PUIGFLOWS · RECURSO

Paginación por Keyset en PostgreSQL: Por Qué OFFSET No Escala (y Cómo Solucionarlo)

Aprende por qué la paginación con LIMIT/OFFSET colapsa en tablas grandes y cómo reemplazarla por paginación por keyset (cursor): comparación de tuplas, el índice compuesto correcto, cursores opacos para tu API, errores comunes y cuándo usar cada enfoque. Con código listo para producción en SQL y Python.

Guía · Intermedio · Josué Puig

Tiempo de lectura: ~35 min · Lectura premium

Español

Casi todos aprendimos a paginar con `LIMIT` y `OFFSET`. Es intuitivo, se lee bien y en una tabla chica funciona sin que nadie note nada raro. El problema es que ese enfoque se degrada justo cuando más lo necesitás: cuando la tabla crece y los usuarios navegan hacia el fondo. En esta guía vas a ver por qué OFFSET no escala, cómo funciona la paginación por keyset (también llamada paginación por cursor), cómo implementarla bien con el índice correcto y una API limpia, y cómo llevarla a producción: NULLs, órdenes mixtos, cursores firmados, conteos totales, integración con GraphQL y ORMs, scroll infinito en el frontend, pruebas y migración desde OFFSET sin romper a nadie.

La idea central cabe en una frase: **en lugar de pedirle a la base que cuente y descarte filas, le damos un punto de referencia para que salte directo a donde quedó.** Ese cambio de enfoque convierte una consulta que se vuelve más lenta cuanto más profundo navegás en una que tarda prácticamente lo mismo en la página 2 que en la 50.000.

El problema con OFFSET

Para hacerlo concreto: pedir la página 5.000 con OFFSET se ve así.

```
SELECT id, titulo, creado_en
FROM articulos
ORDER BY creado_en DESC
LIMIT 20 OFFSET 99980;
```

PostgreSQL no salta mágicamente a la fila 99.981. Recorre desde el principio las 99.980 filas anteriores, las descarta una por una y recién entonces devuelve las 20 que pediste. Cuanto más profundo navegás, más trabajo hace el motor. En tablas de millones de filas, esto convierte consultas de milisegundos en consultas de segundos. El costo es lineal respecto al offset: la página 1 es gratis, la página 10.000 te cobra haber leído todo lo anterior.

Hay un segundo problema, menos visible pero más traicionero: la **correctitud**. Si alguien inserta o elimina una fila mientras el usuario pasa de la página 1 a la 2, todo el conteo se corre. Una fila que estaba en la posición 21 pasa a la 20 y nunca aparece en la página siguiente; el usuario simplemente no la ve. Al revés, si se borra una fila, otra se repite en dos páginas. Con datos que cambian seguido, OFFSET te miente, y lo peor es que miente en silencio: nadie ve un error, solo faltan o sobran registros.

Midiendo el costo real con EXPLAIN ANALYZE

No te creas nada de lo anterior por fe; medilo. La herramienta es `EXPLAIN (ANALYZE, BUFFERS)`, que ejecuta la consulta de verdad y te muestra el plan, los tiempos y -clave- cuántos bloques de disco y caché tuvo que tocar.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, titulo, creado_en
FROM articulos
ORDER BY creado_en DESC
LIMIT 20 OFFSET 99980;
```

En una consulta con OFFSET profundo vas a ver algo como `Limit ... rows=20` sobre un nodo que reporta haber procesado ~100.000 filas, con un número de `shared hit/read` (los bloques leídos) enorme. Esa es la prueba material del problema: el motor tocó cien mil filas para devolverte veinte. La línea `Buffers` es la que más mira un DBA con experiencia, porque el trabajo de E/S es lo que no escala.

Como referencia mental aproximada, sobre una tabla de unos pocos millones de filas bien indexada, OFFSET 20 responde en un puñado de milisegundos, mientras que OFFSET 100.000 puede irse a cientos de milisegundos o más, y sigue creciendo. El keyset, en cambio, se mantiene en ese puñado de

milisegundos sin importar la profundidad. La diferencia no es de constante: es de forma de la curva.

Cómo funciona la paginación por keyset

La idea es darle la vuelta a la pregunta. En lugar de "saltá 99.980 filas y dame las siguientes 20", le decimos a la base "dame las 20 filas que vienen justo después de la última que ya mostré". Para eso usamos los valores de esa última fila como punto de referencia: el cursor.

```
-- Primera página
SELECT id, titulo, creado_en
FROM articulos
ORDER BY creado_en DESC, id DESC
LIMIT 20;

-- Páginas siguientes: pasamos el (creado_en, id) de la última fila vista
SELECT id, titulo, creado_en
FROM articulos
WHERE (creado_en, id) < ('2026-06-20 10:30:00', 8842)
ORDER BY creado_en DESC, id DESC
LIMIT 20;
```

La clave está en la comparación de tuplas `(creado_en, id) < (...)`. PostgreSQL la evalúa de forma nativa: primero compara `creado_en` y, ante un empate, desempata por `id`. Con el índice adecuado, el motor salta directo a la posición del cursor sin recorrer filas de más. El tiempo de respuesta se mantiene casi constante, estés en la página 2 o en la 50.000.

La comparación de tuplas, en profundidad

Vale la pena entender qué hace exactamente esa comparación de fila (row-value comparison del estándar SQL), porque es el corazón de todo. La expresión `(a, b) < (x, y)` NO significa "a < x Y b < y". Significa, en orden lexicográfico, lo mismo que:

```
WHERE creado_en < '2026-06-20 10:30:00'
   OR (creado_en = '2026-06-20 10:30:00' AND id < 8842)
```

Es decir: traé todo lo que sea estrictamente anterior por fecha, y para el caso del empate exacto de fecha, desempatá por id. PostgreSQL entiende la forma de tupla directamente y, lo más importante, sabe traducirla en un acceso eficiente al índice compuesto. Esa traducción es la diferencia entre un `Index Scan` que arranca justo en el cursor y un escaneo que filtra fila por fila.

¿Por qué no escribir el `OR` a mano y listo? Porque históricamente algunos planificadores no aprovechaban bien la forma con `OR` para saltar dentro del índice, y terminaban recorriendo de más. La forma de tupla es más clara para el optimizador y para quien lee la consulta. Dicho eso, vas a necesitar la versión con `OR` sí o sí en cuanto mezcles direcciones de orden (una columna ASC y otra DESC), porque ahí la tupla simple deja de representar lo que querés. Lo vemos más abajo.

Un detalle que confunde a mucha gente: la comparación es entre *tuplas completas*, no columna por columna de forma independiente. `(2, 100) < (3, 1)` es verdadero aunque 100 > 1, porque el primer elemento ya decide. Pensalo como ordenar palabras en un diccionario: primero la primera letra; solo si empata, mirás la segunda.

Por qué necesitás un desempate único

Ordenar solo por `creado_en` no alcanza. Si dos artículos comparten el mismo timestamp, su orden relativo queda indefinido, y ahí es donde empezás a saltarte o repetir filas entre páginas. Por eso agregamos `id` como segundo criterio. La regla general: las columnas del `ORDER BY`, en conjunto, tienen que ser únicas. En la práctica eso casi siempre significa terminar la lista con la clave primaria. No importa si la columna principal "casi nunca" repite: con suficiente tráfico, dos filas van a caer en el mismo milisegundo, y el día que pase vas a perder una fila sin enterarte.

El índice que lo hace rápido

El keyset sin un índice que lo respalde no sirve de nada: volvés a un recorrido secuencial. Necesitás un índice compuesto que coincida con tu `ORDER BY`, incluida la dirección de cada columna.

```
CREATE INDEX idx_articulos_keyset
ON articulos (creado_en DESC, id DESC);
```

Después, confirmá que el plan realmente lo usa con `EXPLAIN ANALYZE`. Querés ver un `Index Scan` (o `Index Only Scan`) y un tiempo de ejecución que no crece con la profundidad; si aparece un `Seq Scan` o un `Sort` costoso, algo no coincide entre el índice y la consulta.

```
EXPLAIN ANALYZE
SELECT id, titulo, creado_en
FROM articulos
WHERE (creado_en, id) < ('2026-06-20 10:30:00', 8842)
ORDER BY creado_en DESC, id DESC
LIMIT 20;
```

Index-only scans e índices de cobertura

Si tu consulta solo necesita columnas que están en el índice, PostgreSQL puede resolverla sin tocar la tabla: es un `Index Only Scan`, el plan más rápido posible. El problema es que normalmente querés devolver más columnas que las del orden (un título, un resumen). La solución es un **índice de cobertura** con la cláusula `INCLUDE`: las columnas de orden van en la clave y las que solo necesitás leer van en el payload.

```
CREATE INDEX idx_articulos_keyset_cover
ON articulos (creado_en DESC, id DESC)
INCLUDE (titulo);
```

Con esto, la consulta que pide `id, titulo, creado_en` puede resolverse íntegra desde el índice. Cuidado: agregar columnas con `INCLUDE` engorda el índice y encarece las escrituras, así que incluí solo lo que de verdad devolvés en el listado, no la fila entera. Para tablas muy anchas, más abajo vemos una alternativa mejor: el deferred join.

Un requisito a menudo olvidado para los index-only scans: la tabla tiene que estar razonablemente "visible" para el visibility map, lo que en la práctica significa un `VACUUM` al día. Si ves muchos `Heap Fetches` en el plan, es señal de que el autovacuum se está quedando atrás.

PostgreSQL 18: skip scan y qué cambia para el keyset

PostgreSQL 18 (lanzado en 2025) trajo el **skip scan** para índices B-tree multicolumna: ahora el planificador puede usar un índice compuesto aunque la consulta no filtre por la primera columna, siempre que esa columna tenga pocos valores distintos. El motor recorre internamente los valores distintos del prefijo y baja al índice para cada uno.

¿Qué tiene que ver con el keyset? Dos cosas. Primero, te da más libertad al diseñar índices: un mismo índice compuesto puede servir para más patrones de consulta de los que servía antes. Segundo, y más importante, **no cambia la regla de oro**: para que el keyset sea rápido, el índice tiene que coincidir con el `ORDER BY` en columnas y dirección. El skip scan ayuda con filtros de igualdad sobre prefijos de baja cardinalidad (por ejemplo, paginar dentro de un `estado` con tres valores posibles), no reemplaza al índice ordenado que el cursor necesita para saltar. Pensalo como una herramienta complementaria, no como un permiso para ignorar el orden del índice.

Órdenes mixtos: cuando ASC y DESC rompen la tupla

La comparación de tuplas `(a, b) < (x, y)` asume que todas las columnas van en la misma dirección. En cuanto necesitás ordenar por una columna ascendente y otra descendente -por ejemplo, precio ascendente y, ante empate, id descendente- la tupla simple deja de representar lo que querés y tenés que descomponer

la condición a mano.

