PuigFlows

El Problema N+1 en ORMs: Detectarlo y Eliminarlo en SQLAlchemy y Prisma

The N+1 Query Problem in ORMs: How to Detect and Eliminate It in SQLAlchemy and Prisma

Guía técnica · Nivel intermedio · Edición del 15 de agosto de 2026
Español (56 min) · English (53 min)

puigflows.com

Versión en español

Qué es el problema N+1 (y por qué casi seguro lo tienes)

El problema N+1 es el fallo de rendimiento más común en aplicaciones que usan un ORM, y también el más traicionero: el código que lo provoca parece perfectamente razonable. Ocurre cuando cargas una lista de N objetos con una consulta y después, al recorrerlos, el ORM lanza una consulta adicional por cada uno para traer sus relaciones. Una consulta se convierte en 1 + N sin que nada en el código lo delate.

```
# Parece una sola consulta... pero no lo es
posts = session.scalars(select(Post).limit(50)).all()  # 1 consulta

for post in posts:
    print(post.author.name)   # 1 consulta extra POR CADA post

# Total: 1 + 50 = 51 consultas contra la base de datos
```

El culpable es el **lazy loading**, la estrategia por defecto de la mayoría de ORMs: las relaciones no se cargan hasta que las tocas. Es una decisión sensata para no traer datos que quizá no uses, pero dentro de un bucle se convierte en una fábrica de consultas.

Por qué importa: las matemáticas del desastre

Supongamos que cada consulta tarda 2 ms de ida y vuelta a la base de datos. Con 50 posts, esos 51 viajes suman unos 100 ms solo en latencia de red, antes de contar el trabajo real de la base de datos. Con 500 filas ya son más de un segundo. Y lo peor: en desarrollo no lo notas, porque con 10 filas de prueba todo va rápido. El N+1 es un problema que *escala con tus datos*, así que explota justo cuando tu aplicación empieza a tener éxito.

Hay un segundo coste menos obvio: cada consulta ocupa una conexión del pool durante su ida y vuelta. Un endpoint con N+1 bajo tráfico multiplica la presión sobre el pool y puede provocar timeouts en endpoints que no tienen nada que ver.

Cómo detectarlo antes que tus usuarios

La señal inequívoca: muchas consultas idénticas seguidas donde solo cambia un parámetro. En SQLAlchemy basta activar el log de SQL en desarrollo:

```
engine = create_engine(DATABASE_URL, echo=True)  # solo en desarrollo
```

Para algo más sistemático, cuenta las consultas con un event listener. Este context manager falla ruidosamente si un bloque de código supera el límite esperado:

```
from contextlib import contextmanager
```

```
from sqlalchemy import event

@contextmanager
def max_queries(engine, limit):
    executed = []

    def counter(conn, cursor, statement, params, context, executemany):
        executed.append(statement)

    event.listen(engine, "before_cursor_execute", counter)
    try:
        yield executed
    finally:
        event.remove(engine, "before_cursor_execute", counter)
        count = len(executed)
        assert count <= limit, (
            f"Se ejecutaron {count} consultas (límite: {limit}). "
            "Probable N+1: revisa las estrategias de carga."
        )
```

En Prisma, el equivalente es escuchar los eventos de query del cliente:

```
const prisma = new PrismaClient({
  log: [{ emit: 'event', level: 'query' }],
});

prisma.$on('query', (e) => {
  console.log(e.query, e.duration + 'ms');
});
```

En producción, las trazas de OpenTelemetry lo hacen evidente: un span de petición con decenas de spans hijos de base de datos idénticos en cascada es la firma visual del N+1.

SQLAlchemy 2.0: elige la estrategia de carga correcta

SQLAlchemy no te obliga a elegir entre lazy loading y traerlo todo: tiene varias estrategias de carga ansiosa (eager loading) que se declaran por consulta con options(). Las dos que resuelven el 95% de los casos son selectinload y joinedload.

selectinload: la opción por defecto para colecciones

selectinload ejecuta una segunda consulta con un IN sobre las claves de los padres. Dos consultas en total, sin importar cuántas filas haya:

```
from sqlalchemy import select
from sqlalchemy.orm import selectinload

stmt = (
    select(Post)
    .options(selectinload(Post.comments))
    .limit(50)
)
```

```
posts = session.scalars(stmt).all()

# Consulta 1: SELECT * FROM posts LIMIT 50
# Consulta 2: SELECT * FROM comments WHERE post_id IN (id1, id2, ..., id50)
```

Como cada consulta trae filas "planas" sin duplicar datos del padre, escala bien con colecciones grandes. Es la elección correcta para relaciones uno-a-muchos y muchos-a-muchos.

joinedload: para relaciones a-uno

joinedload resuelve todo en una sola consulta con un JOIN. Es ideal cuando la relación apunta a un solo objeto (muchos-a-uno o uno-a-uno), porque el JOIN no multiplica filas:

```
from sqlalchemy.orm import joinedload

stmt = (
    select(Post)
    .options(joinedload(Post.author))  # muchos-a-uno: JOIN seguro
    .limit(50)
)
```

La regla práctica: **relación a-uno → joinedload; relación a-muchos → selectinload**. Usar joinedload sobre colecciones grandes provoca "row bloat": el JOIN repite los datos del padre en cada fila hija, y si encadenas dos colecciones obtienes un producto cartesiano que puede devolver miles de filas para 50 posts.

Relaciones anidadas

Las estrategias se encadenan para cargar grafos completos de forma predecible:

```
stmt = (
    select(Post)
    .options(
        joinedload(Post.author),
        selectinload(Post.comments).joinedload(Comment.author),
    )
    .limit(50)
)
# 2 consultas en total: posts+autores (JOIN) y comments+autores (IN + JOIN)
```

raiseload: el guardarraíl que convierte el N+1 en un error

La mejor defensa es hacer que el lazy loading accidental sea imposible. Con raiseload("*"), cualquier acceso a una relación no cargada explícitamente lanza una excepción en lugar de una consulta silenciosa:

```
from sqlalchemy.orm import joinedload, raiseload

stmt = (
    select(Post)
    .options(
        joinedload(Post.author),
        raiseload("*"),  # todo lo demás: error, no consulta
```

```
    )
)
# post.comments -> InvalidRequestError: 'Post.comments' is not available due to lazy='raise'
```

Si usas SQLAlchemy asíncrono ya tienes este comportamiento gratis: el lazy loading implícito falla con MissingGreenlet, lo que convierte cada N+1 potencial en un error visible en desarrollo. Es molesto a propósito, y es bueno que lo sea.

Prisma: include y relationLoadStrategy

En Prisma el N+1 clásico aparece al consultar relaciones dentro de un bucle:

```
// MAL: 1 + N consultas
const posts = await prisma.post.findMany({ take: 50 });
for (const post of posts) {
  const author = await prisma.user.findUnique({
    where: { id: post.authorId },
  });
}
```

La solución idéntica en espíritu al eager loading: pedir las relaciones en la misma consulta con include (o select anidado):

```
// BIEN: número constante de consultas
const posts = await prisma.post.findMany({
  take: 50,
  include: {
    author: true,
    comments: { include: { author: true } },
  },
});
```

Además, Prisma te deja elegir *cómo* se cargan esas relaciones con relationLoadStrategy (disponible en PostgreSQL y MySQL tras activar el preview feature relationJoins en el generator del schema):

```
const posts = await prisma.post.findMany({
  relationLoadStrategy: 'join',   // 1 consulta con LATERAL JOIN + agregación JSON
  // relationLoadStrategy: 'query' // varias consultas + merge en la aplicación
  take: 50,
  include: { author: true },
});
```

La estrategia join hace el trabajo en la base de datos (en PostgreSQL usa LATERAL JOIN y agregación JSON); query ejecuta una consulta por tabla y une los resultados en la aplicación, como hace selectinload. La misma intuición aplica: join para relaciones a-uno o resultados pequeños, query cuando las colecciones son grandes y el JOIN duplicaría demasiados datos.

Un detalle que sorprende: dentro de un mismo tick, Prisma agrupa automáticamente los findUnique idénticos con un dataloader interno, lo que mitiga el N+1 en resolvers de GraphQL. No dependas de ello para código

secuencial con await dentro de bucles: ahí el batching no puede ayudarte.

Los errores que reintroducen el N+1

- **La serialización automática.** Cargas los posts sin relaciones y se los pasas a Pydantic, a un serializer de DRF o a JSON.stringify con un getter. El serializador recorre las relaciones y dispara el N+1 fuera de tu vista. La consulta debe cargar todo lo que el schema de respuesta necesita.
- **El acceso "solo para contar".** len(post.comments) carga la colección entera para tirar los objetos. Usa una subconsulta de conteo o una columna anotada.
- **Encadenar joinedload sobre dos colecciones.** No es N+1, pero el producto cartesiano resultante puede ser peor. Colecciones: siempre selectinload.
- **Confiar en la memoria.** El compañero que añade un campo al serializer seis meses después no sabe que la consulta no carga esa relación. Sin guardarraíles (raiseload, tests que cuentan consultas), el N+1 siempre vuelve.

Blindaje: tests que cuentan consultas

El N+1 es un bug de regresión clásico: lo arreglas hoy y alguien lo reintroduce en tres meses. La defensa definitiva es un test que falla si el número de consultas crece con el número de filas:

```
def test_list_posts_query_count(session, engine, make_posts):
    make_posts(50)  # fixture que crea 50 posts con autor y comentarios

    with max_queries(engine, limit=3):
        result = list_posts_with_comments(session)

    assert len(result) == 50
```

La clave del test no es el número exacto, sino que el límite sea *constante*: si con 50 filas pasa y con 500 también, el endpoint es O(1) en consultas. En el ecosistema Django, assertNumQueries hace exactamente esto; el context manager de arriba te da lo mismo en SQLAlchemy.

Checklist final

- Activa el log de SQL en desarrollo y desconfía de consultas repetidas que solo cambian un parámetro.
- Relación a-uno → joinedload / join. Relación a-muchos → selectinload / query.
- Nunca encadenes joins sobre dos colecciones en la misma consulta.
- Usa raiseload("*") (o el modo asíncrono) para convertir el lazy loading accidental en un error de desarrollo.
- La consulta debe cargar exactamente lo que el serializador necesita: ni más, ni menos.
- Protege tus endpoints críticos con tests que cuentan consultas y fallan si el número deja de ser constante.

El N+1 no se elimina una vez: se mantiene a raya con hábitos. Carga explícita, guardarraíles que convierten el descuido en error y tests que vigilan el número de consultas. Con esas tres piezas, tu ORM deja de ser una caja de sorpresas y vuelve a ser lo que prometía: una herramienta que te ahorra trabajo sin costarte el rendimiento.

Edición ampliada: del parche puntual a un sistema que no recae

Todo lo anterior arregla el N+1 que ya encontraste. Esta ampliación va de lo demás: detectarlo en producción cuando nadie lo está buscando, los rincones donde reaparece con otra cara (async, colecciones grandes, agregados, GraphQL, la capa de serialización) y cómo montar guardarraíles que lo conviertan en un error de CI en lugar de un incidente de sábado por la noche.

Detección en producción: pg_stat_statements y el patrón de la consulta repetida

En desarrollo ves las consultas pasar por el log. En producción nadie mira el log, así que el N+1 se detecta por su huella estadística: una consulta corta, normalizada, con un número de ejecuciones desproporcionado respecto al tráfico. La extensión pg_stat_statements agrupa las consultas por su forma (los parámetros se sustituyen por marcadores), y esa agrupación es exactamente lo que necesitas: mil ejecuciones de la misma consulta con distinto id colapsan en una sola fila con calls=1000.

```
-- Sospechosos de N+1: muchas llamadas, poco tiempo por llamada
SELECT
  calls,
  round(mean_exec_time::numeric, 2) AS media_ms,
  round((calls * mean_exec_time)::numeric, 0) AS total_ms,
  left(query, 90) AS consulta
FROM pg_stat_statements
WHERE calls > 1000
  AND mean_exec_time < 5      -- cada una es "rapida"
ORDER BY calls DESC
LIMIT 20;
```

La trampa psicológica del N+1 es que ninguna consulta individual es lenta: cada una tarda dos milisegundos y por eso jamás aparece en el log de consultas lentas. El daño está en el producto calls × media, no en la media. Ordena por esa columna y las víctimas del lazy loading suben a la superficie. Si usas un APM, busca la misma señal en las trazas: decenas de spans de base de datos idénticos y consecutivos colgando de una misma petición HTTP — Sentry, por ejemplo, detecta este patrón automáticamente y lo agrupa como "N+1 Query" en su sección de problemas de rendimiento; Datadog y New Relic muestran la cascada de spans repetidos en la vista de traza. Una traza sana de un endpoint típico tiene entre una y cinco consultas; cuando veas cuarenta spans de tres milisegundos con la misma forma, ya sabes qué relación olvidó su selectinload.

Contar consultas desde dentro: instrumentación con eventos

El APM te avisa tarde: el código ya está desplegado. La instrumentación propia te avisa en el momento exacto en que un cambio dispara el número de consultas, y en SQLAlchemy cuesta quince líneas gracias al sistema de eventos del engine:

```
from contextlib import contextmanager
from sqlalchemy import event

@contextmanager
def contador_de_consultas(engine):
    """Cuenta las consultas SQL ejecutadas dentro del bloque."""
    consultas = []
```

```
    def registrar(conn, cursor, statement, params, context, executemany):
        consultas.append(statement)

    event.listen(engine, "before_cursor_execute", registrar)
    try:
        yield consultas
    finally:
        event.remove(engine, "before_cursor_execute", registrar)

# Uso en un test o en un endpoint sospechoso:
with contador_de_consultas(engine) as consultas:
    pedidos = obtener_pedidos_con_clientes(session)

print(f"{len(consultas)} consultas")  # esperabas 2, salen 51: N+1
```

En una aplicación FastAPI puedes ir un paso más lejos y colgar el contador de un middleware: cada respuesta sale con una cabecera X-DB-Queries y, si el número supera un umbral (digamos quince), se emite un warning estructurado con la ruta. Eso convierte el N+1 en algo que aparece en tus dashboards el mismo día en que se introduce, con la ruta exacta y sin esperar a que un usuario note la lentitud. El equivalente en Prisma es el evento de log de consultas — lo veremos en su sección — y la idea es idéntica: el número de consultas por petición es una métrica de primera clase, no una curiosidad de debugging.

Async SQLAlchemy: donde el lazy loading directamente no existe

Si usas SQLAlchemy con asyncio (FastAPI moderno, por ejemplo), el problema cambia de naturaleza: el lazy loading no es que sea lento — es que está prohibido. Acceder a una relación no cargada lanza MissingGreenlet, porque la carga perezosa necesitaría hacer IO dentro de un acceso a atributo, y eso no puede ser un punto de await. El error desconcierta la primera vez ("estaba leyendo un atributo, ¿por qué me habla de greenlets?"), pero es el N+1 manifestándose como excepción en lugar de como lentitud silenciosa.

```
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload
from sqlalchemy import select

async def pedidos_con_lineas(session: AsyncSession):
    resultado = await session.execute(
        select(Pedido)
        .options(
            selectinload(Pedido.lineas).selectinload(Linea.producto),
            selectinload(Pedido.cliente),
        )
        .where(Pedido.estado == "pendiente")
    )
    return resultado.scalars().all()
    # pedido.lineas[0].producto.nombre funciona: todo vino cargado
    # cualquier relacion NO listada arriba -> MissingGreenlet
```

La lectura correcta: async te obliga a hacer lo que en sync deberías hacer por disciplina. Toda relación que la respuesta vaya a tocar se declara en la consulta, y punto. De ahí sale una receta de migración que funciona sorprendentemente bien: antes de pasar un código sync a async, añade lazy="raise" a tus relaciones (o

raiseload("*") a tus consultas) y arregla cada punto que explote. Cuando el código sync ya no hace ninguna carga perezosa, la migración a async es casi mecánica — todos los MissingGreenlet potenciales ya fueron eliminados con mensajes de error mucho más claros.

Colecciones grandes: selectinload no sabe de límites

Las estrategias de carga ansiosa tienen un punto ciego del que casi nadie habla hasta que duele: cargan la colección completa. "Los últimos tres pedidos de cada cliente" no se puede expresar con selectinload — no existe un límite por relación — así que el eager loading ingenuo trae los diez mil pedidos del cliente veterano para quedarse con tres. Has eliminado el N+1 y a cambio has creado un problema de memoria.

La solución idiomática en PostgreSQL es una función de ventana (o un LATERAL join) que numera las filas por grupo y corta, combinada con contains_eager para que SQLAlchemy pueble la relación desde tu join manual en lugar de emitir el suyo:

```
from sqlalchemy import select, func
from sqlalchemy.orm import contains_eager, aliased

# Subconsulta: numera los pedidos de cada cliente, del mas reciente al mas antiguo
fila = func.row_number().over(
    partition_by=Pedido.cliente_id,
    order_by=Pedido.creado_en.desc(),
).label("fila")

sub = select(Pedido, fila).subquery()
PedidoTop = aliased(Pedido, sub)

consulta = (
    select(Cliente)
    .join(PedidoTop, PedidoTop.cliente_id == Cliente.id)
    .where(sub.c.fila <= 3)
    .options(contains_eager(Cliente.pedidos.of_type(PedidoTop)))
)
clientes = session.execute(consulta).unique().scalars().all()
# cliente.pedidos contiene como mucho 3 elementos: los mas recientes
```

El detalle crucial es contains_eager: le dice al ORM "la relación ya viene en este join, no la cargues tú". Sin esa opción, SQLAlchemy haría el join para filtrar y luego, al acceder a cliente.pedidos, dispararía su propia consulta — el N+1 de vuelta por la puerta de atrás, con el agravante de que ahora devolvería los diez mil pedidos, no tres. En Prisma el equivalente es directo porque include acepta take y orderBy anidados (include: { pedidos: { take: 3, orderBy: { creadoEn: "desc" } } }), y con la estrategia join lo resuelve con un LATERAL en una sola consulta.

