PuigFlows

puigflows.com · Guia Premium

Backups y PITR en PostgreSQL: Archivado de WAL, Incrementales Nativos, pgBackRest y Simulacros de Restauración

PostgreSQL Backups and PITR: WAL Archiving, Native Incremental Backups, pgBackRest and Restore Drills

Josue Garcia

Guia · Nivel intermedio · Lectura estimada: 45 min

Actualizado el 27 de agosto de 2026

Edicion bilingue · Espanol / English

ESPANOL

Backups y PITR en PostgreSQL: Archivado de WAL, Incrementales Nativos, pgBackRest y Simulacros de Restauración

Hay una conversación que se repite en casi todas las postmortems de pérdida de datos y siempre suena igual: «sí, teníamos backups». Lo que casi nunca hay es alguien capaz de decir cuándo fue la última vez que restauró uno, cuánto tardó y qué faltaba al terminar. Esa distancia —entre tener copias y poder volver a operar— es de lo que trata esta guía.

PostgreSQL ofrece dos familias de herramientas que resuelven problemas distintos y que la gente confunde constantemente: el volcado lógico con pg_dump, y el backup físico con archivado continuo de WAL, que es lo único que permite recuperación a un punto en el tiempo (PITR). Vamos a ver las dos, cuándo usar cada una, cómo configurar el archivado sin llenar el disco, cómo funcionan los backups incrementales nativos que llegaron en PostgreSQL 17, cómo se hace un PITR real paso a paso, y cómo pgBackRest convierte todo esto en algo operable sin scripts caseros.

Primero los dos números: RPO y RTO

Antes de tocar una línea de configuración necesitas dos cifras acordadas con el negocio, no elegidas por ti en un rato libre.

- RPO (Recovery Point Objective): cuántos datos puedes permitirte perder, medido en tiempo. Si tu RPO son 5 minutos, un backup diario a las 3:00 no se acerca: te expone a perder hasta 24 horas de escrituras.
- RTO (Recovery Time Objective): cuánto puedes tardar en volver a dar servicio. Aquí entra el tamaño de la base, la velocidad del almacenamiento y —esto se olvida siempre— el tiempo de reproducir el WAL acumulado desde el backup base.

Estos dos números determinan casi todo lo demás. Un RPO de segundos exige archivado continuo de WAL, o pg_receivewal en streaming, o directamente una réplica. Un RTO de minutos sobre una base de 2 TB descarta restaurar desde un pg_dump, porque el volcado lógico tiene que reconstruir todos los índices y eso puede llevar horas. Cuando alguien pide «backups robustos» sin dar estos números, la conversación que toca tener es esa, no la de qué herramienta usar.

Backup lógico: pg_dump y lo que no te cuenta

pg_dump exporta el contenido de una base de datos como instrucciones SQL o en un formato propio comprimido. Se ejecuta dentro de una transacción con snapshot consistente, así que la copia es coherente aunque haya escrituras durante el volcado, y no bloquea a los escritores normales.

> BASH

```
# Formato "directory": el único que permite volcado y restauración en paralelo
pg_dump \
  --format=directory \
  --jobs=4 \
  --compress=zstd:3 \
  --file=/backups/tienda_2026-08-27 \
  --dbname="postgresql://backup_user@db.interno:5432/tienda"

# Restauración en paralelo sobre una base vacía
createdb tienda_restaurada
pg_restore \
  --dbname=tienda_restaurada \
  --jobs=4 \
  --exit-on-error \
  /backups/tienda_2026-08-27
```

Dos detalles marcan la diferencia. `--jobs` sólo funciona con el formato `directory`, y es lo que convierte una restauración de tres horas en una de cuarenta minutos, porque la creación de índices se paraleliza. Y `--exit-on-error` debería ser tu defecto: sin él, `pg_restore` continúa tras un fallo y puede terminar dejándote una base incompleta que aparenta estar bien.

Los tres olvidos clásicos

`pg_dump` vuelca *una* base de datos. No incluye roles, contraseñas ni configuración a nivel de clúster. La primera vez que alguien restaura en un servidor nuevo y descubre que ningún `GRANT` apunta a un rol existente suele ser a las tres de la mañana:

```BASH
# Objetos globales: roles, pertenencias a roles, tablespaces
pg_dumpall --globals-only --file=/backups/globals_2026-08-27.sql

# Restaurar SIEMPRE los globales antes que las bases
psql --dbname=postgres --file=/backups/globals_2026-08-27.sql
```

El segundo olvido son las extensiones: `pg_dump` escribe `CREATE EXTENSION`, pero no los binarios. Si restauras en un contenedor que no tiene instalado `pgvector` o `postgis`, la restauración se detiene en la primera línea. El tercero es que un volcado lógico no conserva el estado físico: los índices se reconstruyen, así que el resultado sale desfragmentado y, en tablas grandes, ese trabajo es la mayor parte del RTO.

Dicho esto, `pg_dump` tiene un superpoder que el backup físico no tiene: es portable entre versiones mayores y entre arquitecturas. Es la herramienta correcta para migrar de PostgreSQL 16 a 18, para copiar una base a staging o para exportar un solo esquema. Como estrategia de recuperación ante desastres, por sí sola es insuficiente.

Archivado continuo de WAL: el cimiento del PITR

PostgreSQL escribe cada cambio primero en el *write-ahead log*. Si conservas un backup físico del directorio de datos y *todos* los segmentos de WAL generados desde ese momento, puedes reconstruir el estado de la base

en cualquier instante posterior. Eso es PITR, y es lo que te salva del DELETE sin WHERE de las 16:42.

```ini
# postgresql.conf
wal_level = replica
archive_mode = on
archive_timeout = 300s       # fuerza cambio de segmento cada 5 min: techo del RPO

# El comando DEBE fallar si el destino ya existe
archive_command = 'test ! -f /wal_archive/%f && cp --no-clobber %p /wal_archive/%f'
```

El detalle que hace o rompe todo el sistema

El contrato de archive_command es simple y despiadado: si devuelve 0, PostgreSQL da el segmento por archivado y lo puede reciclar. Si devuelve cualquier otra cosa, lo reintenta indefinidamente y *no borra nada*. De ahí salen los dos fallos más caros de esta área.

- Un comando que devuelve 0 sin copiar. Un cp hacia un punto de montaje NFS caído puede devolver éxito escribiendo en el directorio local vacío que quedó debajo. Meses después el archivo tiene huecos y el PITR es imposible. Por eso el test ! -f del ejemplo: si el fichero ya existe, el comando falla y te enteras, en vez de sobrescribir en silencio un segmento distinto con el mismo nombre, algo que ocurre de verdad tras una recuperación mal gestionada.
- Un comando que falla y nadie mira. Si el destino se llena, PostgreSQL acumula WAL en pg_wal hasta agotar el disco del servidor, y entonces el motor se para. El backup mal configurado tumba la producción que pretendía proteger.

La consulta que debería estar en tu dashboard desde el primer día:

```sql
SELECT archived_count,
    last_archived_wal,
    last_archived_time,
    failed_count,
    last_failed_wal,
    last_failed_time,
    now() - last_archived_time AS retraso_archivado
FROM pg_stat_archiver;
```

Alerta sobre dos cosas: que failed_count crezca, y que retraso_archivado supere holgadamente tu archive_timeout. La segunda es la que detecta el archivado que se ha quedado colgado sin llegar a dar error.

Desde PostgreSQL 15 existe además archive_library, que carga un módulo en proceso en lugar de lanzar un shell por cada segmento. Es bastante más eficiente cuando generas mucho WAL. El módulo basic_archive de contrib es el ejemplo de referencia, pero en la práctica, si usas una biblioteca, lo normal es que sea la de tu herramienta de backup.

El backup base con pg_basebackup

Con el archivado en marcha necesitas una copia física del clúster que sirva de punto de partida. pg_basebackup se conecta por el protocolo de replicación y copia el directorio de datos entero, coordinándose con el servidor para marcar el inicio y el fin del backup.

```bash
pg_basebackup \
  --pgdata=/backups/base/2026-08-27 \
  --format=tar \
  --compress=server-zstd:6 \
  --wal-method=stream \
  --checkpoint=fast \
  --progress \
  --verbose \
  --dbname="postgresql://replicador@db.interno:5432/postgres"
```

Las opciones que de verdad importan:

- --wal-method=stream abre una segunda conexión que recibe el WAL generado durante el backup. Sin ella (modo fetch), si el archivado se retrasa o falla mientras copias, el backup base resulta inservible porque le faltan segmentos. Con stream la copia es autocontenida.
- --checkpoint=fast pide un checkpoint inmediato en lugar de esperar al ritmo normal. Empiezas antes a cambio de un pico de E/S; en un servidor ajustado, evalúa si te compensa.
- --compress=server-zstd:6 comprime en el servidor: ahorras ancho de banda a cambio de CPU en el primario. Si el primario va justo de CPU, usa client-zstd.

Si prefieres control manual —por ejemplo para coordinar un snapshot de volumen— la API de bajo nivel cambió en PostgreSQL 15: el modo exclusivo desapareció y las funciones se llaman ahora pg_backup_start() y pg_backup_stop(). Conviene saberlo porque circulan muchísimos scripts y tutoriales con pg_start_backup(), que sencillamente ya no existe.

```sql
-- La sesión debe permanecer abierta durante todo el backup
SELECT pg_backup_start(label => 'snapshot_zfs', fast => true);

-- ... aquí tomas el snapshot del volumen ...

-- Devuelve el LSN final, el contenido de backup_label y el mapa de tablespaces.
-- Hay que guardar backup_label junto al snapshot o la copia no arrancará.
SELECT * FROM pg_backup_stop(wait_for_archive => true);
```

Verifica el backup en el momento de crearlo

```bash
# Comprueba el backup contra su backup_manifest: ficheros, tamaños y checksums,
```

```
# y que el WAL necesario para recuperarlo está presente y es legible.
pg_verifybackup --progress /backups/base/2026-08-27

# PostgreSQL 18 añadió la verificación de backups en formato tar
pg_verifybackup --format=tar /backups/base/2026-08-27
```

Esto no sustituye a una restauración de prueba, pero detecta corrupción de transporte y ficheros truncados en segundos. Debería ser un paso obligatorio del script de backup: si `pg_verifybackup` devuelve distinto de 0, el backup no se marca como bueno.

Backups incrementales nativos (PostgreSQL 17 y posteriores)

Hasta PostgreSQL 16, para tener backups incrementales había que irse a pgBackRest o Barman. Desde la 17 el propio motor sabe resumir qué bloques ha tocado el WAL, y `pg_basebackup` puede copiar sólo esos.

```ini
# postgresql.conf — requiere reinicio
summarize_wal = on
wal_summary_keep_time = '10d'   # valor por defecto; súbelo si tu cadena es más larga
```

Con `summarize_wal = on` arranca un proceso auxiliar que escribe resúmenes en `pg_wal/summaries`. Esos resúmenes son los que permiten saber qué bloques cambiaron entre dos momentos sin releer el WAL entero.

```bash
# Domingo: backup completo
pg_basebackup -D /backups/dom --checkpoint=fast --wal-method=stream

# Lunes: incremental respecto al manifest del backup anterior
pg_basebackup -D /backups/lun \
  --incremental=/backups/dom/backup_manifest \
  --wal-method=stream

# Martes: incremental respecto al del lunes (la cadena crece)
pg_basebackup -D /backups/mar \
  --incremental=/backups/lun/backup_manifest \
  --wal-method=stream
```

Un backup incremental **no se puede restaurar directamente**. Hay que reconstruir un completo sintético con `pg_combinebackup`, pasándole la cadena entera en orden:

```bash
pg_combinebackup \
  /backups/dom /backups/lun /backups/mar \
  --output=/var/lib/postgresql/18/restaurado
```

Aquí está la trampa, y conviene interiorizarla antes de escribir una política de retención: **la cadena es indivisible**.

Si tu script de limpieza borra el completo del domingo porque «ya tiene siete días», los seis incrementales que cuelgan de él se convierten en basura ilegible. Cualquier retención tiene que razonar sobre cadenas completas, no sobre backups sueltos. Y si los resúmenes de WAL de todo el intervalo no están disponibles —porque wal_summary_keep_time se quedó corto— el incremental falla al crearse, que al menos es un fallo ruidoso y a tiempo.

Para depurar, pg_walsummary te deja inspeccionar el contenido de un fichero de resumen y ver exactamente qué bloques se consideran modificados.

PITR paso a paso

Escenario real: a las 16:42 alguien ejecuta un UPDATE sin WHERE sobre pedidos. Son las 17:05 y quieres el estado de las 16:41:30.

1. Para el servidor y salva lo que hay

```bash
sudo systemctl stop postgresql

# Nunca sobrescribas el directorio dañado: puede contener WAL sin archivar
sudo mv /var/lib/postgresql/18/main /var/lib/postgresql/18/main.roto
```

Ese mv no es paranoia. El directorio actual contiene los segmentos de WAL más recientes, que probablemente aún no han llegado al archivo. Si los pierdes, tu punto de recuperación máximo retrocede hasta el último segmento archivado, y con él parte de las escrituras buenas posteriores al error.

2. Restaura el backup base

```bash
sudo -u postgres mkdir -p /var/lib/postgresql/18/main
sudo -u postgres chmod 0700 /var/lib/postgresql/18/main
sudo -u postgres tar -xf /backups/base/2026-08-27/base.tar.zst \
  -C /var/lib/postgresql/18/main

# Vacía pg_wal: el WAL vendrá del archivo, no de la copia
sudo -u postgres rm -rf /var/lib/postgresql/18/main/pg_wal/*
```

3. Configura el objetivo de recuperación

```ini
# postgresql.conf del directorio restaurado
restore_command = 'cp /wal_archive/%f %p'
recovery_target_time = '2026-08-27 16:41:30+02'
recovery_target_inclusive = off      # parar ANTES de esa marca, sin incluirla
```

```
recovery_target_action = 'pause'     # NO promocionar: primero comprobar
```

```
BASH

# Este fichero vacío es lo que le dice a PostgreSQL que arranque en recuperación
sudo -u postgres touch /var/lib/postgresql/18/main/recovery.signal
```

`recovery_target_action = 'pause'` es la opción que más disgustos evita. Con `promote`, PostgreSQL alcanza el objetivo, crea una línea temporal nueva y abre la base para escritura; si te pasaste de instante, acabas de consolidar el error y toca empezar de cero. Con `pause` el motor se queda esperando en sólo lectura y puedes mirar los datos con calma.

4. Arranca, comprueba y confirma

```
BASH

sudo systemctl start postgresql
tail -f /var/log/postgresql/postgresql-18-main.log
```

```
SQL

-- ¿Seguimos en recuperación?
SELECT pg_is_in_recovery(), pg_last_wal_replay_lsn();

-- Comprueba los datos concretos que motivaron la recuperación
SELECT id, estado, total, actualizado_en
FROM pedidos
WHERE actualizado_en > '2026-08-27 16:00:00+02'
ORDER BY actualizado_en DESC
LIMIT 20;
```

Si los datos son los correctos, promociona. Si te has pasado o te has quedado corto, para el servidor, ajusta `recovery_target_time`, vuelve a restaurar el backup base y repite: no se puede rebobinar una recuperación en curso.

```
SQL

SELECT pg_promote(wait => true);
```

Líneas temporales: la parte que confunde a todo el mundo

Cada vez que promocionas tras una recuperación, PostgreSQL crea una *timeline* nueva y escribe un fichero `.history` en el archivo. Existe precisamente para que puedas hacer varios intentos sin machacar el historial original: la timeline 1 sigue intacta.

