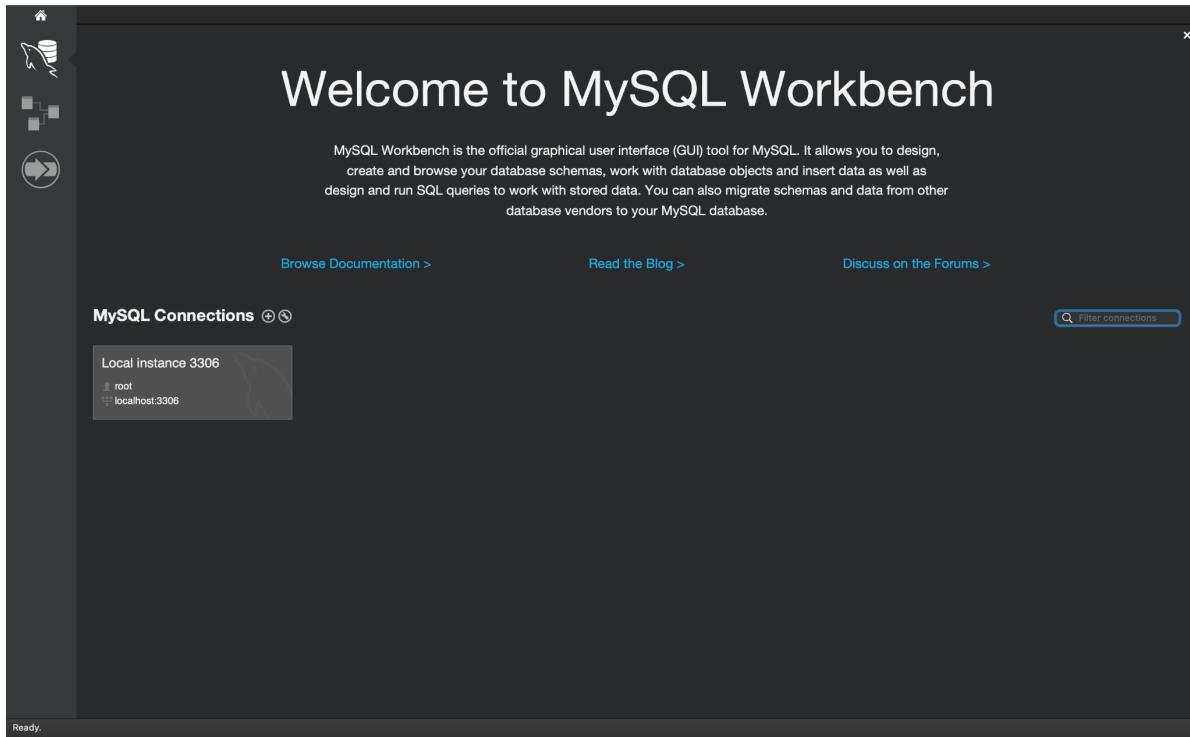


Database Design and Implementation

Database implementation

Currently, we implemented the entire database locally in MySQL. We used MySQL workbench to access and work on the database easily.

Here are the screenshots of the database (InsightifyDB):



This screenshot shows the MySQL Workbench interface for a local instance (3306). The main window displays the 'Info' tab for the 'InsightifyDB' schema. Key details shown include:

- Default collation: utf8mb4_0900_ai_ci
- Default characterset: utf8mb4
- Table count: 5
- Database size (rough estimate): 198.3 MiB

The left sidebar lists SCHEMAS with 'InsightifyDB' selected. The bottom status bar indicates 'SQL Editor Opened.'

This screenshot shows the MySQL Workbench interface after executing the 'show databases' query. The results are displayed in a Result Grid:

Database
information_schema
InsightifyDB
mysql
performance_schema
sys

The bottom status bar indicates 'Query Completed'.

