Stage 3

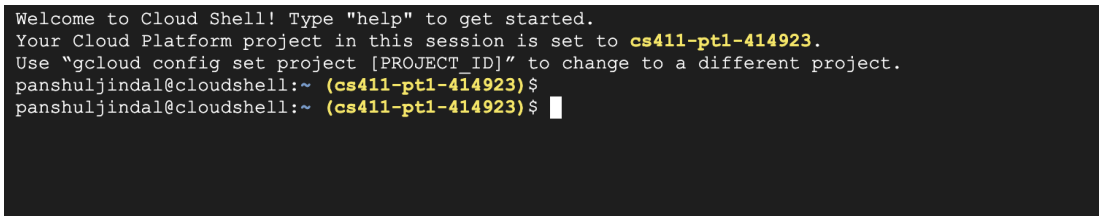
Part 1:

1. The following are our main tables:

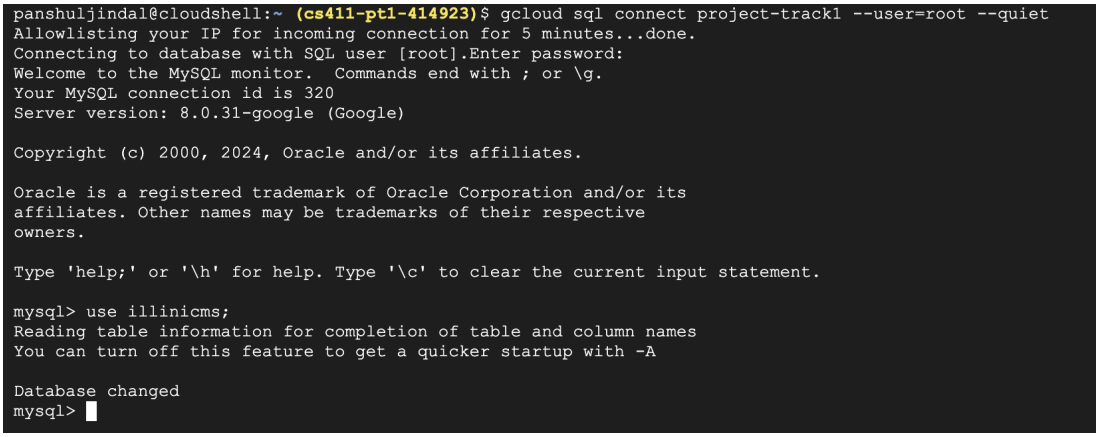
```
mysql> show tables
    -> ;
+----------------------+
| Tables_in_illinicms  |
+----------------------+
| Assignment           |
| Attendance           |
| ClassGroupRecordings |
| ClassroomGroups      |
| ClassroomUsers       |
| Classrooms           |
| Courses              |
| Grades               |
| Users                |
+----------------------+
9 rows in set (0.00 sec)
```

2. We have uploaded our Database Design pdf to our GitHub repo

3. A. Here is the screenshot of our connection:
   - **a.** Terminal before connection



   - **b.** Terminal after connection



B. And the proof of 1000 rows in our tables:

```
mysql> SELECT table_name, table_rows FROM information_schema.tables WHERE table_schema = 'illin
icms';
+----------------------+------------+
| TABLE_NAME           | TABLE_ROWS |
+----------------------+------------+
| Assignment           |        300 |
| Attendance           |       5918 |
| ClassGroupRecordings |          0 |
| ClassroomGroups      |       1144 |
| ClassroomUsers       |       4970 |
| Classrooms           |        144 |
| Courses              |         40 |
| Grades               |       1720 |
| Users                |       1600 |
+----------------------+------------+
9 rows in set (0.00 sec)
```

4. The following are our 4 advanced SQL queries for our application and their respective output screenshots:

a. Query1 - Parent Dashboard

We want parents to know where their child stands in the class. For this, we write a query to select the subjectName, className, assignmentId, and grade for that particular student. We also display the maximumGrade achievable and toppers grade for that particular assignment.

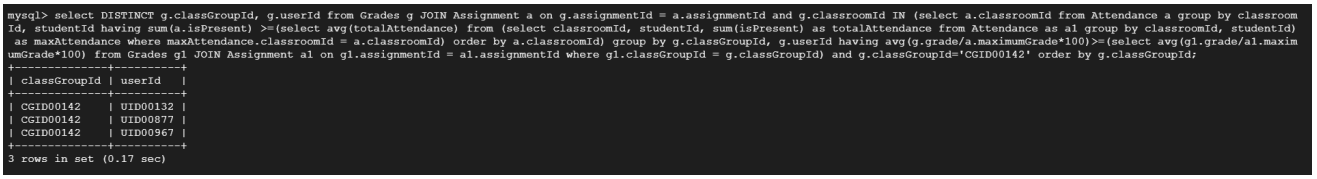
select co.subjectName, cr.className,cg.classGroupId, a.assignmentId, g.grade, maxGrade.topperGrade, a.maximumGrade as maximumPossibleGrade from ClassroomUsers cu NATURAL JOIN ClassroomGroups cg NATURAL JOIN Courses co NATURAL JOIN Classrooms cr NATURAL JOIN Assignment a NATURAL JOIN Grades g JOIN (select assignmentId, max(grade) as topperGrade from Grades group by assignmentId) as maxGrade ON g.assignmentId = maxGrade.assignmentId where cu.userId='UID00020';

```
mysql> select co.subjectName, cr.className,cg.classGroupId, a.assignmentId, g.grade, maxGrade.topperGrade, a.maximumGrade as maximumPossibleGrade from ClassroomUsers cu NATURAL JOIN Classroo
mGroups cg NATURAL JOIN Courses co NATURAL JOIN Classrooms cr NATURAL JOIN Assignment a NATURAL JOIN Grades g JOIN (select assignmentId, max(grade) as topperGrade from Grades group by assign
mentId) as maxGrade ON g.assignmentId = maxGrade.assignmentId where cu.userId='UID00020' LIMIT 15;
+---------------+-----------+--------------+--------------+-------+------------+---------------------+
| subjectName   | className | classGroupId | assignmentId | grade | topperGrade | maximumPossibleGrade |
+---------------+-----------+--------------+--------------+-------+------------+---------------------+
| Mathematics   | 1-A       | CGID00000    | AID00193     |    95 |         95 |                 100 |
| Science       | 1-A       | CGID00001    | AID00005     |    80 |         80 |                 100 |
| Science       | 1-A       | CGID00001    | AID00175     |    56 |         62 |                  70 |
| Social Studies | 1-A       | CGID00002    | AID00156     |    51 |         54 |                  60 |
| Social Studies | 1-A       | CGID00002    | AID00277     |    50 |         50 |                  50 |
| Social Studies | 1-A       | CGID00002    | AID00299     |    45 |         54 |                  60 |
| Social Studies | 1-A       | CGID00002    | AID00300     |    46 |         67 |                  70 |
| Social Studies | 1-A       | CGID00002    | AID00301     |    68 |         90 |                 100 |
| Social Studies | 1-A       | CGID00002    | AID00302     |    30 |         89 |                  90 |
| Social Studies | 1-A       | CGID00002    | AID00303     |    48 |         48 |                  50 |
| Social Studies | 1-A       | CGID00002    | AID00304     |    34 |         64 |                  80 |
| Social Studies | 1-A       | CGID00002    | AID00305     |    64 |         95 |                 100 |
| Social Studies | 1-A       | CGID00002    | AID00306     |    66 |        100 |                 100 |
| Social Studies | 1-A       | CGID00002    | AID00307     |    97 |         97 |                 100 |
| Social Studies | 1-A       | CGID00002    | AID00308     |    50 |         50 |                  50 |
+---------------+-----------+--------------+--------------+-------+------------+---------------------+
15 rows in set (0.00 sec)
```

b. Query2 - STAR students in a classGroup

We want to display a list of STAR students in a particular classGroup. The requirement for being a STAR student is that a student must have grades above average in all assignments from that classGroup and attendance above average in that classGroup. We plan to display this statistic in every classGroup for teachers. We write a query to select classGroupId, userId for every STAR student.

select DISTINCT g.classGroupId, g.userId from Grades g JOIN Assignment a on g.assignmentId = a.assignmentId and g.classroomId IN (select a.classroomId from Attendance a group by classroomId, studentId having sum(a.isPresent) >=(select avg(totalAttendance) from (select classroomId, studentId, sum(isPresent) as totalAttendance from Attendance as a1 group by classroomId, studentId) as maxAttendance where maxAttendance.classroomId = a.classroomId) order by a.classroomId) group by g.classGroupId, g.userId having avg(g.grade/a.maximumGrade*100)>=(select avg(g1.grade/a1.maximumGrade*100) from Grades g1 JOIN Assignment a1 on g1.assignmentId = a1.assignmentId where g1.classGroupId = g.classGroupId) and g.classGroupId='CGID00142' order by g.classGroupId;



```
mysql> select DISTINCT g.classGroupId, g.userId from Grades g JOIN Assignment a on g.assignmentId = a.assignmentId and g.classroomId IN (select a.classroomId from Attendance a group by classroom
Id, studentId having sum(a.isPresent) >=(select avg(totalAttendance) from (select classroomId, studentId, sum(isPresent) as totalAttendance from Attendance as a1 group by classroomId, studentId)
 as maxAttendance where maxAttendance.classroomId = a.classroomId) order by a.classroomId) group by g.classGroupId, g.userId having avg(g.grade/a.maximumGrade*100)>=(select avg(g1.grade/a1.maxim
umGrade*100) from Grades g1 JOIN Assignment a1 on g1.assignmentId = a1.assignmentId where g1.classGroupId = g.classGroupId) and g.classGroupId='CGID00142' order by g.classGroupId;
+--------------+----------+
| classGroupId | userId   |
+--------------+----------+
| CGID00142    | UID00132 |
| CGID00142    | UID00877 |
| CGID00142    | UID00967 |
+--------------+----------+
3 rows in set (0.17 sec)
```