La consecuencia práctica es que si más tarde necesitas recuperar a un instante *anterior* a esa promoción, tienes que decirle a PostgreSQL en qué línea buscar, porque por defecto sigue la más reciente que encuentra en

el archivo:

```ini
recovery_target_timeline = '1'    # o 'current', o un ID concreto
```

Y un olvido habitual: tu **archive_command** tiene que archivar también los ficheros **.history**. Si filtras por el patrón de 24 caracteres de los segmentos, las recuperaciones que cruzan timelines fallarán justo en el segundo intento, que es cuando ya estás nervioso. Usar **%f** tal cual, como en los ejemplos, ya lo cubre.

pgBackRest: cuando dejas de escribir scripts

Todo lo anterior funciona, y conviene entenderlo porque es lo que hay debajo. Pero mantener a mano la retención, la verificación, la compresión, el cifrado, la subida a almacenamiento de objetos y el paralelismo termina siendo un proyecto en sí mismo. pgBackRest (2.59.1 en agosto de 2026) resuelve todo eso y añade cosas que a mano no vas a construir: backup incremental a nivel de bloque, restauración delta, checksums por fichero revalidados en la restauración y validación de checksums de página durante el propio backup.

```ini
; /etc/pgbackrest/pgbackrest.conf
[global]
repo1-path=/var/lib/pgbackrest
repo1-type=s3
repo1-s3-bucket=backups-produccion
repo1-s3-region=eu-west-1
repo1-s3-endpoint=s3.eu-west-1.amazonaws.com
repo1-cipher-type=aes-256-cbc
repo1-cipher-pass=CLAVE_INYECTADA_DESDE_EL_GESTOR_DE_SECRETOS
repo1-retention-full=4
repo1-retention-diff=7
repo1-bundle=y          ; agrupa ficheros pequeños: clave con almacenamiento de objetos
repo1-block=y           ; backup incremental por bloques
process-max=4
compress-type=zst
compress-level=6
start-fast=y
archive-async=y
spool-path=/var/spool/pgbackrest
log-level-console=info
log-level-file=detail

[produccion]
pg1-path=/var/lib/postgresql/18/main
pg1-port=5432
```

```ini
# postgresql.conf
archive_mode = on
```

```
archive_command = 'pgbackrest --stanza=produccion archive-push %p'
```

BASH

```
# Una sola vez, al montar la stanza
pgbackrest --stanza=produccion stanza-create

# Valida configuración, permisos y archivado de punta a punta.
# Merece un hueco en tu cron semanal.
pgbackrest --stanza=produccion check

# Domingo
pgbackrest --stanza=produccion --type=full backup
# Resto de días
pgbackrest --stanza=produccion --type=incr backup

# Auditoría del repositorio: checksums de todo lo almacenado
pgbackrest --stanza=produccion verify

# PITR con restauración delta
pgbackrest --stanza=produccion restore \
  --delta \
  --type=time \
  --target="2026-08-27 16:41:30+02" \
  --target-action=pause
```

Ese `--delta` es la diferencia entre un RTO de horas y uno de minutos en bases grandes: compara checksums contra lo que ya hay en el directorio de datos y copia sólo los ficheros que difieren, en vez de vaciarlo y traerlo todo de nuevo. Y `check` es la única forma barata de saber que el archivado funciona de verdad de extremo a extremo, no que la configuración parezca correcta.

3-2-1-1-0 y el escenario que casi nadie planifica

La regla clásica 3-2-1 (tres copias, dos medios, una fuera) se quedó corta cuando el ransomware empezó a buscar y cifrar los propios repositorios de backup. La versión que se usa hoy es 3-2-1-1-0: añade **una copia inmutable u offline** y **cero errores de verificación**.

En la nube eso significa activar Object Lock en el bucket del repositorio, en modo *compliance*, con un periodo de retención igual o mayor que tu ventana de recuperación. Y significa que la credencial que usa el servidor de base de datos debe poder escribir objetos nuevos pero no borrarlos ni sobrescribirlos. Si tu servidor de producción tiene permisos para vaciar el bucket de backups, tus backups tienen exactamente la misma superficie de ataque que tu base de datos.

JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "EscrituraDeNuevosObjetos",
```

```
      "Effect": "Allow",
      "Action": ["s3:PutObject", "s3:GetObject", "s3:ListBucket"],
      "Resource": [
        "arn:aws:s3:::backups-produccion",
        "arn:aws:s3:::backups-produccion/*"
      ]
    },
    {
      "Sid": "NuncaBorrarNiRelajarLaRetencion",
      "Effect": "Deny",
      "Action": [
        "s3:DeleteObject",
        "s3:DeleteObjectVersion",
        "s3:PutBucketVersioning",
        "s3:PutObjectRetention",
        "s3:BypassGovernanceRetention"
      ],
      "Resource": [
        "arn:aws:s3:::backups-produccion",
        "arn:aws:s3:::backups-produccion/*"
      ]
    }
  ]
}
```

La expiración por retención la ejecuta entonces un principal distinto, con su propia credencial, desde un sitio que no es el servidor de base de datos. Es un poco más de trabajo de montaje y es la diferencia entre perder una base y perder la empresa.

El simulacro de restauración: lo único que demuestra algo

Un backup tiene dos estados: restaurado o hipotético. La única forma de salir de la hipótesis es restaurar de verdad, de forma automática y periódica, y medir cuánto tarda. Ese número es tu RTO real, no el que aparece en el documento de arquitectura.

PYTHON

```
#!/usr/bin/env python3
"""Simulacro de restauracion: restaura el ultimo backup en una instancia
efimera, comprueba integridad y contenido, y publica metricas.

Pensado para cron semanal. Devuelve codigo distinto de 0 si algo falla,
para que lo recoja tu sistema de alertas.
"""

import subprocess
import sys
import time
from datetime import datetime, timezone

STANZA = "produccion"
DATA_DIR = "/var/lib/postgresql/drill"
PORT = "55432"
METRICS = "/var/lib/node_exporter/textfile/pgbackrest_drill.prom"
```

```
def run(cmd: list[str]) -> subprocess.CompletedProcess:
    print("$ " + " ".join(cmd), flush=True)
    return subprocess.run(cmd, check=True, text=True, capture_output=True)


def escribir_metricas(duracion: float, exito: int) -> None:
    with open(METRICS, "w") as f:
        f.write("# TYPE pgbackrest_restore_drill_seconds gauge\n")
        f.write("pgbackrest_restore_drill_seconds " + str(round(duracion, 1)) + "\n")
        f.write("# TYPE pgbackrest_restore_drill_success gauge\n")
        f.write("pgbackrest_restore_drill_success " + str(exito) + "\n")


def main() -> int:
    inicio = time.monotonic()
    print("Simulacro iniciado:", datetime.now(timezone.utc).isoformat())

    # 1. Restaurar el backup mas reciente hasta el ultimo punto consistente
    run([
        "pgbackrest", "--stanza=" + STANZA, "restore",
        "--pg1-path=" + DATA_DIR,
        "--type=immediate",
        "--target-action=promote",
        "--delta",
    ])

    # 2. Arrancar la instancia restaurada en un puerto aislado
    run(["pg_ctl", "-D", DATA_DIR, "-o", "-p " + PORT, "-w", "-t", "900", "start"])

    try:
        # 3. Esperar a que salga de recuperacion
        for _ in range(180):
            r = subprocess.run(
                ["psql", "-p", PORT, "-tAc", "SELECT pg_is_in_recovery()"],
                text=True, capture_output=True,
            )
            if r.stdout.strip() == "f":
                break
            time.sleep(5)
        else:
            print("ERROR: la instancia no salio de recuperacion", file=sys.stderr)
            escribir_metricas(time.monotonic() - inicio, 0)
            return 1

        # 4. Comprobar CONTENIDO, no solo que el proceso arranca
        comprobaciones = {
            "tablas_de_usuario": (
                "SELECT count(*) FROM pg_class c "
                "JOIN pg_namespace n ON n.oid = c.relnamespace "
                "WHERE c.relkind = 'r' "
                "AND n.nspname NOT IN ('pg_catalog', 'information_schema')"
            ),
            "pedidos_ultimas_24h": (
                "SELECT count(*) FROM pedidos "
                "WHERE creado_en > now() - interval '24 hours'"
            ),
            "indices_invalidos": "SELECT count(*) FROM pg_index WHERE NOT indisvalid",
```

```
        "roles": "SELECT count(*) FROM pg_roles WHERE rolcanlogin",
    }

    res = {}
    for nombre, sql in comprobaciones.items():
        salida = run(["psql", "-p", PORT, "-tAc", sql])
        res[nombre] = int(salida.stdout.strip())
        print("  " + nombre + " = " + str(res[nombre]))

    fallos = []
    if res["tablas_de_usuario"] == 0:
        fallos.append("la base restaurada no tiene tablas de usuario")
    if res["pedidos_ultimas_24h"] == 0:
        fallos.append("sin pedidos en 24h: el backup esta obsoleto")
    if res["indices_invalidos"] > 0:
        fallos.append("hay indices invalidos tras la restauracion")
    if res["roles"] < 2:
        fallos.append("faltan roles: revisa pg_dumpall --globals-only")

    if fallos:
        for f in fallos:
            print("ERROR: " + f, file=sys.stderr)
        escribir_metricas(time.monotonic() - inicio, 0)
        return 1

finally:
    subprocess.run(["pg_ctl", "-D", DATA_DIR, "-m", "immediate", "stop"])

duracion = time.monotonic() - inicio
print("OK: simulacro completado en " + str(round(duracion)) + "s")
escribir_metricas(duracion, 1)
return 0


if __name__ == "__main__":
    sys.exit(main())
```

Fíjate en la comprobación de pedidos_ultimas_24h. Verificar sólo que la instancia arranca deja pasar el fallo más insidioso de todos: un backup que se restaura sin un solo error pero contiene datos de hace tres semanas, porque el job lleva ese tiempo fallando en silencio. Y exportar la duración como métrica te permite alertar cuando tu RTO real se acerca al comprometido, en lugar de descubrirlo durante el incidente.

YAML

```
# Alertas de Prometheus asociadas al simulacro y al archivado
groups:
  - name: postgres-backups
    rules:
      - alert: SimulacroDeRestauracionFallido
        expr: pgbackrest_restore_drill_success == 0
        for: 5m
        labels: { severity: critical }
        annotations:
          summary: "El simulacro semanal de restauración ha fallado"

      - alert: SimulacroDeRestauracionSinEjecutarse
```

```
    expr: time() - node_textfile_mtime_seconds{file="pgbackrest_drill.prom"} > 10 * 24 * 3600
    labels: { severity: warning }
    annotations:
      summary: "Han pasado más de 10 días sin simulacro de restauración"

  - alert: RTORealCercaDelObjetivo
    expr: pgbackrest_restore_drill_seconds > 0.8 * 3600
    labels: { severity: warning }
    annotations:
      summary: "El RTO medido supera el 80% del objetivo de 1 hora"

  - alert: ArchivadoDeWALFallando
    expr: increase(pg_stat_archiver_failed_count[15m]) > 0
    labels: { severity: critical }
    annotations:
      summary: "archive_command está fallando: el PITR se está rompiendo ahora"
```

Comprimir y paralelizar: que el backup quepa en la ventana

Hasta aquí el backup ha sido una cuestión de corrección. A partir de cierto tamaño se convierte también en una cuestión de reloj: si tu copia completa tarda nueve horas y tu ventana de bajo tráfico son cuatro, el problema ya no es si el backup es válido, sino que nunca termina cuando debería.

Hay dos palancas y conviene entender qué aprieta cada una. La compresión reduce lo que viaja por la red y lo que ocupa en el repositorio, a cambio de CPU. El paralelismo reduce el tiempo de pared, a cambio de CPU y de E/S concurrente. Desde PostgreSQL 15, pg_basebackup puede comprimir en el servidor o en el cliente, y esa elección decide dónde duele.

BASH

```
# Compresión en el CLIENTE: el servidor envía los datos sin comprimir.
# Ahorras disco pero no ancho de banda. Útil si producción va justa de CPU.
pg_basebackup -h db-prod -D /backup/base -Ft \
  --compress=client-zstd:level=3 -Xstream -P

# Compresión en el SERVIDOR: comprime antes de enviar.
# Ahorras ancho de banda y disco; pagas CPU en la máquina de producción.
# workers=N reparte zstd entre varios hilos.
pg_basebackup -h db-prod -D /backup/base -Ft \
  --compress=server-zstd:level=9,workers=4 -Xstream -P
```

La pregunta que casi nadie se hace antes de elegir es cuánto da de sí el enlace. Hay una forma limpia de medirlo sin escribir un solo byte: `--target=blackhole` descarta el contenido y te deja ver el caudal real. Con un detalle que hace fallar el primer intento de todo el mundo: como el streaming de WAL lo implementa el cliente y no el servidor, blackhole es incompatible con `-Xstream`, que es el valor por defecto. Tienes que pedir `-Xnone` o `-Xfetch` explícitamente.

BASH

```
# Caudal real del backup sin almacenar nada.
```

```
time pg_basebackup -h db-prod --target=blackhole -Xnone -P

# Repite con compresión de servidor: si el tiempo baja, tu cuello era la red;
# si sube, era la CPU. Dos ejecuciones te ahorran una semana de suposiciones.
time pg_basebackup -h db-prod --target=blackhole -Xnone \
  --compress=server-zstd:level=9,workers=4 -P
```

En el lado lógico el paralelismo es todavía más rentable, pero sólo con el formato directorio. `pg_dump -Fd -j N` vuelca varias tablas a la vez, y `pg_restore -j N` reconstruye índices y restricciones en paralelo, que es donde se va la mayor parte del tiempo de una restauración lógica.

BASH

```
# Volcado paralelo: sólo el formato directorio (-Fd) admite -j
pg_dump -h db-prod -d tienda -Fd -j 8 -f /backup/tienda.dir

# Restauración paralela, con más memoria para construir los índices.
# PGOPTIONS afecta sólo a esta sesión: no tocas la configuración del servidor.
PGOPTIONS="-c maintenance_work_mem=2GB" \
  pg_restore -d tienda_restaurada -j 8 /backup/tienda.dir
```

Un límite que conviene interiorizar antes de dimensionar nada: pg_dump y pg_restore paralelizan *entre* tablas, nunca *dentro* de una tabla. Si el 70 % de tu base es una única tabla de hechos, subir `-j` de 8 a 16 no te dará nada: al final del volcado quedará un solo proceso trabajando en esa tabla mientras los otros quince miran. Cuando ese es tu perfil, la respuesta no es más paralelismo, es particionar la tabla o pasarte definitivamente al backup físico.

El backup que copia fielmente tus datos corruptos

Hay un fallo que no aparece en casi ninguna checklist y que anula todo lo anterior: un backup impecable de una base de datos que ya estaba corrupta. El archivado funciona, pg_verifybackup pasa, el simulacro levanta la instancia sin una queja, y sin embargo la página 4812 de tu tabla de pedidos lleva tres semanas con bytes que ningún proceso escribió a propósito. Cuando lo descubres, tus siete copias contienen exactamente el mismo daño.

La primera defensa es antigua y hasta hace poco venía apagada: las sumas de verificación de datos. Con checksums activos, PostgreSQL calcula un resumen de cada página al escribirla y lo comprueba al leerla; si no cuadra, la lectura falla con un error explícito en lugar de devolverte basura en silencio. **PostgreSQL 18 cambió el valor por defecto de initdb y ahora los activa**; se desactivan a propósito con `--no-data-checksums`. Si tu clúster nació antes, lo más probable es que los tengas apagados y no lo sepas.

SQL

```
-- ¿Están activos?
SHOW data_checksums;

-- Contador acumulado de fallos por base de datos.
-- Cualquier valor distinto de cero es un incidente abierto, no una métrica.
```

```
SELECT datname, checksum_failures, checksum_last_failure
FROM pg_stat_database
WHERE checksum_failures > 0;
```

Activarlos en un clúster existente exige parar el servidor: pg_checksums --enable reescribe todas las páginas y necesita el clúster apagado y cerrado limpiamente. En bases grandes son horas. La buena noticia es que esto tiene fecha de caducidad: PostgreSQL 19 incorpora la activación y desactivación en caliente mediante un proceso en segundo plano que va ensuciando páginas para que reciban su checksum en la siguiente escritura, sin ventana de parada.

BASH

```
# El clúster debe estar PARADO y haber cerrado limpiamente.
pg_ctl -D /var/lib/postgresql/18/main stop -m fast
pg_checksums --check -D /var/lib/postgresql/18/main   # primero comprueba
pg_checksums --enable --progress -D /var/lib/postgresql/18/main
pg_ctl -D /var/lib/postgresql/18/main start
```

Los checksums detectan daño en el almacenamiento, pero no ven la corrupción lógica: punteros de línea que apuntan donde no deben, tuplas cuyo xmin es posterior a su xmax, índices que prometen filas que ya no existen. Para eso está la extensión amcheck y su envoltorio de línea de comandos pg_amcheck, que recorre tablas e índices y devuelve una fila por cada problema encontrado en lugar de abortar en el primero. La documentación es refrescantemente honesta sobre su alcance: amcheck puede demostrar que hay corrupción, nunca que no la hay. A cambio, ningún error de corrupción que reporte es un falso positivo.

BASH

