

When TOAST Burns: Diagnosing & Surviving OLD Exhaustion & Collision in PostgreSQL

Mohamed Ali

Sr Database Engineer @  Databricks [ Lakebase]

POSTGRES SUMMIT US 2026



A visual deep-dive into PostgreSQL **TOAST** — how oversized values are stored, and how a busy table can exhaust its chunk_id OIDs. Use the arrow keys or **Next** to begin.

ABOUT THE SPEAKER

Who am I?

I'm **Mohamed Ali**, a Senior Database Engineer at **Databricks**, where I work on **Lakebase**.

Experience

Over **19 years** working with **Oracle** and **PostgreSQL** — including nearly **10 years at AWS RDS** on PostgreSQL engine development for **Amazon RDS for PostgreSQL** and **Aurora PostgreSQL**.

Open source

- pg-collector — github.com/awslabs/pg-collector ↗
- PG Counter Metrics (PGCM) — github.com/awslabs/pg-counter-metrics ↗

Agenda

Where we're headed — from how PostgreSQL stores oversized values, to why a busy table runs out of OIDs, and what to do about it.

01 The 8 kB problem

Why a row can't span pages — and where big values must go

02 What TOAST is & storage modes

PLAIN, EXTENDED, EXTERNAL, MAIN — and how to set them

03 Shrinking a row

Compression and the 4-round out-of-line eviction algorithm

04 Inside the TOAST table

Chunks, chunk_id, chunk_seq, and the toast identity

05 How chunk_id OIDs are assigned

One cluster-wide counter feeding every TOAST table

06 Issue #1: TOAST OID exhaustion

What it is, why it happens, and how PostgreSQL logs the search

07 Diagnose & identify

Symptoms, the server log, and chunk_id density

08 Wait events

What backends wait on — standard PostgreSQL and Lakebase

09 Reproduce & mitigate

Force it, then clean up, archive, or partition

10 Issue #2: collision after OID wraparound

Transient, localized stalls even when space is free

11 Wrap-up & the upstream fix

Collision vs exhaustion — and the 8-byte TOAST fix (v19 → v20)

Where does a 1 MB value go inside an 8 kB page?

PostgreSQL keeps every row inside a fixed 8 kB page, and a row is not allowed to span pages. So what happens when you insert a huge JSON/JSONB document, a long TEXT block, or a big BYTEA blob?

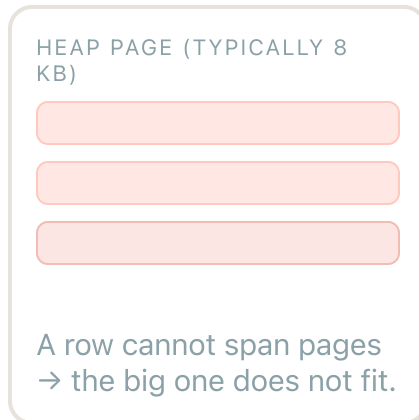
The answer is **TOAST**. Let's build up the whole picture, one step at a time.

WHY 8 KB?

The page size is a compile-time constant, `BLCKSZ`, defaulting to **8192 bytes** — a balance between I/O efficiency, WAL overhead, and wasted space. It **can** be changed, but only by rebuilding PostgreSQL from source (`--with-blocksize, 1–32 kB`) plus a fresh `initdb`.

What is TOAST?

The **O**versized-**A**tttribute **S**torage **T**echnique — PostgreSQL's built-in way to store values too large to fit comfortably in a normal row.



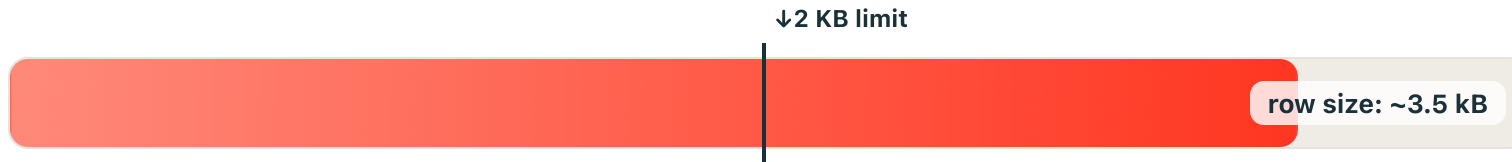
The problem: a single wide row would blow past the (typically 8 kB) page limit.

TOAST's job is to take the bulky part of a value **out of the main row** and stash it in a hidden companion table, so the main row stays small.

Only **variable-length** types can be toasted: text, bytea, json/jsonb, numeric, arrays, and similar. Fixed-width types like int never are.

When does a row get TOASTed?

Only when the row is too wide. The trigger is `TOAST_TUPLE_THRESHOLD = 2032 bytes` (a quarter of an 8 kB page). Rows under that are left alone.



The moving figure is the current **row size**. Watch it grow as more data is added →

Over threshold! TOAST kicks in — compress first, then move out-of-line if needed.

2 KB OR 2032 B?

Docs say "normally **2 kB**" — that's the rounded name. The exact value is **2032 bytes**. It is not a clean 2048 (2×1024) because the threshold is a quarter page *minus* per-page overhead: (8192 – page header – 4 line pointers) / 4, aligned down. So "2 kB" and "2032 B" are the same threshold — one rounded, one exact.

Not a setting. `TOAST_TUPLE_THRESHOLD` is a compile-time C macro (2032 B) that *triggers* the TOAST path — you cannot change it with `SET`, `postgresql.conf`, or `ALTER TABLE`; only by recompiling. The knob you *can* turn is the per-table `toast_tuple_target`, which sets how small TOAST tries to shrink the row once triggered.

What can happen to a column

Each column has a **storage mode** that decides whether it may be compressed, moved out-of-line, or both. Set it with:

```
ALTER TABLE table_name ALTER COLUMN column_name SET STORAGE storage_type;
```

PLAIN	EXTENDED default	EXTERNAL	MAIN
Compress no	Compress yes	Compress no	Compress yes
Out-of-line no	Out-of-line yes	Out-of-line yes	Out-of-line last resort
fixed-width types	most varlena types	faster substrings, more storage	keep inline if possible

The default, **EXTENDED**, is why most large text/JSON values are automatically compressed and, if needed, pushed out-of-line without you doing anything.

EXTERNAL stores out-of-line but skips compression, so *substring* reads on wide text/bytea are faster — only the needed slice is fetched (an uncompressed value can be read in parts) — at the cost of more storage. **MAIN** prefers to keep the value inline and compressed; it still goes out-of-line, but only as a last resort when the row otherwise won't fit on a page.

PLAIN forbids both compression and out-of-line storage — so a value that can't fit inline has nowhere to go. Force PLAIN on a wide text and a big insert fails with `row is too big: size ..., maximum size 8160` — the single row (tuple) exceeds the largest size that fits on one ~8 kB page.

```
DROP TABLE IF EXISTS big_table;
CREATE TABLE big_table (
  id BIGSERIAL PRIMARY KEY,
  text TEXT
);
ALTER TABLE big_table ALTER COLUMN text SET STORAGE PLAIN;

INSERT INTO big_table (text)
SELECT repeat(md5(random()::text), 1000000);
-- ERROR:  row is too big: size 32000040, maximum size 8160
```

How TOAST shrinks a wide row: **compress** first, then **move out**

Once a row is over the 2032 B threshold, TOAST has two tools to get it back under:
compress the value, and **move** it out of the main table. It always tries the cheaper one — compression — first, and only moves the value out-of-line if that isn't enough.

1. Compress

Shrink the value in place with pglz or lz4.



Fits now?

If the row is back under 2032 B, stop here.



2. Move out-of-line

Still too big → store the value in the TOAST table.

Compression can be enough on its own. Out-of-line storage only happens if the value is *still* too large after compressing (and if the column's storage mode allows it).

The target is tunable. TOAST keeps shrinking the row until it drops under the target size — by default the same **2032 B** threshold. You can change that goal per table with the `toast_tuple_target` storage parameter (128–8160 B):

```
ALTER TABLE table_name SET (toast_tuple_target = 4096);
```

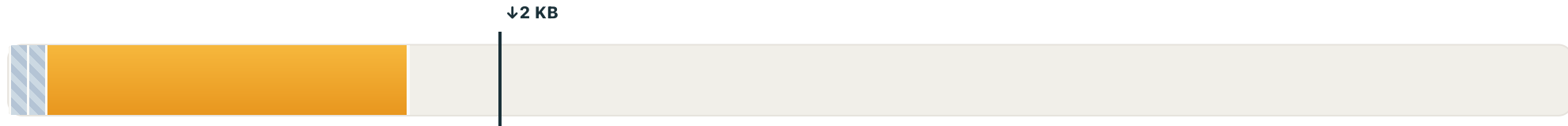
The algorithm is tunable too. lz4 has been available since PostgreSQL 14 — usually faster with a better ratio than pglz. In **PostgreSQL 19** it becomes the **default**: when the server is built with LZ4 support (the normal case), `default_toast_compression` now boots to lz4 instead of pglz. You can still override it per column:

```
ALTER TABLE table_name ALTER COLUMN column_name SET COMPRESSION lz4;
```

The cluster-wide default is set by the `default_toast_compression` GUC (v14–18 default pglz; v19+ default lz4 when built with LZ4).

