PuigFlows

Row-Level Security en PostgreSQL: Aislamiento Multi-Tenant que No se Puede Olvidar

Row-Level Security in PostgreSQL: Multi-Tenant Isolation You Cannot Forget

---

Guia tecnica · Nivel intermedio · Lectura estimada 50 min

Edicion bilingue ES / EN · puigflows.com

Version en espanol

Row-Level Security en PostgreSQL: Aislamiento Multi-Tenant que No se Puede Olvidar

El WHERE que alguien va a olvidar

En una aplicación multi-tenant, casi todas las consultas llevan la misma cláusula: `WHERE tenant_id = ?`. Funciona hasta que alguien escribe un endpoint nuevo un viernes por la tarde, un job de reportes hace un JOIN sin filtrar, o el ORM genera una subconsulta que se salta el scope por defecto. El resultado no es un bug cualquiera: es que un cliente ve los datos de otro.

Row-Level Security (RLS) mueve ese filtro al único sitio donde no se puede olvidar: dentro de PostgreSQL. Una vez activada, el motor añade la condición a cada consulta que toque la tabla, venga de tu API, de un script de migración o de alguien con `psql` abierto en producción. No sustituye al filtrado en la aplicación —lo sigues queriendo por rendimiento y claridad—, pero convierte un fallo de aislamiento de "probable con el tiempo" en "requiere saltarse dos capas a la vez".

Esta guía cubre el modelo completo: el diseño de roles que hace que las políticas se apliquen de verdad, la sintaxis de las políticas, cómo propagar el tenant sin que se filtre entre peticiones de un pool, el truco de rendimiento que separa una política gratis de una que multiplica por N el coste de cada consulta, las puertas traseras (vistas, funciones, claves foráneas) y los tests que impiden que todo esto se degrade con el tiempo.

Cómo funciona RLS, en dos frases

Cuando activas RLS en una tabla, PostgreSQL adjunta las políticas de esa tabla como predicados adicionales a cualquier `SELECT`, `INSERT`, `UPDATE` o `DELETE` que la toque. Si no hay ninguna política aplicable, el comportamiento por defecto es denegar: la tabla se comporta como si estuviera vacía.

Ese "deny by default" es la propiedad que hace que RLS merezca la pena. Olvidarse de escribir una política rompe la funcionalidad de forma ruidosa e inmediata; olvidarse de un `WHERE` filtra datos en silencio durante meses. Prefieres el primer tipo de error.

USING y WITH CHECK no son lo mismo

Cada política puede tener dos expresiones, y hacen cosas distintas:

- `USING` filtra filas *existentes*. Se aplica al `SELECT` y a las filas candidatas de `UPDATE` y `DELETE`. Las filas que no cumplen la expresión sencillamente no existen para esa sesión: un `DELETE` que no las alcanza devuelve cero filas afectadas, no un error.

- `WITH CHECK` valida filas *nuevas*. Se aplica al `INSERT` y al resultado de un `UPDATE`. Si falla, PostgreSQL lanza un error explícito: `new row violates row-level security policy`.

Si defines `USING` y omites `WITH CHECK` en una política `FOR ALL`, PostgreSQL reutiliza la expresión de `USING` también para la validación. Es cómodo, pero escribe las dos de forma explícita: el día que necesites que un tenant lea filas archivadas pero no pueda crearlas, la diferencia importa.

PERMISSIVE y RESTRICTIVE

Las políticas permisivas (el defecto) se combinan con **OR**: basta que una se cumpla para que la fila pase. Las restrictivas se combinan con **AND**: todas tienen que cumplirse. En multi-tenant quieres tu política de aislamiento como **RESTRICTIVE**, por debajo de las políticas permisivas de negocio. Así, el día que alguien añada una política permisiva del tipo "los administradores ven todo", el filtro de tenant se seguirá aplicando por encima de ella.

Primero los roles, después las políticas

El error más común con RLS no está en las políticas: está en el rol con el que se conecta la aplicación.

Por defecto, RLS **no se aplica al dueño de la tabla**. Tampoco a superusuarios ni a roles con el atributo `BYPASSRLS`. Si tu aplicación se conecta con el mismo usuario que corre las migraciones —lo habitual cuando esto se monta con prisa—, las políticas están ahí, se ven en `pg_policies`, y no filtran absolutamente nada. Todo parece configurado y nada está protegido.

Hay dos defensas y quieres las dos:

```
-- 1) Rol de aplicación separado, sin privilegios especiales
--    y que NO es dueño de las tablas.
CREATE ROLE app_user LOGIN PASSWORD '...';
-- Sin SUPERUSER. Sin BYPASSRLS. Sin CREATEROLE.

GRANT USAGE ON SCHEMA public TO app_user;
GRANT SELECT, INSERT, UPDATE, DELETE
    ON ALL TABLES IN SCHEMA public TO app_user;

-- Que también aplique a las tablas futuras:
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_user;

-- 2) FORCE: las políticas se aplican también al dueño de la tabla.
ALTER TABLE invoices ENABLE ROW LEVEL SECURITY;
ALTER TABLE invoices FORCE  ROW LEVEL SECURITY;
```

`FORCE ROW LEVEL SECURITY` cierra el agujero del dueño. No afecta a superusuarios ni a roles con `BYPASSRLS`: esos siempre pasan por encima, por diseño, para que las copias de seguridad y el mantenimiento sigan funcionando. Por eso esa lista tiene que ser corta y estar auditada:

```
SELECT rolname, rolsuper, rolbypassrls
FROM pg_roles
WHERE rolcanlogin AND (rolsuper OR rolbypassrls)
ORDER BY rolname;
```

Si el usuario de tu API aparece en ese resultado, RLS no te está protegiendo de nada.

El esquema y la política base

Modelo de trabajo: una tabla `tenants` y tablas de negocio con una columna `tenant_id` no nula. La decisión de diseño más importante es que `tenant_id` esté en *todas* las tablas, incluso en aquellas donde podrías derivarlo

con un JOIN. Denormalizar esa columna es justo lo que permite que la política sea una comparación de igualdad barata en lugar de una subconsulta por fila.

```
CREATE TABLE tenants (
    id   uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    slug text NOT NULL UNIQUE
);

CREATE TABLE invoices (
    id          uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id   uuid NOT NULL REFERENCES tenants(id),
    number      text NOT NULL,
    total_cents bigint NOT NULL,
    status      text NOT NULL DEFAULT 'draft',
    created_at  timestamptz NOT NULL DEFAULT now()
);

-- Unicidad SIEMPRE con el tenant dentro: el número de factura
-- se repite entre tenants y eso es correcto.
CREATE UNIQUE INDEX invoices_tenant_number_key
    ON invoices (tenant_id, number);

-- El índice que va a usar la política. tenant_id primero, siempre.
CREATE INDEX invoices_tenant_created_idx
    ON invoices (tenant_id, created_at DESC);
```

La función de contexto

El tenant activo viaja en un parámetro de sesión personalizado. Envolver su lectura en una función centraliza la conversión de tipo y el caso "no hay tenant", y deja las políticas legibles.

```
CREATE OR REPLACE FUNCTION current_tenant_id() RETURNS uuid
LANGUAGE sql STABLE AS $$
    SELECT NULLIF(current_setting('app.tenant_id', true), '')::uuid;
$$;
```

Tres detalles que parecen menores y no lo son:

- El segundo argumento `true` de `current_setting` significa "si el parámetro no existe, devuelve NULL en lugar de lanzar un error". Sin él, cualquier consulta sin contexto revienta con `unrecognized configuration parameter` —incluidas las del health check.
- `NULLIF(..., '')` convierte la cadena vacía en NULL. Sin esto, un `app.tenant_id` vacío provoca un error de conversión a `uuid` en mitad de la consulta, con un mensaje que no dice nada útil.
- `STABLE`, no `VOLATILE`: el valor no cambia dentro de una consulta y el planificador necesita saberlo para poder optimizar.

Con NULL como resultado, la comparación `tenant_id = NULL` devuelve NULL, que no es `true`, así que no pasa ninguna fila. Sin contexto, cero datos. Ese es exactamente el comportamiento que quieres: una conexión mal configurada falla de forma visible en lugar de servir la base de datos entera.

La política

```
CREATE POLICY tenant_isolation ON invoices
    FOR ALL
    TO app_user
    USING      (tenant_id = (SELECT current_tenant_id()))
    WITH CHECK (tenant_id = (SELECT current_tenant_id()));
```

Fíjate en el `(SELECT ...)` alrededor de la llamada. No es una cuestión de estilo: es la diferencia entre una política que cuesta prácticamente nada y una que cuesta una llamada a función por cada fila examinada. Lo desarrollo en la sección de rendimiento.

Cuando aparecen las políticas de negocio

Tarde o temprano vas a querer reglas además del tenant: que un rol *viewer* no vea borradores, que soporte acceda a un subconjunto. En cuanto añades una segunda política permisiva, el `OR` puede abrirte un agujero. La forma robusta es separar el aislamiento del permiso:

```
-- Guardarraíl restrictivo: se combina con AND sobre TODO lo demás.
CREATE POLICY tenant_guard ON invoices
    AS RESTRICTIVE FOR ALL TO app_user
    USING      (tenant_id = (SELECT current_tenant_id()))
    WITH CHECK (tenant_id = (SELECT current_tenant_id()));

-- Políticas permisivas de negocio, que ya no pueden cruzar tenants.
CREATE POLICY invoices_read ON invoices
    FOR SELECT TO app_user
    USING (status <> 'draft' OR current_setting('app.role', true) = 'editor');

CREATE POLICY invoices_write ON invoices
    FOR ALL TO app_user
    USING      (current_setting('app.role', true) = 'editor')
    WITH CHECK (current_setting('app.role', true) = 'editor');
```

Ahora una política permisiva nueva y mal escrita puede como mucho conceder demasiado *dentro* del tenant. El cruce entre clientes lo impide el guardarraíl restrictivo, que nadie va a tocar.

Propagar el tenant sin filtrarlo entre peticiones

Aquí es donde se rompen la mayoría de implementaciones reales. La forma correcta de fijar el contexto es esta:

```
BEGIN;
SELECT set_config('app.tenant_id', '9f3c1b2e-...', true);  -- true = local
SELECT id, number FROM invoices WHERE status = 'open';
COMMIT;  -- el contexto desaparece con la transacción
```

Dos reglas, y las dos tienen consecuencias de seguridad.

Regla 1: local a la transacción, nunca a la sesión

Un `SET` normal dura toda la sesión. Con un pool de conexiones —tanto el de tu aplicación como PgBouncer en modo *transaction*— esa sesión se reutiliza para la siguiente petición, que puede ser de otro cliente. El parámetro

sobrevive, la política lo lee, y el tenant B ejecuta sus consultas con el contexto del tenant A. La aplicación no tiene ningún bug visible: las consultas están bien parametrizadas y el código parece correcto.

`SET LOCAL` y su equivalente funcional `set_config(nombre, valor, true)` se revierten al terminar la transacción, con COMMIT o con ROLLBACK. Si usas PgBouncer en modo *transaction*, esta no es una recomendación: es la única variante que funciona.

Regla 2: `set_config`, no interpolación de cadenas

`SET LOCAL app.tenant_id = $1` no es SQL válido: los comandos `SET` no aceptan parámetros. La tentación inmediata es construir la sentencia concatenando el identificador, que es exactamente donde vive la inyección SQL. `set_config()` es una función normal y sí acepta un placeholder. Úsala siempre.

Python: SQLAlchemy 2.0

```
from contextlib import contextmanager

from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker

engine = create_engine(
    "postgresql+psycopg://app_user:...@db:5432/app",
    pool_pre_ping=True,
)
SessionFactory = sessionmaker(engine)


@contextmanager
def tenant_session(tenant_id: str):
    """Sesión con el contexto de tenant fijado al ámbito de la transacción."""
    with SessionFactory() as session:
        with session.begin():                    # BEGIN ... COMMIT / ROLLBACK
            session.execute(
                text("SELECT set_config('app.tenant_id', :tid, true)"),
                {"tid": str(tenant_id)},
            )
            yield session
        # Al salir: COMMIT, y el parámetro local se descarta con él.
```

El uso queda limpio y, lo importante, el aislamiento ya no depende de que quien escriba la consulta se acuerde de nada:

```
with tenant_session(request.state.tenant_id) as s:
    rows = s.execute(
        text("""
            SELECT id, number, total_cents
            FROM invoices
            ORDER BY created_at DESC
            LIMIT 50
        """)
    ).all()
# Ni una mención a tenant_id. PostgreSQL la ha puesto por ti.
```

Un detalle de integración: si tu middleware no consigue resolver el tenant de la petición, no abras la sesión. Falla con un 401 o un 403 antes de tocar la base de datos. Una transacción sin contexto no es peligrosa (no ve nada), pero produce errores confusos en lugar de un mensaje claro.

TypeScript: node-postgres

```
import { Pool, PoolClient } from "pg";

const pool = new Pool({ connectionString: process.env.DATABASE_URL, max: 20 });

export async function withTenant<T>(
  tenantId: string,
  fn: (db: PoolClient) => Promise<T>,
): Promise<T> {
  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    // set_config admite parámetros; SET LOCAL no.
    await client.query("SELECT set_config('app.tenant_id', $1, true)", [tenantId]);
    const result = await fn(client);
    await client.query("COMMIT");
    return result;
  } catch (err) {
    await client.query("ROLLBACK");
    throw err;
  } finally {
    client.release();
  }
}

const invoices = await withTenant(req.tenantId, async (db) => {
  const { rows } = await db.query(
    "SELECT id, number, total_cents FROM invoices ORDER BY created_at DESC LIMIT 50",
  );
  return rows;
});
```

Con Prisma el patrón equivalente es un `$transaction` interactivo cuya primera sentencia es el `set_config`, envuelto en una extensión de cliente para que ningún repositorio pueda saltárselo. Con Drizzle, un `db.transaction()` con la misma primera línea. La idea no cambia: **una transacción explícita por petición, y el contexto se fija dentro de ella**.

Rendimiento: el `(SELECT ...)` que lo cambia todo

Escribe la política así y funciona:

```
USING (tenant_id = current_tenant_id())
```

Escríbela así y además es rápida:

```
USING (tenant_id = (SELECT current_tenant_id()))
```

La diferencia es que la subconsulta escalar hace que el planificador genere un nodo **InitPlan**: se evalúa una sola vez por ejecución de la consulta y el resultado queda disponible como un parámetro. Sin ella, aunque la función esté marcada como `STABLE`, PostgreSQL puede acabar evaluándola por cada fila candidata dentro del predicado de seguridad.

El efecto de segundo orden es el que de verdad importa. Con el valor resuelto a un parámetro, el planificador puede usarlo como *clave de búsqueda* en un índice. Sin él, el predicado se queda como filtro después de leer las filas. En el `EXPLAIN` se ve directamente:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, number FROM invoices ORDER BY created_at DESC LIMIT 50;

-- Lo que quieres ver:
--   InitPlan 1 (returns $0)
--     ->  Result
--   Index Scan using invoices_tenant_created_idx on invoices
--         Index Cond: (tenant_id = $0)

-- Lo que NO quieres ver:
--   Seq Scan on invoices
--         Filter: (tenant_id = current_tenant_id())
--         Rows Removed by Filter: 4821390
```

La regla general: `Index Cond` bien, `Filter` con la función dentro, mal. Y ese `Rows Removed by Filter` creciendo con el número total de tenants es la firma de un sistema que se va a degradar exactamente cuando el negocio vaya bien.

Una advertencia: el envoltorio `(SELECT ...)` solo es válido para expresiones que **no dependen de la fila**. Si la expresión referencia columnas de la tabla, no puedes sacarla a un InitPlan.

Índices: `tenant_id` va primero

RLS añade un predicado de igualdad sobre `tenant_id` a todas tus consultas. Un índice sobre `(created_at)` deja de servir; el que quieres es `(tenant_id, created_at)`. Revisa todos los índices de las tablas con RLS y antepón la columna de tenant. Es un cambio mecánico y suele ser la mitad de la ganancia de rendimiento al adoptar RLS.

Políticas que consultan otras tablas

Cuando un usuario puede pertenecer a varios tenants, la política tiene que mirar una tabla de membresías. Hacerlo de forma ingenua tiene dos problemas: cuesta caro y, si `memberships` también tiene RLS, PostgreSQL falla con `infinite recursion detected in policy for relation "memberships"`.

La solución a ambos es una función `SECURITY DEFINER` que resuelva la lista una sola vez:

```
CREATE OR REPLACE FUNCTION allowed_tenant_ids() RETURNS uuid[]
LANGUAGE sql STABLE SECURITY DEFINER
SET search_path = public, pg_temp
AS $$
    SELECT coalesce(array_agg(m.tenant_id), '{}')
    FROM memberships m
```

```
    WHERE m.user_id = NULLIF(current_setting('app.user_id', true), '')::uuid;
$$;

REVOKE EXECUTE ON FUNCTION allowed_tenant_ids() FROM PUBLIC;
GRANT  EXECUTE ON FUNCTION allowed_tenant_ids() TO app_user;

CREATE POLICY tenant_guard ON invoices
    AS RESTRICTIVE FOR ALL TO app_user
    USING      (tenant_id = ANY ((SELECT allowed_tenant_ids())))
    WITH CHECK (tenant_id = ANY ((SELECT allowed_tenant_ids())));
```

El `SET search_path` no es decorativo: en una función `SECURITY DEFINER` es obligatorio, porque sin él quien pueda crear objetos en un esquema anterior del path puede secuestrar la resolución de nombres y ejecutar código con los privilegios del dueño.

Vistas, funciones y otras puertas traseras

RLS protege tablas. Todo lo que envuelve una tabla es un posible desvío.

- **Vistas.** Por defecto una vista se ejecuta con los permisos de su *dueño*, no de quien la consulta. Si la vista pertenece al dueño de las tablas, cualquiera que pueda leerla ve todas las filas de todos los tenants. Desde PostgreSQL 15 la solución es explícita: `CREATE VIEW v WITH (security_invoker = true) AS ...`. Audita las vistas existentes, no solo las nuevas.

- **Funciones `SECURITY DEFINER`.** Se ejecutan como su dueño, así que aplican el RLS *del dueño*. Si el dueño es el propietario de las tablas y no has puesto `FORCE`, la función es un bypass completo. Úsalas de forma quirúrgica y con `search_path` fijo.

- **Vistas materializadas.** Se refrescan con el rol del dueño y el snapshot resultante contiene los datos de todos los tenants. Una matview no hereda RLS: si la expones al rol de aplicación, necesita su propia columna `tenant_id` y su propia política, o directamente no debe ser accesible.

- **`COPY`.** `COPY tabla TO` sí aplica las políticas de `SELECT`. `COPY ... FROM`, en cambio, **no está soportado** sobre tablas con RLS activada: tus rutas de carga masiva tienen que pasar por `INSERT`.

- **`pg_dump`.** Un volcado con un rol sujeto a RLS necesita `--enable-row-security` y produce un dump *parcial*, que es una trampa peligrosa para las copias de seguridad. Haz los backups con un rol que sí pueda saltarse RLS y trátalo como lo que es: una credencial crítica.

Restricciones e integridad referencial: los canales encubiertos

La documentación de PostgreSQL lo dice sin rodeos: las comprobaciones de integridad referencial —claves primarias, índices únicos y claves foráneas— **siempre se saltan RLS**. Tiene que ser así para que la integridad de los datos sea consistente para todo el mundo. Y tiene dos consecuencias prácticas.

Un índice único revela la existencia de filas ajenas. Si `invoices(number)` fuera único a secas, insertar `'A-001'` devolvería un error de duplicado cuando otro tenant ya lo tiene. Eso es un oráculo: permite enumerar datos que no puedes leer. La solución coincide con lo que ya querías por corrección funcional: mete `tenant_id` en todos

los índices únicos.

Una clave foránea puede cruzar tenants. La comprobación lee la fila padre saltándose RLS, así que nada impide que una factura del tenant A apunte a un cliente del tenant B si alguien consigue colar el identificador. La defensa fuerte no es una política: es una clave foránea compuesta que incluya el tenant.

