PUIGFLOWS · GUÍA TÉCNICA

Autovacuum y Bloat en PostgreSQL: MVCC, Freezing, Wraparound y Tuning por Tabla

PostgreSQL Autovacuum and Bloat: MVCC, Freezing, Wraparound and Per-Table Tuning

Josué Puig

Guía · Nivel intermedio · Lectura estimada: 56 min

Edición del 19 de agosto de 2026 · Contenido bilingüe (Español / English)

VERSIÓN EN ESPAÑOL

Autovacuum y Bloat en PostgreSQL: MVCC, Freezing, Wraparound y Tuning por Tabla

Hay una clase de incidente en PostgreSQL que nunca aparece en los logs de la aplicación. No lanza excepciones, no dispara alertas de error y no rompe ningún test. La base de datos simplemente se va volviendo más lenta, semana tras semana, hasta que una consulta que tardaba 12 ms tarda 900 ms. Cuando por fin alguien mira el tamaño en disco, descubre una tabla de 40 GB que contiene 8 GB de datos vivos.

Eso es **bloat** , y la causa raíz casi siempre es la misma: autovacuum lleva semanas o meses sin poder hacer su trabajo, y nadie se enteró porque nada falló.

Esta guía cubre el ciclo completo: por qué PostgreSQL genera bloat por diseño, qué hace VACUUM exactamente (y qué no hace, que es donde vive la mayor parte de la confusión), cómo medirlo de verdad, cómo afinar autovacuum tabla por tabla en lugar de tocar postgresql.conf a ciegas, y cómo salir del peor escenario que existe en PostgreSQL: el wraparound de transaction IDs, el único fallo capaz de dejar tu base de datos rechazando escrituras.

1. MVCC: por qué el bloat es inevitable

PostgreSQL nunca modifica una fila en el sitio. Un UPDATE escribe una *versión nueva* de la fila y marca la anterior como caducada; un DELETE ni siquiera borra, solo marca. Esto es lo que permite que un SELECT largo siga viendo un estado coherente de los datos mientras otras transacciones escriben encima: cada versión de fila lleva un `xmin` (la transacción que la creó) y un `xmax` (la que la eliminó o la sustituyó), y cada snapshot decide qué versiones puede ver.

Se ve muy claro con el `ctid` , que es la dirección física de la fila (página, posición dentro de la página):

```
CREATE TABLE demo (id int PRIMARY KEY, saldo numeric);
INSERT INTO demo VALUES (1, 100);

SELECT ctid, xmin, xmax, * FROM demo;
--  ctid  | xmin | xmax | id | saldo
-- -------+------+------+----+-------
--  (0,1) |  742 |    0 |  1 |   100

UPDATE demo SET saldo = 200 WHERE id = 1;

SELECT ctid, xmin, xmax, * FROM demo;
--  ctid  | xmin | xmax | id | saldo
-- -------+------+------+----+-------
--  (0,2) |  743 |    0 |  1 |   200
```

La fila "se movió" de la posición 1 a la 2 de la página 0. La versión antigua sigue físicamente en el fichero, ocupando espacio, invisible para cualquier transacción nueva. Se llama *dead tuple* (tupla muerta), y hasta que alguien la limpie es peso muerto que tus índices y tus secuential scans tienen que atravesar.

> *Consecuencia práctica que sorprende a mucha gente: una tabla que solo recibe UPDATEs, sin una sola inserción neta, crece. Un contador que se actualiza mil veces por segundo genera mil versiones muertas por segundo.*

2. Qué hace VACUUM exactamente (y qué no)

VACUUM es el recolector de basura de ese diseño. En una pasada normal hace cuatro cosas:

- **Recupera espacio** de las tuplas muertas y lo registra en el *Free Space Map* para que futuras escrituras lo reutilicen.
- **Limpia las entradas de índice** que apuntaban a esas tuplas.
- **Actualiza el Visibility Map** , marcando páginas donde todo es visible para todos. Esto es lo que habilita los *index-only scans* , así que un vacuum al día vale más de lo que parece.
- **Congela** transaction IDs antiguos, que es el tema de la sección 8.

Y ahora lo que **no** hace, que es la fuente número uno de malentendidos:

- **No devuelve el espacio al sistema operativo.** El fichero de 40 GB sigue midiendo 40 GB. VACUUM lo convierte en 40 GB con 32 GB reutilizables. La única excepción es la truncación de páginas completamente vacías *al final* de la tabla, que además necesita un lock ACCESS EXCLUSIVE breve.
- **No reordena físicamente** las filas ni compacta las páginas parcialmente llenas.
- **No recalcula estadísticas.** Eso es ANALYZE, un proceso distinto con sus propios umbrales.

Si tu tabla ya está hinchada, VACUUM detiene la hemorragia pero no recupera el disco. Para eso hace falta reescribir la tabla (sección 11).

3. Cuándo se dispara autovacuum: la fórmula que causa el 80% de los problemas

El launcher de autovacuum se despierta cada `autovacuum_naptime` (1 minuto por defecto) y compara, para cada tabla, las tuplas muertas estimadas contra este umbral:

```
umbral = autovacuum_vacuum_threshold          -- 50 por defecto
       + autovacuum_vacuum_scale_factor       -- 0.2 por defecto
       * reltuples                            -- filas estimadas de la tabla
```

Ese `0.2` es el problema. Significa "espera a que el 20% de la tabla esté muerto". En una tabla de 10.000 filas son 2.050 tuplas muertas y es razonable. En una tabla de 50 millones de filas son **10 millones de tuplas muertas** antes de que autovacuum se moleste en mirarla. Para entonces las consultas ya se han degradado y la limpieza va a ser una pasada larga y cara justo en el peor momento.

Hay dos umbrales más que conviene tener en la cabeza:

- **ANALYZE** usa `autovacuum_analyze_threshold` (50) + `autovacuum_analyze_scale_factor` (0.1). Estadísticas obsoletas producen planes malos, que es un problema distinto del bloat pero llega por la misma puerta.
- **Tablas de solo inserción** (logs, eventos, series temporales) no generan tuplas muertas, así que durante años nunca recibían vacuum y acumulaban una deuda enorme de congelación. Desde PostgreSQL 13 existen `autovacuum_vacuum_insert_threshold` (1000) y `autovacuum_vacuum_insert_scale_factor` (0.2) para eso.

PostgreSQL 18 añade por fin un tope absoluto, `autovacuum_vacuum_max_threshold` , con valor por defecto 100.000.000: por muy grande que sea la tabla, al superar ese número de tuplas muertas

autovacuum entra igualmente. Es una red de seguridad útil, pero no sustituye al ajuste por tabla: 100 millones de tuplas muertas siguen siendo demasiadas para casi cualquier tabla real.

Para ver qué umbral tiene hoy cada tabla, con los overrides ya aplicados:

```
SELECT c.relname,
       s.n_live_tup,
       s.n_dead_tup,
       (coalesce((SELECT option_value::float
                    FROM pg_options_to_table(c.reloptions)
                   WHERE option_name = 'autovacuum_vacuum_threshold'),
                 current_setting('autovacuum_vacuum_threshold')::float)
        + coalesce((SELECT option_value::float
                      FROM pg_options_to_table(c.reloptions)
                     WHERE option_name = 'autovacuum_vacuum_scale_factor'),
                   current_setting('autovacuum_vacuum_scale_factor')::float)
          * c.reltuples)::bigint AS umbral_actual
FROM pg_class c
JOIN pg_stat_user_tables s ON s.relid = c.oid
WHERE c.relkind = 'r'
ORDER BY s.n_dead_tup DESC
LIMIT 20;
```

Si en alguna fila `n_dead_tup` está cerca de `umbral_actual` y ese umbral son millones, ya sabes dónde empezar.

4. Detectar bloat: tres niveles de precisión

Nivel 1: el chequeo barato de todos los días

```
SELECT relname,
       n_live_tup,
       n_dead_tup,
       round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct_muertas,
       pg_size_pretty(pg_total_relation_size(relid)) AS tamano,
       last_autovacuum,
       autovacuum_count
FROM pg_stat_user_tables
WHERE n_dead_tup > 10000
ORDER BY n_dead_tup DESC
LIMIT 20;
```

Cuesta milisegundos y sirve para el dashboard. Dos avisos importantes sobre cómo interpretarlo:

- `n_dead_tup` es una *estimación* del colector de estadísticas, no un recuento. Un crash o un `pg_stat_reset()` la pone a cero sin que el bloat haya desaparecido.
- Un `n_dead_tup` bajo **no** significa que no haya bloat. Si autovacuum ya limpió las tuplas pero el espacio quedó disperso en páginas medio vacías, el contador está a cero y el fichero sigue igual de gordo. Esto engaña a mucha gente.

La columna que más información da suele ser `last_autovacuum` . Si es NULL o de hace tres semanas en una tabla que recibe escrituras constantes, tienes un vacuum bloqueado, no un vacuum lento.

Nivel 2: medición exacta con pgstattuple

Cuando el nivel 1 levanta una sospecha, mide de verdad:

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;

-- Exacto, pero lee la tabla entera: úsalo con criterio en producción
SELECT table_len, tuple_percent, dead_tuple_percent, free_percent
FROM pgstattuple('public.pedidos');

-- Muestreo: mucho más barato, suficiente para decidir
SELECT approx_tuple_percent, approx_free_percent, dead_tuple_percent
FROM pgstattuple_approx('public.pedidos');

-- Los índices también se hinchan, y a menudo más que la tabla
SELECT avg_leaf_density, leaf_fragmentation
FROM pgstatindex('public.pedidos_pkey');
```

Cómo leerlo: en una tabla sana `free_percent` suele quedarse por debajo del 20%. Por encima del 40% ya estás pagando I/O por leer aire. En índices B-tree, un `avg_leaf_density` por debajo de 60-70% indica que un REINDEX CONCURRENTLY te va a devolver espacio y latencia con muy poco riesgo.

El detalle que justifica esta sección: es perfectamente normal que el nivel 1 diga "0% de tuplas muertas" y pgstattuple reporte 37% de espacio libre disperso. Son dos cosas distintas.

Nivel 3: ver el vacuum mientras corre

```
SELECT p.pid,
       p.relid::regclass AS tabla,
       p.phase,
       pg_size_pretty(p.heap_blks_total * 8192::bigint)    AS total,
       pg_size_pretty(p.heap_blks_scanned * 8192::bigint)  AS escaneado,
       p.index_vacuum_count,
       a.query_start,
       now() - a.query_start AS duracion
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a USING (pid);
```

Si ves `index_vacuum_count` subiendo por encima de 1, el vacuum está haciendo varias pasadas sobre todos los índices porque no le cabe la lista de tuplas muertas en memoria. Eso multiplica el coste y se arregla con memoria (sección 6).

5. Afinar autovacuum por tabla, no globalmente

El error más común al descubrir el problema es bajar `autovacuum_vacuum_scale_factor` a 0.01 en postgresql.conf. Funciona para las tres tablas grandes y provoca vacuums constantes e inútiles en las doscientas tablas pequeñas del esquema.

La configuración correcta es por relación. Perfil para una tabla con escritura intensa:

```
ALTER TABLE eventos SET (
  autovacuum_vacuum_scale_factor  = 0.01,   -- 1% en vez de 20%
  autovacuum_vacuum_threshold     = 1000,
  autovacuum_analyze_scale_factor = 0.005,  -- estadísticas frescas
```

```
  autovacuum_analyze_threshold    = 500,
  autovacuum_vacuum_cost_delay    = 0       -- sin freno: que termine rápido
);

-- Tabla de solo inserción (logs, métricas, eventos)
ALTER TABLE logs_acceso SET (
  autovacuum_vacuum_insert_scale_factor = 0.01,
  autovacuum_vacuum_insert_threshold    = 10000
);

-- Comprobar lo aplicado
SELECT relname, reloptions FROM pg_class WHERE relname IN ('eventos','logs_acceso');

-- Volver al comportamiento global
ALTER TABLE eventos RESET (autovacuum_vacuum_scale_factor);
```

Un punto de partida razonable: cualquier tabla por encima de un millón de filas con escritura frecuente merece `scale_factor` entre 0.01 y 0.05. Las tablas pequeñas se quedan con los valores globales, que para ellas son correctos.

El `autovacuum_vacuum_cost_delay = 0` merece un matiz: quita el freno para esa tabla concreta. En una instancia con I/O holgado es exactamente lo que quieres para tus tablas calientes. En una instancia ya saturada de I/O, ponerlo a cero en una tabla de 500 GB puede empeorar la latencia del resto. Mídelo antes de generalizarlo.

6. Por qué tu vacuum va lento: el presupuesto de coste

Autovacuum tiene un limitador de velocidad deliberado, para no arrasar el I/O de la instancia. Funciona con un sistema de puntos:

- `vacuum_cost_page_hit` = 1 punto (página ya en shared_buffers)
- `vacuum_cost_page_miss` = 2 puntos (hay que leerla de disco)
- `vacuum_cost_page_dirty` = 20 puntos (hay que ensuciarla)

Cuando el trabajador acumula `vacuum_cost_limit` puntos (200 por defecto), duerme `autovacuum_vacuum_cost_delay` milisegundos (2 ms por defecto). Haz la cuenta: 200 puntos cada 2 ms son 100.000 puntos por segundo. Si el vacuum está ensuciando páginas a 20 puntos cada una, eso son 5.000 páginas por segundo, es decir **unos 39 MB/s** .

Cualquier NVMe moderno hace veinte veces eso. Estás limitando tu mantenimiento a la velocidad de un disco duro de 2010.

Y aquí está la trampa que arruina muchos intentos de arreglo: **el presupuesto es compartido entre todos los trabajadores de autovacuum** . Subir `autovacuum_max_workers` de 3 a 10 sin tocar el límite de coste no acelera nada; reparte los mismos 39 MB/s entre diez procesos y cada uno va más lento.

```
# postgresql.conf - punto de partida para instancias con SSD/NVMe
autovacuum_vacuum_cost_limit = 2000    # ~390 MB/s de techo, 10x el defecto
autovacuum_vacuum_cost_delay = 2ms
autovacuum_max_workers = 5             # súbelo DESPUÉS de subir el límite
autovacuum_naptime = 30s
maintenance_work_mem = 1GB             # memoria por operación de vacuum manual
autovacuum_work_mem = 1GB              # idem para trabajadores de autovacuum
vacuum_buffer_usage_limit = 16MB       # anillo de buffers (defecto 2MB desde PG16)
```

Memoria: la razón oculta de las pasadas múltiples

VACUUM guarda en memoria los identificadores de las tuplas muertas que va encontrando. Si esa estructura se llena, tiene que parar, recorrer *todos* los índices de la tabla, vaciar la lista y volver a empezar. Con seis índices y tres pasadas son dieciocho recorridos completos de índice.

Hasta PostgreSQL 16 esa estructura era un array ordenado con un tope duro de 1 GB, así que subir `maintenance_work_mem` por encima de 1 GB no servía de nada. PostgreSQL 17 lo reemplazó por un *TidStore* basado en un árbol radix adaptativo: mucho más compacto, sin el tope de 1 GB, y con eso las pasadas múltiples pasaron de ser habituales a ser raras. Si estás en PG16 o anterior y ves `index_vacuum_count > 1` , esa sola actualización te arregla el problema.

Para vacuums manuales puntuales puedes saltarte los límites globales:

```
VACUUM (VERBOSE, PARALLEL 4, BUFFER_USAGE_LIMIT '256MB') pedidos;
```

`PARALLEL` paraleliza la fase de limpieza de índices (no el escaneo de la tabla), así que solo ayuda si la tabla tiene varios índices.

7. HOT updates y fillfactor: prevenir en vez de limpiar

Hay un atajo dentro del motor que evita generar basura en primer lugar. Si un UPDATE cumple dos condiciones, PostgreSQL hace un *HOT update* (Heap-Only Tuple):

1. No modifica ninguna columna indexada.
2. La nueva versión cabe en la misma página que la antigua.

En ese caso no se crea ninguna entrada nueva en los índices, y la versión vieja puede reciclarse mediante el *page pruning* oportunista que hace cualquier lectura de esa página, sin esperar a VACUUM. Es la diferencia entre una tabla que se mantiene sola y una que necesita mantenimiento constante.

Mide qué proporción de tus updates consigue este camino rápido:

```
SELECT relname,
       n_tup_upd,
       n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd / NULLIF(n_tup_upd, 0), 1) AS pct_hot
FROM pg_stat_user_tables
WHERE n_tup_upd > 100000
ORDER BY n_tup_upd DESC
LIMIT 20;
```

Por debajo del 50% en una tabla con mucha escritura, hay margen real de mejora. Dos palancas:

Reducir el fillfactor para dejar hueco libre en cada página y que la nueva versión quepa al lado de la vieja:

```
ALTER TABLE sesiones SET (fillfactor = 80);   -- 20% de la página reservado
-- Ojo: solo afecta a páginas nuevas. Para aplicarlo a los datos
-- existentes hay que reescribir la tabla (VACUUM FULL o pg_repack).
```

Entre 70 y 85 es el rango habitual para tablas con updates frecuentes. Bajarlo más significa leer más páginas para el mismo número de filas, así que no es gratis: es un intercambio entre densidad de lectura y

coste de escritura.

Quitar índices sobre columnas que se actualizan mucho. Un índice sobre `updated_at` en una tabla que se actualiza constantemente destruye la posibilidad de HOT updates para cada UPDATE. Antes de mantenerlo, comprueba si alguien lo usa:

```
SELECT indexrelid::regclass AS indice,
       relid::regclass       AS tabla,
       idx_scan,
       pg_size_pretty(pg_relation_size(indexrelid)) AS tamano
FROM pg_stat_user_indexes
WHERE idx_scan < 50
ORDER BY pg_relation_size(indexrelid) DESC;
```

Un índice con cero escaneos desde el último reinicio estadístico cuesta espacio, cuesta escritura y bloquea HOT updates a cambio de nada. (Verifica primero que no sea un índice único o de soporte de una clave foránea, y que las estadísticas cubran un periodo representativo: no borres un índice que solo se usa en el cierre mensual.)

8. Freezing y wraparound: el fallo que apaga la base de datos

Los transaction IDs de PostgreSQL son enteros de 32 bits. Las comparaciones son circulares: para cualquier XID, la mitad del espacio se considera "pasado" y la otra mitad "futuro". Eso da unos 2.000 millones de transacciones de margen. Cuando el contador da la vuelta, transacciones antiguas empezarían a parecer del futuro y sus filas desaparecerían de golpe. PostgreSQL prefiere pararse antes que permitir eso.

La defensa es el *freezing* : marcar filas suficientemente antiguas como "visibles siempre, para todos", sacándolas de la aritmética de XIDs. VACUUM lo hace de forma incremental. Los parámetros que gobiernan el proceso:

- `vacuum_freeze_min_age` (50M): edad mínima de un XID para que un vacuum lo congele.
- `vacuum_freeze_table_age` (150M): a partir de aquí, el siguiente vacuum es *agresivo* y escanea toda la tabla en lugar de saltarse las páginas ya visibles.
- `autovacuum_freeze_max_age` (200M): se lanza un autovacuum *anti-wraparound* aunque autovacuum esté desactivado para esa tabla.
- `vacuum_failsafe_age` (1.600M): modo de emergencia. El vacuum ignora el freno de coste y se salta la limpieza de índices para ir lo más rápido posible.
- Equivalentes para MultiXact: `autovacuum_multixact_freeze_max_age` (400M) y `vacuum_multixact_failsafe_age` (1.600M). Se disparan con uso intensivo de `SELECT ... FOR SHARE` y claves foráneas.

Esta es la consulta que debería estar en tu dashboard desde hoy:

```
-- Edad de congelación por relación
SELECT c.oid::regclass AS relacion,
       age(c.relfrozenxid) AS edad_xid,
       round(100.0 * age(c.relfrozenxid)
             / current_setting('autovacuum_freeze_max_age')::numeric, 1) AS pct_umbral,
       pg_size_pretty(pg_total_relation_size(c.oid)) AS tamano
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
```

```
WHERE c.relkind IN ('r', 'm', 't')
  AND n.nspname NOT IN ('pg_catalog', 'information_schema')
ORDER BY age(c.relfrozenxid) DESC
LIMIT 20;

-- Vista de conjunto por base de datos
SELECT datname,
       age(datfrozenxid) AS edad_xid,
       2147483647 - age(datfrozenxid) AS xids_restantes
FROM pg_database
ORDER BY age(datfrozenxid) DESC;
```

Umbrales de alerta que funcionan en la práctica: aviso a los 500 millones, alerta seria a los 1.000 millones, llamada de madrugada a los 1.500 millones. A los 2.000 millones menos unos pocos, PostgreSQL empieza a rechazar comandos con el mensaje "database is not accepting commands to avoid wraparound data loss".

Runbook: qué hacer cuando ya está pasando

El instinto es entrar en modo mono-usuario. **Casi nunca es lo correcto** , y es peligroso: el modo mono-usuario desactiva precisamente la protección que evita la corrupción, y cada transacción que ejecutas ahí dentro te acerca al abismo. Las versiones modernas de PostgreSQL reservan un margen de unos 3 millones de XIDs justo para que un administrador pueda resolverlo en caliente. Sigue este orden:

```
-- 1. ¿Qué está reteniendo el horizonte? Transacciones vivas
SELECT pid, state, age(backend_xmin) AS edad_xmin,
       now() - xact_start AS duracion, left(query, 80) AS consulta
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL
ORDER BY age(backend_xmin) DESC;

-- 2. Slots de replicación abandonados (el sospechoso habitual)
SELECT slot_name, active, xmin, catalog_xmin,
       age(xmin) AS edad_xmin,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS wal_retenido
FROM pg_replication_slots
ORDER BY age(xmin) DESC NULLS LAST;

-- 3. Transacciones preparadas huérfanas (2PC olvidado)
SELECT gid, prepared, owner, database FROM pg_prepared_xacts ORDER BY prepared;

-- 4. Elimina el bloqueador
SELECT pg_terminate_backend(12345);            -- transacción zombi
SELECT pg_drop_replication_slot('replica_muerta');
ROLLBACK PREPARED 'gid_olvidado';

-- 5. Ahora sí, congela las tablas más antiguas, sin freno
SET vacuum_cost_delay = 0;
VACUUM (FREEZE, VERBOSE) tabla_mas_antigua;
```

El paso 4 es el que resuelve el incidente. Los pasos 1 a 3 son el que te dice cuál de las cuatro causas tienes. Saltarte el diagnóstico y lanzar VACUUM FREEZE directamente no sirve de nada si el horizonte sigue bloqueado: el vacuum correrá durante horas y no eliminará ni una tupla.

9. Los cuatro bloqueadores: cuando VACUUM corre y no limpia nada

Este es el concepto que separa a quien ha peleado con esto de quien solo ha leído la documentación. **VACUUM solo puede eliminar una tupla muerta si ninguna transacción del sistema podría necesitar verla todavía.** Ese punto de corte se llama el horizonte de xmin, y basta una sola cosa vieja para congelarlo entero:

1. **Transacciones de larga duración** , incluidas las que están `idle in transaction` . Un BEGIN sin COMMIT en un pool de conexiones puede paralizar la limpieza de toda la instancia durante días.
2. **Slots de replicación inactivos.** Una réplica que se retiró sin borrar su slot retiene el horizonte y el WAL indefinidamente. Es la causa número uno de bloat inexplicable.
3. **hot_standby_feedback = on** en una réplica que ejecuta consultas analíticas largas: el primario retrasa su limpieza para que la réplica no cancele consultas.
4. **Transacciones preparadas huérfanas** , típicamente de un coordinador XA que murió a mitad.

Defensas que conviene tener puestas desde el principio:

```
# postgresql.conf
idle_in_transaction_session_timeout = 5min   # mata el BEGIN olvidado
transaction_timeout = 30min                  # nuevo en PG17: tope total por transacción
statement_timeout = 60s                      # ajústalo por rol, no globalmente

# Limita el WAL que un slot puede retener antes de invalidarse
max_slot_wal_keep_size = 100GB
# PG17+: invalida slots inactivos automáticamente
idle_replication_slot_timeout = 24h
```

Un detalle muy útil: desde PostgreSQL 16, la salida de `VACUUM VERBOSE` incluye el corte de removibilidad y cuántos segundos de antigüedad tiene. Si ves algo como "removable cutoff ... older than 86400 seconds", ya sabes que tienes un horizonte bloqueado y no un problema de tuning.

10. Qué ha cambiado en PostgreSQL 17 y 18

Si estás en PG13 o PG14, estas dos versiones cambian bastante el panorama del mantenimiento:

PostgreSQL 17

- **TidStore** : la estructura de tuplas muertas pasa de array ordenado a árbol radix adaptativo. Mucha más densidad por byte y adiós al tope de 1 GB. En tablas grandes con varios índices, la mejora de tiempo de vacuum es del orden de 2x a 6x según el caso.
- Privilegio `MAINTAIN` : puedes delegar VACUUM, ANALYZE, REINDEX y CLUSTER sin repartir superusuario.
- `transaction_timeout` : el guardarraíl que faltaba contra las transacciones eternas.

PostgreSQL 18

- `autovacuum_vacuum_max_threshold` (defecto 100.000.000): tope absoluto de tuplas muertas, para que las tablas enormes no queden a merced del scale factor.
- **Congelación anticipada (eager freezing)** : un VACUUM normal ahora intenta congelar también páginas que ya son "all-visible" pero no "all-frozen", en vez de dejar toda esa deuda para el

anti-wraparound. Se controla con `vacuum_max_eager_freeze_failure_rate` (defecto 0.03): es la fracción de páginas que el vacuum puede intentar congelar sin éxito antes de desactivar el escaneo anticipado en esa pasada. Los éxitos se limitan internamente al 20% de las páginas candidatas, para repartir el coste entre varias ejecuciones. El efecto neto es que las pasadas anti-wraparound, esas que duran seis horas y aparecen sin avisar, se vuelven mucho más cortas.

- `autovacuum_max_workers` pasa a poder modificarse con un simple reload; el techo duro lo fija `autovacuum_worker_slots` , que sí requiere reinicio. Se acabó reiniciar la instancia para subir trabajadores durante un incidente.

> *Regla práctica: si tu mayor dolor son las pasadas anti-wraparound largas en tablas de cientos de GB, PG18 es una actualización con retorno inmediato. Si tu dolor son los vacuums lentos por falta de memoria, es PG17 lo que necesitas.*

11. El bloat ya está ahí: cómo recuperar el espacio

Ajustar autovacuum evita el bloat futuro. Para el que ya tienes hay que reescribir la relación. Cuatro opciones, de la más brusca a la más elegante:

VACUUM FULL — reescribe la tabla completa y devuelve el espacio al sistema operativo. Toma un lock ACCESS EXCLUSIVE durante toda la operación: nadie lee ni escribe, ni siquiera un SELECT. Necesita espacio libre en disco equivalente a la tabla resultante más sus índices. Es la opción correcta solo si tienes ventana de mantenimiento o la tabla es pequeña.

```
VACUUM (FULL, ANALYZE, VERBOSE) pedidos;   -- bloquea la tabla por completo
```

CLUSTER — lo mismo, pero reordenando físicamente las filas según un índice. Mismo lock, misma restricción. Útil si además tienes un patrón de acceso por rangos que se beneficia de la localidad.

pg_repack — extensión externa que hace el trabajo en línea. Crea una copia sombra, captura los cambios concurrentes con triggers y al final intercambia las relaciones. Solo mantiene el lock exclusivo unos instantes al principio y al final. A cambio: hay que instalar la extensión y su cliente, necesita el doble de espacio, y en tablas muy escritas la cola de triggers puede crecer bastante.

```
# Reempaquetar una tabla concreta, sin cortar el servicio
pg_repack --no-superuser-check -d midb -t public.pedidos --jobs 4

# Solo los índices (más barato y muchas veces suficiente)
pg_repack -d midb -t public.pedidos --only-indexes
```

pg_squeeze — alternativa más reciente que usa decodificación lógica en lugar de triggers para seguir los cambios concurrentes. Menos sobrecarga en el primario y sin locks largos sobre el catálogo, a cambio de requerir `wal_level = logical` . Puede programarse para actuar sola cuando una tabla supera un umbral de bloat.

Y una nota sobre hacia dónde va esto: en PGConf.dev 2026 se presentó un comando nativo `REPACK CONCURRENTLY` , propuesto para PostgreSQL 19, que llevaría el reempaquetado en línea al core sin extensiones. Todavía no es algo con lo que puedas contar en producción, pero conviene tenerlo en el radar antes de construir tooling propio a su alrededor.

Los índices primero. Antes de plantearte reescribir una tabla de 400 GB, mira los índices. `REINDEX INDEX CONCURRENTLY` existe desde PostgreSQL 12, no bloquea escrituras y con frecuencia recupera la mayor

parte del espacio perdido:

```
REINDEX INDEX CONCURRENTLY pedidos_cliente_id_idx;
REINDEX TABLE CONCURRENTLY pedidos;   -- todos los índices de la tabla

-- Si algo falla a mitad quedan índices invalidos: hay que limpiarlos
SELECT indexrelid::regclass FROM pg_index WHERE NOT indisvalid;
```

12. Monitorización: script y alertas

Este script reúne las tres señales que importan (bloat, edad de congelación y bloqueadores del horizonte) en un solo informe ejecutable desde cron o desde un job de CI:

```
"""Informe de salud de vacuum para PostgreSQL. Requiere: pip install psycopg[binary]"""
import psycopg

BLOAT = """
SELECT relname,
       n_dead_tup,
       round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct,
       pg_size_pretty(pg_total_relation_size(relid)) AS size,
       last_autovacuum
FROM pg_stat_user_tables
WHERE n_dead_tup > %s
ORDER BY n_dead_tup DESC LIMIT 10;
"""

FREEZE = """
SELECT c.oid::regclass::text,
       age(c.relfrozenxid) AS xid_age
FROM pg_class c JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','m','t')
  AND n.nspname NOT IN ('pg_catalog','information_schema')
  AND age(c.relfrozenxid) > %s
ORDER BY 2 DESC LIMIT 10;
"""

HORIZON = """
SELECT 'backend'  AS kind, pid::text AS id, age(backend_xmin) AS xid_age
  FROM pg_stat_activity WHERE backend_xmin IS NOT NULL
UNION ALL
SELECT 'slot', slot_name, age(xmin)
  FROM pg_replication_slots WHERE xmin IS NOT NULL
UNION ALL
SELECT 'prepared', gid, age(transaction)
  FROM pg_prepared_xacts
ORDER BY xid_age DESC NULLS LAST LIMIT 5;
"""

def report(dsn: str, dead_min: int = 100_000, freeze_warn: int = 500_000_000) -> int:
    problems = 0
    with psycopg.connect(dsn, autocommit=True) as conn:
        print("== Tablas con más tuplas muertas ==")
        for name, dead, pct, size, last_av in conn.execute(BLOAT, (dead_min,)):
            flag = "  <-- REVISAR" if last_av is None else ""
            print(f"{name:35} {dead:>12,} ({pct}%)  {size:>10}  ultimo={last_av}{flag}")
            problems += 1

        print("\n== Relaciones cerca del anti-wraparound ==")
```

```
        for name, age in conn.execute(FREEZE, (freeze_warn,)):
            print(f"{name:35} edad_xid={age:>12,}")
            problems += 1

        print("\n== Qué retiene el horizonte de limpieza ==")
        for kind, ident, age in conn.execute(HORIZON):
            print(f"{kind:10} {ident:25} edad_xid={age}")
    return problems

if __name__ == "__main__":
    import os, sys
    sys.exit(1 if report(os.environ["DATABASE_URL"]) else 0)
```

La sección del horizonte es la que aporta el valor real: convierte "el vacuum va lento" en "el slot de replicación llamado replica_analitica lleva 900 millones de XIDs reteniendo la limpieza", que es una frase sobre la que se puede actuar.

Y las alertas equivalentes con el postgres_exporter de Prometheus:

```
groups:
  - name: postgres-vacuum
    rules:
      - alert: PostgresXIDWraparoundWarning
        expr: pg_database_xid_age > 500000000
        for: 30m
        labels: { severity: warning }
        annotations:
          summary: "Edad de XID alta en {{ $labels.datname }}"
          description: "age(datfrozenxid) = {{ $value }}. Revisa horizonte y anti-wraparound."

      - alert: PostgresXIDWraparoundCritical
        expr: pg_database_xid_age > 1200000000
        for: 5m
        labels: { severity: critical }
        annotations:
          summary: "Riesgo de wraparound en {{ $labels.datname }}"

      - alert: PostgresAutovacuumStalled
        expr: |
          (time() - pg_stat_user_tables_last_autovacuum > 86400)
          and (pg_stat_user_tables_n_dead_tup > 1000000)
        for: 1h
        labels: { severity: warning }
        annotations:
          summary: "Sin autovacuum en 24h con más de 1M de tuplas muertas"

      - alert: PostgresReplicationSlotHoldingXmin
        expr: pg_replication_slot_xmin_age > 200000000
        for: 15m
        labels: { severity: warning }
        annotations:
          summary: "El slot {{ $labels.slot_name }} está bloqueando la limpieza"
```

13. TOAST: la tabla que se hincha donde nadie mira

Cuando una fila supera aproximadamente 2 kB, PostgreSQL no la guarda entera en la tabla principal. Comprime los campos grandes y, si aún así no caben, los parte en trozos y los mueve a una tabla auxiliar llamada TOAST (*The Oversized-Attribute Storage Technique*). Cualquier tabla con columnas `text` , `jsonb`

, `bytea` o arrays tiene potencialmente su propia tabla TOAST, con su propio heap, su propio índice y su propio bloat.

El problema práctico es de visibilidad: `pg_stat_user_tables` no suma las tuplas muertas del TOAST a las de su tabla padre. Puedes tener una tabla con `n_dead_tup = 400` y sesenta gigas de basura en su TOAST sin que ninguna de tus consultas de diagnóstico habituales diga una palabra.

```
-- TOAST por tamaño, con sus propias estadísticas de vacuum
SELECT c.relname                                AS tabla,
       t.relname                                AS toast,
       pg_size_pretty(pg_relation_size(t.oid))  AS tam_toast,
       s.n_live_tup,
       s.n_dead_tup,
       s.last_autovacuum
FROM pg_class c
JOIN pg_class t                ON t.oid = c.reltoastrelid
LEFT JOIN pg_stat_all_tables s ON s.relid = t.oid
WHERE c.relkind = 'r'
  AND c.reltoastrelid <> 0
ORDER BY pg_relation_size(t.oid) DESC
LIMIT 20;
```

El segundo detalle que sorprende a mucha gente: **los parámetros de almacenamiento de la tabla padre no se heredan en su TOAST** . Si afinaste `documentos` con un scale factor de 0.02, su TOAST sigue funcionando con el 0.2 global. Los ajustes del TOAST se escriben con el prefijo `toast.` y hay que ponerlos de forma explícita.

```
ALTER TABLE documentos SET (
  autovacuum_vacuum_scale_factor       = 0.02,
  autovacuum_vacuum_threshold          = 2000,
  toast.autovacuum_vacuum_scale_factor = 0.02,
  toast.autovacuum_vacuum_threshold    = 2000,
  toast.autovacuum_vacuum_cost_delay   = 2
);
```

Para medir el bloat real de un TOAST concreto, `pgstattuple` funciona igual que sobre cualquier otra relación, apuntando al esquema `pg_toast` :

```
SELECT * FROM pgstattuple('pg_toast.pg_toast_16584');
```

Y ahora la parte de diseño, que suele ser la solución de verdad. Un `UPDATE` solo genera basura en el TOAST si toca una columna toasteada. Si tu tabla guarda un `jsonb` de 200 kB y actualizas cincuenta veces al día un contador que vive dentro de ese mismo JSON, cada actualización reescribe la cadena TOAST entera: doscientos kilobytes de escritura para cambiar un número. Sacar los campos pequeños y calientes a columnas propias —un `estado` , un `contador` , un `actualizado_en` — elimina el problema en origen mucho mejor que cualquier ajuste de autovacuum.

14. Réplicas: hot_standby_feedback, conflictos de recuperación y slots que no mueren

La lista de bloqueadores del horizonte se queda corta en un punto importante: en la mayoría de las instalaciones que tienen réplicas, el bloat del primario lo decide una máquina distinta.

El mecanismo es directo. Con `hot_standby_feedback = on` , la réplica informa al primario del xmin más antiguo que necesitan sus consultas, y el primario retrasa su limpieza para respetarlo. Con esa opción apagada, el primario limpia a su ritmo y la réplica cancela cualquier consulta que necesitase las filas que acaban de desaparecer, con el conocido `canceling statement due to conflict with recovery` . No hay una opción buena: es un intercambio entre bloat en el primario y consultas canceladas en la réplica, y conviene elegirlo a propósito en lugar de heredarlo.

Lo primero es medir cuánto te está costando ahora mismo:

```
-- Cuánto retiene cada réplica el horizonte del primario
SELECT pid, application_name, state,
       backend_xmin,
       age(backend_xmin) AS edad_xmin
FROM pg_stat_replication
ORDER BY age(backend_xmin) DESC NULLS LAST;

-- Conflictos que ya han ocurrido (ejecutar en la réplica)
SELECT datname, confl_snapshot, confl_lock, confl_bufferpin, confl_deadlock
FROM pg_stat_database_conflicts
WHERE confl_snapshot + confl_lock + confl_bufferpin + confl_deadlock > 0;
```

Si `confl_snapshot` crece, tus lecturas mueren por la limpieza del primario y el reflejo inmediato es encender `hot_standby_feedback` . Antes de hacerlo, mira `max_standby_streaming_delay` : es el dial intermedio que casi nadie usa. Subirlo a treinta segundos permite que la réplica pause la aplicación del WAL —y por tanto se retrase un poco— en lugar de cancelar consultas, sin tocar el horizonte del primario en absoluto.

Los slots de replicación son el caso más peligroso, porque no necesitan que exista nada al otro lado. Un slot huérfano de una réplica desmantelada hace meses retiene el horizonte y el WAL de forma indefinida:

```
SELECT slot_name, slot_type, active,
       age(xmin)          AS edad_xmin,
       age(catalog_xmin)  AS edad_catalog_xmin,
       inactive_since,
       pg_size_pretty(
         pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)
       ) AS wal_retenido
FROM pg_replication_slots
ORDER BY age(xmin) DESC NULLS LAST;
```

Aquí hay una trampa que se repite mucho: `max_slot_wal_keep_size` protege tu disco de WAL, pero **no protege tu horizonte de limpieza** . Un slot puede llevar días reteniendo el xmin y ser invalidado solo cuando el WAL acumulado cruza el límite. Son dos recursos distintos con dos umbrales distintos, y solo uno de los dos tiene freno automático.

PostgreSQL 18 añade por fin la pieza que faltaba: `idle_replication_slot_timeout` invalida los slots que llevan demasiado tiempo sin conexión, midiendo desde el valor de `inactive_since` . Viene desactivado por defecto, con valor 0.

```
-- PostgreSQL 18+: invalidar slots abandonados
ALTER SYSTEM SET idle_replication_slot_timeout = '24h';
SELECT pg_reload_conf();
```

Dos matices antes de confiarte. La invalidación se evalúa en el checkpoint, así que puede llegar hasta un `checkpoint_timeout` más tarde de lo que esperas, y no se aplica a los slots sincronizados hacia una

standby. Si estás en 17 o anterior, la alternativa es una alerta sobre `inactive_since` y una regla escrita: quien desmantela una réplica, borra su slot.

La configuración que mejor funciona en la práctica separa los papeles en lugar de buscar un ajuste único. La réplica de alta disponibilidad va con `hot_standby_feedback = off` y un `max_standby_streaming_delay` corto, porque su trabajo es estar lista para un failover, no servir informes. La réplica analítica, donde viven las consultas de veinte minutos, va con el feedback encendido y con una alerta sobre la edad de su `backend_xmin` , de modo que el coste en bloat sea una decisión medida y no un descubrimiento de tres meses después.

15. Bloat de índices: bottom-up deletion, la fase cara del vacuum y cuándo hace falta REINDEX

Es habitual encontrar índices más hinchados que la tabla que indexan, y la razón es estructural. Cuando VACUUM elimina entradas de un índice B-tree, las páginas que se quedan vacías se marcan como reutilizables, pero el árbol no se reequilibra ni devuelve el espacio al fichero. Un patrón sostenido de inserciones y borrados deja el índice con páginas medio llenas para siempre.

Desde PostgreSQL 14 existe una defensa importante que mucha gente todavía no sabe que tiene activada: la *bottom-up index deletion* . Cuando una página de índice está a punto de dividirse, PostgreSQL intenta primero eliminar de esa misma página las entradas que son versiones antiguas de la misma fila lógica. En cargas con muchos UPDATE sobre columnas no indexadas —el caso típico de una tabla de pedidos con un índice por cliente y otro por fecha— esto evita la mayoría de las divisiones de página y mantiene el índice estable durante meses. No ayuda con borrados reales: ahí solo VACUUM libera las entradas.

Para saber si un índice concreto está mal, `pgstatindex` da la respuesta directa:

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;

SELECT index_size, tree_level, avg_leaf_density, leaf_fragmentation
FROM pgstatindex('idx_pedidos_cliente_creado');
```

Un índice recién construido ronda un `avg_leaf_density` de 90. Por debajo de 50 de forma sostenida hay espacio real que recuperar, y un `leaf_fragmentation` alto indica además que los recorridos por rango van a leer más páginas de las necesarias, que es exactamente el tipo de degradación que no aparece en ningún log.