Example 1 — MAIN is protected

A document's row where PostgreSQL **sacrifices the EXTENDED columns to keep the MAIN one on the page**. Target = 2 KB (2032 B). Step through → to watch it shrink.



row: 1540 B — fits ✓

■ **doc_id** int · PLAIN · 4 B ■ **content** text · EXTENDED · → 18 B ptr ■ **author_notes** text · EXTENDED · → 18 B ptr ■ **abstract** text · MAIN · 1500 B

TOAST TABLE (OUT-OF-LINE)

content → chunks

author_notes → chunks

Round 1 — compress EXTENDED: content 3000→1200 B, author_notes 1800→900 B. Neither is alone over 2 KB, so nothing moves yet. MAIN untouched.

Round 2 — move the biggest inline value out: content → 18 B pointer, its data goes to the TOAST table. Still over.

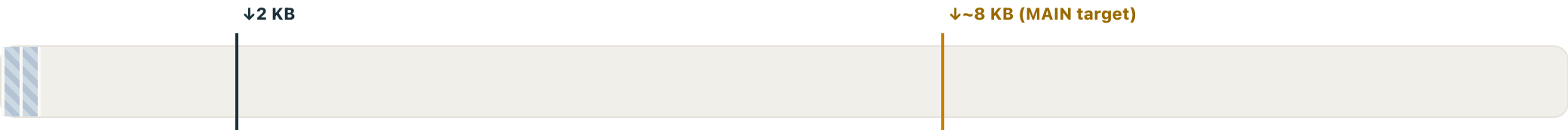
Round 2 continues: author_notes → 18 B pointer. Row now **1540 B** — fits, TOAST stops.

MAIN survived on the page. abstract was never compressed and never moved — evicting the two EXTENDED columns was enough, so **Rounds 3 & 4 never ran**.

If it still hadn't fit: Round 3 would compress abstract (MAIN); only then Round 4 raises the target to **~8160 B** (nearly a full page) and moves it out-of-line — the true last resort. That target bump is *how* MAIN clings to the page.

Example 2 — when MAIN gets evicted too

Same table, but now the data (MAIN) value is huge and compresses poorly. Even after everything else is gone, the row won't fit — so **all four rounds fire**, and MAIN is compressed *and* moved out as a last resort.



row: 40 B — fits ✓

■ **doc_id** int · PLAIN · 4 B ■ **tags** text · EXTENDED · → 18 B ptr ■ **data** text · MAIN · → 18 B ptr

TOAST TABLE (OUT-OF-LINE)

tags → chunks

data → chunks

Round 1 — compress EXTENDED: tags 1500→700 B. data (MAIN) is not touched yet. Still far over.

Round 2 — externalize EXTENDED: tags → 18 B pointer. Nothing EXTENDED/EXTERNAL left inline, yet the row is still ~12 KB.

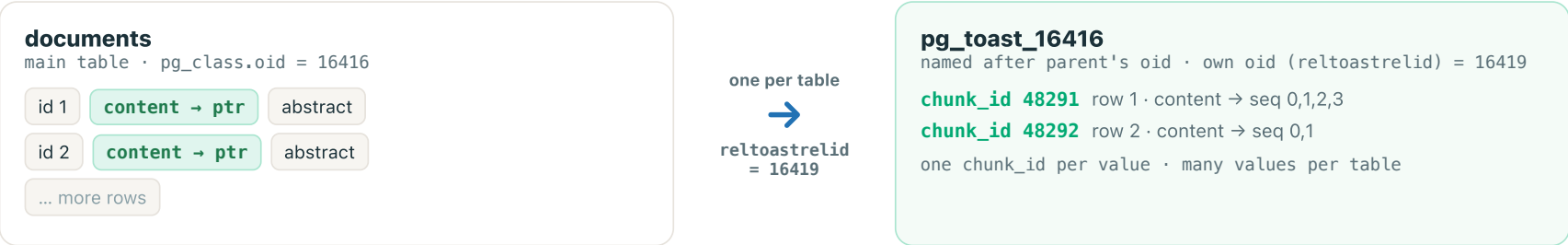
Round 3 — compress MAIN: now data is compressed 12000→9000 B (poorly compressible). Still over — even past the **~8 KB MAIN target**.

Round 4 — last resort: the target is raised to ~8 KB and data (MAIN) is moved out-of-line → 18 B pointer. Row now **40 B** — fits.

MAIN was evicted — the true last resort. This only happens when the row can't fit even after every EXTENDED/EXTERNAL column is out *and* MAIN is compressed — i.e. the compressed MAIN value alone is bigger than the ~8 KB (nearly full-page) target. In Example 1 that never happened, so MAIN stayed put.

Where do the chunks live? One TOAST table per table

Any table with a TOASTable column gets **exactly one** companion table in the pg_toast schema, created once with the table. The <oid> in pg_toast_<oid> is the **owning table's OID** — it is per **table**, never per row.



Chunking: one value becomes many rows

An out-of-line value is sliced into pieces of at most `TOAST_MAX_CHUNK_SIZE` — the docs say **~2 KB (exactly 1996 bytes)**. Each piece becomes one row in the companion table `pg_toast_<oid>`.

one out-of-line value = 7000 B → cut every ~2 KB (1996 B) → $\text{ceil}(7000 / 1996) = 4$ chunks

seq 0 1996 B	seq 1 1996 B	seq 2 1996 B	seq 3 1012 B
-----------------	-----------------	-----------------	-----------------

↓ each slice is stored as one row (same `chunk_id`, rising `chunk_seq`)

`PG_TOAST_16416`

CHUNK_ID	CHUNK_SEQ	CHUNK_DATA
48291	0	1996 B
48291	1	1996 B
48291	2	1996 B
48291	3	1012 B

Why ~2000 bytes? `TOAST_MAX_CHUNK_SIZE` is sized so that **four** chunk rows fit on one 8 kB page (`EXTERN_TUPLES_PER_PAGE = 4`) — hence "about 2000 B". After subtracting per-tuple overhead the exact value is **1996 B**. Changing it requires an `initdb`.

Fast retrieval: every TOAST table has a **unique** btree index `pg_toast_<oid>_index` on (`chunk_id`, `chunk_seq`). Reads fetch a value's chunks by `chunk_id` in order; slice reads use both columns to jump to a specific chunk. The uniqueness also guards against duplicate `chunk_id` OIDs.

Not a parameter. `TOAST_MAX_CHUNK_SIZE` is a compile-time C macro (derived from `EXTERN_TUPLES_PER_PAGE = 4`), not a setting — there is no `GUC`, `postgresql.conf` entry, or `ALTER TABLE` option for it. It is fixed when PostgreSQL is built, and changing it requires editing the source, recompiling, and a fresh `initdb`.

The key idea: the whole value shares **one** `chunk_id` (green column, all 48291). A bigger value just means a higher `chunk_seq` — more rows with that same id. Size grows `chunk_seq`, never `chunk_id`.

So `OID/chunk_id` usage scales with the number of **distinct toasted values**, not with how big any one value is. One 1 GB value = 1 `chunk_id` (~538k chunk rows, all sharing it); a million small values = a million `chunk_ids`.

What stays behind in the row

The big value is replaced in the main row by a tiny **TOAST pointer** (~18 bytes) — a `varatt_external` struct that says "this value lives over there."

ID 42	DOC (POINTER) → value 48291 in pg_toast_16416
----------	--

va_rawsize original size + header	va_extinfo external size + compression	va_valueid → chunk_id (48291)	va_toastrelid which TOAST table (relid 16419)
---	--	---	---

Instead of megabytes inline, the row carries a fixed ~18-byte reference. The main table stays compact and fast to scan.

Reading it back (detoasting)

When a query actually needs the column, PostgreSQL follows the pointer and reassembles the value.

1. Read pointer

Get chunk_id + toast table from the row.



2. Fetch chunks

Look up all rows with that chunk_id, ordered by chunk_seq.



3. Reassemble

Concatenate the pieces, then decompress.

Detoasting is lazy. If a query never touches the large column, its chunks are never read at all — so wide values do not slow down queries that ignore them.

OID ALLOCATION — THE COMMON CASE

One counter hands out IDs to *every* TOAST table

Each out-of-line value needs a `chunk_id`. In the normal case that is simply **the next number from one cluster-wide counter** — `GetNewObjectId()`. Every TOAST table (and every catalog object) draws from the **same** sequence, so a single table's ids are usually **not contiguous**.

`GetNewObjectId()`
one cluster-wide OID counter

next OID →

48294

handing **48293** → **images**

pg_toast_16416 orders.payload

chunk_id 48290

chunk_id 48292

pg_toast_24880 logs.body

chunk_id 48291

pg_toast_31002 images.data

chunk_id 48293

- 1) **orders** inserts a wide payload → grabs **48290**; counter moves to 48291.
- 2) **logs** inserts → grabs the very next id **48291**; counter moves to 48292.
- 3) **orders** inserts again → grabs **48292** — note its ids (48290, 48292) skip over logs' 48291.
- 4) **images** inserts → grabs **48293**. Four values, four consecutive ids, spread across three tables.

No density check in the common case. The counter just returns the next number; a value's `chunk_id` does **not** depend on how full its TOAST table is. Only when a proposed id turns out to be *already taken* does `GetNewIdWithIndex()` loop and retry — that is the next page.

