Alumni Database Management System

Responsibility of Group1

We revisited the ER Diagram and Relational Schemas of Assignment 1 and on the basis of feedback from the last assignment, we modified the ER Diagram.

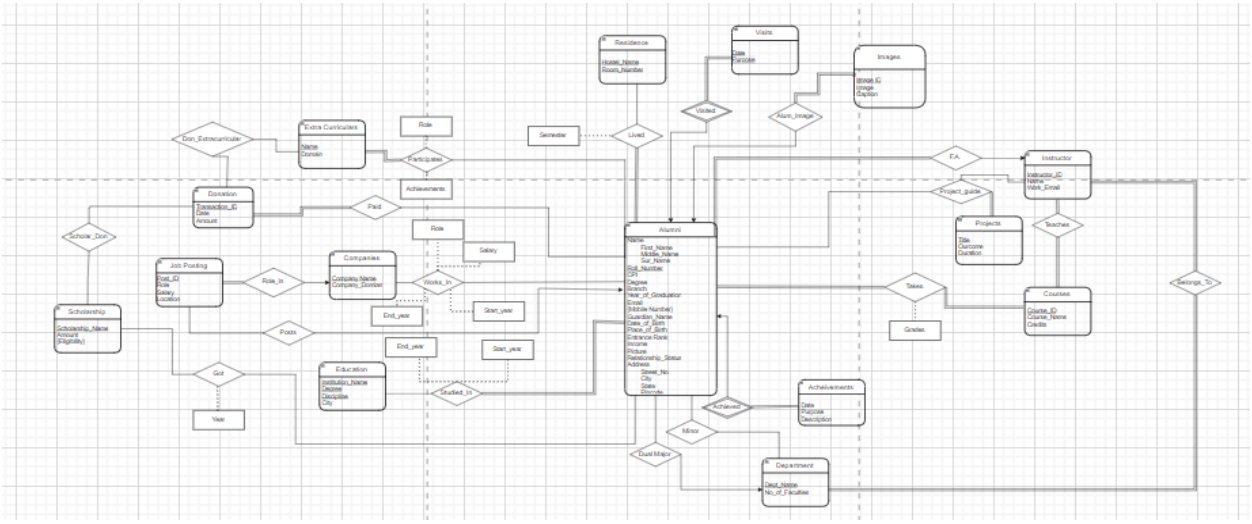
Changes done in the ER Diagram on suggestions from TA:
JEE Rank attribute for the alumni was changed to the Entrance Rank because a student might get enrolled via different examinations and not necessarily through JEE.
Donations from Alumni were changed from total to partial. Earlier we had considered donation as total from the alumni side because graduating students have to pay a certain amount to be enrolled in the alumni network. However, we were told to not consider that.

Education was changed from total to partial for the relationship studied because TA said that we can also incorporate top institutions which didn't have any alumni studying in the college in the education entity relation.

The last change that was done was to include the Image entity in our relation table which will have different images for different alumni that might be relevant to the alumni database. For example, batch photos can be incorporated in our database. This relationship is many many because a single image might be linked to multiple alumni and an alumni might have multiple relevant images in the database. It's also a total participation from the Image side because each image must be associated with at least one alumni.

Entity-Relationship Diagram

Link to the draw.io project: Assignment-1_ER_Diagram.drawio



Population of Tables with Data

Population of the tables with Data was one of the most crucial parts. For generating data, we have used Faker library and Random library in python to generate different data for different entities. We used these libraries to generate the fake data for our tables and then wrote it in a csv file. This csv file was then later imported in mySQL workbench to populate the tables written in mySQL. It was a difficult task because we had to ensure that we take care of different types of constraints while populating the tables. We had to take care that primary keys didn't get repeated for any tables, total/partial/one-one/many-one/one-many/one-one participation was taken care of and appropriate data types were used to fill the table. We created different scripts for each table and filled the entity tables at first. After getting done with it, we used the obtained entity tables to fill the relationship tables (in csv) keeping in mind the different constraints. Finally all these tables in CSV files were imported in mySQL.

Google Drive Link of CSV files and SQL dump

Indexing

We have used indexing in a few entity and relationship tables wherever we thought that search could be optimized. However, it's worth noting that indexing has some trade-offs, such as increased storage space and decreased write performance (since indexes must be updated whenever the indexed columns are modified).

1) **Alumni:** Separate indexing: {Branch, Degree} and {Roll_number}

   Reason for Roll_Number: Indexing the Roll number attribute of the Alumni entity can be beneficial if many queries filter or sort data based on this attribute. When a query condition is applied to an indexed column, the database can use the index to quickly locate the rows that match the condition rather than scanning the entire table.

```
CREATE INDEX alumni_idx ON Alumni(Income);
SET profiling = 1;
select * from Alumni where Roll_Number < 85602320;
SHOW PROFILE;
```

   Query execution time before indexing: 0.045519 s
   Query execution time after indexing: 0.0245 s

   Reason for Branch, Degree: In the context of the Alumni entity, Branch and Degree are likely to be commonly used attributes in queries that retrieve data about alumni who have studied in a particular branch or have a particular degree. For example, queries like "Find all alumni who have a Bachelor's degree in Computer Science" or "Find all alumni

who studied in the Electrical Engineering branch" can be optimized by creating indexes on the Branch and Degree columns, respectively.

Additionally, if the Branch and Degree attributes are used as foreign keys in other tables (such as the Education entity), indexing them can improve JOIN performance when querying those tables.

```
CREATE INDEX alumni_idx_1 ON Alumni(Branch, Degree);
SET profiling = 1;
select * from Alumni where Branch = "Biology";
SHOW PROFILE;
```

Query execution time before indexing: 0.0066 s
Query execution time after indexing: 0.0045 s

2) Instructor : Instructor ID
Reason: Queries that filter by Instructor_ID can quickly locate the exact record(s) that match the query condition without searching the entire table. Additionally, if the Instructor_ID attribute is used as a foreign key in other tables (such as the Belongs To or F.A. relationships), indexing it can improve JOIN performance when querying those tables.

```
CREATE INDEX instructor_idx ON Instructor(Instructor_ID);
SET profiling = 1;
select * from Instructor where Instructor_ID > 1090;
SHOW PROFILE;
```

Query execution time before indexing: 0.00108
Query execution time after indexing: 0.0008

3) Courses: Course ID
Reason: Queries that filter by Course ID can quickly locate the exact record(s) that match the query condition without searching the entire table. Additionally, indexing the Course ID attribute as a foreign key in other tables (such as the Takes relationship) can improve JOIN performance when querying those tables.

```
CREATE INDEX course_idx_1 ON Courses(Course_ID);
SET profiling = 1;
select * from Courses where Course_ID != 'WA889';
SHOW PROFILE;
```

Query execution time before indexing:0.0042
Query execution time after indexing: 0.0032

4) Scholarship: Scholarship Name and Amount
   Reason: In the context of the Scholarship entity, queries that filter or sort by Scholarship Amount or Scholarship Name could be common. For example, a query that retrieves all scholarships with a certain amount or a query that lists all scholarships in alphabetical order. If the Scholarship Name attribute has low cardinality (i.e., there are only a few distinct values), then the benefit of indexing it may be limited.

```
CREATE INDEX scolarship_idx ON Scholarships(Scholarship_Name, Amount);
SET profiling = 1;
select * from Scholarships where Amount > 3000;
SHOW PROFILE;
```

   Query execution time before indexing:0.0033
   Query execution time after indexing:0.0031

5) Achievements: Purpose
   Reason: Users would more frequently search based on the purpose of the achievement by the alumni. It might be the case that the user wants to contact some alumni who were good in that domain.

```
CREATE INDEX ach1 ON Achievements(Purpose);
SET profiling = 1;
select * from Achievements where Purpose = 'Academic';
SHOW PROFILE;
```

   Query execution time before indexing:0.0003
   Query execution time after indexing:0.00015

6) Department: Dept_Name, Minor: Dept_Name, Major: Dept_Name
   Reason: Users will only group rows in these tables based on department name to filter the alumni or number of faculties according to their departments.

```
CREATE INDEX deptidx_1 ON Department(Department_name);
SET profiling = 1;
select * from Department where Department_name != 'Biology';
SHOW PROFILE;
```

7) Companies: Company Name
   Reasons: Users will filter and search alumni based on which companies they are working on. Users might need some contacts of alumni who are working at D.E Shaw. That's why indexing based on Company Names will be efficient.

```
CREATE INDEX compidx ON Companies(Company_name);
SET profiling = 1;
select * from Companies where Company_name = 'Wagner, Castro and Kennedy';
SHOW PROFILE;
```

Query execution time before indexing:0.000026
Query execution time after indexing:0.000010

8) Extra_Curricular: Name
   Reasons: Users will search for alumni based on what extra-curricular activities alumni did in their college years. If they want to conduct a session in that activity or want some donation for that activity, it will be helpful to search based on the name of the activity.

```
CREATE INDEX ext1 ON Extra_curricular(Name_);
SET profiling = 1;
select * from Extra_curricular where Name_ = 'budget';
SHOW PROFILE;
```

   Query execution time before indexing:0.000008
   Query execution time after indexing:0.000007

9) Works_in: Role
   Reason: Users will search and filter alumni in this relationship table based on their roles of jobs in any institution or company they are associated with. It will help them to reachout the alumni working in a particular role who can guide the students for that role.

```
CREATE INDEX wor1 ON Works_In(Role_);
SET profiling = 1;
select * from Works_In where Role_= 'Patent attorney';
SHOW PROFILE;
```

   Query execution time before indexing:0.00033
   Query execution time after indexing:0.00011

10) Education: Institution name
    Reason: We have made the education table such that it contains many institutions worldwide, irrespective of whether alumni are studying in it. Many rows of the same institute name will likely offer different disciplines and degrees. If we group them based on the institute names, it will be easier to search.

```
CREATE INDEX edx ON Education(Institution_name);
SET profiling = 1;
select * from Education where Institution_name = 'Emily Clark';
SHOW PROFILE;
```

    Query execution time before indexing:0.00012
    Query execution time after indexing:0.000079

Engine	Support	Comment	Transactions	XA	Savepoints
PERFORMANCE_SCHEMA	YES	Performance Schema	NO	NO	NO
MyISAM	YES	MyISAM storage engine	NO	NO	NO
InnoDB	DEFAULT	Supports transactions, row-level locking, and fo...	YES	YES	YES
ndbinfo	NO	MySQL Cluster system information storage engine	NULL	NULL	NULL
BLACKHOLE	YES	/dev/null storage engine (anything you write to ...	NO	NO	NO
ARCHIVE	YES	Archive storage engine	NO	NO	NO

