**Name: Zijie Yu**

**Database Management Systems (CSC434)**

**Final Project**

**1) - Define the information content of your database.**

**a)-Define a set of entities and appropriate attributes for each entity. Minimum 10 entities.**

Note: The first attributes are primary key, the Blue color entity means entity created automaticly by Visual Paradigm because of many to many relationship (intermediate table).

* **Students:** Student\_ID; First\_Name; Last\_Name; Country; Age; Phone; Email; Expected\_graduation\_year; Address;
* **Major**: Major\_Name; Department; Minimum\_GPA; DepartmentId
* **Students\_Major:** Students\_StudentId; Major\_MajorId
* **Course:** CourseId; Course\_Name; Seats Avaliable; Professor\_ProfessorId; SubjectId; ClassSceduleId;
* **Student\_Course:** Students\_StudentId; Course\_CourseId
* **Classroom:** ClassroomId; Classroom\_Name; Buildings; Location; Capacity;
* **Classroom\_Course:** Classroom\_ClassroomId; Course\_CourseId
* **Course\_Scedule**: CourseSceduleId; Meeting\_Day; Time\_Start; Time\_End
* **Subject:** SubjectId; Subject\_Name; DepartmentId
* Professors: ProfessorId; First\_Name; Last\_Name; Email; Phone; Address
* **Subject\_Professors:** Subject\_SubjectId; Professors\_ProfessorId
* **Titles:** TitleId; Type
* **Titles:** Titles\_Titled; Professors\_ProfessorId
* **Manager:** ManagerId; First\_Name; Last\_Name; Phone; Address
* **Advisor:** AdvisorId; First\_Name; Last\_Name; Phone; Address
* **Department:** DepartmentId; Name; AdvisorId; ManagerId

**b)-Define a set of relationships that might exist between/among entities and attributes. Such relationships may include one-to-one, one-to-many and many-to-many associations.**

**Students – Course:** Many to many

(One student can take many course, and one course can have many students)

**Students – Major:** Many to many

(One student can have one or more majore, and one major can have many students.)

**Major – Department:** Many to one

( One major must belong to one department, but one department can have many major)

**Department – Manager:** One to one

(One department can only have one manager, and one manager can only manage one specific department)

**Department – Advisor:** One to one

(One department can only have one advisor, and one advisor can only manage one specific department)

**Department – Subject:** One to many

(One deparment can have many subjects, but one subject can only belong to one department.)

**Subject – Course:** One to many

(One subject can have many different course, but one course must have one subject)

**Course – Course\_Schedule:** Many to one

(One course\_Schedule can have many different course, but one course can only have on specific schedule.)

**Course – Classroom:** Many to many

(One course may have one or more classroom, and one classroom can have many different course.)

**Professor – titles:** Many to many

(One professor can have one or more titles, and one titles can have many different professor.)

**Professor – subject:** Many to many

(One professor can teach many different subject course, and one subject course can have many different professor.)

**c)-Define a set of constraints that may be imposed on data.**

* One department can only have 1 manager.
* One department can only have 1 advisor.
* Student must take at least one course
* Every Professor must have proper title, sush as Ph.d, Acaemic or assistanmt
* One major can only belong to one department.
* One course can only belong to one subject, such as CSC or MATH.
* A course must have at least one classroom.

**2) - Define an E-R Diagram for your database design.**

**3) - Define a relational schema for your database design.**

**a)-Define one or more realistic key(s) for every relation scheme. Use both simple and composite keys.**

* **Student:** The primary key should be SutdentId, since StudentId is unque.
* **Students\_Course:** Both Students\_StudentId and Course\_CourseId are foreign keys.
* **Course:** CourseId should be primary key, because it is unique. And Professors\_professorId, SubjectId, CourseScheduleId are foregin keys.
* **Students\_Major:** Both Students\_StudentId and Major\_MajorName are foreign keys.
* **Major:** MajorName is primary key because it is unique. The DepartmentId is foregn key.
* **Department:** The primary key should be DepartmentId, the foregn keys are AdvisorId and ManageId.
* **Manager:** ManagerId should be primary key since it is unique.
* **Advisor:** AdvisorId should be primary key.
* **Course\_Schedule:** The primary key should be CourseScheduledId since it is unique.
* **Subject:** SubjectId should primary key and the DepartmentId should be foreign key.
* **Ttitles:** The titleId should be primary key.
* **Titles\_Professor:** Both Titles\_TtitleId and Professor\_ProfessorId are foreign keys.
* **Professors:** ProfessorId should be primary key since it is unique.
* **Subject\_Professors:** Both Subject\_SubjectId and Professors\_ProfessorId are foreign keys.
* **Classroom:** The primary key should be classroomId since it is unique.
* **Classroom\_Course:** Both Classroom\_ClassroomId and Course\_CourseId are foreign keys.

**b)-Define a realistic set of Functional / Multi-Valued Dependencies (when appropriate) for every relation scheme.**

**Student:**

StudentId 🡪 First\_Name

StudentId 🡪 Last\_Name

StudentId 🡪Country

StudentId 🡪 Age

StudentId 🡪Phone

StudentId 🡪 Email

StudentId 🡪 Address

**Major:**

MajorName 🡪 Department

MajorName 🡪 Minimum\_GPA

**Manager:**

ManagerId 🡪 First\_Name

ManagerId 🡪 Last\_Name

ManagerId 🡪 Phone

ManagerId 🡪 Address

**Classroom:**

ClassroomId 🡪 Classroom\_Name

ClassroomId 🡪 Buildings

ClassroomId 🡪 Capacity

ClassroomId 🡪 Location

**Course:**

CourseId 🡪 Course\_Name

CourseId 🡪 Seats Avaliable

**Department:**

DepartmentId 🡪 Name

**Professors:**

ProfessorId 🡪 First\_Name

ProfessorId 🡪 Last\_Name

ProfessorId 🡪 Email

ProfessorId 🡪 Phone

ProfessorId 🡪 Address

**Advisor:**

AdvisorId 🡪 First\_Name

AdvisorId 🡪 Last\_Name

AdvisorId 🡪 Phone

AdvisorId 🡪 Address

**Titles**

TitleId 🡪 Type

**Subjec**

SubjectId 🡪 Subject\_Name

**Course\_Schedule**

CourseScheduledId 🡪 Meeting\_Day

CourseScheduledId 🡪 Time\_Start

CourseScheduledId 🡪 Time\_End

**C-Check whether your relational schema is in 2NF, 3NF, BCNF, 4NF.**

**d)-Put your relational schema in the highest normal form that is possible.**

**Note that, every relation scheme should be in a specific normal form in order to have the relational schema in that normal form.**

**NOTE: Please provide a detailed explanation for every question when appropriate.**

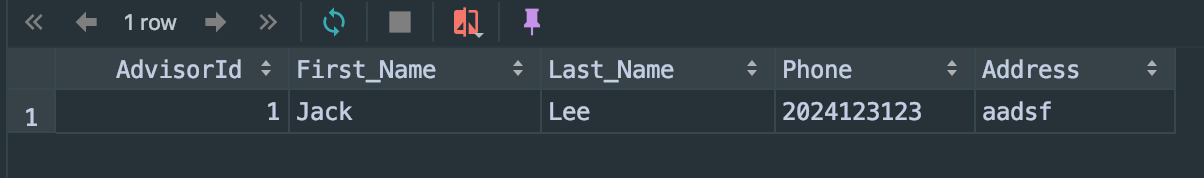
**4) Implementation: Create your database using Oracle, or MySQL, or… to Perform the following operations.**

1. **You are required to execute SQL queries that include the following operations. For each query, provide the SQL statements along with the output. For each of the following, try different SQL statements (i.e., using one relation, more than one relations,...).**

**select involving one/more conditions in Where Clause**

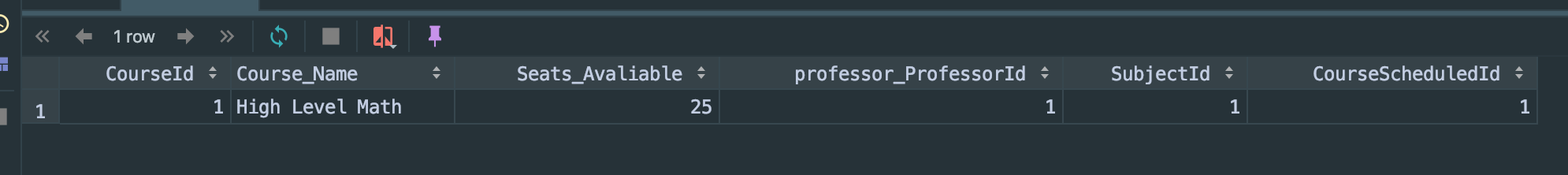
*select* \* *from* advisor  
*where* Last\_Name = "Lee";

**Output:**

****

*select* \* *from* Course  
*where* SubjectId = 1  
*and* CourseScheduledId = 1;

