coroot :~#

PERCONA UNIVERSITY · MONTEVIDEO

Beyond “Postgres is up”
Detecting failures before they become incidents

Nikolay Sivko · Coroot
Agustín Gallego · Percona

Where this talk comes from

coroot:~# an open-source observability platform

THE GOAL

Make an engineer's life easier

Not busier. One more dashboard is not help.

THE STANCE

Not just that something is wrong

Saying that is the easy half. We try to answer what is happening, and why.

THE WORK

Percona and Coroot, together

We have been extending what Coroot can tell you about a database.

THE METHOD

Break it, then look

We simulated failures against each database and asked what it takes to name the cause.

So that nobody has to open psql, walk through system views, and google what a column means at 3 a.m.

A database is a set of promises

Durable. A commit survives a power cut.

Consistent. Every query gets its own snapshot.

Available. A node dies, another takes over.

Recoverable. Rewind to 14:32, before the bad DELETE.

Still fast. After the data grows 50x.

Kept by machinery you never call

CHECKPOINTER

Crash recovery stays bounded, with no fsync per write

AUTOVACUUM · RECLAIM

Dead tuples are reclaimed and reused

AUTOVACUUM · FREEZE

Transaction IDs never wrap

ANALYZE

The planner keeps choosing good plans

WAL ARCHIVING

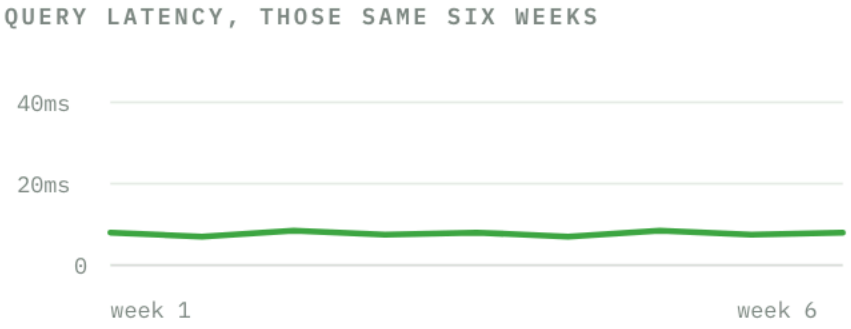
Yesterday stays restorable

REPLICATION

A standby worth promoting

Not on the query path. Still able to take you down.

When this machinery breaks, queries stay fast and the dashboard stays flat. The day it broke and the day you find out are **weeks apart**.



Nothing to see. That is exactly the problem.

What goes wrong, and what it costs

WHAT BREAKS IN THE BACKGROUND	WHAT IT COSTS YOU
Autovacuum falls behind	Tables bloat, every query gets slower
Transaction ID freezing falls behind	The database refuses writes
The standby stops streaming	Failover promotes a stale copy
Checkpoints drift out of tune	Crash recovery takes hours

So let’s break things on purpose

01

Let’s understand it

What is the mechanism doing?

02

Let’s break it

Force the real failure on a real cluster

03

Let’s watch it

Which system view moves first

04

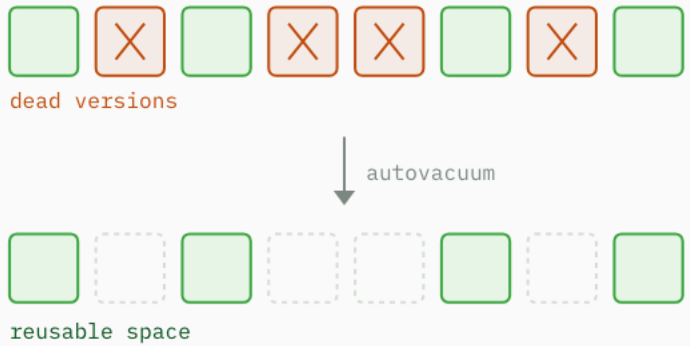
Let’s automate it

A check that names the cause and the fix

SCENARIO 1 OF 3

Autovacuum

Nothing is overwritten in place. Every write leaves the previous version of the row behind.



INSERT

Writes one new row version.

UPDATE

Writes a new version. The old one stays, marked as replaced.

DELETE

Writes nothing new. The row is only marked as removed.

