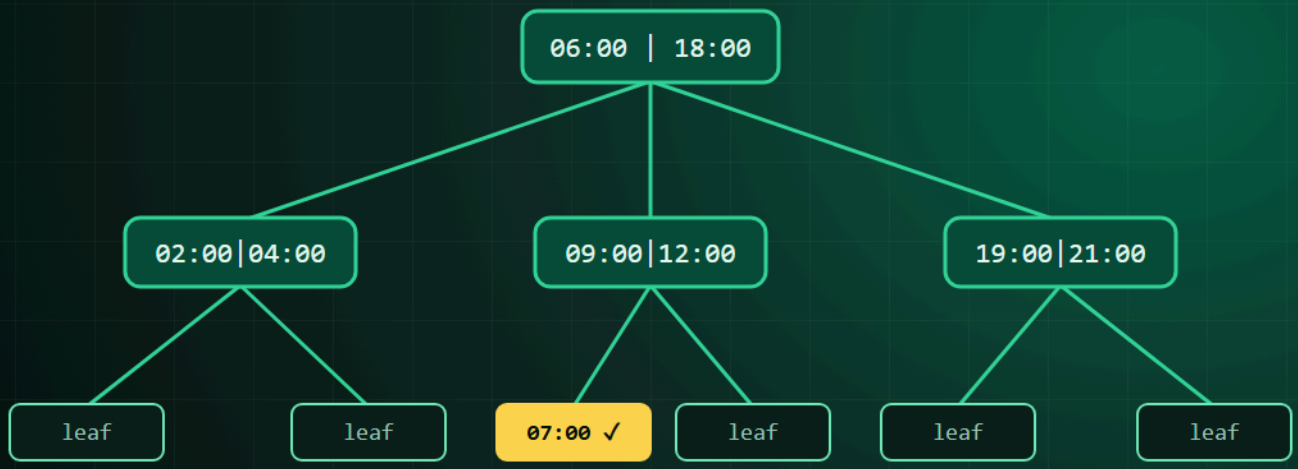
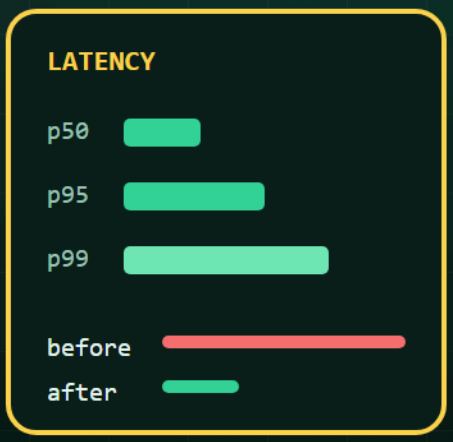



```
EXPLAIN (ANALYZE, BUFFERS)

Limit (rows=20)
  -> Index Scan using
     class_sessions_studio_id_starts_at_idx
       on class_sessions
     Index Cond: studio_id = 12
     Buffers: shared hit=5
```



index · plan · estimate · buffers · lock · vacuum

THE PRACTICAL GUIDE

Database Performance Basics

Indexes, query plans, and practical tuning for developers

With PostgreSQL and MySQL — EXPLAIN, B-trees,
N+1 queries, locking, pooling, and partitioning

MEASURE · EXPLAIN · INDEX · REWRITE · VERIFY

THENEXBOOKS

Database Performance Basics

Indexes, Query Plans, and Practical Tuning for Developers
With PostgreSQL and MySQL

First Edition

THENEXBOOKS

Database Performance Basics
Indexes, Query Plans, and Practical Tuning for Developers

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Query plans, timings, and configuration defaults vary by database version, hardware, data, and workload; all plan output and timings shown in this book are illustrative. Always test changes on your own systems and consult the official documentation for the versions you run.

PostgreSQL is a trademark of the PostgreSQL Community Association of Canada. MySQL is a trademark of Oracle and/or its affiliates. SQL Server is a trademark of Microsoft Corporation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by any of these organizations.

Wavecrest, the running example in this book, is fictional, as are all studios, members, and people mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

Almost every slow application eventually leads to the database. A page that took a few milliseconds during development takes seconds in production. A report that ran fine last quarter now times out. A booking button works for everyone except during the one minute when everyone presses it at once. The code did not change much — the data, the traffic, and the queries that nobody looked at closely did.

The good news is that most database performance problems are not mysterious. They come from a small set of causes: queries that read far more data than they need, missing or mismatched indexes, plans built on wrong estimates, too many round trips from the application, contention on hot rows, and maintenance that fell behind. Once you understand how a database stores data, how the planner chooses a plan, and how indexes actually work, you can find and fix these problems methodically instead of by guesswork.

