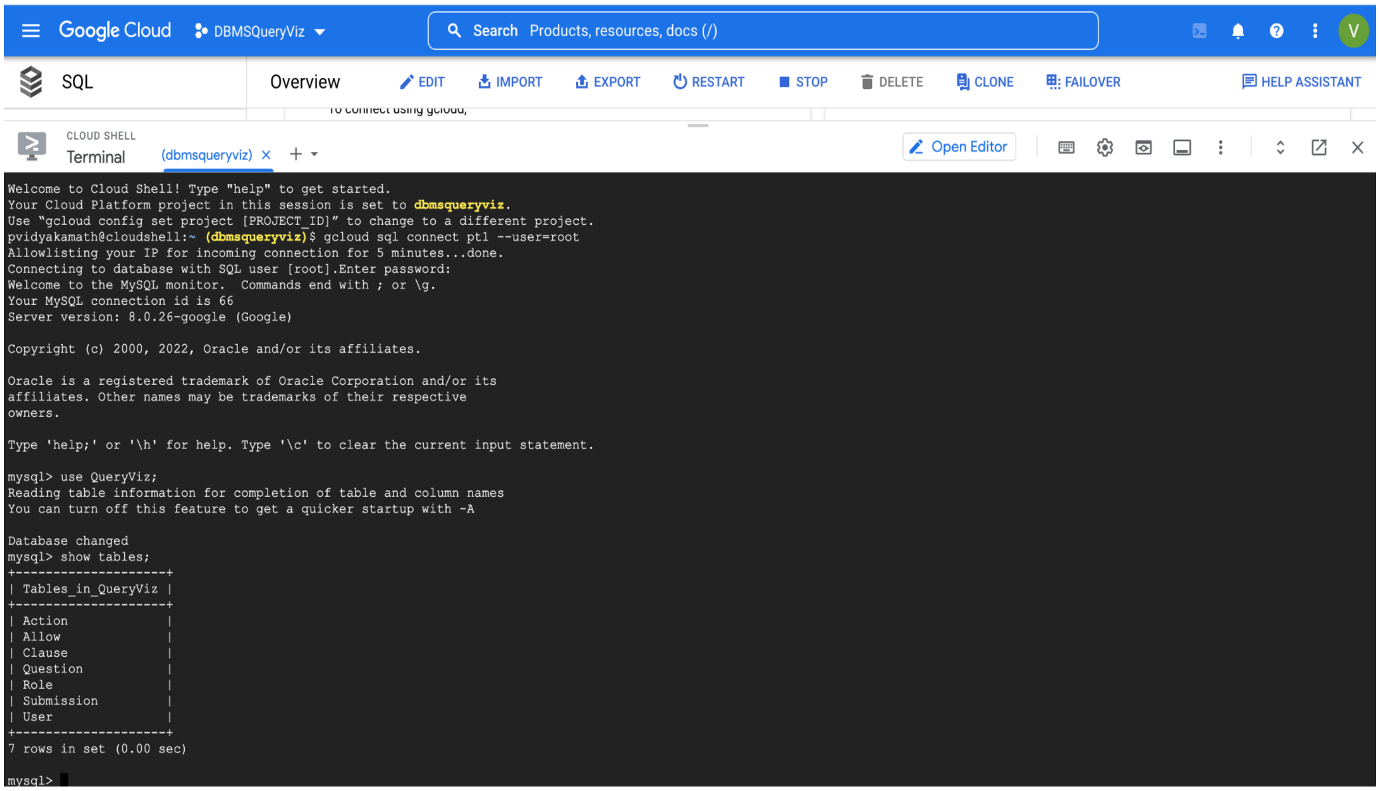
## Database Implementation

We are using MySql v8.0 Database on Google Cloud Platform.
**Database name: QueryViz**

## GCP Connection



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to dbmsqueryviz.
Use "gcloud config set project [PROJECT_ID]" to change to a different project.
pvidyakamath@cloudshell:~ (dbmsqueryviz)$ gcloud sql connect pt1 --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 66
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

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

Database changed
mysql> show tables;
+--------------------+
| Tables_in_QueryViz |
+--------------------+
| Action             |
| Allow              |
| Clause             |
| Question           |
| Role               |
| Submission         |
| User               |
+--------------------+
7 rows in set (0.00 sec)

mysql> 
```

## DDL Commands

CREATE TABLE Role (
ID int auto_increment primary key,
Name varchar(30) not null,
Description varchar(400) not null
);

CREATE TABLE Action (
·ID int auto_increment primary key,
Description varchar(400) not null
);

CREATE TABLE Allow (
RoleID int references Role(ID) on delete cascade on update cascade,
ActionID int references Action(ID) on delete cascade on update cascade,
primary key (RoleID, ActionID)
);

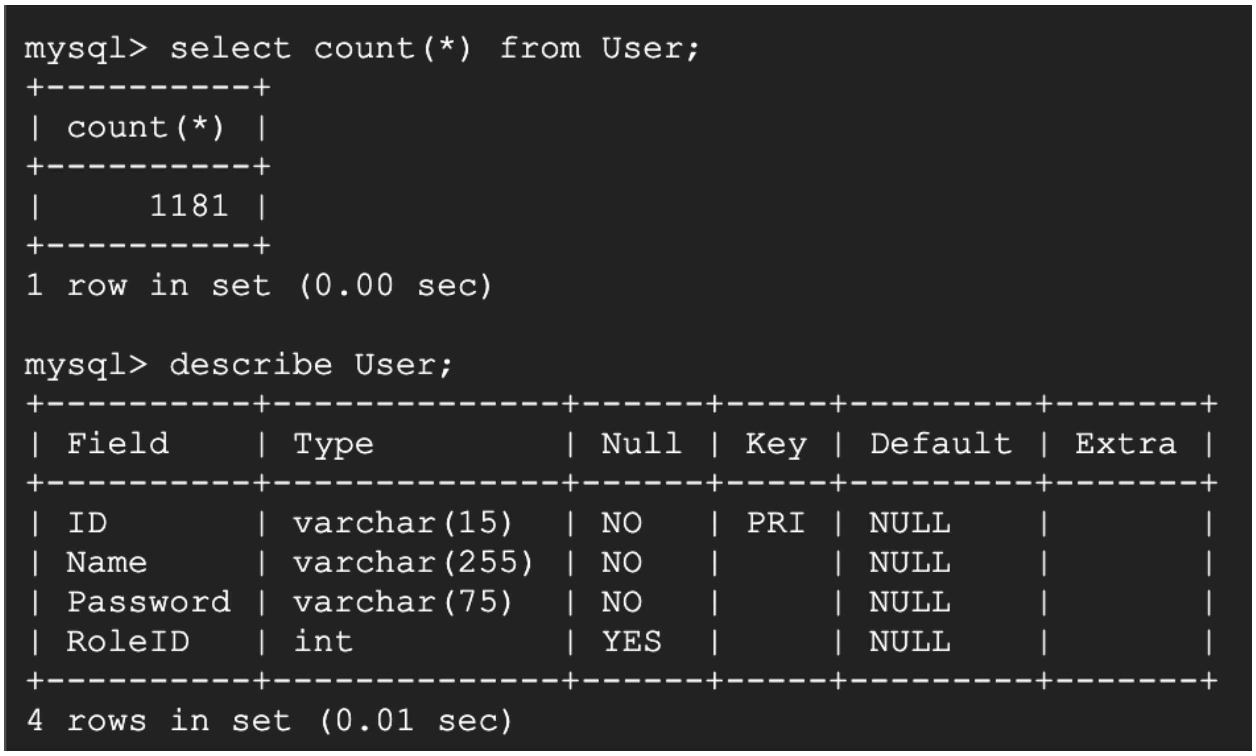
```
CREATE TABLE User (
ID varchar(15) primary key,
Name varchar(255) not null,
Password varchar(75) not null,
RoleID int references Role(ID) on delete cascade on update cascade
);

