

# Bio-Log Database Project

By Ary Hernandez, Jacquelyn Johnson, and Andrew Samuel

Spring 2020

Professor Yuan

## Table of Contents

<b>Abstract.....</b>	<b>3</b>
<b>Mission Statement.....</b>	<b>3</b>
<b>Mission Objectives.....</b>	<b>3</b>
<b>Major User Views.....</b>	<b>5</b>
<b>E/R Diagram.....</b>	<b>6</b>
<b>Normalization.....</b>	<b>9</b>
<b>Use Cases.....</b>	<b>12</b>
<b>Test &amp; Outputs.....</b>	<b>30</b>
<b>Conclusion.....</b>	<b>39</b>
<b>References.....</b>	<b>40</b>

**Abstract**

The proposed database will allow for a logical and concise storage of data collected from the Gen III MicroPlate Bio-log lab being conducted at Lone Star College-Montgomery. This database application will allow students and professors the ability to query through large amounts of data collected, draw inferences from the data to extend hypothesis, and further the research in the field of microbiology pertaining to this specific lab. The platform is a MySQL database with the application GUI written in VB using Windows Forms App(.NET Framework). The result is a database which enables students and professors to enter, manage, and query data through a user-friendly interface.

**Mission Statement**

The purpose of the Bio-log database project is to maintain the data collected during the course of performing the Gen III Microplate Lab being conducted at Lone Star College-Montgomery under the direction of Dr. Julie Harless. The Microplate Lab involved the study of microorganisms and reading their reactions and electrical outputs. This database will allow for such data to be organized in a way that hypothesis can be formulated and tested thoroughly. The students and professors will be able to query the data in such a way to allow for new labs to be developed and further their research in the study of microorganisms and their behaviors.

**Mission Objectives**

To maintain (enter, update, and delete) data on State

To maintain (enter, update, and delete) data on University

To maintain (enter, update, and delete) data on Campus

To maintain (enter, update, and delete) data on Professor

To maintain (enter, update, and delete) data on Class

To maintain (enter, update, and delete) data on Semester

To maintain (enter, update, and delete) data on Location

To maintain (enter, update, and delete) data on Student

To maintain (enter and update) data on Data

To perform searches on State

To perform searches on University

To perform searches on Campus

To perform searches on Class

To perform searches on Semester

To perform searches on Location

To perform searches on Student

To perform searches on Data

To track status of Data

To report on State

To report on University

To report on Campus

To report on Professor

To report on Class

To report on Semester

To report on Location

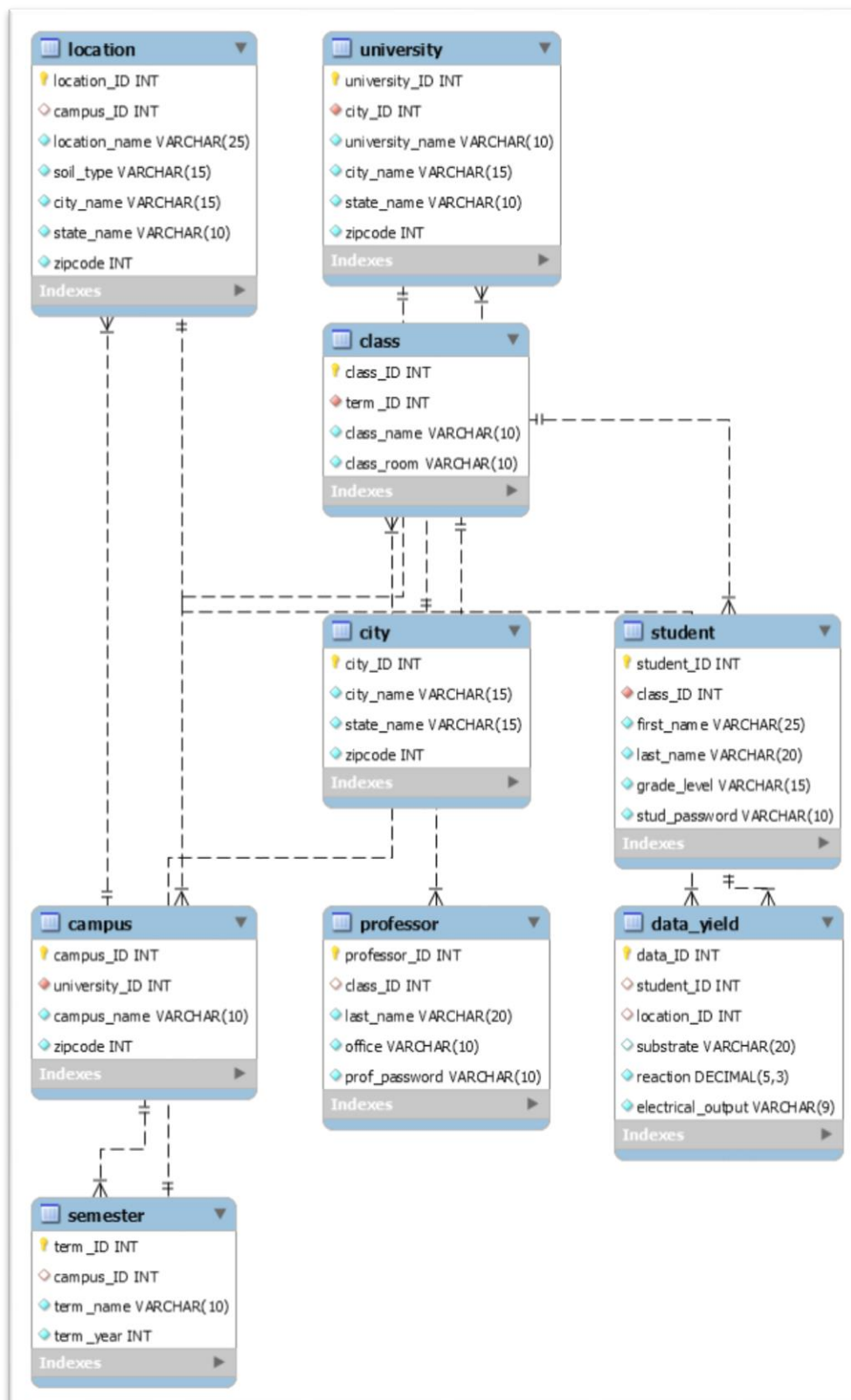
To report on Student

To report on Data

**Major User Views**

Data		Student			Professor		
		Update	Delete	View	Update	Delete	View
City	<u>StateName</u>			X	X	X	X
	<u>CityName</u>			X	X	X	X
University	<u>UniversityID</u>				X	X	X
	<u>UniversityName</u>			X	X	X	X
Campus	<u>CampusName</u>			X	X	X	X
Location	<u>Location_Name</u>	X	X	X	X	X	X
	<u>SoilType</u>			X	X	X	X
Professor	<u>ProfessorID</u>						X
	<u>ProfessorName</u>			X	X		X
	<u>OfficeNo.</u>			X	X		X
	Phone			X	X		
	Email			X	X		
Class	<u>ClassID</u>			X	X	X	X
	<u>ClassName</u>			X	X	X	X
	<u>ClassTime</u>			X	X	X	X
Student	<u>StudentID</u>	X		X	X	X	X
	<u>StudentName</u>			X	X	X	X
	Email			X	X	X	
Semester	<u>SemesterNo.</u>			X	X	X	X
	<u>SemesterYear</u>			X	X	X	
Data	<u>ReactionType</u>	X		X	X	X	X
	<u>ElectricalOutput</u>	X		X	X	X	X
	<u>Substrate</u>			X	X	X	

## E/R Diagram



**Table Explanations:****1. City**

The purpose of the ‘City’ table is to hold the city name and state name where the university is located. Since there can be more than one university in a city, this table is useful when querying information. This table allows for the expansion of the use of this application/database as this lab is available to be performed at any university interested in the electrical output of bacteria in different soil types. The primary key for this table is CityID and the foreign key for this table is UniversityID.