This book teaches those foundations thoroughly and practically. It is written for developers rather than full-time database administrators: we focus on the things you control every day — schemas, indexes, queries, ORM usage, transactions, and connections — and on the few operational topics you need to understand to work well with your database, not to run it single-handedly.

Who This Book Is For

- **Application and backend developers** who write SQL directly or through an ORM and want their features to stay fast as data grows.
- **Full-stack developers** who own the database along with everything else and need a reliable method for slow-query problems.
- **Junior DBAs and data engineers** who want solid mental models of plans, indexes, and concurrency.
- **Tech leads** who review schemas and migrations and want to catch performance problems before they reach production.

What You Will Learn

- A mental model of where database time goes: pages, the buffer cache, I/O, CPU, and waiting.
- How to measure properly with latency percentiles, throughput, and baselines — before changing anything.
- How the query planner works, how to read `EXPLAIN` in PostgreSQL and MySQL, and why estimates go wrong.
- How B-tree indexes work inside, how to order composite index columns, and when to use covering, partial, expression, and specialized indexes.
- When indexes hurt, and how to build an indexing strategy for a real workload.

- Query patterns that defeat indexes, fast pagination and top-N queries, reporting queries, and materialized views.
- How to fix N+1 queries, batch work, and load or delete large amounts of data safely.
- Transactions, MVCC, locks, hot rows, deadlocks, connection pooling, vacuum, and configuration basics.
- How to find slow queries in production, partition large tables, and scale beyond one server.

How This Book Is Organized

Part I — Performance Foundations (Chapters 1–4) builds the mental model and a performance lab. **Part II — Reading Query Plans** (Chapters 5–9) explains the planner, `EXPLAIN` in both engines, statistics, and join algorithms. **Part III — Indexing** (Chapters 10–15) covers indexes from the inside out and ends with a practical strategy. **Part IV — Writing Fast Queries** (Chapters 16–20) turns that knowledge into faster SQL and application access patterns. **Part V — Concurrency and Operations** (Chapters 21–25) covers transactions, locking, connections, maintenance, and monitoring. **Part VI — Scaling and Case Study** (Chapters 26–28) covers partitioning and scaling, and walks through a complete tuning investigation. The appendices provide a cheat sheet, diagnostic query templates, a performance checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Wavecrest

Throughout the book we tune the database of **Wavecrest**, a fictional chain of fitness studios. Members book classes in a mobile app, check in at the door with a QR code, and buy memberships and class packs; studio managers run reports on attendance and revenue. That gives us every classic performance problem in one realistic schema: a booking rush when next week's schedule opens, a check-in table that grows every day, time-range queries, per-studio filtering, month-over-month reporting, and an ORM-based API that issues more queries than anyone expects. In Chapter 4 you will build the Wavecrest lab database yourself, so you can run every example.

Versions and Conventions

PostgreSQL is our primary engine and appears in every chapter; examples target PostgreSQL 16 and 17, and features introduced in later versions are labeled. MySQL 8 with the InnoDB storage engine is our second engine; where its behavior differs in a way that matters, we show the MySQL equivalent. SQL Server and other systems appear in short notes. **All `EXPLAIN` output and timings in this book are illustrative**: your numbers will differ with your hardware, data, and version. We draw conclusions from plan shapes and row counts, not from absolute milliseconds.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version details, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause slowdowns, outages, or data loss in production.

Code appears in `monospace`. Engine-specific snippets are labeled `-- PostgreSQL` or `-- MySQL`, and long plan lines are abbreviated with `...`.