```
# --install-missing crea la extensión amcheck donde falte.
# --heapallindexed verifica que toda tupla visible del heap está en el índice.
# --parent-check añade la verificación de la estructura padre-hijo del B-tree.
pg_amcheck --database=tienda \
  --heapallindexed --parent-check \
  --jobs=4 --progress --install-missing
```

El sitio correcto para ejecutar esto no es producción: es la instancia efímera que ya levantas en el simulacro de restauración. Allí la CPU es tuya, --parent-check no bloquea a nadie, y estás comprobando exactamente los bytes que usarías en un desastre real. Un pg_amcheck semanal sobre la copia restaurada convierte tu simulacro en algo que verifica dos cosas a la vez: que puedes restaurar, y que lo que restauras está sano.

Encontrar el instante exacto: pg_waldump, XID y LSN

La sección de PITR daba por hecho que sabes a qué hora hay que volver. En un incidente real casi nunca lo sabes. Sabes que alguien ejecutó algo entre las 14:25 y las 14:40, que el ticket se abrió a las 14:52 y que el reloj del portátil desde el que se lanzó la consulta iba dos minutos adelantado. Restaurar a recovery_target_time = '2026-08-27 14:30:00' es una apuesta: si te pasas por poco, arrastras el desastre a la copia nueva; si te quedas corto, tiras a la basura minutos de trabajo legítimo de todos los demás.

Hay una alternativa que no requiere adivinar. El WAL contiene el registro de cada cambio con su identificador de transacción y su LSN, y pg_waldump lo lee en texto plano. Puedes localizar la transacción culpable y detener la recuperación justo antes de ella.

BASH

```
# 1. Acota la ventana. Averigua qué segmentos cubren la franja horaria.
ls -l --time-style=full-iso /archivo/wal/ | awk '$6 >= "2026-08-27" '

# 2. Vuelca los registros de esos segmentos y busca la operación.
#    -s y -e acotan por LSN si ya tienes una aproximación.
pg_waldump -p /archivo/wal 000000010000004A000000C1 000000010000004A000000C4 \
  | grep -E 'DELETE|TRUNCATE|DROP' \
  | head -50

# Salida típica (recortada):
# rmgr: Heap  len: 54  tx: 918273645  lsn: 4A/C2A1F038  desc: DELETE off: 12 ...
```

Con ese tx en la mano, la recuperación deja de ser aproximada. recovery_target_xid apunta a la transacción exacta y recovery_target_inclusive = false le dice a PostgreSQL que pare *antes* de aplicarla, no después.

INI

```
# postgresql.conf de la instancia de recuperación
restore_command = 'cp /archivo/wal/%f %p'

# Opción A: parar justo antes de la transacción culpable.
recovery_target_xid = '918273645'
recovery_target_inclusive = off

# Opción B: parar en un LSN concreto (el más preciso de todos).
# recovery_target_lsn = '4A/C2A1F038'

# En ambos casos: pausa al llegar al objetivo, no promociones a ciegas.
recovery_target_action = pause
```

Los cinco objetivos de recuperación no son intercambiables y merece la pena tenerlos claros antes de un incidente, no durante. recovery_target_time es cómodo pero depende de que la marca de tiempo del commit sea la que crees. recovery_target_xid es exacto para un cambio identificado. recovery_target_lsn es el más quirúrgico y el único que no tiene ambigüedad. recovery_target_name requiere que alguien hubiera llamado a pg_create_restore_point() antes del desastre, que en la práctica sólo ocurre si lo pones en tu procedimiento de despliegue. Y recovery_target = 'immediate' para en cuanto el backup base alcanza un estado consistente, que es lo que quieres cuando el objetivo no es recuperar datos sino comprobar que el backup sirve.

Un consejo que sale caro aprender por la vía práctica: crea un punto de restauración con nombre justo antes de cada migración de esquema. Cuesta una línea en el pipeline y convierte un PITR angustioso en una instrucción de una sola frase.

SQL

```
-- En el paso previo de tu migración, dentro del pipeline de despliegue:
```

```
SELECT pg_create_restore_point('pre-migracion-2026-08-27-facturas');

-- Y si algo sale mal, el objetivo de recuperación ya no se adivina:
--   recovery_target_name = 'pre-migracion-2026-08-27-facturas'
```

Sólo necesito una tabla: la restauración parcial

El caso más frecuente de todos no es la pérdida del centro de datos. Es alguien que ejecutó un **UPDATE** sin **WHERE** sobre una tabla de 40.000 filas mientras el resto de la base de datos, con sus tres terabytes, sigue perfectamente sana y atendiendo tráfico. Y aquí choca de frente la limitación fundamental del backup físico: es todo o nada. El backup base y el WAL reconstruyen un clúster entero en un instante entero. No existe un **--only-table**.

La solución no es un truco, es un procedimiento, y conviene tenerlo escrito antes de necesitarlo. Se restaura una instancia paralela, efímera, en otro puerto, se hace PITR hasta justo antes del desastre, se extrae de ahí lo que se necesita con herramientas lógicas y se inyecta en producción.

BASH

```
#!/usr/bin/env bash
set -euo pipefail

# --- Restauración quirúrgica de una sola tabla ---
PUERTO=5433
DIR=/srv/rescate/pgdata
OBJETIVO="2026-08-27 14:29:50+02"

# 1. Instancia efímera a partir del repositorio, en otro puerto.
pgbackrest --stanza=main --delta \
  --pg1-path="$DIR" \
  --type=time --target="$OBJETIVO" \
  --target-action=pause restore

# 2. Arranca aislada: sin red, sin archivado, sin que nadie la confunda con prod.
pg_ctl -D "$DIR" -o "-p $PUERTO -c listen_addresses=localhost -c archive_mode=off" start

# 3. Espera a que alcance el objetivo y promociona SOLO esta instancia.
until psql -p "$PUERTO" -Atc "SELECT pg_is_in_recovery()" | grep -q '^f$'; do
  psql -p "$PUERTO" -Atc "SELECT pg_wal_replay_resume()" >/dev/null 2>&1 || true
  sleep 2
done

# 4. Extrae únicamente lo que necesitas.
pg_dump -p "$PUERTO" -d tienda -t public.precios_producto \
  --data-only -Fc -f /tmp/precios_rescatados.dump

# 5. Cárgalo en producción en un esquema aparte. NUNCA directamente encima.
psql -h db-prod -d tienda -c "CREATE SCHEMA IF NOT EXISTS rescate;"
pg_restore -h db-prod -d tienda --data-only \
  --schema=public -t precios_producto /tmp/precios_rescatados.dump \
  2>/dev/null || echo "revisar: cargar manualmente en rescate.precios_producto"
```

El paso 5 es donde se pierden los incidentes. La tentación es restaurar la tabla encima de la de producción y

terminar antes; el problema es que entre el desastre y el rescate han pasado dos horas en las que ha habido escrituras legítimas sobre esa misma tabla. Cargar la copia antigua encima no repara, sobrescribe. Lo correcto es dejar los datos rescatados en un esquema aparte y reconciliar con SQL, decidiendo explícitamente qué gana en cada conflicto.

```sql
-- Reconciliación explícita: sólo restauramos las filas que el accidente tocó
-- y que nadie ha vuelto a modificar legítimamente después.
UPDATE public.precios_producto p
SET    precio_cents = r.precio_cents,
       actualizado_en = now()
FROM   rescate.precios_producto r
WHERE  p.id = r.id
  AND  p.precio_cents IS DISTINCT FROM r.precio_cents
  AND  p.actualizado_en < TIMESTAMPTZ '2026-08-27 14:29:50+02';

-- Y comprueba el alcance ANTES de confirmar, no después.
-- (ejecuta el UPDATE dentro de una transacción y revisa el conteo)
```

En el mundo lógico la restauración parcial es mucho más directa, y merece la pena conocer `-L`: pg_restore puede listar el índice de un volcado, dejarte editar ese listado a mano y restaurar exactamente los objetos que dejes en él. Es la herramienta correcta cuando necesitas cinco tablas y sus secuencias, pero no las cuarenta restantes.

```bash
# 1. Extrae el índice del volcado
pg_restore -l /backup/tienda.dump > /tmp/listado.txt

# 2. Edita /tmp/listado.txt y comenta con ';' todo lo que no quieras
grep -E 'TABLE DATA public (pedidos|lineas_pedido)' /tmp/listado.txt > /tmp/seleccion.txt

# 3. Restaura sólo esa selección, en orden de dependencias
pg_restore -d tienda_restaurada -L /tmp/seleccion.txt /backup/tienda.dump
```

Los slots de replicación que llenan pg_wal y tumban el servidor

Esto parece un tema de replicación y no de backups, pero pertenece aquí por una razón muy concreta: la forma más habitual de que un sistema de backup mate la producción no es fallando, es reteniendo. Si `archive_command` empieza a fallar, PostgreSQL no borra los segmentos que aún no ha archivado. Si además hay un slot de replicación abandonado, tampoco borra los que ese slot aún no ha consumido. `pg_wal` crece, el disco se llena y el servidor entra en modo de sólo lectura o directamente se cae. El backup, que existía para protegerte de una caída, la ha causado.

La defensa tiene dos parámetros y una consulta. `max_slot_wal_keep_size` (por defecto -1, es decir, sin límite) fija cuánto WAL puede retener un slot antes de que PostgreSQL lo invalide y siga adelante. Es una decisión deliberada: prefieres romper una réplica a perder el primario. Y `idle_replication_slot_timeout`, que llegó en PostgreSQL 18, invalida los slots que llevan demasiado tiempo sin que nadie se conecte a ellos, que es la firma

exacta del slot que alguien creó para una prueba y olvidó borrar.

```ini
# postgresql.conf
# Un slot no puede retener más de 64 GB de WAL. Al superarlo se invalida.
max_slot_wal_keep_size = 64GB

# PostgreSQL 18+: invalida slots inactivos más de 24 horas.
# La invalidación ocurre en el checkpoint, así que puede tardar
# hasta checkpoint_timeout en materializarse.
idle_replication_slot_timeout = 24h

# WAL que se conserva para réplicas SIN slot (mecanismo independiente).
wal_keep_size = 8GB
```

La consulta que debería estar en tu dashboard mide dos cosas distintas que la gente mezcla: el retraso de cada slot en bytes, y la salud del archivado. Un slot con 200 GB de retraso y un archivado que lleva cuarenta minutos sin avanzar son el mismo incidente visto desde dos ángulos.

```sql
-- Retraso por slot, en bytes y legible. wal_status cuenta la historia:
-- reserved (bien), extended (creciendo), unreserved (en riesgo), lost (invalidado).
SELECT slot_name,
       slot_type,
       active,
       wal_status,
       pg_size_pretty(
         pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)
       ) AS retraso,
       safe_wal_size IS NOT NULL AS tiene_limite
FROM   pg_replication_slots
ORDER  BY pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn) DESC;

-- Salud del archivado: si failed_count crece o last_archived_time se congela,
-- tienes minutos, no horas, antes de que el disco sea el problema.
SELECT archived_count,
       last_archived_wal,
       last_archived_time,
       failed_count,
       last_failed_wal,
       last_failed_time,
       now() - last_archived_time AS desde_el_ultimo_archivado
FROM   pg_stat_archiver;
```

Una nota sobre **max_slot_wal_keep_size** que conviene entender antes de ponerlo: invalidar un slot no es gratis. Si el slot pertenecía a una réplica de streaming, esa réplica tendrá que reconstruirse desde un backup base; si pertenecía a una suscripción de replicación lógica o a un conector CDC como Debezium, se pierde el punto de continuidad y hay que resembrar. El parámetro no evita el dolor, elige quién lo sufre. Y esa elección —una réplica rota frente a un primario caído— es casi siempre la correcta, siempre que tengas una alerta que te avise *antes* de que se alcance el límite y no un descubrimiento a posteriori.

pg_rewind: reincorporar el primario antiguo sin copiar tres terabytes

Después de un PITR o de un failover te queda un cadáver incómodo: el servidor antiguo, con datos válidos hasta el punto de divergencia y basura a partir de ahí. La opción evidente es borrarlo y reconstruirlo con un backup base completo. En una base de tres terabytes eso son horas de red y de disco para recuperar una máquina que difiere del nuevo primario en unos pocos gigabytes.

pg_rewind hace exactamente eso: compara las historias de línea temporal de origen y destino, localiza el punto de divergencia y copia únicamente los bloques que cambiaron desde entonces. El resultado es un directorio de datos equivalente a un backup base del origen, obtenido en una fracción del tiempo.

Tiene tres requisitos que hay que dejar preparados antes, porque no se pueden activar a posteriori sobre el clúster que ya divergió. El destino necesita wal_log_hints = on o checksums de datos activos, para que pg_rewind pueda saber qué bloques cambiaron. Necesita full_page_writes activo, que es el valor por defecto y jamás deberías desactivar. Y necesita tener en su pg_wal los segmentos que llegan hasta el punto de divergencia; si el servidor antiguo estuvo mucho rato funcionando después, esos segmentos pueden haber desaparecido, y ahí entra -c, que los recupera del archivo.

BASH

```
# El destino DEBE estar parado y haber cerrado limpiamente.
pg_ctl -D /var/lib/postgresql/18/main stop -m fast

# Antes de nada: una copia del directorio, aunque sea comprimida.
# pg_rewind es destructivo y elimina cualquier posibilidad de análisis forense.
tar -I 'zstd -3' -cf /rescate/pgdata-previo-rewind.tar.zst \
  -C /var/lib/postgresql/18 main

# Rebobina contra el nuevo primario.
#   -c / --restore-target-wal: recupera del archivo el WAL que falte.
#   --dry-run: imprime lo que haría sin tocar nada. Úsalo siempre la primera vez.
pg_rewind --target-pgdata=/var/lib/postgresql/18/main \
  --source-server="host=db-nuevo-primario port=5432 user=rewind dbname=postgres" \
  --restore-target-wal \
  --progress --dry-run

# Si la salida es la esperada, repite sin --dry-run.
```

Después del rebobinado el directorio no está listo: pg_rewind deja los datos pero no la configuración de réplica. Hay que crear standby.signal y apuntar el primary_conninfo al nuevo primario. Y conviene verificar la línea temporal resultante antes de dar nada por bueno.

BASH

```
touch /var/lib/postgresql/18/main/standby.signal
cat >> /var/lib/postgresql/18/main/postgresql.auto.conf <<'EOF'
primary_conninfo = 'host=db-nuevo-primario port=5432 user=replicador application_name=db-antiguo'
restore_command  = 'cp /archivo/wal/%f %p'
EOF

pg_ctl -D /var/lib/postgresql/18/main start
```

```
# Comprueba que sigue al primario y en qué línea temporal está.
psql -Atc "SELECT pg_is_in_recovery(), pg_last_wal_replay_lsn();"
psql -h db-nuevo-primario -Atc \
  "SELECT application_name, state, sync_state,
          pg_size_pretty(pg_wal_lsn_diff(sent_lsn, replay_lsn)) AS retraso
   FROM pg_stat_replication;"
```

Y la advertencia que da nombre a la mitad de los hilos de la lista de correo sobre este tema: **pg_rewind no es una herramienta de backup**. No te protege de nada, no guarda nada y no sustituye a un solo elemento de este artículo. Es una optimización de reincorporación, y su precio es que destruye el estado del servidor antiguo. Si ese servidor antiguo es la única copia que tienes de los datos que existían justo antes del incidente —porque el PITR se hizo sobre él, por ejemplo— rebobinarlo es la última mala decisión de un día que ya iba mal. De ahí el tar del principio.

Retención: la aritmética que decide el coste de tu repositorio

La retención es la decisión que más gente toma por omisión y más cara sale, en las dos direcciones. Retener de menos significa descubrir que la corrupción que detectaste hoy empezó hace cinco semanas y que tu copia más antigua tiene cuatro. Retener de más significa una factura de almacenamiento que crece sin que nadie la mire hasta que alguien de finanzas pregunta.

El punto de partida es una cuenta que se puede hacer en cinco minutos. Necesitas tres cifras: el tamaño comprimido de un backup completo, el volumen diario de WAL y la tasa de cambio diaria, que es lo que determina el tamaño de los incrementales.