**2. University**

The purpose of the ‘University’ table is to hold the information about the university at which the lab is being conducted. This table will hold the university name and has the foreign key CityID. The primary key for the table is UniversityID.

**3. Campus**

The purpose of the ‘Campus’ table is to hold information about the exact campus at which the lab is being performed. The logic behind this is as follows: University of Houston has campuses in a variety of places, Clear Lake, Downtown, Victoria, etc. There are many universities that also have multiple campuses, like Lone Star College System, for which this database is being designed. By implementing a ‘Campus’ table, the data collected at the different campuses can be kept distinct. The primary key for this table is campusID and the foreign keys are universityID, classID, and professorID.

**4. Location**

The purpose of the ‘Location’ table is to hold the data for the different, distinct locations that will be studied at each campus. The purpose of the lab itself is to “see” the reactions to the substrate that the bacteria have where the bacteria were collected from different soil conditions. For example, samples could be collected from pond mud, dry dirt, and loamy soil respectively. As each of the locations would be unique, there is a need for a table to hold the data collected about each one. The primary key for this table is locationID and the foreign key is campusID.

**5. Student**

The purpose of the ‘Student’ table is to hold identifying information about the student performing the lab and collecting the data. The information held in this table is the name of the student, what year they are in concerning their studies, their email, and the password that they will use to log in to the user interface and document the progress of the lab they are working on. As there can be many students who complete this lab across all of the possible campuses, this table is necessary. The primary key for this table is studentID and the foreign key for this table is classID.

**6. Professor**

The purpose of the ‘Professor’ table is to hold identifying information about the professor at the campus where the lab is being conducted. This table will contain the following information about the professor: name, office, email, and phone as query-able data. It is possible that there can be more than one professor at a campus that is assisting students in the collection of data and completion of the lab. The primary key of this table is professorID and the foreign key is classID.

## 7. Class

The purpose of the 'Class' table is to hold information about the classes to which the students belong. This is necessary as there can be multiple sections of a class offered at a time in which there are students in different sections that are performing the same lab. The class name will also contain the section number within it so as to differentiate between the unique classes. The primary key for this table is classID and the foreign keys are campusID and semesterNo.

## 8. Semester

The purpose of the 'Semester' table is to hold identifying information concerning the semester during which the lab was performed: fall, spring, and summer. It will also contain the year during which the semester occurs to further differentiate the time and aid in future comparative queries of the data. The primary key for this table is semesterNo and the foreign keys for this table are ProfessorID, classID, and campusID.

## 9. Data

The purpose of the 'Data' table is to hold all the data that is collected over the course of the lab. This table will be modified the most for each lab, mainly through updating reactions and the resulting electrical output for the substrate being tested based on the location that the sample was collected from. The primary key for this table is dataID and the foreign keys are studentID and locationID.



## Normalization

### City Table

1NF: Each value in 1NF should only have one single atomic value so the city table complies with 1NF standards; therefore, it is 1NF

2NF: No non-prime attribute and partial dependency must exist in the table to be in 2NF form, the non-key columns are dependent on the primary key, cityID. Additionally, the primary key is a single column identifier, therefore it is 2NF.

3NF: A table is in 3NF form if there is no transitive dependency so, city table is in 3NF since no column depends on another.

BCNF: Boyce-Codd Normal Form states that every functional dependency  $A \rightarrow B$ , A must be a super key.

Primary Key: cityID

### University Table

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: All attributes are independent of the primary key, university\_ID, therefore it is in 2NF.

3NF: Primary key university\_ID; all other attributes do not rely on it; therefore, it is in 3NF form.

BCNF: Only primary key is university\_ID, with city\_ID as a foreign key to University table but a primary for the city table. Thus, BCNF is achieved.

Primary Key: universityID

Foreign Key: cityID

### Campus Table

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: No partial dependency in attributes is observed, thus 2NF is achieved.

3NF: No transitive functional dependencies are observed in the campus table; hence, it is in 3NF.

BCNF: Since for every functional dependency requires  $A \rightarrow B$ , A there is a super key present, which is campusID. Hence, BCNF form.

Primary Key: campusID

Foreign Key: universityID

**Location Table**

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: No partial dependencies are present in the location table; there contains a single column primary key (locationID).

3NF: No transitive dependencies are present in the location table. All columns are not transitively dependent on the primary key (locationID). Because of this, it is in 3NF.

BCNF: Because for every functional dependency  $A \rightarrow B$ , B there is a super key in this table, this table is Boyce-Codd Normal Form.

Primary Key: locationID

Foreign Key: campusID

**Student Table**

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: All columns in student table fully depend on studentID primary key, therefore no partial dependency is present. Additionally, the primary key is presenting a single column. Thus, the table is in 2NF.

3NF: All columns are not transitively dependent on the primary key (studentID), therefore 3NF is active.

BCNF: Following the functional dependency standard of  $A \rightarrow B$ , A there exists a super key, because of this, BCNF is achieved.

Primary Key: studentID

Foreign Key: classID

**Professor Table**

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: Single column primary key is present, and no partial dependency is present since all columns depend on table primary key. Therefore, 2NF is achieved in the professor table.

3NF: All columns are not transitively dependent on the primary key (professorID) thus, 3NF is achieved.

BCNF: An extension of 3NF, BCNF is reached because every functional dependency implies  $A \rightarrow B$ , A implying a super key, which exists. Therefore, BCNF is met.

Primary Key: professorID

Foreign Key: classID

**Class Table**

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: No partial dependency is present in Class table. All columns depend on the table's primary key and because there is only one column holding the primary key. Because of this, Class table is in 2NF.

3NF: Since class\_ID  $\rightarrow$  rest of all attributes, no transitive functional dependencies are in the table. Therefore 3NF is set.

**BCNF:** Class table is in BCNF because class\_ID is the super key in relation to class table.

Primary Key: classID

Foreign Key: termNo

**Semester Table**

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: No partial dependency is present in Semester table. Additionally, a single column primary key proves true in the table. Therefore, the table is in 2NF

3NF: The table has no transitive functional dependencies because no non-prime attribute depends on other non-prime attribute. Instead it depends upon the prime attributes or primary key (semesterNo)

**BCNF:** Since for every functional dependency ( $A \rightarrow B$ ), A there is a super key in this table, therefore table is in BCNF.

Primary Key: termNo

Foreign Key: campusID

**Data Table**

1NF: There is only one single atomic value in each attribute of the table, so it is in 1NF.

2NF: No partial dependency is in the data table. Also, a single column key is present (data\_ID) for the table. Thus, 2NF is achieved,

3NF: No partial dependency found in the table. Therefore, 3NF is active. Additionally, a column's value does not rely upon another column through a second intermediate column.

**BCNF:** Since for every functional dependency ( $A \rightarrow B$ ), A there is a super key in this table, therefore table is in BCNF.

Primary Key: dataID

Foreign Key: studentID

Foreign Key: locationID

## Use Cases

### Entity: City

#### 1. Insert New City

Actor: Professor

Steps:

- User clicks “New City” button
- User sees a form appear to collect information
- User inputs information for the State name and the city name
- User clicks the “Continue” button

MySQL statement:

```
INSERT INTO City (stateName, cityName) VALUES (stateName, cityName);
```

Explanation: This use case allows the Professor to add a new city where a new campus is located that will be collecting data for this lab.

