



# Have you ever worked on SQL tuning? Brief about the best practices

Yes, I have worked extensively on SQL tuning. In my experience, SQL performance issues usually stem from **missing indexes**, **poor query execution plans**, or **architectural anti-patterns** like the N+1 problem.

When tuning, I follow a systematic approach: **Identify** → **Analyze** → **Optimize**.

## 1. Identify (Find the Slow Query)

- **Monitoring Tools:** I use tools like **Datadog APM**, **New Relic**, or cloud-specific tools like **AWS RDS Performance Insights** to spot slow queries (high latency or high CPU usage).
- **Slow Query Log:** I enable the database's "Slow Query Log" (e.g., in MySQL/Postgres) to capture queries taking longer than a threshold (e.g., > 1 second).<sup>[1]</sup> <sup>[2]</sup>

## 2. Analyze (Understand Why It's Slow)

Once identified, I run the `EXPLAIN` (or `EXPLAIN ANALYZE` in Postgres) command to see the **Execution Plan**. I look for:

- **Full Table Scans:** Is the database reading 1 million rows to find just one? (Bad)<sup>[1]</sup>
- **Missing Indexes:** Are filtering (`WHERE`) or joining (`JOIN ON`) columns indexed?<sup>[3]</sup>
- **Rows Examined vs. Rows Returned:** If the DB scans 10,000 rows to return 10, the index selectivity is poor.<sup>[4]</sup>

## 3. Optimize (Best Practices I Follow)

### A. Indexing Strategy

- **Index High-Usage Columns:** I ensure columns used in `WHERE`, `JOIN`, and `ORDER BY` clauses are indexed.<sup>[5]</sup> <sup>[3]</sup>
- **Composite Indexes:** For queries filtering on multiple columns (e.g., `WHERE status = 'ACTIVE' AND region = 'US'`), I create a **Composite Index** (`(status, region)`) instead of single indexes, paying attention to the **Leftmost Prefix Rule**.<sup>[1]</sup>
- **Covering Indexes:** I try to include selected columns in the index itself (using `INCLUDE` in SQL Server/Postgres) to perform an **Index Only Scan**, avoiding a lookup in the main table heap.<sup>[1]</sup>

### B. Query Refactoring

- **Avoid SELECT \*:** Only fetch columns I need. This reduces I/O and network overhead.<sup>[6]</sup>
- **Fix N+1 Problem:** Instead of fetching a User and then querying their Orders one by one (1000 users = 1001 queries), I use JOIN or IN clauses (e.g., WHERE user\_id IN (...)) to fetch everything in 2 queries.<sup>[7] [4]</sup>
- **Pagination Optimization:** For large offsets (e.g., LIMIT 10 OFFSET 100000), the DB still scans 100,010 rows. I switch to **Keyset Pagination** (Seeking) using WHERE id > last\_seen\_id LIMIT 10, which is O(1) instead of O(N).<sup>[8] [9]</sup>

## C. Architecture / Schema

- **Denormalization:** If a query joins 5+ tables and is still slow, I sometimes denormalize data by creating a "read-optimized" table or materialized view.<sup>[3]</sup>
- **Read Replicas:** For heavy read traffic, I offload reporting/analytics queries to a **Read Replica** to protect the primary DB instance.

## Example Success Story (Interview Snippet)

"In a previous project, an API was timing out because of a query doing a Full Table Scan on a 50-million-row Transactions table. EXPLAIN showed it was filtering by merchant\_id and sorting by created\_at without a proper index. I created a composite index on (merchant\_id, created\_at). The query time dropped from **12 seconds** to **50 milliseconds**."

✱

1. <https://learnomatic.org/indexing-strategies-and-execution-plans-in-oracle-database/>
2. <https://www.developernation.net/blog/8-indexing-strategies-to-optimize-database-performance/>
3. <https://www.dbvis.com/thetable/top-sql-performance-tuning-interview-questions-and-answers/>
4. <https://readysset.io/blog/investigating-and-optimizing-over-querying>
5. <https://www.geeksforgeeks.org/mysql/mysql-indexing-best-practices/>
6. <https://www.geeksforgeeks.org/sql/sql-performance-tuning/>
7. <https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping>
8. <https://dev.to/mrpercival/improving-the-performance-of-limitoffset-in-mysql-42h8>
9. <https://learn.microsoft.com/en-us/ef/core/performance/efficient-querying>
10. <https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/429036f4-3bc3-4a7a-8c6f-90246e73b9b9/image.jpg>
11. <https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/24f18c0a-b56e-4862-a619-e2959036a5c2/image.jpg>
12. <https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/97e9ed86-ed11-4035-a18b-1eb05c1bab4e/image.jpg>
13. <https://ppl-ai-file-upload.s3.amazonaws.com/web/direct-files/attachments/images/81815274/87f62423-96f2-4071-9802-8f6699e0ecd8/image.jpg>
14. <https://www.remotely.works/blog/top-advanced-sql-interview-questions-and-answers>
15. <https://upesonline.ac.in/blog/advanced-sql-interview-questions>

16. <https://codesignal.com/blog/28-sql-interview-questions-and-answers-from-beginner-to-senior-level/>
17. [https://www.reddit.com/r/dataengineering/comments/trwho8/when\\_asked\\_in\\_a\\_job\\_interview\\_how\\_to/](https://www.reddit.com/r/dataengineering/comments/trwho8/when_asked_in_a_job_interview_how_to/)
18. <https://roadmap.sh/questions/sql-queries>
19. [https://www.linkedin.com/posts/aditya-chandak-24692825\\_sql-performance-tuning-interview-questions-activity-7364517650158280704-QQbF](https://www.linkedin.com/posts/aditya-chandak-24692825_sql-performance-tuning-interview-questions-activity-7364517650158280704-QQbF)
20. <https://www.acceldata.io/blog/mastering-database-indexing-strategies-for-peak-performance>
21. <https://www.prisma.io/docs/orm/prisma-client/queries/query-optimization-performance>
22. <https://www.datacamp.com/blog/top-sql-interview-questions-and-answers-for-beginners-and-intermediate-practitioners>
23. <https://experienceleague.adobe.com/en/docs/experience-manager-cloud-service/content/operations/query-and-indexing-best-practices>
24. <https://www.vervecopilot.com/interview-questions/top-30-most-common-sql-query-interview-questions-and-answers-you-should-prepare-for>