

Stage Three: Database Implementation and Indexing

Database Implementation

The relation schema from the previous stage is shown below. These were the tables we implemented.

1. Course(courseID: varchar(8) [PK], title: varchar(255), deptID: int [FK to Department.deptID])
2. Student(UIN: int [PK], password: varchar(255), firstName: varchar(255), lastName: varchar(255), deptID: int [FK to Department.deptID], standing: VARCHAR(255))
3. Enrollment(studentUIN: int [PK, FK to Student.UIN], sectionCourseID: varchar(8) [PK, FK to Course.courseID], sectionProf: int [PK, FK to Professor.profID], sectionTerm: varchar(8) [PK], credits: int)
4. Section(sectionCourseID: varchar(8) [PK, FK to Course.courseID], sectionProf: int [PK, FK to Professor.profID], sectionTerm: varchar(8) [PK], buildingTaught: varchar(255), roomNum: int)
5. Professor(profID: int [PK], firstName: varchar(255), lastName: varchar(255), deptID: int [FK to Department.deptID])
6. Department(deptID: int [PK], deptName: varchar(255), mainBuilding: varchar(255))
7. Review(reviewID: int [PK], reviewerUIN: int [FK to Student.UIN], courseID: varchar(10) [FK to Course.courseID], profID: int [FK to Professor.profID], reviewStanding: string, courseTerm: varchar(4), profComments: varchar(5000), courseComments: varchar(5000), reviewerGrade: varchar(2), difficulty: int, usefulness: int, avgHrs: real, maxHrs: real, profRating: int)

The DDL commands for each of these is shown below. Since we created a database from scratch we did not worry about having to delete an existing table with the same name. Regardless, we executed a *drop table if exists <tableName>* command for each table before creating them.

- **Create Table** Course(
 - courseID varchar(20),
 - title varchar(255),
 - deptID int,
 - Primary key (courseID),
 - Foreign key (deptID) references Department(deptID) on delete set Null on update cascade
);

- **Create table** Student(
 - UIN int,
 - password varchar(255),
 - firstName varchar(255),
 - lastName varchar(255),
 - standing varchar(20),
 - deptID int,
 - Primary key (UIN),
 - Foreign key (deptID) references Department(deptID) on delete set Null on update cascade
);

- **Create table** Enrollment(
 - studentUIN int,
 - sectionCourseID varchar(20),
 - sectionProfID int,
 - sectionTerm varchar(20),
 - Credits int,
 - Primary key (studentUIN, sectionCourseID, sectionProfID, sectionTerm),
 - Foreign key (studentUIN) references Student(UIN) on delete cascade on update cascade,
 - Foreign key (sectionCourseID, sectionProfID, sectionTerm) references Section(sectionCourseID, sectionProfID, sectionTerm) on delete cascade on update cascade
);

- **Create table** Section(
 - sectionCourseID varchar(20),
 - sectionProfID int,
 - sectionTerm varchar(20),
 - buildingTaught varchar(255),
 - roomNum int,
 - Primary key (sectionCourseID, sectionProfID, sectionTerm),
 - Foreign key (sectionCourseID) references Course(courseID) on delete cascade on update cascade,
 - Foreign key (sectionProfID) references Professor(profID) on delete cascade on update cascade
);

- **Create table** Professor(
 - profID int,
 - firstName varchar(255),
 - lastName varchar(255),
 - deptID int,
 - Primary key (profID),
 - Foreign key (deptID) references Department(deptID) on delete set Null on update cascade
);

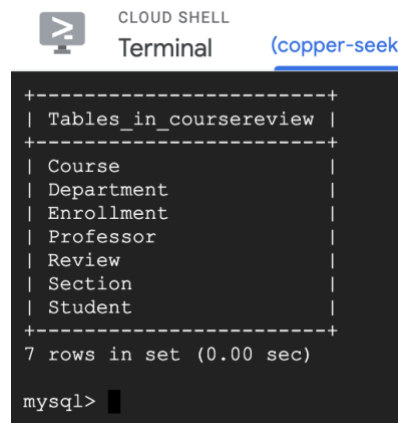
```

- Create table Department(
    deptID int,
    deptName varchar(255),
    mainBuilding varchar(255),
    Primary key (deptID)
);

- Create table Review(
    reviewID int,
    reviewerUIN int,
    courseID varchar(20),
    profID int,
    reviewStanding varchar(20),
    courseTerm varchar(20),
    profComments varchar(5000),
    courseComments varchar(5000),
    reviewerGrade varchar(10),
    difficulty int,
    usefulness int,
    avgHrs real,
    maxHrs real,
    profRating real,
    Primary Key (reviewID),
    Foreign key (reviewerUIN) references Student(UIN) on delete set null on update cascade,
    Foreign key (courseID) references Course(courseID) on delete set null on update cascade,
    Foreign key (profID) references Professor(profID) on delete set null on update cascade
);