An old version becomes a **dead tuple** once no running transaction can still see it. The oldest transaction still running draws that line: the **vacuum horizon**.

Autovacuum falls behind

Postgres cleans up after itself in the background. Here is what happens when it cannot.

Dead tuples are not reclaimed → Tables and indexes bloat → Every query reads more pages → **Queries slow down. The disk fills.**

What Postgres already counts per table

One row per table in `pg_stat_user_tables`, kept up to date by the cumulative statistics system.

N_DEAD_TUP

Dead tuples

Waiting to be reclaimed. An estimate, counted in rows.

N_LIVE_TUP

Live rows

Visible to new transactions. An estimate as well.

LAST_AUTOVACUUM

When it last ran

The difference between “never ran” and “ran and removed nothing”.

How Postgres decides to vacuum

```
-- autovacuum picks up a table when
n_dead_tup >= autovacuum_vacuum_threshold
            + autovacuum_vacuum_scale_factor * n_live_tup
```

Defaults: threshold 50, scale factor 0.2

Which literally means **50 tuples, plus 20% of the live tuples in that table**

Alert on the ratio, not the count

```
pressure = n_dead_tup / (threshold + scale_factor * n_live_tup)
```

PRESSURE ≈ 1

Autovacuum is keeping up

The same number for a 100 MB table and a 4 TB one.

PRESSURE > 2, SUSTAINED

It should have been vacuumed

Postgres decided to run. Something stopped it finishing.

AND ≥ 512 MIB DEAD

The noise gate

A tiny table can sit at 50x and cost nothing. Skip it.

Autovacuum runs and removes nothing

HOW THIS HAPPENS

- A service opens a transaction, calls a third party inside it, and the call hangs
- Or someone types BEGIN in psql and goes to lunch

One open transaction pins the horizon for the whole database.

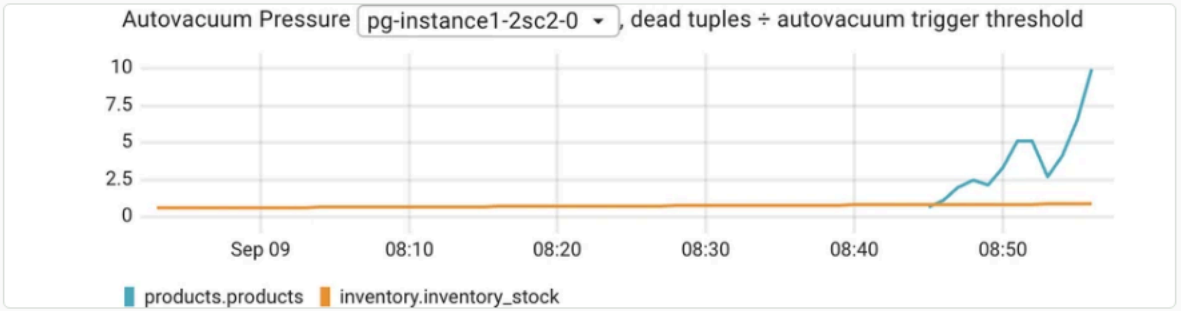
```
BEGIN ISOLATION LEVEL
  REPEATABLE READ;
SELECT 1;  -- then nothing, ever
```

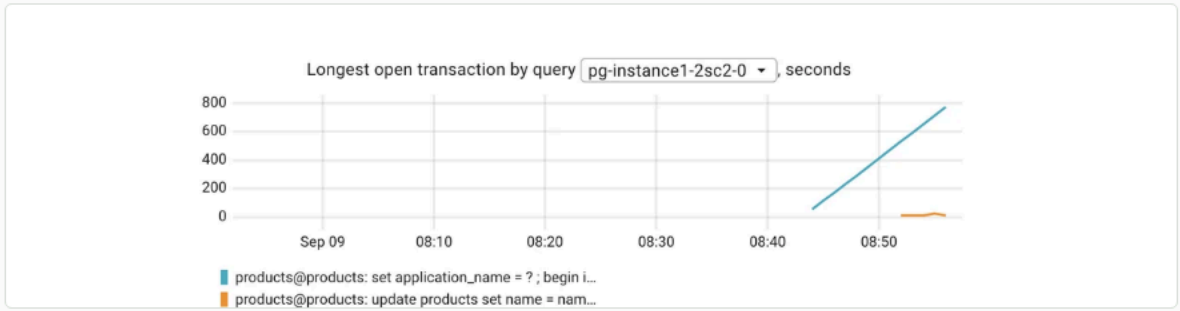


“a running transaction is blocking cleanup”