```
CREATE TABLE customers (
    id        uuid NOT NULL DEFAULT gen_random_uuid(),
    tenant_id uuid NOT NULL REFERENCES tenants(id),
    name      text NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (tenant_id, id)          -- necesario para poder referenciarla
);

ALTER TABLE invoices
    ADD COLUMN customer_id uuid NOT NULL,
    ADD CONSTRAINT invoices_customer_same_tenant_fk
        FOREIGN KEY (tenant_id, customer_id)
        REFERENCES customers (tenant_id, id);
```

Con esa restricción, una referencia cruzada entre tenants es imposible *a nivel estructural*: falla aunque RLS esté desactivada, aunque el script lo ejecute un superusuario y aunque el bug esté en la aplicación. Es el tipo de garantía que no depende de que nadie se acuerde de nada.

Testing: el aislamiento se demuestra, no se supone

RLS es una de esas cosas que o tienen tests o se degradan. Dos niveles.

Tests de comportamiento

```
import pytest
from sqlalchemy import text
from sqlalchemy.exc import ProgrammingError


def test_un_tenant_no_ve_al_otro(tenant_a, tenant_b):
    with tenant_session(tenant_a) as s:
        s.execute(
            text("INSERT INTO invoices (tenant_id, number, total_cents) "
                 "VALUES (:t, 'A-001', 1000)"),
            {"t": tenant_a},
        )
    with tenant_session(tenant_b) as s:
        s.execute(
            text("INSERT INTO invoices (tenant_id, number, total_cents) "
                 "VALUES (:t, 'B-001', 2000)"),
            {"t": tenant_b},
        )

    with tenant_session(tenant_a) as s:
        numeros = [r[0] for r in s.execute(text("SELECT number FROM invoices")).all()]
```

```
    assert numeros == ["A-001"]     # B-001 sencillamente no existe para A


def test_no_se_puede_escribir_en_otro_tenant(tenant_a, tenant_b):
    # WITH CHECK lanza SQLSTATE 42501 -> ProgrammingError en SQLAlchemy.
    with pytest.raises(ProgrammingError, match="row-level security"):
        with tenant_session(tenant_a) as s:
            s.execute(
                text("INSERT INTO invoices (tenant_id, number, total_cents) "
                     "VALUES (:t, 'X-001', 1)"),
                {"t": tenant_b},          # tenant ajeno
            )


def test_sin_contexto_no_se_ve_nada(engine):
    with engine.connect() as c:           # sin set_config
        total = c.execute(text("SELECT count(*) FROM invoices")).scalar()
    assert total == 0
```

Ese tercer test es el más valioso de los tres y el que casi nadie escribe. Prueba la propiedad de *deny by default*: si un día alguien cambia la política y deja de comparar contra el contexto, este test se pone rojo aunque los otros dos sigan pasando.

El test que impide la erosión

El fallo más probable a medio plazo no es una política mal escrita: es una tabla nueva a la que nadie le activó RLS. Ese caso se detecta con una consulta al catálogo, y debe correr en CI:

```
EXENTAS = {"tenants", "alembic_version", "feature_flags"}


def test_toda_tabla_con_tenant_id_tiene_rls_forzada(engine):
    sql = text("""
        SELECT c.relname
        FROM pg_class c
        JOIN pg_namespace n ON n.oid = c.relnamespace
        WHERE n.nspname = 'public'
          AND c.relkind IN ('r', 'p')          -- tablas y particionadas
          AND EXISTS (
              SELECT 1 FROM pg_attribute a
              WHERE a.attrelid = c.oid
                AND a.attname  = 'tenant_id'
                AND NOT a.attisdropped
          )
          AND NOT (c.relrowsecurity AND c.relforcerowsecurity)
    """)
    with engine.connect() as conn:
        faltan = {r[0] for r in conn.execute(sql)} - EXENTAS

    assert not faltan, f"Tablas con tenant_id sin RLS forzada: {sorted(faltan)}"
```

Añade una migración con una tabla nueva que tenga `tenant_id` y olvídate de las dos líneas de `ALTER TABLE`: el build se cae antes de llegar a producción. Veinte líneas de test que cubren el 80% del riesgo real a lo largo del tiempo.

Migraciones: la tabla nueva que nadie protegió

Casi ningún incidente de RLS empieza con una política mal escrita. Empieza con una tabla que se creó un martes por la tarde y a la que nadie le puso `ENABLE ROW LEVEL SECURITY`. El modelo de PostgreSQL es abierto por defecto: una tabla recién creada no tiene RLS, y si tu rol de aplicación tiene `SELECT` sobre el esquema, la ve entera. El resto del sistema puede estar impecable y esa tabla sigue siendo un agujero.

La defensa no es la disciplina. La disciplina se erosiona en cuanto entra gente nueva al equipo o alguien tiene prisa antes de una demo. La defensa es un control automático que falla ruidosamente.

Auditar en un segundo qué tablas están desprotegidas

Esta consulta te da, para cada tabla de tus esquemas de aplicación, si tiene RLS habilitado, si está forzado y cuántas políticas la cubren. Es la primera consulta que ejecuto cuando alguien me dice "creo que tenemos RLS":

```
SELECT
  n.nspname                       AS esquema,
  c.relname                       AS tabla,
  c.relrowsecurity                AS rls_habilitado,
  c.relforcerowsecurity           AS rls_forzado,
  count(p.polname)                AS politicas
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_policy p ON p.polrelid = c.oid
WHERE c.relkind IN ('r', 'p')            -- tablas y tablas particionadas
  AND n.nspname NOT IN ('pg_catalog', 'information_schema')
GROUP BY 1, 2, 3, 4
ORDER BY c.relrowsecurity ASC, politicas ASC, 1, 2;
```

Lo que buscas está arriba del todo: filas con `rls_habilitado = false`, o con `rls_habilitado = true` pero `politicas = 0`. El segundo caso es curioso y merece una nota: una tabla con RLS habilitado y cero políticas no es permisiva, es lo contrario. PostgreSQL aplica una política implícita de denegación total, así que la tabla devuelve cero filas para todo el mundo excepto el dueño y los roles con `BYPASSRLS`. Es un fallo cerrado, que es el fallo correcto, pero suele significar que alguien habilitó RLS y se olvidó de la mitad del trabajo.

Fíjate también en `relforcerowsecurity`. Si el dueño de la tabla y el rol de la aplicación coinciden —cosa que pasa más de lo que debería— sin `FORCE` las políticas no se aplican al dueño y tu aislamiento es decorativo.

Convertirlo en un test que rompe el build

Envuelve esa consulta en un test. No en un documento, no en una casilla de una lista de revisión: en algo que ponga el CI en rojo. Una lista de excepciones explícita es aceptable siempre que sea corta y esté justificada en el propio código:

```
# tests/test_rls_cobertura.py
import pytest

# Tablas que legítimamente no llevan RLS. Cada entrada necesita un motivo.
EXENTAS = {
    "public.schema_migrations",   # metadatos de migración, sin datos de tenant
    "public.planes",              # catálogo global de planes, mismo para todos
```

```
    "public.paises",              # tabla de referencia, solo lectura
}

CONSULTA = """
SELECT n.nspname || '.' || c.relname AS tabla,
       c.relrowsecurity,
       c.relforcerowsecurity,
       count(p.polname) AS politicas
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_policy p ON p.polrelid = c.oid
WHERE c.relkind IN ('r', 'p')
  AND n.nspname = 'public'
GROUP BY 1, 2, 3
"""

def test_todas_las_tablas_tienen_rls(conexion_admin):
    problemas = []
    for tabla, habilitado, forzado, politicas in conexion_admin.execute(CONSULTA):
        if tabla in EXENTAS:
            continue
        if not habilitado:
            problemas.append(f"{tabla}: RLS no habilitado")
        elif not forzado:
            problemas.append(f"{tabla}: RLS habilitado pero sin FORCE")
        elif politicas == 0:
            problemas.append(f"{tabla}: RLS habilitado sin ninguna politica")
    assert not problemas, "Tablas sin proteger:\n" + "\n".join(problemas)
```

El detalle que hace que este test funcione a largo plazo es la lista `EXENTAS`. Sin ella, la primera tabla de catálogo que alguien añada dejará el test en rojo, alguien lo marcará como `skip` "temporalmente" y habrás perdido el control entero. Con ella, añadir una excepción es un cambio visible en una revisión de código, que es exactamente donde quieres esa conversación.

Un event trigger que lo grita al momento

Si quieres el aviso antes incluso de que el código llegue al CI, PostgreSQL tiene event triggers sobre DDL. Este emite un `WARNING` en cuanto se crea una tabla en `public`, en la misma sesión de `psql` o de la herramienta de migración:

```
CREATE OR REPLACE FUNCTION avisar_tabla_sin_rls()
RETURNS event_trigger
LANGUAGE plpgsql
AS $$
DECLARE
  obj record;
BEGIN
  FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands()
  LOOP
    IF obj.command_tag = 'CREATE TABLE'
       AND obj.schema_name = 'public' THEN
      RAISE WARNING
        'Tabla % creada sin RLS. Anade ENABLE + FORCE ROW LEVEL SECURITY y una politica.',
        obj.object_identity;
```

```
    END IF;
  END LOOP;
END;
$$;

CREATE EVENT TRIGGER tr_avisar_tabla_sin_rls
  ON ddl_command_end
  WHEN TAG IN ('CREATE TABLE')
  EXECUTE FUNCTION avisar_tabla_sin_rls();
```

Podrías usar `RAISE EXCEPTION` en lugar de `WARNING` para abortar la creación directamente, pero no lo recomiendo: rompe herramientas de migración, restauraciones de `pg_restore` y tablas temporales de trabajo de formas que cuesta depurar. El `WARNING` más el test de CI cubre el mismo riesgo sin los efectos secundarios.

La plantilla de migración

Reduce la fricción. Si crear una tabla nueva con RLS correcto requiere recordar cinco sentencias, alguien olvidará dos. Ten una función auxiliar y que la migración sea una línea:

```
CREATE OR REPLACE FUNCTION proteger_tabla_tenant(nombre_tabla text)
RETURNS void
LANGUAGE plpgsql
AS $$
BEGIN
  EXECUTE format('ALTER TABLE %I ENABLE ROW LEVEL SECURITY', nombre_tabla);
  EXECUTE format('ALTER TABLE %I FORCE ROW LEVEL SECURITY', nombre_tabla);
  EXECUTE format($f$
    CREATE POLICY aislamiento_tenant ON %I
      AS PERMISSIVE FOR ALL
      TO app_usuario
      USING (tenant_id = (SELECT app.tenant_actual()))
      WITH CHECK (tenant_id = (SELECT app.tenant_actual()))
  $f$, nombre_tabla);
  EXECUTE format(
    'CREATE INDEX IF NOT EXISTS %I ON %I (tenant_id)',
    'idx_' || nombre_tabla || '_tenant', nombre_tabla);
END;
$$;

-- En la migración:
CREATE TABLE facturas (
  id          uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id   uuid NOT NULL,
  importe     numeric(12,2) NOT NULL,
  creada_en   timestamptz NOT NULL DEFAULT now()
);
SELECT proteger_tabla_tenant('facturas');
```

Ojo con el `format` y `%I`: es la forma correcta de interpolar identificadores en SQL dinámico. Usar concatenación de cadenas aquí abre una inyección por nombre de tabla que, en una función que corre como el dueño del esquema, es tan grave como suena.

El pool de conexiones, a fondo

Ya vimos que `SET LOCAL` es obligatorio y que `SET` a nivel de sesión es un fallo de aislamiento esperando a ocurrir. Merece la pena entender por qué, porque la explicación te dice exactamente qué configuraciones de pool son seguras y cuáles no.

Por qué transaction mode y SET LOCAL encajan

PgBouncer en `pool_mode = transaction` asigna una conexión del servidor a un cliente sólo mientras dura una transacción. En cuanto llega el `COMMIT` o el `ROLLBACK`, esa conexión vuelve al pool y el siguiente cliente puede recibirla. Ese es el momento peligroso: cualquier estado que sobreviva a la transacción viaja con la conexión al siguiente inquilino.

`SET LOCAL`, y su equivalente funcional `set_config(clave, valor, true)` con el tercer argumento en `true`, ata el valor a la transacción. Al terminar, PostgreSQL lo revierte. No hay ventana. Es la única propiedad que hace que RLS y el pooling en modo transacción convivan sin sorpresas.

Con `SET` de sesión el valor persiste. Si la petición A configura el tenant 42 y termina, la petición B del tenant 7 puede recibir esa conexión con `app.tenant_id = 42` todavía puesto. Si el código de B configura su propio tenant, no pasa nada. Si por cualquier motivo no lo hace —una ruta que se saltó el middleware, un healthcheck, un job que reusó el pool— B lee los datos de 42. Y lo hará sin ningún error, devolviendo filas plausibles. Es la peor clase de fallo que existe.

```
-- Peligroso con pooling en modo transaccion:
SET app.tenant_id = '42';        -- sobrevive al COMMIT

-- Correcto:
BEGIN;
SELECT set_config('app.tenant_id', '42', true);   -- true = local a la transaccion
SELECT * FROM facturas;
COMMIT;                                            -- el valor desaparece aqui
```

DISCARD ALL no es tu red de seguridad

Existe la tentación de decir "PgBouncer ejecuta `server_reset_query` entre clientes, así que estoy cubierto". En modo transacción, por defecto, `server_reset_query` *no* se ejecuta: la opción `server_reset_query_always` viene desactivada precisamente porque en transaction mode se asume que no queda estado que limpiar. Y aunque lo activases, estarías pagando un viaje de ida y vuelta extra por transacción para arreglar algo que `SET LOCAL` arregla gratis. Configúralo bien y no dependas de la limpieza.

Sentencias preparadas: el detalle que solía morder

Durante años, usar sentencias preparadas con PgBouncer en modo transacción era terreno minado: la sentencia se preparaba en una conexión del servidor y el `EXECUTE` podía acabar en otra, con el consiguiente error de "prepared statement does not exist". La solución habitual era desactivarlas en el driver, lo que cuesta rendimiento en consultas repetitivas —justo las que genera un ORM.

Esto ya está resuelto. PgBouncer 1.21 introdujo soporte para sentencias preparadas a nivel de protocolo en modo transacción mediante `max_prepared_statements`, que mantiene una caché LRU de sentencias por

conexión del servidor y reescribe los identificadores de forma transparente. Y PostgreSQL 17 añadió la pieza que faltaba: la capacidad de cerrar a nivel de protocolo una sentencia preparada, lo que elimina los errores que aparecían cuando el driver hacía `DEALLOCATE` por su cuenta.

