Database Design.MD

````
# Database Design

## DDL Commands
```sql
CREATE TABLE IF NOT EXISTS Users (
    UserID INT PRIMARY KEY,
    Name VARCHAR(255),
    Email VARCHAR(255)
);

CREATE TABLE IF NOT EXISTS Professors (
    ProfessorID INT PRIMARY KEY,
    Name VARCHAR(255),
    Department VARCHAR(255),
    RMP_Link VARCHAR(255)
);

CREATE TABLE IF NOT EXISTS Courses (
    CourseID INT PRIMARY KEY, --random
    CourseNumber VARCHAR(255), --cs411
    ProfessorID INT,
    Title VARCHAR(255),
    FOREIGN KEY (ProfessorID) REFERENCES Professors(Professor
ID)
);

CREATE TABLE IF NOT EXISTS Ratings (
    RatingID INT PRIMARY KEY,
    Score DECIMAL(5, 2),
    WouldTakeAgain BOOLEAN,
    CourseID INT,
    UserID INT,
````
````

````
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID),
    FOREIGN KEY (UserID) REFERENCES Users(UserID)
);

CREATE TABLE IF NOT EXISTS Comments (
    CommentID INT PRIMARY KEY,
    Content TEXT,
    CourseID INT,
    UserID INT,
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID),
    FOREIGN KEY (UserID) REFERENCES Users(UserID)
);

```

-- Grade -- Calculate AVG, MEDIAN, 25%, 75%ile
-- rating,num_ratings,level_of_difficulty

## Advanced Queries

**Query 1**: Courses with the highest average rating, including

```sql
SELECT c.CourseID, c.Title, p.Name AS ProfessorName, p.Departmer
FROM Courses c
JOIN Professors p ON c.ProfessorID = p.ProfessorID
JOIN Ratings r ON c.CourseID = r.CourseID
GROUP BY c.CourseID, c.Title, p.Name, p.Department
ORDER BY AverageRating DESC;
```

**Query 2**: Determine the percentage of users who would take a
```sql
SELECT c.CourseID, c.Title,
````

```
    COALESCE(
    (SELECT ROUND(SUM(CASE WHEN r.WouldTakeAgain THEN 1 ELSE 0 E
    FROM Ratings r
    WHERE r.CourseID = c.CourseID
    GROUP BY r.CourseID),
    0) AS PercentageWouldTakeAgain
FROM Courses c
LEFT JOIN Ratings r ON c.CourseID = r.CourseID
GROUP BY c.CourseID, c.Title
ORDER BY PercentageWouldTakeAgain DESC;
```

Query 3: Retrieve popular courses, defined by the number of
```sql
SELECT c.CourseID, c.Title, p.Name AS ProfessorName,
       COUNT(DISTINCT com.CommentID) AS NumberOfComments,
       COALESCE(AVG(r.Score), 00) AS AverageRating
FROM Courses c
LEFT JOIN Comments com ON c.CourseID = com.CourseID
LEFT JOIN Ratings r ON c.CourseID = r.CourseID
JOIN Professors p ON c.ProfessorID = p.ProfessorID
GROUP BY c.CourseID, c.Title, p.Name
ORDER BY NumberOfComments DESC, AverageRating DESC;
```

Query 4: Average GPA of all courses taught by a specific Pro
```sql
SELECT
    Pr.Name AS ProfessorName,
    Pr.Department,
    AVG(OverallCourseScore) AS AvgProfessorScore
FROM
    (
        SELECT
            C.ProfessorID,
```

```
            C.Title AS CourseTitle,
            AVG(R.Score) AS OverallCourseScore
        FROM Courses C
        JOIN Ratings R ON C.CourseID = R.CourseID
        GROUP BY C.CourseID
    ) AS CourseScores
JOIN Professors Pr ON CourseScores.ProfessorID = Pr.ProfessorID
GROUP BY Pr.ProfessorID
```

Database Connection

```
shawkatabrar@cloudshell:~ (cs-411-nullpointers)$ gcloud config set project cs-411-nullpointers
Updated property [core/project].
shawkatabrar@cloudshell:~ (cs-411-nullpointers)$ goloud sql connect testbed --user=chef
-bash: goloud: command not found
shawkatabrar@cloudshell:~ (cs-411-nullpointers)$ gcloud sql connect testbed --user=chef
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [chef].Enter password:
ERROR 2003 (HY000): Can't connect to MySQL server on '35.239.129.65:3306' (110)
shawkatabrar@cloudshell:~ (cs-411-nullpointers)$ goloud sql connect testbed --user=chef
-bash: goloud: command not found
shawkatabrar@cloudshell:~ (cs-411-nullpointers)$ gcloud sql connect testbed --user=chef
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [chef].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 12580
Server version: 8.0.31-google (Google)
```

```
mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| omniscient         |
| performance_schema |
| sandwiches         |
| sys                |
+--------------------+
6 rows in set (0.00 sec)

mysql> use omniscient
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+----------------------+
| Tables_in_omniscient |
+----------------------+
| Comments             |
| Courses              |
| Department           |
| Professors           |
| Ratings              |
| Users                |
+----------------------+
6 rows in set (0.00 sec)
```

Showing the insertion of at least 1000 rows to each table

```
mysql> SELECT COUNT(*) FROM Comments;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM Courses;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM Professors;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM Ratings;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM Users;
+----------+
| COUNT(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql>
```