Columnas anchas: el N+1 de los bytes

Hay una variante del problema que no multiplica consultas sino bytes. Tu tabla de productos tiene una columna descripcion_html de doscientos kilobytes y una ficha_tecnica JSONB del mismo calibre; el listado solo necesita nombre y precio, pero el ORM trae las filas completas. Cincuenta productos por página por doscientos kilobytes son diez megabytes que viajan, se deserializan y se descartan — en cada petición. Es el mismo error de fondo (cargar por comodidad lo que no se va a usar), y tiene la misma medicina: carga

explícita.

```
from sqlalchemy.orm import load_only, defer

# Opcion 1: lista blanca de columnas para el listado
consulta = select(Producto).options(
    load_only(Producto.id, Producto.nombre, Producto.precio)
)

# Opcion 2: columnas pesadas diferidas por defecto, en el modelo
class Producto(Base):
    __tablename__ = "productos"
    # ...
    descripcion_html: Mapped[str] = mapped_column(deferred=True)
    ficha_tecnica: Mapped[dict] = mapped_column(JSONB, deferred=True)
```

Ojo con la segunda opción: una columna diferida que sí se usa se convierte en... una consulta perezosa por fila. Diferir descripcion_html y luego renderizarla en un bucle de cincuenta productos es fabricar un N+1 artesanal. La regla: defer para columnas que solo lee la vista de detalle, load_only explícito para los listados, y el test que cuenta consultas (ya llegamos) vigilando que nadie rompa el contrato. En Prisma, select y omit cumplen el mismo papel, con omit especialmente cómodo para excluir globalmente columnas pesadas desde la configuración del cliente.

El N+1 de los agregados: contar sin iterar

"Mostrar cada categoría con su número de productos" es el N+1 más común después del de las relaciones: un COUNT por categoría dentro del bucle de renderizado. Cincuenta categorías, cincuenta COUNTs. La solución vive en SQL desde hace cuarenta años — GROUP BY — y el trabajo es solo expresarla en tu ORM:

```
from sqlalchemy import select, func

# Una sola consulta: categoria + numero de productos
consulta = (
    select(Categoria, func.count(Producto.id).label("num_productos"))
    .outerjoin(Producto)
    .group_by(Categoria.id)
    .order_by(Categoria.nombre)
)
for categoria, num_productos in session.execute(consulta):
    print(categoria.nombre, num_productos)
```

Si el conteo se necesita en muchos sitios, SQLAlchemy permite adjuntarlo al modelo con column_property (una subconsulta correlacionada que se carga con la entidad), y Prisma lo resuelve con include: { _count: { select: { productos: true } } } — una sola consulta con el agregado incluido. La versión sofisticada del mismo pecado es el agregado sobre relación cargada: len(categoria.productos) parece inocente, pero si la relación no está cargada dispara la carga completa de la colección para tirar todo menos el número. Contar en Python lo que la base de datos puede contar en SQL es pagar dos veces: una en consultas, otra en memoria.

Prisma a fondo: relationLoadStrategy, la API fluida y el dataloader escondido

La sección básica mostró include; la decisión interesante en Prisma es cómo se materializa ese include. Con el feature flag relationJoins activado, relationLoadStrategy acepta dos valores: join (por defecto una vez activado el flag) genera una única consulta usando LATERAL joins y agregación JSON en PostgreSQL — la base de datos devuelve el árbol ya montado —, mientras que query emite una consulta por tabla y une los resultados en la aplicación, que es esencialmente lo que hace selectinload en SQLAlchemy.

```
// schema.prisma
generator client {
  provider        = "prisma-client-js"
  previewFeatures = ["relationJoins"]
}

// Eleccion por consulta:
const pedidos = await prisma.pedido.findMany({
  relationLoadStrategy: "query",   // varias consultas, une la app
  include: { cliente: true, lineas: { include: { producto: true } } },
});
```

¿Cuál elegir? join minimiza viajes y suele ganar con relaciones a-uno y árboles moderados; query rinde mejor cuando el fan-out es grande y la agregación JSON en el servidor de base de datos empieza a costar CPU y memoria — la misma disyuntiva joinedload/ selectinload con otros nombres. Como siempre: mide con tus datos, no con los del benchmark de otro.

Prisma esconde además una defensa que merece ser conocida: cuando usas la API fluida (prisma.pedido.findUnique(...).cliente()) para resolver relaciones, las llamadas que coinciden en el mismo tick del event loop se agrupan automáticamente en lotes — un dataloader interno. Por eso algunos N+1 "accidentales" en resolvers de GraphQL con Prisma no duelen tanto como deberían. Pero es una red de seguridad con condiciones (las llamadas deben ser compatibles y simultáneas), no una licencia para iterar. Y para vigilar todo esto, Prisma expone el evento de consulta:

```
const prisma = new PrismaClient({
  log: [{ emit: "event", level: "query" }],
});

let consultas = 0;
prisma.$on("query", (e) => {
  consultas += 1;
  if (e.duration > 200) {
    console.warn("consulta lenta:", e.duration, "ms");
  }
});
// Por peticion: resetea el contador en un middleware HTTP
// y alerta si un endpoint supera su presupuesto.
```

GraphQL: el generador industrial de N+1

GraphQL merece sección propia porque su modelo de ejecución fabrica el problema por diseño: cada campo se resuelve de forma independiente, así que una consulta que pide veinte pedidos con su cliente ejecuta el resolver de cliente veinte veces. Nadie escribió un bucle; el runtime lo escribió por ti. La respuesta canónica es el patrón DataLoader: acumular las claves pedidas durante un tick de ejecución, resolverlas todas en una

sola consulta por lotes, y cachear dentro de la petición para que el mismo cliente pedido dos veces se resuelva una.

```
# Python con Strawberry: un DataLoader por relacion
from strawberry.dataloader import DataLoader

async def cargar_clientes(ids: list[int]) -> list[Cliente]:
    resultado = await session.execute(
        select(Cliente).where(Cliente.id.in_(ids))
    )
    por_id = {c.id: c for c in resultado.scalars()}
    return [por_id[i] for i in ids]   # mismo orden que las claves

loader_clientes = DataLoader(load_fn=cargar_clientes)

# En el resolver del campo cliente:
async def resolver_cliente(pedido) -> Cliente:
    return await loader_clientes.load(pedido.cliente_id)
# 20 pedidos -> 1 sola consulta WHERE id IN (...las 20 claves...)
```

Dos contratos del patrón que no se pueden violar: la función de lote debe devolver los resultados en el mismo orden que las claves recibidas (incluyendo None para las que no existan), y el loader debe crearse por petición, nunca compartido entre usuarios — su caché interna no distingue permisos y serviría datos de una petición a otra. En el ecosistema TypeScript, la librería dataloader de referencia impone los mismos contratos; con Prisma, la API fluida ya te da parte del efecto, pero un DataLoader explícito sigue siendo la herramienta correcta cuando la resolución cruza servicios o necesita control fino del batching.

La capa de serialización también dispara consultas

Un N+1 particularmente traicionero no vive en tu handler sino después de él: en la serialización. Devuelves una lista de objetos ORM y tu framework de esquemas — Pydantic con from_attributes, un serializer al estilo DRF, la serialización automática de tu framework web — recorre los atributos declarados para construir la respuesta. Si el esquema declara campos que corresponden a relaciones no cargadas, el serializador los toca, y cada toque es una consulta perezosa. El handler parece limpio; el log muestra cincuenta consultas emitidas durante la construcción del JSON.

```
class PedidoOut(BaseModel):
    model_config = ConfigDict(from_attributes=True)
    id: int
    total: Decimal
    cliente: ClienteOut     # <- si Pedido.cliente no vino cargado,
                            #    ESTE campo dispara una consulta por pedido

@app.get("/pedidos", response_model=list[PedidoOut])
def listar(session: Session = Depends(get_session)):
    return session.execute(
        select(Pedido).options(
            selectinload(Pedido.cliente),  # carga lo que el esquema declara
            raiseload("*"),                # y lo no declarado, que explote
        )
    ).scalars().all()
```

La combinación selectinload + raiseload("*") es el contrato perfecto entre consulta y esquema: la consulta carga exactamente lo que el esquema serializa, y cualquier desajuste — alguien añade un campo al esquema sin tocar la consulta — se manifiesta como error ruidoso en desarrollo, no como degradación silenciosa en producción. Si prefieres desacoplar del todo, mapea a DTOs explícitos en el handler; más verboso, pero el serializador ya no puede tocar nada que no le hayas dado.

Batching manual con IN: cuando no hay ORM que te salve

A veces la carga no pasa por relaciones del ORM: lees ids de una cola, cruzas datos de dos bases distintas, o llamas a un servicio por cada elemento. El patrón de fondo es siempre el mismo y merece existir como utilidad en tu código: junta las claves, resuélvelas en lotes, indexa el resultado.

```
from itertools import islice

def por_lotes(iterable, tamano):
    it = iter(iterable)
    while lote := list(islice(it, tamano)):
        yield lote

def cargar_por_ids(session, modelo, ids, tamano_lote=1000):
    """Resuelve ids en lotes con IN; devuelve un dict id -> entidad."""
    resultado = {}
    for lote in por_lotes(set(ids), tamano_lote):
        filas = session.execute(
            select(modelo).where(modelo.id.in_(lote))
        ).scalars()
        resultado.update({f.id: f for f in filas})
    return resultado

productos = cargar_por_ids(session, Producto, ids_desde_la_cola)
for item in items:
    procesar(item, productos.get(item.producto_id))
```

Los lotes de mil no son manía: los drivers y las bases de datos limitan el número de parámetros por consulta (PostgreSQL admite decenas de miles, pero el rendimiento del parser degrada mucho antes), y un IN con doscientas mil claves es su propio incidente. Deduplica las claves antes de consultar, decide qué hacer con las que no aparecen (el .get con None explícito del ejemplo) y, si esto cruza servicios en lugar de tablas, el mismo patrón se llama "endpoint de batch" y vale exactamente igual.

Cuándo el N+1 es aceptable (y cuándo la respuesta es denormalizar)

La honestidad técnica obliga a decirlo: no todo N+1 merece ser cazado. Un script administrativo que recorre veinte filas una vez al día puede permitirse sus veinte consultas; la legibilidad vale más que los sesenta milisegundos. Los criterios que convierten el N+1 en problema real son N creciente con los datos (hoy 20, en un año 4.000), estar en la ruta caliente de una petición interactiva, o multiplicarse por anidación (N clientes × M pedidos). Sin ninguna de las tres condiciones, arreglar el N+1 es optimización prematura con coste de complejidad.

En el extremo opuesto están los casos donde ni el mejor eager loading basta: el listado que necesita el número de pedidos, la fecha del último y el total acumulado de cada uno de mil clientes visibles. Ahí la

respuesta ya no es cargar mejor sino no calcular en caliente: contadores denormalizados mantenidos por triggers o por la aplicación (con la disciplina de actualizarlos en la misma transacción), vistas materializadas refrescadas por cron para datos que toleran minutos de retraso, o directamente una tabla de resumen que un job reconstruye. Denormalizar es aceptar redundancia a cambio de lecturas O(1); es la herramienta correcta cuando la agregación es cara, frecuente y tolerante a una pizca de staleness — y una fuente de bugs sutiles cuando se usa sin esas tres condiciones.

Caso práctico: /pedidos, de 4,2 segundos a 90 milisegundos

Números reales de un endpoint real (anonimizado): un listado de cincuenta pedidos con cliente, líneas y producto de cada línea. La versión ingenua — bucle sobre pedidos, acceso a pedido.cliente, pedido.lineas y linea.producto — emitía 1 + 50 + 50 + 380 = 481 consultas: una por la lista, una por cada cliente, una por cada colección de líneas y una por cada producto de cada línea. A dos milisegundos y pico de ida y vuelta por consulta, el endpoint tardaba 4,2 segundos, con la base de datos al 4% de CPU: todo el tiempo era latencia de red multiplicada.

El arreglo, en tres pasos medibles. Primero, selectinload sobre las tres relaciones: 481 consultas se convirtieron en 4 (la lista, los clientes en un IN, las líneas en otro, los productos en otro) y la latencia cayó a 210 ms. Segundo, load_only para excluir la descripción HTML de los productos — que pesaba el 80% de los bytes transferidos y el listado no mostraba —: 130 ms. Tercero, raiseload("*") como guardarraíl y un test de presupuesto que fija "4 consultas" como contrato del endpoint: 90 ms estables, y la garantía de que el próximo refactor no deshace el trabajo en silencio. Ninguno de los tres pasos requirió tocar el esquema ni cachear nada: solo decirle al ORM exactamente qué cargar.

Presupuestos de consultas en CI: el contrato que no se degrada

El test que cuenta consultas de la sección anterior tiene una versión industrializada: presupuestos por endpoint versionados junto al código. La idea es que cada endpoint crítico declare cuántas consultas tiene permitido emitir, y que superarlo rompa el build con un mensaje que explica qué hacer:

```
PRESUPUESTOS = {
    "GET /pedidos": 4,
    "GET /pedidos/{id}": 3,
    "GET /clientes": 2,
}

@pytest.mark.parametrize("ruta,limite", PRESUPUESTOS.items())
def test_presupuesto_de_consultas(cliente_http, engine, ruta, limite):
    metodo, path = ruta.split(" ", 1)
    path = path.replace("{id}", "1")
    with contador_de_consultas(engine) as consultas:
        respuesta = cliente_http.request(metodo, path)
    assert respuesta.status_code == 200
    assert len(consultas) <= limite, (
        f"{ruta} emitio {len(consultas)} consultas (limite {limite}). "
        "Si el aumento es legitimo, sube el presupuesto en este test "
        "explicando por que en el commit."
    )
```

La fricción es deliberada: subir el presupuesto exige tocar el test y justificarlo, así que el N+1 introducido por accidente no puede colarse. En el mundo Prisma se consigue lo mismo contando eventos query en los tests de integración. Dos matices para que el sistema envejezca bien: fija presupuestos con un pequeño margen (el exacto se rompe con cambios inocentes, como una consulta extra de healthcheck de pool) y ejecuta estos tests contra una base con datos suficientes — con tres filas en la tabla, un N+1 emite cuatro consultas y pasa el presupuesto que rompería con cincuenta.

Preguntas frecuentes

¿joinedload o selectinload? La regla en una línea

A-uno (muchos pedidos → un cliente): joinedload, porque el join no multiplica filas. A-muchos (un pedido → muchas líneas): selectinload, porque evita el producto cartesiano y deduplicar en memoria. Anidado: mézclalos por nivel según la misma regla. Y si dudas, selectinload es el default seguro para todo, con un viaje extra de coste.

¿La caché no me ahorra arreglar el N+1?

Te lo esconde, que es peor. El día que la caché expira en frío, las 481 consultas siguen ahí, todas a la vez, con el tráfico del pico. Arregla primero la forma de las consultas y cachea después si aún hace falta: la caché es un multiplicador de rendimiento, no un sustituto de consultas sanas.

¿Cómo capturo el N+1 en revisión de código antes de que llegue a un test?

Busca el patrón, no las consultas: cualquier acceso a atributo-relación dentro de un bucle (for pedido in pedidos: pedido.cliente...) es sospechoso por inspección visual, sin ejecutar nada. Es un check barato de code review que atrapa la mayoría de los casos, y la razón por la que raiseload convierte los que se escapen en errores imposibles de ignorar.

¿Esto aplica igual con Django, Rails o Go?

El problema es idéntico y las herramientas se llaman distinto: select_related/ prefetch_related y assertNumQueries en Django; includes y la gema bullet en Rails; los ORMs de Go (GORM, Ent) con sus Preload/With. Los cuatro pilares — carga explícita, límite de bytes, guardarraíl de errores y test que cuenta consultas — se traducen uno a uno a cualquier ecosistema con ORM.

El N+1 de escritura: updates en bucle y el unit of work

Todo lo anterior habla de lecturas, pero el bucle que escribe tiene exactamente la misma patología: for producto in productos: producto.precio *= 1.1 seguido de un commit emite un UPDATE por fila. SQLAlchemy agrupa los flushes dentro del unit of work y los envía en lote cuando puede (executemany), lo que amortigua el golpe; aun así, para actualizaciones masivas cuyo nuevo valor puede expresarse en SQL, la diferencia entre "N updates de dos milisegundos" y "un update de veinte" sigue siendo un orden de magnitud, más el ahorro de no cargar las filas en memoria primero:

```
from sqlalchemy import update, insert

# En lugar de cargar + iterar + mutar + flush:
session.execute(
    update(Producto)
    .where(Producto.categoria_id == categoria_id)
    .values(precio=Producto.precio * 1.1)
)

# Y para inserciones masivas, una sola sentencia con lista de dicts:
session.execute(
    insert(Evento),
    [{"tipo": e.tipo, "payload": e.payload} for e in eventos],
)
```

En Prisma, updateMany y createMany cumplen el mismo papel. La señal para buscar esta variante: un job "que tarda horas" cuyo tiempo se va en miles de sentencias diminutas — el mismo perfil de pg_stat_statements que ya sabes leer, pero con UPDATE en lugar de SELECT. La contrapartida honesta: el update masivo en SQL se salta los eventos del ORM (validaciones, hooks, señales de auditoría que vivan en el modelo); si dependes de ellos, o los reimplementas en la sentencia (updated_at=func.now()) o aceptas el bucle con lotes.

Exports grandes: yield_per y el peligro del streaming con relaciones

El endpoint de exportación — "dame los cien mil pedidos del año en CSV" — combina mal dos necesidades: streaming para no cargar todo en memoria, y relaciones para que cada fila salga completa. yield_per resuelve la primera trayendo las filas en tandas; el matiz es cómo se lleva con el eager loading. La carga por joins (joinedload sobre colecciones) es incompatible con el streaming — el ORM no puede garantizar que una entidad esté completa hasta ver todas sus filas del join —, mientras que selectinload funciona: se ejecuta una vez por tanda, cargando las relaciones de esa tanda.

```
consulta = (
    select(Pedido)
    .options(selectinload(Pedido.cliente))
    .execution_options(yield_per=500)
)
for tanda in session.execute(consulta).partitions():
    for (pedido,) in tanda:
        escribir_fila_csv(pedido)   # memoria estable: ~500 pedidos vivos
# selectinload se ejecuta una vez por tanda de 500,
# no una vez por pedido: 200 tandas -> 400 consultas totales, no 100.001
```

Para exports verdaderamente masivos, la respuesta madura suele ser bajar un nivel: una consulta con join explícito que devuelva tuplas planas (sin objetos ORM, sin identity map) o directamente COPY de PostgreSQL hacia el proceso que escribe el fichero. El ORM brilla gestionando objetos con identidad y ciclo de vida; un export es una transformación de filas, y tratarlo como tal es más rápido y más simple.

