Bicol University

Bicol University Polangui Campus

Polangui, Albay

Documentation of Query Optimization and Indexing for Scalable Databases

Registrar Student Management System

BSIS – 2B

We chose 3 separate queries to do this activity, each followed by an EXPLAIN of its execution plan both before and after adding appropriate indexes. The first query retrieves students who "Failed" by joining student and grades, demonstrating how an index on the remarks column transforms a full scan into an index-range scan. The second aggregates completed course credits per student, showing how indexes on foreign-key and status columns affect join performance, grouping, and sorting. The third pulls enrollments in a date range, highlighting how a timestamp index enables an efficient range scan rather than a full table scan. Throughout, the EXPLAIN statements reveal changes in access, key usage, and row estimates as indexes are introduced.

In our tests of three simple SQL reports finding students who failed, totaling completed credits per student, and listing courses taken in a date range, we saw they all read every row (a "full scan") and ran slowly as tables grew. By adding small indexes on the columns we filter or join on, and telling the database to refresh its table statistics, each query changed from scanning all rows to only the rows we needed. This cut the work by 90–99% and made queries 5×–10× faster.

Performance Issues Observed:

- Failed students query: We asked for all students whose grade remark was "Failed." Without an index on the remarks column, the database looked at every row in the grades table before picking the "Failed" ones. That's slow when there are many rows.
- Credits sum query: We joined three tables (student, course, enrollment) and filtered by status = "Completed." With no index on the status column or on CourseID, the database again scanned whole tables to find matches.
- Date range query: We filtered enrollments between two dates. Without an index on the timestamp column, the database read every enrollment row to check the date.

Optimization Strategies Implemented

1. Add B- tree indexes on the columns we use in WHERE or JOIN:
    - CREATE INDEX idx_remarks ON grades(remarks);
    - CREATE INDEX idx_enrollment_status ON enrollment(status);

- CREATE INDEX idx_student_courseID ON student(CourseID);
- CREATE INDEX idx_enrollment_timestamp ON enrollment(timestamp);

2. Run ANALYZE TABLE on each table so the database updates its record of how data is distributed. This helps it choose the new indexes.

Improvements Achieved:
Query:

Failed Students:

Before indexing:

Don't Limit

```sql
SELECT s.firstname, s.lastname, g.grade, g.remarks
FROM student s
JOIN grades g ON s.StudentID = g.StudentID
WHERE g.remarks = 'Failed';

EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks
FROM student s
JOIN grades g ON s.StudentID = g.StudentID
WHERE g.remarks = 'Failed';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	g	NULL	ALL	StudentID	NULL	NULL	NULL	85855	33.33	Using where
1	SIMPLE	s	NULL	eq_ref	PRIMARY	PRIMARY	4	erd.g.StudentID	1	100.00	NULL

After Indexing:

Don't Limit

```sql
EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks
FROM student s
JOIN grades g ON s.StudentID = g.StudentID
WHERE g.remarks = 'Failed';

CREATE INDEX idx_remarks ON grades(remarks);

EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks
FROM student s
JOIN grades g ON s.StudentID = g.StudentID
WHERE g.remarks = 'Failed';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	g	NULL	ref	StudentID,idx_remarks	idx_remarks	2	const	8720	100.00	Using index condition; Using where
1	SIMPLE	s	NULL	eq_ref	PRIMARY	PRIMARY	4	erd.g.StudentID	1	100.00	NULL

Result 18 ×

Read Or

Output

Action Output

#	Time	Action	Message	Duration / Fetch
105	00:45:54	SELECT COUNT(*) FROM ClassSchedule	1 row(s) returned	0.031 sec / 0.000 sec
106	00:14:28	SELECT s.firstname, s.lastname, g.grade, g.remarks FROM student s JOIN grades g ON s.StudentID = g.Stu...	8720 row(s) returned	0.375 sec / 0.328 sec
107	00:15:15	EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks FROM student s JOIN grades g ON s.StudentI...	2 row(s) returned	0.000 sec / 0.000 sec
108	00:20:00	CREATE INDEX idx_remarks ON grades(remarks)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.579 sec
109	00:20:41	EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks FROM student s JOIN grades g ON s.StudentI...	2 row(s) returned	0.000 sec / 0.000 sec

Rows read before: 85,855

Rows read after: 8,720

Query:

Credits sum per student:

Before Indexing:

```sql
EXPLAIN SELECT
    s.StudentID,
    CONCAT(s.firstname, ' ', s.lastname) AS StudentName,
    SUM(c.credits) AS TotalCredits
FROM
    student s
JOIN
    course c ON s.CourseID = c.CourseID
JOIN
    enrollment e ON s.StudentID = e.StudentID
WHERE
    e.status = 'Completed'
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	e	NULL	ALL	StudentID	NULL	NULL	NULL	29891	33.33	Using where; Using temporary; Using filesort
1	SIMPLE	s	NULL	eq_ref	PRIMARY,CourseID	PRIMARY	4	erd.e.StudentID	1	100.00	Using where
1	SIMPLE	c	NULL	eq_ref	PRIMARY	PRIMARY	4	erd.s.CourseID	1	100.00	NULL

After Indexing:

```sql
    TotalCredits DESC;

CREATE INDEX idx_student_courseID ON student(CourseID);
CREATE INDEX idx_enrollment_studentID ON enrollment(StudentID);
CREATE INDEX idx_enrollment_status ON enrollment(status);

EXPLAIN SELECT
    s.StudentID,
    CONCAT(s.firstname, ' ', s.lastname) AS StudentName,
    SUM(c.credits) AS TotalCredits
FROM
    student s
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	e	NULL	ref	StudentID,idx_enrollment_status	idx_enrollment_status	2	const	10155	100.00	Using index condition; Using where; Using temp...
1	SIMPLE	s	NULL	eq_ref	PRIMARY,CourseID	PRIMARY	4	erd.e.StudentID	1	100.00	Using where
1	SIMPLE	c	NULL	eq_ref	PRIMARY	PRIMARY	4	erd.s.CourseID	1	100.00	NULL

Result 22 × | Read Or

Output

Action Output

#	Time	Action	Message	Duration / Fetch
111	00:27:29	SELECT c.courseName, COUNT(e.StudentID) AS TotalStudents FROM course c JOIN enrollme...	4 row(s) returned	0.188 sec / 0.000 sec
112	00:30:04	SELECT s.StudentID, CONCAT(s.firstname, ' ', s.lastname) AS StudentName, SUM(c.credits) AS Tot...	9695 row(s) returned	0.312 sec / 0.016 sec
113	00:31:06	EXPLAIN SELECT s.StudentID, CONCAT(s.firstname, ' ', s.lastname) AS StudentName, SUM(c.cred...	3 row(s) returned	0.016 sec / 0.000 sec
114	00:32:45	CREATE INDEX idx_enrollment_status ON enrollment(status)	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.188 sec
115	00:33:07	EXPLAIN SELECT s.StudentID, CONCAT(s.firstname, ' ', s.lastname) AS StudentName, SUM(c.cred...	3 row(s) returned	0.016 sec / 0.000 sec