There are only 3 rows in this output.

c. Query3 - Assignment Average Grades

To estimate the difficulty of the assignments for future reference, we want to calculate the average grade as well as the maximum grade of each assignment across all students of a ClassGroup. We write a query to display the assignmentId, averageGrade, maximum grade as MaxStudentScore, maximum achievable grade for each assignment as MaxPossibleGrade. This will be displayed as a statistic on the assignments page.

select a.assignmentId, Round(AVG(g.grade), 2) as averageGrade, max(g.grade) as MaxStudentScore, a.maximumGrade as MaxPossibleGrade from Assignment a NATURAL JOIN Grades g where a.classGroupId = 'CGID00002' GROUP BY a.assignmentId, a.maximumGrade;

```
mysql> select a.assignmentId, Round(AVG(g.grade), 2) as averageGrade, max(g.grade) as MaxStudentScore, a.maximumGrade as MaxPossibleGrade  from Assignment a NATURAL JOIN Grades g where a.cla
ssGroupId = 'CGID00002' GROUP BY a.assignmentId, a.maximumGrade LIMIT 15;
+--------------+--------------+-----------------+------------------+
| assignmentId | averageGrade | MaxStudentScore | MaxPossibleGrade |
+--------------+--------------+-----------------+------------------+
| AID00156     |           41 |              54 |               60 |
| AID00277     |        39.57 |              50 |               50 |
| AID00299     |        42.57 |              54 |               60 |
| AID00300     |        50.86 |              67 |               70 |
| AID00301     |        72.86 |              90 |              100 |
| AID00302     |        58.86 |              89 |               90 |
| AID00303     |        42.71 |              48 |               50 |
| AID00304     |        47.71 |              64 |               80 |
| AID00305     |           70 |              95 |              100 |
| AID00306     |        70.71 |             100 |              100 |
| AID00307     |        66.14 |              97 |              100 |
| AID00308     |        41.14 |              50 |               50 |
| AID00309     |        42.71 |              50 |               50 |
| AID00310     |        76.14 |              98 |              100 |
| AID00311     |        65.86 |              88 |               90 |
+--------------+--------------+-----------------+------------------+
15 rows in set (0.00 sec)
```

d. Query4 - ClassGroup Topper

We want the admins to get a list of toppers from each classgroup. For this, we write a query which selects classGroupId, userId, firstName, lastName of the topper, and Average grade of the topper across all assignments as TopperAverage.

select g.classGroupId, u.userId as TopperUserId, u.firstName AS TopperFirstName, u.lastName AS TopperLastName, Round(max(g.grade/a.maximumGrade*100), 2) as TopperAverage from Assignment a NATURAL JOIN Grades g NATURAL JOIN Users u group by g.classGroupId, u.userId having (g.classGroupId, TopperAverage) IN (select g.classGroupId, Round(max(g.grade/a.maximumGrade*100), 2) as TopperAverage from Assignment a JOIN Grades g on a.assignmentId=g.assignmentId group by g.classGroupId order by g.classGroupId) order by g.classGroupId;

```
mysql> select g.classGroupId, u.userId as TopperUserId, u.firstName AS TopperFirstName, u.lastName AS TopperLastName, Round(max(g.grade/a.maximumGrade*100), 2) as TopperAverage from Assignment a
 NATURAL JOIN Grades g NATURAL JOIN Users u group by g.classGroupId, u.userId having (g.classGroupId, TopperAverage) IN (select g.classGroupId, Round(max(g.grade/a.maximumGrade*100), 2) as Toppe
rAverage from Assignment a NATURAL JOIN Grades g group by g.classGroupId order by g.classGroupId) order by g.classGroupId LIMIT 15;
+--------------+--------------+----------------+---------------+--------------+
| classGroupId | TopperUserId | TopperFirstName | TopperLastName | TopperAverage |
+--------------+--------------+----------------+---------------+--------------+
| CGID00000    | UID00020     | Burdette       | Bruen         |           95 |
| CGID00001    | UID00443     | Tamara         | Jones         |        88.57 |
| CGID00002    | UID00020     | Burdette       | Bruen         |          100 |
| CGID00002    | UID00612     | Reyes          | Mayert        |          100 |
| CGID00004    | UID00612     | Reyes          | Mayert        |          100 |
| CGID00007    | UID00323     | Marlene        | Stracke       |        91.43 |
| CGID00009    | UID00323     | Marlene        | Stracke       |           94 |
| CGID00012    | UID00298     | Daphnee        | Thompson      |        97.78 |
| CGID00013    | UID00251     | Wilburn        | Parker        |           94 |
| CGID00014    | UID00903     | Frieda         | Goyette       |           95 |
| CGID00015    | UID00883     | Ramiro         | MacGyver      |          100 |
| CGID00016    | UID00544     | Chaya          | Hintz         |          100 |
| CGID00018    | UID00357     | Sigmund        | Harvey        |        88.57 |
| CGID00019    | UID00357     | Sigmund        | Harvey        |        91.11 |
| CGID00020    | UID00363     | Jameson        | Block         |        94.44 |
+--------------+--------------+----------------+---------------+--------------+
15 rows in set (0.02 sec)
```

Displaying the top 15 rows.

Part 2: Indexing

Explain Analyze results before indexing:

1. Query 1:

```
-> Nested loop inner join  (cost=49.55 rows=0) (actual time=2.304..2.574 rows=25 loops=1)
    -> Nested loop inner join  (cost=18.30 rows=12) (actual time=0.058..0.277 rows=25 loops=1)
        -> Nested loop inner join  (cost=13.92 rows=12) (actual time=0.050..0.128 rows=25 loops=1)
            -> Nested loop inner join  (cost=10.17 rows=5) (actual time=0.037..0.065 rows=5 loops=1)
                -> Nested loop inner join  (cost=5.14 rows=5) (actual time=0.030..0.045 rows=5 loops=1)
                    -> Nested loop inner join  (cost=3.39 rows=5) (actual time=0.024..0.036 rows=5 loops=1)
                        -> Covering index lookup on cu using idx_classroomusers_userId_classroomId_classGroupId (userId='UID00020')  (cost=1.64 rows=5) (actual time=0.014..0.016 rows=5
 loops=1)
                        -> Single-row index lookup on co using PRIMARY (courseId=cu.courseId)  (cost=0.27 rows=1) (actual time=0.003..0.003 rows=1 loops=5)
                    -> Single-row index lookup on cr using PRIMARY (classroomId=cu.classroomId)  (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=5)
                -> Single-row covering index lookup on cg using PRIMARY (classGroupId=cu.classGroupId, classroomId=cu.classroomId, courseId=cu.courseId)  (cost=0.93 rows=1) (actual tim
e=0.004..0.004 rows=1 loops=5)
            -> Index lookup on a using PRIMARY (classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId)  (cost=0.55 rows=2) (actual time=0.008..0.012 rows=5 loo
ps=5)
        -> Single-row index lookup on g using PRIMARY (userId='UID00020', classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId, assignmentId=a.assignmentId)
(cost=0.26 rows=1) (actual time=0.006..0.006 rows=1 loops=25)
    -> Index lookup on maxGrade using <auto_key0> (assignmentId=a.assignmentId)  (actual time=0.091..0.092 rows=1 loops=25)
        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=2.241..2.241 rows=265 loops=1)
            -> Table scan on <temporary>  (actual time=1.854..1.887 rows=265 loops=1)
                -> Aggregate using temporary table  (actual time=1.854..1.854 rows=265 loops=1)
                    -> Table scan on Grades  (cost=177.75 rows=1720) (actual time=0.041..0.882 rows=1914 loops=1)
 |
```

Indexing-1 (Minimal Reduction of Cost (approximately .1)):

We have performed indexing on the userId column of the ClassroomUsers table which is referenced in a where clause (where cu.userId='UID00020'). Since you are checking where the child stands you need the specific user id. By indexing this one would hope to reduce the time it takes to find the specific userId. However, the cost difference was only 0.1 in the nested inner loop joins and the lookup for the userId.

```
mysql> CREATE INDEX cu_user_id on ClassroomUsers(userId);
Query OK, 0 rows affected (0.17 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
-> Nested loop inner join  (cost=49.45 rows=0) (actual time=2.461..2.797 rows=25 loops=1)
    -> Nested loop inner join  (cost=18.20 rows=12) (actual time=0.134..0.411 rows=25 loops=1)
        -> Nested loop inner join  (cost=13.82 rows=12) (actual time=0.118..0.212 rows=25 loops=1)
            -> Nested loop inner join  (cost=10.07 rows=5) (actual time=0.102..0.133 rows=5 loops=1)
                -> Nested loop inner join  (cost=5.04 rows=5) (actual time=0.091..0.109 rows=5 loops=1)
                    -> Nested loop inner join  (cost=3.29 rows=5) (actual time=0.081..0.095 rows=5 loops=1)
                        -> Covering index lookup on cu using cu_user_id (userId='UID00020')  (cost=1.54 rows=5) (actual time=0.067..0.070 rows=5 loops=1)
                        -> Single-row index lookup on co using PRIMARY (courseId=cu.courseId)  (cost=0.27 rows=1) (actual time=0.005..0.005 rows=1 loops=5)
                    -> Single-row index lookup on cr using PRIMARY (classroomId=cu.classroomId)  (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=5)
                -> Single-row covering index lookup on cg using PRIMARY (classGroupId=cu.classGroupId, classroomId=cu.classroomId, courseId=cu.courseId)  (cost=0.93 rows=1) (actual tim
e=0.005..0.005 rows=1 loops=5)
            -> Index lookup on a using PRIMARY (classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId)  (cost=0.55 rows=2) (actual time=0.010..0.015 rows=5 loo
ps=5)
        -> Single-row index lookup on g using PRIMARY (userId='UID00020', classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId, assignmentId=a.assignmentId)
(cost=0.26 rows=1) (actual time=0.007..0.008 rows=1 loops=25)
    -> Index lookup on maxGrade using <auto_key0> (assignmentId=a.assignmentId)  (actual time=0.095..0.095 rows=1 loops=25)
        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=2.319..2.319 rows=265 loops=1)
            -> Table scan on <temporary>  (actual time=1.903..1.939 rows=265 loops=1)
                -> Aggregate using temporary table  (actual time=1.902..1.902 rows=265 loops=1)
                    -> Table scan on Grades  (cost=177.75 rows=1720) (actual time=0.049..0.906 rows=1914 loops=1)
 |
```