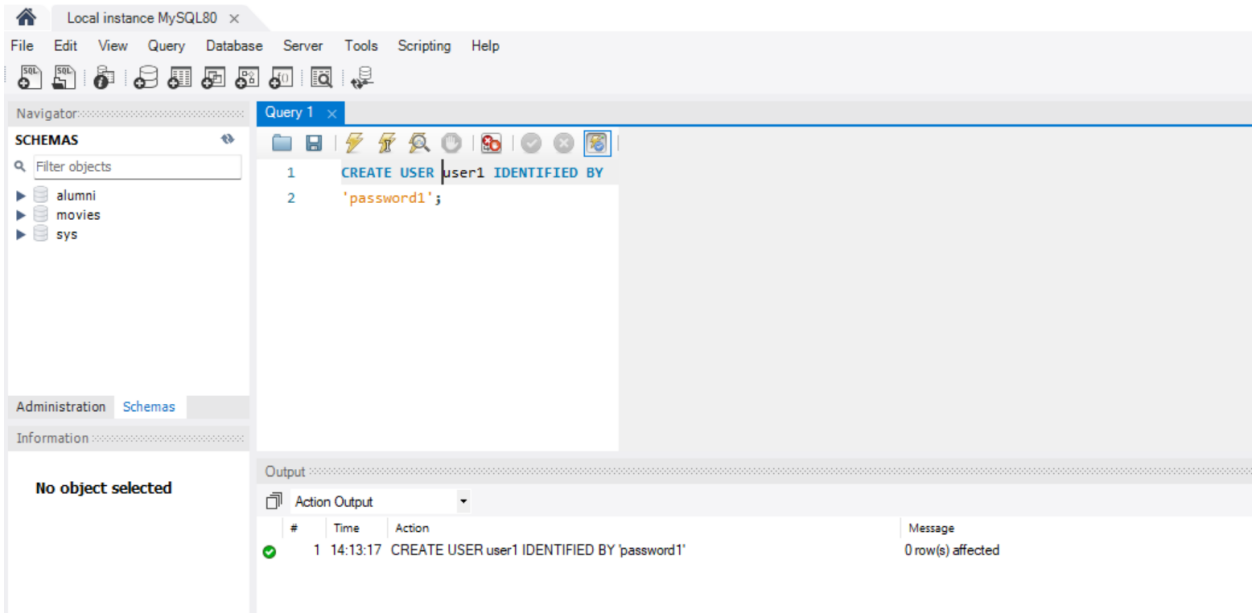
For table extension, we have used InnoDB as the storage engine, which is available in MySQL that provides support for transactions, foreign keys, and row-level locking. It is the default storage engine used by MySQL, and it is designed to provide high performance and reliability for transactional workloads.

We have used InnoDB for all the tables, and as it is the default engine, then there was no need to change the engine.

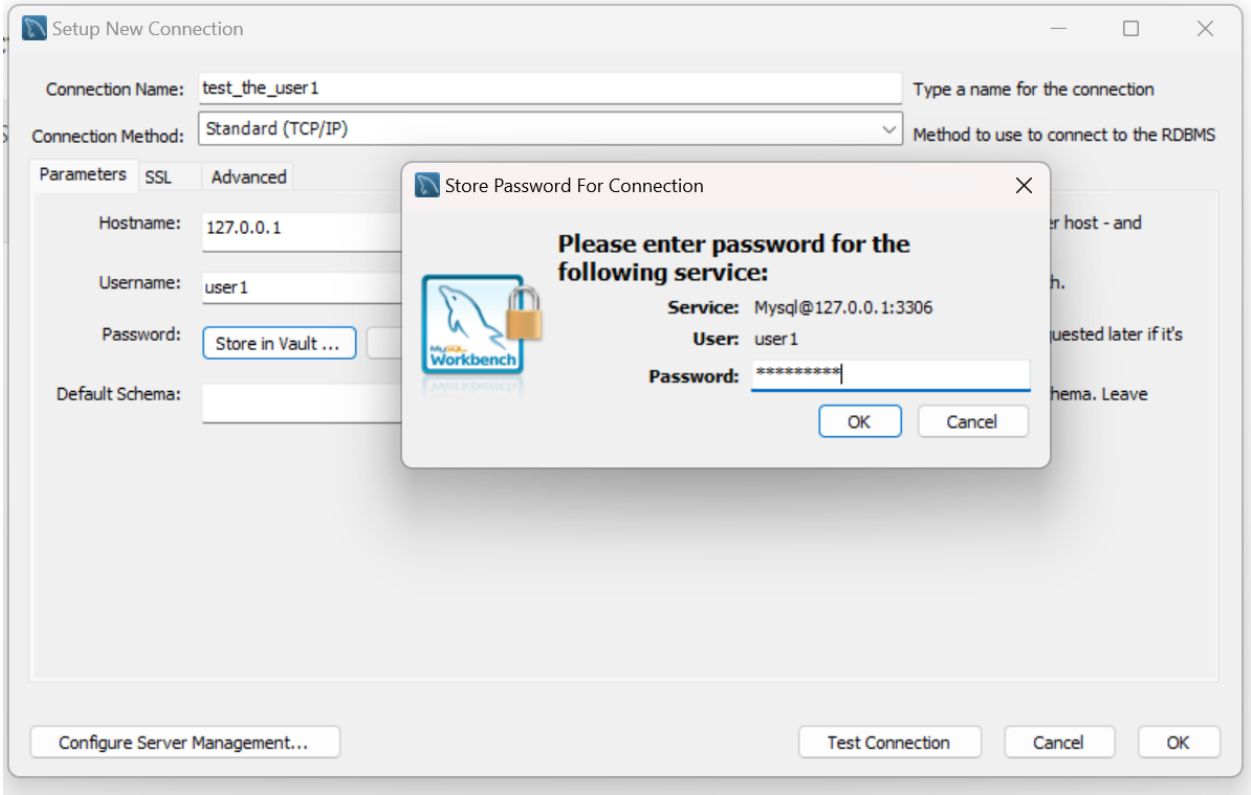
Responsibility of Group 2

Question 1

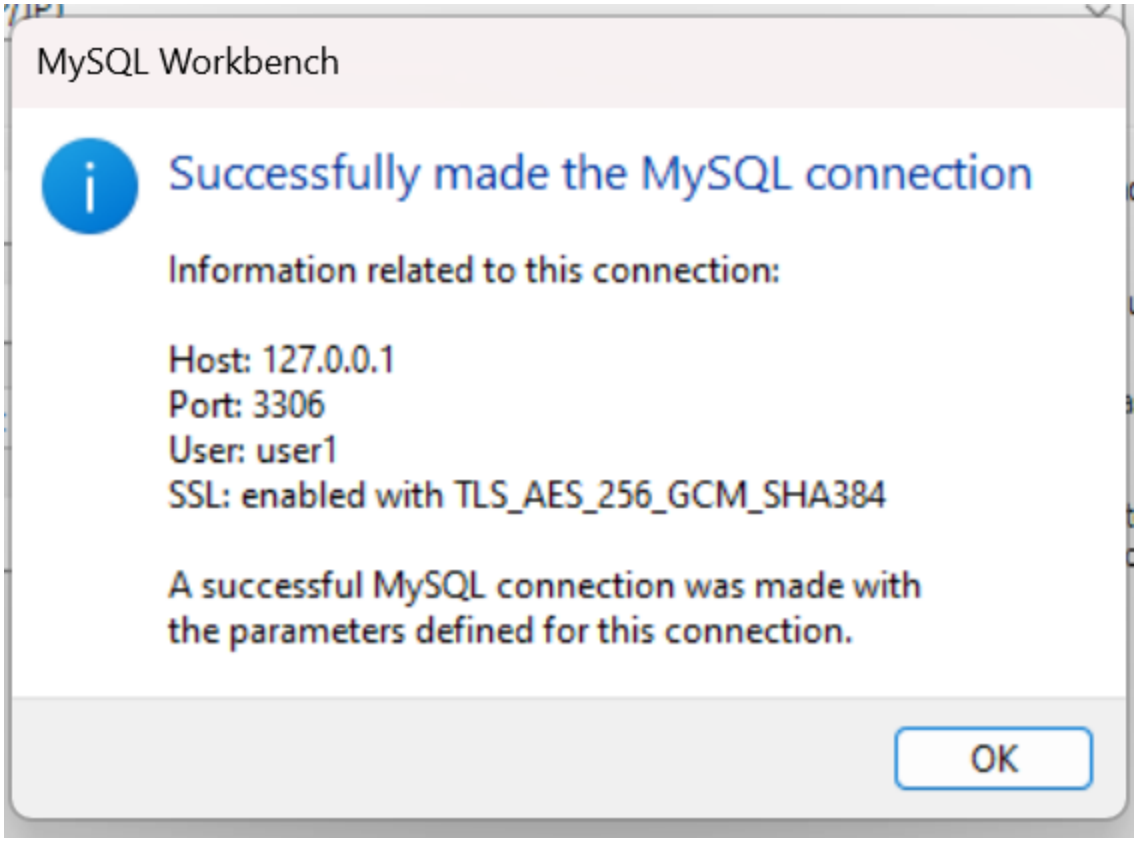
Part 1



Making the User and setting the password



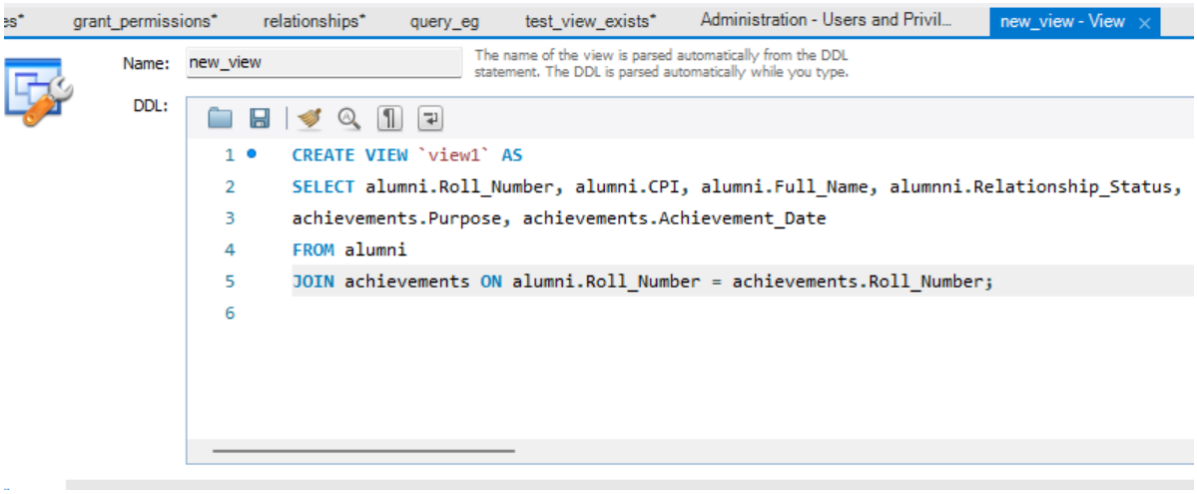
Setting up the connection



Done with the connection.

Part 2

Making View1



This contains the columns Roll_Number, CPI, Full_Name from Alumni Table.
It contains the Purpose, Achievement_Date column from Achievements table.

It also contains the “user-defined” ENUM column “Relationship_Status” from Alumnni Table.

