SQL PERFORMANCE MASTERCLASS

A Practical Ebook for Writing Fast, Scalable SQL

Professional SQL Training for Developers, Instructors, and Technical Teams

EXPLAIN - INDEXES - JOINS - PAGINATION - AGGREGATION

PostgreSQL and MySQL performance patterns for real-world applications

2026 Edition

By AYSUN iTAI | iTAI WEB SOLUTIONS

About This Ebook

This ebook is designed as a practical, classroom-ready guide to SQL performance. It teaches the mental model behind fast SQL: measure first, understand the execution plan, then fix the real cause of slowness.

FROM THE AUTHOR
Created and published by AYSUN iTAI, founder of iTAI WEB SOLUTIONS. This ebook is designed to help developers write SQL that performs confidently in real-world applications.

Educational disclaimer: SQL behavior can vary between database engines and versions. Always test performance changes in your own environment before applying them to production.

Table of Contents

1. The Professional SQL Performance Mindset

A query that works on a small table can become painfully slow in production. Professional SQL is not only about returning the correct data. It is about returning the correct data efficiently, predictably, and safely as the database grows.

The key questions

- How many rows will this query read?
- Is the database using an index or scanning the whole table?
- Am I fetching more columns than the application needs?
- Is the query sorting, grouping, or joining more data than necessary?
- Will this still be fast when the table has ten million rows?

> **CORE PRINCIPLE**
>
> Do not guess. Measure the query, read the plan, and optimize the bottleneck that actually exists.

Small database thinking vs production database thinking

Small database habit	Production database habit
Write the query and move on	Inspect the execution plan
Use SELECT * because it is easy	Select only the columns needed
Add indexes randomly	Add indexes based on WHERE, JOIN, and ORDER BY patterns
Use OFFSET everywhere	Use cursor pagination for large lists
Buy more server power	Fix N+1 queries and inefficient joins first

2. How the Query Optimizer Thinks

When you write SQL, you describe what result you want. The database decides how to produce that result. This decision is made by the query optimizer.

The optimizer compares possible execution plans and chooses the plan it estimates to be cheapest. It relies on table statistics, indexes, row counts, and data distribution.

Common plan operations

Plan node	Meaning	What to watch
Seq Scan	Reads the whole table	Fine on tiny tables, risky on large tables
Index Scan	Uses an index, then fetches rows	Usually good for selective filters
Index Only Scan	Answers from the index alone	Excellent when a covering index works
Nested Loop	Loops through one side and probes another	Good only when the inner side is small or indexed
Hash Join	Builds a hash table for joining	Common for large equality joins
Sort	Explicitly sorts rows	May be expensive or spill to disk

STATISTICS MATTER

If estimated rows and actual rows are very different, update statistics before adding new indexes.

```
-- PostgreSQL: refresh statistics
ANALYZE users;

-- Check when tables were last analyzed
SELECT relname, last_analyze, last_autoanalyze
FROM pg_stat_user_tables
WHERE relname = 'users';
```

3. EXPLAIN and EXPLAIN ANALYZE

EXPLAIN is the professional starting point for performance tuning. It shows how the database plans to execute your query.

EXPLAIN

```
EXPLAIN
SELECT id, email
FROM users
WHERE email = 'alice@example.com';
```

EXPLAIN shows the estimated plan without actually running the query.

EXPLAIN ANALYZE

```
EXPLAIN ANALYZE
SELECT id, email
FROM users
WHERE email = 'alice@example.com';
```

EXPLAIN ANALYZE runs the query and shows actual execution statistics. This is the most useful form when diagnosing real slowness.

> **SAFETY WARNING**
>
> EXPLAIN ANALYZE actually executes INSERT, UPDATE, and DELETE statements. Wrap dangerous statements in a transaction and roll them back.

```
BEGIN;
EXPLAIN ANALYZE
DELETE FROM users WHERE status = 'inactive';
ROLLBACK;
```

What to look for

- Large table scans on production-size tables.
- Huge differences between estimated rows and actual rows.
- Sort operations over many rows.
- Nested loops where the inner side is large and not indexed.
- Queries reading far more rows than they return.

4. Index Mastery

Indexes are the highest-return SQL performance tool. An index lets the database find rows without scanning the entire table.

