PuigFlows

Particionado de Tablas en PostgreSQL

Range, List, Hash y Retención sin Downtime — Edición ampliada
Table Partitioning in PostgreSQL: Range, List, Hash and Zero-Downtime Retention — Expanded edition

Guía Premium · Premium Guide · puigflows.com · Agosto 2026

Particionado de Tablas en PostgreSQL: Range, List, Hash y Retención sin Downtime

Guía · Intermedio · Proyectos Reales · ~35 min de lectura · PuigFlows

Cuando una tabla supera los cientos de gigabytes, los problemas dejan de ser teóricos: los índices ya no caben en memoria, autovacuum tarda horas en recorrerla y borrar datos antiguos con DELETE bloquea, genera bloat y castiga la replicación. El particionado declarativo de PostgreSQL divide esa tabla en piezas físicas más pequeñas —las particiones— manteniendo una sola tabla lógica para tu aplicación. Bien hecho, convierte el borrado de un año de datos en soltar un archivo; mal hecho, añade complejidad sin mejorar nada. Esta guía cubre cómo hacerlo bien.

Por qué particionar (y qué no soluciona)

El particionado aporta tres beneficios concretos. Primero, retención barata: eliminar una partición completa con DROP TABLE o DETACH PARTITION es una operación de metadatos casi instantánea, frente a un DELETE masivo que reescribe páginas, dispara autovacuum y llena el WAL. Segundo, partition pruning: si tus consultas filtran por la clave de partición, el planificador ignora las particiones irrelevantes y escanea una fracción de los datos. Tercero, mantenimiento local: vacuum, análisis y reindexado operan partición a partición, con locks y tiempos mucho menores.

Lo que el particionado no hace: no acelera consultas que no filtran por la clave de partición (de hecho las puede empeorar, porque hay que abrir todas las particiones), no sustituye a un buen índice y no es sharding: todo sigue viviendo en el mismo servidor. Como regla práctica, empieza a considerarlo cuando una tabla se acerca a los 100 GB o cuando la retención de datos por tiempo es un requisito real. Para una tabla de 5 GB, un índice parcial bien elegido suele rendir más que cualquier esquema de particiones.

Range, List y Hash: elige el método correcto

RANGE es el método más común: divide por intervalos de una columna ordenable, casi siempre una fecha. Es el ajuste natural para eventos, logs, métricas y cualquier dato con retención temporal.

```
CREATE TABLE events (
  id bigint GENERATED ALWAYS AS IDENTITY,
  tenant_id uuid NOT NULL,
  event_type text NOT NULL,
  payload jsonb NOT NULL DEFAULT '{}',
  created_at timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

CREATE TABLE events_2026_07 PARTITION OF events
  FOR VALUES FROM ('2026-07-01') TO ('2026-08-01');

CREATE TABLE events_2026_08 PARTITION OF events
  FOR VALUES FROM ('2026-08-01') TO ('2026-09-01');
```

Fíjate en dos detalles. El límite superior es exclusivo: la partición de julio acepta hasta 2026-07-31 23:59:59.999999 y el 1 de agosto cae en la siguiente, sin huecos ni solapes. Y la clave primaria incluye created_at: en una tabla particionada, toda restricción única debe contener la clave de partición (enseguida veremos por qué).

LIST divide por valores discretos: región, país, tipo de cliente. Útil cuando quieres aislar subconjuntos con ciclos de vida distintos.

```
CREATE TABLE orders (
  id bigint NOT NULL,
  region text NOT NULL,
  total numeric(12,2),
  created_at timestamptz NOT NULL DEFAULT now()
) PARTITION BY LIST (region);

CREATE TABLE orders_eu PARTITION OF orders FOR VALUES IN ('es', 'fr', 'de', 'it');
CREATE TABLE orders_latam PARTITION OF orders FOR VALUES IN ('mx', 'ar', 'co', 'cl');
```

HASH reparte filas de forma uniforme según el hash de una columna. No permite pruning por rangos ni retención por tiempo; su caso de uso es repartir una tabla enorme sin un criterio temporal claro, por ejemplo para paralelizar el mantenimiento.

```
CREATE TABLE sessions (
  session_id uuid NOT NULL,
  data jsonb
) PARTITION BY HASH (session_id);

CREATE TABLE sessions_p0 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE sessions_p1 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE sessions_p2 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE sessions_p3 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

Si lo necesitas, puedes combinar métodos con subparticiones (por ejemplo, RANGE mensual y dentro HASH por tenant), pero cada nivel multiplica el número de particiones. Doce particiones mensuales con cuatro subparticiones hash son 48 tablas; 365 particiones diarias son un problema de planificación. Mantén el total en unos pocos cientos como máximo.

Partition pruning: verifica que funciona

El pruning es lo que hace rentable todo lo demás: el planificador demuestra que una partición no puede contener filas que cumplan el WHERE y ni siquiera la abre. Pero solo ocurre si el filtro actúa directamente sobre la clave de partición, sin funciones que la envuelvan. Compara:

```
-- [SI] Pruning: el rango se compara directo contra created_at
EXPLAIN (COSTS OFF)
SELECT count(*) FROM events
WHERE created_at >= '2026-07-01' AND created_at < '2026-08-01';

-- Aggregate
--   ->  Seq Scan on events_2026_07 events
--       Filter: ...

-- [NO] Sin pruning: date_trunc() envuelve la columna
EXPLAIN (COSTS OFF)
SELECT count(*) FROM events
WHERE date_trunc('month', created_at) = '2026-07-01';
-- escanea TODAS las particiones
```

El pruning ocurre en tres momentos: al planificar (valores literales), al inicializar la ejecución (parámetros como now() - interval '7 days', visible como Subplans Removed en el EXPLAIN) y durante la ejecución (valores que llegan de un join). La consecuencia práctica: usar now() o parámetros preparados no impide el pruning, pero envolver la columna en funciones sí. Acostúmbrate a verificar con EXPLAIN que las consultas críticas solo tocan las particiones esperadas; es la prueba de fuego de todo el diseño.

Índices, claves primarias y únicas

Desde PostgreSQL 11, un índice creado sobre la tabla padre se propaga automáticamente a todas las particiones, presentes y futuras:

```
CREATE INDEX idx_events_tenant_created
ON events (tenant_id, created_at DESC);
```

La restricción importante son las claves únicas: PostgreSQL no tiene índices únicos globales entre particiones, así que toda PRIMARY KEY o UNIQUE debe incluir la clave de partición. Por eso nuestra tabla usa PRIMARY KEY (id, created_at). Si necesitas unicidad global real sobre otra columna (por ejemplo, un external_id), tienes que garantizarla fuera: una tabla auxiliar no particionada con la restricción única, o unicidad garantizada por el productor del dato.

Un truco operativo para tablas grandes: crea el índice en el padre con ON ONLY, construye los índices por partición con CREATE INDEX CONCURRENTLY y termina con ALTER INDEX ... ATTACH PARTITION. Así evitas el lock prolongado de construir todos los índices de golpe.

Automatización con pg_partman y pg_cron

Crear particiones a mano funciona hasta la primera vez que alguien olvida crear la del mes siguiente y las escrituras empiezan a fallar (o a caer en la partición DEFAULT, que es peor). pg_partman automatiza creación y retención:

```
CREATE EXTENSION pg_partman;

SELECT partman.create_parent(
  p_parent_table := 'public.events',
  p_control := 'created_at',
  p_interval := '1 month',
  p_premake := 4  -- mantén 4 particiones futuras creadas
);

-- Retención: suelta particiones con datos de más de 12 meses
UPDATE partman.part_config
SET retention = '12 months',
    retention_keep_table = false
WHERE parent_table = 'public.events';
```

Y con pg_cron programas el mantenimiento dentro de la propia base de datos:

```
CREATE EXTENSION pg_cron;

SELECT cron.schedule(
  'partman-maintenance',
  '@hourly',
  $$CALL partman.run_maintenance_proc()$$
);
```

Cada hora, pg_partman crea las particiones futuras que falten y aplica la política de retención. En servicios gestionados (RDS, Cloud SQL, Azure) ambas extensiones suelen estar disponibles; si no, el mismo mantenimiento puede lanzarse desde un cron externo o un scheduler de tu aplicación.

Retención: DETACH y DROP en lugar de DELETE

La forma correcta de borrar datos antiguos en una tabla particionada nunca es DELETE. Es esto:

```
-- Desengancha la partición sin bloquear lecturas ni escrituras
ALTER TABLE events DETACH PARTITION events_2025_07 CONCURRENTLY;

-- La tabla sigue existiendo por si quieres archivarla (pg_dump, COPY a S3...)
-- y cuando ya no la necesites:
DROP TABLE events_2025_07;
```

DETACH ... CONCURRENTLY (PostgreSQL 14+) usa locks débiles y no interrumpe el tráfico, con dos condiciones: no puede ejecutarse dentro de una transacción y la tabla no debe tener partición DEFAULT. Frente al DELETE equivalente, la diferencia es brutal: metadatos frente a millones de tuplas muertas, WAL mínimo frente a gigabytes, cero bloat frente a semanas de autovacuum.

Migrar una tabla existente sin downtime

No puedes convertir una tabla normal en particionada con un ALTER TABLE. La migración estándar sin cortar el servicio funciona así:

```
-- 1. Crea la nueva tabla particionada con un nombre temporal
CREATE TABLE events_new (LIKE events INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
PARTITION BY RANGE (created_at);

-- 2. Añade a la tabla vieja un CHECK que describa su rango real.
--    NOT VALID + VALIDATE evita el lock largo del escaneo:
ALTER TABLE events ADD CONSTRAINT events_range_chk
CHECK (created_at >= '2020-01-01' AND created_at < '2026-08-01') NOT VALID;
ALTER TABLE events VALIDATE CONSTRAINT events_range_chk;

-- 3. El intercambio, en una transacción breve:
BEGIN;
ALTER TABLE events RENAME TO events_legacy;
ALTER TABLE events_new RENAME TO events;
-- Gracias al CHECK ya validado, ATTACH no necesita escanear la tabla:
ALTER TABLE events ATTACH PARTITION events_legacy
  FOR VALUES FROM ('2020-01-01') TO ('2026-08-01');
COMMIT;