View1 Print

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Roll_Number	CPI	Full_Name	Relationship_Status	Purpose	Achievement_Date
10316544	3.60	Jeremy Drake	Married	Sports	2022-02-15
10447530	2.87	David Morales	Married	Sports	2022-05-22
10885496	8.02	John Brown	Unmarried	Academic	2022-05-19
11027859	1.09	Sabrina Hunter	Unmarried	Academic	2022-04-03
11238079	0.19	Jill May	Unmarried	Sports	2022-11-10
11364440	1.04	Michael Vazquez	Married	Sports	2022-08-23
11898376	4.85	Joseph Jacobson	Unmarried	Other	2022-01-02
12506648	6.59	Stephanie Larson	Married	Academic	2022-01-03
12526533	5.83	Taylor Perry	Married	Other	2022-02-07
12587496	7.17	William Jones	Married	Other	2022-02-10
12587496	7.17	William Jones	Married	Sports	2022-03-24
12951747	0.53	Daniel Watson	Married	Other	2022-01-02
12977205	7.48	Walter Smith	Married	Academic	2022-09-18
12978235	0.94	Theresa Ross	Married	Cultural	2022-07-03

Making View2

Name: view2

The name of the view is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

DDL:

```
1  CREATE VIEW `view2` AS
2  SELECT alumni.Roll_Number, alumni.Full_Name, alumni.CPI, alumni.Relationship_Status,
3  studied_in.Degree, studied_in.Discipline
4  FROM alumni
5  JOIN studied_in ON alumni.Roll_Number = studied_in.Roll_Number;
```

This view contains the Roll_Number, Full_Name, CPI columns from Alumni Table.
It contains the Degree and Discipline columns from the studied_in Table.
It also contains the “user-defined” ENUM column “Relationship_Status” from Alumnni Table.

View2 Print

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Roll_Number	Full_Name	CPI	Relationship_Status	Degree	Discipline
24641065	Katherine Pham	9.69	Unmarried	10	Art
28559195	Elizabeth Russell	8.07	Unmarried	10	Art
29312536	Sandra Stanton	8.29	Married	10	Art
64966317	Brian Medina	3.88	Unmarried	10	Art
67456864	Heidi Ayala	7.24	Married	10	Art
89704129	Jonathan Whitehead	7.05	Unmarried	10	Art
10165745	Lindsey Williams	6.78	Unmarried	12	Art
24344137	Caitlin Cunningham	7.77	Married	12	Art
31983263	Jason Scott	2.47	Unmarried	12	Art
48008670	Kristin Levy	6.35	Unmarried	12	Art
65111846	David Rodriguez	1.90	Married	12	Art
18394963	John Schultz	6.45	Married	10	Comput...
29779812	Jeffrey Davis	3.13	Unmarried	10	Comput...
33785505	Austin Barrett	2.91	Married	10	Comput...