```
-- Orden deseado: precio ASC, id DESC
-- La forma de tupla NO sirve acá. Descomponemos:
SELECT id, nombre, precio
FROM productos
WHERE precio > 49.90
   OR (precio = 49.90 AND id < 8842)
ORDER BY precio ASC, id DESC
LIMIT 20;
```

Fijate el patrón: para la columna ASC usamos `>` (queremos lo que viene después, hacia arriba), y para el desempate DESC usamos `<`. El índice tiene que reflejar exactamente esas direcciones para que el plan sea eficiente:

```
CREATE INDEX idx_productos_precio
ON productos (precio ASC, id DESC);
```

Si tenés tres o más columnas de orden con direcciones mezcladas, la condición crece en una cascada de `OR` anidados. No es difícil, pero es fácil equivocarse a mano, así que conviene generar la cláusula desde código a partir de la lista de columnas de orden, o apoyarte en una librería que ya lo haga (lo vemos en la sección de ORMs).

Manejo de valores NULL

Los NULL son la fuente número uno de bugs sutiles en keyset. En PostgreSQL, por defecto, `ORDER BY ... DESC` pone los NULL primero (`NULLS FIRST`) y `ASC` los pone al final (`NULLS LAST`). El problema es que cualquier comparación contra `NULL` con `<` o `>` devuelve `NULL` (ni verdadero ni falso), así que tu condición de cursor descarta filas que no debería.

La regla práctica: paginá siempre por columnas `NOT NULL` cuando puedas. `creado_en` y la clave primaria casi siempre lo son, y por eso son cursores ideales. Si tenés que ordenar por una columna que admite NULL (por ejemplo, `publicado_en`, que es NULL para borradores), fijá explícitamente el orden de los NULL y maneja el cruce entre el bloque no-NULL y el bloque NULL.

```
-- Orden: publicado_en DESC NULLS LAST, id DESC
-- Cursor sobre una fila con publicado_en NO nulo:
SELECT id, titulo, publicado_en
FROM articulos
WHERE (publicado_en IS NOT NULL AND publicado_en < :cur_pub)
   OR (publicado_en IS NOT NULL AND publicado_en = :cur_pub AND id < :cur_id)
   OR (publicado_en IS NULL)              -- el bloque de NULLs viene después
ORDER BY publicado_en DESC NULLS LAST, id DESC
LIMIT 20;
```

Cuando el cursor ya entró en el bloque de NULLs (la fila de referencia tiene `publicado_en IS NULL`), la condición se simplifica a "filas con publicado_en NULL e id menor al del cursor". Es manejable, pero suma complejidad real, y es la razón por la que conviene evitar columnas nullable en el orden siempre que se pueda. El índice también debe declarar el mismo `NULLS LAST` para que el plan lo aproveche.

Paginación bidireccional: anterior y siguiente

Hasta acá vimos avanzar hacia "siguiente". Para soportar "anterior" se invierte la lógica: se da vuelta el operador de comparación y la dirección del `ORDER BY`, y al final se reordena el resultado para devolverlo en el orden natural.

```
-- Página ANTERIOR respecto del cursor (creado_en, id)
SELECT * FROM (
  SELECT id, titulo, creado_en
  FROM articulos
```

```
  WHERE (creado_en, id) > ('2026-06-20 10:30:00', 8842)
  ORDER BY creado_en ASC, id ASC   -- invertido
  LIMIT 20
) sub
ORDER BY creado_en DESC, id DESC;   -- reordenamos al orden natural
```

El subselect trae las 20 filas inmediatamente anteriores en orden invertido (las más cercanas al cursor primero), y el select externo las devuelve en el orden que el usuario espera. Con esto podés ofrecer botones de "anterior" y "siguiente" sin volver nunca a OFFSET. Para saber si hay página anterior o siguiente, aplicá el mismo truco de pedir una fila de más (`LIMIT 21`) en cada dirección.

Cursores opacos para tu API

Exponer `creado_en` e `id` crudos en la URL funciona, pero acopla tu API al esquema de la base de datos. El día que quieras cambiar la columna de orden, rompés a todos los clientes que guardaron esos cursores. La práctica habitual es codificar el cursor (por ejemplo en Base64) y tratarlo como un token opaco que el cliente solo devuelve sin interpretar.

```
import base64, json

def encode_cursor(creado_en: str, id: int) -> str:
    payload = json.dumps({"c": creado_en, "i": id})
    return base64.urlsafe_b64encode(payload.encode()).decode()

def decode_cursor(cursor: str) -> tuple[str, int]:
    payload = json.loads(base64.urlsafe_b64decode(cursor.encode()))
    return payload["c"], payload["i"]
```

Con eso, un endpoint de listado queda simple y consistente. Fijate el detalle de pedir una fila de más (`limite + 1`): nos sirve para saber si hay página siguiente sin ejecutar un `COUNT` aparte.

```
@app.get("/articulos")
def listar(cursor: str | None = None, limite: int = 20):
    limite = min(limite, 100)  # techo defensivo: nunca confíes en el cliente
    if cursor:
        creado_en, last_id = decode_cursor(cursor)
        filas = db.fetch(
            """SELECT id, titulo, creado_en FROM articulos
               WHERE (creado_en, id) < (%s, %s)
               ORDER BY creado_en DESC, id DESC
               LIMIT %s""",
            (creado_en, last_id, limite + 1),
        )
    else:
        filas = db.fetch(
            """SELECT id, titulo, creado_en FROM articulos
               ORDER BY creado_en DESC, id DESC
               LIMIT %s""",
            (limite + 1,),
        )

    hay_mas = len(filas) > limite
    filas = filas[:limite]
    siguiente = encode_cursor(filas[-1]["creado_en"], filas[-1]["id"]) if hay_mas else None
    return {"datos": filas, "next_cursor": siguiente}
```

El cliente no necesita saber nada del interior del cursor: pide `/articulos`, recibe los datos y un `next_cursor`, y para la página siguiente vuelve a llamar con ese valor. Cuando `next_cursor` es null, llegó al final.

Cursores firmados y versionados

Base64 no es seguridad: cualquiera puede decodificarlo y editarlo. Para la mayoría de los listados eso no

importa, pero si el cursor pudiera usarse para inferir o forzar datos sensibles, firmalo con HMAC y verificá la firma al recibirlo. Así detectás cualquier manipulación.

```
import base64, hmac, hashlib, json

SECRET = b"clave-larga-y-secreta-de-entorno"

def encode_cursor(payload: dict) -> str:
    body = json.dumps({"v": 1, **payload}, separators=(",", ":")).encode()
    sig = hmac.new(SECRET, body, hashlib.sha256).digest()[:16]
    return base64.urlsafe_b64encode(body + b"." + sig).decode()

def decode_cursor(cursor: str) -> dict:
    raw = base64.urlsafe_b64decode(cursor.encode())
    body, sig = raw.rsplit(b".", 1)
    expected = hmac.new(SECRET, body, hashlib.sha256).digest()[:16]
    if not hmac.compare_digest(sig, expected):
        raise ValueError("cursor inválido")
    return json.loads(body)
```

Dos detalles que valen oro en producción. El campo `"v": 1` versiona el formato del cursor: el día que cambies la columna de orden, incrementás la versión y podés rechazar (o migrar) cursores viejos en lugar de devolver resultados incorrectos en silencio. Y la comparación de la firma usa `hmac.compare_digest`, que es de tiempo constante, para no filtrar información por el tiempo de respuesta. Ante un cursor inválido o manipulado, respondé un `400` claro en vez de tragarte el error.

Keyset con filtros y búsqueda

En la vida real casi nunca paginás "toda la tabla": filtrás por categoría, por autor, por estado, o combinás con una búsqueda de texto. El keyset sigue funcionando, pero el índice tiene que poner las columnas de filtro por igualdad **antes** de las columnas de orden.

```
-- Listar artículos de una categoría, paginados
SELECT id, titulo, creado_en
FROM articulos
WHERE categoria_id = 7
  AND (creado_en, id) < ('2026-06-20 10:30:00', 8842)
ORDER BY creado_en DESC, id DESC
LIMIT 20;

-- El índice correcto pone la igualdad primero:
CREATE INDEX idx_articulos_cat_keyset
ON articulos (categoria_id, creado_en DESC, id DESC);
```

La regla mnemotécnica clásica para índices compuestos es **"igualdad, orden, rango"**: primero las columnas que filtrás con `=`, después las del `ORDER BY`, y al final cualquier rango. Si filtrás por varios valores posibles de categoría, acordate de que PostgreSQL 18 con skip scan puede ayudarte cuando esa columna tiene baja cardinalidad. Para búsqueda de texto, podés combinar un filtro por `tsvector` con keyset sobre un campo de ranking estable más el id, pero ojo: si ordenás por relevancia (un score calculado), ese valor tiene que ser determinista y estar materializado, no recalculado en cada consulta, o el cursor deja de tener sentido entre páginas.

El problema del conteo total y "página X de Y"

El keyset tiene un costo de diseño honesto: no te da gratis el total de filas ni el número de página. Eso es justamente lo que lo hace rápido (no cuenta lo que no necesita), pero a veces el producto pide un "1.234 resultados" o un "página 3 de 50". Tenés varias salidas, de menor a mayor exactitud.

- **No mostrar el total.** La opción más honesta para feeds y scroll infinito: nadie en Twitter espera ver "tweet 4.500 de 12.000". Mostrá "siguiente/anterior" y listo.

- **Conteo aproximado.** Para "alrededor de 12 mil resultados", usá la estimación del planificador, que es instantánea porque sale de las estadísticas.
- **Conteo exacto cacheado.** Si de verdad necesitás el número exacto, calculalo aparte (un job, un contador incremental con triggers, o un `COUNT` cacheado por unos minutos) y no en cada request.

```
-- Estimación instantánea del total de una consulta filtrada
SELECT reltuples::bigint AS estimado
FROM pg_class
WHERE relname = 'articulos';

-- Estimación más fina, respetando el filtro, leyendo el plan:
EXPLAIN (FORMAT JSON)
SELECT 1 FROM articulos WHERE categoria_id = 7;
-- el campo "Plan Rows" del JSON es la estimación de filas
```

La idea de fondo: separá la navegación (rápida, por keyset) de la métrica de volumen (cara, aproximada o diferida). Mezclarlas en la misma consulta es lo que te devuelve al problema de OFFSET.

Deferred join para tablas anchas

Cuando la tabla tiene muchas columnas o columnas grandes (texto largo, JSON), incluso un Index Scan puede ser lento porque, por cada fila, el motor va a la tabla a buscar todo el resto del registro. El patrón **deferred join** (o "late row lookup") resuelve esto en dos pasos: primero usás el índice para traer solo los ids de la página -rapidísimo, casi siempre index-only- y después hacés un join con la tabla para hidratar únicamente esas 20 filas.

