**Marks:** 100

# Introduction

This project verifies competency in creating T-SQL stored procedures, functions, and triggers, and writing as simple script.

It is a timed individual assignment to be completed during scheduled class time. You are not supposed to discuss this test with people other than your instructor while working on it.

Points assigned to various parts are indicated in the marking rubric at the end of this document.

**You will be submitting this document with your answers pasted into it.**

* **Any code should be pasted as a text and not an image.**
* **The screenshots should show the results from tests.**
* **Make sure to provide test statements even if your module has errors and cannot be executed.**

# Project Tasks

Run attached script CreateCollege.sql to create database College. For your convenience, a copy of an ERD for this database is provided in the attached document College ERD.docx.

1. Create a stored procedure named spInsertDepartment that takes as a parameter a value for the department name and adds a new row into the Departments table.

* Even thoughDepartments table allows null department name, validate that department name is provided and is not an empty string. Throw an error with message “Department name must be provided” if incorrect.
* Department namehas to be unique. The procedure should throw an errorwith message “Department name must be unique” upon attempt to insert a department with a name that is already in the table.

Code three tests executing this procedure:1) with null or empty department name, 2) with a unique department name, and 3) with a duplicate department name.

Paste here code of your procedure, statements for the three tests, and screenshots that shows results of the tests.

USE College;

GO

IF OBJECT\_ID('spInsertDepartment') IS NOT NULL

DROP PROC spInsertDepartment;

GO

CREATE OR ALTER PROC spInsertDepartment

@DepartmentName VARCHAR(40) = NULL

AS

BEGIN TRY

-- Validating presence of DepartmentName:

IF @DepartmentName IS NULL OR @DepartmentName = ''

THROW 50007, 'Department name must be provided', 1;

-- Validating uniqueness of DepartmentName

IF @DepartmentName IN (SELECT DepartmentName FROM Departments)

THROW 50007, 'Department name must be unique', 1;

-- All basic validation was successfully passed, time to insert a row:

INSERT INTO Departments

VALUES (@DepartmentName);

PRINT 'Success! One row was inserted.';

END TRY

BEGIN CATCH

PRINT 'An error occurred. Row was not inserted.';

PRINT 'Error number: ' +

CONVERT(varchar, ERROR\_NUMBER());

PRINT 'Error message: ' +

CONVERT(varchar, ERROR\_MESSAGE());

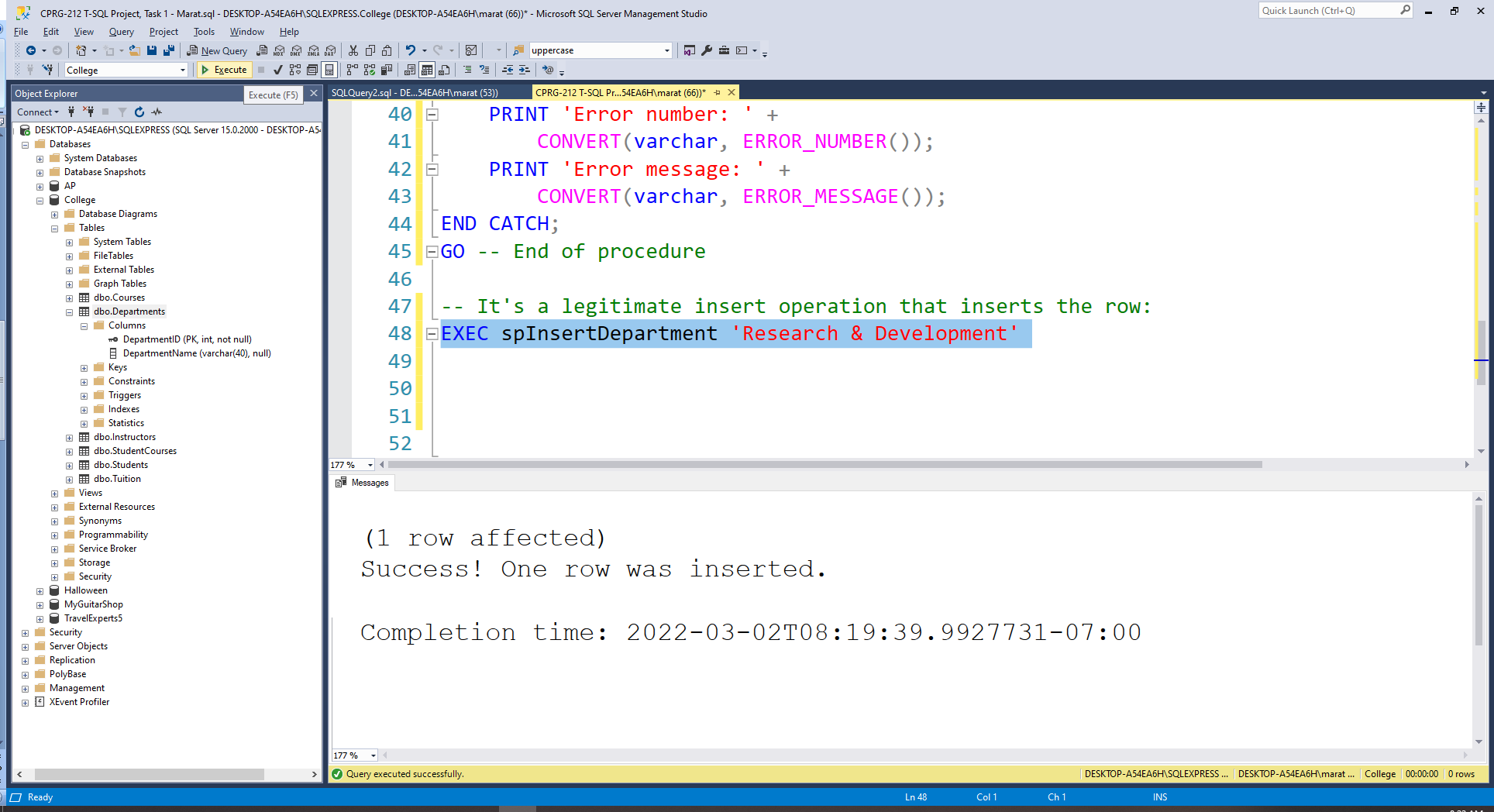
END CATCH;