```SQL
-- Tamaño de la base (aproxima el completo sin comprimir)
SELECT pg_size_pretty(pg_database_size(current_database()));

-- Volumen de WAL de las últimas 24 h, a partir del contador de archivado.
-- Cada segmento son 16 MB por defecto.
SELECT archived_count,
       pg_size_pretty(archived_count::bigint * 16 * 1024 * 1024) AS wal_acumulado,
       stats_reset,
       now() - stats_reset AS ventana
FROM   pg_stat_archiver;
```

Con esos números, una política de un completo semanal, diferenciales diarios y WAL continuo sobre una base de 500 GB que comprime a 120 GB, cambia un 4 % al día y genera 90 GB de WAL diarios sale así: cuatro completos retenidos son 480 GB, seis diferenciales por semana rondan los 29 GB cada uno (unos 700 GB en el mes) y el WAL de treinta días son 2,7 TB. Casi cuatro terabytes, de los cuales el 70 % es WAL. Ese reparto es lo que suele sorprender, y es la razón por la que la retención de archivo merece tanta atención como la de backups.

```INI
```

```
[global]
repo1-path=/var/lib/pgbackrest
repo1-cipher-type=aes-256-cbc

# Retener 4 backups completos. Al expirar uno, pgBackRest elimina
# automáticamente los diferenciales e incrementales que colgaban de él.
repo1-retention-full=4
repo1-retention-full-type=count

# Diferenciales: 6 por completo (uno al día entre completos semanales).
repo1-retention-diff=6

# LA LÍNEA QUE MÁS GENTE OLVIDA.
# Sin ella, pgBackRest conserva TODO el WAL de todos los backups retenidos.
# Con ella, sólo conserva el WAL necesario para hacer PITR sobre los N
# completos más recientes; los anteriores quedan restaurables al instante
# de su creación, pero ya no a un punto intermedio.
repo1-retention-archive=2
repo1-retention-archive-type=full

# El expire es caro en repositorios S3 grandes: desacóplalo del backup
# para que un borrado lento no alargue tu ventana.
repo1-retention-history=365
```

La trampa de **repo1-retention-archive** merece una frase propia porque funciona al revés de la intuición. Si no la configuras, no pierdes WAL: acumulas todo el de la ventana de retención completa y pagas por ello. Si la configuras demasiado agresiva, pierdes la capacidad de PITR sobre los backups antiguos sin que nada te avise; el backup sigue apareciendo en **pgbackrest info**, sigue siendo restaurable, pero sólo al instante exacto en que se hizo. Es una degradación silenciosa de tu RPO histórico, y sólo la descubres el día que alguien pide volver a un martes de hace tres semanas.

```BASH

# Desacopla el expire del backup: el backup no debería esperar
# a que S3 termine de borrar 400.000 objetos.
pgbackrest --stanza=main --no-expire backup --type=incr

# Y en una tarea aparte, fuera de la ventana crítica:
pgbackrest --stanza=main expire

# Comprueba qué hay de verdad en el repositorio y en qué estado.
# 'repo status' y el rango de WAL de cada backup son lo que importa.
pgbackrest --stanza=main info --output=json | \
  jq -r '.[0].backup[] | "\(.label) \(.type) \(.timestamp.stop) wal:\(.archive.start)-\(.archive.stop)"'
```

Una última consideración que no es técnica pero decide la política: hay retenciones que no eliges tú. Si procesas datos de tarjetas, si tienes obligaciones fiscales de conservación o si operas bajo un marco que exige poder demostrar el estado de los datos en una fecha concreta, la retención mínima viene dada. Y su reverso también existe: el derecho de supresión implica que un dato borrado a petición de un usuario no puede seguir vivo en un backup indefinidamente. La salida habitual es documentar la ventana de retención como parte de la política de privacidad y reaplicar las supresiones pendientes tras cualquier restauración. Ese segundo paso casi nunca está escrito en el runbook, y debería.

Cifrado y claves: el backup que no puedes descifrar

Cifrar el repositorio es fácil y casi todo el mundo lo hace bien. Lo que se hace mal es la gestión de la clave, y el fallo tiene una forma reconocible: una dependencia circular. La clave del repositorio vive en el gestor de secretos; el gestor de secretos autentica contra un servicio de identidad; ese servicio consulta una base de datos; esa base de datos es la que acabas de perder. Todo el sistema es correcto y aun así no puedes abrir tus backups.

La configuración en sí son tres líneas. pgBackRest cifra en el cliente antes de subir, de modo que el proveedor de almacenamiento nunca ve texto en claro, y eso se combina —no se sustituye— con el cifrado en reposo del propio bucket.

```ini
[global]
# Cifrado del repositorio en el lado del cliente.
repo1-cipher-type=aes-256-cbc
repo1-cipher-pass=CLAVE_LARGA_ALEATORIA_INYECTADA_EN_TIEMPO_DE_EJECUCION

# Y además, cifrado en reposo del propio bucket con una clave de KMS.
repo1-s3-key-type=auto
repo1-type=s3
repo1-s3-bucket=backups-postgres-prod
repo1-s3-region=eu-west-1
repo1-storage-ca-file=/etc/ssl/certs/ca-certificates.crt
```

Dos advertencias sobre esa clave. La primera: `repo1-cipher-pass` no se puede rotar sin rehacer el repositorio. No hay un comando de recifrado; cambiar la clave significa empezar un repositorio nuevo y mantener el antiguo hasta que su retención expire. Por eso conviene generarla larga y aleatoria desde el principio y no reutilizarla entre entornos. La segunda: si esa clave está sólo en el fichero de configuración del servidor de producción, y el servidor de producción es lo que ha ardido, tienes un repositorio de tres terabytes de ruido perfectamente cifrado.

El patrón que rompe la circularidad se llama *break-glass* y no es sofisticado: una copia de la clave, fuera de banda, que no dependa de ninguno de los sistemas que la clave sirve para recuperar. Lo importante no es la forma que tome, sino que su recuperación esté probada y que quede rastro de quién la usa.

```bash
# Genera la clave del repositorio una sola vez, con entropía suficiente.
openssl rand -base64 48

# Guárdala en el gestor de secretos para el día a día...
aws secretsmanager create-secret \
  --name prod/pgbackrest/cipher-pass \
  --secret-string "$(openssl rand -base64 48)" \
  --kms-key-id alias/backups-prod

# ...y saca una copia break-glass a un lugar que NO dependa de producción:
# otra cuenta cloud, otro proveedor, o un sobre sellado en una caja fuerte.
# Lo importante es que el procedimiento de acceso esté escrito y probado.
```

```
# En tiempo de ejecución, inyecta la clave por variable de entorno o
# por fichero montado. Nunca la dejes fija en un fichero versionado.
export PGBACKREST_REPO1_CIPHER_PASS="$(aws secretsmanager get-secret-value \
  --secret-id prod/pgbackrest/cipher-pass --query SecretString --output text)"
pgbackrest --stanza=main backup --type=incr
```

Y el paso que convierte esto de política en garantía: incluir el descifrado en el simulacro. Un simulacro que usa la clave que ya está cargada en la máquina de producción no demuestra nada sobre tu capacidad de descifrar en un desastre. El simulacro correcto se ejecuta desde una máquina limpia, recupera la clave por el camino break-glass y restaura con ella. Es incómodo precisamente porque prueba lo que hace falta probar. Hacerlo una vez al trimestre, con dos personas distintas cada vez, es lo que separa una política de cifrado de una ilusión de cifrado.

Servicios gestionados: lo que RDS hace por ti y lo que no

Si estás en RDS, Cloud SQL o Azure Database, buena parte de este artículo la ejecuta otro. El archivado continuo, el backup base, la retención y el PITR están montados y probados por gente cuyo trabajo es que funcionen. Es una ventaja real y no conviene fingir lo contrario. Lo que sí conviene es tener claro qué queda fuera, porque los huecos son concretos y no salen en el panel.

El primero es el alcance de la ventana. Los backups automáticos de RDS te dan PITR dentro del periodo de retención configurado, hasta 35 días. Fuera de ese periodo sólo existen las instantáneas manuales, y una instantánea manual restaura al instante en que se tomó, no a un punto intermedio. Si tu obligación de conservación es de un año, el panel de RDS no la cubre por sí solo.

El segundo es más sutil y es el que sorprende en un simulacro. La replicación de backups automáticos entre regiones existe y consigue un RPO de minutos, pero **las copias de instantáneas entre regiones o entre cuentas pierden la capacidad de PITR**: al restaurarlas vuelves al instante en que se creó la copia, no a un punto que tú elijas. Es decir, tu copia de seguridad geográfica tiene un RPO peor que tu copia principal, y esa diferencia hay que conocerla antes de escribirla en un documento de continuidad. Hay además cuotas que se alcanzan sin darse cuenta: por defecto, veinte backups automáticos entre regiones por cuenta.

BASH

```
# PITR en RDS: restaura a una instancia NUEVA, nunca sobre la existente.
aws rds restore-db-instance-to-point-in-time \
  --source-db-instance-identifier tienda-prod \
  --target-db-instance-identifier tienda-rescate \
  --restore-time 2026-08-27T12:29:50Z \
  --db-subnet-group-name privada \
  --no-publicly-accessible \
  --db-instance-class db.r6g.xlarge

# Comprueba hasta dónde puedes volver AHORA MISMO. Esta cifra
# debería estar en tu dashboard, no descubrirse durante un incidente.
aws rds describe-db-instances \
  --db-instance-identifier tienda-prod \
  --query 'DBInstances[0].LatestRestorableTime'
```

El tercer hueco es el que ninguna funcionalidad del proveedor puede tapar, y es el que justifica seguir haciendo tus propios volcados aunque estés en un gestionado: el radio de explosión de la cuenta. Los backups automáticos viven en la misma cuenta que la base de datos. Unas credenciales comprometidas con permisos suficientes, o un error de infraestructura como código que destruye un recurso con **skip_final_snapshot**, se llevan por delante la instancia y sus copias a la vez. La mitigación es aburrida y funciona: un volcado lógico periódico exportado a un bucket de *otra* cuenta, con Object Lock, al que producción no tenga permiso de borrado.

YAML