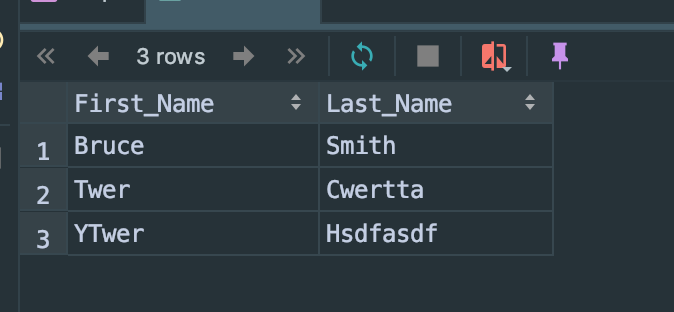
**Output:**

****

For another more difficult exmaple, if I want to find studnets who major is computer science:

*select* First\_Name, Last\_Name *from* Students, Students\_Major, Major  
*where* studentsId = Students\_StudentId  
*and* MajorName = Major\_MajorName  
*and* majorName = "computer science";

**output:**



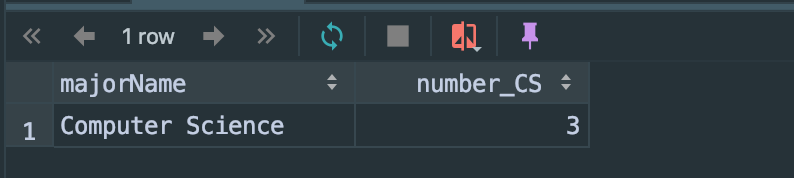
**select with aggregate functions (i.e., SUM,MIN,MAX,AVG,COUNT)**

**Count:**

If I want to know how many students’ major is computer science. Then the SQL should be:

*select* majorName, *count*(StudentsId) *as* number\_CS *from* students, Students\_Major, Major  
*where* StudentsId = Students\_StudentId  
*and* Major\_MajorName = MajorName  
*and* MajorName = "Computer Science"  
*group by* majorName;

Output：

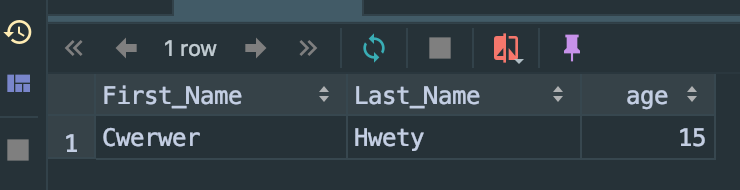


**Min:**

If I want to find youngest studnets in college, I will use sql below:

*select* First\_Name, Last\_Name, age *from* students *where*age = (*select* min(age) *from* Students);

Output:

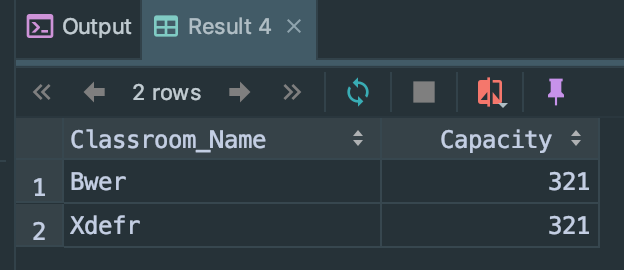


**Max:**

If I want to find which classroom can contain more students than any other classroom, I will use sql below:

*select* Classroom\_Name, Capacity *from* Classroom  
*where* Capacity = (*select max*(Capacity) *from* Classroom)

**Output:**

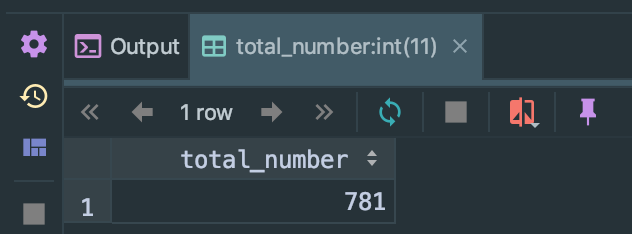


**Sum:**

If I want to find how many total classroom capacity avaliable, I will use sql below:

*select sum*(Capacity) *as* total\_number *from* Classroom;

Output:

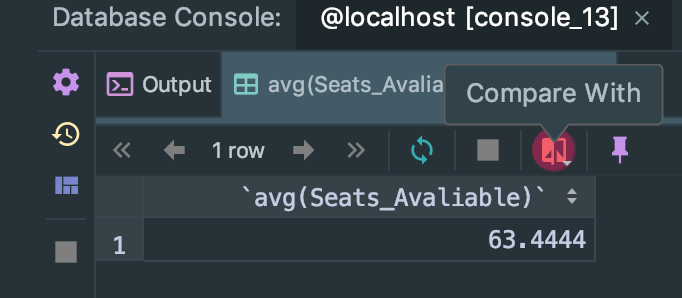


**AVG:**

If I want to find the average seats avaliable for each course, I will use sql below:

*select avg*(Seats\_Avaliable) *from* Course;

**Output:**

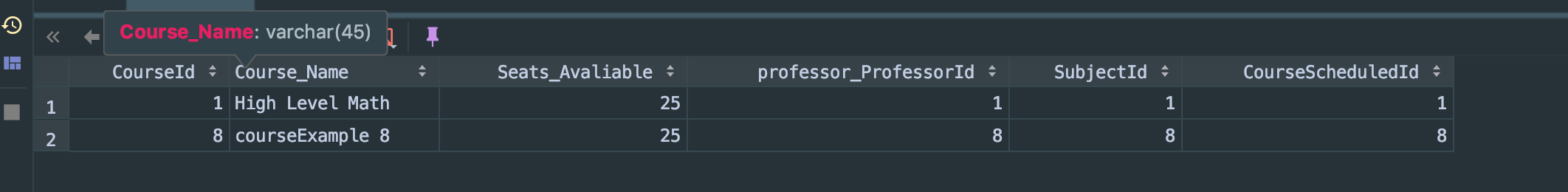


**select with Having, Group By, Order By clause**

**Only “Having”:**

*select* \* *from* Course  
*having* Seats\_Avaliable = 25;

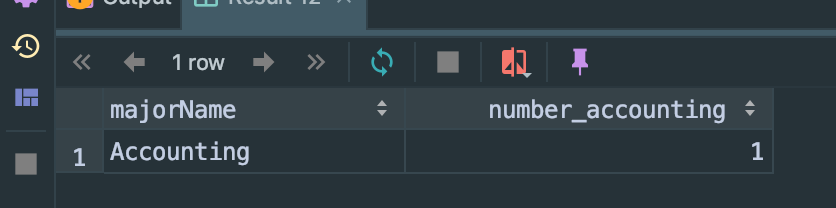
Output (In this case, having = where):



**Having and group by:**

*select* majorName, *count*(StudentsId) *as* number\_accounting *from* students, Students\_Major, Major  
*where* StudentsId = Students\_StudentId  
*and* Major\_MajorName = MajorName  
*group by* majorName  
*having* MajorName = "Accounting";

**Output**

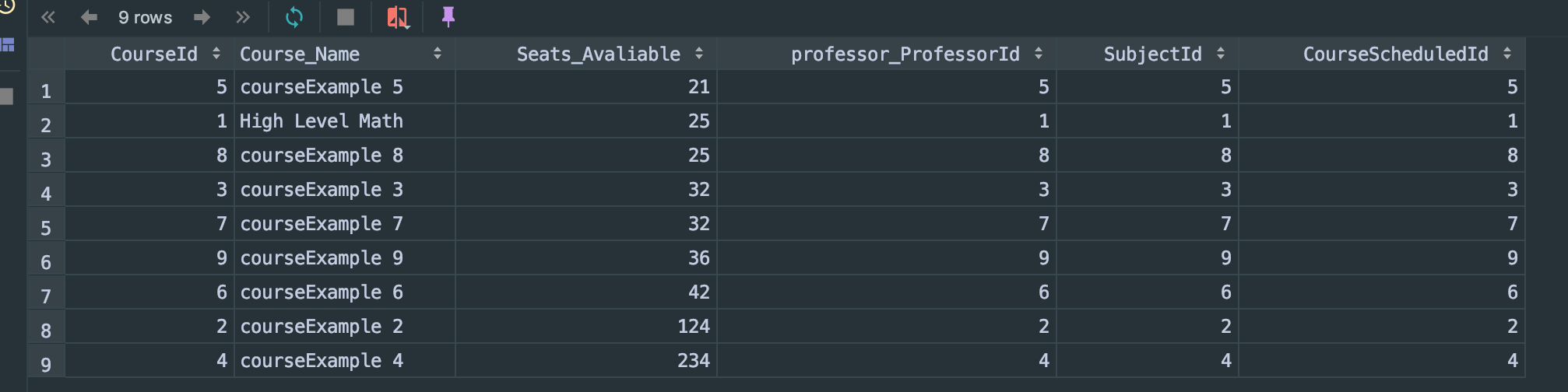
****

**Order by:**

If I want to compare course’e seats availably, I will write sql below:

*select* \* *from* Course  
*order by* Seats\_Avaliable;

**Output:**

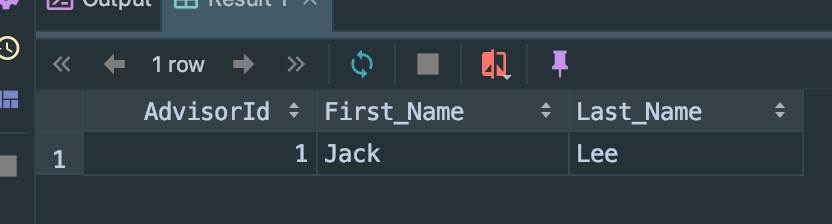


**Nested Select:**

If I want to know who is CAS advisor, I will do sql below:

*select* AdvisorId, First\_Name, Last\_Name *from* Advisor  
*where* AdvisorId = (*select* AdvisorId *from* Department  
 *where name* = "CAS")

**output:**

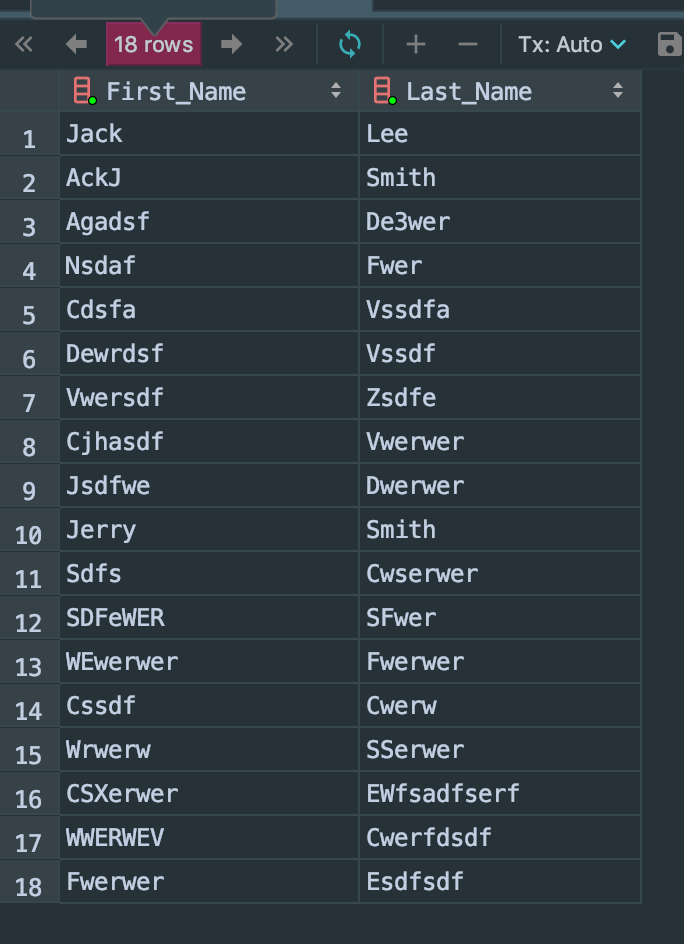


**select involving the Union operation**

If I want to see all advisor and all manager in all departments, I will use sql below:

*select* First\_Name, Last\_Name *from* Advisor  
*union  
select* First\_Name, Last\_Name *from* Manager

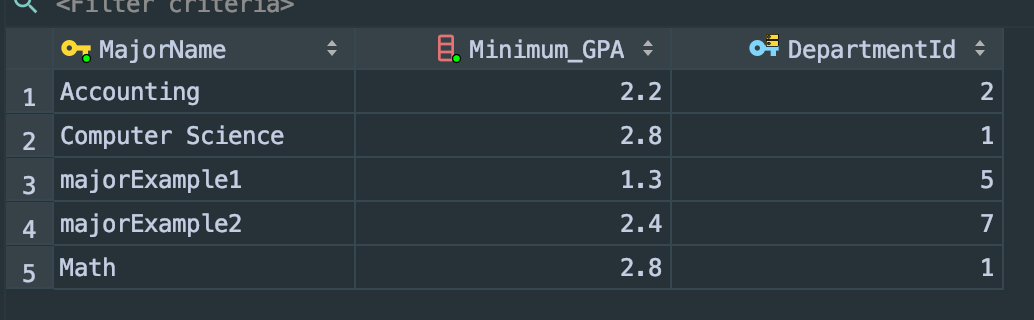
**output:**



**- Insert:**

**insert one tuple into a table:**

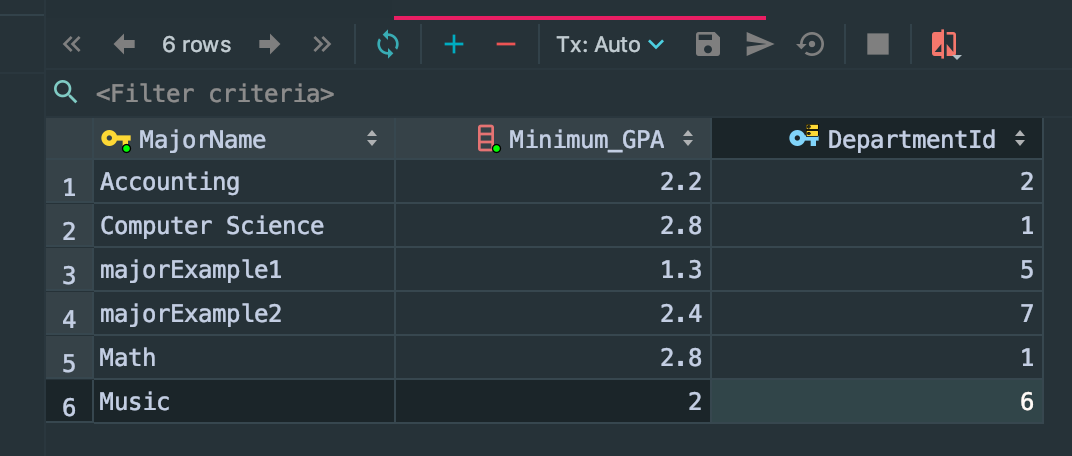
Table before insert:



SQL:

*insert into* Major (MajorName, Minimum\_GPA, DepartmentId)  
*values* ("Music", 2.0, 6)

Talbe after insert:



**insert a set of tuples (by using another select statement)**

xxxx

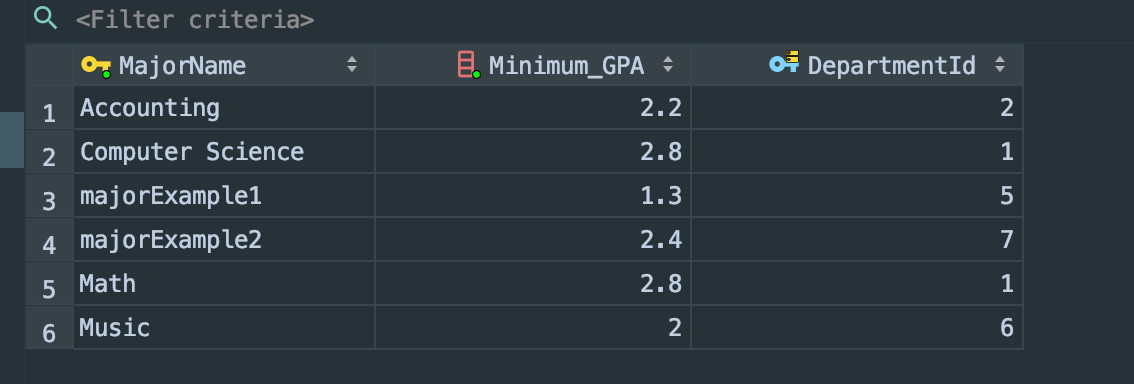
**insert involving two tables**

**- Delete: delete one tuple or a set of tuples: from one table, from multiple tables.**

**From one table:**

My requirement is delete all about musict major information.