Contents

Preface

PART I — PERFORMANCE FOUNDATIONS

1 Why Databases Get Slow: A Mental Model of Database Performance

2 How a Database Stores Data: Pages, Rows, the Buffer Cache, and I/O

3 Measuring Before Tuning: Latency, Throughput, Percentiles, and Baselines

4 Setting Up a Performance Lab: PostgreSQL, MySQL, Realistic Test Data, and Tools

PART II — READING QUERY PLANS

5 How the Query Planner Works: Parsing, Planning, and Cost Estimation

6 Reading EXPLAIN in PostgreSQL

7 Reading EXPLAIN in MySQL (with a Note on SQL Server Plans)

8 Statistics, Cardinality Estimates, and Why Plans Go Wrong

9 Join Algorithms: Nested Loop, Hash Join, and Merge Join

PART III — INDEXING

10 B-Tree Indexes from the Inside

11 Composite Indexes and Column Order

12 Covering Indexes, Index-Only Scans, and Clustered Tables

13 Partial, Expression, and Specialized Indexes (GIN, GiST, BRIN, Full-Text)

14 When Indexes Hurt: Write Cost, Bloat, Duplicate and Unused Indexes

15 An Indexing Strategy for Real Workloads

PART IV — WRITING FAST QUERIES

16 Query Patterns That Defeat Indexes: Sargability, Functions, Casts, OR, and LIKE

17 Pagination, Sorting, and Top-N Queries

18 Aggregation, Reporting Queries, and Materialized Views

19 Application Access Patterns: N+1 Queries, ORMs, Batching, and Round Trips

20 Bulk Loading, Batch Updates, and Large Deletes

PART V — CONCURRENCY AND OPERATIONS

21 Transactions, Isolation Levels, and MVCC Performance

22 Locks, Contention, Hot Rows, and Deadlocks

23 Connections and Connection Pooling

24 Memory, Configuration, and Maintenance: Vacuum, Autovacuum, and InnoDB Purge

25 Monitoring and Finding Slow Queries: pg_stat_statements, the Slow Query Log, and Performance Schema

PART VI — SCALING AND CASE STUDY

26 Partitioning and Archiving Large Tables

27 Scaling Beyond One Server: Caching, Read Replicas, and Sharding Basics

28 Case Study: Tuning the Wavecrest Booking Database

APPENDICES

A Database Performance Cheat Sheet

B Diagnostic Query Templates and Reference

C Performance Checklist

D Glossary

PART I

Performance Foundations

Where database time actually goes, how data is stored in pages and cached in memory, how to measure latency and throughput properly, and a hands-on performance lab with the Wavecrest schema.

1 · Why Databases Get Slow

2 · Pages, Rows, the Buffer Cache, and I/O

3 · Measuring Before Tuning

4 · Setting Up a Performance Lab

CHAPTER 1 · PART I — PERFORMANCE FOUNDATIONS

Why Databases Get Slow: A Mental Model of Database Performance

Slow databases are rarely mysterious once you know where time can go. This chapter builds the mental model the rest of the book relies on: a query's journey from the application to the disk and back, the difference between doing work and waiting, and the short list of causes behind almost every performance problem you will meet.

Every Sunday at six in the evening, Wavecrest opens next week's class schedule. Within seconds, thousands of members open the mobile app and race to book the popular early-morning cycle classes and the Saturday yoga sessions. For most of the week the booking API answers in a blink. For those first few minutes on Sunday, it crawls. Some requests time out, members complain on social media, and a studio manager who is trying to run a revenue report at the same moment gives up and goes home.

Nobody changed the code on Sunday. The database server is the same one that was fast on Tuesday. So what happened? Answering that kind of question is what this book is about, and it starts with a way of thinking rather than a list of tricks. If you have a clear picture of what a database does when it runs a query, most slowdowns stop being mysterious. You can look at a symptom, form a hypothesis about which part of the machine is struggling, and test it. Without that picture, tuning becomes guesswork: adding indexes at random, increasing memory settings because a blog post said so, or rewriting queries without knowing whether the rewrite helps.

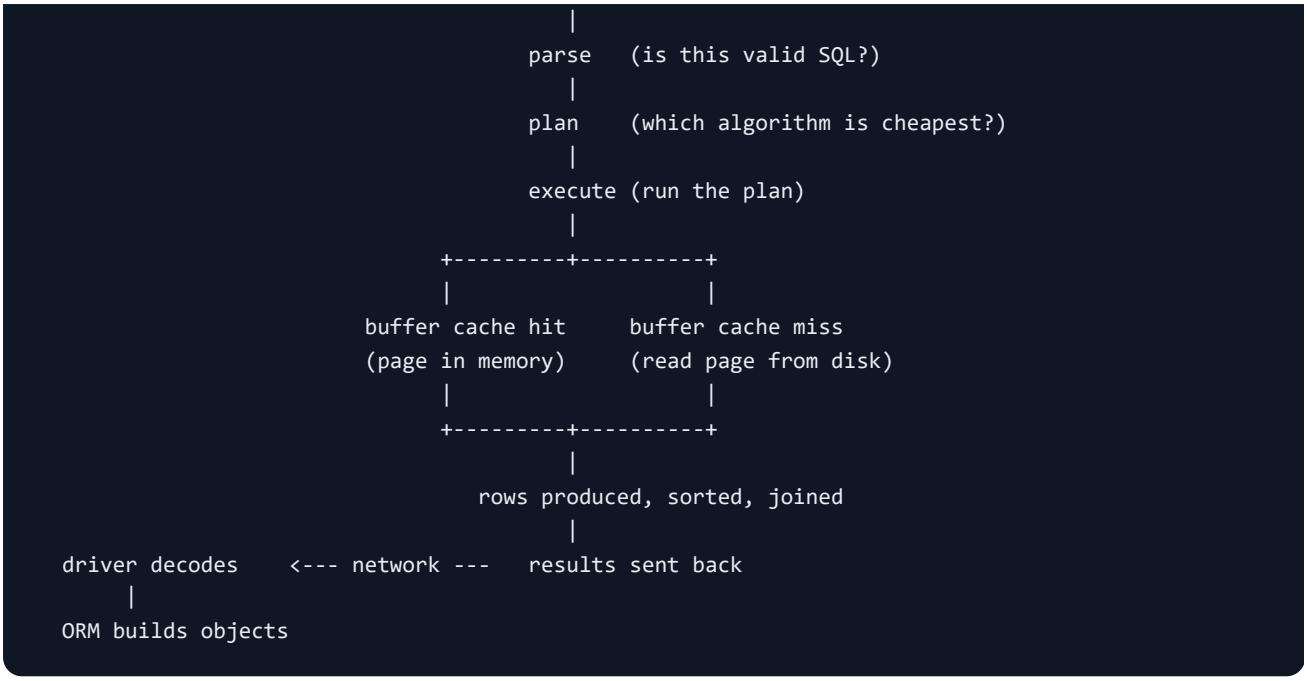
This chapter builds the picture. We will not tune anything yet; that starts in earnest in Part II. Instead, we will follow a query through the system, name the places where time is spent, and sort the causes of slowness into a handful of families. The rest of the book fills in each family in depth.

