Tiger Data

Zooming in on OOM

A Deep Dive into Postgres and Linux Memory

Oleksii Kliukin · Dimitris Sideris

Platform team · **TigerData**, Swiss PGDay 2026 🔥

A bad day in production

We run **thousands** of Postgres services on Kubernetes.
Memory overcommit is unavoidable.

```
LOG:  server process (PID 2420332) was terminated by signal 9: Killed
DETAIL:  Failed process was running: REFRESH MATERIALIZED VIEW CONCURRENTLY …
LOG:  terminating any other active server processes
```

Three questions

1. How much memory does Postgres **actually** request?
2. Why do Postgres and Linux metrics **disagree by gigabytes**?
3. What happens when the OS **can't** fulfill a request?

01

Killed by the kernel

A short play

How Postgres asks the OS for memory and what happens under **overcommit**.

Cast: 🐘 Postgres · 🐧 Linux kernel · 🐃 glibc *(we need one volunteer!)*

`vm.overcommit_memory = 2`

never overcommit

`malloc` returns **NULL** → Postgres survives

`vm.overcommit_memory = 1`

always overcommit

> `malloc` never fails... until **SIGKILL** 🔫

What just happened

- `malloc` returns a **promise**, not RAM
- RAM is spent on **first touch** (page fault)
- Modes: `0` heuristic *(default)* · `1` always · `2` strict
- In containers, the cgroup `memory.max` is a second, harder ceiling
- **SIGKILL = no cleanup, no rollback**

Why this is brutal for Postgres

- A backend dies by **signal 9**, uncatchable
- Postmaster must assume shared memory is unsafe
- → terminates **all** backends → crash recovery
- Seconds–minutes of downtime, every connection dropped
- Patroni can't save you: **fail over** vs **wait**, both bad

Our safeguard, and the new symptom

- For years: a custom `LD_PRELOAD` library capped allocations
- It worked, far fewer OOM kills
- Then workloads grew… **the OOMs crept back**
- Allocations were **bypassing our cap**. *Why?*

02

Where does the memory go?

Allocation ≠ memory

- **Virtual memory** (VSZ): what you asked for
- **RSS**: what you actually touched, real pages
- Demand paging: a page costs nothing until written
- Under overcommit, `malloc` almost never returns NULL

The allocator layer: glibc

Postgres calls `malloc` → glibc **ptmalloc**, which has two paths to the kernel:

brk → arena
small allocations · freed memory kept for reuse → RSS stays high

mmap
≥ 128 KB · `munmap` on free → RSS drops

One process per connection

postmaster
listens on the port · forks one backend per connection · no threads

`fork()` ↓

backend
session A

backend
session B

backend
session …

× one per connection · private heap each

`↓ each inherits one shared region, at the same virtual address`

Shared memory
`mmap(MAP_SHARED | ANON)` · one region for the whole instance

Shared memory is the only bridge. Everything else is **per-process**, and multiplies with connections.

The Postgres memory model

Shared memory
once per instance
`shared_buffers` · WAL buffers · locks
→ `pg_shmem_allocations`

+

Backend
`work_mem` ×N · temp buffers · catalog & relcache

Backend
…

Backend
…

× hundreds of connections

What the kernel also counts

Page tables
the big one
PTEs mapping `shared_buffers`, one set per backend

Socket buffers
kernel TCP send/recv, one socket per connection

Kernel stacks
one per process, every backend

Slab
`task_struct` · dentries · inodes · fd tables

× every backend · scales with connections

Charged to cgroup `memory.max`. None of it in `pg_shmem_allocations` or any Postgres metric.