La conexión retenida: el coste invisible del lazy loading

Hay un daño del N+1 que no aparece en ninguna métrica de latencia hasta que tumba el servicio: la retención

de conexiones. Mientras tu handler itera perezosamente — consulta, procesa, consulta, procesa — la conexión sigue ocupada (checked out) del pool durante todo el recorrido, incluyendo el tiempo de CPU entre consultas. Con un pool de veinte conexiones y handlers que retienen la suya doscientos milisegundos en vez de diez, tu techo de concurrencia real acaba de dividirse por veinte: las peticiones nuevas no esperan a la base de datos, esperan al pool.

Por eso el patrón sano es cargar todo al principio (pocas consultas densas, seguidas), soltar la conexión y procesar después con los datos ya en memoria. El eager loading no solo reduce el número de consultas: compacta el uso de la conexión en una ráfaga corta al inicio del handler, que es exactamente lo que un pool necesita para multiplexar bien. Si además hay transacción abierta, el efecto empeora: una transacción que abarca el bucle perezoso mantiene sus locks y su snapshot vivos durante todo el paseo, hinchando el trabajo de vacuum y el riesgo de contención. La regla compuesta: transacciones cortas, cargas densas al principio, y el procesamiento fuera de ambas.

Observabilidad continua: OpenTelemetry y la métrica que faltaba

La instrumentación automática de OpenTelemetry para SQLAlchemy y para Prisma crea un span por consulta, lo que convierte el N+1 en algo visible en cualquier backend de trazas: la traza del endpoint enfermo muestra una escalera de spans idénticos que ningún dashboard de latencia media habría revelado. Merece la pena añadir una métrica derivada que casi nadie exporta y que es la más útil de todas: consultas por petición, como histograma etiquetado por ruta.

```
from opentelemetry.instrumentation.sqlalchemy import SQLAlchemyInstrumentor
from opentelemetry import metrics

SQLAlchemyInstrumentor().instrument(engine=engine)

medidor = metrics.get_meter("app")
consultas_por_peticion = medidor.create_histogram(
    "db.queries_per_request",
    description="Consultas SQL emitidas por peticion HTTP",
)

# En el middleware, al terminar cada peticion:
consultas_por_peticion.record(
    len(consultas), {"http.route": ruta, "http.method": metodo}
)
```

Con esa métrica, el N+1 tiene alerta propia: el p95 de db.queries_per_request de una ruta que salta de 4 a 60 tras un deploy es una regresión inequívoca, detectable en minutos y atribuible al commit exacto — sin esperar a que la latencia se degrade lo suficiente como para que alguien abra un ticket. Es la versión en producción del presupuesto de CI: el mismo contrato, vigilado en el otro extremo del ciclo de vida.

Tabla de decisión: qué estrategia para qué relación

Como referencia rápida, la matriz que resume todo lo anterior. Relación a-uno en ruta caliente: joinedload (un join barato, cero viajes extra). Colección mediana (decenas a cientos): selectinload. Colección con límite por padre ("los últimos 3"): window function o LATERAL con contains_eager; en Prisma, include con take.

Colección enorme sin límite: replantea el endpoint — probablemente pide paginación anidada o un export. Columnas anchas en listados: load_only o select/omit. Agregados por padre: GROUP BY, column_property o _count, nunca len() en Python. Async: todo lo anterior más lazy="raise" en los modelos, porque el lazy loading ni siquiera es una opción. GraphQL: DataLoader por relación, creado por petición. Y en todos los casos, raiseload("*") o su equivalente como cierre: lo no declarado debe fallar, no funcionar despacio.

Checklist de producción ampliada

- **Detección:** pg_stat_statements activo y revisado (orden por calls × media), APM con detección de N+1 habilitada, y la métrica db.queries_per_request exportada con alerta sobre su p95 por ruta.
- **Prevención en el código:** raiseload("*") (o lazy="raise") como política por defecto en consultas de endpoints, carga explícita de cada relación que el esquema de salida declara, load_only en listados con columnas anchas.
- **Prevención en CI:** presupuesto de consultas por endpoint crítico, tests ejecutados contra datos con volumen realista (cincuenta filas mínimo, no tres), y presupuestos con margen para no romperse por ruido.
- **Escrituras y jobs:** updates masivos expresados en SQL (update/ updateMany), inserciones con executemany/createMany, exports con yield_per + selectinload o directamente COPY.
- **Arquitectura:** transacciones cortas con cargas densas al inicio, conexiones devueltas al pool antes del procesamiento pesado, DataLoader por petición en GraphQL, y denormalización solo donde la agregación caliente lo justifique.

El hilo conductor de toda esta guía cabe en una frase: el ORM ejecuta exactamente lo que le pides, y el N+1 aparece cuando pides sin darte cuenta. Hacer explícita cada carga — y hacer que lo implícito falle ruidosamente — es todo el secreto. Lo demás es medición para saber dónde empezar y tests para no volver atrás.

Django como espejo: los mismos patrones con otros nombres

Aunque esta guía se centra en SQLAlchemy y Prisma, ver los mismos conceptos en Django ayuda a fijarlos — y es probable que convivas con los tres. select_related es el joinedload de Django: un join para relaciones a-uno, resuelto en la misma consulta. prefetch_related es selectinload: una segunda consulta con IN para colecciones, unida en Python. El objeto Prefetch con un queryset personalizado cubre el caso "los últimos 3 por padre" y el de filtrar la colección cargada. only() y defer() son load_only y defer casi literalmente. Y assertNumQueries, integrado en el framework de tests desde siempre, es el presupuesto de consultas que construimos a mano más arriba:

```
# Django: mismos patrones, otra sintaxis
pedidos = (
    Pedido.objects
    .select_related("cliente")                # a-uno: join
    .prefetch_related(
        Prefetch("lineas",
                 queryset=Linea.objects.select_related("producto"))
    )
    .only("id", "total", "creado_en", "cliente__nombre")
)
```

```
class TestPedidos(TestCase):
    def test_presupuesto(self):
        with self.assertNumQueries(3):
            self.client.get("/pedidos/")
```

Lo que Django no trae de serie es el guardarraíl al estilo raiseload; el hueco lo cubren paquetes como django-zen-queries (bloquea consultas dentro de secciones marcadas) o la vigilancia de nplusone. En Rails, la gema bullet avisa en desarrollo de cada includes que falta y de cada uno que sobra — este último caso, el eager loading innecesario, es el pecado simétrico del que casi nadie habla: cargar relaciones que la vista no usa también cuesta, solo que en bytes en lugar de viajes.

Catálogo de anti-patrones: los disfraces del N+1

Para cerrar la parte técnica, la lista de disfraces con los que el N+1 pasa desapercibido en revisión de código. El helper inocente: un método formatear(pedido) que internamente toca pedido.cliente.nombre — el bucle está limpio, la consulta vive tres niveles más abajo en la pila. El property traicionero: un @property del modelo que cruza una relación, invocado desde una plantilla o un serializador donde nadie ve SQL. El if que solo a veces: una rama condicional que accede a la relación solo para pedidos premium — el test con datos normales pasa, producción con mezcla real dispara. El conteo disfrazado: if pedido.lineas: para saber si hay líneas, que carga la colección entera para evaluar un booleano (la forma barata es una subconsulta exists o un conteo agregado). El re-fetch por identidad: volver a consultar por id un objeto que ya tienes en la sesión, multiplicado por bucle. Y el más moderno: el agente o el job de IA que llama a una herramienta "obtener_detalle(id)" dentro de un bucle sobre cien ids, reproduciendo con llamadas HTTP el mismo patrón que este artículo describe con SQL — el batching como principio no entiende de capas.

Glosario

Lazy loading (carga perezosa): resolver una relación en el momento del primer acceso al atributo, con una consulta implícita. **Eager loading (carga ansiosa):** declarar en la consulta qué relaciones cargar, resolviéndolas por adelantado en una o pocas consultas. **selectinload / prefetch_related / estrategia query:** carga de colecciones con una consulta adicional por relación usando IN sobre las claves de los padres. **joinedload / select_related / estrategia join:** carga en la misma consulta mediante join; ideal para a-uno. **contains_eager:** poblar una relación desde un join que tú escribiste, en lugar de dejar que el ORM emita el suyo. **raiseload / lazy="raise":** convertir todo acceso perezoso en excepción; el guardarraíl central de esta guía. **Identity map:** caché por sesión que garantiza una única instancia por fila; deduplica objetos pero no evita consultas de colecciones. **DataLoader:** acumulador por petición que agrupa cargas individuales en lotes; la defensa canónica en GraphQL. **Fan-out:** número de hijos por padre en una relación; el multiplicador que convierte N+1 en N×M+1. **Presupuesto de consultas:** número máximo de consultas que un endpoint tiene permitido emitir, verificado por un test.

Caso de estudio 2: el dashboard multi-tenant que degradaba por cliente grande

Un segundo caso real, porque el primero era casi de libro y este tiene la forma retorcida con la que el N+1 suele llegar de verdad. Un dashboard B2B mostraba, para cada proyecto del tenant, sus miembros con avatar y su última actividad. Con los tenants pequeños (tres proyectos, diez miembros) volaba; el cliente más

grande (ciento veinte proyectos, dos mil miembros) tardaba once segundos y a veces agotaba el timeout. El perfil no mostraba ninguna consulta lenta: mostraba cuatro mil consultas rápidas.

La cascada tenía tres pisos: proyectos → miembros (lazy), miembro → usuario (lazy, para el avatar), y una consulta de "última actividad" por proyecto ejecutada en un property del modelo. Los dos primeros pisos se resolvieron con selectinload anidado (dos consultas con IN, independientes del número de proyectos). El tercero era la trampa interesante: la última actividad por proyecto es exactamente el patrón "top-1 por padre", así que acabó en una window function con row_number() sobre la tabla de actividades, materializada con contains_eager — una consulta, ciento veinte filas. Once segundos pasaron a 240 ms sin tocar ni el esquema ni la UI, y el presupuesto de CI quedó fijado en cinco consultas. La moraleja del caso: el N+1 escala con los datos del cliente más grande, no con los del entorno de pruebas — por eso el tenant enorme lo sufría y el de demo jamás lo mostró. Dimensiona tus datos de test pensando en tu peor cliente, no en el promedio.

Para seguir profundizando

Este problema toca fronteras que en esta web tienen guía propia: si tus listados crecen, la paginación por keyset explica por qué OFFSET colapsa y cómo paginar con cursores (y se combina de forma natural con los patrones de carga de aquí); la guía de índices en PostgreSQL cubre la otra mitad del rendimiento — que las pocas consultas que emitas sean rápidas —; y la de connection pooling con PgBouncer explica qué pasa con las conexiones que el lazy loading retiene. El N+1 rara vez viaja solo: un endpoint que lo sufre suele sufrir también paginación por OFFSET y algún índice ausente, y los tres arreglos se refuerzan entre sí.

Dos preguntas más que llegan siempre

¿raiseload en producción o solo en desarrollo?

En ambos, pero con matiz. En desarrollo y en tests, siempre: quieres el error lo antes posible. En producción, la pregunta real es qué prefieres cuando aparezca un acceso no previsto: ¿un error 500 inmediato y visible, o una degradación silenciosa que quizá nadie note en meses? Para endpoints críticos, el 500 con alerta es preferible — falla una petición y lo arreglas hoy. Para rutas secundarias, algunos equipos usan raiseload solo en los entornos previos y aceptan el riesgo en producción. Lo indefendible es no usarlo en ningún sitio: es la única herramienta que convierte esta clase entera de bugs en errores deterministas.

¿El identity map no me protege ya de consultas repetidas?

Solo a medias, y es un malentendido frecuente. El identity map garantiza que la misma fila sea el mismo objeto Python dentro de una sesión, y evita re- consultar una entidad ya cargada cuando la pides por identidad (session.get). Pero el acceso perezoso a una colección no pregunta "¿tengo estos objetos?" sino "¿tengo esta relación cargada?" — y si no lo está, emite la consulta aunque cada línea resultante ya estuviera en la sesión. Veinte pedidos del mismo cliente disparan las veinte consultas de lineas aunque compartan cliente cacheado. El identity map deduplica objetos; no deduplica cargas de relaciones. La única defensa contra eso es la carga explícita.

Nueva ampliación: herencia polimórfica, colecciones write-only y el

N+1 que cruza servicios

Esta ampliación cubre tres frentes que quedaban fuera del mapa. El primero es el N+1 que fabrica la herencia de tablas unidas, con su antídoto nativo: selectin_polymorphic. El segundo, las colecciones tan grandes que la respuesta correcta no es cargarlas mejor sino no cargarlas nunca: las relaciones write-only de SQLAlchemy 2.0. El tercero, la versión del problema que no emite ni una línea de SQL porque vive entre servicios HTTP. Y cierra con dos detalles de ingeniería con los que tarde o temprano vas a chocar: el .unique() que SQLAlchemy te exige después de un joinedload sobre colecciones, y lo que selectinload hace de verdad por debajo cuando tus padres se cuentan por miles.

El N+1 escondido en la herencia: selectin_polymorphic

La herencia de tablas unidas (joined table inheritance) reparte cada entidad entre una tabla base y una tabla por subtipo: los datos comunes de los empleados en una tabla, y las columnas propias de ingenieros y gerentes en las suyas. Es un diseño limpio... hasta que consultas la clase base y tocas un atributo del subtipo. SQLAlchemy carga las filas base con una consulta y, al acceder al primer atributo específico, emite un SELECT contra la tabla del subtipo. Por cada fila. Es el N+1 disfrazado de polimorfismo: nadie escribió un bucle sobre relaciones; solo se leyó un atributo heredado.

```
class Empleado(Base):
    __tablename__ = "empleados"
    id: Mapped[int] = mapped_column(primary_key=True)
    nombre: Mapped[str]
    tipo: Mapped[str]
    __mapper_args__ = {"polymorphic_identity": "empleado", "polymorphic_on": "tipo"}

class Ingeniero(Empleado):
    __tablename__ = "ingenieros"
    id: Mapped[int] = mapped_column(ForeignKey("empleados.id"), primary_key=True)
    lenguaje_principal: Mapped[str]
    __mapper_args__ = {"polymorphic_identity": "ingeniero"}

# Consulta sobre la clase base:
empleados = session.scalars(select(Empleado)).all()   # 1 consulta
for e in empleados:
    if isinstance(e, Ingeniero):
        print(e.lenguaje_principal)   # 1 SELECT por ingeniero: N+1
```

La solución nativa es selectin_polymorphic(), primo hermano de selectinload: después de cargar las filas base, emite una consulta adicional por subtipo con un IN sobre las claves primarias, en lugar de una consulta por fila. Dos o tres consultas totales, da igual cuántos empleados haya:

```
from sqlalchemy.orm import selectin_polymorphic

stmt = select(Empleado).options(
    selectin_polymorphic(Empleado, [Ingeniero, Gerente])
)
empleados = session.scalars(stmt).all()
# Consulta 1: SELECT ... FROM empleados
# Consulta 2: SELECT ... FROM ingenieros WHERE id IN (claves de ingenieros)
# Consulta 3: SELECT ... FROM gerentes WHERE id IN (claves de gerentes)
```

```
# Acceder a atributos del subtipo ya no emite nada
```

Si un subtipo se consulta siempre completo, puedes fijar el comportamiento en el propio mapeo con polymorphic_load="selectin" y olvidarte de declararlo por consulta. La alternativa clásica, with_polymorphic, resuelve lo mismo en una sola consulta con joins a todas las tablas de subtipos; paga con filas más anchas y un join extra por subtipo. El criterio es el de siempre: pocos subtipos con columnas ligeras, join único; muchos subtipos o columnas pesadas, selectin. Y el guardarraíl también existe en este terreno, aunque es más artesanal: las columnas diferidas admiten raiseload (defer con raiseload=True), de modo que el acceso no previsto a una columna cara falle en desarrollo en lugar de emitir una consulta silenciosa por fila.

Colecciones que jamás deberían cargarse enteras: WriteOnlyMapped

Hay relaciones que ningún eager loading puede salvar, porque el error no está en cómo se cargan sino en cargarlas: los eventos de un usuario (cientos de miles), las lecturas de un sensor, los mensajes de un canal. Para esas, SQLAlchemy 2.0 tiene una respuesta estructural: declarar la relación como write- only. Una relación WriteOnlyMapped no carga sus elementos nunca — ni perezosa ni ansiosamente —; ofrece add() y remove() para escribir, y un generador de consultas .select() para leer exactamente el subconjunto que necesitas, con filtros, orden y límite:

```
from sqlalchemy.orm import WriteOnlyMapped

class Usuario(Base):
    __tablename__ = "usuarios"
    id: Mapped[int] = mapped_column(primary_key=True)
    eventos: WriteOnlyMapped["Evento"] = relationship(
        cascade="all, delete-orphan", passive_deletes=True
    )

# Escribir: sin cargar nada en memoria
usuario.eventos.add(Evento(tipo="login"))

# Leer: una consulta explícita y acotada, no "la colección"
ultimos = session.scalars(
    usuario.eventos.select()
    .order_by(Evento.creado_en.desc())
    .limit(20)
).all()
```

El acceso estilo lista — usuario.eventos[0], len(usuario.eventos), iterar en un for — directamente no existe: no es que falle en tiempo de ejecución, es que el tipo no ofrece esas operaciones, así que el error aparece en el editor y en mypy antes de ejecutar nada. Es la filosofía de raiseload — lo implícito debe fallar — aplicada un nivel antes, en el sistema de tipos. La señal para adoptarlo: cualquier relación cuyo tamaño crece sin límite con el uso del producto. Las colecciones acotadas (las líneas de un pedido) siguen siendo territorio de selectinload; las no acotadas (el historial de un usuario) son write-only y se leen con consultas paginadas — idealmente por keyset, como explica la guía de paginación de esta web. Si vienes de SQLAlchemy 1.x: lazy="dynamic" hacía algo parecido; write_only es su sucesor tipado, sin la trampa de que alguien itere la relación "dinámica" por accidente y cargue la tabla entera.