The Journey of a Single Query

Consider one of the most frequent requests in the Wavecrest app: a member opens the "My classes" screen, and the API asks the database for that member's upcoming bookings. From the application developer's point of view, this is a single line of code — perhaps an ORM call such as `member.upcoming_bookings()`. From the database's point of view, it is a surprisingly long trip.

The path of one query (simplified)

```
Application                      Database server
-----------                      ---------------
ORM builds SQL
     |
driver serializes  --- network --->  connection / session process
```



Each box is a place where time can go. Let us walk through them in order.

1. **Building and sending the request.** The ORM or query builder turns method calls into SQL text, the driver encodes it using the database's wire protocol, and the bytes cross the network. On a local network this takes a fraction of a millisecond, but it happens for *every* statement. A request that issues one query pays it once; a request that issues two hundred queries pays it two hundred times.
2. **Getting a connection.** The statement needs a server-side session to run in. If the application uses a connection pool and all connections are busy, the request waits in the pool before it even reaches the database. That wait is invisible in database-side metrics, which is one reason application and database measurements often disagree. Chapter 23 covers connections and pooling.
3. **Parsing and planning.** The server checks that the SQL is valid, resolves table and column names, and then the *query planner* (called the optimizer in MySQL) chooses how to execute it. For a query that touches several tables there may be many possible plans: which table to read first, whether to use an index, which join algorithm to use. The planner estimates the cost of candidate plans and picks the cheapest one it finds. Planning is usually quick, but its *decision* is the single biggest influence on how long execution takes. Chapter 5 explains how the planner works.
4. **Executing the plan.** The executor runs the chosen plan: it reads rows from tables and indexes, filters them, joins them, sorts them, and aggregates them. This is where most of the work usually happens.
5. **Reading data pages.** Tables and indexes are stored in fixed-size pages. When the executor needs a page, it first looks in the database's memory cache. A cache hit costs a memory access; a miss means a read from storage, which is far slower even on modern SSDs. Chapter 2 is devoted to this layer.

6. **Returning results.** The rows are encoded and sent back across the network. A query that returns ten rows sends little data; a query that returns fifty thousand rows because the application forgot a `LIMIT` sends a lot, and the application then spends more time decoding them and building objects.

The total response time the member experiences is the sum of all these steps, plus whatever the application does before and after. When someone says "the database is slow", the first useful question is: *which* of these steps is slow? The answer determines the fix. A missing index affects step 4 and step 5. A chatty ORM affects step 1 and step 6. An exhausted connection pool affects step 2. No amount of indexing will fix a pool problem, and no pool setting will fix a missing index.

Work and Waiting

The most useful single distinction in performance work is between **doing work** and **waiting**. A query's elapsed time is made of both.

- **Service time** is time spent actually doing something: CPU cycles evaluating predicates and comparing keys, reading pages from storage, writing log records, sending bytes on the network.
- **Wait time** is time spent unable to make progress: waiting for a lock held by another transaction, waiting for a free connection, waiting in a queue for a CPU core that other queries are using, or waiting for a disk that is busy serving other requests.