-- 4. Crea las particiones nuevas a partir de ahí (o deja que pg_partman lo haga)
```

La clave es el paso 2: ATTACH PARTITION necesita demostrar que todas las filas caben en el rango declarado. Sin el CHECK previo, escanea la tabla completa manteniendo el lock; con él, el attach es instantáneo. La tabla histórica queda como una partición gigante y los datos nuevos fluyen a particiones mensuales; con la política de retención, el problema se disuelve solo con el tiempo. Si además necesitas repartir el histórico en particiones mensuales, el patrón es el mismo que un backfill por lotes: copiar por rangos con INSERT ... SELECT, verificar recuentos y soltar la partición legacy al final.

Errores comunes que invalidan el particionado

Consultas que no filtran por la clave de partición. Sin pruning, cada consulta abre todas las particiones y el rendimiento empeora respecto a la tabla única. Diseña la clave de partición a partir de tus consultas reales, no al revés.

Demasiadas particiones. Miles de particiones inflan el tiempo de planificación y el uso de memoria del planificador. Mensual suele ganar a diario; si necesitas más granularidad, subparticiona con cabeza y mantén el total en cientos, no miles.

Confiar en la partición DEFAULT. Parece una red de seguridad, pero las filas que caen en ella quedan fuera de la política de retención, y añadir nuevas particiones exige a PostgreSQL verificar que la DEFAULT no contiene filas del nuevo rango, con locks que pueden parar el tráfico. Con pg_partman y p_premake, es mejor no tener DEFAULT y tratar una inserción sin partición como el error que realmente es.

Olvidar que UPDATE puede mover filas. Si un UPDATE cambia el valor de la clave de partición, PostgreSQL lo ejecuta como DELETE + INSERT entre particiones: más caro y con semántica distinta frente a triggers y a lecturas concurrentes. Las claves de partición deberían ser efectivamente inmutables.

Claves únicas que no incluyen la clave de partición. El error aparece en el CREATE TABLE, así que lo detectarás pronto, pero la trampa real es asumir que UNIQUE (id, created_at) garantiza ids únicos globalmente: no lo hace. Un mismo id podría existir en dos meses distintos si tu generador de ids no lo impide.

Novedades de PostgreSQL 18

Las versiones recientes han ido puliendo los puntos débiles: PostgreSQL 18 acelera el pruning cuando hay muchas particiones gracias a un algoritmo de selección más eficiente en el planificador, y amplía los casos en que funcionan los partition-wise joins (joins resueltos partición a partición, activables con enable_partitionwise_join = on). Si vienes de una versión anterior a la 14, ganas además DETACH CONCURRENTLY; y si vienes de antes de la 12, el pruning en ejecución y los locks más ligeros en ATTACH cambian las reglas del juego. Merece la pena revisar las notas de versión antes de fijar tu diseño.

Checklist de producción

- La clave de partición aparece en el WHERE de tus consultas más frecuentes, sin funciones que la envuelvan.
- EXPLAIN confirma pruning en las consultas críticas (busca que solo aparezcan las particiones esperadas o Subplans Removed).

- Toda clave única incluye la clave de partición, y la unicidad global real está resuelta fuera si hace falta.
- La creación de particiones futuras está automatizada (pg_partman + pg_cron) y monitorizada: una alerta si quedan menos de N particiones futuras.
- La retención usa DETACH CONCURRENTLY + DROP, nunca DELETE masivo.
- No hay partición DEFAULT, o está vacía y vigilada.
- El número total de particiones se mantiene en cientos como máximo.

El particionado no es un ajuste de rendimiento genérico: es una decisión de arquitectura sobre el ciclo de vida de tus datos. Si tus consultas filtran por tiempo y tus datos caducan, es probablemente la mejora de operabilidad más grande que puedes hacerle a una tabla enorme. Si no, empieza por los índices.

Edición ampliada: particionado en producción, a fondo

Hasta aquí, la guía esencial. Lo que sigue es la parte que normalmente se aprende a base de incidentes: cómo elegir la clave de partición, qué pasa con las estadísticas, los locks que muerden de verdad, la replicación, el archivado del histórico y cómo probar todo esto antes de que llegue a producción.

Cómo elegir la clave de partición (y el caso multi-tenant)

La decisión correcta sale de tres preguntas, en este orden. Primera: ¿qué filtran tus consultas más frecuentes? Saca el top de pg_stat_statements y mira sus WHERE; la clave de partición tiene que aparecer ahí de forma literal, sin funciones que la envuelvan. Segunda: ¿cómo caducan los datos? Si hay retención temporal, RANGE por fecha es casi siempre la respuesta, porque convierte el borrado en un DETACH. Tercera: ¿cómo se reparten las escrituras? Que todas las escrituras aterricen en la última partición es normal y sano en series temporales; que aterricen todas en una sola partición de LIST significa que conservas los mismos hotspots que tenías sin particionar.

El conflicto clásico es multi-tenant: ¿particiono por tiempo o por tenant? Hay tres variantes con trade-offs muy distintos:

- **RANGE por fecha + índice (tenant_id, created_at)**. La opción por defecto. La retención funciona sola y las consultas de un tenant en un rango podan por fecha y resuelven por índice. Deja de bastar cuando un tenant gigante domina cada partición y necesitas aislarlo.
- **LIST por tenant**. Solo con pocos tenants grandes y estables (decenas, no miles). Aísla el ruido entre clientes, permite mover un tenant a su propio tablespace y hace trivial el borrado contractual de un cliente completo. Pero crear una partición desde la aplicación cada vez que se registra un tenant es fricción operativa pura: no lo uses con registro self-service.
- **HASH por tenant con subparticiones RANGE por fecha**. Reparte miles de tenants en N buckets estables y conserva la retención temporal. El precio: multiplicas el número de particiones y renuncias al aislamiento individual por tenant.

Regla final: la clave debe ser efectivamente inmutable y estar presente en la mayoría del top 20 de consultas. Si dudas entre dos diseños, gana el que produce pruning en más consultas reales, no el que queda más elegante en el diagrama.

Estadísticas y autovacuum: el detalle que casi todos olvidan

Autovacuum procesa las particiones hoja —cada una es una tabla normal— pero nunca la tabla padre. Una tabla particionada no tiene filas propias, así que autovacuum no la visita y, en consecuencia, jamás ejecuta ANALYZE sobre ella. Las estadísticas del padre existen y el planificador las usa para estimar joins y agregados sobre la tabla lógica completa: si nunca lanzas un ANALYZE manual del padre, esas estimaciones se quedan congeladas en el día de la migración.

```
-- ¿Cuándo se analizó cada pieza por última vez?
SELECT relname, last_analyze, last_autoanalyze, n_live_tup
FROM pg_stat_user_tables
WHERE relname LIKE 'events%'
ORDER BY relname;

-- El padre requiere ANALYZE manual; prográmalo con pg_cron:
SELECT cron.schedule(
  'analyze-events-parent',
  '30 3 * * *',
  $$ANALYZE public.events$$
);
```

Hay tres momentos en los que el ANALYZE manual no es opcional: después de un backfill grande (las particiones recién pobladas tienen estadísticas vacías y el planificador elegirá seq scans absurdos), después de ATTACH PARTITION (la partición llega con sus estadísticas, pero las del padre siguen sin reflejarla) y después de cambiar default_statistics_target o crear estadísticas extendidas. Para columnas correlacionadas dentro de una partición —tenant_id y event_type, por ejemplo— CREATE STATISTICS sobre cada partición mejora estimaciones de forma notable, y con pg_partman puedes definirlo una vez en la tabla template para que las particiones futuras nazcan con él.

BRIN frente a B-tree en particiones temporales

En particiones de series temporales append-only hay un índice sistemáticamente infrautilizado: BRIN. Un índice BRIN sobre created_at no guarda una entrada por fila, sino el mínimo y el máximo de cada bloque de páginas (128 por defecto, ajustable con pages_per_range). En una partición donde los datos se insertan en orden temporal, la correlación física entre posición y fecha es casi perfecta, y un BRIN de unos pocos cientos de KB filtra rangos de fechas casi tan bien como un B-tree de varios GB.

```
-- B-tree clásico sobre la partición: preciso pero enorme
CREATE INDEX idx_events_2026_08_created_btree
ON events_2026_08 (created_at);

-- BRIN: órdenes de magnitud más pequeño
CREATE INDEX idx_events_2026_08_created_brin
ON events_2026_08 USING brin (created_at)
WITH (pages_per_range = 64);

-- Compara tamaños reales:
SELECT indexrelname,
       pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE relname = 'events_2026_08';
```

Las condiciones para que BRIN funcione: inserciones en orden (o casi) de la columna indexada, consultas por rangos y ausencia de UPDATEs masivos que recoloquen filas. Un backfill desordenado destruye la correlación y con ella la selectividad del BRIN; si haces backfill, insértalo ordenado por fecha o lanza un CLUSTER antes de crear el índice. El patrón maduro en muchos sistemas de eventos: B-tree en (tenant_id, created_at) para las consultas por tenant, BRIN en created_at para barridos analíticos amplios, y nada más.

Claves foráneas en tablas particionadas

Desde PostgreSQL 12 las claves foráneas funcionan en ambas direcciones: una tabla particionada puede declarar FKs hacia tablas normales, y una tabla normal puede referenciar a una tabla particionada. Esto segundo tiene una implicación directa del requisito de unicidad: la FK debe apuntar a una clave que incluya la clave de partición.

```
-- FK desde la tabla particionada hacia un catálogo: sin sorpresas
ALTER TABLE events
  ADD CONSTRAINT fk_events_tenant
  FOREIGN KEY (tenant_id) REFERENCES tenants (id);

-- FK hacia la tabla particionada: debe incluir la clave de partición
CREATE TABLE event_annotations (
  event_id bigint NOT NULL,
  event_created_at timestamptz NOT NULL,
  note text NOT NULL,
  FOREIGN KEY (event_id, event_created_at)
    REFERENCES events (id, created_at) ON DELETE CASCADE
);
```

Las trampas están en el ciclo de vida. Un DETACH PARTITION falla si una fila de la tabla referenciante apunta a filas de esa partición: PostgreSQL no va a dejar filas huérfanas para que tu retención sea cómoda. En pipelines de eventos con retención, la solución habitual es duplicar la política: particiona también la tabla referenciante por la misma clave temporal y suéltalas en pareja, o directamente relaja la FK y valida por aplicación, aceptando el trade-off. Y un detalle de rendimiento: ON DELETE CASCADE sobre una partición gigante convierte tu DETACH + DROP casi gratis en un borrado masivo en la tabla hija; si el volumen referenciado es grande, la cascada te devuelve al problema que el particionado había eliminado.

Locks, DETACH interrumpido y cómo no bloquear producción

Todas las operaciones de gestión de particiones toman locks sobre la tabla padre, y la tabla padre es, por definición, la más caliente de tu sistema. La regla número uno es la misma que en cualquier migración: pon lock_timeout antes de cualquier DDL sobre el padre y reintenta si pierdes la carrera, porque un DDL esperando un lock bloquea detrás de él a todas las consultas nuevas.

```
SET lock_timeout = '3s';

-- Si expira, no pasa nada: reintenta en un bucle con backoff
ALTER TABLE events ATTACH PARTITION events_2026_09
FOR VALUES FROM ('2026-09-01') TO ('2026-10-01');
```

DETACH CONCURRENTLY merece atención especial porque funciona en dos transacciones internas. Si el proceso muere a mitad —un deploy que reinicia el pod, un statement_timeout agresivo—, la partición queda en un estado intermedio: sigue apareciendo en el catálogo marcada como en proceso de detach, y las consultas siguen funcionando, pero no puedes hacerle ATTACH ni otro DETACH. La salida está documentada y es explícita:

```
-- Completa un detach que quedó a medias
ALTER TABLE events DETACH PARTITION events_2025_07 FINALIZE;
```

Vigila también los locks al crear particiones: CREATE TABLE ... PARTITION OF toma un lock fuerte sobre el padre, mientras que crear la tabla suelta y luego hacer ATTACH (con el CHECK previo, como en la migración) solo necesita SHARE UPDATE EXCLUSIVE. pg_partman ya usa el patrón correcto; si escribes tu propia automatización, imítalo. Y recuerda el detalle de la partición DEFAULT: cada ATTACH tiene que demostrar que la DEFAULT no contiene filas del rango nuevo, así que una DEFAULT poblada convierte cada alta de partición en un escaneo con lock. Es otra razón más para no tenerla.

Muchas particiones: planificación, prepared statements y memoria

El coste oculto del particionado no está en ejecutar consultas sino en planificarlas. El planificador tiene que considerar cada partición candidata, abrir sus estadísticas y bloquearla; con miles de particiones, la planificación puede costar más que la ejecución. De ahí la recomendación de mantenerse en cientos. Tres frentes concretos:

Prepared statements. Un plan genérico no conoce los valores de los parámetros, así que no puede podar en tiempo de planificación: el plan referencia todas las particiones y confía en el pruning de inicialización (el Subplans Removed del EXPLAIN). Funciona, pero el plan cacheado arrastra metadatos de todas las particiones y la memoria del backend crece. Si ves planes genéricos gordos y lentos con tablas muy particionadas, prueba plan_cache_mode = force_custom_plan para esa sesión o rol: pagas replanificar cada vez, a cambio de podas en frío y planes pequeños.

Locks por consulta. Cada partición tocada por un plan es un lock más en la transacción. Consultas sin pruning sobre cientos de particiones pueden agotar la fast-path de locks e incluso obligarte a subir max_locks_per_transaction; el síntoma

en monitorización es contención en LWLock:LockManager bajo concurrencia alta. La solución de fondo nunca es subir el límite: es recuperar el pruning.

PostgreSQL 18. Las versiones recientes han atacado exactamente esto: PostgreSQL 18 elimina las particiones irrelevantes antes en el proceso de planificación —en tablas con cientos de particiones puede recortar el tiempo de planificación a la mitad—, permite partition-wise joins en más casos reduciendo su consumo de memoria y mejora las estimaciones de coste sobre particionadas. Si tu carga es de muchas particiones con consultas cortas, el salto a 18 se nota directamente en p99 de latencia.

Partition-wise joins y aggregates

Cuando dos tablas están particionadas por la misma clave, PostgreSQL puede resolver el join partición a partición en lugar de mezclar los dos conjuntos completos: events_2026_07 con event_annotations_2026_07, y así sucesivamente. Lo mismo con agregados: agregar por partición y combinar al final. Ambas optimizaciones existen pero vienen desactivadas por defecto porque encarecen la planificación:

```
SET enable_partitionwise_join = on;
SET enable_partitionwise_aggregate = on;

EXPLAIN (COSTS OFF)
SELECT e.tenant_id, count(*)
FROM events e
JOIN event_annotations a
  ON a.event_id = e.id AND a.event_created_at = e.created_at
WHERE e.created_at >= '2026-07-01' AND e.created_at < '2026-08-01'
GROUP BY e.tenant_id;
-- Busca en el plan: Append de joins por pareja de particiones
-- en lugar de un join de dos Appends gigantes
```

Cuándo compensa: joins analíticos grandes entre tablas co-particionadas, agregados sobre muchas particiones y hardware con paralelismo disponible, porque cada rama del Append puede paralelizarse. Cuándo no: consultas OLTP cortas que ya podan a una sola partición; ahí solo añades trabajo al planificador. El punto medio razonable es activarlos por rol o por sesión en la capa analítica y dejarlos apagados para el tráfico transaccional, midiendo siempre con EXPLAIN ANALYZE antes y después.

Almacenamiento por capas: tablespaces y archivado en frío

Las particiones son tablas físicas independientes, y eso abre una palanca de coste que una tabla monolítica no tiene: cada partición puede vivir en un almacenamiento distinto. Datos del mes corriente en NVMe local, trimestres anteriores en discos baratos, y lo más antiguo fuera de la base de datos.

```
-- Crea el tablespace en el volumen barato
CREATE TABLESPACE cold_storage LOCATION '/mnt/cold';

-- Mueve una partición antigua (lock exclusivo: hazlo en ventana tranquila)
ALTER TABLE events_2025_11 SET TABLESPACE cold_storage;

-- Los índices se mueven aparte:
ALTER INDEX idx_events_2025_11_tenant_created SET TABLESPACE cold_storage;
```

Para el archivado fuera de la base, el patrón encaja con la retención que ya tienes: DETACH CONCURRENTLY, exportar la tabla suelta y soltar. La exportación puede ser un pg_dump de esa tabla concreta o un COPY comprimido directo a un bucket:

```
ALTER TABLE events DETACH PARTITION events_2025_07 CONCURRENTLY;

-- Exporta a Parquet/CSV comprimido (aquí CSV + zstd hacia S3)
\copy events_2025_07 TO PROGRAM 'zstd -q -o /tmp/events_2025_07.csv.zst && aws s3 cp /tmp/events_2025_07.csv.zst
s3://archive/events/' CSV

DROP TABLE events_2025_07;
```

La prueba de fuego de cualquier archivado es la restauración: documenta y ensaya el camino de vuelta (crear la tabla con el esquema de entonces, COPY de vuelta, ATTACH con su rango original). Un archivo que nunca has restaurado es, a

efectos prácticos, un archivo que no existe. Y si auditoría o negocio piden consultas ocasionales sobre el histórico, considera dejar esas particiones en cold_storage dentro de la base: un tablespace lento pero consultable suele ser más barato que un proceso de restauración de emergencia.

Replicación lógica y pg_dump con particiones

La replicación lógica y el particionado se llevan bien solo si decides conscientemente en qué nivel publicas los cambios. Por defecto, una publicación sobre la tabla particionada emite los cambios como si vinieran de cada partición hoja, lo que obliga al suscriptor a tener exactamente las mismas particiones con los mismos nombres. Casi nunca es lo que quieres. El parámetro clave existe desde PostgreSQL 13:

```
CREATE PUBLICATION events_pub
FOR TABLE events
WITH (publish_via_partition_root = true);
```

Con publish_via_partition_root, los cambios se publican como cambios de la tabla lógica events, y el suscriptor puede tener el layout que quiera: otra granularidad de particiones, otra clave, o directamente una tabla sin particionar. Es la pieza que permite, por ejemplo, replicar tu tabla de eventos particionada por mes hacia un almacén analítico particionado por día.

Con pg_dump, el equivalente es --load-via-partition-root: el dump genera INSERTs contra la tabla padre en lugar de contra cada partición, de modo que el enrutado de tuplas decide en destino. Es más lento, pero imprescindible cuando restauras hacia un esquema de particiones distinto del original. Dos avisos operativos: los DDL de gestión de particiones (CREATE ... PARTITION OF, ATTACH, DETACH) no viajan por replicación lógica —tu automatización de particiones tiene que correr también en el suscriptor— y la sincronización inicial de una tabla enorme conviene trocearla suscribiendo por fases para no saturar el WAL sender.

Monitorización: las consultas de tu dashboard

Un sistema particionado introduce modos de fallo nuevos y silenciosos: la creación de particiones futuras se detiene, la retención deja de ejecutarse, la DEFAULT empieza a llenarse. Ninguno da error el día que ocurre; todos son incidentes semanas después. Las cuatro consultas que deberían estar en un dashboard (o exportadas a Prometheus vía postgres_exporter):

```
-- 1. Tamaño por partición: detecta desequilibrios y crecimiento anómalo
SELECT c.relname,
       pg_size_pretty(pg_total_relation_size(c.oid)) AS total
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhrelid
WHERE i.inhparent = 'events'::regclass
ORDER BY pg_total_relation_size(c.oid) DESC;

-- 2. ¿Cuántas particiones FUTURAS quedan creadas? Alerta si < 2
SELECT count(*) AS future_partitions
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhrelid
JOIN pg_catalog.pg_partition_tree('events') pt ON pt.relid = c.oid
WHERE (regexp_match(c.relname, '(\d{4})_(\d{2})$')) IS NOT NULL
  AND to_date((regexp_match(c.relname, '(\d{4}_\d{2})$'))[1], 'YYYY_MM')
      > date_trunc('month', now());

-- 3. Filas en la DEFAULT (si existe): debería ser siempre 0
SELECT count(*) FROM ONLY events_default;

-- 4. Último mantenimiento de pg_partman sin errores
SELECT jobname, status, start_time
FROM cron.job_run_details
ORDER BY start_time DESC LIMIT 10;
```

pg_partman trae además check_default(), que recorre las particiones DEFAULT de todas las jerarquías gestionadas y devuelve las que contienen filas: es el healthcheck perfecto para un job diario. Completa el cuadro con las métricas de siempre —bloat, autovacuum atrasado, edad de transacciones— pero medidas por partición, porque una partición caliente con autovacuum atrasado se esconde en los promedios de la tabla lógica.

Subparticionado y template tables con pg_partman 5

pg_partman 5 es solo particionado declarativo: el soporte de particionado por triggers desapareció y la versión mínima de PostgreSQL es la 14. La pieza más útil que quizá no conoces es la tabla template. Hay propiedades que el particionado nativo no propaga del padre a las particiones nuevas; pg_partman las resuelve dejándote definirlas una vez en una tabla plantilla que clona en cada partición que crea:

```
-- La template se crea junto a create_parent(); localízala:
SELECT template_table FROM partman.part_config
WHERE parent_table = 'public.events';

-- Define en la template lo que las particiones futuras deben heredar:
ALTER TABLE partman.template_public_events
  ALTER COLUMN payload SET STORAGE EXTERNAL;
CREATE INDEX ON partman.template_public_events USING brin (created_at);
ALTER TABLE partman.template_public_events SET (fillfactor = 90);
```

Ojo: los cambios en la template solo afectan a particiones futuras; las existentes hay que alterarlas a mano. Para el subparticionado, pg_partman ofrece create_sub_parent(), que aplica una segunda dimensión a cada partición hija. Úsalo con la calculadora delante: la multiplicación de particiones es exactamente la que te lleva de un sistema de cientos de tablas manejables a uno de miles que castiga cada planificación. En la mayoría de sistemas multi-tenant, RANGE mensual más un buen índice compuesto gana a cualquier subparticionado.

Backfill del histórico por lotes (con script)

Tras la migración sin downtime, la tabla legacy queda como una partición gigante. Funciona, pero no participa de la retención mensual ni del pruning fino. Repartir ese histórico en particiones mensuales es un backfill clásico: copiar por rangos, en lotes cortos, verificando recuentos y sin saturar la replicación.

```
import time
import psycopg

BATCH_HOURS = 6          # rango copiado por iteración
PAUSE_SECONDS = 0.5      # respiro para replicación y autovacuum

def backfill_month(conn, year: int, month: int) -> None:
    start = f"{year:04d}-{month:02d}-01"
    end = f"{year + month // 12:04d}-{month % 12 + 1:02d}-01"
    with conn.cursor() as cur:
        cur.execute("SET lock_timeout = '3s'")
        cur.execute(
            """INSERT INTO events
               SELECT * FROM events_legacy
               WHERE created_at >= %s AND created_at < %s
               ON CONFLICT DO NOTHING""",
            (start, end),
        )
        conn.commit()

def verify(conn, start: str, end: str) -> bool:
    with conn.cursor() as cur:
        cur.execute(
            """SELECT
                 (SELECT count(*) FROM events_legacy
                  WHERE created_at >= %s AND created_at < %s),
                 (SELECT count(*) FROM events
                  WHERE created_at >= %s AND created_at < %s
                    AND tableoid <> 'events_legacy'::regclass)""",
            (start, end, start, end),
        )
        legacy, copied = cur.fetchone()
        return legacy == copied
```

Las decisiones importantes están fuera del código: lotes definidos por rango de la clave de partición (no por OFFSET), commit por lote para no retener un snapshot eterno, ON CONFLICT DO NOTHING para que el script sea reejecutable, y la verificación comparando recuentos con tableoid para distinguir la copia del original. Cuando el último mes verifica, el final es el que ya conoces: DETACH CONCURRENTLY de la partición legacy, un ANALYZE del padre, y la tabla histórica pasa a

archivo. Hazlo en horas valle y vigila el lag de réplicas entre lotes: un backfill que tumba la replicación no ahorra tiempo, lo aplaza con intereses.

¿Y TimescaleDB o Citus? Cuándo salir del particionado nativo

El particionado declarativo cubre la mayoría de casos, pero hay dos extensiones que conviene conocer para saber dónde está tu frontera. TimescaleDB automatiza sobre PostgreSQL lo que esta guía hace a mano: sus hypertables crean chunks por tiempo automáticamente, sin pg_partman ni pg_cron, y añaden dos cosas que el particionado nativo no tiene: compresión columnar por chunk (habitual ver 90% o más de reducción en series temporales) y agregados continuos que materializan resúmenes de forma incremental. Si tu carga es métricas, IoT o telemetría con dashboards agregados, TimescaleDB hace gratis lo que aquí son tres secciones de operativa.

Citus resuelve el problema contrario: cuando el cuello de botella no es el ciclo de vida de los datos sino que un solo servidor ya no da más, Citus convierte la tabla en una tabla distribuida repartida en shards entre varios nodos. Es sharding real —lo que el particionado nativo explícitamente no es— con su complejidad correspondiente: co-location de tablas para que los joins no crucen la red, transacciones distribuidas, un coordinador que planifica. La escalera razonable: índices bien elegidos, después particionado nativo, después una extensión especializada, y solo al final sharding aplicativo. Cada peldaño se salta solo con una razón medida, no por anticipación.

Probar el particionado en CI

El diseño de particiones se rompe en silencio: alguien añade una consulta con date_trunc() sobre la clave, un ORM genera un filtro con cast implícito, y el pruning desaparece sin que ningún test funcional falle. La defensa es tratar el pruning como un contrato y testearlo, con la base real en un contenedor:

```
import json
import pytest
from testcontainers.postgres import PostgresContainer

@pytest.fixture(scope="session")
def pg():
    with PostgresContainer("postgres:18") as container:
        yield container

def scanned_partitions(cur, sql: str, params=()) -> set[str]:
    cur.execute(f"EXPLAIN (FORMAT JSON) {sql}", params)
    plan = json.dumps(cur.fetchone()[0])
    return {name for name in KNOWN_PARTITIONS if name in plan}

def test_range_query_prunes_to_one_partition(events_cursor):
    parts = scanned_partitions(
        events_cursor,
        """SELECT count(*) FROM events
           WHERE created_at >= %s AND created_at < %s""",
        ("2026-07-01", "2026-08-01"),
    )
    assert parts == {"events_2026_07"}

def test_unfiltered_query_is_flagged(events_cursor):
    parts = scanned_partitions(
        events_cursor, "SELECT count(*) FROM events WHERE event_type = 'click'"
    )
    assert len(parts) > 1  # documenta el coste: toca todas
```

El truco es extraer del EXPLAIN en JSON qué particiones aparecen en el plan y afirmar exactamente cuáles deberían ser. Añade un test que cree la partición del mes siguiente y ejecute el mantenimiento de pg_partman contra el contenedor: valida que tu configuración de create_parent y retención hace lo que crees. Son tests baratos —segundos, no minutos— y detectan la regresión de rendimiento más cara de detectar en producción: la que no lanza ningún error.

Caso práctico: de 800 GB monolíticos a particiones mensuales

Números de un escenario realista para aterrizar todo lo anterior. Una tabla events de 800 GB y 4.100 millones de filas, tres años de datos, creciendo 1,2 GB al día. Síntomas: autovacuum tardaba 14 horas en completar una pasada, el índice de

(tenant_id, created_at) ocupaba 210 GB y no cabía en los 128 GB de RAM, y el job trimestral de borrado (DELETE de 90 días) tardaba un fin de semana entero, generaba 300 GB de WAL y dejaba la réplica con horas de lag.

El plan siguió exactamente la secuencia de esta guía. Primero, dos semanas antes, el CHECK NOT VALID + VALIDATE sobre la tabla vieja (el VALIDATE tardó 6 horas sin bloquear nada). Después, el intercambio de nombres y el ATTACH instantáneo en una transacción que mantuvo el lock 180 milisegundos, medido con lock_timeout de 3 segundos y un reintento. pg_partman quedó configurado con particiones mensuales, premake de 4 y retención de 13 meses. El backfill del histórico corrió 11 noches a razón de tres meses por noche, con verificación de recuentos por mes; dos meses fallaron la verificación por un bug del ON CONFLICT con una unique parcial antigua, se repitieron, y el DETACH final de la legacy cerró la migración.

Resultados a 30 días: el p99 de la consulta principal (eventos de un tenant en los últimos 7 días) bajó de 480 ms a 45 ms —pruning a dos particiones más índice caliente en memoria—. La retención pasó de un DELETE de fin de semana a un DETACH + DROP de 2 segundos mensuales, y el WAL del proceso de retención pasó de 300 GB a prácticamente cero. Autovacuum procesa ahora la partición del mes corriente en 40 minutos. El único coste real: la planificación media subió un 8% en las consultas que no filtran por fecha, que eran el 3% del tráfico y se reescribieron en el siguiente sprint. Ninguno de estos números es exótico: son el resultado esperable de aplicar el patrón completo a una tabla con perfil temporal claro.

Preguntas frecuentes

¿Puedo convertir una tabla en particionada sin migrar los datos? No. No existe ALTER TABLE ... PARTITION BY. La ruta sin downtime es la de esta guía: tabla nueva, CHECK validado, intercambio de nombres y ATTACH de la vieja como primera partición.

¿Cuál es el número máximo razonable de particiones? El límite duro son decenas de miles, pero el límite sano es cientos. A partir de ahí pagas en planificación, memoria de planes y locks por consulta. PostgreSQL 18 sube ese techo de forma notable, pero no convierte 10.000 particiones diarias en una buena idea.

¿El particionado acelera los INSERT? No directamente: cada fila paga un enrutado a su partición, un coste pequeño pero real. Lo que mejora es todo lo de alrededor: índices más pequeños y calientes, autovacuum local y checkpoints más suaves. En cargas de escritura intensiva el beneficio neto suele ser positivo, pero por efectos de segundo orden.

¿Puedo tener índices distintos en cada partición? Sí, y es una técnica legítima: las particiones recientes pueden llevar índices caros para consultas operativas y las antiguas quedarse solo con el BRIN para analítica. Gestiona la diferencia con la template de pg_partman o con tu automatización, no a mano.

¿Qué pasa con las secuencias y los ids? La secuencia es única para toda la tabla lógica, así que los ids no colisionan entre particiones. Pero recuerda: la unicidad real de id solo está garantizada dentro de cada combinación (id, created_at) salvo que tu generador —secuencia, UUIDv7, Snowflake— la garantice por construcción.

¿Cómo cambio la clave de partición de una tabla ya particionada? No se puede in situ: es una nueva migración completa hacia una tabla con la clave nueva, usando el mismo patrón de intercambio. Por eso la elección de clave merece más análisis que ningún otro paso.

¿Particionar por UUID? Por HASH, sí, si el objetivo es solo repartir volumen. Por RANGE solo tiene sentido con UUIDv7 u otro formato ordenado por tiempo; con UUIDv4 aleatorio, RANGE no da pruning útil para consultas temporales.

¿Y si mi tabla no llega a 100 GB pero crece rápido? Particiona cuando la proyección a 12-18 meses cruce el umbral, no antes. Particionar pronto es deuda operativa sin retorno; migrar tarde es el fin de semana más largo de tu año. La proyección te dice cuál de los dos errores estás a punto de cometer.

Vacuum, freezing y wraparound: la ventaja escondida

Hay un beneficio del particionado que rara vez aparece en las comparativas y que en tablas gigantes es el más importante de todos: el anti-wraparound. PostgreSQL necesita congelar tuplas viejas antes de que el contador de transacciones dé la vuelta, y en una tabla monolítica de 800 GB ese vacuum de emergencia es un proceso de horas que no puedes cancelar y que compite por I/O con tu tráfico. Con particiones, cada pieza se congela por separado: las particiones históricas se congelan una vez y no vuelven a tocarse jamás, y el trabajo recurrente se concentra en la partición activa, que es órdenes de magnitud más pequeña.

```
-- ¿Qué particiones están más cerca del límite de wraparound?
SELECT c.relname,
       age(c.relfrozenxid) AS xid_age,
       pg_size_pretty(pg_relation_size(c.oid)) AS size
FROM pg_class c
JOIN pg_inherits i ON i.inhrelid = c.oid
WHERE i.inhparent = 'events'::regclass
ORDER BY age(c.relfrozenxid) DESC;

-- Congela agresivamente una partición cerrada, una sola vez:
VACUUM (FREEZE, VERBOSE) events_2025_12;
```

El patrón operativo: cuando una partición deja de recibir escrituras (el mes se cierra), lánzale un VACUUM FREEZE explícito en hora valle. A partir de ahí, el visibility map la marca como completamente congelada y los vacuums anti-wraparound la saltan por completo. También puedes ajustar la agresividad de autovacuum solo donde importa, con parámetros por partición sobre la partición activa: autovacuum_vacuum_scale_factor más bajo para la partición del mes corriente, valores por defecto para el resto. Ese nivel de control quirúrgico es imposible en una tabla monolítica.

Particiones y ORMs: dónde se rompe el pruning sin que nadie toque SQL

La mayoría de regresiones de pruning no vienen de SQL escrito a mano sino del ORM. Tres patrones concretos a vigilar. Primero, los casts implícitos: si la columna es timestamptz y el driver envía un timestamp sin zona (o viceversa), el planificador puede introducir una conversión sobre la columna y el pruning desaparece; en Django, USE_TZ y el tipo de columna deben estar alineados, y en SQLAlchemy conviene declarar DateTime(timezone=True) explícito. Segundo, los filtros generados: un date_range de conveniencia que por debajo hace date_trunc o ::date sobre la columna rompe el pruning exactamente igual que hacerlo a mano. Tercero, las migraciones: los autogeneradores (Django migrations, Alembic, Prisma Migrate) no entienden PARTITION BY, así que la tabla particionada se define con SQL crudo en una migración manual y se marca como gestionada externamente para que el autogenerador no intente recrearla.

```
# Django: consulta que PODA (rango directo sobre la columna)
Event.objects.filter(
    created_at__gte=start, created_at__lt=end, tenant_id=tenant
)

