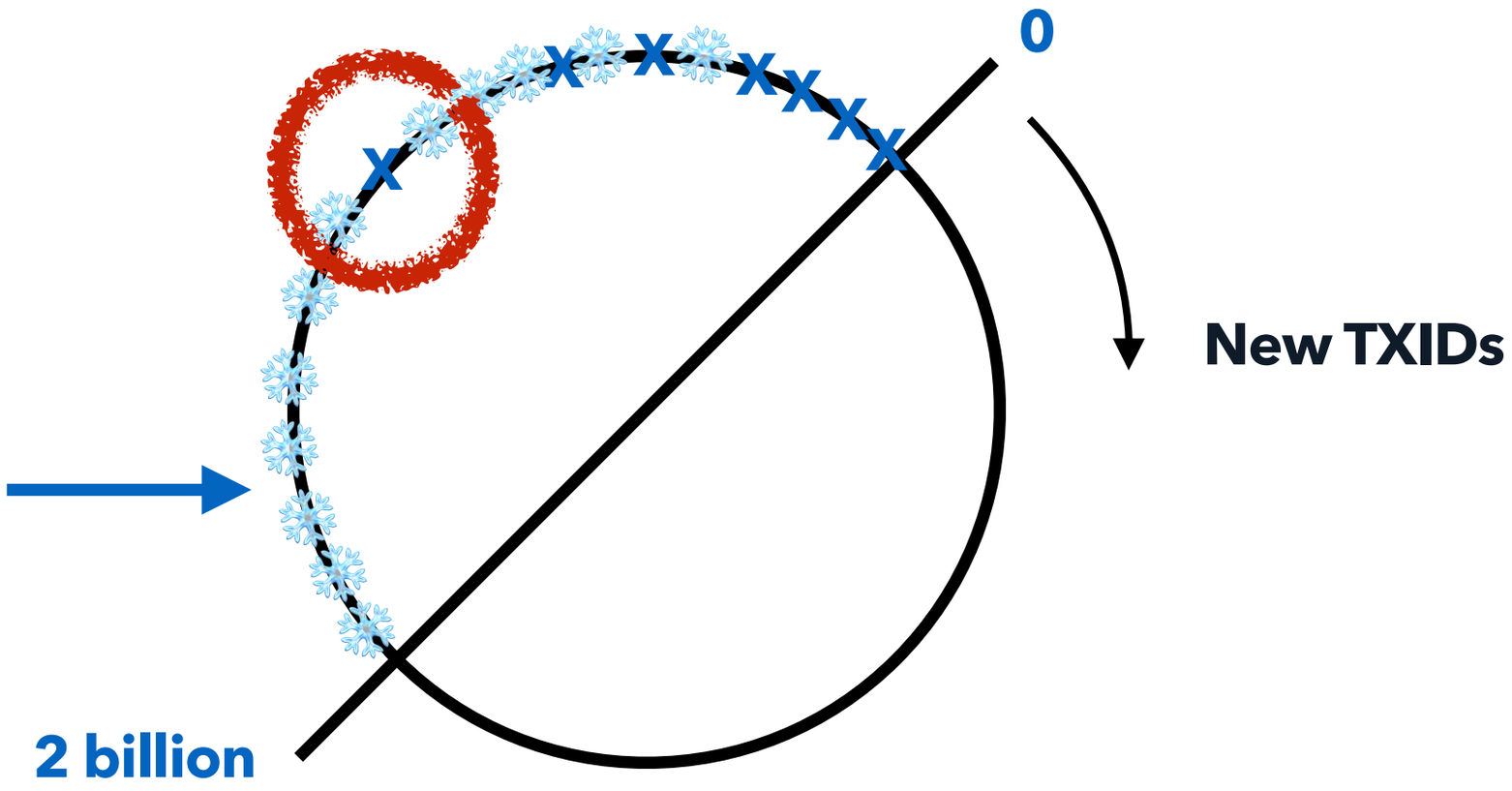
(they take cost-delay into account)

VACUUM runtime is heavily **impacted by index sizes and amount of indexes**

Enter, the Failsafe VACUUM (PG14+)

vacuum_failsafe_age

1.6 billion



pganalyze

Failsafe VACUUM =

No cost delay, skip all index vacuums

(just get those table pages frozen already!)