The multiplier: per-backend × connections

Tiger Data

- `work_mem` is **per sort/hash node, per backend**, one query can use it many times
- Every backend also **maps** `shared_buffers` → its own **page-table entries**
- ≈ `connections × shared_buffers × 8 B / 4 KB` → **GBs** of page tables *(kernel memory!)*
- Plus **slab**: kernel per-process bookkeeping
- Even idle connections cost

Why Postgres and Linux disagree by GBs

- **Postgres counts:** logical allocations
- **Linux counts:** RSS + page tables + slab + allocator retention
- The **gap** = page tables + ptmalloc free lists + kernel overhead
- That gap is what the kernel **counts against the limit**
- It behaves like a **floor** under every query, and it **rises with each connection**

03

The guard

The guard, in one line

- An `LD_PRELOAD` library intercepts `malloc`
- Over the limit? Return `NULL` + `ENOMEM`, before the kernel runs out
- Postgres turns NULL into a clean `ERROR: out of memory`

```
errno = ENOMEM;    /* tell the caller we're out of memory */
return NULL;       /* refuse the allocation, don't let the kernel kill us */
```

How it intercepts

- Override the whole `malloc` family: `malloc` · `free` · `realloc` · `posix_memalign` · …
- Hot path stays **local** (a pre-reserved chunk); shared counter touched only when it runs out · **lock-free**

Account
size → per-process slot + global counter

→

Decide
counter + size < limit? · blockable?

→

Return
fits → `malloc()` over → `NULL` + `ENOMEM`

Blocking the right process

Blockable
query backends & parallel workers
hooked at auth (`ClientAuthentication` / `ExecutorStart`)
over the limit → `NULL` → clean `ERROR`, session lives

No hook yet
autovacuum workers & other auxiliaries
can eat a lot of memory, but no hook marks them
idea: intercept `fork` / `exec` of the new pid, then read `/proc/<pid>` to identify it

Never block
postmaster, checkpointer, walwriter, archiver launchers, replication, Patroni
a failed `malloc` here → crash, failover, or lost WAL

Return `NULL` only to processes that can survive it.

Still getting killed

- The guard turned most OOM kills into a clean `ERROR: out of memory`
- But a residual stream of **SIGKILLs kept coming**
- Sometimes the guard was **well under its limit** when the kernel struck

> The cap worked on paper. The kernel disagreed.

04

Why does the OOM killer fire?

Flying blind

- The guard capped allocations… yet OOMs returned
- All we knew: *"a container was OOM-killed"*
- **v1:** a DaemonSet on containerd's `/tasks/oom` topic
- No PID, no process, no query, can't line up with PG logs
- **You can't fix what you can't see**

eBPF to the rescue

- Hook the kernel OOM killer itself: the `mark_victim` tracepoint
- Capture the **victim** *and* the **invoker**
- **Host** *and* **container** PIDs (namespace-aware)

```
struct oom_event {
    __u64 timestamp;
    __u32 victim_pid;    __u32 invoker_pid;
    __u32 victim_nspid;  __u32 invoker_nspid;
    __u32 container_init_pid;
};
```