#### 2. Delete City

Actor: Professor

Steps:

- User clicks “Delete City” button
- User sees a window displaying all cities in the database
- User enters the cityID number in a text field that they wish to delete
- User clicks “Delete” button
- User sees a dialog box warning that this action cannot be undone
- User clicks “Confirm Deletion” button on the dialog box

MySQL statement:

```
DELETE FROM City WHERE cityID = cityID;
```

Explanation: This use case will allow for the deletion of a city

#### 3. View Cities

Actor: Professor/ Student

Steps:

- User clicks “Show Cities” button
- User sees a window that displays the contents of the City table

MySQL statement:

```
SELECT (cityName, stateName) FROM City;
```

Explanation: This use case allows the system users to view the cities in the table

#### 4. Update City

Actor: Professor

Steps:

- User clicks “Show Cities” button
- User sees a window that displays the contents of the City table
- User clicks “Update City” button
- User inputs the cityID of the city to be updated
- User is prompted for the new cityName and stateName
- User inputs newCityName and newStateName into labeled fields
- User clicks “Update Record”

MySQL statement:

```
UPDATE City SET cityName = newCityName WHERE cityName = cityName_original  
UPDATE City SET stateName=newStateName WHERE stateName=stateName_original
```

Explanation: This use case allows a professor to correct the city/state of a record

#### 5. Count Cities

Actor: Professor

Steps:

- User clicks “Count Cities” button
- User sees a window that displays the number of cities by state

MySQL statement:

```
SELECT COUNT(cityID), stateName  
FROM City  
GROUP BY stateName
```

Explanation: This use case displays the number of cities by state

**Entity: University**

#### 6. Insert New University

Actor: Professor

Steps:

- User clicks “New University” button
- User sees a form appear to collect information
- User inputs information for the university name
- User clicks the “Continue” button

MySQL statement:

```
INSERT INTO University (universityName) VALUES (universityName);
```

Explanation: This use case allows a professor to add a new university record to the University table

#### 7. Delete University

Actor: Professor

Steps:

- User clicks “Delete University” button
- User sees a list of universities
- User inputs the universityID of the university to delete
- User clicks “Delete” button
- User sees a message dialog box informing that this action cannot be undone
- User clicks “Confirm Deletion” button

MySQL statement:

```
DELETE FROM University WHERE universityID=universityID;
```

Explanation: This use case allows a professor to delete a university from the University table

### 8. View Universities

Actor: Professor/Student

Steps:

- User clicks “Show Universities” button
- User sees window showing all universities in the University table

MySQL statement:

```
SELECT * FROM University;
```

### 9. Count Universities

Actor: Professor/Student

Steps:

- User clicks “Count Universities” button
- User is displayed a window with number of universities by state

MySQL statement:

```
SELECT COUNT(uniID) stateName  
FROM UNIVERSITY  
GROUP BY stateName
```

Explanation: User can see the number of universities by state.

### 10. Update University

Actor: Professor

Steps:

- User clicks “Show Universities” button
- User sees window showing all universities in the University table
- User clicks “Update University” button
- User inputs universityID of university record to update
- User is prompted for new university name

- User clicks “Update Record” button

MySQL statement:

```
UPDATE University SET universityName=newUnivName WHERE  
universityName=universityName_original;
```

Explanation: This use case allows the professor to update the name of a university in the University table

## Entity: Campus

### 11. Insert New Campus

Actor: Professor

Steps:

- User clicks “Add Campus” button
- User selects the city where the new campus is located
- User selects the university the campus belongs to; universityID is captured
- User inputs new campus name
- User clicks “Continue” button

MySQL statement:

```
INSERT INTO Campus (campusName, universityID) VALUES (campusName, universityID);
```

Explanation: This use case allows a campus to be added to the Campus table and foreign key to be linked at creation of record

### 12. Delete Campus

Actor: Professor

Steps:

- User clicks “Delete Campus” button
- User sees a window displaying the contents of the Campus table
- User inputs the campusID of the campus to be deleted
- User clicks “Delete” button
- User sees a message dialog box informing the user that this action is not reversible
- User clicks “Confirm Deletion” button

MySQL statement:

```
DELETE FROM Campus WHERE campusID=campusID;
```

Explanation: This use case allows a professor to delete a campus from the Campus table

### 13. View Campuses

Actor: Professor/Student

Steps:

- User clicks “View Campuses” button

- User sees a window displaying the contents of the Campus table

MySQL statement:

```
SELECT * FROM Campus;
```

Explanation: This use case will display the contents of the Campus table for all users

#### 14. Count Campuses

Actor: Professor/Student

Steps:

- User selects “Count Campuses” button
- User will be presented a window with number of campuses by university

MySQL statement:

```
SELECT COUNT(campusID), universityName  
FROM Campus  
GROUP BY universityName
```

Explanation: With this, the user can see the number of campuses by university system. Some university systems have multiple campuses. Ex. (UH system)

#### 15. Update Campus

Actor: Professor

Steps:

- User clicks “View Campuses” button
- User sees a window displaying the contents of the Campus table
- User clicks “Update Campus” button
- User is prompted for the campusID of the campus to update
- User inputs the new campus name
- User clicks “Update Record” button

MySQL statement:

```
UPDATE Campus SET campusName=newCampusName WHERE  
campusName=campusName_original;
```

Explanation: This use case updates the name of the campus

### **Entity: Location**

#### 16. Insert New Location

Actor: Professor

Steps:

- User clicks “Add New Location” button
- User sees a form pop up to collect information
- User selects the campus on which the location is found; campusID captured
- User inputs the name of the location and soil type found there
- User clicks “Continue” button



MySQL statement:

```
INSERT INTO Location (locationName, soilType, campusID) VALUES (locationName, soilType, campusID);
```

Explanation: This use case adds a new location to the Location table and foreign key of campusID to be linked at creation of record

### 17. Delete a Location

Actor: Professor

Steps:

- User clicks “Delete Location” button
- User inputs locationID of location to delete
- User clicks “Delete” button
- User sees a message dialog box informing them that the action cannot be reversed
- User clicks “Confirm Deletion” button

MySQL statement:

```
DELETE FROM Location WHERE locationID=locationID;
```

Explanation: This use case allows a location to be deleted

### 18. View Locations

Actor: Professor/Student

Steps:

- User clicks “View Locations” button
- User sees a window displaying the contents of the Location table

MySQL statement:

```
SELECT * FROM Location;
```

Explanation: This use case allows a user to view all the locations in the Location table

### 19. Update Location Name

Actor: Professor

Steps:

- User clicks “View Locations” button
- User sees a window displaying the contents of the Location table
- User clicks “Update Location” button
- User is prompted for the locationID of the location record to be updated
- User inputs the locationID
- User inputs new location name
- User clicks “Update Record” button

MySQL statement:

```
UPDATE Location SET locationName=newLocationName WHERE locationName=locationName_original;
```

Explanation: This use case allows a location name to be updated by the professor

## 20. Update Location Soil Type

Actor: Professor/Student

Steps:

- User clicks “View Locations” button
- User sees a window displaying the contents of the Location table
- User clicks “Update Location Soil Type” button
- User is prompted for the locationID of the location record to be updated
- User inputs the locationID
- User inputs new soil type
- User clicks “Update Record” button

MySQL statement:

```
UPDATE Location SET soilType=newSoilType WHERE soilType=soilType_original;
```

Explanation: This use case allows a soil type to be updated at a location by the professor or the student

## 21. Delete a Location Soil Type

Actor: Professor

Steps:

- User clicks “Delete Soil Type” button
- User inputs locationID of location where soil type needs to be deleted
- User sees soil type listed for that location
- User inputs the name of the soil type
- User clicks “Delete” button
- User sees a message dialog box informing them that the action cannot be reversed
- User clicks “Confirm Deletion” button

MySQL statement:

```
UPDATE Location SET soilType=NULL WHERE soilType=soilType_original;
```

Explanation: This use case allows a soil type to appear deleted to the user and inserts a NULL value in its place in the Location table

## 22. Count Soil Type

Actor: Professor

Steps:

- User clicks on “Count Soil Types” button
- User is displayed the count of soil types by location

mySQL statement:

```
SELECT COUNT(locationID), soil_type
FROM Location
GROUP BY soil_type
```

Explanation: The user can see the types of soil found by location.