```yaml
# CronJob que exporta un volcado lógico fuera de la cuenta de producción.
apiVersion: batch/v1
kind: CronJob
metadata:
  name: exportacion-logica-fuera-de-cuenta
spec:
  schedule: "0 3 * * *"
  concurrencyPolicy: Forbid
  jobTemplate:
    spec:
      backoffLimit: 2
      template:
        spec:
          restartPolicy: Never
          serviceAccountName: exportador-backups   # IRSA: asume rol en la cuenta archivo
          containers:
            - name: pgdump
              image: postgres:18-alpine
              command: ["/bin/sh", "-c"]
              args:
                - |
                  set -euo pipefail
                  FECHA=$(date -u +%Y%m%dT%H%M%SZ)
                  pg_dump --format=custom --compress=zstd:3 \
                    --file=/tmp/tienda-$FECHA.dump
                  # El bucket destino vive en otra cuenta y tiene Object Lock
                  # en modo COMPLIANCE: ni siquiera su propietario puede borrar.
                  aws s3 cp /tmp/tienda-$FECHA.dump \
                    s3://archivo-frio-tienda/logico/tienda-$FECHA.dump
              env:
                - name: PGHOST
                  value: tienda-prod.abc123.eu-west-1.rds.amazonaws.com
                - name: PGUSER
                  value: exportador
                - name: PGPASSWORD
                  valueFrom:
                    secretKeyRef: { name: pg-exportador, key: password }
              resources:
                requests: { cpu: "500m", memory: "1Gi" }
                limits:   { memory: "2Gi" }
```

Conviene también no confundir dos cosas que Aurora ofrece y que suenan parecido. *Backtrack* rebobina el clúster hacia atrás en el tiempo sobre la misma instancia, es muy rápido y está pensado para deshacer un error reciente; su ventana es corta y no es un backup. El PITR desde backups sí crea un clúster nuevo y llega hasta donde alcance tu retención. Usar backtrack como sustituto del PITR es cómodo hasta el día en que el error

tiene más horas que la ventana de backtrack.

Un caso completo: 14:32, un UPDATE sin WHERE

Sirve de poco enumerar herramientas si no se ve cómo encajan bajo presión. Este es el recorrido de un incidente real en su forma resumida, con las cifras que importan y las dos decisiones que lo salvaron.

14:32. Una tarea de mantenimiento ejecuta un `UPDATE precios_producto SET precio_cents = precio_cents * 100` con el `WHERE` comentado durante una prueba local que nadie descomentó al desplegar. 38.400 filas afectadas. La transacción confirma.

14:41. Atención al cliente reporta pedidos con importes absurdos. El primer instinto del equipo es escribir un `UPDATE` inverso dividiendo entre 100. Alguien señala lo que no se ve: entre las 14:32 y las 14:41 ha habido cambios de precio legítimos, y la división ciega los corrompería también. Esa es la primera decisión que salva el incidente: *no reparar en caliente*.

14:44. Se declara el incidente y se abre el runbook de restauración parcial. La base tiene 1,9 TB, así que restaurar producción entera está descartado: el RTO estimado ronda las cinco horas y la tienda funciona correctamente para todo lo que no sean precios.

14:47. Se localiza la transacción con `pg_waldump` sobre los tres segmentos que cubren la franja. Aparece el xid 4.418.902.117 y su LSN. Se anota el LSN inmediatamente anterior como objetivo.

14:52. Arranca `pgbackrest restore --delta --type=lsn` sobre una instancia efímera de rescate en otro puerto. El `--delta` es la segunda decisión importante: la máquina de rescate conservaba un directorio de datos de un simulacro de la semana anterior, y la restauración delta sólo trae los bloques que difieren. 1,9 TB se convierten en 61 GB.

15:09. La instancia de rescate alcanza el objetivo y queda en pausa. Un `SELECT count(*) FROM precios_producto WHERE precio_cents > 1000000` devuelve cero: los precios están sanos.

15:14. `pg_dump --data-only -t precios_producto` extrae la tabla y la carga en un esquema `rescate` de producción. 38.400 filas, 4 MB.

15:23. Se ejecuta la reconciliación dentro de una transacción abierta: sólo se restauran las filas cuyo `actualizado_en` es anterior al incidente y cuyo precio actual difiere del rescatado. El conteo antes del `COMMIT` da 38.331 filas, no 38.400. La diferencia son 69 productos con cambios de precio legítimos posteriores, exactamente los que el `UPDATE` inverso habría destruido.

15:26. Commit. Incidente cerrado. RTO real: 54 minutos desde la declaración. RPO efectivo sobre los datos afectados: cero.

Lo interesante del caso no son las herramientas, que son las de siempre. Es que ninguna de las tres cosas que hicieron corto el incidente se improvisó. El runbook de restauración parcial existía porque alguien lo escribió tras un susto anterior. La máquina de rescate tenía un directorio de datos reciente porque el simulacro semanal lo dejaba ahí en lugar de borrarlo, lo que redujo la restauración de horas a diecisiete minutos. Y el equipo sabía leer `pg_waldump` porque formaba parte del ejercicio trimestral. El día del incidente no se aprende nada; sólo se ejecuta lo que ya se sabía.

Glosario

- RPO — Recovery Point Objective: cuántos datos, medidos en tiempo, puedes permitirte perder.
- RTO — Recovery Time Objective: cuánto puedes tardar en volver a estar operativo.
- WAL — Write-Ahead Log: el registro secuencial de todos los cambios, escrito antes de tocar los ficheros de datos. Es el cimiento del PITR y de la replicación.
- LSN — Log Sequence Number: la posición de un registro dentro del WAL. Identifica un instante del clúster sin ambigüedad.
- Backup base — Copia física del directorio de datos completo, tomada de forma consistente. Es el punto de partida sobre el que se reproduce el WAL.
- PITR — Point-In-Time Recovery: restaurar un backup base y reproducir WAL hasta un instante concreto.
- Línea temporal — Identificador que PostgreSQL incrementa cada vez que una recuperación se promociona, para que dos historias divergentes no se mezclen. Se documenta en los ficheros .history.
- Backup diferencial — Contiene lo que cambió desde el último backup completo.
- Backup incremental — Contiene lo que cambió desde el backup anterior, sea del tipo que sea. Encadena, y por eso la cadena se rompe entera si falta un eslabón.
- Restauración delta — Restauración que compara el destino con el repositorio y sólo trae los ficheros que difieren, en lugar de vaciar y copiar todo.
- Slot de replicación — Mecanismo que garantiza que el primario no borra WAL que un consumidor aún no ha recibido. Muy útil y la causa más común de discos llenos.
- Object Lock — Retención inmutable a nivel de objeto en el almacenamiento. En modo COMPLIANCE, ni el propietario del bucket puede borrar antes de tiempo.
- 3-2-1-1-0 — Tres copias, en dos medios, una fuera de sitio, una inmutable o desconectada, y cero errores en la verificación.
- Simulacro de restauración — Restauración completa y automatizada, con verificación de contenido, ejecutada de forma periódica. Lo único que convierte «tenemos backups» en un hecho.

Ocho errores que se repiten

1. Confiar en pg_dump como estrategia única. No hay PITR posible: tu RPO es la distancia entre volcados, y el RTO en bases grandes se dispara por la reconstrucción de índices. Sirve para migrar de versión, no para recuperarte de un incidente a las cuatro de la tarde.
2. No monitorizar pg_stat_archiver. El archivado se rompe en silencio y sólo te enteras cuando intentas recuperar, que es el peor momento posible. Alerta sobre failed_count y sobre el retraso del último archivado.
3. Un archive_command que devuelve 0 sin haber copiado. Si es un script, empieza por set -euo pipefail, comprueba códigos de salida y haz que sobrescribir un fichero existente sea un error, no un acierto.
4. Borrar el completo del que cuelgan los incrementales. Con incrementales nativos la retención tiene que razonar por cadenas. Con pgBackRest, deja que repo1-retention-full lo gestione y no toques el repositorio a mano jamás.
5. Restaurar con recovery_target_action = 'promote' a la primera. Usa pause, verifica los datos y promociona después. Promocionar es irreversible y te obliga a repetir la restauración entera.
6. Guardar los backups en el mismo sitio, y con las mismas credenciales, que la base. Un snapshot en el mismo proyecto de nube, borrable por la misma cuenta, no protege contra el escenario que más daño hace.

7. No archivar los ficheros .history. Rompe cualquier recuperación que cruce líneas temporales, justo en el segundo intento.
8. No haber medido nunca el RTO. Si no sabes cuánto tarda tu restauración con el volumen de datos de hoy, no tienes un plan de recuperación: tienes una intención.

Checklist de producción

- RPO y RTO escritos y acordados con negocio, no inferidos por el equipo de infraestructura.
- wal_level = replica o superior, archive_mode = on y un archive_command que falla ruidosamente.
- archive_timeout alineado con el RPO en bases con poca escritura.
- Alertas sobre pg_stat_archiver: fallos y retraso del último archivado.
- Espacio libre de pg_wal monitorizado, con alerta antes del 80 %.
- Backup completo periódico más incrementales, con retención que respeta las cadenas.
- pg_verifybackup o pgbackrest verify como paso obligatorio del pipeline.
- Al menos una copia inmutable u offline, escrita con credenciales que no pueden borrar.
- Repositorio cifrado en reposo y clave gestionada fuera del servidor de base de datos.
- Simulacro de restauración automatizado, semanal, con comprobaciones de contenido.
- RTO medido, exportado como métrica y con alerta de tendencia.
- pg_dumpall --globals-only dentro del ciclo y validado en el simulacro.
- Runbook de recuperación escrito con los comandos exactos, ejecutado al menos una vez por alguien que no lo escribió.
- Procedimiento capaz de desplegar la misma versión menor de PostgreSQL que corría el clúster.

Preguntas frecuentes

¿Una réplica de streaming sustituye a los backups?

No. Una réplica protege contra la caída de un servidor, pero replica fielmente tus errores: el DROP TABLE llega al standby en milisegundos. Protege la disponibilidad, no la corrección. Lo que sí puedes hacer es tomar el backup base *desde* la réplica para no cargar al primario, algo que tanto pg_basebackup como pgBackRest soportan.

¿Puedo restaurar un backup físico en otra versión mayor de PostgreSQL?

No. El formato en disco no es compatible entre versiones mayores, y además debe coincidir la arquitectura. Para saltar de versión, pg_dump / pg_restore o pg_upgrade. La consecuencia operativa es importante: tu procedimiento de recuperación tiene que poder desplegar exactamente la misma versión menor que corría el clúster, así que fija esa versión en tus imágenes y guárdala junto al backup.

¿Cada cuánto conviene un backup completo?

Depende de cuánto WAL genera tu base y de tu RTO, no del calendario. La pregunta útil es: cuánto WAL tendría que reproducir si el desastre ocurriera justo antes del siguiente completo, y cuánto tarda esa reproducción.

Completo semanal con incrementales diarios es un punto de partida razonable para la mayoría; si generas cientos de gigabytes de WAL al día, acorta el ciclo.

¿Sirve un snapshot del volumen?

Sí, si es atómico sobre *todos* los volúmenes del clúster —incluidos los tablespaces separados y `pg_wal` si vive aparte— y lo coordinas con `pg_backup_start()` y `pg_backup_stop()`, guardando el `backup_label` que devuelve la segunda. Si los snapshots no son consistentes entre sí, obtienes una copia incoherente que puede arrancar y parecer sana durante días. Y seguirás necesitando archivado de WAL para hacer PITR entre snapshots.

¿Y en un servicio gestionado como RDS o Cloud SQL?

El proveedor gestiona el archivado y ofrece PITR dentro de una ventana de retención, pero eso no cierra el asunto. Sigue siendo tuya la responsabilidad de comprobar que la ventana cubre tu RPO, de exportar copias fuera de la cuenta del proveedor —para el caso del borrado accidental de la instancia o de un problema con la propia cuenta— y de probar la restauración. Además, la restauración PITR gestionada crea una instancia nueva, así que tu RTO real incluye repuntar la aplicación, revisar parámetros y recrear réplicas.

¿Puedo recuperar hasta un punto concreto que no sea una hora?

Sí. Además de `recovery_target_time` tienes `recovery_target_xid`, `recovery_target_lsn` y `recovery_target_name`. Esta última es especialmente útil: si vas a ejecutar una migración arriesgada, llama antes a `pg_create_restore_point('antes_migracion_v42')` y tendrás una marca exacta a la que volver, sin depender de relojes ni de zonas horarias.

ENGLISH

PostgreSQL Backups and PITR: WAL Archiving, Native Incremental Backups, pgBackRest and Restore Drills

There is a conversation that shows up in almost every data-loss postmortem, and it always sounds the same: "yes, we had backups." What almost never shows up is someone who can say when they last restored one, how long it took, and what was missing when it finished. That gap — between having copies and being able to operate again — is what this guide is about.

PostgreSQL gives you two families of tools that solve different problems and that people confuse constantly: logical dumps with pg_dump, and physical backups with continuous WAL archiving, which is the only thing that enables point-in-time recovery (PITR). We will cover both, when to use each, how to configure archiving without filling your disk, how the native incremental backups introduced in PostgreSQL 17 actually work, how to perform a real PITR step by step, and how pgBackRest turns all of this into something you can operate without homegrown scripts.

Start with two numbers: RPO and RTO

Before touching a single configuration line you need two figures agreed with the business, not picked by you in a spare moment.

- RPO (Recovery Point Objective): how much data you can afford to lose, measured in time. If your RPO is 5 minutes, a daily backup at 3:00 is not even close — it exposes you to losing up to 24 hours of writes.
- RTO (Recovery Time Objective): how long you can take to be serving again. This is where database size, storage speed, and — always forgotten — the time to replay the WAL accumulated since the base backup all come in.

These two numbers determine nearly everything else. An RPO measured in seconds demands continuous WAL archiving, or pg_receivewal streaming, or a replica outright. An RTO measured in minutes on a 2 TB database rules out restoring from a pg_dump, because a logical dump has to rebuild every index and that can take hours. When someone asks for "solid backups" without giving you these numbers, that is the conversation to have — not the one about which tool to use.

Logical backups: pg_dump and what it does not tell you

pg_dump exports the contents of a single database as SQL statements or in its own compressed format. It runs inside a transaction with a consistent snapshot, so the copy is coherent even if writes happen during the dump, and it does not block normal writers.

BASH

```
# The "directory" format is the only one that allows parallel dump and restore
pg_dump \
```

```
  --format=directory \
  --jobs=4 \
  --compress=zstd:3 \
  --file=/backups/shop_2026-08-27 \
  --dbname="postgresql://backup_user@db.internal:5432/shop"

# Parallel restore into an empty database
createdb shop_restored
pg_restore \
  --dbname=shop_restored \
  --jobs=4 \
  --exit-on-error \
  /backups/shop_2026-08-27
```

Two details make the difference. `--jobs` only works with the `directory` format, and it is what turns a three-hour restore into a forty-minute one, because index creation is parallelised. And `--exit-on-error` should be your default: without it, `pg_restore` carries on after a failure and can leave you with an incomplete database that looks fine.

The three classic omissions

`pg_dump` dumps *one* database. It does not include roles, passwords, or cluster-level configuration. The first time somebody restores onto a fresh server and discovers that no `GRANT` points at an existing role is usually at three in the morning:

```bash
# Global objects: roles, role memberships, tablespaces
pg_dumpall --globals-only --file=/backups/globals_2026-08-27.sql

# ALWAYS restore globals before the databases
psql --dbname=postgres --file=/backups/globals_2026-08-27.sql
```

The second omission is extensions: `pg_dump` writes `CREATE EXTENSION`, but not the binaries. If you restore into a container that does not have `pgvector` or `postgis` installed, the restore stops on the first line. The third is that a logical dump does not preserve physical state: indexes get rebuilt, so the result comes out defragmented and, on large tables, that work is the bulk of your RTO.

That said, `pg_dump` has one superpower physical backups do not: it is portable across major versions and architectures. It is the right tool for moving from PostgreSQL 16 to 18, for copying a database into staging, or for exporting a single schema. As a disaster recovery strategy on its own, it is not enough.

Continuous WAL archiving: the foundation of PITR

PostgreSQL writes every change to the write-ahead log first. If you keep a physical backup of the data directory plus *every* WAL segment generated since that moment, you can reconstruct the state of the database at any later instant. That is PITR, and it is what saves you from the `DELETE` without a `WHERE` at 16:42.

```ini
# postgresql.conf
wal_level = replica
archive_mode = on
archive_timeout = 300s       # force a segment switch every 5 min: your RPO ceiling

# The command MUST fail if the destination already exists
archive_command = 'test ! -f /wal_archive/%f && cp --no-clobber %p /wal_archive/%f'
```

The detail that makes or breaks the whole system

The contract for archive_command is simple and merciless: if it returns 0, PostgreSQL considers the segment archived and is free to recycle it. If it returns anything else, PostgreSQL retries indefinitely and *deletes nothing*. Two of the most expensive failures in this area follow directly from that.

- A command that returns 0 without copying. A cp to a dead NFS mount point can succeed by writing into the empty local directory underneath it. Months later the archive has holes and PITR is impossible. That is why the example uses test ! -f: if the file is already there, the command fails and you find out, instead of silently overwriting a different segment that happens to share a name — something that genuinely happens after a mishandled recovery.
- A command that fails and nobody watches. If the destination fills up, PostgreSQL accumulates WAL in pg_wal until the server disk is exhausted, and then the engine stops. A misconfigured backup takes down the production it was meant to protect.

The query that should be on your dashboard from day one:

```sql
SELECT archived_count,
    last_archived_wal,
    last_archived_time,
    failed_count,
    last_failed_wal,
    last_failed_time,
    now() - last_archived_time AS archiving_lag
FROM pg_stat_archiver;
```

Alert on two things: failed_count growing, and archiving_lag comfortably exceeding your archive_timeout. The second one catches archiving that has quietly stalled without ever returning an error.

Since PostgreSQL 15 there is also archive_library, which loads an in-process module instead of spawning a shell per segment. It is considerably more efficient when you generate a lot of WAL. The basic_archive module in contrib is the reference example, but in practice, if you use a library it will normally be the one shipped by your backup tool.

The base backup with pg_basebackup

With archiving running you need a physical copy of the cluster to serve as a starting point. pg_basebackup

connects over the replication protocol and copies the whole data directory, coordinating with the server to mark the start and end of the backup.

```bash
pg_basebackup \
  --pgdata=/backups/base/2026-08-27 \
  --format=tar \
  --compress=server-zstd:6 \
  --wal-method=stream \
  --checkpoint=fast \
  --progress \
  --verbose \
  --dbname="postgresql://replicator@db.internal:5432/postgres"
```

The options that actually matter:

- --wal-method=stream opens a second connection that receives the WAL generated during the backup. Without it (fetch mode), if archiving lags or fails while you are copying, the base backup is useless because segments are missing. With stream the copy is self-contained.
- --checkpoint=fast requests an immediate checkpoint instead of waiting for the normal pace. You start sooner at the cost of an I/O spike; on a tightly tuned server, decide whether that trade is worth it.
- --compress=server-zstd:6 compresses on the server: you save network bandwidth in exchange for CPU on the primary. If the primary is CPU-bound, use client-zstd instead.

If you prefer manual control — for instance to coordinate a volume snapshot — the low-level API changed in PostgreSQL 15: exclusive mode was removed and the functions are now pg_backup_start() and pg_backup_stop(). This is worth knowing because a great many scripts and tutorials still use pg_start_backup(), which simply no longer exists.

```sql
-- The session must stay open for the entire backup
SELECT pg_backup_start(label => 'zfs_snapshot', fast => true);

-- ... take the volume snapshot here ...

-- Returns the end LSN, the contents of backup_label and the tablespace map.
-- You must store backup_label alongside the snapshot or the copy will not start.
SELECT * FROM pg_backup_stop(wait_for_archive => true);
```

Verify the backup at creation time

```bash
# Checks the backup against its backup_manifest: files, sizes and checksums,
# and that the WAL needed to recover it is present and readable.
pg_verifybackup --progress /backups/base/2026-08-27

# PostgreSQL 18 added verification of tar-format backups
```

```
pg_verifybackup --format=tar /backups/base/2026-08-27
```

This is no substitute for a test restore, but it catches transport corruption and truncated files in seconds. It should be a mandatory step in your backup script: if pg_verifybackup returns non-zero, the backup does not get marked as good.

Native incremental backups (PostgreSQL 17 and later)

Up to PostgreSQL 16, incremental backups meant reaching for pgBackRest or Barman. Since 17, the server itself can summarise which blocks the WAL has touched, and pg_basebackup can copy only those.

```ini
# postgresql.conf — requires a restart
summarize_wal = on
wal_summary_keep_time = '10d'   # the default; raise it if your chain is longer
```

With summarize_wal = on, a background worker starts writing summaries into pg_wal/summaries. Those summaries are what let the server know which blocks changed between two moments without re-reading the entire WAL.