Art

Part 3

Giving user1 SELECT, UPDATE and DELETE privileges on alumni table.

```
1 • use alumni;
2 • GRANT SELECT, UPDATE, DELETE on alumni to user1;
3
4 • show grants for user1;
5
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

Grants for user1@%
GRANT USAGE ON *.* TO `user1`@`%`
GRANT SELECT, UPDATE, DELETE ON `alumni`....

Result Grid | Form Editor | Read Only | Context Help | Snippets

Result 4

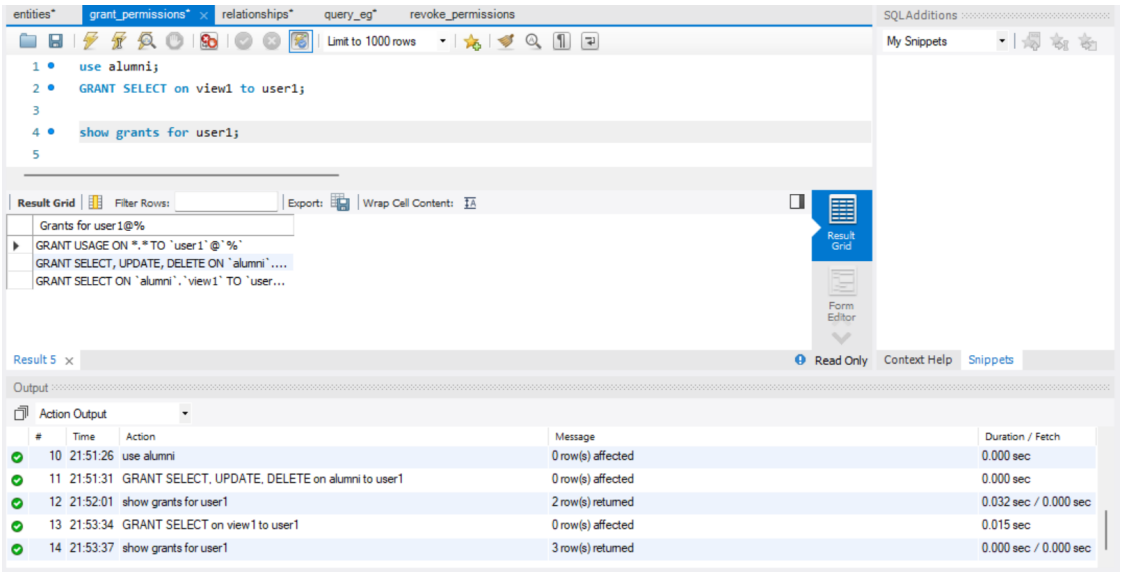
Output

Action Output

#	Time	Action	Message	Duration / Fetch
8	21:44:05	select * from alumni.view1 LIMIT 0, 1000	500 row(s) returned	0.031 sec / 0.000 sec
9	21:50:34	CREATE USER user1 IDENTIFIED BY 'password1'	0 row(s) affected	0.015 sec
10	21:51:26	use alumni	0 row(s) affected	0.000 sec
11	21:51:31	GRANT SELECT, UPDATE, DELETE on alumni to user1	0 row(s) affected	0.000 sec
12	21:52:01	show grants for user1	2 row(s) returned	0.032 sec / 0.000 sec

Part 4

Giving user1 SELECT privileges on “view1”.

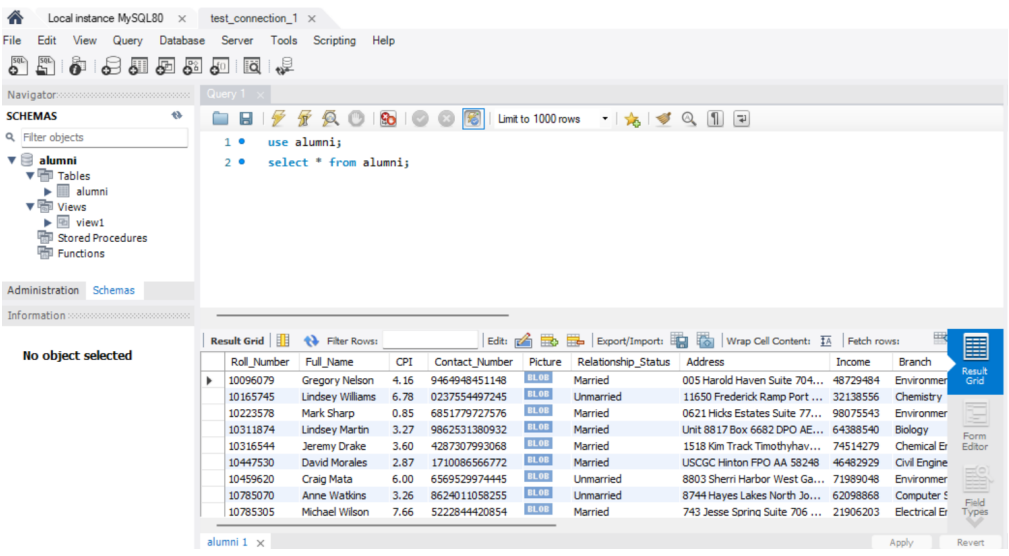


Part 5

Logging in as “user1”. The left panel displays the limited acces privileges that “user1” has. He can only see the “alumni” table and the “view1” view.

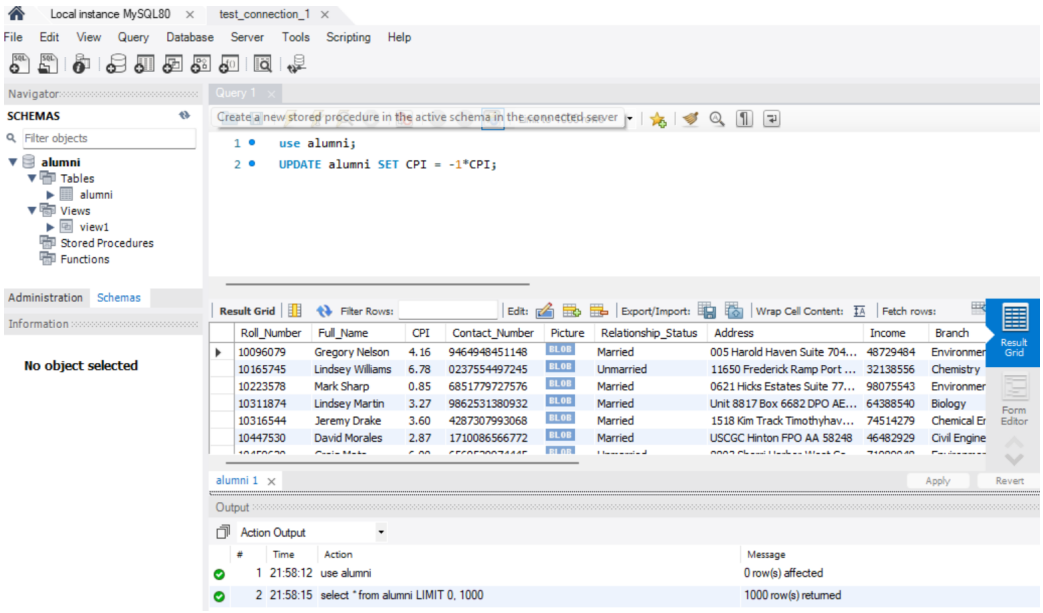
Operations on Table 1 - Alumni

SELECT

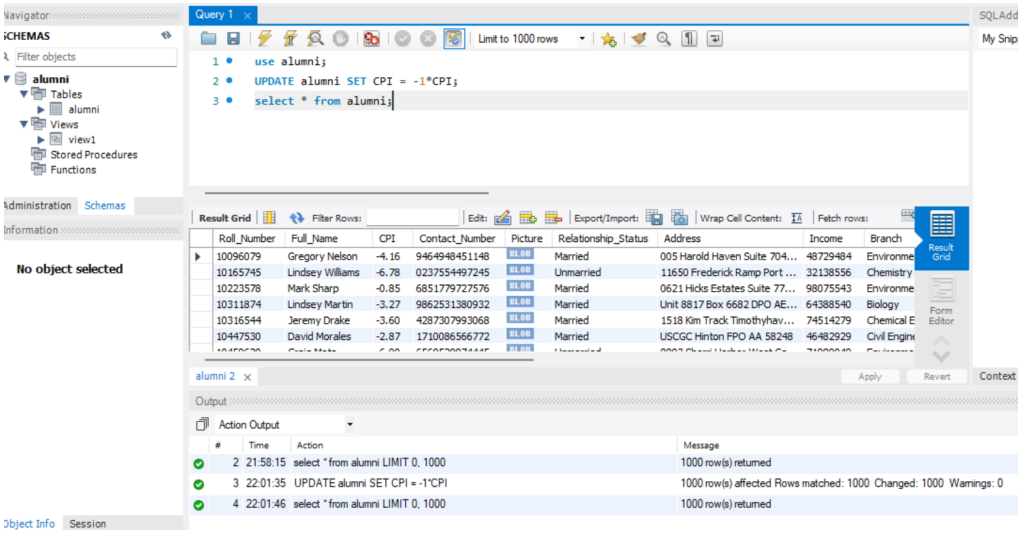


UPDATE

Before



After

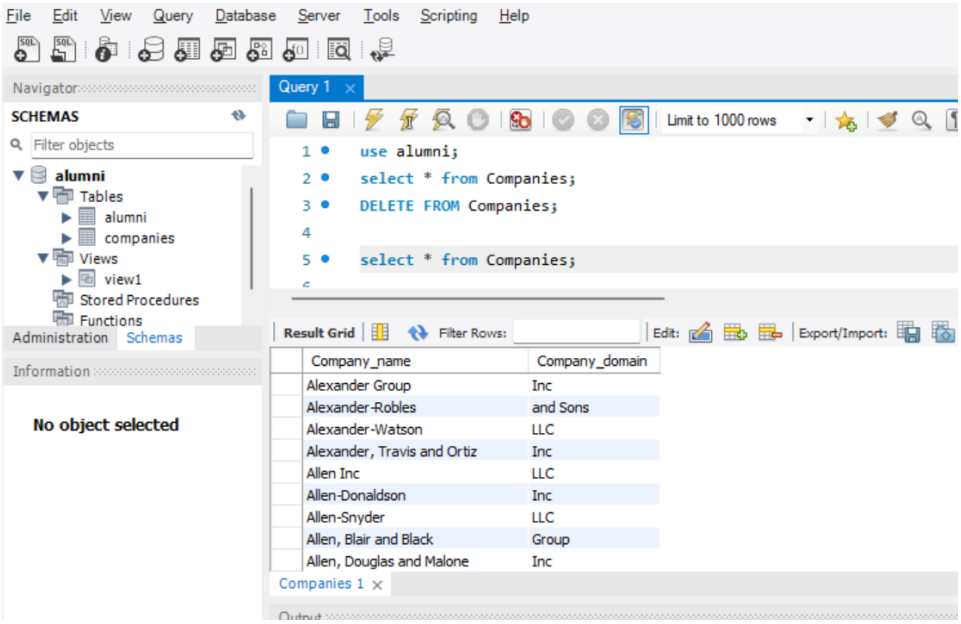


User1 can update the CPI to its negative value.

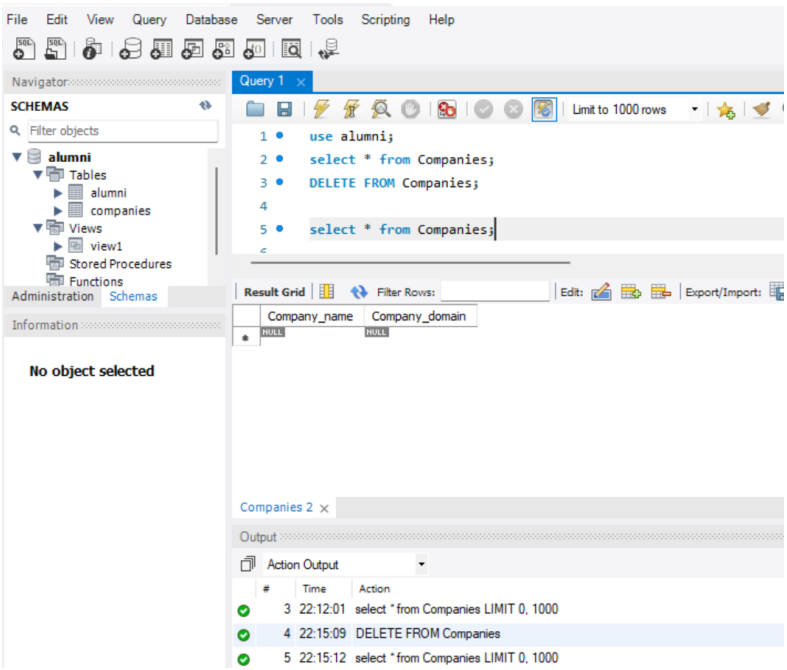
DELETE

Before

To avoid inability to delete due to foreign key constraint, I granted user1 access to a “Companies” table.



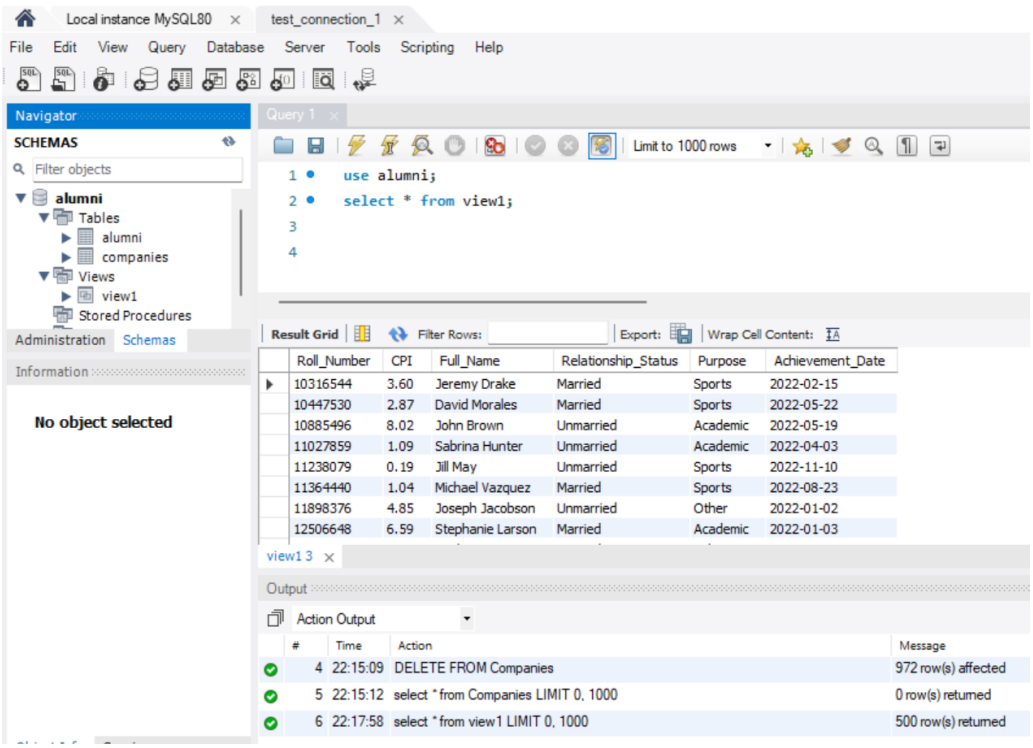
After



The DELETE operation worked.

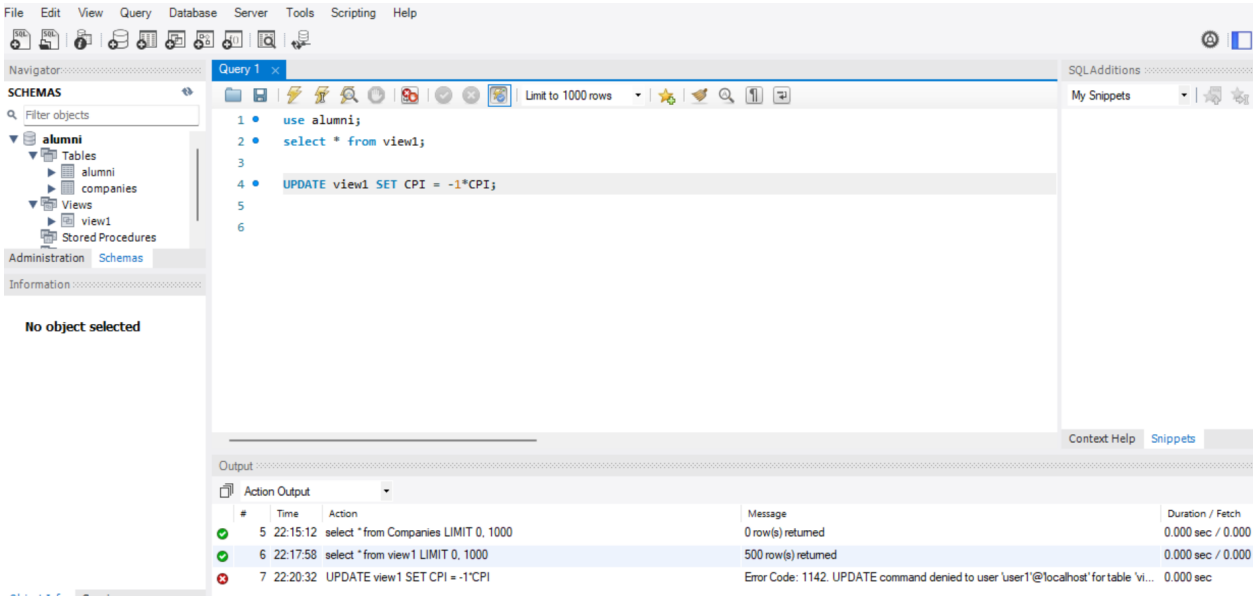
Operations on View1

SELECT



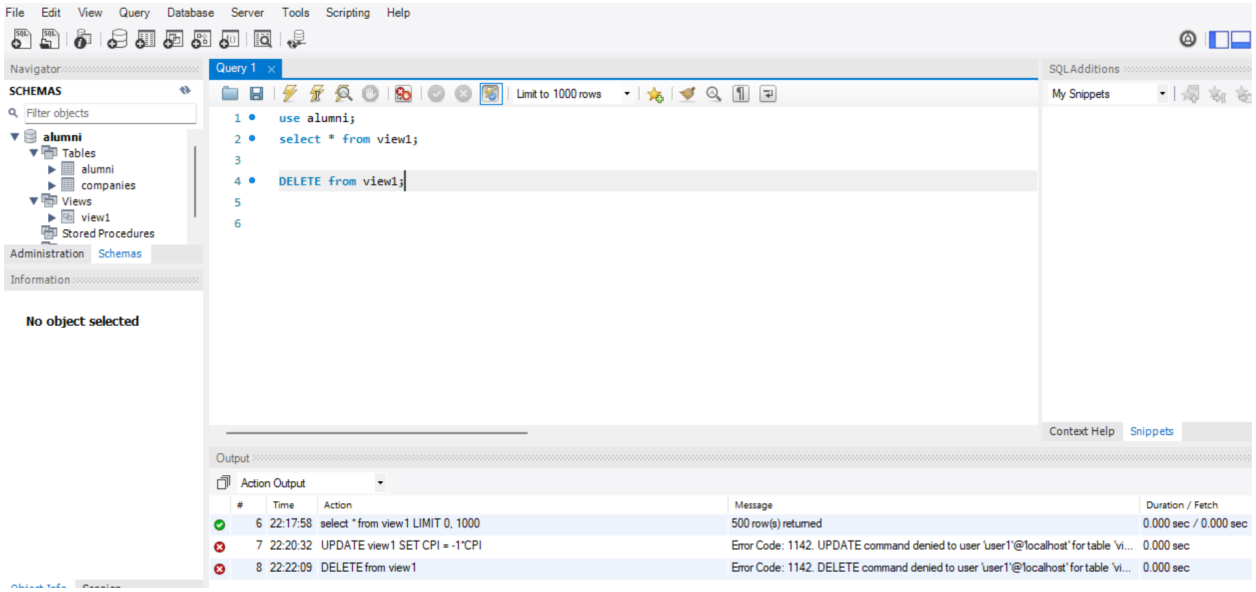
User1 can do the SELECT operation on view1

UPDATE



We can observe in the Error log that UPDATE permission is denied to user1 on view1. So our granted permission system is working.

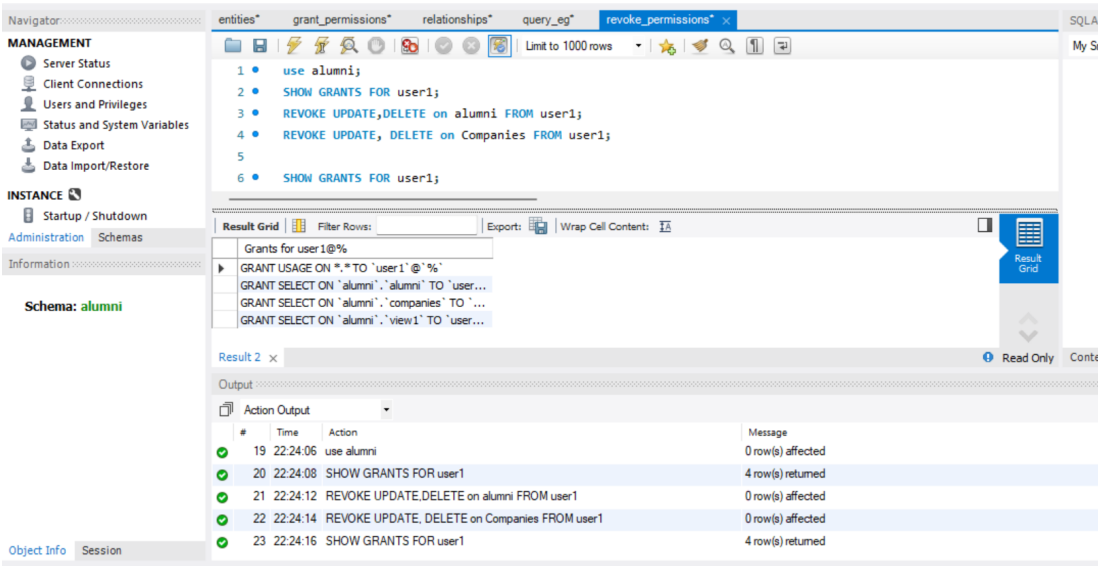
DELETE



We can observe in the Error log that DELETE permission is denied to user1 on view1. So our granted permission system is working.

Part 6

The code for REVOKING permissions is-



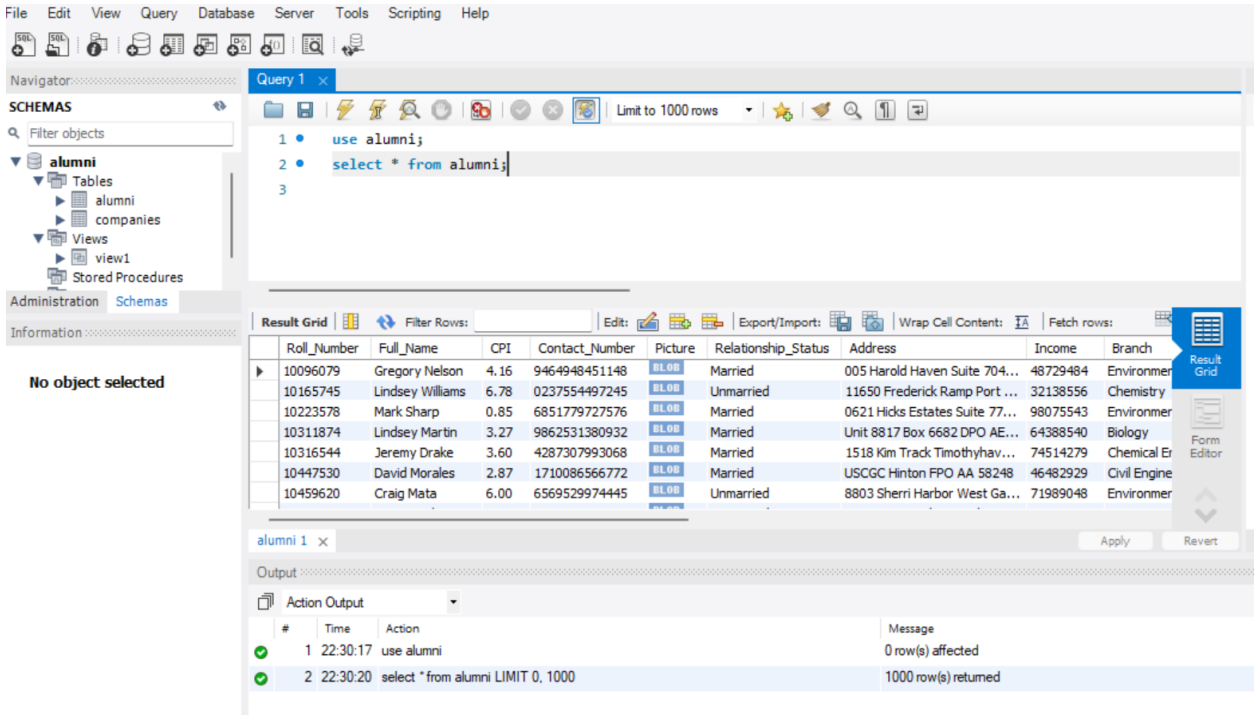
We can see in the updated permissions that only SELECT operations are available to user1 on Alumni Table, Company table and view1.

I added access to Company 1 because DELETING from alumni led to referential integrity constraints, and I wanted to correctly show the DELETE operation on a table like Company from where I could delete and not receive an error for violation of integrity.

Part 7

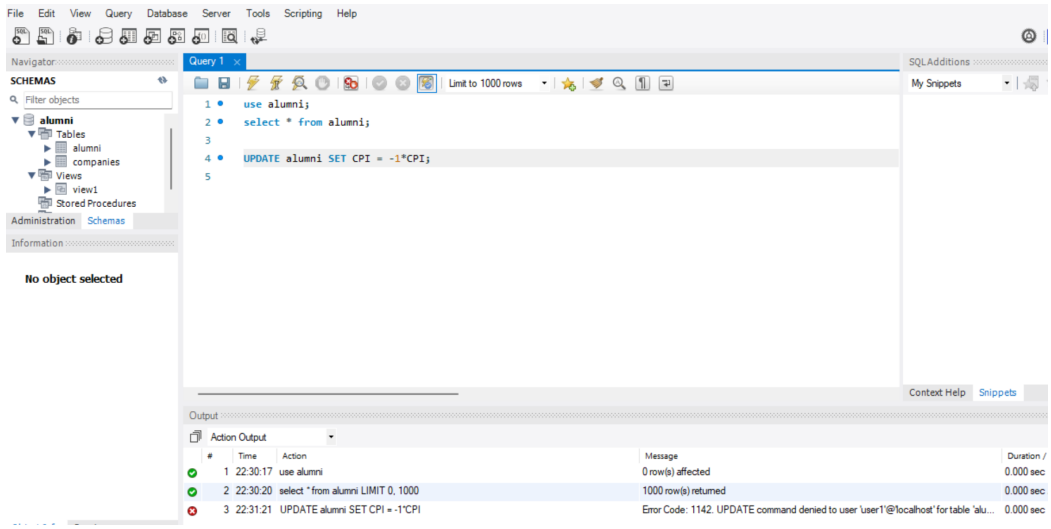
Operations on Table 1 - Alumni

SELECT



As expected, the SELECT operation still works.

UPDATE

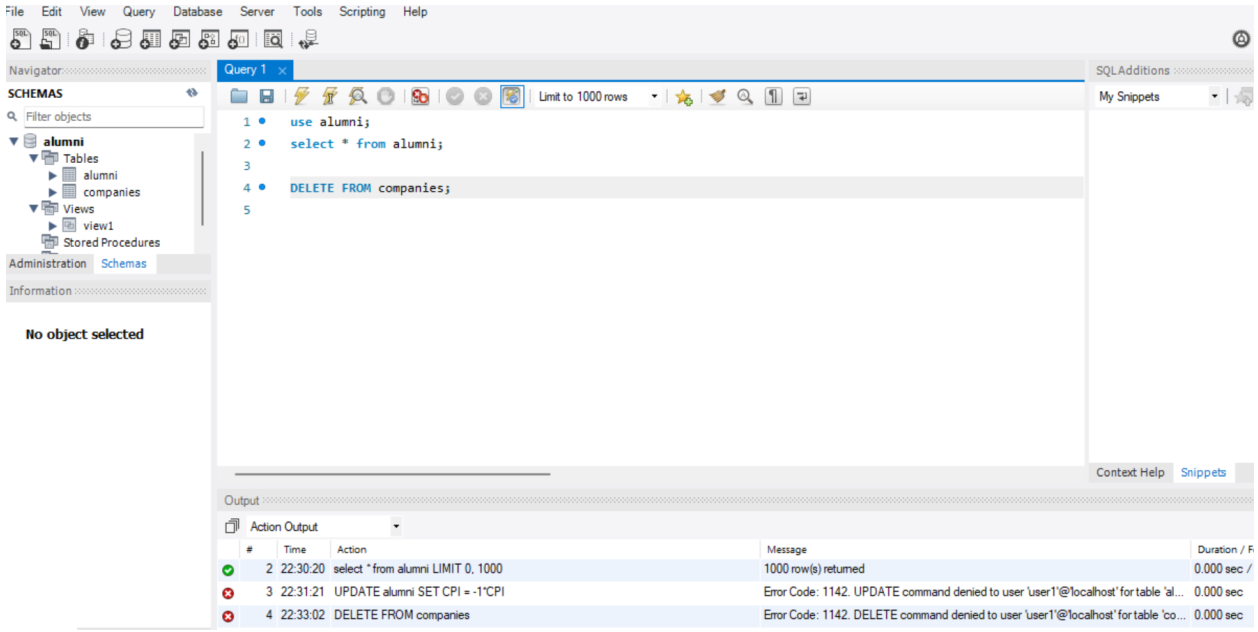


Like we wanted, user1 cannot UPDATE alumni table anymore as he doesn't have the permission now.

DELETE

Before

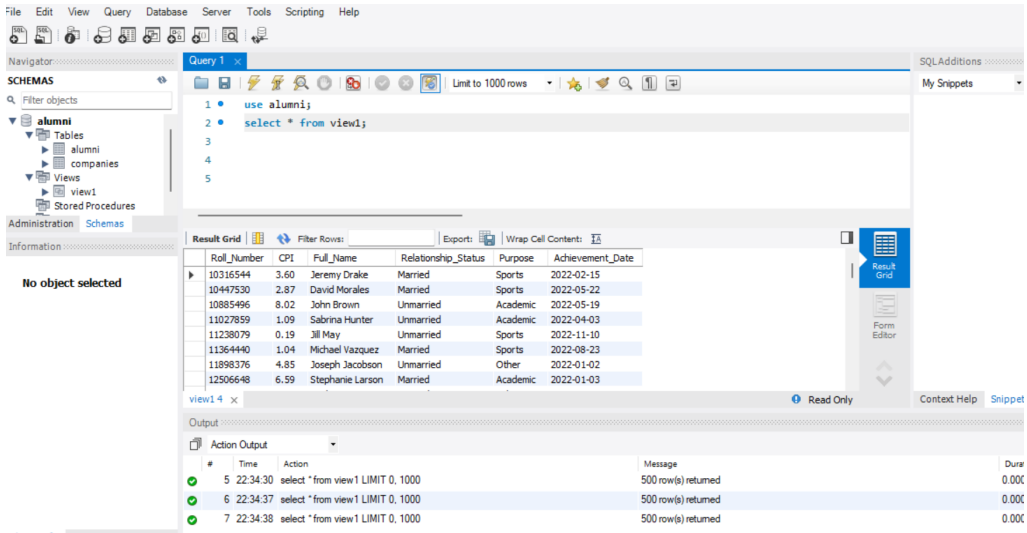
To avoid inability to delete due to foreign key constraint, I granted user1 access to a “Companies” table.



As expected, user1 can no longer issue DELETE commands on Companies table. This is evident from the Error Log.

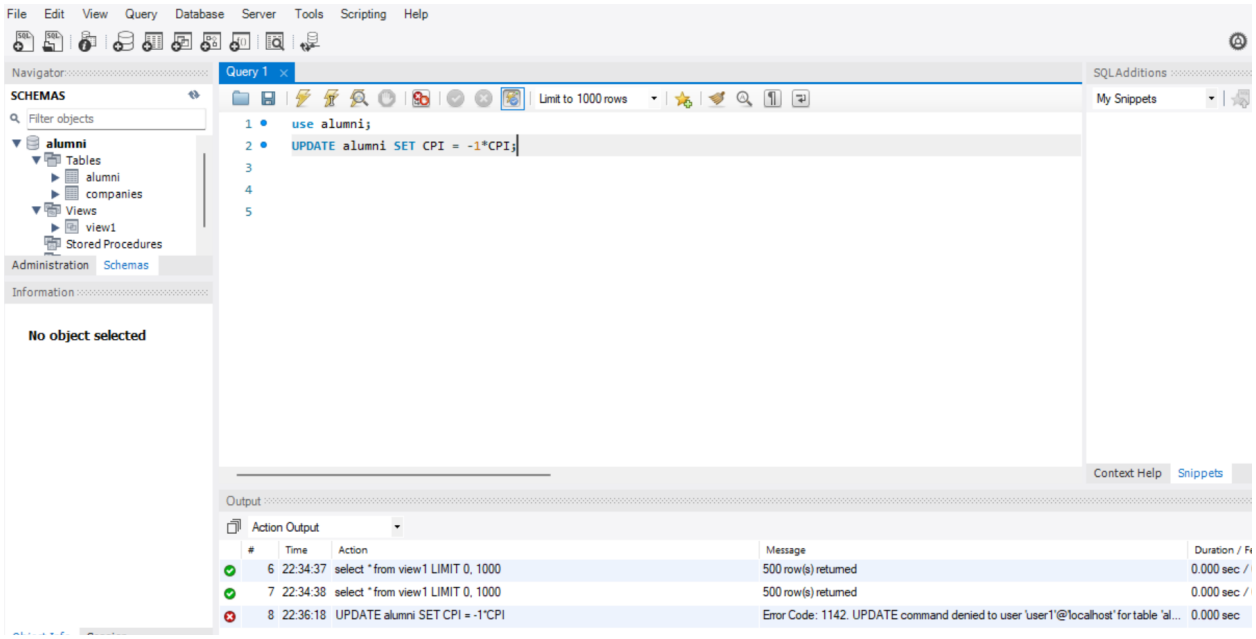
Operations on View1

SELECT



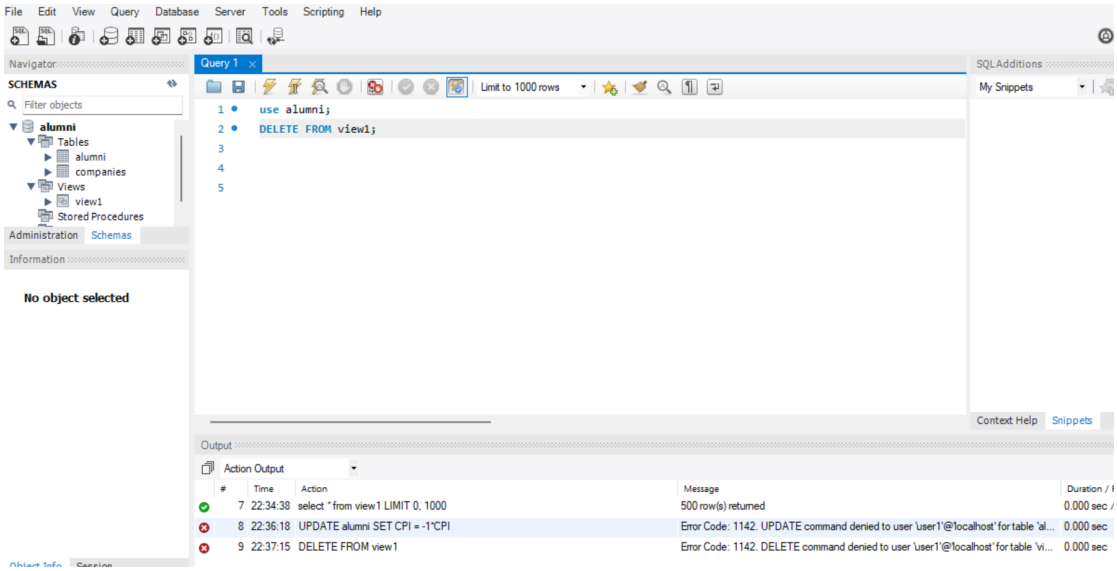
As expected, user1 can SELECT from view1 because he has SELECT access.

UPDATE



We can observe in the Error log that UPDATE permission is denied to user1 on view1. So our granted permission system is working.

DELETE



We can observe in the Error log that DELETE permission is denied to user1 on view1. So our granted permission system is working.

Question 2

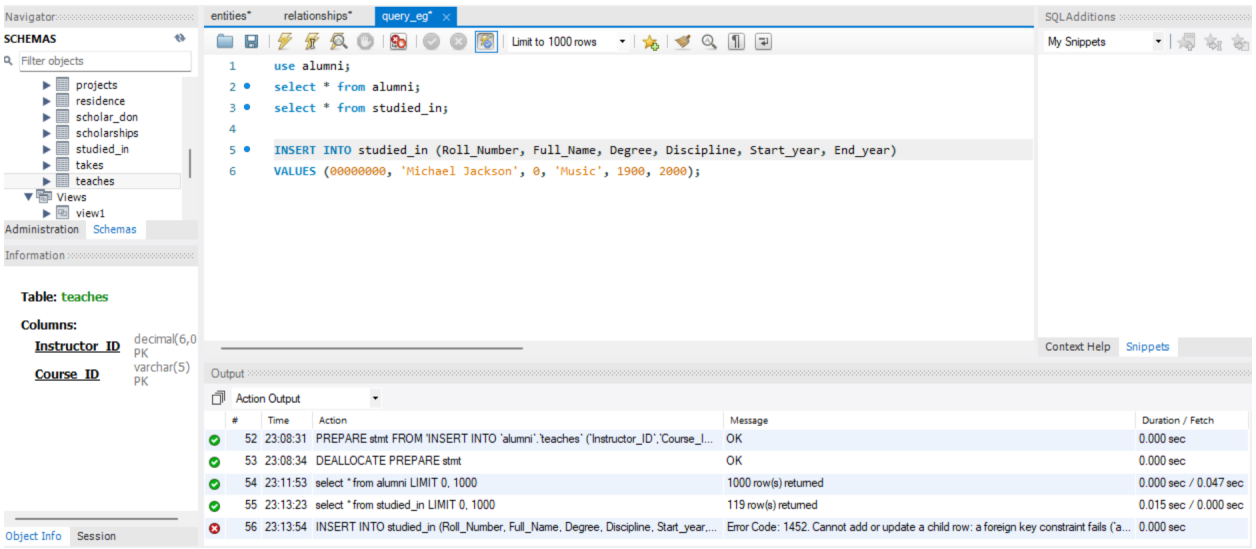
There are a few situations where there would be violation of referential integrity in our database. These are related to Foreign Key Constraints being violated.