### **Entity: Professor**

#### 23. Insert New Professor

Actor: Professor

Steps:

- User clicks “Add Professor” button
- User verifies their own credentials to maintain integrity of the database
- User sees a form pop up to collect information
- User inputs professor name, office, phone, and email
- User clicks “Continue”

MySQL statement:

```
INSERT INTO Professor (professorName, office, phoneNumber, email) VALUES  
(professorName, office, phoneNumber, email);
```

Explanation: This use case allows for the creation of a new record into the Professor table

#### 24. Delete Professor

Actor: Professor

Steps:

- User clicks “Delete Professor”
- User verifies their own credentials to maintain integrity of the database
- User sees a window displaying the current records in the Professor table
- User is prompted for the professorID to delete
- User inputs the professorID
- User clicks “Delete”
- User sees a message dialog box informing the user that this action is irreversible
- User clicks “Confirm Deletion” button

MySQL statement:

```
DELETE FROM Professor WHERE professorID=professorID;
```

Explanation: This use case allows a record to be deleted from the professor table

#### 25. View Professor Information

Actor: Professor/Student

Steps:

- User clicks “View Professors” button
- User sees a window displaying all the professors in the Professor table

MySQL statement:

```
SELECT * FROM Professor;
```

Explanation: This use case allows users to view the records in the Professor table

## 26. Update Professor Information

Actor: Professor

Steps:

- User clicks “View Professors” button
- User sees a window displaying all the professors in the Professor table
- User clicks “Update Professor” button
- User verifies their own credentials to maintain integrity of the database
- User is prompted for the professorID of the record to update
- User inputs the professorID of the record to update
- User selects the information to be updated
- User inputs the new values for the information that needs to be updated
- User clicks “Update Record”

MySQL statement:

```
UPDATE Professor SET professorName=newProfessorName WHERE
professorName=professorName_original;
UPDATE Professor SET office=newOffice WHERE office=office_original;
UPDATE Professor SET phoneNumber=newPhone WHERE
phoneNumber=phoneNumber_original;
UPDATE Professor SET email=newEmail WHERE email=email_original;
```

Explanation: This use case allows for a professor record to be updated. The MySQL statement that will be executed depends wholly on the selected information and will only update the information that is selected to be updated. This will be verified and handled programmatically on the GUI side.

## 27. Count Professor

Actor: Professor/Student

Steps:

- User clicks “Count Professor” button
- User is displayed number of classes is professor teaching by course ID

MySQL statement:

```
SELECT COUNT(professor_ID)
FROM Professor
GROUP BY classID;
```

Explanation: Shows number of professors teaching a specific class

**Entity: Class**

## 28. Insert New Class

Actor: Professor

Steps:

- User clicks “Add New Class” button

- User sees a form pop up to collect information for the new record
- User selects the campus and semester from cascading menus; campusID and semesterID are captured
- User inputs class name and class time
- User clicks “Add”

MySQL statement:

```
INSERT INTO Class (className, classTime, campusID, semesterID) VALUES (className, classTime, campusID, semesterID);
```

Explanation: This use case adds a new record to the Class table and links the foreign keys for campusID and semesterID at creation.

### 29. Delete Class

Actor: Professor

Steps:

- User clicks “Delete Class”
- User verifies their credentials to maintain database integrity
- User sees window displaying contents of Class table
- User prompted for classID of class to delete
- User inputs classID
- User clicks “Delete”
- User sees pop up message informing the action is irreversible
- User clicks “Confirm Deletion”

MySQL statement:

```
DELETE FROM Class WHERE classID=classID;
```

Explanation: Allows for deletion of a class record from the Class table

### 30. View Classes

Actor: Professor/Student

Steps:

- User clicks “View Classes”
- User sees window displaying all records in Class table

MySQL statement:

```
SELECT * FROM Class;
```

### 31. Update Class

Actor: Professor

Steps:

- User clicks “View Classes”
- User sees window displaying all records in Class table
- User clicks “Update Class”
- User verifies credentials
- User prompted for classID

- User inputs classID
- User inputs new class name and class time
- User clicks “Update Record”

MySQL statement:

```
UPDATE Class SET className=newClassName WHERE className=className_original;  
UPDATE Class SET classTime=newClassTime WHERE classTime=classTime_original;
```

Explanation: Allows for the name and time of a class to be updated

### 32. Count Classes

Actor: Professor/Student

Steps:

- User clicks on “Count Classes” button
- User is displayed count of classes by semester

MySQL statement:

```
SELECT COUNT(classID), semesterNo  
FROM Class  
GROUP BY semesterNo
```

Explanation: The user can see the number of classes in each campus

### **Entity: Student**

### 33. Insert New Student

Actor: Professor

Steps:

- User clicks “Add New Student”
- User verifies credentials
- User sees a form pop up to collect information for record
- User inputs student name (first and last), grade level, and email
- User selects the class to which the new student belongs; classID captured
- User clicks “Continue”

MySQL statement:

```
INSERT INTO Student (firstName, lastName, gradeLevel, email) VALUES (firstName,  
lastName, gradeLevel, email);
```

Explanation: creates a new record in the Student table and links classID as foreign key to the record

### 34. Delete Student

Actor: Professor

Steps:

- User clicks “Delete Student”

- User verifies credentials
- User sees all student records from Student table
- User prompted for studentID to delete
- User inputs studentID
- User clicks “Delete”
- User sees message stating the action is irreversible
- User clicks “Confirm Deletion”

MySQL statement:

**DELETE FROM** Student **WHERE** studentID=studentID;

Explanation: Allows a student record to be deleted

### 35. View Students

Actor: Professor/Student

Steps:

- User clicks “View Students”
- User sees window displaying contents of Student table

MySQL statement:

**SELECT \* FROM** Student;

Explanation: Displays contents of Student table

### 36. Update Student

Actor: Professor/Student

Steps:

- User clicks “View Students”
- User sees window displaying contents of Student table
- User selects record to update
- User verifies credentials; if student actor, must verify that record is their own
- User selects information to update
- User inputs new information for selected fields
- User clicks “Update Record”

MySQL statement:

**UPDATE** Student **SET** firstName=newFirstName **WHERE** firstName=firstName\_origin;

**UPDATE** Student **SET** lastName=newLastName **WHERE** lastName=lastName\_origin;

**UPDATE** Student **SET** gradeLevel=newGrLevel **WHERE** gradeLevel=grLevel\_origin;

**UPDATE** Student **SET** email=newEmail **WHERE** email=email\_origin;

Explanation: Allows a student/professor to update information in a student record