Top 15 from Advanced Queries:

Top 15 from Query 1:

```
with pool.connect() as db_conn:
  results = db_conn.execute(sqlalchemy.text("""
      SELECT c.CourseID, c.Title, p.Name AS ProfessorName, p.Department, AVG(r.Score) AS AverageRating
      FROM Courses c
      JOIN Professors p ON c.ProfessorID = p.ProfessorID
      JOIN Ratings r ON c.CourseID = r.CourseID
      GROUP BY c.CourseID, c.Title, p.Name, p.Department
      ORDER BY AverageRating DESC LIMIT 15;
  """)).fetchall()

  # show results
  for row in results:
    print(row)
```

```
(791, 'innovate end-to-end communities', 'Stanley Barnett', 'goal', Decimal('5.000000'))
(663, 'benchmark mission-critical eyeballs', 'Debra Jones', 'article', Decimal('4.990000'))
(503, 'extend value-added eyeballs', 'James Taylor', 'decide', Decimal('4.970000'))
(794, 'disintermediate impactful e-services', 'Jeremy Bernard', 'sport', Decimal('4.970000'))
(746, 'seize bricks-and-clicks interfaces', 'Mary Chavez', 'policy', Decimal('4.960000'))
(39, 'benchmark end-to-end web services', 'Michelle Leblanc', 'environmental', Decimal('4.940000'))
(278, 'enhance frictionless web services', 'Brooke Smith', 'director', Decimal('4.930000'))
(225, 'extend frictionless convergence', 'Leslie Alexander', 'education', Decimal('4.920000'))
(619, 'grow collaborative e-tailers', 'Richard Hicks', 'team', Decimal('4.910000'))
(31, 'reinvent rich networks', 'Stanley Barnett', 'goal', Decimal('4.910000'))
(280, 'envisioneer viral eyeballs', 'Michael Stanley', 'your', Decimal('4.905000'))
(706, 'reinvent intuitive e-services', 'Margaret Johnson', 'include', Decimal('4.890000'))
(749, 'unleash transparent systems', 'Jeffery Klein', 'put', Decimal('4.890000'))
(945, 'reinvent B2C mindshare', 'Jonathan Burke', 'example', Decimal('4.870000'))
(305, 'iterate transparent channels', 'Ethan Torres', 'reflect', Decimal('4.850000'))
```

Top 15 from Query 2:

```
with pool.connect() as db_conn:
  results = db_conn.execute(sqlalchemy.text("""
    SELECT c.CourseID, c.Title,
    COALESCE(
    (SELECT ROUND(SUM(CASE WHEN r.WouldTakeAgain THEN 1 ELSE 0 END) * 100.0 / COUNT(r.WouldTakeAgain), 2)
    FROM Ratings r
    WHERE r.CourseID = c.CourseID
    GROUP BY r.CourseID),
    0) AS PercentageWouldTakeAgain
    FROM Courses c
    LEFT JOIN Ratings r ON c.CourseID = r.CourseID
    GROUP BY c.CourseID, c.Title
    ORDER BY PercentageWouldTakeAgain DESC LIMIT 15;
  """)).fetchall()

  # show results
  for row in results:
    print(row)
```

```
(41, 'visualize bricks-and-clicks interfaces', Decimal('100.00'))
(2, 'repurpose turn-key portals', Decimal('100.00'))
(4, 'architect vertical e-services', Decimal('100.00'))
(5, 'transition strategic supply-chains', Decimal('100.00'))
(43, 'enhance dot-com web-readiness', Decimal('100.00'))
(26, 'visualize cutting-edge e-business', Decimal('100.00'))
(8, 'harness compelling action-items', Decimal('100.00'))
(9, 'architect front-end infrastructures', Decimal('100.00'))
(20, 'expedite compelling models', Decimal('100.00'))
(40, 'redefine bricks-and-clicks web-readiness', Decimal('100.00'))
(12, 'innovate clicks-and-mortar vortals', Decimal('100.00'))
(13, 'expedite vertical paradigms', Decimal('100.00'))
(57, 'strategize rich niches', Decimal('100.00'))
(18, 'deliver web-enabled ROI', Decimal('100.00'))
(16, 'e-enable cross-media content', Decimal('100.00'))
```