1) If Table A has the primary key of Table B as a foreign key, and we **insert** a new tuple into Table A with a value for foreign key (say “x”). If the value “x” doesn’t exist as a primary key anywhere in Table B, then we will get a an error that a referenced value doesn’t exist.
2) If Table A has the primary key of Table B as a foreign key, and we **modify** a tuple (and it’s foreign key value) in Table A with a new value for foreign key (say “x”). If the value “x” doesn’t exist as a primary key anywhere in Table B, then we will get a an error that a referenced value doesn’t exist.
3) If Table A has the primary key of Table B as a foreign key, and we **delete** a particular row only in Table B (and not in Table A), whose primary key is being referenced somewhere in Table A, then we would get an error saying that the value in parent row cannot be deleted, because Table A still references it. If we were to delete such a row in Table B without deleting the referencing rows in Table A, then Table A would end up referencing a value from Table B which no longer exists. This would be a violation of the referential integrity.
4) If Table A has the primary key of Table B as a foreign key, and we **update** a particular row only in Table B (and not in Table A), whose primary key is being referenced somewhere in Table A, then we would get an error saying that the value in parent row cannot be updated, because Table A still references the un-updated value of it. If we were to update such a row in Table B without updating the referencing rows in Table A, then Table A would end up referencing an old value from Table B, which has since been updated. This would be a violation of the referential integrity.

Our database has multiple instances of foreign key references, and these situations can arise there. I will take one example of each case and show the outputs. In the below examples, I’m taking Table A = studied_in and Table B = alumni.

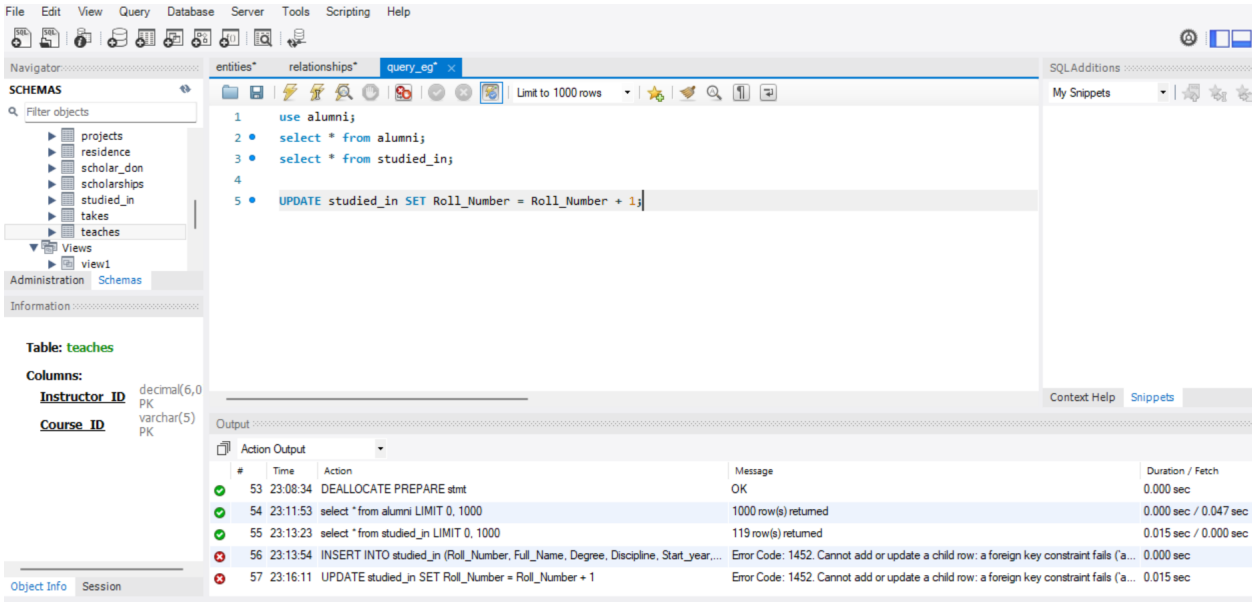
Errors

Case 1



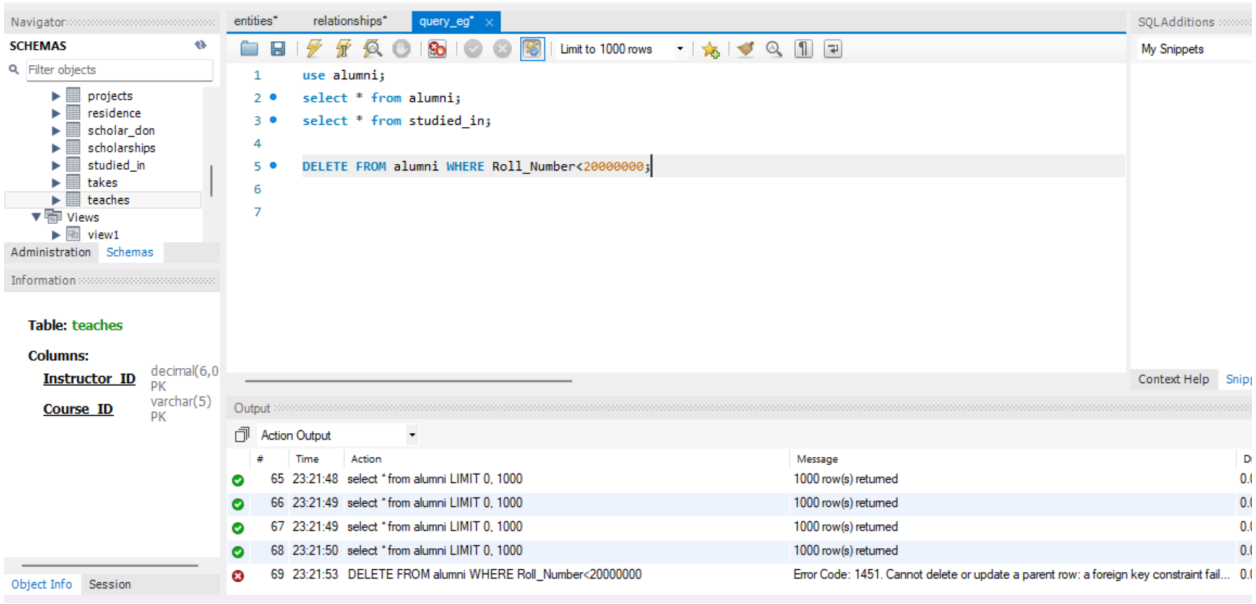
The foreign key constraint error shows up in the error log because 00000000 is not a valid roll number.

Case 2



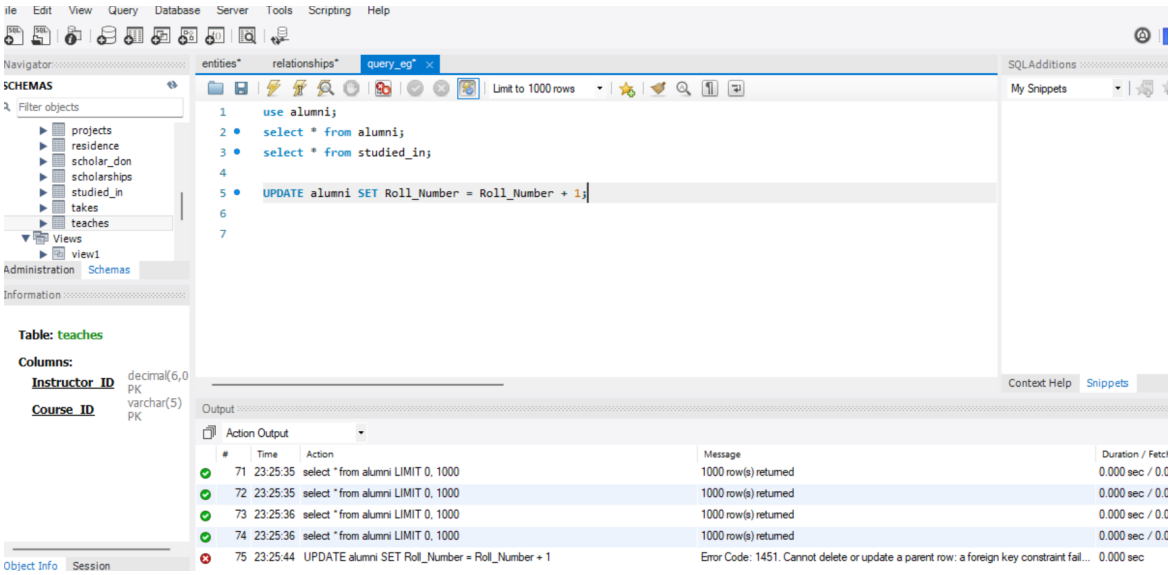
In the above query, I'm updating the "Roll_Number" foreign key in Table A. Not all Roll Numbers incremented by 1 are still valid Roll Numbers (our data is random), so some of the updated Roll Numbers are no longer valid and don't refer to any primary key in Table B. This leads to violation of referential integrity.

Case 3



There are some rows in “studied_in” which refer to Roll_Number values less than 20000000. So, when we try to delete this in the parent row of “alumni”, we get a referential integrity error as per the error log because the primary keys being deleted in “alumni” are still being referenced in “studied_in”.

Case 4



Here, I tried to update primary key values in “alumni” by incrementing them by 1 and this leads to rows in “studied_in” referring to old primary keys which have now been updated. This leads to violation of referential integrity as the referenced value no longer exists in the referenced table.

Solutions

We can solve these issues by adding referential actions like ON DELETE CASCADE, ON UPDATE CASCADE. This will take care of Case 3 and Case 4. The two actions mean that if a record is deleted/updated in Table B, then all the corresponding records in Table A will automatically be deleted/updated in Table A. This would ensure that there is no violation of referential integrity. Case 1 and Case 2 are errors on the side of database user, and he/she should take care not to insert tuples that refer to foreign keys that don't exist, and not update the referencing values to foreign keys that don't exist.

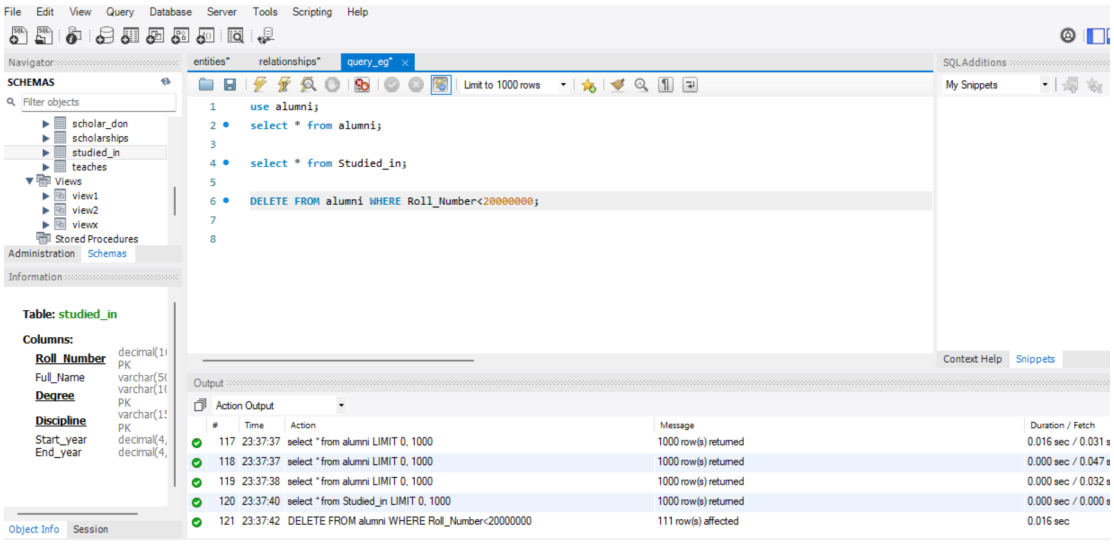
I have currently isolated the "alumni" and "studied_in" tables to show the relationship between their constraints only. Of course, this solution needs to be updated in every table that references "alumni" as a foreign key.

Case 3