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'SCHEMAS' tree, with 'InsightifyDB' selected. Under 'InsightifyDB', the 'Tables' node is expanded, showing tables such as Courses, CourseSchedule, Enrollments, Instructors, and Students. The main area contains a 'Query Grid' with the following content:

```

use InsightifyDB;
show tables;

```

The 'Result Grid' below shows the output of the 'show tables' command:

Tables_in_insightify...
Courses
CourseSchedule
Enrollments
Instructors
Students

The 'Action Output' section at the bottom lists the executed commands:

Action	Time	Response	Duration / Fetch Time
show databases	17:34:50	5 row(s) returned	0.0025 sec / 0.00000...
use InsightifyDB	17:35:25	0 row(s) affected	0.00069 sec
show tables	17:35:25	5 row(s) returned	0.0033 sec / 0.00000...

Creating Tables

For this stage, we added the following five tables:

- 1) Courses
- 2) Students
- 3) Instructors
- 4) CourseSchedule
- 5) Enrollments

Here are the DDL commands we used for the creation of these tables:

```

CREATE TABLE Courses(
    CRN VARCHAR(7) NOT NULL PRIMARY KEY,
    Title VARCHAR(104) NOT NULL,
    Description VARCHAR(1384)
);

```

```
CREATE TABLE Students(
    Firstname VARCHAR(11) NOT NULL,
    Lastname  VARCHAR(11) NOT NULL,
    NetId     VARCHAR(15) NOT NULL PRIMARY KEY
);

CREATE TABLE Instructors(
    ID INTEGER NOT NULL PRIMARY KEY,
    Instructor_name VARCHAR(25) NOT NULL
);

CREATE TABLE CourseSchedule(
    CRN VARCHAR(7) NOT NULL,
    YearTerm VARCHAR(11) NOT NULL,
    instructor_id INTEGER NOT NULL,
    PRIMARY KEY(CRN,YearTerm),
    CONSTRAINT CourseSchedule_fk FOREIGN KEY (CRN) REFERENCES Courses(CRN),
    CONSTRAINT CourseSchedule_fk1 FOREIGN KEY (instructor_id) REFERENCES
Instructors(ID)
);

CREATE TABLE Enrollments(
    NetId VARCHAR(15) NOT NULL,
    CRN VARCHAR(7) NOT NULL,
    Term VARCHAR(11) NOT NULL,
    Credits INTEGER NOT NULL,
    Grade VARCHAR(2) NOT NULL,
    Score NUMERIC(3,1) NOT NULL,
    PRIMARY KEY(NetId,CRN),
    CONSTRAINT enrollments_fk FOREIGN KEY (NetId) REFERENCES Students(NetId),
    CONSTRAINT enrollments_fk1 FOREIGN KEY (CRN) REFERENCES Courses(CRN)
);
```

Here are the screenshots of the tables:

The screenshot shows the MySQL Workbench interface with the following details:

- Schemas:** InsightifyDB
- Tables:** Courses, CourseSchedule, Enrollments, Instructors, Students
- Action Output:**

Time	Action	Response	Duration / Fetch Time
17:34:50	show databases	5 row(s) returned	0.0025 sec / 0.00000...
17:35:25	use InsightifyDB	0 row(s) affected	0.00069 sec
17:35:25	show tables	5 row(s) returned	0.0033 sec / 0.00000...
- Message:** Query Completed

Inserting data into the Tables

We inserted the data into the tables in the following way:

1. Courses:
 - a. Source: <https://github.com/chin123/gpa/blob/master/uiuc-course-catalog.csv>
 - b. #rows: 8589
2. Students:
 - a. Source: Generated using faker package
 - b. #rows: 48962
3. Instructors:
 - a. Source: <https://github.com/chin123/gpa/blob/master/uiuc-gpa-dataset.csv>
 - b. #rows: 8683
4. CourseSchedule:
 - a. Source: <https://github.com/chin123/gpa/blob/master/uiuc-gpa-dataset.csv>
 - b. #rows: 30682
5. Enrollments:
 - a. Source: Generated using random package
 - b. #rows: 1533620

Here are the screenshots showing the count of rows in each table:

This screenshot shows the MySQL Workbench interface with a query results window. The query executed is `select count(*) from Courses;`. The result grid displays a single row with the value 8589. The right-hand sidebar contains tabs for Result Grid, Form Editor, Field Types, Query Stats, and Execution Plan.

count(*)
8589

Action Output

Time	Action
7 18:46:46	select count(*) from Courses LIMIT 0, 1000

Response

1 row(s) returned

Duration / Fetch Time

0.025 sec / 0.000017...