```bash
# Sunday: full backup
pg_basebackup -D /backups/sun --checkpoint=fast --wal-method=stream

# Monday: incremental against the previous backup's manifest
pg_basebackup -D /backups/mon \
  --incremental=/backups/sun/backup_manifest \
  --wal-method=stream

# Tuesday: incremental against Monday's (the chain grows)
pg_basebackup -D /backups/tue \
  --incremental=/backups/mon/backup_manifest \
  --wal-method=stream
```

An incremental backup **cannot be restored directly**. You have to reconstruct a synthetic full backup with pg_combinebackup, handing it the entire chain in order:

```bash
pg_combinebackup \
  /backups/sun /backups/mon /backups/tue \
  --output=/var/lib/postgresql/18/restored
```

Here is the trap, and it is worth internalising before you write a retention policy: **the chain is indivisible**. If your cleanup script deletes Sunday's full backup because it is "over seven days old," the six incrementals hanging off it become unreadable garbage. Any retention scheme has to reason about complete chains, not individual

backups. And if WAL summaries for the whole interval are unavailable — because wal_summary_keep_time was too short — the incremental fails at creation time, which is at least a loud and timely failure.

For debugging, pg_walsummary lets you inspect a summary file and see exactly which blocks are considered modified.

PITR step by step

A realistic scenario: at 16:42 somebody runs an UPDATE without a WHERE against orders. It is 17:05 and you want the state as of 16:41:30.

1. Stop the server and preserve what is there

```bash
sudo systemctl stop postgresql

# Never overwrite the damaged directory: it may hold unarchived WAL
sudo mv /var/lib/postgresql/18/main /var/lib/postgresql/18/main.broken
```

That mv is not paranoia. The current directory holds the most recent WAL segments, which have probably not reached the archive yet. Lose them and your latest possible recovery point rolls back to the last archived segment — taking with it some of the good writes that happened after the mistake.

2. Restore the base backup

```bash
sudo -u postgres mkdir -p /var/lib/postgresql/18/main
sudo -u postgres chmod 0700 /var/lib/postgresql/18/main
sudo -u postgres tar -xf /backups/base/2026-08-27/base.tar.zst \
  -C /var/lib/postgresql/18/main

# Empty pg_wal: WAL will come from the archive, not from the copy
sudo -u postgres rm -rf /var/lib/postgresql/18/main/pg_wal/*
```

3. Configure the recovery target

```ini
# postgresql.conf inside the restored directory
restore_command = 'cp /wal_archive/%f %p'
recovery_target_time = '2026-08-27 16:41:30+02'
recovery_target_inclusive = off      # stop BEFORE that mark, not including it
recovery_target_action = 'pause'     # do NOT promote: verify first
```

```bash
# This empty file is what tells PostgreSQL to start in recovery mode
sudo -u postgres touch /var/lib/postgresql/18/main/recovery.signal
```

`recovery_target_action = 'pause'` is the setting that prevents the most grief. With `promote`, PostgreSQL reaches the target, creates a new timeline and opens the database for writes; if you overshot, you have just cemented the mistake and you start over. With `pause` the engine waits in read-only mode and you can inspect the data calmly.

4. Start, verify, then commit

```bash
sudo systemctl start postgresql
tail -f /var/log/postgresql/postgresql-18-main.log
```

```sql
-- Still in recovery?
SELECT pg_is_in_recovery(), pg_last_wal_replay_lsn();

-- Check the specific data that triggered the recovery
SELECT id, status, total, updated_at
FROM orders
WHERE updated_at > '2026-08-27 16:00:00+02'
ORDER BY updated_at DESC
LIMIT 20;
```

If the data is right, promote. If you overshot or stopped too early, shut the server down, adjust `recovery_target_time`, restore the base backup again and repeat — there is no rewinding a recovery in progress.

```sql
SELECT pg_promote(wait => true);
```

Timelines: the part that confuses everyone

Every time you promote after a recovery, PostgreSQL creates a new *timeline* and writes a `.history` file to the archive. It exists precisely so you can make several attempts without destroying the original history: timeline 1 is still intact.

The practical consequence is that if you later need to recover to an instant *before* that promotion, you have to tell PostgreSQL which line to follow, because by default it follows the latest one it finds in the archive:

```ini
```

```
recovery_target_timeline = '1'    # or 'current', or a specific ID
```

And a common oversight: your **archive_command** must archive **.history** files too. If you filter on the 24-character segment pattern, cross-timeline recoveries will fail on the second attempt — exactly when you are already stressed. Using **%f** as-is, like in the examples, covers it.

pgBackRest: when you stop writing scripts

Everything above works, and it is worth understanding because it is what sits underneath. But maintaining retention, verification, compression, encryption, object-storage upload and parallelism by hand ends up being a project of its own. pgBackRest (2.59.1 as of August 2026) solves all of that and adds things you will not build yourself: block-level incremental backup, delta restore, per-file checksums revalidated during restore, and page checksum validation during the backup itself.

INI

```
; /etc/pgbackrest/pgbackrest.conf
[global]
repo1-path=/var/lib/pgbackrest
repo1-type=s3
repo1-s3-bucket=production-backups
repo1-s3-region=eu-west-1
repo1-s3-endpoint=s3.eu-west-1.amazonaws.com
repo1-cipher-type=aes-256-cbc
repo1-cipher-pass=KEY_INJECTED_FROM_YOUR_SECRETS_MANAGER
repo1-retention-full=4
repo1-retention-diff=7
repo1-bundle=y          ; bundles small files: crucial with object storage
repo1-block=y           ; block-level incremental backup
process-max=4
compress-type=zst
compress-level=6
start-fast=y
archive-async=y
spool-path=/var/spool/pgbackrest
log-level-console=info
log-level-file=detail

[production]
pg1-path=/var/lib/postgresql/18/main
pg1-port=5432
```

INI

```
# postgresql.conf
archive_mode = on
archive_command = 'pgbackrest --stanza=production archive-push %p'
```

BASH

```
# Once, when setting up the stanza
pgbackrest --stanza=production stanza-create

# Validates configuration, permissions and archiving end to end.
# Worth a slot in your weekly cron.
pgbackrest --stanza=production check

# Sunday
pgbackrest --stanza=production --type=full backup
# Every other day
pgbackrest --stanza=production --type=incr backup

# Repository audit: checksums across everything stored
pgbackrest --stanza=production verify

# PITR with delta restore
pgbackrest --stanza=production restore \
 --delta \
 --type=time \
 --target="2026-08-27 16:41:30+02" \
 --target-action=pause
```

That `--delta` is the difference between an RTO of hours and one of minutes on large databases: it compares checksums against whatever is already in the data directory and copies only the files that differ, instead of wiping it and pulling everything down again. And `check` is the cheapest way to know that archiving really works end to end, rather than that the configuration merely looks right.

3-2-1-1-0 and the scenario almost nobody plans for

The classic 3-2-1 rule (three copies, two media, one offsite) stopped being enough once ransomware started hunting for and encrypting backup repositories themselves. The version in use today is 3-2-1-1-0: it adds **one immutable or offline copy** and **zero verification errors**.

In the cloud that means enabling Object Lock on the repository bucket, in *compliance* mode, with a retention period at least as long as your recovery window. And it means the credential used by the database server must be able to write new objects but not delete or overwrite them. If your production server has permission to empty the backup bucket, your backups have exactly the same attack surface as your database.

```JSON

{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Sid": "WriteNewObjectsOnly",
   "Effect": "Allow",
   "Action": ["s3:PutObject", "s3:GetObject", "s3:ListBucket"],
   "Resource": [
    "arn:aws:s3:::production-backups",
    "arn:aws:s3:::production-backups/*"
   ]
  },
```

```
    {
      "Sid": "NeverDeleteOrWeakenRetention",
      "Effect": "Deny",
      "Action": [
        "s3:DeleteObject",
        "s3:DeleteObjectVersion",
        "s3:PutBucketVersioning",
        "s3:PutObjectRetention",
        "s3:BypassGovernanceRetention"
      ],
      "Resource": [
        "arn:aws:s3:::production-backups",
        "arn:aws:s3:::production-backups/*"
      ]
    }
  ]
}
```

Retention expiry is then executed by a different principal, with its own credential, from somewhere that is not the database server. It is slightly more setup work, and it is the difference between losing a database and losing the company.

The restore drill: the only thing that proves anything

A backup has two states: restored, or hypothetical. The only way out of the hypothesis is to restore for real, automatically and regularly, and measure how long it takes. That number is your actual RTO — not the one in the architecture document.

PYTHON

```
#!/usr/bin/env python3
"""Restore drill: restores the latest backup into an ephemeral instance,
checks integrity and content, and publishes metrics.

Designed for a weekly cron. Returns non-zero if anything fails, so your
alerting system picks it up.
"""

import subprocess
import sys
import time
from datetime import datetime, timezone

STANZA = "production"
DATA_DIR = "/var/lib/postgresql/drill"
PORT = "55432"
METRICS = "/var/lib/node_exporter/textfile/pgbackrest_drill.prom"


def run(cmd: list[str]) -> subprocess.CompletedProcess:
    print("$ " + " ".join(cmd), flush=True)
    return subprocess.run(cmd, check=True, text=True, capture_output=True)
```

```
def write_metrics(duration: float, success: int) -> None:
    with open(METRICS, "w") as f:
        f.write("# TYPE pgbackrest_restore_drill_seconds gauge\n")
        f.write("pgbackrest_restore_drill_seconds " + str(round(duration, 1)) + "\n")
        f.write("# TYPE pgbackrest_restore_drill_success gauge\n")
        f.write("pgbackrest_restore_drill_success " + str(success) + "\n")


def main() -> int:
    started = time.monotonic()
    print("Drill started:", datetime.now(timezone.utc).isoformat())

    # 1. Restore the most recent backup up to the last consistent point
    run([
        "pgbackrest", "--stanza=" + STANZA, "restore",
        "--pg1-path=" + DATA_DIR,
        "--type=immediate",
        "--target-action=promote",
        "--delta",
    ])

    # 2. Start the restored instance on an isolated port
    run(["pg_ctl", "-D", DATA_DIR, "-o", "-p " + PORT, "-w", "-t", "900", "start"])

    try:
        # 3. Wait for it to leave recovery
        for _ in range(180):
            r = subprocess.run(
                ["psql", "-p", PORT, "-tAc", "SELECT pg_is_in_recovery()"],
                text=True, capture_output=True,
            )
            if r.stdout.strip() == "f":
                break
            time.sleep(5)
        else:
            print("ERROR: instance never left recovery", file=sys.stderr)
            write_metrics(time.monotonic() - started, 0)
            return 1

        # 4. Check CONTENT, not just that the process starts
        checks = {
            "user_tables": (
                "SELECT count(*) FROM pg_class c "
                "JOIN pg_namespace n ON n.oid = c.relnamespace "
                "WHERE c.relkind = 'r' "
                "AND n.nspname NOT IN ('pg_catalog', 'information_schema')"
            ),
            "orders_last_24h": (
                "SELECT count(*) FROM orders "
                "WHERE created_at > now() - interval '24 hours'"
            ),
            "invalid_indexes": "SELECT count(*) FROM pg_index WHERE NOT indisvalid",
            "login_roles": "SELECT count(*) FROM pg_roles WHERE rolcanlogin",
        }

        res = {}
        for name, sql in checks.items():
            out = run(["psql", "-p", PORT, "-tAc", sql])
            res[name] = int(out.stdout.strip())
```

```
        print("  " + name + " = " + str(res[name]))

    failures = []
    if res["user_tables"] == 0:
        failures.append("restored database has no user tables")
    if res["orders_last_24h"] == 0:
        failures.append("no orders in the last 24h: the backup is stale")
    if res["invalid_indexes"] > 0:
        failures.append("invalid indexes present after restore")
    if res["login_roles"] < 2:
        failures.append("roles missing: check pg_dumpall --globals-only")

    if failures:
        for f in failures:
            print("ERROR: " + f, file=sys.stderr)
        write_metrics(time.monotonic() - started, 0)
        return 1

finally:
    subprocess.run(["pg_ctl", "-D", DATA_DIR, "-m", "immediate", "stop"])

duration = time.monotonic() - started
print("OK: drill completed in " + str(round(duration)) + "s")
write_metrics(duration, 1)
return 0


if __name__ == "__main__":
    sys.exit(main())
```

Note the orders_last_24h check. Verifying only that the instance starts lets through the most insidious failure of all: a backup that restores without a single error but contains data from three weeks ago, because the job has been failing silently that whole time. And exporting the duration as a metric lets you alert when your real RTO creeps toward the committed one, instead of discovering it mid-incident.

YAML

```
# Prometheus alerts for the drill and for archiving
groups:
  - name: postgres-backups
    rules:
      - alert: RestoreDrillFailed
        expr: pgbackrest_restore_drill_success == 0
        for: 5m
        labels: { severity: critical }
        annotations:
          summary: "The weekly restore drill failed"

      - alert: RestoreDrillNotRunning
        expr: time() - node_textfile_mtime_seconds{file="pgbackrest_drill.prom"} > 10 * 24 * 3600
        labels: { severity: warning }
        annotations:
          summary: "More than 10 days without a restore drill"

      - alert: MeasuredRTONearTarget
        expr: pgbackrest_restore_drill_seconds > 0.8 * 3600
```

```
    labels: { severity: warning }
    annotations:
      summary: "Measured RTO exceeds 80% of the 1-hour objective"

  - alert: WALArchivingFailing
    expr: increase(pg_stat_archiver_failed_count[15m]) > 0
    labels: { severity: critical }
    annotations:
      summary: "archive_command is failing: PITR is breaking right now"
```

Compression and parallelism: making the backup fit the window

So far, backups have been a question of correctness. Past a certain size they also become a question of the clock: if your full copy takes nine hours and your low-traffic window is four, the problem is no longer whether the backup is valid, it is that it never finishes when it should.

There are two levers, and it pays to understand what each one moves. Compression reduces what travels over the network and what sits in the repository, in exchange for CPU. Parallelism reduces wall-clock time, in exchange for CPU and concurrent I/O. Since PostgreSQL 15, pg_basebackup can compress on the server or on the client, and that choice decides where the pain lands.

BASH

```
# CLIENT-side compression: the server sends data uncompressed.
# You save disk but not bandwidth. Useful when production is tight on CPU.
pg_basebackup -h db-prod -D /backup/base -Ft \
  --compress=client-zstd:level=3 -Xstream -P

# SERVER-side compression: compress before sending.
# You save bandwidth and disk; you pay CPU on the production machine.
# workers=N spreads zstd across several threads.
pg_basebackup -h db-prod -D /backup/base -Ft \
  --compress=server-zstd:level=9,workers=4 -Xstream -P
```

The question almost nobody asks before choosing is how much throughput the link actually gives. There is a clean way to measure it without writing a single byte: `--target=blackhole` discards the contents and shows you the real rate. With one detail that trips up everyone's first attempt: because WAL streaming is implemented by the client rather than the server, blackhole is incompatible with `-Xstream`, which is the default. You have to ask for `-Xnone` or `-Xfetch` explicitly.

BASH

```
# Real backup throughput, storing nothing.
time pg_basebackup -h db-prod --target=blackhole -Xnone -P

# Repeat with server compression: if the time drops, the network was your
# bottleneck; if it rises, CPU was. Two runs save you a week of guessing.
time pg_basebackup -h db-prod --target=blackhole -Xnone \
  --compress=server-zstd:level=9,workers=4 -P
```

On the logical side parallelism pays off even more, but only with the directory format. pg_dump -Fd -j N dumps several tables at once, and pg_restore -j N rebuilds indexes and constraints in parallel, which is where most of the time in a logical restore goes.

```bash
# Parallel dump: only the directory format (-Fd) accepts -j
pg_dump -h db-prod -d shop -Fd -j 8 -f /backup/shop.dir

# Parallel restore, with more memory for building indexes.
# PGOPTIONS applies to this session only: you do not touch server config.
PGOPTIONS="-c maintenance_work_mem=2GB" \
  pg_restore -d shop_restored -j 8 /backup/shop.dir
```

One limit worth internalising before sizing anything: pg_dump and pg_restore parallelise *across* tables, never *within* a table. If 70% of your database is a single fact table, raising -j from 8 to 16 buys you nothing: at the end of the dump one process will be grinding through that table while the other fifteen watch. When that is your profile, the answer is not more parallelism, it is partitioning the table or moving to physical backups for good.

The backup that faithfully copies your corrupted data

There is one failure that appears on almost no checklist and that voids everything above: a flawless backup of a database that was already corrupt. Archiving works, pg_verifybackup passes, the drill brings the instance up without a complaint, and yet page 4812 of your orders table has spent three weeks holding bytes no process wrote on purpose. By the time you find out, all seven of your copies contain exactly the same damage.

The first defence is old and until recently shipped disabled: data checksums. With checksums on, PostgreSQL computes a digest of every page as it writes it and verifies it on read; if it does not match, the read fails with an explicit error instead of quietly handing you garbage. **PostgreSQL 18 changed the initdb default and now enables them**; you opt out deliberately with --no-data-checksums. If your cluster was born earlier, the odds are you have them off and do not know it.