Top 15 from Query 3:

```
with pool.connect() as db_conn:
  results = db_conn.execute(sqlalchemy.text("""
      SELECT c.CourseID, c.Title, p.Name AS ProfessorName,
      COUNT(DISTINCT com.CommentID) AS NumberOfComments,
      COALESCE(AVG(r.Score), 00) AS AverageRating
      FROM Courses c
      LEFT JOIN Comments com ON c.CourseID = com.CourseID
      LEFT JOIN Ratings r ON c.CourseID = r.CourseID
      JOIN Professors p ON c.ProfessorID = p.ProfessorID
      GROUP BY c.CourseID, c.Title, p.Name
      ORDER BY NumberOfComments DESC, AverageRating DESC LIMIT 15;
  """)).fetchall()

  # show results
  for row in results:
    print(row)
```

```
(806, 'reinvent plug-and-play initiatives', 'David Collins', 6, Decimal('1.780000'))
(914, 'reinvent plug-and-play deliverables', 'Lori Smith', 6, Decimal('0.000000'))
(615, 'expedite collaborative infrastructures', 'Dustin Elliott', 5, Decimal('2.610000'))
(665, 'embrace efficient technologies', 'Michael Obrien', 5, Decimal('0.000000'))
(818, 'drive interactive action-items', 'Diane Daniels', 4, Decimal('4.840000'))
(420, 'syndicate enterprise e-tailers', 'Danny Powers', 4, Decimal('4.730000'))
(754, 'repurpose mission-critical action-items', 'Andrew Murray', 4, Decimal('4.065000'))
(850, 'incubate bleeding-edge e-business', 'Eric Chapman', 4, Decimal('3.580000'))
(164, 'drive dynamic platforms', 'Ralph Fox', 4, Decimal('3.320000'))
(872, 'revolutionize enterprise schemas', 'Valerie Sandoval', 4, Decimal('3.140000'))
(837, 'utilize back-end e-commerce', 'Debbie Hodge', 4, Decimal('3.090000'))
(61, 'transition end-to-end niches', 'Bradley Patterson', 4, Decimal('2.970000'))
(144, 'generate dynamic eyeballs', 'Jennifer Rivera', 4, Decimal('2.960000'))
(255, 'engage extensible web services', 'Kelly Parks', 4, Decimal('2.806667'))
(492, 'cultivate value-added functionalities', 'Jason Powell', 4, Decimal('2.460000'))
```

Top 15 from Query 4:

```python
with pool.connect() as db_conn:

    result = db_conn.execute(sqlalchemy.text("""
        SELECT
          Pr.Name AS ProfessorName,
          Pr.Department,
          AVG(OverallCourseScore) AS AvgProfessorScore
      FROM
          (
              SELECT
                  C.ProfessorID,
                  C.Title AS CourseTitle,
                  AVG(R.Score) AS OverallCourseScore
              FROM Courses C
              JOIN Ratings R ON C.CourseID = R.CourseID
              GROUP BY C.CourseID
          ) AS CourseScores
      JOIN Professors Pr ON CourseScores.ProfessorID = Pr.ProfessorID
      GROUP BY Pr.ProfessorID LIMIT 15;
      """))

    # show results
    for row in result:
      print(row)
```

```
('Joseph Morton', 'nothing', Decimal('1.8700000000'))
('Elizabeth Burton', 'man', Decimal('2.9800000000'))
('Scott Bennett', 'member', Decimal('4.1700000000'))
('Dylan Browning', 'second', Decimal('3.5100000000'))
('Jonathan Flynn', 'where', Decimal('3.3200000000'))
('John Jackson', 'inside', Decimal('3.0600000000'))
('Katherine Galloway', 'heavy', Decimal('2.9733330000'))
('Tiffany Carter', 'free', Decimal('2.2400000000'))
('William Gomez', 'forget', Decimal('2.2333332500'))
('Holly West', 'pattern', Decimal('4.0300000000'))
('Jennifer Gibson', 'special', Decimal('2.8800000000'))
('Hector Kirby', 'model', Decimal('1.2000000000'))
('Stacey Evans', 'religious', Decimal('3.9850000000'))
('Dana Davis', 'rule', Decimal('3.4500000000'))
('Jeremy Bernard', 'sport', Decimal('3.7875000000'))
```

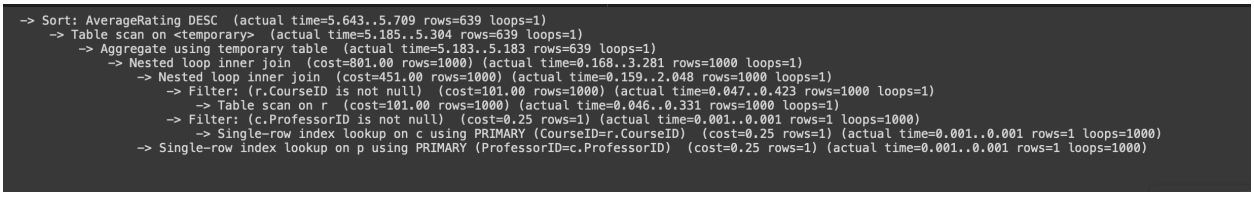
Query 1 Indexing:

```
**Query 1**: Courses with the highest average rating, including


SELECT c.CourseID, c.Title, p.Name AS ProfessorName, p.Departmer
FROM Courses c
JOIN Professors p ON c.ProfessorID = p.ProfessorID
JOIN Ratings r ON c.CourseID = r.CourseID
```

```
GROUP BY c.CourseID, c.Title, p.Name, p.Department
ORDER BY AverageRating DESC;
```

EXPLAIN ANALYZE OUTPUT With Default Index



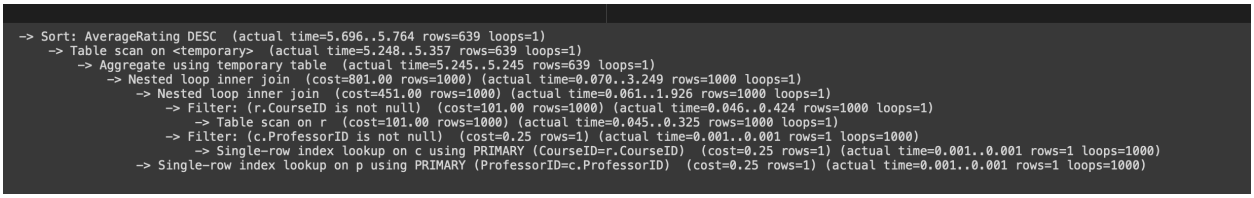
We can see that the cost is pretty low and the time to run as well. However we can attempt to index.