```
SELECT a.*
FROM articulos a
JOIN (
  SELECT id
  FROM articulos
  WHERE (creado_en, id) < ('2026-06-20 10:30:00', 8842)
  ORDER BY creado_en DESC, id DESC
  LIMIT 20
) pagina USING (id)
ORDER BY a.creado_en DESC, a.id DESC;
```

El subselect interno toca solo el índice y devuelve 20 ids; el join externo lee de la tabla apenas esas 20 filas completas. La diferencia se nota cuando las filas son anchas: pasás de leer páginas de disco de filas gordas que vas a descartar, a leer exactamente lo que devolvés. Es la misma idea que el índice de cobertura, pero sin tener que duplicar columnas dentro del índice.

Integración con GraphQL: conexiones estilo Relay

Si exponés una API GraphQL, lo más probable es que sigas la especificación de Cursor Connections de Relay, que es keyset con otro vestido. El esquema define `edges` (cada uno con un `node` y su `cursor`) y un `pageInfo` con `hasNextPage`, `hasPreviousPage`, `startCursor` y `endCursor`. Los argumentos `first`/`after` paginan hacia adelante y `last`/`before` hacia atrás.

```
type ArticuloConnection {
  edges: [ArticuloEdge!]!
  pageInfo: PageInfo!
}
type ArticuloEdge {
  node: Articulo!
  cursor: String!
}
type PageInfo {
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
```

```
  startCursor: String
  endCursor: String
}
type Query {
  articulos(first: Int, after: String, last: Int, before: String): ArticuloConnection!
}
```

El truco mental es que el `cursor` de Relay es exactamente tu cursor de keyset codificado, y `after` es el valor que va en tu cláusula `WHERE (creado_en, id) < ...`. La especificación es deliberadamente agnóstica de la base de datos, pero recomienda keyset justamente porque las cláusulas `WHERE` aprovechan los índices y OFFSET no. Implementás el resolver con la misma consulta de keyset que ya tenés, calculás `hasNextPage` con el truco de la fila extra, y serializás cada cursor con tu `encode_cursor`.

Implementación con ORMs

No siempre escribís SQL a mano. Estos son los patrones en los ORMs más comunes; el principio es idéntico en todos: condición de cursor + orden estable + límite, con un índice que coincida.

SQLAlchemy

Podés expresar la comparación de tuplas con `tuple_`, o apoyarte en la librería `sqlakeyset`, que maneja por vos los cursores, las direcciones mixtas y los NULL.

```
from sqlalchemy import tuple_, select

# Manual: comparación de tuplas nativa
stmt = (
    select(Articulo)
    .where(tuple_(Articulo.creado_en, Articulo.id) < (cur_creado, cur_id))
    .order_by(Articulo.creado_en.desc(), Articulo.id.desc())
    .limit(20)
)

# Con sqlakeyset: cursores opacos listos para usar
from sqlakeyset import select_page
q = select(Articulo).order_by(Articulo.creado_en.desc(), Articulo.id.desc())
page = select_page(session, q, per_page=20, page=cursor)  # cursor opaco
siguiente = page.paging.bookmark_next if page.paging.has_next else None
```

Django

El ORM de Django no tiene comparación de tuplas directa, así que se arma la condición con objetos `Q`. Para casos serios, la librería `django-cursor-pagination` encapsula todo esto.

```
from django.db.models import Q

qs = (
    Articulo.objects
    .filter(Q(creado_en__lt=cur_creado) |
            Q(creado_en=cur_creado, id__lt=cur_id))
    .order_by("-creado_en", "-id")[:20]
)
```

Prisma

Prisma trae paginación por cursor de fábrica con `cursor` + `take`. Un detalle importante de su API: el cursor es *inclusivo*, así que se usa `skip: 1` para no repetir la fila de referencia.

```
const pagina = await prisma.articulo.findMany({
  take: 20,
  skip: 1,                       // saltea la fila del cursor (es inclusivo)
  cursor: { id: ultimoId },      // cursor por clave única
  orderBy: [{ creadoEn: 'desc' }, { id: 'desc' }],
});
```

Una advertencia sobre Prisma y motores distribuidos: en algunos backends su helper de cursor puede generar planes que no aprovechan el índice como esperás. Como siempre, verificá el SQL real que emite tu ORM y pasalo por `EXPLAIN ANALYZE`; ninguna abstracción te exime de mirar el plan.

Scroll infinito en el frontend

Del lado del cliente, el next_cursor convierte el scroll infinito en algo trivial: guardás el cursor, y cuando el usuario llega al final pedís la página siguiente y la concatenás. Un ejemplo con React, sin dependencias:

```
function useArticulos() {
  const [items, setItems] = React.useState([]);
  const [cursor, setCursor] = React.useState(null);
  const [hayMas, setHayMas] = React.useState(true);
  const [cargando, setCargando] = React.useState(false);

  async function cargarMas() {
    if (cargando || !hayMas) return;
    setCargando(true);
    const url = "/articulos?limite=20" + (cursor ? "&cursor=" + encodeURIComponent(cursor) :
"");
    const res = await fetch(url).then(r => r.json());
    setItems(prev => [...prev, ...res.datos]);
    setCursor(res.next_cursor);
    setHayMas(res.next_cursor !== null);
    setCargando(false);
  }

  return { items, cargarMas, hayMas, cargando };
}
```

Combinalo con un `IntersectionObserver` sobre un elemento centinela al final de la lista para disparar `cargarMas()` automáticamente. Como cada página se ancla en el contenido real (no en un número de offset), el scroll infinito no se saltea ni repite elementos aunque entren publicaciones nuevas mientras el usuario lee: la base de los problemas de "feeds que parpadean" suele ser, justamente, OFFSET por debajo.

Pruebas: garantizar que no se saltan ni repiten filas

La mejor prueba para una paginación es una de propiedad: recorré todas las páginas hasta el final y verificá que el conjunto resultante es exactamente la tabla, sin huecos ni duplicados. Esto atrapa los bugs clásicos (desempate olvidado, cursor inclusivo, dirección invertida) que en una sola página no se notan.

```
def test_paginacion_recorre_todo(client, seed_500_articulos):
    vistos = []
    cursor = None
    while True:
        url = "/articulos?limite=37"  # tamaño "feo" a propósito
        if cursor:
            url += "&cursor=" + cursor
        r = client.get(url).json()
        vistos.extend(item["id"] for item in r["datos"])
        cursor = r["next_cursor"]
        if cursor is None:
            break

    ids_db = [a.id for a in Articulo.query.order_by(
        Articulo.creado_en.desc(), Articulo.id.desc()).all()]

    assert vistos == ids_db                # mismo orden, sin huecos
    assert len(vistos) == len(set(vistos))  # cero duplicados
```

Un par de pruebas extra que vale la pena tener: una con timestamps duplicados a propósito (para verificar el desempate), una con la tabla mutando entre páginas (insertás filas en el medio del recorrido y confirmás que

las ya vistas no se repiten), y una de cursor manipulado (que devuelva 400, no 500). El tamaño de página "feo" como 37 ayuda a destapar errores de borde que un múltiplo redondo esconde.

Observabilidad y operación en producción

Una vez en producción, lo que querés vigilar cambia respecto de OFFSET. La gran ventaja del keyset es que la latencia deja de depender de la profundidad, así que el primer indicador de salud es que tu p95 y p99 del endpoint de listado sean planos sin importar cuán lejos navegue la gente. Si ves la latencia crecer con la profundidad, algo volvió a un Seq Scan: revisá que el índice siga existiendo y coincidiendo con el orden.

- **Poné un techo al tamaño de página.** `limite = min(limite, 100)`. Sin techo, un cliente pidiendo `limite=100000` te reintroduce el problema que viniste a resolver.
- **Logueá cursores inválidos.** Un pico de 400 por cursores rotos suele indicar un cambio de formato no versionado o un cliente viejo; el campo de versión del cursor te dice exactamente qué pasó.
- **Vigilá los Heap Fetches.** Si dependés de index-only scans y el autovacuum se atrasa, la latencia sube de a poco. Un dashboard del lag de autovacuum sobre las tablas calientes te avisa antes que el usuario.
- **Revisá los planes tras cada migración.** Un `ALTER` de tipo de columna o un cambio de collation puede invalidar en silencio el uso del índice. Tené un test de regresión que falle si el plan deja de ser un Index Scan.

Migrar desde OFFSET sin romper clientes

Si ya tenés una API en producción con `?page=N`, no podés cambiar el contrato de un día para el otro. La migración se hace en capas. Primero, agregá el índice compuesto correcto; no rompe nada y ya mejora algunas consultas. Segundo, soportá ambos esquemas a la vez: si llega `cursor`, usás keyset; si llega `page`, mantenés OFFSET. Tercero, empezá a devolver `next_cursor` en todas las respuestas para que los clientes nuevos lo adopten. Cuarto, deprecá `page` con aviso y métricas de uso, y recién cuando el tráfico por `page` sea marginal, lo retirás.