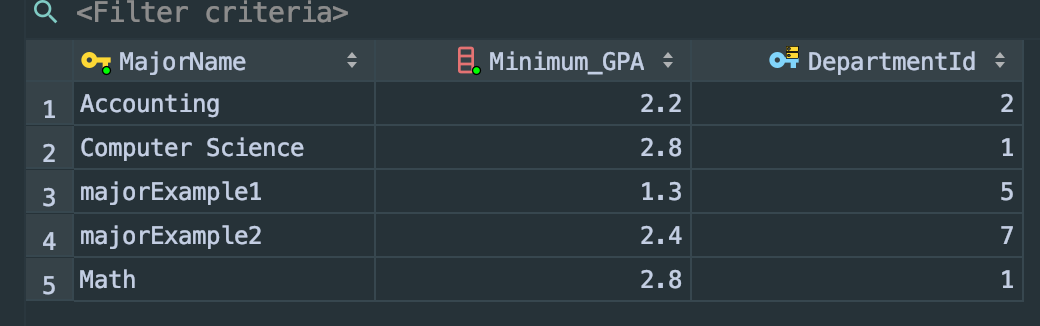
Table before delete:



Mysql:

*delete from* Major  
*where* MajorName = "music";

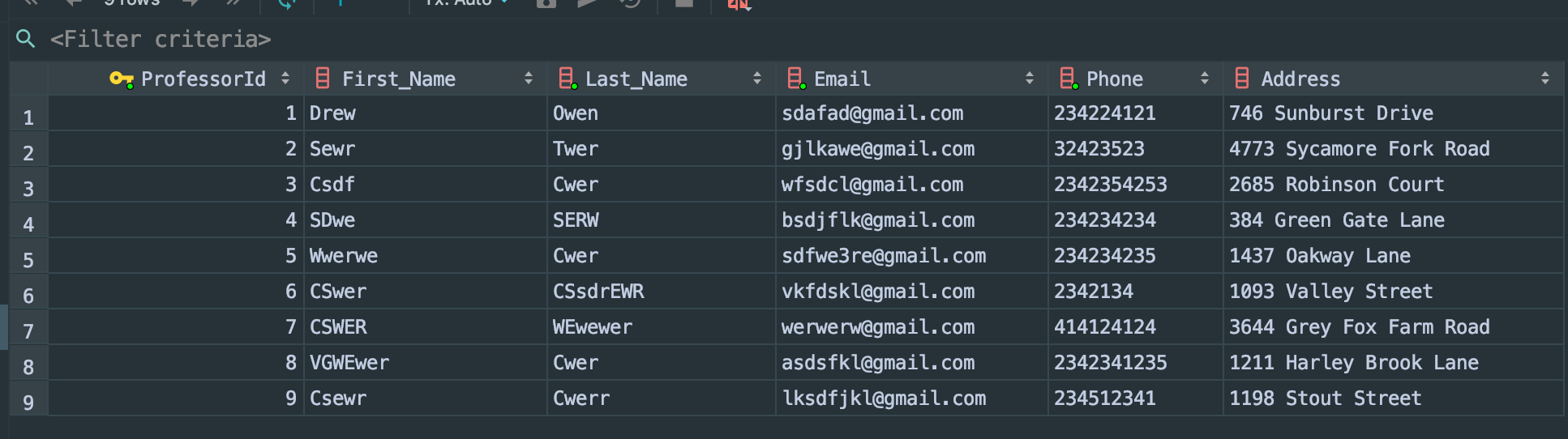
Table after delete:



From mutiple table to delete:

My requeirement is delete all prfessors Title is PH. D

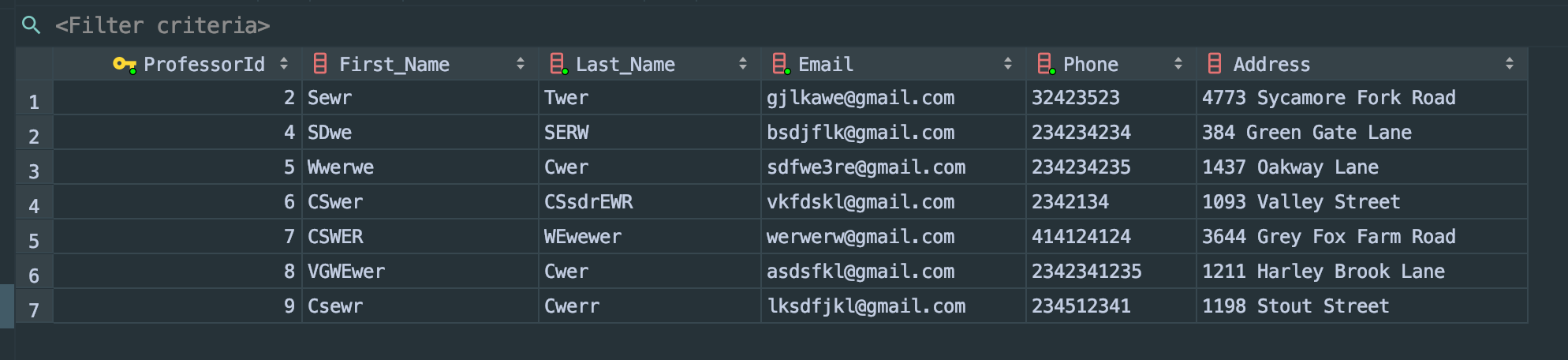
Table before delete:



Mysql:

*delete  
from* Professors  
*where* ProfessorId *in* (*select* professors\_ProfessorId  
 *from* Titles\_Professors  
 *where* Titles\_TitleId = (*select* TitleId *from* Titles *where* type = "PH. D"));

Table after delete:

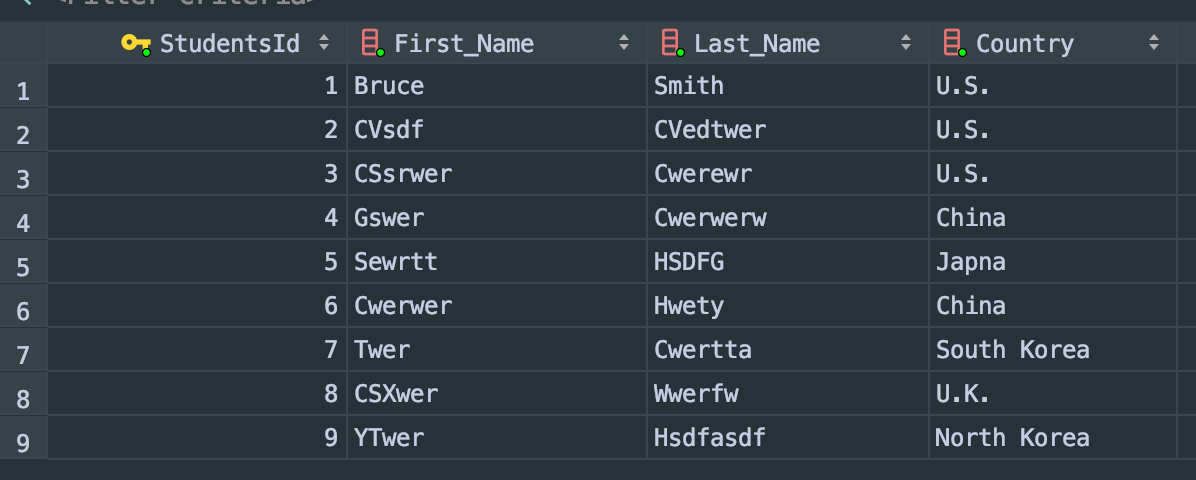


**- Update: update one tuple or a set of tuples: from one table, from multiple tables.**

**From one table:**

My requirement is to change student Bruce Smith’s country from U.S. to U.K.

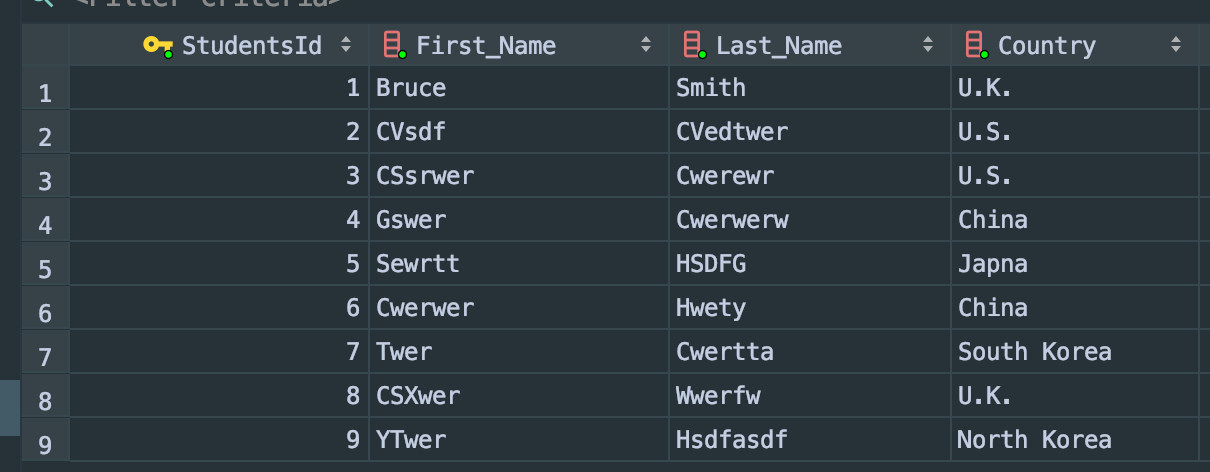
Table before update:



Mysql:

*update* students  
*set* country = "U.K."  
*where* First\_Name = "Bruce"  
*and* Last\_Name = "Smith";

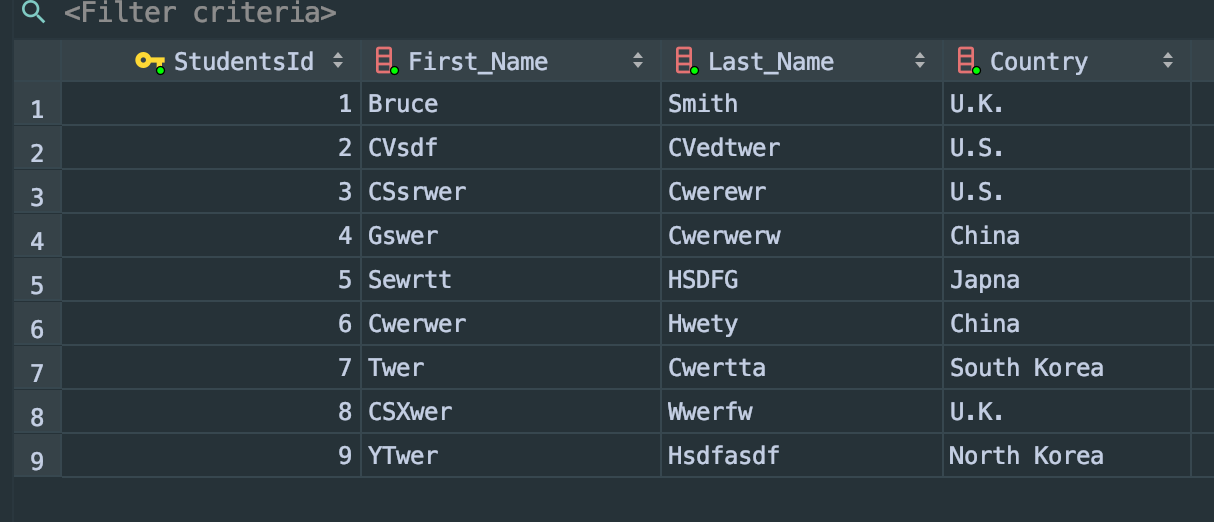
Table after update:



**From multiple table to update:**

My requirement is update all computer science studnets’ country to Canada.

Table beofore update:



Mysql:

*update* Students  
*set* Country = "Canada"  
*where* StudentsId *in* (*select* Students\_StudentId  
 *from* Students\_Major  
 *where* Major\_MajorName = "Computer Science");

Table after update:



**- Create View:**

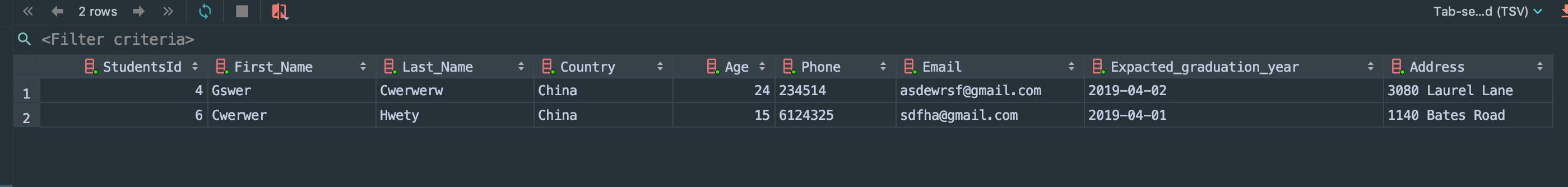
**- based on one relation and more than one relation:**

**One relation view:**

If I want to create a view to show all Chinese students, I will write sql below:

*create view* CHIN\_students\_info *as  
select* \*  
*from* students  
*where* Country = "China";

**output:**

****

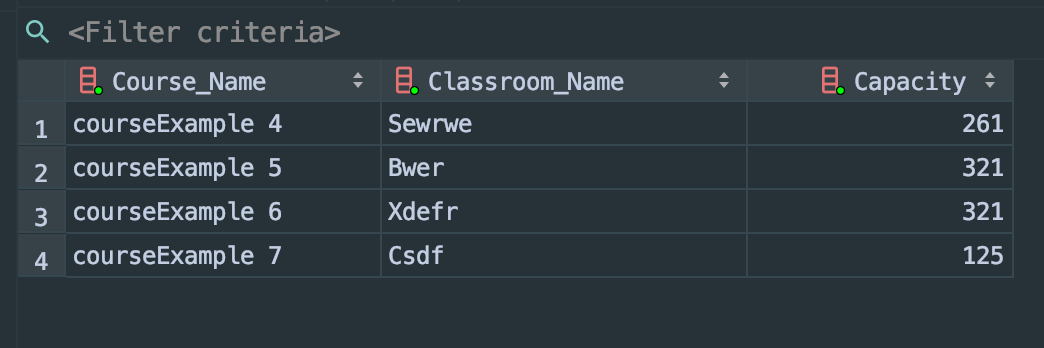
**More than one relation view:**

If I want to create a view I called high\_capacity\_Classroom\_Course which would show a courser’s classroom capacity can contain more than 100 students.

Mysql:

*create view* High\_capacity\_Classroom\_Course *as  
 select* Course\_Name, Classroom\_Name, Capacity *from* course, Classroom, Classroom\_Course  
*where* courseId = CourseCourseId  
*and* Classromm\_ClassroomId = ClassroomId  
*and* Capacity > 100;

**Output:**

****

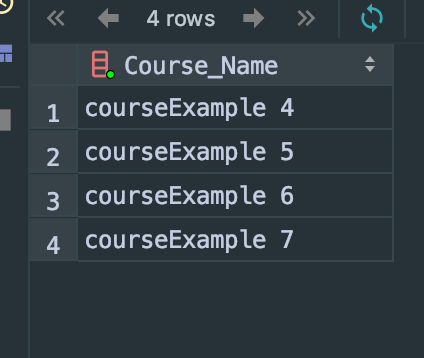
**- operate on View (i.e., select, insert, delete, update,..)**

I’m going to operate my view I created last question high\_capacity\_Classroom\_Course.

**Select VIEW:**

*select* Course\_Name *from* high\_capacity\_classroom\_course;

**Output:**

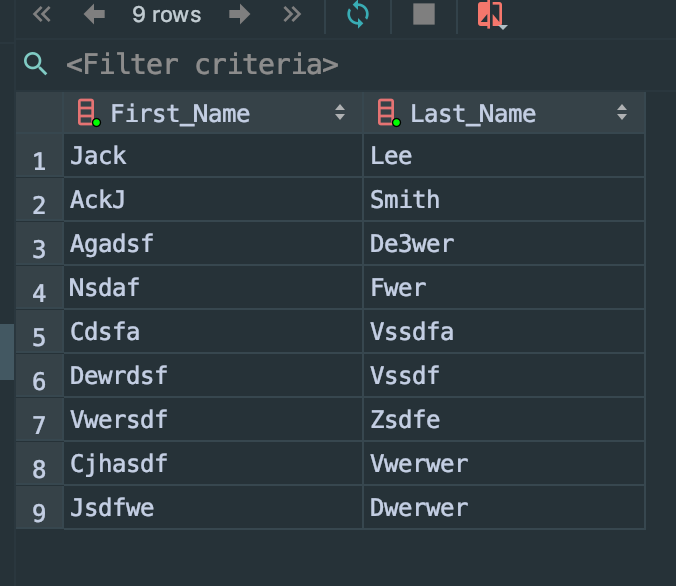


**Insert VIEW:**

**First, let’s create another view:**

*create view* Advisor\_name *as  
 select* First\_Name, Last\_Name *from* Advisor;

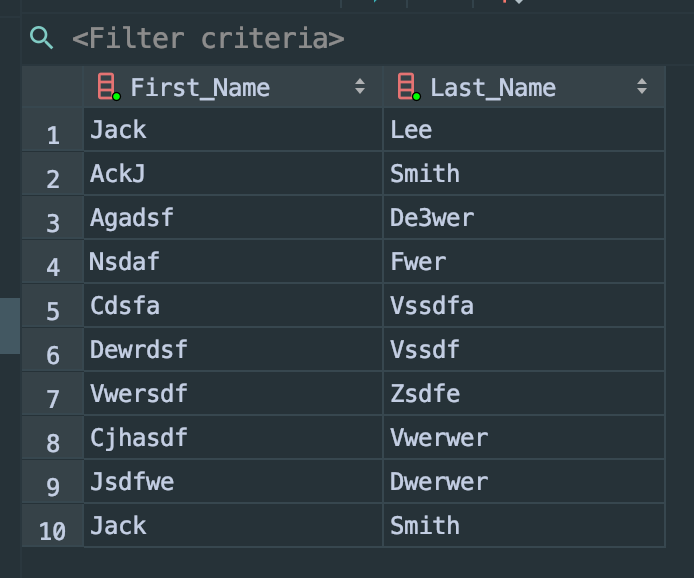
Table before insert:



Insert view sql:

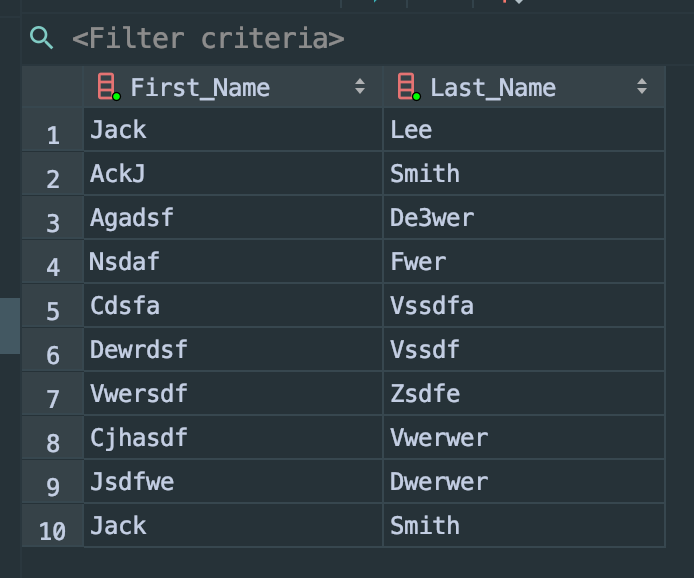
*insert into* advisor\_name (First\_Name, Last\_Name) *VALUES* ("Jack", "Smith");

Table after insert:



**Delete VIEW:**

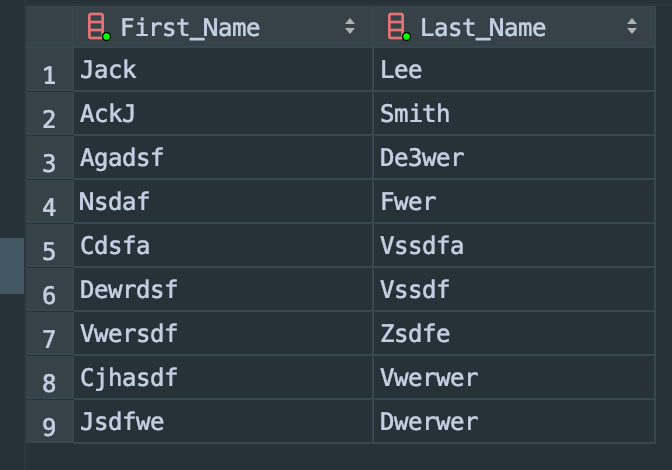
**Table before delete:**



Sql:

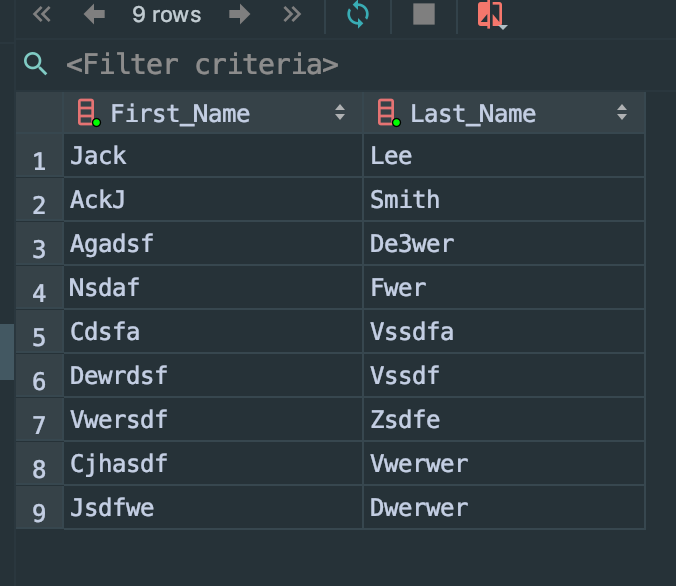
*delete from* advisor\_name  
*where* First\_Name = "Jack"  
*and* Last\_Name = "Smith";

Table after delete:



**Update VIEW:**

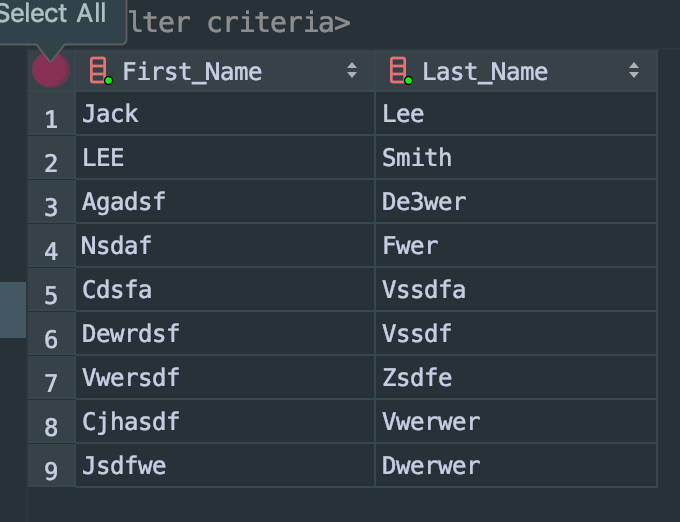
Table before update:



SQL:

*update* advisor\_name *set* First\_Name = "LEE"  
*where* Last\_Name = "Smith";

Table after update:



**TRIGGER**

**B) Also, create at least 4 different practical/useful triggers (written in MySQL) for your database to perform the following tasks:**

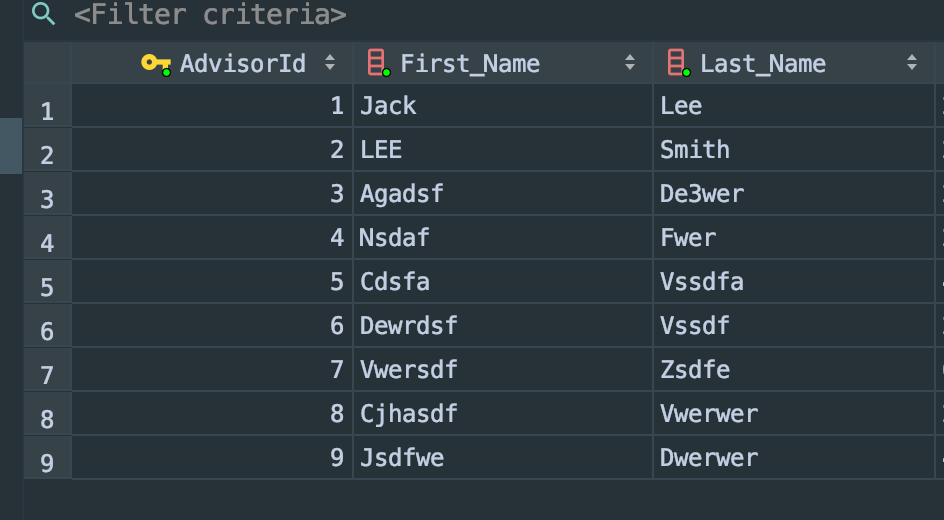
**- enforcing referential integrity**

I create a trigger Advisor\_delete. If user delete department, then the department advisor also will delete automaticly.

*Delimiter* $$  
*create trigger* Advisor\_delete *after delete on* Department  
 *for each row  
 begin  
 delete from* Advisor  
 *where* AdvisorId = *old*.AdvisorId;  
 *end* $$

Let’s check if it is work.

Advisor table:



After I run sql below to delete apartment:

*delete from* Department  
*where* DepartmentId = 1;

The advisor table become to:



So, the trigger successfully delete the CAS advisor automaticly.

I also create Manage\_delete trigger, when user delete department, the trigger will automaticly delete the department manaeger.

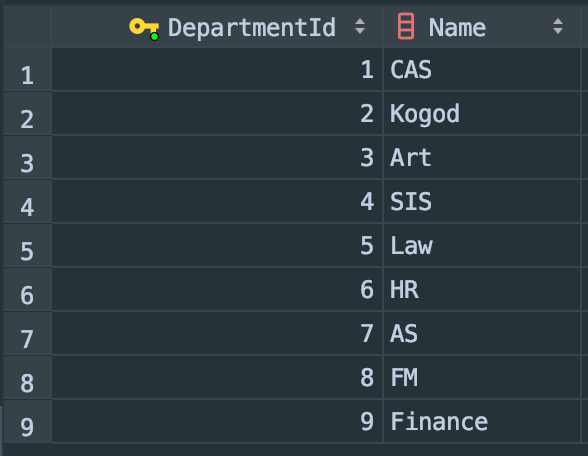
*Delimiter* $$  
*create trigger* Manager\_delete *after delete on* Department  
 *for each row  
 begin  
 delete from* Manager  
 *where* ManagerId = *old*.ManagerId;  
 *end* $$

Since the method is the same, I didn’t take the screenshot for this trigger.

