

The database wasn't slow. It was being asked too much.

An internal back office for an energy retailer, built on Laravel and Filament over Oracle 11g. **Every page of the admin panel ran 928 database queries and took 22 seconds to load.**

QUERIES PER PAGE	PAGE LOAD	MENU QUERIES	HOTSPOTS
928 → 36 same page, before and after	22.3 s → 2.7 s measured in development	200+ → 1 one cached call	3 menu, record pages, bulk import

ROLE **Senior backend contractor** on an ongoing engagement — diagnosis and fixes across the admin panel, record pages and a bulk import flow.

CORE STACK PHP 8 · Laravel · Filament · Livewire · Oracle 11g · Redis

NOTE *Client withheld. Figures are from measurements recorded in the project, taken in the development environment; where no after-figure was recorded, none is claimed.*

THE SYMPTOMS

Three places hurt. Every page in the admin panel was slow, whatever it showed. One record's status page ran 930 queries on its own. And the processing button on a bulk import would hang: the spinner turned and the work never visibly finished.

The obvious reading was a slow database, or an underpowered server. The numbers pointed elsewhere: the same questions were being asked hundreds of times per click.

COUNT FIRST, THEN FIX

The first step was measurement: a per-request query counter that also flagged duplicates, with page timings recorded before and after each change. Duplicates are the useful part: a query that runs once is a cost, but the same query running two hundred times is a pattern, and patterns have a single cause you can remove.

Every fix below started from a number, and every claimed result is one that was measured.

01 A menu that queried the database once per screen

The admin navigation supported per-screen overrides, and every Filament resource looked up its own override separately — over two hundred queries just to draw the menu, on every page, before the page's own content had loaded. The fix was to load all overrides in one query, cache the result in Redis, and invalidate the cache whenever an override is saved or deleted. The same page went from **928 queries to 36**, and from **22.3 seconds to 2.7**.

02 Lazy loading on the record pages

The status page loaded related records one access at a time: location data for the customer, a redundant user lookup, and workflow-step records re-queried instead of reused. Classic N+1. Fixed with eager loading, related models fetched in a single query, and the step-status collection cached. The before-figure was 930 queries; no after-figure was recorded, so I don't quote one.

03 A bulk import doing one round trip per row

The import ran synchronously inside the web request, and for every row it checked whether the record existed, inserted it, and fetched reference data again — each a separate round trip across a network link. I moved the work to set-based SQL: a single *NOT EXISTS* anti-join instead of a per-row check, inserts batched 500 rows at a time with a per-row fallback on error, reference data loaded once and matched in memory, and two configuration reads merged into one.

One cause was specific to the database version. Laravel's *chunk()* pages with *OFFSET*, which Oracle 11g emulates by wrapping the query in *ROWNUM* — so every chunk re-scanned the entire source view from the top. On the larger dataset, that meant dozens of full scans per click. Streaming the view once with a cursor, and paging keyed tables by ID instead of offset, removed them. Duration logging went in with the fix; before/after timings for this flow were not recorded.

WHAT THIS MEANS IF YOU'RE HIRING

Slow applications are rarely slow because of the hardware. They're slow because of a handful of patterns repeated thousands of times, and finding them takes someone willing to count before guessing. The ORM makes each query easy to write; it doesn't show what they add up to.

None of this needed a new server, a rewrite or a framework change. Every fix lives in the application layer, where the questions were being asked, so each one could be reviewed, shipped and verified on its own.

Richard Trujillo • Desemax. Senior backend and AI-integration engineer: PHP / Laravel / Symfony, TypeScript / Node / Bun, real-time and LLM-backed systems, Python data pipelines. Eleven years of backend engineering, independent since 2015. Based in Milan, working across Europe.

richard@desemax.it · richardtrujillo.desemax.it