# Consulta que NO poda: función sobre la columna de partición
Event.objects.annotate(
    month=TruncMonth("created_at")
).filter(month=some_month)
```

La regla que resume los tres casos: en el WHERE, la clave de partición aparece desnuda y con el tipo exacto de la columna, y cualquier transformación se aplica al parámetro, nunca a la columna. Es la misma regla de siempre de los índices, elevada a crítica porque aquí el castigo no es un índice ignorado sino abrir cientos de particiones. Los tests de pruning en CI de la sección anterior existen precisamente para cazar al ORM cuando la incumple.

Runbook: los tres incidentes clásicos y su salida

Incidente 1: las escrituras fallan con "no partition of relation found for row". La creación de particiones futuras se detuvo —pg_cron caído, el job de partman fallando en silencio, o nadie lo programó— y llegó la primera fila del mes nuevo. Mitigación inmediata: crear la partición que falta a mano (CREATE TABLE ... PARTITION OF con su rango), que es una operación de milisegundos si usas el patrón de tabla suelta más ATTACH. Corrección de fondo: alerta sobre el número de particiones futuras (la consulta del dashboard) y revisión de cron.job_run_details en el healthcheck. Este incidente es el argumento definitivo a favor de p_premake = 4: te da cuatro meses de margen para enterarte.

Incidente 2: la DEFAULT está creciendo. Alguien la creó como red de seguridad y ahora contiene filas que ninguna política de retención va a tocar, y cada ATTACH nuevo escanea la DEFAULT con lock. Salida ordenada: identificar los rangos presentes en la DEFAULT, crear las particiones correctas para esos rangos, mover las filas por lotes (INSERT ... SELECT + DELETE por rango), y cuando quede vacía, decidir en equipo si se elimina o se mantiene vacía con alerta. La parte difícil no es técnica: es descubrir por qué llegaron filas fuera de rango, porque suele ser un bug de datos (relojes desajustados, imports con fechas basura) que seguirá ocurriendo.

Incidente 3: la latencia sube tras añadir particiones. El número de particiones cruzó el umbral en el que la planificación se nota: planes genéricos que referencian cientos de particiones, memoria de backend creciendo, contención en el lock manager. Diagnóstico: comparar EXPLAIN ANALYZE con y sin parámetros literales, y mirar si el plan cacheado

referencia muchas más particiones que las que ejecuta. Salidas por orden de preferencia: recuperar pruning en las consultas culpables, consolidar granularidad (de diaria a mensual con la técnica de migración de siempre), plan_cache_mode = force_custom_plan para los roles afectados, y la actualización a PostgreSQL 18, cuyo pruning temprano en planificación ataca exactamente este perfil de problema.

Los tres incidentes comparten moraleja: el particionado mueve el riesgo de las consultas al ciclo de vida. Las consultas se vuelven predecibles; a cambio, la gestión de particiones se convierte en un proceso de producción más, con sus alertas, su runbook y su dueño. Presupuesta esa operativa el día que decidas particionar, no el día del primer incidente.

Bonus: RANGE sobre enteros y la checklist de operación ampliada

No todo particionado por rango es temporal. RANGE funciona igual de bien sobre un bigint —un id de secuencia— y es el patrón correcto para tablas tipo cola o log de procesamiento donde la retención se expresa en "todo lo anterior al id X ya está procesado": particiones de, por ejemplo, 50 millones de ids, retención soltando las particiones completamente procesadas, y pruning perfecto para los workers que siempre leen el rango activo. La mecánica es idéntica a la de fechas, incluida la automatización: pg_partman soporta particionado por id nativo con p_interval numérico.

```
CREATE TABLE jobs (
  id bigint GENERATED ALWAYS AS IDENTITY,
  status text NOT NULL DEFAULT 'pending',
  payload jsonb,
  PRIMARY KEY (id)
) PARTITION BY RANGE (id);

SELECT partman.create_parent(
  p_parent_table := 'public.jobs',
  p_control := 'id',
  p_interval := '50000000'
);
```

Y para cerrar, la checklist de operación continua —lo que revisas cada trimestre, no solo el día del diseño—: las alertas de particiones futuras y de DEFAULT tienen dueño y se han disparado en un simulacro; el runbook de los tres incidentes está escrito y enlazado desde la alerta; el ANALYZE del padre está programado y su última ejecución es reciente; las particiones cerradas del último trimestre recibieron su VACUUM FREEZE; los tests de pruning en CI cubren las cinco consultas más caras de pg_stat_statements; la restauración desde el archivo frío se ensayó al menos una vez este año; y el número total de particiones, proyectado a 18 meses con el crecimiento actual, sigue en cientos. Si alguno de estos puntos falla, ya sabes cuál es el trabajo de la semana que viene: es infinitamente más barato ahora que durante el incidente.

Glosario rápido

Partition pruning: optimización por la que el planificador o el ejecutor descartan particiones que no pueden contener filas relevantes para la consulta. **Tuple routing:** el enrutado automático de cada fila insertada hacia su partición según el valor de la clave. **Partición hoja:** la tabla física que almacena filas, frente a la tabla particionada padre, que solo define estructura. **DEFAULT partition:** partición que recibe las filas que no encajan en ningún rango declarado; cómoda en apariencia, problemática en operación. **Partition-wise join/aggregate:** ejecución de joins y agregados partición a partición cuando las tablas comparten esquema de particionado. **Premake:** en pg_partman, número de particiones futuras que se mantienen creadas por adelantado. **Template table:** tabla plantilla de pg_partman de la que las particiones nuevas heredan propiedades que el particionado nativo no propaga. **DETACH CONCURRENTLY:** desenganche de una partición con locks débiles, en dos transacciones, sin cortar tráfico. **FINALIZE:** paso que completa un DETACH CONCURRENTLY interrumpido. **Anti-wraparound vacuum:** vacuum obligatorio que congela tuplas viejas para proteger el contador de transacciones; con particiones, se reduce a la partición activa. Tener este vocabulario compartido en el equipo acorta cada conversación de diseño y cada incidente: la mitad de los malentendidos sobre particionado son, en el fondo, malentendidos sobre estos diez términos.

Table Partitioning in PostgreSQL: Range, List, Hash and Zero-Downtime Retention

Guide · Intermediate · Real Projects · ~35 min read · PuigFlows

Once a table grows past a few hundred gigabytes, the problems stop being theoretical: indexes no longer fit in memory, autovacuum takes hours to walk the table, and deleting old data with DELETE locks, bloats, and hammers replication. PostgreSQL's declarative partitioning splits that table into smaller physical pieces — partitions — while your application keeps seeing a single logical table. Done well, it turns deleting a year of data into dropping a file; done poorly, it adds complexity without improving anything. This guide covers how to do it well.

Why partition (and what it does not solve)

Partitioning delivers three concrete benefits. First, cheap retention: removing a whole partition with DROP TABLE or DETACH PARTITION is a near-instant metadata operation, versus a mass DELETE that rewrites pages, triggers autovacuum, and floods the WAL. Second, partition pruning: if your queries filter on the partition key, the planner skips irrelevant partitions and scans a fraction of the data. Third, local maintenance: vacuum, analyze, and reindex operate partition by partition, with much smaller locks and durations.

What partitioning does not do: it does not speed up queries that don't filter on the partition key (it can actually make them worse, since every partition must be opened), it does not replace a good index, and it is not sharding — everything still lives on the same server. As a rule of thumb, start considering it when a table approaches 100 GB or when time-based data retention is a real requirement. For a 5 GB table, a well-chosen partial index usually beats any partitioning scheme.

Range, List, and Hash: pick the right method

RANGE is the most common method: it splits by intervals of an orderable column, almost always a date. It is the natural fit for events, logs, metrics, and any data with time-based retention.

```
CREATE TABLE events (
  id bigint GENERATED ALWAYS AS IDENTITY,
  tenant_id uuid NOT NULL,
  event_type text NOT NULL,
  payload jsonb NOT NULL DEFAULT '{}',
  created_at timestamptz NOT NULL DEFAULT now(),
  PRIMARY KEY (id, created_at)
) PARTITION BY RANGE (created_at);

CREATE TABLE events_2026_07 PARTITION OF events
  FOR VALUES FROM ('2026-07-01') TO ('2026-08-01');

CREATE TABLE events_2026_08 PARTITION OF events
  FOR VALUES FROM ('2026-08-01') TO ('2026-09-01');
```

Note two details. The upper bound is exclusive: the July partition accepts rows up to 2026-07-31 23:59:59.999999, and August 1st lands in the next one — no gaps, no overlaps. And the primary key includes created_at: on a partitioned table, every unique constraint must contain the partition key (we'll see why shortly).

LIST splits by discrete values: region, country, customer type. Useful when you want to isolate subsets with different lifecycles.

```
CREATE TABLE orders (
  id bigint NOT NULL,
  region text NOT NULL,
  total numeric(12,2),
  created_at timestamptz NOT NULL DEFAULT now()
) PARTITION BY LIST (region);

CREATE TABLE orders_eu PARTITION OF orders FOR VALUES IN ('es', 'fr', 'de', 'it');
CREATE TABLE orders_latam PARTITION OF orders FOR VALUES IN ('mx', 'ar', 'co', 'cl');
```

HASH spreads rows uniformly based on a column's hash. It allows no range pruning and no time-based retention; its use case is splitting a huge table that has no clear temporal axis, for example to parallelize maintenance.

```
CREATE TABLE sessions (
  session_id uuid NOT NULL,
  data jsonb
) PARTITION BY HASH (session_id);

CREATE TABLE sessions_p0 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE sessions_p1 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE sessions_p2 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE sessions_p3 PARTITION OF sessions FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

If you need it, methods can be combined through sub-partitioning (say, monthly RANGE with HASH by tenant inside), but every level multiplies the partition count. Twelve monthly partitions with four hash sub-partitions is 48 tables; 365 daily partitions is a planning problem. Keep the total in the low hundreds at most.

Partition pruning: verify it actually works

Pruning is what makes everything else pay off: the planner proves a partition cannot contain rows matching the WHERE clause and never even opens it. But it only happens when the filter applies directly to the partition key, with no functions wrapping it. Compare:

```
-- [YES] Pruning: the range is compared directly against created_at
EXPLAIN (COSTS OFF)
SELECT count(*) FROM events
WHERE created_at >= '2026-07-01' AND created_at < '2026-08-01';

-- Aggregate
--   ->  Seq Scan on events_2026_07 events
--       Filter: ...

-- [NO] No pruning: date_trunc() wraps the column
EXPLAIN (COSTS OFF)
SELECT count(*) FROM events
WHERE date_trunc('month', created_at) = '2026-07-01';
-- scans ALL partitions
```

Pruning happens at three moments: at planning time (literal values), at executor startup (parameters such as now() - interval '7 days', visible as Subplans Removed in EXPLAIN output), and during execution (values arriving from a join). The practical takeaway: using now() or prepared-statement parameters does not defeat pruning, but wrapping the column in functions does. Make a habit of confirming with EXPLAIN that critical queries touch only the expected partitions — it is the acid test of the whole design.

Indexes, primary keys, and unique constraints

Since PostgreSQL 11, an index created on the parent table automatically propagates to all partitions, present and future:

```
CREATE INDEX idx_events_tenant_created
ON events (tenant_id, created_at DESC);
```

The important restriction is unique keys: PostgreSQL has no global unique indexes across partitions, so every PRIMARY KEY or UNIQUE constraint must include the partition key. That is why our table uses PRIMARY KEY (id, created_at). If you need true global uniqueness on another column (say, an external_id), you must enforce it elsewhere: an auxiliary non-partitioned table holding the unique constraint, or uniqueness guaranteed by whatever produces the value.

An operational trick for large tables: create the index on the parent with ON ONLY, build each partition's index with CREATE INDEX CONCURRENTLY, then finish with ALTER INDEX ... ATTACH PARTITION. This avoids the long lock of building every index at once.

Automation with pg_partman and pg_cron

Creating partitions by hand works until the first time someone forgets next month's partition and writes start failing (or landing in the DEFAULT partition, which is worse). pg_partman automates creation and retention:

```
CREATE EXTENSION pg_partman;

SELECT partman.create_parent(
  p_parent_table := 'public.events',
  p_control := 'created_at',
  p_interval := '1 month',
  p_premake := 4  -- keep 4 future partitions pre-created
);

-- Retention: drop partitions holding data older than 12 months
UPDATE partman.part_config
SET retention = '12 months',
    retention_keep_table = false
WHERE parent_table = 'public.events';
```

And with pg_cron you schedule maintenance inside the database itself:

```
CREATE EXTENSION pg_cron;

SELECT cron.schedule(
  'partman-maintenance',
  '@hourly',
  $$CALL partman.run_maintenance_proc()$$
);
```

Every hour, pg_partman creates any missing future partitions and applies the retention policy. On managed services (RDS, Cloud SQL, Azure) both extensions are usually available; if not, the same maintenance can be driven by an external cron or your application's scheduler.

