Excellent. Now we move into **real SQL engineering territory**.

This is a **deep dive into query optimization**, not surface-level advice.

We'll cover:

1. How MySQL actually executes queries
2. How to read `EXPLAIN` properly
3. Index internals (B+Tree)
4. Composite indexes & leftmost rule
5. Covering indexes
6. Join optimization
7. Subquery vs join performance
8. Window function cost
9. Pagination optimization
10. Real optimization workflow

We'll use the `advanced_db` dataset created earlier.

🔴1️⃣ How MySQL Executes a Query

When you run:

```
SELECT * FROM employees WHERE salary > 80000;
```

Internally MySQL does:

1. Parse SQL
2. Check syntax
3. Optimize (choose execution plan)
4. Execute

The optimizer chooses:

- Which index to use
- Join order
- Access method (index scan vs full scan)

🔴2️⃣FULL TABLE SCAN (Worst Case)

Without index:

```
SELECT * FROM employees WHERE salary > 80000;
```

MySQL checks **every row**.

This is O(n).

For 10 rows → fine
For 10 million rows → disaster

🔴3️⃣USING EXPLAIN (Most Important Tool)

```
EXPLAIN SELECT * FROM employees WHERE salary > 80000;
```

Key columns to understand:

Column	Meaning
type	Access method
possible_keys	Available indexes
key	Index actually used
rows	Estimated rows scanned
Extra	Extra info

Access Types (Ordered Best → Worst)

type	Meaning
const	Single row
eq_ref	Perfect join

ref	Indexed lookup
range	Range scan
index	Full index scan
ALL	Full table scan ❌

If you see:

```
type = ALL
```

You have a performance problem.

🔴4️⃣ ADD INDEX & TEST

```
CREATE INDEX idx_salary ON employees(salary);
```

Now run:

```
EXPLAIN SELECT * FROM employees WHERE salary > 80000;
```

Expected:

```
type = range
key = idx_salary
```

Now MySQL uses **B+Tree index range scan**.

🔴5️⃣ How B+Tree Index Works

Index stores:

```
salary → primary key pointer
```

Sorted order:

```
60000
70000
75000
80000
90000
```

Binary search is used → O(log n)

Then rows are fetched.

🔴6️⃣ Composite Index (CRITICAL)

Suppose query:

```
SELECT * FROM employees
WHERE dept_id = 1 AND salary > 70000;
```

Better index:

```
CREATE INDEX idx_dept_salary
ON employees(dept_id, salary);
```

This is powerful because:

MySQL uses **leftmost prefix rule**.

Leftmost Rule

Index `(dept_id, salary)` works for:

✅ WHERE dept_id = ?
✅ WHERE dept_id = ? AND salary > ?

❌ WHERE salary > ? alone

Because dept_id is first column.

🔴7️⃣ Covering Index (Advanced Optimization)

If query is:

```
SELECT dept_id, salary
FROM employees
WHERE dept_id = 1;
```

If index is:

```
CREATE INDEX idx_cover ON employees(dept_id, salary);
```

MySQL may not even touch the table.

Because:
All needed columns are inside index.

EXPLAIN shows:

```
Using index
```

That is extremely fast.

🔴8️⃣ JOIN OPTIMIZATION

Bad:

```
SELECT *
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id;
```

If no index on:

```
orders.customer_id
```

MySQL may scan entire orders table.

Fix:

```
CREATE INDEX idx_orders_customer
ON orders(customer_id);
```

Foreign key columns should almost always be indexed.

🔴9️⃣ JOIN ORDER MATTERS

MySQL optimizer decides join order.

Example:

```
SELECT *
FROM employees e
JOIN departments d ON e.dept_id = d.dept_id
WHERE d.dept_name = 'Engineering';
```

If index on:

```
departments.dept_name
```

MySQL first filters departments → then joins employees.

🔴 🔟 Subquery vs Join

Slow:

```
SELECT *
FROM employees
WHERE dept_id IN
(SELECT dept_id FROM departments WHERE dept_name = 'Engineering');
```

Better:

```
SELECT e.*
FROM employees e
```

```
JOIN departments d
ON e.dept_id = d.dept_id
WHERE d.dept_name = 'Engineering';
```

Joins usually optimize better.

🔴 1️⃣1️⃣ Avoid SELECT *

Bad:

```
SELECT * FROM employees;
```

Why?

- Transfers unnecessary columns
- Prevents covering index usage

Better:

```
SELECT emp_id, first_name, salary
FROM employees;
```

🔴 1️⃣2️⃣ Pagination Optimization

Bad:

```
SELECT * FROM employees
LIMIT 100000, 10;
```

This skips 100k rows → slow.

Better approach:

Use indexed column:

```
SELECT *
```

```
FROM employees
WHERE emp_id > 100000
LIMIT 10;
```

This is keyset pagination.

🔴 1️⃣3️⃣ ORDER BY Optimization

Bad:

```
SELECT * FROM employees
ORDER BY salary;
```

If no index on salary → filesort.

Fix:

```
CREATE INDEX idx_salary ON employees(salary);
```

Then EXPLAIN should show:

```
Using index
```

🔴 1️⃣4️⃣ Avoid Functions on Indexed Columns

Bad:

```
SELECT * FROM employees
WHERE YEAR(hire_date) = 2023;
```

This disables index.

Better:

```
WHERE hire_date BETWEEN '2023-01-01' AND '2023-12-31';
```

🔴 15️⃣ Window Functions Cost

```
SELECT first_name,
       RANK() OVER (ORDER BY salary DESC)
FROM employees;
```

Window functions require sorting.

Large datasets → heavy memory usage.

Index on salary helps but still sorting happens.

🔴 16️⃣ Transaction & Lock Optimization

Long transactions → lock contention.

Bad:

```
START TRANSACTION;
SELECT ...
(wait 5 minutes)
UPDATE ...
COMMIT;
```

Keep transactions short.

🔴 17️⃣ Real Optimization Workflow

When query is slow:

Run:

```
EXPLAIN query;
```

1.

2. Check:
    - type = ALL? ❌
    - rows large?
    - no key used?
3. Add appropriate index
4. Avoid:
    - functions on indexed columns
    - OR across multiple columns (may disable index)
5. Re-test EXPLAIN

🔴 18 Index Strategy Best Practices

Situation	Index Strategy
Foreign keys	Always index
WHERE column	Index
WHERE + ORDER BY	Composite index
Low cardinality (gender, status)	Avoid indexing
Many updates	Avoid too many indexes

🔴 19 Cardinality Matters

If column has only:

- 2 values (status: active/inactive)

Index not helpful.

If column has:

- Millions of distinct values (email, emp_id)

Very useful.

🔴 2️⃣0️⃣ Performance Interview Questions

- What is covering index?
- What is leftmost prefix rule?
- Why avoid function on indexed column?
- What is filesort?
- What is range scan?
- What is keyset pagination?
- What is index selectivity?

🟢 Practice Optimization Exercises

Run these and optimize:

1.

```
SELECT * FROM orders WHERE status='Completed';
```

2.

```
SELECT * FROM employees WHERE salary > 70000 ORDER BY hire_date;
```

3.

```
SELECT c.customer_name, SUM(o.total_amount)
FROM customers c
JOIN orders o ON c.customer_id=o.customer_id
```

```
GROUP BY c.customer_id;
```

Add appropriate indexes and compare EXPLAIN.

🟢 Final Key Principle

Indexes speed up reads
 Indexes slow down writes

Balance is required.