OFFSET Pagination Gets Slower Every Page. Keyset Pagination Doesn't. Here Are the Numbers.

Unpublished spec sample. All figures below come from the attached script (`keyset_bench.py`*), run on PostgreSQL 16.4 on an 8-vCPU Linux box. Run it yourself and your absolute times will differ, but the shape won't.*

"Page 40,000 of results is slow" is one of those tickets that sits in the backlog for months because nothing is technically broken. The query is indexed. The plan uses the index. It's still slow, and it gets slower the deeper someone scrolls.

The cause is almost always `LIMIT ... OFFSET ...`. This article measures exactly how much it costs, shows the keyset (or "seek") query that replaces it, and covers two correctness traps: one that OFFSET has without anyone noticing, and one that a naive keyset query introduces.

The test table

Two million orders with a composite index on the sort key:

```
CREATE TABLE orders (
  id          bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  customer_id int           NOT NULL,
  created_at  timestamptz   NOT NULL,
  total       numeric(10,2) NOT NULL
);

CREATE INDEX orders_created_id ON orders (created_at, id);
```

The data deliberately puts four orders on every `created_at` value. Real order tables have timestamp ties too (batch imports, second-precision clocks), and ties are where naive pagination quietly breaks.

The OFFSET query

```
SELECT id, created_at, total
FROM orders
ORDER BY created_at, id
LIMIT 50 OFFSET 1000000;
```

This looks like "jump to row one million and read 50." That's not what PostgreSQL does. An index can't jump to the N-th entry, so the executor walks the index from the start, fetches each row, and discards it until it has skipped the offset. Here's the plan for the deepest page we tested (offset 1,900,000):

```
Limit  (actual time=328.814..328.824 rows=50 loops=1)
  Buffers: shared hit=9092 read=12162
  ->  Index Scan using orders_created_id on orders
```

```
        (actual time=0.003..270.111 rows=1900050 loops=1)
Execution Time: 328.840 ms
```

Look at `rows=1900050`. To return 50 rows, it read 1.9 million and touched more than 21,000 buffers.

The keyset query

Keyset pagination doesn't count rows to skip. It remembers where the last page ended and asks for what comes after it:

```
SELECT id, created_at, total
FROM orders
WHERE (created_at, id) > ($1, $2)   -- values from the last row of the previous page
ORDER BY created_at, id
LIMIT 50;
```

Same deep page, same index:

```
Limit  (actual time=0.013..0.023 rows=50 loops=1)
  Buffers: shared hit=4
  ->  Index Scan using orders_created_id on orders
        Index Cond: (ROW(created_at, id) > ROW('2024-03-23 12:09:45-06', 1899999))
Execution Time: 0.037 ms
```

The row comparison becomes an `Index Cond`. PostgreSQL descends the B-tree straight to the boundary and reads 50 entries. Four buffers.

The numbers

Median of 7 runs per point, `EXPLAIN (ANALYZE, BUFFERS)` execution time. The script also checks that both queries return identical rows at every depth.

Rows skipped	OFFSET (ms)	OFFSET buffers	Keyset (ms)	Keyset buffers
0	0.016	4	0.015	4
1,000	0.172	15	0.019	5
10,000	1.458	115	0.017	4
100,000	14.675	1,122	0.019	4
500,000	75.671	5,595	0.018	4
1,000,000	153.252	11,188	0.020	5
1,900,000	344.951	21,253	0.022	4

OFFSET cost grows linearly with depth: about 0.15 ms per thousand rows skipped on this machine. Keyset stays flat at roughly 0.02 ms no matter where you are. On page 1 the two are identical, so testing on the first page never shows the problem.

A third of a second doesn't sound dramatic until you remember that crawlers, exports and "load more" loops request every page in order. Paging through all 2 million rows at 50 per page with OFFSET does about 40,000 queries whose average

cost is half the table. With keyset, every one of those queries is the cheap one.

Bug #1 that keyset fixes: duplicates when rows are inserted

OFFSET pages are defined by position, so any insert ahead of the reader shifts everything. The script fetches page 1, inserts a single older order, then fetches page 2:

- **OFFSET page 2:** 1 row that already appeared on page 1 (the last row got pushed across the boundary).
- **Keyset page 2:** 0 duplicates. The cursor is a value, not a position, so the insert doesn't move it.

With deletes it's the mirror image: rows silently skipped instead of repeated. On a busy table, users of OFFSET pagination are seeing this today and blaming the front end.

Bug #2 to avoid: a cursor without a tie-breaker

The common first attempt at keyset pagination uses only the sort column:

```
WHERE created_at > $1   -- wrong when created_at has ties
```

If the previous page ended halfway through a group of rows sharing one timestamp, the rest of that group is skipped forever. In our data, the boundary after page 3 falls inside such a group, and the timestamp-only cursor loses 1 row there. With four-way ties a page boundary can lose up to 3. The fix is the one used above. Add a unique column as the last sort key and compare as a row value:

```
WHERE (created_at, id) > ($1, $2)
ORDER BY created_at, id
```

Two details matter:

1. **The index must match the sort order exactly**: `(created_at, id)`, same column order. Then the row comparison is used as an index condition rather than a filter.
2. **Write the row-value form, not the expanded `OR`.** `created_at > $1 OR (created_at = $1 AND id > $2)` is logically the same, but PostgreSQL treats it as a `Filter`, not an index condition. We tried it at the 1,000,000-row boundary. The planner still walked the index from the start (`Rows Removed by Filter: 1000000`) and took 154.9 ms, which is no better than OFFSET.

For descending order, flip both: `ORDER BY created_at DESC, id DESC` with `WHERE (created_at, id) < ($1, $2)`. Mixed directions (`created_at DESC, id ASC`) can't use a single row comparison. Keep both directions the same.

What keyset pagination costs you

- **No "jump to page 731."** You can go next and previous, and to the first and last page. Arbitrary page numbers need OFFSET or a precomputed position. For search results and feeds, nobody misses page numbers. For admin grids, keep OFFSET for the first few pages and switch to keyset past a depth limit.

- **The API returns a cursor instead of a page number.** Usually it's the last row's `(created_at, id)` encoded as an opaque string. Clients pass it back and never parse it.
- **Total counts are a separate question.** `count(*)` over 2 million rows is its own cost either way. Show "50 of many" or use an estimate.

The takeaway

If a list endpoint can be paged deep, by users, crawlers or export jobs, replace OFFSET with a row-value keyset predicate on an index that matches the `ORDER BY`, and include a unique tie-breaker. On this 2-million-row table that took the deepest page from 345 ms to 0.02 ms, removed the duplicate-row bug, and changed one `WHERE` clause.

Reproduce: `DSN="host=... dbname=scratch" python keyset_bench.py` (needs `psycopg2`). It creates and drops a table named `orders` in the target database. Use a scratch database.