Query Completed

This screenshot shows the MySQL Workbench interface with a query results window. The query executed is `select count(*) from CourseSchedule;`. The result grid displays a single row with the value 30682. The right-hand sidebar contains tabs for Result Grid, Form Editor, Field Types, Query Stats, and Execution Plan.

count(*)
30682

Action Output

Time	Action
8 18:47:22	select count(*) from CourseSchedule LIMIT 0, 1000

Response

1 row(s) returned

Duration / Fetch Time

0.036 sec / 0.000016...

Query Completed

Local instance 3306

Administration Schemas Query 1

MANAGEMENT

- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

INSTANCE

- Startup / Shutdown
- Server Logs
- Options File

PERFORMANCE

- Dashboard
- Performance Reports
- Performance Schema Setup

Query 1

```
1 • select count(*) from Enrollments;
2
```

Result Grid

count(*)
1533620

Result 6

Action Output

Action	Time	Response	Duration / Fetch Time
select count(*) from Enrollments LIMIT 0, 1000	18:48:07	1 row(s) returned	0.358 sec / 0.000022...

Query Completed

Read Only

Local instance 3306

Administration Schemas Query 1

MANAGEMENT

- Server Status
- Client Connections
- Users and Privileges
- Status and System Variables
- Data Export
- Data Import/Restore