El .unique() obligatorio: qué te está diciendo SQLAlchemy

Si has usado joinedload sobre una colección en SQLAlchemy 2.0, habrás chocado con este error: "The unique() method must be invoked on this Result". No es burocracia: es la factura del row bloat que describimos al principio, hecha explícita. El JOIN devuelve una fila por cada hija, así que un pedido con ocho líneas aparece ocho veces en el resultado crudo; unique() deduplica los objetos padre apoyándose en el identity map antes de entregártelos:

```
stmt = select(Pedido).options(joinedload(Pedido.lineas))
pedidos = session.execute(stmt).unique().scalars().all()
# Sin .unique(): InvalidRequestError
# Con .unique(): 50 pedidos, aunque el JOIN devolviera 400 filas
```

SQLAlchemy podría deduplicar en silencio — es lo que hacía la Query de 1.x — pero te obliga a escribirlo para que el coste sea visible: si necesitas deduplicar, es que estás transfiriendo filas repetidas desde la base de datos. Cada .unique() que tecleas es una invitación a preguntarte si esa colección no debería cargarse con selectinload. La nota simétrica: selectinload nunca necesita unique(), porque sus consultas devuelven filas planas sin duplicar padres.

selectinload por dentro: lotes de 500, IN expandido y PgBouncer

Saber qué hace selectinload exactamente explica dos comportamientos que desconciertan la primera vez. El primero: los lotes. selectinload no mete diez mil claves en un solo IN; trocea en lotes de 500 y emite una consulta por lote. Para diez mil padres son veinte consultas, no una — y sigue siendo la decisión correcta: el número de viajes crece con N/500 en vez de con N, y evita los IN gigantes que castigan al parser y al planificador. Si en el log ves varias consultas IN consecutivas donde esperabas una, no es un bug: es el batching.

```
stmt = select(Cliente).options(selectinload(Cliente.pedidos))
clientes = session.scalars(stmt).all()   # 1.200 clientes
# SELECT ... FROM pedidos WHERE cliente_id IN (...500 claves...)
# SELECT ... FROM pedidos WHERE cliente_id IN (...500 claves...)
# SELECT ... FROM pedidos WHERE cliente_id IN (...200 claves...)
# 4 consultas en total, no 1.201
```

El segundo: la forma de la consulta. SQLAlchemy construye el IN con parámetros expandidos — la sentencia se reescribe con el número exacto de marcadores en cada ejecución —, y eso tiene consecuencias río abajo. En pg_stat_statements, hasta PostgreSQL 17 cada cardinalidad distinta del IN aparecía como una entrada separada: al auditar N+1 conviene sumar las variantes, no mirar solo la primera fila. PostgreSQL 18 agrupa por fin las listas de constantes en una sola entrada (verás IN ($1 /*, ... */)). Y con prepared statements el efecto es el contrario: para el driver, cada cardinalidad es una sentencia distinta que preparar. Con asyncpg — cuyo caché de preparadas indexa por texto de consulta — un endpoint con IN de tamaño variable multiplica variantes en el caché; si además hay PgBouncer en modo transaction por medio, necesitas max_prepared_statements bien dimensionado (PgBouncer 1.21+) o desactivar el caché del driver. La guía de connection pooling de esta web cubre ese pantano con detalle.

El N+1 sin SQL: fan-out HTTP entre servicios

Cuando el monolito se parte en servicios, el N+1 no desaparece: cambia de protocolo. El servicio de pedidos necesita el nombre de cada cliente, y el bucle "por cada pedido, GET /clientes/id" reproduce el patrón con latencias diez o veinte veces peores — y con timeouts, reintentos y rate limits que el SQL no tenía. La solución conserva la estructura que ya conoces del ORM: juntar claves, resolver en lote, indexar el resultado. Aplicada en dos capas.

Primera capa: si el servicio remoto ofrece un endpoint de lote (GET /clientes?ids=... o un POST con la lista), úsalo — es tu selectinload entre servicios. Las mismas reglas de los lotes SQL aplican: deduplica antes de pedir, trocea en grupos razonables (500 vuelve a ser un buen número), y decide qué hacer con las claves que no vuelven. Segunda capa, cuando no existe endpoint de lote: paraleliza con límite de concurrencia. Un gather sin semáforo no es paralelismo, es una estampida contra el servicio vecino:

```
import asyncio
import httpx

async def cargar_clientes(ids: set[int]) -> dict[int, dict]:
    limite = asyncio.Semaphore(10)   # máx. 10 peticiones en vuelo

    async def uno(client, cid):
        async with limite:
            r = await client.get(f"/clientes/{cid}")
            r.raise_for_status()
            return cid, r.json()

    async with httpx.AsyncClient(base_url=URL_CLIENTES, timeout=5.0) as client:
        pares = await asyncio.gather(*(uno(client, i) for i in ids))
    return dict(pares)

# En el handler: reunir claves, deduplicar, resolver UNA vez
ids = {p.cliente_id for p in pedidos}
clientes = await cargar_clientes(ids)
for p in pedidos:
    salida.append(construir_fila(p, clientes.get(p.cliente_id)))
```

Fíjate en que el semáforo convierte "N peticiones secuenciales" en "N peticiones en oleadas de diez": con 50 clientes y 40 ms por petición, el bucle secuencial tarda dos segundos y la versión con semáforo unos 240 ms. El endpoint de lote sigue ganando (una petición, un viaje), y si diseñas tú el servicio remoto, ofrécelo: acepta hasta un máximo documentado de ids, devuelve un mapa id-objeto y explicita qué pasa con los ids inexistentes. El patrón DataLoader de la sección de GraphQL también funciona aquí tal cual — acumular claves por petición y resolverlas juntas —, y es la pieza que convierte un BFF en algo eficiente. Y una regla de arquitectura que ahorra toda la conversación: si una vista necesita datos de tres servicios para cada fila de una lista, o falta un endpoint de composición (BFF, vista materializada, réplica de lectura) o el corte de servicios está dibujado por el organigrama y no por los datos.

Radar de agosto de 2026: el estado de las piezas

Para cerrar, el estado actual de las herramientas citadas, porque varias siguen moviéndose. En Prisma, relationLoadStrategy sigue detrás del preview flag relationJoins (PostgreSQL, CockroachDB y MySQL; SQL Server pendiente), y hay un matiz operativo importante: al activar el flag, join pasa a ser la estrategia por

defecto. Revisa tus endpoints con colecciones de fan- out grande después de activarlo, porque el comportamiento por defecto deja de parecerse a selectinload. En SQLAlchemy, todo lo usado en esta ampliación — write_only, selectin_polymorphic, polymorphic_load — es API estable de la serie 2.x. En PostgreSQL, la agrupación de listas IN en pg_stat_statements llegó con la versión 18: si sigues en 17 o anterior, recuerda sumar las variantes por cardinalidad al auditar. Y los APM siguen puliendo la detección automática: la firma de "muchos spans idénticos bajo una petición" ya la agrupan como problema con nombre propio tanto Sentry como los grandes vendors. Nada de esto cambia el fondo: las herramientas detectan mejor cada año, pero la carga explícita — y hacer que lo implícito falle — sigue siendo trabajo tuyo.

Ampliación de agosto de 2026: carga filtrada, el planificador y el N+1 en arquitecturas de lectura

Hasta aquí hemos tratado el N+1 como un problema de conteo: mil consultas donde debería haber dos. Pero cuando llevas el arreglo a producción aparecen tres frentes que casi nadie documenta. El primero es que filtrar una colección y cargarla con antelación son operaciones distintas, y confundirlas produce datos incorrectos además de consultas de más. El segundo es que el planificador de PostgreSQL tiene su propia opinión sobre tu arreglo, y a veces esa opinión es peor que el problema original. El tercero es que en cuanto la base de datos deja de estar en el mismo host, cada consulta que ahorras vale bastante más milisegundos de lo que sugiere tu portátil. Esta ampliación cubre esos tres frentes.

contains_eager: cuando el JOIN que filtras también debe poblar la colección

Este es el error más caro de la lista porque no se manifiesta como lentitud sino como datos equivocados. Supón que quieres los usuarios que tienen pedidos pendientes, y que en la respuesta aparezcan solo esos pedidos pendientes. La versión intuitiva es unir por la relación, filtrar y añadir un selectinload para no caer en el N+1:

```
# INCORRECTO: el filtro y la carga viven en mundos separados
stmt = (
    select(User)
    .join(User.orders)
    .where(Order.status == "pending")
    .options(selectinload(User.orders))
)
```

El JOIN sirve para decidir qué usuarios devolver. El selectinload, en cambio, lanza una segunda consulta completamente independiente: *SELECT ... FROM orders WHERE orders.user_id IN (...)*, sin rastro del filtro por estado. Resultado: cada usuario llega con todos sus pedidos, incluidos los entregados hace dos años. La consulta es rápida, la lista está mal, y el bug sobrevive a la revisión de código porque el *SELECT* parece razonable.

La herramienta correcta es contains_eager(), que le dice al ORM que las filas del JOIN que ya has escrito son las que deben poblar la colección. No hay segunda consulta y el filtro se respeta:

```
from sqlalchemy import select
```

```
from sqlalchemy.orm import contains_eager

stmt = (
   select(User)
   .join(User.orders)
   .where(Order.status == "pending")
   .options(contains_eager(User.orders))
   .execution_options(populate_existing=True)
)
users = session.scalars(stmt).unique().all()
# 1 sola consulta, y user.orders contiene EXCLUSIVAMENTE los pendientes
```

Hay tres advertencias que conviene interiorizar antes de usarlo. La primera es populate_existing=True: si esos usuarios ya estaban en la sesión con la colección cargada, el identity map ganaría y verías la colección antigua; esa opción fuerza la sobreescritura. La segunda es que la colección resultante es una vista parcial y no es persistente: en cuanto el objeto expire o lo vuelvas a cargar en otra consulta, user.orders recupera su significado normal de *todos los pedidos*. La tercera es la más peligrosa: nunca escribas a través de una colección parcial. Si la relación tiene cascade="all, delete-orphan" y haces user.orders.remove(x) sobre una colección que solo contiene los pendientes, SQLAlchemy razona sobre el conjunto que ve, no sobre el que existe en la tabla. Trata las colecciones de contains_eager como estrictamente de lectura.

Si lo único que quieres es filtrar la colección sin que el filtro decida qué padres devolver, la opción ligera es aplicar el criterio dentro del propio loader:

```
# Todos los usuarios; de cada uno, solo sus pedidos pendientes
stmt = select(User).options(
   selectinload(User.orders.and_(Order.status == "pending"))
)
```

La regla mental: si el filtro debe reducir la lista de padres, va en el *WHERE* con contains_eager; si solo debe reducir el contenido de la colección, va dentro del and_() del loader.

with_loader_criteria: el filtro que no se te puede olvidar en ningún sitio

El soft delete y el aislamiento por tenant son los dos sitios donde las optimizaciones de carga se pudren con el tiempo. Alguien arregla un N+1 añadiendo selectinload y, sin darse cuenta, se salta el *deleted_at IS NULL* que el lazy loading original aplicaba desde un helper. El bug no es de rendimiento, es de corrección, y aparece semanas después como *por qué salen pedidos borrados en el panel*.

with_loader_criteria() resuelve esto aplicando una condición a todas las apariciones de una entidad dentro de una consulta, sin importar la estrategia de carga ni la profundidad a la que se cargue:

```
from sqlalchemy.orm import with_loader_criteria

stmt = (
   select(User)
   .options(selectinload(User.orders).selectinload(Order.items))
   .options(
      with_loader_criteria(Order, Order.deleted_at.is_(None), include_aliases=True),
      with_loader_criteria(Item, Item.deleted_at.is_(None), include_aliases=True),
```

```
    )
)
```

Lo interesante es que se puede elevar a política global con un evento de sesión, de modo que ninguna consulta futura necesite acordarse:

```
from sqlalchemy import event
from sqlalchemy.orm import Session, with_loader_criteria

@event.listens_for(Session, "do_orm_execute")
def _apply_soft_delete(state):
    if not state.is_select:
        return
    if state.execution_options.get("include_deleted", False):
        return
    state.statement = state.statement.options(
        with_loader_criteria(
            SoftDeleteMixin,
            lambda cls: cls.deleted_at.is_(None),
            include_aliases=True,
        )
    )
```

Dos detalles finos. Usar la forma lambda cls: ... en lugar de una expresión literal permite que el criterio se aplique a cualquier subclase del mixin y que SQLAlchemy lo cachee correctamente por clase. Y include_aliases=True es imprescindible: sin él, las apariciones de la entidad bajo un alias — que es exactamente lo que genera joinedload — se quedan sin filtrar, y vuelves a tener el bug pero solo en las rutas optimizadas, que es la peor forma posible de tenerlo. La escotilla de escape es session.execute(stmt, execution_options={"include_deleted": True}) para los pocos casos que sí necesitan ver lo borrado.

El resto del catálogo: subqueryload, immediateload y noload

Casi todo el mundo conoce selectinload y joinedload, y ahí se detiene. Merece la pena saber qué hacen los otros tres, aunque solo sea para reconocerlos en código heredado.

subqueryload fue el antecesor de selectinload. En lugar de emitir un *IN* con las claves ya conocidas, reescribe la consulta original como subconsulta y la vuelve a unir. Eso significa que el planificador replanifica la consulta padre entera en cada carga de relación, que necesita un *ORDER BY* determinista para ser correcto con *LIMIT*, y que no se puede combinar con yield_per. Si lo ves en un repositorio, cámbialo por selectinload y mide: en tablas grandes la diferencia suele estar entre el 30 % y varios múltiplos.

immediateload emite la carga perezosa inmediatamente, un *SELECT* por objeto. Sigue siendo N+1 en número de consultas, así que suena inútil, pero tiene un nicho real: cuando procesas con yield_per o en contextos donde el objeto va a salir de la sesión, garantiza que la relación esté poblada antes de que el acceso perezoso pueda fallar. Es la opción honesta para colecciones diminutas dentro de un streaming, no una estrategia de rendimiento.

noload devuelve la colección vacía sin consultar. Es tentador para aligerar serializaciones y es una fuente inagotable de bugs silenciosos: una lista vacía es indistinguible de *no hay elementos*. Salvo que estés

construyendo objetos para escritura y sepas que nadie leerá esa relación, prefiere raiseload, que falla ruidosamente en lugar de mentir.

```
# Streaming de 200k filas sin explotar la memoria ni caer en lazy loads
stmt = (
    select(Order)
    .options(selectinload(Order.items), raiseload("*"))
    .execution_options(yield_per=1000)
)
for order in session.scalars(stmt):
    process(order)   # items ya cargados por lotes; cualquier otra relacion falla
```

Qué hace el planificador con tu arreglo: nested loop, hash join y work_mem

Aquí es donde la conversación deja de ser sobre el ORM. Cuando cambias mil consultas por un *JOIN*, le estás entregando a PostgreSQL un problema distinto, y el plan que elija determina si el arreglo es una mejora de diez veces o una regresión bajo concurrencia.

Con joinedload, el planificador tiene dos caminos razonables. Si el número de padres es pequeño y existe índice por la clave foránea, elegirá un *nested loop* con *index scan*: rápido, memoria constante, comportamiento predecible. Si los padres son muchos, preferirá un *hash join*, que construye en memoria una tabla hash de una de las relaciones. Y ahí entra work_mem: si la tabla hash no cabe, PostgreSQL parte la operación en lotes y los escribe a disco. Tu consulta única sigue siendo una sola consulta, pero ahora hace E/S temporal, y con veinte peticiones concurrentes cada una reclamando su propio work_mem el servidor se degrada de una forma que no se parece en nada a lo que viste en local.

La forma de verlo es mirar el plan real, no el estimado:

```
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT u.*, o.*
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
WHERE u.tenant_id = 42;

-- Señales a buscar en la salida:
--   Hash Join  ... Batches: 9  Memory Usage: 4096kB  Disk Usage: 71232kB
--                ^ mas de 1 batch = se ha ido a disco: sube work_mem o cambia de estrategia
--   Rows Removed by Join Filter: 1204338
--                ^ estas leyendo filas para tirarlas: falta un indice o el filtro va tarde
--   Buffers: shared hit=812 read=44190
--                ^ read alto = no esta en cache: el coste real es E/S, no CPU
```

Con selectinload el perfil es distinto y, casi siempre, más aburrido en el buen sentido. La segunda consulta es un *WHERE user_id IN (...)* sobre una lista de claves conocidas, que se resuelve con *index scan* o *bitmap heap scan* y no necesita memoria de trabajo apreciable. Esa previsibilidad es la razón real por la que selectinload es la opción por defecto para colecciones: no es que siempre sea más rápida en el caso medio, es que es mucho más difícil que sea catastrófica en el caso malo.

Regla práctica: si el *EXPLAIN ANALYZE* de tu joinedload muestra más de un *batch* o un *Disk Usage* distinto de cero, no subas work_mem como primer reflejo. Cambia a selectinload y vuelve a medir; suele resolver el

problema sin tocar configuración global que afecta a todo lo demás.

El anti-patrón inverso: cuando arreglar el N+1 crea una consulta monstruo

El entusiasmo tiene un coste. Una vez que alguien descubre joinedload, la tentación es encadenarlo por todas las relaciones del agregado hasta que el endpoint haga exactamente una consulta. El problema es que cada colección unida multiplica las filas del resultado. Si un usuario tiene 50 pedidos y cada pedido 20 líneas, unir ambas colecciones a la vez devuelve 1.000 filas *por usuario*, con los datos del usuario repetidos mil veces por la red. Con 200 usuarios en la página son 200.000 filas para reconstruir 200 objetos.

```
# Producto cartesiano: 200 x 50 x 20 = 200.000 filas transmitidas
select(User).options(
  joinedload(User.orders).joinedload(Order.items)
)

# Amplificacion lineal: 3 consultas, ~11.000 filas en total
select(User).options(
  selectinload(User.orders).selectinload(Order.items)
)
```

La heurística es simple y aguanta bien: **como máximo una colección por** joinedload **en una misma consulta**. Las relaciones a-uno (*many-to-one*) puedes encadenarlas sin miedo porque no multiplican filas; son las colecciones las que lo hacen. En cuanto haya dos niveles de colección, selectinload es la respuesta.

Hay una métrica que merece la pena instrumentar junto al conteo de consultas: la *amplificación de filas*, es decir, filas devueltas por la base de datos dividido entre objetos construidos por el ORM. Un valor cercano a uno significa que estás transmitiendo lo justo. Un valor de 300 significa que has cambiado un problema de latencia por un problema de ancho de banda y de CPU de deduplicación, que es lo que hace .unique() por ti en silencio. Reducir consultas de 1.000 a 1 y subir la amplificación de 1 a 500 no siempre es una victoria; medirlo evita descubrirlo por las malas.