Alert r2xpqq1ewa10 Firing

Rule	Postgres autovacuum
Application	pg
Severity	Warning
Opened at	Sep 09, 08:53:01 (6m ago)
Duration	6m (ongoing)
Summary	autovacuum is falling behind on 1 postgres instance
Description	Dead rows are accumulating in a PostgreSQL database faster than autovacuum can reclaim them, risking bloat and degraded performance. Check for snapshot-pinning transactions/slots and autovacuum tuning.
Findings	⚠ pg-instance1-2sc2-0: products.products: 10x over the autovacuum trigger threshold, ~1GB of dead rows; a running transaction is blocking cleanup (holds the vacuum horizon). The oldest is in "products", open 13m24s: set application_name = ? ; begin isolation level repeatable read ; select count (*) from products ; select pg_sleep (?)





How old is it, and which transaction is it?

THE QUERY

```
-- how old is the oldest snapshot?
SELECT max(age(backend_xmin))
FROM pg_stat_activity
WHERE backend_type = 'client backend';
-- and which transaction is it?
SELECT datname, usename, query,
  EXTRACT(EPOCH FROM now()-xact_start)
FROM pg_stat_activity;
```

WHAT IT RETURNS

```
   max
----------
   412877

    db     | secs |     query
-----------+------+---------------
 products |  804 | set applicat…
```

Client backends only, transactions open longer than 10 seconds, top few by age.

WHICH BECOMES

```
pg_oldest_xmin_age{holder="running_transaction"} 412877
pg_transaction_seconds{db="products", user="products", query="set application_name = ? ; begin…"} 804
```

Who is holding the vacuum horizon?

Exactly four things can hold it, and each one lives in a different view.

PG_STAT_ACTIVITY

A running transaction

PG_STAT_ACTIVITY · WALSENDER

Standby feedback

PG_REPLICATION_SLOTS

A replication slot

PG_PREPARED_XACTS

A prepared transaction

The replica is gone. Its slot is not.

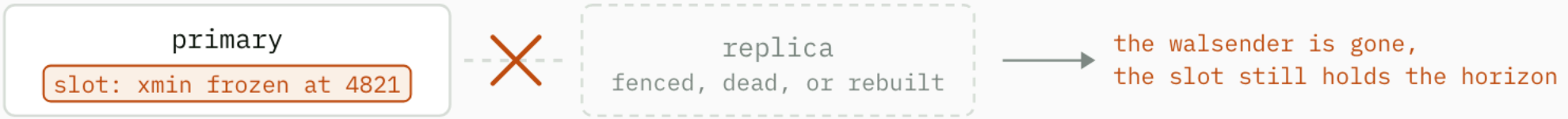
HOW THIS HAPPENS

- A replica serves reports, so `hot_standby_feedback` is on
- The primary holds the horizon back for it, and records that in the replica's slot
- The replica is fenced, or dies, or gets rebuilt somewhere else

The slot stays on the primary, frozen at the last position the replica asked for.

HARD TO CATCH BY HAND

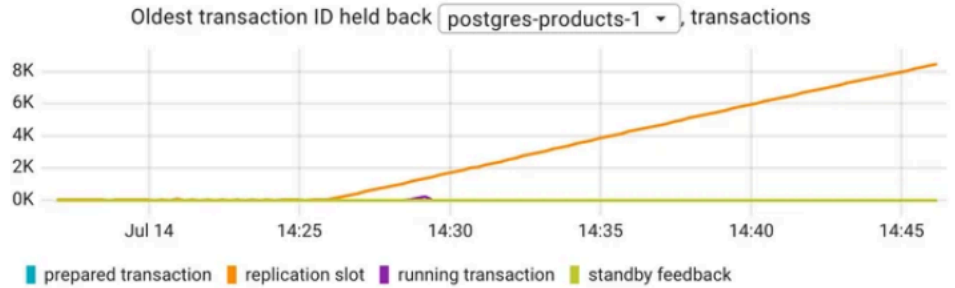
The walsender exits. Nothing looks stuck. The only trace is the old horizon the slot is still holding, in `pg_replication_slots`.