INSTANCE

- Startup / Shutdown
- Server Logs
- Options File

PERFORMANCE

- Dashboard
- Performance Reports
- Performance Schema Setup

Query 1

```
1 • select count(*) from Instructors;
2
```

Result Grid

count(*)
8683

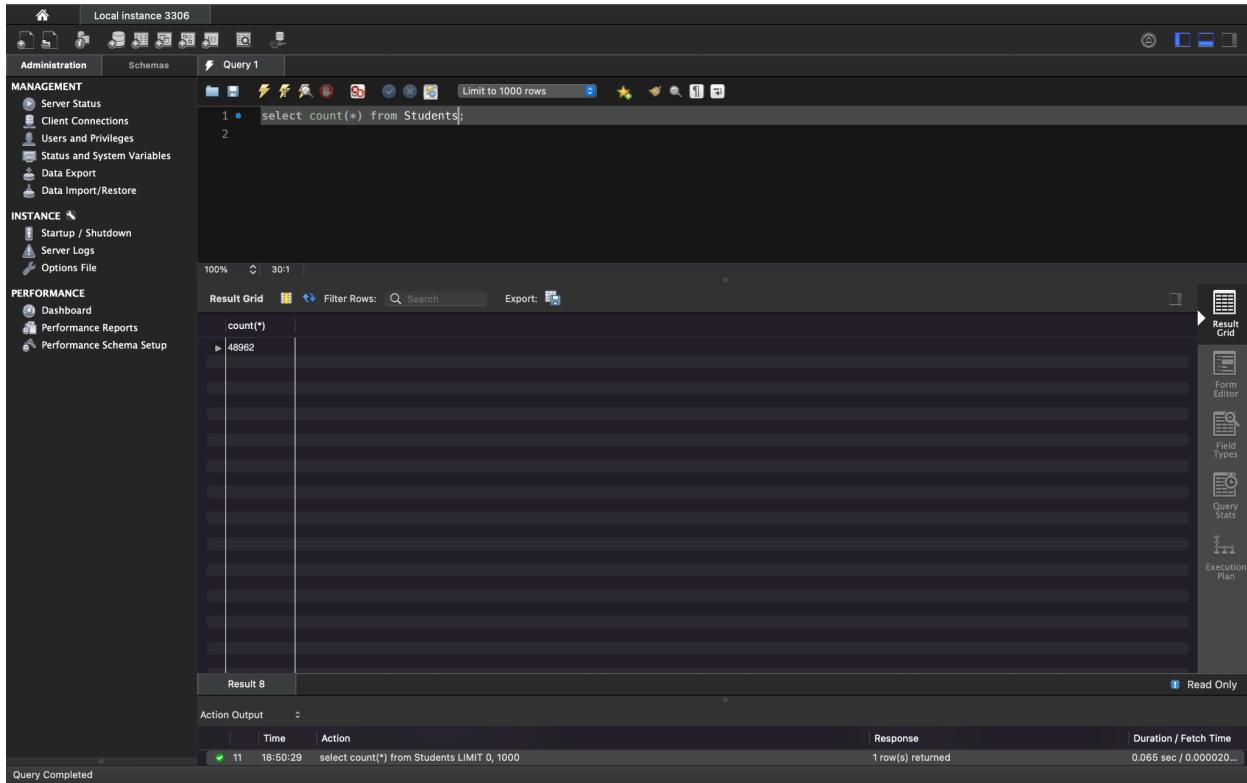
Result 7

Action Output

Action	Time	Response	Duration / Fetch Time
select count(*) from Instructors LIMIT 0, 1000	18:49:05	1 row(s) returned	0.029 sec / 0.000012...

Query Completed

Read Only



Advanced Queries

Here are the 2 advanced queries and their explanation:

Query 1

```
SELECT CS.YearTerm, I.Instructor_name, AVG(E.Score)
FROM CourseSchedule CS JOIN Instructors I on (CS.instructor_id=I.ID) Join
Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term)
WHERE CS.CRN = "CS411"
GROUP BY CS.YearTerm,I.Instructor_name
limit 15;
```

Explanation: This query finds the list of all terms and the instructor for that term and the average score that term given a particular course (CRN). In our application, CRN is the course number (like CS411). In our application, the student first selects a course and then we display the term, instructor and average score data. In the query, we are assuming the given CRN as “CS411”. The student can check more details than this, but for now we are implementing this feature.

Here are the top 15 rows returned when this query is executed:

The screenshot shows a SQL database interface with the following details:

- Schemas:** InsighifyDB
- Tables:** Courses, CourseSchedule, Enrollments, Instructors, Students
- Query:**

```
1 SELECT CS.YearTerm, I.Instructor_name, AVG(E.Score)
2 FROM CourseSchedule CS JOIN Instructors I on (CS.instructor_id=I.ID) Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term)
3 WHERE CS.CRN = "CS411"
4 GROUP BY CS.YearTerm,I.Instructor_name
5 limit 15;
```
- Result Grid:** Shows 15 rows of data with columns: YearTerm, Instructor_name, and AVG(E.Score). The data includes various terms from Fall-2010 to Fall-2016 and different instructors like Saurabh Sinha, Kevin C Chang, Peixiang Zhao, and Timothy Weninger.
- Right Panel:** Includes tabs for Result Grid, Form Editor, Field Types, Query Stats, and Execution Plan.
- Status Bar:** Shows "Query Completed" and "Result 7".

Query 2

```
SELECT Grade, COUNT(*)
FROM CourseSchedule CS Join Enrollments E on (CS.CRN=E.CRN and
CS.YearTerm=E.Term)
WHERE CS.CRN = "CS411" and CS.instructor_id = 2931
GROUP BY Grade
ORDER BY Grade
limit 15;
```

Explanation: This query finds the Grade distribution (Grade and the total number of students who got this grade) for a particular course and particular instructor. In our application, a student will be able to see how a particular instructor grades students for a course. The student will be able to get other metrics and other types of queries as well (like grade distribution for a particular term), but this data (in current query) will be shown at the high level when a student checks for an instructor and course.

