DataShelf Hub

DATABASES & DATA SYSTEMS

The Query Performance Playbook

Indexes, execution plans and the art of a fast SELECT

● FREE SAMPLE CHAPTER

Omar El Alaoui
Data & systems engineer

PDF · CH. 01

ABOUT THIS SAMPLE

A free first chapter

What follows is Chapter 1 of *The Query Performance Playbook*, reproduced in full and provided at no cost by DataShelf Hub. It is unabridged — nothing has been trimmed or summarised. If it's useful to you, the remaining seven chapters continue exactly where this one leaves off.

This sample is for your personal use while you decide whether the book is right for you. Please don't republish or redistribute it — share a link to the book's page instead.

TITLE	The Query Performance Playbook
AUTHOR	Omar El Alaoui
SHELF	Databases & Data Systems
FORMAT	PDF, DRM-free, 248 pages
THIS SAMPLE	Chapter 1 of 8 — 14 pages
PUBLISHER	DataShelf Hub · datashelfhub.com

WHAT'S IN THE FULL BOOK

Table of contents

8 chapters · 24 sections

01 **How the Database Thinks** — In this sample

- What the optimizer is solving
- Rows, pages and I/O
- Reading a slow query's story

02 **Reading Execution Plans**

- Scan versus seek
- Join algorithms compared
- Estimated versus actual rows

03 **Indexes That Earn Their Keep**

- B-tree fundamentals
- Composite and covering indexes
- When an index does more harm than good

04 **Queries the Planner Likes**

- Sargable predicates
- Avoiding accidental scans
- Rewriting subqueries and CTEs

Continued on the next page →

TABLE OF CONTENTS, CONTINUED

Chapters 5–8

05

Joins, Aggregation and Sorting

- Join order and selectivity
- Grouping efficiently
- Sorts that spill to disk

06

Statistics, Cardinality and Skew

- Keeping stats fresh
- Skewed data and bad plans
- Hints and plan stability

07

Transactions and Contention

- Isolation levels in practice
- Lock waits and deadlocks
- Designing away contention

08

A Tuning Workflow

- Finding the queries that matter
- Measuring before and after
- Knowing when to stop

CHAPTER ONE

01

How the Database Thinks

A query is a request, not an instruction — the database decides how to satisfy it, and tuning starts with understanding that decision.

You write `SELECT * FROM orders WHERE customer_id = 42`. It feels like an instruction: go here, filter this, return that. It isn't. It's a description of the result you want, handed to a piece of software called the query optimizer, which is free to satisfy that description any way it sees fit — scan the whole table, use an index, use a different index you forgot existed, read the data in a completely different order than you wrote the query in. You specified the what. The database decided the how, and it decided in about a millisecond, before your query ever touched a single row.

Most query tuning advice skips straight to "add an index here" without explaining that decision, which makes the advice feel like superstition — sometimes it helps, sometimes it doesn't, and you're never quite sure why. This chapter is about the decision itself: what the optimizer is actually trying to do, what it has to work with, and why understanding that turns tuning from guesswork into something closer to reading.

1.1 What the Optimizer Is Solving

Give the optimizer a query and it has to answer one question: of every physically possible way to produce this result, which one will cost the least? "Cost" here isn't wall-clock time — the optimizer never runs your query to find out how long it takes. It's an estimate, built from statistics about your data, of how much work — how many rows examined, how many pages read from disk or cache — each candidate plan would require.

For anything beyond a trivial query, the space of possible plans is enormous. A three-table join alone has six possible join orders, each combinable with different join algorithms and different access paths per table, and the optimizer can't afford to cost every single combination — it prunes aggressively, using heuristics and statistics, and settles on a plan it believes is good, not one it's proven is optimal.

> ***"The plan you get is the optimizer's best guess under time pressure, built from statistics about data it hasn't actually looked at yet. Tuning is the art of giving that guess better information."***

This is the single most useful reframe in query tuning: a slow query is rarely the database being slow. It's usually the optimizer making a reasonable decision from bad or missing information — stale statistics, a missing index it would have used if one existed, a predicate shaped in a way that hides how selective it actually is. Fix the information, and the same optimizer, unchanged, often produces a dramatically better plan on its own.

That reframe matters because it changes what you look for when a query is slow. The instinct is to look at the query's logic — is the JOIN wrong, is the WHERE clause wrong — but the query can be logically perfect and still run for eight seconds because the optimizer chose a full scan over an index it didn't know would help, or misjudged how many rows a filter would match by three orders of magnitude.

WHAT THE OPTIMIZER ACTUALLY HAS TO WORK WITH

Table & column statistics	row counts, value distributions, null fractions — usually sampled, not exact
Available indexes	which access paths exist at all — it can't use an index you didn't create
Cost model	rough estimates of I/O and CPU per operation, calibrated to typical hardware

Every chapter after this one is really about improving one of those three inputs. Chapter 3 is about giving the optimizer better access paths through indexes. Chapter 6 is about keeping its statistics honest so its estimates don't drift away from reality. Chapter 4 is about shaping queries so the optimizer can actually recognise when an index applies, instead of writing something logically equivalent but invisible to it.

