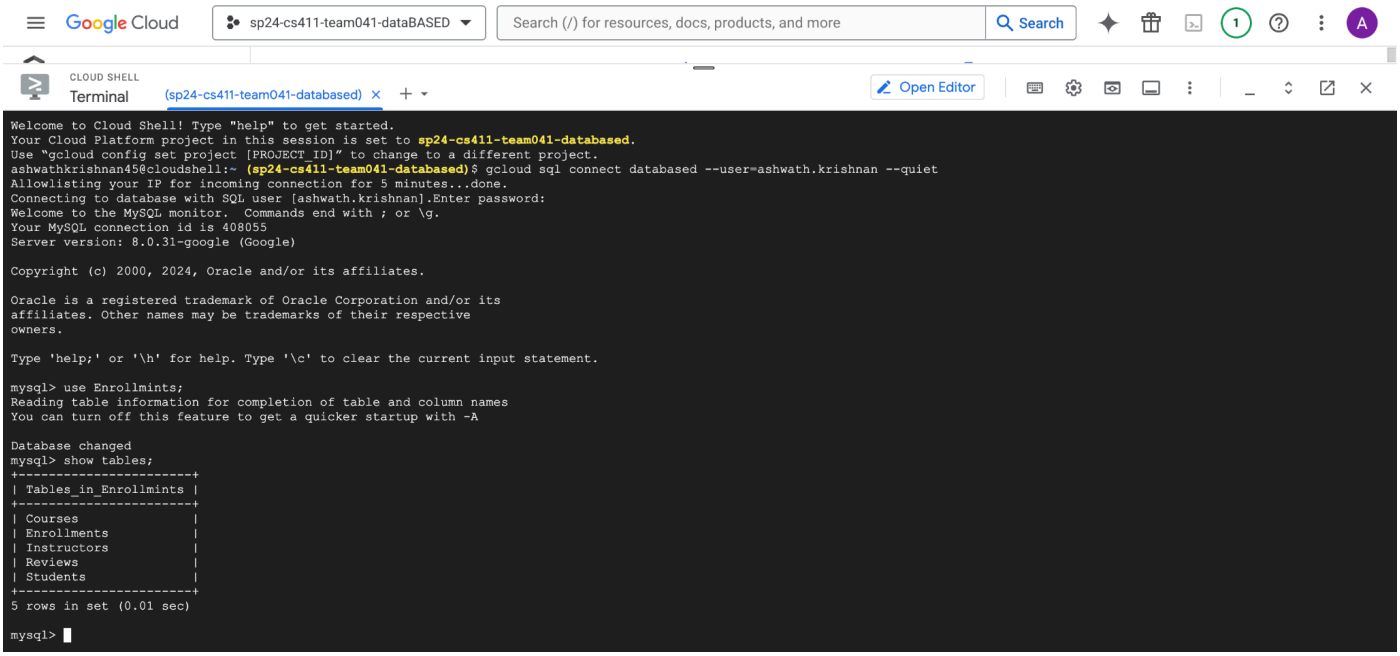
Stage 3: Database Implementation and Indexing

The Database has been implemented on GCP



Data Definition Language commands to create tables:

Courses:

```
Create table Courses (
        CRN int,
        Course_Code varchar(10),
        Course_Name varchar(255),
        Semester varchar(20),
        Credits varchar(15),
        Breadth varchar(255),
        Department varchar(50),
        Is_Online boolean,
        Time varchar(50),
        Day varchar(10),
        Location varchar(255),
        Type varchar(50),
        Section varchar(10),
        Present_Enrollments int,
```

```
    Max_Enrollments int,
    InstructorID int,
    Start_Time time,
    End_Time time,
    primary key(CRN, Semester),
    foreign key(InstructorID) references Instructors(InstructorID)
    on update cascade
    on delete cascade
);
```

Instructors:

```
Create table Instructors(
    InstructorID int,
    ProfessorName varchar(255),
    WebLink varchar(1000)
);
```

Students:

```
Create table Students(
    NetID int,
    Student_Name varchar(255),
    Is_International boolean,
    Degree_Name varchar(255),
    Semester int
);
```

Reviews:

```
Create table Reviews (
    Review_ID int,
    Course_Code varchar(10),
    Review varchar(5000),
    Semester varchar(20),
    InstructorID int,
    Rating float,
    Primary key(Review_ID),
    Foreign key(InstructorID) references Instructors(InstructorID)
    on update cascade
    on delete cascade
```

);