La reconstrucción en caliente es `REINDEX INDEX CONCURRENTLY` . Es incomparablemente más barata que reescribir el heap, pero no es gratis: necesita espacio en disco para una copia completa del índice, hace dos pasadas sobre la tabla y espera a que terminen las transacciones que empezaron antes que ella. Si falla a mitad, deja un índice inválido con sufijo `_ccnew` que hay que limpiar a mano, y que mientras tanto se sigue manteniendo en cada INSERT sin que ninguna consulta lo use.

```
REINDEX INDEX CONCURRENTLY idx_pedidos_cliente_creado;

-- Restos de intentos fallidos: coste de escritura sin beneficio de lectura
SELECT indexrelid::regclass AS indice_invalido,
       indrelid::regclass   AS tabla
FROM pg_index
WHERE NOT indisvalid;
```

```
DROP INDEX CONCURRENTLY idx_pedidos_cliente_creado_ccnew;
```

Hay un detalle operativo que merece la pena conocer antes de necesitarlo. La fase de índices es la más cara de un VACUUM, porque obliga a recorrer cada índice entero de principio a fin. Cuando estás corriendo contra el reloj de un wraparound, se puede saltar:

```
-- Solo para ganar tiempo en una emergencia de wraparound
VACUUM (INDEX_CLEANUP OFF, VERBOSE) eventos;
```

Esto permite congelar y avanzar el `relfrozenxid` a una velocidad muy superior, a cambio de dejar en los índices entradas que apuntan a punteros de línea muertos. Es una herramienta de rescate, no una configuración: en cuanto la instancia esté fuera de peligro hay que dejar que un VACUUM normal haga la pasada completa. Por defecto, desde PostgreSQL 14 la opción está en `AUTO` , que ya se salta la limpieza de índices por su cuenta cuando hay tan pocas tuplas muertas que no compensa el recorrido.

Un último apunte de dimensionamiento que conecta con el presupuesto de memoria: el número de pasadas sobre los índices depende de si todos los identificadores de tuplas muertas caben en `maintenance_work_mem` . Con la estructura TidStore introducida en PostgreSQL 17, esa memoria rinde muchísimo más que antes y el viejo techo efectivo de 1 GB desapareció. Si has actualizado hace poco, merece la pena volver a medir cuántas pasadas hace tu VACUUM en las tablas grandes: es frecuente pasar de cinco o seis a una sola sin cambiar ningún parámetro.

16. Tablas particionadas: el vacuum va por partición, las estadísticas del padre no

En una tabla particionada, autovacuum nunca procesa la tabla padre porque no contiene filas. Trabaja partición por partición, cada una con su propio contador de tuplas muertas y su propia fórmula de disparo. Eso es justo lo que quieres, y abre la puerta a un ajuste imposible en una tabla monolítica: tratar de forma distinta la partición caliente y las frías.

```
-- La partición del mes en curso recibe toda la escritura
ALTER TABLE eventos_2026_08 SET (
  autovacuum_vacuum_scale_factor = 0.01,
  autovacuum_vacuum_threshold    = 5000,
  autovacuum_vacuum_cost_delay   = 2,
  fillfactor                     = 85
);

-- Una partición ya cerrada: congelarla una vez y olvidarse
VACUUM (FREEZE, ANALYZE) eventos_2026_06;
```

Congelar explícitamente una partición que ya no recibe escrituras es una de las mejores inversiones de tiempo que existen en una base de datos grande. Esas páginas quedan marcadas como all-frozen en el mapa de visibilidad y ningún vacuum futuro, ni siquiera un anti-wraparound, volverá a leerlas. En una tabla de eventos con veinticuatro particiones mensuales, esto convierte el mantenimiento de veinticuatro relaciones en el mantenimiento de una.

Lo que no conviene hacer es desactivar autovacuum en las particiones frías dando por hecho que ya están limpias. `autovacuum_enabled = false` no impide el vacuum anti-wraparound —PostgreSQL lo lanza igual cuando la edad lo exige, y menos mal—, pero sí impide todo lo demás, incluido el ANALYZE. El resultado habitual es que la partición acaba recibiendo un anti-wraparound enorme, sin preparar y en el peor

momento.

La otra sorpresa está en las estadísticas. Autovacuum lanza ANALYZE sobre las particiones, pero **no sobre la tabla padre** . Las estadísticas heredadas del padre, que son las que el planificador usa cuando una consulta no puede podar particiones, solo se actualizan si alguien ejecuta el ANALYZE a mano. En tablas particionadas por fecha con consultas que cruzan varios meses, esto se manifiesta como planes que empeoran poco a poco sin ningún cambio en el código ni en el esquema.

```
-- Programarlo semanalmente, por ejemplo con pg_cron
SELECT cron.schedule(
  'analyze-eventos-padre',
  '0 4 * * 0',
  'ANALYZE eventos'
);
```

Y el punto que más ahorra a largo plazo: en una tabla particionada, la retención de datos no debería generar trabajo de vacuum en absoluto. Borrar treinta millones de filas con un `DELETE` crea treinta millones de tuplas muertas que alguien tendrá que limpiar, más el WAL correspondiente, más el bloat que queda después. Desconectar la partición del mes que ya caducó y eliminarla no crea ninguna.

```
ALTER TABLE eventos DETACH PARTITION eventos_2025_08 CONCURRENTLY;
DROP TABLE eventos_2025_08;
```

Dicho de otra forma: si tu tabla más grande tiene una política de retención y sigue creciendo por acumulación de bloat, el problema probablemente no sea autovacuum, sino que estás borrando filas donde deberías estar eliminando particiones.

17. Autovacuum en servicios gestionados: lo que puedes tocar y lo que no

Si tu PostgreSQL vive en RDS, Aurora, Cloud SQL o Azure Database, la mitad de esta guía se aplica tal cual y la otra mitad cambia de sitio. Lo que cambia es el acceso: no hay `postgresql.conf` , hay grupos de parámetros, y una parte de los ajustes está gestionada por el propio proveedor.

En RDS y Aurora, el parámetro `rds.adaptive_autovacuum` viene activado por defecto y hace algo sensato: cuando la edad de las transacciones de la instancia se acerca a los umbrales de peligro, AWS ajusta automáticamente los parámetros de autovacuum para que sea más agresivo. La recomendación oficial es dejarlo encendido, y es buena. A partir de PostgreSQL 18 va un paso más allá y escala `autovacuum_max_workers` de forma dinámica al acercarse el wraparound, algo que hasta ahora exigía un reinicio en el peor momento posible.

Ese detalle conecta con una novedad de PostgreSQL 18 que conviene entender aunque no uses AWS. `autovacuum_max_workers` pasó a ser modificable en caliente, y se añadió `autovacuum_worker_slots` , que reserva los slots de proceso en el arranque y fija el techo absoluto. Poner `autovacuum_max_workers` por encima de `autovacuum_worker_slots` no tiene ningún efecto: los trabajadores salen de ese conjunto reservado.

```
-- PostgreSQL 18: el techo se fija en el arranque, el ajuste va en caliente
SHOW autovacuum_worker_slots;      -- requiere reinicio para cambiarlo

ALTER SYSTEM SET autovacuum_max_workers = 8;
```

```
SELECT pg_reload_conf();
```

La contrapartida de subir trabajadores es la memoria, y aquí es fácil hacerse daño. Cada trabajador puede usar hasta `autovacuum_work_mem` . En Aurora ese valor se calcula por defecto como `GREATEST(DBInstanceClassMemory/32768, 65536)` , de modo que en una instancia grande puede rondar 1 GB por trabajador. Pasar de tres a dieciséis trabajadores con 1 GB cada uno significa reservar hasta 16 GB de memoria para el mantenimiento, compitiendo con la caché y con las consultas. La pregunta correcta no es cuántos trabajadores quieres, sino cuánta memoria total estás dispuesto a dedicar a autovacuum; el número de trabajadores se despeja de ahí.

Con independencia del proveedor, hay dos palancas que siempre son tuyas y que casi nadie aprovecha lo suficiente. La primera es el ajuste por tabla, que no depende de ningún grupo de parámetros ni de ningún reinicio:

```
ALTER TABLE pedidos SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 2000,
  autovacuum_vacuum_cost_delay   = 2,
  autovacuum_vacuum_cost_limit   = 2000,
  fillfactor                     = 85
);
```

La segunda es el registro. `log_autovacuum_min_duration` es, con diferencia, el parámetro con mejor relación entre coste y valor de toda esta guía. Ponerlo en 0 durante unos días te dice qué tablas visita autovacuum, cuánto tarda cada pasada, cuántas tuplas elimina y —lo más útil de todo— cuántas tuplas muertas *no ha podido* eliminar todavía, que es la señal directa de un horizonte bloqueado. Sin eso, cualquier ajuste posterior es a ciegas.

```
ALTER SYSTEM SET log_autovacuum_min_duration = 0;   -- unos días, luego '250ms'
SELECT pg_reload_conf();
```

Una advertencia de operación: en 0 registra absolutamente todas las pasadas, incluidas las de las tablas del catálogo, así que en una instancia con miles de relaciones el volumen de log es considerable. La secuencia sensata es dejarlo en 0 el tiempo justo para construir el mapa, y luego subirlo a un umbral que solo capture lo que tarda de verdad.

18. Caso práctico: de 41 GB a 9 GB sin ventana de mantenimiento

Vale la pena recorrer el ciclo completo sobre un caso concreto, porque el orden de los pasos importa bastante más que cada paso por separado.

El síntoma: una tabla `pedidos` de 41 GB en disco, con una latencia de lectura que había pasado de 15 ms a 700 ms en tres meses, sin cambios de código y sin crecimiento real del negocio. La primera medición, con la variante barata de pgstattuple:

```
SELECT * FROM pgstattuple_approx('pedidos');
-- table_len: 41 GB | approx_tuple_percent: 22 | dead_tuple_percent: 61
```

Nueve gigas de datos vivos dentro de cuarenta y uno. Con ese número delante, el reflejo habitual es programar un VACUUM FULL o un pg_repack esa misma noche. Sería un error de orden: reescribir la tabla sin resolver la causa devuelve al mismo sitio en seis semanas, y además consume la ventana de

calma que vas a necesitar después. Antes hay que responder a una pregunta concreta: por qué autovacuum, que estaba activado y pasando por la tabla, no eliminaba nada.

```
SELECT last_autovacuum, autovacuum_count, n_dead_tup, n_live_tup
FROM pg_stat_user_tables
WHERE relname = 'pedidos';
-- Pasaba cada hora. Y n_dead_tup no bajaba nunca.
```

Ese es el patrón inconfundible de un horizonte bloqueado: VACUUM corre, no falla, no registra errores y no puede eliminar nada porque alguien en el sistema todavía podría necesitar ver esas versiones. Localizar al responsable son tres consultas:

```
-- 1. Transacciones vivas más antiguas, incluidas las idle in transaction
SELECT pid, state,
       age(backend_xid)  AS edad_xid,
       age(backend_xmin) AS edad_xmin,
       now() - xact_start AS duracion,
       left(query, 60)    AS consulta
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL OR backend_xid IS NOT NULL
ORDER BY greatest(age(backend_xid), age(backend_xmin)) DESC
LIMIT 5;

-- 2. Slots de replicación
SELECT slot_name, active, age(xmin) AS edad_xmin, inactive_since
FROM pg_replication_slots
ORDER BY age(xmin) DESC NULLS LAST;

-- 3. Transacciones preparadas huérfanas
SELECT gid, prepared, age(transaction) AS edad
FROM pg_prepared_xacts
ORDER BY prepared;
```

El culpable estaba en la segunda: un slot llamado `replica_informes` , inactivo desde hacía setenta y cuatro días, de una réplica apagada durante una migración y cuyo slot nadie borró. Retenía además 380 GB de WAL que ya habían obligado a ampliar el disco dos veces, sin que nadie relacionase las dos cosas.

```
SELECT pg_drop_replication_slot('replica_informes');
```

Con el horizonte liberado, el siguiente VACUUM sí tuvo efecto. Conviene subir la memoria de mantenimiento en la propia sesión antes de lanzarlo, para que no tenga que hacer varias pasadas por los índices:

```
SET maintenance_work_mem = '2GB';
VACUUM (VERBOSE, ANALYZE) pedidos;
```

La tabla siguió midiendo 41 GB, porque VACUUM no encoge el fichero, pero esos 32 GB de hueco pasaron a ser reutilizables y la latencia bajó de 700 ms a unos 200 ms sin tocar nada más. Lo que quedaba era espacio en disco y páginas frías ensuciando la caché, y para eso sí tocaba reescribir. Con pg_repack, sin bloquear la tabla:

```
pg_repack -d produccion -t public.pedidos --jobs 4 --wait-timeout 30
```

Dos avisos sobre pg_repack: necesita espacio libre para una copia completa de la tabla y de sus índices, y

toma un lock exclusivo breve al principio y otro al final. Por eso conviene lanzarlo con `--wait-timeout` y a una hora tranquila, aunque técnicamente no requiera parada.

Y la parte que evita la repetición, que es la que de verdad cierra el incidente:

```
ALTER TABLE pedidos SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 2000,
  fillfactor                     = 90
);
```

Más una alerta sobre slots inactivos con más de veinticuatro horas sin conexión, que es exactamente la que habría convertido tres meses de degradación silenciosa en un aviso de cinco minutos. El resultado final: 9,4 GB en disco, latencia de lectura en 14 ms, 380 GB de WAL liberados y un runbook una página más largo.

La lección que queda no es ninguna de las órdenes anteriores, sino la secuencia: medir, encontrar el bloqueador, quitarlo, dejar que VACUUM haga su trabajo, recuperar el espacio solo si sigue haciendo falta y, por último, ajustar para que no vuelva a pasar. Empezar por el final —que es lo que hace casi todo el mundo— arregla el síntoma durante un mes.

19. El visibility map: el segundo trabajo de VACUUM

Hasta aquí hemos tratado a VACUUM como si su única misión fuera liberar espacio. No lo es. Cada pasada actualiza también el **visibility map** , un fichero auxiliar minúsculo —dos bits por página de datos— que acompaña a cada tabla. El primer bit dice "todas las tuplas de esta página son visibles para cualquier transacción". El segundo dice "todas están congeladas".

Esos dos bits sostienen dos optimizaciones importantes. La primera beneficia al propio VACUUM: si una página está marcada como all-visible, la siguiente pasada la salta sin leerla, que es la razón por la que un vacuum sobre una tabla tranquila termina en segundos aunque la tabla ocupe 200 GB. La segunda es mucho más visible para tus usuarios: el *index-only scan* .

Un index-only scan responde una consulta leyendo solo el índice, sin tocar la tabla. El problema es que un índice de PostgreSQL no guarda información de visibilidad: sabe que existe una entrada apuntando a una tupla, pero no sabe si esa tupla sigue viva para tu transacción. Para poder saltarse el heap, el ejecutor consulta el visibility map. Si la página está marcada all-visible, se ahorra el acceso. Si no lo está, tiene que ir a la tabla a comprobar la fila, y eso es un *heap fetch* .

La consecuencia práctica incomoda: **un índice de cobertura perfecto rinde como un index scan corriente si el visibility map está desactualizado** . Y el visibility map se desactualiza exactamente cuando autovacuum no llega. El bloat, entonces, no solo te obliga a leer más páginas: además te quita el atajo que servía para no leerlas.

Se mide en el EXPLAIN, en una línea que mucha gente pasa por alto:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_id, created_at
FROM orders
WHERE created_at >= now() - interval '7 days';

--  Index Only Scan using orders_created_at_customer_idx on orders
--    (actual time=0.031..118.402 rows=412903 loops=1)
--    Index Cond: (created_at >= (now() - '7 days'::interval))
```

```
--    Heap Fetches: 398211          <-- el atajo no está funcionando
--    Buffers: shared hit=41238 read=390122
```

Ese `Heap Fetches` es el número que importa. Si se acerca al número de filas devueltas, tu index-only scan es un index scan disfrazado. Tras un vacuum de la tabla, la misma consulta suele bajar a `Heap Fetches: 0` y a una fracción de los buffers leídos.

Para ver el mapa por dentro está la extensión `pg_visibility` , que viene en contrib y no cuesta nada instalar:

```
CREATE EXTENSION IF NOT EXISTS pg_visibility;

-- Resumen rápido: cuántas páginas están marcadas visibles y congeladas
SELECT * FROM pg_visibility_map_summary('orders');
--  all_visible | all_frozen
-- -------------+------------
--       118204 |      92311

-- Cobertura real en porcentaje
SELECT
  round(100.0 * count(*) FILTER (WHERE all_visible) / count(*), 1) AS pct_visible,
  round(100.0 * count(*) FILTER (WHERE all_frozen)  / count(*), 1) AS pct_frozen,
  count(*) AS total_pages
FROM pg_visibility('orders');
```

Una cobertura de all_visible por debajo del 80% en una tabla grande y mayoritariamente estable es una señal clara de que autovacuum no está pasando lo suficiente por ahí. Y hay un caso especialmente traicionero: `pg_visibility` también sirve para verificar que el mapa no *miente* . Si el visibility map dice que una página es all-visible y no lo es, obtendrás resultados incorrectos en index-only scans, sin ningún error. Eso pasa tras ciertos fallos de hardware o bugs de almacenamiento:

```
-- Debe devolver cero filas. Si devuelve alguna, el mapa está corrupto.
SELECT * FROM pg_check_visible('orders');
SELECT * FROM pg_check_frozen('orders');

-- Reconstrucción: fuerza a VACUUM a leer todas las páginas
VACUUM (DISABLE_PAGE_SKIPPING) orders;
```

`DISABLE_PAGE_SKIPPING` le dice a VACUUM que ignore el visibility map y lea la tabla entera. Es caro y no es algo de rutina, pero es la herramienta correcta cuando sospechas del mapa o cuando acabas de restaurar desde un backup físico dudoso.

Tres reglas prácticas que salen de todo esto:

- Monitoriza `Heap Fetches` en las consultas que dependen de índices de cobertura. Es la métrica que traduce "autovacuum va retrasado" a "esta pantalla tarda tres veces más".
- Después de una carga masiva, ejecuta un `VACUUM` explícito aunque no haya una sola tupla muerta. Sin él, ninguna página nueva tendrá el bit all-visible y los index-only scans sobre esos datos no existirán.
- No confundas cobertura del visibility map con ausencia de bloat. Una tabla puede tener 95% de páginas all-visible y aun así estar hinchada: son cosas distintas que el mismo proceso mantiene.

20. Tablas append-only: el vacuum que durante años no llegaba nunca

Repasa la fórmula de disparo de la sección 3 y busca el término que representa las inserciones. No está. Durante la mayor parte de la historia de PostgreSQL, autovacuum solo miraba tuplas muertas: UPDATEs y DELETEs. Una tabla que solo recibe INSERTs no genera tuplas muertas, así que el contador nunca subía y autovacuum nunca la visitaba.

Eso describe a una familia entera de tablas muy comunes: eventos, logs de auditoría, telemetría, tablas de hechos de un almacén de datos, historiales de precios. Tablas enormes, que solo crecen, y a las que el mantenimiento automático ignoraba por completo.

Las consecuencias eran tres, y ninguna daba la cara hasta que era tarde:

1. **Ningún index-only scan.** Sin vacuum no hay visibility map, y sin visibility map las consultas analíticas sobre esas tablas leían el heap entero aunque el índice tuviera todas las columnas necesarias.
2. **Una tormenta de congelación diferida.** Como nada disparaba el vacuum, la tabla llegaba intacta a `autovacuum_freeze_max_age` . Entonces, de golpe, autovacuum arrancaba una pasada anti-wraparound sobre 400 GB que nadie había tocado nunca, normalmente a las tres de la tarde de un martes.
3. **Estadísticas viejas.** El autoanalyze sí se dispara con inserciones, pero solo actualiza estadísticas; no toca el visibility map ni congela nada.

PostgreSQL 13 cerró el agujero con dos parámetros nuevos que introducen un segundo umbral, paralelo al de tuplas muertas:

```
umbral_insert = autovacuum_vacuum_insert_threshold
              + autovacuum_vacuum_insert_scale_factor * n_filas

-- Valores por defecto
autovacuum_vacuum_insert_threshold    = 1000
autovacuum_vacuum_insert_scale_factor = 0.2
```

Autovacuum ahora compara `n_ins_since_vacuum` contra ese umbral, y si lo supera lanza un vacuum aunque no haya una sola tupla muerta. Poner el umbral en `-1` desactiva por completo el disparo por inserciones para esa tabla, que es justo lo que querrías en un caso muy concreto: una tabla de staging que se llena y se trunca cada hora y sobre la que un vacuum es puro desperdicio.

El scale factor por defecto de 0.2 sigue siendo demasiado perezoso para tablas grandes: en una tabla de 500 millones de filas exige 100 millones de inserciones antes de mover un dedo. Para tablas append-only grandes, lo razonable es fijar un umbral absoluto y anular el factor proporcional:

```
-- Tabla de eventos append-only, grande, consultada por rangos de fecha
ALTER TABLE events SET (
  autovacuum_vacuum_insert_threshold    = 500000,
  autovacuum_vacuum_insert_scale_factor = 0.0,
  autovacuum_analyze_scale_factor       = 0.01
);

-- Comprobar el estado actual
SELECT relname,
```

```
       n_ins_since_vacuum,
       n_live_tup,
       last_vacuum,
       last_autovacuum
FROM pg_stat_user_tables
WHERE relname = 'events';
```

Con `scale_factor = 0.0` y umbral de 500.000, la tabla recibe un vacuum cada medio millón de filas nuevas, sea cual sea su tamaño. Ese vacuum es barato —solo lee las páginas nuevas, gracias al propio visibility map que está manteniendo— y hace tres cosas a la vez: marca las páginas como visibles, las congela mientras están calientes en caché y evita la tormenta anti-wraparound futura.

Si la tabla está particionada por fecha, y debería estarlo, los parámetros van **en las particiones** , no en el padre. La tabla padre no tiene almacenamiento propio y autovacuum trabaja partición a partición. Un patrón que funciona bien es aplicar ajustes agresivos a la partición activa y congelar en firme las particiones cerradas:

```
-- Partición del mes en curso: recibe todas las escrituras
ALTER TABLE events_2026_08 SET (
  autovacuum_vacuum_insert_threshold    = 200000,
  autovacuum_vacuum_insert_scale_factor = 0.0
);

-- Particiones cerradas: una congelación definitiva y a olvidarse
VACUUM (FREEZE, ANALYZE) events_2026_07;
```

Hay además un truco de carga que ahorra el trabajo antes de crearlo. Si haces `COPY` dentro de la misma transacción que crea o trunca la tabla, la opción `FREEZE` escribe las filas ya congeladas y con las páginas marcadas como visibles desde el primer momento:

```
BEGIN;
TRUNCATE events_staging;
COPY events_staging FROM '/data/eventos.csv' WITH (FORMAT csv, FREEZE);
COMMIT;
-- Las páginas nacen all-visible y all-frozen: cero trabajo pendiente
-- para autovacuum, e index-only scans disponibles de inmediato.
```

La restricción es estricta: `FREEZE` exige que la tabla haya sido creada o truncada en la misma transacción. Fuera de ese caso, la alternativa es un `VACUUM (FREEZE, ANALYZE)` justo después de la carga, mientras los datos siguen en la caché del sistema operativo y leerlos es casi gratis.

Un último detalle sobre las versiones. Si administras un PostgreSQL 12 o anterior —quedan más de los que parece— no tienes ninguno de estos parámetros. La única defensa es un `VACUUM` programado con cron sobre las tablas append-only. No es elegante, pero evita la tormenta.

21. Bloat del catálogo: pg_attribute, tablas temporales y DDL

Las tablas del catálogo del sistema son tablas normales. Tienen filas, sufren MVCC, acumulan tuplas muertas y se hinchan igual que las tuyas. La diferencia es que casi nadie las mira, y que cuando se hinchan los síntomas no se parecen en nada a los de una tabla de negocio.

El síntoma típico es este: las consultas van bien, pero *abrir una conexión* tarda 300 ms; el tiempo de planificación de consultas triviales sube a decenas de milisegundos; un `\d tabla` en psql se queda

pensando. Todo eso ocurre porque cada planificación consulta el catálogo, y si `pg_attribute` tiene 4 GB de los cuales 200 MB son datos vivos, cada búsqueda paga el precio.

Las causas son siempre las mismas tres:

- **Tablas temporales creadas y destruidas sin parar.** Cada `CREATE TEMP TABLE` inserta filas en `pg_class` , `pg_attribute` , `pg_type` y `pg_depend` . Cada `DROP` las mata. Una aplicación que crea una temporal por petición genera miles de tuplas muertas por segundo en el catálogo.
- **DDL frecuente por diseño.** Arquitecturas multi-tenant con un esquema o un juego de tablas por cliente, sistemas que crean particiones al vuelo, suites de tests que hacen migraciones en bucle.
- **Herramientas que usan tablas temporales por dentro.** Algunos ORMs, procesos ETL y extensiones lo hacen sin que tú escribas una sola línea de DDL.

La medición es idéntica a la de cualquier tabla, solo que la vista es `pg_stat_sys_tables` :

```
SELECT relname,
       n_live_tup,
       n_dead_tup,
       round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct_dead,
       pg_size_pretty(pg_total_relation_size(relid)) AS size,
       last_autovacuum
FROM pg_stat_sys_tables
WHERE schemaname = 'pg_catalog'
  AND n_dead_tup > 1000
ORDER BY n_dead_tup DESC
LIMIT 10;

-- Los sospechosos habituales, por este orden:
-- pg_attribute, pg_class, pg_type, pg_depend, pg_shdepend, pg_statistic
```

Autovacuum sí procesa el catálogo, con los mismos umbrales y sujeto a los mismos bloqueadores de la sección 9. Si tienes una transacción abierta desde hace seis horas, el catálogo tampoco se limpia. Pero hay un matiz propio: `pg_attribute` y `pg_class` son tablas *anchas en filas por objeto* , así que un pico de DDL las lleva de 20.000 a 2 millones de filas en un rato y el scale factor del 20% se vuelve un umbral gigantesco justo cuando menos conviene.

Los parámetros por tabla también funcionan en el catálogo, y ese es el arreglo estructural:

```
-- Requiere superusuario
ALTER TABLE pg_catalog.pg_attribute SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 1000
);
ALTER TABLE pg_catalog.pg_class SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 1000
);

-- Limpieza puntual si ya está hinchado
VACUUM (ANALYZE, VERBOSE) pg_catalog.pg_attribute;
```

Si el catálogo ya está muy hinchado y el vacuum normal no recupera nada, la única salida es `VACUUM FULL pg_catalog.pg_attribute` , y aquí conviene ser muy claro: eso toma un lock ACCESS EXCLUSIVE sobre una tabla que *toda* consulta necesita. Bloquea la base de datos entera durante lo que dure. No es un mantenimiento en caliente; es una ventana planificada, y ni `pg_repack` ni `pg_squeeze` pueden ayudarte porque no operan sobre catálogos.

Por eso el trabajo real es preventivo. La alternativa a crear y destruir temporales es reutilizarlas:

```
-- Anti-patrón: una tabla nueva en el catálogo por cada petición
CREATE TEMP TABLE resultados_tmp AS SELECT ...;
-- ... uso ...
DROP TABLE resultados_tmp;

-- Mejor: crear una vez por sesión y vaciarla en cada transacción
CREATE TEMP TABLE IF NOT EXISTS resultados_tmp (
  id bigint,
  total numeric
) ON COMMIT DELETE ROWS;

-- Mejor todavía: no usar tabla temporal
WITH resultados AS (SELECT ...)
SELECT * FROM resultados JOIN ...;
```

`ON COMMIT DELETE ROWS` vacía la tabla al terminar cada transacción pero mantiene su definición, así que el catálogo se toca una vez por sesión en lugar de una vez por petición. Con un pooler en modo transaction y sesiones de vida larga, la diferencia es de varios órdenes de magnitud.

Y un aviso sobre el almacenamiento de las propias tablas temporales: viven en el esquema temporal de su sesión y **autovacuum no las toca nunca** . Solo la sesión propietaria puede vacuumarlas. Si tienes un proceso de larga duración que llena y actualiza una temporal durante horas, esa temporal se hincha sin control y nadie la va a limpiar por ti. En ese caso, un `VACUUM` explícito dentro de la propia sesión es obligatorio.

22. VACUUM en paralelo: qué paraleliza de verdad

Desde PostgreSQL 13, `VACUUM` acepta la opción `PARALLEL` . Vale la pena entender exactamente qué reparte entre workers, porque el nombre promete más de lo que da.

Un vacuum tiene tres fases: escanear el heap y anotar los identificadores de las tuplas muertas, limpiar los índices de esas entradas, y volver al heap a liberar el espacio. **Solo la fase de índices se paraleliza** , y el reparto es de un worker por índice. El escaneo del heap, que en tablas con pocos índices es la mayor parte del trabajo, sigue siendo de un solo proceso.

De ahí salen las condiciones para que sirva de algo:

- La tabla necesita **varios índices** . Con uno solo, el paralelismo no aporta nada porque no hay nada que repartir.
- Cada índice debe superar `min_parallel_index_scan_size` (512 kB por defecto) para ser candidato.
- El número real de workers queda limitado por `max_parallel_maintenance_workers` , que por defecto es 2.
- Y el más importante: **autovacuum nunca usa el paralelismo** hasta PostgreSQL 19. Da igual lo que configures; los workers de autovacuum procesan tabla e índices en serie.

Esa última línea explica un patrón que se ve mucho en producción: una tabla ancha con doce índices monopoliza un worker de autovacuum durante cuarenta minutos mientras el resto de la base de datos espera turno. La solución hasta ahora ha sido sacar esa tabla del circuito automático y darle un vacuum manual en la ventana tranquila:

```
-- Comprobar el techo actual
SHOW max_parallel_maintenance_workers;  -- 2 por defecto

-- Subirlo solo para esta sesión de mantenimiento
SET max_parallel_maintenance_workers = 6;

-- Vacuum manual con paralelismo explícito y traza de lo que hace
VACUUM (PARALLEL 4, VERBOSE, ANALYZE) orders;

-- La salida indica cuántos workers se lanzaron de verdad:
-- INFO:  launched 3 parallel vacuum workers for index vacuuming
--        (planned: 4)
```

Fíjate en el "planned: 4" frente a "launched: 3". Los workers salen del mismo pool que las consultas paralelas, acotado por `max_worker_processes` y `max_parallel_workers` . Si el servidor está ocupado, pides cuatro y te dan los que haya. Nunca falla por eso: simplemente va más lento.

Hay una segunda forma de paralelismo que se confunde a menudo con esta y que es igual de útil: `vacuumdb --jobs` . No reparte el trabajo de *una* tabla, sino que abre varias conexiones y vacuuma *varias tablas a la vez* . Para el mantenimiento programado de una base entera es lo que quieres:

```
# Ocho tablas en paralelo, cada una con sus indices en paralelo
vacuumdb --all --analyze --jobs 8 --parallel 4 --verbose

# Solo las tablas mas urgentes por edad de congelacion
vacuumdb --all --freeze --jobs 4 --min-xid-age 500000000
```

Y una advertencia de operación: cada worker paralelo consume su propia porción de `maintenance_work_mem` . Con `maintenance_work_mem = 2GB` y cuatro workers puedes estar comprometiendo bastante más memoria de la que tenías en la cabeza. En un servidor ajustado, baja la memoria antes de subir el paralelismo, no al revés.

23. Actualizaciones mayores: pg_upgrade, la tormenta de freeze y las estadísticas

Una actualización mayor de PostgreSQL interactúa con todo lo que hemos visto, y el resultado sorprende a mucha gente el primer lunes después del cambio.

La buena noticia primero: `pg_upgrade` conserva `relfrozenxid` . No hay que volver a congelar nada, y el reloj del wraparound sigue donde estaba. Lo que *no* se conservaba, hasta PostgreSQL 17 incluido, eran las estadísticas del planificador. La base arrancaba con cero información sobre la distribución de tus datos y el planificador tomaba decisiones a ciegas: seq scans donde había índices perfectos, nested loops sobre millones de filas, timeouts en endpoints que iban bien el viernes.

PostgreSQL 18 arregla la parte principal: `pg_upgrade` ahora transfiere la mayoría de las estadísticas del optimizador. Pero quedan dos huecos, y el segundo es el que toca directamente a este artículo:

- Las **estadísticas extendidas** creadas con `CREATE STATISTICS` no se transfieren. Si dependes de ellas para correlaciones entre columnas, hay que regenerarlas.
- Las **estadísticas acumuladas** —las de `pg_stat_user_tables` : `n_dead_tup` , `n_ins_since_vacuum` , `n_mod_since_analyze` — se pierden siempre. Y esos son exactamente los contadores que disparan autovacuum.

Esto último produce un efecto curioso: después de la actualización, autovacuum cree que ninguna tabla tiene tuplas muertas. Todos los contadores están a cero. Una tabla que llevaba un mes acumulando bloat vuelve a empezar desde cero y no se disparará hasta acumular otra vez su umbral completo, aunque el bloat real siga ahí. La base parece tranquila, y lo que está pasando es que el mantenimiento está ciego.

La secuencia recomendada tras un `pg_upgrade` a 18, en este orden:

```
# 1. Estadisticas minimas para lo que falte, en fases, y rapido.
#    --missing-stats-only evita machacar las que si se transfirieron.
vacuumdb --all --analyze-in-stages --missing-stats-only --jobs 8

# 2. Ahora un ANALYZE completo, que ademas repuebla
#    los contadores acumulados que disparan autovacuum.
vacuumdb --all --analyze-only --jobs 8

# 3. Regenerar las estadisticas extendidas que se perdieron
psql -c "SELECT format('ANALYZE %I.%I;', n.nspname, c.relname)
         FROM pg_statistic_ext e
         JOIN pg_class c ON c.oid = e.stxrelid
         JOIN pg_namespace n ON n.oid = c.relnamespace;"

# 4. Y una pasada de vacuum de verdad, sin prisa, fuera de hora punta
vacuumdb --all --jobs 4 --verbose
```

El paso 1 es el que evita el desastre inmediato: `--analyze-in-stages` hace tres pasadas con objetivos de estadísticas crecientes, de modo que el planificador tiene *algo* con lo que trabajar en cuestión de minutos en lugar de esperar horas a un ANALYZE completo. Con `--missing-stats-only` , en PostgreSQL 18, además no destruye las estadísticas buenas que sí sobrevivieron al salto.

Si vienes de PostgreSQL 17 o anterior, el paso 1 no lleva `--missing-stats-only` —esa opción no existe— y hay que asumir que se analiza todo. Reserva tiempo de ventana para ello y no abras el tráfico hasta que la primera fase haya terminado.

24. Lo que viene en PostgreSQL 19: autovacuum que prioriza y REPACK en el core

PostgreSQL 19 publicó su primera beta el 4 de junio de 2026 y la versión estable se espera para la segunda mitad del año. Nada de esto debería tocar tu producción todavía, pero dos de sus cambios afectan de lleno a lo que cuenta esta guía y conviene saber hacia dónde va el terreno.

Autovacuum aprende a hacer triaje

Hasta ahora, un worker de autovacuum construía su lista de tablas candidatas y las recorría más o menos en el orden en que aparecen en `pg_class` . Muy democrático y bastante absurdo: una tabla a punto de provocar una parada por wraparound recibía el mismo trato que una que acababa de cruzar el umbral de ANALYZE.

PostgreSQL 19 introduce un sistema de puntuación. Antes de elegir por dónde empezar, calcula cinco puntuaciones por tabla —edad del transaction ID, edad del multixact, tuplas muertas pendientes, tuplas recién insertadas y cambios desde el último ANALYZE— y se queda con la más alta como prioridad de esa tabla. Cada componente es en esencia una razón: cuánto se ha pasado la tabla de su umbral de disparo.

Lo mejor es que deja de ser una caja negra. Hay una vista nueva:

```
SELECT relname,
       ceil(score)               AS score,
       ceil(vacuum_score)        AS vacuum_score,
       ceil(vacuum_insert_score) AS insert_score,
       ceil(analyze_score)       AS analyze_score,
       ceil(xid_score)           AS xid_score,
       for_wraparound
FROM pg_stat_autovacuum_scores
ORDER BY score DESC
LIMIT 20;

--    relname   | score | vacuum_score | insert_score | analyze_score
-- -------------+-------+--------------+--------------+---------------
--  append_log  | 10000 |            0 |          500 |         10000
--  churn_queue |  6800 |         2800 |          200 |          6800
```

Durante años, diagnosticar por qué autovacuum estaba haciendo *eso* en lugar de lo que tú querías consistía en leer código fuente y hacer conjeturas informadas. Ahora está en una vista que puedes poner en un dashboard.

Y se puede inclinar la balanza. Cinco de los seis parámetros nuevos son pesos que multiplican cada componente antes de decidir el máximo:

```
-- Todos valen 1.0 por defecto: todas las preocupaciones pesan igual
autovacuum_freeze_score_weight           = 1.0
autovacuum_multixact_freeze_score_weight = 1.0
autovacuum_vacuum_score_weight           = 1.0
autovacuum_vacuum_insert_score_weight    = 1.0
autovacuum_analyze_score_weight          = 1.0

-- Ejemplo: recuperar espacio importa el doble, refrescar
-- estadisticas la mitad. Son parametros sighup: basta un reload.
ALTER SYSTEM SET autovacuum_vacuum_score_weight  = 2.0;
ALTER SYSTEM SET autovacuum_analyze_score_weight = 0.5;
SELECT pg_reload_conf();
```

Hay dos sutilezas que merecen atención. La primera: poner los cinco pesos a `0.0` devuelve el comportamiento anterior a la versión 19, el orden de catálogo de toda la vida. Es una salida de emergencia documentada. La segunda: los dos pesos de congelación no se comportan igual que los demás. Subirlos por encima de 1.0 no solo multiplica la puntuación, también *divide los umbrales* por el peso. Un `autovacuum_freeze_score_weight = 2.0` significa que la presión de congelación empieza a considerarse urgente a la mitad de `autovacuum_freeze_max_age` . Es potente y es fácil pasarse.

El sexto parámetro es de otra naturaleza: `autovacuum_max_parallel_workers` permite por fin que un worker de autovacuum recluta ayudantes para procesar índices en paralelo, cerrando el hueco de la sección 22. Hay que optar explícitamente porque el valor por defecto es 0:

```
ALTER SYSTEM SET autovacuum_max_parallel_workers = 4;
SELECT pg_reload_conf();
```

REPACK: la respuesta del core al bloat que ya está ahí

La sección 11 de esta guía existe porque recuperar espacio de una tabla hinchada tenía dos malas respuestas: `VACUUM FULL` , que toma un lock ACCESS EXCLUSIVE y bloquea todo, o una extensión externa como `pg_repack` o `pg_squeeze` , con su propia instalación y su propio conjunto de aristas.

PostgreSQL 19 mete la solución en el core con un comando nuevo, `REPACK` , que absorbe lo que hacían `VACUUM FULL` y `CLUSTER` . Lo interesante es la variante concurrente, construida sobre la maquinaria de decodificación lógica que `pg_squeeze` fue el primero en explorar:

```
-- Reescritura clasica, con lock exclusivo (equivale a VACUUM FULL)
REPACK orders;

-- Reescritura sin lock exclusivo: la tabla sigue aceptando escrituras
REPACK CONCURRENTLY orders;

-- Reordenando fisicamente por un indice (equivale a CLUSTER)
REPACK orders USING INDEX orders_created_at_idx;

-- Seguimiento en vivo
SELECT * FROM pg_stat_progress_repack;
```

Por dentro, el modo concurrente copia la tabla y reproduce mediante decodificación lógica los cambios que van ocurriendo durante la copia, por eso aparece un parámetro nuevo, `max_repack_replication_slots` , con valor por defecto 5. Los dos comandos antiguos siguen funcionando en la versión 19, pero el desarrollo futuro va a `REPACK` .

La lectura práctica para quien planifica: si hoy mantienes `pg_repack` instalado únicamente para desinflar tablas cada cierto tiempo, esa dependencia tiene fecha de caducidad. No migres nada todavía —esto es beta— pero anótalo en el plan de la próxima actualización mayor.

25. Validar tu configuración antes de que lo haga producción

Todo lo anterior son decisiones, y una decisión de configuración que nadie ha medido es una hipótesis. La buena noticia es que el bloat es uno de los pocos problemas de bases de datos que se reproduce con fidelidad en un portátil, porque no depende del volumen absoluto de datos sino de la relación entre la tasa de escritura y la capacidad del vacuum.

El banco de pruebas mínimo tiene tres piezas: una tabla con el patrón de escritura que te preocupa, una carga sostenida, y una medición honesta del estado estacionario.