The distinction matters because the two have different cures. If a query is slow because it does too much work — reading three million rows to return sixteen — the fix is to make it do less: a better index, a better query, a better plan. If a query is slow because it waits — for example, a booking insert blocked behind another transaction that is updating the same class session row — then making the query itself more efficient may not help at all. You need to reduce the contention: shorter transactions, a different locking strategy, or a design that avoids a single hot row.

Here is a compact way to think about the resources a database uses and what happens when each one runs short.

Resource	Work that uses it	Symptom when it runs short
CPU	Filtering, joining, sorting, hashing, parsing	High CPU; all queries slow down together
Memory	Caching pages, sorts, hash tables	More disk reads; sorts spill to temp files
Storage I/O	Reading uncached pages, writing logs and pages	High I/O wait; latency spikes
Network	Sending statements and result sets	Chatty requests slow; large results slow
Locks	Protecting rows and structures	Queries idle while blocked; low CPU
Connections	Server sessions for clients	Requests wait in the pool; errors at the limit

Notice the "low CPU" entry for locks. One of the most confusing situations for newcomers is a database that is barely busy — CPU is low, disks are quiet — yet requests are timing out. That pattern almost always means waiting, not working. The queries are parked behind a lock, or the application cannot get a connection. Chapter 22 shows how to see lock waits directly with `pg_locks` and `pg_stat_activity` in PostgreSQL and with Performance Schema in MySQL.

> **NOTE**
>
> Both PostgreSQL and MySQL expose "wait events" that tell you what a session is currently waiting for — a lock, an I/O operation, a client message, and so on. In PostgreSQL they appear in the `wait_event_type` and `wait_event` columns of `pg_stat_activity`; in MySQL they are instrumented in Performance Schema. Sampling these regularly is one of the fastest ways to tell work from waiting. Chapter 25 builds this into a monitoring routine.

The Most Important Ratio: Work Done Versus Rows Returned

If you remember one idea from this chapter, make it this one: **a query is efficient when the amount of data it touches is close to the amount of data it returns.** A query that returns sixteen rows but reads three million to find them is doing roughly two hundred thousand times more work than necessary. A query that returns sixteen rows and reads a few dozen pages to find them is close to ideal.

Let us make that concrete with the Wavecrest `bookings` table. Suppose the table holds a few million rows — you will generate data like this in Chapter 4 — and the API asks for one member's bookings.

PostgreSQL · a member's bookings

```
-- PostgreSQL
SELECT id, session_id, status
FROM bookings
WHERE member_id = 48213;
```

Without an index that starts with `member_id`, the database has only one option: read the whole table and check every row. The plan below shows what that looks like. We read EXPLAIN in detail in Chapter 6; for now, focus on the row counts and the number of pages ("buffers") touched.

PostgreSQL · EXPLAIN (ANALYZE, BUFFERS) — illustrative; your timings will differ

```
Seq Scan on bookings  (cost=0.00..69700.00 rows=17 width=21)
                      (actual time=0.041..412.337 rows=16 loops=1)
  Filter: (member_id = 48213)
  Rows Removed by Filter: 3299984
  Buffers: shared hit=2112 read=26338
Planning Time: 0.090 ms
Execution Time: 412.402 ms
```

The query returned 16 rows. To find them, it examined about 3.3 million rows and touched 28,450 pages (2,112 were already in memory, 26,338 had to be read). That ratio — millions of rows examined for a handful returned — is the signature of a missing or unusable index. The absolute milliseconds depend on hardware, cache state, and concurrent load, so do not read too much into "412 ms". Read the shape: a full scan with nearly every row thrown away.

Now the same query after an index on `bookings(member_id)` exists. (We will create indexes like this deliberately in Part III; the name `bookings_member_id_idx` follows the naming convention used throughout the book.)

PostgreSQL · EXPLAIN (ANALYZE, BUFFERS) — illustrative; your timings will differ

```
Index Scan using bookings_member_id_idx on bookings
      (cost=0.43..72.10 rows=17 width=21)
      (actual time=0.029..0.061 rows=16 loops=1)
  Index Cond: (member_id = 48213)
  Buffers: shared hit=19 read=1
Planning Time: 0.112 ms
Execution Time: 0.083 ms
```

Sixteen rows returned, twenty pages touched. The work is now proportional to the answer, not to the size of the table. This is the fundamental promise of an index, and it explains why indexing is the single most effective tool in this book. But it is not the only tool, and the ratio idea applies more widely than indexes:

- A report that aggregates a year of check-ins to show one number per studio touches millions of rows by necessity; its efficiency question is whether it touches each row *once*, and whether it could read a pre-aggregated summary instead (Chapter 18).
- A paginated list that uses `OFFSET 100000` reads and discards one hundred thousand rows to return twenty (Chapter 17).
- An API endpoint that loads a member, then loads each booking separately, then each session separately, may touch only a few rows per query but pays a network round trip for each one (Chapter 19).