```
@app.get("/articulos")
def listar(cursor: str | None = None, page: int | None = None, limite: int = 20):
    limite = min(limite, 100)
    if cursor is not None or page is None:
        # Camino nuevo: keyset (también el default sin parámetros)
        return listar_por_keyset(cursor, limite)
    # Camino legado: OFFSET, marcado como deprecado en la respuesta
    resp = listar_por_offset(page, limite)
    resp["deprecation"] = "Usá next_cursor; ?page será retirado."
    return resp
```

Lo importante es no quedarte para siempre en el modo dual: el código que mantiene dos paginaciones a la vez es más difícil de razonar y de testear. Poné una fecha, comunicala y cumplila.

Errores comunes

- **Olvidar el desempate único:** ordenar por una columna no única (una fecha, un nombre, un precio) sin agregar la clave primaria termina en filas repetidas o salteadas.
- **Que el índice no coincida con el ORDER BY:** si el índice es `(creado_en DESC, id DESC)` pero la consulta ordena en ASC, PostgreSQL no puede aprovecharlo bien. La dirección importa tanto como las columnas.
- **Usar una columna mutable como cursor:** si paginás por `actualizado_en` y esa fila se actualiza entre dos páginas, el cursor deja de ser estable. Preferí valores inmutables, como `creado_en` combinado con `id`.
- **Pretender saltar a una página arbitraria:** el keyset es secuencial, pensado para "siguiente" y "anterior",

no para "ir a la página 4.000". Si tu producto necesita números de página clicables, esta no es la herramienta.

- **Paginar por una columna nullable sin manejar los NULL:** una comparación contra NULL no es ni verdadera ni falsa, y se te caen filas. Usá columnas NOT NULL o maneja el cruce explícitamente.
- **Confundir el cursor inclusivo con el exclusivo:** según la herramienta (Prisma es inclusivo, el SQL con `<` es exclusivo), repetís o te saltás la fila del borde. Testealo con la prueba de recorrido completo.
- **Recalcular un score de orden en cada consulta:** si ordenás por relevancia y el score cambia entre páginas, el cursor pierde sentido. Materializá el valor de orden.

Cuándo usar cada uno

OFFSET sigue siendo una opción razonable para conjuntos chicos y mayormente estáticos donde el usuario espera números de página: tablas de administración, listados de un CMS, resultados de búsqueda acotados. Keyset gana cuando controlás los dos extremos y necesitás velocidad y consistencia: scroll infinito, feeds, exportaciones masivas, sincronización entre servicios o recorrer millones de registros de log.

Como regla práctica: si la tabla puede superar unos pocos miles de filas y vas a paginar en profundidad, arrancá con keyset desde el día uno. Migrar más tarde, cuando ya hay clientes dependiendo del comportamiento de OFFSET, siempre sale más caro que diseñarlo bien desde el principio.

Lista de verificación para producción

- El `ORDER BY` termina en una columna (o combinación) única, normalmente la clave primaria.
- Existe un índice compuesto que coincide con las columnas del orden y sus direcciones.
- Confirmaste con `EXPLAIN (ANALYZE, BUFFERS)` un Index Scan plano respecto de la profundidad.
- Las columnas del cursor son inmutables y, idealmente, NOT NULL.
- El cursor es opaco (Base64), versionado y, si hace falta, firmado con HMAC.
- Hay un techo al tamaño de página en el servidor.
- Soportás "siguiente" y "anterior", cada uno con el truco de la fila extra para hasNext/hasPrev.
- El total se muestra aproximado o diferido, nunca con un COUNT por request.
- Tenés una prueba que recorre todas las páginas y verifica cero huecos y cero duplicados.
- Para tablas anchas, evaluaste deferred join o índice de cobertura.

Preguntas frecuentes

¿El keyset funciona en MySQL, SQLite u otros motores?

Sí. La comparación de tuplas es estándar SQL y la soportan MySQL/MariaDB y SQLite, entre otros. El principio (condición de cursor + orden estable + índice) es universal; solo cambian detalles de sintaxis y de cómo cada optimizador aprovecha el índice. Verificá siempre el plan en tu motor.

¿Puedo ordenar por una columna calculada o por varias tablas con join?

Podés, siempre que el valor de orden sea estable entre páginas y exista una forma de indexarlo (por ejemplo, una columna materializada o un índice sobre la expresión). Si el orden surge de un cálculo que cambia, el cursor deja de ser confiable.

¿Qué tamaño de página conviene?

Lo suficientemente chico para responder rápido y no cargar de más al cliente, y lo suficientemente grande para no hacer mil requests. Entre 20 y 50 suele ser un buen punto de partida para UIs; para exportaciones o sincronización podés subir, siempre con un techo.

¿Y si necesito sí o sí "ir a la página N"?

Entonces el keyset puro no alcanza y tenés un requisito de producto que pide números de página. Opciones: aceptar OFFSET para esas vistas concretas (asumiendo el costo), o un esquema híbrido donde precalculás "anclas" cada N filas. Pero antes preguntate en serio si el usuario realmente salta a la página 4.000 o si alcanza con buscar y filtrar.

Un benchmark concreto, paso a paso

Para ver la diferencia con tus propios ojos, armá una tabla de prueba con datos sintéticos. PostgreSQL trae `generate_series`, que es perfecto para esto.

```
CREATE TABLE articulos (
  id          bigserial PRIMARY KEY,
  titulo      text NOT NULL,
  creado_en   timestamptz NOT NULL
);

INSERT INTO articulos (titulo, creado_en)
SELECT 'Articulo ' || g,
       now() - (g || ' minutes')::interval
FROM generate_series(1, 5_000_000) AS g;

CREATE INDEX idx_articulos_keyset
ON articulos (creado_en DESC, id DESC);

ANALYZE articulos;
```

Ahora compará. Primero, OFFSET profundo:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, titulo, creado_en FROM articulos
ORDER BY creado_en DESC, id DESC
LIMIT 20 OFFSET 2_000_000;
```

En el plan vas a ver un nodo `Limit` que, por debajo, recorrió dos millones de filas para tirarlas, con una cuenta de buffers altísima y un tiempo que se mide en cientos de milisegundos (o más, según el hardware). Ahora el keyset, parándonos en un cursor cualquiera del medio de la tabla:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, titulo, creado_en FROM articulos
WHERE (creado_en, id) < (now() - interval '2000000 minutes', 3000000)
ORDER BY creado_en DESC, id DESC
LIMIT 20;
```

Acá el plan es un `Index Scan` que lee apenas las 20 filas pedidas más unas pocas de overhead del árbol: un puñado de buffers y un tiempo de un dígito de milisegundos. Lo revelador es repetir la prueba moviendo el cursor más y más profundo: el número casi no se mueve. Esa "línea plana" frente a la "rampa" del OFFSET es, en una sola imagen, todo el argumento de esta guía.

Qué mirar en el `EXPLAIN ANALYZE`: el tipo de nodo (querés `Index Scan`/`Index Only Scan`, no `Seq Scan` ni `Sort`), la línea `Buffers: shared hit/read` (el trabajo real de E/S), y la diferencia entre filas estimadas y reales (si difieren por órdenes de magnitud, te falta un `ANALYZE` o tenés estadísticas pobres).

Cursores compuestos y claves ordenadas por tiempo

Cuando el orden involucra tres o más columnas, el cursor simplemente lleva todos los valores de desempate, y la condición usa la tupla completa (si todas las direcciones coinciden) o la cascada de OR (si se mezclan). Por ejemplo, ordenar por `prioridad DESC, creado_en DESC, id DESC`:

```
SELECT id, titulo, prioridad, creado_en
FROM tareas
```

```
WHERE (prioridad, creado_en, id) < (:p, :c, :i)
ORDER BY prioridad DESC, creado_en DESC, id DESC
LIMIT 20;

CREATE INDEX idx_tareas_keyset
ON tareas (prioridad DESC, creado_en DESC, id DESC);
```

Hay un caso especial que simplifica todo: si tu clave primaria ya está ordenada por tiempo, podés paginar por ella sola y ahorrarte el desempate. Es el atractivo de **UUIDv7** (estandarizado en 2024) y de claves tipo ULID: incluyen un componente temporal en los bits más significativos, así que ordenan cronológicamente y son únicas a la vez. Con una clave así, `ORDER BY id DESC` y un cursor de una sola columna alcanzan, sin exponer un contador secuencial adivinable.

```
-- Con UUIDv7 como PK: el id ya ordena por tiempo y es único
SELECT id, titulo FROM articulos
WHERE id < :cursor_id
ORDER BY id DESC
LIMIT 20;
```

Ojo con un matiz de seguridad: un id secuencial (bigserial) expuesto en el cursor revela el volumen y el ritmo de creación de registros (el clásico "ataque del número de factura"). Los cursores opacos mitigan la lectura casual, pero si el dato es sensible, preferí claves no adivinables como UUIDv7 desde el diseño.

Exportaciones y ETL: recorrer tablas enormes

El keyset no es solo para UIs. Es la mejor forma de recorrer una tabla completa en lotes para un export, una migración o un pipeline de ETL, sin mantener una transacción gigante abierta ni sufrir el costo cuadrático de paginar con OFFSET lote tras lote.

```
def recorrer_todo(db, lote=1000):
    cursor = None
    while True:
        if cursor:
            filas = db.fetch(
                """SELECT id, datos FROM eventos
                   WHERE id > %s ORDER BY id ASC LIMIT %s""",
                (cursor, lote))
        else:
            filas = db.fetch(
                """SELECT id, datos FROM eventos
                   ORDER BY id ASC LIMIT %s""", (lote,))
        if not filas:
            break
        yield from filas
        cursor = filas[-1]["id"]
```

Cada lote es una consulta corta e independiente que aprovecha el índice, así que el recorrido completo es lineal en el tamaño de la tabla, no cuadrático. Como bonus, al no depender de una sola transacción larga, no bloqueás el autovacuum ni acumulás un snapshot enorme; si el proceso se cae, reanudás desde el último cursor guardado en lugar de empezar de cero. Para una tabla de cientos de millones de filas, esta es la diferencia entre un export que termina y uno que nunca lo hace.

Consistencia, aislamiento y datos que cambian

Una pregunta que aparece tarde, ya en producción: ¿qué pasa con la consistencia mientras el usuario pagina y la tabla cambia por debajo? El keyset ya resuelve el peor síntoma de OFFSET -saltarse o repetir filas cuando se insertan o borran registros- porque cada página se ancla en un valor real, no en una posición. Si entra una fila nueva "más arriba" que tu cursor, no afecta a las páginas que ya recorriste: el cursor sigue apuntando al mismo lugar lógico.

Eso no significa que veas una "foto" perfectamente coherente de toda la tabla a lo largo de una sesión de paginado. Cada página es una transacción independiente, así que entre la página 3 y la 4 el mundo pudo cambiar. Para la inmensa mayoría de los feeds y listados, eso es exactamente lo que querés: el usuario ve lo último al pedir cada página. Pero si necesitás una vista estrictamente consistente -un reporte financiero que recorre millones de filas y no puede mezclar dos estados del mundo- tenés dos caminos.

- **Snapshot por transacción larga:** abrís una transacción `REPEATABLE READ` y paginás dentro de ella. Ves una foto congelada, pero mantenés un snapshot abierto que frena al autovacuum y no escala a sesiones de usuario; sirve para un export batch, no para una UI.
- **Anclar a un instante:** agregás `AND creado_en <= :inicio_de_sesion` a todas las páginas, usando el timestamp del primer request como tope. Así el recorrido ignora todo lo insertado después de que el usuario empezó, sin transacciones largas. Es la técnica que usan muchos feeds para que el scroll infinito no "salte" cuando llegan publicaciones nuevas.

Para los borrados, el keyset es naturalmente robusto: si la fila del cursor se elimina, la comparación `<` sigue funcionando porque compara valores, no la existencia de esa fila puntual. La página siguiente trae lo que venga después de esos valores, exista o no la fila original. No necesitás tombstones ni lógica especial; es una ventaja silenciosa sobre cualquier esquema basado en posición.

OFFSET vs Keyset: resumen comparativo

Para tenerlo a mano, así quedan las dos técnicas frente a frente en las dimensiones que importan.

- **Rendimiento en profundidad:** OFFSET se degrada linealmente con la profundidad; keyset se mantiene plano sin importar la página.
- **Consistencia con datos que cambian:** OFFSET salta y repite filas ante inserciones/borrados; keyset se ancla en valores reales y no sufre ese corrimiento.
- **Saltar a una página arbitraria:** OFFSET lo permite de forma trivial; keyset no (es secuencial, "siguiente/anterior").
- **Total de filas y "página X de Y":** fácil con OFFSET; con keyset hay que estimarlo o diferirlo.
- **Complejidad de implementación:** OFFSET es trivial; keyset pide pensar desempate, índice, NULLs y formato de cursor.
- **Acoplamiento de la API:** OFFSET expone números de página estables; keyset expone cursores opacos que podés versionar.

La lectura honesta: no hay un ganador absoluto, hay un ganador por contexto. Para un panel de administración con miles de filas y necesidad de números de página, OFFSET es la herramienta correcta y más simple. Para cualquier cosa que crezca y se navegue en profundidad -feeds, timelines, APIs públicas, exportaciones- keyset es la opción que no te va a traicionar a escala. El error caro no es elegir uno u otro, sino elegir OFFSET por inercia para un sistema que claramente va a necesitar keyset, y descubrirlo recién cuando la tabla ya tiene millones de filas y miles de clientes dependiendo del contrato viejo.

Por qué el salto es barato: anatomía del B-tree

Para confiar de verdad en el keyset ayuda entender por qué "saltar al cursor" es tan barato. El índice por defecto de PostgreSQL es un B-tree: un árbol balanceado donde cada nodo apunta a rangos de valores, y las hojas, al final, guardan las entradas ordenadas. Buscar un valor (o el punto donde empezaría) es bajar del nodo raíz a la hoja correcta, y eso cuesta del orden de `log(n)` pasos. En una tabla de mil millones de filas, ese logaritmo es apenas unos pocos niveles: el motor llega a la posición del cursor tocando un puñado de páginas, no millones.

Una vez en la hoja correcta, las entradas están encadenadas en orden, así que leer las siguientes 20 es

seguir el hilo: lecturas secuenciales, baratas y predecibles. Esa es la asimetría que explica todo. OFFSET 1.000.000 obliga a caminar un millón de entradas desde el principio porque el motor no tiene forma de saber, sin contarlas, cuáles descartar. El cursor del keyset, en cambio, le da al B-tree exactamente la llave que necesita para bajar directo. No es magia ni un caso optimizado a mano: es la estructura de datos haciendo lo que mejor sabe hacer.

Esto también explica por qué la dirección del índice importa tanto. Un B-tree creado como `(creado_en DESC, id DESC)` tiene sus hojas encadenadas en ese sentido; si tu consulta pide el orden inverso, el motor puede recorrerlo hacia atrás, pero si pedís una combinación de direcciones que el índice no refleja, ya no hay un único hilo ordenado para seguir y aparece un `Sort` que arruina la fiesta. Coincidir índice y `ORDER BY` no es una formalidad: es lo que mantiene el acceso en "bajar y seguir el hilo" en lugar de "leer todo y ordenar".

El costo del otro lado: índices y escrituras

Ningún índice es gratis. Cada uno que agregás acelera lecturas pero encarece cada `INSERT`, `UPDATE` y `DELETE`, porque el motor tiene que mantener también esa estructura, y ocupa espacio en disco y en caché. Para keyset esto importa por dos motivos concretos.

Primero, el índice de cobertura con `INCLUDE` es tentador pero engorda: si metés tres columnas grandes en el payload, cada actualización de esas columnas reescribe entradas del índice y cada página de índice cachea menos entradas útiles. Incluí solo lo que devolvés en el listado; para todo lo demás, el deferred join suele rendir mejor. Segundo, si tu patrón de escritura es intenso (muchas inserciones por segundo), de a poco el índice acumula "hinchazón" (bloat) y conviene vigilar su tamaño y, si hace falta, reconstruirlo con `REINDEX CONCURRENTLY` sin bloquear la tabla.