```
-- setup.sql: una tabla que imita una cola de trabajo
DROP TABLE IF EXISTS bench_queue;
CREATE TABLE bench_queue (
  id       bigserial PRIMARY KEY,
  state    int  NOT NULL DEFAULT 0,
  claimed_by text,
  updated_at timestamptz NOT NULL DEFAULT now(),
  payload  jsonb NOT NULL
);
CREATE INDEX ON bench_queue (state, id);
CREATE INDEX ON bench_queue (updated_at);

INSERT INTO bench_queue (payload)
SELECT jsonb_build_object('n', g)
FROM generate_series(1, 2000000) g;

VACUUM (FREEZE, ANALYZE) bench_queue;
```

El script de carga reproduce el patrón que hincha las colas: reclamar filas, actualizarlas y borrarlas. Es un fichero para `pgbench` :

```
-- churn.sql
\set id random(1, 2000000)
BEGIN;
UPDATE bench_queue
   SET state = 1, claimed_by = 'w' || :client_id, updated_at = now()
 WHERE id = :id;
UPDATE bench_queue
   SET state = 2, updated_at = now()
 WHERE id = :id;
DELETE FROM bench_queue WHERE id = :id AND random() < 0.05;
END;
```

Y la ejecución. La clave metodológica es correr dos veces: una con la configuración actual y otra con la propuesta, cambiando **una sola cosa** entre ambas:

```
# Linea base: 20 minutos de carga sostenida, 16 clientes
pgbench -n -f churn.sql -c 16 -j 4 -T 1200 -P 60 bench

# Durante la carga, en otra sesion, muestreo cada 30 segundos
watch -n 30 "psql -qAt -d bench -c \"
  SELECT n_dead_tup,
         pg_size_pretty(pg_total_relation_size('bench_queue')),
         last_autovacuum
  FROM pg_stat_user_tables WHERE relname = 'bench_queue';\""
```

Lo que buscas no es un número bonito, sino una **forma de curva** . Si `n_dead_tup` sube, autovacuum entra, baja, y el ciclo se repite alrededor de una media estable, tu configuración aguanta ese ritmo de escritura. Si la media de cada ciclo es más alta que la del anterior, autovacuum va perdiendo terreno y en producción eso se traduce en bloat que crece indefinidamente. Es la misma señal que un backlog de cola que no se drena.

La medición final, cuando la carga ha terminado, la da `pgstattuple` , que ya conoces de la sección 4:

```
SELECT
  round(free_percent::numeric, 1)  AS pct_libre,
  round(dead_tuple_percent::numeric, 1) AS pct_muerto,
  pg_size_pretty(table_len)        AS tamano
FROM pgstattuple('bench_queue');

-- Y el coste que pagaste por mantenerlo asi:
SELECT vacuum_count, autovacuum_count,
       n_tup_upd, n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd / NULLIF(n_tup_upd,0), 1) AS pct_hot
FROM pg_stat_user_tables WHERE relname = 'bench_queue';
```

Ese `pct_hot` del final es la métrica que cierra el círculo con la sección 7. Si sale bajo, prueba a bajar el `fillfactor` de la tabla a 80 y repite el experimento entero. Verás subir los HOT updates y, con ellos, bajar tanto el trabajo del vacuum como el bloat de los índices. Es la clase de mejora que en producción, sin banco de pruebas, jamás te habrías atrevido a justificar.

Dos cosas que este banco de pruebas *no* mide, y conviene tenerlas presentes: el impacto de la latencia de tu almacenamiento real —un vacuum en NVMe local y otro en un volumen de red con IOPS aprovisionadas no se parecen en nada— y la interferencia con el resto de la carga de la base de datos. Para ambas cosas, el sitio correcto es un entorno de staging con el mismo tipo de disco que producción y una réplica de tu tráfico real. Pero el patrón de la curva, que es la decisión importante, sale ya en el portátil.

26. Ocho errores que se repiten

1. **Desactivar autovacuum** "porque consume I/O". Estás cambiando un coste continuo y predecible por un anti-wraparound de seis horas en el peor momento posible. Si autovacuum molesta, el problema es el tuning, no autovacuum.
2. **Ajustar solo globalmente.** Un scale factor de 0.01 en postgresql.conf castiga a doscientas tablas pequeñas para arreglar tres grandes.
3. **Subir** `autovacuum_max_workers` **sin subir** `vacuum_cost_limit` **.** El presupuesto es compartido: más trabajadores sobre el mismo presupuesto significa que cada uno va más lento.
4. **Tratar VACUUM FULL como "un VACUUM más fuerte".** Bloquea la tabla entera. Lanzarlo a las 10 de la mañana sobre la tabla de pedidos es una caída autoinfligida.
5. **Ignorar el bloat de índices.** Es habitual que los índices estén más hinchados que la tabla, y REINDEX CONCURRENTLY es incomparablemente más barato que reescribir el heap.
6. **Alertar solo sobre** `n_dead_tup` **.** No detecta ni la deuda de congelación ni el horizonte bloqueado, que son las dos formas en que esto se convierte en incidente grave.
7. **Dejar slots de replicación de réplicas retiradas.** Retienen WAL y horizonte para siempre. Audítalos cada vez que desmontes una réplica.
8. **Borrar millones de filas sin plan posterior.** Un DELETE masivo deja el espacio dentro del fichero. Si no vas a reinsertar un volumen parecido, planifica el reempaquetado; si es un borrado recurrente por fechas, usa particionado y `DROP PARTITION` , que es una operación de metadatos y no genera ni una tupla muerta.

27. Checklist de producción

- Alertas activas sobre `age(datfrozenxid)` a 500M (aviso) y 1.200M (crítico).
- Alerta sobre tablas con más de 1M de tuplas muertas y sin autovacuum en 24 horas.
- Alerta sobre slots de replicación con xmin antiguo o WAL retenido creciente.
- `idle_in_transaction_session_timeout` y (en PG17+) `transaction_timeout` configurados.
- Overrides por tabla en toda relación con más de un millón de filas y escritura frecuente.
- `vacuum_cost_limit` acorde al almacenamiento real, no el valor por defecto de 200.
- `maintenance_work_mem` y `autovacuum_work_mem` en 1 GB o más si hay tablas grandes.
- Revisión trimestral con `pgstattuple_approx` sobre las veinte tablas más grandes.
- Auditoría de slots de replicación en el runbook de retirada de réplicas.
- Particionado por fecha en cualquier tabla con política de retención, en lugar de DELETE periódico.
- Runbook de wraparound escrito, con las consultas de diagnóstico, y probado al menos una vez.

Preguntas frecuentes

¿VACUUM bloquea mi tabla? Un VACUUM normal toma un lock SHARE UPDATE EXCLUSIVE: convive con SELECT, INSERT, UPDATE y DELETE sin problema. Solo entra en conflicto con otro VACUUM, con ANALYZE, con CREATE INDEX y con ALTER TABLE. VACUUM FULL es otra cosa completamente distinta

y sí bloquea todo.

Lancé VACUUM y la tabla sigue midiendo lo mismo. ¿Está roto? No, está funcionando como debe. VACUUM recicla espacio dentro del fichero, no lo devuelve al sistema operativo. Si el espacio en disco es el problema, necesitas pg_repack, pg_squeeze o VACUUM FULL.

¿Cuánto bloat es aceptable? Por debajo del 20% de espacio libre es normal y hasta deseable, porque ese hueco alimenta los HOT updates. Entre 20% y 40% conviene investigar. Por encima del 40% sostenido, hay algo mal en el tuning o hay un horizonte bloqueado.

¿Puedo cancelar un autovacuum que está molestando? Un autovacuum normal se cancela solo si una consulta necesita un lock incompatible. Puedes matarlo con `pg_terminate_backend()` , pero volverá. Y si en `pg_stat_activity` la consulta dice "to prevent wraparound", **no lo mates** : es la protección de emergencia y matarla repetidamente es exactamente el camino hacia la parada de la base de datos.

¿Sirve de algo un VACUUM programado por cron si ya tengo autovacuum? Rara vez, y suele ser un parche sobre un tuning mal hecho. La excepción legítima es el vacuum manual con `FREEZE` tras una carga masiva, o antes de una ventana crítica en la que no quieres que salte un anti-wraparound.

¿Y en un servicio gestionado (RDS, Cloud SQL, Aurora)? Todo lo anterior aplica igual, con dos matices: los overrides por tabla con ALTER TABLE funcionan siempre, y en cambio algunos GUCs globales están limitados o se gestionan por grupos de parámetros. En esos entornos, el ajuste por relación no es solo una buena práctica, suele ser tu única palanca.

ENGLISH VERSION

PostgreSQL Autovacuum and Bloat: MVCC, Freezing, Wraparound and Per-Table Tuning

There is a class of PostgreSQL incident that never shows up in your application logs. It throws no exceptions, fires no error alerts and breaks no tests. The database just gets slower, week after week, until a query that took 12 ms takes 900 ms. When someone finally looks at the on-disk size, they find a 40 GB table holding 8 GB of live data.

That is **bloat** , and the root cause is almost always the same: autovacuum has been unable to do its job for weeks or months, and nobody noticed because nothing failed.

This guide covers the full cycle: why PostgreSQL produces bloat by design, what VACUUM actually does (and what it does not do, which is where most of the confusion lives), how to measure it properly, how to tune autovacuum table by table instead of editing postgresql.conf blindly, and how to get out of the worst scenario PostgreSQL has: transaction ID wraparound, the one failure mode that can leave your database refusing writes.

1. MVCC: why bloat is unavoidable

PostgreSQL never modifies a row in place. An UPDATE writes a *new version* of the row and marks the previous one as expired; a DELETE does not even remove anything, it just marks. This is what lets a long-running SELECT keep seeing a consistent view of the data while other transactions write over it: every row version carries an `xmin` (the transaction that created it) and an `xmax` (the one that deleted or replaced it), and every snapshot decides which versions it is allowed to see.

You can see it clearly with `ctid` , the physical address of the row (page, slot within the page):

```
CREATE TABLE demo (id int PRIMARY KEY, balance numeric);
INSERT INTO demo VALUES (1, 100);

SELECT ctid, xmin, xmax, * FROM demo;
--  ctid  | xmin | xmax | id | balance
-- -------+------+------+----+---------
--  (0,1) |  742 |    0 |  1 |     100

UPDATE demo SET balance = 200 WHERE id = 1;

SELECT ctid, xmin, xmax, * FROM demo;
--  ctid  | xmin | xmax | id | balance
-- -------+------+------+----+---------
--  (0,2) |  743 |    0 |  1 |     200
```

The row "moved" from slot 1 to slot 2 of page 0. The old version is still physically in the file, taking up space, invisible to any new transaction. It is called a *dead tuple* , and until something cleans it up it is dead weight your indexes and sequential scans have to walk past.

> *A practical consequence that surprises a lot of people: a table that only receives UPDATEs, with no net inserts at all, grows. A counter updated a thousand times per second produces a thousand dead versions per second.*

2. What VACUUM actually does (and what it does not)

VACUUM is the garbage collector for that design. On a normal pass it does four things:

- **Reclaims space** from dead tuples and records it in the *Free Space Map* so future writes reuse it.
- **Cleans up the index entries** that pointed at those tuples.
- **Updates the Visibility Map** , marking pages where everything is visible to everyone. This is what enables *index-only scans* , so a daily vacuum buys you more than it looks like.
- **Freezes** old transaction IDs, which is the subject of section 8.

And now what it does **not** do, which is the number one source of misunderstanding:

- **It does not return space to the operating system.** Your 40 GB file is still 40 GB. VACUUM turns it into 40 GB with 32 GB of reusable room inside. The only exception is truncating completely empty pages *at the end* of the table, which additionally needs a brief ACCESS EXCLUSIVE lock.
- **It does not physically reorder** rows or compact partially filled pages.
- **It does not recompute statistics.** That is ANALYZE, a separate process with its own thresholds.

If your table is already bloated, VACUUM stops the bleeding but does not give you the disk back. For that you have to rewrite the table (section 11).

3. When autovacuum fires: the formula behind 80% of the problems

The autovacuum launcher wakes up every `autovacuum_naptime` (1 minute by default) and, for each table, compares the estimated dead tuples against this threshold:

```
threshold = autovacuum_vacuum_threshold        -- 50 by default
          + autovacuum_vacuum_scale_factor     -- 0.2 by default
          * reltuples                          -- estimated rows in the table
```

That `0.2` is the problem. It means "wait until 20% of the table is dead". On a 10,000-row table that is 2,050 dead tuples and perfectly reasonable. On a 50-million-row table it is **10 million dead tuples** before autovacuum bothers to look. By then your queries have already degraded and the cleanup will be a long, expensive pass at exactly the wrong moment.

Two more thresholds worth keeping in mind:

- **ANALYZE** uses `autovacuum_analyze_threshold` (50) + `autovacuum_analyze_scale_factor` (0.1). Stale statistics produce bad plans, which is a different problem from bloat but arrives through the same door.
- **Insert-only tables** (logs, events, time series) generate no dead tuples, so for years they never got vacuumed and built up an enormous freezing debt. Since PostgreSQL 13 there are `autovacuum_vacuum_insert_threshold` (1000) and `autovacuum_vacuum_insert_scale_factor` (0.2) for exactly this.

PostgreSQL 18 finally adds an absolute cap, `autovacuum_vacuum_max_threshold` , defaulting to 100,000,000: no matter how large the table, once it crosses that number of dead tuples autovacuum runs anyway. It is a useful safety net, but it does not replace per-table tuning: 100 million dead tuples is still far too many for almost any real table.

To see the threshold each table has today, with overrides already applied:

```
SELECT c.relname,
       s.n_live_tup,
       s.n_dead_tup,
       (coalesce((SELECT option_value::float
                  FROM pg_options_to_table(c.reloptions)
                 WHERE option_name = 'autovacuum_vacuum_threshold'),
                current_setting('autovacuum_vacuum_threshold')::float)
        + coalesce((SELECT option_value::float
                    FROM pg_options_to_table(c.reloptions)
                   WHERE option_name = 'autovacuum_vacuum_scale_factor'),
                  current_setting('autovacuum_vacuum_scale_factor')::float)
          * c.reltuples)::bigint AS current_threshold
FROM pg_class c
JOIN pg_stat_user_tables s ON s.relid = c.oid
WHERE c.relkind = 'r'
ORDER BY s.n_dead_tup DESC
LIMIT 20;
```

If any row shows `n_dead_tup` approaching a `current_threshold` measured in millions, you know where to start.

4. Detecting bloat: three levels of precision

Level 1: the cheap daily check

```
SELECT relname,
       n_live_tup,
       n_dead_tup,
       round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct_dead,
       pg_size_pretty(pg_total_relation_size(relid)) AS size,
       last_autovacuum,
       autovacuum_count
FROM pg_stat_user_tables
WHERE n_dead_tup > 10000
ORDER BY n_dead_tup DESC
LIMIT 20;
```

It costs milliseconds and belongs on your dashboard. Two important caveats about reading it:

- `n_dead_tup` is an *estimate* from the statistics collector, not a count. A crash or a `pg_stat_reset()` zeroes it without any bloat having gone away.
- A low `n_dead_tup` does **not** mean there is no bloat. If autovacuum already cleaned the tuples but the space is scattered across half-empty pages, the counter reads zero and the file is just as fat as before. This trips up a lot of people.

The most informative column is usually `last_autovacuum` . If it is NULL or three weeks old on a table under constant writes, you have a blocked vacuum, not a slow one.

Level 2: exact measurement with pgstattuple

When level 1 raises a suspicion, measure properly:

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;

-- Exact, but reads the whole table: use it deliberately in production
SELECT table_len, tuple_percent, dead_tuple_percent, free_percent
FROM pgstattuple('public.orders');

-- Sampled: far cheaper, good enough to make a decision
SELECT approx_tuple_percent, approx_free_percent, dead_tuple_percent
FROM pgstattuple_approx('public.orders');

-- Indexes bloat too, and often worse than the table
SELECT avg_leaf_density, leaf_fragmentation
FROM pgstatindex('public.orders_pkey');
```

How to read it: on a healthy table `free_percent` usually stays under 20%. Above 40% you are paying I/O to read empty space. On B-tree indexes, an `avg_leaf_density` below 60-70% means a REINDEX CONCURRENTLY will hand you back space and latency at very low risk.

The detail that justifies this section: it is entirely normal for level 1 to report "0% dead tuples" while pgstattuple reports 37% scattered free space. They are two different things.

Level 3: watching a vacuum while it runs

```
SELECT p.pid,
       p.relid::regclass AS table_name,
       p.phase,
       pg_size_pretty(p.heap_blks_total * 8192::bigint)   AS total,
       pg_size_pretty(p.heap_blks_scanned * 8192::bigint) AS scanned,
       p.index_vacuum_count,
       a.query_start,
       now() - a.query_start AS duration
FROM pg_stat_progress_vacuum p
JOIN pg_stat_activity a USING (pid);
```

If you see `index_vacuum_count` climbing above 1, the vacuum is making multiple passes over every index because the dead tuple list does not fit in memory. That multiplies the cost and is fixed with memory (section 6).

5. Tune autovacuum per table, not globally

The most common mistake on discovering the problem is dropping `autovacuum_vacuum_scale_factor` to 0.01 in postgresql.conf. It works for the three big tables and causes constant, pointless vacuums on the two hundred small ones in the schema.

The right configuration is per relation. A profile for a write-heavy table:

```
ALTER TABLE events SET (
  autovacuum_vacuum_scale_factor  = 0.01,   -- 1% instead of 20%
  autovacuum_vacuum_threshold     = 1000,
  autovacuum_analyze_scale_factor = 0.005,  -- fresh statistics
  autovacuum_analyze_threshold    = 500,
  autovacuum_vacuum_cost_delay    = 0       -- no brake: let it finish fast
);

-- Insert-only table (logs, metrics, events)
```

```
ALTER TABLE access_logs SET (
  autovacuum_vacuum_insert_scale_factor = 0.01,
  autovacuum_vacuum_insert_threshold    = 10000
);

-- Verify what was applied
SELECT relname, reloptions FROM pg_class WHERE relname IN ('events','access_logs');

-- Go back to the global behaviour
ALTER TABLE events RESET (autovacuum_vacuum_scale_factor);
```

A reasonable starting point: any table above a million rows with frequent writes deserves a `scale_factor` between 0.01 and 0.05. Small tables keep the global values, which are correct for them.

`autovacuum_vacuum_cost_delay = 0` deserves a caveat: it removes the brake for that specific table. On an instance with I/O headroom that is exactly what you want for your hot tables. On an instance already saturated with I/O, setting it to zero on a 500 GB table can hurt everyone else's latency. Measure before you generalise it.

6. Why your vacuum is slow: the cost budget

Autovacuum has a deliberate speed limiter so it does not flatten the instance's I/O. It works on a points system:

- `vacuum_cost_page_hit` = 1 point (page already in shared_buffers)
- `vacuum_cost_page_miss` = 2 points (has to be read from disk)
- `vacuum_cost_page_dirty` = 20 points (has to be dirtied)

When a worker accumulates `vacuum_cost_limit` points (200 by default) it sleeps for `autovacuum_vacuum_cost_delay` milliseconds (2 ms by default). Do the arithmetic: 200 points every 2 ms is 100,000 points per second. If the vacuum is dirtying pages at 20 points each, that is 5,000 pages per second, or **roughly 39 MB/s** .

Any modern NVMe does twenty times that. You are throttling your maintenance to the speed of a 2010 spinning disk.

And here is the trap that ruins many attempted fixes: **the budget is shared across all autovacuum workers** . Raising `autovacuum_max_workers` from 3 to 10 without touching the cost limit speeds up nothing; it splits the same 39 MB/s across ten processes and each one gets slower.

```
# postgresql.conf - starting point for SSD/NVMe instances
autovacuum_vacuum_cost_limit = 2000    # ~390 MB/s ceiling, 10x the default
autovacuum_vacuum_cost_delay = 2ms
autovacuum_max_workers = 5             # raise this AFTER raising the limit
autovacuum_naptime = 30s
maintenance_work_mem = 1GB             # memory per manual vacuum operation
autovacuum_work_mem = 1GB              # same for autovacuum workers
vacuum_buffer_usage_limit = 16MB       # buffer ring (default 2MB since PG16)
```

Memory: the hidden reason for multiple passes

VACUUM keeps the identifiers of the dead tuples it finds in memory. If that structure fills up, it has to stop, walk *every* index on the table, empty the list and start over. With six indexes and three passes that is

eighteen full index scans.

Up to PostgreSQL 16 that structure was a sorted array with a hard 1 GB cap, so raising `maintenance_work_mem` beyond 1 GB did nothing at all. PostgreSQL 17 replaced it with a *TidStore* built on an adaptive radix tree: far denser per byte, no 1 GB cap, and with it multiple passes went from routine to rare. If you are on PG16 or older and you see `index_vacuum_count > 1` , that upgrade alone fixes the problem.

For one-off manual vacuums you can bypass the global limits:

```
VACUUM (VERBOSE, PARALLEL 4, BUFFER_USAGE_LIMIT '256MB') orders;
```

`PARALLEL` parallelises the index cleanup phase (not the heap scan), so it only helps when the table has several indexes.

7. HOT updates and fillfactor: prevention instead of cleanup

There is a shortcut inside the engine that avoids producing garbage in the first place. If an UPDATE meets two conditions, PostgreSQL performs a *HOT update* (Heap-Only Tuple):

1. It modifies no indexed column.
2. The new version fits on the same page as the old one.

In that case no new index entries are created, and the old version can be reclaimed by the opportunistic *page pruning* any read of that page performs, without waiting for VACUUM. It is the difference between a table that maintains itself and one that needs constant attention.