Basic index

```
CREATE INDEX idx_users_email
ON users(email);

SELECT id, name, email
FROM users
WHERE email = 'alice@example.com';
```

Composite indexes

A composite index contains multiple columns. Column order is critical because the database uses the index from left to right.

```
CREATE INDEX idx_orders_user_status_created
ON orders(user_id, status, created_at DESC);
```

This index is strong for a query like this:

```
SELECT id, total, created_at
FROM orders
WHERE user_id = 42
AND status = 'completed'
ORDER BY created_at DESC
LIMIT 20;
```

> **COMPOSITE INDEX RULE**
>
> Put equality columns first. Put the range or sorting column last.

Partial indexes

A partial index only indexes rows matching a condition. This can make an index much smaller and faster.

```
CREATE INDEX idx_users_active_email
ON users(email)
WHERE deleted_at IS NULL;

SELECT id, email
FROM users
WHERE email = 'alice@example.com'
AND deleted_at IS NULL;
```

Covering indexes

A covering index contains all columns needed by the query, allowing the database to avoid extra table reads.

```
-- PostgreSQL
CREATE INDEX idx_users_status_cover
ON users(status) INCLUDE (id, email);
```

```
SELECT id, email
FROM users
WHERE status = 'active';
```

When not to add an index

- Tiny tables where a full scan is cheaper.
- Low-cardinality columns such as boolean flags.
- High-write tables where the index is rarely used.
- Duplicate indexes that add write cost without adding value.
- Columns wrapped in functions unless you create a matching functional index.

5. Slow Query Patterns and Professional Fixes

Pattern 1: SELECT *

Slow / careless	Fast / intentional

`SELECT *`
`FROM users`
`WHERE id = 1;`
	`SELECT id, name, email`
	`FROM users`
	`WHERE id = 1;`

Selecting only needed columns reduces I/O, network transfer, and memory usage. It can also enable index-only scans.

Pattern 2: N+1 queries

Slow / repeated loop	Fast / single query

`SELECT id`
`FROM orders`
`WHERE user_id = 10;`
 	
`-- Then inside a loop:`
`SELECT *`
`FROM products`
`WHERE id = ?;`
	`SELECT o.id, p.name, p.price`
	`FROM orders o`
	`JOIN order_items oi ON oi.order_id = o.id`
	`JOIN products p ON p.id = oi.product_id`
	`WHERE o.user_id = 10;`

N+1 is one of the most common ORM performance bugs. If your code runs queries inside a loop, inspect it immediately.

Pattern 3: functions on indexed columns

Slow / breaks normal index	Fast / normalized or indexed

`SELECT *`
`FROM users`
`WHERE LOWER(email) = 'alice@example.com';`
	`SELECT *`
	`FROM users`
	`WHERE email = 'alice@example.com';`
	
	`-- Or create a functional index:`
	`CREATE INDEX idx_users_lower_email`
	`ON users(LOWER(email));`

Pattern 4: leading wildcard LIKE

Slow / cannot use B-tree well	Fast / prefix or full-text

`SELECT *`
`FROM products`
`WHERE name LIKE '%docker%';`
	`SELECT *`
	`FROM products`
	`WHERE name LIKE 'docker%';`
	
	`-- For keyword search, use full-text search.`

Pattern 5: OR conditions

Sometimes slow	Often better

`SELECT *`
`FROM users`
`WHERE email = 'alice@example.com'`
	`SELECT *`
	`FROM users`
	`WHERE email = 'alice@example.com'`

`OR username = 'alice';`	`UNION`
	`SELECT *`
	`FROM users`
	`WHERE username = 'alice';`

Pattern 6: correlated subqueries

Slow / repeated subquery	Fast / join and group

`SELECT u.id, u.name,`
`  (SELECT COUNT(*)`
`   FROM orders o`
`   WHERE o.user_id = u.id) AS order_count`
`FROM users u;`
	`SELECT u.id, u.name, COUNT(o.id) AS order_count`
	`FROM users u`
	`LEFT JOIN orders o ON o.user_id = u.id`
	`GROUP BY u.id, u.name;`

PROFESSIONAL HABIT

Before scaling servers, fix SELECT *, N+1, missing indexes, bad pagination, and unnecessary correlated subqueries.