```
-- Tamaño de un índice y de la tabla, para vigilar el bloat
SELECT
  pg_size_pretty(pg_relation_size('idx_articulos_keyset')) AS indice,
  pg_size_pretty(pg_relation_size('articulos'))            AS tabla;

-- Reconstrucción sin bloquear lecturas/escrituras (PostgreSQL 12+)
REINDEX INDEX CONCURRENTLY idx_articulos_keyset;
```

La regla de equilibrio es simple: indexá lo que de verdad paginás y consultás, no "por las dudas". Un índice keyset bien elegido -las columnas del orden, terminando en la clave primaria, con la dirección correcta- suele ser todo lo que necesitás, y su costo de escritura es perfectamente asumible frente a lo que te ahorra en lecturas. El antipatrón es acumular media docena de índices solapados "por si acaso": pagás la escritura de todos y usás dos.

Conclusión

La paginación por keyset no es un truco exótico: es la forma correcta de paginar en cualquier sistema que vaya a crecer. Cambiás "contá y descartá" por "saltá directo al cursor", apoyás esa consulta en un índice que coincide con tu orden, y de golpe la latencia se vuelve plana y los resultados, consistentes incluso con datos que cambian. El precio es modesto: perdés el "ir a la página N" gratis y tenés que pensar el desempate, los NULL y el formato del cursor. A cambio, ganás una paginación que rinde igual en la fila 20 que en la 20 millones. Si estás diseñando una API o un feed hoy, empezá por keyset; tu yo del futuro, con la tabla ya en millones de filas, te lo va a agradecer.

English

Almost all of us learned to paginate with `LIMIT` and `OFFSET`. It's intuitive, it reads well, and on a small table nobody notices anything wrong. The problem is that this approach degrades exactly when you need it most: when the table grows and users navigate deep into it. In this guide you'll see why OFFSET doesn't scale, how keyset pagination (also called cursor pagination) works, how to implement it properly with the right index and a clean API, and how to take it to production: NULLs, mixed orderings, signed cursors, total counts, GraphQL and ORM integration, infinite scroll on the frontend, testing, and migrating away from OFFSET without breaking anyone.

The core idea fits in one sentence: **instead of asking the database to count and discard rows, we hand it a reference point so it jumps straight to where it left off.** That change of framing turns a query that gets slower the deeper you page into one that takes practically the same time on page 2 as on page 50,000.

The problem with OFFSET

To make it concrete, asking for page 5,000 with OFFSET looks like this.

```
SELECT id, title, created_at
FROM articles
ORDER BY created_at DESC
LIMIT 20 OFFSET 99980;
```

PostgreSQL doesn't magically jump to row 99,981. It scans from the beginning through the previous 99,980 rows, throws them away one by one, and only then returns the 20 you asked for. The deeper you page, the more work the engine does. On tables with millions of rows, this turns millisecond queries into multi-second ones. The cost is linear in the offset: page 1 is free, page 10,000 charges you for having read everything before it.

There's a second problem, less visible but more treacherous: **correctness**. If someone inserts or deletes a row while the user moves from page 1 to page 2, the whole count shifts. A row that sat at position 21 slides to position 20 and never shows up on the next page; the user simply never sees it. Conversely, if a row is deleted, another one repeats across two pages. With data that changes often, OFFSET lies to you, and the worst part is that it lies silently: nobody sees an error, records just go missing or show up twice.

Measuring the real cost with EXPLAIN ANALYZE

Don't take any of the above on faith; measure it. The tool is `EXPLAIN (ANALYZE, BUFFERS)`, which actually runs the query and shows you the plan, the timings and -crucially- how many disk and cache blocks it had to touch.

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, title, created_at
FROM articles
ORDER BY created_at DESC
LIMIT 20 OFFSET 99980;
```

On a query with a deep OFFSET you'll see something like `Limit ... rows=20` over a node that reports having processed ~100,000 rows, with a huge `shared hit/read` count (the blocks read). That's the material proof of the problem: the engine touched a hundred thousand rows to return you twenty. The `Buffers` line is the one an experienced DBA watches most, because I/O work is what doesn't scale.

As a rough mental reference, on a table of a few million well-indexed rows, OFFSET 20 answers in a handful of milliseconds, while OFFSET 100,000 can climb to hundreds of milliseconds or more, and keeps growing. Keyset, by contrast, stays in that handful of milliseconds regardless of depth. The difference isn't a constant: it's the shape of the curve.

How keyset pagination works

The idea is to flip the question around. Instead of "skip 99,980 rows and give me the next 20", we tell the database "give me the 20 rows that come right after the last one I already showed". To do that we use the values of that last row as a reference point: the cursor.

```
-- First page
SELECT id, title, created_at
FROM articles
ORDER BY created_at DESC, id DESC
LIMIT 20;

-- Following pages: pass the (created_at, id) of the last row seen
SELECT id, title, created_at
FROM articles
WHERE (created_at, id) < ('2026-06-20 10:30:00', 8842)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

The key is the tuple comparison `(created_at, id) < (...)`. PostgreSQL evaluates it natively: it compares `created_at` first and, on a tie, breaks it with `id`. With the right index, the engine jumps straight to the cursor's position without scanning extra rows. Response time stays almost constant, whether you're on page 2 or page 50,000.

Tuple comparison, in depth

It's worth understanding exactly what that row-value comparison (from the SQL standard) does, because it's the heart of everything. The expression `(a, b) < (x, y)` does NOT mean "a < x AND b < y". It means, in lexicographic order, the same as:

```
WHERE created_at < '2026-06-20 10:30:00'
   OR (created_at = '2026-06-20 10:30:00' AND id < 8842)
```

That is: bring everything strictly earlier by date, and for the exact-date-tie case, break the tie by id. PostgreSQL understands the tuple form directly and, most importantly, knows how to translate it into an efficient access on the composite index. That translation is the difference between an `Index Scan` that starts right at the cursor and a scan that filters row by row.

Why not just write the `OR` by hand and be done? Because historically some planners didn't exploit the `OR` form well to skip inside the index, and ended up scanning too much. The tuple form is clearer for the optimizer and for whoever reads the query. That said, you'll absolutely need the `OR` version the moment you mix sort directions (one column ASC and another DESC), because there the plain tuple stops representing what you want. We'll see that below.

One detail that trips many people up: the comparison is between *whole tuples*, not column by column independently. `(2, 100) < (3, 1)` is true even though 100 > 1, because the first element already decides. Think of it like sorting words in a dictionary: first letter first; only on a tie do you look at the second.

Why you need a unique tiebreaker

Ordering by `created_at` alone isn't enough. If two articles share the same timestamp, their relative order is undefined, and that's where you start skipping or repeating rows across pages. That's why we add `id` as a second key. The general rule: the `ORDER BY` columns, taken together, must be unique. In practice that almost always means ending the list with the primary key. It doesn't matter if the leading column "almost never" repeats: with enough traffic, two rows will land in the same millisecond, and the day that happens you'll lose a row without noticing.

The index that makes it fast

Keyset without an index to back it is worthless: you fall back to a sequential scan. You need a composite index that matches your `ORDER BY`, including the direction of each column.

```
CREATE INDEX idx_articles_keyset
ON articles (created_at DESC, id DESC);
```

Then confirm the plan actually uses it with `EXPLAIN ANALYZE`. You want to see an `Index Scan` (or `Index Only Scan`) and an execution time that doesn't grow with depth; if a `Seq Scan` or an expensive `Sort` shows up, something doesn't match between the index and the query.