“a replication slot is blocking cleanup”

Alert 4wlkv6tiwl24 Firing

Rule	Postgres autovacuum
Application	postgres-products
Severity	Warning
Opened at	Jul 14, 14:36:56 (10m ago)
Duration	10m (ongoing)
Summary	autovacuum is falling behind on 1 postgres instance
Description	Dead rows are accumulating in a PostgreSQL database faster than autovacuum can reclaim them, risking bloat and degraded performance. Check for snapshot-pinning transactions/slots and autovacuum tuning.
Findings	⚠ postgres-products-1: products.wsat_1: 3x over the autovacuum trigger threshold, ~1GB of dead rows; a replication slot is blocking cleanup (holds the vacuum horizon)





Same metric, a different holder

THE QUERY

```
SELECT max(GREATEST(age(xmin),
                    age(catalog_xmin)))
FROM pg_replication_slots;
```

WHAT IT RETURNS

```
   max
----------
 8231455
```

WHICH BECOMES

```
pg_oldest_xmin_age{holder="replication_slot"} 8231455
pg_oldest_xmin_age{holder="running_transaction"} 0
```

The vacuum runs, and gets nowhere

HOW THIS HAPPENS

Cost settings tuned years ago, for spinning disks

The server is on NVMe now, and vacuum still sleeps between pages.

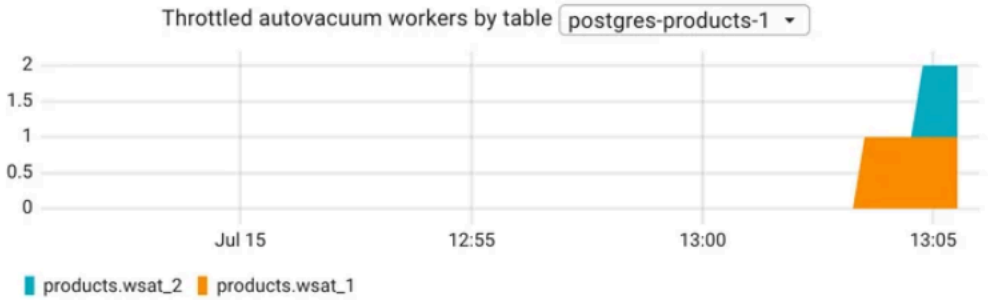
```
ALTER TABLE wsat_1 SET (
  autovacuum_vacuum_cost_delay = 100
);  -- sleep 100ms every cost_limit
```



“vacuum throttled by this table’s cost_delay”

Alert `ti031lldmtvq` Firing

Rule	Postgres autovacuum
Application	postgres-products
Severity	Warning
Opened at	Jul 15, 13:06:01 (1m ago)
Duration	1m (ongoing)
Summary	autovacuum is falling behind on 1 postgres instance
Description	Dead rows are accumulating in a PostgreSQL database faster than autovacuum can reclaim them, risking bloat and degraded performance. Check for snapshot-pinning transactions/slots and autovacuum tuning.
Findings	⚠ postgres-products-1: products.wsat_2: 3x over the autovacuum trigger threshold, ~2GB of dead rows; vacuum throttled by this table's cost_delay=100ms, adjust the cost settings



Is the vacuum working, or sleeping?

THE QUERY

```
SELECT c.relname,
       a.wait_event = 'VacuumDelay'
FROM pg_stat_progress_vacuum p
JOIN pg_class c ON c.oid = p.relid
LEFT JOIN pg_stat_activity a
       ON a.pid = p.pid;
```

WHAT IT RETURNS

```
 relname | throttled
---------+-----------
 wsat_2  | t
 wsat_1  | t
```