GO -- End of procedure

Test 1:

-- It's a legitimate insert operation that inserts a row:

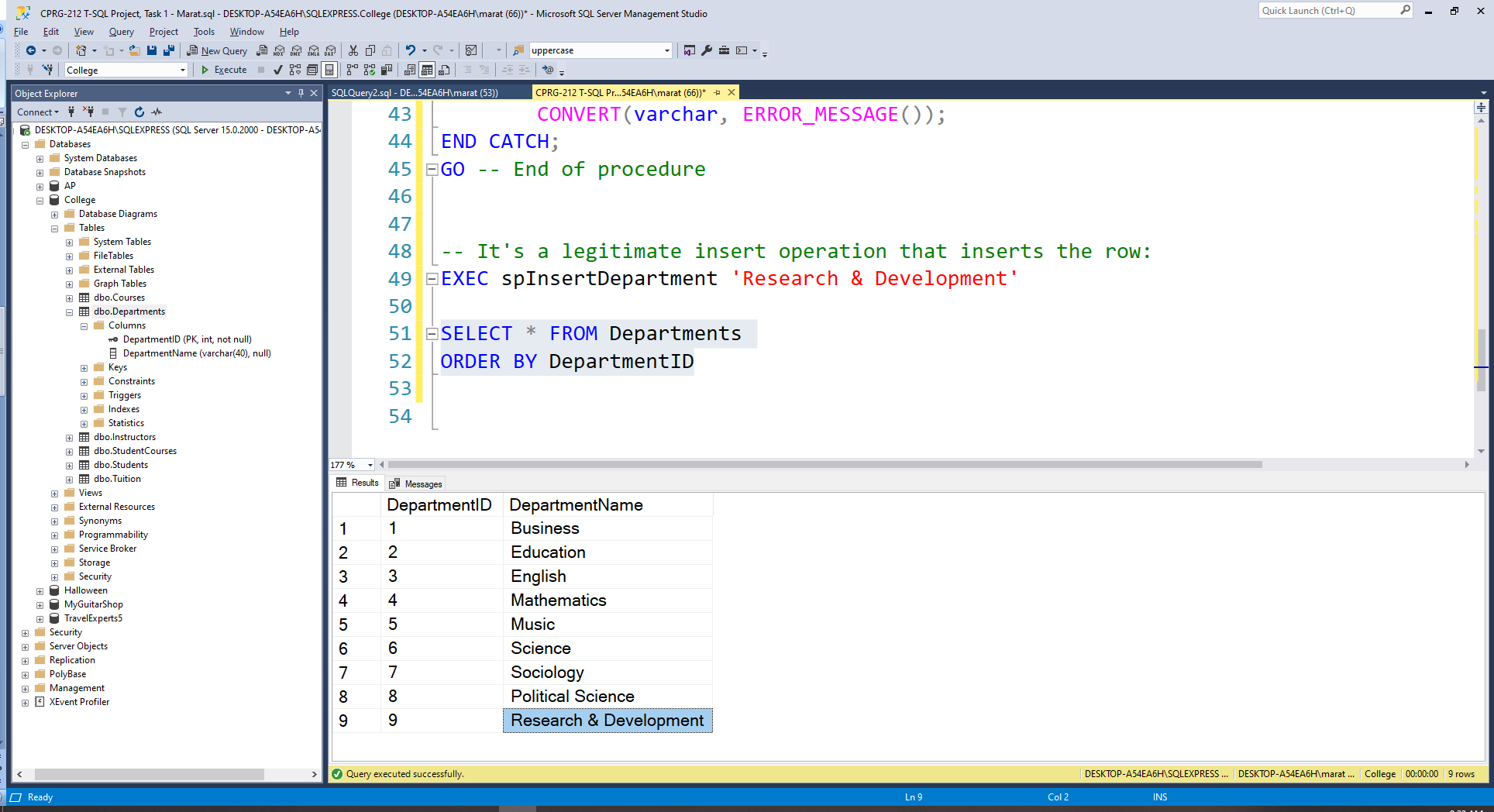
EXEC spInsertDepartment 'Research & Development'