CREATE TABLE Question (
ID int auto_increment primary key,

Description varchar(700) not null,
Solution varchar(1500),
Difficulty enum(‘Easy’, ‘Medium’, ‘Hard’) not null
);

CREATE TABLE Submission (
ID int primary key,
Timestamp Datetime,
TotalExecutionTime real,
IsSuccess boolean,
SubmittedQuery varchar(1500),
QuestionID int references Question(ID) on delete cascade on update cascade,
UserID varchar(15) references User(ID) on delete cascade on update cascade
);

CREATE TABLE Clause (
ExecutionOrder int,
SubmissionID int references Submission(ID) on delete cascade on update cascade,
Type varchar(20) not null,
RelativeExecutionTime real,
Depth int,
Primary key (ExecutionOrder, SubmissionID)
);
```

## Data Records

We used csv files to insert data into each of the below tables. Anonymized student submissions across 10 SQL queries for the previous semester were kindly shared to us by the Instructor. These had over 20000 submissions. We did the following processing to obtain records for our tables:

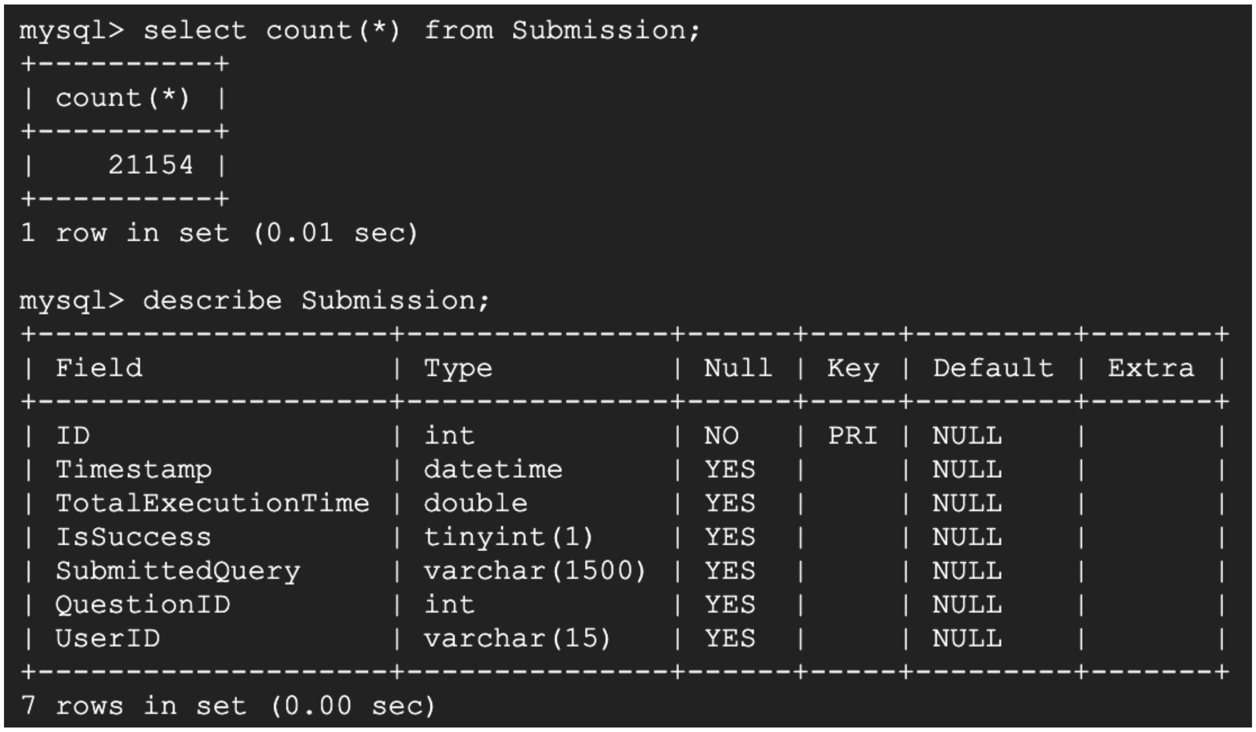
- Picked questions and submissions relevant to a particular schema, from the given dataset, as we are limiting checking of student solutions to one schema(Students, Courses and Enrollments) which will be static in the database.
- Worked out solutions on our own to these questions and added it to the Question table
- The submissions given to us had about 180 users. The remaining usernames and passwords were generated synthetically.
- The submitted query was also present in the submission file dump and this was processed with our query parser (still work in progress) to generate the clause types and few other metadata for the clauses table.

### User



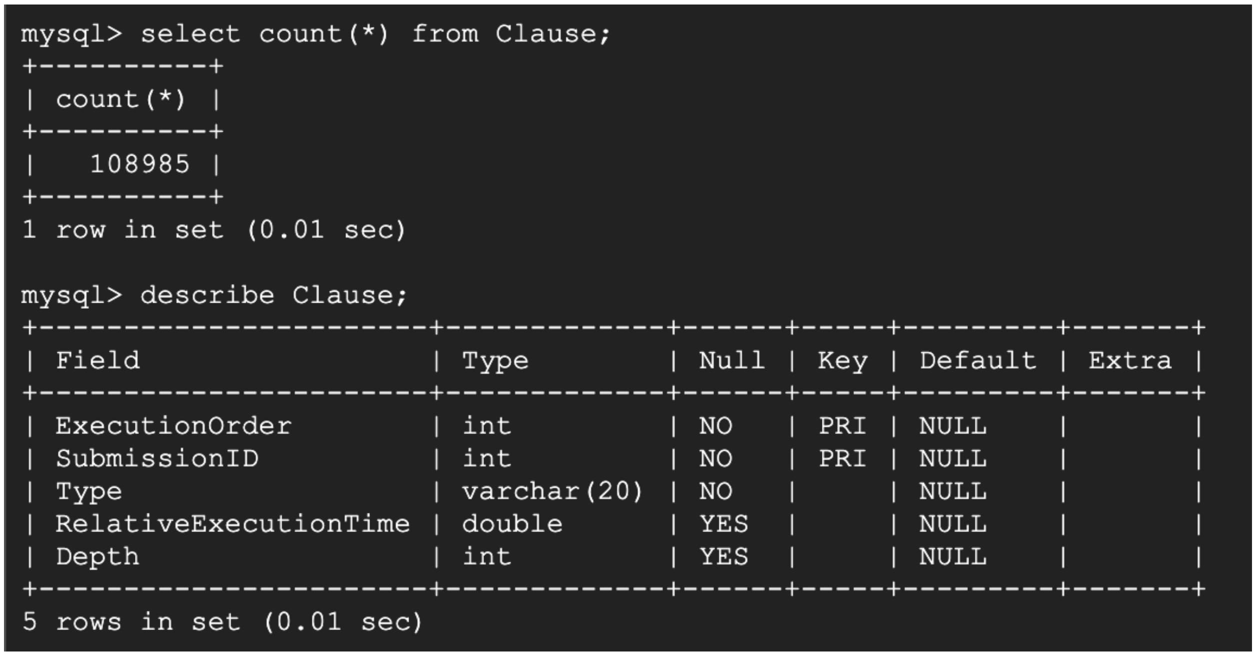
```
mysql> select count(*) from User;
+----------+
| count(*) |
+----------+
|     1181 |
+----------+
1 row in set (0.00 sec)

mysql> describe User;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| ID       | varchar(15)  | NO   | PRI | NULL    |       |
| Name     | varchar(255) | NO   |     | NULL    |       |
| Password | varchar(75)  | NO   |     | NULL    |       |
| RoleID   | int          | YES  |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
4 rows in set (0.01 sec)
```

**Submission**