```
37  CREATE TABLE Studied_In(
38      Roll_Number numeric(10),
39      Full_Name varchar(50),
40      Degree varchar(10),
41      Discipline varchar(150),
42      Start_year numeric(4),
43      End_year numeric(4),
44      PRIMARY KEY(Roll_Number, Discipline, Degree),
45      FOREIGN KEY(Roll_Number) REFERENCES Alumni(Roll_Number)
46      ON DELETE CASCADE
47      ON UPDATE CASCADE,
48      FOREIGN KEY(Discipline, Degree) REFERENCES Education(Discipline, Degree)
49      );
```

This is after implementing the solution.

Test-



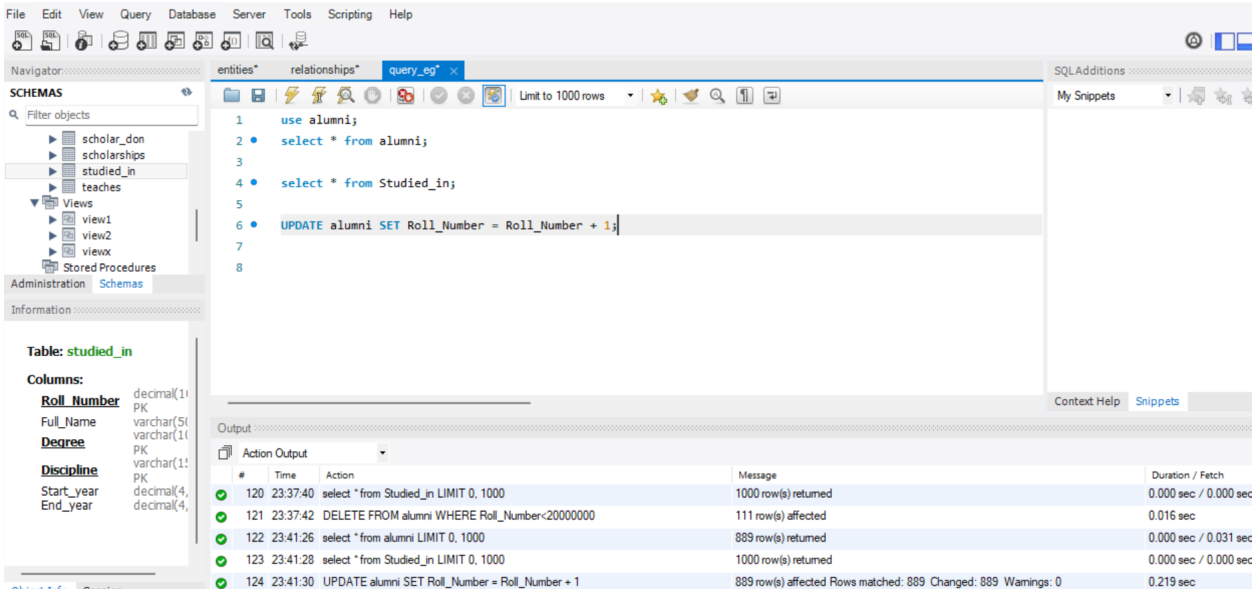
The deletion operation works. It works on 111 rows. This doesn't throw a referential integrity error.

Case 4