-- Checking changes in the DB after successful Insert operation:

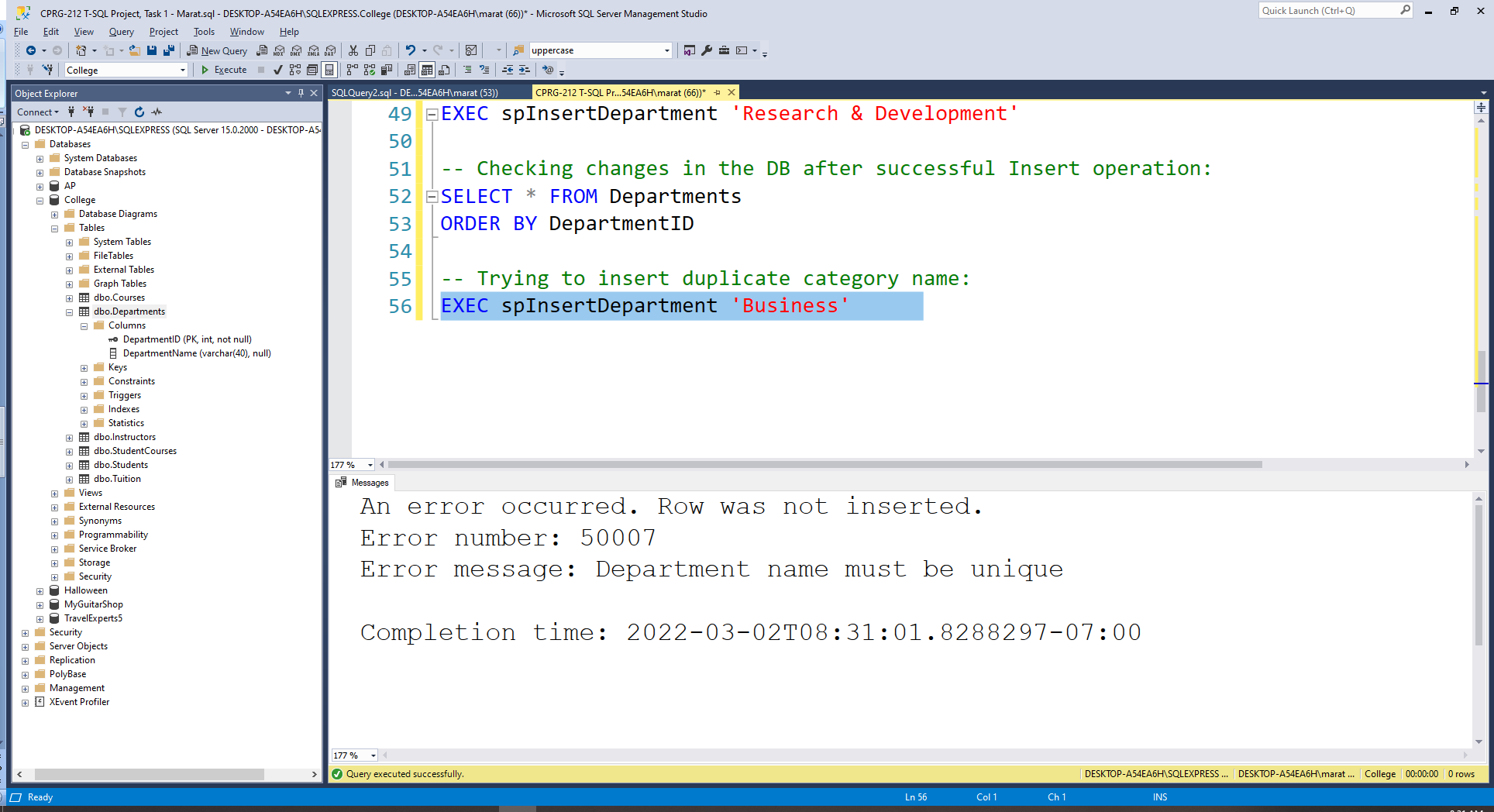
SELECT \* FROM Departments