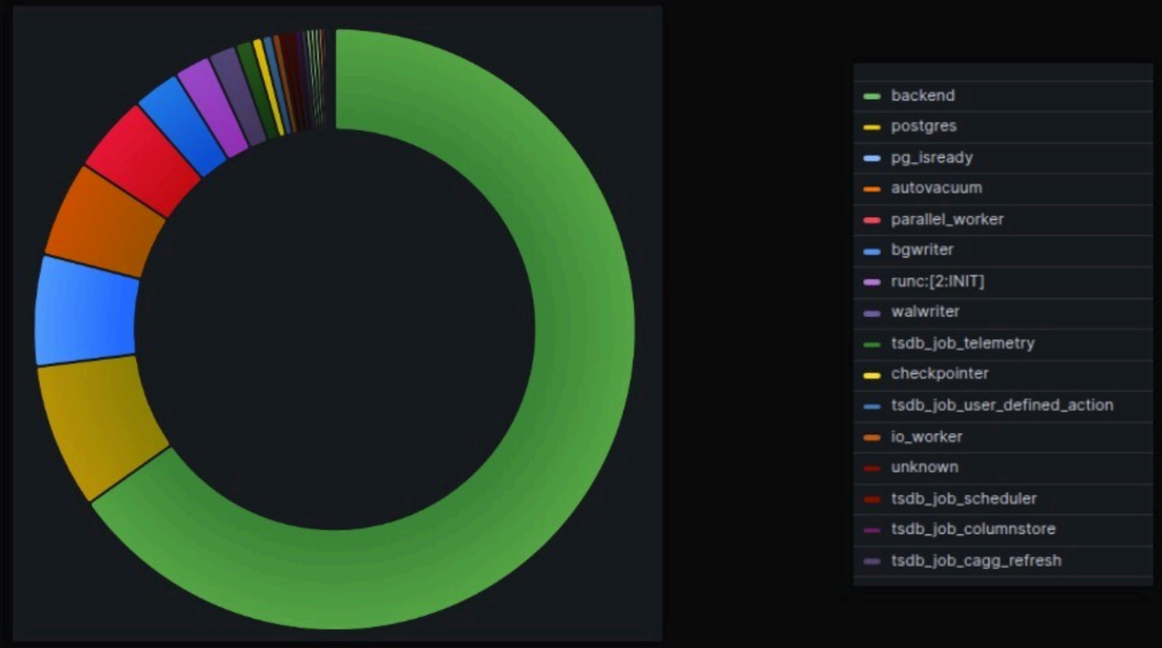
Naming the victim

- Resolve PID → identity via `/proc/<pid>/cmdline`

```
switch {
case strings.HasPrefix(cmdline, "postgres:"): // "autovacuum worker", "tsdbadmin@tsdb …"
    return strings.TrimPrefix(cmdline, "postgres:"), nil
case strings.HasPrefix(cmdline, "postgres"):
    return "postmaster", nil
}
```

- Before eBPF, the biggest bucket was literally **unknown** (35,659)

User backends



~two-thirds are ordinary query backends (67%, +3% parallel workers), not the postmaster, not background jobs.

Two root causes

1. `free()` didn't return memory to the OS (ptmalloc)
2. The **limit formula** under-counted page tables & slab

05

Building a reliable cap

Root cause #1: the arena trap

ptmalloc's two paths to the kernel:

- `brk` → the arena; freed memory stays in internal free lists, **not returned**
- `mmap` → standalone mapping; `munmap` on free **returns it to the OS**

The **128 KB** threshold decides which, and we **froze** it. *(our `mallopt(M_TRIM_THRESHOLD, …)` disables glibc's dynamic adjustment → threshold stuck at 128 KB)*

`free()` doesn't return it to the kernel

free() drops brk and returns the top free pages, but stops at the first used chunk, so the interleaved free below stays resident

ptmalloc returns this heap to the OS only by lowering the `brk` horizon, and only down to the **topmost used chunk**, so the freed pages below it stay resident. `malloc_trim(0)` can `madvise` them back, but only on demand and only whole free pages.

Postgres memory contexts

A tree of contexts

every `palloc` lands in the current context

`TopMemoryContext` · process life

├ `CacheMemoryContext` · plans, relcache

├ `MessageContext` · one client message

└ `PortalContext` · one query

reset → frees the whole subtree, no per-object `free()`

palloc → malloc

`palloc(N)` → a slot in the current `AllocSet` block

block full → one real `malloc()`

blocks grow geometrically: 8 KB → 8 MB

→ N pallocs, only a few mallocs

Reset frees a context instantly, but the memory often stays in the allocator, **not handed back to the OS**.

Why Postgres never escaped to `mmap`

Tiger Data

- AllocSet doubles blocks **8 KB → 8 MB**, the big ones *would* `mmap` ✓
- But a cursor's `PortalContext` uses `ALLOCSET_SMALL_SIZES` → **capped at 8 KB**
- 8 KB < 128 KB → **every block goes through `brk`**, into the arena
- Trigger: hundreds of OR/AND clauses on a **570-chunk hypertable** → TimescaleDB expands into **thousands** of per-chunk nodes

The evidence

`PortalContext: 724,496,896 bytes in 88,436 blocks` → exactly **8,192 B/block**

TIME	CGROUP	GUARD	ARENA	note
09:05:35	2238 MB	1779 MB	1755 MB	
09:05:36	2016 MB	60 MB	1520 MB	backend frees → guard 60, kernel 1546
09:05:42	3520 MB	1568 MB	3020 MB	next backend stacks on top…
09:05:46	53 MB	2146 MB	4 MB	💀 OOM KILL

After each free, **~1500 MB stays trapped in the arena**, guard sees 60 MB, the kernel sees 1546 MB.