Measure what fraction of your updates gets this fast path:

```
SELECT relname,
       n_tup_upd,
       n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd / NULLIF(n_tup_upd, 0), 1) AS pct_hot
FROM pg_stat_user_tables
WHERE n_tup_upd > 100000
ORDER BY n_tup_upd DESC
LIMIT 20;
```

Below 50% on a write-heavy table there is real room to improve. Two levers:

Lower the fillfactor so each page keeps free room and the new version fits next to the old one:

```
ALTER TABLE sessions SET (fillfactor = 80);   -- 20% of the page reserved
-- Note: this only affects new pages. To apply it to existing data
-- you have to rewrite the table (VACUUM FULL or pg_repack).
```

70 to 85 is the usual range for update-heavy tables. Going lower means reading more pages for the same number of rows, so it is not free: it is a trade between read density and write cost.

Drop indexes on frequently updated columns. An index on `updated_at` in a constantly updated table destroys the HOT path for every single UPDATE. Before keeping it, check whether anyone uses it:

```
SELECT indexrelid::regclass AS index_name,
       relid::regclass       AS table_name,
       idx_scan,
       pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE idx_scan < 50
ORDER BY pg_relation_size(indexrelid) DESC;
```

An index with zero scans since the last stats reset costs space, costs writes and blocks HOT updates in exchange for nothing. (First confirm it is not a unique index or a foreign key support index, and that your statistics cover a representative period: do not drop an index that is only used during month-end close.)

8. Freezing and wraparound: the failure that shuts the database down

PostgreSQL transaction IDs are 32-bit integers. Comparisons are circular: for any given XID, half the space counts as "the past" and half as "the future". That gives roughly 2 billion transactions of headroom. When the counter wraps, old transactions would start to look like future ones and their rows would vanish. PostgreSQL would rather stop than allow that.

The defence is *freezing* : marking sufficiently old rows as "always visible, to everyone", taking them out of XID arithmetic entirely. VACUUM does this incrementally. The parameters that govern it:

- `vacuum_freeze_min_age` (50M): minimum XID age for a vacuum to freeze it.
- `vacuum_freeze_table_age` (150M): past this point the next vacuum is *aggressive* and scans the whole table instead of skipping all-visible pages.
- `autovacuum_freeze_max_age` (200M): an *anti-wraparound* autovacuum is launched even if autovacuum is disabled for that table.
- `vacuum_failsafe_age` (1,600M): emergency mode. The vacuum ignores the cost brake and skips index cleanup to go as fast as it can.
- MultiXact equivalents: `autovacuum_multixact_freeze_max_age` (400M) and `vacuum_multixact_failsafe_age` (1,600M). These fire under heavy `SELECT ... FOR SHARE` and foreign key usage.

This is the query that should go on your dashboard today:

```
-- Freeze age per relation
SELECT c.oid::regclass AS relation,
       age(c.relfrozenxid) AS xid_age,
       round(100.0 * age(c.relfrozenxid)
             / current_setting('autovacuum_freeze_max_age')::numeric, 1) AS pct_of_threshold,
       pg_size_pretty(pg_total_relation_size(c.oid)) AS size
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r', 'm', 't')
  AND n.nspname NOT IN ('pg_catalog', 'information_schema')
ORDER BY age(c.relfrozenxid) DESC
LIMIT 20;

-- Database-level overview
SELECT datname,
       age(datfrozenxid) AS xid_age,
       2147483647 - age(datfrozenxid) AS xids_remaining
FROM pg_database
```

```
ORDER BY age(datfrozenxid) DESC;
```

Alert thresholds that work in practice: warning at 500 million, serious alert at 1 billion, wake someone up at 1.5 billion. At 2 billion minus a small reserve, PostgreSQL starts refusing commands with "database is not accepting commands to avoid wraparound data loss".

Runbook: what to do when it is already happening

The instinct is to drop into single-user mode. **It is almost never the right move** , and it is dangerous: single-user mode disables the very protection that prevents corruption, and every transaction you run in there takes you closer to the edge. Modern PostgreSQL reserves a margin of about 3 million XIDs precisely so an administrator can fix this online. Follow this order:

```
-- 1. What is holding the horizon? Live transactions
SELECT pid, state, age(backend_xmin) AS xmin_age,
       now() - xact_start AS duration, left(query, 80) AS query
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL
ORDER BY age(backend_xmin) DESC;

-- 2. Abandoned replication slots (the usual suspect)
SELECT slot_name, active, xmin, catalog_xmin,
       age(xmin) AS xmin_age,
       pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)) AS wal_retained
FROM pg_replication_slots
ORDER BY age(xmin) DESC NULLS LAST;

-- 3. Orphaned prepared transactions (forgotten 2PC)
SELECT gid, prepared, owner, database FROM pg_prepared_xacts ORDER BY prepared;

-- 4. Remove the blocker
SELECT pg_terminate_backend(12345);            -- zombie transaction
SELECT pg_drop_replication_slot('dead_replica');
ROLLBACK PREPARED 'forgotten_gid';

-- 5. Now freeze the oldest tables, with no brake
SET vacuum_cost_delay = 0;
VACUUM (FREEZE, VERBOSE) oldest_table;
```

Step 4 is what resolves the incident. Steps 1 to 3 are what tell you which of the four causes you have. Skipping the diagnosis and firing VACUUM FREEZE straight away achieves nothing if the horizon is still held: the vacuum will run for hours and remove not a single tuple.

9. The four blockers: when VACUUM runs and cleans nothing

This is the concept that separates people who have fought this from people who have only read the docs. **VACUUM can only remove a dead tuple if no transaction in the system could still need to see it.** That cutoff is called the xmin horizon, and one single old thing is enough to freeze the whole thing:

1. **Long-running transactions** , including ones sitting `idle in transaction` . A BEGIN without a COMMIT in a connection pool can paralyse cleanup across the entire instance for days.
2. **Inactive replication slots.** A replica that was decommissioned without dropping its slot holds the horizon and the WAL indefinitely. This is the number one cause of unexplained bloat.

3. **hot_standby_feedback = on** on a replica running long analytical queries: the primary delays its cleanup so the replica does not cancel queries.
4. **Orphaned prepared transactions** , typically from an XA coordinator that died halfway through.

Defences worth having in place from day one:

```
# postgresql.conf
idle_in_transaction_session_timeout = 5min   # kills the forgotten BEGIN
transaction_timeout = 30min                  # new in PG17: total cap per transaction
statement_timeout = 60s                      # tune it per role, not globally

# Limit how much WAL a slot may retain before it is invalidated
max_slot_wal_keep_size = 100GB
# PG17+: invalidate inactive slots automatically
idle_replication_slot_timeout = 24h
```

One very useful detail: since PostgreSQL 16, `VACUUM VERBOSE` output includes the removable cutoff and how old it is. If you see something like "removable cutoff ... older than 86400 seconds", you know you have a held horizon and not a tuning problem.

10. What changed in PostgreSQL 17 and 18

If you are on PG13 or PG14, these two releases meaningfully change the maintenance picture:

PostgreSQL 17

- **TidStore** : the dead tuple structure moves from a sorted array to an adaptive radix tree. Far more density per byte and goodbye to the 1 GB cap. On large tables with several indexes, vacuum time improvements are in the 2x to 6x range depending on the case.
- `MAINTAIN` privilege: you can delegate VACUUM, ANALYZE, REINDEX and CLUSTER without handing out superuser.
- `transaction_timeout` : the guardrail that was missing against never-ending transactions.

PostgreSQL 18

- `autovacuum_vacuum_max_threshold` (default 100,000,000): an absolute dead tuple cap, so huge tables are no longer at the mercy of the scale factor.
- **Eager freezing** : a normal VACUUM now also tries to freeze pages that are already all-visible but not all-frozen, instead of leaving that whole debt to anti-wraparound. It is controlled by `vacuum_max_eager_freeze_failure_rate` (default 0.03): the fraction of pages the vacuum may try and fail to freeze before it disables eager scanning for that pass. Successes are internally capped at 20% of the candidate pages, so the cost is amortised across several runs. The net effect is that anti-wraparound passes, the six-hour ones that show up unannounced, get much shorter.
- `autovacuum_max_workers` can now be changed with a plain reload; the hard ceiling is set by `autovacuum_worker_slots` , which does require a restart. No more restarting the instance to add workers mid-incident.

> *Rule of thumb: if your biggest pain is long anti-wraparound passes on hundred-GB tables, PG18 pays for itself immediately. If your pain is slow vacuums starved of memory, PG17 is the one you need.*

11. The bloat is already there: reclaiming the space

Tuning autovacuum prevents future bloat. For what you already have, the relation has to be rewritten. Four options, from bluntest to most elegant:

VACUUM FULL — rewrites the whole table and returns the space to the operating system. It holds an ACCESS EXCLUSIVE lock for the entire operation: nobody reads or writes, not even a SELECT. It needs free disk equal to the resulting table plus its indexes. It is the right choice only if you have a maintenance window or the table is small.

```
VACUUM (FULL, ANALYZE, VERBOSE) orders;   -- locks the table completely
```

CLUSTER — the same, but physically reordering rows by an index. Same lock, same restriction. Useful if you also have a range access pattern that benefits from locality.

pg_repack — an external extension that does the work online. It builds a shadow copy, captures concurrent changes with triggers and swaps the relations at the end. It only holds the exclusive lock for moments at the start and the finish. In exchange: you have to install the extension and its client, it needs twice the space, and on very write-heavy tables the trigger queue can grow considerably.

```
# Repack one table without taking the service down
pg_repack --no-superuser-check -d mydb -t public.orders --jobs 4

# Indexes only (cheaper and often enough)
pg_repack -d mydb -t public.orders --only-indexes
```

pg_squeeze — a newer alternative that uses logical decoding instead of triggers to track concurrent changes. Less overhead on the primary and no long catalog locks, at the price of requiring `wal_level = logical`. It can be scheduled to act on its own when a table crosses a bloat threshold.

And a note on where this is heading: at PGConf.dev 2026 a native `REPACK CONCURRENTLY` command was previewed, targeted at PostgreSQL 19, which would bring online repacking into core with no extension. It is not something you can rely on in production yet, but it is worth knowing about before you build your own tooling around this.

Indexes first. Before you consider rewriting a 400 GB table, look at the indexes. `REINDEX INDEX CONCURRENTLY` has existed since PostgreSQL 12, does not block writes, and frequently recovers most of the lost space:

```
REINDEX INDEX CONCURRENTLY orders_customer_id_idx;
REINDEX TABLE CONCURRENTLY orders;   -- every index on the table

-- If something fails halfway you are left with invalid indexes: clean them up
SELECT indexrelid::regclass FROM pg_index WHERE NOT indisvalid;
```

12. Monitoring: script and alerts

This script pulls the three signals that matter (bloat, freeze age and horizon blockers) into a single report you can run from cron or a CI job:

```
"""PostgreSQL vacuum health report. Requires: pip install psycopg[binary]"""
import psycopg

BLOAT = """
SELECT relname,
       n_dead_tup,
       round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct,
       pg_size_pretty(pg_total_relation_size(relid)) AS size,
       last_autovacuum
FROM pg_stat_user_tables
WHERE n_dead_tup > %s
ORDER BY n_dead_tup DESC LIMIT 10;
"""

FREEZE = """
SELECT c.oid::regclass::text,
       age(c.relfrozenxid) AS xid_age
FROM pg_class c JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','m','t')
  AND n.nspname NOT IN ('pg_catalog','information_schema')
  AND age(c.relfrozenxid) > %s
ORDER BY 2 DESC LIMIT 10;
"""

HORIZON = """
SELECT 'backend'  AS kind, pid::text AS id, age(backend_xmin) AS xid_age
  FROM pg_stat_activity WHERE backend_xmin IS NOT NULL
UNION ALL
SELECT 'slot', slot_name, age(xmin)
  FROM pg_replication_slots WHERE xmin IS NOT NULL
UNION ALL
SELECT 'prepared', gid, age(transaction)
  FROM pg_prepared_xacts
ORDER BY xid_age DESC NULLS LAST LIMIT 5;
"""

def report(dsn: str, dead_min: int = 100_000, freeze_warn: int = 500_000_000) -> int:
    problems = 0
    with psycopg.connect(dsn, autocommit=True) as conn:
        print("== Tables with the most dead tuples ==")
        for name, dead, pct, size, last_av in conn.execute(BLOAT, (dead_min,)):
            flag = "  <-- INVESTIGATE" if last_av is None else ""
            print(f"{name:35} {dead:>12,} ({pct}%)  {size:>10}  last={last_av}{flag}")
            problems += 1

        print("\n== Relations approaching anti-wraparound ==")
        for name, age in conn.execute(FREEZE, (freeze_warn,)):
            print(f"{name:35} xid_age={age:>12,}")
            problems += 1

        print("\n== What is holding the cleanup horizon ==")
        for kind, ident, age in conn.execute(HORIZON):
            print(f"{kind:10} {ident:25} xid_age={age}")
    return problems

if __name__ == "__main__":
    import os, sys
    sys.exit(1 if report(os.environ["DATABASE_URL"]) else 0)
```

The horizon section is where the real value is: it turns "the vacuum is slow" into "the replication slot named analytics_replica has been holding cleanup back by 900 million XIDs", which is a sentence you can act on.

And the equivalent alerts with the Prometheus postgres_exporter:

```
groups:
  - name: postgres-vacuum
    rules:
      - alert: PostgresXIDWraparoundWarning
        expr: pg_database_xid_age > 500000000
        for: 30m
        labels: { severity: warning }
        annotations:
          summary: "High XID age on {{ $labels.datname }}"
          description: "age(datfrozenxid) = {{ $value }}. Check the horizon and
anti-wraparound."

      - alert: PostgresXIDWraparoundCritical
        expr: pg_database_xid_age > 1200000000
        for: 5m
        labels: { severity: critical }
        annotations:
          summary: "Wraparound risk on {{ $labels.datname }}"

      - alert: PostgresAutovacuumStalled
        expr: |
          (time() - pg_stat_user_tables_last_autovacuum > 86400)
          and (pg_stat_user_tables_n_dead_tup > 1000000)
        for: 1h
        labels: { severity: warning }
        annotations:
          summary: "No autovacuum in 24h with more than 1M dead tuples"

      - alert: PostgresReplicationSlotHoldingXmin
        expr: pg_replication_slot_xmin_age > 200000000
        for: 15m
        labels: { severity: warning }
        annotations:
          summary: "Slot {{ $labels.slot_name }} is blocking cleanup"
```

13. TOAST: the table that bloats where nobody looks

When a row grows past roughly 2 kB, PostgreSQL does not keep it whole in the main table. It compresses the large fields and, if they still do not fit, slices them into chunks and moves them into a side table called TOAST (The Oversized-Attribute Storage Technique). Any table with `text` , `jsonb` , `bytea` or array columns potentially has its own TOAST table, with its own heap, its own index and its own bloat.

The practical problem is visibility: `pg_stat_user_tables` does not roll TOAST dead tuples up into the parent table. You can have a table showing `n_dead_tup = 400` and sixty gigabytes of garbage in its TOAST without a single one of your usual diagnostic queries saying a word.

```
-- TOAST relations by size, with their own vacuum statistics
SELECT c.relname                                AS table_name,
       t.relname                                AS toast_name,
       pg_size_pretty(pg_relation_size(t.oid))  AS toast_size,
       s.n_live_tup,
       s.n_dead_tup,
       s.last_autovacuum
FROM pg_class c
JOIN pg_class t                ON t.oid = c.reltoastrelid
LEFT JOIN pg_stat_all_tables s ON s.relid = t.oid
```

```
WHERE c.relkind = 'r'
  AND c.reltoastrelid <> 0
ORDER BY pg_relation_size(t.oid) DESC
LIMIT 20;
```

The second detail that surprises a lot of people: **storage parameters on the parent table are not inherited by its TOAST table** . If you tuned `documents` with a scale factor of 0.02, its TOAST is still running on the global 0.2. TOAST settings are written with the `toast.` prefix and have to be set explicitly.

```
ALTER TABLE documents SET (
  autovacuum_vacuum_scale_factor       = 0.02,
  autovacuum_vacuum_threshold          = 2000,
  toast.autovacuum_vacuum_scale_factor = 0.02,
  toast.autovacuum_vacuum_threshold    = 2000,
  toast.autovacuum_vacuum_cost_delay   = 2
);
```

To measure real bloat on a specific TOAST relation, `pgstattuple` works exactly as it does anywhere else, pointed at the `pg_toast` schema:

```
SELECT * FROM pgstattuple('pg_toast.pg_toast_16584');
```

And now the design part, which is usually the real fix. An `UPDATE` only creates TOAST garbage if it touches a toasted column. If your table stores a 200 kB `jsonb` document and you bump a counter that lives inside that same JSON fifty times a day, every update rewrites the entire TOAST chain: two hundred kilobytes of writes to change one number. Pulling the small, hot fields out into their own columns — a `status` , a `counter` , an `updated_at` — removes the problem at the source far better than any autovacuum tuning.

14. Replicas: hot_standby_feedback, recovery conflicts and slots that never die

The list of horizon blockers falls short on one important point: in most installations that run replicas, bloat on the primary is decided by a different machine.

The mechanism is direct. With `hot_standby_feedback = on` , the replica reports back the oldest xmin its queries need, and the primary holds off its cleanup to respect it. With that option off, the primary cleans at its own pace and the replica cancels any query that needed the rows that just disappeared, with the familiar `canceling statement due to conflict with recovery` . There is no good option here: it is a trade between bloat on the primary and cancelled queries on the replica, and it is worth choosing deliberately rather than inheriting it.

Start by measuring what it is costing you right now:

```
-- How much each replica is holding back the primary horizon
SELECT pid, application_name, state,
       backend_xmin,
       age(backend_xmin) AS xmin_age
FROM pg_stat_replication
ORDER BY age(backend_xmin) DESC NULLS LAST;

-- Conflicts that already happened (run this on the replica)
SELECT datname, confl_snapshot, confl_lock, confl_bufferpin, confl_deadlock
```

```
FROM pg_stat_database_conflicts
WHERE confl_snapshot + confl_lock + confl_bufferpin + confl_deadlock > 0;
```

If `confl_snapshot` is climbing, your reads are dying because of cleanup on the primary, and the instinctive fix is to turn on `hot_standby_feedback` . Before you do, look at `max_standby_streaming_delay` : it is the middle dial almost nobody uses. Raising it to thirty seconds lets the replica pause WAL replay — and therefore lag a little — instead of cancelling queries, without touching the primary horizon at all.

Replication slots are the most dangerous case, because they do not require anything to exist on the other end. An orphaned slot from a replica decommissioned months ago holds the horizon and the WAL indefinitely:

```
SELECT slot_name, slot_type, active,
       age(xmin)          AS xmin_age,
       age(catalog_xmin)  AS catalog_xmin_age,
       inactive_since,
       pg_size_pretty(
         pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)
       ) AS wal_retained
FROM pg_replication_slots
ORDER BY age(xmin) DESC NULLS LAST;
```

There is a trap here that shows up constantly: `max_slot_wal_keep_size` protects your WAL disk, but **it does not protect your cleanup horizon** . A slot can hold the xmin back for days and only get invalidated once accumulated WAL crosses the limit. These are two different resources with two different thresholds, and only one of them has an automatic brake.

PostgreSQL 18 finally adds the missing piece: `idle_replication_slot_timeout` invalidates slots that have gone too long without a connection, measured from the slot's `inactive_since` value. It ships disabled, with a default of 0.

```
-- PostgreSQL 18+: invalidate abandoned slots
ALTER SYSTEM SET idle_replication_slot_timeout = '24h';
SELECT pg_reload_conf();
```

Two caveats before you rely on it. Invalidation is evaluated at checkpoint time, so it can land up to one `checkpoint_timeout` later than you expect, and it does not apply to slots being synced to a standby. If you are on 17 or earlier, the alternative is an alert on `inactive_since` plus a written rule: whoever decommissions a replica deletes its slot.

The setup that works best in practice splits the roles instead of looking for one setting that fits both. The high-availability replica runs with `hot_standby_feedback = off` and a short `max_standby_streaming_delay` , because its job is to be ready for a failover, not to serve reports. The analytics replica, where the twenty-minute queries live, runs with feedback on and an alert on the age of its `backend_xmin` , so the bloat cost is a measured decision rather than a discovery three months later.

15. Index bloat: bottom-up deletion, the expensive vacuum phase and when REINDEX is needed

It is common to find indexes more bloated than the table they index, and the reason is structural. When VACUUM removes entries from a B-tree index, pages left empty are marked reusable, but the tree is not

rebalanced and the space is not returned to the file. A sustained pattern of inserts and deletes leaves the index with half-full pages forever.