6. JOIN Optimization

JOINs are essential, but expensive joins are one of the easiest ways to slow down an application.

Index join columns

```
SELECT u.name, o.total
FROM users u
JOIN orders o ON o.user_id = u.id;
```

The primary key users.id is usually already indexed. The foreign key orders.user_id should also be indexed.

```
CREATE INDEX idx_orders_user_id
ON orders(user_id);
```

Three common join algorithms

Join type	Best when	Risk
Nested Loop	Outer side is small and inner side is indexed	Terrible if both sides are large
Hash Join	Large equality joins	Can use a lot of memory
Merge Join	Both sides are sorted or indexed	Sorting first can be expensive

Filter before joining

Less clear

```
SELECT u.name, o.total
FROM users u
JOIN orders o ON o.user_id = u.id
WHERE o.status = 'pending'
AND u.country = 'US';
```

Clear intent

```
WITH pending_orders AS (
  SELECT user_id, total
  FROM orders
  WHERE status = 'pending'
)
SELECT u.name, p.total
FROM users u
JOIN pending_orders p ON p.user_id = u.id
WHERE u.country = 'US';
```

Modern optimizers may produce the same plan, but explicit filtering makes the intention clear and can be more robust across systems.

7. Pagination That Scales

Pagination is a hidden performance trap. OFFSET pagination feels simple, but it becomes slower as users go deeper into the result set.

OFFSET pagination

```
SELECT id, title, created_at
FROM posts
ORDER BY created_at DESC
LIMIT 20 OFFSET 10000;
```

The database still has to walk through the skipped rows. The deeper the page, the more work it does.

Cursor pagination

```
-- First page
SELECT id, title, created_at
FROM posts
ORDER BY created_at DESC, id DESC
LIMIT 20;


-- Next page, using the last row from the previous page as cursor
SELECT id, title, created_at
FROM posts
WHERE (created_at, id) < ('2026-03-15 10:00:00', 1523)
ORDER BY created_at DESC, id DESC
LIMIT 20;
```

```
CREATE INDEX idx_posts_created_id
ON posts(created_at DESC, id DESC);
```

WHERE CURSOR PAGINATION SHINES

Use cursor pagination for feeds, inboxes, logs, activity streams, chat messages, dashboards, and infinite scroll.

Deferred join pagination

Heavy OFFSET

```
SELECT id, title, body, metadata, author_id
FROM posts
ORDER BY id
LIMIT 20 OFFSET 10000;
```

Deferred join

```
SELECT p.*
FROM posts p
JOIN (
  SELECT id
  FROM posts
  ORDER BY id
  LIMIT 20 OFFSET 10000
) page ON page.id = p.id;
```

Deferred joins help when you must keep an OFFSET-based API but want to avoid reading wide rows that will be discarded.

8. Aggregation, Window Functions, and Reports

COUNT variants

```
-- Counts all rows, including rows with NULL values
SELECT COUNT(*) FROM orders;

-- Counts only rows where coupon_code is not NULL
SELECT COUNT(coupon_code) FROM orders;

-- Counts unique users, more expensive because it deduplicates
SELECT COUNT(DISTINCT user_id) FROM orders;
```

Use the COUNT version that matches your intention. DISTINCT counts are more expensive because the database must remove duplicates.

Window functions

Repeated calculation	Single pass window function

`SELECT id, total,`
`  (SELECT AVG(total)`
`   FROM orders o2`
`   WHERE o2.user_id = o1.user_id) AS user_avg`
`FROM orders o1;`
	`SELECT id, total,`
	`  AVG(total) OVER (PARTITION BY user_id) AS user_avg`
	`FROM orders;`

Window functions calculate values across related rows without collapsing the result set.

Materialized views

For reports and dashboards, do not recalculate expensive aggregates on every request. Precompute them.

```
CREATE MATERIALIZED VIEW daily_revenue AS
SELECT DATE_TRUNC('day', created_at) AS day,
       SUM(total) AS revenue,
       COUNT(*) AS order_count
FROM orders
WHERE status = 'completed'
GROUP BY DATE_TRUNC('day', created_at);

CREATE UNIQUE INDEX idx_daily_revenue_day
ON daily_revenue(day);

REFRESH MATERIALIZED VIEW daily_revenue;
```

BEST USE CASE

Materialized views are ideal for dashboards, financial summaries, daily reports, and expensive aggregations that do not need to be real-time.