Retention: DETACH and DROP instead of DELETE

The right way to remove old data from a partitioned table is never DELETE. It is this:

```
-- Detach the partition without blocking reads or writes
ALTER TABLE events DETACH PARTITION events_2025_07 CONCURRENTLY;

-- The table still exists in case you want to archive it (pg_dump, COPY to S3...)
-- and once you no longer need it:
DROP TABLE events_2025_07;
```

DETACH ... CONCURRENTLY (PostgreSQL 14+) takes weak locks and does not interrupt traffic, with two conditions: it cannot run inside a transaction, and the table must not have a DEFAULT partition. Against the equivalent DELETE, the difference is brutal: metadata versus millions of dead tuples, minimal WAL versus gigabytes, zero bloat versus weeks of autovacuum.

Migrating an existing table with zero downtime

You cannot turn a regular table into a partitioned one with ALTER TABLE. The standard no-downtime migration works like this:

```
-- 1. Create the new partitioned table under a temporary name
CREATE TABLE events_new (LIKE events INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
PARTITION BY RANGE (created_at);

-- 2. Add a CHECK to the old table describing its actual range.
--    NOT VALID + VALIDATE avoids the long lock of the scan:
ALTER TABLE events ADD CONSTRAINT events_range_chk
CHECK (created_at >= '2020-01-01' AND created_at < '2026-08-01') NOT VALID;
ALTER TABLE events VALIDATE CONSTRAINT events_range_chk;

-- 3. The swap, in one short transaction:
BEGIN;
ALTER TABLE events RENAME TO events_legacy;
ALTER TABLE events_new RENAME TO events;
-- Thanks to the validated CHECK, ATTACH needs no table scan:
ALTER TABLE events ATTACH PARTITION events_legacy
  FOR VALUES FROM ('2020-01-01') TO ('2026-08-01');
COMMIT;

-- 4. Create the new partitions from here on (or let pg_partman do it)
```

The key is step 2: ATTACH PARTITION must prove that every row fits the declared range. Without the prior CHECK, it scans the whole table while holding its lock; with it, the attach is instant. The historical table remains as one giant partition while new data flows into monthly partitions; with the retention policy in place, the problem dissolves on its own over time. If you also need to spread the historical data into monthly partitions, the pattern is the same as any batched backfill: copy by ranges with INSERT ... SELECT, verify counts, and drop the legacy partition at the end.

Common mistakes that silently defeat partitioning

Queries that don't filter on the partition key. Without pruning, every query opens every partition and performance gets worse than the single table. Design the partition key from your actual queries, not the other way around.

Too many partitions. Thousands of partitions inflate planning time and planner memory. Monthly usually beats daily; if you need more granularity, sub-partition deliberately and keep the total in the hundreds, not thousands.

Relying on the DEFAULT partition. It looks like a safety net, but rows landing there fall outside your retention policy, and adding new partitions forces PostgreSQL to verify the DEFAULT holds no rows for the new range, with locks that can stall traffic. With pg_partman and p_premake, it is better to have no DEFAULT and treat an insert with no matching partition as the error it really is.

Forgetting that UPDATE can move rows. If an UPDATE changes the partition key value, PostgreSQL executes it as a DELETE + INSERT across partitions: more expensive, and with different semantics for triggers and concurrent reads. Partition keys should be effectively immutable.

Unique keys that don't include the partition key. The error shows up at CREATE TABLE time, so you'll catch it early — but the real trap is assuming UNIQUE (id, created_at) guarantees globally unique ids: it does not. The same id could exist in two different months if your id generator doesn't prevent it.

What's new in PostgreSQL 18

Recent releases keep polishing the weak spots: PostgreSQL 18 speeds up pruning when there are many partitions thanks to a more efficient selection algorithm in the planner, and extends the cases where partition-wise joins work (joins resolved partition by partition, enabled with enable_partitionwise_join = on). If you are coming from a version older than 14, you also gain DETACH CONCURRENTLY; and from before 12, runtime pruning and lighter ATTACH locks change the game entirely. The release notes are worth a read before you lock in your design.

Production checklist

- The partition key appears in the WHERE clause of your most frequent queries, with no functions wrapping it.
- EXPLAIN confirms pruning on critical queries (look for only the expected partitions, or Subplans Removed).
- Every unique key includes the partition key, and true global uniqueness is handled elsewhere if needed.
- Future partition creation is automated (pg_partman + pg_cron) and monitored: alert when fewer than N future partitions remain.

- Retention uses DETACH CONCURRENTLY + DROP, never mass DELETE.
- There is no DEFAULT partition, or it is empty and watched.
- The total partition count stays in the hundreds at most.

Partitioning is not a generic performance tweak: it is an architectural decision about your data's lifecycle. If your queries filter by time and your data expires, it is probably the single biggest operability upgrade you can give a huge table. If not, start with indexes.

Expanded edition: production partitioning, in depth

So far, the essential guide. What follows is the part people usually learn through incidents: how to choose the partition key, what happens to statistics, the locks that actually bite, replication, archiving historical data, and how to test all of it before it reaches production.

Choosing the partition key (and the multi-tenant case)

The right decision comes from three questions, in this order. First: what do your most frequent queries filter on? Pull the top of pg_stat_statements and look at their WHERE clauses; the partition key must appear there literally, with no functions wrapping it. Second: how does the data expire? If there is time-based retention, RANGE by date is almost always the answer, because it turns deletion into a DETACH. Third: how are writes distributed? All writes landing in the latest partition is normal and healthy for time series; all writes landing in a single LIST partition means you kept the same hotspots you had before partitioning.

The classic conflict is multi-tenant: partition by time or by tenant? Three variants with very different trade-offs:

- **RANGE by date + index on (tenant_id, created_at)**. The default option. Retention runs itself, and a tenant's queries over a range prune by date and resolve via the index. It stops being enough when one giant tenant dominates every partition and needs isolation.
- **LIST by tenant**. Only with a few large, stable tenants (dozens, not thousands). It isolates noise between customers, lets you move a tenant to its own tablespace, and makes contractual deletion of a whole customer trivial. But creating a partition from application code every time a tenant signs up is pure operational friction: do not use it with self-service registration.
- **HASH by tenant with RANGE sub-partitions by date**. Spreads thousands of tenants across N stable buckets while keeping time-based retention. The price: you multiply the partition count and give up per-tenant isolation.

Final rule: the key must be effectively immutable and present in most of your top 20 queries. If you hesitate between two designs, the winner is the one that produces pruning in more real queries, not the one that looks more elegant on the diagram.

Statistics and autovacuum: the detail almost everyone forgets

Autovacuum processes the leaf partitions — each is a normal table — but never the parent. A partitioned table has no rows of its own, so autovacuum never visits it and, as a consequence, never runs ANALYZE on it. The parent's statistics exist and the planner uses them to estimate joins and aggregates over the whole logical table: if you never run a manual ANALYZE on the parent, those estimates stay frozen on migration day.

```
-- When was each piece last analyzed?
SELECT relname, last_analyze, last_autoanalyze, n_live_tup
FROM pg_stat_user_tables
WHERE relname LIKE 'events%'
ORDER BY relname;

-- The parent needs manual ANALYZE; schedule it with pg_cron:
SELECT cron.schedule(
  'analyze-events-parent',
  '30 3 * * *',
  $$ANALYZE public.events$$
);
```

There are three moments when a manual ANALYZE is not optional: after a large backfill (freshly loaded partitions have empty statistics and the planner will pick absurd seq scans), after ATTACH PARTITION (the partition arrives with its stats,

but the parent's still don't reflect it), and after changing default_statistics_target or creating extended statistics. For correlated columns within a partition — tenant_id and event_type, say — CREATE STATISTICS per partition improves estimates noticeably, and with pg_partman you can define it once on the template table so future partitions are born with it.

BRIN versus B-tree on time-based partitions

On append-only time-series partitions there is one systematically underused index: BRIN. A BRIN index on created_at does not store one entry per row, but the minimum and maximum of each block of pages (128 by default, tunable with pages_per_range). In a partition where data is inserted in time order, the physical correlation between position and date is nearly perfect, and a BRIN of a few hundred KB filters date ranges almost as well as a multi-GB B-tree.

```
-- Classic B-tree on the partition: precise but huge
CREATE INDEX idx_events_2026_08_created_btree
ON events_2026_08 (created_at);

-- BRIN: orders of magnitude smaller
CREATE INDEX idx_events_2026_08_created_brin
ON events_2026_08 USING brin (created_at)
WITH (pages_per_range = 64);

-- Compare actual sizes:
SELECT indexrelname,
       pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE relname = 'events_2026_08';
```

The conditions for BRIN to work: insertions in (near) order of the indexed column, range queries, and no massive UPDATEs relocating rows. An unordered backfill destroys the correlation and with it the BRIN's selectivity; if you backfill, insert ordered by date or run CLUSTER before creating the index. The mature pattern in many event systems: a B-tree on (tenant_id, created_at) for per-tenant queries, a BRIN on created_at for wide analytical sweeps, and nothing else.

Foreign keys on partitioned tables

Since PostgreSQL 12, foreign keys work in both directions: a partitioned table can declare FKs to normal tables, and a normal table can reference a partitioned one. The latter carries a direct implication of the uniqueness requirement: the FK must point at a key that includes the partition key.

```
-- FK from the partitioned table to a catalog: no surprises
ALTER TABLE events
  ADD CONSTRAINT fk_events_tenant
  FOREIGN KEY (tenant_id) REFERENCES tenants (id);

-- FK to the partitioned table: must include the partition key
CREATE TABLE event_annotations (
  event_id bigint NOT NULL,
  event_created_at timestamptz NOT NULL,
  note text NOT NULL,
  FOREIGN KEY (event_id, event_created_at)
    REFERENCES events (id, created_at) ON DELETE CASCADE
);
```

The traps live in the lifecycle. A DETACH PARTITION fails if a row in the referencing table points at rows in that partition: PostgreSQL will not orphan rows to make your retention convenient. In event pipelines with retention, the usual solution is to duplicate the policy: partition the referencing table by the same time key and drop them as a pair, or relax the FK and validate in the application, accepting the trade-off. And one performance detail: ON DELETE CASCADE against a giant partition turns your almost-free DETACH + DROP into a mass delete on the child table; if the referenced volume is large, the cascade sends you right back to the problem partitioning had eliminated.

Locks, interrupted DETACH, and how not to block production

Every partition-management operation takes locks on the parent table, and the parent is, by definition, the hottest table in your system. Rule number one is the same as in any migration: set lock_timeout before any DDL on the parent and retry if you lose the race, because a DDL waiting on a lock queues every new query behind it.

```
SET lock_timeout = '3s';

-- If it expires, no harm done: retry in a loop with backoff
ALTER TABLE events ATTACH PARTITION events_2026_09
FOR VALUES FROM ('2026-09-01') TO ('2026-10-01');
```

DETACH CONCURRENTLY deserves special attention because it works in two internal transactions. If the process dies midway — a deploy restarting the pod, an aggressive statement_timeout — the partition is left in an intermediate state: it still shows up in the catalog marked as being detached, and queries keep working, but you cannot ATTACH it or DETACH it again. The way out is documented and explicit:

```
-- Complete a detach that was left halfway
ALTER TABLE events DETACH PARTITION events_2025_07 FINALIZE;
```

Watch the locks when creating partitions too: CREATE TABLE ... PARTITION OF takes a strong lock on the parent, while creating the standalone table and then ATTACHing it (with the prior CHECK, as in the migration) only needs SHARE UPDATE EXCLUSIVE. pg_partman already uses the correct pattern; if you write your own automation, imitate it. And remember the DEFAULT partition detail: every ATTACH must prove the DEFAULT holds no rows for the new range, so a populated DEFAULT turns every partition addition into a locked scan. One more reason not to have one.