```
mysql> select count(*) from Submission;
+----------+
| count(*) |
+----------+
|    21154 |
+----------+
1 row in set (0.01 sec)

mysql> describe Submission;
+--------------------+---------------+------+-----+---------+-------+
| Field              | Type          | Null | Key | Default | Extra |
+--------------------+---------------+------+-----+---------+-------+
| ID                 | int           | NO   | PRI | NULL    |       |
| Timestamp          | datetime      | YES  |     | NULL    |       |
| TotalExecutionTime | double        | YES  |     | NULL    |       |
| IsSuccess          | tinyint(1)    | YES  |     | NULL    |       |
| SubmittedQuery     | varchar(1500) | YES  |     | NULL    |       |
| QuestionID         | int           | YES  |     | NULL    |       |
| UserID             | varchar(15)   | YES  |     | NULL    |       |
+--------------------+---------------+------+-----+---------+-------+
7 rows in set (0.00 sec)
```

**Clause**



```
mysql> select count(*) from Clause;
+----------+
| count(*) |
+----------+
|   108985 |
+----------+
1 row in set (0.01 sec)

mysql> describe Clause;
+-----------------------+-------------+------+-----+---------+-------+
| Field                 | Type        | Null | Key | Default | Extra |
+-----------------------+-------------+------+-----+---------+-------+
| ExecutionOrder        | int         | NO   | PRI | NULL    |       |
| SubmissionID          | int         | NO   | PRI | NULL    |       |
| Type                  | varchar(20) | NO   |     | NULL    |       |
| RelativeExecutionTime | double      | YES  |     | NULL    |       |
| Depth                 | int         | YES  |     | NULL    |       |
+-----------------------+-------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

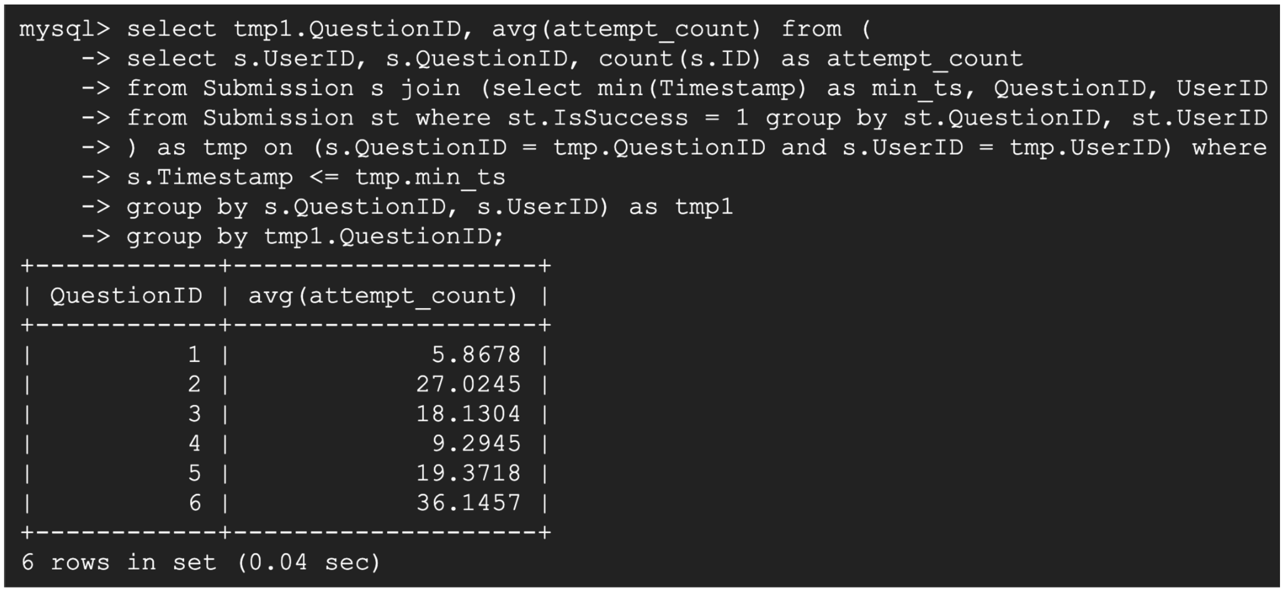
**Advanced Queries**

Usage: The below two advanced queries will be used to calculate the statistics of submissions displayed on the instructor dashboard (Refer Fig 5: Dashboard with metrics about Students' Performance in the Proposal document)

## 1. Average number of attempts before a successful submission for each question.

**SELECT** tmp1.QuestionID, AVG(attempt_count)
**FROM**
 (**SELECT** s.UserID,
  s.QuestionID,
  count(s.ID) AS attempt_count
 **FROM** Submission s
 **JOIN**
  (**SELECT** min(Timestamp) AS min_ts,
   QuestionID,
   UserID
  **FROM** Submission st
  **WHERE** st.IsSuccess = 1
  **GROUP BY** st.QuestionID,
   st.UserID) **AS** tmp **ON** (s.QuestionID = tmp.QuestionID **AND** s.UserID = tmp.UserID)
 **WHERE** s.Timestamp <= tmp.min_ts
 **GROUP BY** s.QuestionID,
  s.UserID) AS tmp1
**GROUP BY** tmp1.QuestionID;

- Screenshot showing the result of the query



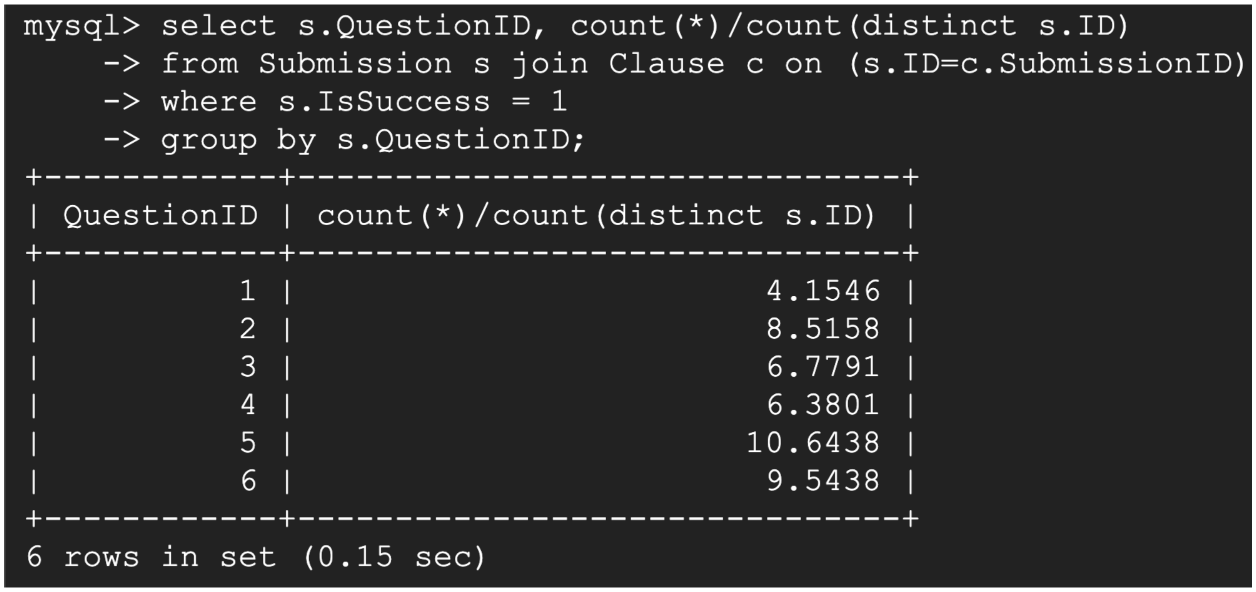
```
mysql> select tmp1.QuestionID, avg(attempt_count) from (
    -> select s.UserID, s.QuestionID, count(s.ID) as attempt_count
    -> from Submission s join (select min(Timestamp) as min_ts, QuestionID, UserID
    -> from Submission st where st.IsSuccess = 1 group by st.QuestionID, st.UserID
    -> ) as tmp on (s.QuestionID = tmp.QuestionID and s.UserID = tmp.UserID) where
    -> s.Timestamp <= tmp.min_ts
    -> group by s.QuestionID, s.UserID) as tmp1
    -> group by tmp1.QuestionID;
+------------+--------------------+
| QuestionID | avg(attempt_count) |
+------------+--------------------+
|          1 |             5.8678 |
|          2 |            27.0245 |
|          3 |            18.1304 |
|          4 |             9.2945 |
|          5 |            19.3718 |
|          6 |            36.1457 |
+------------+--------------------+
6 rows in set (0.04 sec)
```