### 37. Count Students

Actor: Professor/Student

Steps:

- User clicks “Count Students”
- User sees window displaying number of students by grade level

MySQL statement:

```
SELECT COUNT(studentID), grade_level
FROM Student
GROUP BY grade_level
```

Explanation: The user will be able to see the number of freshmen, sophomores, juniors and seniors per campus

### Entity: Semester

#### 38. Insert New Semester

Actor: Professor

Steps:

- User clicks “Add New Semester”
- User selects professor, campus, and class from menus; professorID, campusID, classID captured
- User inputs semester year and semester season
- User clicks “Continue”

MySQL statement:

```
INSERT INTO Semester (semesterYear, semesterSeason, professorID, classID, campusID)
VALUES (semesterYear, semesterSeason, professorID, classID, campusID);
```

Explanation: creates new record in Semester table and links foreign keys for professorID, classID, campusID upon creation

#### 39. Delete Semester

Actor: Professor

Steps:

- User clicks “Delete Semester”
- User verifies credentials
- User sees contents of Semester table
- User inputs semesterNO of semester to delete
- User clicks “Delete”
- User informed that action is irreversible
- User clicks “Confirm Deletion”

MySQL statement:

```
DELETE FROM Semester WHERE semesterNO=semesterNO;
```

Explanation: Deletes record from Semester table

#### 40. View Semesters

Actor: Professor

Steps:

- User clicks “View Semesters”
- User sees window displaying contents of Semester table



MySQL statement:

```
SELECT * FROM Semester;
```

Explanation: Displays contents of Semester table

#### 41. Update Semester

Actor: Professor

Steps:

- User clicks “View Semesters”
- User sees window displaying contents of Semester table
- User clicks “Update Semester”
- User verifies credentials
- User inputs new semester year and new semester season
- User clicks “Update Record”

MySQL statement:

```
UPDATE Semester SET semesterYear=newSemYear WHERE semesterYear=semYear_origin;  
UPDATE Semester SET semesterSeason=newSeason WHERE  
semesterSeason=semesterSeason_origin;
```

Explanation: Updates information in semester record

#### 42. Count Semester

Actor: Professor/Student

Steps:

- User clicks “Count Semesters”
- User is displayed number of semesters by semester season (Spring, Fall or Summer)

MySQL statement:

```
SELECT COUNT(semesterNo), semesterSeason  
FROM Semester  
GROUP BY semesterSeason
```

Explanation: Shows number of semesters grouped by the season.

### **Entity: Data**

#### 43. Insert New Lab Data

Actor: Student

Steps:

- User clicks “New Lab Data”
- User inputs their studentID
- User selects semester and location from menu; semesterNO and locationID captured
- User sees form pop up for information collection
- User inputs reaction type and electrical output for selected substrate

- User clicks “Continue”

MySQL statement:

```
INSERT INTO Data (reactionType, electricalOutput, substrate, studentID, semesterNO, locationID) VALUES (reactionType, electricalOutput, substrate, studentID, semesterNO, locationID);
```

Explanation: Adds a new record to Data that holds the lab data being collected by that student and links the foreign keys studentID, semesterNO, and locationID at creation.

#### 44. Delete Data

Actor: Professor

Steps:

- User clicks “Delete Lab Data”
- User verifies credentials
- User sees window displaying contents from Data table
- User inputs dataID of record to delete
- User clicks “Delete”
- User informed action is irreversible
- User clicks “Confirm Deletion”

MySQL statement:

```
DELETE FROM Data WHERE dataID=dataID;
```

Explanation: Deletes record from table

#### 45. View Lab Data

Actor: Professor/Student

Steps:

- User clicks “View Lab Data”
- User selects studentID
- User sees window displaying contents of Data table for selected studentID

MySQL statement:

```
SELECT * FROM Data WHERE studentID=studentID;
```

Explanation: Displays lab data of selected studentID from Data table

#### 46. Update Lab Data

Actor: Professor/ Student

Steps:

- User clicks “View Lab Data”
- User selects studentID
- User sees window displaying contents of Data table for selected studentID
- User clicks “Update Lab Data”
- User selects dataID for record to update

- User inputs data for selected substrate
- User clicks “Update Record”

MySQL statement:

```
UPDATE Data SET reactionType=newReaction WHERE (reactionType=reactioType_origin,  
substrate=substrate_origin);  
UPDATE Date SET electricalOutput=newElec WHERE  
(electricalOutput=electricalOutput_origin, substrate=substrate_origin);
```

Explanation: allows a user to update a record

#### 47. Count Data

Actor: Professor/Student

Steps:

- User clicks “Count Data” button
- User is displayed number of data organized by each reaction type

MySQL statement:

```
SELECT COUNT(dataID), reactionType  
FROM Data  
GROUP BY reactionType
```

Explanation: The user can the see the number of reactions by reaction type.

#### 48. Average Electrical Output

Actor: Professor/Student

Steps:

- User clicks “Average Electrical Output”
- User is displayed lists of Electrical outputs and their average

MySQL statement:

```
SELECT AVG (electricalOutput)  
FROM Data
```

Explanation: Display the averages of recorded electrical outputs

#### 49. Average Reaction Value

Actors: Professor/Student

Steps:

- User clicks “Average Reaction Value”
- User is displayed data of reaction value averages

MySQL statement:

```
SELECT AVG (reaction)  
FROM Data
```

Explanation: Display averages of reaction values to the user

### Join City & University

```
SELECT city_name.City, state_name.City, university_name.University, campus_name.Campus
FROM City JOIN University ON City.city_ID = University.city_ID
JOIN Campus ON University.university_ID = Campus.university_ID;
```

```
1 • SELECT city.city_name, city.state_name, university.university_name, campus.campus_name
2 FROM city JOIN university ON city.city_ID = university.city_ID
3 JOIN campus ON university.university_ID = campus.university_ID;
```

city_name	state_name	university_name	campus_name
Houston	Texas	University of Houston	Main
Houston	Texas	University of Houston	Northwest
San-Antonio	Texas	University of Houston-Downtown	Southeast

### Join Campus & Semester

```
SELECT campus_name.Campus, term_name.Semester, term_year.Semester, class_name.Class
FROM Campus JOIN Semester ON Campus.campus_ID = Semester.campus_ID
JOIN Class ON Semester.term_ID = Class.term_ID;
```

```
1 SELECT campus.campus_name, semester.term_name, semester.term_year, class.class_name
2 FROM campus JOIN semester ON campus.campus_ID = semester.campus_ID
3 JOIN class ON semester.term_ID = class.term_ID;
```

campus_name	term_name	term_year	class_name
Main	Fall	2020	Data Structures
Main	Fall	2021	Senior Seminar
Northwest	Spring	2021	Intro to C++

Join Semester and Class

```

SELECT term_name.Semester, term_year.Semester, class_ID.Class, last_name.Professor,
class_name.Class, class_room.Class
FROM Semester JOIN Class ON Semester.term_ID = Class.term_ID
JOIN Professors ON Class.class_ID = Professor.class_ID;

```

```

1 • SELECT semester.term_name, semester.term_year, class.class_ID, professor.last_name, class.class_name
2 FROM semester JOIN class ON semester.term_ID = class.term_ID
3 JOIN professor ON class.class_ID = professor.class_ID;

```

term_name	term_year	class_ID	last_name	class_name	class_room
Fall	2020	1100	Singh	Data Structures	A110
Fall	2020	1100	Harris	Data Structures	A110
Spring	2021	1310	Chang	Intro to C++	C213
Spring	2021	1310	Solbahr	Intro to C++	C213
Fall	2021	4320	Li	Senior Seminar	A450
Fall	2021	4320	Lin	Senior Seminar	A450