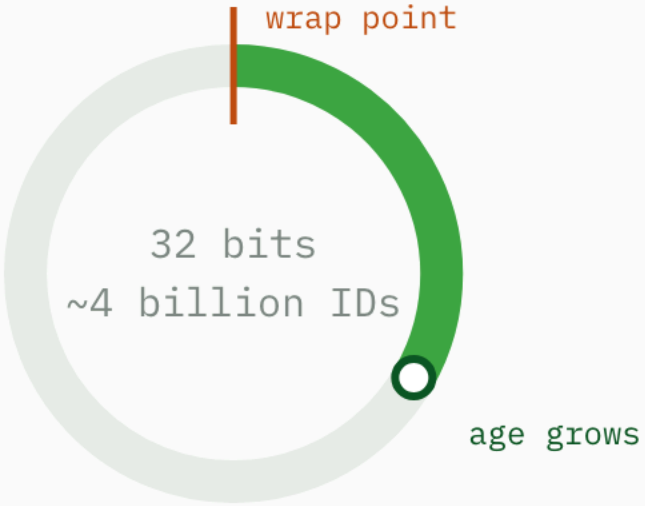
WHICH BECOMES

```
pg_table_vacuum_throttled{schema="products", table="wsat_2"} 1
```

SCENARIO 2 OF 3

Transaction ID wraparound

Every transaction gets an ID from a 32-bit counter. Postgres must keep the oldest one from falling too far behind.



Autovacuum has a second job

Everything so far was about reclaiming space. The same daemon quietly does something else.

EVERY TRANSACTION GETS AN ID

32 bits wide, so about 4 billion of them exist.

POSTGRES COMPARES THOSE IDS

That is how it decides which rows a transaction may see.

THE COUNTER WRAPS

Past 4 billion, the oldest rows would start to look like the future.

So VACUUM **freezes** old rows: stamps them visible to everyone, permanently, and the counter can wrap safely.

Transaction ID freezing falls behind

VACUUM can only freeze rows older than the vacuum horizon. So whatever pins that horizon also stops the age from resetting.

Something pins the vacuum horizon → Nothing older can be frozen, so the age never resets → The age climbs towards the 2 billion limit → **Postgres warns, then refuses new writes**

What the countdown actually looks like

Postgres escalates on its own as the age grows.

AGE	WHAT POSTGRES DOES
200M	Starts an anti-wraparound vacuum, even where autovacuum is off
1.6B	Vacuum drops its cost limits and goes as fast as it can
2.1B	**Writes are refused. The database goes read only.**

The first two thresholds are autovacuum_freeze_max_age and vacuum_failsafe_age.

When to actually page someone

Not the raw age. The age as a share of the limit.

```
risk = max(pg_xid_age, pg_multixact_age) / 2^31
```

It fires above 50%. Half the transaction IDs are already spent.

Then it names the holder. The oldest one, if it accounts for most of that age.

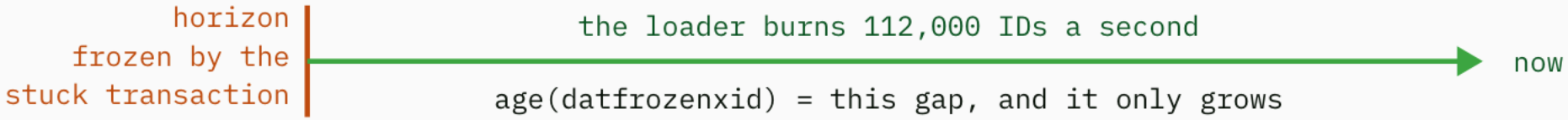
One stuck transaction, one busy loader

HOW THIS HAPPENS

- A report opens a transaction, takes its snapshot, and then hangs
- A loader puts every row in its own savepoint, so one commit uses 65 transaction IDs instead of one

One holds the horizon still. The other spends 112,000 IDs a second.

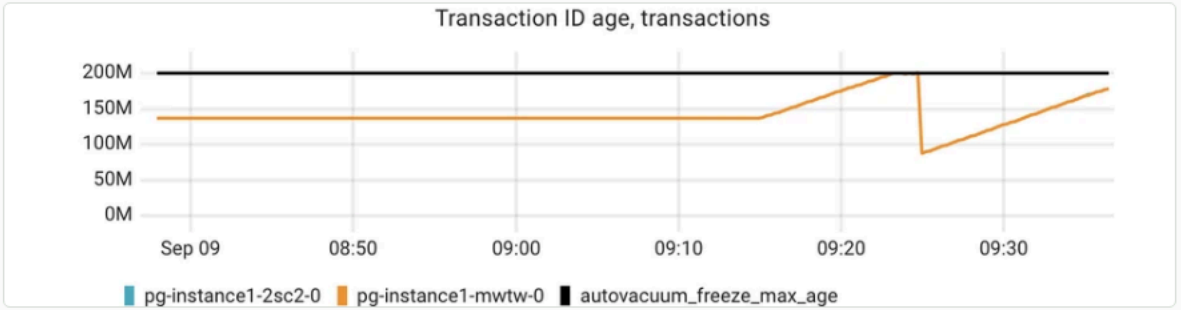
```
-- long transaction
BEGIN ISOLATION LEVEL REPEATABLE READ;
SELECT ...;          -- then it stalls
-- batch loader
BEGIN;
  SAVEPOINT r; INSERT ...; RELEASE r;
  ...                      -- 65 per commit
COMMIT;              -- 65 XIDs, not 1
```