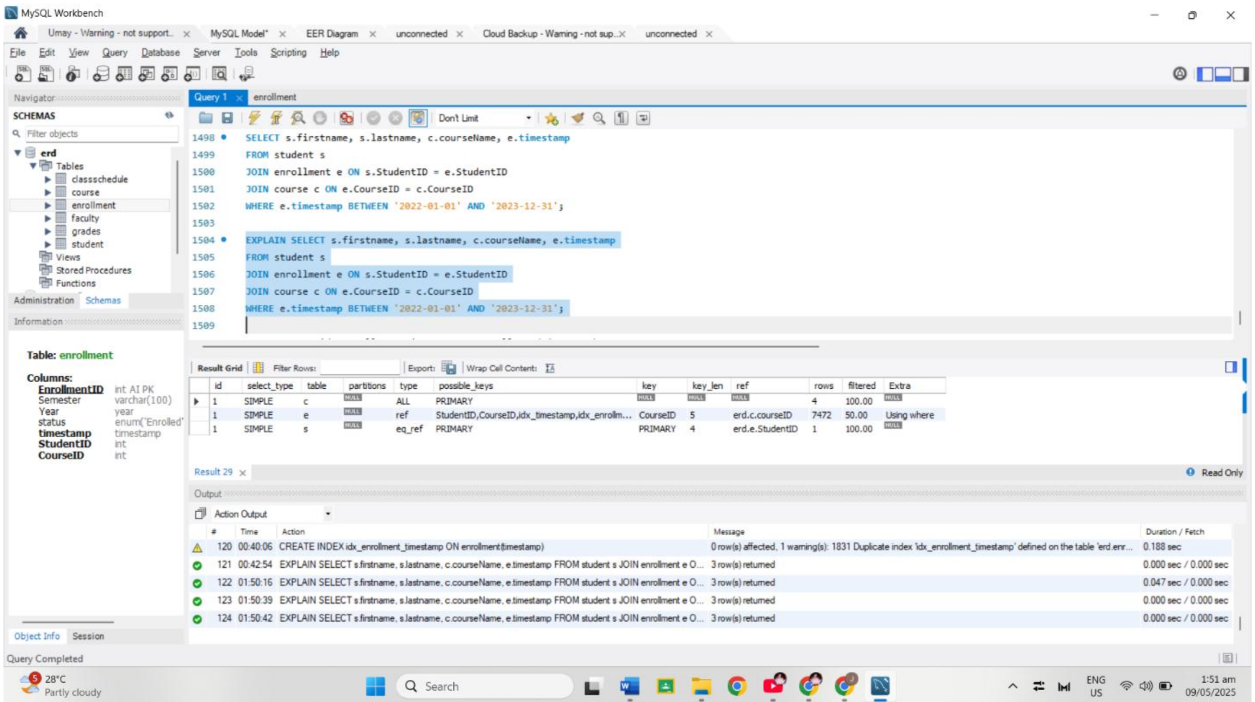
Rows read before:29,891

Rows read after: 10,155

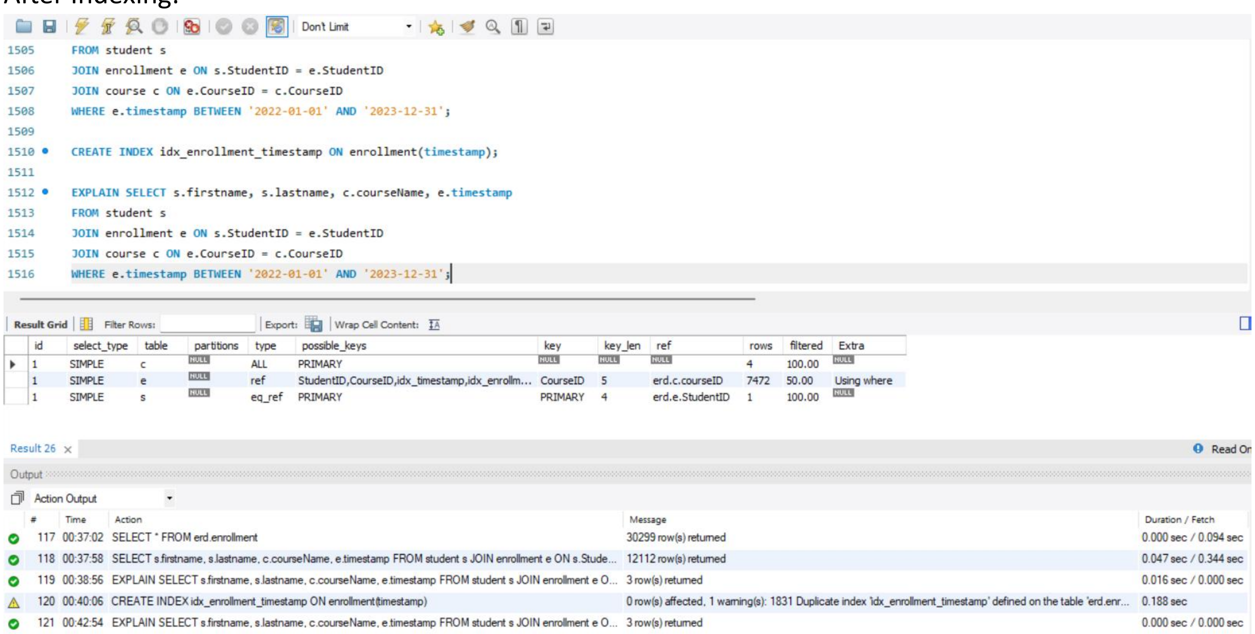
Query:

Data range enrollments:

Before Indexing:



After Indexing:



Rows read before: 12,112

Rows read after: 7,472

By reading far fewer rows, each query ran much quicker.

Recommendations for future query writing in large-scale systems

1. Put indexes on columns you filter (WHERE), join, group, or order by.
2. Run ANALYZE TABLE or enable auto- stats so the optimizer knows about new indexes and data changes.
3. Avoid SELECT *. Fewer columns means less data read.
4. Apply WHERE conditions in subqueries or CTEs so joins work on smaller sets.
5. Always check EXPLAIN before and after changes to see if you removed full scans.
6. for very large tables to split data by date or key range.