```

We populated the tables with webscraped review data from RateMyProfessor and course data. In addition to some randomly generated data to fill in additional review components, we generated the Review, Student and Professor data. Department data was gathered directly from UIUC website. The Section and Enrollment data was also generated from the review information. Since we combined scraped data with generated data, there is currently no accurate association between courses and departments. For example, the course CS411 may be associated with the anthropology department. The database (called 'coursereview') is hosted on GCP as shown by the screenshot below. We also provide the count for each table.



```

CLOUD SHELL
Terminal (copper-seek)

+-----+
| Tables_in_coursereview |
+-----+
| Course                  |
| Department              |
| Enrollment              |
| Professor               |
| Review                 |
| Section                |
| Student                |
+-----+
7 rows in set (0.00 sec)

mysql>

```

```
mysql> select count(*) from Course;
+-----+
| count(*) |
+-----+
|      4368 |
+-----+
1 row in set (0.01 sec)
```

```
mysql> select count(*) from Student;
+-----+
| count(*) |
+-----+
|      5208 |
+-----+
1 row in set (0.26 sec)
```

```
mysql> select count(*) from Enrollment;
+-----+
| count(*) |
+-----+
|    259062 |
+-----+
1 row in set (1.72 sec)
```

```
mysql> select count(*) from Section;
+-----+
| count(*) |
+-----+
|     12966 |
+-----+
1 row in set (0.01 sec)
```

```
mysql> select count(*) from Professor;
+-----+
| count(*) |
+-----+
|      4188 |
+-----+
1 row in set (0.06 sec)
```

```
mysql> select count(*) from Department;
+-----+
| count(*) |
+-----+
|       151 |
+-----+
1 row in set (0.05 sec)
```

```
mysql> select count(*) from Review;
+-----+
| count(*) |
+-----+
|     17564 |
+-----+
1 row in set (0.80 sec)
```

Advanced Queries

For the first query, we wanted to extract information regarding the standing of the students who wrote reviews for courses taught by professors with high average ratings. This can be useful to see if there is any correlation between the student standing and the review they write. It can also be a good indication of what students these professors usually teach.

The query is as follows:

```
SELECT reviewStanding, COUNT(reviewStanding) FROM Review WHERE profID IN
(SELECT profID FROM Professor p JOIN Review r USING(profID) WHERE p.deptID = 104
GROUP BY profID HAVING profRating > avg(profRating))
GROUP BY reviewStanding;
```

We use a subquery to extract all professors in the department with ID=104 who have a rating that is higher than the average. We then aggregate the number of reviews for these professors grouping by reviewStanding. Since there are only 4 standings, this returns 4 tuples only as shown below.

► Freshman	121
Junior	131
Senior	92
Sophomore	109

For the second query, we wanted to find the average avgHrs and average maxHrs for grouped by department. Users can use this to gauge how time consuming an average class in that department would be. We order this in descending order based on the average hours.

```
select deptID, avg(avgHrs) as avgAvgHrs, avg(maxHrs) as avgMaxHrs
from Department natural join Course natural join Review
group by deptID
order by avgAvgHrs limit 15;
```

The top 15 results for this query are shown below

deptID	avgAvgHrs	avgMaxHrs
144	3.130434782608696	7.1521739130434785
150	3.3157894736842106	6.578947368421052
113	3.3544303797468356	6.987341772151899
192	3.388888888888889	7.222222222222222
145	3.393939393939394	7.03030303030303
211	3.3972602739726026	6.863013698630137
137	3.42	6.88
155	3.4408602150537635	7.172043010752688
207	3.4461538461538463	7.092307692307692
248	3.4857142857142858	7.257142857142857
112	3.488372093023256	6.837209302325581
247	3.4956521739130433	7.017391304347826
114	3.5	7.055555555555555
239	3.5128205128205128	6.846153846153846
231	3.52073732718894	7.013824884792626

Indexing Analysis

For each query we first tested the performance without any indexes.

Query one:

```
EXPLAIN:
-> Table scan on <temporary> (actual time=0.002..0.002 rows=4 loops=1)
-> Aggregate using temporary table (actual time=20309.334..20309.335 rows=4 loops=1)
-> Filter: <in_optimizer>(Review.profilID,<exists>(select #2)) (cost=1791.45 rows=16872) (actual time=29.594..20306.943 rows=453 loops=1)
-> Table scan on Review (cost=1791.45 rows=16872) (actual time=0.029..14.010 rows=17564 loops=1)
```

Query two (total time 50.202ms)

```
EXPLAIN:
-> Sort: avgAvgHrs (actual time=50.188..50.202 rows=151 loops=1)
-> Table scan on <temporary> (actual time=0.001..0.009 rows=151 loops=1)
-> Aggregate using temporary table (actual time=50.111..50.133 rows=151 loops=1)
-> Nested loop inner join (cost=13601.85 rows=16872) (actual time=0.044..44.155 rows=17564 loops=1)
```

To improve query performance, we tried adding indexes to relevant columns of the tables involved in the query. The foreign and primary keys all have default indexes so joins could not be altered. Instead we looked for other columns that are being searched or returned.

For query one, we had multiple options for indexing since our query was utilizing multiple different attributes from both the Professor and Review table. The inner query could not be improved since it only references a foreign key in the Professor department which already has a default index.