Since PostgreSQL 14 there is an important defence many people do not know they already have enabled: *bottom-up index deletion* . When an index page is about to split, PostgreSQL first tries to remove entries on that same page that are older versions of the same logical row. In workloads with many UPDATEs on non-indexed columns — the classic orders table with an index on customer and another on date — this avoids most page splits and keeps the index stable for months. It does not help with genuine deletes: only VACUUM frees those entries.

To find out whether a specific index is in bad shape, `pgstatindex` answers directly:

```
CREATE EXTENSION IF NOT EXISTS pgstattuple;

SELECT index_size, tree_level, avg_leaf_density, leaf_fragmentation
FROM pgstatindex('idx_orders_customer_created');
```

A freshly built index sits around an `avg_leaf_density` of 90. Sustained below 50 means there is real space to reclaim, and high `leaf_fragmentation` also means range scans will read more pages than they need — exactly the kind of degradation that never shows up in a log.

The online rebuild is `REINDEX INDEX CONCURRENTLY` . It is incomparably cheaper than rewriting the heap, but it is not free: it needs disk space for a full copy of the index, makes two passes over the table, and waits for transactions that started before it. If it fails halfway it leaves an invalid index with a `_ccnew` suffix that has to be cleaned up by hand — and that meanwhile keeps being maintained on every INSERT without any query ever using it.

```
REINDEX INDEX CONCURRENTLY idx_orders_customer_created;

-- Leftovers from failed attempts: write cost with no read benefit
SELECT indexrelid::regclass AS invalid_index,
       indrelid::regclass   AS table_name
FROM pg_index
WHERE NOT indisvalid;

DROP INDEX CONCURRENTLY idx_orders_customer_created_ccnew;
```

There is one operational detail worth knowing before you need it. The index phase is the most expensive part of a VACUUM, because it forces a full scan of every index from end to end. When you are racing a wraparound clock, you can skip it:

```
-- Emergency use only, to buy time during a wraparound
VACUUM (INDEX_CLEANUP OFF, VERBOSE) events;
```

This lets you freeze and advance `relfrozenxid` far faster, in exchange for leaving index entries pointing at dead line pointers. It is a rescue tool, not a configuration: as soon as the instance is out of danger, let a normal VACUUM do the full pass. Since PostgreSQL 14 the default is `AUTO` , which already skips index cleanup on its own when there are so few dead tuples that the scan is not worth it.

One last sizing note that connects back to the memory budget: the number of index passes depends on whether all dead tuple identifiers fit in `maintenance_work_mem` . With the TidStore structure introduced in PostgreSQL 17, that memory goes much further than it used to and the old effective 1 GB ceiling is gone. If you upgraded recently, it is worth re-measuring how many passes your VACUUM makes on large tables: going from five or six down to one without changing a single parameter is common.

16. Partitioned tables: vacuum runs per partition, parent statistics do not

In a partitioned table, autovacuum never processes the parent, because it holds no rows. It works partition by partition, each with its own dead tuple counter and its own trigger formula. That is exactly what you want, and it opens the door to a kind of tuning that is impossible on a monolithic table: treating the hot partition and the cold ones differently.

```
-- The current month partition takes all the writes
ALTER TABLE events_2026_08 SET (
  autovacuum_vacuum_scale_factor = 0.01,
  autovacuum_vacuum_threshold    = 5000,
  autovacuum_vacuum_cost_delay   = 2,
  fillfactor                     = 85
);

-- A closed partition: freeze it once and forget it
VACUUM (FREEZE, ANALYZE) events_2026_06;
```

Explicitly freezing a partition that no longer receives writes is one of the best uses of your time in a large database. Those pages are marked all-frozen in the visibility map and no future vacuum, not even an anti-wraparound one, will read them again. On an events table with twenty-four monthly partitions, this turns maintenance of twenty-four relations into maintenance of one.

What you should not do is disable autovacuum on cold partitions assuming they are already clean. `autovacuum_enabled = false` does not prevent anti-wraparound vacuum — PostgreSQL runs it anyway when the age demands it, thankfully — but it does prevent everything else, including ANALYZE. The usual outcome is that the partition eventually gets one enormous, unprepared anti-wraparound pass at the worst possible moment.

The other surprise is in the statistics. Autovacuum runs ANALYZE on the partitions, but **not on the parent table** . The parent's inherited statistics, which the planner uses whenever a query cannot prune partitions, are only refreshed if somebody runs ANALYZE by hand. On date-partitioned tables with queries that span several months, this shows up as plans that quietly get worse with no change in code or schema.

```
-- Schedule it weekly, for example with pg_cron
SELECT cron.schedule(
  'analyze-events-parent',
  '0 4 * * 0',
  'ANALYZE events'
);
```

And the point that saves the most over time: in a partitioned table, data retention should generate no vacuum work at all. Deleting thirty million rows with a `DELETE` creates thirty million dead tuples somebody has to clean up, plus the WAL, plus the bloat left behind. Detaching and dropping the expired month creates none.

```
ALTER TABLE events DETACH PARTITION events_2025_08 CONCURRENTLY;
DROP TABLE events_2025_08;
```

Put differently: if your largest table has a retention policy and still keeps growing through accumulated bloat, the problem is probably not autovacuum — it is that you are deleting rows where you should be

dropping partitions.

17. Autovacuum on managed services: what you can touch and what you cannot

If your PostgreSQL lives on RDS, Aurora, Cloud SQL or Azure Database, half of this guide applies unchanged and the other half moves house. What changes is access: there is no `postgresql.conf` , there are parameter groups, and some of the settings are managed by the provider itself.

On RDS and Aurora, `rds.adaptive_autovacuum` is on by default and does something sensible: as the transaction age on the instance approaches the danger thresholds, AWS automatically adjusts autovacuum parameters to be more aggressive. The official recommendation is to leave it on, and it is a good one. Starting with PostgreSQL 18 it goes one step further and scales `autovacuum_max_workers` dynamically as wraparound approaches — something that until now required a restart at the worst possible time.

That detail connects to a PostgreSQL 18 change worth understanding even if you are not on AWS. `autovacuum_max_workers` became changeable without a restart, and a new `autovacuum_worker_slots` parameter reserves the backend slots at startup and sets the hard ceiling. Setting `autovacuum_max_workers` above `autovacuum_worker_slots` has no effect: workers are drawn from that reserved pool.

```
-- PostgreSQL 18: ceiling fixed at startup, tuning done live
SHOW autovacuum_worker_slots;      -- changing this still needs a restart

ALTER SYSTEM SET autovacuum_max_workers = 8;
SELECT pg_reload_conf();
```

The cost of adding workers is memory, and this is where it is easy to hurt yourself. Each worker can use up to `autovacuum_work_mem` . On Aurora that value defaults to `GREATEST(DBInstanceClassMemory/32768, 65536)` , which on a large instance can land near 1 GB per worker. Going from three to sixteen workers at 1 GB each means reserving up to 16 GB of memory for maintenance, competing with cache and with queries. The right question is not how many workers you want, but how much total memory you are willing to give autovacuum; the worker count falls out of that.

Regardless of provider, two levers are always yours and are consistently underused. The first is per-table tuning, which depends on no parameter group and no restart:

```
ALTER TABLE orders SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 2000,
  autovacuum_vacuum_cost_delay   = 2,
  autovacuum_vacuum_cost_limit   = 2000,
  fillfactor                     = 85
);
```

The second is logging. `log_autovacuum_min_duration` is by far the best cost-to-value parameter in this entire guide. Setting it to 0 for a few days tells you which tables autovacuum visits, how long each pass takes, how many tuples it removes and — most useful of all — how many dead tuples it *could not* remove yet, which is the direct signal of a blocked horizon. Without that, any tuning afterwards is guesswork.

```
ALTER SYSTEM SET log_autovacuum_min_duration = 0;   -- a few days, then '250ms'
SELECT pg_reload_conf();
```

One operational warning: at 0 it logs every single pass, including catalog tables, so on an instance with thousands of relations the log volume is considerable. The sensible sequence is to leave it at 0 just long enough to build the map, then raise it to a threshold that only captures what genuinely takes time.

18. Worked example: from 41 GB to 9 GB with no maintenance window

It is worth walking the full cycle on one concrete case, because the order of the steps matters considerably more than any individual step.

The symptom: an `orders` table at 41 GB on disk, with read latency that had gone from 15 ms to 700 ms over three months, with no code changes and no real business growth. First measurement, using the cheap variant of pgstattuple:

```
SELECT * FROM pgstattuple_approx('orders');
-- table_len: 41 GB | approx_tuple_percent: 22 | dead_tuple_percent: 61
```

Nine gigabytes of live data inside forty-one. With that number in front of you, the instinct is to schedule a VACUUM FULL or a pg_repack for that same night. That is an ordering mistake: rewriting the table without fixing the cause puts you back in the same place in six weeks, and it burns the quiet window you are going to need afterwards. First you have to answer a specific question — why was autovacuum, which was enabled and visiting the table, removing nothing?

```
SELECT last_autovacuum, autovacuum_count, n_dead_tup, n_live_tup
FROM pg_stat_user_tables
WHERE relname = 'orders';
-- It ran hourly. And n_dead_tup never went down.
```

That is the unmistakable signature of a blocked horizon: VACUUM runs, does not fail, logs no errors, and cannot remove anything because someone in the system might still need to see those versions. Finding the culprit takes three queries:

```
-- 1. Oldest live transactions, including idle in transaction
SELECT pid, state,
       age(backend_xid)  AS xid_age,
       age(backend_xmin) AS xmin_age,
       now() - xact_start AS duration,
       left(query, 60)    AS query
FROM pg_stat_activity
WHERE backend_xmin IS NOT NULL OR backend_xid IS NOT NULL
ORDER BY greatest(age(backend_xid), age(backend_xmin)) DESC
LIMIT 5;

-- 2. Replication slots
SELECT slot_name, active, age(xmin) AS xmin_age, inactive_since
FROM pg_replication_slots
ORDER BY age(xmin) DESC NULLS LAST;

-- 3. Orphaned prepared transactions
SELECT gid, prepared, age(transaction) AS age
```

```
FROM pg_prepared_xacts
ORDER BY prepared;
```

The culprit was in the second one: a slot named `reporting_replica` , inactive for seventy-four days, belonging to a replica shut down during a migration whose slot nobody deleted. It was also holding 380 GB of WAL that had already forced two disk expansions without anyone connecting the two facts.

```
SELECT pg_drop_replication_slot('reporting_replica');
```

With the horizon released, the next VACUUM actually did something. It is worth raising maintenance memory in the session before launching it, so it does not need several passes over the indexes:

```
SET maintenance_work_mem = '2GB';
VACUUM (VERBOSE, ANALYZE) orders;
```

The table still measured 41 GB, because VACUUM does not shrink the file, but those 32 GB of space became reusable and latency dropped from 700 ms to around 200 ms with nothing else changed. What remained was disk space and cold pages polluting the cache, and for that a rewrite was justified — with pg_repack, without locking the table:

```
pg_repack -d production -t public.orders --jobs 4 --wait-timeout 30
```

Two warnings about pg_repack: it needs free space for a full copy of the table and its indexes, and it takes a brief exclusive lock at the start and another at the end. That is why you run it with `--wait-timeout` and at a quiet hour, even though technically it needs no downtime.

And the part that prevents a repeat, which is what actually closes the incident:

```
ALTER TABLE orders SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 2000,
  fillfactor                     = 90
);
```

Plus an alert on inactive slots with more than twenty-four hours without a connection — precisely the alert that would have turned three months of silent degradation into a five-minute notification. Final result: 9.4 GB on disk, read latency back at 14 ms, 380 GB of WAL released, and a runbook one page longer.

The lesson is not any of the commands above, it is the sequence: measure, find the blocker, remove it, let VACUUM do its job, reclaim space only if it is still needed, and finally tune so it does not happen again. Starting at the end — which is what almost everyone does — fixes the symptom for about a month.

19. The visibility map: VACUUM's second job

So far we have treated VACUUM as if its only mission were reclaiming space. It is not. Every pass also updates the **visibility map** , a tiny auxiliary file —two bits per data page— that travels alongside each table. The first bit says "every tuple on this page is visible to every transaction". The second says "every tuple on this page is frozen".

Those two bits carry two significant optimizations. The first benefits VACUUM itself: if a page is marked all-visible, the next pass skips it without reading it, which is why a vacuum over a quiet table finishes in

seconds even when the table occupies 200 GB. The second is far more visible to your users: the *index-only scan* .

An index-only scan answers a query by reading the index alone, without touching the table. The catch is that a PostgreSQL index stores no visibility information: it knows an entry exists pointing at a tuple, but not whether that tuple is still alive for your transaction. To skip the heap, the executor consults the visibility map. If the page is marked all-visible, it saves the access. If it is not, it has to go to the table and check the row, and that is a *heap fetch* .

The practical consequence is uncomfortable: **a perfect covering index performs like an ordinary index scan when the visibility map is stale** . And the visibility map goes stale precisely when autovacuum falls behind. So bloat does not just force you to read more pages: it also removes the shortcut that let you avoid reading them.

You measure it in EXPLAIN, on a line most people skip past:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT customer_id, created_at
FROM orders
WHERE created_at >= now() - interval '7 days';

--  Index Only Scan using orders_created_at_customer_idx on orders
--    (actual time=0.031..118.402 rows=412903 loops=1)
--    Index Cond: (created_at >= (now() - '7 days'::interval))
--    Heap Fetches: 398211          <-- the shortcut is not working
--    Buffers: shared hit=41238 read=390122
```

That `Heap Fetches` is the number that matters. If it approaches the number of rows returned, your index-only scan is an index scan in disguise. After vacuuming the table, the same query typically drops to `Heap Fetches: 0` and to a fraction of the buffers read.

To inspect the map itself there is the `pg_visibility` extension, shipped in contrib and free to install:

```
CREATE EXTENSION IF NOT EXISTS pg_visibility;

-- Quick summary: how many pages are marked visible and frozen
SELECT * FROM pg_visibility_map_summary('orders');
--  all_visible | all_frozen
-- -------------+------------
--       118204 |      92311

-- Actual coverage as a percentage
SELECT
  round(100.0 * count(*) FILTER (WHERE all_visible) / count(*), 1) AS pct_visible,
  round(100.0 * count(*) FILTER (WHERE all_frozen)  / count(*), 1) AS pct_frozen,
  count(*) AS total_pages
FROM pg_visibility('orders');
```

All-visible coverage below 80% on a large, mostly stable table is a clear sign that autovacuum is not passing through often enough. And there is one especially treacherous case: `pg_visibility` also lets you verify that the map is not *lying* . If the visibility map claims a page is all-visible when it is not, you get wrong results from index-only scans, with no error whatsoever. That happens after certain hardware failures or storage bugs:

```
-- Both should return zero rows. Any row means the map is corrupt.
SELECT * FROM pg_check_visible('orders');
SELECT * FROM pg_check_frozen('orders');

-- Rebuild: force VACUUM to read every page
VACUUM (DISABLE_PAGE_SKIPPING) orders;
```

`DISABLE_PAGE_SKIPPING` tells VACUUM to ignore the visibility map and read the whole table. It is expensive and not a routine operation, but it is the right tool when you suspect the map, or right after restoring from a physical backup you do not fully trust.

Three practical rules follow from all this:

- Monitor `Heap Fetches` on the queries that rely on covering indexes. It is the metric that translates "autovacuum is behind" into "this screen takes three times longer".
- After a bulk load, run an explicit `VACUUM` even if there is not a single dead tuple. Without it, none of the new pages carries the all-visible bit and index-only scans over that data simply will not happen.
- Do not confuse visibility map coverage with absence of bloat. A table can be 95% all-visible and still be badly bloated: they are different things maintained by the same process.

20. Append-only tables: the vacuum that for years never came

Go back to the trigger formula in section 3 and look for the term representing inserts. There is none. For most of PostgreSQL's history, autovacuum only looked at dead tuples: UPDATEs and DELETEs. A table that only receives INSERTs produces no dead tuples, so the counter never rose and autovacuum never visited it.

That describes a whole family of very common tables: events, audit logs, telemetry, warehouse fact tables, price histories. Enormous tables that only grow, and that automatic maintenance ignored completely.

There were three consequences, and none of them showed up until it was too late:

1. **No index-only scans.** No vacuum means no visibility map, and without a visibility map, analytical queries over those tables read the entire heap even when the index held every column they needed.
2. **A deferred freezing storm.** Since nothing triggered a vacuum, the table reached `autovacuum_freeze_max_age` untouched. Then, all at once, autovacuum launched an anti-wraparound pass over 400 GB nobody had ever vacuumed, usually at three in the afternoon on a Tuesday.
3. **Stale statistics.** Autoanalyze does fire on inserts, but it only refreshes statistics; it touches neither the visibility map nor freezing.

PostgreSQL 13 closed the gap with two new parameters that introduce a second threshold, running in parallel with the dead-tuple one:

```
insert_threshold = autovacuum_vacuum_insert_threshold
                 + autovacuum_vacuum_insert_scale_factor * n_rows

-- Defaults
autovacuum_vacuum_insert_threshold    = 1000
autovacuum_vacuum_insert_scale_factor = 0.2
```

Autovacuum now compares `n_ins_since_vacuum` against that threshold, and fires a vacuum if it is

exceeded even with zero dead tuples. Setting the threshold to `-1` disables insert-based triggering entirely for that table, which is exactly what you want in one narrow case: a staging table that is filled and truncated every hour, where a vacuum is pure waste.

The default scale factor of 0.2 is still far too lazy for large tables: on a 500-million-row table it demands 100 million inserts before it lifts a finger. For large append-only tables, the sensible move is a fixed absolute threshold with the proportional factor turned off:

```
-- Large append-only events table, queried by date ranges
ALTER TABLE events SET (
  autovacuum_vacuum_insert_threshold    = 500000,
  autovacuum_vacuum_insert_scale_factor = 0.0,
  autovacuum_analyze_scale_factor       = 0.01
);

-- Check the current state
SELECT relname,
       n_ins_since_vacuum,
       n_live_tup,
       last_vacuum,
       last_autovacuum
FROM pg_stat_user_tables
WHERE relname = 'events';
```

With `scale_factor = 0.0` and a threshold of 500,000, the table gets a vacuum every half million new rows regardless of its size. That vacuum is cheap —it only reads the new pages, thanks to the very visibility map it is maintaining— and it does three things at once: it marks pages visible, it freezes them while they are still warm in cache, and it prevents the future anti-wraparound storm.

If the table is partitioned by date, and it should be, these parameters belong **on the partitions** , not on the parent. The parent has no storage of its own and autovacuum works partition by partition. A pattern that works well is aggressive settings on the active partition and a definitive freeze on the closed ones:

```
-- Current month partition: takes all the writes
ALTER TABLE events_2026_08 SET (
  autovacuum_vacuum_insert_threshold    = 200000,
  autovacuum_vacuum_insert_scale_factor = 0.0
);

-- Closed partitions: freeze once and forget about them
VACUUM (FREEZE, ANALYZE) events_2026_07;
```

There is also a loading trick that avoids creating the work in the first place. If you run `COPY` inside the same transaction that creates or truncates the table, the `FREEZE` option writes the rows already frozen and the pages already marked visible:

```
BEGIN;
TRUNCATE events_staging;
COPY events_staging FROM '/data/events.csv' WITH (FORMAT csv, FREEZE);
COMMIT;
-- Pages are born all-visible and all-frozen: zero pending work
-- for autovacuum, and index-only scans available immediately.
```

The restriction is strict: `FREEZE` requires that the table was created or truncated in the same transaction. Outside that case, the alternative is a `VACUUM (FREEZE, ANALYZE)` right after the load, while the data is still in the operating system cache and reading it is nearly free.

One last note on versions. If you administer PostgreSQL 12 or older —and there are more of those around than you would think— you have none of these parameters. Your only defense is a cron-scheduled `VACUUM` over the append-only tables. Not elegant, but it prevents the storm.

21. Catalog bloat: pg_attribute, temp tables and DDL

The system catalog tables are ordinary tables. They have rows, they obey MVCC, they accumulate dead tuples and they bloat exactly like yours. The difference is that almost nobody looks at them, and that when they bloat the symptoms look nothing like those of a business table.

The typical symptom is this: queries are fine, but *opening a connection* takes 300 ms; planning time for trivial queries climbs into the tens of milliseconds; a `\d table` in psql sits there thinking. All of that happens because every planning cycle consults the catalog, and if `pg_attribute` is 4 GB of which 200 MB is live data, every lookup pays the price.

The causes are always the same three:

- **Temporary tables created and destroyed nonstop.** Every `CREATE TEMP TABLE` inserts rows into `pg_class` , `pg_attribute` , `pg_type` and `pg_depend` . Every `DROP` kills them. An application that creates one temp table per request generates thousands of dead catalog tuples per second.
- **Frequent DDL by design.** Multi-tenant architectures with a schema or a table set per customer, systems that create partitions on the fly, test suites that run migrations in a loop.
- **Tools that use temp tables internally.** Some ORMs, ETL processes and extensions do it without you writing a single line of DDL.

Measuring is identical to any other table, except the view is `pg_stat_sys_tables` :