Here are the top 15 rows returned when this query is executed:

The screenshot shows the MySQL Workbench interface. The left sidebar displays the 'SCHEMAS' tree with 'InsightifyDB' selected. The main area contains a SQL editor window with the following query:

```

SELECT Grade, COUNT(*)
FROM CourseSchedule CS Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term)
WHERE CS.CRN = "CS411" and CS.instructor_id = 2931
GROUP BY Grade
ORDER BY Grade
limit 15;
    
```

The results are displayed in a grid titled 'Result Grid' with the following data:

Grade	COUNT(*)
A	42
A-	45
A+	45
B	45
B-	44
B+	44
C	39
C-	46
C+	35
D	35
D-	50
D+	43
F	42

The status bar at the bottom indicates 'Query Completed' and 'Read Only'.

Indexing on Query 1

EXPLAIN ANALYZE on Query 1 without adding any new index:

```

mysql>
mysql> EXPLAIN ANALYZE SELECT CS.YearTerm, I.Instructor_name, AVG(E.Score)
   FROM CourseSchedule CS JOIN Instructors I ON (CS.instructor_id=I.ID) Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term)
   WHERE CS.CRN = "CS411"
   GROUP BY CS.YearTerm,I.Instructor_name;
+-----+
| EXPLAIN
+-----+
| > Table scan on <temporary> (actual time=18.100 ,18.100 rows=31 loops=1)
|   -> Aggregate using temporary table (actual time=18.095 ..18.095 rows=31 loops=1)
|     -> Nested loop inner join (cost=3636.11 rows=1567) (actual time=0 .352 ..14.765 rows=1567 loops=1)
|       -> Nested loop inner join (cost=3087.66 rows=1567) (actual time=0 .260 ..13.702 rows=1567 loops=1)
|         -> Index lookup on E using mytable.fki (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0 .233 ..4.312 rows=1567 loops=1)
|           -> Covering Index lookup on CS using PRIMARY (CRN='CS411', YearTerm,Term) (cost=0.99 rows=1) (actual time=0 .004 ..0.006 rows=1 loops=1567)
|             -> Single-row Index lookup on I using PRIMARY (ID=cs.instructor_id) (cost=0.25 rows=1) (actual time=0 .000 ..0.000 rows=1 loops=1567)
| 
+-----+
1 row in set (0.03 sec)

mysql> 
    
```

- All the primary keys of all tables will already be indexed by default. So, we tried to index using different combinations of attributes which are not primary keys and not indexed by default.

- We can see above that the execution time is 0.03 seconds when we did not add any new index.

Adding Index on Score attribute of Enrollments table:

```

mysql>
mysql>
mysql> mysql> create index score_idx on Enrollments(Score);
Query OK, 0 rows affected (0.28 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql>
mysql> EXPLAIN ANALYZE SELECT CS.YearTerm, I.Instructor_name, AVG(E.Score) FROM CourseSchedule CS JOIN Instructors I on (CS.instructor_id=I.ID) Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE CS.CRN = "CS411" GROUP BY CS.YearTerm,I.Instructor_name;
+-----+
| EXPLAIN
+-----+
| > Table scan on <temporary> (actual time=19.785..19.802 rows=31 loops=1)
  -> Aggregate using temporary table (actual time=19.771..19.774 rows=1 loops=1)
    -> Nested loop inner join (Cost=3087.66 rows=1567) (actual time=0.200..14.913 rows=1567 loops=1)
      -> Nested loop inner join (Cost=3087.66 rows=1567) (actual time=0.151..14.101 rows=1567 loops=1)
        -> Index lookup on E using mytable.fki (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0.132..4.952 rows=1567 loops=1)
          -> Covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm=Term) (cost=0.99 rows=1) (actual time=0.005..0.006 rows=1 loops=1567)
            -> Single-row index lookup on I using PRIMARY (ID=cs.instructor_id) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=1567)
+-----+
1 row in set (0.02 sec)

mysql> |

```

- As we can see above, when we added index on Score attribute and ran EXPLAIN ANALYZE, we got the execution time as 0.02 which is marginally less than the time taken before without adding any index (0.03)
- We indexed on this attribute to experiment indexing on a new attribute part of aggregation.
- We saw a very marginal improvement probably because of the very low size of data. The execution time without new indexes itself is very low at 0.03 seconds.