```
# pgbouncer.ini
[pgbouncer]
pool_mode = transaction
max_prepared_statements = 200     ; 0 desactiva el soporte
default_pool_size = 25
server_reset_query = DISCARD ALL
server_reset_query_always = 0     ; no hace falta en transaction mode
```

Importante: esto cubre las sentencias preparadas *del protocolo* (las que emiten los drivers cuando usas consultas parametrizadas). Los comandos SQL `PREPARE`, `EXECUTE` y `DEALLOCATE` escritos a mano se reenvían tal cual al servidor y siguen siendo incompatibles con el modo transacción. Si tienes código que los usa, reescríbelo con parámetros normales.

Serverless, HTTP y drivers sin transacción

Los drivers que hablan con la base de datos por HTTP —el driver serverless de Neon, la API de datos de RDS, PostgREST— complican el asunto porque cada llamada puede ser una transacción implícita distinta. Si tu `set_config` va en una llamada y tu `SELECT` en otra, el contexto ya no existe cuando llega la consulta y la política evalúa contra `NULL`: cero filas, o peor, si escribiste la función de contexto de forma laxa, todas.

Hay dos salidas limpias. La primera es agrupar ambas sentencias en una transacción explícita que el driver envíe como una unidad. La segunda, más robusta, es dejar de propagar el tenant por variable de sesión y hacer que venga firmado en el propio token, que es lo que hace Supabase con las claims del JWT. Volvemos a ello en la sección de integraciones.

RDS Proxy y el "pinning"

Si usas Amazon RDS Proxy, conviene saber que ciertas operaciones provocan *pinning*: el proxy deja de multiplexar y ata la conexión del cliente a una del servidor hasta que se cierra. Usar `SET` a nivel de sesión es una de esas operaciones. Es decir, la variante insegura de propagar el tenant además destruye el beneficio del proxy. `SET LOCAL` dentro de una transacción no provoca pinning. Es agradable cuando la opción correcta también es la rápida.

Trabajos en segundo plano, ETL y copias de seguridad

RLS protege el camino que va del usuario a la base de datos. La mitad de los accesos reales a una base de datos de producción no siguen ese camino: migraciones, informes nocturnos, exportaciones, replicación, backups, el script que alguien corre desde su portátil. Cada uno necesita una decisión explícita.

BYPASSRLS: quién lo tiene y por qué

El atributo de rol `BYPASSRLS`, que sólo puede otorgar un superusuario, permite a un rol saltarse las políticas por completo. Los superusuarios y los dueños de tabla con `BYPASSRLS` lo hacen siempre. Es una herramienta legítima: un proceso de agregación que calcula métricas de toda la plataforma necesita ver todas las filas. Lo que no es legítimo es que ese atributo lo lleve el rol con el que se conecta tu API.

```
-- Rol de la aplicacion: sin privilegios especiales, sujeto a RLS
CREATE ROLE app_usuario LOGIN PASSWORD :'clave_app'
  NOBYPASSRLS NOSUPERUSER NOCREATEDB NOCREATEROLE;

-- Rol analitico: ve todo, pero solo lectura y solo desde la red interna
CREATE ROLE analitica LOGIN PASSWORD :'clave_analitica' BYPASSRLS;
GRANT USAGE ON SCHEMA public TO analitica;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO analitica;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
  GRANT SELECT ON TABLES TO analitica;

-- Auditoria: que roles pueden saltarse RLS
SELECT rolname, rolsuper, rolbypassrls
FROM pg_roles
WHERE rolbypassrls OR rolsuper
ORDER BY rolname;
```

Esa última consulta debería tener un resultado corto y aburrido, y debería estar en el mismo test de CI que la cobertura de tablas. Un `BYPASSRLS` nuevo que aparece sin que nadie lo haya pedido es una señal que quieres ver.

pg_dump, restauraciones y el flag que casi nadie conoce

Por defecto `pg_dump` falla si intenta volcar una tabla con RLS activo usando un rol que está sujeto a las políticas, en lugar de exportar silenciosamente un subconjunto. Es un comportamiento deliberado y muy sensato: un backup parcial que parece completo es peor que un backup que no existe.

Si lo que quieres de verdad es un volcado filtrado por tenant —para dar a un cliente sus propios datos, por ejemplo— existe `--enable-row-security`, que le dice a `pg_dump` que exporte con las políticas activas. En ese caso tienes que establecer el contexto tú mismo, y el resultado es explícitamente un volcado parcial:

```
# Backup completo: usa un rol con BYPASSRLS o el dueño. Falla ruidosamente
# si el rol esta sujeto a politicas, que es lo que quieres.
pg_dump -U postgres -Fc -f completo.dump miapp

# Exportacion por tenant: politicas activas y contexto fijado
PGOPTIONS="-c app.tenant_id=a1b2c3d4-0000-0000-0000-000000000000" \
  pg_dump -U app_usuario --enable-row-security \
          --data-only -t facturas -t clientes \
          -f tenant_a1b2.sql miapp
```

Dos advertencias sobre la segunda forma. Una: `PGOPTIONS` con `-c` establece el parámetro a nivel de sesión, no de transacción, lo cual está bien aquí porque es un proceso de un solo uso con su propia conexión, pero no traslades ese patrón a la aplicación. Dos: revisa el resultado antes de entregarlo. Una política mal escrita en una tabla secundaria puede incluir filas de otros tenants en un fichero que estás a punto de enviar por correo.

Replicación lógica

La replicación lógica merece un párrafo propio porque su interacción con RLS sorprende. Si el rol de replicación carece de `SUPERUSER` y de `BYPASSRLS`, las políticas del publicador se ejecutan durante la replicación, lo que significa que el suscriptor puede recibir un subconjunto de las filas sin ningún aviso. Tu réplica está incompleta y nada te lo dice.

La documentación de PostgreSQL recomienda que, si no confías en todos los dueños de tablas, incluyas `options=-crow_security=off` en la cadena de conexión de la suscripción. Con eso, si un dueño añade una política, la replicación se detiene con un error en lugar de aplicarla silenciosamente. Prefieres una alerta a las tres de la mañana antes que una réplica en la que no puedes confiar:

```
CREATE SUBSCRIPTION sub_analitica
  CONNECTION 'host=primario dbname=miapp user=replicador options=-crow_security=off'
  PUBLICATION pub_completa;
```

COPY, cargas masivas y el WITH CHECK

`COPY ... FROM` respeta las políticas `WITH CHECK` igual que un `INSERT`. Esto es correcto pero tiene una consecuencia práctica: una carga de un millón de filas evaluará la expresión de la política un millón de veces. Si tu `WITH CHECK` hace una subconsulta a otra tabla, la carga se vuelve lentísima. Para importaciones masivas de datos de confianza, lo razonable es correr el proceso con un rol `BYPASSRLS` dedicado y validar el `tenant_id` en el propio proceso de carga, o usar una tabla de staging sin RLS y mover los datos con un único `INSERT ... SELECT` ya validado.

Fugas laterales: LEAKPROOF, mensajes de error y funciones

Hasta aquí hemos hablado de filas que se ven o no se ven. Existe una categoría más sutil de fuga: consultas que no devuelven ninguna fila prohibida pero sí revelan información sobre ellas. Es un tema que casi nunca aparece en los tutoriales de RLS y que conviene entender aunque sólo sea para saber cuándo no preocuparse.

Cómo un mensaje de error revela una fila que no puedes ver

Imagina una tabla `facturas` con RLS por tenant y esta consulta, lanzada por un usuario del tenant A:

```
SELECT * FROM facturas WHERE 1 / (importe - 1000) > 0;
```

Si el planificador decide evaluar `1 / (importe - 1000)` *antes* que la condición de la política, y existe en cualquier tenant una factura de exactamente 1000, la consulta aborta con `division by zero`. El usuario del tenant A no ha visto la factura del tenant B, pero acaba de confirmar que existe. Repitiendo la consulta con distintos valores puede extraer datos de filas que la política le prohíbe leer.

PostgreSQL bloquea esto, y el mecanismo es la marca `LEAKPROOF`. Las condiciones de seguridad —las de las políticas RLS y las de las vistas con `security_barrier`— se evalúan antes que cualquier predicado del usuario que pudiera filtrar el contenido de una fila. Sólo los operadores y funciones marcados como `LEAKPROOF` pueden reordenarse o empujarse por debajo de esa barrera, porque son de confianza: invocarlos sobre filas invisibles no revela nada sobre ellas.

Qué significa exactamente LEAKPROOF

Una función es `LEAKPROOF` si no tiene efectos secundarios y, en particular, si no revela información sobre sus argumentos por ningún canal que no sea su valor de retorno. Eso incluye los mensajes de error: una función que lance `ERROR: valor invalido: %s` con el argumento incrustado no es leakproof, por más pura que parezca.

Muchos operadores simples y muy usados sí lo son —los de igualdad, por ejemplo—, y por eso el rendimiento

con RLS suele ser razonable: los filtros más comunes se pueden empujar hacia abajo sin comprometer nada. Sólo un superusuario puede marcar una función como `LEAKPROOF`, precisamente porque hacerlo mal abre exactamente el agujero que el mecanismo pretende cerrar.

```
-- Ver que operadores de tu esquema son leakproof
SELECT p.proname, p.proleakproof, pg_get_function_identity_arguments(p.oid) AS args
FROM pg_proc p
JOIN pg_namespace n ON n.oid = p.pronamespace
WHERE n.nspname = 'public'
ORDER BY p.proleakproof DESC, p.proname;
```

Hay un caso especial útil: las funciones que no reciben argumentos, o que no reciben ninguno procedente de la vista o tabla protegida, no necesitan estar marcadas como `LEAKPROOF` para poder empujarse, porque nunca reciben datos de las filas protegidas. Tu función `app.tenant_actual()` entra en esa categoría, que es una de las razones por las que el truco del `(SELECT ...)` del que hablamos antes funciona tan bien.

El coste de la barrera

Conviene ser honesto sobre el precio. Las vistas creadas con `security_barrier` pueden rendir bastante peor que las normales, y en general no hay forma de evitarlo: si el plan más rápido puede comprometer la seguridad, hay que descartarlo. Lo mismo aplica, en menor medida, a las tablas con RLS. Si ves una diferencia de rendimiento grande e inexplicable entre una consulta con RLS y la misma consulta como dueño de la tabla, mira si algún predicado no leakproof está impidiendo que un filtro selectivo baje hasta el escaneo.

SECURITY DEFINER y search_path: la puerta trasera clásica

Las funciones `SECURITY DEFINER` se ejecutan con los privilegios de quien las creó, no de quien las llama. Son necesarias para operaciones controladas que cruzan tenants, y son un vector de escalada de privilegios de manual si no fijas el `search_path`.

El ataque es sencillo: si tu función `SECURITY DEFINER` hace `SELECT ... FROM facturas` sin cualificar el esquema, un atacante que pueda crear objetos en un esquema que aparezca antes en el `search_path` —`pg_temp` es el sospechoso habitual— coloca ahí su propia `facturas` o su propio operador, y tu función privilegiada acaba ejecutando su código.

```
-- Mal: search_path heredado del llamante
CREATE FUNCTION transferir_propiedad(f_id uuid, nuevo_tenant uuid)
RETURNS void
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
  UPDATE facturas SET tenant_id = nuevo_tenant WHERE id = f_id;
END;
$$;

-- Bien: search_path vacio y todo cualificado
CREATE FUNCTION app.transferir_propiedad(f_id uuid, nuevo_tenant uuid)
RETURNS void
LANGUAGE plpgsql
SECURITY DEFINER
```

```
SET search_path = ''
AS $$
BEGIN
  IF NOT public.puede_administrar(current_user, nuevo_tenant) THEN
    RAISE EXCEPTION 'no autorizado';
  END IF;
  UPDATE public.facturas
     SET tenant_id = nuevo_tenant
   WHERE id = f_id;
END;
$$;

-- Y nunca la dejes abierta a todo el mundo:
REVOKE ALL ON FUNCTION app.transferir_propiedad(uuid, uuid) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION app.transferir_propiedad(uuid, uuid) TO rol_administrador;
```

`SET search_path = ''` es más estricto que `SET search_path = public, pg_temp` y por eso es la recomendación actual: te obliga a cualificar cada nombre, incluidos los tipos y los operadores, y elimina la clase entera de ataques por resolución de nombres. Es más verboso. Merece la pena.

El `REVOKE ... FROM PUBLIC` del final tampoco es opcional. Por defecto, `EXECUTE` sobre funciones nuevas se concede a `PUBLIC`. Una función `SECURITY DEFINER` que cualquiera puede llamar es, literalmente, una API de escalada de privilegios que has publicado sin querer.

Rendimiento avanzado: medir antes de opinar

El coste real de una política, en números

La forma correcta de saber cuánto te cuesta RLS es comparar el mismo plan con y sin políticas, en la misma sesión y sobre los mismos datos. `row_security = off` te lo permite si el rol tiene `BYPASSRLS`:

```
-- Con politicas
SET ROLE app_usuario;
BEGIN;
SELECT set_config('app.tenant_id', :'tenant', true);
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT id, importe FROM facturas WHERE creada_en > now() - interval '30 days';
COMMIT;
RESET ROLE;

-- Sin politicas, mismo dato, misma cache
SET ROLE analitica;   -- rol con BYPASSRLS
SET row_security = off;
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT id, importe FROM facturas
WHERE tenant_id = :'tenant' AND creada_en > now() - interval '30 days';
RESET row_security;
RESET ROLE;
```

Lo que quieres ver en el primer plan es un `Index Cond` que incluya `tenant_id`, no un `Filter`. Si `tenant_id` aparece bajo `Filter`, PostgreSQL está leyendo filas de otros tenants desde disco y descartándolas después. Funciona, es seguro, y es entre uno y dos órdenes de magnitud más lento en una tabla grande. Presta atención

también a `Buffers`: un salto grande en `shared read` entre ambos planes es la prueba objetiva de que la política no se está usando como condición de índice.

Planes genéricos frente a planes personalizados

Este es un efecto de segundo orden que muerde en producción y casi nunca en desarrollo. Cuando ejecutas la misma sentencia preparada varias veces, PostgreSQL evalúa si le compensa cachear un plan genérico —uno que no mira los valores concretos de los parámetros— en lugar de replanificar cada vez. A partir de la sexta ejecución, si el coste estimado del plan genérico no es peor, se queda con él.

Con multi-tenancy esto puede salir mal. Si tienes un tenant con diez millones de filas y quinientos con doscientas, el plan genérico será un compromiso que no le va bien a ninguno: escaneo secuencial para los pequeños, o bucles anidados para el grande. El parámetro `plan_cache_mode` te deja forzar el comportamiento:

```
-- Forzar replanificacion por ejecucion en consultas sensibles al tamano del tenant
BEGIN;
SET LOCAL plan_cache_mode = force_custom_plan;
SELECT set_config('app.tenant_id', :'tenant', true);
SELECT ... ;
COMMIT;
```

No lo pongas globalmente: replanificar tiene un coste y en la mayoría de las consultas no compensa. Aplícalo con `SET LOCAL` sólo en las rutas donde hayas medido que el plan genérico se equivoca. Y mide de verdad, con `pg_stat_statements`, antes y después.

Índices parciales y de cobertura

Ya sabes que `tenant_id` va primero en el índice compuesto. Dos refinamientos que suelen dar mucho por poco esfuerzo:

```
-- Indice de cobertura: la consulta se resuelve sin tocar la tabla
CREATE INDEX idx_facturas_tenant_fecha
  ON facturas (tenant_id, creada_en DESC)
  INCLUDE (importe, estado);

-- Indice parcial: si el 95% de las consultas solo miran facturas activas,
-- no pagues por indexar las archivadas
CREATE INDEX idx_facturas_tenant_activas
  ON facturas (tenant_id, creada_en DESC)
  WHERE estado <> 'archivada';
```

El índice de cobertura importa especialmente con RLS porque evita el *heap fetch*, y menos accesos a la tabla significa menos filas sobre las que evaluar la política. El parcial reduce el tamaño del índice, que en una tabla con años de histórico puede significar la diferencia entre caber en memoria y no caber.

Particionar por tenant: cuándo sí y cuándo no

Si tienes unos pocos tenants enormes, particionar por lista sobre `tenant_id` permite que el planificador descarte particiones enteras y que cada tenant tenga sus propias estadísticas, lo que resuelve de raíz el problema del plan genérico:

```
CREATE TABLE eventos (
  id         bigserial,
  tenant_id  uuid NOT NULL,
  tipo       text NOT NULL,
  ocurrido_en timestamptz NOT NULL DEFAULT now()
) PARTITION BY LIST (tenant_id);

CREATE TABLE eventos_tenant_grande
  PARTITION OF eventos FOR VALUES IN ('a1b2c3d4-0000-0000-0000-000000000000');
CREATE TABLE eventos_resto PARTITION OF eventos DEFAULT;

-- RLS en la tabla padre se hereda por las particiones
ALTER TABLE eventos ENABLE ROW LEVEL SECURITY;
ALTER TABLE eventos FORCE ROW LEVEL SECURITY;
CREATE POLICY aislamiento ON eventos
  FOR ALL TO app_usuario
  USING (tenant_id = (SELECT app.tenant_actual()))
  WITH CHECK (tenant_id = (SELECT app.tenant_actual()));
```

Ahora el aviso: no particiones por tenant si tienes miles de tenants parecidos. Cada partición es una tabla con su propio catálogo, sus propios índices y su propio trabajo de autovacuum. Con diez mil particiones el tiempo de planificación se dispara y las operaciones de mantenimiento se vuelven un problema en sí mismas. La regla práctica que uso: particionar por tenant tiene sentido con decenas de tenants, no con miles. Para miles, particiona por tiempo si hace falta y deja que el índice sobre `tenant_id` haga su trabajo.

Observabilidad: saber qué tenant hizo qué

Cuando RLS funciona bien es invisible, y eso es un problema para depurar. Un usuario dice "no veo mis facturas" y tú tienes que distinguir entre tres causas que producen el mismo síntoma: no hay facturas, el contexto de tenant no se estableció, o la política tiene un error. Sin instrumentación, esa investigación son dos horas. Con instrumentación, dos minutos.

Etiquetar la conexión

El truco más barato y más útil: escribe el tenant y la petición en `application_name` al abrir la transacción, y añádelo al prefijo de log. A partir de ahí, cada consulta lenta, cada deadlock y cada error del log lleva el tenant pegado.