```
37  CREATE TABLE Studied_In(
38      Roll_Number numeric(10),
39      Full_Name varchar(50),
40      Degree varchar(10),
41      Discipline varchar(150),
42      Start_year numeric(4),
43      End_year numeric(4),
44      PRIMARY KEY(Roll_Number, Discipline, Degree),
45      FOREIGN KEY(Roll_Number) REFERENCES Alumni(Roll_Number)
46      ON DELETE CASCADE
47      ON UPDATE CASCADE,
48      FOREIGN KEY(Discipline, Degree) REFERENCES Education(Discipline, Degree)
49      );
```

Solution Implementation

Test-



The updation statement works in this case after setting the cascade on update. The updates in the table is shown in the log. It works on 889 rows. This doesn't throw a referential integrity error.

Responsibility of G1 and G2:

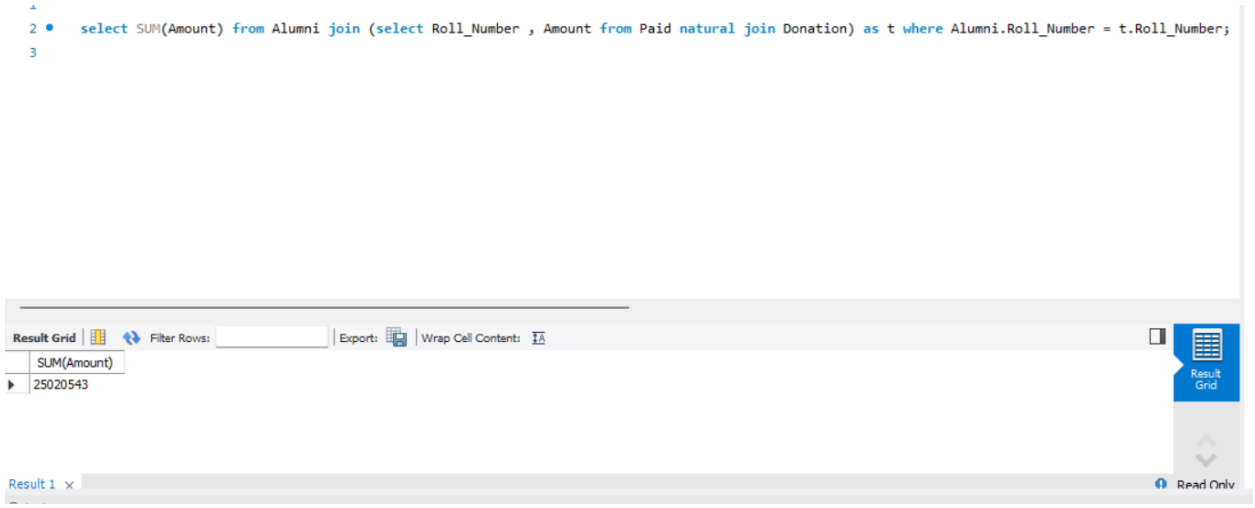
List of Operations:

- Operation - 1

  SQL Query → select SUM(Amount) from Alumni join (select Roll_Number , Amount from Paid natural join Donation) as t where Alumni.Roll_Number = t.Roll_Number;

  What is the function of this operation?
  This SQL query gives the summation over all the donations made to the institute by a particular alumni.

  This operation (f1) satisfies specification 3



- Operation - 2
  SQL Query → (select * from Alumni join (select Roll_Number from Education join Studied_In where Education.Discipline = "Computer Science") as edu) union (select * from Alumni join Companies as c);

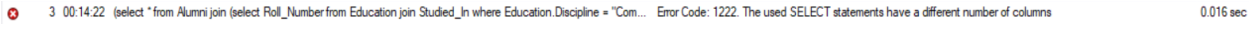
$$\text{Alumni} \bowtie \{\rho_{edu}[\pi_{\text{Roll\_number}}(\sigma_{\text{Education.Discipline} = \text{"Computer Science"}}(\text{Education} \bowtie \text{Studied\_In}))] \cup \rho_c[\text{Alumni} \bowtie \text{Company}]\}$$

What is the function of this operation?
This operation gives the union of Education cross Studied_In where the discipline is “Computer Science” and Alumni cross Companies.

As the cardinalities of the sets are not equal, union operation wont work here, which produces an error.

This operation (f2) satisfies specification 1

Error:



- Operation - 3
  SQL Query → select Role_ , Salary, case
  when Role_ = "Media planner" then 0
  else Salary
  end as New_Salary
  from Alumni join (select Role_ , Salary from Works_In join Companies)
  as c;

$$\rho_c(\pi_{\text{Role\_, Salary}}(\text{works\_In} \bowtie \text{companies}))$$
$$\rho_{\text{NewSalary}}(\text{Case when Role\_ = 'Media planner' then 0 else Salary end})$$
$$\pi_{\text{Role\_, Salary, New\_Salary}}(\text{Alumni} \bowtie c)$$

What is the function of this operation?

This query gives the salary of all roles with making the salary zero of a particular role.

This can be used to show all salary with modifying salaries of few roles making it either 0 or increasing or decreasing by some ratio. This query satisfies the involvement of use of case.
This operation (f3) satisfies specification 3.

