UnCodemy

SQL Query Optimization Guide PDF

To make a SQL query faster, read its execution plan, filter early on indexed columns, select only the columns you need, avoid functions on indexed columns in WHERE, join on indexed keys, avoid unneeded DISTINCT and sorting, and test changes on realistic data. Always measure before and after. This guide shows common slow patterns and their fixes.

Key takeaways

- Measure first: use the execution plan, not guesses.
- Indexes speed up reads on filtered, joined and sorted columns but slow down writes.
- Wrapping an indexed column in a function usually prevents the index from being used.
- Select only needed columns and rows; avoid SELECT * in production queries.
- Test on realistic data volumes, since small tables hide problems.

What is SQL query optimization?

SQL query optimization is the practice of rewriting queries and designing indexes and tables so that the database returns the same result using less time, memory and input and output.

How do I begin optimising?

First, find the slow query using the database's logs or monitoring. Then see how the database runs it with the **execution plan** (`EXPLAIN` in MySQL and PostgreSQL, `EXPLAIN QUERY PLAN` in SQLite). The plan shows whether the database scans the whole table or uses an index, the join order and the estimated rows. Change one thing at a time and compare time and plan before and after.

What does an index do?

An index is a sorted structure that lets the database find rows without scanning everything, like the index of a book. Index columns that appear often in `WHERE`, `JOIN` and `ORDER BY`. Primary keys are indexed automatically in most systems. Indexes take space and slow down inserts and updates, so add them for real query needs and drop unused ones.

```
CREATE INDEX idx_orders_customer ON orders(customer_id);
EXPLAIN QUERY PLAN
SELECT order_id, order_total FROM orders WHERE customer_id = 1;
```

The plan should say the query searches using the index instead of scanning the table.

How do I write filters that can use indexes?

Keep the indexed column alone on one side of the comparison. A function on the column usually prevents index use.

```
Slow:   WHERE YEAR(order_date) = 2026
Better: WHERE order_date >= '2026-01-01' AND order_date < '2027-01-01'

Slow:   WHERE LOWER(email) = 'a@x.com'
Better: store emails in one case, or create an expression index if supported
```

A leading wildcard such as `LIKE '%shirt'` cannot use a normal index, while `LIKE 'shirt%'` can.

Why should I avoid SELECT *?

It reads and sends more data than needed, can stop the use of covering indexes and breaks when table columns change. List the columns you need.

How should I write joins?

Join on indexed columns of the same type. Filter rows before joining when possible so that fewer rows are joined. Make sure the join condition is complete, because a missing condition creates a huge product of rows. Use `EXISTS` instead of a join plus `DISTINCT` when you only need to know whether a match exists.

```
SELECT c.name
FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id);
```

Are subqueries slow?

Not always; modern optimisers handle many well. Correlated subqueries that run once per row can be slow on large tables. Rewrite them as joins or with window functions when the plan shows repeated scans. Test both forms.

How do I reduce sorting and grouping cost?

Sorting large results is expensive. Use `LIMIT` when you need the top rows, and index the sort column when the pattern is frequent. Avoid `DISTINCT` to hide duplicate rows caused by a bad join; fix the join. Aggregate before joining when you only need totals from a big table.

How does pagination affect speed?

`OFFSET` with a large number forces the database to read and skip many rows. For deep pages use keyset pagination: filter by the last seen key, for example `WHERE id > 1000 ORDER BY id LIMIT 20`.

What table design choices help?

Use suitable data types (integers for IDs, dates for dates), avoid storing several values in one column, keep tables normalised for writes and consider summary tables or materialised views for heavy reports. Partitioning large tables by date can speed up queries that filter by date.

How do statistics and caching matter?

The optimiser uses statistics about data distribution to choose a plan. After big data loads, update statistics (`ANALYZE`). Cache results of expensive, repeated queries in the application or a cache layer when freshness rules allow.

What is a safe tuning routine?

1. Capture the slow query and its time. 2. Read the plan. 3. Change one thing: an index, a filter or a rewrite. 4. Check the plan and time again on realistic data. 5. Check that results are identical. 6. Document the change and monitor.

What mistakes slow down real systems?

Missing indexes on foreign keys, queries that fetch thousands of rows when the page shows twenty, running a query inside a loop in application code instead of one set-based query, comparing columns of different types so that conversion disables indexes and repeating the same heavy report many times a day without caching. Each of these is common and fixable once measured.

How do I practise optimisation?

Create a table with a few hundred thousand made-up rows, run a filtered query, read the plan, add an index and compare. Then try a function on the column and see how the plan changes. Seeing the plan switch from a scan to an index search, and back, makes the rules easy to remember.

Frequently asked questions

How do I find out why a SQL query is slow?

Read its execution plan with EXPLAIN to see whether it scans the whole table, which indexes it uses and how it joins tables.

Why is my index not used?

A function on the indexed column, a leading wildcard in LIKE or a type mismatch often prevents index use.

Do indexes always make queries faster?

They speed up reads on filtered, joined and sorted columns but take space and slow down inserts and updates.

Is SELECT * bad?

It reads more data than needed and can stop covering index use, so list only the needed columns in production queries.

How do I make deep pagination faster?

Use keyset pagination by filtering on the last seen key instead of a large OFFSET.

Sources and further reading

- PostgreSQL: Performance Tips
- MySQL: Optimization
- SQLite query planner

About this PDF

These PDFs are study material for learning and revision. They are not a certificate, a degree or a promise of a job, and technology changes, so check official documentation for the latest details. Spot an error? Write to info@uncodemy.com and the trainer team will review it.

More free resources and courses: uncodemy.com | info@uncodemy.com | +91 98183 66550