The fix: jemalloc

- `malloc_trim(0)` *can* `madvise` free pages back, but it is **manual** and **page-granular** *(a live chunk sharing a page keeps it; ptmalloc never does this on its own)*
- **jemalloc**, decay off → returns freed memory to the OS **immediately**:

```
const char *malloc_conf = "narenas:2,dirty_decay_ms:0,muzzy_decay_ms:0";
```

- Now RSS tracks the guard's accounting → the cgroup is never surprised

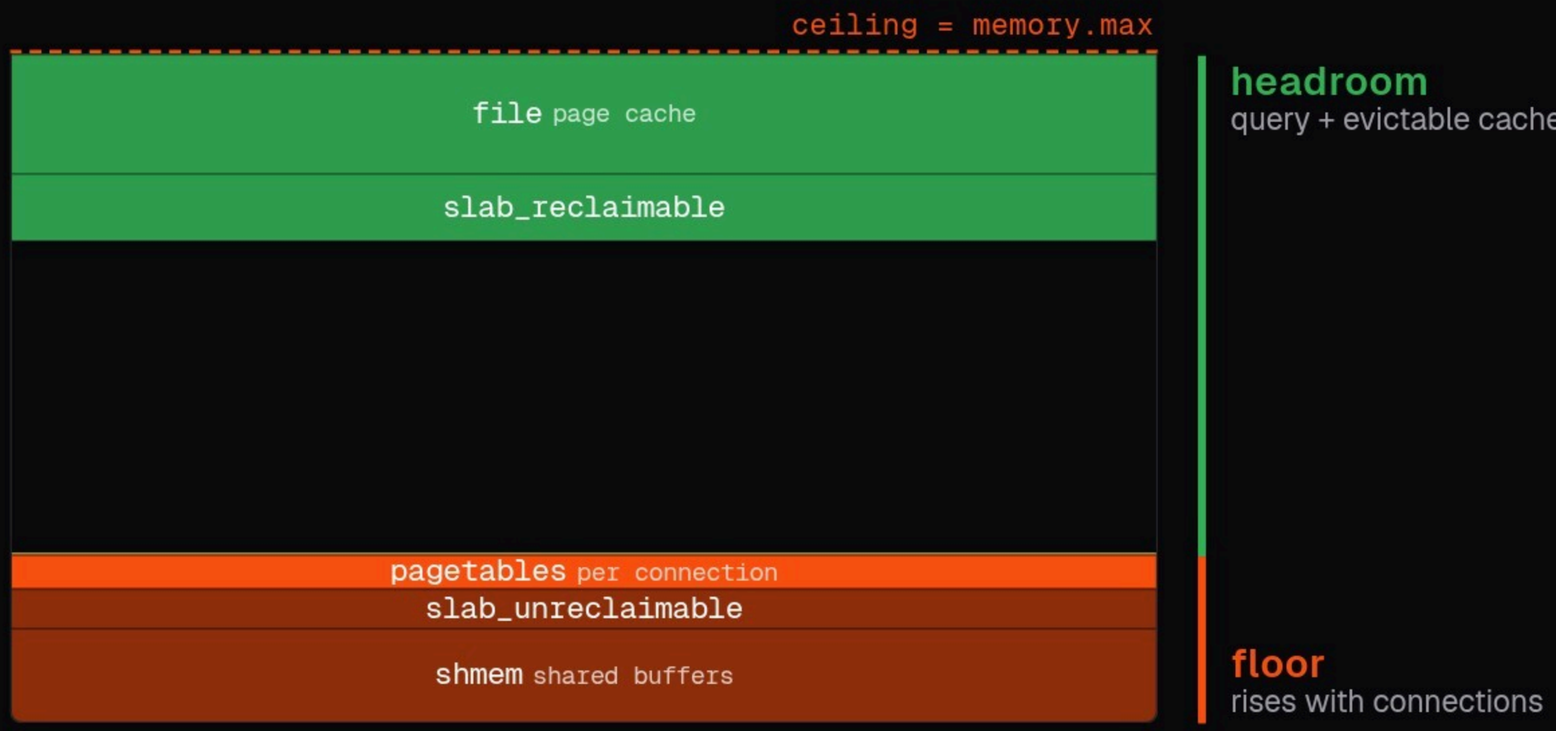
jemalloc in action

Same workload, same container, different allocator

```
TIME       CGROUP    ANON      GUARD    note
07:49:34   3097 MB   2103 MB   2075 MB
07:49:35   1097 MB    101 MB     59 MB  backend frees
07:49:44   3102 MB   2100 MB   2035 MB
07:49:46   1190 MB    186 MB    108 MB  backend frees
```

After each free, the guard **follows the allocated memory properly**

Root cause #2: Picking the right limit



Idle: mostly idle connections, small **page tables**, no query memory. The **floor** is just shmem and unreclaimable slab, leaving plenty of **headroom**.

Root cause #2: Picking the right limit

Connections go active: each backend builds page tables and runs queries, so the floor rises and **anon** (work_mem) appears. Headroom shrinks.