JoinClass & Student

```

SELECT class_name.Class, first_name.Student, last_name.Student, data_ID.Data_Yield,
substrate.Data_Yield, reaction.Data_Yield, electrical_output.Data_Yield
FROM Class JOIN Student ON Class.class_ID = Student.class_ID
JOIN Data_Yield ON Student.student_ID = Data_Yield.student_ID

```

```

1 • SELECT class.class_name, student.first_name, student.last_name, data_yield.data_ID, data_yield.subs
2 FROM class JOIN student ON class.class_ID = student.class_ID
3 JOIN data_yield ON student.student_ID = data_yield.student_ID;

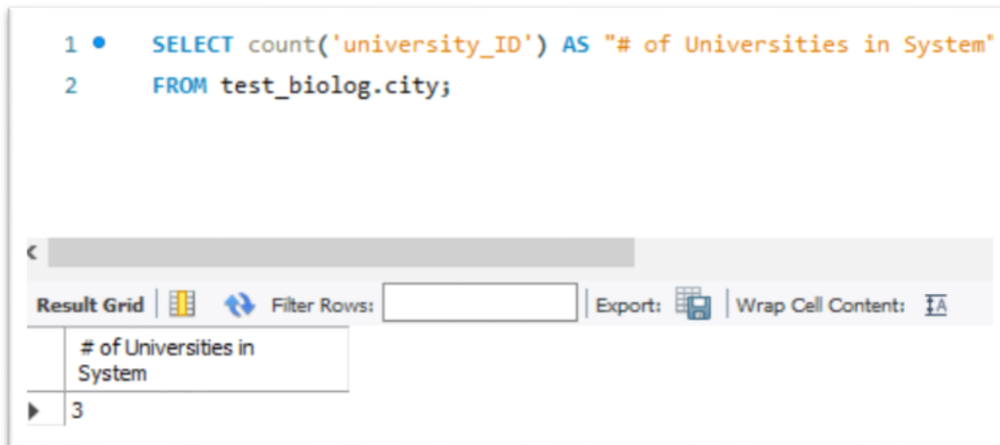
```

class_name	first_name	last_name	data_ID	substrate	reaction	electrical_output
Data Structures	Mark	Smith	6002	1	1.475	No Output
Senior Seminar	Zack	Martin	6001	-1	2.135	No Output
Senior Seminar	Zack	Martin	6003	-1	0.037	No Output

## Tests & Outputs

### Aggregate function for City table

```
SELECT count('university_ID') AS "# of Universities in System"  
FROM test_biolog.city;
```

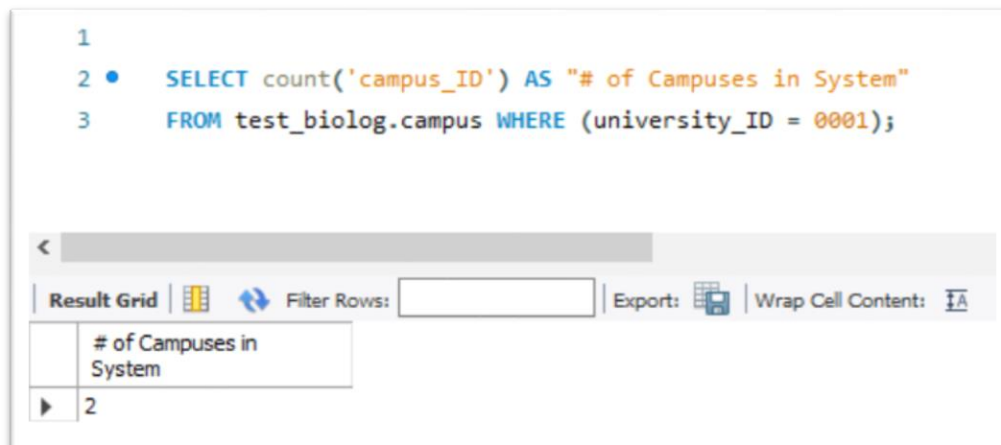


The screenshot shows a database query interface. At the top, the SQL query is displayed: `1 • SELECT count('university_ID') AS "# of Universities in System"` and `2 FROM test_biolog.city;`. Below the query, there is a toolbar with options: "Result Grid", "Filter Rows:", "Export:", and "Wrap Cell Content:". The "Result Grid" is active, showing a single row with the column header "# of Universities in System" and the value "3".

# of Universities in System
3

### Aggregate function for Campus table

```
SELECT count('campus_ID') AS "# of Campuses in System"  
FROM test_biolog.campus WHERE (university_ID = 001);
```

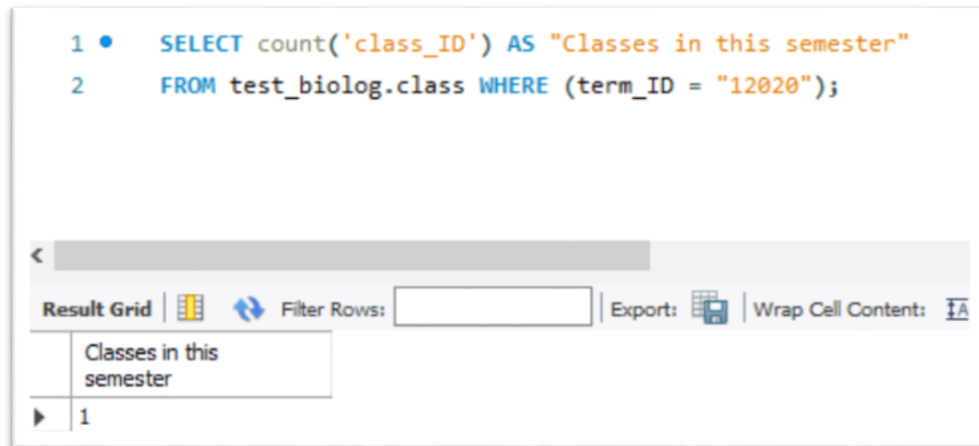


The screenshot shows a database query interface. At the top, the SQL query is displayed: `1`, `2 • SELECT count('campus_ID') AS "# of Campuses in System"`, and `3 FROM test_biolog.campus WHERE (university_ID = 001);`. Below the query, there is a toolbar with options: "Result Grid", "Filter Rows:", "Export:", and "Wrap Cell Content:". The "Result Grid" is active, showing a single row with the column header "# of Campuses in System" and the value "2".