Enrollments:

```
Create table Enrollments(
        NetID int,
        CRN int,
        Semester varchar(20),
        primary key(NetID, CRN, Semester),
        foreign key(CRN, Semester) references Courses(CRN, Semester),
        foreign key(NetID) references Students(NetID)
        on update cascade
        on delete cascade
);
```

COUNT OF ROWS FOR EACH TABLE:

```
mysql> show tables;
+-----------------------+
| Tables_in_Enrollmints |
+-----------------------+
| Courses               |
| Enrollments           |
| Instructors           |
| Reviews               |
| Students              |
+-----------------------+
5 rows in set (0.01 sec)

mysql> select count(*) from Courses;
+----------+
| count(*) |
+----------+
|     1494 |
+----------+
1 row in set (0.01 sec)

mysql> select count(*) from Enrollments;
+----------+
| count(*) |
+----------+
|     8905 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from Instructors;
+----------+
| count(*) |
+----------+
|      247 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from Reviews;
+----------+
| count(*) |
+----------+
|      456 |
+----------+
1 row in set (0.00 sec)

mysql> select count(*) from Students;
+----------+
| count(*) |
+----------+
|     2000 |
+----------+
1 row in set (0.00 sec)
```

COMPLEX QUERIES:

1. List the courses offered by the highest-rated professors

   Query:

```
Select distinct c.Course_Code, c.Course_Name,
subquery.InstructorID, subquery.rating
From Courses c natural join
        (Select avg(rating) as rating, InstructorID
        From Reviews
        Group by InstructorID
        Order by avg(rating) desc) subquery
where c.Semester = "Spring 2024"
order by subquery.rating desc;
```

```
+-------------+-------------------------------------------------+--------------+--------------------+
| Course_Code | Course_Name                                     | InstructorID | rating             |
+-------------+-------------------------------------------------+--------------+--------------------+
| CS 199      | Undergraduate Open Seminar in Computer Science  |        12015 |  3.479034948767277 |
| CS 341      | System Programming                              |        12015 |  3.479034948767277 |
| CS 411      | Database Systems                                |        12011 | 3.2837634848993877 |
| CS 277      | Algorithms and Data Structures for Data Science |        12204 |  3.230302265712193 |
| CS 444      | Deep Learning for Computer Vision               |        12145 | 2.7807091041044756 |
| CS 199      | Undergraduate Open Seminar in Computer Science  |        12037 | 2.6792704463005066 |
| CS 124      | Introduction to Computer Science I              |        12037 | 2.6792704463005066 |
| CS 510      | Advanced Information Retrieval                  |        12238 |  2.435778667529424 |
| CS 173      | Discrete Structures                             |        12057 | 2.3361438512802124 |
| CS 199      | Undergraduate Open Seminar in Computer Science  |        12057 | 2.3361438512802124 |
+-------------+-------------------------------------------------+--------------+--------------------+
10 rows in set (0.01 sec)
```

The number of rows is 10 since we only have the Reviews of 9 Professors. And on further restricting the courses to “Spring 2024”, we only get 10 courses for the Spring semester.

To reduce this cost, we will create an index on Courses for (InstructorID, Semester), so that Instructors for the same Semester can be indexed together.

```
------+
| -> Limit: 10 row(s)  (actual time=0.999..1.001 rows=10 loops=1)
    -> Sort: subquery.rating DESC, limit input to 10 row(s) per chunk  (actual time=0.999..0.999 rows=10 loops=1)
        -> Table scan on <temporary>  (cost=3777.32..3814.28 rows=2758) (actual time=0.986..0.989 rows=10 loops=1)
            -> Temporary table with deduplication  (cost=3777.31..3777.31 rows=2758) (actual time=0.985..0.985 rows=10 loops=1)
                -> Nested loop inner join  (cost=3501.49 rows=2758) (actual time=0.705..0.925 rows=43 loops=1)
                    -> Filter: (subquery.InstructorID is not null)  (cost=0.12..53.80 rows=456) (actual time=0.637..0.641 rows=10 loops=1)
                        -> Table scan on subquery  (cost=2.50..2.50 rows=0) (actual time=0.636..0.638 rows=10 loops=1)
                            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=0.635..0.635 rows=10 loops=1)
                                -> Sort: rating DESC  (actual time=0.628..0.628 rows=10 loops=1)
                                    -> Stream results  (cost=93.70 rows=456) (actual time=0.060..0.617 rows=10 loops=1)
                                        -> Group aggregate: avg(Reviews.Rating)  (cost=93.70 rows=456) (actual time=0.058..0.612 rows=10 loops=1)
                                            -> Index scan on Reviews using InstructorID  (cost=48.10 rows=456) (actual time=0.052..0.571 rows=456 loops=1)
                    -> Index lookup on c using InstructorID (InstructorID=subquery.InstructorID), with index condition: (c.Semester = 'Spring 2024')  (cost=1.51 rows=6) (actual time
=0.021..0.028 rows=4 loops=10)
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

This might be particularly beneficial because it directly supports the join and filter conditions used in the query, allowing for faster retrieval by combining both key columns.

CREATE INDEX idx_courses_ins_sem ON Courses(InstructorID, Semester);