Indexing method one: Index on Courses.ProfessorID

This query joins the Courses table with the Professors table based on the ProfessorID column. An index on Courses.ProfessorID can speed up this join operation by quickly locating all courses taught by each professor. This is particularly beneficial because the join condition directly references Courses.ProfessorID, making lookups based on this column frequent and critical for the join performance.

```
CREATE INDEX index_courses_professorid ON Courses(ProfessorID);
```

EXPLAIN ANALYZE OUTPUT Index 1



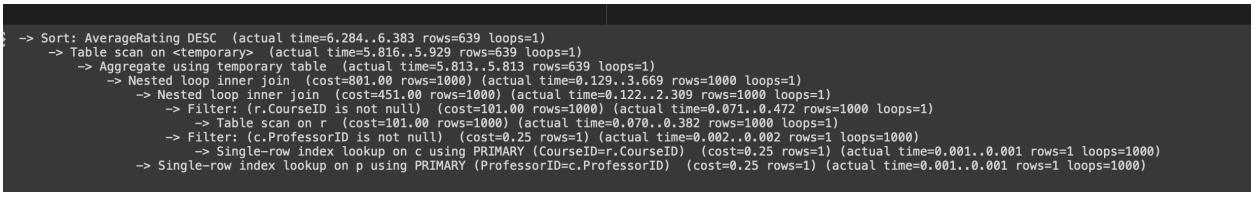
There doesn’t seem to be any change and there is no index lookup using the index so the index doesn’t seem to help at all.

Indexing method two: Index on Ratings.CourseID

The query also involves a join between the Courses and Ratings tables on the CourseID column, followed by an aggregation operation to calculate the average rating. An index on Ratings.CourseID will let there be faster matching of ratings to courses, which is essential for both the join operation and the subsequent aggregation. Since this column is used for a key operation, an index here can greatly reduce the amount of data the database needs to scan, improving query performance.

```
CREATE INDEX index_ratings_courseid ON Ratings(CourseID);
```

EXPLAIN ANALYZE OUTPUT Index 2



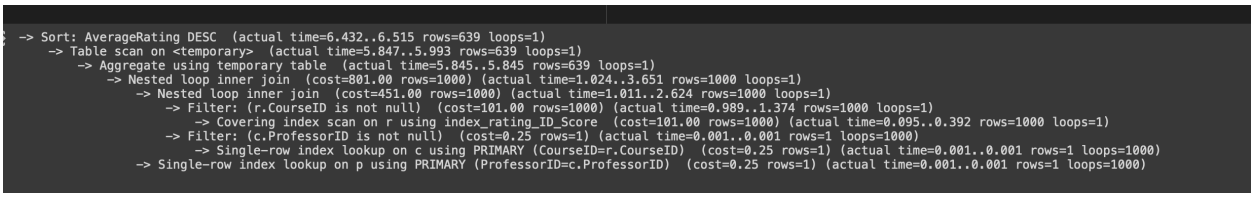
The cost stays the same and time changes a bit but this indexing also doesn’t change anything. The index still isn't being reached.

Indexing method three: Index on Ratings.CourseID

An index on both CourseID and Score in the Ratings table could further optimize the performance of the average function used on Score in the query. This index supports the join operation by CourseID and could optimizes the calculation of the average score. The database might be able to use the index to directly compute the average without having to sort or scan the entire Score column for each course.

```
CREATE INDEX index_rating_ID_Score ON Ratings(CourseID, Scor
e);
```

EXPLAIN ANALYZE OUTPUT Index 3



Here we can see that there is an index scan on index_rating_ID_Score and that there is no effect on the cost at all. This is likely because there are only 1000 rows in the data and there isn't a big change that indexing makes.

Result:

We will choose indexing on Ratings(CourseID, Score) since we can see that the index is being scanned in the Explain Analyze output. However there really isn't much of a change overall.

Query 2 Indexing:

```
'''**Query 2**: Determine the percentage of users who would take
'''
SELECT c.CourseID, c.Title,
    COALESCE(
    (SELECT ROUND(SUM(CASE WHEN r.WouldTakeAgain THEN 1 ELSE 0 E
    FROM Ratings r
    WHERE r.CourseID = c.CourseID
    GROUP BY r.CourseID),
    0) AS PercentageWouldTakeAgain
FROM Courses c
LEFT JOIN Ratings r ON c.CourseID = r.CourseID
GROUP BY c.CourseID, c.Title
ORDER BY PercentageWouldTakeAgain DESC;
```