Réplicas de lectura y latencia: por qué el N+1 duele el triple fuera de tu portátil

Esta es la razón por la que un N+1 pasa desapercibido en desarrollo y arrasa en producción, y no tiene nada que ver con el tamaño de los datos. Tiene que ver con el ida y vuelta.

En local, la base de datos vive en el mismo host: cada consulta trivial cuesta unos 0,1–0,3 ms de ida y vuelta. Doscientas consultas son 40 ms; nadie lo nota. En producción, con la aplicación y la réplica en zonas de disponibilidad distintas, ese mismo ida y vuelta pasa a 1–3 ms; las mismas doscientas consultas son ahora entre 200 y 600 ms de puro tiempo de red. Si la réplica está en otra región — algo cada vez más común en despliegues multirregión con lectura local y escritura remota — hablamos de 30–80 ms por consulta, y doscientas consultas se convierten en un tiempo de espera que ningún usuario va a tolerar.

Lo perverso es cómo se ve esto en los paneles. La métrica de *duración media de consulta* de tu base de datos estará impecable: cada una de esas consultas tarda 0,2 ms en ejecutarse. El tiempo se va en la red y en la sesión esperando, y eso no aparece en las estadísticas del servidor. De ahí la regla que conviene grabar a fuego: **la métrica que predice el N+1 no es la duración de las consultas, es el número de consultas por petición**. La primera es una propiedad de la base de datos; la segunda es una propiedad de tu código, y es la

única de las dos que empeora sola.

```
# Presupuesto de latencia, en una servilleta
# consultas_por_peticion x rtt_ms = suelo de latencia inevitable
#
#   mismo host     200 x 0.2 ms  =   40 ms   (invisible)
#   misma AZ       200 x 1.0 ms  =  200 ms   (molesto)
#   entre AZ       200 x 3.0 ms  =  600 ms   (incidencia)
#   entre regiones 200 x 45 ms   = 9000 ms   (timeout)
#
# El arreglo baja 200 -> 3. En la ultima fila eso es 9 s -> 135 ms.
```

Hay un segundo efecto, más sutil, si estás detrás de PgBouncer en modo transacción. Cada carga perezosa que ocurre fuera de una transacción explícita puede tomar y devolver una conexión del pool. Doscientas cargas perezosas son doscientos ciclos de adquisición, cada uno con su contención de cerrojo en el pooler. Bajo carga, el N+1 deja de ser un problema de tu petición y pasa a ser un problema de todas las demás, porque tu petición está monopolizando el pooler. Es el mecanismo habitual por el que un endpoint lento acaba tumbando endpoints que no tienen nada que ver.

Prisma: una extensión de cliente que grita cuando aparece el N+1

En el lado de TypeScript, las extensiones de cliente de Prisma permiten interceptar toda operación sin envolver el cliente a mano. Combinadas con AsyncLocalStorage puedes contar consultas por petición y hacer que el proceso falle en desarrollo cuando un mismo modelo se consulta de forma sospechosamente repetida:

```
import { AsyncLocalStorage } from "node:async_hooks";
import { PrismaClient } from "@prisma/client";

type QueryCounter = Map<string, number>;
export const requestScope = new AsyncLocalStorage<QueryCounter>();

const UMBRAL = 10;

export const prisma = new PrismaClient().$extends({
  query: {
    $allModels: {
      async $allOperations({ model, operation, args, query }) {
        const counter = requestScope.getStore();
        if (counter) {
          const key = model + "." + operation;
          const n = (counter.get(key) ?? 0) + 1;
          counter.set(key, n);
          if (n === UMBRAL + 1 && process.env.NODE_ENV !== "production") {
            throw new Error(
              "Posible N+1: " + key + " se ha ejecutado " + n + " veces en una sola peticion"
            );
          }
        }
        return query(args);
      },
    },
  },
```

```
});
```

```
// Middleware que abre el ambito por peticion (Express, Fastify o similar)
app.use((req, res, next) => {
  const counter: QueryCounter = new Map();
  requestScope.run(counter, () => {
    res.on("finish", () => {
      const total = [...counter.values()].reduce((a, b) => a + b, 0);
      res.setHeader; // ya enviado: registramos en el log estructurado
      logger.info({ path: req.path, consultas: total, detalle: Object.fromEntries(counter) });
    });
    next();
  });
});
```

Este patrón tiene una ventaja sobre el logging tradicional: no te obliga a leer registros. En desarrollo lanza una excepción con el nombre del modelo culpable en el momento exacto en que el patrón aparece, y en producción emite un campo numérico por petición que puedes alertar. Es el equivalente en Prisma del presupuesto de consultas en CI que vimos antes, pero funcionando en tiempo real.

Sobre relationLoadStrategy conviene recordar la diferencia de fondo, porque el nombre confunde: join resuelve la relación con un *LATERAL JOIN* en PostgreSQL y devuelve todo en una consulta, mientras que query emite una consulta por tabla y las une en la aplicación, que es conceptualmente lo mismo que hace selectinload. La intuición de cuándo usar cada una es la misma que en SQLAlchemy: join para relaciones a-uno y colecciones pequeñas, query cuando el fan-out es grande y el *JOIN* empezaría a multiplicar filas.

Paginación por keyset y relaciones: cargar sin volver al punto de partida

Un último patrón que combina dos temas y que casi siempre se implementa mal. Cuando paginas con *OFFSET* profundo y además cargas relaciones, pagas dos veces: la base de datos tiene que recorrer y descartar todas las filas anteriores al offset, y el eager loading se aplica sobre ese trabajo desperdiciado. La paginación por keyset elimina el primer coste, y el orden correcto de las operaciones elimina el segundo.

```
from sqlalchemy import select, tuple_
from sqlalchemy.orm import selectinload

def pagina_de_pedidos(session, cursor=None, limite=20):
    stmt = select(Order).order_by(Order.created_at.desc(), Order.id.desc()).limit(limite)
    if cursor is not None:
        creado, ident = cursor
        stmt = stmt.where(tuple_(Order.created_at, Order.id) < (creado, ident))

    # Primero se acota la pagina; solo despues se cargan las relaciones
    stmt = stmt.options(selectinload(Order.items), selectinload(Order.customer))
    pedidos = session.scalars(stmt).all()

    siguiente = (pedidos[-1].created_at, pedidos[-1].id) if pedidos else None
    return pedidos, siguiente
```

La clave es el orden mental: *acotar, luego cargar*. El *WHERE* por tupla usa el índice compuesto (created_at, id) y salta directamente al punto de corte sin importar la profundidad de la página, y el selectinload posterior

opera sobre exactamente veinte claves. La página mil cuesta lo mismo que la página uno, con el mismo número de consultas. Con *OFFSET*, la página mil cuesta veinte mil filas descartadas antes de empezar.

Una nota de corrección que se pasa por alto: la comparación tiene que ser por tupla, no por columnas sueltas encadenadas con *AND*. Si dos pedidos comparten created_at al milisegundo — algo que ocurre en cuanto hay importaciones por lotes — comparar solo por fecha se salta filas o las repite. La tupla (created_at, id) impone un orden total y hace que la paginación sea determinista.

Qué medir mañana por la mañana

Si de toda esta ampliación te llevas solo tres cosas, que sean estas. Una: instrumenta *consultas por petición* como métrica de primera clase junto a la latencia, porque es la única que detecta el N+1 antes de que sea una incidencia. Dos: cada vez que sustituyas un N+1 por un *JOIN*, mira el *EXPLAIN (ANALYZE, BUFFERS)* del resultado y comprueba que no has cambiado mil consultas baratas por una consulta que se va a disco. Y tres: cuando filtres una colección, decide conscientemente si el filtro reduce los padres — contains_eager — o solo el contenido — and_() dentro del loader — porque esa distinción es la diferencia entre un endpoint rápido y un endpoint que devuelve datos que no son.

English version

What the N+1 problem is (and why you almost certainly have it)

The N+1 problem is the most common performance bug in applications that use an ORM, and also the sneakiest: the code that causes it looks perfectly reasonable. It happens when you load a list of N objects with one query and then, as you iterate over them, the ORM fires an additional query for each one to fetch its relations. One query silently becomes 1 + N, and nothing in the code gives it away.

```
# Looks like a single query... but it is not
posts = session.scalars(select(Post).limit(50)).all()  # 1 query

for post in posts:
    print(post.author.name)   # 1 extra query FOR EACH post

# Total: 1 + 50 = 51 round trips to the database
```

The culprit is **lazy loading**, the default strategy in most ORMs: relations are not loaded until you touch them. That is a sensible default for avoiding data you might never use, but inside a loop it turns into a query factory.

Why it matters: the math of the disaster

Say each query takes 2 ms of round trip to the database. With 50 posts, those 51 trips add up to about 100 ms of pure network latency, before counting any actual database work. With 500 rows you are past one second. Worst of all: you will not notice it in development, because everything is fast with 10 test rows. N+1 is a problem that *scales with your data*, so it blows up exactly when your application starts to succeed.

There is a second, less obvious cost: every query holds a connection from the pool for its full round trip. An endpoint with an N+1 under real traffic multiplies pressure on the pool and can cause timeouts in endpoints that have nothing to do with it.

How to detect it before your users do

The unmistakable signal: many identical consecutive queries where only one parameter changes. In SQLAlchemy, turning on SQL logging in development is often enough:

```
engine = create_engine(DATABASE_URL, echo=True)  # development only
```

For something more systematic, count queries with an event listener. This context manager fails loudly if a block of code exceeds the expected limit:

```
from contextlib import contextmanager
from sqlalchemy import event

@contextmanager
```

```
def max_queries(engine, limit):
    executed = []

    def counter(conn, cursor, statement, params, context, executemany):
        executed.append(statement)

    event.listen(engine, "before_cursor_execute", counter)
    try:
        yield executed
    finally:
        event.remove(engine, "before_cursor_execute", counter)
        count = len(executed)
        assert count <= limit, (
            f"{count} queries executed (limit: {limit}). "
            "Likely N+1: review your loading strategies."
        )
```

In Prisma, the equivalent is listening to the client's query events:

```
const prisma = new PrismaClient({
  log: [{ emit: 'event', level: 'query' }],
});

prisma.$on('query', (e) => {
  console.log(e.query, e.duration + 'ms');
});
```

In production, OpenTelemetry traces make it obvious: a request span with dozens of identical cascading database child spans is the visual signature of an N+1.

SQLAlchemy 2.0: pick the right loading strategy

SQLAlchemy does not force you to choose between lazy loading and fetching everything: it offers several eager loading strategies declared per query with options(). The two that solve 95% of cases are selectinload and joinedload.

selectinload: the default choice for collections

selectinload runs a second query with an IN clause over the parent keys. Two queries total, no matter how many rows there are:

```
from sqlalchemy import select
from sqlalchemy.orm import selectinload

stmt = (
    select(Post)
    .options(selectinload(Post.comments))
    .limit(50)
)
posts = session.scalars(stmt).all()

# Query 1: SELECT * FROM posts LIMIT 50
```

```
# Query 2: SELECT * FROM comments WHERE post_id IN (id1, id2, ..., id50)
```

Because each query returns "flat" rows without duplicating parent data, it scales well with large collections. It is the right choice for one-to-many and many-to-many relationships.

joinedload: for to-one relationships

joinedload resolves everything in a single query with a JOIN. It is ideal when the relation points to a single object (many-to-one or one-to-one), because the JOIN does not multiply rows:

```
from sqlalchemy.orm import joinedload

stmt = (
    select(Post)
    .options(joinedload(Post.author))  # many-to-one: safe JOIN
    .limit(50)
)
```

The rule of thumb: **to-one relation → joinedload; to-many relation → selectinload**. Using joinedload on large collections causes row bloat: the JOIN repeats the parent's data on every child row, and if you chain two collections you get a cartesian product that can return thousands of rows for 50 posts.

Nested relationships

Strategies chain together to load complete object graphs predictably:

```
stmt = (
    select(Post)
    .options(
        joinedload(Post.author),
        selectinload(Post.comments).joinedload(Comment.author),
    )
    .limit(50)
)
# 2 queries total: posts+authors (JOIN) and comments+authors (IN + JOIN)
```

raiseload: the guardrail that turns N+1 into an error

The best defense is making accidental lazy loading impossible. With raiseload("*"), any access to a relation that was not explicitly loaded raises an exception instead of firing a silent query:

```
from sqlalchemy.orm import joinedload, raiseload

stmt = (
    select(Post)
    .options(
        joinedload(Post.author),
        raiseload("*"),  # everything else: error, not a query
    )
)
# post.comments -> InvalidRequestError: 'Post.comments' is not available due to lazy='raise'
```

If you use async SQLAlchemy you get this behavior for free: implicit lazy loading fails with MissingGreenlet, which turns every potential N+1 into a visible error during development. It is annoying on purpose, and that is a good thing.

Prisma: include and relationLoadStrategy

In Prisma, the classic N+1 shows up when querying relations inside a loop:

```
// BAD: 1 + N queries
const posts = await prisma.post.findMany({ take: 50 });
for (const post of posts) {
  const author = await prisma.user.findUnique({
    where: { id: post.authorId },
  });
}
```

The fix is identical in spirit to eager loading: ask for the relations in the same query with include (or a nested select):

```
// GOOD: constant number of queries
const posts = await prisma.post.findMany({
  take: 50,
  include: {
    author: true,
    comments: { include: { author: true } },
  },
});
```

On top of that, Prisma lets you choose *how* those relations are loaded with relationLoadStrategy (available on PostgreSQL and MySQL after enabling the relationJoins preview feature in your schema's generator block):

```
const posts = await prisma.post.findMany({
  relationLoadStrategy: 'join',   // 1 query with LATERAL JOIN + JSON aggregation
  // relationLoadStrategy: 'query' // several queries + merge in the application
  take: 50,
  include: { author: true },
});
```

The join strategy does the work in the database (on PostgreSQL it uses LATERAL JOINs and JSON aggregation); query runs one query per table and merges the results in the application, just like selectinload. The same intuition applies: join for to-one relations or small result sets, query when collections are large and a JOIN would duplicate too much data.

One surprising detail: within the same tick, Prisma automatically batches identical findUnique calls with an internal dataloader, which mitigates N+1 in GraphQL resolvers. Do not rely on it for sequential code that awaits inside loops: batching cannot help you there.

The mistakes that reintroduce N+1

- **Automatic serialization.** You load the posts without relations and hand them to Pydantic, a DRF serializer, or JSON.stringify with a getter. The serializer walks the relations and triggers the N+1 out of sight. The query must load everything the response schema needs.
- **Accessing a collection "just to count it".** len(post.comments) loads the entire collection only to throw the objects away. Use a count subquery or an annotated column.
- **Chaining joinedload over two collections.** Not an N+1, but the resulting cartesian product can be worse. Collections: always selectinload.
- **Trusting memory.** The teammate who adds a field to the serializer six months later does not know the query never loads that relation. Without guardrails (raiseload, query-counting tests), the N+1 always comes back.

Hardening: tests that count queries

N+1 is a classic regression bug: you fix it today and someone reintroduces it in three months. The definitive defense is a test that fails if the number of queries grows with the number of rows:

```
def test_list_posts_query_count(session, engine, make_posts):
    make_posts(50)  # fixture that creates 50 posts with author and comments

    with max_queries(engine, limit=3):
        result = list_posts_with_comments(session)

    assert len(result) == 50
```

The key is not the exact number but that the limit is *constant*: if the test passes with 50 rows and also with 500, the endpoint is O(1) in queries. In the Django ecosystem, assertNumQueries does exactly this; the context manager above gives you the same thing in SQLAlchemy.

Final checklist

- Turn on SQL logging in development and be suspicious of repeated queries where only a parameter changes.
- To-one relation → joinedload / join. To-many relation → selectinload / query.
- Never chain joins over two collections in the same query.
- Use raiseload("*") (or async mode) to turn accidental lazy loading into a development-time error.
- The query must load exactly what the serializer needs: no more, no less.
- Protect your critical endpoints with query-counting tests that fail when the number stops being constant.

You do not eliminate N+1 once: you keep it at bay with habits. Explicit loading, guardrails that turn carelessness into errors, and tests that watch the query count. With those three pieces, your ORM stops being a box of surprises and goes back to being what it promised: a tool that saves you work without costing you performance.

Extended edition: from one-off fix to a system that does not relapse

Everything above fixes the N+1 you already found. This extension is about the rest: detecting it in production when nobody is looking for it, the corners where it reappears wearing a different face (async, large collections, aggregates, GraphQL, the serialization layer), and how to build guardrails that turn it into a CI error instead of a Saturday-night incident.

Detection in production: pg_stat_statements and the repeated-query pattern

In development you watch queries scroll through the log. In production nobody reads the log, so N+1 is detected by its statistical footprint: a short, normalized query with an execution count wildly out of proportion to traffic. The pg_stat_statements extension groups queries by shape (parameters replaced with placeholders), and that grouping is exactly what you need: a thousand executions of the same query with different ids collapse into a single row with calls=1000.

```
-- N+1 suspects: many calls, little time per call
SELECT
  calls,
  round(mean_exec_time::numeric, 2) AS mean_ms,
  round((calls * mean_exec_time)::numeric, 0) AS total_ms,
  left(query, 90) AS query
FROM pg_stat_statements
WHERE calls > 1000
  AND mean_exec_time < 5      -- each one is "fast"
ORDER BY calls DESC
LIMIT 20;
```

The psychological trap of N+1 is that no individual query is slow: each takes two milliseconds, which is why it never shows up in the slow-query log. The damage lives in the product calls × mean, not in the mean. Sort by that column and the victims of lazy loading float to the surface. If you run an APM, look for the same signal in traces: dozens of identical, consecutive database spans hanging off a single HTTP request — Sentry, for instance, detects this pattern automatically and groups it as an "N+1 Query" performance issue; Datadog and New Relic show the cascade of repeated spans in the trace view. A healthy trace for a typical endpoint has one to five queries; when you see forty three-millisecond spans with the same shape, you already know which relationship forgot its selectinload.