```sql
-- Are they on?
SHOW data_checksums;

-- Cumulative failure counter per database.
-- Any non-zero value is an open incident, not a metric.
SELECT datname, checksum_failures, checksum_last_failure
FROM pg_stat_database
WHERE checksum_failures > 0;
```

Enabling them on an existing cluster requires stopping the server: pg_checksums --enable rewrites every page and needs the cluster shut down and cleanly closed. On large databases that is hours. The good news is that this has an expiry date: PostgreSQL 19 adds online enabling and disabling via a background worker that dirties pages so they pick up their checksum on the next write, with no downtime window.

```bash
# The cluster must be STOPPED and have shut down cleanly.
pg_ctl -D /var/lib/postgresql/18/main stop -m fast
pg_checksums --check -D /var/lib/postgresql/18/main   # verify first
pg_checksums --enable --progress -D /var/lib/postgresql/18/main
pg_ctl -D /var/lib/postgresql/18/main start
```

Checksums catch damage in storage, but they cannot see logical corruption: line pointers aiming where they should not, tuples whose xmin is later than their xmax, indexes promising rows that no longer exist. That is what the amcheck extension and its command-line wrapper **pg_amcheck** are for: they walk tables and indexes and return one row per problem found instead of aborting on the first. The documentation is refreshingly honest about its scope: amcheck can prove corruption is present, never that it is absent. In exchange, no corruption error it reports is ever a false positive.

```bash
# --install-missing creates the amcheck extension wherever it is absent.
# --heapallindexed verifies every visible heap tuple is present in the index.
# --parent-check adds verification of the B-tree parent/child structure.
pg_amcheck --database=shop \
  --heapallindexed --parent-check \
  --jobs=4 --progress --install-missing
```

The right place to run this is not production: it is the throwaway instance you already spin up for the restore drill. There the CPU is yours, **--parent-check** blocks nobody, and you are checking exactly the bytes you would use in a real disaster. A weekly **pg_amcheck** against the restored copy turns your drill into something that verifies two things at once: that you can restore, and that what you restore is healthy.

Finding the exact instant: pg_waldump, XID and LSN

The PITR section assumed you know what time to go back to. In a real incident you almost never do. You know somebody ran something between 14:25 and 14:40, that the ticket was opened at 14:52, and that the clock on the laptop the query was fired from was two minutes fast. Restoring to **recovery_target_time = '2026-08-27 14:30:00'** is a bet: overshoot slightly and you drag the disaster into the fresh copy; undershoot and you throw away minutes of legitimate work from everyone else.

There is an alternative that does not require guessing. The WAL contains a record of every change with its transaction id and its LSN, and **pg_waldump** reads it in plain text. You can locate the offending transaction and stop recovery just before it.

```bash
# 1. Narrow the window. Find which segments cover the time range.
ls -l --time-style=full-iso /archive/wal/ | awk '$6 >= "2026-08-27" '

# 2. Dump the records in those segments and search for the operation.
#    -s and -e bound the scan by LSN once you have an approximation.
```

```
pg_waldump -p /archive/wal 000000010000004A000000C1 000000010000004A000000C4 \
  | grep -E 'DELETE|TRUNCATE|DROP' \
  | head -50

# Typical output (trimmed):
# rmgr: Heap  len: 54  tx: 918273645  lsn: 4A/C2A1F038  desc: DELETE off: 12 ...
```

With that **tx** in hand, recovery stops being approximate. **recovery_target_xid** points at the exact transaction and **recovery_target_inclusive = false** tells PostgreSQL to stop *before* applying it, not after.

```ini
# postgresql.conf on the recovery instance
restore_command = 'cp /archive/wal/%f %p'

# Option A: stop right before the offending transaction.
recovery_target_xid = '918273645'
recovery_target_inclusive = off

# Option B: stop at a specific LSN (the most precise of all).
# recovery_target_lsn = '4A/C2A1F038'

# Either way: pause on reaching the target, do not promote blindly.
recovery_target_action = pause
```

The five recovery targets are not interchangeable, and it is worth being clear on them before an incident rather than during one. **recovery_target_time** is convenient but depends on the commit timestamp being what you think it is. **recovery_target_xid** is exact for an identified change. **recovery_target_lsn** is the most surgical and the only one with no ambiguity at all. **recovery_target_name** requires that somebody called **pg_create_restore_point()** before the disaster, which in practice only happens if you put it in your deployment procedure. And **recovery_target = 'immediate'** stops as soon as the base backup reaches a consistent state, which is what you want when the goal is not recovering data but proving the backup works.

One piece of advice that is expensive to learn the hard way: create a named restore point right before every schema migration. It costs one line in the pipeline and turns an anxious PITR into a single-sentence instruction.

```sql
-- In the pre-step of your migration, inside the deployment pipeline:
SELECT pg_create_restore_point('pre-migration-2026-08-27-invoices');

-- And if something goes wrong, the recovery target is no longer a guess:
--   recovery_target_name = 'pre-migration-2026-08-27-invoices'
```

I only need one table back: partial restore

The most common case of all is not losing the data centre. It is somebody running an **UPDATE** without a **WHERE** against a 40,000-row table while the rest of the database, all three terabytes of it, stays perfectly healthy and keeps serving traffic. And here you run headfirst into the fundamental limitation of physical backups: they

are all or nothing. The base backup plus WAL reconstruct an entire cluster at an entire instant. There is no `--only-table`.

The solution is not a trick, it is a procedure, and it is worth having written down before you need it. You restore a parallel, throwaway instance on another port, PITR it to just before the disaster, extract what you need with logical tools, and inject that into production.

BASH

```
#!/usr/bin/env bash
set -euo pipefail

# --- Surgical restore of a single table ---
PORT=5433
DIR=/srv/rescue/pgdata
TARGET="2026-08-27 14:29:50+02"

# 1. Throwaway instance from the repository, on another port.
pgbackrest --stanza=main --delta \
  --pg1-path="$DIR" \
  --type=time --target="$TARGET" \
  --target-action=pause restore

# 2. Start it isolated: no network, no archiving, nobody mistakes it for prod.
pg_ctl -D "$DIR" -o "-p $PORT -c listen_addresses=localhost -c archive_mode=off" start

# 3. Wait for it to reach the target, then promote ONLY this instance.
until psql -p "$PORT" -Atc "SELECT pg_is_in_recovery()" | grep -q '^f$'; do
  psql -p "$PORT" -Atc "SELECT pg_wal_replay_resume()" >/dev/null 2>&1 || true
  sleep 2
done

# 4. Extract only what you need.
pg_dump -p "$PORT" -d shop -t public.product_prices \
  --data-only -Fc -f /tmp/prices_rescued.dump

# 5. Load it into production in a SEPARATE schema. Never straight on top.
psql -h db-prod -d shop -c "CREATE SCHEMA IF NOT EXISTS rescue;"
```

Step 5 is where incidents are lost. The temptation is to restore the table on top of the production one and be done sooner; the problem is that two hours have passed between the disaster and the rescue, and there have been legitimate writes to that same table. Loading the old copy on top does not repair, it overwrites. The right move is to land the rescued data in a separate schema and reconcile with SQL, deciding explicitly what wins each conflict.

SQL

```
-- Explicit reconciliation: restore only the rows the accident touched
-- and that nobody has legitimately modified since.
UPDATE public.product_prices p
SET    price_cents = r.price_cents,
       updated_at  = now()
FROM   rescue.product_prices r
WHERE  p.id = r.id
```

```
  AND  p.price_cents IS DISTINCT FROM r.price_cents
  AND  p.updated_at < TIMESTAMPTZ '2026-08-27 14:29:50+02';

-- And check the blast radius BEFORE you commit, not after.
-- (run the UPDATE inside an open transaction and review the row count)
```

In the logical world partial restore is far more direct, and `-L` is worth knowing: pg_restore can list the table of contents of a dump, let you edit that list by hand, and restore exactly the objects you leave in it. It is the right tool when you need five tables and their sequences, but not the other forty.

```bash
# 1. Extract the dump's table of contents
pg_restore -l /backup/shop.dump > /tmp/toc.txt

# 2. Edit /tmp/toc.txt and comment out with ';' everything you do not want
grep -E 'TABLE DATA public (orders|order_lines)' /tmp/toc.txt > /tmp/selection.txt

# 3. Restore only that selection, in dependency order
pg_restore -d shop_restored -L /tmp/selection.txt /backup/shop.dump
```

The replication slots that fill pg_wal and take the server down

This looks like a replication topic rather than a backup one, but it belongs here for a very concrete reason: the most common way for a backup system to kill production is not by failing, it is by retaining. If `archive_command` starts failing, PostgreSQL will not delete segments it has not archived yet. If there is also an abandoned replication slot, it will not delete the ones that slot has not consumed either. `pg_wal` grows, the disk fills, and the server goes read-only or simply falls over. The backup, which existed to protect you from an outage, caused one.

The defence is two parameters and one query. `max_slot_wal_keep_size` (default -1, meaning unlimited) caps how much WAL a slot may retain before PostgreSQL invalidates it and moves on. That is a deliberate trade: you would rather break a replica than lose the primary. And `idle_replication_slot_timeout` , which arrived in PostgreSQL 18, invalidates slots nobody has connected to for too long, which is the exact signature of the slot somebody created for a test and forgot to drop.

```ini
# postgresql.conf
# A slot may not retain more than 64 GB of WAL. Past that it is invalidated.
max_slot_wal_keep_size = 64GB

# PostgreSQL 18+: invalidate slots idle for more than 24 hours.
# Invalidation happens at checkpoint, so it can take up to
# checkpoint_timeout to actually materialise.
idle_replication_slot_timeout = 24h

# WAL kept for replicas WITHOUT a slot (an independent mechanism).
wal_keep_size = 8GB
```

The query that belongs on your dashboard measures two different things people tend to conflate: how far behind each slot is in bytes, and the health of archiving. A slot 200 GB behind and an archiver that has not advanced in forty minutes are the same incident seen from two angles.

```sql
-- Lag per slot, in bytes and human-readable. wal_status tells the story:
-- reserved (fine), extended (growing), unreserved (at risk), lost (invalidated).
SELECT slot_name,
       slot_type,
       active,
       wal_status,
       pg_size_pretty(
         pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn)
       ) AS lag,
       safe_wal_size IS NOT NULL AS has_cap
FROM   pg_replication_slots
ORDER  BY pg_wal_lsn_diff(pg_current_wal_lsn(), restart_lsn) DESC;

-- Archiver health: if failed_count climbs or last_archived_time freezes,
-- you have minutes, not hours, before the disk becomes the problem.
SELECT archived_count,
       last_archived_wal,
       last_archived_time,
       failed_count,
       last_failed_wal,
       last_failed_time,
       now() - last_archived_time AS since_last_archive
FROM   pg_stat_archiver;
```

One note on max_slot_wal_keep_size worth understanding before you set it: invalidating a slot is not free. If the slot belonged to a streaming replica, that replica will have to be rebuilt from a base backup; if it belonged to a logical replication subscription or a CDC connector such as Debezium, the continuity point is gone and you have to reseed. The parameter does not avoid the pain, it chooses who suffers it. And that choice — a broken replica versus a downed primary — is almost always the right one, provided you have an alert that warns you *before* the cap is reached rather than a discovery after the fact.

pg_rewind: bringing the old primary back without copying three terabytes

After a PITR or a failover you are left with an awkward corpse: the old server, holding valid data up to the divergence point and garbage after it. The obvious option is to wipe it and rebuild from a full base backup. On a three-terabyte database that is hours of network and disk to recover a machine that differs from the new primary by a few gigabytes.

pg_rewind does exactly that: it compares the timeline histories of source and target, locates the divergence point, and copies only the blocks that changed since. The result is a data directory equivalent to a base backup of the source, obtained in a fraction of the time.

It has three prerequisites that must be in place beforehand, because they cannot be turned on after the fact on a cluster that has already diverged. The target needs wal_log_hints = on or data checksums enabled, so pg_rewind can tell which blocks changed. It needs full_page_writes on, which is the default and which you

should never disable. And it needs the WAL segments reaching back to the divergence point in its pg_wal; if the old server ran for a long time afterwards those segments may be gone, and that is where -c comes in, fetching them from the archive.

```bash
# The target MUST be stopped and have shut down cleanly.
pg_ctl -D /var/lib/postgresql/18/main stop -m fast

# First of all: a copy of the directory, even a compressed one.
# pg_rewind is destructive and removes any chance of forensic analysis.
tar -I 'zstd -3' -cf /rescue/pgdata-pre-rewind.tar.zst \
  -C /var/lib/postgresql/18 main

# Rewind against the new primary.
#   -c / --restore-target-wal: fetch missing WAL from the archive.
#   --dry-run: print what it would do without touching anything. Always use it first.
pg_rewind --target-pgdata=/var/lib/postgresql/18/main \
  --source-server="host=db-new-primary port=5432 user=rewind dbname=postgres" \
  --restore-target-wal \
  --progress --dry-run

# If the output looks right, repeat without --dry-run.
```

After the rewind the directory is not ready: pg_rewind leaves the data but not the standby configuration. You have to create standby.signal and point primary_conninfo at the new primary. And it is worth verifying the resulting timeline before calling anything done.

```bash
touch /var/lib/postgresql/18/main/standby.signal
cat >> /var/lib/postgresql/18/main/postgresql.auto.conf <<'EOF'
primary_conninfo = 'host=db-new-primary port=5432 user=replicator application_name=db-old'
restore_command  = 'cp /archive/wal/%f %p'
EOF

pg_ctl -D /var/lib/postgresql/18/main start

# Check it is following the primary and which timeline it is on.
psql -Atc "SELECT pg_is_in_recovery(), pg_last_wal_replay_lsn();"
psql -h db-new-primary -Atc \
  "SELECT application_name, state, sync_state,
          pg_size_pretty(pg_wal_lsn_diff(sent_lsn, replay_lsn)) AS lag
   FROM pg_stat_replication;"
```

And the warning that accounts for half the mailing-list threads on this topic: **pg_rewind is not a backup tool**. It protects you from nothing, stores nothing, and replaces not one element of this article. It is a re-attachment optimisation, and its price is that it destroys the old server's state. If that old server is the only copy you hold of the data as it existed just before the incident — because the PITR was performed on it, for instance — rewinding it is the last bad decision of a day that was already going badly. Hence the tar at the top.

Retention: the arithmetic that decides what your repository costs

Retention is the decision most people make by omission and that costs the most, in both directions. Retaining too little means discovering that the corruption you found today started five weeks ago and your oldest copy is four weeks old. Retaining too much means a storage bill that grows unwatched until somebody in finance asks about it.

The starting point is a calculation you can do in five minutes. You need three numbers: the compressed size of a full backup, the daily WAL volume, and the daily change rate, which is what determines incremental size.

```sql
-- Database size (approximates the uncompressed full)
SELECT pg_size_pretty(pg_database_size(current_database()));

-- WAL volume over the last 24h, from the archiver counter.
-- Each segment is 16 MB by default.
SELECT archived_count,
       pg_size_pretty(archived_count::bigint * 16 * 1024 * 1024) AS wal_total,
       stats_reset,
       now() - stats_reset AS observed_window
FROM   pg_stat_archiver;
```

With those numbers, a policy of one weekly full, daily differentials and continuous WAL over a 500 GB database that compresses to 120 GB, changes 4% a day and generates 90 GB of WAL daily works out like this: four retained fulls are 480 GB, six differentials per week at roughly 29 GB each come to about 700 GB over a month, and thirty days of WAL is 2.7 TB. Nearly four terabytes, of which 70% is WAL. That split is what usually surprises people, and it is why archive retention deserves as much attention as backup retention.