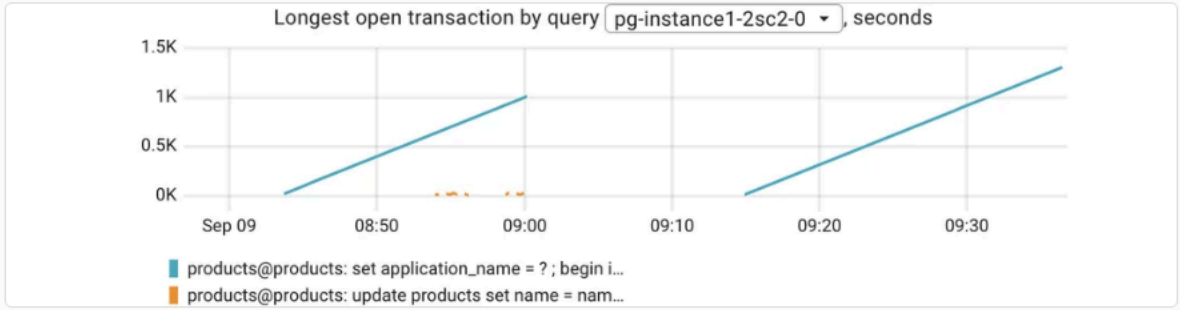


“vacuum cannot freeze past the oldest transaction ID”

Alert s7lv855e64vv Firing

Rule	Postgres transaction ID wraparound
Application	pg
Severity	Critical
Opened at	Sep 09, 09:37:57 (4s ago)
Duration	4s (ongoing)
Summary	transaction ID wraparound risk on 2 postgres instances
Description	PostgreSQL is approaching transaction ID wraparound. If the age reaches the limit the database will stop accepting writes. Vacuum freezing is likely blocked by a held xmin horizon (a replication slot, long-running or prepared transaction).
Findings	⚠ pg-instance1-2sc2-0: database "inventory" is 8% toward transaction ID wraparound (age 181M); vacuum cannot freeze past the oldest transaction ID, held back by a long-running transaction; end that transaction. The oldest is in "products", open 21m56s: set application_name = ? ; begin isolation level repeatable read ; select count (*) from products ; select pg_sleep (?)





The dip in the age is an earlier holder being ended: the horizon advances and the age falls back. Threshold lowered to 5% for this reproduction, the default is 50%.

Ask every database how old its oldest ID is

THE QUERY

```
SELECT datname,
       age(datfrozenxid),
       mxid_age(datminmxid)
FROM pg_database;
```

WHAT IT RETURNS

```
  datname  |    age    | mxid_age
-----------+-----------+----------
 products  | 412877877 |     1204
 postgres  |     18422 |        0
```

WHICH BECOMES

```
pg_xid_age{db="products"} 412877877
pg_multixact_age{db="products"} 1204
```

SCENARIO 3 OF 3

Replication

A standby is only useful if it is caught up. Two separate steps decide whether it is.



Replication happens in two steps

The primary sends its WAL to the replica. The replica receives it, then replays it. Either step can fall behind.

SHIPPING LAG

Sent, not received

The network between them is slow or broken.

APPLY LAG

Received, not replayed

The replica is busy, often behind a long read query.

Why lag matters

A replica usually does two jobs at once. Lag breaks both, and neither one returns an error.

IT SERVES READS

Answers get stale

Reports, dashboards and read-only traffic keep working, on data that is minutes or hours old.