Counting queries from the inside: event instrumentation

The APM warns you late: the code is already deployed. Your own instrumentation warns you at the exact moment a change inflates the query count, and in SQLAlchemy it costs fifteen lines thanks to the engine's event system:

```
from contextlib import contextmanager
from sqlalchemy import event

@contextmanager
def query_counter(engine):
    """Counts SQL queries executed inside the block."""
    queries = []

    def record(conn, cursor, statement, params, context, executemany):
```

```
        queries.append(statement)

    event.listen(engine, "before_cursor_execute", record)
    try:
        yield queries
    finally:
        event.remove(engine, "before_cursor_execute", record)

# Usage in a test or a suspicious endpoint:
with query_counter(engine) as queries:
    orders = get_orders_with_customers(session)

print(f"{len(queries)} queries")  # you expected 2, you get 51: N+1
```

In a FastAPI application you can go one step further and hang the counter off a middleware: every response ships with an X-DB-Queries header and, when the number crosses a threshold (say fifteen), a structured warning fires with the route. That turns N+1 into something that shows up on your dashboards the very day it is introduced, with the exact route, without waiting for a user to notice the slowness. The Prisma equivalent is the query log event — we will get to it in its section — and the idea is identical: queries per request is a first-class metric, not a debugging curiosity.

Async SQLAlchemy: where lazy loading simply does not exist

If you use SQLAlchemy with asyncio (modern FastAPI, for example), the problem changes nature: lazy loading is not slow — it is forbidden. Touching an unloaded relationship raises MissingGreenlet, because lazy loading would need to perform IO inside an attribute access, and that cannot be an await point. The error is baffling the first time ("I was reading an attribute, why is it talking about greenlets?"), but it is N+1 manifesting as an exception instead of silent slowness.

```
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.orm import selectinload
from sqlalchemy import select

async def orders_with_lines(session: AsyncSession):
    result = await session.execute(
        select(Order)
        .options(
            selectinload(Order.lines).selectinload(Line.product),
            selectinload(Order.customer),
        )
        .where(Order.status == "pending")
    )
    return result.scalars().all()
    # order.lines[0].product.name works: everything came preloaded
    # any relationship NOT listed above -> MissingGreenlet
```

The correct reading: async forces you to do what you should be doing in sync code out of discipline. Every relationship the response will touch gets declared in the query, period. That yields a migration recipe that works surprisingly well: before moving a sync codebase to async, add lazy="raise" to your relationships (or raiseload("*") to your queries) and fix every spot that blows up. Once the sync code performs no lazy loads at all, the async migration is nearly mechanical — every potential MissingGreenlet has already been eliminated,

with much clearer error messages.

Large collections: selectinload knows nothing about limits

Eager loading strategies have a blind spot almost nobody mentions until it hurts: they load the entire collection. "Each customer's three most recent orders" cannot be expressed with selectinload — there is no per-relationship limit — so naive eager loading fetches the veteran customer's ten thousand orders just to keep three. You eliminated the N+1 and created a memory problem in exchange.

The idiomatic PostgreSQL solution is a window function (or a LATERAL join) that numbers rows per group and cuts, combined with contains_eager so SQLAlchemy populates the relationship from your manual join instead of emitting its own:

```
from sqlalchemy import select, func
from sqlalchemy.orm import contains_eager, aliased

# Subquery: number each customer's orders, newest first
row = func.row_number().over(
    partition_by=Order.customer_id,
    order_by=Order.created_at.desc(),
).label("row")

sub = select(Order, row).subquery()
TopOrder = aliased(Order, sub)

query = (
    select(Customer)
    .join(TopOrder, TopOrder.customer_id == Customer.id)
    .where(sub.c.row <= 3)
    .options(contains_eager(Customer.orders.of_type(TopOrder)))
)
customers = session.execute(query).unique().scalars().all()
# customer.orders holds at most 3 elements: the most recent ones
```

The crucial detail is contains_eager: it tells the ORM "this relationship already came in this join, do not load it yourself." Without that option, SQLAlchemy would use the join for filtering and then, on accessing customer.orders, fire its own query — N+1 back through the rear door, with the aggravating factor that it would now return all ten thousand orders, not three. In Prisma the equivalent is direct, because include accepts nested take and orderBy (include: { orders: { take: 3, orderBy: { createdAt: "desc" } } }), and with the join strategy it resolves to a single LATERAL query.

Wide columns: the N+1 of bytes

There is a variant of the problem that multiplies bytes rather than queries. Your products table has a two-hundred-kilobyte description_html column and a JSONB spec sheet of similar heft; the listing only needs name and price, but the ORM fetches full rows. Fifty products per page times two hundred kilobytes is ten megabytes traveling, deserializing, and getting discarded — on every request. It is the same underlying mistake (loading out of convenience what will not be used), and it takes the same medicine: explicit loading.

```
from sqlalchemy.orm import load_only, defer
```

```
# Option 1: column allowlist for the listing
query = select(Product).options(
    load_only(Product.id, Product.name, Product.price)
)

# Option 2: heavy columns deferred by default, on the model
class Product(Base):
    __tablename__ = "products"
    # ...
    description_html: Mapped[str] = mapped_column(deferred=True)
    spec_sheet: Mapped[dict] = mapped_column(JSONB, deferred=True)
```

Watch out with the second option: a deferred column that does get used becomes... one lazy query per row. Deferring description_html and then rendering it in a loop of fifty products is hand-crafting an N+1. The rule: defer for columns only the detail view reads, explicit load_only for listings, and the query-counting test (we are getting there) watching that nobody breaks the contract. In Prisma, select and omit play the same role, with omit especially convenient for globally excluding heavy columns in the client configuration.

The N+1 of aggregates: counting without iterating

"Show each category with its product count" is the most common N+1 after the relationship kind: one COUNT per category inside the render loop. Fifty categories, fifty COUNTs. The solution has lived in SQL for forty years — GROUP BY — and the work is merely expressing it in your ORM:

```
from sqlalchemy import select, func

# A single query: category + product count
query = (
    select(Category, func.count(Product.id).label("product_count"))
    .outerjoin(Product)
    .group_by(Category.id)
    .order_by(Category.name)
)
for category, product_count in session.execute(query):
    print(category.name, product_count)
```

If the count is needed in many places, SQLAlchemy lets you attach it to the model with column_property (a correlated subquery loaded with the entity), and Prisma solves it with include: { _count: { select: { products: true } } } — one query with the aggregate included. The sophisticated version of the same sin is the aggregate over a loaded relationship: len(category.products) looks innocent, but if the relationship is not loaded it triggers a full collection load just to throw everything away except the number. Counting in Python what the database can count in SQL means paying twice: once in queries, once in memory.

Prisma in depth: relationLoadStrategy, the fluent API, and the hidden dataloader

The basic section showed include; the interesting decision in Prisma is how that include materializes. With the relationJoins feature flag enabled, relationLoadStrategy accepts two values: join (the default once the flag is on) generates a single query using LATERAL joins and JSON aggregation on PostgreSQL — the database returns the tree already assembled — while query issues one query per table and joins the results in

the application, which is essentially what selectinload does in SQLAlchemy.

```
// schema.prisma
generator client {
  provider        = "prisma-client-js"
  previewFeatures = ["relationJoins"]
}

// Per-query choice:
const orders = await prisma.order.findMany({
  relationLoadStrategy: "query",   // several queries, app-side join
  include: { customer: true, lines: { include: { product: true } } },
});
```

Which to choose? join minimizes round trips and tends to win with to-one relations and moderate trees; query performs better when fan-out is large and server-side JSON aggregation starts costing the database CPU and memory — the same joinedload/selectinload dilemma under different names. As always: measure with your data, not with someone else's benchmark.

Prisma also hides a defense worth knowing: when you use the fluent API (prisma.order.findUnique(...).customer()) to resolve relations, calls that coincide in the same event-loop tick are automatically batched — an internal dataloader. That is why some "accidental" N+1s in GraphQL resolvers backed by Prisma hurt less than they should. But it is a safety net with conditions (the calls must be compatible and simultaneous), not a license to iterate. And to watch all of this, Prisma exposes the query event:

```
const prisma = new PrismaClient({
  log: [{ emit: "event", level: "query" }],
});

let queries = 0;
prisma.$on("query", (e) => {
  queries += 1;
  if (e.duration > 200) {
    console.warn("slow query:", e.duration, "ms");
  }
});
// Per request: reset the counter in an HTTP middleware
// and alert when an endpoint exceeds its budget.
```

GraphQL: the industrial-scale N+1 generator

GraphQL deserves its own section because its execution model manufactures the problem by design: every field resolves independently, so a query asking for twenty orders with their customer runs the customer resolver twenty times. Nobody wrote a loop; the runtime wrote it for you. The canonical answer is the DataLoader pattern: accumulate the requested keys during one execution tick, resolve them all in a single batched query, and cache within the request so the same customer asked for twice resolves once.

```
# Python with Strawberry: one DataLoader per relation
from strawberry.dataloader import DataLoader
```

```
async def load_customers(ids: list[int]) -> list[Customer]:
    result = await session.execute(
        select(Customer).where(Customer.id.in_(ids))
    )
    by_id = {c.id: c for c in result.scalars()}
    return [by_id[i] for i in ids]   # same order as the keys

customer_loader = DataLoader(load_fn=load_customers)

# In the customer field resolver:
async def resolve_customer(order) -> Customer:
    return await customer_loader.load(order.customer_id)
# 20 orders -> 1 single query WHERE id IN (...all 20 keys...)
```

Two contracts of the pattern that must not be violated: the batch function must return results in the same order as the received keys (including None for missing ones), and the loader must be created per request, never shared across users — its internal cache knows nothing about permissions and would serve one request's data to another. In the TypeScript ecosystem, the reference dataloader library enforces the same contracts; with Prisma, the fluent API already gives you part of the effect, but an explicit DataLoader remains the right tool when resolution crosses services or needs fine-grained batching control.

The serialization layer fires queries too

A particularly treacherous N+1 lives not in your handler but after it: in serialization. You return a list of ORM objects and your schema framework — Pydantic with from_attributes, a DRF- style serializer, your web framework's automatic serialization — walks the declared attributes to build the response. If the schema declares fields that correspond to unloaded relationships, the serializer touches them, and every touch is a lazy query. The handler looks clean; the log shows fifty queries emitted during JSON construction.

```
class OrderOut(BaseModel):
    model_config = ConfigDict(from_attributes=True)
    id: int
    total: Decimal
    customer: CustomerOut   # <- if Order.customer was not loaded,
                            #    THIS field fires one query per order

@app.get("/orders", response_model=list[OrderOut])
def list_orders(session: Session = Depends(get_session)):
    return session.execute(
        select(Order).options(
            selectinload(Order.customer),  # load what the schema declares
            raiseload("*"),                # and let the undeclared blow up
        )
    ).scalars().all()
```

The selectinload + raiseload("*") combination is the perfect contract between query and schema: the query loads exactly what the schema serializes, and any mismatch — someone adds a field to the schema without touching the query — manifests as a loud error in development, not as silent degradation in production. If you prefer full decoupling, map to explicit DTOs in the handler; more verbose, but the serializer can no longer touch anything you did not hand it.

Manual batching with IN: when no ORM will save you

Sometimes the loading does not go through ORM relationships: you read ids off a queue, join data across two different databases, or call a service per element. The underlying pattern is always the same and deserves to exist as a utility in your codebase: gather the keys, resolve them in batches, index the result.

```
from itertools import islice

def batched(iterable, size):
    it = iter(iterable)
    while batch := list(islice(it, size)):
        yield batch

def load_by_ids(session, model, ids, batch_size=1000):
    """Resolves ids in batches with IN; returns a dict id -> entity."""
    result = {}
    for batch in batched(set(ids), batch_size):
        rows = session.execute(
            select(model).where(model.id.in_(batch))
        ).scalars()
        result.update({r.id: r for r in rows})
    return result

products = load_by_ids(session, Product, ids_from_queue)
for item in items:
    process(item, products.get(item.product_id))
```

Batches of a thousand are not superstition: drivers and databases cap the number of parameters per query (PostgreSQL accepts tens of thousands, but parser performance degrades much earlier), and an IN with two hundred thousand keys is its own incident. Deduplicate keys before querying, decide what to do with the ones that do not come back (the explicit .get with None in the example), and if this crosses services instead of tables, the same pattern is called a "batch endpoint" and applies exactly the same.

When N+1 is acceptable (and when the answer is denormalization)

Technical honesty requires saying it: not every N+1 deserves hunting. An admin script walking twenty rows once a day can afford its twenty queries; readability is worth more than the sixty milliseconds. The criteria that turn N+1 into a real problem are N growing with the data (20 today, 4,000 in a year), sitting on the hot path of an interactive request, or multiplying through nesting (N customers × M orders). Absent all three, fixing the N+1 is premature optimization with a complexity cost.

At the opposite extreme are cases where even the best eager loading is not enough: the listing that needs each of a thousand visible customers' order count, latest order date, and running total. There the answer is no longer loading better but not computing on the fly: denormalized counters maintained by triggers or by the application (with the discipline of updating them in the same transaction), materialized views refreshed by cron for data that tolerates minutes of delay, or outright a summary table a job rebuilds. Denormalizing means accepting redundancy in exchange for O(1) reads; it is the right tool when the aggregation is expensive, frequent, and tolerant of a pinch of staleness — and a source of subtle bugs when used without those three conditions.

Case study: /orders, from 4.2 seconds to 90 milliseconds

Real numbers from a real endpoint (anonymized): a listing of fifty orders with customer, lines, and each line's product. The naive version — loop over orders, touch order.customer, order.lines, and line.product — emitted 1 + 50 + 50 + 380 = 481 queries: one for the list, one per customer, one per line collection, one per product per line. At two- and- change milliseconds of round trip per query, the endpoint took 4.2 seconds, with the database at 4% CPU: all the time was network latency, multiplied.

The fix, in three measurable steps. First, selectinload on the three relationships: 481 queries became 4 (the list, the customers in one IN, the lines in another, the products in another) and latency fell to 210 ms. Second, load_only to exclude the products' HTML description — which weighed 80% of the transferred bytes and the listing never showed —: 130 ms. Third, raiseload("*") as guardrail and a budget test pinning "4 queries" as the endpoint's contract: a stable 90 ms, and the guarantee that the next refactor cannot silently undo the work. None of the three steps required touching the schema or caching anything: just telling the ORM exactly what to load.

Query budgets in CI: the contract that does not decay

The query- counting test from the earlier section has an industrialized version: per- endpoint budgets versioned alongside the code. The idea is that every critical endpoint declares how many queries it is allowed to emit, and exceeding it breaks the build with a message explaining what to do:

```
BUDGETS = {
    "GET /orders": 4,
    "GET /orders/{id}": 3,
    "GET /customers": 2,
}

@pytest.mark.parametrize("route,limit", BUDGETS.items())
def test_query_budget(http_client, engine, route, limit):
    method, path = route.split(" ", 1)
    path = path.replace("{id}", "1")
    with query_counter(engine) as queries:
        response = http_client.request(method, path)
    assert response.status_code == 200
    assert len(queries) <= limit, (
        f"{route} emitted {len(queries)} queries (limit {limit}). "
        "If the increase is legitimate, raise the budget in this test "
        "and explain why in the commit."
    )
```

The friction is deliberate: raising the budget requires touching the test and justifying it, so an accidentally introduced N+1 cannot sneak through. In the Prisma world you achieve the same by counting query events in integration tests. Two nuances so the system ages well: set budgets with a small margin (exact ones break on innocent changes, like an extra pool healthcheck query) and run these tests against a database with enough data — with three rows in the table, an N+1 emits four queries and passes the budget it would shatter with fifty.

Frequently asked questions

joinedload or selectinload? The one-line rule

To-one (many orders → one customer): joinedload, because the join does not multiply rows. To-many (one order → many lines): selectinload, because it avoids the cartesian product and in-memory deduplication. Nested: mix them per level by the same rule. And when in doubt, selectinload is the safe default for everything, at the cost of one extra round trip.

Doesn't caching spare me from fixing the N+1?

It hides it from you, which is worse. The day the cache expires cold, the 481 queries are still there, all at once, under peak traffic. Fix the shape of the queries first and cache afterwards if still needed: a cache is a performance multiplier, not a substitute for healthy queries.

How do I catch N+1 in code review before it reaches a test?

Look for the pattern, not the queries: any relationship-attribute access inside a loop (for order in orders: order.customer...) is suspicious on visual inspection, without running anything. It is a cheap code-review check that catches most cases, and the reason raiseload turns the ones that slip through into errors impossible to ignore.

Does this apply the same in Django, Rails, or Go?

The problem is identical and the tools have different names: select_related/ prefetch_related and assertNumQueries in Django; includes and the bullet gem in Rails; Go ORMs (GORM, Ent) with their Preload/ With. The four pillars — explicit loading, byte limits, error guardrails, and query-counting tests — translate one-to-one into any ecosystem with an ORM.

The write-side N+1: updates in a loop and the unit of work

Everything above talks about reads, but the loop that writes has exactly the same pathology: for product in products: product.price *= 1.1 followed by a commit emits one UPDATE per row. SQLAlchemy groups flushes inside the unit of work and sends them batched when it can (executemany), which cushions the blow; even so, for bulk updates whose new value can be expressed in SQL, the difference between "N two-millisecond updates" and "one twenty-millisecond update" is still an order of magnitude, plus the savings of not loading the rows into memory first:

```
from sqlalchemy import update, insert

# Instead of load + iterate + mutate + flush:
session.execute(
    update(Product)
    .where(Product.category_id == category_id)
    .values(price=Product.price * 1.1)
)

# And for bulk inserts, one statement with a list of dicts:
session.execute(
    insert(Event),
    [{"type": e.type, "payload": e.payload} for e in events],
```

```
)
```

In Prisma, updateMany and createMany play the same role. The signal for hunting this variant: a job "that takes hours" whose time goes into thousands of tiny statements — the same pg_stat_statements profile you already know how to read, but with UPDATE instead of SELECT. The honest counterpart: a bulk SQL update skips the ORM's events (validations, hooks, audit signals living on the model); if you depend on them, either reimplement them in the statement (updated_at=func.now()) or accept the loop, batched.