OID ALLOCATION

How a chunk_id is assigned

A chunk_id is just an **OID**, chosen **once per value** (not per chunk) by `GetNewOidWithIndex()`. It asks a cluster-wide counter for a candidate, then retries until it finds one that is free in this TOAST table's index.

```
GETNEWOIDWITHINDEX( PG_TOAST_16416, INDEX, COL 1 )
```

candidate: 48293 → **free → chunk_id**

index has: **48291** **48292** **48293**

retries: **3**

- 1) Counter proposes **48291** → already in the index (SnapshotAny counts dead rows too) → **collision, retry**.
- 2) Next candidate **48292** → also present → **collision, retry**.
- 3) Next candidate **48293** → not found → **assigned** as this value's chunk_id; the chunks are then written with it.

Once per value. The id is chosen before the chunking loop; every chunk reuses it with a rising chunk_seq. A value's size never consumes more ids.

Unique per TOAST table. The check scans this table's unique index `pg_toast_<oid>_index`. Two different TOAST tables may legally hold the same chunk_id.

Why SnapshotAny? The collision test counts **dead-but-not-yet-vacuumed** rows as "in use", so a just-deleted id isn't reused before VACUUM clears it.

The counter is cluster-wide. `GetNewObjectId()` is one 32-bit counter shared by the whole cluster; it wraps at ~4.29 billion back to `FirstNormalObjectId` (16384). After a wrap, candidates may already exist — that is when the retries above begin to pile up.

Normally it is one try. With plenty of free ids the first candidate is unused, so there is no loop. The retry story matters only once the OID space around the counter gets dense — the start of the **TOAST OID exhaustion** topic (the "still searching for an unused OID" log), coming next.

THE SERVER LOG

When the search drags on, Postgres tells you

If the retry loop keeps colliding, `GetNewOidWithIndex()` starts writing to the server log — but with **exponential back-off** so it never floods it. The first message appears after **1,000,000** retries, then the interval doubles up to **128,000,000**, after which it repeats every 128M until an id is found.

1M x2 2M x2 4M x2 8M x2 16M x2 32M x2 64M x2 128M then +128M each ...

```
DETAIL:  OID candidates have been checked 1,000,000 times, but no
unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_16416"
DETAIL:  OID candidates have been checked 2,000,000 times, but no
unused OID has been found yet.
... then 4M, 8M, 16M, 32M, 64M, 128M (interval keeps doubling), after
which it repeats every +128M ...
LOG:  new OID has been assigned in relation "pg_toast_16416" after
5,242,880 retries
```

```
// src/backend/catalog/catalog.c
#define GETNEWOID_LOG_THRESHOLD 1000000
#define GETNEWOID_LOG_MAX_INTERVAL 128000000
```

Why back off? The code doubles the interval *"not to fill up the server log with similar messages"* — one line at 1M, 2M, 4M... instead of a million identical lines.

It also logs success. When an id is finally found, if it took more than 1M retries Postgres emits `new OID has been assigned ...` after N retries, so the incident has a clear end.

Seeing `still searching for an unused OID in relation "pg_toast_..."` means that TOAST table's OID space is so crowded the counter is lapping it — the symptom of **TOAST OID exhaustion**. Try it live on the next page.

INTERACTIVE

Try it: the GetNewOidWithIndex() loop

Drag **TOAST table fill** to change how crowded the OID space is, then **Step** through the loop or **Run** it. Watch the code lines light up, the counter wrap at 2^{32} , retries climb, and the real log fire.

```
GetNewOidWithIndex(rel, indexId, col) {
    retries = 0; before_log = 1,000,000;
    do {
        newOid = GetNewObjectId();           // cluster counter, wraps
        collides = scan(index, newOid, SnapshotAny); // dead rows count
        if (retries >= before_log) { LOG("still searching..."); before_log =
            retries + 1;
        } while (collides);
        return newOid;                       // → chunk_id
    }
}
```

TOAST table fill: 0.0% 

Step

Run

Reset

OID COUNTER	CANDIDATE	VERDICT	RETRIES
4,294,847,296	—	idle	0

server log — move the slider up, then Step or Run...

At low fill the first candidate is free (1 try). Push fill toward 100% — many live/dead values crowding this one TOAST table — and the loop retries millions of times. That is **TOAST OID exhaustion**, and why you would see still searching for an unused OID in relation "pg_toast_..." in the server log.

MAJOR ISSUES

Issue #1 — TOAST OID exhaustion

CRITICAL when one TOAST table runs out of unused `chunk_ids`

WHAT

A single TOAST table accumulates so many distinct out-of-line values that its `chunk_id` space (a 32-bit OID) becomes densely occupied. Because `chunk_id` must be **unique within that TOAST table**, `GetNewIdWithIndex()` has to retry to find a free id. As occupancy climbs toward the **~4.29 billion** ceiling, every new wide value needs more and more retries: `INSERT/UPDATE` of TOASTed values slows sharply and Postgres logs still searching for an unused OID in relation "`pg_toast_...`". Exhaustion is counted in **distinct TOASTed values** (one OID per value) — not per chunk or per row.

WHY

- **chunk_id is a 32-bit OID** — a hard ceiling of ~4.29B distinct values per TOAST table, handed out by the **cluster-wide** `GetNewObjectId()` counter.
- **Uniqueness is enforced per TOAST table** (unique index on `chunk_id`, `chunk_seq`). The collision check uses `SnapshotAny`, so **dead-but-not-yet-vacuumed** values still count as "in use".
- **The counter wraps** at 2^{32} back to `FirstNormalObjectId` (16384); after a wrap it revisits ranges already crowded by a big table's ids → collisions.
- **Amplifiers:** very large tables of wide values; high churn plus vacuum lag / long-running transactions that keep dead ids "in use"; high write volume burning ids faster.
- **Consequence:** each retry is an index probe, so cost grows with density — the retry loop in `GetNewIdWithIndex()` is the root cause.

MOST AT RISK

The classic profile is a **single, unpartitioned append-only table** that pours values into **one** giant TOAST table:

- **Unpartitioned** — all rows share **one** TOAST table, so every `chunk_id` piles into a **single** OID space instead of being spread across many per-partition TOAST tables.
- **Wide data types that TOAST routinely** — large `TEXT`, `BYTEA`, `JSON/JSONB`, so nearly every row adds a **distinct** `chunk_id`.
- **Insert-heavy, delete-light** — append-mostly with few deletes/updates, so values **accumulate and never free** their ids — monotonic growth straight toward the ~4.29B ceiling.

Mitigation direction: partition the table (one TOAST table per partition → independent OID spaces), or age out old data so values are freed — explored in the fixes.

ISSUE #1 — REPRODUCTION

Reproduce it: exhaust a TOAST table

Long-running and disk-heavy — it deliberately drives one TOAST table's chunk_id space to ~100%.

1 • Create a TOAST-heavy, append-only table

```
DROP TABLE IF EXISTS big_table;
CREATE TABLE big_table (
  id    BIGSERIAL PRIMARY KEY,
  text  TEXT
);
-- EXTERNAL = out-of-line, no compression: every row lands in TOAST
ALTER TABLE big_table ALTER COLUMN text SET STORAGE EXTERNAL;
```

2 • Load until the counter laps the table

```
CREATE OR REPLACE PROCEDURE insert_big_batches_proc(max_rows BIGINT)
LANGUAGE plpgsql AS $$
DECLARE batch_size INT := 50000; inserted BIGINT := 0;
BEGIN
  WHILE inserted < max_rows LOOP
    INSERT INTO big_table (text)
    SELECT repeat(md5(random()::text), 100)
    FROM generate_series(1, batch_size);
    inserted := inserted + batch_size;
    COMMIT;
    RAISE NOTICE 'Inserted % / % rows', inserted, max_rows;
  END LOOP;
END; $$;

-- Long-running: billions of wide values consume chunk_id OIDs
CALL insert_big_batches_proc(5000000000);
```

Confirm it (second session): identify & size

3 · Find the TOAST table

```
SELECT r.relname,  
       t.relname AS TOAST,  
       i.relname AS toast_index  
FROM pg_class r, pg_class i, pg_index d, pg_class t  
WHERE r.relname = 'big_table'  
      AND d.indrelid = r.reltoastrelid  
      AND i.oid = d.indexrelid  
      AND t.oid = r.reltoastrelid;
```

relname	toast	toast_index
big_table	pg_toast_69643053	pg_toast_69643053_index

(1 row)

4 · Check its size

```
SELECT t.relname AS table_name, r.relname AS toast_name, r.oid AS toast_oid,  
       pg_relation_size(t.reltoastrelid) AS toast_size_bytes,  
       pg_size_pretty(pg_relation_size(t.reltoastrelid)) AS toast_size  
FROM pg_class r  
INNER JOIN pg_class t ON r.oid = t.reltoastrelid  
WHERE t.relname = 'big_table';
```

table_name	toast_name	toast_oid	toast_size_bytes	toast_size
big_table	pg_toast_69643053	69643058	17590450323456	16 TB

(1 row)

Confirm it: chunk_id density

A full count (DISTINCT chunk_id) over the TOAST table (slow — a full scan of billions of rows) against the usable OID space:

```
-- reserved = the 16384 low OIDs Postgres keeps for system use; a chunk_id is
-- always >= 16384, so the real ceiling is (2^32 - reserved), not a full 2^32.
WITH k AS (
  SELECT count(DISTINCT chunk_id)::bigint AS used_oids,
         count(*)::bigint                AS chunk_rows
  FROM   pg_toast.pg_toast_69643053
),
c AS (SELECT 4294967296::bigint AS oid_space,  -- 2^32
          16384::bigint      AS reserved)  -- reserved low OIDs
SELECT
  used_oids,
  chunk_rows,
  round(chunk_rows::numeric / NULLIF(used_oids,0), 2) AS avg_chunks_per_value,
  (oid_space - reserved - used_oids) AS remaining_oids,
  round(100.0 * used_oids / (oid_space - reserved), 4) AS pct_used,
  round(100.0 * (oid_space - reserved - used_oids)
        / (oid_space - reserved), 4) AS pct_remaining,
  CASE
    WHEN used_oids::numeric / (oid_space - reserved) >= 0.98 THEN 'CRITICAL'
    WHEN used_oids::numeric / (oid_space - reserved) >= 0.90 THEN 'WARNING'
    WHEN used_oids::numeric / (oid_space - reserved) >= 0.75 THEN 'WATCH'
    ELSE 'OK'
  AS status
END
FROM k, c;
```

used_oids	chunk_rows	avg_chunks_per_value	remaining_oids	pct_used	pct_remaining	status
4294200000	8588400000	2.00	750912	99.9825	0.0175	CRITICAL
(1 row)						

Only **750,912** free ids remain of ~4.29B (**99.98%** used). New inserts now spin in the retry loop and log still searching for an unused OID in relation "pg_toast_69643053".