```
1
2  select Role_ , Salary, case
3  when Role_ = "Media planner" then 0
4  else Salary
5  end as New_Salary
6  from Alumni join ( select Role_ , Salary from Works_In join Companies) as c;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content: | Fetch rows:

Role_	Salary	New_Salary
Marine scientist	17374439	17374439
Psychologist, clinical	16455364	16455364
Media planner	93706306	0
Lecturer, higher education	39991362	39991362
Community arts worker	13709170	13709170

Result 2 | Read Only

- Operation - 4
  SQL Query → insert into Images(imageId , image , caption) values (87528752875287528752 , "" , "Caption-1");

Images ← Images ∪ {(87528752875287528752 , "" , "Caption-1")}

insert into alumImages(roll_number, imageId) values (10096079, 87528752875287528752);

alumImages ← alumImages ∪ {(10096079 , 87528752875287528752)}

What is the function of this operation?

These queries insert an image in the table alumImages.

This operation (f4) satisfies specification 2

- Operation - 5
  SQL Query → insert into Paid(Transaction_ID , Roll_Number) values (54512 , 5454);

$$Paid \leftarrow Paid \cup \{(54512, 5454)\}$$

What is the function of this operation?

This query shows that we cannot add Transaction_ID , Roll_Number pairs in the paid table if no alumni of that roll number exists in the database. This means that we can not add a transaction for which the person paying does not exist.

This operation (f5) satisfies specification 1

Here we get the issue of foreign key constraint.

Error:

Work Distribution:

Dhairya Shah (G1):

1) Populated the Donation, Department, Job Posting tables which come under the entity set using python.
2) Populated the Belongs_To, Dual_Major, FA, Got tables which come under the relationship tables using python.
3) Implemented indexing and explained the reasons wherever used.
4) Formed the tables of above mentioned entities and relationships in mysql.
5) Ensured that all the data populated works well with the indexing motives, and the constraints defined in the previous assignments.

Harshvardhan Vala (G1):

1) Populated the Alumni Table, Image Table and the Projects table.
2) Also populated the Works_In Relationship Table, Image Relationship Table and Role_In Relationship Table.
3) Wrote the code for each table
4) Helped in debugging different codes and solving conflicts of certain constraints such as total and partial participation, many-one, many-many, one-one relationships, etc.
5) Incorporated the images and caption in the database along with user defined data type.
6) Documentation of the project report.

Joy Makwana (G1) :

1) Populated the Department Table, Companies Table, Education Table
2) Populated the relationship tables - Don_Extracurricular Table, Studied_In, Takes and Teaches Table
3) Final compilation of all the tables and csv files at one place and populating the sql tables using the corresponding csv files.
4) Worked on the extension of tables and documentation of the report.

Medhansh Singh (G1):

1) Populated the Scholarship Table, Course Table,Achievements Table for entities.
2) Populate the Scholar_Don, Posts table, Lived Table for relationships.
3) Made necessary changes to the ER diagram and documented the same.

Nokzendi Aier (G1):

1) Populated Visits, Residence, Extracurricular Tables for entities.
2) Populated the Project Guide, Minor, Paid, Participates Relationship Tables.
3) Wrote python code to populate each tables keeping in mind the many/one to many/one relationships and total/partial participation.
4) Helped in compiling the overall database and ensuring sanity checks for different tables and corresponding code.

.

Bhavesh Jain (G2):

1) Working on nested queries, queries which give error, conversion of queries into relational algebra.
2) Ideating the tables, and contributing in definitions of tables and their domains.
3) I helped in the referential integrity solution.
4) I helped in user defined datatypes.
5) Worked on checking cases where referential integrity is violated.

Rahul Chembakasseril (G2):

1) I worked on setting up the user, tables, views as part of G2's responsibility. I helped implement and test the granting permissions and revoking permission component of the assignment.
2) I also helped identify and test the four situations where referential integrity is violated. I also helped implement the solutions to these situations and proved that the solutions work.
3) I also helped ideate over multiple queries for our database as part of the combined work of G1 and G2.

Kanishk Singhal(G2):

1) I wrote queries which gives constraint violations as part in G1 + G2.
2) I wrote queries which we ran in SQL to satisfy the mentioned properties of using join, nested, case.
3) I helped in testing the granting permissions and revoking permission as G2 member.
4) Brainstorming and ideating to modify queries and writing their documentation was also a task for me mentioned G1+G2's responsibilities.

Kalash Kankaria (G2):

1) Worked with members to find queries which give constraint violations as was the part of work of G1+G2.
2) Converted various nested queries into its specific relational algebra and displayed them as a part of G1+G2.
3) Helped in writing nested queries which we ran in SQL to satisfy the mentioned properties of using join and case.
4) Found and implemented indexing in the relations and explained the reasons wherever used, this was a part of G1(Helped Dhairya).

Inderjeet Singh (G2):

1) Helped in setting up the user, views as was the part of G2's responsibility.

2) Wrote SQL queries of nested operations that would follow certain specifications as was the part of G1 + G2.
3) Wrote SQL queries that would produce errors due to constraints in our database as was the part of G1+G2.
4) Helped in writing the relational algebra for the various queries that I wrote.