Large exports: yield_per and the danger of streaming with relationships

The export endpoint — "give me the year's hundred thousand orders as CSV" — combines two needs badly: streaming so you do not load everything into memory, and relationships so each row comes out complete. yield_per solves the first by fetching rows in batches; the nuance is how it gets along with eager loading. Join-based collection loading (joinedload on collections) is incompatible with streaming — the ORM cannot guarantee an entity is complete until it has seen all its join rows — while selectinload works: it runs once per batch, loading that batch's relationships.

```
query = (
    select(Order)
    .options(selectinload(Order.customer))
    .execution_options(yield_per=500)
)
for batch in session.execute(query).partitions():
    for (order,) in batch:
        write_csv_row(order)   # stable memory: ~500 live orders
# selectinload runs once per batch of 500,
# not once per order: 200 batches -> 400 total queries, not 100,001
```

For truly massive exports, the mature answer is usually to drop one level: a query with an explicit join returning flat tuples (no ORM objects, no identity map) or straight PostgreSQL COPY into the process writing the file. The ORM shines at managing objects with identity and lifecycle; an export is a row transformation, and treating it as such is both faster and simpler.

The held connection: lazy loading's invisible cost

There is a form of N+1 damage that shows up in no latency metric until it takes the service down: connection retention. While your handler iterates lazily — query, process, query, process — the connection stays checked out of the pool for the entire walk, including the CPU time between queries. With a pool of twenty connections and handlers holding theirs for two hundred milliseconds instead of ten, your real concurrency ceiling just got divided by twenty: new requests are not waiting on the database, they are waiting on the pool.

That is why the healthy pattern is load everything up front (a few dense queries, back to back), release the connection, and process afterwards with the data already in memory. Eager loading does not just reduce the query count: it compacts connection usage into one short burst at the start of the handler, which is exactly what a pool needs to multiplex well. With an open transaction the effect worsens: a transaction spanning the lazy walk keeps its locks and snapshot alive the whole way, inflating vacuum work and contention risk. The compound rule: short transactions, dense loads first, processing outside both.

Continuous observability: OpenTelemetry and the missing metric

OpenTelemetry's auto-instrumentation for SQLAlchemy and for Prisma creates one span per query, which makes N+1 visible in any tracing backend: the sick endpoint's trace shows a staircase of identical spans that no average-latency dashboard would ever reveal. It is worth adding a derived metric almost nobody exports and which is the most useful of all: queries per request, as a histogram labeled by route.

```
from opentelemetry.instrumentation.sqlalchemy import SQLAlchemyInstrumentor
from opentelemetry import metrics

SQLAlchemyInstrumentor().instrument(engine=engine)

meter = metrics.get_meter("app")
queries_per_request = meter.create_histogram(
    "db.queries_per_request",
    description="SQL queries emitted per HTTP request",
)

# In the middleware, when each request finishes:
queries_per_request.record(
    len(queries), {"http.route": route, "http.method": method}
)
```

With that metric, N+1 gets its own alert: a route whose db.queries_per_request p95 jumps from 4 to 60 after a deploy is an unambiguous regression, detectable in minutes and attributable to the exact commit — without waiting for latency to degrade enough for someone to open a ticket. It is the production-side version of the CI budget: the same contract, watched at the other end of the lifecycle.

Decision table: which strategy for which relationship

As a quick reference, the matrix that summarizes everything above. To-one relation on a hot path: joinedload (one cheap join, zero extra round trips). Medium collection (tens to hundreds): selectinload. Collection with a per-parent limit ("the latest 3"): window function or LATERAL with contains_eager; in Prisma, include with take. Huge collection with no limit: rethink the endpoint — it probably wants nested pagination or an export. Wide columns in listings: load_only or select/omit. Aggregates per parent: GROUP BY, column_property, or _count — never len() in Python. Async: all of the above plus lazy="raise" on the models, because lazy loading is not even an option. GraphQL: one DataLoader per relation, created per request. And in every case, raiseload("*") or its equivalent as the closer: whatever is undeclared should fail, not work slowly.

Extended production checklist

- **Detection:** pg_stat_statements enabled and reviewed (sorted by calls × mean), APM with N+1 detection turned on, and the db.queries_per_request metric exported with an alert on its per-route p95.
- **Prevention in code:** raiseload("*") (or lazy="raise") as the default policy in endpoint queries, explicit loading of every relationship the output schema declares, load_only on listings with wide columns.
- **Prevention in CI:** query budget per critical endpoint, tests run against realistically sized data (fifty rows minimum, not three), and budgets with margin so they do not break on noise.
- **Writes and jobs:** bulk updates expressed in SQL (update/ updateMany), inserts with executemany/ createMany, exports with yield_per + selectinload or straight COPY.

- **Architecture:** short transactions with dense loads up front, connections returned to the pool before heavy processing, per-request DataLoaders in GraphQL, and denormalization only where hot aggregation justifies it.

The through-line of this whole guide fits in one sentence: the ORM executes exactly what you ask for, and N+1 appears when you ask without realizing it. Making every load explicit — and making the implicit fail loudly — is the entire secret. The rest is measurement to know where to start, and tests so you never slide back.

Django as a mirror: the same patterns under other names

Although this guide centers on SQLAlchemy and Prisma, seeing the same concepts in Django helps cement them — and odds are you live with all three. select_related is Django's joinedload: a join for to-one relations, resolved in the same query. prefetch_related is selectinload: a second IN query for collections, joined in Python. The Prefetch object with a custom queryset covers the "latest 3 per parent" case and filtering the loaded collection. only() and defer() are load_only and defer almost literally. And assertNumQueries, built into the test framework forever, is the query budget we built by hand above:

```
# Django: same patterns, different syntax
orders = (
    Order.objects
    .select_related("customer")                # to-one: join
    .prefetch_related(
        Prefetch("lines",
                 queryset=Line.objects.select_related("product"))
    )
    .only("id", "total", "created_at", "customer__name")
)

class TestOrders(TestCase):
    def test_budget(self):
        with self.assertNumQueries(3):
            self.client.get("/orders/")
```

What Django does not ship out of the box is a raiseload-style guardrail; packages like django-zen-queries (blocks queries inside marked sections) or nplusone's monitoring fill the gap. In Rails, the bullet gem warns in development about every missing includes and every superfluous one — that last case, unnecessary eager loading, is the symmetric sin almost nobody discusses: loading relationships the view never uses also costs, just in bytes instead of round trips.

Anti-pattern catalog: the disguises of N+1

To close the technical part, the list of disguises N+1 wears past code review. The innocent helper: a format_order(order) method that internally touches order.customer.name — the loop is clean, the query lives three levels down the stack. The treacherous property: a model @property crossing a relationship, invoked from a template or serializer where nobody sees SQL. The sometimes-if: a conditional branch touching the relationship only for premium orders — the test with normal data passes, production with a real mix fires. The disguised count: if order.lines: to check whether lines exist, which loads the whole collection to evaluate a

boolean (the cheap form is an exists subquery or an aggregate count). The identity re-fetch: querying by id an object already in the session, multiplied by a loop. And the most modern one: the AI agent or job calling a get_detail(id) tool inside a loop over a hundred ids, reproducing over HTTP the very pattern this article describes in SQL — batching as a principle does not care about layers.

Glossary

Lazy loading: resolving a relationship at first attribute access, with an implicit query. **Eager loading:** declaring in the query which relationships to load, resolving them up front in one or few queries. **selectinload / prefetch_related / query strategy:** collection loading via one extra query per relation using IN over the parents' keys. **joinedload / select_related / join strategy:** loading in the same query through a join; ideal for to-one. **contains_eager:** populating a relationship from a join you wrote, instead of letting the ORM emit its own. **raiseload / lazy="raise":** turning every lazy access into an exception; the central guardrail of this guide. **Identity map:** per-session cache guaranteeing a single instance per row; deduplicates objects but does not prevent collection queries. **DataLoader:** per-request accumulator that groups individual loads into batches; the canonical defense in GraphQL. **Fan-out:** children per parent in a relationship; the multiplier that turns N+1 into N×M+1. **Query budget:** the maximum number of queries an endpoint may emit, enforced by a test.

Case study 2: the multi-tenant dashboard that degraded per large customer

A second real case, because the first was almost textbook and this one has the twisted shape N+1 usually arrives in. A B2B dashboard showed, for each of the tenant's projects, its members with avatars and its latest activity. Small tenants (three projects, ten members) flew; the largest customer (one hundred twenty projects, two thousand members) took eleven seconds and sometimes hit the timeout. The profile showed no slow query: it showed four thousand fast ones.

The cascade had three floors: projects → members (lazy), member → user (lazy, for the avatar), and a "latest activity" query per project executed inside a model property. The first two floors were solved with nested selectinload (two IN queries, independent of project count). The third was the interesting trap: latest activity per project is exactly the "top-1 per parent" pattern, so it ended up as a window function with row_number() over the activities table, materialized with contains_eager — one query, one hundred twenty rows. Eleven seconds became 240 ms without touching schema or UI, and the CI budget was pinned at five queries. The case's moral: N+1 scales with your largest customer's data, not with your test environment's — which is why the huge tenant suffered it and the demo tenant never showed it. Size your test data for your worst customer, not the average one.

Further reading

This problem touches borders that have their own guides on this site: if your listings grow, keyset pagination explains why OFFSET collapses and how to paginate with cursors (and combines naturally with the loading patterns here); the PostgreSQL indexing guide covers the other half of performance — making the few queries you do emit fast —; and the connection pooling guide with PgBouncer explains what happens to the connections lazy loading holds. N+1 rarely travels alone: an endpoint that suffers it usually also suffers OFFSET pagination and a missing index, and the three fixes reinforce each other.

Two more questions that always come up

raiseload in production, or only in development?

Both, with nuance. In development and tests, always: you want the error as early as possible. In production, the real question is what you prefer when an unanticipated access appears: an immediate, visible 500, or a silent degradation nobody may notice for months? For critical endpoints, the 500 with an alert is preferable — one request fails and you fix it today. For secondary routes, some teams use raiseload only in pre-production environments and accept the risk in production. What is indefensible is using it nowhere: it is the only tool that turns this entire class of bugs into deterministic errors.

Doesn't the identity map already protect me from repeated queries?

Only halfway, and it is a frequent misunderstanding. The identity map guarantees that the same row is the same Python object within a session, and avoids re-querying an entity already loaded when you ask for it by identity (session.get). But lazy access to a collection does not ask "do I have these objects?" — it asks "is this relationship loaded?" — and if it is not, it emits the query even if every resulting row was already in the session. Twenty orders from the same customer fire twenty lines queries even though they share a cached customer. The identity map deduplicates objects; it does not deduplicate relationship loads. The only defense against those is explicit loading.

New expansion: polymorphic inheritance, write-only collections and the N+1 that crosses services

This expansion covers three fronts that were still off the map. First, the N+1 that joined table inheritance manufactures, with its native antidote: selectin_polymorphic. Second, collections so large that the right answer is not to load them better but to never load them at all: SQLAlchemy 2.0's write-only relationships. Third, the version of the problem that emits no SQL whatsoever because it lives between HTTP services. It closes with two engineering details you will sooner or later run into: the .unique() call SQLAlchemy forces on you after a joinedload over collections, and what selectinload actually does under the hood when your parent rows number in the thousands.

The N+1 hiding in inheritance: selectin_polymorphic

Joined table inheritance splits each entity between a base table and one table per subtype: the common employee data in one table, and the columns specific to engineers and managers in their own. It is a clean design... until you query the base class and touch a subtype attribute. SQLAlchemy loads the base rows with one query and, on access to the first subtype-specific attribute, emits a SELECT against the subtype's table. For every row. It is the N+1 dressed up as polymorphism: nobody wrote a loop over relationships; someone just read an inherited attribute.

```
class Employee(Base):
    __tablename__ = "employees"
    id: Mapped[int] = mapped_column(primary_key=True)
    name: Mapped[str]
```

```
    kind: Mapped[str]
    __mapper_args__ = {"polymorphic_identity": "employee", "polymorphic_on": "kind"}

class Engineer(Employee):
    __tablename__ = "engineers"
    id: Mapped[int] = mapped_column(ForeignKey("employees.id"), primary_key=True)
    primary_language: Mapped[str]
    __mapper_args__ = {"polymorphic_identity": "engineer"}

# Query against the base class:
employees = session.scalars(select(Employee)).all()   # 1 query
for e in employees:
    if isinstance(e, Engineer):
        print(e.primary_language)   # 1 SELECT per engineer: N+1
```

The native fix is selectin_polymorphic(), a close cousin of selectinload: after loading the base rows, it emits one additional query per subtype with an IN over the primary keys, instead of one query per row. Two or three queries total, no matter how many employees there are:

```
from sqlalchemy.orm import selectin_polymorphic

stmt = select(Employee).options(
    selectin_polymorphic(Employee, [Engineer, Manager])
)
employees = session.scalars(stmt).all()
# Query 1: SELECT ... FROM employees
# Query 2: SELECT ... FROM engineers WHERE id IN (engineer keys)
# Query 3: SELECT ... FROM managers WHERE id IN (manager keys)
# Accessing subtype attributes no longer emits anything
```

If a subtype is always queried in full, you can pin the behavior on the mapping itself with polymorphic_load="selectin" and stop declaring it per query. The classic alternative, with_polymorphic, solves the same problem in a single wide query with joins to every subtype table; it pays with wider rows and one extra join per subtype. The criterion is the usual one: few subtypes with light columns, single join; many subtypes or heavy columns, selectin. And the guardrail exists in this territory too, though it is more artisanal: deferred columns accept raiseload (defer with raiseload=True), so that an unplanned access to an expensive column fails in development instead of silently emitting one query per row.

Collections that should never be fully loaded: WriteOnlyMapped

Some relationships cannot be saved by any eager loading, because the mistake is not how they are loaded but loading them at all: a user's events (hundreds of thousands), a sensor's readings, a channel's messages. For those, SQLAlchemy 2.0 has a structural answer: declare the relationship write-only. A WriteOnlyMapped relationship never loads its elements — neither lazily nor eagerly —; it offers add() and remove() for writing, and a .select() query builder for reading exactly the subset you need, with filters, ordering and a limit:

```
from sqlalchemy.orm import WriteOnlyMapped

class User(Base):
    __tablename__ = "users"
```

```
    id: Mapped[int] = mapped_column(primary_key=True)
    events: WriteOnlyMapped["Event"] = relationship(
        cascade="all, delete-orphan", passive_deletes=True
    )

# Write: without loading anything into memory
user.events.add(Event(kind="login"))

# Read: an explicit, bounded query — not "the collection"
latest = session.scalars(
    user.events.select()
    .order_by(Event.created_at.desc())
    .limit(20)
).all()
```

List-style access — user.events[0], len(user.events), iterating in a for loop — simply does not exist: it is not that it fails at runtime, it is that the type does not offer those operations, so the mistake shows up in your editor and in mypy before anything runs. It is the raiseload philosophy — the implicit must fail — applied one level earlier, in the type system. The signal for adopting it: any relationship whose size grows without bound as the product gets used. Bounded collections (an order's line items) remain selectinload territory; unbounded ones (a user's history) are write-only and get read with paginated queries — ideally keyset-based, as this site's pagination guide explains. If you come from SQLAlchemy 1.x: lazy="dynamic" did something similar; write_only is its typed successor, without the trap of someone accidentally iterating the "dynamic" relationship and loading the whole table.

The mandatory .unique(): what SQLAlchemy is telling you

If you have used joinedload over a collection in SQLAlchemy 2.0, you have hit this error: "The unique() method must be invoked on this Result". It is not bureaucracy: it is the row bloat bill from the first part of this guide, made explicit. The JOIN returns one row per child, so an order with eight line items appears eight times in the raw result; unique() deduplicates the parent objects using the identity map before handing them to you:

```
stmt = select(Order).options(joinedload(Order.items))
orders = session.execute(stmt).unique().scalars().all()
# Without .unique(): InvalidRequestError
# With .unique(): 50 orders, even if the JOIN returned 400 rows
```

SQLAlchemy could deduplicate silently — that is what 1.x's Query did — but it makes you write it so the cost stays visible: if you need to deduplicate, you are transferring repeated rows from the database. Every .unique() you type is an invitation to ask whether that collection should be loaded with selectinload instead. The symmetric note: selectinload never needs unique(), because its queries return flat rows without duplicating parents.

selectinload under the hood: batches of 500, expanded IN and PgBouncer

Knowing exactly what selectinload does explains two behaviors that are puzzling the first time. First: the batches. selectinload does not put ten thousand keys into a single IN; it chunks them into batches of 500 and emits one query per batch. For ten thousand parents that is twenty queries, not one — and it is still the right

call: the number of round trips grows with N/500 instead of N, and it avoids the giant INs that punish the parser and the planner. If your log shows several consecutive IN queries where you expected one, it is not a bug: it is the batching.

```
stmt = select(Customer).options(selectinload(Customer.orders))
customers = session.scalars(stmt).all()   # 1,200 customers
# SELECT ... FROM orders WHERE customer_id IN (...500 keys...)
# SELECT ... FROM orders WHERE customer_id IN (...500 keys...)
# SELECT ... FROM orders WHERE customer_id IN (...200 keys...)
# 4 queries total, not 1,201
```

Second: the query's shape. SQLAlchemy builds the IN with expanding parameters — the statement is rewritten with the exact number of placeholders on every execution — and that has downstream consequences. In pg_stat_statements, up to PostgreSQL 17 every distinct IN cardinality showed up as a separate entry: when auditing for N+1, sum the variants rather than looking only at the top row. PostgreSQL 18 finally squashes constant lists into a single entry (you will see IN ($1 /*, ... */)). With prepared statements the effect is the opposite: to the driver, each cardinality is a distinct statement to prepare. With asyncpg — whose prepared statement cache is keyed by query text — an endpoint with variable-size INs multiplies variants in the cache; if PgBouncer in transaction mode also sits in between, you need max_prepared_statements sized properly (PgBouncer 1.21+) or the driver's cache disabled. This site's connection pooling guide covers that swamp in detail.

The N+1 without SQL: HTTP fan-out between services

When the monolith gets split into services, the N+1 does not disappear: it changes protocol. The orders service needs each customer's name, and the loop "for each order, GET /customers/id" reproduces the pattern with latencies ten or twenty times worse — plus timeouts, retries and rate limits that SQL never had. The solution keeps the structure you already know from the ORM: gather keys, resolve in batch, index the result. Applied in two layers.