Diagnosing & identifying it

Start with what you notice — the symptoms that point at a single exhausted TOAST table.

1 • Symptoms you notice

- INSERT/UPDATE of wide values (large TEXT, BYTEA, JSON) suddenly slows to a crawl.
- **It's localized — other writes are fine.** Inserts into other tables (and narrow rows that don't get TOASTed) keep working normally.
- High CPU / I/O but **little write throughput** — lots of work, few rows actually landing.
- **Retry storm from app timeouts** — if the app has a statement/connection **timeout + automatic retry**, a write stuck in the OID-search loop blows past the timeout, is cancelled, and is **retried** — each attempt restarts the expensive search. Concurrent retries pile up, multiplying OidGen contention and I/O, and the cancelled attempts can leave **dead chunk rows** that (via SnapshotAny) crowd the space even more. The slowdown feeds on itself.



Wait events: the stalled backend is **read-bound**

The retry loop calls `GetNewOidWithIndex()` over and over, each time doing a tiny lookup on the TOAST index to test a candidate `chunk_id`. Millions of lookups → the session spends almost all its time **reading index pages**. So the waits you see are really “I’m waiting to read a page.”

IO:DataFileRead

The backend is **actually fetching a page from storage** (a cache miss).
Analogy: you walking to the warehouse to get the box.

IPC:BufferIO

The backend wants a page **another backend is already fetching**, so it just waits.
Analogy: standing at the shelf while a coworker brings the box.

It depends on how many sessions are stuck:

One session

Purely **read-bound** — just `IO:DataFileRead`. **No** `BufferIO` (no one to wait on) and little `OidGen`. This is the common captured case.

Many sessions

Reads are shared → waiters pile on `IPC:BufferIO`, and the single global `LWLock:OidGen` turns into a cluster-wide bottleneck (noisy neighbor). `LWLock:BufferMapping` also rises — every one of those millions of page lookups must find or place the buffer in the shared buffer-mapping hash table, whose partition locks get hammered under the load.

`LWLock:OidGen` guards the single, cluster-wide OID counter — `GetNewObjectId()` takes it on every allocation, so one table’s retry storm serializes OID hand-out for the whole cluster.

Naming change: it was `OidGenLock` in PG ≤ 12; from **PG 13** the predefined LWLock wait events dropped the Lock suffix, so it’s now reported as `OidGen`.

Naming change: PG ≤ 13 showed the “wait for another backend’s read” as `LWLock:buffer_io`; since **PG 14** it’s a condition variable reported as `IPC:BufferIO`. Same meaning — new type & name.

ISSUE #1 — DETECTION (CONT.)

In Lakebase

Lakebase separates compute from storage — pages come from a remote **pageserver** (**PS**) and are cached locally in the **Local File Cache (LFC)**. So the same “waiting to read a page” shows up under **Neon** names.

Lakebase (Neon) wait events

- **One session:** **Neon/PS_ReadIO** dominates (fetch an index page from the PS) + **Neon/FileCache_Write** (write it into the LFC). No **IPC:BufferIO**.
- **Many sessions:** **Neon/PS_ReadIO** still dominates, plus **IPC:BufferIO** waiters and the same global **LWLock:OidGen** bottleneck.

Equivalence with vanilla: **I0:DataFileRead** ↔ **Neon/PS_ReadIO** (physical page read) · shared-buffer fill ↔ **Neon/FileCache_Write** · **IPC:BufferIO** and **LWLock:OidGen** behave the same on both.

Noisy neighbor: the global **LWLock:OidGen** is shared — one exhausted TOAST table can slow OID allocation for the **whole cluster**, even for writes to healthy tables.

ISSUE #1 — DETECTION (CONT.)

In Lakebase: sample one backend's waits

`neon_backend_wait_report(pid, seconds)` reports a **single backend's** wait events over a window — **waits** (how often it was seen waiting), **wait_time** (total ms), and **ms_per_wait** (mean). Point it at the stuck write's PID.

Sampling the stuck INSERT (pid 6741) for 60 s — almost entirely read-bound on the PS:

```
databricks_postgres=> select * from neon_backend_wait_report(6741, 60);
type_name |      event_name      | waits | wait_time | ms_per_wait
-----+-----+-----+-----+-----
Extension | Neon/PS_ReadIO       | 58689 | 58511.73 | 0.997
Extension | Neon/FileCache_Write | 58689 | 283.74  | 0.005
LWLock    | neon_lwlsn           | 1163  | 7.07    | 0.006
LWLock    | BufferMapping         | 50    | 1.43    | 0.029
LWLock    | lfc_lock             | 126   | 0.72    | 0.006
(5 rows)
```

Neon/PS_ReadIO = 58,511 ms of 60,000 ms (~97.5%) spent fetching index pages from the PS — each one a candidate-OID probe against the near-full TOAST index.

ISSUE #1 — DETECTION (CONT.)

The server log names the table

2 • The definitive signal — the server log

- Postgres logs still searching for an unused OID and **names the exact TOAST table**; the DETAIL shows how many candidates were checked — **doubling each time** (1M, 2M, 4M, 8M...).
- On success: new OID has been assigned ... after N retries marks the end of the incident.

```
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 1000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 2000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 4000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 8000000 times, but no unused OID has been found yet.
```

Blind spot: this log only starts after 1,000,000 retries (core's GETNEWOID_LOG_THRESHOLD). The retry storm from the previous page can be well underway below that floor — a shorter burst, or a statement **cancelled / timed out before 1M**, is just as slow but logs **nothing**. We come back to this blind spot later.

Confirm & quantify: chunk_id density

A full count (DISTINCT chunk_id) over the TOAST table (slow — a full scan of billions of rows) against the usable OID space:

```
-- reserved = the 16384 low OIDs Postgres keeps for system use; a chunk_id is
-- always >= 16384, so the real ceiling is (2^32 - reserved), not a full 2^32.
WITH k AS (
  SELECT count(DISTINCT chunk_id)::bigint AS used_oids,
         count(*)::bigint                AS chunk_rows
  FROM   pg_toast.pg_toast_69643053
),
c AS (SELECT 4294967296::bigint AS oid_space, -- 2^32
          16384::bigint      AS reserved)  -- reserved low OIDs
SELECT
  used_oids,
  chunk_rows,
  round(chunk_rows::numeric / NULLIF(used_oids,0), 2) AS avg_chunks_per_value,
  (oid_space - reserved - used_oids)                  AS remaining_oids,
  round(100.0 * used_oids / (oid_space - reserved), 4) AS pct_used,
  round(100.0 * (oid_space - reserved - used_oids)
        / (oid_space - reserved), 4)                  AS pct_remaining,
  CASE
    WHEN used_oids::numeric / (oid_space - reserved) >= 0.98 THEN 'CRITICAL'
    WHEN used_oids::numeric / (oid_space - reserved) >= 0.90 THEN 'WARNING'
    WHEN used_oids::numeric / (oid_space - reserved) >= 0.75 THEN 'WATCH'
    ELSE 'OK'
  END
  AS status
FROM k, c;
```

Heads-up: this scans the **entire** TOAST table (billions of chunk rows), so it can run for **minutes to hours** — depending on table size and the **DB host** (CPU, memory, storage / I/O). Run it off-peak.

used_oids	chunk_rows	avg_chunks_per_value	remaining_oids	pct_used	pct_remaining	status
4294200000	8588400000	2.00	750912	99.9825	0.0175	CRITICAL
(1 row)						

Only **750,912** free ids remain of ~4.29B (**99.98%** used). New inserts now spin in the retry loop and log still searching for an unused OID in relation "pg_toast_69643053".

ISSUE #1 — DETECTION (CONT.)

Catch it in the act: the backend is stuck in the OID retry loop

Definitive proof: find the PID of the stalled INSERT/UPDATE, then grab a backtrace. If it is stuck probing for a free chunk_id, the stack sits inside `GetNewOidWithIndex()`.