## 2. Average number of clauses executed for successful submission for each question.

**SELECT** s.QuestionID,
 **COUNT**(*)/**COUNT**(**DISTINCT** s.ID)
**FROM** Submission s
**JOIN** Clause c **ON** (s.ID=c.SubmissionID)

**WHERE** s.IsSuccess = 1
**GROUP BY** s.QuestionID;

- Screenshot showing the result of the query



```
mysql> select s.QuestionID, count(*)/count(distinct s.ID)
    -> from Submission s join Clause c on (s.ID=c.SubmissionID)
    -> where s.IsSuccess = 1
    -> group by s.QuestionID;
+------------+-------------------------------+
| QuestionID | count(*)/count(distinct s.ID) |
+------------+-------------------------------+
|          1 |                        4.1546 |
|          2 |                        8.5158 |
|          3 |                        6.7791 |
|          4 |                        6.3801 |
|          5 |                       10.6438 |
|          6 |                        9.5438 |
+------------+-------------------------------+
6 rows in set (0.15 sec)
```

## Indexing Analysis Reports

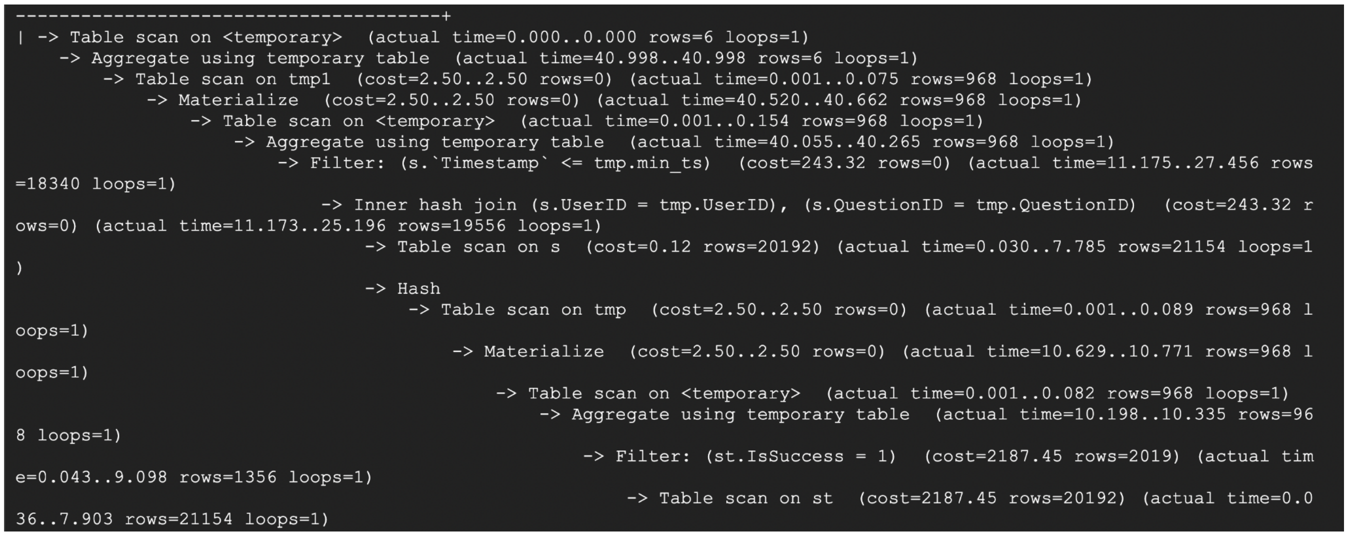
### Advanced Query 1

**SELECT** tmp1.QuestionID, AVG(attempt_count) **— 1**
**FROM**
 (**SELECT** s.UserID,
  s.QuestionID,
  count(s.ID) AS attempt_count **—2**
 **FROM** Submission s
 **JOIN — 3**
  (**SELECT** min(Timestamp) AS min_ts,
   QuestionID,
   UserID **— 4**
  **FROM** Submission st
  **WHERE** st.IsSuccess = 1 **— 5**
  **GROUP BY** st.QuestionID, **— 6**
   st.UserID) **AS** tmp **ON** (s.QuestionID = tmp.QuestionID **AND** s.UserID = tmp.UserID)
 **WHERE** s.Timestamp <= tmp.min_ts **— 7**
 **GROUP BY** s.QuestionID,
  s.UserID **— 8**
) AS tmp1

**GROUP BY** tmp1.QuestionID; **– 9**

**Indexing analysis**

**Actual Time Units: 40.998**



```
----------------------------------------+
| -> Table scan on <temporary>  (actual time=0.000..0.000 rows=6 loops=1)
    -> Aggregate using temporary table  (actual time=40.998..40.998 rows=6 loops=1)
        -> Table scan on tmp1  (cost=2.50..2.50 rows=0) (actual time=0.001..0.075 rows=968 loops=1)
            -> Materialize  (cost=2.50..2.50 rows=0) (actual time=40.520..40.662 rows=968 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.154 rows=968 loops=1)
                    -> Aggregate using temporary table  (actual time=40.055..40.265 rows=968 loops=1)
                        -> Filter: (s.`Timestamp` <= tmp.min_ts)  (cost=243.32 rows=0) (actual time=11.175..27.456 rows
=18340 loops=1)
                            -> Inner hash join (s.UserID = tmp.UserID), (s.QuestionID = tmp.QuestionID)  (cost=243.32 r
ows=0) (actual time=11.173..25.196 rows=19556 loops=1)
                                -> Table scan on s  (cost=0.12 rows=20192) (actual time=0.030..7.785 rows=21154 loops=1
)
                                -> Hash
                                    -> Table scan on tmp  (cost=2.50..2.50 rows=0) (actual time=0.001..0.089 rows=968 l
oops=1)
                                        -> Materialize  (cost=2.50..2.50 rows=0) (actual time=10.629..10.771 rows=968 l
oops=1)
                                            -> Table scan on <temporary>  (actual time=0.001..0.082 rows=968 loops=1)
                                                -> Aggregate using temporary table  (actual time=10.198..10.335 rows=96
8 loops=1)
                                                    -> Filter: (st.IsSuccess = 1)  (cost=2187.45 rows=2019) (actual tim
e=0.043..9.098 rows=1356 loops=1)
                                                        -> Table scan on st  (cost=2187.45 rows=20192) (actual time=0.0
36..7.903 rows=21154 loops=1)
```