However, we tried adding indexes for the reviewStanding and profRating attributes in the Review table. First we tried just the reviewStanding index. We see below that it used the index and the total time for the query was reduced slightly. The structure of the query has also changed accordingly. This makes sense we are counting on the reviewStanding which is likely more efficient when the entries ordered based on reviewStanding rather than a different attribute.

```
EXPLAIN: -> Group aggregate: count(Review.reviewStanding) (actual time=4840.050..19876.669 rows=4 loops=1)
          -> Filter: <in_optimizer>(Review.proflD,<exists>(select #2)) (cost=1791.45 rows=16872) (actual time=3.001..19875.543 rows=453 loops=1)
          -> Index scan on Review using idx_reviewStanding (cost=1791.45 rows=16872) (actual time=0.029..26.874 rows=17564 loops=1)
          -> Select #2 (subquery in condition; dependent)
```

We also tried just using an index on profRating. We see that the query structure is the same as with the original query but the time has been reduced somewhat. This makes sense since ordering the profRating makes it easier to determine when the aggregation threshold is surpassed.

```
EXPLAIN: -> Table scan on <temporary> (actual time=0.002..0.003 rows=4 loops=1)
          -> Aggregate using temporary table (actual time=18862.107..18862.108 rows=4 loops=1)
          -> Filter: <in_optimizer>(Review.proflD,<exists>(select #2)) (cost=1791.45 rows=16872) (actual time=27.401..18859.891 rows=453 loops=1)
          -> Table scan on Review (cost=1791.45 rows=16872) (actual time=0.038..13.127 rows=17564 loops=1)
```

Lastly, we tried combining both indexes. This resulted in the same structure as with just the reviewStanding index which makes sense. The total time is reduced again slightly which also makes sense since both indexes can be used to reduce the total query time.

```
EXPLAIN: -> Group aggregate: count(Review.reviewStanding) (actual time=4557.196..18419.362 rows=4 loops=1)
          -> Filter: <in_optimizer>(Review.proflD,<exists>(select #2)) (cost=1791.45 rows=16872) (actual time=2.538..18418.330 rows=453 loops=1)
          -> Index scan on Review using idx_reviewStanding (cost=1791.45 rows=16872) (actual time=0.027..24.995 rows=17564 loops=1)
          -> Select #2 (subquery in condition; dependent)
```

However, all these times are quite similar regardless of the indexes which suggests that they do not contribute a substantial change to query performance. This is clear since the number of entries that need to be explored does not change much as shown by the rows and loops variables in the analyze results.

For query two, there are more limited options since the query is simpler. DeptID already has an index in Course since it is a foreign key. However, avgHrs and maxHrs in the Review table did not yet have an index.

We first added an index for maxHrs which minimally decreased performance:

```
EXPLAIN: -> Sort: avgAvgHrs (actual time=50.820..50.834 rows=151 loops=1)
          -> Table scan on <temporary> (actual time=0.001..0.010 rows=151 loops=1)
          -> Aggregate using temporary table (actual time=50.737..50.760 rows=151 loops=1)
          -> Nested loop inner join (cost=13601.85 rows=16872) (actual time=0.047..44.634 rows=17564 loops=1)
```

We also tried only having an index for avgHrs which we thought would result in better performance since we are grouping by this variable. However, we saw another minimal decrease in performance:

```
EXPLAIN:      -> Sort: avgAvgHrs (actual time=51.162..51.194 rows=151 loops=1)
              -> Table scan on <temporary> (actual time=0.001..0.009 rows=151 loops=1)
              -> Aggregate using temporary table (actual time=51.071..51.104 rows=151 loops=1)
              -> Nested loop inner join (cost=13601.85 rows=16872) (actual time=0.047..44.776 rows=17564 loops=1)
```

Lastly, we tried a combination with both of the above indexes. This resulted in a minimal improvement in performance compared to the original query (total time 49.795ms)

```
EXPLAIN:      -> Sort: avgAvgHrs (actual time=49.781..49.795 rows=151 loops=1)
              -> Table scan on <temporary> (actual time=0.001..0.010 rows=151 loops=1)
              -> Aggregate using temporary table (actual time=49.704..49.726 rows=151 loops=1)
              -> Nested loop inner join (cost=13601.85 rows=16872) (actual time=0.049..43.447 rows=17564 loops=1)
```

We can see that in each case the structure of the table is not changed which makes sense since the logic of the join, aggregation, table scan, and finally sorting do not change. The minimal time improvement occurs at the aggregation level which makes sense since we added the index for both the variables that are being aggregated. It is possible that with just a single index the other aggregated attribute which is not indexed is the bottleneck and the added index simply adds some overhead since the system needs to decide between more index options. Adding both indexes may limit this bottleneck which allows a minimal improvement in performance. This difference is also proportionally minimal so this result may simply be due to random variations since there are minimal differences in timing for different iterations of the same exact query with the same indexes. We did not try to add indexes to other attributes since no other attributes were related to this query that did not already have default indexes.