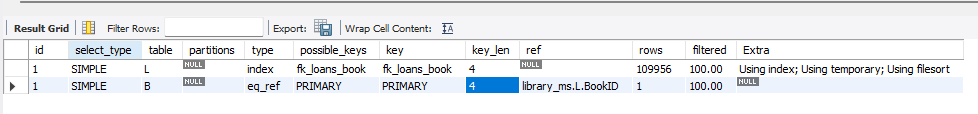
**Part 1.**

**Slow-Performing Queries:** Query 3 (Most Active Members), Query 5 (Total Books Available by Category), and Query 6 (Most Borrowed Book Categories) are slower compared to Query 1 and Query 2. The EXPLAIN output for these queries shows factors (full table scans, high rows count, temporary tables, or filesort operations) that likely contribute to their longer duration."

Query 1: Top 5 Most Borrowed Books

This is the result of explain:

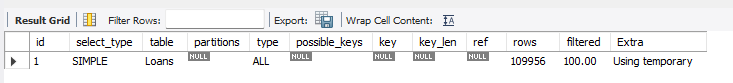


This is the Duration/ Fetch:

0.171 sec / 0.000 sec

Query 2: How Many Books Are Borrowed Each Month?

This is the result of explain:

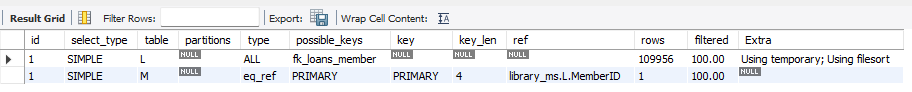


This is the Duration/ Fetch:

0.063 sec / 0.000 sec

Query 3: Most Active Members

This is the result of explain:

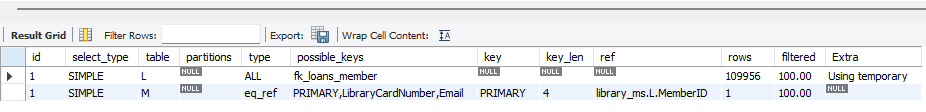


This is the Duration/ Fetch:

0.703 sec / 0.063 sec

Query 4: Average Loan Duration Per Member

This is the result of explain:

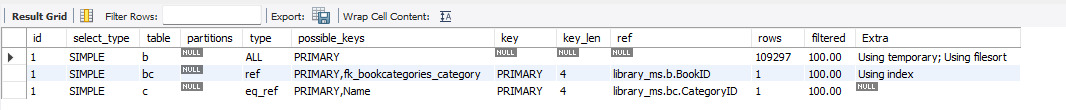


This is the Duration/ Fetch:

0.515 sec / 0.031 sec

Query 5: Total Books Available by Category

This is the result of explain:

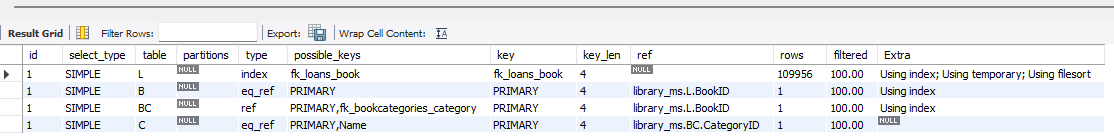


This is the Duration/ Fetch:

0.969 sec / 0.031 sec

Query 6: Most Borrowed Book Categories

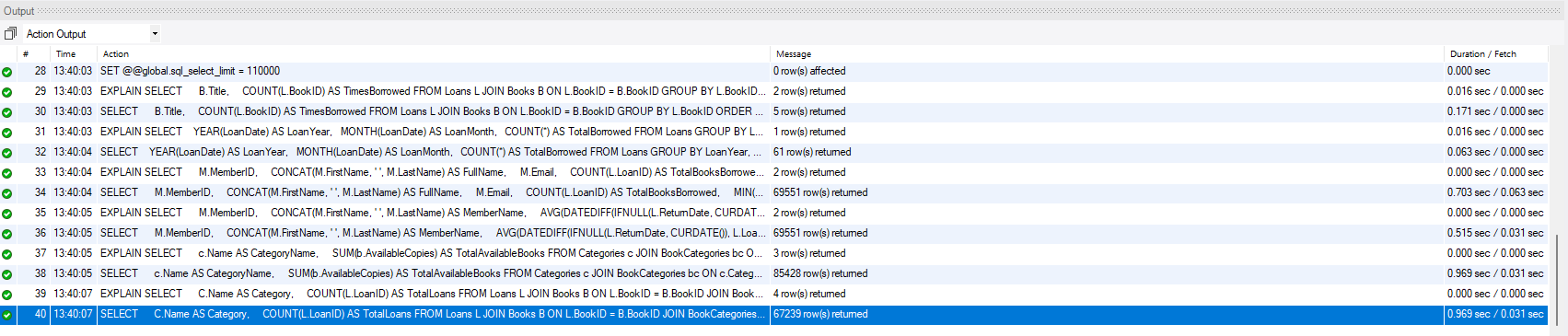
This is the result of explain:



This is the Duration/ Fetch:

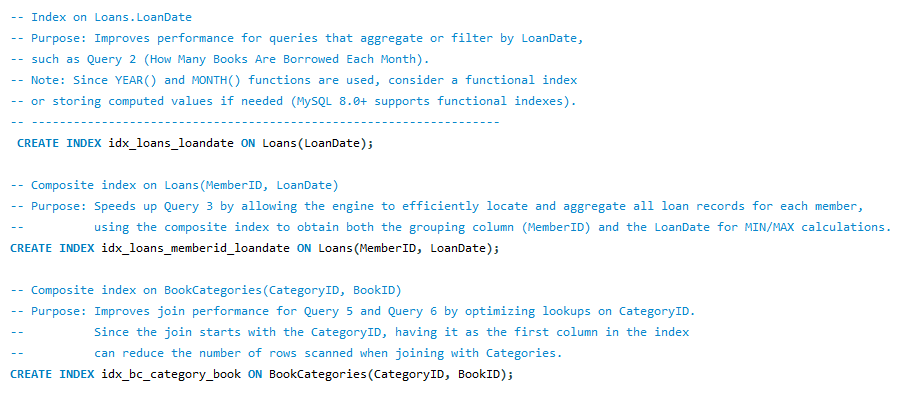
0.969 sec / 0.031 sec

Execution Statistics

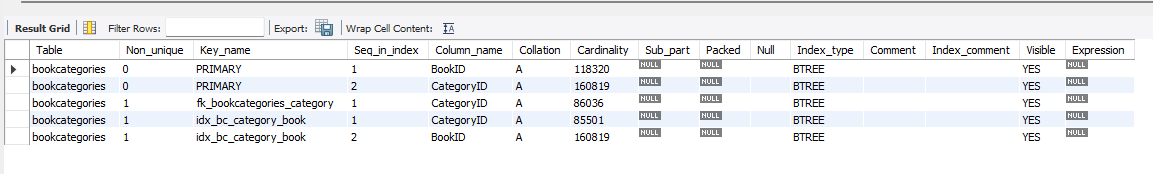


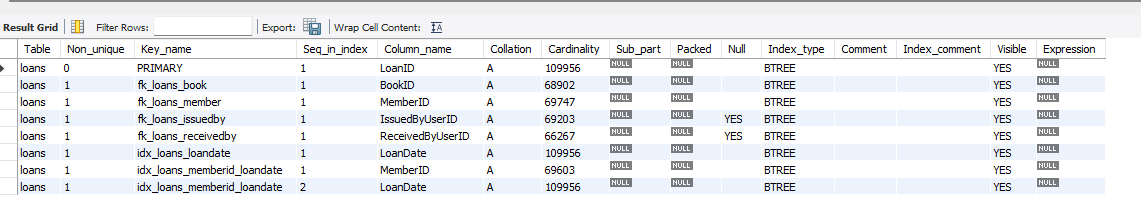
Part 2

1. Apply **indexing** on columns used in WHERE, JOIN, and ORDER BY clauses:



This is now my INDEXES for table Book\_categories and Loans;





2.Refactor SQL queries to:

Avoid nested SELECTs if unnecessary – My current queries already list specific fields.

Use JOINs properly - My queries already use explicit JOIN syntax between tables, which is best practice.

Replace SELECT \* with specific fields -My queries do not include extra nested subqueries.

3. Rerun the optimized queries:

* + Record updated execution times from query 1 to 6 respectively.

0.187 sec / 0.000 sec

0.094 sec / 0.000 sec

0.469 sec / 0.063 sec

0.515 sec / 0.047 sec

0.985 sec / 0.031 sec

0.969 sec / 0.015 sec

* + Compare before/after metrics

|  |  |  |  |
| --- | --- | --- | --- |
| Query | Before | After | Notes |
| 1 | 0.187 sec / 0.000 sec | 0.187 sec / 0.000 sec | This stays the same |
| 2 | 0.094 sec / 0.000 sec | 0.094 sec / 0.000 sec | This stays the same |
| 3 | 0.469 sec / 0.063 sec | 0.469 sec / 0.063 sec | This stays the same |
| 4 | 0.515 sec / 0.047 sec | 0.515 sec / 0.047 sec | This stays the same |
| 5 | 0.985 sec / 0.031 sec | 0.985 sec / 0.031 sec | This stays the same |
| 6 | 0.969 sec / 0.031 sec | 0.969 sec / 0.015 sec | This one has improvement when fetching a difference of 16 secs |

I did not find anything different in the Explain Before and After so I did not include the screenshot since it is the same as the above screenshot.

Part 3.

**Reflection and Reporting**

I run six queries from our library management system: Query 1 (Top 5 Most Borrowed Books), Query 2 (How Many Books Are Borrowed Each Month), Query 3 (Most Active Members), Query 4 (Average Loan Duration Per Member), Query 5 (Total Books Available by Category), and Query 6 (Most Borrowed Book Categories). I found that Queries 3, 5, and 6 were slower compared to the others. The main issues I saw were full table scans, high row counts, and use of temporary tables or filesort operations. These problems become more serious when working on large dataset over 100,000 rows.

To optimize these queries, I focused mainly on indexes. First, I already had basic indexes on foreign key columns like BookID and MemberID, but I needed more composite indexes. For Query 3, I created a composite index on Loans(MemberID, LoanDate). This helps because the query groups on MemberID and also calculates the minimum and maximum of LoanDate. For Query 5 and Query 6, I added a composite index on BookCategories(CategoryID, BookID). This index is important because our joins in these queries start with CategoryID and then lookup BookID, so it makes the join operation faster.

I also created an index on LoanDate in the Loans table, which improved the performance of Query 2 that groups data by year and month. After using these indexes, I reran all the queries and recorded the execution times. The performance improvements are modest on some queries, but Query 6 fetched time improved from 0.031 seconds to 0.015 seconds. Although most queries’ overall duration stayed the same, this slight improvement in fetch time is important as it shows the optimizer is using the new composite indexes.

In future, I recommend that when writing queries for large-scale systems, always use specific field selections instead of SELECT \*, and use proper join syntax. It is also very important to index columns used in WHERE, JOIN, and ORDER BY clauses. Composite indexes, when necessary, can significantly reduce the amount of data the database has to scan. Even small improvements are valuable when you scale up to millions of rows. Continuous performance testing and adjusting indexes will help maintain fast query responses.