EXPLAIN ANALYZE OUTPUT With Default Index

```
-> Sort: PercentageWouldTakeAgain DESC  (actual time=486.031..486.108 rows=1000 loops=1)
    -> Table scan on <temporary>  (cost=200005.04..212507.52 rows=1000000) (actual time=485.538..485.686 rows=1000 loops=1)
        -> Temporary table with deduplication  (cost=200005.02..200005.02 rows=1000000) (actual time=485.536..485.536 rows=1000 loops=1)
            -> Left hash join (r.CourseID = c.CourseID)  (cost=100005.02 rows=1000000) (actual time=0.482..1.798 rows=1361 loops=1)
                -> Table scan on c  (cost=101.50 rows=1000) (actual time=0.035..0.687 rows=1000 loops=1)
                -> Hash
                    -> Table scan on r  (cost=0.10 rows=1000) (actual time=0.044..0.270 rows=1000 loops=1)
-> Select #2 (subquery in projection; dependent)
    -> Table scan on <temporary>  (actual time=0.352..0.352 rows=1 loops=1361)
        -> Aggregate using temporary table  (actual time=0.352..0.352 rows=1 loops=1361)
            -> Filter: (r.CourseID = c.CourseID)  (cost=11.00 rows=100) (actual time=0.194..0.350 rows=1 loops=1361)
                -> Table scan on r  (cost=11.00 rows=1000) (actual time=0.019..0.276 rows=1000 loops=1361)
```

We notice that the cost seems very extreme so we could benefit from indexing.

Indexing method one: Index on Ratings.CourseID

The subquery and the main query's join condition both filter ratings based on courseID. An index on ratings.CourseID would make these operations more efficient by allowing the database engine to quickly locate all ratings for each course. This is particularly beneficial because the where and on clauses directly reference ratings.courseID which means that lookups based on this column are frequent and critical for performance.

```
CREATE INDEX index_ratings_courseID ON Ratings(CourseID);
```

EXPLAIN ANALYZE OUTPUT Index 1

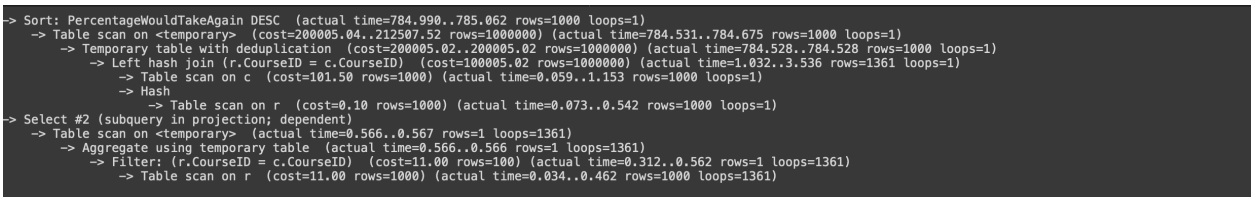
```
-> Sort: PercentageWouldTakeAgain DESC  (actual time=10.658..10.725 rows=1000 loops=1)
    -> Table scan on <temporary>  (cost=664.66..686.69 rows=1565) (actual time=10.187..10.336 rows=1000 loops=1)
        -> Temporary table with deduplication  (cost=664.64..664.64 rows=1565) (actual time=10.185..10.185 rows=1000 loops=1)
            -> Nested loop left join  (cost=508.15 rows=1565) (actual time=0.071..2.614 rows=1361 loops=1)
                -> Table scan on c  (cost=101.50 rows=1000) (actual time=0.056..0.364 rows=1000 loops=1)
                -> Covering index lookup on r using index_ratings_courseid (CourseID=c.CourseID)  (cost=0.25 rows=2) (actual time=0.002..0.002 rows=1 loops=1000)
-> Select #2 (subquery in projection; dependent)
    -> Group aggregate: count(r.WouldTakeAgain), sum((case when r.WouldTakeAgain then 1 else 0 end))  (cost=0.70 rows=2) (actual time=0.004..0.004 rows=1 loops=1361)
        -> Index lookup on r using index_ratings_courseid (CourseID=c.CourseID)  (cost=0.55 rows=2) (actual time=0.003..0.004 rows=1 loops=1361)
```

We can see that there is a huge cost decrease once indexing on Ratings Course ID.

Indexing method two: Index on Course.CourseID
Although courses might be a smaller table compared to ratings, an index on courses.courseID could possibly still enhance the join operation. Since the query involves a left join based on courseID
and there is a group by on Course ID, the database can benefit from an index on this column in the courses table to quickly map each course to its ratings. This index could help in speeding up the access to course information during the join operation.

```
CREATE INDEX index_courses_courseID ON Courses(CourseID);
```

EXPLAIN ANALYZE OUTPUT Index 2



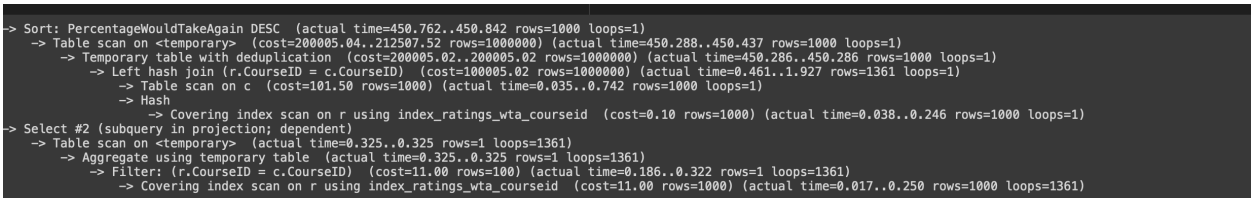
This definitely did not make it better so this indexing method is not good and should not be used.