9. PostgreSQL Performance Toolkit

pg_stat_statements

pg_stat_statements helps identify the queries consuming the most total time in PostgreSQL.

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

SELECT query,
       calls,
       ROUND(total_exec_time::numeric, 2) AS total_ms,
       ROUND(mean_exec_time::numeric, 2) AS mean_ms,
       rows
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

JSONB indexes

```
CREATE INDEX idx_products_data_gin
ON products USING GIN(data);

SELECT *
FROM products
WHERE data @> '{"category": "electronics"}';

CREATE INDEX idx_products_category
ON products((data->>'category'));

SELECT *
FROM products
WHERE data->>'category' = 'electronics';
```

VACUUM and bloat

PostgreSQL keeps old row versions after updates and deletes. Autovacuum usually handles this, but large tables should be monitored.

```
SELECT relname, n_dead_tup, n_live_tup,
       ROUND(n_dead_tup * 100.0 / NULLIF(n_live_tup + n_dead_tup, 0), 1) AS dead_pct
FROM pg_stat_user_tables
ORDER BY n_dead_tup DESC
LIMIT 10;

VACUUM ANALYZE orders;
```

10. MySQL Performance Toolkit

Slow query log

```
SET GLOBAL slow_query_log = ON;
SET GLOBAL long_query_time = 1;
SET GLOBAL log_queries_not_using_indexes = ON;
```

The slow query log helps you discover which queries need tuning instead of guessing.

EXPLAIN in MySQL

```
EXPLAIN
SELECT *
FROM orders
WHERE user_id = 42;
```

Field	Meaning
type	Access type. ALL usually means a full scan.
key	Index chosen. NULL means no index was used.
rows	Estimated rows examined.
Extra	Watch for Using filesort or Using temporary.

Primary key design

InnoDB stores rows ordered by the primary key. Sequential integer primary keys are usually efficient. Random UUID primary keys can cause page splits and larger secondary indexes.

```
CREATE TABLE orders (
  id BIGINT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
  user_id BIGINT UNSIGNED NOT NULL,
  created_at DATETIME NOT NULL
);
```

11. Final Checklist and Exercises

Production SQL checklist

1. Run EXPLAIN or EXPLAIN ANALYZE before optimizing.
2. Select only the columns the application needs.
3. Index columns used in WHERE filters.
4. Index columns used in JOIN conditions.
5. Avoid queries inside application loops.
6. Avoid functions on indexed columns unless you use a functional index.
7. Use cursor pagination for deep or infinite scrolling lists.
8. Order composite indexes with equality columns first and range or sort columns last.
9. Use partial indexes for common filtered workloads.
10. Measure production queries with pg_stat_statements or the MySQL slow query log.

Classroom exercise

Given this table:

```
CREATE TABLE orders (
  id BIGSERIAL PRIMARY KEY,
  user_id BIGINT NOT NULL,
  status TEXT NOT NULL,
  total NUMERIC(10, 2),
  created_at TIMESTAMPTZ NOT NULL
);
```

And this query:

```
SELECT *
FROM orders
WHERE user_id = 42
AND status = 'completed'
ORDER BY created_at DESC
LIMIT 20;
```

Questions

11. What is wrong with SELECT *?
12. Which columns are used for filtering?
13. Which column is used for sorting?
14. What composite index would help?
15. How would you rewrite the query to select only needed columns?

Suggested solution

```
SELECT id, total, created_at
FROM orders
WHERE user_id = 42
AND status = 'completed'
ORDER BY created_at DESC
LIMIT 20;


CREATE INDEX idx_orders_user_status_created
ON orders(user_id, status, created_at DESC);
```

WHY THIS WORKS

user_id and status are equality filters. created_at is used for sorting. The composite index follows the professional rule: equality first, sort/range last.

Closing principle

SQL performance is not about memorizing tricks. It is about learning how the database thinks. The best developers measure first, read the execution plan, and fix the root cause instead of treating the symptom.

Before optimizing, run EXPLAIN ANALYZE.