In every case, the diagnostic question is the same: *how much did we touch, and how much did we need?*

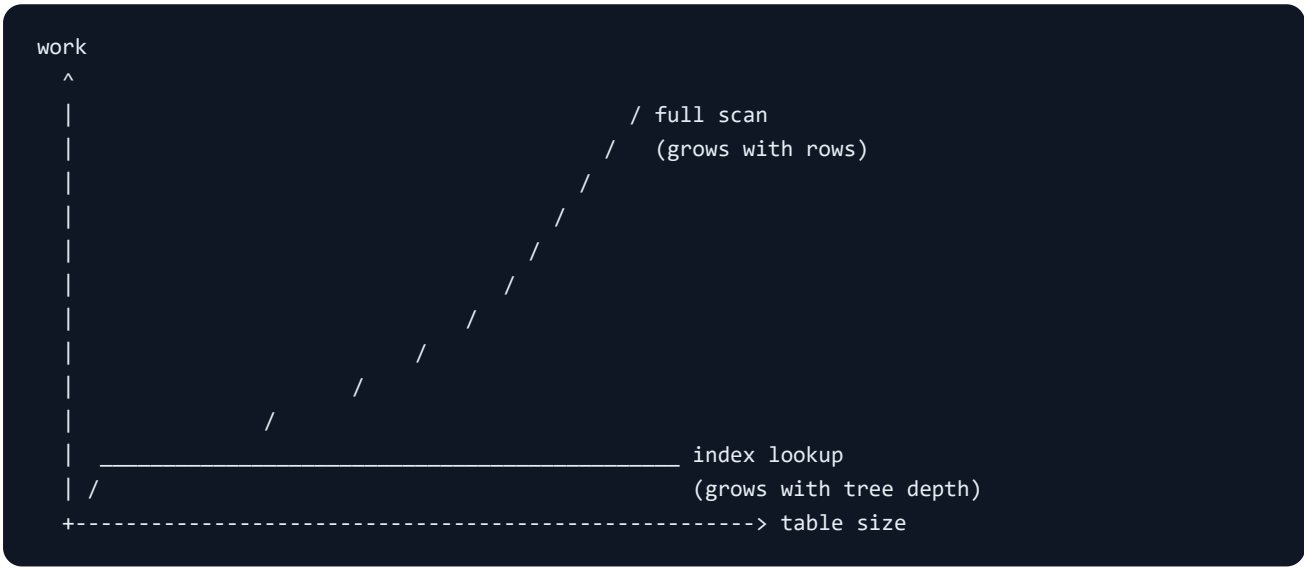
> **TIP**
>
> In MySQL, the slow query log records `Rows_examined` and `Rows_sent` for each logged statement, and `EXPLAIN` shows an estimated `rows` figure for each table access. In PostgreSQL, `EXPLAIN ANALYZE` reports `Rows Removed by Filter` and actual row counts per plan node. Get into the habit of comparing examined rows with returned rows before you look at anything else.

Why It Was Fast Yesterday: Growth Changes Everything

Many performance problems are not caused by a change in code at all. They are caused by growth. The Wavecrest `check_ins` table gains a row every time a member walks through a studio door. In the first month after launch it held a few thousand rows, and every query against it was fast no matter how it was written. A year later it holds millions of rows, and a query that used to take a millisecond now takes seconds.

The reason is that different access methods scale differently with table size. A full table scan does work proportional to the number of rows: double the table, double the work. A B-tree index lookup does work proportional to the *depth* of the tree, which grows with the logarithm of the table size — for practical purposes, it barely grows at all. Chapter 10 explains why.

How the cost of two access methods grows (shape only)



When a table is small, both lines are close to the floor, and the difference is invisible in testing. This is why so many performance problems escape into production: the developer's laptop database has a few hundred rows, the test suite has a few thousand, and every query is fast. The problem only appears when real data volume arrives. The lab you build in Chapter 4 exists precisely to avoid this trap — you will test against millions of rows, not hundreds.

The cache cliff

Growth also causes a second, less gradual effect. While a table and its indexes fit comfortably in memory, even inefficient queries are served from the cache and run at memory speed. The day the frequently used data — the *working set* — grows beyond what fits in memory, those same queries start reading from storage, and their latency can jump sharply even though the table only grew a little. Performance does not always degrade smoothly; sometimes it falls off a cliff. Chapter 2 explains the buffer cache and why the working set, not the total database size, is what matters.