1 · Find the PID — the session writing wide values (from `pg_stat_activity`):

```
SELECT pid, state, wait_event_type, wait_event, now() - query_start AS running, query
FROM   pg_stat_activity
WHERE  state = 'active' AND query ~* '(insert|update)'
ORDER BY query_start;
-- pid | state | wait_event_type | wait_event | running | query
-- ----+-----+-----+-----+-----+-----
-- 6741 | active | IO              | DataFileRead | 00:02:11 | INSERT INTO big_table ...
```

2 · Backtrace that PID — `gdb` (or `pstack` for a quick, read-only snapshot):

```
gdb -ex "set height 0" -ex "thread apply all bt" --batch -p 6741
# or, lighter weight:
pstack 6741
```

3 · Read the stack — the tell-tale frames (retrying the OID collision check):

```
... (top frames: buffered read / btree probe of the TOAST index) ...
#15 0x0000005580228c7c in ReadBuffer_common (...)
#16 0x00000055801a9d20 in _bt_search (...)
#17 0x00000055801aa884 in _bt_first (...)
#18 0x00000055801ab9f0 in systable_getnext (...)
OID CHECK: Checking whether a generated object ID is already in use.
#19 0x000000558011f8ec in GetNewOidWithIndex (oidcolumn=<optimized out>, indexId=<optimized out>, relation=<optimized out>) at catalog.c:488
OID CHECK: Checking whether a generated object ID is already in use.
#20 GetNewOidWithIndex (relation=relation@entry=0x7f27f09b28, indexId=69643059, oidcolumn=oidcolumn@entry=1) at catalog.c:448
#21 0x0000005580142abc in toast_save_datum (...)
#22 0x0000005580140f30 in heap_toast_insert_or_update (...)
#23 0x00000055801248a0 in heap_insert (...)
... (executor: ExecInsert -> ExecModifyTable -> standard_ExecutorRun) ...
```

Two frames in `GetNewOidWithIndex()` (`catalog.c:448` → the `index_getnext` probe at `:488`) mean the backend is **spinning through candidate OIDs and checking the TOAST index for each one**. Combined with the still searching for an unused OID log line, this is conclusive: the TOAST table's `chunk_id` space is (near-)exhausted.

ISSUE #1 — DETECTION (CONT.)

See it live from the client

Those LOG lines go to the **server log** by default. Raise `client_min_messages` to LOG and `psql` prints them **right in your session** — real-time proof, no log access needed.

```
-- LOG-level messages normally go only to the server log; raise the client
-- threshold so your session sees them too:
SET client_min_messages = LOG;
SHOW client_min_messages;    -- => log
```

```
databricks_postgres=> SET client_min_messages = LOG;
SET
databricks_postgres=> SHOW client_min_messages;
 client_min_messages
-----
log
(1 row)

databricks_postgres=> INSERT INTO big_table (text)
                      SELECT repeat(md5(random()::text), 100);
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 1000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 2000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 4000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_69643053"
DETAIL:  OID candidates have been checked 8000000 times, but no unused OID has been found yet.
^Ccancel request sent
ERROR:  canceling statement due to user request
Time: 5052208.556 ms (01:24:12.209)
```

A **single** INSERT ran for **1 h 24 m** without landing a row — the retry counter **doubles** (1M → 2M → 4M → 8M...) and the statement had to be cancelled by hand. The named relation (`pg_toast_69643053`) points straight at the exhausted TOAST table.

Fast fixes to buy time

Actions you can deploy right now with minimal disruption. Each relieves OID pressure temporarily — none removes the underlying ceiling.

1 • Clean up existing data

Delete obsolete or unneeded rows, then `VACUUM` so their `chunk_ids` stop counting as “in use” and become reusable — easing OID pressure and restoring insert speed.

PROS

- + Fast, no schema change — instant pressure relief.
- + Restores insert performance while free ids exist.

CONS

- **No guarantee** — ids still come from the sequential `GetNewObjectId()` counter, which keeps probing candidates that may already be taken. If the table stays dense you can still hit **OID collisions** and the retry loop that tanks performance.
- **Diminishing returns** — as the table keeps growing there is less to delete.
- On a huge table the `VACUUM` needed to actually reclaim ids is heavy and can lag.

2 • Archive data to a new table

Move outdated or cold rows into a separate archive table, then purge them from the live table — shrinking the live TOAST table's distinct-value count so Postgres manages ids more easily.

PROS

- + Reduces the live table's size and OID pressure.
- + Keeps historical data accessible in the archive.

CONS

- Still depends on `VACUUM` of the live table to actually free ids.
- Temporary — won't resolve exhaustion if the table keeps growing.

Swap to a fresh table & near-term tweaks

3 · Write to a new table (rename + view)

Rename the exhausted table to `tasks_old`, create a fresh empty `tasks_new` (a brand-new, empty `chunk_id` space), send all new writes there, and expose both as one logical table with a `UNION ALL` view:

```
CREATE VIEW tasks AS
  SELECT * FROM tasks_old
  UNION ALL
  SELECT * FROM tasks_new;
```

PROS

- + New inserts bypass the exhausted TOAST table immediately.
- + Transparent to readers — the view keeps queries unchanged.

CONS

- **UPDATEs on old rows re-TOAST in the old table** → can revive exhaustion; needs app-side copy-on-update (fetch old → insert new), adding latency.
- As rows migrate/update into `tasks_new`, it can eventually exhaust too — not scalable.

Also · Medium-term tweaks to buy headroom

- **Rewrite / repack** (`VACUUM FULL`, `CLUSTER`, or `pg_repack`) into a fresh TOAST table — only helps if a lot was **dead**; a table full of **live** distinct values stays full.
- **Create fewer out-of-line values** — switch wide columns to `lz4` and/or tune `toast_tuple_target` so more values stay **inline** and never get a `chunk_id`.
- **Aggressive autovacuum on the TOAST table** (per-table `autovacuum_*` reloptions) so dead ids never pile up.

ISSUE #1 — LONG-TERM & SCALABLE

The real fix: table partitioning

Split the large table into partitions (for example by **date** or **region**). **Each partition gets its own TOAST table and its own independent chunk_id OID space**, spreading OID usage so no single partition nears the ~4.29B ceiling.

Partition by a natural key (date / region)

BENEFITS

- + Each partition = an **independent** ~4.29B chunk_id space → capacity scales with partition count.
- + **DROP old partitions** to free entire OID spaces instantly — ideal for time-series / append-only data.
- + Faster queries and inserts via **partition pruning** (less data scanned).

CONSIDERATIONS

- Needs careful design — the right partition key and **even data distribution**.
- Query patterns and indexing must suit partitioning; a poor scheme can hurt performance.
- Migrating an existing large table into partitions takes planning and effort.

Bottom line: cleanup, archiving, and the rename-and-view trick buy time; **partitioning is the only approach that removes the ceiling** rather than deferring it.

MAJOR ISSUES

Issue #2 — TOAST OID collision (after OID wraparound)

HIGH the same retry storm as Issue #1 — but it strikes **before** a table is exhausted

WHAT

The cluster-wide OID counter wraps at 2^{32} and resets to `FirstNormalObjectId` (16384), then re-traverses ranges a TOAST table **already filled** with a dense block of `chunk_ids` (e.g. from a big historical bulk load). `GetNewObjectId()` keeps proposing candidates that are **already taken**, so `GetNewOidWithIndex()` spins in the retry loop and logs `still searching for an unused OID` — even though the table still has **plenty of free ids** overall. It's the exhaustion symptom without the exhaustion.

WHY

- **One shared counter.** Every OID comes from a single cluster-wide 32-bit counter (`varsup.c`). Its own comment: *"counter wraparound will occur eventually ... it is unwise to assume they are unique."*
- **Position, not fullness.** What matters is **where the counter is right now**, not how full the table is. When it enters a range a table has already packed with `chunk_ids`, each proposed OID lands on a taken one — so it collides and retries until the counter advances **past** that occupied block.
- **The whole cluster feeds one counter.** All OID consumers (every TOASTed value, temp tables, large objects, DDL) advance the same counter, so a busy cluster reaches the next wrap faster.
- **Dead ids still collide.** The check uses `SnapshotAny`, so dead-but-unvacuumed `chunk_ids` block candidates too.
- **Transient but recurring.** The storm clears once the counter walks past the occupied block — then can return on the next lap or at the next dense block.

MOST AT RISK

A **long-lived, OID-hungry cluster** that has already wrapped the counter at least once, plus a table holding a **dense, contiguous block** of `chunk_ids`:

- **Already wrapped (or close).** Clusters old/busy enough to have burned ~4.29B OIDs at least once.
- **Big one-shot bulk loads.** A large historical import lays down a **tight, contiguous** `chunk_id` block that the counter will later revisit all at once — the perfect collision target.
- **Unpartitioned single TOAST table.** Concentrates all `chunk_ids` into one OID space, making dense blocks (and thus collisions) more likely than spreading across per-partition TOAST tables.