Indexing-2(Increase in Cost From 49.5 to 2196.27); Not Recommended

```
mysql>
mysql> CREATE INDEX g_assignment_id on Grades(assignmentId);
Query OK, 0 rows affected, 1 warning (0.12 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Nested loop inner join  (cost=2196.27 rows=21500) (actual time=6.584..7.239 rows=25 loops=1)
    -> Nested loop inner join  (cost=15.02 rows=12) (actual time=0.085..0.681 rows=25 loops=1)
        -> Nested loop inner join  (cost=10.64 rows=12) (actual time=0.072..0.600 rows=25 loops=1)
            -> Nested loop inner join  (cost=6.89 rows=5) (actual time=0.058..0.528 rows=5 loops=1)
                -> Nested loop inner join  (cost=5.14 rows=5) (actual time=0.046..0.146 rows=5 loops=1)
                    -> Nested loop inner join  (cost=3.39 rows=5) (actual time=0.038..0.134 rows=5 loops=1)
                        -> Covering index lookup on cu using idx_classroomusers_userId_classroomId_classGroupId (userId='UID00020')  (cost=1.64 rows=5) (actual time=0.026..0.032 rows=5 loop
s=1)
                        -> Single-row index lookup on co using PRIMARY (courseId=cu.courseId)  (cost=0.27 rows=1) (actual time=0.020..0.020 rows=1 loops=5)
                    -> Single-row index lookup on cr using PRIMARY (classroomId=cu.classroomId)  (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=5)
                -> Single-row covering index lookup on cg using PRIMARY (classGroupId=cu.classGroupId, classroomId=cu.classroomId, courseId=cu.courseId)  (cost=0.27 rows=1) (actual time=0.0
76..0.076 rows=1 loops=5)
            -> Index lookup on a using PRIMARY (classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId)  (cost=0.55 rows=2) (actual time=0.010..0.014 rows=5 loops=5)
        -> Single-row index lookup on g using PRIMARY (userId='UID00020', classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId, assignmentId=a.assignmentId)  (cost
=0.26 rows=1) (actual time=0.003..0.003 rows=1 loops=25)
    -> Index lookup on maxGrade using <auto_key0> (assignmentId=a.assignmentId)  (actual time=0.262..0.262 rows=1 loops=25)
        -> Materialize  (cost=521.75..521.75 rows=1720) (actual time=6.491..6.491 rows=265 loops=1)
            -> Group aggregate: max(Grades.grade)  (cost=349.75 rows=1720) (actual time=0.433..6.026 rows=265 loops=1)
                -> Index scan on Grades using idx_assid_grd  (cost=177.75 rows=1720) (actual time=0.423..5.530 rows=1914 loops=1)
 |
```

We have performed indexing on the assignmentId column of the Grades table which is referenced when doing a group by while calculating the max grade. Since there is a group by assignmentId within grades, we assumed it would lower the running time. However, the cost of the outer nested loop inner join jumped from 49.55 to 2196.27 which is a drastic decrease in performance.

Indexing-3(No Reduction of Cost)

```
mysql> CREATE INDEX a_class_group_id  on Assignment(classGroupId);
Query OK, 0 rows affected (0.08 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Nested loop inner join  (cost=49.55 rows=0) (actual time=2.436..2.866 rows=25 loops=1)
    -> Nested loop inner join  (cost=18.30 rows=12) (actual time=0.068..0.443 rows=25 loops=1)
        -> Nested loop inner join  (cost=13.92 rows=12) (actual time=0.059..0.157 rows=25 loops=1)
            -> Nested loop inner join  (cost=10.17 rows=5) (actual time=0.045..0.084 rows=5 loops=1)
                -> Nested loop inner join  (cost=5.14 rows=5) (actual time=0.036..0.059 rows=5 loops=1)
                    -> Nested loop inner join  (cost=3.39 rows=5) (actual time=0.027..0.046 rows=5 loops=1)
                        -> Covering index lookup on cu using idx_classroomusers_userId_classroomId_classGroupId (userId='UID00020')  (cost=1.64 rows=5) (actual time=0.016..0.021 rows=5
 loops=1)
                        -> Single-row index lookup on co using PRIMARY (courseId=cu.courseId)  (cost=0.27 rows=1) (actual time=0.004..0.004 rows=1 loops=5)
                    -> Single-row index lookup on cr using PRIMARY (classroomId=cu.classroomId)  (cost=0.27 rows=1) (actual time=0.002..0.002 rows=1 loops=5)
                -> Single-row covering index lookup on cg using PRIMARY (classGroupId=cu.classGroupId, classroomId=cu.classroomId, courseId=cu.courseId)  (cost=0.93 rows=1) (actual tim
e=0.005..0.005 rows=1 loops=5)
            -> Index lookup on a using PRIMARY (classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId)  (cost=0.55 rows=2) (actual time=0.009..0.014 rows=5 loo
ps=5)
        -> Single-row index lookup on g using PRIMARY (userId='UID00020', classroomId=cu.classroomId, courseId=cu.courseId, classGroupId=cu.classGroupId, assignmentId=a.assignmentId)
(cost=0.26 rows=1) (actual time=0.011..0.011 rows=1 loops=25)
    -> Index lookup on maxGrade using <auto_key0> (assignmentId=a.assignmentId)  (actual time=0.096..0.097 rows=1 loops=25)
        -> Materialize  (cost=0.00..0.00 rows=0) (actual time=2.361..2.361 rows=265 loops=1)
            -> Table scan on <temporary>  (actual time=1.977..2.012 rows=265 loops=1)
                -> Aggregate using temporary table  (actual time=1.976..1.976 rows=265 loops=1)
                    -> Table scan on Grades  (cost=177.75 rows=1720) (actual time=0.048..0.957 rows=1914 loops=1)
 |
```

We have performed indexing on the classGroupId column of Assignment which is used within the Natural Join of the Assignment and ClassroomGroup tables. Unfortunately this did not change the cost of the nested loop inner Joins.

2. **Query 2:**

Before any indexing:

```
| -> Sort: g.classGroupId, g.userId  (actual time=425.136..425.136 rows=3 loops=1)
    -> Filter: ((avg(((g.grade / a.maximumGrade) * 100)) >= (select #5)) and (g.classGroupId = 'CGID00142'))  (actual time=423.987..424.157 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=423.928..424.080 rows=698 loops=1)
            -> Aggregate using temporary table  (actual time=423.925..423.925 rows=698 loops=1)
                -> Filter: <in_optimizer>(g.classroomId,g.classroomId in (select #2))  (cost=51637.93 rows=51600) (actual time=180.281..184.326 rows=1914 loops=1)
                    -> Inner hash join (g.assignmentId = a.assignmentId)  (cost=51637.93 rows=51600) (actual time=32.446..34.753 rows=1914 loops=1)
                        -> Table scan on g  (cost=0.08 rows=1720) (actual time=0.067..1.237 rows=1914 loops=1)
                        -> Hash
                            -> Table scan on a  (cost=31.88 rows=300) (actual time=0.121..31.059 rows=315 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((g.classroomId = `<materialized_subquery>`.classroomId))  (cost=1808.50..1808.50 rows=1) (actual time=0.473..0.473 rows=1 loops=312)
                            -> Limit: 1 row(s)  (cost=1808.40..1808.40 rows=1) (actual time=0.472..0.472 rows=1 loops=312)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (classroomId=g.classroomId)  (actual time=0.472..0.472 rows=1 loops=312)
                                    -> Materialize with deduplication  (cost=1808.40..1808.40 rows=5918) (actual time=146.742..146.742 rows=142 loops=1)
                                        -> Filter: (sum(a.isPresent) >= (select #3))  (cost=1216.60 rows=5918) (actual time=135.959..146.507 rows=500 loops=1)
                                            -> Group aggregate: sum(a.isPresent)  (cost=1216.60 rows=5918) (actual time=97.450..102.399 rows=994 loops=1)
                                                -> Index scan on a using PRIMARY  (cost=624.80 rows=5918) (actual time=97.407..99.634 rows=5964 loops=1)
                                            -> Select #3 (subquery in condition; dependent)
                                                -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2400.20..2400.20 rows=1) (actual time=0.041..0.041 rows=1 loops=994)
                                                    -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.038..0.040 rows=8 loops=994)
                                                        -> Materialize  (cost=1808.40..1808.40 rows=5918) (actual time=36.222..36.222 rows=994 loops=1)
                                                            -> Group aggregate: sum(a1.isPresent)  (cost=1216.60 rows=5918) (actual time=0.087..33.155 rows=994 loops=1)
                                                                -> Index scan on a1 using PRIMARY  (cost=624.80 rows=5918) (actual time=0.069..29.704 rows=5964 loops=1)
                    -> Select #3 (subquery in projection; dependent)
                        -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2400.20..2400.20 rows=1) (actual time=0.041..0.041 rows=1 loops=994)
                            -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.038..0.040 rows=8 loops=994)
                                -> Materialize  (cost=1808.40..1808.40 rows=5918) (actual time=36.222..36.222 rows=994 loops=1)
                                    -> Group aggregate: sum(a1.isPresent)  (cost=1216.60 rows=5918) (actual time=0.087..33.155 rows=994 loops=1)
                                        -> Index scan on a1 using PRIMARY  (cost=624.80 rows=5918) (actual time=0.069..29.704 rows=5964 loops=1)
        -> Select #5 (subquery in condition; dependent)
            -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=625.39 rows=1) (actual time=0.336..0.336 rows=1 loops=698)
                -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=569.30 rows=561) (actual time=0.270..0.333 rows=22 loops=698)
                    -> Table scan on a1  (cost=0.26 rows=300) (actual time=0.020..0.094 rows=315 loops=698)
                    -> Hash
                        -> Index lookup on g1 using classGroupId (classGroupId=g.classGroupId)  (cost=6.54 rows=19) (actual time=0.153..0.180 rows=22 loops=698)
-> Select #5 (subquery in projection; dependent)
    -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=625.39 rows=1) (actual time=0.336..0.336 rows=1 loops=698)
        -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=569.30 rows=561) (actual time=0.270..0.333 rows=22 loops=698)
            -> Table scan on a1  (cost=0.26 rows=300) (actual time=0.020..0.094 rows=315 loops=698)
            -> Hash
                -> Index lookup on g1 using classGroupId (classGroupId=g.classGroupId)  (cost=6.54 rows=19) (actual time=0.153..0.180 rows=22 loops=698)
|
```

