



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:

The screenshot shows a SQL query in the editor:

```

1397
1398
1399
1400 • SELECT s.firstname, s.lastname, g.grade, g.remarks
1401 FROM student s
1402 JOIN grades g ON s.StudentID = g.StudentID
1403 WHERE g.remarks = 'Failed';
1404
1405 • EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks
1406 FROM student s
1407 JOIN grades g ON s.StudentID = g.StudentID
1408 WHERE g.remarks = 'Failed';

```

Below the query, the 'Result Grid' shows the execution plan:

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	g	NONE	ALL	StudentID	NONE	NONE	NONE	85855	33.33	Using where
	1	SIMPLE	s	NONE	eq_ref	PRIMARY	PRIMARY	4	erd.g.StudentID	1	100.00	NONE

After Indexing:

The screenshot shows the same SQL query in the editor:

```

1404
1405 • EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks
1406 FROM student s
1407 JOIN grades g ON s.StudentID = g.StudentID
1408 WHERE g.remarks = 'Failed';
1409
1410 • CREATE INDEX idx_remarks ON grades(remarks);
1411
1412 • EXPLAIN SELECT s.firstname, s.lastname, g.grade, g.remarks
1413 FROM student s
1414 JOIN grades g ON s.StudentID = g.StudentID
1415 WHERE g.remarks = 'Failed';

```

Below the query, the 'Result Grid' shows the execution plan after indexing:

	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	g	NONE	ref	StudentID,idx_remarks	idx_remarks	2	const	8720	100.00	Using index condition; Using where
	1	SIMPLE	s	NONE	eq_ref	PRIMARY	PRIMARY	4	erd.g.StudentID	1	100.00	NONE

Below the execution plan, the 'Action Output' window shows the following log:

#	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.StudentID = g.Stu...	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.StudentID = g.Stu...	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:

```
1440 • EXPLAIN SELECT
1441     s.StudentID,
1442     CONCAT(s.firstname, ' ', s.lastname) AS StudentName,
1443     SUM(c.credits) AS TotalCredits
1444 FROM
1445     student s
1446 JOIN
1447     course c ON s.CourseID = c.CourseID
1448 JOIN
1449     enrollment e ON s.StudentID = e.StudentID
1450 WHERE
1451     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	HULL	ALL	StudentID	HULL	HULL	HULL	29891	33.33	Using where; Using temporary; Using filesort
	1	SIMPLE	s	HULL	eq_ref	PRIMARY, CourseID	PRIMARY	4	erd.e.StudentID	1	100.00	Using where
	1	SIMPLE	c	HULL	eq_ref	PRIMARY	PRIMARY	4	erd.s.CourseID	1	100.00	HULL

After Indexing:

```
1466     TotalCredits DESC;
1467
1468 • CREATE INDEX idx_student_courseID ON student(CourseID);
1469 • CREATE INDEX idx_enrollment_studentID ON enrollment(StudentID);
1470 • CREATE INDEX idx_enrollment_status ON enrollment(status);
1471
1472 • EXPLAIN SELECT
1473     s.StudentID,
1474     CONCAT(s.firstname, ' ', s.lastname) AS StudentName,
1475     SUM(c.credits) AS TotalCredits
1476 FROM
1477     student s
```

Result Grid												
Filter Rows:												
Export: <a href="#">Wrap Cell Content:</a> <a href="#">TS</a>												
	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
▶	1	SIMPLE	e	<small>HULL</small>	ref	StudentID,idx_enrollment_status	idx_enrollment_status	2	const	10155	100.00	Using index condition; Using where; Using temp...
	1	SIMPLE	s	<small>HULL</small>	eq_ref	PRIMARY,CourseID	PRIMARY	4	erd.e.StudentID	1	100.00	Using where
	1	SIMPLE	c	<small>HULL</small>	eq_ref	PRIMARY	PRIMARY	4	erd.s.CourseID	1	100.00	<small>HULL</small>

Result 22											
Output											
Action Output											
#	Time	Action	Message								Duration / Fetch
111	00:27:29	SELECT c.courseName, COUNT(e.StudentID) AS TotalStudents FROM course c JOIN enrollment...	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:

The screenshot shows the MySQL Workbench interface. The query editor contains the following SQL code:

```
1498 SELECT s.firstname, s.lastname, c.courseName, e.timestamp
1499 FROM student s
1500 JOIN enrollment e ON s.StudentID = e.StudentID
1501 JOIN course c ON e.CourseID = c.CourseID
1502 WHERE e.timestamp BETWEEN '2022-01-01' AND '2023-12-31';
1503
1504 EXPLAIN SELECT s.firstname, s.lastname, c.courseName, e.timestamp
1505 FROM student s
1506 JOIN enrollment e ON s.StudentID = e.StudentID
1507 JOIN course c ON e.CourseID = c.CourseID
1508 WHERE e.timestamp BETWEEN '2022-01-01' AND '2023-12-31';
1509
```

The left sidebar shows the database schema with tables: classschedule, course, enrollment, faculty, grades, student, and views. The 'enrollment' table is selected, showing its columns: EnrollmentID (int), Semester (int), Year (int), Status (enum), Timestamp (timestamp), StudentID (int), and CourseID (int).

The 'Result Grid' shows the execution plan for the query. The first row (id 1) is a SIMPLE table scan for the 'course' table. The second row (id 1) is a SIMPLE table scan for the 'student' table. The third row (id 1) is a SIMPLE table scan for the 'enrollment' table. The 'Extra' column for the enrollment row indicates 'Using where'.

The 'Action Output' shows the execution of the query, with a message indicating that 1831 duplicate index 'idx\_enrollment\_timestamp' defined on the table 'enr...' were affected.

After Indexing:

The screenshot shows the MySQL Workbench interface after indexing. The query editor contains the following SQL code:

```
1505 FROM student s
1506 JOIN enrollment e ON s.StudentID = e.StudentID
1507 JOIN course c ON e.CourseID = c.CourseID
1508 WHERE e.timestamp BETWEEN '2022-01-01' AND '2023-12-31';
1509
1510 CREATE INDEX idx_enrollment_timestamp ON enrollment(timestamp);
1511
1512 EXPLAIN SELECT s.firstname, s.lastname, c.courseName, e.timestamp
1513 FROM student s
1514 JOIN enrollment e ON s.StudentID = e.StudentID
1515 JOIN course c ON e.CourseID = c.CourseID
1516 WHERE e.timestamp BETWEEN '2022-01-01' AND '2023-12-31';
```

The 'Result Grid' shows the execution plan for the query. The first row (id 1) is a SIMPLE table scan for the 'course' table. The second row (id 1) is a SIMPLE table scan for the 'student' table. The third row (id 1) is a SIMPLE table scan for the 'enrollment' table. The 'Extra' column for the enrollment row indicates 'Using where'.

The 'Action Output' shows the execution of the query, with a message indicating that 1831 duplicate index 'idx\_enrollment\_timestamp' defined on the table 'enr...' were affected.

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.