To be able to optimize the query, we firstly looked at the way the query planner handles it.
For this query we see that the sequence of steps followed by the query planner was as follows:

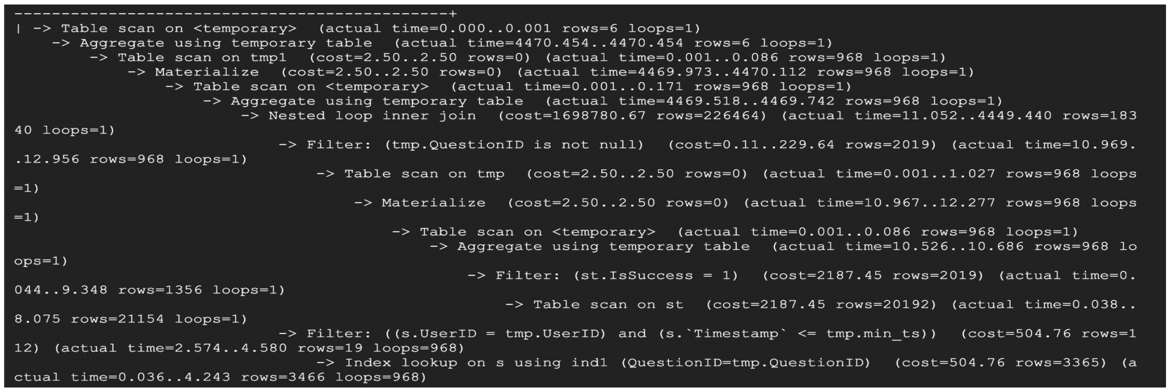
1. Table scan on submissions(aliased as st) along with filtering of IsSuccess column. Corresponds to statement 5.
2. Sorting to form groups based on QuestionID and UserID, aggregating for required values within groups(corresponding to statement 4 and 6).
3. Materializing the temporary table tmp
4. Scan the tmp table to form a hash on userId, QuestionID
5. Scan submissions table to use for the join along with the hash created before on userId, QuestionID
6. The inner hash join step joins the two tables using the hash and simultaneously performs filtering on the timestamp column(the join corresponds to statement 3 and the filtering corresponds to statement 7)
7. Sorting and grouping based on userID and QuestionID(statement 8) and aggregating to get values as required in statement 2.
8. Materializing this as tmp1 table
9. Sorting and grouping based on QuestionID(statement 9) and aggregating to get values as required in statement 1.

Seeing that there is a self join based on QuestionID and UserID and a group by based on these fields as well as QuestionID alone, it would be valuable to explore these fields as indices.

**Option 1: Creating index on QuestionID**
**Time Units: 4470.454**

The index is reflected in the query plan, as we see that the hash join is replaced by a nested loop inner join and for equality on the QuestionID field, ind1 is used for index lookup. However the overall cost of the hash join was 243, but just the index lookup cost for QuestionID(along with filtering of equal userId) is 504. Further, the nested loop join is the most costly operation (1.69 x $10^6$) and leads to an extremely large execution time. This is the reason that this index did not work well and the overall time was **4470.454**

CREATE INDEX ind1 ON Submission(QuestionID);



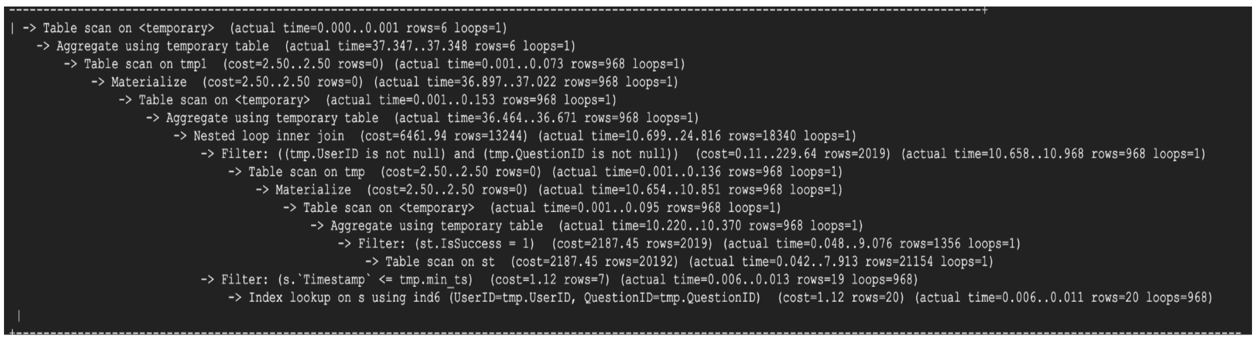
```
-------------------------------------------+
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=6 loops=1)
    -> Aggregate using temporary table  (actual time=4470.454..4470.454 rows=6 loops=1)
        -> Table scan on tmp1  (cost=2.50..2.50 rows=0) (actual time=0.001..0.086 rows=968 loops=1)
            -> Materialize  (cost=2.50..2.50 rows=0) (actual time=4469.973..4470.112 rows=968 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.171 rows=968 loops=1)
                    -> Aggregate using temporary table  (actual time=4469.518..4469.742 rows=968 loops=1)
                        -> Nested loop inner join  (cost=1698780.67 rows=226464) (actual time=11.052..4449.440 rows=183
40 loops=1)
                            -> Filter: (tmp.QuestionID is not null)  (cost=0.11..229.64 rows=2019) (actual time=10.969.
.12.956 rows=968 loops=1)
                                -> Table scan on tmp  (cost=2.50..2.50 rows=0) (actual time=0.001..1.027 rows=968 loops
=1)
                                    -> Materialize  (cost=2.50..2.50 rows=0) (actual time=10.967..12.277 rows=968 loops
=1)
                                        -> Table scan on <temporary>  (actual time=0.001..0.086 rows=968 loops=1)
                                            -> Aggregate using temporary table  (actual time=10.526..10.686 rows=968 lo
ops=1)
                                                -> Filter: (st.IsSuccess = 1)  (cost=2187.45 rows=2019) (actual time=0.
044..9.348 rows=1356 loops=1)
                                                    -> Table scan on st  (cost=2187.45 rows=20192) (actual time=0.038..
8.075 rows=21154 loops=1)
                            -> Filter: ((s.UserID = tmp.UserID) and (s.`Timestamp` <= tmp.min_ts))  (cost=504.76 rows=1
12) (actual time=2.574..4.580 rows=19 loops=968)
                                -> Index lookup on s using ind1 (QuestionID=tmp.QuestionID)  (cost=504.76 rows=3365) (a
ctual time=0.036..4.243 rows=3466 loops=968)
```

We wanted to explore if the cost of Nested loop inner join would decrease below that of the hash join if we added an index on UserId as well, along with QuestionID. This is because, in the current case we see that for each record of the outer table, indexing is done to get all matching records with the same QuestionID and then there is record by record filtering of matching UserID.