Indexing-1 (Reduction of Cost from 1808.4 to 1789.06 - Select#2):

We have performed indexing on the isPresent column of Attendance which is a crucial attribute to determine whether a student is a STAR student or not, which is used for analyzing the average attendance of students per classGroup, used in two aggregation functions - sum and average.

```
mysql> CREATE INDEX idx_isp_att ON Attendance(isPresent);
Query OK, 0 rows affected (0.34 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Sort: g.classGroupId, g.userId  (actual time=245.299..245.300 rows=3 loops=1)
    -> Filter: ((avg(((g.grade / a.maximumGrade) * 100)) >= (select #5)) and (g.classGroupId = 'CGID00142'))  (actual time=245.025..245.194 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=244.995..245.130 rows=698 loops=1)
            -> Aggregate using temporary table  (actual time=244.993..244.993 rows=698 loops=1)
                -> Filter: <in_optimizer>(g.classroomId,g.classroomId in (select #2))  (cost=51636.80 rows=51600) (actual time=75.080..78.435 rows=1914 loops=1)
                    -> Inner hash join (g.assignmentId = a.assignmentId)  (cost=51636.80 rows=51600) (actual time=2.150..4.038 rows=1914 loops=1)
                        -> Table scan on g  (cost=0.08 rows=1720) (actual time=0.148..1.166 rows=1914 loops=1)
                        -> Hash
                            -> Table scan on a  (cost=30.75 rows=300) (actual time=0.581..0.730 rows=315 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((g.classroomId = `<materialized_subquery>`.classroomId))  (cost=1789.16..1789.16 rows=1) (actual time=0.235..0.235 rows=1 loops=312)
                            -> Limit: 1 row(s)  (cost=1789.06..1789.06 rows=1) (actual time=0.235..0.235 rows=1 loops=312)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (classroomId=g.classroomId)  (actual time=0.235..0.235 rows=1 loops=312)
                                    -> Materialize with deduplication  (cost=1789.06..1789.06 rows=5918) (actual time=72.904..72.904 rows=142 loops=1)
                                        -> Filter: (sum(a.isPresent) >= (select #3))  (cost=1197.26 rows=5918) (actual time=62.441..72.679 rows=500 loops=1)
                                            -> Group aggregate: sum(a.isPresent)  (cost=1197.26 rows=5918) (actual time=50.998..55.771 rows=994 loops=1)
                                                -> Index scan on a using PRIMARY  (cost=605.46 rows=5918) (actual time=50.974..53.028 rows=5964 loops=1)
                                            -> Select #3 (subquery in condition; dependent)
                                                -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2380.86..2380.86 rows=1) (actual time=0.016..0.016 rows=1 loops=994)
                                                    -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.013..0.015 rows=8 loops=994)
                                                        -> Materialize  (cost=1789.06..1789.06 rows=5918) (actual time=11.345..11.345 rows=994 loops=1)
                                                            -> Group aggregate: sum(a1.isPresent)  (cost=1197.26 rows=5918) (actual time=0.090..8.828 rows=994 loops=1)
                                                                -> Index scan on a1 using PRIMARY  (cost=605.46 rows=5918) (actual time=0.084..6.115 rows=5964 loops=1)
                    -> Select #3 (subquery in projection; dependent)
                        -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2380.86..2380.86 rows=1) (actual time=0.016..0.016 rows=1 loops=994)
                            -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.013..0.015 rows=8 loops=994)
                                -> Materialize  (cost=1789.06..1789.06 rows=5918) (actual time=11.345..11.345 rows=994 loops=1)
                                    -> Group aggregate: sum(a1.isPresent)  (cost=1197.26 rows=5918) (actual time=0.090..8.828 rows=994 loops=1)
                                        -> Index scan on a1 using PRIMARY  (cost=605.46 rows=5918) (actual time=0.084..6.115 rows=5964 loops=1)
        -> Select #5 (subquery in condition; dependent)
            -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=624.25 rows=1) (actual time=0.233..0.233 rows=1 loops=698)
                -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=568.17 rows=561) (actual time=0.166..0.230 rows=22 loops=698)
                    -> Table scan on a1  (cost=0.20 rows=300) (actual time=0.020..0.096 rows=315 loops=698)
                    -> Hash
                        -> Index lookup on g1 using classGroupId (classGroupId=g.classGroupId)  (cost=6.54 rows=19) (actual time=0.051..0.074 rows=22 loops=698)
-> Select #5 (subquery in projection; dependent)
    -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=624.25 rows=1) (actual time=0.233..0.233 rows=1 loops=698)
        -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=568.17 rows=561) (actual time=0.166..0.230 rows=22 loops=698)
            -> Table scan on a1  (cost=0.20 rows=300) (actual time=0.020..0.096 rows=315 loops=698)
            -> Hash
                -> Index lookup on g1 using classGroupId (classGroupId=g.classGroupId)  (cost=6.54 rows=19) (actual time=0.051..0.074 rows=22 loops=698)
```

Indexing-2 (Reduction of Cost from 624.25 to 580.05 - Select#5):

We performed index on the classGroupId and the userId on the Grades table in which classGroupId attribute is used to perform Join on Grades and Assignment, and classGroupId and userId is used in the groupBy clause while grouping Grades. This indexing technique helps in reducing the cost since we can easily find the distinct classGroupId's and userId's out of all the values present in each column.

```
mysql> CREATE INDEX idx_grades_classgroupid_userid ON Grades(classGroupId, userId);
Query OK, 0 rows affected (0.10 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Sort: g.classGroupId, g.userId  (actual time=186.211..186.211 rows=3 loops=1)
    -> Filter: ((avg(((g.grade / a.maximumGrade) * 100)) >= (select #5)) and (g.classGroupId = 'CGID00142'))  (actual time=186.028..186.196 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=185.999..186.126 rows=698 loops=1)
            -> Aggregate using temporary table  (actual time=185.997..185.997 rows=698 loops=1)
                -> Filter: <in_optimizer>(g.classroomId,g.classroomId in (select #2))  (cost=51636.80 rows=51600) (actual time=18.327..21.830 rows=1914 loops=1)
                    -> Inner hash join (g.assignmentId = a.assignmentId)  (cost=51636.80 rows=51600) (actual time=0.388..2.306 rows=1914 loops=1)
                        -> Table scan on g  (cost=0.08 rows=1720) (actual time=0.068..1.090 rows=1914 loops=1)
                        -> Hash
                            -> Table scan on a  (cost=30.75 rows=300) (actual time=0.085..0.196 rows=315 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((g.classroomId = `<materialized_subquery>`.classroomId))  (cost=1783.75..1783.75 rows=1) (actual time=0.059..0.059 rows=1 loops=312)
                            -> Limit: 1 row(s)  (cost=1783.65..1783.65 rows=1) (actual time=0.059..0.059 rows=1 loops=312)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (classroomId=g.classroomId)  (actual time=0.059..0.059 rows=1 loops=312)
                                    -> Materialize with deduplication  (cost=1783.65..1783.65 rows=5918) (actual time=17.921..17.921 rows=142 loops=1)
                                        -> Filter: (sum(a.isPresent) >= (select #3))  (cost=1191.85 rows=5918) (actual time=7.615..17.719 rows=500 loops=1)
                                            -> Group aggregate: sum(a.isPresent)  (cost=1191.85 rows=5918) (actual time=0.078..4.807 rows=994 loops=1)
                                                -> Index scan on a using PRIMARY  (cost=600.05 rows=5918) (actual time=0.061..2.107 rows=5964 loops=1)
                                            -> Select #3 (subquery in condition; dependent)
                                                -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2375.45..2375.45 rows=1) (actual time=0.012..0.012 rows=1 loops=994)
                                                    -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.009..0.011 rows=8 loops=994)
                                                        -> Materialize  (cost=1783.65..1783.65 rows=5918) (actual time=7.493..7.493 rows=994 loops=1)
                                                            -> Group aggregate: sum(a1.isPresent)  (cost=1191.85 rows=5918) (actual time=0.061..5.027 rows=994 loops=1)
                                                                -> Index scan on a1 using PRIMARY  (cost=600.05 rows=5918) (actual time=0.054..2.286 rows=5964 loops=1)
                    -> Select #3 (subquery in projection; dependent)
                        -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2375.45..2375.45 rows=1) (actual time=0.012..0.012 rows=1 loops=994)
                            -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.009..0.011 rows=8 loops=994)
                                -> Materialize  (cost=1783.65..1783.65 rows=5918) (actual time=7.493..7.493 rows=994 loops=1)
                                    -> Group aggregate: sum(a1.isPresent)  (cost=1191.85 rows=5918) (actual time=0.061..5.027 rows=994 loops=1)
                                        -> Index scan on a1 using PRIMARY  (cost=600.05 rows=5918) (actual time=0.054..2.286 rows=5964 loops=1)
        -> Select #5 (subquery in condition; dependent)
            -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=580.17 rows=1) (actual time=0.229..0.229 rows=1 loops=698)
                -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=528.05 rows=521) (actual time=0.164..0.226 rows=22 loops=698)
                    -> Table scan on a1  (cost=0.22 rows=300) (actual time=0.020..0.094 rows=315 loops=698)
                    -> Hash
                        -> Index lookup on g1 using idx_grades_classgroupid_userid (classGroupId=g.classGroupId)  (cost=6.08 rows=17) (actual time=0.048..0.074 rows=22 loops=698)
-> Select #5 (subquery in projection; dependent)
    -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=580.17 rows=1) (actual time=0.229..0.229 rows=1 loops=698)
        -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=528.05 rows=521) (actual time=0.164..0.226 rows=22 loops=698)
            -> Table scan on a1  (cost=0.22 rows=300) (actual time=0.020..0.094 rows=315 loops=698)
            -> Hash
                -> Index lookup on g1 using idx_grades_classgroupid_userid (classGroupId=g.classGroupId)  (cost=6.08 rows=17) (actual time=0.048..0.074 rows=22 loops=698)
```