**- enforcing attribute domain constraints**

I’m going to make trigger that constraints college’s department.

Right now, my database department is below:



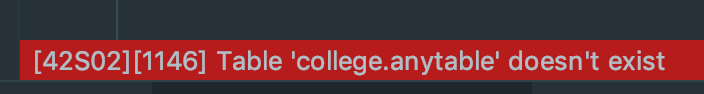
I don’t want to user add any other major to a new department. So, the trigger should be:

*Delimiter* $$  
*Create trigger* Department\_domain\_checking *before insert on* Major  
*For each row  
Begin  
Declare* temp *int*; *set* temp=0;  
*Select count*(\*) *into* temp  
*From* Department *where* Department.DepartmentId=new.DepartmentId;  
*If* temp=0 *then  
Insert into* mylog *values*('Invalid Major');  
*Insert into* anytable *values*('This major is not offered'); *End if*;  
*End*; $$

Let’s test (I create a new departmentId 15):

*insert into* major (MajorName, Minimum\_GPA, DepartmentId) *values* ("test",2.3,15);

Then the error pop up:



We are not allow to add major to a new department anymore (15 is new department ID). The trigger is succesfuly prevent that happen.

**- creating database log**

For this trigger, I’m going to create new table called log.

*Create table* StudentLog (message *varchar*(70));

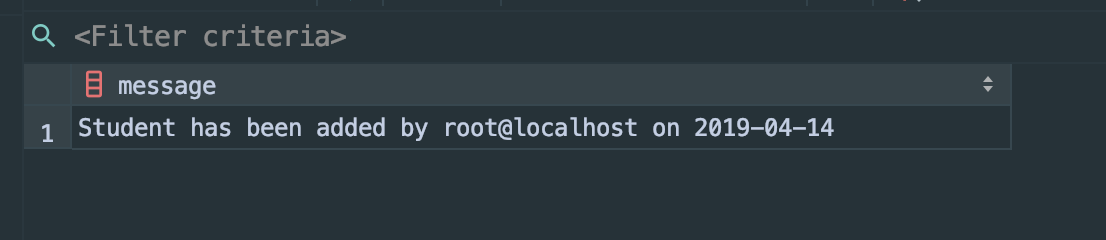
Trigger:

*Delimiter* $$  
*create trigger* Student\_Log *after insert on* students  
 *for each row  
 begin  
 insert into* StudentLog *values*(*concat*('Student has been added by ',*current\_user*(), ' on ',*current\_date*()));  
 *end*;  
 $$

Let’s active by using:

*insert into* students (First\_Name, Last\_Name, Country, Age, Phone, Email, Expacted\_graduation\_year, Address)  
*values* ("Bruce", "Guo", "Japna", 21, "20234234", "asdfaea@gmail.com", "2021-04-09", "addresss test 1");

Then the studentLog automaticly add information below:



**- gathering statistics**

I’m going to create student’s age summary trigger.

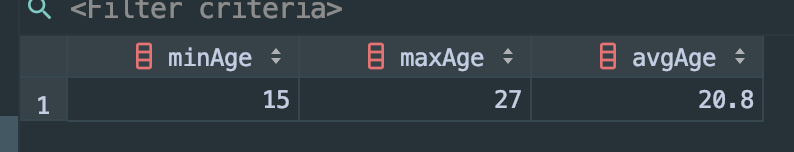
First, create table student\_age\_summary

*create table* student\_age\_summary(minAge *double*, maxAge *double*, avgAge *double*);

The trigger code should be:

*Delimiter* $$  
*create trigger* age\_sum *after insert on* Students  
*for each row  
begin  
delete from* student\_age\_summary;  
*insert* student\_age\_summary  
*select min*(age),*max*(age),*avg*(age) *from* students;  
*end*; $$

After I insert a new student in student table, the trigger automaticly made Age summary for me:



**ColdFusion**

**Use MySQL and ColdFusion to create a Web-based application to enable the user to do the following operations:**

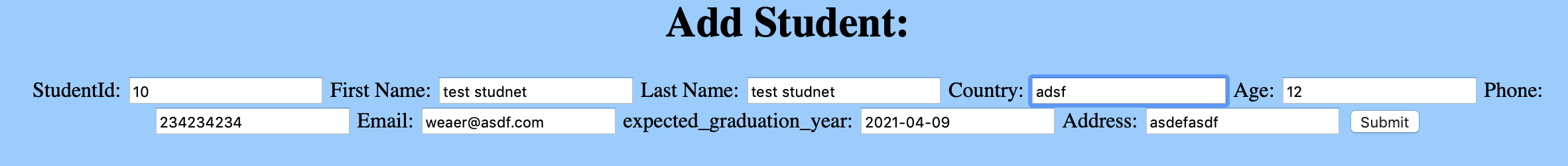
**Add a record**

I made “add student” on web server.

Student table before add:

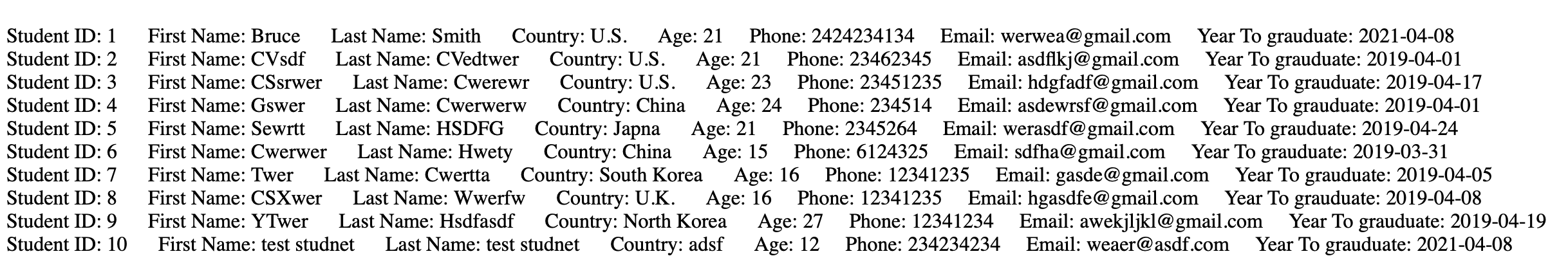


Then, I add student on the web server:





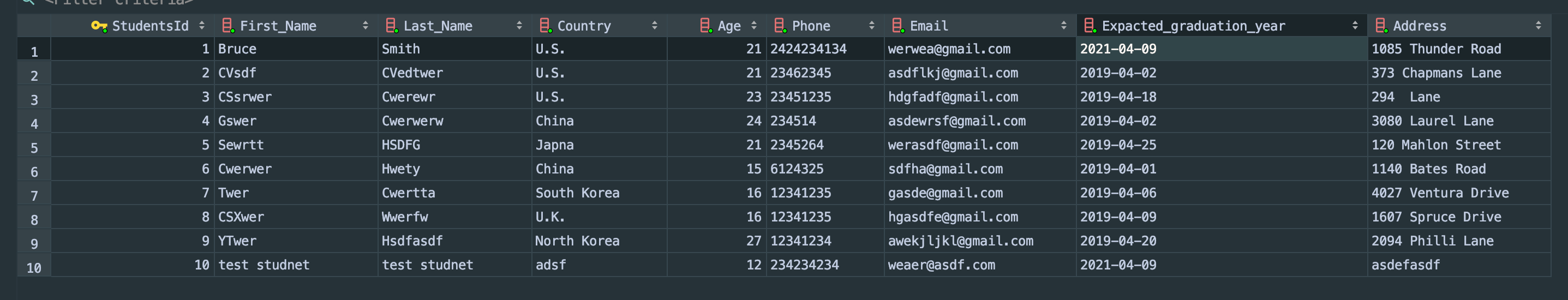
Then, let’s check student table:



We can see that student 10 has been added.