**Option 2: Create a single index on (QuestionID,UserID)**
**Time Units: 37.348**

Here we observe that, during the JOIN operation(statement 3) both the indices are used and the index lookup only gets 19 relevant records as opposed to option 1(only index QuestionID) where 3000+ rows were looked at. The overall filtering time hence drops from 9.098 to 0.013. The inner join time also drops from 25.196 to 24.816. This leads to overall time reduction

CREATE INDEX ind6 ON Submission(UserID, QuestionID);

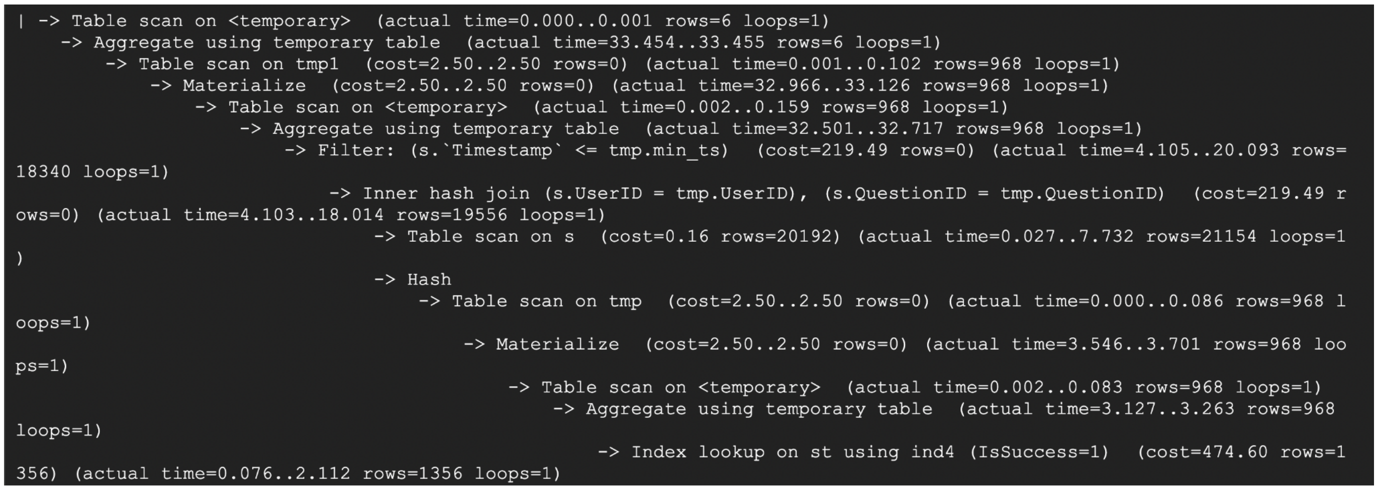


```
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=6 loops=1)
    -> Aggregate using temporary table  (actual time=37.347..37.348 rows=6 loops=1)
        -> Table scan on tmp1  (cost=2.50..2.50 rows=0) (actual time=0.001..0.073 rows=968 loops=1)
            -> Materialize  (cost=2.50..2.50 rows=0) (actual time=36.897..37.022 rows=968 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.153 rows=968 loops=1)
                    -> Aggregate using temporary table  (actual time=36.464..36.671 rows=968 loops=1)
                        -> Nested loop inner join  (cost=6461.94 rows=13244) (actual time=10.699..24.816 rows=18340 loops=1)
                            -> Filter: ((tmp.UserID is not null) and (tmp.QuestionID is not null))  (cost=0.11..229.64 rows=2019) (actual time=10.658..10.968 rows=968 loops=1)
                                -> Table scan on tmp  (cost=2.50..2.50 rows=0) (actual time=0.001..0.136 rows=968 loops=1)
                                    -> Materialize  (cost=2.50..2.50 rows=0) (actual time=10.654..10.851 rows=968 loops=1)
                                        -> Table scan on <temporary>  (actual time=0.001..0.095 rows=968 loops=1)
                                            -> Aggregate using temporary table  (actual time=10.220..10.370 rows=968 loops=1)
                                                -> Filter: (st.IsSuccess = 1)  (cost=2187.45 rows=2019) (actual time=0.048..9.076 rows=1356 loops=1)
                                                    -> Table scan on st  (cost=2187.45 rows=20192) (actual time=0.042..7.913 rows=21154 loops=1)
                            -> Filter: (s.`Timestamp` <= tmp.min_ts)  (cost=1.12 rows=7) (actual time=0.006..0.013 rows=19 loops=968)
                                -> Index lookup on s using ind6 (UserID=tmp.UserID, QuestionID=tmp.QuestionID)  (cost=1.12 rows=20) (actual time=0.006..0.011 rows=20 loops=968)
 |
```

**Option 3: Creating a single index on IsSuccess**
**Time Units: 33.455**

In the query, we have a filtering based on the field IsSuccess(statement 5). We wanted to have an index on this column so that all submission records need not be checked. Given that it is a binary column we felt that the search space reduction would be close to half. Also since we are not placing indices on the columns of the JOIN, the hash based join is used. We can compare this performance with the original query. The actual execution time has improved from 40.998 to 33.454. As anticipated the index on IsSuccess prevented a table scan thereby dropping the cost from 2187.45 to 474.6, also it had to only scan 1356 rows as opposed to the 20192. This caused a time improvement from 10.198 to 3.127 in for all steps that were post the aggregate step (statement 4). The rest of the steps remain unchanged from the original query plan and hence the improvement of 7 time units is understood. (33.455)

CREATE INDEX ind4 ON Submission(IsSuccess);

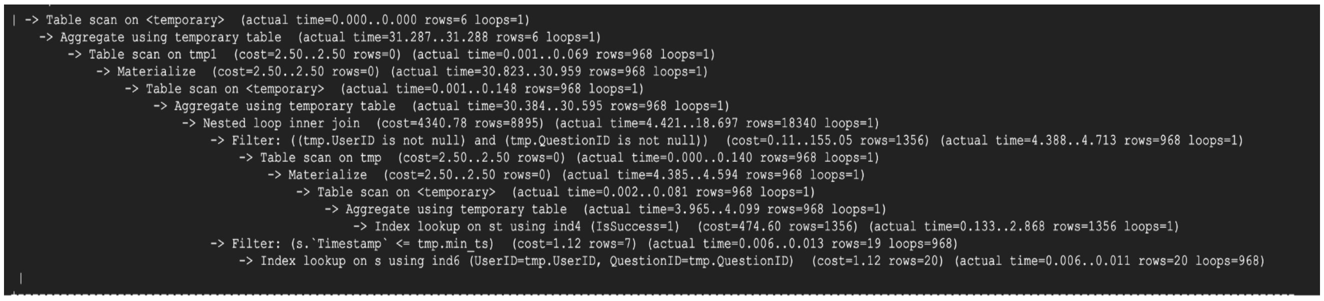


```
| -> Table scan on <temporary>  (actual time=0.000..0.001 rows=6 loops=1)
    -> Aggregate using temporary table  (actual time=33.454..33.455 rows=6 loops=1)
        -> Table scan on tmp1  (cost=2.50..2.50 rows=0) (actual time=0.001..0.102 rows=968 loops=1)
            -> Materialize  (cost=2.50..2.50 rows=0) (actual time=32.966..33.126 rows=968 loops=1)
                -> Table scan on <temporary>  (actual time=0.002..0.159 rows=968 loops=1)
                    -> Aggregate using temporary table  (actual time=32.501..32.717 rows=968 loops=1)
                        -> Filter: (s.`Timestamp` <= tmp.min_ts)  (cost=219.49 rows=0) (actual time=4.105..20.093 rows=
18340 loops=1)
                            -> Inner hash join (s.UserID = tmp.UserID), (s.QuestionID = tmp.QuestionID)  (cost=219.49 r
ows=0) (actual time=4.103..18.014 rows=19556 loops=1)
                                -> Table scan on s  (cost=0.16 rows=20192) (actual time=0.027..7.732 rows=21154 loops=1
)
                                -> Hash
                                    -> Table scan on tmp  (cost=2.50..2.50 rows=0) (actual time=0.000..0.086 rows=968 l
oops=1)
                                        -> Materialize  (cost=2.50..2.50 rows=0) (actual time=3.546..3.701 rows=968 loo
ps=1)
                                            -> Table scan on <temporary>  (actual time=0.002..0.083 rows=968 loops=1)
                                                -> Aggregate using temporary table  (actual time=3.127..3.263 rows=968
loops=1)
                                                    -> Index lookup on st using ind4 (IsSuccess=1)  (cost=474.60 rows=1
356) (actual time=0.076..2.112 rows=1356 loops=1)
```

Now we see the benefit of having Option 2,3. We wanted to check if we can have indices on both these columns to achieve a higher speedup. Note: We cannot combine them as one index because for IsSuccess we care about filtering in the entire search space. If we have it as a single index(QuestionID,UserID), then only when QuestionID and UserID are the same for multiple submissions, they are sorted based on IsSuccess.