Indexing-3 (Reduction of Cost from 1808.4 to 1783.65 - Select#2):

We observed similar results as the above indexing on adding index to grade and maximumGrade on Grades and Assignment respectively. The index scan is happening on grade in the Grades table rather than having a normal scan on grades as done in the before indexing query. As a result, we do observe some drop in the cost.

```
mysql> CREATE INDEX idx_grade_gd ON Grades(grade);
Query OK, 0 rows affected, 1 warning (0.15 sec)
Records: 0  Duplicates: 0  Warnings: 1

mysql> CREATE INDEX idx_mxmgrd_ass ON Assignment(maximumGrade);
Query OK, 0 rows affected, 1 warning (0.09 sec)
Records: 0  Duplicates: 0  Warnings: 1
```

```
| -> Sort: g.classGroupId, g.userId  (actual time=202.745..202.745 rows=3 loops=1)
    -> Filter: ((avg(((g.grade / a.maximumGrade) * 100)) >= (select #5)) and (g.classGroupId = 'CGID00142'))  (actual time=202.524..202.704 rows=3 loops=1)
        -> Table scan on <temporary>  (actual time=202.466..202.643 rows=698 loops=1)
            -> Aggregate using temporary table  (actual time=202.463..202.463 rows=698 loops=1)
                -> Filter: <in_optimizer>(g.classroomId,g.classroomId in (select #2))  (cost=51636.80 rows=51600) (actual time=20.205..25.913 rows=1914 loops=1)
                    -> Inner hash join (g.assignmentId = a.assignmentId)  (cost=51636.80 rows=51600) (actual time=0.434..2.667 rows=1914 loops=1)
                        -> Covering index scan on g using g_grade  (cost=0.08 rows=1720) (actual time=0.046..1.118 rows=1914 loops=1)
                        -> Hash
                            -> Covering index scan on a using a_maximum_grade  (cost=30.75 rows=300) (actual time=0.123..0.212 rows=315 loops=1)
                    -> Select #2 (subquery in condition; run only once)
                        -> Filter: ((g.classroomId = `<materialized_subquery>`.classroomId))  (cost=1783.75..1783.75 rows=1) (actual time=0.013..0.013 rows=1 loops=1698)
                            -> Limit: 1 row(s)  (cost=1783.65..1783.65 rows=1) (actual time=0.012..0.012 rows=1 loops=1698)
                                -> Index lookup on <materialized_subquery> using <auto_distinct_key> (classroomId=g.classroomId)  (actual time=0.012..0.012 rows=1 loops=1698)
                                    -> Materialize with deduplication  (cost=1783.65..1783.65 rows=5918) (actual time=19.747..19.747 rows=142 loops=1)
                                        -> Filter: (sum(a.isPresent) >= (select #3))  (cost=1191.85 rows=5918) (actual time=8.727..19.503 rows=500 loops=1)
                                            -> Group aggregate: sum(a.isPresent)  (cost=1191.85 rows=5918) (actual time=0.056..5.233 rows=994 loops=1)
                                                -> Index scan on a using PRIMARY  (cost=600.05 rows=5918) (actual time=0.043..2.483 rows=5964 loops=1)
                                        -> Select #3 (subquery in condition; dependent)
                                            -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2375.45..2375.45 rows=1) (actual time=0.013..0.013 rows=1 loops=994)
                                                -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.010..0.011 rows=8 loops=994)
                                                    -> Materialize  (cost=1783.65..1783.65 rows=5918) (actual time=7.549..7.549 rows=994 loops=1)
                                                        -> Group aggregate: sum(a1.isPresent)  (cost=1191.85 rows=5918) (actual time=0.034..4.907 rows=994 loops=1)
                                                            -> Index scan on a1 using PRIMARY  (cost=600.05 rows=5918) (actual time=0.029..2.174 rows=5964 loops=1)
                    -> Select #3 (subquery in projection; dependent)
                        -> Aggregate: avg(maxAttendance.totalAttendance)  (cost=2375.45..2375.45 rows=1) (actual time=0.013..0.013 rows=1 loops=994)
                            -> Index lookup on maxAttendance using <auto_key0> (classroomId=a.classroomId)  (actual time=0.010..0.011 rows=8 loops=994)
                                -> Materialize  (cost=1783.65..1783.65 rows=5918) (actual time=7.549..7.549 rows=994 loops=1)
                                    -> Group aggregate: sum(a1.isPresent)  (cost=1191.85 rows=5918) (actual time=0.034..4.907 rows=994 loops=1)
                                        -> Index scan on a1 using PRIMARY  (cost=600.05 rows=5918) (actual time=0.029..2.174 rows=5964 loops=1)
    -> Select #5 (subquery in condition; dependent)
        -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=624.25 rows=1) (actual time=0.245..0.245 rows=1 loops=698)
            -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=568.17 rows=561) (actual time=0.150..0.242 rows=22 loops=698)
                -> Covering index scan on a1 using a_maximum_grade  (cost=0.20 rows=300) (actual time=0.018..0.092 rows=315 loops=698)
                -> Hash
                    -> Index lookup on g1 using classGroupId (classGroupId=g.classGroupId)  (cost=6.54 rows=19) (actual time=0.057..0.086 rows=22 loops=698)
-> Select #5 (subquery in projection; dependent)
    -> Aggregate: avg(((g1.grade / a1.maximumGrade) * 100))  (cost=624.25 rows=1) (actual time=0.245..0.245 rows=1 loops=698)
        -> Inner hash join (a1.assignmentId = g1.assignmentId)  (cost=568.17 rows=561) (actual time=0.150..0.242 rows=22 loops=698)
            -> Covering index scan on a1 using a_maximum_grade  (cost=0.20 rows=300) (actual time=0.018..0.092 rows=315 loops=698)
            -> Hash
                -> Index lookup on g1 using classGroupId (classGroupId=g.classGroupId)  (cost=6.54 rows=19) (actual time=0.057..0.086 rows=22 loops=698)
 |
```

3. Query 3:

Before Indexing:

```
-----------------------------------------+
| -> Table scan on <temporary>  (actual time=0.903..0.906 rows=18 loops=1)
    -> Aggregate using temporary table  (actual time=0.901..0.901 rows=18 loops=1)
        -> Nested loop inner join  (cost=52.98 rows=140) (actual time=0.133..0.742 rows=126 loops=1)
            -> Index lookup on a using classGroupId (classGroupId='CGID00002')  (cost=4.05 rows=18) (actual time=0.080..0.085 rows=18 loops=1)
            -> Index lookup on g using classGroupId (classGroupId='CGID00002', classroomId=a.classroomId, courseId=a.courseId, assignmentId=a.assignmentId)  (cost=1.98 rows=8) (actual time=0.031
..0.036 rows=7 loops=18)
 |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Indexing-1(Reduction of Cost from 52.98 to 3.67- overall):

We observed a significant reduction in the cost of using an index on classGroupID on the Assignment table.This can be attributed to the use of the above attribute in the when clause in the last part of the subquery which reduces the search time significantly.