Tuning Tip:

For very large tables with a lot of activity,
increase the cost limit
(or reduce the cost delay)
to make sure they finish on time,
before the failsafe.

Tuning Tip:

For very large tables,
consider lowering the per table
autovacuum_freeze_min_age.

This will enable a normal VACUUM thats already
touching a page, to freeze it eagerly
(delaying a future anti-wraparound VACUUM).

pganalyze

Dead Tuples Not Yet Removable

Dead Tuples Not Yet Removable

```
LOG: automatic vacuum of table "mydb.public.mytable":
 pages: 0 removed, 21146 remain, 0 skipped due to pins
 tuples: 0 removed, 152873 remain, 26585 are dead but not yet removable
 buffer usage: 104893 hits, 54167 misses, 2165 dirtied
 avg read rate: 3.548 MB/s, avg write rate: 0.142 MB/s
 system usage: CPU 0.28s/0.97u sec elapsed 119.26 sec
```

Turn on autovacuum logs with

log_autovacuum_min_duration = 0

Dead Tuples Not Yet Removable

```
LOG: automatic vacuum of table "mydb.public.mytable":
 pages: 0 removed, 21146 remain, 0 skipped due to pins
 tuples: 0 removed, 152873 remain, 26585 are dead but not yet removable
 buffer usage: 104893 hits, 54167 misses, 2165 dirtied
 avg read rate: 3.548 MB/s, avg write rate: 0.142 MB/s
 system usage: CPU 0.28s/0.97u sec elapsed 119.26 sec
```

VACUUM just spent two minutes (119 seconds), doing absolutely nothing useful.

Causes of Dead Tuples Not Yet Removable

1) **Long running transactions (that's the big one!)**
   => Check age(backend_xmin) and age(backend_xid)
   in pg_stat_activity
2) **Forgotten replication slots**
3) **hot_standby_feedback = on**
4) **Uncommitted prepared transactions**

pganalyze

VACUUMing in the cloud
RDS & Aurora, Cloud SQL & AlloyDB

Amazon RDS & Aurora

- Both Aurora and regular RDS have **“Adaptive Autovacuum”**
- Triggers when transaction ID age reaches 500 million TXIDs
- Reduces cost delay and increases resource consumption
- Modifies **server-wide** parameters:
  - autovacuum_vacuum_cost_delay
  - autovacuum_vacuum_cost_limit
  - autovacuum_work_mem
  - autovacuum_naptime
- Its better to tune per-table VACUUM settings instead, and have regular VACUUMs do more work

Aurora Autovacuum Costs

- Aurora defaults set **vacuum_cost_page_miss** to **0** (usually 5), but increases **autovacuum_vacuum_cost_delay** to **5ms** (usually 2ms)
- Makes it cheaper to get a page from the disk than fetching something from the cache
- But makes vacuum take longer pauses between activity
- Presumably this models the higher concurrency that the Aurora storage system supports, with its dedicated page server, and encourages merging of I/O requests

Cloud SQL & AlloyDB

*“Cloud SQL added an **emergency maintenance mode** which removes the need to use the single-user mode to resolve the wraparound problem.*