Adding Index on Instructor_name attribute of Instructors table (after adding index on Score):

```

mysql>
mysql> EXPLAIN ANALYZE SELECT CS.YearTerm, I.Instructor_name, AVG(E.Score) FROM CourseSchedule CS JOIN Instructors I on (CS.instructor_id=I.ID) Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE CS.CRN = "CS411" GROUP BY CS.YearTerm, I.Instructor_name;
+-----+
| EXPLAIN
+-----+
| > Table scan on <temporary> (actual time=11.886..11.895 rows=31 loops=1)
  -> Aggregate using temporary table (actual time=11.882..11.883 rows=31 loops=1)
    -> Nested loop inner join (Cost=3636.11 rows=1567) (actual time=0.061..0.936 rows=1567 loops=1)
      -> Nested loop inner join (Cost=3087.66 rows=1567) (actual time=0.055..0.892 rows=1567 loops=1)
        -> Index lookup on E using mytable.fki (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0.043..2.680 rows=1567 loops=1)
          -> Covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm=Term) (cost=0.99 rows=1) (actual time=0.003..0.004 rows=1 loops=1567)
            -> Single-row index lookup on I using PRIMARY (ID=cs.instructor_id) (cost=0.25 rows=1) (actual time=0.000..0.000 rows=1 loops=1567)
+-----+
1 row in set (0.01 sec)

mysql> |

```

- We added an index on Instructor_name as it is part of “Group by”. We think adding index on this attribute will help as “GROUP BY” internally may require table scan on this attribute.

- As we can see above, when we added index on Score attribute and ran EXPLAIN ANALYZE, we got the execution time as 0.01 which is marginally less than the time taken before (0.02)
- We saw a very marginal improvement probably because of the very low size of data. The execution time without new indexes itself is very low at 0.03 seconds.

Index with Instructor_name alone:

```

mysql> show index from Instructors;
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Instructors | 0 | PRIMARY | 1 | ID | A | 8643 | NULL | NULL | BTREE | | YES | NULL |
| Instructors | 1 | instructor_name_idx | 1 | Instructor_name | A | 8643 | NULL | NULL | BTREE | | YES | NULL |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
mysql>
mysql>
mysql>
mysql>
mysql>
mysql> EXPLAIN ANALYZE SELECT CS.YearTerm, I.Instructor_name, AVG(E.Score) FROM CourseSchedule CS JOIN Instructors I on (CS.instructor_id=I.ID) Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE YearTerm,I.Instructor_name;
+-----+
| EXPLAIN
+-----+
+-----+
| > Table scan on <temporary> (actual_time=13.224,13.233 rows=31 loops=1)
  > Aggregate using temporary table (actual_time=13.215,13.215 rows=31 loops=1)
    > Nested loop inner join (cost=3636.11 rows=1567) (actual_time=0.258..10.815 rows=1567 loops=1)
      > Nested loop inner join (cost=3087.66 rows=1567) (actual_time=0.232..10.057 rows=1567 loops=1)
        > Index lookup on E using mytable_fk1 (CRN='CS411') (cost=1386.56 rows=1567) (actual_time=0.183..3.324 rows=1567 loops=1)
        > Covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm='e.Term') (cost=0.99 rows=1) (actual_time=0.003..0.004 rows=1 loops=1567)
      > Single-row index lookup on I using PRIMARY (ID=cs.instructor_id) (cost=0.25 rows=1) (actual_time=0.000..0.000 rows=1 loops=1567)
| 
+-----+
1 row in set (0.02 sec)

mysql> 

```

- As we can see above, when we added an index on Instructor_name attribute (without adding index on Score) and ran EXPLAIN ANALYZE, we got the execution time as 0.02 which is exactly the same as the time taken when we had only the Score index.
- This is probably because of the very low size of data. The execution time without new indexes itself is very low at 0.03 seconds.