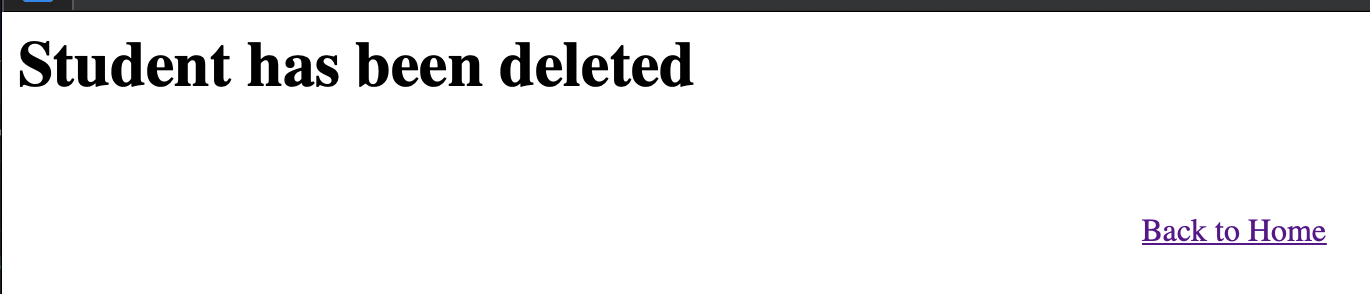
**Delete a record**

Student table before delete:

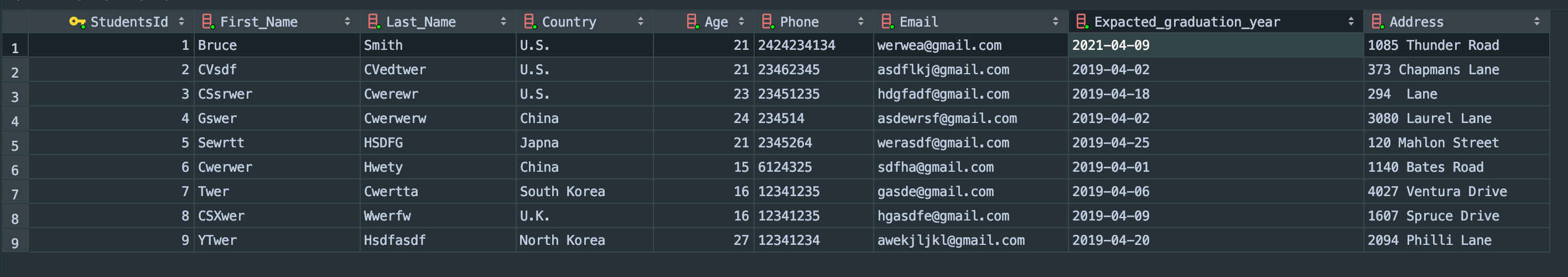


Then, I write the studentID in web server:





Let’s check student table now:

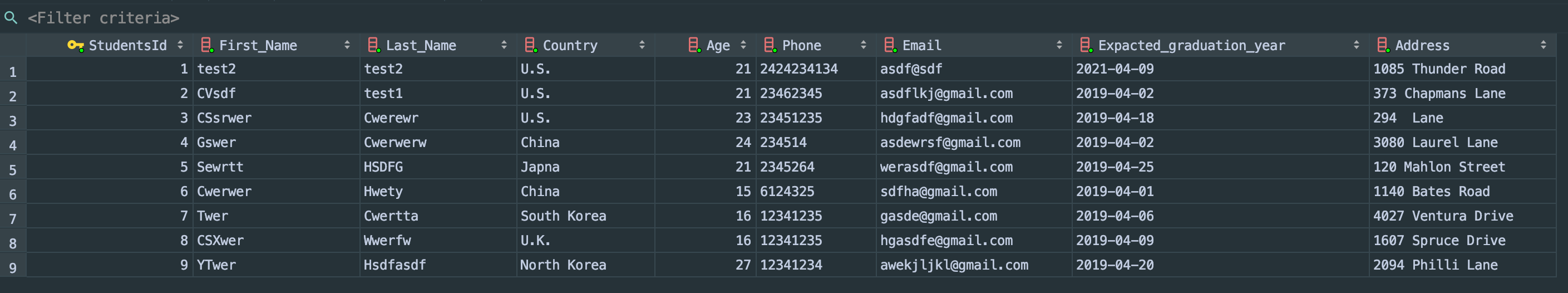


Student 10 has been successfully deleted.

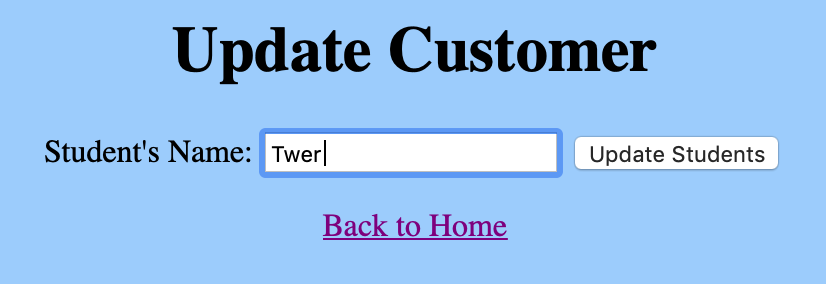
**Update a record**

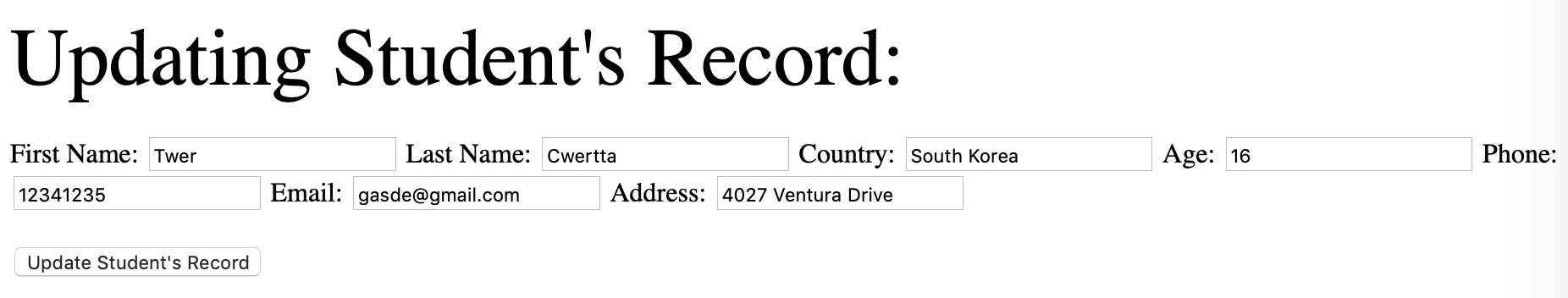
I made student update coldfusion.

Student table before update:



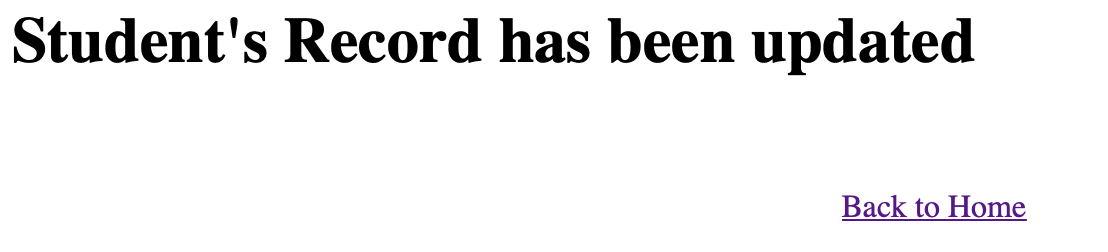
Then I put student Twer name to the web server:



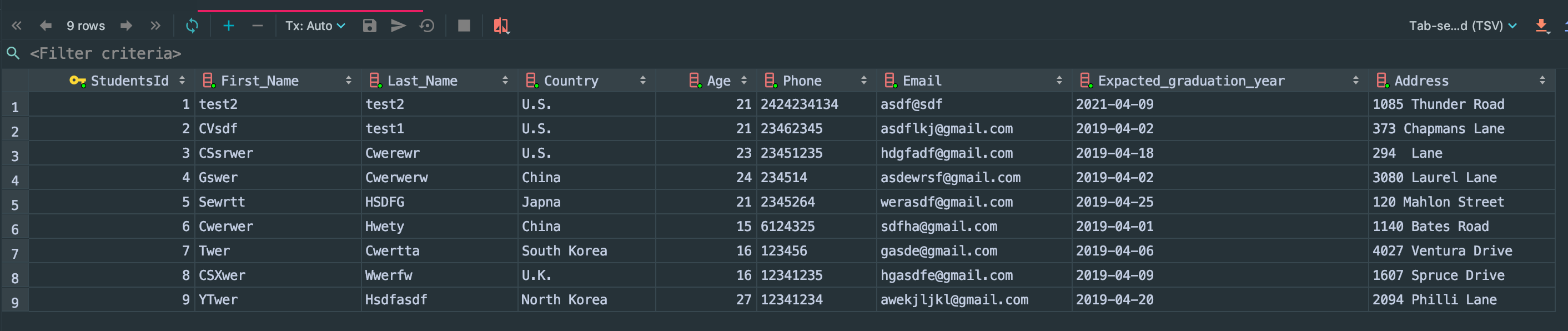
****

Change the phone number to 123456

After click button:



Finally, let’t check student’s table:



Student Twer’s phone has been updated to 123456.

**Query (at least 3 select statements on one relation)**