Indexing method three: Index on Rating(CourseID, WouldTakeAgain)

A composite index that includes both courseID and would take again could further optimize the subquery's performance. This is because the subquery not only filters by courseID but also needs to access the wouldtakeagain attribute to calculate the percentages. With a composite index, the database could efficiently filter and calculate the required information without having to fetch the entire row.

```
CREATE INDEX index_ratings_wta_courseid ON Ratings(WouldTakeAga
```

EXPLAIN ANALYZE OUTPUT Index 3



This also did not have a good performance.

Result:

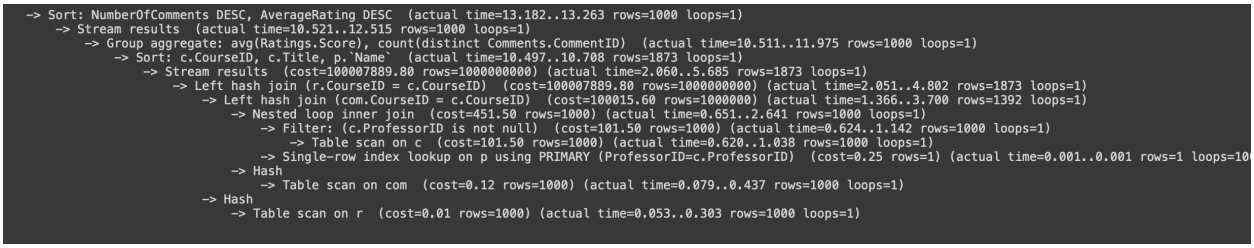
After all three indexes, we see that only indexing on Ratings(CourseID) would be very helpful in decreasing cost as explained by the Explain analyze output.

Query 3 Indexing

```
'''Query 3: Retrieve popular courses, defined by the number o
f comments and average rating. Include courses with no commen
ts or ratings.'''

SELECT c.CourseID, c.Title, p.Name AS ProfessorName,
       COUNT(DISTINCT com.CommentID) AS NumberOfComments,
       COALESCE(AVG(r.Score), 00) AS AverageRating
FROM Courses c
LEFT JOIN Comments com ON c.CourseID = com.CourseID
LEFT JOIN Ratings r ON c.CourseID = r.CourseID
JOIN Professors p ON c.ProfessorID = p.ProfessorID
GROUP BY c.CourseID, c.Title, p.Name
ORDER BY NumberOfComments DESC, AverageRating DESC;
```

EXPLAIN ANALYZE OUTPUT With Default Index



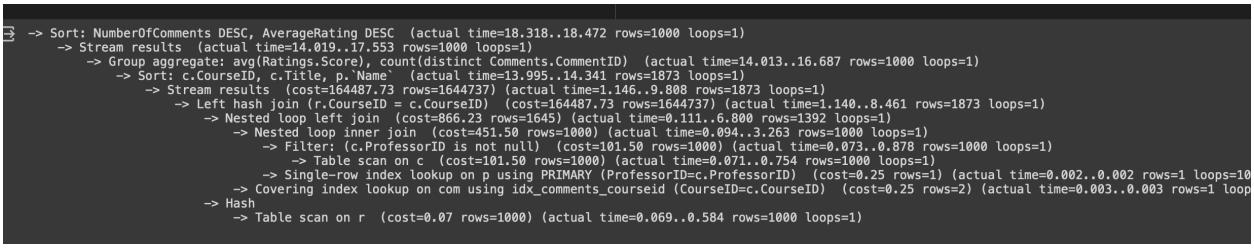
We notice that there are a lot of rows being read and the cost is pretty high because of that.

Indexing method one: Index on Comments.ProfessorID and Comments.CommentID

Creating a composite index on CourseID and CommentsID in the Comments table could significantly enhance the performance of the given SQL query. Firstly, it optimizes the join operation between the Courses and Comments tables on CourseID, ensuring that the database engine can quickly find all comments related to each course. Secondly, including CommentsID in the index further eases the process of counting distinct comment IDs for each course, as it allows the database engine to efficiently traverse the index to count unique comment IDs without needing to fetch and inspect the full rows from the Comments table.

```
CREATE INDEX index_comments_courseid ON Comments(CourseID, Comme
```

EXPLAIN ANALYZE OUTPUT Index 1



The time ends up increasing but we can see that the cost decreases with the stream results. The number of rows read are much less. So this is a pretty good index.

Indexing method two: Index on Ratings.CourseID and Ratings.Score

This query involves a join with the ratings table and computes the average score for each course. A composite index on course and score can quicken these operations. The courseID is used for the join, and scores is involved in calculating the average of scores. While courseID helps in quickly locating the ratings relevant to each course, including score in the index might help the database engine more efficiently calculate the average.

```
CREATE INDEX index_ratings_courseid_score ON Ratings(CourseI
D, Score);
```

