Sorting, Limiting, and Pagination

ORDER BY, LIMIT/OFFSET, and NULLS FIRST/LAST -- what Pageable/Sort/Page<T>, already covered in the Spring Data JPA course, actually compiles down to underneath, on this project's own real topic table. The 8th lesson in the PostgreSQL Foundations category.

Intermediate 20 min

Last updated: August 25, 2026
Read online: https://www.learnforgex.com/en/topics/sorting-limiting-and-pagination

---

"Pagination, Sorting, and Projections," back in the Spring Data JPA course, covered `Pageable`, `Sort`, and `Page<T>` — a Java API resolved, at the controller level, into a repository call. This lesson looks at what that repository call actually becomes once it reaches PostgreSQL: raw `ORDER BY`, `LIMIT`, and `OFFSET`, on this project's own real `topic` table.

ORDER BY: Sorting Rows

Without an `ORDER BY`, PostgreSQL makes no promise about row order at all — not "insertion order," not "primary key order," genuinely unspecified, and it can change between runs of the identical query. `ORDER BY` is what makes order a guarantee rather than an accident:

```
SELECT slug, sort_order FROM topic
WHERE category_id = 1
ORDER BY sort_order;
```

Run against this project's own real `postgresql-foundations` category, this returns every topic covered so far, genuinely in `sort_order` — the exact column "Databases, Schemas, Tables, and Basic SQL Syntax" onward have all had, and the exact value this course's own migrations set explicitly for every topic (`sort_order = 1` for the first lesson, `2` for the second, and so on).

Multiple Sort Keys and Direction

`ORDER BY` accepts more than one column, each with its own direction — `ASC` (ascending, the default) or `DESC` (descending):

```
SELECT slug, difficulty, sort_order FROM topic
ORDER BY difficulty ASC, sort_order DESC;
```

Sorting is applied left to right: rows are grouped by `difficulty` first, and only *within* each identical `difficulty` value are they then ordered by `sort_order` — the second column only breaks ties left by the first, it doesn't independently re-sort the whole result.

LIMIT and OFFSET

`LIMIT` caps how many rows come back; `OFFSET` skips a number of rows before starting to return them:

```
SELECT slug FROM topic
ORDER BY sort_order
LIMIT 5 OFFSET 0;   -- the first 5 rows

SELECT slug FROM topic
ORDER BY sort_order
LIMIT 5 OFFSET 5;   -- the next 5 rows
```

`LIMIT` / `OFFSET` only make sense paired with an `ORDER BY` — without one, "the first 5 rows" and "the next 5 rows" aren't well-defined concepts to begin with, since row order itself isn't guaranteed.

Paging Through Results: A Real Example

Chaining `ORDER BY` with `LIMIT` / `OFFSET` at increasing offsets is exactly how a page of results is built, one page at a time:

```
-- Page 1 (first page, 3 topics per page)
SELECT slug FROM topic
ORDER BY sort_order
LIMIT 3 OFFSET 0;

-- Page 2
SELECT slug FROM topic
ORDER BY sort_order
LIMIT 3 OFFSET 3;

-- Page 3
SELECT slug FROM topic
ORDER BY sort_order
LIMIT 3 OFFSET 6;
```

Run against this project's own `postgresql-foundations` category, page 1 returns this course's first three lessons; page 3 lands on this very lesson and its immediate neighbors. `OFFSET` for page *n* (zero-indexed) with a page size of `s` is always `n * s` — the same arithmetic a paginated API endpoint does internally before it ever reaches SQL.

The Cost of OFFSET on Large Tables

`OFFSET` doesn't skip rows for free — PostgreSQL still has to scan and discard every one of them before it can start returning the rows actually wanted, so `OFFSET 100000` does meaningfully more work than `OFFSET 10`, even though both return the same number of rows. This project's own tables are far too small for the difference to matter, but it's worth naming now: "Indexes and Query Performance with EXPLAIN," later in this course, comes back to this specifically, along with a faster alternative (keyset pagination) for cases where it does.

From Raw SQL to Pageable/Page: What Spring Data JPA Does Underneath

"Pagination, Sorting, and Projections" already covered `Pageable` / `Sort` / `Page<T>` on the Java side — a `PageRequest.of(1, 3, Sort.by("sortOrder"))` passed into a repository method is exactly the Java-level equivalent of the page-2 query above, and Spring Data JPA translates it into precisely this shape: an `ORDER BY sort_order`, a `LIMIT 3`, and an `OFFSET 3`, plus a second `SELECT count(*)` query behind the scenes to compute `Page<T>`'s total element/page counts. Nothing new needs explaining about the Java API itself here — what matters is recognizing that `Pageable` was never a separate mechanism from `LIMIT` / `OFFSET`, only a typed wrapper generating exactly the SQL this lesson just wrote by hand.