```
EXPLAIN ANALYZE
SELECT id, title, created_at
FROM articles
WHERE (created_at, id) < ('2026-06-20 10:30:00', 8842)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

Index-only scans and covering indexes

If your query only needs columns that are in the index, PostgreSQL can resolve it without touching the table: that's an `Index Only Scan`, the fastest possible plan. The catch is that you usually want to return more columns than the ordering ones (a title, a summary). The solution is a **covering index** with the `INCLUDE` clause: the ordering columns go in the key and the ones you only need to read go in the payload.

```
CREATE INDEX idx_articles_keyset_cover
ON articles (created_at DESC, id DESC)
INCLUDE (title);
```

With this, a query asking for `id, title, created_at` can be resolved entirely from the index. Careful: adding columns with `INCLUDE` fattens the index and makes writes more expensive, so include only what you actually return in the listing, not the whole row. For very wide tables, there's a better alternative below: the deferred join.

An often-forgotten requirement for index-only scans: the table has to be reasonably "visible" to the visibility map, which in practice means a `VACUUM` kept up to date. If you see lots of `Heap Fetches` in the plan, that's a sign autovacuum is falling behind.

PostgreSQL 18: skip scan and what changes for keyset

PostgreSQL 18 (released in 2025) brought **skip scan** for multicolumn B-tree indexes: the planner can now use a composite index even when the query doesn't filter on the first column, as long as that column has few distinct values. The engine internally walks the distinct values of the prefix and dives into the index for each one.

What does that have to do with keyset? Two things. First, it gives you more freedom when designing indexes: a single composite index can now serve more query patterns than before. Second, and more important, **it doesn't change the golden rule**: for keyset to be fast, the index has to match the `ORDER BY` in columns and direction. Skip scan helps with equality filters on low-cardinality prefixes (for example, paginating within a `status` with three possible values), it doesn't replace the ordered index the cursor needs in order to jump. Think of it as a complementary tool, not a license to ignore the index's order.

Mixed orderings: when ASC and DESC break the tuple

The tuple comparison `(a, b) < (x, y)` assumes every column goes in the same direction. The moment you need to order by one ascending column and another descending -say, price ascending and, on a tie, id descending- the plain tuple stops representing what you want and you have to decompose the condition by hand.

```
-- Desired order: price ASC, id DESC
```

```
-- The tuple form does NOT work here. We decompose:
SELECT id, name, price
FROM products
WHERE price > 49.90
   OR (price = 49.90 AND id < 8842)
ORDER BY price ASC, id DESC
LIMIT 20;
```

Notice the pattern: for the ASC column we use `>` (we want what comes after, going up), and for the DESC tiebreaker we use `<`. The index must reflect exactly those directions for the plan to be efficient:

```
CREATE INDEX idx_products_price
ON products (price ASC, id DESC);
```

If you have three or more ordering columns with mixed directions, the condition grows into a cascade of nested `OR`s. It's not hard, but it's easy to get wrong by hand, so it's best to generate the clause from code based on the list of ordering columns, or lean on a library that already does it (we'll see that in the ORM section).

Handling NULL values

NULLs are the number-one source of subtle keyset bugs. In PostgreSQL, by default, `ORDER BY ... DESC` puts NULLs first (`NULLS FIRST`) and `ASC` puts them last (`NULLS LAST`). The problem is that any comparison against `NULL` with `<` or `>` returns `NULL` (neither true nor false), so your cursor condition discards rows it shouldn't.

The practical rule: always paginate by `NOT NULL` columns when you can. `created_at` and the primary key almost always are, which is why they're ideal cursors. If you must order by a nullable column (for example, `published_at`, which is NULL for drafts), explicitly fix the NULL ordering and handle the crossing between the non-NULL block and the NULL block.

```
-- Order: published_at DESC NULLS LAST, id DESC
-- Cursor on a row with a NON-null published_at:
SELECT id, title, published_at
FROM articles
WHERE (published_at IS NOT NULL AND published_at < :cur_pub)
   OR (published_at IS NOT NULL AND published_at = :cur_pub AND id < :cur_id)
   OR (published_at IS NULL)              -- the NULL block comes afterward
ORDER BY published_at DESC NULLS LAST, id DESC
LIMIT 20;
```

Once the cursor has entered the NULL block (the reference row has `published_at IS NULL`), the condition simplifies to "rows with published_at NULL and id smaller than the cursor's". It's manageable, but it adds real complexity, and it's the reason to avoid nullable columns in the ordering whenever possible. The index must also declare the same `NULLS LAST` so the plan can use it.

Bidirectional pagination: previous and next

So far we've moved "next". To support "previous" you invert the logic: flip the comparison operator and the `ORDER BY` direction, and at the end reorder the result to return it in the natural order.

```
-- PREVIOUS page relative to the cursor (created_at, id)
SELECT * FROM (
  SELECT id, title, created_at
  FROM articles
  WHERE (created_at, id) > ('2026-06-20 10:30:00', 8842)
  ORDER BY created_at ASC, id ASC   -- inverted
  LIMIT 20
) sub
ORDER BY created_at DESC, id DESC;   -- back to natural order
```

The subselect brings the 20 rows immediately before in inverted order (those closest to the cursor first), and the outer select returns them in the order the user expects. With this you can offer "previous" and "next" buttons without ever going back to OFFSET. To know whether there's a previous or next page, apply the same trick of asking for one extra row (`LIMIT 21`) in each direction.

Opaque cursors for your API

Exposing raw `created_at` and `id` in the URL works, but it couples your API to the database schema. The day you want to change the sort column, you break every client that stored those cursors. The common practice is to encode the cursor (in Base64, for example) and treat it as an opaque token that the client just hands back without interpreting.

```
import base64, json

def encode_cursor(created_at: str, id: int) -> str:
    payload = json.dumps({"c": created_at, "i": id})
    return base64.urlsafe_b64encode(payload.encode()).decode()

def decode_cursor(cursor: str) -> tuple[str, int]:
    payload = json.loads(base64.urlsafe_b64decode(cursor.encode()))
    return payload["c"], payload["i"]
```

With that in place, a listing endpoint stays simple and consistent. Notice the trick of asking for one extra row (`limit + 1`): it lets us know whether there's a next page without running a separate `COUNT`.

```
@app.get("/articles")
def list_articles(cursor: str | None = None, limit: int = 20):
    limit = min(limit, 100)  # defensive cap: never trust the client
    if cursor:
        created_at, last_id = decode_cursor(cursor)
        rows = db.fetch(
            """SELECT id, title, created_at FROM articles
               WHERE (created_at, id) < (%s, %s)
               ORDER BY created_at DESC, id DESC
               LIMIT %s""",
            (created_at, last_id, limit + 1),
        )
    else:
        rows = db.fetch(
            """SELECT id, title, created_at FROM articles
               ORDER BY created_at DESC, id DESC
               LIMIT %s""",
            (limit + 1,),
        )

    has_more = len(rows) > limit
    rows = rows[:limit]
    next_cursor = encode_cursor(rows[-1]["created_at"], rows[-1]["id"]) if has_more else None
    return {"data": rows, "next_cursor": next_cursor}
```

The client doesn't need to know anything about the cursor's internals: it calls `/articles`, gets the data and a `next_cursor`, and asks for the next page by calling again with that value. When `next_cursor` is null, it has reached the end.

Signed and versioned cursors

Base64 is not security: anyone can decode and edit it. For most listings that doesn't matter, but if the cursor could be used to infer or force sensitive data, sign it with HMAC and verify the signature on the way in. That's how you detect any tampering.

```
import base64, hmac, hashlib, json
```

```
SECRET = b"long-secret-key-from-the-environment"

def encode_cursor(payload: dict) -> str:
    body = json.dumps({"v": 1, **payload}, separators=(",", ":")).encode()
    sig = hmac.new(SECRET, body, hashlib.sha256).digest()[:16]
    return base64.urlsafe_b64encode(body + b"." + sig).decode()

def decode_cursor(cursor: str) -> dict:
    raw = base64.urlsafe_b64decode(cursor.encode())
    body, sig = raw.rsplit(b".", 1)
    expected = hmac.new(SECRET, body, hashlib.sha256).digest()[:16]
    if not hmac.compare_digest(sig, expected):
        raise ValueError("invalid cursor")
    return json.loads(body)
```

Two details worth their weight in gold in production. The `"v": 1` field versions the cursor format: the day you change the sort column, you bump the version and can reject (or migrate) old cursors instead of silently returning wrong results. And the signature comparison uses `hmac.compare_digest`, which is constant-time, so as not to leak information through response timing. On an invalid or tampered cursor, respond with a clear `400` rather than swallowing the error.

Keyset with filters and search

In real life you almost never paginate "the whole table": you filter by category, by author, by status, or combine it with a text search. Keyset still works, but the index has to put the equality-filter columns **before** the ordering columns.

```
-- List articles in one category, paginated
SELECT id, title, created_at
FROM articles
WHERE category_id = 7
  AND (created_at, id) < ('2026-06-20 10:30:00', 8842)
ORDER BY created_at DESC, id DESC
LIMIT 20;

-- The right index puts equality first:
CREATE INDEX idx_articles_cat_keyset
ON articles (category_id, created_at DESC, id DESC);
```

The classic mnemonic for composite indexes is **"equality, sort, range"**: first the columns you filter with `=`, then the `ORDER BY` ones, and finally any range. If you filter by several possible category values, remember that PostgreSQL 18 with skip scan can help when that column has low cardinality. For text search, you can combine a `tsvector` filter with keyset over a stable ranking field plus the id, but watch out: if you order by relevance (a computed score), that value must be deterministic and materialized, not recomputed on every query, or the cursor stops making sense across pages.

The total-count problem and "page X of Y"

Keyset has an honest design cost: it doesn't give you the total row count or the page number for free. That's exactly what makes it fast (it doesn't count what it doesn't need), but sometimes the product asks for a "1,234 results" or a "page 3 of 50". You have several ways out, from least to most exact.

- **Don't show the total.** The most honest option for feeds and infinite scroll: nobody on Twitter expects to see "tweet 4,500 of 12,000". Show "next/previous" and move on.
- **Approximate count.** For "about 12 thousand results", use the planner's estimate, which is instant because it comes from the statistics.
- **Cached exact count.** If you truly need the exact number, compute it separately (a job, an incremental counter with triggers, or a `COUNT` cached for a few minutes) and not on every request.

```
-- Instant estimate of a table's total
SELECT reltuples::bigint AS estimate
FROM pg_class
WHERE relname = 'articles';

-- Finer estimate, respecting the filter, by reading the plan:
EXPLAIN (FORMAT JSON)
SELECT 1 FROM articles WHERE category_id = 7;
-- the "Plan Rows" field of the JSON is the row estimate
```

The underlying idea: separate navigation (fast, by keyset) from the volume metric (expensive, approximate or deferred). Mixing them in the same query is what drags you back to the OFFSET problem.