Issue #2 — TOAST OID Collision After Wraparound

The exhaustion symptom — without the exhaustion

1 WRAPAROUND

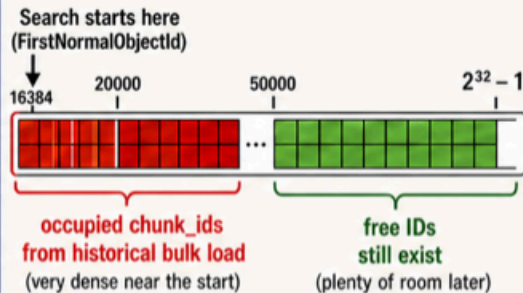
OID counter hits the limit...



cluster-wide counter wraps

2 COLLISION

TOAST table chunk_id space (uint32)

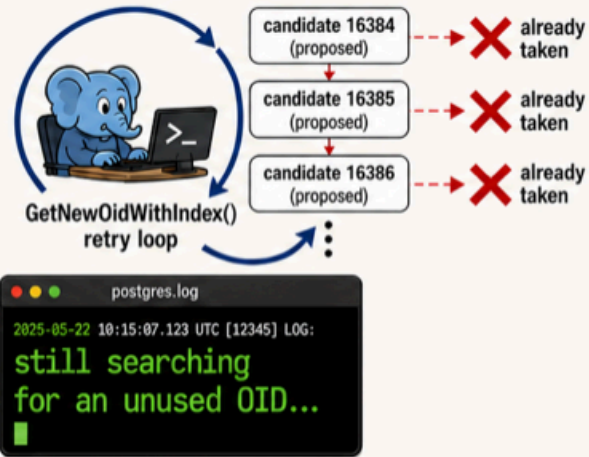


Free overall $\neq$ free at the candidate location

The table is not exhausted.

3 RETRY LOOP

Worker keeps proposing occupied IDs near the start.



OID collision after wraparound:
it **looks like** exhaustion, **but** the table still has free IDs.

After a wrap the counter restarts at FirstNormalObjectId (16384) and walks straight into a dense block of existing chunk_ids — **free overall $\neq$ free at the candidate**. It looks like exhaustion, but the table still has free ids.

ISSUE #2 — REPRODUCTION

Reproduce it: force a real wraparound (setup)

Start clean. Run on a **fresh cluster/DB** so `next_oid` is low and the wrap timing is predictable — or reset the counter first: `pg_ctl stop → pg_resetwal -o <oid> "$PGDATA" → pg_ctl start`. Test clusters only.

Load `big_table` **first** so its `chunk_ids` fill the **low** OID range, then push the counter up to just under 2^{32} with `big_table_2`.

```
-- 1) Dense LOW block – load big_table FIRST
DROP TABLE IF EXISTS big_table;
CREATE TABLE big_table (id BIGSERIAL PRIMARY KEY, text TEXT);
ALTER TABLE big_table ALTER COLUMN text SET STORAGE EXTERNAL;  -- no compression → straight to TOAST

CREATE OR REPLACE PROCEDURE insert_big_batches_proc(max_rows BIGINT)
LANGUAGE plpgsql AS $$
DECLARE batch_size INT := 50000; inserted BIGINT := 0;
BEGIN
    WHILE inserted < max_rows LOOP
        INSERT INTO big_table (text)
        SELECT repeat(md5(random()::text), 100) FROM generate_series(1, batch_size);
        inserted := inserted + batch_size; COMMIT;
    END LOOP;
END $$;

CALL insert_big_batches_proc(3000000000);  -- ~3B chunk_ids fill the LOW range

-- 2) big_table_2 uses an identical proc (insert_big_batches_proc_2 → big_table_2)
--    Push next_oid to JUST under 2^32 (stop before it wraps):
CALL insert_big_batches_proc_2(1375305832);
```

ISSUE #2 — REPRODUCTION (CONT.)

Watch the counter approach the wrap

With the dense low block in place and `big_table_2` pushing `next_oid` up, check how close the cluster-wide counter is to wrapping at 2^{32} .

3) Watch the counter approach the wrap:

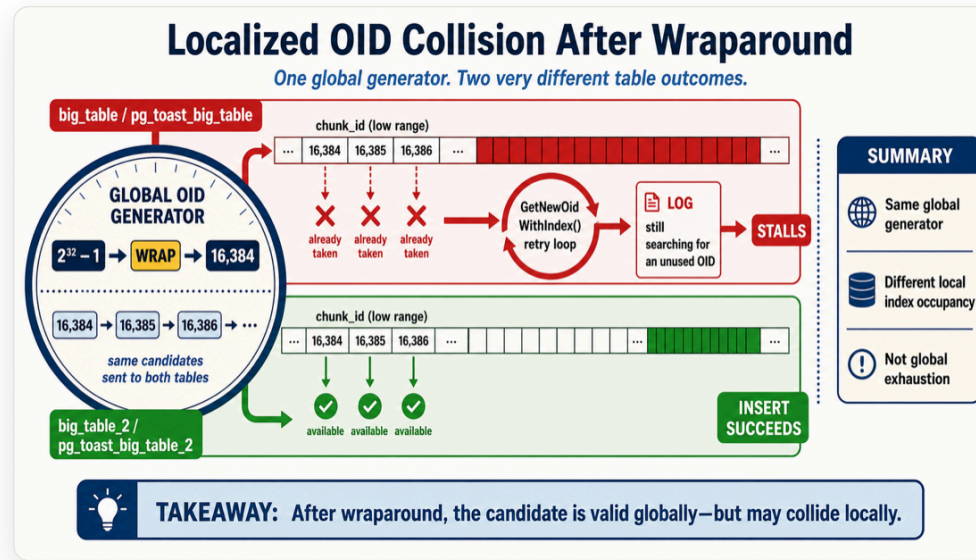
```
SELECT (2::bigint^32) - next_oid::bigint          AS oids_remaining,  
       ROUND(((2::bigint^32) - next_oid::bigint)::numeric  
             / (2::bigint^32)::numeric * 100, 4)  AS percent_remaining  
FROM   pg_control_checkpoint();
```

```
oids_remaining | percent_remaining  
-----+-----  
3506068 | 0.0816 -- ~3.5M OIDs from the top
```

ISSUE #2 — REPRODUCTION (CONT.)

Cross the wrap & prove it's localized

Drive the counter over 2^{32} under concurrency, then show the collision is **per-table**.



```
# one wide insert into the DENSE table
echo "INSERT INTO big_table (text) SELECT repeat(md5(random()::text), 100);" > toast_insert.sql

# 500 clients push next_oid over 2^32 → back to 16384, into big_table's low block
pgbench --client=500 --jobs=2 --progress=60 --transactions=20000 -f toast_insert.sql
```

```
progress: 300.0 s, 315.9 tps ...
progress: 360.0 s, 0.0 tps, lat 0.000 ms ... ← throughput collapses at the wrap
```

ISSUE #2 — REPRODUCTION (CONT.)

Prove the collision is localized

Prove it's a localized collision, not global exhaustion — run both right after the wrap:

```
-- Session A → STALLS: low candidate OIDs collide with big_table's low chunk_ids
INSERT INTO big_table (text) SELECT repeat(md5(random()::text), 100);

-- Session B → FAST: big_table_2's chunk_ids live high, so low candidates are free for it
INSERT INTO big_table_2 (text) SELECT repeat(md5(random()::text), 100);
```

Same cluster, same counter — one table stalls, the other flies. **Which** one stalls depends on where the counter currently sits: right after the wrap it's `big_table`; once it climbs past ~3B it flips to `big_table_2`.

Cost: a real wrap burns $>2^{32}$ OIDs — roughly **~14 TB** and many hours. To demonstrate the same mechanism cheaply, use the `pg_resetwal -o` page next.

```
postgres=> SELECT
  (2::bigint^32) - next_oid::bigint AS oids_remaining,
  ROUND(
    ((2::bigint^32) - next_oid::bigint)::numeric / (2::bigint^32)::numeric * 100,
    4
  ) AS percent_remaining
FROM pg_control_checkpoint();
oids_remaining | percent_remaining
-----+-----
1539988 | 0.0359
(1 row)
```

```
postgres=> SELECT next_oid FROM pg_control_checkpoint();
next_oid
-----
4293427308
(1 row)
```

```
postgres=> SELECT
  (2::bigint^32) - next_oid::bigint AS oids_remaining,
  ROUND(
    ((2::bigint^32) - next_oid::bigint)::numeric / (2::bigint^32)::numeric * 100,
    4
  ) AS percent_remaining
FROM pg_control_checkpoint();
oids_remaining | percent_remaining
-----+-----
425555584 | 99.0824
(1 row)
```

```
postgres=> SELECT
  (2::bigint^32) - next_oid::bigint AS oids_remaining,
  ROUND(
    ((2::bigint^32) - next_oid::bigint)::numeric / (2::bigint^32)::numeric * 100,
    4
  ) AS percent_remaining
FROM pg_control_checkpoint();
oids_remaining | percent_remaining
-----+-----
425555584 | 99.0824
(1 row)
```

```
postgres=>
```

Global OID Counter wraparound

0 TPS

```
]# pgbench -i --fillfactor=90 --scale=2000
```

```
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data (client-side)...
vacuuming...
creating primary keys...
done in 267.97 s (drop tables 0.00 s, create tables 0.03 s, client-side generate 199.96 s, vacuum 1.07 s, primary keys 66.91 s).
```

```
]# pgbench --client=500 --jobs=2 --progress=60 --transactions=20000 -f toast_insert.sql
```

```
pgbench (18.1, server 17.6)
starting vacuum...end.
progress: 60.0 s, 12522.9 tps, lat 35.836 ms stddev 187.070, 0 failed
progress: 120.0 s, 15101.0 tps, lat 33.108 ms stddev 17.472, 0 failed
progress: 180.0 s, 15203.1 tps, lat 32.876 ms stddev 16.408, 0 failed
progress: 240.0 s, 15427.9 tps, lat 32.406 ms stddev 16.992, 0 failed
progress: 300.0 s, 315.9 tps, lat 30.954 ms stddev 10.709, 0 failed
progress: 360.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 420.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 480.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 540.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 600.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 660.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 720.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 780.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 840.0 s, 233.7 tps, lat 19806.837 ms stddev 102806.371, 0 failed
progress: 900.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 960.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 1020.0 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 1080.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
progress: 1140.1 s, 0.0 tps, lat 0.000 ms stddev 0.000, 0 failed
```

Live proof. The counter runs down to **1.5M** OIDs left (**0.0359%**), then **wraps** back to ~99% remaining — and at that instant pgbench TOAST inserts fall from ~15,000 tps to **0 TPS**. The table isn't full (99% shows "remaining"); the counter just re-entered an occupied block.