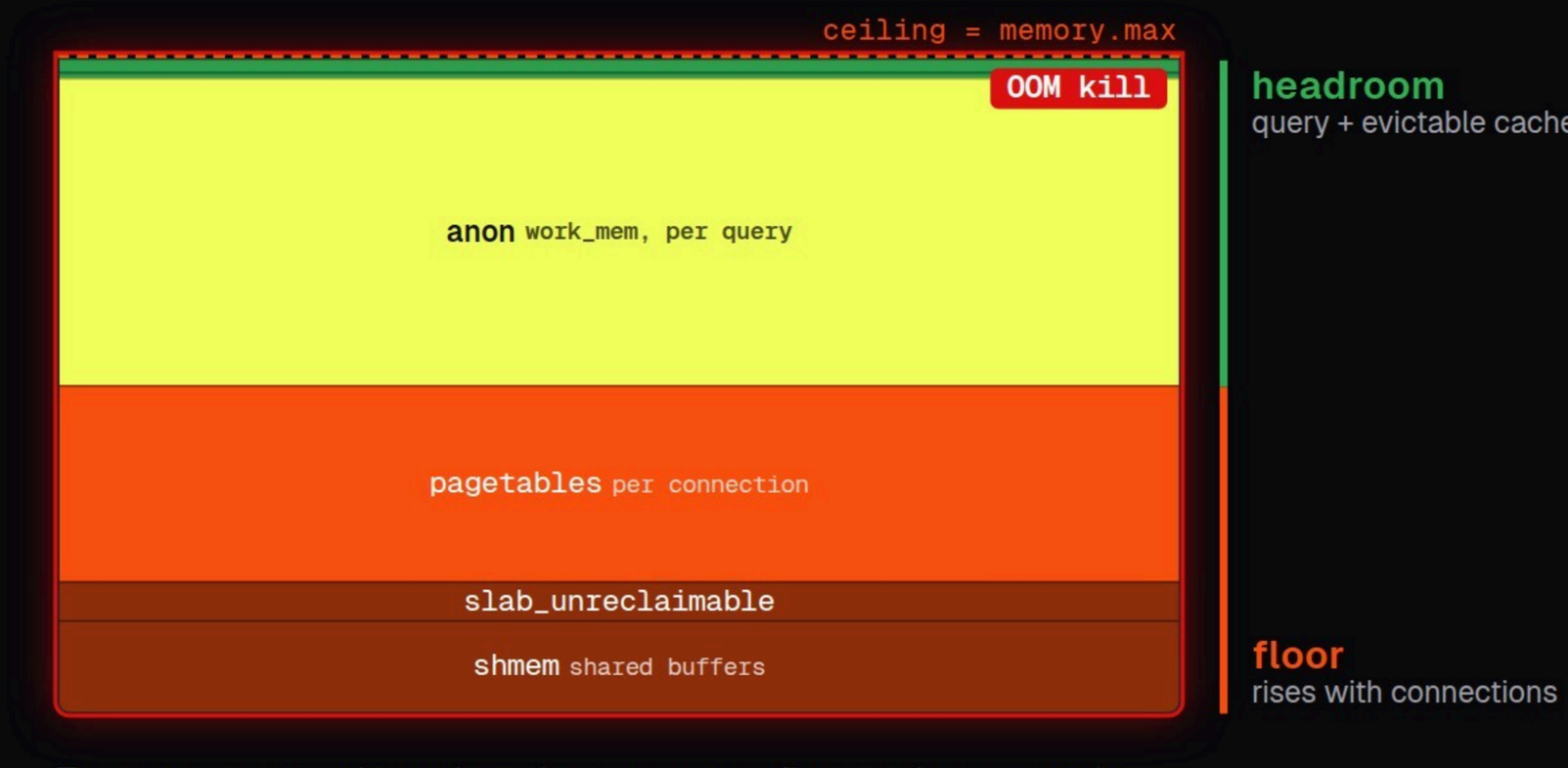
Root cause #2: Picking the right limit

Steady load: the query's anon sits in the headroom, with free space and evictable cache still in reserve.

Root cause #2: Picking the right limit

Under pressure the kernel evicts **file cache** and reclaimable slab, yielding room so anon can keep growing.

Root cause #2: Picking the right limit



Too many connections plus a hungry query: floor and anon reach `memory.max` with almost nothing left to reclaim, so the kernel **kills**.

The new formula

```
pagetables  = effective_connections * shared_buffers_bytes * 8 // 4096
slab        = 100 * 1024 * 1024
limit_bytes = total_ram - (shmem_bytes + kernel_overhead)
```

- shared memory from `pg_shmem_allocations`
- cross-checked against cgroup `memory.stat`, take the larger

limit = memory.max – floor (the same floor we watched rise)

Mechanism vs. policy

Inside Postgres
LD_PRELOAD library + extension
intercept · account · block
the mechanism, rarely changes

Outside Postgres
control script / node agent
compute & push the limit
the policy, changes often

They share just **one number in shared memory**, the limit.

> Change the tuning logic → **no Postgres restart**. The policy evolves while every backend keeps running.

Autotune v1: Patroni callbacks

- A control script reruns on Patroni `on_start` / `on_reload` / `on_restart`
- ⚠ Re-tunes only at those events, so the limit goes **stale** between them

Autotune v2: continuous, in the node agent

- Move tuning **into the node agent** (the DaemonSet from §3)
- Continuously re-tune from live cgroup `memory.stat`
- The observability agent becomes the **control loop**

> The thing that watched the kills now **prevents** them.

Rolling it out safely

- Feature-flagged, **off by default**; **shadow mode** first
- Every autotune **decision** logged → a **TimescaleDB hypertable** (we dogfood)
- …plus OOM events enriched with **victim / invoker**
- → **back-test** a new limit against real kills before enforcing