Many partitions: planning cost, prepared statements, and memory

The hidden cost of partitioning is not in executing queries but in planning them. The planner has to consider every candidate partition, open its statistics, and lock it; with thousands of partitions, planning can cost more than execution. Hence the advice to stay in the hundreds. Three concrete fronts:

Prepared statements. A generic plan does not know the parameter values, so it cannot prune at planning time: the plan references every partition and relies on startup pruning (the Subplans Removed you see in EXPLAIN). It works, but the cached plan drags metadata for every partition and backend memory grows. If you see fat, slow generic plans on heavily partitioned tables, try plan_cache_mode = force_custom_plan for that session or role: you pay for replanning each time, in exchange for cold pruning and small plans.

Locks per query. Every partition a plan touches is one more lock in the transaction. Queries without pruning over hundreds of partitions can exhaust the fast-path lock slots and even force you to raise max_locks_per_transaction; the monitoring symptom is contention on LWLock:LockManager under high concurrency. The real fix is never raising the limit: it is getting pruning back.

PostgreSQL 18. Recent releases have attacked exactly this: PostgreSQL 18 eliminates irrelevant partitions earlier in the planning process — on tables with hundreds of partitions it can cut planning time in half — allows partition-wise joins in more cases while reducing their memory usage, and improves cost estimates on partitioned tables. If your workload is many partitions with short queries, the jump to 18 shows up directly in p99 latency.

Partition-wise joins and aggregates

When two tables are partitioned by the same key, PostgreSQL can resolve the join partition by partition instead of mixing the two full sets: events_2026_07 with event_annotations_2026_07, and so on. Same with aggregates: aggregate per partition and combine at the end. Both optimizations exist but ship disabled because they make planning more expensive:

```
SET enable_partitionwise_join = on;
SET enable_partitionwise_aggregate = on;

EXPLAIN (COSTS OFF)
SELECT e.tenant_id, count(*)
FROM events e
JOIN event_annotations a
  ON a.event_id = e.id AND a.event_created_at = e.created_at
WHERE e.created_at >= '2026-07-01' AND e.created_at < '2026-08-01'
GROUP BY e.tenant_id;
-- Look for: an Append of per-partition-pair joins
-- instead of one join of two giant Appends
```

When it pays off: large analytical joins between co-partitioned tables, aggregates over many partitions, and hardware with parallelism available, because each branch of the Append can be parallelized. When it does not: short OLTP queries that already prune to a single partition; there you only add planner work. The reasonable middle ground is enabling them per role or session in the analytics layer and keeping them off for transactional traffic, always measuring with EXPLAIN ANALYZE before and after.

Tiered storage: tablespaces and cold archiving

Partitions are independent physical tables, and that opens a cost lever a monolithic table does not have: each partition can live on different storage. Current month on local NVMe, previous quarters on cheap disks, and the oldest data outside the database.

```
-- Create the tablespace on the cheap volume
CREATE TABLESPACE cold_storage LOCATION '/mnt/cold';

-- Move an old partition (exclusive lock: do it in a quiet window)
ALTER TABLE events_2025_11 SET TABLESPACE cold_storage;

-- Indexes move separately:
ALTER INDEX idx_events_2025_11_tenant_created SET TABLESPACE cold_storage;
```

For archiving outside the database, the pattern fits the retention you already have: DETACH CONCURRENTLY, export the standalone table, and drop. The export can be a pg_dump of that specific table or a compressed COPY straight to a bucket:

```
ALTER TABLE events DETACH PARTITION events_2025_07 CONCURRENTLY;

-- Export to Parquet/compressed CSV (here CSV + zstd to S3)
\copy events_2025_07 TO PROGRAM 'zstd -q -o /tmp/events_2025_07.csv.zst && aws s3 cp /tmp/events_2025_07.csv.zst
s3://archive/events/' CSV

DROP TABLE events_2025_07;
```

The acid test of any archive is restoration: document and rehearse the way back (create the table with the schema of that era, COPY back, ATTACH with its original range). An archive you have never restored is, for practical purposes, an archive that does not exist. And if audit or business asks for occasional queries over history, consider keeping those partitions in cold_storage inside the database: a slow but queryable tablespace is usually cheaper than an emergency restore process.

Logical replication and pg_dump with partitions

Logical replication and partitioning get along only if you consciously decide at which level you publish changes. By default, a publication on the partitioned table emits changes as if they came from each leaf partition, which forces the subscriber to have exactly the same partitions with the same names. That is almost never what you want. The key parameter has existed since PostgreSQL 13:

```
CREATE PUBLICATION events_pub
FOR TABLE events
WITH (publish_via_partition_root = true);
```

With publish_via_partition_root, changes are published as changes to the logical table events, and the subscriber can have whatever layout it wants: a different partition granularity, a different key, or a plain unpartitioned table. It is the piece that lets you, for example, replicate your monthly-partitioned events table to an analytics store partitioned by day.

With pg_dump, the equivalent is --load-via-partition-root: the dump generates INSERTs against the parent table instead of against each partition, so tuple routing decides at the destination. It is slower, but essential when restoring into a partition layout different from the original. Two operational warnings: partition-management DDL (CREATE ... PARTITION OF, ATTACH, DETACH) does not travel through logical replication — your partition automation must also run on the subscriber — and the initial sync of a huge table is best staged by subscribing in phases so you do not saturate the WAL sender.

Monitoring: the queries for your dashboard

A partitioned system introduces new, silent failure modes: future-partition creation stops, retention stops running, the DEFAULT starts filling up. None of them errors on the day it happens; all of them are incidents weeks later. The four queries that should live on a dashboard (or be exported to Prometheus via postgres_exporter):

```
-- 1. Size per partition: spot imbalance and anomalous growth
SELECT c.relname,
       pg_size_pretty(pg_total_relation_size(c.oid)) AS total
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhrelid
WHERE i.inhparent = 'events'::regclass
ORDER BY pg_total_relation_size(c.oid) DESC;

-- 2. How many FUTURE partitions exist? Alert if < 2
SELECT count(*) AS future_partitions
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhrelid
JOIN pg_catalog.pg_partition_tree('events') pt ON pt.relid = c.oid
WHERE (regexp_match(c.relname, '(\d{4})_(\d{2})$')) IS NOT NULL
  AND to_date((regexp_match(c.relname, '(\d{4}_\d{2})$'))[1], 'YYYY_MM')
      > date_trunc('month', now());

-- 3. Rows in the DEFAULT (if it exists): should always be 0
SELECT count(*) FROM ONLY events_default;

-- 4. Latest pg_partman maintenance runs without errors
SELECT jobname, status, start_time
FROM cron.job_run_details
ORDER BY start_time DESC LIMIT 10;
```

pg_partman also ships check_default(), which walks the DEFAULT partitions of every managed hierarchy and returns the ones containing rows: the perfect healthcheck for a daily job. Complete the picture with the usual metrics — bloat, lagging autovacuum, transaction age — but measured per partition, because a hot partition with lagging autovacuum hides inside the averages of the logical table.

Sub-partitioning and template tables with pg_partman 5

pg_partman 5 is declarative partitioning only: trigger-based partitioning support is gone and the minimum PostgreSQL version is 14. The most useful piece you may not know is the template table. Some properties are not propagated from parent to new partitions by native partitioning; pg_partman solves this by letting you define them once on a template table that it clones into every partition it creates:

```
-- The template is created along with create_parent(); find it:
SELECT template_table FROM partman.part_config
WHERE parent_table = 'public.events';

-- Define on the template what future partitions should inherit:
ALTER TABLE partman.template_public_events
  ALTER COLUMN payload SET STORAGE EXTERNAL;
CREATE INDEX ON partman.template_public_events USING brin (created_at);
ALTER TABLE partman.template_public_events SET (fillfactor = 90);
```

Careful: template changes only affect future partitions; existing ones must be altered by hand. For sub-partitioning, pg_partman offers create_sub_parent(), which applies a second dimension to every child partition. Use it with a calculator in hand: partition multiplication is exactly what takes you from a manageable system of hundreds of tables to one of

thousands that punishes every planning cycle. In most multi-tenant systems, monthly RANGE plus a good composite index beats any sub-partitioning.

Backfilling history in batches (with a script)

After the zero-downtime migration, the legacy table remains as one giant partition. It works, but it does not participate in monthly retention or fine-grained pruning. Spreading that history into monthly partitions is a classic backfill: copy by ranges, in short batches, verifying counts, without saturating replication.

```
import time
import psycopg

BATCH_HOURS = 6          # range copied per iteration
PAUSE_SECONDS = 0.5      # breathing room for replication and autovacuum

def backfill_month(conn, year: int, month: int) -> None:
    start = f"{year:04d}-{month:02d}-01"
    end = f"{year + month // 12:04d}-{month % 12 + 1:02d}-01"
    with conn.cursor() as cur:
        cur.execute("SET lock_timeout = '3s'")
        cur.execute(
            """INSERT INTO events
               SELECT * FROM events_legacy
               WHERE created_at >= %s AND created_at < %s
               ON CONFLICT DO NOTHING""",
            (start, end),
        )
        conn.commit()

def verify(conn, start: str, end: str) -> bool:
    with conn.cursor() as cur:
        cur.execute(
            """SELECT
                 (SELECT count(*) FROM events_legacy
                  WHERE created_at >= %s AND created_at < %s),
                 (SELECT count(*) FROM events
                  WHERE created_at >= %s AND created_at < %s
                    AND tableoid <> 'events_legacy'::regclass)""",
            (start, end, start, end),
        )
        legacy, copied = cur.fetchone()
        return legacy == copied
```

The important decisions live outside the code: batches defined by ranges of the partition key (not by OFFSET), a commit per batch so you never hold an eternal snapshot, ON CONFLICT DO NOTHING so the script is safely re-runnable, and verification comparing counts using tableoid to tell the copy from the original. When the last month verifies, the ending is the one you already know: DETACH CONCURRENTLY on the legacy partition, an ANALYZE of the parent, and the historical table goes to the archive. Run it off-peak and watch replica lag between batches: a backfill that knocks over replication does not save time, it defers it with interest.

What about TimescaleDB or Citus? When to leave native partitioning

Declarative partitioning covers most cases, but two extensions are worth knowing so you can see where your boundary is. TimescaleDB automates on top of PostgreSQL what this guide does by hand: its hypertables create time-based chunks automatically, with no pg_partman or pg_cron, and add two things native partitioning does not have: per-chunk columnar compression (90%+ reduction is common on time series) and continuous aggregates that materialize summaries incrementally. If your workload is metrics, IoT, or telemetry with aggregated dashboards, TimescaleDB gives you for free what here amounts to three sections of operations.

Citus solves the opposite problem: when the bottleneck is not data lifecycle but a single server running out of headroom, Citus turns the table into a distributed table spread in shards across several nodes. That is real sharding — which native partitioning explicitly is not — with the corresponding complexity: table co-location so joins do not cross the network, distributed transactions, a coordinator doing the planning. The sensible ladder: well-chosen indexes, then native partitioning, then a specialized extension, and only at the end application-level sharding. You skip a rung only with a measured reason, not out of anticipation.