Index selection:

- As per the metrics we saw in the above three experiments, the best design would be to add index on both Score and Instructor_name attributes
- But, these new index bring down the execution time only marginally down
- This is probably because of the very low size of data. The execution time without new indexes itself is very low at 0.03 seconds.
- Ideally, for such low volume databases, it would be wiser to not add new indexes as they may take up more memory with only very marginal improvements in execution time which in any case does not impact the calling application

Indexing on Query 2

EXPLAIN ANALYZE on Query 2 without adding any new index:

```
mysql>
mysql>
mysql> EXPLAIN ANALYZE SELECT Grade, COUNT(*) FROM CourseSchedule CS Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE CS.CRN = "CS411" and CS.instructor_id = 2931 GROUP BY Grade ORDER BY Grade;
+-----+
| EXPLAIN
+-----+
| > Sort: e.Grade (actual time=4.357..4.358 rows=13 loops=1)
|   -> Table scan on <temporary> (actual time=3.903..3.905 rows=13 loops=1)
|     -> Aggregate using temporary table (actual time=3..896..3.896 rows=13 loops=1)
|       -> Nested loop inner join (cost=3087.66 rows=1567) (actual time=2.103..3.500 rows=555 loops=1)
|         -> Index lookup on E using mytable.fki (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0.092..2.675 rows=1567 loops=1)
|           -> Single-row covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm=e.Term, instructor_id=2931) (cost=0.99 rows=1) (actual time=0.000..0.000 rows=0 loops=1567)
+-----+
1 row in set (0.02 sec)
```

- We can see above that the execution time is 0.03 seconds when we did not add any new index.

Index on CRN:

```
mysql> EXPLAIN ANALYZE SELECT Grade, COUNT(*) FROM CourseSchedule CS Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE CS.CRN = "CS411" and CS.instructor_id = 2931 GROUP BY Grade ORDER BY Grade;
+-----+
| EXPLAIN
+-----+
| > Sort: e.Grade (actual time=6.117..6.118 rows=13 loops=1)
|   -> Table scan on <temporary> (actual time=6.073..6.075 rows=13 loops=1)
|     -> Aggregate using temporary table (actual time=6.073..6.073 rows=13 loops=1)
|       -> Nested loop inner join (cost=1935.01 rows=1567) (actual time=3.457..5.496 rows=555 loops=1)
|         -> Index lookup on E using mytable.fki (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0.077..4.096 rows=1567 loops=1)
|           -> Single-row covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm=e.Term, instructor_id=2931) (cost=0.25 rows=1) (actual time=0.001..0.001 rows=0 loops=1567)
+-----+
1 row in set (0.02 sec)

mysql> show index from CourseSchedule;
+-----+
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-----+
| CourseSchedule | 0 | PRIMARY | 1 | CRN | A | 3315 | NULL | NULL | BTREE | YES | NULL |
| CourseSchedule | 0 | PRIMARY | 2 | YearTerm | A | 24492 | NULL | NULL | BTREE | YES | NULL |
| CourseSchedule | 0 | PRIMARY | 3 | instructor_id | A | 24492 | NULL | NULL | BTREE | YES | NULL |
| CourseSchedule | 1 | CourseSchedule_fk1 | 1 | instructor_id | A | 6321 | NULL | NULL | BTREE | YES | NULL |
| CourseSchedule | 1 | crn_indx | 1 | CRN | A | 3575 | NULL | NULL | BTREE | YES | NULL |
+-----+
5 rows in set (0.03 sec)

mysql> ||
```

- As we can see above, when we added an index on CRN attribute and ran EXPLAIN ANALYZE, we got the execution time as 0.02 which is exactly the same as the time taken when we had no additional index.
- This is probably because of the very low size of data and already insignificant execution time.