```
mysql> CREATE INDEX idx_classgroupid ON Assignment(classGroupID);
Query OK, 0 rows affected (0.33 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> Explain ANALYZE select a.assignmentId, Round(AVG(g.grade), 2) as averageGrade, max(g.grade) as MaxStudentScore, a.maximumGrade as MaxPossibleGrade  from Assignment a NATURAL JOIN Grades g where a.classGrou
pId = 'CGID00002' GROUP BY a.assignmentId, a.maximumGrade;
+------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                             |
+------------------------------------------------------------------------------------------------------------+
| -> Table scan on <temporary>  (actual time=1.137..1.140 rows=18 loops=1)
    -> Aggregate using temporary table  (actual time=1.136..1.136 rows=18 loops=1)
        -> Nested loop inner join  (cost=3.67 rows=7) (actual time=0.258..0.980 rows=126 loops=1)
            -> Filter: (a.classGroupId = 'CGID00002')  (cost=1.11 rows=1) (actual time=0.187..0.318 rows=18 loops=1)
                -> Intersect rows sorted by row ID  (cost=1.11 rows=1) (actual time=0.183..0.308 rows=18 loops=1)
                    -> Index range scan on a using classGroupId over (classGroupId = 'CGID00002')  (cost=0.40 rows=18) (actual time=0.114..0.126 rows=18 loops=1)
                    -> Index range scan on a using idx_classgroupid over (classGroupId = 'CGID00002')  (cost=0.36 rows=18) (actual time=0.046..0.055 rows=18 loops=1)
            -> Index lookup on g using classGroupId (classGroupId='CGID00002', classroomId=a.classroomId, courseId=a.courseId, assignmentId=a.assignmentId)  (cost=2.56 rows=7) (actual time=0.033..0.035 rows=7 loo
ps=18)
 |
+------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Indexing -2(Increase of Cost from 52.98 to 1549.85 - overall) Not Recommended:

On using index on the assignmentID for both Grades and Assignment, we see a drop in the cost for the last part of the subquery from 1.98 to 1.62, however, it comes at the cost of an overall increase in cost of execution of the Nested inner loop. Indexes do not always work in favour of lowering the cost, and this is an example that index might reduce the cost of a part of a subquery, but could eventually lead to an overall increase in cost.

```
mysql> CREATE INDEX idx_assignmentid_assignment ON Assignment(assignmentID);
Query OK, 0 rows affected (0.10 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_assignmentid_grade ON Grades(assignmentID);
Query OK, 0 rows affected (0.14 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> Explain ANALYZE select a.assignmentId, Round(AVG(g.grade), 2) as averageGrade, max(g.grade) as MaxStudentScore, a.maximumGrade as MaxPossibleGrade  from Assignment a NATURAL JOIN Grades g where a.classGrou
pId = 'CGID00002' GROUP BY a.assignmentId, a.maximumGrade;
+------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                            |
+------------------------------------------------------------------------------------------------------------+
| -> Table scan on <temporary>  (actual time=0.906..0.909 rows=18 loops=1)
    -> Aggregate using temporary table  (actual time=0.905..0.905 rows=18 loops=1)
        -> Nested loop inner join  (cost=1549.85 rows=117) (actual time=0.220..0.789 rows=126 loops=1)
            -> Index lookup on a using classGroupId (classGroupId='CGID00002')  (cost=4.05 rows=18) (actual time=0.155..0.158 rows=18 loops=1)
            -> Index lookup on g using idx_assignmentid_grade (assignmentId=a.assignmentId), with index condition: ((g.courseId = a.courseId) and (g.classroomId = a.classroomId) and (g.classGroupId = 'CGID00002')
)  (cost=1.62 rows=6) (actual time=0.033..0.034 rows=7 loops=18)
 |
+------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Indexing-3 (Reduction of Cost from 52.98 to 50.16):
There isn't much effect on adding indexing to the maximumGrade of Assignment since it is only used in the select statement and not as a filter to retrieve any new information or perform any lookup. The assignmentId of Assignment is used to perform group by which helps to reduce the slight cost we have seen in the above example.

```
mysql> CREATE INDEX idx_assignmentid ON Assignment(assignmentId);
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> CREATE INDEX idx_maxgrade ON Assignment(maximumGrade);
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> Explain ANALYZE select a.assignmentId, Round(AVG(g.grade), 2) as averageGrade, max(g.grade) as MaxStudentScore, a.maximumGrade as MaxPossibleGrade  from Assignment a NATURAL JOIN Grades g where a.classGrou
pId = 'CGID00002' GROUP BY a.assignmentId, a.maximumGrade;
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| EXPLAIN
                                                                                                                                                                                      |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Table scan on <temporary>  (actual time=0.954..0.957 rows=18 loops=1)
    -> Aggregate using temporary table  (actual time=0.952..0.952 rows=18 loops=1)
        -> Nested loop inner join  (cost=50.16 rows=132) (actual time=0.189..0.832 rows=126 loops=1)
            -> Index lookup on a using classGroupId (classGroupId='CGID00002')  (cost=4.05 rows=18) (actual time=0.130..0.135 rows=18 loops=1)
            -> Index lookup on g using classGroupId (classGroupId='CGID00002', classroomId=a.classroomId, courseId=a.courseId, assignmentId=a.assignmentId)  (cost=1.87 rows=7) (actual time=0.036..0.038 rows=7 loo
ps=18)
 |
+------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

4. **Query 4:**

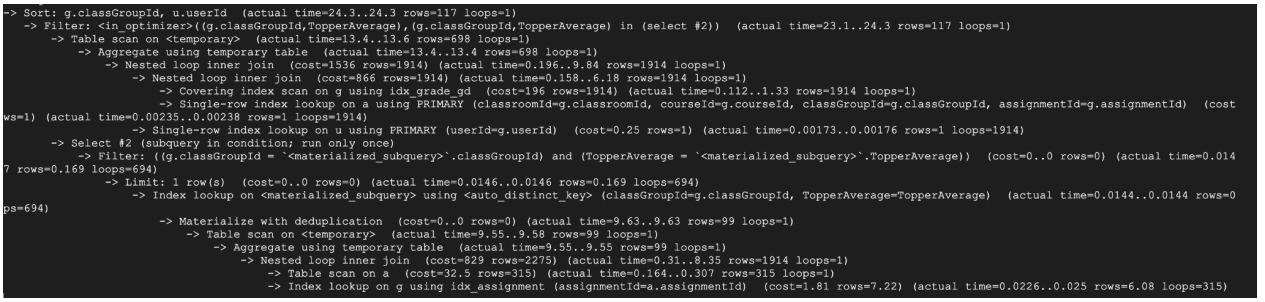
Before Indexing:

```
-> Sort: g.classGroupId, u.userId  (actual time=17..17 rows=117 loops=1)
    -> Filter: <in_optimizer>((g.classGroupId,TopperAverage),(g.classGroupId,TopperAverage) in (select #2))  (actual time=15.7..16.8 rows=117 loops=1)
        -> Table scan on <temporary>  (actual time=12..12.2 rows=698 loops=1)
            -> Aggregate using temporary table  (actual time=12..12 rows=698 loops=1)
                -> Nested loop inner join  (cost=1536 rows=1914) (actual time=0.488..8.87 rows=1914 loops=1)
                    -> Nested loop inner join  (cost=866 rows=1914) (actual time=0.409..7.39 rows=1914 loops=1)
                        -> Table scan on g  (cost=196 rows=1914) (actual time=0.266..1.56 rows=1914 loops=1)
                        -> Single-row index lookup on a using PRIMARY (classroomId=g.classroomId, courseId=g.courseId, classGroupId=g.classGroupId, assignmentId=g.assignmentId)  (cost
ows=1) (actual time=0.00281..0.00285 rows=1 loops=1914)
                    -> Single-row index lookup on u using PRIMARY (userId=g.userId)  (cost=0.25 rows=1) (actual time=552e-6..588e-6 rows=1 loops=1914)
        -> Select #2 (subquery in condition; run only once)
            -> Filter: ((g.classGroupId = `<materialized_subquery>`.classGroupId) and (TopperAverage = `<materialized_subquery>`.TopperAverage))  (cost=0..0 rows=0) (actual time=0.006
0602 rows=0.167 loops=699)
                -> Limit: 1 row(s)  (cost=0..0 rows=0) (actual time=0.00585..0.00585 rows=0.167 loops=699)
                    -> Index lookup on <materialized_subquery> using <auto_distinct_key> (classGroupId=g.classGroupId, TopperAverage=TopperAverage)  (actual time=0.00573..0.00573 rows
loops=699)
                        -> Materialize with deduplication  (cost=0..0 rows=0) (actual time=3.59..3.59 rows=99 loops=1)
                            -> Table scan on <temporary>  (actual time=3.53..3.54 rows=99 loops=1)
                                -> Aggregate using temporary table  (actual time=3.52..3.52 rows=99 loops=1)
                                    -> Inner hash join (g.assignmentId = a.assignmentId)  (cost=60328 rows=60291) (actual time=1.34..2.58 rows=1914 loops=1)
                                        -> Table scan on g  (cost=0.0758 rows=1914) (actual time=0.0492..0.895 rows=1914 loops=1)
                                        -> Hash
                                            -> Table scan on a  (cost=32.5 rows=315) (actual time=0.995..1.08 rows=315 loops=1)
```

Indexing-1 (Reduction of Cost from 60328 to 829 on Inner Join):

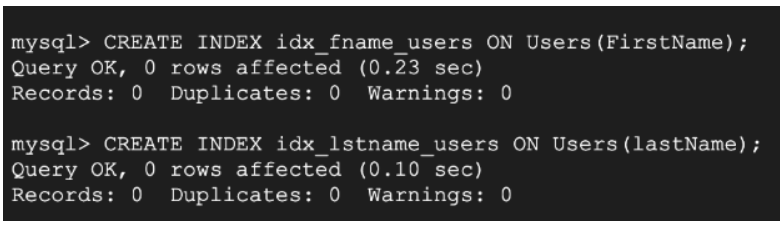
On performing indexing on assignmentId on table Grades, we see a significant reduction in cost. This can be attributed to the use of the above column in the Having condition being used to check for different grades belonging to assignmentId. We can also see a significant reduction from the Index lookup on g to just 1.81, which again is a significant reduction in the cost of using indexing.