```
-- postgresql.conf
log_line_prefix = '%m [%p] %q%u@%d app=%a '
log_min_duration_statement = 500ms

# En el mismo sitio donde estableces el tenant
async def con_tenant(conn, tenant_id: str, request_id: str):
    await conn.execute(
        "SELECT set_config('app.tenant_id', $1, true), "
        "       set_config('application_name', $2, true)",
        tenant_id, f"api t={tenant_id[:8]} r={request_id[:8]}",
    )
```

Fíjate en que `application_name` también se establece con `set_config(..., true)`. Es un parámetro de sesión

normal, así que si lo pones con `SET` se queda pegado a la conexión y acabarás atribuyendo las consultas de un tenant a otro en tus dashboards. El mismo error, el mismo remedio.

Auditar quién tocó qué

Para las tablas sensibles, un trigger de auditoría que registre el tenant efectivo en el momento de la escritura te da algo con lo que responder preguntas incómodas más adelante:

```
CREATE TABLE auditoria (
  id           bigserial PRIMARY KEY,
  tabla        text        NOT NULL,
  operacion    text        NOT NULL,
  fila_id      uuid,
  tenant_ctx   uuid,          -- tenant que decia ser la sesion
  tenant_fila  uuid,          -- tenant de la fila afectada
  rol          text        NOT NULL DEFAULT current_user,
  app          text        NOT NULL DEFAULT current_setting('application_name'),
  ocurrido_en  timestamptz NOT NULL DEFAULT clock_timestamp()
);

CREATE OR REPLACE FUNCTION auditar_cambio()
RETURNS trigger
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = ''
AS $$
DECLARE
  fila record;
BEGIN
  fila := COALESCE(NEW, OLD);
  INSERT INTO public.auditoria (tabla, operacion, fila_id, tenant_ctx, tenant_fila)
  VALUES (
    TG_TABLE_NAME,
    TG_OP,
    fila.id,
    NULLIF(current_setting('app.tenant_id', true), '')::uuid,
    fila.tenant_id
  );
  RETURN fila;
END;
$$;

CREATE TRIGGER tr_auditar_facturas
  AFTER INSERT OR UPDATE OR DELETE ON facturas
  FOR EACH ROW EXECUTE FUNCTION auditar_cambio();
```

Guardar las dos columnas —`tenant_ctx` y `tenant_fila`— parece redundante porque la política garantiza que coinciden. Justamente por eso es valioso: cualquier fila de auditoría donde difieran significa que la escritura entró por un camino que no pasó por las políticas, es decir, un rol con `BYPASSRLS`, un job, una migración o el dueño de la tabla. Una alerta sobre esa condición es una de las mejores señales de seguridad que puedes tener, y cuesta una consulta:

```
SELECT tabla, rol, app, count(*), max(ocurrido_en)
```

```
FROM auditoria
WHERE tenant_ctx IS DISTINCT FROM tenant_fila
  AND ocurrido_en > now() - interval '1 day'
GROUP BY 1, 2, 3
ORDER BY 4 DESC;
```

Nota que la tabla `auditoria` no lleva RLS y el rol de la aplicación no debería tener `SELECT` sobre ella. Un registro de auditoría que el propio sujeto auditado puede leer o modificar no sirve de mucho.

El cero filas sospechoso

El fallo más común en producción, con diferencia, no es una fuga: es que el contexto no se estableció y todo devuelve vacío. Detéctalo explícitamente en lugar de dejar que se manifieste como un ticket de soporte. Una función de contexto que lance excepción cuando falta el tenant convierte un fallo silencioso en un error 500 con traza, que es infinitamente más fácil de arreglar:

```
CREATE OR REPLACE FUNCTION app.tenant_actual()
RETURNS uuid
LANGUAGE plpgsql
STABLE
SET search_path = ''
AS $$
DECLARE
  v text := current_setting('app.tenant_id', true);
BEGIN
  IF v IS NULL OR v = '' THEN
    RAISE EXCEPTION 'contexto de tenant no establecido'
      USING ERRCODE = 'insufficient_privilege',
            HINT = 'Falta set_config(''app.tenant_id'', ..., true) en la transaccion';
  END IF;
  RETURN v::uuid;
END;
$$;
```

Hay una decisión de diseño aquí que conviene tomar conscientemente. Lanzar excepción es lo correcto para una API donde toda petición tiene un tenant. Si tienes rutas legítimas sin tenant —un endpoint de salud, una página pública de precios— esas rutas no deben tocar tablas con RLS, o necesitarás una variante que devuelva `NULL` y políticas que lo traten como "ninguna fila". Lo que no puedes hacer es devolver `NULL` y escribir la política de forma que `NULL` case con todo. Ese es el bug que convierte un despiste en una brecha.

Seguridad a nivel de columna, junto a RLS

RLS decide qué filas ves. No decide qué columnas. Son mecanismos ortogonales y a menudo necesitas los dos: el equipo de soporte puede ver las filas de todos los tenants pero no los números de tarjeta; un rol de analítica ve todas las filas pero no los nombres.

```
-- Privilegios por columna: soporte no ve el hash de la tarjeta
GRANT SELECT (id, tenant_id, importe, estado, creada_en)
  ON facturas TO rol_soporte;

-- Intentar leer la columna prohibida falla de forma explicita
```

```
SET ROLE rol_soporte;
SELECT tarjeta_hash FROM facturas LIMIT 1;
-- ERROR: permission denied for column tarjeta_hash
```

Para casos donde prefieres enmascarar en vez de denegar, una vista con `security_barrier` hace el trabajo, siempre que recuerdes que sin esa opción la vista no es una frontera de seguridad:

```
CREATE VIEW facturas_soporte WITH (security_barrier = true) AS
SELECT id,
       tenant_id,
       importe,
       estado,
       'XXXX-' || right(tarjeta_ultimos4, 4) AS tarjeta_mostrable,
       creada_en
FROM facturas;

GRANT SELECT ON facturas_soporte TO rol_soporte;
REVOKE ALL ON facturas FROM rol_soporte;
```

A partir de PostgreSQL 15 existe además `CREATE VIEW ... WITH (security_invoker = true)`, que hace que la vista se ejecute con los permisos y las políticas de quien la consulta en lugar de las del dueño. Para vistas sobre tablas con RLS esto suele ser lo que quieres: sin `security_invoker`, una vista propiedad de un rol privilegiado se salta las políticas de las tablas base y se convierte en la puerta trasera de la que hablamos antes.

RLS frente a schema-por-tenant y base-por-tenant

RLS no es la única forma de aislar tenants, y elegir mal cuesta caro porque migrar entre modelos es un proyecto, no un refactor. Un resumen honesto de los tres enfoques:

Fila compartida con RLS. Una sola base, una sola tabla por entidad, `tenant_id` en cada fila. Escala a decenas de miles de tenants sin despeinarse. Las migraciones son una sola operación. El coste operativo por tenant tiende a cero. A cambio, el aislamiento es lógico: un error en una política es un incidente para todos, y las cargas de trabajo compiten por los mismos recursos, así que un tenant pesado degrada al resto. Es el modelo por defecto para SaaS con muchos clientes pequeños o medianos.

Un esquema por tenant. Misma base de datos, un esquema distinto por cliente. El aislamiento es más fuerte y más fácil de razonar: si el `search_path` apunta al esquema equivocado, normalmente falla en vez de devolver datos ajenos. Permite personalizar el esquema por cliente, cosa que en el enterprise a veces se paga bien. El problema es operativo: cada migración se multiplica por el número de tenants, y con unos cientos de esquemas el catálogo de PostgreSQL empieza a notarse en el tiempo de planificación y en las conexiones. Funciona bien hasta el orden de cientos.

Una base o una instancia por tenant. Aislamiento máximo, incluido el de recursos y el de radio de explosión. Es lo que quieres si un cliente exige residencia de datos en su región o su propia clave de cifrado. También es lo más caro: *N* backups, *N* ventanas de mantenimiento, *N* despliegues de migración, y un problema de conexiones que resolverás con más infraestructura. Razonable con decenas de clientes grandes; inviable con miles.

Y una nota práctica: estos modelos se combinan. El patrón más común en SaaS maduro es RLS para la larga

cola de clientes y una instancia dedicada para los tres o cuatro que pagan lo suficiente como para exigirla. Si diseñas la capa de acceso a datos para que el tenant se resuelva en un solo sitio —el middleware que abre la transacción—, mover un cliente de un modelo a otro es un cambio de configuración y no una reescritura.

Adoptar RLS en una aplicación que ya está en producción

Casi nadie empieza con RLS. Lo normal es llegar a él con doscientas consultas que ya llevan `WHERE tenant_id = ?` y el miedo razonable a que activar políticas rompa algo el viernes. Se puede hacer sin downtime y sin drama, en fases, con la propiedad de que cada fase es reversible.

Fase 1: preparar sin activar

Crea el rol de aplicación separado del dueño de las tablas, la función de contexto y todas las políticas. Todavía no habilites RLS. En este punto las políticas existen en el catálogo y no hacen absolutamente nada, así que el riesgo es cero. Aprovecha para arreglar lo aburrido: tablas sin `tenant_id`, columnas que lo permiten `NULL`, índices que no lo tienen primero.

```
-- Un NOT NULL sin bloquear la tabla (PostgreSQL 12+)
ALTER TABLE facturas ADD CONSTRAINT facturas_tenant_no_nulo
  CHECK (tenant_id IS NOT NULL) NOT VALID;
ALTER TABLE facturas VALIDATE CONSTRAINT facturas_tenant_no_nulo;
```

El `NOT VALID` seguido de `VALIDATE` es el patrón para no coger un `ACCESS EXCLUSIVE` largo: la primera sentencia es instantánea y sólo aplica a filas nuevas, la segunda escanea la tabla con un bloqueo mucho más suave. En una tabla de cien millones de filas esa diferencia es la que hay entre un despliegue normal y una caída.

Fase 2: modo sombra

Antes de activar nada, verifica que la política *habría* devuelto lo mismo que tus `WHERE` actuales. Se hace en una réplica o en una copia, comparando conteos por tenant con y sin política:

```
-- En un entorno de pruebas con datos reales
ALTER TABLE facturas ENABLE ROW LEVEL SECURITY;
ALTER TABLE facturas FORCE ROW LEVEL SECURITY;

DO $$
DECLARE
  t record;
  con_politica bigint;
  con_where    bigint;
  fallos       int := 0;
BEGIN
  FOR t IN SELECT id FROM tenants LOOP
    PERFORM set_config('app.tenant_id', t.id::text, true);
    SET LOCAL ROLE app_usuario;
    SELECT count(*) INTO con_politica FROM facturas;
    RESET ROLE;
    SELECT count(*) INTO con_where FROM facturas WHERE tenant_id = t.id;

    IF con_politica <> con_where THEN
      RAISE WARNING 'tenant %: politica=% where=%', t.id, con_politica, con_where;
```

```
      fallos := fallos + 1;
    END IF;
  END LOOP;
  RAISE NOTICE 'tenants con discrepancia: %', fallos;
END;
$$;
```

Cualquier discrepancia es información valiosísima antes de tocar producción. Las causas típicas: filas huérfanas con un `tenant_id` que ya no existe, tablas de unión donde el tenant se deduce por join en vez de estar denormalizado, y registros históricos importados de una migración antigua sin tenant. Todas se arreglan con datos, no con políticas.

Fase 3: activar, con los WHERE todavía puestos

Ahora sí: `ENABLE` y `FORCE` en producción, tabla por tabla, empezando por la menos crítica. Deja los `WHERE tenant_id = ?` del código exactamente donde están. Son redundantes con la política y esa redundancia es justo lo que quieres durante la transición: si te equivocaste en una política y es demasiado laxa, el `WHERE` te salva; si te equivocaste y es demasiado estricta, verás cero filas y lo detectarás en cuestión de minutos. Además, tener el filtro explícito ayuda al planificador y evita sorpresas de rendimiento en el mismo despliegue en que cambias la semántica.

Fase 4: retirar los WHERE, o no

Pasadas unas semanas sin incidentes, puedes empezar a quitar los filtros manuales. Mi opinión, después de haber vivido las dos versiones: quítalos de las consultas nuevas y de las que toques de todas formas, pero no hagas una campaña de limpieza masiva. El beneficio de borrar un `WHERE` que ya funciona es estético; el riesgo de un *pull request* de trescientos ficheros que nadie revisa de verdad es real. Lo que sí merece la pena eliminar sin excepción es cualquier construcción dinámica de ese filtro, porque es donde vive la inyección.

Hay una excepción a esa moderación: si tu objetivo es que un desarrollador nuevo no pueda escribir una consulta insegura, en algún momento hay que quitar el filtro de en medio, porque mientras esté ahí la gente sigue pensando que el aislamiento es responsabilidad suya. Es una decisión de cultura de equipo tanto como técnica.

Integraciones concretas

Supabase y PostgREST

En Supabase el tenant no viaja en una variable de sesión que tú estableces, sino en las claims del JWT que PostgREST pone en `request.jwt.claims`. El patrón cambia, pero las reglas son las mismas:

```
-- Leer una claim del token de forma segura
CREATE OR REPLACE FUNCTION app.tenant_del_token()
RETURNS uuid
LANGUAGE sql
STABLE
SET search_path = ''
AS $$
  SELECT NULLIF(
    current_setting('request.jwt.claims', true)::jsonb ->> 'tenant_id',
```

```
    ''
  )::uuid;
$$;

CREATE POLICY aislamiento_tenant ON public.facturas
  AS PERMISSIVE FOR ALL
  TO authenticated                          -- nunca a anon, nunca a public
  USING      (tenant_id = (SELECT app.tenant_del_token()))
  WITH CHECK (tenant_id = (SELECT app.tenant_del_token()));
```

Tres detalles que marcan la diferencia aquí. El `TO authenticated` evita que la política se evalúe para el rol anónimo, lo que además ahorra trabajo. El `(SELECT ...)` alrededor de la función es el mismo truco de InitPlan que ya conoces, y en Supabase es la recomendación oficial precisamente porque el impacto en tablas grandes es enorme: sin él, la función se evalúa una vez por fila. Y el segundo argumento `true` en `current_setting` hace que devuelva `NULL` en lugar de lanzar error cuando la claim no existe, que es lo que quieres para que la política simplemente no case.

Un aviso específico de este entorno: la clave `service_role` se salta RLS por completo. Si acaba en el cliente, en una variable de entorno con prefijo público o en un repositorio, todas las políticas que has escrito dejan de existir. Trátala como una credencial de superusuario, porque a efectos prácticos lo es.

Django

Django no tiene soporte nativo de RLS, pero un middleware que abra la transacción y fije el contexto cubre el caso completo:

```
# middleware.py
from django.db import connection, transaction

class TenantRLSMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        tenant_id = resolver_tenant(request)   # de subdominio, JWT, sesion...
        if tenant_id is None:
            return self.get_response(request)  # rutas publicas: sin tocar RLS

        with transaction.atomic():
            with connection.cursor() as cur:
                cur.execute(
                    "SELECT set_config('app.tenant_id', %s, true)",
                    [str(tenant_id)],
                )
            return self.get_response(request)
```

Dos cosas importantes. Primera: `ATOMIC_REQUESTS = True` en la configuración de la base de datos hace lo mismo de forma más limpia, envolviendo cada petición en una transacción; si lo activas, el middleware sólo tiene que hacer el `set_config`. Segunda: los comandos de gestión y las tareas de Celery no pasan por el middleware. Cada uno necesita fijar su propio contexto, y ese es exactamente el sitio donde la gente se olvida y acaba dando permisos de `BYPASSRLS` "temporalmente" a la aplicación entera.