```ini
[global]
repo1-path=/var/lib/pgbackrest
repo1-cipher-type=aes-256-cbc

# Retain 4 full backups. When one expires, pgBackRest automatically
# removes the differentials and incrementals that hung off it.
repo1-retention-full=4
repo1-retention-full-type=count

# Differentials: 6 per full (one a day between weekly fulls).
repo1-retention-diff=6

# THE LINE MOST PEOPLE FORGET.
# Without it, pgBackRest keeps ALL the WAL for every retained backup.
# With it, it only keeps the WAL needed to PITR across the N most recent
# fulls; older ones remain restorable to their creation instant, but no
# longer to an intermediate point in time.
repo1-retention-archive=2
repo1-retention-archive-type=full

# Expire is expensive on large S3 repositories: decouple it from backup
# so a slow delete does not stretch your window.
```

```
repo1-retention-history=365
```

The repo1-retention-archive trap deserves its own sentence because it works backwards from intuition. If you do not configure it, you do not lose WAL: you accumulate all of it across the full retention window and pay for it. If you configure it too aggressively, you lose PITR capability on older backups with nothing warning you; the backup still shows up in pgbackrest info, still restores, but only to the exact instant it was taken. That is a silent degradation of your historical RPO, and you only discover it the day somebody asks to go back to a Tuesday three weeks ago.

BASH

```
# Decouple expire from backup: the backup should not wait
# for S3 to finish deleting 400,000 objects.
pgbackrest --stanza=main --no-expire backup --type=incr

# And in a separate job, outside the critical window:
pgbackrest --stanza=main expire

# Check what is actually in the repository and in what state.
# 'repo status' and each backup's WAL range are what matter.
pgbackrest --stanza=main info --output=json | \
  jq -r '.[0].backup[] | "\(.label) \(.type) \(.timestamp.stop) wal:\(.archive.start)-\(.archive.stop)"'
```

One last consideration that is not technical but decides the policy: some retentions are not yours to choose. If you handle card data, if you have tax retention obligations, or if you operate under a framework that requires proving the state of data on a given date, the minimum retention is given to you. And the reverse exists too: the right to erasure means data deleted at a user's request cannot live on indefinitely in a backup. The usual way out is to document the retention window as part of your privacy policy and to reapply pending erasures after any restore. That second step is almost never written in the runbook, and it should be.

Encryption and keys: the backup you cannot decrypt

Encrypting the repository is easy and almost everyone gets it right. What goes wrong is key management, and the failure has a recognisable shape: a circular dependency. The repository key lives in the secrets manager; the secrets manager authenticates against an identity service; that service queries a database; that database is the one you just lost. Every component is correct and you still cannot open your backups.

The configuration itself is three lines. pgBackRest encrypts on the client before uploading, so the storage provider never sees plaintext, and that combines with — rather than replaces — the bucket's own encryption at rest.

INI

```
[global]
# Client-side repository encryption.
repo1-cipher-type=aes-256-cbc
repo1-cipher-pass=LONG_RANDOM_KEY_INJECTED_AT_RUNTIME
```

```
# And on top of that, encryption at rest on the bucket with a KMS key.
repo1-s3-key-type=auto
repo1-type=s3
repo1-s3-bucket=backups-postgres-prod
repo1-s3-region=eu-west-1
repo1-storage-ca-file=/etc/ssl/certs/ca-certificates.crt
```

Two warnings about that key. First: repo1-cipher-pass cannot be rotated without rebuilding the repository. There is no re-encryption command; changing the key means starting a new repository and keeping the old one until its retention expires. So generate it long and random from the start, and do not reuse it across environments. Second: if that key exists only in the configuration file of the production server, and the production server is what burned down, you own three terabytes of beautifully encrypted noise.

The pattern that breaks the circularity is called *break-glass* and it is not sophisticated: a copy of the key, out of band, that depends on none of the systems the key exists to recover. What matters is not the form it takes, but that retrieving it has been rehearsed and that its use leaves a trace.

BASH

```
# Generate the repository key once, with enough entropy.
openssl rand -base64 48

# Store it in the secrets manager for day-to-day use...
aws secretsmanager create-secret \
  --name prod/pgbackrest/cipher-pass \
  --secret-string "$(openssl rand -base64 48)" \
  --kms-key-id alias/backups-prod

# ...and keep a break-glass copy somewhere that does NOT depend on production:
# another cloud account, another provider, or a sealed envelope in a safe.
# What matters is that the access procedure is written down and tested.

# At runtime, inject the key via environment variable or mounted file.
# Never leave it hard-coded in a version-controlled file.
export PGBACKREST_REPO1_CIPHER_PASS="$(aws secretsmanager get-secret-value \
  --secret-id prod/pgbackrest/cipher-pass --query SecretString --output text)"
pgbackrest --stanza=main backup --type=incr
```

And the step that turns this from policy into guarantee: include decryption in the drill. A drill that uses the key already loaded on the production machine proves nothing about your ability to decrypt during a disaster. The correct drill runs from a clean machine, retrieves the key through the break-glass path, and restores with it. It is inconvenient precisely because it tests the thing that needs testing. Doing it once a quarter, with two different people each time, is what separates an encryption policy from an illusion of one.

Managed services: what RDS does for you and what it does not

If you are on RDS, Cloud SQL or Azure Database, a good part of this article is run by somebody else. Continuous archiving, base backups, retention and PITR are built and tested by people whose job is to make them work. That is a real advantage and there is no point pretending otherwise. What is worth being clear about is what falls outside, because the gaps are specific and they do not show up in the console.

The first is the reach of the window. RDS automated backups give you PITR within the configured retention period, up to 35 days. Beyond that only manual snapshots exist, and a manual snapshot restores to the instant it was taken, not to an intermediate point. If your retention obligation is a year, the RDS console does not cover it on its own.

The second is subtler and it is the one that surprises people during a drill. Cross-region replication of automated backups exists and achieves an RPO of minutes, but **cross-region or cross-account snapshot copies lose PITR capability**: restoring them takes you back to the moment the copy was created, not to a point you choose. In other words, your geographic copy has a worse RPO than your primary one, and that difference needs to be understood before it goes into a continuity document. There are also quotas you hit without noticing: by default, twenty cross-region automated backups per account.

```bash
# PITR on RDS: restores to a NEW instance, never over the existing one.
aws rds restore-db-instance-to-point-in-time \
  --source-db-instance-identifier shop-prod \
  --target-db-instance-identifier shop-rescue \
  --restore-time 2026-08-27T12:29:50Z \
  --db-subnet-group-name private \
  --no-publicly-accessible \
  --db-instance-class db.r6g.xlarge

# Check how far back you can go RIGHT NOW. This number belongs on
# your dashboard, not discovered during an incident.
aws rds describe-db-instances \
  --db-instance-identifier shop-prod \
  --query 'DBInstances[0].LatestRestorableTime'
```

The third gap is the one no provider feature can close, and it is what justifies still taking your own dumps even on a managed service: account blast radius. Automated backups live in the same account as the database. Compromised credentials with enough permissions, or an infrastructure-as-code mistake that destroys a resource with `skip_final_snapshot`, take out the instance and its copies together. The mitigation is boring and it works: a periodic logical dump exported to a bucket in a *different* account, with Object Lock, that production has no permission to delete from.

```yaml
# CronJob exporting a logical dump outside the production account.
apiVersion: batch/v1
kind: CronJob
metadata:
  name: logical-export-off-account
spec:
  schedule: "0 3 * * *"
  concurrencyPolicy: Forbid
  jobTemplate:
    spec:
      backoffLimit: 2
      template:
        spec:
          restartPolicy: Never
```

```
    serviceAccountName: backup-exporter   # IRSA: assumes a role in the archive account
    containers:
      - name: pgdump
        image: postgres:18-alpine
        command: ["/bin/sh", "-c"]
        args:
          - |
            set -euo pipefail
            STAMP=$(date -u +%Y%m%dT%H%M%SZ)
            pg_dump --format=custom --compress=zstd:3 \
              --file=/tmp/shop-$STAMP.dump
            # The destination bucket lives in another account and has
            # Object Lock in COMPLIANCE mode: not even its owner can delete.
            aws s3 cp /tmp/shop-$STAMP.dump \
              s3://cold-archive-shop/logical/shop-$STAMP.dump
        env:
          - name: PGHOST
            value: shop-prod.abc123.eu-west-1.rds.amazonaws.com
          - name: PGUSER
            value: exporter
          - name: PGPASSWORD
            valueFrom:
              secretKeyRef: { name: pg-exporter, key: password }
        resources:
          requests: { cpu: "500m", memory: "1Gi" }
          limits:   { memory: "2Gi" }
```

It is also worth not confusing two Aurora features that sound alike. *Backtrack* rewinds the cluster backwards in time on the same instance, is very fast, and is designed to undo a recent mistake; its window is short and it is not a backup. PITR from backups does create a new cluster and reaches as far as your retention allows. Using backtrack as a PITR substitute is comfortable right up until the day the mistake is older than the backtrack window.

A full case: 14:32, an UPDATE without a WHERE

Listing tools is worth little if you never see how they fit together under pressure. This is one real incident in summarised form, with the numbers that mattered and the two decisions that saved it.

14:32. A maintenance job runs `UPDATE product_prices SET price_cents = price_cents * 100` with the `WHERE` commented out during a local test that nobody uncommented before deploying. 38,400 rows affected. The transaction commits.

14:41. Customer support reports orders with absurd totals. The team's first instinct is to write a reverse `UPDATE` dividing by 100. Somebody points out what is not visible: between 14:32 and 14:41 there have been legitimate price changes, and a blind division would corrupt those too. That is the first decision that saves the incident: *do not repair in place*.

14:44. The incident is declared and the partial-restore runbook is opened. The database is 1.9 TB, so restoring all of production is off the table: estimated RTO is around five hours and the shop works correctly for everything that is not prices.

14:47. The transaction is located with `pg_waldump` across the three segments covering the time range. xid

4,418,902,117 and its LSN appear. The immediately preceding LSN is recorded as the target.

14:52. `pgbackrest restore --delta --type=lsn` starts against a throwaway rescue instance on another port. The `--delta` is the second important decision: the rescue machine still held a data directory from the previous week's drill, and a delta restore only fetches the blocks that differ. 1.9 TB becomes 61 GB.

15:09. The rescue instance reaches the target and pauses. A `SELECT count(*) FROM product_prices WHERE price_cents > 1000000` returns zero: the prices are healthy.

15:14. `pg_dump --data-only -t product_prices` extracts the table and loads it into a `rescue` schema in production. 38,400 rows, 4 MB.

15:23. Reconciliation runs inside an open transaction: only rows whose `updated_at` predates the incident and whose current price differs from the rescued one are restored. The count before `COMMIT` reads 38,331 rows, not 38,400. The difference is 69 products with legitimate later price changes — exactly the ones the reverse `UPDATE` would have destroyed.

15:26. Commit. Incident closed. Actual RTO: 54 minutes from declaration. Effective RPO on the affected data: zero.

What is interesting about the case is not the tools, which are the usual ones. It is that none of the three things that kept the incident short were improvised. The partial-restore runbook existed because somebody wrote it after an earlier scare. The rescue machine had a recent data directory because the weekly drill left it there instead of deleting it, which cut the restore from hours to seventeen minutes. And the team could read `pg_waldump` because it was part of the quarterly exercise. You learn nothing on the day of the incident; you only execute what you already knew.

Glossary

- RPO — Recovery Point Objective: how much data, measured in time, you can afford to lose.
- RTO — Recovery Time Objective: how long you can take to be operational again.
- WAL — Write-Ahead Log: the sequential record of every change, written before the data files are touched. It is the foundation of both PITR and replication.
- LSN — Log Sequence Number: the position of a record within the WAL. It identifies an instant in the cluster unambiguously.
- Base backup — A physical copy of the entire data directory, taken consistently. It is the starting point on which WAL is replayed.
- PITR — Point-In-Time Recovery: restoring a base backup and replaying WAL up to a specific instant.
- Timeline — An identifier PostgreSQL increments every time a recovery is promoted, so that two divergent histories never mix. Documented in the .history files.
- Differential backup — Contains what changed since the last full backup.
- Incremental backup — Contains what changed since the previous backup, whatever its type. It chains, which is why the whole chain breaks if one link is missing.
- Delta restore — A restore that compares the destination against the repository and only fetches files that differ, instead of wiping and copying everything.
- Replication slot — A mechanism guaranteeing the primary does not delete WAL a consumer has not

received yet. Very useful, and the most common cause of full disks.
- Object Lock — Immutable retention at the object level in storage. In COMPLIANCE mode, not even the bucket owner can delete early.
- 3-2-1-1-0 — Three copies, on two media, one off-site, one immutable or offline, and zero errors on verification.
- Restore drill — A complete, automated restore with content verification, run on a schedule. The only thing that turns "we have backups" into a fact.

Eight mistakes that keep recurring

1. Relying on pg_dump as your only strategy. No PITR is possible: your RPO is the gap between dumps, and RTO on large databases balloons because of index rebuilds. It is a migration tool, not an incident-recovery tool.
2. Not monitoring pg_stat_archiver. Archiving breaks silently and you find out when you try to recover, which is the worst possible moment. Alert on failed_count and on the lag of the last archived segment.
3. An archive_command that returns 0 without having copied. If it is a script, start with set -euo pipefail, check exit codes, and make overwriting an existing file an error rather than a success.
4. Deleting the full backup that the incrementals depend on. With native incrementals, retention has to reason in chains. With pgBackRest, let repo1-retention-full handle it and never touch the repository by hand.
5. Restoring with recovery_target_action = 'promote' on the first try. Use pause, verify the data, then promote. Promotion is irreversible and forces you to redo the whole restore.
6. Keeping backups in the same place, with the same credentials, as the database. A snapshot in the same cloud project, deletable by the same account, does not protect against the scenario that does the most damage.
7. Not archiving .history files. It breaks any recovery that crosses timelines, right on the second attempt.
8. Never having measured your RTO. If you do not know how long your restore takes with today's data volume, you do not have a recovery plan — you have an intention.

Production checklist

- RPO and RTO written down and agreed with the business, not inferred by the infrastructure team.
- wal_level = replica or higher, archive_mode = on, and an archive_command that fails loudly.
- archive_timeout aligned with the RPO on low-write databases.
- Alerts on pg_stat_archiver: failures and last-archived lag.
- pg_wal free space monitored, with an alert before 80% usage.
- Periodic full backup plus incrementals, with retention that respects chains.
- pg_verifybackup or pgbackrest verify as a mandatory pipeline step.
- At least one immutable or offline copy, written with credentials that cannot delete.
- Repository encrypted at rest, with the key managed outside the database server.
- Automated weekly restore drill with content checks, not just startup checks.
- Measured RTO exported as a metric, with a trend alert.
- pg_dumpall --globals-only included in the cycle and validated in the drill.
- Recovery runbook written with exact commands, executed at least once by someone who did not write it.

- A procedure capable of deploying the same PostgreSQL minor version the cluster was running.

Frequently asked questions

Does a streaming replica replace backups?

No. A replica protects against a server failing, but it faithfully replicates your mistakes: a `DROP TABLE` reaches the standby in milliseconds. It protects availability, not correctness. What you can do is take the base backup *from* the replica so you do not load the primary — both `pg_basebackup` and pgBackRest support that.

Can I restore a physical backup onto a different major version?

No. The on-disk format is not compatible across major versions, and the architecture must match too. To move between versions, use `pg_dump` / `pg_restore` or `pg_upgrade`. The operational consequence matters: your recovery procedure must be able to deploy exactly the same minor version the cluster was running, so pin that version in your images and record it alongside the backup.

How often should I take a full backup?

It depends on how much WAL your database generates and on your RTO, not on the calendar. The useful question is: how much WAL would I have to replay if disaster struck right before the next full backup, and how long does that replay take? Weekly full with daily incrementals is a reasonable starting point for most; if you generate hundreds of gigabytes of WAL per day, shorten the cycle.

Is a volume snapshot good enough?

Yes, if it is atomic across *all* the cluster's volumes — including separate tablespaces and `pg_wal` if it lives elsewhere — and you coordinate it with `pg_backup_start()` and `pg_backup_stop()`, storing the `backup_label` the latter returns. If the snapshots are not consistent with each other you get an incoherent copy that may start up and look healthy for days. And you still need WAL archiving to do PITR between snapshots.

What about a managed service like RDS or Cloud SQL?

The provider handles archiving and offers PITR within a retention window, but that does not close the matter. It is still your responsibility to confirm the window covers your RPO, to export copies outside the provider account — for the case of an accidental instance deletion or a problem with the account itself — and to test the restore. Managed PITR also creates a brand- new instance, so your real RTO includes repointing the application, reviewing parameters and recreating replicas.

Can I recover to something other than a timestamp?

Yes. Besides `recovery_target_time` you have `recovery_target_xid`, `recovery_target_lsn` and `recovery_target_name`. That last one is particularly useful: before running a risky migration, call `pg_create_restore_point('before_migration_v42')` and you get an exact marker to return to, with no dependence on clocks or time zones.