```
| -> Sort: subquery.rating DESC  (actual time=1.093..1.093 rows=10 loops=1)
    -> Table scan on <temporary>  (cost=592.60..610.05 rows=1197) (actual time=1.082..1.083 rows=10 loops=1)
        -> Temporary table with deduplication  (cost=592.59..592.59 rows=1197) (actual time=1.080..1.080 rows=10 loops=1)
            -> Nested loop inner join  (cost=472.86 rows=1197) (actual time=0.869..1.023 rows=43 loops=1)
                -> Filter: (subquery.InstructorID is not null)  (cost=0.12..53.80 rows=456) (actual time=0.820..0.823 rows=10 loops=1)
                    -> Table scan on subquery  (cost=2.50..2.50 rows=0) (actual time=0.818..0.820 rows=10 loops=1)
                        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=0.816..0.816 rows=10 loops=1)
                            -> Sort: rating DESC  (actual time=0.807..0.807 rows=10 loops=1)
                                -> Stream results  (cost=93.70 rows=456) (actual time=0.076..0.790 rows=10 loops=1)
                                    -> Group aggregate: avg(Reviews.Rating)  (cost=93.70 rows=456) (actual time=0.072..0.784 rows=10 loops=1)
                                        -> Index scan on Reviews using idx_reviews_instr  (cost=48.10 rows=456) (actual time=0.065..0.737 rows=456 loops=1)
                -> Index lookup on c using idx_courses_ins_sem (InstructorID=subquery.InstructorID, Semester='Spring 2024')  (cost=0.66 rows=3) (actual time=0.013..0.019 rows=4 loops=10)
|
```

As can be seen, the cost has significantly reduced from 3777.32 to **592.60**

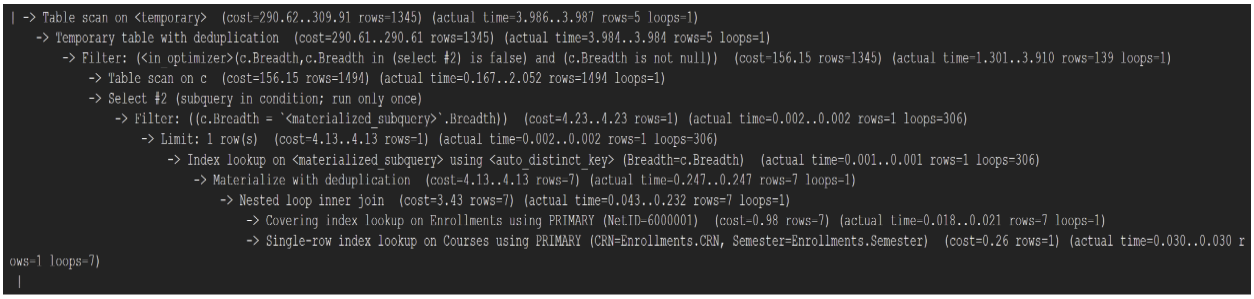
2. <u>List the Breadth Courses that a student with NetID=600002 could choose to complete their Breadth requirements</u>

   <u>Query:</u>

   ```
   SELECT DISTINCT c.Breadth FROM Courses c
   WHERE c.Breadth NOT IN
       (SELECT DISTINCT Courses.Breadth
       FROM Enrollments JOIN Courses ON Enrollments.CRN =
       Courses.CRN AND Enrollments.Semester = Courses.Semester
       WHERE Enrollments.NetID = 6000002)
       AND c.Breadth IS NOT NULL;
   ```

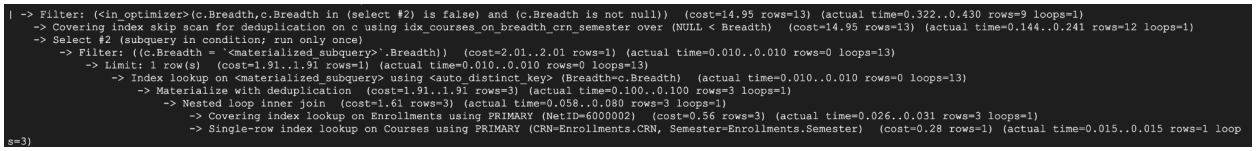
```
+-----------------------------------------+
| Breadth                                 |
+-----------------------------------------+
| Database and Information Systems        |
| Interactive Computing                   |
| Architecture, Compilers, Parallel Computing |
| Bioinformatics and Computational Biology |
| Computers and Education                 |
| Security and Privacy                    |
+-----------------------------------------+
6 rows in set (0.01 sec)
```

The output only contains 6 rows since the total number of Breadths is only 11 and the student has already covered 5 Breadths.



The query involves a JOIN operation between the Enrollments and Courses tables based on the CRN and Semester columns. Having these columns indexed in the Courses table means that the database can more efficiently perform this join, as it can quickly find matching rows in the Courses table using the index. Without an index, the database might have to perform a full scan of the Courses table to find the rows that satisfy the conditions set in the main query and the subquery. To help with deduplication, we can try the following index on Breadth, CRN, and Semester.

CREATE INDEX idx_courses_on_breadth_crn_semester ON Courses(Breadth, CRN, Semester);



Cost before index: 290.62
Cost after creating index: **14.95**

Indexing Breadth, CRN, and Semester helps in reducing the time it takes to execute queries involving these columns by improving join performance, filtering speed, and the efficiency of operations like DISTINCT.

3. List the Advanced Courses that a student with NetID=600002 could choose to complete their Depth requirements

   Query:

   SELECT c.Course_Code, c.Course_Name
   FROM Courses c
   LEFT JOIN Enrollments e ON e.CRN = c.CRN AND e.Semester = c.Semester AND e.NetID = 600002
   WHERE c.Course_Code LIKE 'CS 5%%' AND c.Course_Code <> 'CS 591' AND e.CRN IS NULL
   LIMIT 15;