Deferred join for wide tables

When the table has many columns or large columns (long text, JSON), even an Index Scan can be slow because, for each row, the engine goes to the table to fetch the rest of the record. The **deferred join** pattern (or "late row lookup") solves this in two steps: first you use the index to fetch only the page's ids -blazing fast, almost always index-only- and then you join with the table to hydrate only those 20 rows.

```
SELECT a.*
FROM articles a
JOIN (
  SELECT id
  FROM articles
  WHERE (created_at, id) < ('2026-06-20 10:30:00', 8842)
  ORDER BY created_at DESC, id DESC
  LIMIT 20
) page USING (id)
ORDER BY a.created_at DESC, a.id DESC;
```

The inner subselect touches only the index and returns 20 ids; the outer join reads just those 20 full rows from the table. The difference shows when rows are wide: you go from reading disk pages full of fat rows you'll discard, to reading exactly what you return. It's the same idea as the covering index, but without having to duplicate columns inside the index.

GraphQL integration: Relay-style connections

If you expose a GraphQL API, you most likely follow Relay's Cursor Connections specification, which is keyset in different clothes. The schema defines `edges` (each with a `node` and its `cursor`) and a `pageInfo` with `hasNextPage`, `hasPreviousPage`, `startCursor` and `endCursor`. The `first`/`after` arguments paginate forward and `last`/`before` backward.

```
type ArticleConnection {
  edges: [ArticleEdge!]!
  pageInfo: PageInfo!
}
type ArticleEdge {
  node: Article!
  cursor: String!
}
type PageInfo {
  hasNextPage: Boolean!
  hasPreviousPage: Boolean!
  startCursor: String
  endCursor: String
}
type Query {
  articles(first: Int, after: String, last: Int, before: String): ArticleConnection!
}
```

The mental trick is that Relay's `cursor` is exactly your encoded keyset cursor, and `after` is the value that

goes into your `WHERE (created_at, id) < ...` clause. The specification is deliberately database-agnostic, but it recommends keyset precisely because `WHERE` clauses leverage indexes and OFFSET doesn't. You implement the resolver with the same keyset query you already have, compute `hasNextPage` with the extra-row trick, and serialize each cursor with your `encode_cursor`.

Implementing it with ORMs

You don't always write SQL by hand. These are the patterns in the most common ORMs; the principle is identical in all of them: cursor condition + stable order + limit, with a matching index.

SQLAlchemy

You can express the tuple comparison with `tuple_`, or lean on the `sqlakeyset` library, which handles cursors, mixed directions and NULLs for you.

```
from sqlalchemy import tuple_, select

# Manual: native tuple comparison
stmt = (
    select(Article)
    .where(tuple_(Article.created_at, Article.id) < (cur_created, cur_id))
    .order_by(Article.created_at.desc(), Article.id.desc())
    .limit(20)
)

# With sqlakeyset: opaque cursors out of the box
from sqlakeyset import select_page
q = select(Article).order_by(Article.created_at.desc(), Article.id.desc())
page = select_page(session, q, per_page=20, page=cursor)  # opaque cursor
next_ = page.paging.bookmark_next if page.paging.has_next else None
```

Django

Django's ORM has no direct tuple comparison, so you build the condition with `Q` objects. For serious cases, the `django-cursor-pagination` library encapsulates all of this.

```
from django.db.models import Q

qs = (
    Article.objects
    .filter(Q(created_at__lt=cur_created) |
            Q(created_at=cur_created, id__lt=cur_id))
    .order_by("-created_at", "-id")[:20]
)
```

Prisma

Prisma ships cursor pagination out of the box with `cursor` + `take`. An important API detail: the cursor is *inclusive*, so you use `skip: 1` to avoid repeating the reference row.

```
const page = await prisma.article.findMany({
  take: 20,
  skip: 1,                       // skip the cursor row (it's inclusive)
  cursor: { id: lastId },        // cursor by unique key
  orderBy: [{ createdAt: 'desc' }, { id: 'desc' }],
});
```

A warning about Prisma and distributed engines: on some backends its cursor helper can generate plans that don't use the index the way you expect. As always, check the actual SQL your ORM emits and run it through `EXPLAIN ANALYZE`; no abstraction excuses you from looking at the plan.

Infinite scroll on the frontend

On the client side, the next_cursor makes infinite scroll trivial: you store the cursor, and when the user reaches the end you ask for the next page and concatenate it. An example with React, no dependencies:

```
function useArticles() {
  const [items, setItems] = React.useState([]);
  const [cursor, setCursor] = React.useState(null);
  const [hasMore, setHasMore] = React.useState(true);
  const [loading, setLoading] = React.useState(false);

  async function loadMore() {
    if (loading || !hasMore) return;
    setLoading(true);
    const url = "/articles?limit=20" + (cursor ? "&cursor=" + encodeURIComponent(cursor) : ""
);
    const res = await fetch(url).then(r => r.json());
    setItems(prev => [...prev, ...res.data]);
    setCursor(res.next_cursor);
    setHasMore(res.next_cursor !== null);
    setLoading(false);
  }

  return { items, loadMore, hasMore, loading };
}
```

Combine it with an `IntersectionObserver` on a sentinel element at the end of the list to fire `loadMore()` automatically. Because each page is anchored to real content (not an offset number), infinite scroll neither skips nor repeats items even as new posts arrive while the user reads: the root of "flickering feed" problems is usually, precisely, OFFSET underneath.

Testing: guaranteeing no skipped or repeated rows

The best test for pagination is a property test: walk every page to the end and verify the resulting set is exactly the table, with no gaps and no duplicates. This catches the classic bugs (forgotten tiebreaker, inclusive cursor, inverted direction) that don't show on a single page.

```
def test_pagination_walks_everything(client, seed_500_articles):
    seen = []
    cursor = None
    while True:
        url = "/articles?limit=37"  # deliberately "ugly" size
        if cursor:
            url += "&cursor=" + cursor
        r = client.get(url).json()
        seen.extend(item["id"] for item in r["data"])
        cursor = r["next_cursor"]
        if cursor is None:
            break

    db_ids = [a.id for a in Article.query.order_by(
        Article.created_at.desc(), Article.id.desc()).all()]

    assert seen == db_ids              # same order, no gaps
    assert len(seen) == len(set(seen)) # zero duplicates
```

A couple of extra tests worth having: one with deliberately duplicated timestamps (to verify the tiebreaker), one with the table mutating between pages (you insert rows mid-walk and confirm already-seen ones don't repeat), and one with a tampered cursor (returning 400, not 500). The "ugly" page size like 37 helps surface edge errors that a round multiple hides.

Observability and production operations

Once in production, what you want to watch changes relative to OFFSET. Keyset's great advantage is that

latency stops depending on depth, so the first health indicator is that your listing endpoint's p95 and p99 are flat regardless of how far people navigate. If you see latency grow with depth, something fell back to a Seq Scan: check that the index still exists and matches the order.

- **Cap the page size.** `limit = min(limit, 100)`. Without a cap, a client asking for `limit=100000` reintroduces the very problem you came to solve.
- **Log invalid cursors.** A spike of 400s from broken cursors usually signals an unversioned format change or an old client; the cursor's version field tells you exactly what happened.
- **Watch Heap Fetches.** If you rely on index-only scans and autovacuum falls behind, latency creeps up. A dashboard of autovacuum lag on the hot tables warns you before the user does.
- **Review plans after every migration.** A column-type `ALTER` or a collation change can silently invalidate index usage. Keep a regression test that fails if the plan stops being an Index Scan.

Migrating from OFFSET without breaking clients

If you already have a production API with `?page=N`, you can't change the contract overnight. The migration goes in layers. First, add the right composite index; it breaks nothing and already improves some queries. Second, support both schemes at once: if `cursor` arrives, use keyset; if `page` arrives, keep OFFSET. Third, start returning `next_cursor` in every response so new clients adopt it. Fourth, deprecate `page` with a notice and usage metrics, and only when `page` traffic is marginal do you retire it.

```
@app.get("/articles")
def list_articles(cursor: str | None = None, page: int | None = None, limit: int = 20):
    limit = min(limit, 100)
    if cursor is not None or page is None:
        # New path: keyset (also the default with no params)
        return list_by_keyset(cursor, limit)
    # Legacy path: OFFSET, flagged as deprecated in the response
    resp = list_by_offset(page, limit)
    resp["deprecation"] = "Use next_cursor; ?page will be retired."
    return resp
```

The important thing is not to stay in dual mode forever: code that maintains two paginations at once is harder to reason about and to test. Set a date, communicate it, and stick to it.

Common pitfalls

- **Forgetting the unique tiebreaker:** ordering by a non-unique column (a date, a name, a price) without adding the primary key ends in repeated or skipped rows.
- **An index that doesn't match the ORDER BY:** if the index is `(created_at DESC, id DESC)` but the query orders ASC, PostgreSQL can't use it well. Direction matters as much as the columns.
- **Using a mutable column as the cursor:** if you paginate by `updated_at` and that row gets updated between two pages, the cursor stops being stable. Prefer immutable values, like `created_at` combined with `id`.
- **Trying to jump to an arbitrary page:** keyset is sequential, built for "next" and "previous", not "go to page 4,000". If your product needs clickable page numbers, this isn't the tool.
- **Paginating by a nullable column without handling NULLs:** a comparison against NULL is neither true nor false, and rows fall through. Use NOT NULL columns or handle the crossing explicitly.
- **Confusing inclusive and exclusive cursors:** depending on the tool (Prisma is inclusive, SQL with `<` is exclusive), you repeat or skip the boundary row. Test it with the full-walk test.
- **Recomputing an ordering score on every query:** if you order by relevance and the score changes between pages, the cursor loses meaning. Materialize the ordering value.

When to use each

OFFSET is still a reasonable choice for small, mostly static sets where the user expects page numbers: admin tables, CMS listings, bounded search results. Keyset wins when you control both ends and need speed and consistency: infinite scroll, feeds, bulk exports, syncing between services, or paging through millions of log records.

As a rule of thumb: if the table can grow past a few thousand rows and you'll paginate deep, start with keyset from day one. Migrating later, once clients already depend on OFFSET's behavior, always costs more than designing it right from the start.

Production checklist

- The `ORDER BY` ends in a unique column (or combination), usually the primary key.
- A composite index exists that matches the ordering columns and their directions.
- You confirmed with `EXPLAIN (ANALYZE, BUFFERS)` an Index Scan flat with respect to depth.
- The cursor columns are immutable and, ideally, NOT NULL.
- The cursor is opaque (Base64), versioned and, if needed, HMAC-signed.
- There's a server-side cap on page size.
- You support "next" and "previous", each with the extra-row trick for hasNext/hasPrev.
- The total is shown approximate or deferred, never with a COUNT per request.
- You have a test that walks all pages and verifies zero gaps and zero duplicates.
- For wide tables, you evaluated a deferred join or covering index.

Frequently asked questions

Does keyset work in MySQL, SQLite or other engines?

Yes. Tuple comparison is standard SQL and is supported by MySQL/MariaDB and SQLite, among others. The principle (cursor condition + stable order + index) is universal; only syntax details and how each optimizer uses the index change. Always check the plan on your engine.