EXPLAIN ANALYZE OUTPUT Index 2

```
-> Sort: NumberOfComments DESC, AverageRating DESC  (actual time=12.546..12.628 rows=1000 loops=1)
  -> Stream results  (actual time=9.917..12.064 rows=1000 loops=1)
    -> Group aggregate: avg(Ratings.Score), count(distinct Comments.CommentID)  (actual time=9.911..11.491 rows=1000 loops=1)
      -> Sort: c.CourseID, c.Title, p.`Name`  (actual time=9.807..10.030 rows=1873 loops=1)
        -> Stream results  (cost=882069.70 rows=1564945) (actual time=1.173..7.826 rows=1873 loops=1)
          -> Nested loop left join  (cost=882069.70 rows=1564945) (actual time=1.166..6.926 rows=1873 loops=1)
            -> Left hash join (com.CourseID = c.CourseID)  (cost=100015.60 rows=1000000) (actual time=1.150..3.435 rows=1392 loops=1)
              -> Nested loop inner join  (cost=451.50 rows=1000) (actual time=0.051..2.022 rows=1000 loops=1)
                -> Filter: (c.ProfessorID is not null)  (cost=101.50 rows=1000) (actual time=0.038..0.468 rows=1000 loops=1)
                  -> Table scan on c  (cost=101.50 rows=1000) (actual time=0.036..0.368 rows=1000 loops=1)
                -> Single-row index lookup on p using PRIMARY (ProfessorID=c.ProfessorID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
              -> Hash
                -> Table scan on com  (cost=0.12 rows=1000) (actual time=0.061..0.629 rows=1000 loops=1)
            -> Covering index lookup on r using index_ratings_courseid_score (CourseID=c.CourseID)  (cost=0.63 rows=2) (actual time=0.002..0.002 rows=1 loops=1392)
```

We can see that cost does decrease but not as much as the previous index. However we could still use this index and have multiple indexes.

Indexing method three: Index on Courses.ProfessorID

This query joins the Courses table with the Professors table using ProfessorID. An index on Courses.ProfessorID will make this join operation faster by reducing the time needed to find matching rows in the ProfessorID table. This is particularly beneficial because this join is essential for retrieving the ProfessorID for each course, which is a part of the query's SELECT clause.

```
CREATE INDEX index_courses_professorID ON Ratings(CourseID, Sco
```

EXPLAIN ANALYZE OUTPUT Index 3

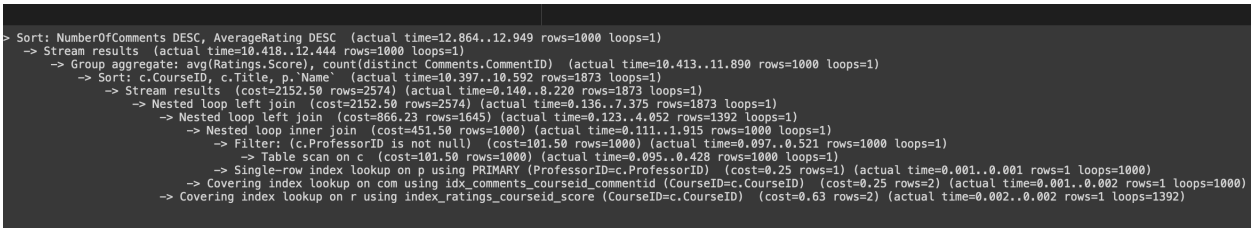
```
-> Sort: NumberOfComments DESC, AverageRating DESC  (actual time=9.262..9.397 rows=1000 loops=1)
  -> Stream results  (actual time=6.588..8.791 rows=1000 loops=1)
    -> Group aggregate: avg(Ratings.Score), count(distinct Comments.CommentID)  (actual time=6.583..8.222 rows=1000 loops=1)
      -> Sort: c.CourseID, c.Title, p.`Name`  (actual time=6.567..6.811 rows=1873 loops=1)
        -> Stream results  (cost=100007889.80 rows=1000000000) (actual time=1.536..4.773 rows=1873 loops=1)
          -> Left hash join (r.CourseID = c.CourseID)  (cost=100007889.80 rows=1000000000) (actual time=1.529..3.956 rows=1873 loops=1)
            -> Left hash join (com.CourseID = c.CourseID)  (cost=100015.60 rows=1000000) (actual time=0.813..2.893 rows=1392 loops=1)
              -> Nested loop inner join  (cost=451.50 rows=1000) (actual time=0.203..1.988 rows=1000 loops=1)
                -> Filter: (c.ProfessorID is not null)  (cost=101.50 rows=1000) (actual time=0.187..0.614 rows=1000 loops=1)
                  -> Table scan on c  (cost=101.50 rows=1000) (actual time=0.186..0.527 rows=1000 loops=1)
                -> Single-row index lookup on p using PRIMARY (ProfessorID=c.ProfessorID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
              -> Hash
                -> Table scan on com  (cost=0.12 rows=1000) (actual time=0.028..0.258 rows=1000 loops=1)
            -> Hash
              -> Table scan on r  (cost=0.01 rows=1000) (actual time=0.046..0.318 rows=1000 loops=1)
```

We can see here that the index isn’t being scanned so there is no change in cost and this index does not help at all.

Result:

By combining and making two indexes, one for Comments.ProfessorID and Comments.CommentID and one for Ratings.CourseID and Ratings.Score, we

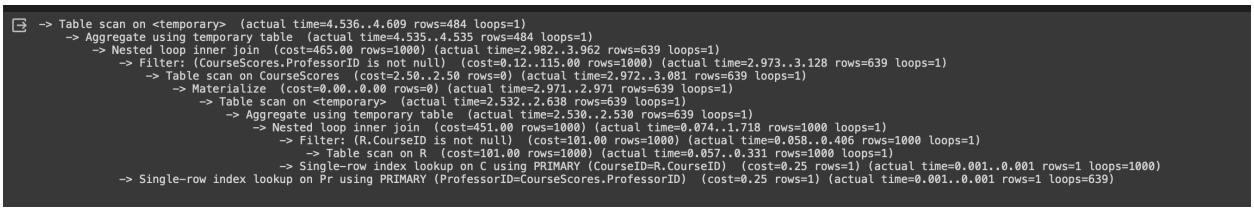
actually decrease cost by a lot as seen below in the Explain Analyze while keeping both indexes.. So the plan is to have two indexes one on the Comments table and one on the ratings table.