Plans that flip

A third growth effect is that the planner's choice itself can change. The planner decides between a full scan and an index scan by estimating how many rows a condition will match. As data grows or its distribution shifts — say, one Wavecrest studio becomes far busier than the others — the estimates can change, and a plan that was good last month can be replaced by one that is not. The query text is identical; the plan is different. Chapter 8 is devoted to statistics and why plans go wrong.

> **WARNING**
>
> A query that is fast in development proves very little. Development databases are usually tiny, fully cached, and used by one person at a time — the three conditions under which almost any query looks fast. Before you conclude that a query is efficient, check its plan against a realistic data volume and look at how much data it touches, not just how long it took on your laptop.

The Usual Suspects

After many years of collective experience, the database community has found that performance problems cluster into a small number of families. When you face a slow system, walking through this list is a surprisingly effective way to find the cause. Each item is covered in depth later in the book.

1. Missing or unusable indexes

The query needs to find a few rows but has no index that lets it jump to them, so it scans. Sometimes an index exists but the query is written in a way that prevents its use — for example, wrapping the indexed column in a function, as in `WHERE date(checked_in_at) = '2026-03-14'`, or comparing a text column with a number so that a conversion is applied to every row. Part III covers indexing; Chapter 16 covers query patterns that defeat indexes.

2. A bad plan

An adequate index exists and the query is written sensibly, but the planner chooses a poor strategy, usually because its row-count estimates are wrong. Stale or insufficient statistics, correlated columns, and skewed data are the typical causes. Chapter 8 explains how to spot the telltale gap between estimated and actual rows.

3. Too much data requested

`SELECT *` on a wide table, a missing `LIMIT`, an unbounded date range, or loading a whole table to display twenty rows. The database may execute the query efficiently, but the amount of data to read, transfer, and decode is simply large.

4. Too many round trips

The classic N+1 pattern: one query to fetch a list, then one query per item. Each individual query is fast, and each appears harmless in isolation, but together they multiply network latency and per-statement overhead. ORMs make this easy to do by accident. Chapter 19 is devoted to it.

5. Contention

Queries interfere with each other: they wait for the same row locks, the same hot pages, or the same connection pool. The Wavecrest booking rush is a textbook case — hundreds of members trying to book the same class session, each transaction wanting to increment the same `booked_count` on the same row. Chapters 21 and 22 cover transactions, locks, and hot rows.

6. Resource saturation

The server runs out of CPU, memory, or I/O capacity. Sometimes this is the root cause (the workload has genuinely outgrown the hardware); more often it is a *consequence* of one of the causes above — a few inefficient queries consuming most of the CPU, for example. Saturation is a symptom to explain, not a diagnosis.

7. Maintenance debt

Databases need housekeeping. In PostgreSQL, updates and deletes leave dead row versions that VACUUM must clean up; if it falls behind, tables and indexes bloat and queries read more pages than necessary. In MySQL's InnoDB, the purge process cleans up old row versions from undo logs, and long-running transactions can hold it back. Out-of-date statistics also count as maintenance debt. Chapter 24 covers vacuum, autovacuum, and purge.

8. Configuration and environment

Memory settings that leave the cache far too small for the working set, a connection limit that is too low (or far too high), or storage that is slower than assumed. Configuration matters, but it is usually the *last* suspect to investigate rather than the first: default configurations are often conservative, yet a well-configured server still cannot rescue a query that scans millions of rows to return one.

Family	Typical Wavecrest symptom	Main chapters
Missing index	"My classes" screen slow for everyone	10–16
Bad plan	Studio report slow for one busy studio	5–9
Too much data	Admin export times out	17, 18
Round trips	Schedule API slow despite fast queries	19
Contention	Sunday booking rush stalls	21, 22
Saturation	Everything slow at peak	23–25
Maintenance debt	Queries slowly degrade over weeks	14, 24

Configuration	Heavy disk reads despite small data	2, 24

Contention and Queueing: Why Busy Systems Slow Down Non-Linearly

Back to Sunday evening. Why does the booking API slow down so dramatically during the rush when it is perfectly fast at other times? Part of the answer is contention on specific rows, which we examine in Chapter 22. The other part is a general property of any shared resource: **queueing**.

Picture a single checkout counter at a supermarket. If customers arrive occasionally, each one is served immediately and the wait is zero. As arrivals increase, customers sometimes arrive while someone else is being served, and a short queue forms. As the counter approaches being busy all the time, queues grow quickly, and a small increase in arrivals produces a large increase in waiting. Queueing theory formalizes this: as utilization of a resource approaches 100%, waiting time grows sharply rather than linearly.