ISSUE #2 — REPRODUCTION (CHEAP)

Force a collision with pg_resetwal -o

No billion-row load: lay down a **large enough** occupied block, then point the OID counter back **into** it.

0 · Check where the global OID counter is now

```
SELECT next_oid FROM pg_control_checkpoint();
```

1 · Lay down a block (same **big_table** code, $\geq 10\text{M}$ rows)

```
CREATE TABLE big_table (id BIGSERIAL PRIMARY KEY, text TEXT);
ALTER TABLE big_table ALTER COLUMN text SET STORAGE EXTERNAL;

-- load  $\geq 10\text{M}$  wide rows  $\rightarrow \geq 10\text{M}$  contiguous chunk_ids (one OID per value)
-- (batched, using insert_big_batches_proc from the reproduction page)
CALL insert_big_batches_proc(10000000);

-- note the chunk_ids we just placed:
SELECT min(chunk_id), max(chunk_id) FROM pg_toast.pg_toast_<big_table oid>;
```

ISSUE #2 — REPRODUCTION (CHEAP, CONT.)

Point the counter into the block & trigger it

2 & 3 · Stop the cluster, point the next OID into that block

```
pg_ctl -D "$PGDATA" stop                # pg_resetwal needs an OFFLINE data dir
pg_resetwal -o <oid_inside_the_block> "$PGDATA"  # e.g. the min(chunk_id) from step 1
pg_ctl -D "$PGDATA" start
```

4 · Make the log visible, then insert into the same table

```
SET client_min_messages = LOG;
INSERT INTO big_table (text)
SELECT repeat(md5(random()::text), 100) FROM generate_series(1, 100);
```

Candidates now land on the ids you just placed → `GetNewOidWithIndex()` retries until it passes the block. **Why $\geq 10M$:** the still searching for an unused OID log fires only after **1,000,000** retries *within a single allocation* (`GETNEWOID_LOG_THRESHOLD`) — so the occupied block must be at least ~1M contiguous `chunk_ids`. 10M gives margin to clearly see the log storm.

Test clusters only. `pg_resetwal` is a low-level surgical tool — never run it on production, and managed platforms (Lakebase) don't expose it. `-o` only changes **new** allocations; it isolates the collision mechanism without a billion-row load.

ISSUE #2 — MEASURED IMPACT

One identical insert: 5.5 ms → 21 s

Same one-row INSERT on the 10M-row block from the repro — the **only** difference is where the OID counter sits.

Baseline — no collision

5.510 ms

counter just past the block · 1 probe

Collision — after `pg_resetwal -o`

21,274.851 ms

counter re-entered the block · 10,000,002 probes

≈ 3,860× slower for one identical INSERT — the whole occupied block is probed before a free OID is found.

```
-- BASELINE (counter is just past the block)
INSERT INTO big_table (text)
SELECT repeat(md5(random()::text), 70) FROM generate_series(1, 1);
INSERT 0 1
Time: 5.510 ms

-- COLLISION (after pg_resetwal -o 4424618)
SET client_min_messages = LOG;
INSERT INTO big_table (text)
SELECT repeat(md5(random()::text), 70) FROM generate_series(1, 1);
LOG:  still searching for an unused OID in relation "pg_toast_4424609"
DETAIL:  OID candidates have been checked 1000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_4424609"
DETAIL:  OID candidates have been checked 2000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_4424609"
DETAIL:  OID candidates have been checked 4000000 times, but no unused OID has been found yet.
LOG:  still searching for an unused OID in relation "pg_toast_4424609"
DETAIL:  OID candidates have been checked 8000000 times, but no unused OID has been found yet.
LOG:  new OID has been assigned in relation "pg_toast_4424609" after 10000002 retries
INSERT 0 1
Time: 21274.851 ms (00:21.275)
```

WRAP-UP

OID collision vs OID exhaustion

Same retry loop, same log line — but two very different root states.

	OID EXHAUSTION (ISSUE #1)	OID COLLISION (ISSUE #2)
State	Space is genuinely (near-)full — billions of live distinct chunk_ids.	Space has plenty free — the counter just wrapped onto an occupied block .
Density check	pct_used $\approx$ 99% $\rightarrow$ CRITICAL.	pct_used low / moderate , yet inserts stall.
Duration	Permanent until ids are freed.	Transient — clears as the counter walks past; can recur.
Scope	The one exhausted table.	Localized to the table whose dense block the counter is crossing; others insert fine.
Trigger	Monotonic growth to the $\sim 4.29\text{B}$ ceiling.	Global counter wraps at 2^{32} into a dense block.
Fix	Free ids (delete + VACUUM, archive); partition to remove the ceiling.	Self-clears; avoid dense contiguous blocks; partition / spread . Long-term = same as #1.

In common: the same retry loop in `GetNewOidWithIndex()` (with `SnapshotAny`), the same still searching for an unused OID log naming the TOAST table, the same wait profile (`LWLock:OidGen` + read-bound), and the same user-visible symptom — wide INSERT/UPDATE stalls.

Metric: the OID-consumption rate

Every failed probe in `GetNewOidWithIndex()` calls `GetNewObjectId()` — so a retry storm **burns OIDs fast**. Watching how quickly the cluster-wide counter advances turns both issues into a single, cheap alarm.

The signal

- Normal writes advance `next_oid` by ~1 per toasted value.
- A storm advances it by **millions/sec** — each retry is one `GetNewObjectId()`.
- So **oids/sec** spiking = a backend stuck probing candidates.

Collision vs exhaustion

- High rate + **density $\approx 99\%$** (`pct_used`) → **exhaustion** (permanent).
- High rate + **low/moderate density** → **collision** (transient, self-clears).
- The rate flags *a storm*; the `chunk_id` density (earlier slide) says *which*.

```
-- Cluster-wide OID position & headroom before it wraps at 2^32
SELECT next_oid,
       (2::bigint^32) - next_oid AS oids_remaining,
       round(100.0 * ((2::bigint^32) - next_oid) / (2::bigint^32), 4) AS pct_to_wrap
FROM pg_control_checkpoint();

-- Rate = wrap-safe delta of next_oid between two samples, over elapsed time
-- oids_consumed = ((cur_oid - prev_oid) % 2^32 + 2^32) % 2^32 -- survives one wrap
-- oids_per_sec = oids_consumed / elapsed_seconds -- the metric to alarm on
```

Reading next_oid: wrap-safety & the source

Why the modular arithmetic? $\text{oids_consumed} = ((\text{cur_oid} - \text{prev_oid}) \% 2^{32} + 2^{32}) \% 2^{32}$

- The counter is **32-bit** and wraps at 2^{32} , so between two samples `cur_oid` can be **numerically smaller** than `prev_oid` (it lapped past the top). A plain `cur_oid - prev_oid` would go **negative** and the rate would look wrong.
- $\% 2^{32}$ folds that difference back into the ring; the extra $+ 2^{32} \% 2^{32}$ forces the result **non-negative** (guards against a negative modulo), giving the true **forward distance** the counter travelled.
- **“Survives one wrap”**: it’s exact as long as **fewer than 2^{32}** OIDs were consumed in the interval (at most one lap). Sample often enough that a storm can’t lap the whole space between reads.