Testing partitioning in CI

Partition designs break silently: someone adds a query with date_trunc() on the key, an ORM generates a filter with an implicit cast, and pruning disappears without a single functional test failing. The defense is treating pruning as a contract and testing it, against the real database in a container:

```
import json
import pytest
from testcontainers.postgres import PostgresContainer

@pytest.fixture(scope="session")
def pg():
    with PostgresContainer("postgres:18") as container:
        yield container

def scanned_partitions(cur, sql: str, params=()) -> set[str]:
    cur.execute(f"EXPLAIN (FORMAT JSON) {sql}", params)
    plan = json.dumps(cur.fetchone()[0])
    return {name for name in KNOWN_PARTITIONS if name in plan}

def test_range_query_prunes_to_one_partition(events_cursor):
    parts = scanned_partitions(
        events_cursor,
        """SELECT count(*) FROM events
           WHERE created_at >= %s AND created_at < %s""",
        ("2026-07-01", "2026-08-01"),
    )
    assert parts == {"events_2026_07"}

def test_unfiltered_query_is_flagged(events_cursor):
    parts = scanned_partitions(
        events_cursor, "SELECT count(*) FROM events WHERE event_type = 'click'"
    )
    assert len(parts) > 1  # documents the cost: touches all
```

The trick is extracting from the JSON EXPLAIN which partitions appear in the plan and asserting exactly which ones should. Add a test that creates next month's partition and runs pg_partman maintenance against the container: it validates that your create_parent and retention configuration does what you think. These tests are cheap — seconds, not minutes — and they catch the performance regression that is most expensive to detect in production: the one that throws no error.

Case study: from an 800 GB monolith to monthly partitions

Numbers from a realistic scenario to ground everything above. An events table of 800 GB and 4.1 billion rows, three years of data, growing 1.2 GB per day. Symptoms: autovacuum took 14 hours to complete a pass, the (tenant_id, created_at) index weighed 210 GB and did not fit in the 128 GB of RAM, and the quarterly deletion job (a 90-day DELETE) took an entire weekend, generated 300 GB of WAL, and left the replica hours behind.

The plan followed exactly the sequence in this guide. First, two weeks ahead, the CHECK NOT VALID + VALIDATE on the old table (the VALIDATE took 6 hours without blocking anything). Then the rename swap and the instant ATTACH in a transaction that held the lock for 180 milliseconds, measured with a 3-second lock_timeout and one retry. pg_partman was configured with monthly partitions, premake of 4, and 13-month retention. The historical backfill ran for 11 nights at three months per night, with per-month count verification; two months failed verification due to a bug involving ON CONFLICT with an old partial unique index, were re-run, and the final DETACH of the legacy partition closed the migration.

Results after 30 days: p99 of the main query (one tenant's events over the last 7 days) dropped from 480 ms to 45 ms — pruning to two partitions plus a hot index in memory. Retention went from a weekend-long DELETE to a monthly 2-second DETACH + DROP, and the WAL of the retention process went from 300 GB to practically zero. Autovacuum now processes the current month's partition in 40 minutes. The only real cost: average planning time rose 8% for queries that do not filter by date, which were 3% of traffic and were rewritten the following sprint. None of these numbers is exotic: they are the expected outcome of applying the full pattern to a table with a clear time-based profile.

Frequently asked questions

Can I convert a table to partitioned without migrating the data? No. There is no ALTER TABLE ... PARTITION BY. The zero-downtime route is the one in this guide: new table, validated CHECK, rename swap, and ATTACH of the old table as the first partition.

What is the maximum reasonable number of partitions? The hard limit is tens of thousands, but the healthy limit is hundreds. Beyond that you pay in planning, plan memory, and locks per query. PostgreSQL 18 raises that ceiling noticeably, but it does not turn 10,000 daily partitions into a good idea.

Does partitioning speed up INSERTs? Not directly: every row pays a routing cost to its partition, small but real. What improves is everything around it: smaller, hotter indexes, local autovacuum, gentler checkpoints. In write-heavy workloads the net effect is usually positive, but through second-order effects.

Can I have different indexes on each partition? Yes, and it is a legitimate technique: recent partitions can carry expensive indexes for operational queries while old ones keep only the BRIN for analytics. Manage the difference through pg_partman's template or your automation, not by hand.

What about sequences and ids? The sequence is shared by the whole logical table, so ids do not collide across partitions. But remember: real id uniqueness is only guaranteed within each (id, created_at) combination unless your generator — sequence, UUIDv7, Snowflake — guarantees it by construction.

How do I change the partition key of an already partitioned table? Not in place: it is a full new migration to a table with the new key, using the same swap pattern. That is why key selection deserves more analysis than any other step.

Partitioning by UUID? By HASH, yes, if the goal is only spreading volume. By RANGE it only makes sense with UUIDv7 or another time-ordered format; with random UUIDv4, RANGE gives no useful pruning for time-based queries.

What if my table is under 100 GB but growing fast? Partition when the 12-18 month projection crosses the threshold, not before. Partitioning early is operational debt with no return; migrating late is the longest weekend of your year. The projection tells you which of the two mistakes you are about to make.

Vacuum, freezing, and wraparound: the hidden advantage

There is a benefit of partitioning that rarely appears in comparisons and that, on giant tables, is the most important of all: anti-wraparound. PostgreSQL needs to freeze old tuples before the transaction counter wraps, and on an 800 GB monolithic table that emergency vacuum is an hours-long process you cannot cancel that competes with your traffic for I/O. With partitions, each piece freezes separately: historical partitions get frozen once and are never touched again, and the recurring work concentrates on the active partition, which is orders of magnitude smaller.

```
-- Which partitions are closest to the wraparound limit?
SELECT c.relname,
       age(c.relfrozenxid) AS xid_age,
       pg_size_pretty(pg_relation_size(c.oid)) AS size
FROM pg_class c
JOIN pg_inherits i ON i.inhrelid = c.oid
WHERE i.inhparent = 'events'::regclass
ORDER BY age(c.relfrozenxid) DESC;

-- Aggressively freeze a closed partition, once:
VACUUM (FREEZE, VERBOSE) events_2025_12;
```

The operational pattern: when a partition stops receiving writes (the month closes), run an explicit VACUUM FREEZE on it off-peak. From then on, the visibility map marks it as fully frozen and anti-wraparound vacuums skip it entirely. You can also tune autovacuum aggressiveness only where it matters, with per-partition parameters on the active partition: a lower autovacuum_vacuum_scale_factor for the current month, defaults for everything else. That level of surgical control is impossible on a monolithic table.

Partitions and ORMs: where pruning breaks without anyone touching SQL

Most pruning regressions come not from hand-written SQL but from the ORM. Three concrete patterns to watch. First, implicit casts: if the column is timestamptz and the driver sends a timestamp without a zone (or vice versa), the planner can introduce a conversion on the column and pruning disappears; in Django, USE_TZ and the column type must be aligned, and in SQLAlchemy it pays to declare DateTime(timezone=True) explicitly. Second, generated filters: a convenience date_range that underneath does date_trunc or ::date on the column breaks pruning exactly as if you did it by

hand. Third, migrations: autogenerators (Django migrations, Alembic, Prisma Migrate) do not understand PARTITION BY, so the partitioned table is defined with raw SQL in a manual migration and marked as externally managed so the autogenerator does not try to recreate it.

```
# Django: query that PRUNES (direct range on the column)
Event.objects.filter(
    created_at__gte=start, created_at__lt=end, tenant_id=tenant
)

# Query that does NOT prune: function over the partition column
Event.objects.annotate(
    month=TruncMonth("created_at")
).filter(month=some_month)
```

The rule that sums up all three cases: in the WHERE clause, the partition key appears bare and with the column's exact type, and any transformation is applied to the parameter, never to the column. It is the same old index rule, elevated to critical because here the penalty is not one ignored index but opening hundreds of partitions. The CI pruning tests from the earlier section exist precisely to catch the ORM when it breaks the rule.

Runbook: the three classic incidents and the way out

Incident 1: writes fail with "no partition of relation found for row". Future-partition creation stopped — pg_cron down, the partman job silently failing, or nobody scheduled it — and the first row of the new month arrived. Immediate mitigation: create the missing partition by hand (CREATE TABLE ... PARTITION OF with its range), a millisecond operation if you use the standalone-table-plus-ATTACH pattern. Root fix: an alert on the number of future partitions (the dashboard query) and a review of cron.job_run_details in the healthcheck. This incident is the definitive argument for p_premake = 4: it gives you four months of margin to find out.

Incident 2: the DEFAULT is growing. Someone created it as a safety net and now it holds rows no retention policy will ever touch, and every new ATTACH scans the DEFAULT under lock. The orderly exit: identify the ranges present in the DEFAULT, create proper partitions for those ranges, move the rows in batches (INSERT ... SELECT + DELETE by range), and once it is empty, decide as a team whether to drop it or keep it empty with an alert. The hard part is not technical: it is discovering why out-of-range rows arrived, because it is usually a data bug (skewed clocks, imports with garbage dates) that will keep happening.

Incident 3: latency rises after adding partitions. The partition count crossed the threshold where planning becomes noticeable: generic plans referencing hundreds of partitions, backend memory growing, lock manager contention. Diagnosis: compare EXPLAIN ANALYZE with and without literal parameters, and check whether the cached plan references many more partitions than it executes. Exits in order of preference: restore pruning in the guilty queries, consolidate granularity (daily to monthly using the usual migration technique), plan_cache_mode = force_custom_plan for the affected roles, and the upgrade to PostgreSQL 18, whose earlier planning-time pruning attacks exactly this problem profile.

The three incidents share a moral: partitioning moves risk from queries to lifecycle. Queries become predictable; in exchange, partition management becomes one more production process, with its alerts, its runbook, and its owner. Budget that operational work the day you decide to partition, not the day of the first incident.

Bonus: RANGE over integers and the extended operations checklist

Not all range partitioning is temporal. RANGE works just as well over a bigint — a sequence id — and it is the right pattern for queue-like tables or processing logs where retention is expressed as "everything before id X is already processed": partitions of, say, 50 million ids, retention by dropping fully processed partitions, and perfect pruning for workers that always read the active range. The mechanics are identical to dates, including automation: pg_partman supports id-based partitioning natively with a numeric p_interval.

```
CREATE TABLE jobs (
  id bigint GENERATED ALWAYS AS IDENTITY,
  status text NOT NULL DEFAULT 'pending',
  payload jsonb,
  PRIMARY KEY (id)
) PARTITION BY RANGE (id);

SELECT partman.create_parent(
  p_parent_table := 'public.jobs',
  p_control := 'id',
  p_interval := '50000000'
);
```

And to close, the continuous-operations checklist — what you review every quarter, not just on design day: the future-partition and DEFAULT alerts have an owner and have fired in a drill; the runbook for the three incidents is written and linked from the alert; the parent ANALYZE is scheduled and its last run is recent; last quarter's closed partitions received their VACUUM FREEZE; the CI pruning tests cover the five most expensive queries in pg_stat_statements; restoration from cold archive was rehearsed at least once this year; and the total partition count, projected 18 months out at current growth, stays in the hundreds. If any of these fails, you know what next week's work is: it is infinitely cheaper now than during the incident.

Quick glossary

Partition pruning: the optimization by which the planner or executor discards partitions that cannot contain rows relevant to the query. **Tuple routing:** the automatic routing of each inserted row to its partition based on the key value. **Leaf partition:** the physical table that stores rows, as opposed to the partitioned parent, which only defines structure. **DEFAULT partition:** the partition that receives rows fitting no declared range; convenient in appearance, problematic in operation. **Partition-wise join/aggregate:** executing joins and aggregates partition by partition when tables share a partitioning scheme. **Premake:** in pg_partman, the number of future partitions kept pre-created. **Template table:** pg_partman's template from which new partitions inherit properties native partitioning does not propagate. **DETACH CONCURRENTLY:** detaching a partition with weak locks, in two transactions, without interrupting traffic. **FINALIZE:** the step that completes an interrupted DETACH CONCURRENTLY. **Anti-wraparound vacuum:** the mandatory vacuum that freezes old tuples to protect the transaction counter; with partitions, it shrinks to the active partition. Sharing this vocabulary across the team shortens every design conversation and every incident: half of all partitioning misunderstandings are, at bottom, misunderstandings about these ten terms.