First layer: if the remote service offers a batch endpoint (GET /customers?ids=... or a POST with the list), use it — it is your selectinload between services. The same SQL batching rules apply: deduplicate before asking, chunk into reasonable groups (500 is once again a good number), and decide what to do with keys that do not come back. Second layer, when no batch endpoint exists: parallelize with a concurrency limit. A gather without a semaphore is not parallelism, it is a stampede against the neighboring service:

```
import asyncio
import httpx

async def load_customers(ids: set[int]) -> dict[int, dict]:
    limit = asyncio.Semaphore(10)   # max 10 requests in flight

    async def one(client, cid):
        async with limit:
            r = await client.get(f"/customers/{cid}")
            r.raise_for_status()
            return cid, r.json()
```

```
    async with httpx.AsyncClient(base_url=CUSTOMERS_URL, timeout=5.0) as client:
        pairs = await asyncio.gather(*(one(client, i) for i in ids))
    return dict(pairs)

# In the handler: gather keys, deduplicate, resolve ONCE
ids = {o.customer_id for o in orders}
customers = await load_customers(ids)
for o in orders:
    output.append(build_row(o, customers.get(o.customer_id)))
```

Notice how the semaphore turns "N sequential requests" into "N requests in waves of ten": with 50 customers at 40 ms per request, the sequential loop takes two seconds and the semaphore version about 240 ms. The batch endpoint still wins (one request, one round trip), and if you design the remote service yourself, offer one: accept up to a documented maximum of ids, return an id-to-object map and be explicit about what happens with nonexistent ids. The DataLoader pattern from the GraphQL section works here verbatim — accumulate keys per request and resolve them together — and it is the piece that makes a BFF efficient. And one architecture rule that saves the whole conversation: if a view needs data from three services for every row of a list, either a composition endpoint is missing (BFF, materialized view, read replica) or the service boundaries were drawn by the org chart rather than by the data.

August 2026 radar: where the pieces stand

To close, the current status of the tools mentioned, because several keep moving. In Prisma, relationLoadStrategy remains behind the relationJoins preview flag (PostgreSQL, CockroachDB and MySQL; SQL Server pending), and there is an important operational nuance: once the flag is on, join becomes the default strategy. Review your endpoints with large fan-out collections after enabling it, because the default stops behaving like selectinload. In SQLAlchemy, everything used in this expansion — write_only, selectin_polymorphic, polymorphic_load — is stable API in the 2.x series. In PostgreSQL, the squashing of IN lists in pg_stat_statements arrived with version 18: if you are still on 17 or earlier, remember to sum the per-cardinality variants when auditing. And APMs keep polishing automatic detection: the signature of "many identical spans under one request" is now grouped as a named problem by Sentry and the big vendors alike. None of this changes the bottom line: the tools detect better every year, but explicit loading — and making the implicit fail — is still your job.

August 2026 expansion: filtered loading, the planner, and N+1 in read architectures

So far we have treated N+1 as a counting problem: a thousand queries where there should be two. But once the fix reaches production, three fronts show up that almost nobody documents. The first is that filtering a collection and eager-loading it are different operations, and confusing them produces wrong data on top of extra queries. The second is that the PostgreSQL planner has its own opinion about your fix, and sometimes that opinion is worse than the original problem. The third is that the moment the database stops living on the same host, every query you save is worth far more milliseconds than your laptop suggests. This expansion covers all three.

contains_eager: when the JOIN you filter on must also populate the collection

This is the most expensive mistake on the list, because it does not show up as slowness — it shows up as incorrect data. Say you want users who have pending orders, and you want the response to contain only those pending orders. The intuitive version joins on the relationship, filters, and adds a selectinload so you do not fall back into N+1:

```
# WRONG: the filter and the load live in separate worlds
stmt = (
    select(User)
    .join(User.orders)
    .where(Order.status == "pending")
    .options(selectinload(User.orders))
)
```

The JOIN decides which users to return. The selectinload, however, fires a completely independent second query: *SELECT ... FROM orders WHERE orders.user_id IN (...)*, with no trace of the status filter. Result: every user arrives with all of their orders, including ones delivered two years ago. The query is fast, the list is wrong, and the bug survives code review because the *SELECT* looks perfectly reasonable.

The right tool is contains_eager(), which tells the ORM that the rows from the JOIN you already wrote are the ones that should populate the collection. No second query, and the filter is respected:

```
from sqlalchemy import select
from sqlalchemy.orm import contains_eager

stmt = (
    select(User)
    .join(User.orders)
    .where(Order.status == "pending")
    .options(contains_eager(User.orders))
    .execution_options(populate_existing=True)
)
users = session.scalars(stmt).unique().all()
# 1 query, and user.orders contains ONLY the pending ones
```

Three warnings are worth internalising before you reach for it. First, populate_existing=True: if those users were already in the session with the collection loaded, the identity map would win and you would see the stale collection; that option forces the overwrite. Second, the resulting collection is a partial view and it is not sticky: as soon as the object expires or you load it in another query, user.orders goes back to meaning *all orders*. Third, and most dangerous: never write through a partial collection. If the relationship has cascade="all, delete-orphan" and you call user.orders.remove(x) on a collection that only holds the pending ones, SQLAlchemy reasons about the set it can see, not the set that exists in the table. Treat contains_eager collections as strictly read-only.

If all you want is to filter the collection without letting the filter decide which parents come back, the lightweight option is to apply the criterion inside the loader itself:

```
# All users; for each one, only their pending orders
stmt = select(User).options(
```

```
    selectinload(User.orders.and_(Order.status == "pending"))
)
```

The mental rule: if the filter should shrink the list of parents, it goes in the *WHERE* with contains_eager; if it should only shrink the contents of the collection, it goes inside the loader's and_().

with_loader_criteria: the filter you cannot forget anywhere

Soft deletes and tenant isolation are the two places where loading optimisations rot over time. Someone fixes an N+1 by adding selectinload and, without noticing, skips the *deleted_at IS NULL* that the original lazy load applied from a helper. The bug is not about performance, it is about correctness, and it surfaces weeks later as *why are deleted orders showing up in the dashboard*.

with_loader_criteria() solves this by applying a condition to every occurrence of an entity inside a query, regardless of loading strategy or how deep it is loaded:

```
from sqlalchemy.orm import with_loader_criteria

stmt = (
    select(User)
    .options(selectinload(User.orders).selectinload(Order.items))
    .options(
        with_loader_criteria(Order, Order.deleted_at.is_(None), include_aliases=True),
        with_loader_criteria(Item, Item.deleted_at.is_(None), include_aliases=True),
    )
)
```

What makes it powerful is that you can promote it to a global policy with a session event, so no future query has to remember:

```
from sqlalchemy import event
from sqlalchemy.orm import Session, with_loader_criteria

@event.listens_for(Session, "do_orm_execute")
def _apply_soft_delete(state):
    if not state.is_select:
        return
    if state.execution_options.get("include_deleted", False):
        return
    state.statement = state.statement.options(
        with_loader_criteria(
            SoftDeleteMixin,
            lambda cls: cls.deleted_at.is_(None),
            include_aliases=True,
        )
    )
```

Two fine details. Using the lambda cls: ... form instead of a literal expression lets the criterion apply to any subclass of the mixin and lets SQLAlchemy cache it correctly per class. And include_aliases=True is not optional: without it, occurrences of the entity under an alias — which is exactly what joinedload generates — go unfiltered, and you get the bug back but only on the optimised paths, which is the worst possible way to

have it. The escape hatch is session.execute(stmt, execution_options={"include_deleted": True}) for the few call sites that genuinely need to see deleted rows.

The rest of the catalogue: subqueryload, immediateload and noload

Almost everyone knows selectinload and joinedload, and stops there. It is worth knowing what the other three do, if only to recognise them in legacy code.

subqueryload was the predecessor of selectinload. Instead of emitting an *IN* with the keys it already knows, it restates the original query as a subquery and joins against it. That means the planner re-plans the entire parent query on every relationship load, that it needs a deterministic *ORDER BY* to be correct alongside *LIMIT*, and that it cannot be combined with yield_per. If you find it in a repository, swap it for selectinload and measure: on large tables the difference is usually somewhere between 30 % and several multiples.

immediateload fires the lazy load immediately, one *SELECT* per object. It is still N+1 in query count, so it sounds useless, but it has a real niche: when you process with yield_per, or in contexts where the object is about to leave the session, it guarantees the relationship is populated before a lazy access can fail. It is the honest option for tiny collections inside a stream, not a performance strategy.

noload returns an empty collection without querying. It is tempting for lightening serialisation and it is an endless source of silent bugs: an empty list is indistinguishable from *there are no items*. Unless you are building objects for writing and you know nobody will read that relationship, prefer raiseload, which fails loudly instead of lying.

```
# Streaming 200k rows without blowing up memory or falling into lazy loads
stmt = (
   select(Order)
   .options(selectinload(Order.items), raiseload("*"))
   .execution_options(yield_per=1000)
)
for order in session.scalars(stmt):
   process(order)   # items already batch-loaded; any other relationship raises
```

What the planner does with your fix: nested loop, hash join and work_mem

This is where the conversation stops being about the ORM. When you trade a thousand queries for a *JOIN*, you are handing PostgreSQL a different problem, and the plan it picks determines whether the fix is a tenfold improvement or a regression under concurrency.

With joinedload, the planner has two reasonable paths. If the number of parents is small and there is an index on the foreign key, it will pick a *nested loop* with an *index scan*: fast, constant memory, predictable behaviour. If there are many parents, it will prefer a *hash join*, which builds an in-memory hash table of one of the relations. And that is where work_mem enters: if the hash table does not fit, PostgreSQL splits the operation into batches and spills them to disk. Your single query is still a single query, but now it does temporary I/O, and with twenty concurrent requests each claiming its own work_mem, the server degrades in a way that looks nothing like what you saw locally.

The way to see it is to look at the actual plan, not the estimate:

```
EXPLAIN (ANALYZE, BUFFERS, SETTINGS)
SELECT u.*, o.*
FROM users u
LEFT JOIN orders o ON o.user_id = u.id
WHERE u.tenant_id = 42;

-- Signals to look for in the output:
--   Hash Join  ... Batches: 9  Memory Usage: 4096kB  Disk Usage: 71232kB
--                ^ more than 1 batch = it spilled: raise work_mem or change strategy
--   Rows Removed by Join Filter: 1204338
--                ^ you are reading rows to throw them away: missing index or late filter
--   Buffers: shared hit=812 read=44190
--                ^ high read = not in cache: the real cost is I/O, not CPU
```

With selectinload the profile is different and, almost always, more boring in the good sense. The second query is a *WHERE user_id IN (...)* over a list of known keys, resolved with an *index scan* or a *bitmap heap scan*, needing no appreciable working memory. That predictability is the real reason selectinload is the default choice for collections: it is not that it always wins in the average case, it is that it is far harder for it to be catastrophic in the bad case.

Practical rule: if *EXPLAIN ANALYZE* on your joinedload shows more than one *batch* or a non-zero *Disk Usage*, do not reach for a higher work_mem first. Switch to selectinload and measure again; that usually solves it without touching a global setting that affects everything else.

The reverse anti-pattern: when fixing N+1 creates a monster query

Enthusiasm has a cost. Once someone discovers joinedload, the temptation is to chain it across every relationship in the aggregate until the endpoint makes exactly one query. The problem is that every joined collection multiplies the result rows. If a user has 50 orders and each order has 20 line items, joining both collections at once returns 1,000 rows *per user*, with the user's own columns repeated a thousand times over the wire. With 200 users on the page, that is 200,000 rows to rebuild 200 objects.

```
# Cartesian product: 200 x 50 x 20 = 200,000 rows transmitted
select(User).options(
   joinedload(User.orders).joinedload(Order.items)
)

# Linear amplification: 3 queries, ~11,000 rows total
select(User).options(
   selectinload(User.orders).selectinload(Order.items)
)
```

The heuristic is simple and holds up well: **at most one collection per joinedload in the same query**. Many-to-one relationships can be chained fearlessly because they do not multiply rows; collections are what does. As soon as there are two levels of collection, selectinload is the answer.

There is one metric worth instrumenting alongside the query count: *row amplification*, meaning rows returned by the database divided by objects built by the ORM. A value near one means you are transmitting exactly what you need. A value of 300 means you have traded a latency problem for a bandwidth and deduplication-CPU problem — that deduplication is what .unique() quietly does for you. Going from 1,000 queries to 1 while

pushing amplification from 1 to 500 is not automatically a win; measuring it saves you from finding out the hard way.

Read replicas and latency: why N+1 hurts three times more off your laptop

This is why an N+1 goes unnoticed in development and wrecks production, and it has nothing to do with data volume. It has to do with the round trip.

Locally, the database lives on the same host: every trivial query costs roughly 0.1–0.3 ms of round trip. Two hundred queries is 40 ms; nobody notices. In production, with the app and the replica in different availability zones, that same round trip becomes 1–3 ms; the same two hundred queries are now 200 to 600 ms of pure network time. If the replica sits in another region — increasingly common in multi-region deployments with local reads and remote writes — you are looking at 30–80 ms per query, and two hundred queries turn into a wait no user will tolerate.

The perverse part is how this looks on dashboards. Your database's *average query duration* metric will be spotless: each of those queries executes in 0.2 ms. The time goes into the network and into the session waiting, and that does not appear in server-side statistics. Hence the rule worth burning in: **the metric that predicts N+1 is not query duration, it is queries per request**. The first is a property of the database; the second is a property of your code, and it is the only one of the two that gets worse on its own.

```
# Latency budget, on the back of a napkin
# queries_per_request x rtt_ms = unavoidable latency floor
#
#   same host     200 x 0.2 ms  =   40 ms   (invisible)
#   same AZ       200 x 1.0 ms  =  200 ms   (annoying)
#   cross-AZ      200 x 3.0 ms  =  600 ms   (incident)
#   cross-region   200 x 45 ms   = 9000 ms   (timeout)
#
# The fix takes 200 -> 3. On the last row that is 9 s -> 135 ms.
```

There is a second, subtler effect if you sit behind PgBouncer in transaction pooling mode. Every lazy load that happens outside an explicit transaction can check a connection out of the pool and hand it back. Two hundred lazy loads are two hundred acquisition cycles, each with its own lock contention inside the pooler. Under load, N+1 stops being a problem with your request and becomes a problem with everyone else's, because your request is monopolising the pooler. That is the usual mechanism by which one slow endpoint takes down endpoints that have nothing to do with it.

Prisma: a client extension that shouts when N+1 appears

On the TypeScript side, Prisma client extensions let you intercept every operation without hand-wrapping the client. Combined with AsyncLocalStorage, you can count queries per request and make the process fail in development when the same model is queried suspiciously often:

```
import { AsyncLocalStorage } from "node:async_hooks";
import { PrismaClient } from "@prisma/client";

type QueryCounter = Map<string, number>;
```

```
export const requestScope = new AsyncLocalStorage<QueryCounter>();

const THRESHOLD = 10;

export const prisma = new PrismaClient().$extends({
 query: {
  $allModels: {
   async $allOperations({ model, operation, args, query }) {
    const counter = requestScope.getStore();
    if (counter) {
     const key = model + "." + operation;
     const n = (counter.get(key) ?? 0) + 1;
     counter.set(key, n);
     if (n === THRESHOLD + 1 && process.env.NODE_ENV !== "production") {
      throw new Error(
       "Possible N+1: " + key + " ran " + n + " times in a single request"
      );
     }
    }
    return query(args);
   },
  },
 },
});
```

```
// Middleware that opens the per-request scope (Express, Fastify or similar)
app.use((req, res, next) => {
 const counter: QueryCounter = new Map();
 requestScope.run(counter, () => {
  res.on("finish", () => {
   const total = [...counter.values()].reduce((a, b) => a + b, 0);
   logger.info({ path: req.path, queries: total, breakdown: Object.fromEntries(counter) });
  });
  next();
 });
});
```

This pattern has an advantage over traditional logging: it does not require you to read logs. In development it throws an exception naming the guilty model at the exact moment the pattern appears, and in production it emits a numeric field per request that you can alert on. It is the Prisma equivalent of the CI query budget we saw earlier, but running in real time.

On relationLoadStrategy, the underlying difference is worth restating because the naming confuses people: join resolves the relationship with a *LATERAL JOIN* on PostgreSQL and returns everything in one query, while query emits one query per table and joins them in the application, which is conceptually the same thing selectinload does. The intuition for choosing is identical to SQLAlchemy's: join for to-one relationships and small collections, query when the fan-out is large and the *JOIN* would start multiplying rows.

Keyset pagination and relationships: loading without ending up back at square one

One last pattern that combines two topics and is almost always implemented badly. When you paginate with a deep *OFFSET* and also load relationships, you pay twice: the database has to walk and discard every row before the offset, and the eager loading is applied on top of that wasted work. Keyset pagination removes the

first cost, and getting the order of operations right removes the second.

```
from sqlalchemy import select, tuple_
from sqlalchemy.orm import selectinload

def orders_page(session, cursor=None, limit=20):
    stmt = select(Order).order_by(Order.created_at.desc(), Order.id.desc()).limit(limit)
    if cursor is not None:
        created, ident = cursor
        stmt = stmt.where(tuple_(Order.created_at, Order.id) < (created, ident))

    # Narrow the page first; only then load the relationships
    stmt = stmt.options(selectinload(Order.items), selectinload(Order.customer))
    orders = session.scalars(stmt).all()

    next_cursor = (orders[-1].created_at, orders[-1].id) if orders else None
    return orders, next_cursor
```

The mental key is the ordering: *narrow, then load*. The tuple *WHERE* uses the composite index on (created_at, id) and jumps straight to the cut- off point regardless of how deep the page is, and the subsequent selectinload operates on exactly twenty keys. Page one thousand costs the same as page one, with the same number of queries. With *OFFSET*, page one thousand costs twenty thousand discarded rows before it even starts.

One correctness note that gets overlooked: the comparison must be on the tuple, not on separate columns chained with *AND*. If two orders share created_at down to the millisecond — which happens the moment you have batch imports — comparing on the date alone either skips rows or repeats them. The (created_at, id) tuple imposes a total order and makes pagination deterministic.

What to measure tomorrow morning

If you take only three things away from this expansion, make them these. One: instrument *queries per request* as a first-class metric alongside latency, because it is the only one that catches N+1 before it becomes an incident. Two: every time you replace an N+1 with a *JOIN*, look at *EXPLAIN (ANALYZE, BUFFERS)* on the result and confirm you have not traded a thousand cheap queries for one query that spills to disk. And three: when you filter a collection, decide deliberately whether the filter narrows the parents — contains_eager — or only the contents — and_() inside the loader — because that distinction is the difference between a fast endpoint and an endpoint that returns data that simply is not true.