None of that is guesswork once you see it this way. It's closing the gap between what the optimizer believes about your data and what's actually true, because every bad plan traces back to a gap somewhere in that belief.

1.2 Rows, Pages and I/O

The unit the optimizer actually reasons in isn't rows — it's pages. Your table doesn't live on disk as a list of rows; it lives as a sequence of fixed-size pages, typically 8KB, each packed with as many rows as fit. When the database needs a row, it doesn't fetch that row — it fetches the whole page containing it, and every other row that happened to be packed alongside it.

This is why "how many rows does this query touch" is the wrong question, and "how many pages does this query have to read" is the right one. A query that touches a thousand scattered rows across a thousand different pages does a thousand page reads. A query that touches the same thousand rows, but they're clustered together on twenty pages, does twenty. Same row count, fifty times the I/O difference — and I/O, not row count, is what actually costs time.

WHY LOCALITY CHANGES EVERYTHING

Scattered rows	1,000 rows on 1,000 pages = 1,000 page reads
Clustered rows	1,000 rows on 20 pages = 20 page reads
Why it happens	insertion order, index structure, and physical clustering all affect where rows land

This is also why an index doesn't automatically make a query fast. An index gives the database a fast way to find which rows match — but if those rows are scattered across many pages, satisfying the query still means many scattered page reads, sometimes more expensive overall than reading the table sequentially would have been. The optimizer knows this, which is exactly why it sometimes ignores an index you were sure it should use — the index would find the rows quickly, but fetching them afterward would still be slow.

Understanding this reframes what a "covering index" — a structure holding everything a query needs without a separate trip back to the table — is actually buying you: it's not just avoiding a lookup, it's avoiding the scattered-page-read problem entirely, because the answer was assembled from one dense, sequential structure instead of jumping around the table.

WORTH REMEMBERING

Cache and buffer pools blur this in practice — a "hot" page already in memory costs nothing to re-read. But the underlying physics don't go away; they just move from disk latency to memory bandwidth, and at scale, locality still wins.

1.3 Reading a Slow Query's Story

Put the two ideas together — cost as an estimate built from imperfect statistics, and I/O as pages rather than rows — and a slow query stops looking like a mystery and starts looking like a story you can read, provided you know where to look: the execution plan the optimizer actually chose, which is the subject of the next chapter in full detail.

For now, the shape of that story is usually one of a small number of plots. The optimizer underestimated how many rows a filter would match, so it chose a plan sized for a small result and got flooded. It had no index for the predicate you're filtering on, so it fell back to reading every page in the table. It found rows efficiently but they were scattered across the table, so the lookup phase dominated the actual work.

Every one of those stories has a specific, legible fix once you can see it: refresh statistics, add an index, restructure the query so an existing index becomes usable, or accept the scattered reads and add a covering index that removes them. None of it is guessing. It's diagnosis, and the tool for that diagnosis — the execution plan — is something every engine exposes, whether you're running Postgres, MySQL, or something else entirely.

That's the discipline this book is building toward: not a list of tricks to try in order, hoping one sticks, but a way of reading what the database actually did and why, so the fix you reach for is the one the evidence points to. Chapter 2 starts there, with the execution plan itself — how to read one, and what it's really telling you.

§ Chapter Summary

Seven things worth carrying into Chapter 2.

BEFORE YOU MOVE ON

01 A query describes a result; the optimizer decides how to produce it, based on statistics that may or may not reflect reality.

02 A slow query is rarely the database being slow — it's usually a reasonable decision made from bad or missing information.

03 The optimizer reasons in pages, not rows. Scattered rows across many pages cost far more I/O than the same rows clustered together.

04 An index doesn't automatically make a query fast — it speeds up finding rows, but fetching scattered rows afterward can still dominate the cost.

05 Every slow query has a legible story: bad estimates, a missing index, or scattered reads. Diagnosis beats guessing every time.

BEFORE CHAPTER 2

Find the slowest query you know of in a system you maintain. Before looking at its execution plan, write down your best guess for why it's slow. Then read the plan in Chapter 2's terms and see how close your guess was — the gap between the two is usually exactly where your tuning instincts need sharpening.

END OF SAMPLE

Seven chapters still to go.

Chapter 1 explains what the optimizer is doing and why; the rest of the book turns that understanding into a practical toolkit — reading execution plans, designing indexes that actually get used, and a repeatable workflow for finding and fixing the queries that matter.

02 Reading Execution Plans

03 Indexes That Earn Their Keep

04 Queries the Planner Likes

05 Joins, Aggregation and Sorting

06 Statistics, Cardinality and Skew

07 Transactions and Contention

08 A Tuning Workflow