# of Campuses in System
2

**Aggregate function for Class table**

```
SELECT count('class_ID') AS "Classes in this semester"  
FROM test_biolog.class WHERE (term_ID = "12020");
```

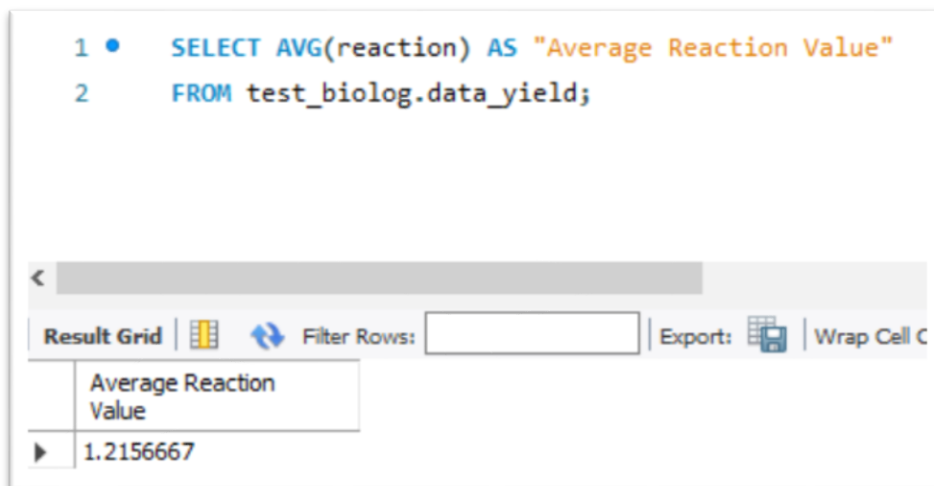


```
1 • SELECT count('class_ID') AS "Classes in this semester"  
2   FROM test_biolog.class WHERE (term_ID = "12020");
```

	Classes in this semester
1	1

**Aggregate function for Data table**

```
SELECT AVG(reaction) AS "Average Reaction Value"  
FROM test_biolog.data_yield;
```

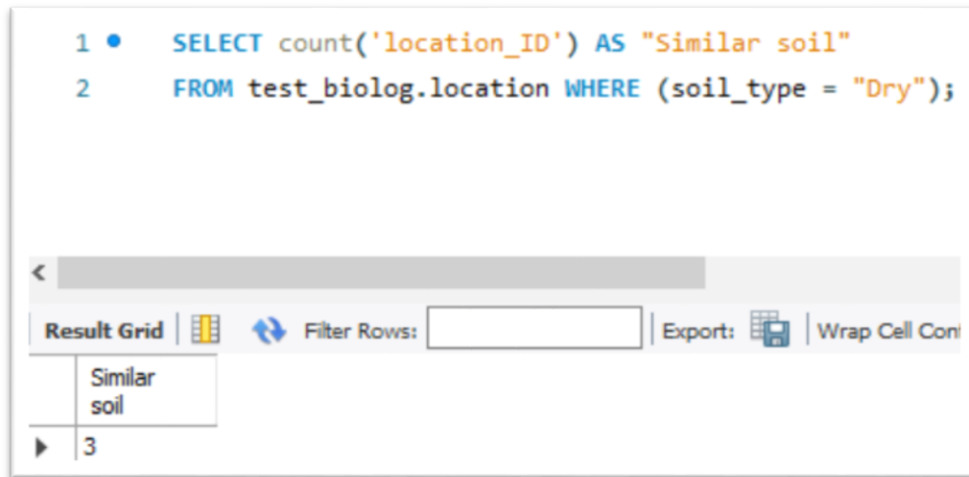


```
1 • SELECT AVG(reaction) AS "Average Reaction Value"  
2   FROM test_biolog.data_yield;
```

	Average Reaction Value
1	1.215667

**Aggregate function for Location table**

```
SELECT count('location_ID') AS "Similar Soil"  
FROM test_biolog.location WHERE (soil_type = "Dry");
```

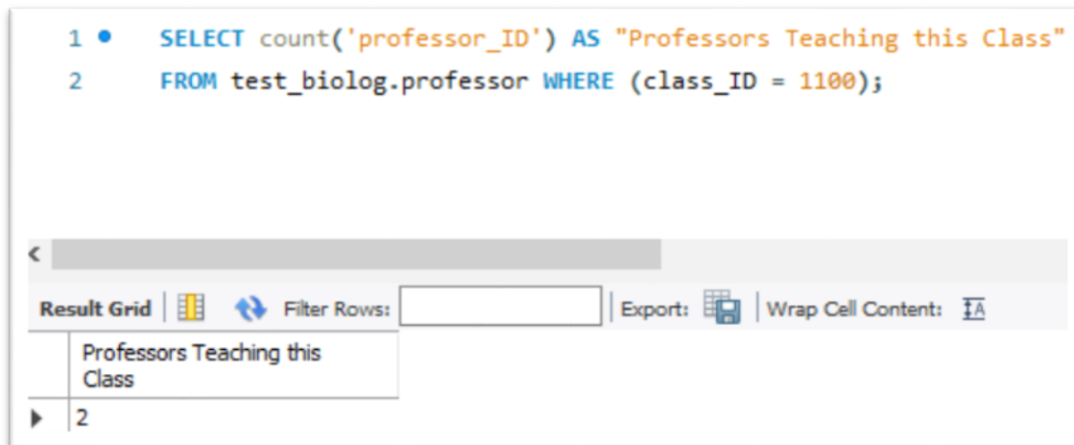


The screenshot shows a SQL query execution interface. The query is:   
1 • SELECT count('location\_ID') AS "Similar soil"  
2 FROM test\_biolog.location WHERE (soil\_type = "Dry");  
Below the query, there is a toolbar with options: Result Grid, Filter Rows, Export, and Wrap Cell Content. The Result Grid shows a single row with the column header "Similar soil" and the value 3.

Similar soil
3

**Aggregate function for Professor table**

```
SELECT count('professor_ID') AS "Professors Teaching this Class"  
FROM test_biolog.professor WHERE (class_ID = 1100);
```



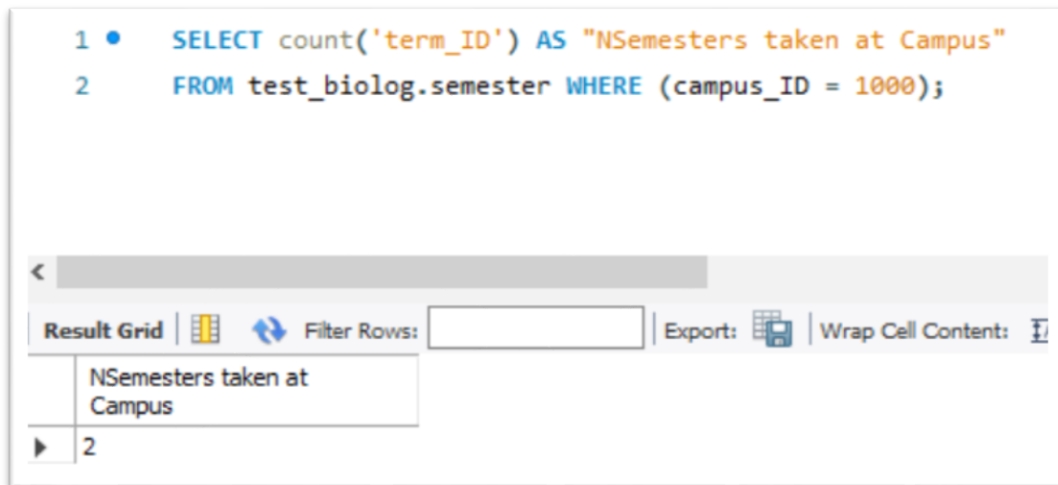
The screenshot shows a SQL query execution interface. The query is:   
1 • SELECT count('professor\_ID') AS "Professors Teaching this Class"  
2 FROM test\_biolog.professor WHERE (class\_ID = 1100);  
Below the query, there is a toolbar with options: Result Grid, Filter Rows, Export, and Wrap Cell Content. The Result Grid shows a single row with the column header "Professors Teaching this Class" and the value 2.