```
# Para Celery, un decorador explicito
from contextlib import contextmanager
from django.db import connection, transaction

@contextmanager
def contexto_tenant(tenant_id):
    with transaction.atomic():
        with connection.cursor() as cur:
            cur.execute("SELECT set_config('app.tenant_id', %s, true)",
                        [str(tenant_id)])
        yield

@shared_task
def generar_informe(tenant_id, mes):
    with contexto_tenant(tenant_id):
        ...   # todo lo de dentro ve solo los datos de ese tenant
```

Prisma

Prisma no expone un gancho de "antes de cada consulta" sobre la conexión, así que el patrón es una transacción interactiva con una extensión de cliente:

```
import { PrismaClient, Prisma } from '@prisma/client'

const prisma = new PrismaClient()

export async function conTenant<T>(
  tenantId: string,
  fn: (tx: Prisma.TransactionClient) => Promise<T>,
): Promise<T> {
  return prisma.$transaction(async (tx) => {
    // Parametrizado: nunca interpolar el tenantId en la cadena
    await tx.$executeRaw`SELECT set_config('app.tenant_id', ${tenantId}, true)`
    return fn(tx)
  })
}

// Uso
const facturas = await conTenant(req.tenantId, (tx) =>
  tx.factura.findMany({ where: { estado: 'pendiente' } }),
)
```

El punto crítico es usar `tx` y no `prisma` dentro del callback. Si por costumbre escribes `prisma.factura.findMany(...)`, esa consulta sale por una conexión distinta del pool, sin el contexto, y verás cero filas o un error de la función de contexto. Es un error fácil de cometer y fácil de detectar si tienes la función que lanza excepción cuando falta el tenant. Una regla de ESLint que prohíba usar el cliente global dentro de un bloque `conTenant` cierra el asunto del todo.

Ocho errores que anulan todo el trabajo

1. **La aplicación se conecta como dueño de las tablas** (o con `BYPASSRLS`). Las políticas existen, se ven en `pg_policies` y no hacen nada. Rol separado y `FORCE ROW LEVEL SECURITY`.

2. `SET` **en lugar de** `SET LOCAL`**.** Con pooling en modo transacción el contexto sobrevive a la petición y lo hereda el siguiente cliente. Es un fallo de aislamiento intermitente, dependiente de la carga y prácticamente imposible de reproducir en local.

3. **Interpolar el identificador de tenant en el SQL del** `SET`**.** Los comandos `SET` no aceptan parámetros, así que la gente concatena. Usa `set_config()` con placeholder.

4. **Índices sin** `tenant_id` **como primera columna.** La política pasa de condición de índice a filtro y el coste de cada consulta crece con el número total de filas de la instancia, no con las del tenant.

5. **Llamar a la función de contexto sin** `(SELECT ...)`**.** Una evaluación por fila en lugar de una por consulta, y adiós al uso del índice.

6. **Vistas sin** `security_invoker`**.** Una vista creada por el dueño de las tablas devuelve todos los tenants a cualquiera que tenga `SELECT` sobre ella.

7. **Índices únicos sin** `tenant_id`**.** Funcionalmente incorrectos (dos clientes no pueden tener la factura número 1) y además filtran la existencia de filas ajenas a través del error de duplicado.

8. **Tablas nuevas sin RLS.** El fallo más probable a los seis meses. Se previene con el test de catálogo en CI, no con disciplina.

Checklist de producción

- El rol de la aplicación no es dueño de ninguna tabla y no tiene `SUPERUSER`, `BYPASSRLS` ni `CREATEROLE`.
- Todas las tablas con `tenant_id` tienen `ENABLE` *y* `FORCE ROW LEVEL SECURITY`, verificado por un test de CI.
- La política de aislamiento es `RESTRICTIVE` y define `USING` y `WITH CHECK` por separado.
- Todas las expresiones de contexto van envueltas en `(SELECT ...)`.
- El contexto se fija con `set_config(..., true)` dentro de una transacción explícita, con el valor pasado como parámetro.
- Todos los índices de tablas multi-tenant empiezan por `tenant_id`, incluidos los únicos.
- Las claves foráneas entre tablas de tenant son compuestas e incluyen `tenant_id`.
- Ninguna vista expuesta al rol de aplicación carece de `security_invoker = true`.
- La lista de roles con `BYPASSRLS` está inventariada, es corta y sus credenciales están rotadas y auditadas.
- Existe un `EXPLAIN` guardado de las tres consultas más calientes mostrando `Index Cond` sobre `tenant_id`.
- Los procesos de backup, ETL y réplica lógica corren con roles conocidos y documentados.

Preguntas frecuentes

¿RLS sustituye al filtrado por tenant en la aplicación? No, y no deberías quererlo. Sigue filtrando en la aplicación: el planificador trabaja mejor, las consultas se leen mejor y no dependes de una única capa. RLS es la red de seguridad que atrapa el caso que se te escapó, no el mecanismo principal.

¿Cuánto cuesta en rendimiento? Bien implementada —política con `(SELECT ...)` e índices encabezados por `tenant_id`— la sobrecarga es del orden de una comparación de igualdad extra por consulta, indistinguible del ruido. Mal implementada puede convertir un `Index Scan` en un `Seq Scan` con una llamada a función por fila. El rango entre ambos extremos es de varios órdenes de magnitud, y lo decide el `EXPLAIN`, no la intuición.

¿Y si necesito un "modo soporte" para ver los datos de un cliente? Un rol distinto con su propia política permisiva, un parámetro de sesión propio y un registro de auditoría de cada uso. Nunca `BYPASSRLS` en la ruta de la aplicación, porque entonces esa credencial se convierte en la llave maestra de toda tu base de datos.

¿Funciona con Prisma, Drizzle o Django? Sí, porque es transparente para el ORM: RLS actúa en el servidor. Lo único que necesitas del cliente es una transacción explícita por petición donde fijar el contexto. En Prisma, un `$transaction` interactivo envuelto en una extensión; en Django, un middleware que abra la transacción y ejecute el `set_config`.

¿Base de datos por tenant o esquema por tenant no sería más simple? Aísla mejor y no necesita nada de esto, pero el coste de operación crece de forma lineal: cada migración se multiplica por el número de tenants y PostgreSQL empieza a sufrir con miles de esquemas. El esquema compartido con RLS es el punto de equilibrio para la mayoría de productos SaaS; base de datos dedicada tiene sentido para clientes grandes con requisitos regulatorios, normalmente como un plan aparte.

¿Puedo aplicar RLS a una tabla existente sin downtime? Sí, y el orden importa. Primero despliega el código que fija el contexto en cada transacción, luego crea las políticas, y solo al final ejecuta `ENABLE` y `FORCE`. Si inviertes el orden, cualquier petición en vuelo que no fije el contexto empieza a ver una base de datos vacía. Los `ALTER TABLE ... ENABLE ROW LEVEL SECURITY` toman un lock exclusivo brevísimo, así que lánzalos con `lock_timeout` puesto.

Para llevarse a casa

RLS no es una función exótica ni un sustituto de escribir buen código. Es un cambio en el modelo de fallos: pasas de un sistema donde una omisión filtra datos en silencio a uno donde una omisión rompe la funcionalidad de forma ruidosa. Ese cambio de dirección es la razón por la que merece la pena el trabajo de configurarlo bien.

Si vas a implementarlo, el orden que menos duele es: rol separado y `FORCE`, política restrictiva con `(SELECT ...)`, contexto con `set_config` transaccional, índices encabezados por `tenant_id`, y los tests de CI desde el primer día. Lo demás son detalles que puedes ir cerrando; esos cinco puntos son los que deciden si el aislamiento es real.

English version

Row-Level Security in PostgreSQL: Multi-Tenant Isolation You Cannot Forget

The WHERE clause somebody will forget

In a multi-tenant application, nearly every query carries the same clause: `WHERE tenant_id = ?`. It works until someone ships a new endpoint on a Friday afternoon, a reporting job writes a JOIN without the filter, or the ORM generates a subquery that slips past the default scope. The result isn't an ordinary bug: it's one customer seeing another customer's data.

Row-Level Security (RLS) moves that filter to the one place where it can't be forgotten: inside PostgreSQL. Once enabled, the engine appends the condition to every query that touches the table — whether it comes from your API, a migration script, or someone with `psql` open against production. It doesn't replace application-level filtering (you still want that for performance and readability), but it turns an isolation failure from "likely, eventually" into "requires bypassing two layers at once".

This guide covers the whole model: the role design that makes policies actually apply, policy syntax, how to propagate the tenant without leaking it across pooled requests, the performance trick that separates a free policy from one that multiplies every query's cost by N, the back doors (views, functions, foreign keys), and the tests that stop all of it from eroding over time.

How RLS works, in two sentences

When you enable RLS on a table, PostgreSQL attaches that table's policies as extra predicates to any `SELECT`, `INSERT`, `UPDATE` or `DELETE` touching it. If no policy applies, the default behaviour is to deny: the table behaves as if it were empty.

That "deny by default" is the property that makes RLS worth the effort. Forgetting to write a policy breaks functionality loudly and immediately; forgetting a `WHERE` leaks data silently for months. You want the first kind of failure.

USING and WITH CHECK are not the same thing

Each policy can carry two expressions, and they do different jobs:

- `USING` filters *existing* rows. It applies to `SELECT` and to the candidate rows of `UPDATE` and `DELETE`. Rows that fail the expression simply don't exist for that session: a `DELETE` that doesn't reach them reports zero rows affected, not an error.

- `WITH CHECK` validates *new* rows. It applies to `INSERT` and to the result of an `UPDATE`. When it fails, PostgreSQL raises an explicit error: `new row violates row-level security policy`.

If you define `USING` and omit `WITH CHECK` on a `FOR ALL` policy, PostgreSQL reuses the `USING` expression for validation too. That's convenient, but write both explicitly: the day you need a tenant to read archived rows without being able to create them, the difference matters.

PERMISSIVE and RESTRICTIVE

Permissive policies (the default) combine with **OR**: one match is enough for the row to pass. Restrictive policies combine with **AND**: all of them must hold. In multi-tenant systems you want your isolation policy to be **RESTRICTIVE**, sitting underneath the permissive business policies. That way, the day someone adds a permissive "admins see everything" policy, the tenant filter still applies on top of it.

Roles first, policies second

The most common RLS mistake isn't in the policies: it's in the role the application connects as.

By default, RLS **does not apply to the table owner**. Nor to superusers, nor to roles carrying the `BYPASSRLS` attribute. If your application connects as the same user that runs migrations — the usual outcome when this gets set up in a hurry — the policies are there, they show up in `pg_policies`, and they filter absolutely nothing. Everything looks configured and nothing is protected.

There are two defences and you want both:

```
-- 1) A separate application role with no special privileges
--    that does NOT own the tables.
CREATE ROLE app_user LOGIN PASSWORD '...';
-- No SUPERUSER. No BYPASSRLS. No CREATEROLE.

GRANT USAGE ON SCHEMA public TO app_user;
GRANT SELECT, INSERT, UPDATE, DELETE
    ON ALL TABLES IN SCHEMA public TO app_user;

-- Cover future tables too:
ALTER DEFAULT PRIVILEGES IN SCHEMA public
    GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO app_user;

-- 2) FORCE: policies apply to the table owner as well.
ALTER TABLE invoices ENABLE ROW LEVEL SECURITY;
ALTER TABLE invoices FORCE  ROW LEVEL SECURITY;
```

`FORCE ROW LEVEL SECURITY` closes the owner loophole. It does not affect superusers or `BYPASSRLS` roles: those always go straight through, by design, so that backups and maintenance keep working. Which is exactly why that list needs to be short and audited:

```
SELECT rolname, rolsuper, rolbypassrls
FROM pg_roles
WHERE rolcanlogin AND (rolsuper OR rolbypassrls)
ORDER BY rolname;
```

If your API user shows up in that result, RLS is protecting you from nothing.

The schema and the base policy

Working model: a `tenants` table plus business tables carrying a non-null `tenant_id` column. The most important design decision is that `tenant_id` lives on *every* table, including the ones where you could derive it

through a JOIN. Denormalising that column is precisely what lets the policy be a cheap equality comparison instead of a per-row subquery.

```
CREATE TABLE tenants (
    id   uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    slug text NOT NULL UNIQUE
);

CREATE TABLE invoices (
    id          uuid PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id   uuid NOT NULL REFERENCES tenants(id),
    number      text NOT NULL,
    total_cents bigint NOT NULL,
    status      text NOT NULL DEFAULT 'draft',
    created_at  timestamptz NOT NULL DEFAULT now()
);

-- Uniqueness ALWAYS includes the tenant: invoice numbers repeat
-- across tenants, and that is correct.
CREATE UNIQUE INDEX invoices_tenant_number_key
    ON invoices (tenant_id, number);

-- The index the policy will use. tenant_id first, always.
CREATE INDEX invoices_tenant_created_idx
    ON invoices (tenant_id, created_at DESC);
```

The context function

The active tenant travels in a custom session parameter. Wrapping the read in a function centralises the type cast and the "no tenant" case, and keeps the policies readable.

```
CREATE OR REPLACE FUNCTION current_tenant_id() RETURNS uuid
LANGUAGE sql STABLE AS $$
    SELECT NULLIF(current_setting('app.tenant_id', true), '')::uuid;
$$;
```

Three details that look minor and aren't:

- The second argument `true` to `current_setting` means "if the parameter doesn't exist, return NULL instead of raising". Without it, any query without context blows up with `unrecognized configuration parameter` — health checks included.
- `NULLIF(..., '')` turns the empty string into NULL. Without it, an empty `app.tenant_id` produces a `uuid` cast error in the middle of the query, with a message that tells you nothing useful.
- `STABLE`, not `VOLATILE`: the value doesn't change within a query and the planner needs to know that in order to optimise.

With NULL as the result, the comparison `tenant_id = NULL` evaluates to NULL, which is not `true`, so no row passes. No context, no data. That is exactly the behaviour you want: a misconfigured connection fails visibly instead of serving the entire database.

The policy

```
CREATE POLICY tenant_isolation ON invoices
    FOR ALL
    TO app_user
    USING      (tenant_id = (SELECT current_tenant_id()))
    WITH CHECK (tenant_id = (SELECT current_tenant_id()));
```

Note the `(SELECT ...)` around the call. This isn't a matter of style: it's the difference between a policy that costs essentially nothing and one that costs a function call per candidate row. Details in the performance section below.

When business policies show up

Sooner or later you'll want rules beyond the tenant: a *viewer* role that can't see drafts, support access to a subset. The moment you add a second permissive policy, the `OR` can open a hole. The robust shape is to separate isolation from permission:

```
-- Restrictive guardrail: ANDed with EVERYTHING else.
CREATE POLICY tenant_guard ON invoices
    AS RESTRICTIVE FOR ALL TO app_user
    USING      (tenant_id = (SELECT current_tenant_id()))
    WITH CHECK (tenant_id = (SELECT current_tenant_id()));

-- Permissive business policies, which can no longer cross tenants.
CREATE POLICY invoices_read ON invoices
    FOR SELECT TO app_user
    USING (status <> 'draft' OR current_setting('app.role', true) = 'editor');

CREATE POLICY invoices_write ON invoices
    FOR ALL TO app_user
    USING      (current_setting('app.role', true) = 'editor')
    WITH CHECK (current_setting('app.role', true) = 'editor');
```

Now a badly written new permissive policy can at worst grant too much *within* a tenant. Cross-customer access is blocked by the restrictive guardrail, which nobody is going to touch.

Propagating the tenant without leaking it between requests

This is where most real implementations break. The correct way to set the context is this:

```
BEGIN;
SELECT set_config('app.tenant_id', '9f3c1b2e-...', true);  -- true = local
SELECT id, number FROM invoices WHERE status = 'open';
COMMIT;  -- the context disappears with the transaction
```

Two rules, and both have security consequences.

Rule 1: transaction-local, never session-wide

A plain `SET` lasts for the whole session. With a connection pool — your application's pool or PgBouncer in *transaction* mode — that session gets reused for the next request, which may belong to a different customer.

The parameter survives, the policy reads it, and tenant B runs its queries with tenant A's context. The application has no visible bug: the queries are properly parameterised and the code looks correct.

`SET LOCAL` and its functional equivalent `set_config(name, value, true)` are reverted when the transaction ends, on COMMIT or ROLLBACK. If you run PgBouncer in *transaction* mode this isn't a recommendation — it's the only variant that works.

Rule 2: `set_config`, not string interpolation

`SET LOCAL app.tenant_id = $1` is not valid SQL: `SET` commands don't take parameters. The immediate temptation is to build the statement by concatenating the identifier, which is exactly where SQL injection lives. `set_config()` is an ordinary function and does accept a placeholder. Always use it.

Python: SQLAlchemy 2.0

```
from contextlib import contextmanager

from sqlalchemy import create_engine, text
from sqlalchemy.orm import sessionmaker

engine = create_engine(
    "postgresql+psycopg://app_user:...@db:5432/app",
    pool_pre_ping=True,
)
SessionFactory = sessionmaker(engine)


@contextmanager
def tenant_session(tenant_id: str):
    """A session whose tenant context is scoped to the transaction."""
    with SessionFactory() as session:
        with session.begin():                    # BEGIN ... COMMIT / ROLLBACK
            session.execute(
                text("SELECT set_config('app.tenant_id', :tid, true)"),
                {"tid": str(tenant_id)},
            )
            yield session
        # On exit: COMMIT, and the local parameter is discarded with it.
```

Usage stays clean and — this is the point — isolation no longer depends on whoever writes the query remembering anything:

```
with tenant_session(request.state.tenant_id) as s:
    rows = s.execute(
        text("""
            SELECT id, number, total_cents
            FROM invoices
            ORDER BY created_at DESC
            LIMIT 50
        """)
    ).all()
# Not a single mention of tenant_id. PostgreSQL added it for you.
```

One integration detail: if your middleware can't resolve the tenant for a request, don't open the session at all. Fail with a 401 or 403 before touching the database. A transaction without context isn't dangerous (it sees nothing), but it produces confusing errors instead of a clear message.

TypeScript: node-postgres

```
import { Pool, PoolClient } from "pg";

const pool = new Pool({ connectionString: process.env.DATABASE_URL, max: 20 });

export async function withTenant<T>(
  tenantId: string,
  fn: (db: PoolClient) => Promise<T>,
): Promise<T> {
  const client = await pool.connect();
  try {
    await client.query("BEGIN");
    // set_config takes parameters; SET LOCAL does not.
    await client.query("SELECT set_config('app.tenant_id', $1, true)", [tenantId]);
    const result = await fn(client);
    await client.query("COMMIT");
    return result;
  } catch (err) {
    await client.query("ROLLBACK");
    throw err;
  } finally {
    client.release();
  }
}

const invoices = await withTenant(req.tenantId, async (db) => {
  const { rows } = await db.query(
    "SELECT id, number, total_cents FROM invoices ORDER BY created_at DESC LIMIT 50",
  );
  return rows;
});
```

With Prisma the equivalent pattern is an interactive `$transaction` whose first statement is the `set_config`, wrapped in a client extension so no repository can skip it. With Drizzle, a `db.transaction()` with the same first line. The idea doesn't change: **one explicit transaction per request, and the context is set inside it**.

Performance: the `(SELECT ...)` that changes everything

Write the policy like this and it works:

```
USING (tenant_id = current_tenant_id())
```

Write it like this and it's also fast:

```
USING (tenant_id = (SELECT current_tenant_id()))
```

The difference is that the scalar subquery makes the planner produce an **InitPlan** node: it is evaluated once per

query execution and the result becomes available as a parameter. Without it, even though the function is marked `STABLE`, PostgreSQL can end up evaluating it for every candidate row inside the security predicate.

The second-order effect is the one that really matters. With the value resolved to a parameter, the planner can use it as an *index key*. Without it, the predicate stays a filter applied after reading rows. `EXPLAIN` shows it directly:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, number FROM invoices ORDER BY created_at DESC LIMIT 50;

-- What you want to see:
--   InitPlan 1 (returns $0)
--     ->  Result
--   Index Scan using invoices_tenant_created_idx on invoices
--         Index Cond: (tenant_id = $0)

-- What you do NOT want to see:
--   Seq Scan on invoices
--         Filter: (tenant_id = current_tenant_id())
--         Rows Removed by Filter: 4821390
```

The rule of thumb: `Index Cond` good, `Filter` with the function inside it bad. And that `Rows Removed by Filter` growing with your total tenant count is the signature of a system that will degrade at exactly the moment business goes well.