IT IS YOUR FAILOVER TARGET

Promotion loses writes

Promote a replica that is hours behind, and everything the primary accepted since is gone.

Cut the network between them

HOW THIS HAPPENS

- A network policy, a firewall rule, or a fenced node
- The replica is still running and still healthy, just unreachable

The replica stops receiving WAL, and falls further behind every second.

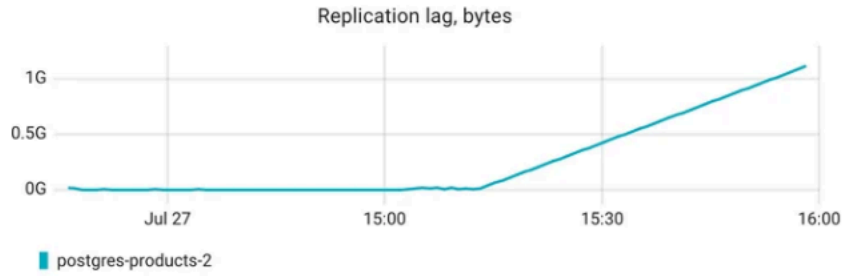
```yaml
# partition the replica off the primary
kind: NetworkChaos
spec:
  action: partition
  direction: both
  selector:
    labelSelector:
      role: replica
```



“the standby is not connected to the primary”

Alert `z6mzhwbtunu6` Firing

Rule	Postgres replication lag
Application	postgres-products
Severity	Warning
Opened at	Jul 27, 15:16:14 (44m ago)
Duration	44m (ongoing)
Summary	1 postgres replica is far behind the primary
Description	PostgreSQL replica is falling behind the primary. Read queries on replicas may return stale data.
Findings	⚠ postgres-products-2: the standby is not connected to the primary (postgres-products-1) - no network connectivity to the primary
Notifications	-





One LSN from the primary, two from the replica

THE QUERY

```
-- primary
SELECT pg_current_wal_lsn() - '0/0';
-- replica
SELECT pg_last_wal_receive_lsn() - '0/0',
       pg_last_wal_replay_lsn()  - '0/0';
SELECT count(1) FROM pg_stat_wal_receiver;
```

shipping lag = current − receive · apply lag = receive − replay

WHAT IT RETURNS

```
      source        |    value
--------------------+-------------
 primary  current   |  5905580032
 replica  receive   |  5368709120
 replica  replay    |  5368709120
 replica  receivers |           0
```

WHICH BECOMES

```
pg_current_wal_lsn 5905580032
pg_last_wal_receive_lsn 5368709120
pg_last_wal_replay_lsn 5368709120
pg_wal_receiver_status{sender_host="10.0.1.4", sender_port="5432"} 0
```

Why a replica falls behind

The alert fires on lag over 30 seconds, then says which of these it is.

THE REASON	WHAT SHOWS IT
They cannot reach each other	Connection attempts between the two are failing
The network is slow	Round-trip time between the two
The network is dropping packets	TCP retransmits between the two
No WAL receiver on the replica	`pg_wal_receiver_status` is 0
Replay is slow	Received is far ahead of replayed
Replay is paused	`pg_is_wal_replay_paused`

The first three come from the network, not from Postgres. We measure both, so the alert can say which.

The same shape, for everything else

The rest work the same way: a threshold the database itself defines, then the cause.

CHECK	FIRES WHEN
Checkpoints	Time since the last one > 3 × `checkpoint_timeout`
WAL archiving	The most recent archive attempt failed
Backups	No successful backup in 24 hours
Stale statistics	A large table past 2x its autoanalyze threshold
Bloat	Estimated wasted space over half the database
Connections	Connections over 90% of `max_connections`

Summary

MONITOR THE MACHINERY

The background work needs its own signals

Vacuum, freezing, replication, checkpoints. None of it shows up in query latency, until it takes the database down.

AUTOMATE THE INVESTIGATION

An alert is half the job

Naming the cause takes telemetry you collect on purpose: thresholds, settings, and who is holding things back.

PROVE THE COVERAGE

Break it on purpose

Chaos testing is how you find out a failure is actually detected, before it happens for real.

ALREADY BUILT

Coroot Community Edition, Apache 2.0

Every check in this talk ships out of the box, minutes after installation.



`github.com/coroot/coroot`

Questions?