```
mysql> SELECT c.Course_Code, c.Course_Name
    -> FROM Courses c
    -> LEFT JOIN Enrollments e ON e.CRN = c.CRN AND e.Semester = c.Semester AND e.NetID = 600002
    -> WHERE c.Course_Code LIKE 'CS 5%%' AND c.Course_Code <> 'CS 591'
    -> AND e.CRN IS NULL
    -> LIMIT 15;
+-------------+-----------------------------------------------+
| Course_Code | Course_Name                                   |
+-------------+-----------------------------------------------+
| CS 500      | Current Topics in Computing Education Research |
| CS 500      | Current Topics in Computing Education Research |
| CS 500      | Current Topics in Computing Education Research |
| CS 500      | Current Topics in Computing Education Research |
| CS 507      | Topics in Cryptography                        |
| CS 508      | Manycore Parallel Algorithms                  |
| CS 510      | Advanced Information Retrieval                |
| CS 510      | Advanced Information Retrieval                |
| CS 511      | Advanced Data Management                      |
| CS 511      | Advanced Data Management                      |
| CS 511      | Advanced Data Management                      |
| CS 512      | Data Mining Principles                        |
| CS 512      | Data Mining Principles                        |
| CS 512      | Data Mining Principles                        |
| CS 513      | Theory & Practice of Data Cleaning            |
+-------------+-----------------------------------------------+
15 rows in set (0.00 sec)
```

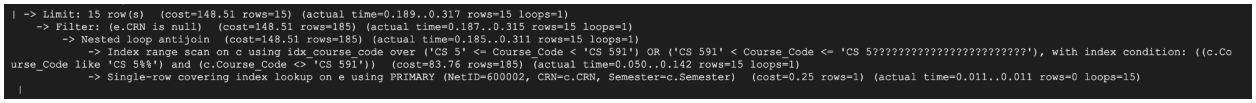
```
| -> Limit: 15 row(s)  (cost=208.43 rows=15) (actual time=0.196..0.237 rows=15 loops=1)
    -> Filter: (e.CRN is null)  (cost=208.43 rows=149) (actual time=0.194..0.235 rows=15 loops=1)
        -> Nested loop antijoin  (cost=208.43 rows=149) (actual time=0.193..0.233 rows=15 loops=1)
            -> Filter: ((c.Course_Code like 'CS 5%%') and (c.Course_Code <> 'CS 591'))  (cost=156.15 rows=149) (actual time=0.182..0.194 rows=15 loops=1)
                -> Table scan on c  (cost=156.15 rows=1494) (actual time=0.095..0.158 rows=175 loops=1)
            -> Single-row covering index lookup on e using PRIMARY (NetID=600002, CRN=c.CRN, Semester=c.Semester)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=0 loops=15)
|
```

To help lessen this cost, we can try this:

CREATE INDEX idx_course_code ON Courses (Course_Code);

Direct Index Use: The index provides a fast lookup for rows that potentially match the pattern 'CS 5%%', thus reducing the number of rows that need to be scanned to satisfy the filter conditions. This is more efficient than

scanning the entire table. Since Course_Code does not have a very broad range of values, we did not observe a significant improvement in cost.



Other indexing designs tried:

CREATE INDEX idx_courses_crn_semester ON Courses (CRN, Semester);

CREATE INDEX idx_enrollments_crn_semester_netid ON Enrollments (CRN, Semester, NetID);

4. Check the schedule of the student with NetID 6000001 for this semester to find if there is a conflict in timings, returning the conflict

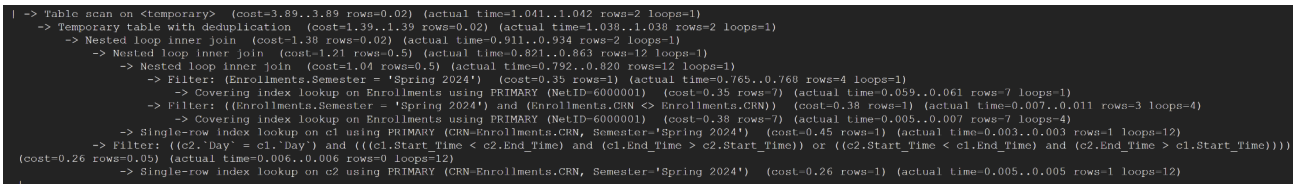
Query:

```
SELECT DISTINCT c1.Course_Code AS Course1, c2.Course_Code AS Course2, c1.Time AS Time1, c2.Time AS Time2
FROM Courses c1 JOIN Courses c2 ON c1.Day = c2.Day AND c1.Semester = c2.Semester AND c1.CRN != c2.CRN
WHERE c1.CRN IN (
        SELECT CRN FROM Enrollments WHERE NetID = 6000001 AND Semester = 'Spring 2024')
AND c2.CRN IN (
        SELECT CRN FROM Enrollments WHERE NetID = 6000001 AND Semester = 'Spring 2024')
AND ((c1.Start_Time < c2.End_Time AND c1.End_Time > c2.Start_Time)
OR (c2.Start_Time < c1.End_Time AND c2.End_Time > c1.Start_Time))
AND c1.Semester = 'Spring 2024';
```

There are only 2 tuples in the output since the student only has 2 courses in his schedule that are conflicts. There may be more for other students.

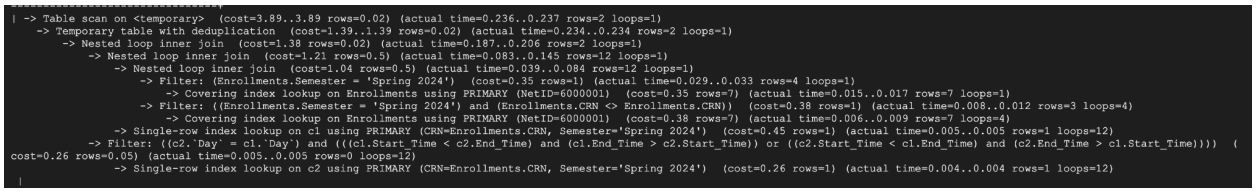
```
+---------+---------+--------------------+--------------------+
| Course1 | Course2 | Time1              | Time2              |
+---------+---------+--------------------+--------------------+
| CS 412  | CS 586  | 11:00AM - 12:15PM | 11:00AM - 12:20PM |
| CS 586  | CS 412  | 11:00AM - 12:20PM | 11:00AM - 12:15PM |
+---------+---------+--------------------+--------------------+
2 rows in set (0.01 sec)
```

Using EXPLAIN ANALYZE, we see many of the costs are associated with nested loop inner joins.



To reduce this cost, we will create an index on Courses for (CRN, Semester, Day), so that courses with the same day can be indexed together for faster retrieval.

CREATE INDEX idx_crn_semester_day ON Courses(CRN, Semester, Day);



However, on implementing this index, it can be seen that no cost improvements have been made. This is because CRN and Semester are already keys for the Courses table. Moreover, The query joins the Courses table to itself based on multiple conditions (Day, Semester, and different CRNs), which complicates how effectively an index can be used. The optimizer might find that maintaining two sets of index scans for the self-join is more costly than other methods like hash joins or nested loops without index support.

Other Indexing designs tried:

CREATE INDEX idx_courses_day_semester_crn_time ON Courses(Day, Semester, CRN, Start_Time, End_Time);
CREATE INDEX idx_enrollments_netid_semester_crn ON Enrollments(NetID, Semester, CRN);