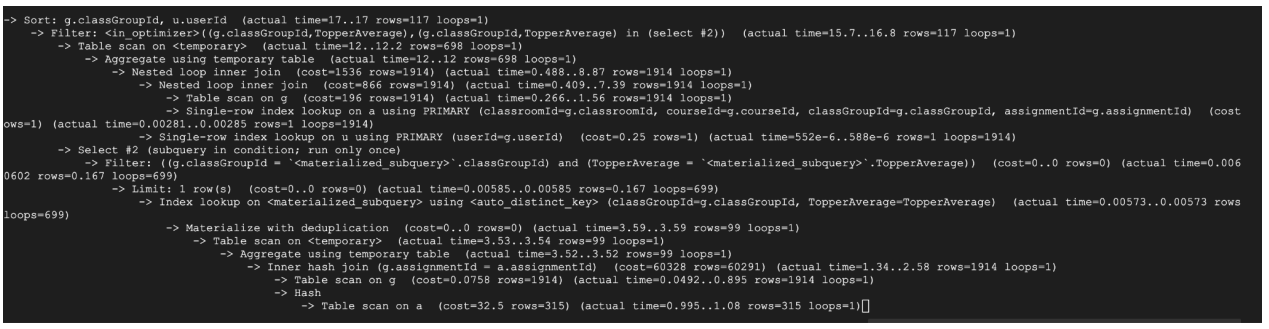
```
mysql> CREATE INDEX idx_assignment on Grades(assignmentId)
    -> ;
Query OK, 0 rows affected, 1 warning (0.10 sec)
Records: 0  Duplicates: 0  Warnings: 1
```



Indexing-2 (No cost reduction):

We performed indexing on the FirstName and LastName of the user to check if indexing would result in any cost reduction on attributes on which the database queries are not performing search but are just used for the SELECT clause. As predicted, there is absolutely no change in cost since we are not using these attributes in any way to derive any results, and are merely using to display the students names.





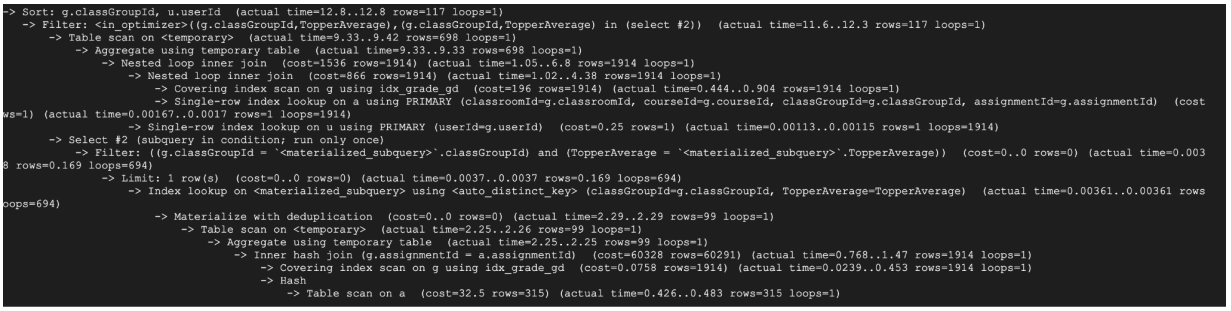
Indexing 3 - (No Cost Reduction):

We performed indexing on two columns namely grade for Grades table and maximumGrade on Assignment table. We hoped that there could be slight change in the cost due to the usage of both the attributes to compute the TopperAverage, which is used in the final computation as well. Since we used TopperAverage as a filtering function and not directly grade or maximum grade we did not receive any change in cost, however, we could observe such changes if we store the result of the last