06

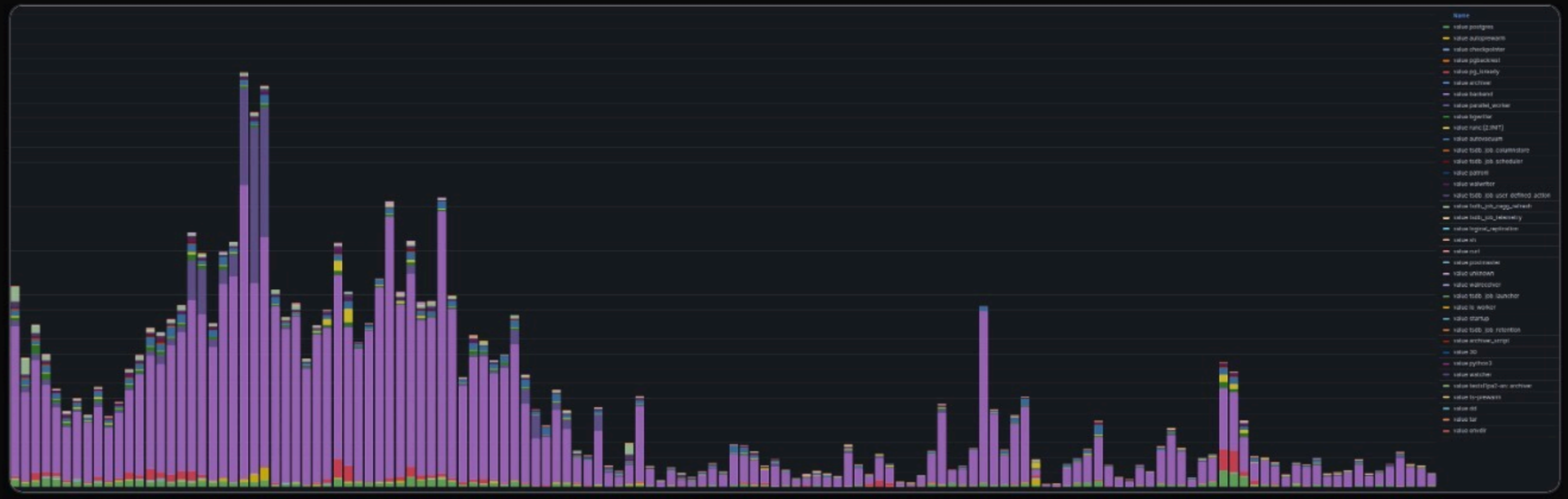
Results

−91.4%

- OOM events: **last 7 days vs. first week of the year**
- **Before:** SIGKILL → postmaster restarts → **whole instance down**
- **After:** recoverable per-query `ERROR: out of memory`

Tiger Data

OOM kills per day: six months



07

Takeaways

Minimize Postgres memory overhead

- **Connection count** drives page tables, pool aggressively
- Right-size `work_mem` (per-operation, per-backend)
- Consider the **allocator** (jemalloc + decay tuning)
- Watch `shared_buffers` × connections **together**

Linux memory toolkit

- **eBPF / bpftrace**, see kernel events (like OOM kills)
- `/proc/<pid>/smaps`, real per-process memory
- `/sys/fs/cgroup/memory.stat` (in the pod), pagetables, slab: the truth
- on the node: `dmesg` / `journalctl -k`, the OOM killer's report (victim + per-task memory)

Postgres memory toolkit

Tiger Data

- `pg_shmem_allocations`, static shared memory
- `pg_dsm_registry_allocations`, dynamic shared memory *(Postgres 19+)*
- `pg_backend_memory_contexts`, per-backend `palloc` contexts
- `pg_log_backend_memory_contexts(pid)`, dump another backend's contexts to the log
- `EXPLAIN (ANALYZE, VERBOSE, BUFFERS, MEMORY)`, per-node memory & buffers

What's next

- **Block what we can't yet**, autovacuum workers & other auxiliary processes are memory-hungry but have no block hook; the promising path: intercept `fork` / `exec` for the new PID, then read `/proc/<pid>` to identify it by name
- **Account for `slab_reclaimable`**, reclaimable in principle, but not freed fast enough under pressure, so it still counts against us
- **Huge pages**, 2 MB pages instead of 4 KB → **far fewer page-table entries** → shrinks the biggest "extra memory" source

Thank you  

Tiger Data

- `malloc` is a **promise**; the kernel keeps score differently
- Account for **page tables, slab, allocator retention**
- Use `vm.overcommit_memory = 2` if you can
- **Collect the data** before making decisions

Questions?

📝 **Don't forget to leave feedback**

2026.pgday.ch/feedback?proposal_id=437