One caveat: the `(SELECT ...)` wrapper is only valid for expressions that **don't depend on the row**. If the expression references table columns, you can't lift it into an InitPlan.

Indexes: `tenant_id` goes first

RLS adds an equality predicate on `tenant_id` to all of your queries. An index on `(created_at)` stops being useful; the one you want is `(tenant_id, created_at)`. Go through every index on RLS-protected tables and prepend the tenant column. It's a mechanical change and it's usually half of the performance work in adopting RLS.

Policies that query other tables

When a user can belong to several tenants, the policy has to consult a memberships table. Doing that naively has two problems: it's expensive, and if `memberships` also has RLS, PostgreSQL fails with `infinite recursion detected in policy for relation "memberships"`.

One fix solves both — a `SECURITY DEFINER` function that resolves the list once:

```
CREATE OR REPLACE FUNCTION allowed_tenant_ids() RETURNS uuid[]
LANGUAGE sql STABLE SECURITY DEFINER
SET search_path = public, pg_temp
AS $$
    SELECT coalesce(array_agg(m.tenant_id), '{}')
    FROM memberships m
    WHERE m.user_id = NULLIF(current_setting('app.user_id', true), '')::uuid;
$$;
```

```
REVOKE EXECUTE ON FUNCTION allowed_tenant_ids() FROM PUBLIC;
GRANT  EXECUTE ON FUNCTION allowed_tenant_ids() TO app_user;

CREATE POLICY tenant_guard ON invoices
    AS RESTRICTIVE FOR ALL TO app_user
    USING      (tenant_id = ANY ((SELECT allowed_tenant_ids())))
    WITH CHECK (tenant_id = ANY ((SELECT allowed_tenant_ids())));
```

The `SET search_path` is not decorative: on a `SECURITY DEFINER` function it is mandatory, because without it anyone who can create objects in an earlier schema on the path can hijack name resolution and run code with the owner's privileges.

Views, functions and other back doors

RLS protects tables. Anything that wraps a table is a potential detour.

- **Views.** By default a view runs with the permissions of its *owner*, not of the caller. If the view belongs to the table owner, anyone who can read it sees every row of every tenant. Since PostgreSQL 15 the fix is explicit: `CREATE VIEW v WITH (security_invoker = true) AS ...`. Audit the existing views, not just the new ones.

- `SECURITY DEFINER` **functions.** They run as their owner, so they apply the *owner's* RLS. If the owner owns the tables and you haven't set `FORCE`, the function is a complete bypass. Use them surgically and with a pinned `search_path`.

- **Materialized views.** They refresh under the owner's role and the resulting snapshot contains every tenant's data. A matview does not inherit RLS: if you expose it to the application role it needs its own `tenant_id` column and its own policy, or it shouldn't be reachable at all.

- `COPY`. `COPY table TO` does apply `SELECT` policies. `COPY ... FROM`, however, is **not supported** on tables with RLS enabled — your bulk-load paths have to go through `INSERT`.

- `pg_dump`. A dump taken by a role subject to RLS requires `--enable-row-security` and produces a *partial* dump, which is a dangerous trap for backups. Take backups with a role that can bypass RLS, and treat that credential as what it is: critical.

Constraints and referential integrity: the covert channels

The PostgreSQL documentation says it plainly: referential integrity checks — primary keys, unique indexes and foreign keys — **always bypass RLS**. They have to, so that data integrity stays consistent for everyone. And that has two practical consequences.

A unique index reveals the existence of other tenants' rows. If `invoices(number)` were unique on its own, inserting `'A-001'` would return a duplicate-key error when another tenant already holds it. That's an oracle: it lets you enumerate data you can't read. The fix coincides with what you wanted for functional correctness anyway — put `tenant_id` inside every unique index.

A foreign key can cross tenants. The check reads the parent row bypassing RLS, so nothing stops an invoice belonging to tenant A from pointing at a customer of tenant B if someone manages to smuggle the identifier in. The strong defence isn't a policy: it's a composite foreign key that includes the tenant.

```
CREATE TABLE customers (
    id        uuid NOT NULL DEFAULT gen_random_uuid(),
    tenant_id uuid NOT NULL REFERENCES tenants(id),
    name      text NOT NULL,
    PRIMARY KEY (id),
    UNIQUE (tenant_id, id)          -- required so it can be referenced
);

ALTER TABLE invoices
    ADD COLUMN customer_id uuid NOT NULL,
    ADD CONSTRAINT invoices_customer_same_tenant_fk
        FOREIGN KEY (tenant_id, customer_id)
        REFERENCES customers (tenant_id, id);
```

With that constraint in place, a cross-tenant reference is impossible *structurally*: it fails even with RLS disabled, even when a superuser runs the script, even when the bug is in the application. That's the kind of guarantee that doesn't depend on anyone remembering anything.

Testing: isolation is proven, not assumed

RLS is one of those things that either has tests or decays. Two levels.

Behavioural tests

```
import pytest
from sqlalchemy import text
from sqlalchemy.exc import ProgrammingError


def test_one_tenant_cannot_see_the_other(tenant_a, tenant_b):
    with tenant_session(tenant_a) as s:
        s.execute(
            text("INSERT INTO invoices (tenant_id, number, total_cents) "
                 "VALUES (:t, 'A-001', 1000)"),
            {"t": tenant_a},
        )
    with tenant_session(tenant_b) as s:
        s.execute(
            text("INSERT INTO invoices (tenant_id, number, total_cents) "
                 "VALUES (:t, 'B-001', 2000)"),
            {"t": tenant_b},
        )

    with tenant_session(tenant_a) as s:
        numbers = [r[0] for r in s.execute(text("SELECT number FROM invoices")).all()]

    assert numbers == ["A-001"]     # B-001 simply does not exist for A
```

```
def test_cannot_write_into_another_tenant(tenant_a, tenant_b):
    # WITH CHECK raises SQLSTATE 42501 -> ProgrammingError in SQLAlchemy.
    with pytest.raises(ProgrammingError, match="row-level security"):
        with tenant_session(tenant_a) as s:
            s.execute(
                text("INSERT INTO invoices (tenant_id, number, total_cents) "
                     "VALUES (:t, 'X-001', 1)"),
                {"t": tenant_b},          # foreign tenant
            )


def test_no_context_sees_nothing(engine):
    with engine.connect() as c:           # no set_config
        total = c.execute(text("SELECT count(*) FROM invoices")).scalar()
    assert total == 0
```

That third test is the most valuable of the three and the one almost nobody writes. It proves the *deny by default* property: if someone later edits the policy so it stops comparing against the context, this test goes red even while the other two keep passing.

The test that prevents erosion

The most likely medium-term failure isn't a badly written policy — it's a new table nobody enabled RLS on. That case is detectable with a catalog query, and it belongs in CI:

```
EXEMPT = {"tenants", "alembic_version", "feature_flags"}


def test_every_table_with_tenant_id_has_forced_rls(engine):
    sql = text("""
        SELECT c.relname
        FROM pg_class c
        JOIN pg_namespace n ON n.oid = c.relnamespace
        WHERE n.nspname = 'public'
          AND c.relkind IN ('r', 'p')          -- tables and partitioned tables
          AND EXISTS (
              SELECT 1 FROM pg_attribute a
              WHERE a.attrelid = c.oid
                AND a.attname  = 'tenant_id'
                AND NOT a.attisdropped
          )
          AND NOT (c.relrowsecurity AND c.relforcerowsecurity)
    """)
    with engine.connect() as conn:
        missing = {r[0] for r in conn.execute(sql)} - EXEMPT

    assert not missing, f"Tables with tenant_id and no forced RLS: {sorted(missing)}"
```

Ship a migration adding a new table with a `tenant_id` and forget the two `ALTER TABLE` lines: the build fails before it ever reaches production. Twenty lines of test covering roughly 80% of the real long-term risk.

Migrations: the new table nobody protected

Almost no RLS incident starts with a badly written policy. It starts with a table someone created on a Tuesday afternoon and never ran `ENABLE ROW LEVEL SECURITY` on. PostgreSQL's model is open by default: a freshly created table has no RLS, and if your application role has `SELECT` on the schema, it sees the whole thing. The rest of your system can be flawless and that one table is still a hole.

The defence is not discipline. Discipline erodes the moment new people join the team or somebody is in a hurry before a demo. The defence is an automated check that fails loudly.

Auditing unprotected tables in one second

This query tells you, for every table in your application schemas, whether RLS is enabled, whether it is forced, and how many policies cover it. It is the first query I run when someone tells me "I think we have RLS":

```
SELECT
  n.nspname                       AS schema,
  c.relname                       AS table_name,
  c.relrowsecurity                AS rls_enabled,
  c.relforcerowsecurity           AS rls_forced,
  count(p.polname)                AS policies
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_policy p ON p.polrelid = c.oid
WHERE c.relkind IN ('r', 'p')            -- tables and partitioned tables
  AND n.nspname NOT IN ('pg_catalog', 'information_schema')
GROUP BY 1, 2, 3, 4
ORDER BY c.relrowsecurity ASC, policies ASC, 1, 2;
```

What you are looking for sits right at the top: rows with `rls_enabled = false`, or with `rls_enabled = true` but `policies = 0`. The second case is interesting and deserves a note: a table with RLS enabled and zero policies is not permissive, it is the opposite. PostgreSQL applies an implicit deny-all, so the table returns zero rows for everyone except the owner and roles with `BYPASSRLS`. That is failing closed, which is the right way to fail, but it usually means someone enabled RLS and forgot half the job.

Pay attention to `relforcerowsecurity` too. If the table owner and the application role are the same — which happens more often than it should — then without `FORCE` the policies do not apply to the owner and your isolation is decorative.

Turning it into a test that breaks the build

Wrap that query in a test. Not in a document, not in a checkbox on a review checklist: in something that turns CI red. An explicit exception list is fine as long as it stays short and each entry justifies itself in the code:

```
# tests/test_rls_coverage.py
import pytest

# Tables that legitimately carry no RLS. Every entry needs a reason.
EXEMPT = {
    "public.schema_migrations",   # migration metadata, no tenant data
    "public.plans",               # global plan catalogue, same for everyone
    "public.countries",           # read-only reference table
}
```

```
QUERY = """
SELECT n.nspname || '.' || c.relname AS table_name,
       c.relrowsecurity,
       c.relforcerowsecurity,
       count(p.polname) AS policies
FROM pg_class c
JOIN pg_namespace n ON n.oid = c.relnamespace
LEFT JOIN pg_policy p ON p.polrelid = c.oid
WHERE c.relkind IN ('r', 'p')
  AND n.nspname = 'public'
GROUP BY 1, 2, 3
"""

def test_every_table_has_rls(admin_connection):
    problems = []
    for table, enabled, forced, policies in admin_connection.execute(QUERY):
        if table in EXEMPT:
            continue
        if not enabled:
            problems.append(f"{table}: RLS not enabled")
        elif not forced:
            problems.append(f"{table}: RLS enabled but not FORCEd")
        elif policies == 0:
            problems.append(f"{table}: RLS enabled with no policy at all")
    assert not problems, "Unprotected tables:\n" + "\n".join(problems)
```

The detail that makes this test survive is the `EXEMPT` list. Without it, the first catalogue table somebody adds turns the test red, somebody marks it `skip` "temporarily", and you have lost the whole control. With it, adding an exception is a visible change in a code review, which is exactly where you want that conversation to happen.

An event trigger that shouts immediately

If you want the warning before the code even reaches CI, PostgreSQL has DDL event triggers. This one emits a `WARNING` the moment a table is created in `public`, in the same `psql` session or migration tool run:

```
CREATE OR REPLACE FUNCTION warn_table_without_rls()
RETURNS event_trigger
LANGUAGE plpgsql
AS $$
DECLARE
  obj record;
BEGIN
  FOR obj IN SELECT * FROM pg_event_trigger_ddl_commands()
  LOOP
    IF obj.command_tag = 'CREATE TABLE'
       AND obj.schema_name = 'public' THEN
      RAISE WARNING
        'Table % created without RLS. Add ENABLE + FORCE ROW LEVEL SECURITY and a policy.',
        obj.object_identity;
    END IF;
  END LOOP;
END;
$$;
```

```
CREATE EVENT TRIGGER tr_warn_table_without_rls
  ON ddl_command_end
  WHEN TAG IN ('CREATE TABLE')
  EXECUTE FUNCTION warn_table_without_rls();
```

You could use `RAISE EXCEPTION` instead of `WARNING` to abort the creation outright, but I would not: it breaks migration tools, `pg_restore` runs and scratch temporary tables in ways that are annoying to debug. The `WARNING` plus the CI test covers the same risk without the side effects.

The migration template

Reduce friction. If creating a new table with correct RLS means remembering five statements, someone will forget two. Keep a helper function and make the migration a one-liner:

```
CREATE OR REPLACE FUNCTION protect_tenant_table(table_name text)
RETURNS void
LANGUAGE plpgsql
AS $$
BEGIN
  EXECUTE format('ALTER TABLE %I ENABLE ROW LEVEL SECURITY', table_name);
  EXECUTE format('ALTER TABLE %I FORCE ROW LEVEL SECURITY', table_name);
  EXECUTE format($f$
    CREATE POLICY tenant_isolation ON %I
      AS PERMISSIVE FOR ALL
      TO app_user
      USING (tenant_id = (SELECT app.current_tenant()))
      WITH CHECK (tenant_id = (SELECT app.current_tenant()))
  $f$, table_name);
  EXECUTE format(
    'CREATE INDEX IF NOT EXISTS %I ON %I (tenant_id)',
    'idx_' || table_name || '_tenant', table_name);
END;
$$;

-- In the migration:
CREATE TABLE invoices (
  id          uuid PRIMARY KEY DEFAULT gen_random_uuid(),
  tenant_id   uuid NOT NULL,
  amount      numeric(12,2) NOT NULL,
  created_at  timestamptz NOT NULL DEFAULT now()
);
SELECT protect_tenant_table('invoices');
```

Note the `format` with `%I`: that is the correct way to interpolate identifiers into dynamic SQL. Using string concatenation here opens a table-name injection which, in a function running as the schema owner, is as serious as it sounds.

Connection pooling, in depth

We have already established that `SET LOCAL` is mandatory and that session-level `SET` is an isolation failure waiting to happen. It is worth understanding why, because the explanation tells you exactly which pooling

configurations are safe and which are not.

Why transaction mode and SET LOCAL fit together

PgBouncer in `pool_mode = transaction` assigns a server connection to a client only for the duration of a transaction. The instant `COMMIT` or `ROLLBACK` lands, that connection goes back to the pool and the next client may receive it. That is the dangerous moment: any state that survives the transaction travels with the connection to the next occupant.

`SET LOCAL`, and its functional equivalent `set_config(key, value, true)` with the third argument set to `true`, binds the value to the transaction. When it ends, PostgreSQL reverts it. There is no window. That single property is what lets RLS and transaction-mode pooling coexist without surprises.

With session-level `SET` the value persists. If request A sets tenant 42 and finishes, request B for tenant 7 may get that connection with `app.tenant_id = 42` still in place. If B's code sets its own tenant, nothing happens. If for any reason it does not — a route that skipped the middleware, a health check, a job that reused the pool — B reads tenant 42's data. And it will do so without any error, returning perfectly plausible rows. It is the worst class of failure there is.

```
-- Dangerous with transaction pooling:
SET app.tenant_id = '42';        -- survives the COMMIT

-- Correct:
BEGIN;
SELECT set_config('app.tenant_id', '42', true);   -- true = transaction-local
SELECT * FROM invoices;
COMMIT;                                            -- the value disappears here
```

DISCARD ALL is not your safety net

There is a temptation to say "PgBouncer runs `server_reset_query` between clients, so I'm covered". In transaction mode, by default, `server_reset_query` is *not* executed: `server_reset_query_always` ships disabled precisely because transaction mode assumes there is no leftover state to clean. And even if you turned it on, you would be paying an extra round trip per transaction to fix something `SET LOCAL` fixes for free. Configure it correctly and do not lean on the cleanup.