```
SELECT relname,
       n_live_tup,
       n_dead_tup,
       round(100.0 * n_dead_tup / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS pct_dead,
       pg_size_pretty(pg_total_relation_size(relid)) AS size,
       last_autovacuum
FROM pg_stat_sys_tables
WHERE schemaname = 'pg_catalog'
  AND n_dead_tup > 1000
ORDER BY n_dead_tup DESC
LIMIT 10;

-- The usual suspects, in this order:
-- pg_attribute, pg_class, pg_type, pg_depend, pg_shdepend, pg_statistic
```

Autovacuum does process the catalog, with the same thresholds and subject to the same blockers from section 9. If you have a transaction open for six hours, the catalog does not get cleaned either. But there is a wrinkle of its own: `pg_attribute` and `pg_class` hold *many rows per object* , so a DDL spike takes them from 20,000 to 2 million rows in no time and the 20% scale factor becomes an enormous threshold at exactly the wrong moment.

Per-table parameters work on the catalog too, and that is the structural fix:

```
-- Requires superuser
ALTER TABLE pg_catalog.pg_attribute SET (
  autovacuum_vacuum_scale_factor = 0.02,
```

```
  autovacuum_vacuum_threshold    = 1000
);
ALTER TABLE pg_catalog.pg_class SET (
  autovacuum_vacuum_scale_factor = 0.02,
  autovacuum_vacuum_threshold    = 1000
);

-- One-off cleanup if it is already bloated
VACUUM (ANALYZE, VERBOSE) pg_catalog.pg_attribute;
```

If the catalog is already badly bloated and a normal vacuum reclaims nothing, the only way out is `VACUUM FULL pg_catalog.pg_attribute` , and here it pays to be blunt: that takes an ACCESS EXCLUSIVE lock on a table that *every* query needs. It blocks the entire database for as long as it runs. This is not online maintenance; it is a planned window, and neither `pg_repack` nor `pg_squeeze` can help you because they do not operate on catalogs.

Which is why the real work is preventive. The alternative to creating and dropping temp tables is reusing them:

```
-- Anti-pattern: a brand new catalog entry per request
CREATE TEMP TABLE results_tmp AS SELECT ...;
-- ... use it ...
DROP TABLE results_tmp;

-- Better: create once per session, empty it each transaction
CREATE TEMP TABLE IF NOT EXISTS results_tmp (
  id bigint,
  total numeric
) ON COMMIT DELETE ROWS;

-- Better still: skip the temp table entirely
WITH results AS (SELECT ...)
SELECT * FROM results JOIN ...;
```

`ON COMMIT DELETE ROWS` empties the table at the end of each transaction while keeping its definition, so the catalog is touched once per session instead of once per request. With a pooler in transaction mode and long-lived sessions, the difference is several orders of magnitude.

And a warning about the storage of temporary tables themselves: they live in their session's temp schema and **autovacuum never touches them** . Only the owning session can vacuum them. If you have a long-running process that fills and updates a temp table for hours, that temp table bloats unchecked and nobody is going to clean it up for you. In that case, an explicit `VACUUM` from inside the session is mandatory.

22. Parallel VACUUM: what actually runs in parallel

Since PostgreSQL 13, `VACUUM` accepts a `PARALLEL` option. It is worth understanding exactly what it distributes across workers, because the name promises more than it delivers.

A vacuum has three phases: scan the heap and record the identifiers of dead tuples, clean those entries out of the indexes, and return to the heap to free the space. **Only the index phase is parallelized** , and the split is one worker per index. The heap scan, which on tables with few indexes is most of the work, remains single-process.

From that follow the conditions for it to be worth anything:

- The table needs **several indexes** . With only one there is nothing to distribute.
- Each index must exceed `min_parallel_index_scan_size` (512 kB by default) to be a candidate.
- The actual worker count is capped by `max_parallel_maintenance_workers` , which defaults to 2.
- And the most important one: **autovacuum never uses parallelism** before PostgreSQL 19. Whatever you configure, autovacuum workers process a table and its indexes serially.

That last line explains a pattern you see a lot in production: a wide table with twelve indexes monopolizes an autovacuum worker for forty minutes while the rest of the database waits its turn. The workaround so far has been to pull that table out of the automatic circuit and give it a manual vacuum during the quiet window:

```
-- Check the current ceiling
SHOW max_parallel_maintenance_workers;  -- 2 by default

-- Raise it for this maintenance session only
SET max_parallel_maintenance_workers = 6;

-- Manual vacuum with explicit parallelism and a trace of what it does
VACUUM (PARALLEL 4, VERBOSE, ANALYZE) orders;

-- The output tells you how many workers actually started:
-- INFO:  launched 3 parallel vacuum workers for index vacuuming
--        (planned: 4)
```

Note the "planned: 4" against "launched: 3". These workers come from the same pool as parallel queries, bounded by `max_worker_processes` and `max_parallel_workers` . If the server is busy, you ask for four and you get what is available. It never fails because of this: it just runs slower.

There is a second kind of parallelism that gets confused with this one and is equally useful: `vacuumdb --jobs` . It does not split the work of *one* table; it opens several connections and vacuums *several tables at once* . For scheduled maintenance of an entire database, that is what you want:

```
# Eight tables in parallel, each with its indexes in parallel
vacuumdb --all --analyze --jobs 8 --parallel 4 --verbose

# Only the most urgent tables by freeze age
vacuumdb --all --freeze --jobs 4 --min-xid-age 500000000
```

And an operational warning: each parallel worker consumes its own share of `maintenance_work_mem` . With `maintenance_work_mem = 2GB` and four workers you may be committing a lot more memory than you had in mind. On a tightly sized server, lower the memory before raising the parallelism, not the other way around.

23. Major upgrades: pg_upgrade, the freeze storm and statistics

A major PostgreSQL upgrade interacts with everything above, and the result surprises a lot of people on the first Monday after the change.

The good news first: `pg_upgrade` preserves `relfrozenxid` . Nothing has to be re-frozen, and the wraparound clock stays where it was. What was *not* preserved, through PostgreSQL 17, were the planner statistics. The database started with zero information about the distribution of your data and the planner made decisions blind: seq scans where perfect indexes existed, nested loops over millions of rows, timeouts on endpoints that were fine on Friday.

PostgreSQL 18 fixes the main part: `pg_upgrade` now transfers most optimizer statistics. But two gaps remain, and the second one lands squarely on this article's territory:

- **Extended statistics** created with `CREATE STATISTICS` are not transferred. If you rely on them for cross-column correlations, they must be regenerated.
- **Cumulative statistics** —the `pg_stat_user_tables` ones: `n_dead_tup` , `n_ins_since_vacuum` , `n_mod_since_analyze` — are always lost. And those are exactly the counters that trigger autovacuum.

That last point produces a curious effect: after the upgrade, autovacuum believes no table has any dead tuples. Every counter is at zero. A table that spent a month accumulating bloat starts from scratch and will not fire until it accumulates its full threshold all over again, even though the real bloat is still sitting there. The database looks calm, and what is actually happening is that maintenance is blind.

The recommended sequence after a `pg_upgrade` to 18, in this order:

```
# 1. Minimal statistics for whatever is missing, in stages, fast.
#    --missing-stats-only avoids clobbering the ones that did transfer.
vacuumdb --all --analyze-in-stages --missing-stats-only --jobs 8

# 2. Now a full ANALYZE, which also repopulates the
#    cumulative counters that drive autovacuum.
vacuumdb --all --analyze-only --jobs 8

# 3. Regenerate the extended statistics that were lost
psql -c "SELECT format('ANALYZE %I.%I;', n.nspname, c.relname)
         FROM pg_statistic_ext e
         JOIN pg_class c ON c.oid = e.stxrelid
         JOIN pg_namespace n ON n.oid = c.relnamespace;"

# 4. And a real vacuum pass, unhurried, outside peak hours
vacuumdb --all --jobs 4 --verbose
```

Step 1 is what prevents the immediate disaster: `--analyze-in-stages` makes three passes with increasing statistics targets, so the planner has *something* to work with within minutes instead of waiting hours for a full ANALYZE. With `--missing-stats-only` , on PostgreSQL 18, it additionally avoids destroying the good statistics that did survive the jump.

If you are coming from PostgreSQL 17 or earlier, step 1 drops `--missing-stats-only` —the option does not exist there— and you have to assume everything gets analyzed. Budget window time for it and do not open traffic until the first stage has finished.

24. What is coming in PostgreSQL 19: autovacuum that prioritizes, and REPACK in core

PostgreSQL 19 released its first beta on June 4, 2026, with the stable version expected in the second half of the year. None of this should touch your production yet, but two of its changes land directly on what this guide covers and it is worth knowing where the ground is moving.

Autovacuum learns triage

Until now, an autovacuum worker built its list of candidate tables and worked through them roughly in the order they appear in `pg_class` . Very democratic and fairly absurd: a table one transaction away from a

wraparound shutdown got the same treatment as one that had just crossed an ANALYZE threshold.

PostgreSQL 19 introduces a scoring system. Before deciding where to start, it computes five scores per table —transaction ID age, multixact ID age, dead tuples waiting to be reclaimed, freshly inserted tuples, and changes since the last ANALYZE— and keeps the highest as that table's priority. Each component is essentially a ratio: how far past its trigger threshold the table has drifted.

The best part is that it stops being a black box. There is a new view:

```
SELECT relname,
       ceil(score)               AS score,
       ceil(vacuum_score)        AS vacuum_score,
       ceil(vacuum_insert_score) AS insert_score,
       ceil(analyze_score)       AS analyze_score,
       ceil(xid_score)           AS xid_score,
       for_wraparound
FROM pg_stat_autovacuum_scores
ORDER BY score DESC
LIMIT 20;

--    relname   | score | vacuum_score | insert_score | analyze_score
-- -------------+-------+--------------+--------------+---------------
--  append_log  | 10000 |            0 |          500 |         10000
--  churn_queue |  6800 |         2800 |          200 |          6800
```

For years, diagnosing why autovacuum was doing *that* instead of what you wanted meant reading source code and making educated guesses. Now it is in a view you can put on a dashboard.

And you can tip the scales. Five of the six new parameters are weights that multiply each component before the maximum is chosen:

```
-- All default to 1.0: every concern weighs the same
autovacuum_freeze_score_weight           = 1.0
autovacuum_multixact_freeze_score_weight = 1.0
autovacuum_vacuum_score_weight           = 1.0
autovacuum_vacuum_insert_score_weight    = 1.0
autovacuum_analyze_score_weight          = 1.0

-- Example: reclaiming space matters twice as much, refreshing
-- statistics half as much. These are sighup parameters:
-- a reload is enough.
ALTER SYSTEM SET autovacuum_vacuum_score_weight  = 2.0;
ALTER SYSTEM SET autovacuum_analyze_score_weight = 0.5;
SELECT pg_reload_conf();
```

Two subtleties deserve attention. First: setting all five weights to `0.0` restores the pre-19 behavior, the good old catalog order. It is a documented escape hatch. Second: the two freeze weights do not behave like the others. Raising them above 1.0 does not merely multiply the score, it also *divides the thresholds* by the weight. An `autovacuum_freeze_score_weight = 2.0` means freeze pressure starts counting as urgent at half of `autovacuum_freeze_max_age` . Powerful, and easy to overdo.

The sixth parameter is of a different nature: `autovacuum_max_parallel_workers` finally lets an autovacuum worker recruit helpers to process indexes in parallel, closing the gap from section 22. You have to opt in explicitly, since the default is 0:

```
ALTER SYSTEM SET autovacuum_max_parallel_workers = 4;
SELECT pg_reload_conf();
```

REPACK: core's answer to the bloat that is already there

Section 11 of this guide exists because reclaiming space from a bloated table had two bad answers: `VACUUM FULL`, which takes an ACCESS EXCLUSIVE lock and blocks everything, or an external extension like `pg_repack` or `pg_squeeze`, with its own installation and its own set of sharp edges.

PostgreSQL 19 brings the solution into core with a new `REPACK` command that absorbs what `VACUUM FULL` and `CLUSTER` used to do. The interesting variant is the concurrent one, built on the logical decoding machinery that `pg_squeeze` pioneered:

```
-- Classic rewrite, exclusive lock (equivalent to VACUUM FULL)
REPACK orders;

-- Rewrite without an exclusive lock: the table keeps taking writes
REPACK CONCURRENTLY orders;

-- Physically reordering by an index (equivalent to CLUSTER)
REPACK orders USING INDEX orders_created_at_idx;

-- Live progress
SELECT * FROM pg_stat_progress_repack;
```

Internally, concurrent mode copies the table and replays, through logical decoding, the changes that happen during the copy, which is why a new parameter appears: `max_repack_replication_slots`, defaulting to 5. Both old commands still work in 19, but future development goes to `REPACK`.

The practical reading for anyone planning ahead: if today you keep `pg_repack` installed solely to deflate tables now and then, that dependency has an expiry date. Do not migrate anything yet —this is beta— but write it into the plan for your next major upgrade.

25. Validating your configuration before production does it for you

Everything above is a set of decisions, and a configuration decision nobody has measured is a hypothesis. The good news is that bloat is one of the few database problems that reproduces faithfully on a laptop, because it does not depend on the absolute volume of data but on the ratio between the write rate and the vacuum's capacity.

The minimal test bench has three pieces: a table with the write pattern that worries you, a sustained load, and an honest measurement of the steady state.

```
-- setup.sql: a table that mimics a work queue
DROP TABLE IF EXISTS bench_queue;
CREATE TABLE bench_queue (
  id       bigserial PRIMARY KEY,
  state    int  NOT NULL DEFAULT 0,
  claimed_by text,
  updated_at timestamptz NOT NULL DEFAULT now(),
  payload  jsonb NOT NULL
);
CREATE INDEX ON bench_queue (state, id);
CREATE INDEX ON bench_queue (updated_at);
```

```
INSERT INTO bench_queue (payload)
SELECT jsonb_build_object('n', g)
FROM generate_series(1, 2000000) g;

VACUUM (FREEZE, ANALYZE) bench_queue;
```

The load script reproduces the pattern that bloats queues: claim rows, update them, delete them. It is a file for `pgbench` :

```
-- churn.sql
\set id random(1, 2000000)
BEGIN;
UPDATE bench_queue
   SET state = 1, claimed_by = 'w' || :client_id, updated_at = now()
 WHERE id = :id;
UPDATE bench_queue
   SET state = 2, updated_at = now()
 WHERE id = :id;
DELETE FROM bench_queue WHERE id = :id AND random() < 0.05;
END;
```

And the run. The methodological key is running it twice: once with your current configuration and once with the proposed one, changing **exactly one thing** between them:

```
# Baseline: 20 minutes of sustained load, 16 clients
pgbench -n -f churn.sql -c 16 -j 4 -T 1200 -P 60 bench

# During the run, from another session, sample every 30 seconds
watch -n 30 "psql -qAt -d bench -c \"
  SELECT n_dead_tup,
         pg_size_pretty(pg_total_relation_size('bench_queue')),
         last_autovacuum
  FROM pg_stat_user_tables WHERE relname = 'bench_queue';\""
```

What you are looking for is not a pretty number but a **curve shape** . If `n_dead_tup` rises, autovacuum kicks in, it falls, and the cycle repeats around a stable average, your configuration keeps up with that write rate. If each cycle's average is higher than the last, autovacuum is losing ground, and in production that translates into bloat that grows without bound. It is the same signal as a queue backlog that never drains.

The final measurement, once the load is done, comes from `pgstattuple` , which you know from section 4:

```
SELECT
  round(free_percent::numeric, 1)       AS pct_free,
  round(dead_tuple_percent::numeric, 1) AS pct_dead,
  pg_size_pretty(table_len)             AS size
FROM pgstattuple('bench_queue');

-- And the price you paid to keep it that way:
SELECT vacuum_count, autovacuum_count,
       n_tup_upd, n_tup_hot_upd,
       round(100.0 * n_tup_hot_upd / NULLIF(n_tup_upd,0), 1) AS pct_hot
FROM pg_stat_user_tables WHERE relname = 'bench_queue';
```

That `pct_hot` at the end is the metric that closes the loop with section 7. If it comes out low, try lowering the table's `fillfactor` to 80 and repeat the whole experiment. You will watch HOT updates rise and, with

them, both the vacuum's workload and index bloat fall. It is the kind of improvement you would never have dared justify in production without a test bench.

Two things this bench does *not* measure, and it is worth keeping them in mind: the impact of your real storage latency —a vacuum on local NVMe and one on a network volume with provisioned IOPS have nothing in common— and the interference with the rest of the database workload. For both, the right place is a staging environment with the same class of disk as production and a replay of your real traffic. But the shape of the curve, which is the decision that matters, already shows up on the laptop.

26. Eight mistakes that keep repeating

1. **Turning autovacuum off** "because it eats I/O". You are trading a continuous, predictable cost for a six-hour anti-wraparound at the worst possible moment. If autovacuum is in the way, the problem is the tuning, not autovacuum.
2. **Tuning only globally.** A 0.01 scale factor in postgresql.conf punishes two hundred small tables to fix three big ones.
3. **Raising** `autovacuum_max_workers` **without raising** `vacuum_cost_limit` **.** The budget is shared: more workers on the same budget means each one is slower.
4. **Treating VACUUM FULL as "a stronger VACUUM".** It locks the entire table. Running it at 10am on the orders table is a self-inflicted outage.
5. **Ignoring index bloat.** Indexes are routinely more bloated than the table, and REINDEX CONCURRENTLY is incomparably cheaper than rewriting the heap.
6. **Alerting only on** `n_dead_tup` **.** It catches neither the freezing debt nor the held horizon, which are the two ways this turns into a serious incident.
7. **Leaving replication slots from decommissioned replicas.** They hold WAL and the horizon forever. Audit them every time you tear down a replica.
8. **Deleting millions of rows with no follow-up plan.** A mass DELETE leaves the space inside the file. If you are not going to reinsert a similar volume, plan the repack; if it is a recurring date-based purge, use partitioning and `DROP PARTITION` , which is a metadata operation and produces no dead tuples at all.

27. Production checklist

- Live alerts on `age(datfrozenxid)` at 500M (warning) and 1.2B (critical).
- Alert on tables with more than 1M dead tuples and no autovacuum in 24 hours.
- Alert on replication slots with an old xmin or growing retained WAL.
- `idle_in_transaction_session_timeout` and (on PG17+) `transaction_timeout` configured.
- Per-table overrides on every relation above a million rows with frequent writes.
- `vacuum_cost_limit` matched to your actual storage, not the default of 200.
- `maintenance_work_mem` and `autovacuum_work_mem` at 1 GB or more when large tables exist.
- Quarterly review with `pgstattuple_approx` over the twenty largest tables.
- Replication slot audit in the replica decommissioning runbook.
- Date partitioning on any table with a retention policy, instead of a periodic DELETE.

- A written wraparound runbook, with the diagnostic queries, rehearsed at least once.

FAQ

Does VACUUM lock my table? A normal VACUUM takes a SHARE UPDATE EXCLUSIVE lock: it coexists happily with SELECT, INSERT, UPDATE and DELETE. It only conflicts with another VACUUM, with ANALYZE, with CREATE INDEX and with ALTER TABLE. VACUUM FULL is an entirely different thing and does block everything.

I ran VACUUM and the table is the same size. Is it broken? No, it is working as designed. VACUUM recycles space inside the file, it does not hand it back to the operating system. If disk space is your problem, you need pg_repack, pg_squeeze or VACUUM FULL.

How much bloat is acceptable? Under 20% free space is normal and even desirable, because that room is what feeds HOT updates. Between 20% and 40% is worth investigating. Sustained above 40%, something is wrong with the tuning or a horizon is held.

Can I cancel an autovacuum that is getting in the way? A normal autovacuum cancels itself when a query needs an incompatible lock. You can kill it with `pg_terminate_backend()` , but it will come back. And if `pg_stat_activity` shows the query saying "to prevent wraparound", **do not kill it** : that is the emergency protection, and killing it repeatedly is precisely the road to the database shutting down.

Is a cron-scheduled VACUUM worth it if I already have autovacuum? Rarely, and it is usually a patch over bad tuning. The legitimate exception is a manual `FREEZE` vacuum after a bulk load, or ahead of a critical window where you do not want an anti-wraparound to fire.

What about managed services (RDS, Cloud SQL, Aurora)? Everything above applies, with two caveats: per-table ALTER TABLE overrides always work, whereas some global GUCs are restricted or managed through parameter groups. In those environments, per-relation tuning is not just good practice, it is usually your only lever.