NULLS FIRST / NULLS LAST

PostgreSQL sorts `NULL` values as larger than any real value by default, which means a plain `ORDER BY estimated_minutes` places every `NULL` at the *end* in ascending order, and at the very *start* in descending order — worth knowing explicitly, since this project's own `topic.estimated_minutes` is nullable ("PostgreSQL Data Types" and "SELECT and Filtering" both used it as the running example of a nullable column). `NULLS FIRST` / `NULLS LAST` override that default directly:

```
SELECT slug, estimated_minutes FROM topic
ORDER BY estimated_minutes ASC NULLS LAST;
```

Every real row in this project's own `topic` table happens to have `estimated_minutes` set, so this specific query wouldn't currently surface any `NULL`s in practice — but the clause is there precisely for the column type ("PostgreSQL Data Types" already established it's genuinely nullable) rather than for any specific row that exists today.

Common Misconceptions

"Rows come back in the order they were inserted, unless told otherwise." Not guaranteed at all — PostgreSQL is free to return rows in whatever order its query plan finds cheapest, which often does resemble insertion order on a small, simple table, which is exactly what makes this misconception feel true until a table grows or a query plan changes. **"`LIMIT` without `ORDER BY` reliably returns 'the first N' rows."** It returns *some* N rows, but which N, and in what order, is unspecified — "first" implies an order that was never established. **"`Pageable` and raw SQL pagination are different features."** They're the same feature at two different layers — `Pageable` is a typed way to produce the exact `ORDER BY` / `LIMIT` / `OFFSET` this lesson covers, not an alternative mechanism.

Best Practices

- Always pair `LIMIT` / `OFFSET` with an explicit `ORDER BY` — this project's own examples never use one without the other, since "the first 5" and "the next 5" are meaningless without a defined order to be first or next *in*.
- List multiple `ORDER BY` columns in the order that actually matters for tie-breaking, and make sure the first column alone doesn't already make later ones irrelevant (sorting by a `UNIQUE` column first, for instance, leaves nothing left to break ties on).
- Be explicit about `NULLS FIRST` / `NULLS LAST` on any nullable sort column rather than relying on PostgreSQL's default (`NULL`s sort last ascending, first descending) — a reader shouldn't have to remember that rule to predict a query's output.
- Recognize `Pageable`'s `PageRequest.of(page, size, sort)` as directly computing `LIMIT size OFFSET page * size` plus an `ORDER BY` — reasoning about a slow paginated endpoint means reasoning about this SQL shape, not a separate Java-level cost model.

Common Mistakes

- Passing a page size and offset combination where the offset was computed with the wrong page size (mixing a page-size-10 offset formula against a page-size-3 query, for instance) — the query runs without error and returns a page of genuinely wrong rows, not an empty result that would be obvious to notice.
- Assuming a `LIMIT` without `ORDER BY` is deterministic across repeated runs of an identical query — it can silently return a different set of rows the next time the same statement runs, especially after the underlying data changes.
- Forgetting that `OFFSET`'s cost grows with the offset itself, then being surprised a "later page" of a paginated endpoint is measurably slower than page one on a large table — not a bug, but a real, well-known cost this lesson names and "Indexes and Query Performance with EXPLAIN" returns to.

- Sorting by a nullable column without deciding where `NULL`s should land, then having them appear at an unexpected end of the result depending only on `ASC` vs. `DESC` — a decision that should be made explicitly with `NULLS FIRST` / `NULLS LAST`, not left to the default.

Summary, Cheat Sheet, and Glossary

Summary

- Row order is never guaranteed without an explicit `ORDER BY`; multiple sort columns apply left to right, each breaking ties left unresolved by the one before it.
- `LIMIT` caps the number of rows returned; `OFFSET` skips rows before returning them — both only meaningful alongside an `ORDER BY`.
- Paging through results is `ORDER BY` plus `LIMIT <size> OFFSET <page * size>`, repeated at increasing offsets — exactly what `Pageable`, already covered in "Pagination, Sorting, and Projections," compiles down to.
- `OFFSET` isn't free — PostgreSQL scans and discards every skipped row, so cost grows with the offset itself; "Indexes and Query Performance with EXPLAIN," later in this course, covers this properly along with a faster alternative.
- `NULL` sorts as larger than any real value by default (last ascending, first descending); `NULLS FIRST` / `NULLS LAST` make the placement explicit on any nullable sort column, like this project's own `topic.estimated_minutes`.

Cheat Sheet

```
SELECT a FROM t ORDER BY a;
SELECT a FROM t ORDER BY a DESC;
SELECT a, b FROM t ORDER BY a ASC, b DESC;
SELECT a FROM t ORDER BY a LIMIT 10;
SELECT a FROM t ORDER BY a LIMIT 10 OFFSET 20;
SELECT a FROM t ORDER BY a NULLS LAST;
```

```
page n (zero-indexed), size s  →  LIMIT s OFFSET (n * s)
```

Glossary

- **ORDER BY**: the clause that establishes a guaranteed row order; without it, row order is unspecified.
- **LIMIT**: caps the number of rows a query returns.
- **OFFSET**: skips a number of rows before a query starts returning them; costs work proportional to the number skipped.
- **Keyset pagination**: a faster alternative to `OFFSET`-based paging for large tables, named here and covered properly in "Indexes and Query Performance with EXPLAIN."