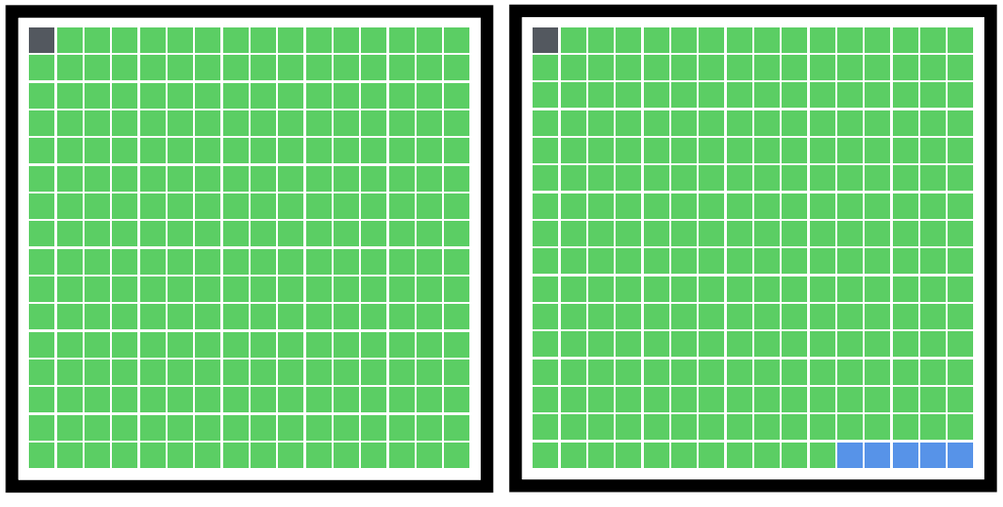
*AlloyDB further enhances and automates vacuum management. Our adaptive vacuum continuously monitors and dynamically **reallocates system resources to the vacuum process** without impacting database performance.*

*If the vacuum still can’t clean up fast enough, AlloyDB **throttles the rate of incoming transactions to avoid the database running out of transaction IDs** and becoming unavailable.”*

- PGConf.EU 2022 Interview with Google Cloud
(also check these older slides re: emergency maintenance mode)

pganalyze

Estimating and fixing table bloat



VACUUM test;

We now have 1 live tuple.

```
=# SELECT * FROM pgstattuple('test');
-[ RECORD 1 ]------+------
table_len          | 16384
tuple_count        | 1
tuple_len          | 160
tuple_percent      | 0.98
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 16160
free_percent       | 98.63
```

live tuples / page density = 0.5

Bloat!

A bad bloat problem on our own database, produced by a bug that caused a lot of UPDATEs (that were not HOT).

```
=> SELECT * FROM pgstattuple('issue_references');
-[ RECORD 1 ]------+-------------
table_len          | 501153767424
tuple_count        | 22117518
tuple_len          | 8623158256
tuple_percent      | 1.72
dead_tuple_count   | 676
dead_tuple_len     | 262123
dead_tuple_percent | 0
free_space         | 476997629164
free_percent       | 95.18
```

1.7% of data, in a 500GB table...

Table bloat =
A less than optimal “page density”

(i.e. how many live tuples are on each page,
vs how many could potentially fit there)

Bloat Estimation Queries...

ioguix / pgsql-bloat-estimation Public

Code Issues 2 Pull requests 2 Actions Projects Wiki Security Insights

master pgsql-bloat-estimation / table / table_bloat.sql

63 lines (63 loc) · 3.32 KB

```
1   /* WARNING: executed with a non-superuser role, the query inspect only tables and materialized view (9.3+) you are granted to read.
2   * This query is compatible with PostgreSQL 9.0 and more
3   */
4   SELECT current_database(), schemaname, tblname, bs*tblpages AS real_s
5     (tblpages-est_tblpages)*bs AS extra_size,
6     CASE WHEN tblpages > 0 AND tblpages - est_tblpages > 0
7       THEN 100 * (tblpages - est_tblpages)/tblpages::float
8       ELSE 0
9     END AS extra_pct, fillfactor,
10    CASE WHEN tblpages - est_tblpages_ff > 0
11      THEN (tblpages-est_tblpages_ff)*bs
12      ELSE 0
13    END AS bloat_size,
14    CASE WHEN tblpages > 0 AND tblpages - est_tblpages_ff > 0
15      THEN 100 * (tblpages - est_tblpages_ff)/tblpages::float
16      ELSE 0
17    END AS bloat_pct, is_na
18    -- , tpl_hdr_size, tpl_data_size, (pst).free_percent + (pst).dead_tuple_percent AS real_frag -- (DEBUG INFO)
19  FROM (
20    SELECT ceil( reltuples / ( (bs-page_hdr)/tpl_size ) ) + ceil( toasttuples / 4 ) AS est_tblpages,
21      ceil( reltuples / ( (bs-page_hdr)*fillfactor/(tpl_size*100) ) ) + ceil( toasttuples / 4 ) AS est_tblpages_ff,
22      tblpages, fillfactor, bs, tblid, schemaname, tblname, heappages, toastpages, is_na
```