Prepared statements: the detail that used to bite

For years, using prepared statements with PgBouncer in transaction mode was a minefield: the statement got prepared on one server connection and the `EXECUTE` could land on another, producing the familiar "prepared statement does not exist" error. The usual fix was to disable them in the driver, which costs performance on repetitive queries — exactly the ones an ORM generates.

This is solved now. PgBouncer 1.21 introduced protocol-level prepared statement support in transaction mode via `max_prepared_statements`, which keeps an LRU cache of statements per server connection and transparently rewrites the identifiers. And PostgreSQL 17 added the missing piece: the ability to close a protocol-level prepared statement, which eliminates the errors that appeared when a driver issued `DEALLOCATE` on its own.

```
# pgbouncer.ini
[pgbouncer]
pool_mode = transaction
max_prepared_statements = 200     ; 0 disables the support
default_pool_size = 25
server_reset_query = DISCARD ALL
server_reset_query_always = 0     ; not needed in transaction mode
```

Important: this covers *protocol-level* prepared statements, the ones drivers emit when you use parameterised queries. The SQL commands `PREPARE`, `EXECUTE` and `DEALLOCATE` written by hand are forwarded straight through to the server and remain incompatible with transaction mode. If you have code that uses them, rewrite it with ordinary parameters.

Serverless, HTTP and transactionless drivers

Drivers that talk to the database over HTTP — Neon's serverless driver, the RDS Data API, PostgREST — complicate matters because each call may be its own implicit transaction. If your `set_config` goes in one call and your `SELECT` in another, the context no longer exists when the query arrives and the policy evaluates against `NULL`: zero rows, or worse, if you wrote the context function loosely, all of them.

There are two clean ways out. The first is to batch both statements into an explicit transaction that the driver sends as one unit. The second, and more robust, is to stop propagating the tenant through a session variable at all and have it arrive signed inside the token itself — which is what Supabase does with JWT claims. We come back to that in the integrations section.

RDS Proxy and pinning

If you use Amazon RDS Proxy, it helps to know that certain operations cause *pinning*: the proxy stops multiplexing and ties the client connection to a server connection until it closes. Using session-level `SET` is one of those operations. In other words, the unsafe way of propagating the tenant also destroys the benefit of the proxy. `SET LOCAL` inside a transaction does not pin. It is pleasant when the correct option is also the fast one.

Background jobs, ETL and backups

RLS protects the path from a user to the database. Half of the real accesses to a production database do not follow that path: migrations, nightly reports, exports, replication, backups, the script somebody runs from their laptop. Each one needs an explicit decision.

BYPASSRLS: who has it and why

The `BYPASSRLS` role attribute, which only a superuser can grant, lets a role skip policies entirely. Superusers and table owners with `BYPASSRLS` always do. It is a legitimate tool: an aggregation process computing platform-wide metrics needs to see every row. What is not legitimate is your API's connection role carrying that attribute.

```
-- Application role: no special privileges, subject to RLS
CREATE ROLE app_user LOGIN PASSWORD :'app_password'
  NOBYPASSRLS NOSUPERUSER NOCREATEDB NOCREATEROLE;

-- Analytics role: sees everything, but read-only and internal network only
CREATE ROLE analytics LOGIN PASSWORD :'analytics_password' BYPASSRLS;
```

```
GRANT USAGE ON SCHEMA public TO analytics;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO analytics;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
  GRANT SELECT ON TABLES TO analytics;

-- Audit: which roles can bypass RLS
SELECT rolname, rolsuper, rolbypassrls
FROM pg_roles
WHERE rolbypassrls OR rolsuper
ORDER BY rolname;
```

That last query should return a short, boring result, and it belongs in the same CI test as your table coverage check. A new `BYPASSRLS` appearing without anyone asking for it is a signal you want to see.

pg_dump, restores and the flag almost nobody knows

By default `pg_dump` errors out if it tries to dump a table with RLS active using a role that is subject to the policies, rather than silently exporting a subset. That is deliberate and very sensible: a partial backup that looks complete is worse than a backup that does not exist.

If what you actually want is a tenant-filtered dump — handing a customer their own data, for instance — there is `--enable-row-security`, which asks `pg_dump` to export with policies active. In that case you set the context yourself, and the result is explicitly a partial dump:

```
# Full backup: use a role with BYPASSRLS or the owner. Fails loudly if the
# role is subject to policies, which is what you want.
pg_dump -U postgres -Fc -f full.dump myapp

# Per-tenant export: policies active, context pinned
PGOPTIONS="-c app.tenant_id=a1b2c3d4-0000-0000-0000-000000000000" \
  pg_dump -U app_user --enable-row-security \
          --data-only -t invoices -t customers \
          -f tenant_a1b2.sql myapp
```

Two warnings about the second form. One: `PGOPTIONS` with `-c` sets the parameter at session level, not transaction level, which is fine here because this is a one-shot process with its own connection — but do not carry that pattern into the application. Two: review the output before you hand it over. A badly written policy on a secondary table can pull other tenants' rows into a file you are about to email.

Logical replication

Logical replication deserves its own paragraph because its interaction with RLS is surprising. If the replication role lacks `SUPERUSER` and `BYPASSRLS`, publisher row security policies execute during replication, which means the subscriber can receive a subset of the rows with no warning whatsoever. Your replica is incomplete and nothing tells you.

The PostgreSQL documentation recommends that, if you do not trust all table owners, you include `options=-crow_security=off` in the subscription connection string. With that, if an owner adds a policy, replication halts with an error instead of silently applying it. You would rather get an alert at three in the morning than a replica you cannot trust:

```
CREATE SUBSCRIPTION sub_analytics
  CONNECTION 'host=primary dbname=myapp user=replicator options=-crow_security=off'
  PUBLICATION pub_full;
```

COPY, bulk loads and WITH CHECK

`COPY ... FROM` respects `WITH CHECK` policies exactly like an `INSERT`. That is correct, and it has a practical consequence: loading a million rows will evaluate the policy expression a million times. If your `WITH CHECK` subqueries another table, the load becomes glacial. For bulk imports of trusted data, the sensible approach is to run the process under a dedicated `BYPASSRLS` role and validate `tenant_id` in the loader itself, or to use a staging table without RLS and move the data with a single already-validated `INSERT ... SELECT`.

Side-channel leaks: LEAKPROOF, error messages and functions

So far we have talked about rows being visible or not visible. There is a subtler category of leak: queries that return no forbidden row but do reveal information about it. It is a topic that almost never appears in RLS tutorials and is worth understanding, if only to know when not to worry about it.

How an error message reveals a row you cannot see

Picture an `invoices` table with per-tenant RLS and this query, issued by a user of tenant A:

```
SELECT * FROM invoices WHERE 1 / (amount - 1000) > 0;
```

If the planner decides to evaluate `1 / (amount - 1000)` *before* the policy condition, and any tenant anywhere has an invoice of exactly 1000, the query aborts with `division by zero`. The tenant A user has not seen tenant B's invoice, but has just confirmed it exists. Repeating the query with different values lets them extract data from rows the policy forbids them to read.

PostgreSQL blocks this, and the mechanism is the `LEAKPROOF` marking. Security conditions — those from RLS policies and from `security_barrier` views — are evaluated before any user-supplied predicate that could leak row contents. Only operators and functions marked `LEAKPROOF` may be reordered or pushed down past that barrier, because they are trusted: invoking them on rows invisible to the user leaks nothing about those rows.

What LEAKPROOF actually means

A function is `LEAKPROOF` if it has no side effects and, in particular, reveals no information about its arguments through any channel other than its return value. That includes error messages: a function that raises `ERROR: invalid value: %s` with the argument embedded is not leakproof, however pure it otherwise looks.

Many simple, heavily used operators are leakproof — the equality operators, for instance — and that is why RLS performance is usually reasonable: the most common filters can be pushed down without compromising anything. Only a superuser can mark a function `LEAKPROOF`, precisely because getting it wrong opens the exact hole the mechanism exists to close.

```
-- See which functions in your schema are leakproof
SELECT p.proname, p.proleakproof, pg_get_function_identity_arguments(p.oid) AS args
FROM pg_proc p
JOIN pg_namespace n ON n.oid = p.pronamespace
```

```
WHERE n.nspname = 'public'
ORDER BY p.proleakproof DESC, p.proname;
```

There is a useful special case: functions that take no arguments, or that receive none from the protected view or table, do not need to be marked `LEAKPROOF` to be pushed down, because they never receive data from the protected rows. Your `app.current_tenant()` function falls into that category, which is one of the reasons the `(SELECT ...)` trick we covered earlier works so well.

The cost of the barrier

It is worth being honest about the price. Views created with `security_barrier` may perform far worse than views created without it, and in general there is no way to avoid this: if the fastest possible plan may compromise security, it has to be rejected. The same applies, to a lesser degree, to tables with RLS. If you see a large, unexplained performance gap between a query under RLS and the same query as the table owner, check whether a non-leakproof predicate is preventing a selective filter from reaching the scan.

SECURITY DEFINER and search_path: the classic back door

`SECURITY DEFINER` functions execute with the privileges of whoever created them, not whoever calls them. They are necessary for controlled cross-tenant operations, and they are a textbook privilege-escalation vector if you do not pin the `search_path`.

The attack is simple: if your `SECURITY DEFINER` function does `SELECT ... FROM invoices` without schema-qualifying, an attacker who can create objects in a schema that comes earlier in the `search_path` — `pg_temp` being the usual suspect — puts their own `invoices` or their own operator there, and your privileged function ends up executing their code.

```
-- Bad: search_path inherited from the caller
CREATE FUNCTION transfer_ownership(inv_id uuid, new_tenant uuid)
RETURNS void
LANGUAGE plpgsql
SECURITY DEFINER
AS $$
BEGIN
  UPDATE invoices SET tenant_id = new_tenant WHERE id = inv_id;
END;
$$;

-- Good: empty search_path and everything qualified
CREATE FUNCTION app.transfer_ownership(inv_id uuid, new_tenant uuid)
RETURNS void
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = ''
AS $$
BEGIN
  IF NOT public.can_administer(current_user, new_tenant) THEN
    RAISE EXCEPTION 'not authorised';
  END IF;
  UPDATE public.invoices
     SET tenant_id = new_tenant
   WHERE id = inv_id;
```

```
END;
$$;

-- And never leave it open to everyone:
REVOKE ALL ON FUNCTION app.transfer_ownership(uuid, uuid) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION app.transfer_ownership(uuid, uuid) TO admin_role;
```

`SET search_path = ''` is stricter than `SET search_path = public, pg_temp` and that is why it is the current recommendation: it forces you to qualify every name, including types and operators, and it eliminates the entire class of name-resolution attacks. It is more verbose. It is worth it.

The trailing `REVOKE ... FROM PUBLIC` is not optional either. By default, `EXECUTE` on new functions is granted to `PUBLIC`. A `SECURITY DEFINER` function anyone can call is, literally, a privilege-escalation API you published by accident.

Advanced performance: measure before you opine

What a policy actually costs, in numbers

The right way to know what RLS costs you is to compare the same plan with and without policies, in the same session and over the same data. `row_security = off` lets you do exactly that if the role has `BYPASSRLS`:

```
-- With policies
SET ROLE app_user;
BEGIN;
SELECT set_config('app.tenant_id', :'tenant', true);
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT id, amount FROM invoices WHERE created_at > now() - interval '30 days';
COMMIT;
RESET ROLE;

-- Without policies, same data, same cache
SET ROLE analytics;   -- role with BYPASSRLS
SET row_security = off;
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT id, amount FROM invoices
WHERE tenant_id = :'tenant' AND created_at > now() - interval '30 days';
RESET row_security;
RESET ROLE;
```

What you want to see in the first plan is an `Index Cond` that includes `tenant_id`, not a `Filter`. If `tenant_id` shows up under `Filter`, PostgreSQL is reading other tenants' rows off disk and discarding them afterwards. It works, it is safe, and it is one to two orders of magnitude slower on a large table. Watch `Buffers` too: a large jump in `shared read` between the two plans is objective proof that the policy is not being used as an index condition.

Generic plans versus custom plans

This is a second-order effect that bites in production and almost never in development. When you execute the same prepared statement repeatedly, PostgreSQL evaluates whether it is worth caching a generic plan — one that ignores the specific parameter values — instead of replanning every time. From the sixth execution

onwards, if the generic plan's estimated cost is not worse, it sticks with it.

With multi-tenancy this can go badly. If you have one tenant with ten million rows and five hundred with two hundred each, the generic plan will be a compromise that suits none of them: sequential scans for the small ones, or nested loops for the big one. The `plan_cache_mode` parameter lets you force the behaviour:

```
-- Force per-execution replanning on queries sensitive to tenant size
BEGIN;
SET LOCAL plan_cache_mode = force_custom_plan;
SELECT set_config('app.tenant_id', :'tenant', true);
SELECT ... ;
COMMIT;
```

Do not set this globally: replanning has a cost and for most queries it does not pay off. Apply it with `SET LOCAL` only on the paths where you have measured the generic plan getting it wrong. And measure properly, with `pg_stat_statements`, before and after.

Partial and covering indexes

You already know `tenant_id` goes first in the composite index. Two refinements that usually pay off handsomely for very little effort:

```
-- Covering index: the query resolves without touching the heap
CREATE INDEX idx_invoices_tenant_date
  ON invoices (tenant_id, created_at DESC)
  INCLUDE (amount, status);

-- Partial index: if 95% of queries only look at active invoices,
-- don't pay to index the archived ones
CREATE INDEX idx_invoices_tenant_active
  ON invoices (tenant_id, created_at DESC)
  WHERE status <> 'archived';
```

The covering index matters especially under RLS because it avoids the heap fetch, and fewer table accesses means fewer rows to evaluate the policy against. The partial index shrinks the index, which on a table with years of history can be the difference between fitting in memory and not.

Partitioning by tenant: when yes and when no

If you have a handful of enormous tenants, list-partitioning on `tenant_id` lets the planner prune whole partitions and gives each tenant its own statistics, which solves the generic-plan problem at the root:

```
CREATE TABLE events (
  id          bigserial,
  tenant_id   uuid NOT NULL,
  kind        text NOT NULL,
  occurred_at timestamptz NOT NULL DEFAULT now()
) PARTITION BY LIST (tenant_id);

CREATE TABLE events_big_tenant
  PARTITION OF events FOR VALUES IN ('a1b2c3d4-0000-0000-0000-000000000000');
CREATE TABLE events_rest PARTITION OF events DEFAULT;
```

```
-- RLS on the parent table is inherited by the partitions
ALTER TABLE events ENABLE ROW LEVEL SECURITY;
ALTER TABLE events FORCE ROW LEVEL SECURITY;
CREATE POLICY tenant_isolation ON events
  FOR ALL TO app_user
  USING (tenant_id = (SELECT app.current_tenant()))
  WITH CHECK (tenant_id = (SELECT app.current_tenant()));
```

Now the warning: do not partition by tenant if you have thousands of similar tenants. Every partition is a table with its own catalogue entries, its own indexes and its own autovacuum work. At ten thousand partitions planning time explodes and maintenance operations become a problem in their own right. The rule of thumb I use: partitioning by tenant makes sense with dozens of tenants, not thousands. For thousands, partition by time if you need to and let the index on `tenant_id` do its job.

Observability: knowing which tenant did what

When RLS works well it is invisible, and that is a problem for debugging. A user says "I can't see my invoices" and you have to distinguish between three causes that produce the same symptom: there are no invoices, the tenant context was never set, or the policy has a bug. Without instrumentation, that investigation takes two hours. With instrumentation, two minutes.

Tag the connection

The cheapest and most useful trick: write the tenant and the request into `application_name` when you open the transaction, and add it to the log prefix. From then on, every slow query, every deadlock and every error in the log carries the tenant with it.

```
-- postgresql.conf
log_line_prefix = '%m [%p] %q%u@%d app=%a '
log_min_duration_statement = 500ms

# In the same place where you set the tenant
async def with_tenant(conn, tenant_id: str, request_id: str):
    await conn.execute(
        "SELECT set_config('app.tenant_id', $1, true), "
        "       set_config('application_name', $2, true)",
        tenant_id, f"api t={tenant_id[:8]} r={request_id[:8]}",
    )
```

Notice that `application_name` is also set with `set_config(..., true)`. It is an ordinary session parameter, so if you set it with `SET` it sticks to the connection and you will end up attributing one tenant's queries to another in your dashboards. Same mistake, same remedy.