Can I order by a computed column or across joined tables?

You can, as long as the ordering value is stable across pages and there's a way to index it (for example, a materialized column or an index on the expression). If the order comes from a calculation that changes, the cursor stops being reliable.

What page size should I use?

Small enough to answer fast and not overload the client, and large enough to avoid a thousand requests. Between 20 and 50 is usually a good starting point for UIs; for exports or syncing you can go higher, always with a cap.

What if I absolutely need "go to page N"?

Then pure keyset isn't enough and you have a product requirement asking for page numbers. Options: accept OFFSET for those specific views (taking the cost), or a hybrid scheme where you precompute "anchors" every N rows. But first, seriously ask whether the user really jumps to page 4,000 or whether search and filtering would do.

A concrete benchmark, step by step

To see the difference with your own eyes, build a test table with synthetic data. PostgreSQL ships `generate_series`, which is perfect for this.

```
CREATE TABLE articles (
  id          bigserial PRIMARY KEY,
  title       text NOT NULL,
  created_at  timestamptz NOT NULL
);

INSERT INTO articles (title, created_at)
SELECT 'Article ' || g,
       now() - (g || ' minutes')::interval
FROM generate_series(1, 5_000_000) AS g;

CREATE INDEX idx_articles_keyset
ON articles (created_at DESC, id DESC);

ANALYZE articles;
```

Now compare. First, a deep OFFSET:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, title, created_at FROM articles
ORDER BY created_at DESC, id DESC
LIMIT 20 OFFSET 2_000_000;
```

In the plan you'll see a `Limit` node that, underneath, scanned two million rows to throw them away, with a very high buffer count and a time measured in hundreds of milliseconds (or more, depending on hardware). Now keyset, standing on some cursor in the middle of the table:

```
EXPLAIN (ANALYZE, BUFFERS)
SELECT id, title, created_at FROM articles
WHERE (created_at, id) < (now() - interval '2000000 minutes', 3000000)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

Here the plan is an `Index Scan` that reads just the 20 rows requested plus a few of tree overhead: a handful of buffers and a single-digit millisecond time. The revealing part is to repeat the test moving the cursor deeper and deeper: the number barely moves. That "flat line" against OFFSET's "ramp" is, in a single image, the entire argument of this guide.

What to look at in `EXPLAIN ANALYZE`: the node type (you want `Index Scan`/`Index Only Scan`, not `Seq Scan` or `Sort`), the `Buffers: shared hit/read` line (the real I/O work), and the difference between estimated and actual rows (if they differ by orders of magnitude, you're missing an `ANALYZE` or have poor statistics).

Composite cursors and time-ordered keys

When the order involves three or more columns, the cursor simply carries all the tiebreaker values, and the condition uses the full tuple (if all directions match) or the OR cascade (if they're mixed). For example, ordering by `priority DESC, created_at DESC, id DESC`:

```
SELECT id, title, priority, created_at
FROM tasks
WHERE (priority, created_at, id) < (:p, :c, :i)
ORDER BY priority DESC, created_at DESC, id DESC
LIMIT 20;

CREATE INDEX idx_tasks_keyset
ON tasks (priority DESC, created_at DESC, id DESC);
```

There's a special case that simplifies everything: if your primary key is already time-ordered, you can paginate by it alone and skip the tiebreaker. That's the appeal of **UUIDv7** (standardized in 2024) and ULID-style keys: they embed a time component in the most significant bits, so they sort chronologically and are unique at the same time. With such a key, `ORDER BY id DESC` and a single-column cursor are enough, without exposing a

guessable sequential counter.

```
-- With UUIDv7 as PK: the id already sorts by time and is unique
SELECT id, title FROM articles
WHERE id < :cursor_id
ORDER BY id DESC
LIMIT 20;
```

Mind one security nuance: a sequential id (bigserial) exposed in the cursor reveals the volume and pace of record creation (the classic "invoice number attack"). Opaque cursors mitigate casual reading, but if the data is sensitive, prefer unguessable keys like UUIDv7 from the design stage.

Exports and ETL: walking huge tables

Keyset isn't just for UIs. It's the best way to walk an entire table in batches for an export, a migration or an ETL pipeline, without holding a giant transaction open or suffering the quadratic cost of paginating with OFFSET batch after batch.

```
def walk_everything(db, batch=1000):
    cursor = None
    while True:
        if cursor:
            rows = db.fetch(
                """SELECT id, data FROM events
                   WHERE id > %s ORDER BY id ASC LIMIT %s""",
                (cursor, batch))
        else:
            rows = db.fetch(
                """SELECT id, data FROM events
                   ORDER BY id ASC LIMIT %s""", (batch,))
        if not rows:
            break
        yield from rows
        cursor = rows[-1]["id"]
```

Each batch is a short, independent query that uses the index, so the full walk is linear in the table's size, not quadratic. As a bonus, by not depending on a single long transaction, you don't block autovacuum or accumulate a huge snapshot; if the process crashes, you resume from the last saved cursor instead of starting over. For a table of hundreds of millions of rows, this is the difference between an export that finishes and one that never does.

Consistency, isolation and changing data

A question that shows up late, already in production: what about consistency while the user paginates and the table changes underneath? Keyset already solves OFFSET's worst symptom -skipping or repeating rows when records are inserted or deleted- because each page is anchored to a real value, not a position. If a new row comes in "above" your cursor, it doesn't affect the pages you've already walked: the cursor still points to the same logical spot.

That doesn't mean you see a perfectly coherent "snapshot" of the whole table across a paging session. Each page is an independent transaction, so between page 3 and page 4 the world may have changed. For the vast majority of feeds and listings, that's exactly what you want: the user sees the latest when they request each page. But if you need a strictly consistent view -a financial report walking millions of rows that can't mix two states of the world- you have two paths.

- **Snapshot via a long transaction:** you open a `REPEATABLE READ` transaction and paginate within it. You see a frozen picture, but you hold a snapshot open that stalls autovacuum and doesn't scale to user sessions; it's for a batch export, not a UI.
- **Anchor to a moment:** you add `AND created_at <= :session_start` to every page, using the first

request's timestamp as a ceiling. That way the walk ignores everything inserted after the user started, without long transactions. It's the technique many feeds use so infinite scroll doesn't "jump" when new posts arrive.

For deletions, keyset is naturally robust: if the cursor's row is deleted, the `<` comparison still works because it compares values, not the existence of that particular row. The next page brings whatever comes after those values, whether or not the original row exists. You need no tombstones or special logic; it's a quiet advantage over any position-based scheme.

OFFSET vs Keyset: a comparison summary

To keep it handy, here's how the two techniques stack up across the dimensions that matter.

- **Deep performance:** OFFSET degrades linearly with depth; keyset stays flat regardless of the page.
- **Consistency with changing data:** OFFSET skips and repeats rows on inserts/deletes; keyset anchors to real values and doesn't suffer that shift.
- **Jumping to an arbitrary page:** OFFSET allows it trivially; keyset doesn't (it's sequential, "next/previous").
- **Total rows and "page X of Y":** easy with OFFSET; with keyset you must estimate or defer it.
- **Implementation complexity:** OFFSET is trivial; keyset asks you to think about the tiebreaker, the index, NULLs and the cursor format.
- **API coupling:** OFFSET exposes stable page numbers; keyset exposes opaque cursors you can version.

The honest read: there's no absolute winner, there's a winner per context. For an admin panel with thousands of rows and a need for page numbers, OFFSET is the right and simpler tool. For anything that grows and gets navigated deep -feeds, timelines, public APIs, exports- keyset is the option that won't betray you at scale. The expensive mistake isn't choosing one or the other, but choosing OFFSET out of inertia for a system that clearly will need keyset, and finding out only when the table already has millions of rows and thousands of clients depending on the old contract.

Why the jump is cheap: anatomy of a B-tree

To truly trust keyset it helps to understand why "jumping to the cursor" is so cheap. PostgreSQL's default index is a B-tree: a balanced tree where each node points to ranges of values, and the leaves, at the end, hold the entries in sorted order. Finding a value (or the point where it would start) is descending from the root node to the right leaf, and that costs on the order of `log(n)` steps. On a billion-row table, that logarithm is just a few levels: the engine reaches the cursor's position touching a handful of pages, not millions.

Once at the right leaf, the entries are chained in order, so reading the next 20 is following the thread: sequential, cheap, predictable reads. That's the asymmetry that explains everything. OFFSET 1,000,000 forces walking a million entries from the start because the engine has no way to know, without counting them, which to discard. Keyset's cursor, by contrast, hands the B-tree exactly the key it needs to descend straight down. It's not magic or a hand-optimized case: it's the data structure doing what it does best.

This also explains why index direction matters so much. A B-tree created as `(created_at DESC, id DESC)` has its leaves chained in that direction; if your query asks for the reverse order, the engine can walk it backward, but if you ask for a combination of directions the index doesn't reflect, there's no longer a single sorted thread to follow and a `Sort` appears that ruins the party. Matching index and `ORDER BY` isn't a formality: it's what keeps access in "descend and follow the thread" instead of "read everything and sort".

The cost on the other side: indexes and writes

No index is free. Each one you add speeds up reads but makes every `INSERT`, `UPDATE` and `DELETE` more

expensive, because the engine has to maintain that structure too, and it takes space on disk and in cache. For keyset this matters for two concrete reasons.

First, the covering index with `INCLUDE` is tempting but bloats: if you put three large columns in the payload, every update of those columns rewrites index entries and each index page caches fewer useful entries. Include only what you return in the listing; for everything else, the deferred join usually performs better. Second, if your write pattern is heavy (many inserts per second), the index gradually accumulates bloat, and it's worth watching its size and, if needed, rebuilding it with `REINDEX CONCURRENTLY` without locking the table.

```
-- Size of an index and the table, to watch for bloat
SELECT
  pg_size_pretty(pg_relation_size('idx_articles_keyset')) AS index_size,
  pg_size_pretty(pg_relation_size('articles'))            AS table_size;

-- Rebuild without blocking reads/writes (PostgreSQL 12+)
REINDEX INDEX CONCURRENTLY idx_articles_keyset;
```

The balancing rule is simple: index what you actually paginate and query, not "just in case". A well-chosen keyset index -the ordering columns, ending in the primary key, with the right direction- is usually all you need, and its write cost is perfectly acceptable against what it saves you on reads. The antipattern is hoarding half a dozen overlapping indexes "just in case": you pay the write cost of all of them and use two.

Conclusion

Keyset pagination isn't an exotic trick: it's the right way to paginate in any system that's going to grow. You swap "count and discard" for "jump straight to the cursor", back that query with an index that matches your order, and suddenly latency goes flat and results stay consistent even with changing data. The price is modest: you lose free "go to page N" and have to think about the tiebreaker, NULLs and the cursor format. In return, you get pagination that performs the same on row 20 as on row 20 million. If you're designing an API or a feed today, start with keyset; your future self, with the table already in the millions of rows, will thank you.