Look scary and complicated

But the idea is simple.

Based on the:

- Number of tuples on the table (per ANALYZE / VACUUM)
- Average column width (as collected by ANALYZE)
- Alignment of columns (as defined by the platform)
- FILLFACTOR (as configured on the table)

What is the **optimal size of the table**,
if all pages were packed efficiently?
(and how does current table size compare to that?)

Best practices for bloat estimation:

- Prefer pgstattuple functions over bloat estimation queries
- Run ANALYZE before running bloat estimation queries
- Avoid if you have very large columns (e.g. JSONB), since the average column width is likely to be incorrect
- Pay close attention to the math behind index bloat estimation, it can often times be completely off
  (better to test a REINDEX on a copy of the database)

Using VACUUM on a bloated table does not fix the bloat.

VACUUM does not move around rows in a table

(it just marks dead row space as reusable,
and pages as all-visible or all-frozen)

To fix bloat, use pg_repack!

It’s like VACUUM FULL, but made for people who are running production databases :-)

CREATE EXTENSION pg_repack; on your database
+ install pg_repack client side utilities on a virtual machine.

```
pg_repack -k -j 5
-h mydb.myaccount.us-east-1.rds.amazonaws.com -d mydb -U myuser
-t issue_references
```

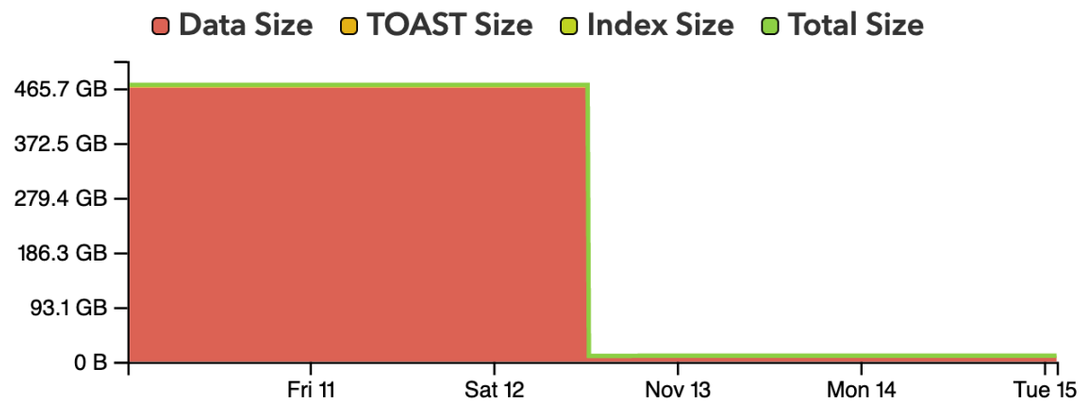
This is an online operation,
except for a short exclusive lock at the end
(similar to REINDEX CONCURRENTLY)

When VACUUM didn't do its job,

you probably need pg_repack to fix it.