Query 4 Indexing

````
**Query 4**: Average GPA of all courses taught by a specific Pr
```sql
SELECT
    Pr.Name AS ProfessorName,
    Pr.Department,
    AVG(OverallCourseScore) AS AvgProfessorScore
FROM
    (
        SELECT
            C.ProfessorID,
            C.Title AS CourseTitle,
            AVG(R.Score) AS OverallCourseScore
        FROM Courses C
        JOIN Ratings R ON C.CourseID = R.CourseID
        GROUP BY C.CourseID
    ) AS CourseScores
JOIN Professors Pr ON CourseScores.ProfessorID = Pr.ProfessorID
GROUP BY Pr.ProfessorID
```
````

EXPLAIN ANALYZE OUTPUT With Default Index

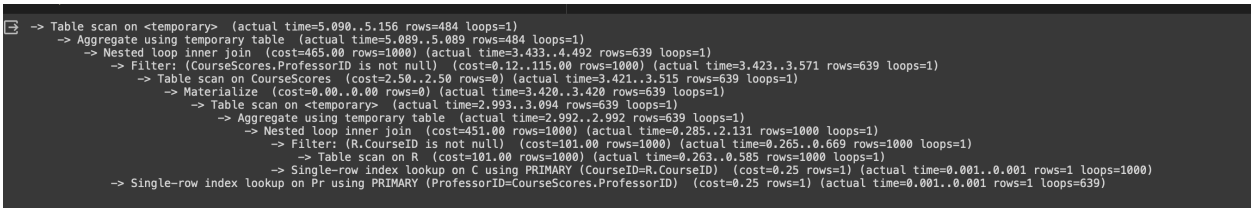


Indexing method one: Index on Ratings.CourseID

This query joins the Courses table with the Ratingstable on Courses.CourseID. An index on Ratings.CourseID will make this join operation more efficient by quickly locating entries in the Ratings table for corresponding entries in the Courses table. Since courseID is used in a join condition, this index could help in reducing the lookup time significantly.

```
CREATE INDEX index_ratings_courseid ON Ratings(CourseID);
```

EXPLAIN ANALYZE OUTPUT With Index 1



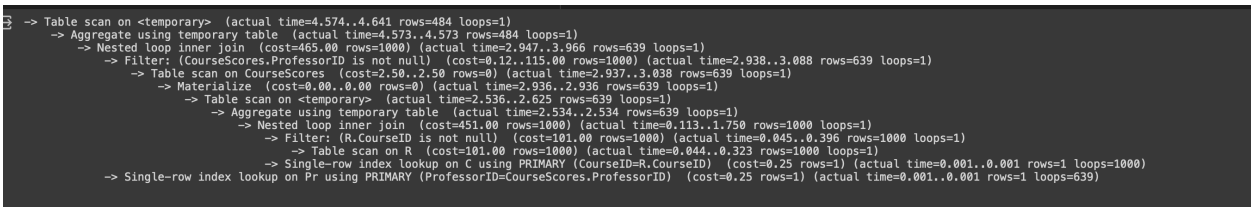
Here we can see that there is no effect on the cost with this index and it is not being used at all.

Indexing method two: Index on Ratings.Score

This query uses Ratings.Score to calculate the average overall course score for all courses taught by one professor. Because there is an aggregation on this score an index on Ratings.Score could potentially be helpful in de lookup time.

```
CREATE INDEX index_ratings_score ON Ratings(score);
```

EXPLAIN ANALYZE OUTPUT With Index 2



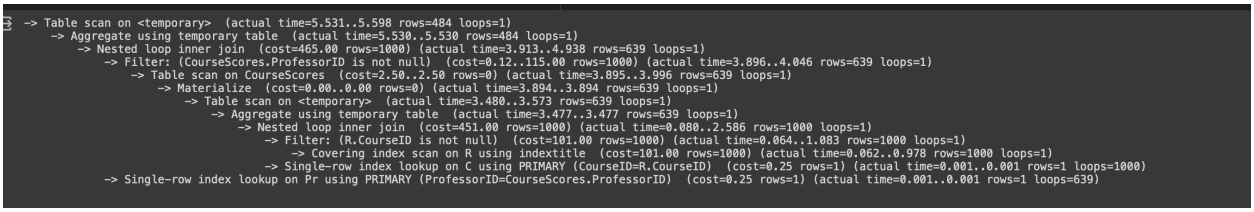
Here there is also no effect whatsoever once there is indexing. The index is not even getting scanned.

Indexing method three: Index on Ratings.Score and Ratings.CourseID

Here we decided to index compositely on both Score and CourseID. This is because of the AVG being used on Score and the CourseID being used to join. We think that indexing on these two together could help costs go down.

```
CREATE INDEX index_ratings_score_courseid ON Ratings(score,Cours
```

EXPLAIN ANALYZE OUTPUT With Index 3



What we found was that there was still no effect on costs or time with this composite index.

Result: No indexing strategy really works for this query and helps cut down. This could likely be because many of the possible indexes are already primary keys and thus default indices for this query.