**Option 4: Creating 2 indices. One on (QuestionID,UserID) and another on IsSuccess.**
**Time Units: 31.288**

We observe that both indices get used. IsSuccess is used for the innermost filter in statement 5 and the other index is used for the join condition in statement 7. The query plan shows the usage of **IsSuccess** by **ind4** and a composite index **(QuestionID,UserID)** by **ind6**. The overall effect is drop in nested loop/join time from 25 to 18 units as opposed to the original query. This leads to an overall improvement of about 8 units.



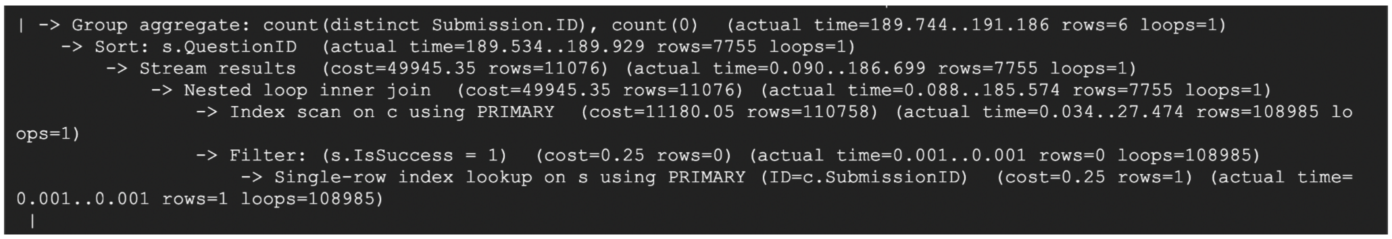
```
| -> Table scan on <temporary>  (actual time=0.000..0.000 rows=6 loops=1)
    -> Aggregate using temporary table  (actual time=31.287..31.288 rows=6 loops=1)
        -> Table scan on tmp1  (cost=2.50..2.50 rows=0) (actual time=0.001..0.069 rows=968 loops=1)
            -> Materialize  (cost=2.50..2.50 rows=0) (actual time=30.823..30.959 rows=968 loops=1)
                -> Table scan on <temporary>  (actual time=0.001..0.148 rows=968 loops=1)
                    -> Aggregate using temporary table  (actual time=30.384..30.595 rows=968 loops=1)
                        -> Nested loop inner join  (cost=4340.78 rows=8895) (actual time=4.421..18.697 rows=18340 loops=1)
                            -> Filter: ((tmp.UserID is not null) and (tmp.QuestionID is not null))  (cost=0.11..155.05 rows=1356) (actual time=4.388..4.713 rows=968 loops=1)
                                -> Table scan on tmp  (cost=2.50..2.50 rows=0) (actual time=0.000..0.140 rows=968 loops=1)
                                    -> Materialize  (cost=2.50..2.50 rows=0) (actual time=4.385..4.594 rows=968 loops=1)
                                        -> Table scan on <temporary>  (actual time=0.002..0.081 rows=968 loops=1)
                                            -> Aggregate using temporary table  (actual time=3.965..4.099 rows=968 loops=1)
                                                -> Index lookup on st using ind4 (IsSuccess=1)  (cost=474.60 rows=1356) (actual time=0.133..2.868 rows=1356 loops=1)
                            -> Filter: (s.`Timestamp` <= tmp.min_ts)  (cost=1.12 rows=7) (actual time=0.006..0.013 rows=19 loops=968)
                                -> Index lookup on s using ind6 (UserID=tmp.UserID, QuestionID=tmp.QuestionID)  (cost=1.12 rows=20) (actual time=0.006..0.011 rows=20 loops=968)
 |
```

**Advanced Query 2**

```
SELECT s.QuestionID,
    COUNT(*)/COUNT(DISTINCT s.ID)
FROM Submission s
JOIN Clause c ON (s.ID=c.SubmissionID)
WHERE s.IsSuccess = 1
GROUP BY s.QuestionID;
```

**Indexing analysis**

**Actual time units**: **189.534**



```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=189.744..191.186 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=189.534..189.929 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=11076) (actual time=0.090..186.699 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=11076) (actual time=0.088..185.574 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.034..27.474 rows=108985 lo
ops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=
0.001..0.001 rows=1 loops=108985)
 |
```

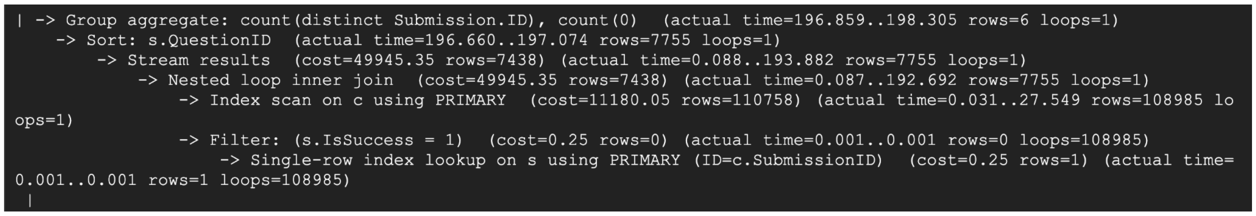
For the original query we see that the following is the query plan:

1. For each submission ID(obtained from primary index as SubmissionID is part of the primary key for the clauses table and hence has a default index) in the clauses table, go to records in the submissions table with the same submission Id(done with indices as SubmissionID is part of the primary key in clause table). Filter records in this bucket where IsSuccess = 1 and merge them with the record from the clauses table.
2. Sort the records based on QuestionID and group them
3. Find the required aggregate functions in each group.

Since there is a filtering based on the IsSuccess field, we tried to add an index on it.

**Option 1: Create a single index on Submission.IsSuccess field**
**Time units: 196.074**

CREATE INDEX ind2 ON Submission(IsSuccess);



```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=196.859..198.305 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=196.660..197.074 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=7438) (actual time=0.088..193.882 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=7438) (actual time=0.087..192.692 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.031..27.549 rows=108985 lo
ops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=
0.001..0.001 rows=1 loops=108985)
 |
```

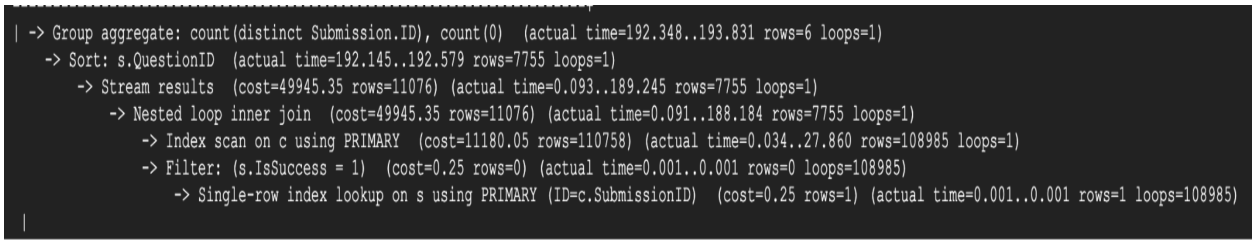
It causes the performance to become worse. It can be seen from above that the index is never used in the query plan. The reason is that for a particular submissionId, there is only 1 record in the submission table which we directly get using the index on submissionId. The IsSuccess field can be checked for this single record without the need to look at any index. Hence there is no value in adding this index and leads to more overhead.