Index on grade:

```
mysql>
mysql>
mysql> create index grade_idx on Enrollments(Grade);
Query OK, 0 rows affected (4.23 sec)
Records: 0  Duplicates: 0  Warnings: 0
mysql>
mysql> EXPLAIN ANALYZE SELECT Grade, COUNT(*) FROM CourseSchedule CS Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE CS.CRN = "CS411" and CS.instructor_id = 2931 GROUP BY Grade ORDER BY Grade;
+-----+
| EXPLAIN
+-----+
I > Sort: e.Grade (actual time=7.157..7.157 rows=13 loops=1)
   -> Table scan on <temporary> (actual time=6.826..6.828 rows=13 loops=1)
      -> Aggregate using temporary table (actual time=6.825..6.825 rows=13 loops=1)
         -> Nested loop inner join (cost=3087.66 rows=1567) (actual time=4.265..6.173 rows=555 loops=1)
            -> Index lookup on E using mytable_fk1 (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0.489..5.288 rows=1567 loops=1)
               -> Single-row covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm=e.Term, instructor_id=2931) (cost=0.99 rows=1) (actual time=0.000..0.000 rows=0 loops=1567)
|
+-----+
1 row in set (0.01 sec)

mysql>
```

- We added an index on Grade as it is part of “Group by”. We think adding index on this attribute will help as “GROUP BY” internally may require table scan on this attribute.
- As we can see above, when we added index on Score attribute and ran EXPLAIN ANALYZE, we got the execution time as 0.01 which is marginally less than the time taken before (0.02)
- We saw a very marginal improvement probably because of the very low volume of data.

Index on instructor ID:

```
mysql>
mysql>
mysql> EXPLAIN ANALYZE SELECT Grade, COUNT(*) FROM CourseSchedule CS Join Enrollments E on (CS.CRN=E.CRN and CS.YearTerm=E.Term) WHERE CS.CRN = "CS411" and CS.instructor_id = 2931 GROUP BY Grade;
+-----+
| EXPLAIN
+-----+
I > Sort: e.Grade (actual time=5.970..5.971 rows=13 loops=1)
   -> Table scan on <temporary> (actual time=5.538..5.541 rows=13 loops=1)
      -> Aggregate using temporary table (actual time=5.534..5.534 rows=13 loops=1)
         -> Nested loop inner join (cost=3087.66 rows=1567) (actual time=2.294..4.834 rows=555 loops=1)
            -> Index lookup on E using mytable_fk1 (CRN='CS411') (cost=1386.56 rows=1567) (actual time=0.244..3.976 rows=1567 loops=1)
               -> Single-row covering index lookup on CS using PRIMARY (CRN='CS411', YearTerm=e.Term, instructor_id=2931) (cost=0.99 rows=1) (actual time=0.000..0.000 rows=0 loops=1567)
|
+-----+
1 row in set (0.02 sec)

mysql> show index from Instructors;
+-----+
| Table    | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-----+
| Instructors |     0 | PRIMARY |          1 | ID          | A          | 8643       |          1 | BTREE   |      | BTREE        |          |           | YES     | NULL      |
| Instructors |     1 | Ins_Id_Idx |          1 | ID          | A          | 8643       |          1 | BTREE   |      | BTREE        |          |           | YES     | NULL      |
+-----+
2 rows in set (0.03 sec)

mysql>
```

- As we can see above, when we added an index on ID attribute and ran EXPLAIN ANALYZE, we got the execution time as 0.02 which is exactly the same as the time taken when we had no additional index.
- This is probably because of the very low size of data and already insignificant execution time.

Index selection:

- As per the metrics we saw in the above three experiments, the best design would be to add an index on the Grade attribute.
- But, this new index bring down the execution time only marginally down
- This is probably because of the very low size of data. The execution time without new indexes itself is very low at 0.02 seconds.
- Ideally, for such low volume databases, it would be wiser to not add new indexes as they may take up more memory with only very marginal improvements in execution time which in any case does not impact the calling application.