Auditing who touched what

For sensitive tables, an audit trigger that records the effective tenant at write time gives you something to answer uncomfortable questions with later:

```
CREATE TABLE audit_log (
  id          bigserial PRIMARY KEY,
```

```
  table_name   text        NOT NULL,
  operation    text        NOT NULL,
  row_id       uuid,
  tenant_ctx   uuid,          -- tenant the session claimed to be
  tenant_row   uuid,          -- tenant of the affected row
  role_name    text        NOT NULL DEFAULT current_user,
  app          text        NOT NULL DEFAULT current_setting('application_name'),
  occurred_at  timestamptz NOT NULL DEFAULT clock_timestamp()
);

CREATE OR REPLACE FUNCTION audit_change()
RETURNS trigger
LANGUAGE plpgsql
SECURITY DEFINER
SET search_path = ''
AS $$
DECLARE
  r record;
BEGIN
  r := COALESCE(NEW, OLD);
  INSERT INTO public.audit_log (table_name, operation, row_id, tenant_ctx, tenant_row)
  VALUES (
    TG_TABLE_NAME,
    TG_OP,
    r.id,
    NULLIF(current_setting('app.tenant_id', true), '')::uuid,
    r.tenant_id
  );
  RETURN r;
END;
$$;

CREATE TRIGGER tr_audit_invoices
  AFTER INSERT OR UPDATE OR DELETE ON invoices
  FOR EACH ROW EXECUTE FUNCTION audit_change();
```

Storing both columns — `tenant_ctx` and `tenant_row` — looks redundant because the policy guarantees they match. That is exactly why it is valuable: any audit row where they differ means the write came in through a path that did not go through the policies, i.e. a `BYPASSRLS` role, a job, a migration or the table owner. An alert on that condition is one of the best security signals you can have, and it costs one query:

```
SELECT table_name, role_name, app, count(*), max(occurred_at)
FROM audit_log
WHERE tenant_ctx IS DISTINCT FROM tenant_row
  AND occurred_at > now() - interval '1 day'
GROUP BY 1, 2, 3
ORDER BY 4 DESC;
```

Note that `audit_log` carries no RLS and the application role should not have `SELECT` on it. An audit trail the audited subject can read or modify is not worth much.

The suspicious zero rows

By a wide margin, the most common production failure is not a leak: it is the context never being set and

everything coming back empty. Detect it explicitly rather than letting it surface as a support ticket. A context function that raises when the tenant is missing turns a silent failure into a 500 with a stack trace, which is infinitely easier to fix:

```
CREATE OR REPLACE FUNCTION app.current_tenant()
RETURNS uuid
LANGUAGE plpgsql
STABLE
SET search_path = ''
AS $$
DECLARE
  v text := current_setting('app.tenant_id', true);
BEGIN
  IF v IS NULL OR v = '' THEN
    RAISE EXCEPTION 'tenant context not set'
      USING ERRCODE = 'insufficient_privilege',
            HINT = 'Missing set_config(''app.tenant_id'', ..., true) in the transaction';
  END IF;
  RETURN v::uuid;
END;
$$;
```

There is a design decision here worth making consciously. Raising is right for an API where every request has a tenant. If you have legitimate tenantless routes — a health endpoint, a public pricing page — those routes must not touch RLS tables, or you will need a variant that returns `NULL` and policies that treat it as "no rows". What you must not do is return `NULL` and write the policy so that `NULL` matches everything. That is the bug that turns an oversight into a breach.

Column-level security, alongside RLS

RLS decides which rows you see. It does not decide which columns. They are orthogonal mechanisms and you often need both: the support team can see every tenant's rows but not card numbers; an analytics role sees every row but not names.

```
-- Column privileges: support cannot read the card hash
GRANT SELECT (id, tenant_id, amount, status, created_at)
  ON invoices TO support_role;

-- Reading the forbidden column fails explicitly
SET ROLE support_role;
SELECT card_hash FROM invoices LIMIT 1;
-- ERROR: permission denied for column card_hash
```

For cases where you would rather mask than deny, a `security_barrier` view does the job, as long as you remember that without that option a view is not a security boundary:

```
CREATE VIEW invoices_support WITH (security_barrier = true) AS
SELECT id,
       tenant_id,
       amount,
       status,
       'XXXX-' || right(card_last4, 4) AS card_display,
```

```
       created_at
FROM invoices;

GRANT SELECT ON invoices_support TO support_role;
REVOKE ALL ON invoices FROM support_role;
```

From PostgreSQL 15 onwards there is also `CREATE VIEW ... WITH (security_invoker = true)`, which makes the view run with the permissions and policies of whoever queries it rather than the owner's. For views over RLS tables this is usually what you want: without `security_invoker`, a view owned by a privileged role bypasses the base tables' policies and becomes the back door we discussed earlier.

RLS versus schema-per-tenant and database-per-tenant

RLS is not the only way to isolate tenants, and choosing wrong is expensive because migrating between models is a project, not a refactor. An honest summary of the three approaches:

Shared rows with RLS. One database, one table per entity, `tenant_id` on every row. Scales to tens of thousands of tenants without breaking a sweat. Migrations are a single operation. Operational cost per tenant tends towards zero. In exchange, isolation is logical: one mistake in a policy is an incident for everybody, and workloads compete for the same resources, so a heavy tenant degrades the rest. It is the default model for SaaS with many small or mid-sized customers.

One schema per tenant. Same database, a different schema per customer. Isolation is stronger and easier to reason about: if `search_path` points at the wrong schema, it usually fails rather than returning somebody else's data. It allows per-customer schema customisation, which in enterprise deals sometimes pays well. The problem is operational: every migration multiplies by the number of tenants, and at a few hundred schemas PostgreSQL's catalogue starts showing up in planning time and connection overhead. Works well into the hundreds.

One database or instance per tenant. Maximum isolation, including resource isolation and blast radius. This is what you want if a customer demands data residency in their region or their own encryption key. It is also the most expensive: *N* backups, *N* maintenance windows, *N* migration deployments, and a connection problem you will solve with more infrastructure. Reasonable with dozens of large customers; unworkable with thousands.

And a practical note: these models combine. The most common pattern in mature SaaS is RLS for the long tail of customers plus a dedicated instance for the three or four who pay enough to demand one. If you design the data access layer so the tenant is resolved in exactly one place — the middleware that opens the transaction — moving a customer from one model to another is a configuration change rather than a rewrite.

Adopting RLS in an application already in production

Almost nobody starts with RLS. The normal path is arriving at it with two hundred queries that already carry `WHERE tenant_id = ?` and a reasonable fear that switching policies on will break something on a Friday. It can be done with no downtime and no drama, in phases, with the property that every phase is reversible.

Phase 1: prepare without enabling

Create the application role separate from the table owner, the context function, and all the policies. Do not enable RLS yet. At this point the policies exist in the catalogue and do absolutely nothing, so the risk is zero. Use the time to fix the boring things: tables without `tenant_id`, columns that allow it to be `NULL`, indexes that do not have it first.

```
-- A NOT NULL without locking the table (PostgreSQL 12+)
ALTER TABLE invoices ADD CONSTRAINT invoices_tenant_not_null
  CHECK (tenant_id IS NOT NULL) NOT VALID;
ALTER TABLE invoices VALIDATE CONSTRAINT invoices_tenant_not_null;
```

`NOT VALID` followed by `VALIDATE` is the pattern for avoiding a long `ACCESS EXCLUSIVE` lock: the first statement is instant and applies only to new rows, the second scans the table under a much gentler lock. On a hundred-million-row table that difference is the difference between a routine deploy and an outage.

Phase 2: shadow mode

Before enabling anything, verify that the policy *would have* returned the same rows your current `WHERE` clauses do. You do this on a replica or a copy, comparing per-tenant counts with and without the policy:

```
-- On a test environment with real data
ALTER TABLE invoices ENABLE ROW LEVEL SECURITY;
ALTER TABLE invoices FORCE ROW LEVEL SECURITY;

DO $$
DECLARE
  t record;
  via_policy bigint;
  via_where  bigint;
  mismatches int := 0;
BEGIN
  FOR t IN SELECT id FROM tenants LOOP
    PERFORM set_config('app.tenant_id', t.id::text, true);
    SET LOCAL ROLE app_user;
    SELECT count(*) INTO via_policy FROM invoices;
    RESET ROLE;
    SELECT count(*) INTO via_where FROM invoices WHERE tenant_id = t.id;

    IF via_policy <> via_where THEN
      RAISE WARNING 'tenant %: policy=% where=%', t.id, via_policy, via_where;
      mismatches := mismatches + 1;
    END IF;
  END LOOP;
  RAISE NOTICE 'tenants with a mismatch: %', mismatches;
END;
$$;
```

Any mismatch is extremely valuable information to have before touching production. The typical causes: orphan rows with a `tenant_id` that no longer exists, join tables where the tenant is derived through a join instead of being denormalised, and historical records imported by an old migration with no tenant at all. All of them are fixed with data, not with policies.

Phase 3: enable, with the WHERE clauses still in place

Now for real: `ENABLE` and `FORCE` in production, table by table, starting with the least critical one. Leave the `WHERE tenant_id = ?` clauses exactly where they are in the code. They are redundant with the policy and that redundancy is precisely what you want during the transition: if you got a policy wrong and it is too permissive, the `WHERE` saves you; if you got it wrong and it is too strict, you will see zero rows and catch it within minutes. It also helps the planner and avoids performance surprises in the same deploy where you change the semantics.

Phase 4: remove the WHERE clauses, or don't

After a few uneventful weeks you can start removing the manual filters. My opinion, having lived both versions: drop them from new queries and from ones you are touching anyway, but do not run a mass cleanup campaign. The benefit of deleting a `WHERE` that already works is aesthetic; the risk of a three-hundred-file pull request nobody really reviews is real. What is worth eliminating without exception is any dynamic construction of that filter, because that is where injection lives.

There is one exception to that moderation: if your goal is that a new developer cannot write an unsafe query, at some point the filter has to go, because as long as it is there people keep believing isolation is their responsibility. That is a team-culture decision as much as a technical one.

Concrete integrations

Supabase and PostgREST

In Supabase the tenant does not travel in a session variable you set; it arrives in the JWT claims that PostgREST places in `request.jwt.claims`. The pattern changes, but the rules do not:

```
-- Read a token claim safely
CREATE OR REPLACE FUNCTION app.tenant_from_token()
RETURNS uuid
LANGUAGE sql
STABLE
SET search_path = ''
AS $$
  SELECT NULLIF(
    current_setting('request.jwt.claims', true)::jsonb ->> 'tenant_id',
    ''
  )::uuid;
$$;

CREATE POLICY tenant_isolation ON public.invoices
  AS PERMISSIVE FOR ALL
  TO authenticated                          -- never anon, never public
  USING      (tenant_id = (SELECT app.tenant_from_token()))
  WITH CHECK (tenant_id = (SELECT app.tenant_from_token()));
```

Three details make the difference here. `TO authenticated` stops the policy from being evaluated for the anonymous role, which also saves work. The `(SELECT ...)` around the function is the same InitPlan trick you already know, and in Supabase it is the official recommendation precisely because the impact on large tables is enormous: without it, the function is evaluated once per row. And the second argument `true` in `current_setting` makes it return `NULL` instead of raising when the claim is absent, which is what you want so the policy simply does not match.

One warning specific to this environment: the `service_role` key bypasses RLS completely. If it ends up in the client, in a publicly prefixed environment variable, or in a repository, every policy you wrote ceases to exist. Treat it as a superuser credential, because for practical purposes it is one.

Django

Django has no native RLS support, but a middleware that opens the transaction and pins the context covers the whole case:

```
# middleware.py
from django.db import connection, transaction

class TenantRLSMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        tenant_id = resolve_tenant(request)   # from subdomain, JWT, session...
        if tenant_id is None:
            return self.get_response(request)  # public routes: don't touch RLS

        with transaction.atomic():
            with connection.cursor() as cur:
                cur.execute(
                    "SELECT set_config('app.tenant_id', %s, true)",
                    [str(tenant_id)],
                )
            return self.get_response(request)
```

Two things matter here. First: `ATOMIC_REQUESTS = True` in the database settings does the same thing more cleanly by wrapping every request in a transaction; if you enable it, the middleware only needs the `set_config`. Second: management commands and Celery tasks do not go through middleware. Each one needs to set its own context, and that is exactly where people forget and end up granting `BYPASSRLS` "temporarily" to the entire application.

```
# For Celery, an explicit decorator
from contextlib import contextmanager
from django.db import connection, transaction

@contextmanager
def tenant_context(tenant_id):
    with transaction.atomic():
        with connection.cursor() as cur:
            cur.execute("SELECT set_config('app.tenant_id', %s, true)",
                        [str(tenant_id)])
        yield

@shared_task
def generate_report(tenant_id, month):
    with tenant_context(tenant_id):
        ...   # everything inside sees only that tenant's data
```

Prisma

Prisma exposes no "before every query" hook on the connection, so the pattern is an interactive transaction wrapped in a helper:

```
import { PrismaClient, Prisma } from '@prisma/client'

const prisma = new PrismaClient()

export async function withTenant<T>(
  tenantId: string,
  fn: (tx: Prisma.TransactionClient) => Promise<T>,
): Promise<T> {
  return prisma.$transaction(async (tx) => {
    // Parameterised: never interpolate tenantId into the string
    await tx.$executeRaw`SELECT set_config('app.tenant_id', ${tenantId}, true)`
    return fn(tx)
  })
}

// Usage
const invoices = await withTenant(req.tenantId, (tx) =>
  tx.invoice.findMany({ where: { status: 'pending' } }),
)
```

The critical point is using `tx` and not `prisma` inside the callback. If out of habit you write `prisma.invoice.findMany(...)`, that query goes out on a different pool connection with no context, and you will get zero rows or an error from the context function. It is an easy mistake to make and an easy one to catch if you have the function that raises when the tenant is missing. An ESLint rule forbidding the global client inside a `withTenant` block closes the matter entirely.

Eight mistakes that undo all the work

1. **The application connects as the table owner** (or with `BYPASSRLS`). The policies exist, they show in `pg_policies`, and they do nothing. Separate role plus `FORCE ROW LEVEL SECURITY`.

2. `SET` **instead of** `SET LOCAL`**.** With transaction-mode pooling the context outlives the request and the next client inherits it. It's an intermittent, load-dependent isolation failure that is practically impossible to reproduce locally.

3. **Interpolating the tenant identifier into the** `SET` **statement.** `SET` commands don't accept parameters, so people concatenate. Use `set_config()` with a placeholder.

4. **Indexes without** `tenant_id` **as the leading column.** The policy degrades from index condition to filter, and the cost of every query scales with the total number of rows in the instance rather than with the tenant's own.

5. **Calling the context function without** `(SELECT ...)`**.** One evaluation per row instead of one per query, and no index usage.

6. **Views without** `security_invoker`**.** A view created by the table owner hands every tenant's rows to anyone holding `SELECT` on it.

7. **Unique indexes without `tenant_id`.** Functionally wrong (two customers can't both have invoice number 1) and they leak the existence of other tenants' rows through duplicate-key errors.

8. **New tables without RLS.** The most likely failure six months in. You prevent it with the CI catalog test, not with discipline.

Production checklist

- The application role owns no tables and has no `SUPERUSER`, `BYPASSRLS` or `CREATEROLE`.
- Every table with a `tenant_id` has both `ENABLE` *and* `FORCE ROW LEVEL SECURITY`, verified by a CI test.
- The isolation policy is `RESTRICTIVE` and defines `USING` and `WITH CHECK` separately.
- All context expressions are wrapped in `(SELECT ...)`.
- The context is set with `set_config(..., true)` inside an explicit transaction, with the value passed as a bound parameter.
- Every index on multi-tenant tables starts with `tenant_id`, unique indexes included.
- Foreign keys between tenant tables are composite and include `tenant_id`.
- No view exposed to the application role is missing `security_invoker = true`.
- The list of roles with `BYPASSRLS` is inventoried, short, and its credentials are rotated and audited.
- You have a saved `EXPLAIN` for your three hottest queries showing `Index Cond` on `tenant_id`.
- Backup, ETL and logical replication processes run under known, documented roles.

FAQ

Does RLS replace tenant filtering in the application? No, and you shouldn't want it to. Keep filtering in the application: the planner does better work, queries read better, and you don't depend on a single layer. RLS is the safety net that catches the case you missed, not the primary mechanism.

What does it cost in performance? Done well — policy wrapped in `(SELECT ...)` and indexes led by `tenant_id` — the overhead is on the order of one extra equality comparison per query, indistinguishable from noise. Done badly it can turn an `Index Scan` into a `Seq Scan` with a function call per row. The range between those two outcomes spans orders of magnitude, and `EXPLAIN` decides which one you got, not intuition.

What if I need a "support mode" to look at a customer's data? A separate role with its own permissive policy, its own session parameter, and an audit record for every use. Never `BYPASSRLS` on the application path, because that credential then becomes the master key to your entire database.

Does it work with Prisma, Drizzle or Django? Yes, because it's transparent to the ORM: RLS acts server-side. All you need from the client is one explicit transaction per request in which to set the context. In Prisma, an interactive `$transaction` wrapped in an extension; in Django, middleware that opens the transaction and runs the `set_config`.

Wouldn't database-per-tenant or schema-per-tenant be simpler? It isolates better and needs none of this, but operational cost grows linearly: every migration is multiplied by the tenant count, and PostgreSQL starts to struggle with thousands of schemas. Shared schema with RLS is the balance point for most SaaS products; a dedicated database makes sense for large customers with regulatory requirements, usually as a separate plan.

Can I apply RLS to an existing table without downtime? Yes, and the order matters. First deploy the code that sets the context on every transaction, then create the policies, and only then run `ENABLE` and `FORCE`. Invert that order and any in-flight request that doesn't set the context starts seeing an empty database. The `ALTER TABLE ... ENABLE ROW LEVEL SECURITY` statements take a very brief exclusive lock, so run them with `lock_timeout` set.

Takeaway

RLS isn't an exotic feature or a substitute for writing good code. It's a change in your failure model: you move from a system where an omission leaks data silently to one where an omission breaks functionality loudly. That change of direction is why the configuration work is worth doing properly.

If you're going to implement it, the least painful order is: separate role and `FORCE`, restrictive policy with `(SELECT ...)`, transactional context via `set_config`, indexes led by `tenant_id`, and the CI tests from day one. Everything else is detail you can close out later; those five points are what decide whether the isolation is real.