perplexity

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

Yes, I have extensively worked on SQL tuning, particularly when dealing with slow-performing APIs where the bottleneck was the database.

SQL tuning is the process of minimizing the resources (CPU, I/O) required to execute a query. My approach generally follows the **"Find, Analyze, Fix"** loop.

1. Identifying the Problem (Find)

First, you need to know *which* query is slow.

- **Slow Query Log:** I configure the database (e.g., MySQL or PostgreSQL) to log any query that takes longer than `200ms`.
- **APM Tools:** Tools like New Relic or Dynatrace often pinpoint exactly which SQL statement is causing high latency.

2. Analyzing the Execution Plan (Analyze)

Once I have the bad query, I run the `EXPLAIN` (or `EXPLAIN ANALYZE`) command on it. This tells me *how* the database is executing the query.

- **Full Table Scan:** This is the biggest red flag. It means the DB is reading every single row to find a match because it lacks an index.
- **Rows Examined vs. Rows Returned:** If the DB examines 1,000,000 rows to return just 10, the query is inefficient.

3. Best Practices & Optimization Techniques (Fix)

Here are the specific techniques I use to fix these issues:

A. Indexing Strategies

- **Add Missing Indexes:** If I search by `email` frequently (`WHERE email = '...'`), I add an index on that column. This changes the complexity from O(n) to O(log n).[1] [2]
- **Composite Indexes:** If a query filters by multiple columns (`WHERE status = 'ACTIVE' AND country = 'US'`), a single index on just `status` isn't enough. I create a **Composite Index** on `(status, country)`.[1]
- **Covering Index:** I try to include all the columns I'm *selecting* in the index itself. If the index has all the data, the DB doesn't even need to look at the main table (Heap), which is

extremely fast.[3]

B. Query Refactoring

- **Avoid `SELECT *`:** I never fetch all columns. I only select what is needed. Fetching unused `BLOB` or `TEXT` columns wastes huge amounts of I/O and network bandwidth.[2] [3]
- **Remove Functions on Columns:**
  - *Bad:* `WHERE YEAR(created_at) = 2023` (This disables the index because the DB must calculate the function for every row).
  - *Good:* `WHERE created_at BETWEEN '2023-01-01' AND '2023-12-31'` (This uses the index effectively).[2]
- **Pagination Optimization:**
  - Standard `LIMIT 10 OFFSET 100000` is slow because the DB still reads 100,010 rows and throws away the first 100k.
  - *Better:* Use "Seek Method" → `WHERE id > 100000 LIMIT 10`.

C. Join Optimization

- **Join Order:** I ensure we join the smallest table first to reduce the intermediate result set size.[1]
- **Avoid Subqueries:** I often rewrite nested subqueries (`WHERE id IN (SELECT ...)`) into `JOIN`s or `EXISTS` clauses, which modern optimizers handle better.[4] [5]

⁂

1. https://www.datacamp.com/blog/sql-query-optimization
2. https://www.geeksforgeeks.org/sql/best-practices-for-sql-query-optimizations/
3. https://blog.devart.com/how-to-optimize-sql-query.html
4. https://www.acceldata.io/blog/query-optimization-in-sql-essential-techniques-tools-and-best-practices
5. https://www.syncfusion.com/blogs/post/top-10-sql-query-optimization-techniques
6. https://www.sqlservercentral.com/forums/topic/database-query-optimization-how-to-improve-query-performance-in-sql
7. https://www.thoughtspot.com/data-trends/data-modeling/optimizing-sql-queries
8. https://www.reddit.com/r/dataengineering/comments/1axd7cy/what_are_your_top_sql_query_optimization_tips/
9. https://www.montecarlodata.com/blog-6-tips-for-better-sql-query-optimization/
10. https://www.youtube.com/watch?v=MpczBulk7R8