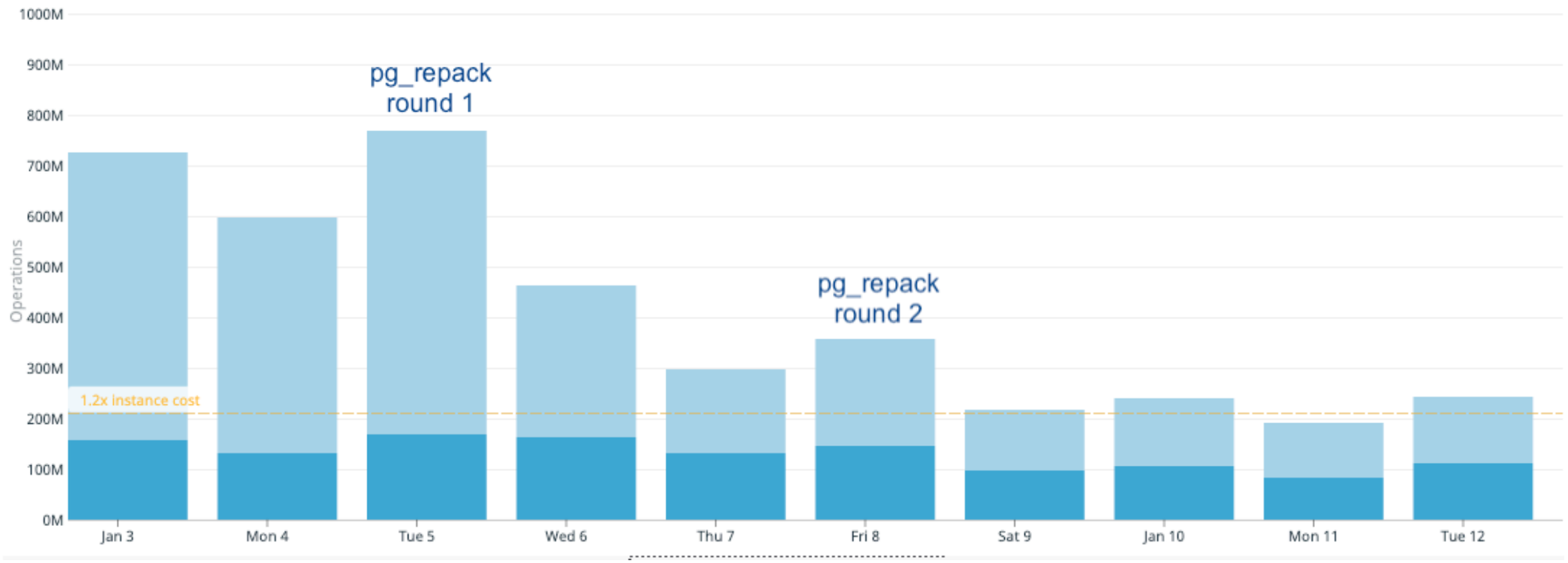
```
-[ RECORD 1 ]------+-------------
table_len          | 501153767424
tuple_count        | 22117518
tuple_len          | 8623158256
tuple_percent      | 1.72
dead_tuple_count   | 676
dead_tuple_len     | 262123
dead_tuple_percent | 0
free_space         | 476997629164
free_percent       | 95.18
```

```
-[ RECORD 1 ]------+-----------
table_len          | 9559777280
tuple_count        | 22629063
tuple_len          | 9047624099
tuple_percent      | 94.64
dead_tuple_count   | 166446
dead_tuple_len     | 132483269
dead_tuple_percent | 1.39
free_space         | 183796600
free_percent       | 1.92
```



pganalyze

Reduce Bloat, Tune Autovacuum => Save Money



“Migrating to Aurora: easy except the bill"

pganalyze

pganalyze

Three chapters we couldn't fit in today:

1) HOT updates and opportunistic pruning
2) Index vacuuming and index bloat
3) autovacuum_work_mem tuning

thanks!

Get a free trial of pganalyze

PGANALYZE.COM

Get free pganalyze eBooks and Postgres blog posts

PGANALYZE.COM/RESOURCES **PGANALYZE.COM/BLOG**

We will send you an email with a recording of this webinar tomorrow!

Feel free to get in touch with us at pganalyze.com/contact