We observed that there is a grouping based on QuestionID and added an index to optimize this.

**Option 2: Create a single index on Submission.QuestionID field**
**Time units: 192.579**

There is an increase of 4 units of time. On comparing with the original query we see that the upto the stream results/Join step they have the same cost, indicating that the overhead appears after that, which is in the sorting based on QuestionID. We see that the table which is getting sorted is a combined table of clauses and submissions but the index we have is only on QuestionID of submissions and so it is not being used anywhere and is an overhead.

CREATE INDEX ind4 ON Submission(QuestionID)



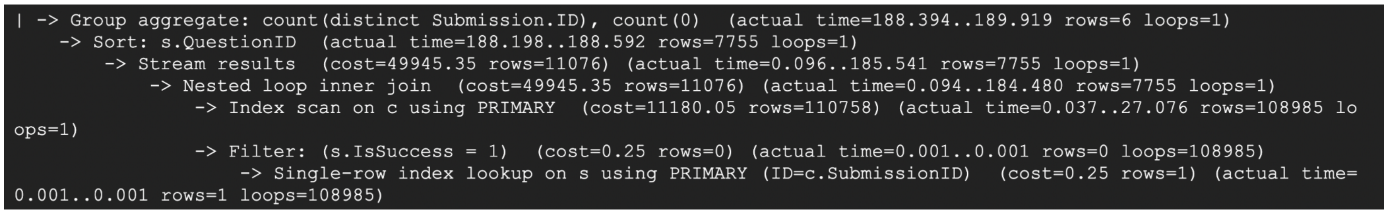
```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=192.348..193.831 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=192.145..192.579 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=11076) (actual time=0.093..189.245 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=11076) (actual time=0.091..188.184 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.034..27.860 rows=108985 loops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=108985)
 |
```

We wanted to add an index on submissionId as it is being used in the aggregate function with a distinct. An index will be able to just count the number of buckets/pointers to get this. However, we already have a Primary index (due to primary key) for this field. We also observed that the columns used for the join i.e SubmissionID of both tables are already primary keys and hence the join is optimized fully. Therefore, we tried several other options, to ensure that we have not missed any possible optimisations, but came to the conclusion that the necessary indices are in place and adding more will only be an overhead for the SQL compiler.

Below are the results of adding different indices:

**Option 3: Create a single index on Submission.UserID field**
**Time units: 188.592**

CREATE INDEX ind1 ON Submission(UserID);

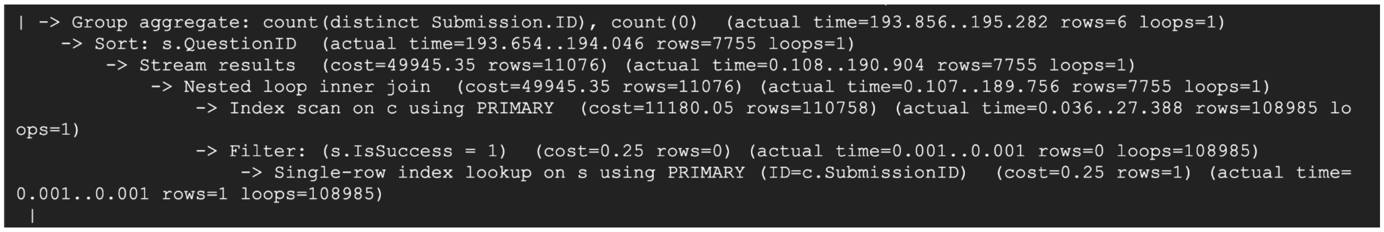


```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=188.394..189.919 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=188.198..188.592 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=11076) (actual time=0.096..185.541 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=11076) (actual time=0.094..184.480 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.037..27.076 rows=108985 lo
ops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=
0.001..0.001 rows=1 loops=108985)
```

**Option 4: Create a single index on (Submission.UserID, Submission.IsSuccess) field**

**Time units: 193.046**

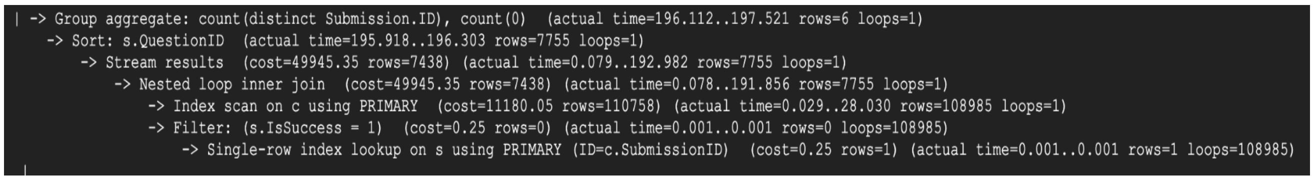
CREATE INDEX ind3 ON Submission(UserID, IsSuccess);



```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=193.856..195.282 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=193.654..194.046 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=11076) (actual time=0.108..190.904 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=11076) (actual time=0.107..189.756 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.036..27.388 rows=108985 lo
ops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=
0.001..0.001 rows=1 loops=108985)
 |
```

**Option 5: Create a single index on (Submission.QuestionID, Submission.IsSuccess) field**
**Time units: 196.303**

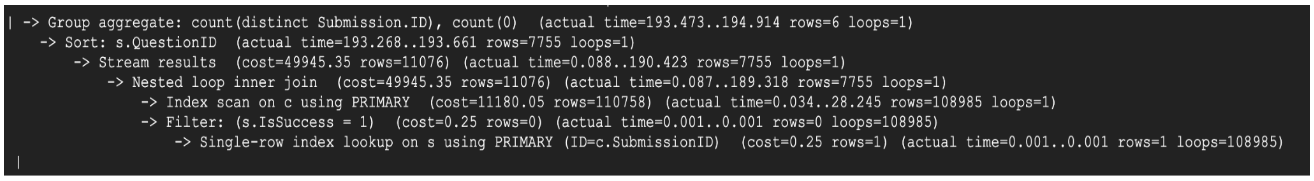
CREATE INDEX ind5 ON Submission(QuestionID, IsSuccess);



```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=196.112..197.521 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=195.918..196.303 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=7438) (actual time=0.079..192.982 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=7438) (actual time=0.078..191.856 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.029..28.030 rows=108985 loops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=108985)
 |
```

**Option 6: Create a single index on (Submission.QuestionID, Submission.ID) field**
**Time units: 193.661**

CREATE INDEX ind6 ON Submission(ID, QuestionID); = 193.661



```
| -> Group aggregate: count(distinct Submission.ID), count(0)  (actual time=193.473..194.914 rows=6 loops=1)
    -> Sort: s.QuestionID  (actual time=193.268..193.661 rows=7755 loops=1)
        -> Stream results  (cost=49945.35 rows=11076) (actual time=0.088..190.423 rows=7755 loops=1)
            -> Nested loop inner join  (cost=49945.35 rows=11076) (actual time=0.087..189.318 rows=7755 loops=1)
                -> Index scan on c using PRIMARY  (cost=11180.05 rows=110758) (actual time=0.034..28.245 rows=108985 loops=1)
                -> Filter: (s.IsSuccess = 1)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=108985)
                    -> Single-row index lookup on s using PRIMARY (ID=c.SubmissionID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=108985)
 |
```