subquery in a separate table and then index on TopperAverage, which would, however, be an additional table aside from our schema.

```
mysql> CREATE INDEX idx_grade_gd ON Grades(grade);
Query OK, 0 rows affected (0.16 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
mysql> CREATE INDEX idx_maxgrd ON Assignment(maximumGrade);
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



Modifications for Stage 2

Summary of stage 2 changes:

- We have changed the UML diagram. We have removed the student, teacher, and parent entities. Instead we have added userType in the user table to identify whether it is a student, parent or teacher.
- Accordingly, we have updated assumptions,cardinality, relational schema, UML and normalization.

Assumptions - Entities:

We are making an Online Learning system targeted at Primary-Secondary schools. New users are added to the Users table. This would contain user profile data on all users and their userType. We have a courses table which would have the information about the courses offered at the school (eg. English, Maths, Science) and a Classrooms table which has details of each classroom (eg. Grade 8-A, Grade 8-B). The ClassroomGroup table has all combinations of courses and classrooms. (eg. Grade 8-A English, Grade 8-A Math, Grade 8-B Science, etc.) So, the ClassroomGroup is a weak entity that depends on Courses and Classrooms for its existence. Assignments and ClassGroupRecordings contain the details of

assignments and recordings for all the fields in ClassroomGroup. They too are therefore weak entities, dependent on the ClassroomGroup. Each ClassroomGroup can contain multiple recordings and multiple assignments, since we can have multiple professors teaching the same subject but to different grades.
Note: We take into account the existence of multiple classrooms for the same school year (Grade) in scenarios where we have nearly 250 students enrolled in 8th Grade, with each room having a capacity of 50 students, so the 250 students are divided into 5 different houses or sections.

Why are these different entities?
Users, Classrooms, Courses, ClassroomGroup, Assignments, and ClassGroupRecordings all represent distinct concepts. By representing them as different entities we can accurately represent complex relationships and capture all relevant information about each entity. These entities can be translated as real-world objects that have a separate existence as compared to a relation or exist as attributes of a separate entity.
The following cardinality-related issues also indicate the need for a new entity:

- ClassGroupRecordings will have multiple recordings for a single ClassroomGroup
- Assignments will have multiple assignments for a single ClassroomGroup

Assumptions - Relations:
The Users table has teachers who grade the Assignments, so Grades is a relation between Users and Assignments. Each assignment will have a single grade. The Users use the ClassRoomGroup, which is defined as a relation called ClassroomUsers, which contains information about those users who are part of a Classroom(teachers and students). Users have attendance in Classrooms, so we have a relation Attendance to depict this. Each user will have multiple attendance records for different days, but only one record on one day.

Following is the assumed cardinality of all the relationships in our table:

1. ClassroomGroup - Courses -> It will have at least 1 course or can have any number of courses **(1..*)**

2. Courses - ClassroomGroup -> A newly added course will not be in the class group and multiple courses can be present in a class group **(0..*)**

3. ClassGroup - Classrooms -> A class group will have at least 1 classroom and can have many classrooms **(1..*)**

4. Classrooms - ClassGroup -> A classroom might not have been added to a ClassGroup on creation and can belong to multiple classGroup (12-a Math, 12-a Sci) **(0..*)**

5. Users - Classrooms -> A user might not belong to a classroom(parent) and might belong to multiple classrooms **(0..*)**

6. Classrooms - Users -> 1 Classroom may belong to many users (many students and teachers) **(1..*)**

7. Assignments - User -> One assignment can belong to multiple students (1..*)

8. User - Assignments -> One user can submit zero to many assignments (0..*)

9. ClassGroup - ClassGroupRecording -> A ClassGroup might not have a recording when the lecture is ongoing and might have multiple for multiple days (0..*)

10. ClassGroupRecording - ClassGroup -> 1 recording belongs to only 1 classGroup (1..1)

11. ClassGroup - Assignment -> A ClassGroup might not have an assignment when the group has just been created and might have multiple for multiple assignments (0..*)

12. Assignment - ClassGroup -> 1 assignment only for 1 classGroup (1..1)

Relational Schema

Users(userId: varchar(10) [PK], firstName: varchar(255), lastName: varchar(255), email: varchar(255), password: varchar(255), userType: Integer, createdAt: Date, address: varchar(255), dob: Date, occupation: varchar(255), studentIds: varchar(255));

Classrooms(classroomId: varchar(10) [PK], className: varchar(20), createdAt: Date)

Courses(courseId: varchar(10) [PK], subjectName: varchar(50), rating: Float)

ClassroomGroup(classGroupId: varchar(10), classroomId: varchar(10) [FK to Classroom.classRoomId], courseId: varchar(10) [FK to Courses.courseId], zoomLink: varchar(255), classStartTimings:varchar(50), classDuration: Float, primary key (classGroupId, classroomId, courseId))

Attendance(studentId: varchar(10) [FK to Users.userId], classroomId: varchar(10) [FK to Classroom.classroomId], attendanceDate: Date, isParentsNotified: Boolean, isPresent: Boolean, primary key(classroomId, studentId))

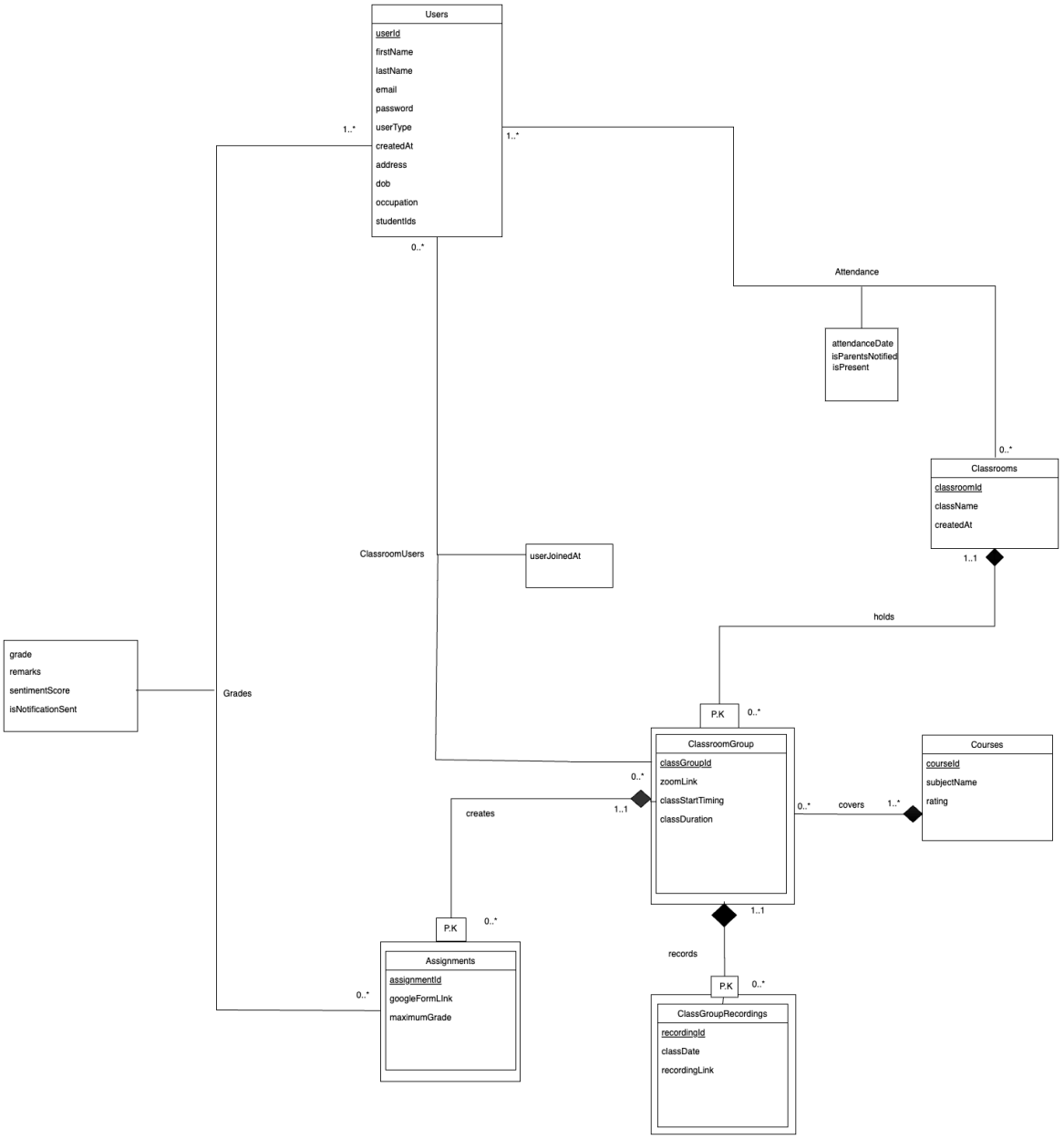
ClassroomUsers(userJoinedAt: Date, userId: varchar(10) [FK to Users.userId], classGroupId: varchar(10), classroomId: varchar(10), courseId: varchar(10), primary key(classroomId, courseId. classgroupId, userId), foreign key (classGroupId, classroomId, courseId) reference ClassroomGroup (classGroupId, classroomId, courseId))

Assignment(assignmentId: varchar(10), classGroupId: varchar(10), classroomId: varchar(10), courseId: varchar(10), foreign key (classGroupId, classroomId, courseId) reference ClassroomGroup (classGroupId, classroomId, courseId), primary key(classroomId, courseId, classGroupId, assignmentId), googleFormLink: varchar(255), maximumGrade: Float)

Grades(assignmentId: varchar(10), userId: varchar(10) [FK to Users.userId], classGroupId: varchar(10), classroomId: varchar(10), courseId: varchar(10), foreign key (assignmentId, classGroupId, classroomId, courseId) reference Assignment(assignmentId, classGroupId, classroomId, courseId), grade: Float, remarks: varchar(255), sentimentScore: INT, isNotificationSent: Boolean, primary key(userId, classroomId, courseId, classGroupId, assignmentId))

ClassGroupRecordings(recordingId: varchar(10), classGroupId: varchar(10), classroomId: varchar(10), courseId: varchar(10), foreign key (classGroupId, classroomId, courseId) reference ClassroomGroup (classGroupId, classroomId, courseId), classDate: Date, recordingLink: varchar(255), primary key (recordingId, classGroupId, classroomId, courseId))

UML



Normalization of Schema

We used 3NF over BCNF to normalize our databases because we opted for dependency preservation which BCNF does not have. Additionally, 3NF is more preferred for practical applications which do not demand the use of complex architecture and modern microservices, making the schema complex. Since, 3NF can establish a balance between the complexity of

our architecture and also remove redundancy, since all non-key attributes are dependent on the primary key, we choose 3NF.

Process of normalization for each entity:

- ***Users***
    - userId->firstName
    - userId->lastName
    - userId->email
    - userId->password
    - userId->userType
    - userId->createdAt
    - userId->address
    - userId->dob
    - userId->occupation
    - userId->studentIds

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.
    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**
        There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
    Created a table with userId as primary key containing firstName, lastName, email, password, userType, createdAt, address, dob, occupation, and studentIds.

3. **If none of the schemas from step 2 is a superkey then create a relation whose schema is a key for the original relation**
    The value of userId is a superkey.

- ***Classroom***
    - classRoomId->className
    - classRoomId->createdAt

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**

Every functional dependency only has a singleton in the right hand side.

b. **Remove unnecessary attributes in the left hand side.**
Every functional dependency has only necessary attributes in the left hand side.

c. **Remove FDs that can be inferred from the rest**
There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
Created a table with classRoomId as a primary key containing className and createdAt date.

3. **If none of the schemas from step 2 is a superkey then create a relation whos schema is a key for the original relation**
   a. The value of classRoomId is a superkey.

- ***Courses***
  - courseId->subjectName
  - courseId->rating

1. **Get Minimal Basis**
   a. **Only Singleton in RHS**
   Every functional dependency only has a singleton in the right hand side.
   b. **Remove unnecessary attributes in the left hand side.**
   Every functional dependency has only necessary attributes in the left hand side.

   c. **Remove FDs that can be inferred from the rest**
   There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
   a. Created a table with courseId as a primary key containing subjectName and rating.

3. **If none of the schemas from step 2 is a superkey then create a relation whose schema is a key for the original relation**
   a. The value of courseId is a superkey.

- ***ClassroomGroup***
  - classGroupId, classroomId, courseId->zoomLink
  - classGroupId, classroomId, courseId->classStartTimings

- classGroupId, classroomId, courseId->classDuration

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.
    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**
        There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
    Created a table with classGroupId, classroomId and courseId as a primary key containing zoomLink, classStartTimings and classDuration.

3. **If none of the schemas from step 2 is a superkey then create a relation whose schema is a key for the original relation**
    The value of classGroupId, classroomId, courseId is a superkey.

- ***Attendance***
    - classroomId, studentId->attendanceDate
    - classroomId, studentId->isParentsNotified
    - classroomId, studentId->IsPresent

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.
    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**
        There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
    Created a table with classroomId, studentId as a primary key containing attendanceDate, isParentsNotified, and isPresent.

3. **If none of the schemas from step 2 is a superkey then create a relation whos schema is a key for the original relation**
    The value of classroomId, studentId is a superkey.

- ***<u>ClassroomUsers</u>***
    - classroomId, courseId. classgroupId, userId->userJoinedAt

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.

    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**
        There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
    Created a table with classroomId, courseId. classgroupId, userId as a primary key containing userJoinedAt date.
3. **If none of the schemas from step 2 is a superkey then create a relation whos schema is a key for the original relation**
    The value of classroomId, courseId. classgroupId, userId is a superkey.

- ***<u>Assignment</u>***
    - classroomId, courseId, classGroupId, assignmentId->googleFormLink
    - classroomId, courseId, classGroupId, assignmentId->maximumGrade

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.
    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**
        There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
    Created a table with classroomId, courseId, classGroupId, assignmentId as a primary key containing googleFormLink and maximum grade.

3. **If none of the schemas from step 2 is a superkey then create a relation whos schema is a key for the original relation**
    The value of classroomId, courseId, classGroupId, assignmentId is a superkey.

- ***Grades***
    - userId, classroomId, courseId, classGroupId, assignmentId->grade
    - userId, classroomId, courseId, classGroupId, assignmentId->remarks
    - userId, classroomId, courseId, classGroupId, assignmentId->sentimentScore
    - userId, classroomId, courseId, classGroupId, assignmentId->isNotificationSent

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.
    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**
        There are no functional dependencies that can be inferred from the rest.
2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
    Created a table with userId, classroomId, courseId, classGroupId, assignmentId as a primary key containing grade, remarks, sentimentScore, and isNotificationSent.

3. **If none of the schemas from step 2 is a superkey then create a relation whos schema is a key for the original relation**
    The value of userId, classroomId, courseId, classGroupId, assignmentId is a superkey.

- ***ClassGroupRecordings***
    - recordingId, classGroupId, classroomId, courseId->classDate
    - recordingId, classGroupId, classroomId, courseId->recordingLink

1. **Get Minimal Basis**
    a. **Only Singleton in RHS**
        Every functional dependency only has a singleton in the right hand side.
    b. **Remove unnecessary attributes in the left hand side.**
        Every functional dependency has only necessary attributes in the left hand side.
    c. **Remove FDs that can be inferred from the rest**

There are no functional dependencies that can be inferred from the rest.

2. **For each FD A->B in minimal basis, use AB as the schema for a new relation**
   Created a table with recordingId, classGroupId, classroomId, courseId as a primary key containing classDate, and recordingLink.

3. **If none of the schemas from step 2 is a superkey then create a relation whose schema is a key for the original relation**
   The value of recordingId, classGroupId, classroomId, courseId is a superkey.