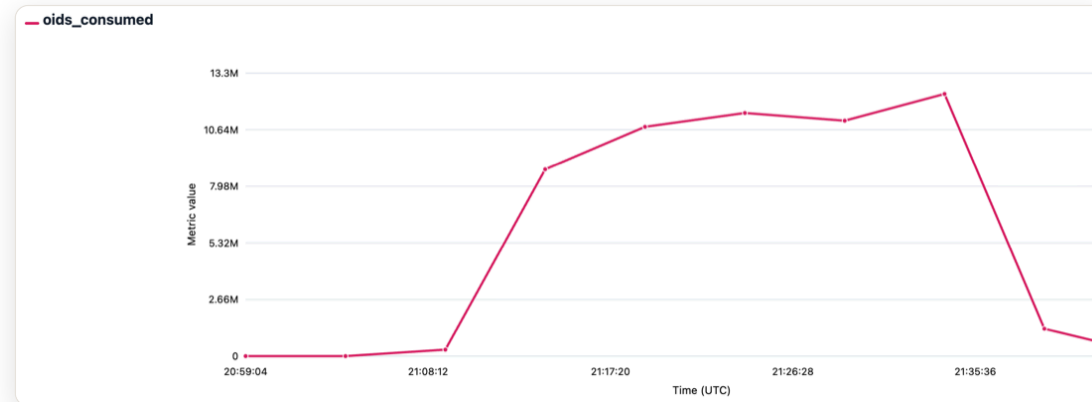
```
-- Example: 200,000 OIDs consumed across a wrap (2^32 = 4,294,967,296)
prev_oid = 4,294,900,000    cur_oid = 132,704    (wrapped)
cur - prev                = -4,294,767,296    ← naive delta: negative, useless
(cur - prev) % 2^32        = -4,294,767,296    ← % keeps the sign
+ 2^32) % 2^32 = 200,000    ← normalized = OIDs actually consumed
```

How `pg_control_checkpoint()` works

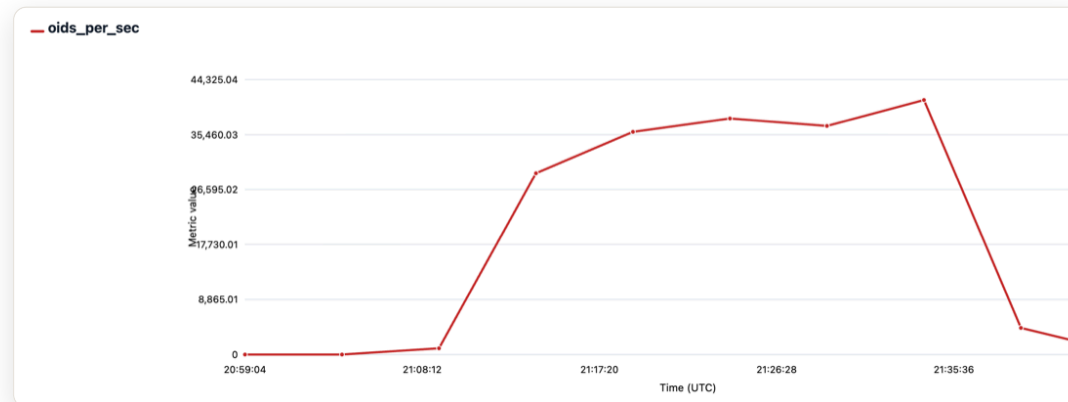
- Superuser function returning **one row** from the cluster **control file** (`global/pg_control`) — the state saved at the **last checkpoint**.
- One column is `next_oid`: the next value the cluster-wide OID counter (`GetNewObjectId()`) will hand out; it wraps at 2^{32} .
- **Cheap**: it reads control state, *not* a table scan — safe to sample often.
- **Only refreshes at a checkpoint**: `next_oid` here is the value written at the **last checkpoint**, and checkpoints are infrequent — by default `checkpoint_timeout = 5 min`. The live in-memory counter runs ahead in between, so this reading is **stair-stepped**, not continuous.
- **So the rate depends on checkpoint timing, not real time**: sample every second/minute and you’ll see **0 oids/sec** between checkpoints, then a big jump right after one — even during an active storm. Force a fresh reading with `CHECKPOINT;` before each sample, or average the rate over several checkpoint intervals. (A live reader of the in-memory counter avoids this entirely.)

What the metric looks like during a storm

Sampled across a real retry storm: the cumulative counter jumps and the per-second rate spikes — exactly the signal to alarm on.



oids_consumed — cumulative OIDs burned (wrap-safe delta) climbs steeply as the storm runs.



oids_per_sec — the rate spikes to millions/sec: a backend stuck probing candidates.

The logging blind spot: nothing below 1,000,000 retries

The only server-log evidence of an OID retry storm comes from `GetNewOidWithIndex()`, and those lines are **gated**. Below ~1 million probes PostgreSQL logs **nothing** — the statement just runs slower, silently. This applies equally to **exhaustion** and **collision**: the floor is about the probe count, not the cause.

```
-- What the server log shows, by retry count (GETNEWOID_LOG_THRESHOLD = 1,000,000)
retries < 1,000,000      → (silent – no log line at all)
retries == 1,000,000     → LOG: still searching for an unused OID in relation "pg_toast_..."
retries = 2M, 4M, 8M, ... → LOG: still searching ...          (logged at each doubling)
resolved, N > 1,000,000  → LOG: new OID has been assigned ... after N retries
```

Case 1 · short burns never cross the floor

- A storm that finds a free OID in **under 1M probes** logs **nothing**.
- You feel the latency on wide INSERT/UPDATES, but the log is empty — no table name, no retry count.
- Happens early in both issues: a dense region the counter only *clips*, or a TOAST table not yet full enough to force $\geq 1M$ probes.

Case 2 · cancelled / timed out before 1M

- App **timeout + retry**, a manual Ctrl-C, or `statement_timeout` ends the statement **before** the probe count reaches 1M.
- Still **no log line** — the storm is aborted below the floor.
- Symptom you see is an **application-side retry storm / timeouts**, with the server log giving no hint that OID allocation is the cause.

Below the floor, don't rely on the log. Lean on signals that show the *pressure* even when it's silent: the **OID-consumption rate** (oids/sec, previous slide), the **chunk_id density** check (collision vs exhaustion), and **wait events** during the stall (LWLock:OidGen, buffer/IO). The `still searching` log only confirms a storm once it's already $\geq 1M$ probes deep.

The community fix: 8-byte TOAST values

The fix in one line: give TOAST tables **8-byte** value ids (opt-in per table) so the id space is effectively unlimited — no **4 billion** ceiling, no wraparound, no retry storm.

pgsql-hackers — “Support for 8-byte TOAST values (aka the TOAST infinite loop problem)”, led by Michael Paquier.

- Adds an `oid8/oid8` type, a second on-disk TOAST pointer, 8-byte OID generation, and an opt-in `toast_value_type` reloption (table rewrite required).
- Reviewers (Tom Lane, Andres Freund, Robert Haas) pushed back on the redirection/abstraction layer; Andres also floated **per-TOAST-table sequences** — which would remove the wraparound collision (Issue #2) outright.

Status: only the foundational `oid8` type landed on HEAD (Jan 2026). The full feature was **dropped from v19** and will be **reworked for v20** without the function-pointer redirections. → On **v19**, the mitigations in this deck are still your only options.

Read the thread: [pgsql-hackers — Support for 8-byte TOAST values ↗](#)

PATCH IDEAS · MAKE IT OBSERVABLE

Closing the gap: core patches for v20+

The real gap today is **visibility** — above all the sub-1M **logging blind spot**. These fixes belong in **core**, in a future major release.

Four core patch ideas

- **A · `pg_next_oid()`** — a live, un-quantized reader of the cluster-wide next OID (no checkpoint lag).
- **B · `OID-search errcontext`** — a cancelled / timed-out search names the table + exact probe count, even below 1M.
- **C · `log-threshold GUC`** — lower the 1M floor so successful sub-1M storms log too.
- **D · `OID-retries metric`** — an always-on, attributable counter per query / table / database.

These are **patch ideas** — I have working **beta versions** of each that I plan to submit to the PostgreSQL community.

PATCH IDEA A

pg_next_oid() — a live reader of the OID counter

Goal: read the *live* cluster-wide next OID for monitoring, with no checkpoint dependency.

`pg_control_checkpoint().next_oid` is frozen at the last checkpoint (default 5 min); this reads the in-memory counter directly for a zero-lag `oids/sec` rate.

PATCH IDEA B

Annotate a cancelled OID search (errcontext)

Goal: when a statement is cancelled, times out, or is terminated **while** searching for an OID — even **below** 1M retries — the error carries a CONTEXT line naming the relation and the exact probe count. Closes blind-spot **Case 2**.

```
ERROR: canceling statement due to statement timeout
CONTEXT: while searching for an unused OID in relation "pg_toast_69643053" (candidates checked 190342 times)
```

PATCH IDEA C

A GUC for the log threshold

Goal: let ops lower the **1,000,000**-retry floor so *successful* sub-1M storms also log still searching for an unused OID. Closes blind-spot **Case 1** for proactive visibility.

Design

Replace the `#define GETNEWOID_LOG_THRESHOLD 1000000` (catalog.c) with an int GUC (e.g. `getnewoid_log_threshold`, default 1M, SIGHUP); seed `retries_before_log` from it and keep the existing exponential back-off doubling.

PATCH IDEA D

An OID-retries metric — every level

Goal: every OID probe is counted today and then thrown away. Add a **cumulative counter** for those retries — **always-on**, counting **every probe (including sub-1M)**, and surfaced through the stats views so it is **directly attributable** to the offending query and table.

D1 · which query — new column `oid_search_retries` on `pg_stat_statements`

```
SELECT query, oid_search_retries
FROM   pg_stat_statements
ORDER BY oid_search_retries DESC LIMIT 5;
```

Benefit: pinpoints the exact statement burning OIDs.

D2 · which table — new column on `pg_stat_all_tables`

```
SELECT c.relname, t.oid_search_retries
FROM   pg_stat_all_tables t
JOIN   pg_class c ON c.reltoastrelid = t.relid
ORDER BY t.oid_search_retries DESC;
```

Benefit: the count lands on the TOAST relid — join `reltoastrelid` to name the owning table whose TOAST is saturating.

D3 · fleet alarm — new column on `pg_stat_database`

```
SELECT datname, oid_search_retries
FROM   pg_stat_database
ORDER BY oid_search_retries DESC;
```

Benefit: a coarse, cheap fleet-wide alarm to watch for OID pressure trending up.



databricks

Q&A