A database has many such counters: CPU cores, storage devices, the connection pool, and — most dramatically — individual locks. A single popular class session row can only be updated by one transaction at a time. If each booking transaction holds that row lock for a few milliseconds, the row can serve only a limited number of bookings per second, no matter how powerful the server is. When the rush exceeds that rate, transactions queue behind each other, and their response time is dominated by waiting.

Two practical lessons follow. First, a system that looks healthy at moderate load tells you little about its behavior near saturation; you have to test at realistic peak load, which Chapter 3 discusses. Second, anything that shortens the time a resource is held — a faster query, a shorter transaction, fewer statements while a lock is held — helps disproportionately under load, because it reduces both service time and the queue that forms behind it.

Round Trips: The Cost the Database Never Sees

Some of the slowest Wavecrest endpoints turn out to have perfectly fast queries. Here is a simplified version of the code behind the weekly schedule screen, written as it often appears in ORM-based applications:

Python · the N+1 pattern (simplified)

```
sessions = db.query(ClassSession).filter(
    ClassSession.studio_id == studio_id,
    ClassSession.starts_at >= week_start,
    ClassSession.starts_at < week_end,
).all()                                  # 1 query, ~80 sessions
```

```
for s in sessions:
    s.instructor.full_name              # +1 query per session
    s.class_type.name                   # +1 query per session
```

One query fetches about eighty sessions; then the loop triggers two lazy-loading queries per session. The endpoint issues around 161 statements. Each takes well under a millisecond inside the database, so the slow query log shows nothing. Yet the endpoint is slow, because every statement also pays for a network round trip, driver work, and ORM overhead. The database-side view says "fast"; the user's view says "slow". Both are correct, and only by measuring at both ends do you see the full picture. The fix — fetching related rows with a join or a single batched query — is the topic of Chapter 19.

A Method for Tuning

With the mental model in place, here is the method we use throughout the book. It is deliberately simple, because the discipline matters more than the details.

1. **Define the problem in measurable terms.** "The app is slow" is not actionable. "The p95 latency of the booking endpoint rises well above its usual level during the Sunday schedule release" is. Chapter 3 explains latency percentiles and baselines.
2. **Find where the time goes.** Is it the application, the network, waiting for a connection, waiting for locks, or executing queries? If it is query execution, which queries? Tools such as `pg_stat_statements` and the MySQL slow query log answer this (Chapter 25).
3. **Explain the slow part.** For a slow query, read its plan and compare the rows it touches with the rows it returns. For waiting, find out what it waits for.
4. **Form a hypothesis and make one change.** "An index on `(member_id, status)` will let this query skip the full scan." Change one thing at a time, so you know which change had which effect.
5. **Verify with the same measurement.** Re-run the plan and the measurement from step 1. Did the rows touched drop? Did the percentile improve? Did anything else get worse — for example, write throughput after adding an index?
6. **Record what you learned.** A short note of the before and after, with plans attached, saves hours the next time the same symptom appears.

This loop sounds obvious, yet the most common tuning mistake is to skip step 2 and jump straight to a fix. Teams add memory, upgrade the server, or add a cache in front of the database, when the real problem was one query missing one index. Sometimes the expensive fix even appears to work for a while, because a bigger server absorbs the waste — until data grows again.

Common Myths

A few beliefs cause a lot of wasted effort. It is worth naming them early.

- **"More hardware will fix it."** Hardware helps when the workload is efficient and has genuinely outgrown the server. It helps much less when queries do orders of magnitude more work than necessary; that waste simply grows back as data grows.
- **"Index every column."** Indexes speed up reads that can use them and slow down every write, because each index must be maintained on insert, update, and delete. They also consume memory and storage. Chapter 14 covers the cost side; Chapter 15 presents a strategy.
- **"The ORM handles performance."** ORMs generate reasonable SQL for simple cases, but they cannot know your data volumes or access patterns. You still need to look at the SQL they produce.
- **"Averages tell you how the system performs."** An average hides the slow requests that users actually complain about. Chapter 3 shows why percentiles are the right lens.
- **"If the query is fast when I run it, it is fine."** Running a query by hand, once, on a warm cache with no concurrent load, measures one favorable case. Production is many concurrent cases, some of them unfavorable.

Where Wavecrest Goes From Here

Throughout the book, Wavecrest serves as our laboratory. Its schema is small enough to understand at a glance and rich enough to reproduce most real-world problems: an append-heavy `check_ins` table that grows without bound, time-range queries on `class_sessions`, per-studio filtering that behaves differently for large and small studios, a booking rush that creates hot rows, and reporting queries over months of history. In Chapter 2 we look at how these tables are physically stored; in Chapter 3 we learn to measure; and in Chapter 4 we build the lab, fill it with a few million rows of realistic data, and set up the tools used in every later chapter.