Professors Teaching this Class
2



**Aggregate function for Semester table**

```
SELECT count('term_ID') AS "Semesters taken at Campus"  
FROM test_biolog.semester WHERE (campus_ID = 1000);
```

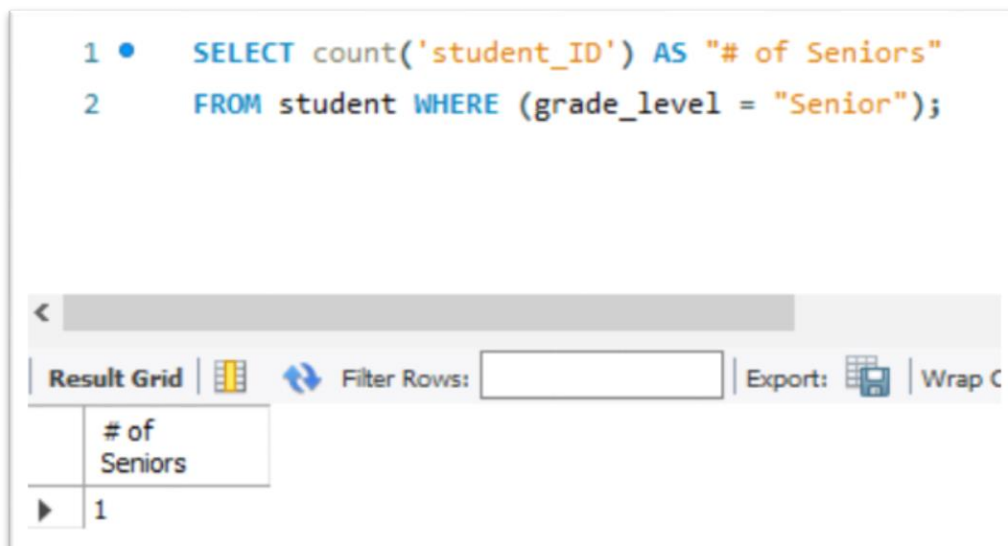


```
1 • SELECT count('term_ID') AS "NSemesters taken at Campus"  
2   FROM test_biolog.semester WHERE (campus_ID = 1000);
```

Result Grid	
NSemesters taken at Campus	2

**Aggregate function for Student table**

```
SELECT count('student_ID') AS "# of Seniors"  
FROM student WHERE (grade_level = "Senior");
```



```
1 • SELECT count('student_ID') AS "# of Seniors"  
2   FROM student WHERE (grade_level = "Senior");
```

Result Grid	
# of Seniors	1

**Aggregate function for University table**

SELECT count(\*) AS “# of Universities in City”

FROM test\_biolog.university WHERE (city\_name = “Houston”);

```
1 • SELECT count(*) AS "# of Universities in City"
2   FROM test_biolog.university WHERE (city_name = "Houston");
```

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

	# of Universities in City
▶	2

**View Campus table**

```
1 • SELECT * FROM test_biolog.campus;
```



Result Grid | Filter Rows:  | Edit:

	campus_ID	university_ID	campus_name	zipcode
▶	1000	1	Main	77400
	2000	1	Northwest	77200
	3000	2	Southeast	87440
✱	NULL	NULL	NULL	NULL

**View City table**

```
1  SELECT * FROM test_biolog.city;
```

<



Result Grid   Filter Rows:  | Edit

	city_ID	city_name	state_name	zipcode
▶	1001	Houston	Texas	77400
	1002	San-Antonio	Texas	87400
	1003	Sugar Land	Texas	78400
*	NULL	NULL	NULL	NULL

**View Class table**

```
1  SELECT * FROM test_biolog.class;
```

<

Result Grid   Filter Rows:  | Edit:

	class_ID	term_ID	class_name	class_room
▶	1100	12020	Data Structures	A110
	1310	12022	Intro to C++	C213
	4320	12021	Senior Seminar	A450
*	NULL	NULL	NULL	NULL

## View Data table

```
1 SELECT * FROM test_biolog.data_yield;
```

Result Grid

	data_ID	student_ID	location_ID	substrate	reaction	electrical_output
▶	6001	1	5001	-1	2.135	No Output
	6002	3	5001	1	1.475	No Output
	6003	1	5003	-1	0.037	No Output
✱	NULL	NULL	NULL	NULL	NULL	NULL

### View Location table

[illegible]

**View Professor table**

```
1  SELECT * FROM test_biolog.professor;
```

	professor_ID	class_ID	last_name	office	prof_password
▶	101	1100	Singh	B008	S101
	102	1310	Chang	A130	C102
	103	4320	Li	C230	L103
	104	4320	Lin	OMB745	Lin104
	105	1100	Harris	OMB722	H105
	106	1310	Soibahm	OMB742	S106
•	NULL	NULL	NULL	NULL	NULL






**View Semester table**

```
1  SELECT * FROM test_biolog.professor;
```

	professor_ID	class_ID	last_name	office	prof_password
▶	101	1100	Singh	B008	S101
	102	1310	Chang	A130	C102
	103	4320	Li	C230	L103
	104	4320	Lin	OMB745	Lin104
	105	1100	Harris	OMB722	H105
	106	1310	Soibahm	OMB742	S106
•	NULL	NULL	NULL	NULL	NULL

**View Student table**








1      **SELECT \* FROM test\_biolog.student**

<   Filter Rows:  | Edit:    | Export/Im

	student_ID	class_ID	first_name	last_name	grade_level	stud_password
▶	1	4320	Zack	Martin	Senior	Z001M
	2	1100	Andrew	Chen	Junior	A002C
	3	1100	Mark	Smith	Sophomore	M003S
	4	1310	Manny	Hernandez	Freshman	M004H
✱	NULL	NULL	NULL	NULL	NULL	NULL

**View University table**

1      **SELECT \* FROM test\_biolog.university;**

<   Filter Rows:  | Edit:    | Export/Import:  

	university_ID	city_ID	university_name	city_name	state_name	zipcode
▶	1	1001	University of Houston	Houston	Texas	77400
	2	1002	University of Houston-Downtown	Houston	Texas	77410
	3	1003	University of Texas-San Antonio	San-Antonio	Texas	87400
✱	NULL	NULL	NULL	NULL	NULL	NULL

## Conclusion

This database was created as way to feasibly store and update data records in a biology lab. With this, our goal was to help professors and students manage the entries of data pertaining to microorganisms while fulfilling all necessary assignment requirements given by professor Yuan. The database implementation made use of entity tables to ensure consistency across the system by modeling their relationships and functions.

The database was also intended to be implemented with a user interface using Visual Basic, but due to time constraints, will be completed after the due date of the project. This, however, was not required in the assignment and was planned in order to enhance the use of the database system.

Lastly, we believe the database can serve as a basic template for other projects that share similar needs. For example, it can be used for other kinds of data recording other than the Bio-log it was created for. Of course, several improvements can be made if given the additional time, considering this database was implemented within a semester term by a team with no prior database system experience. However, we are confident this project served as a good introduction to database management systems especially considering we were involved with it since the beginning, from the written SQL statements to the filling of the tables.

## References

“SQL Server Tutorial.” *Qhmit.com*, 2017, [qhmit.com/sql\\_server/tutorial/](http://qhmit.com/sql_server/tutorial/).

Connolly, Thomas M., and Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson Education Limited, 2015.

“DBMS - Normalization.” *Tutorialspoint*, [www.tutorialspoint.com/dbms/database\\_normalization.htm](http://www.tutorialspoint.com/dbms/database_normalization.htm).

“Database Normalization (Explained in Simple English).” *Essential SQL*, 23 Apr. 2020, [www.essentialsql.com/get-ready-to-learn-sql-database-normalization-explained-in-simple-english/](http://www.essentialsql.com/get-ready-to-learn-sql-database-normalization-explained-in-simple-english/).