ORDER BY DepartmentID



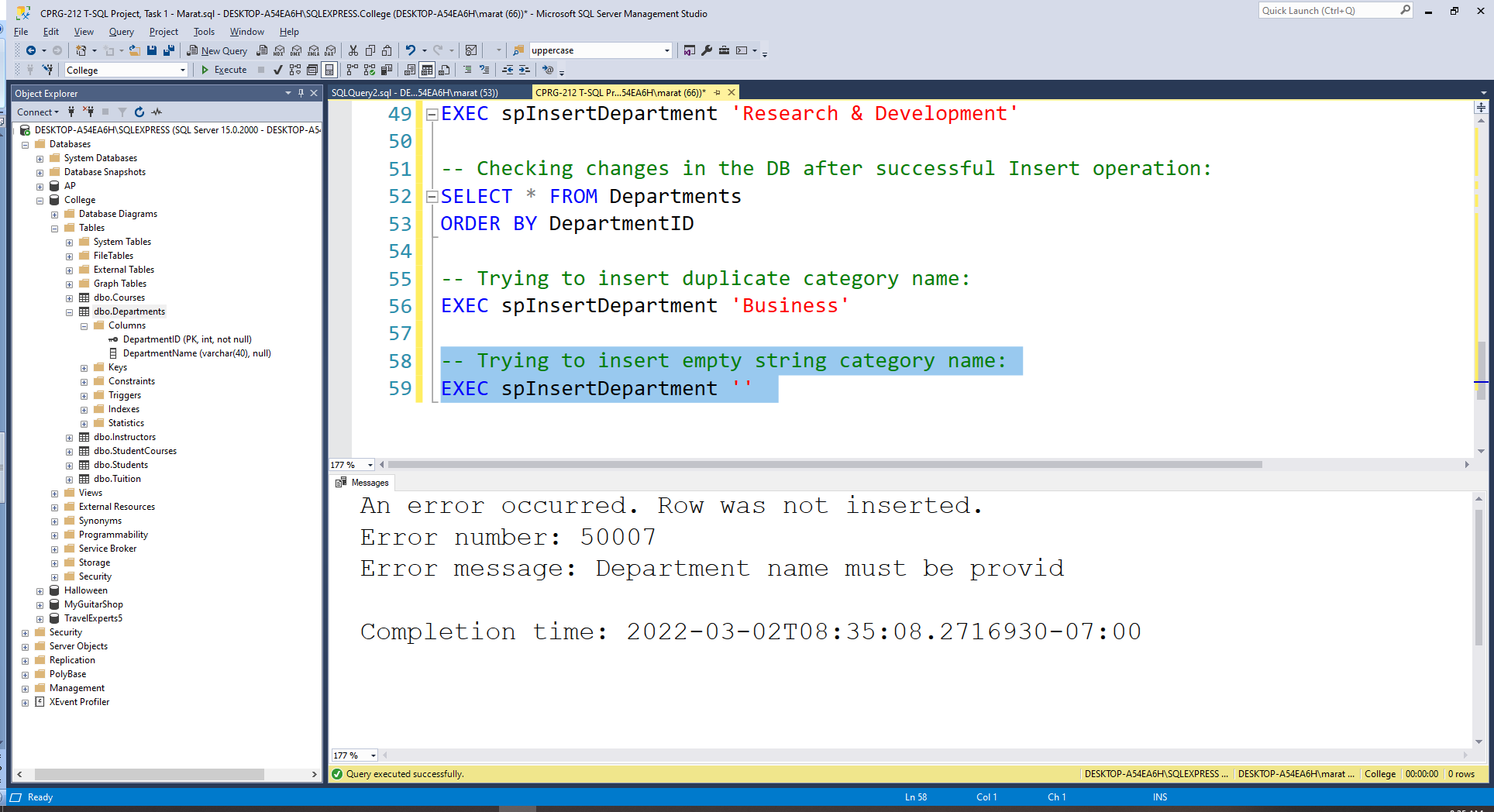
-- Trying to insert duplicate category name:

EXEC spInsertDepartment 'Business'



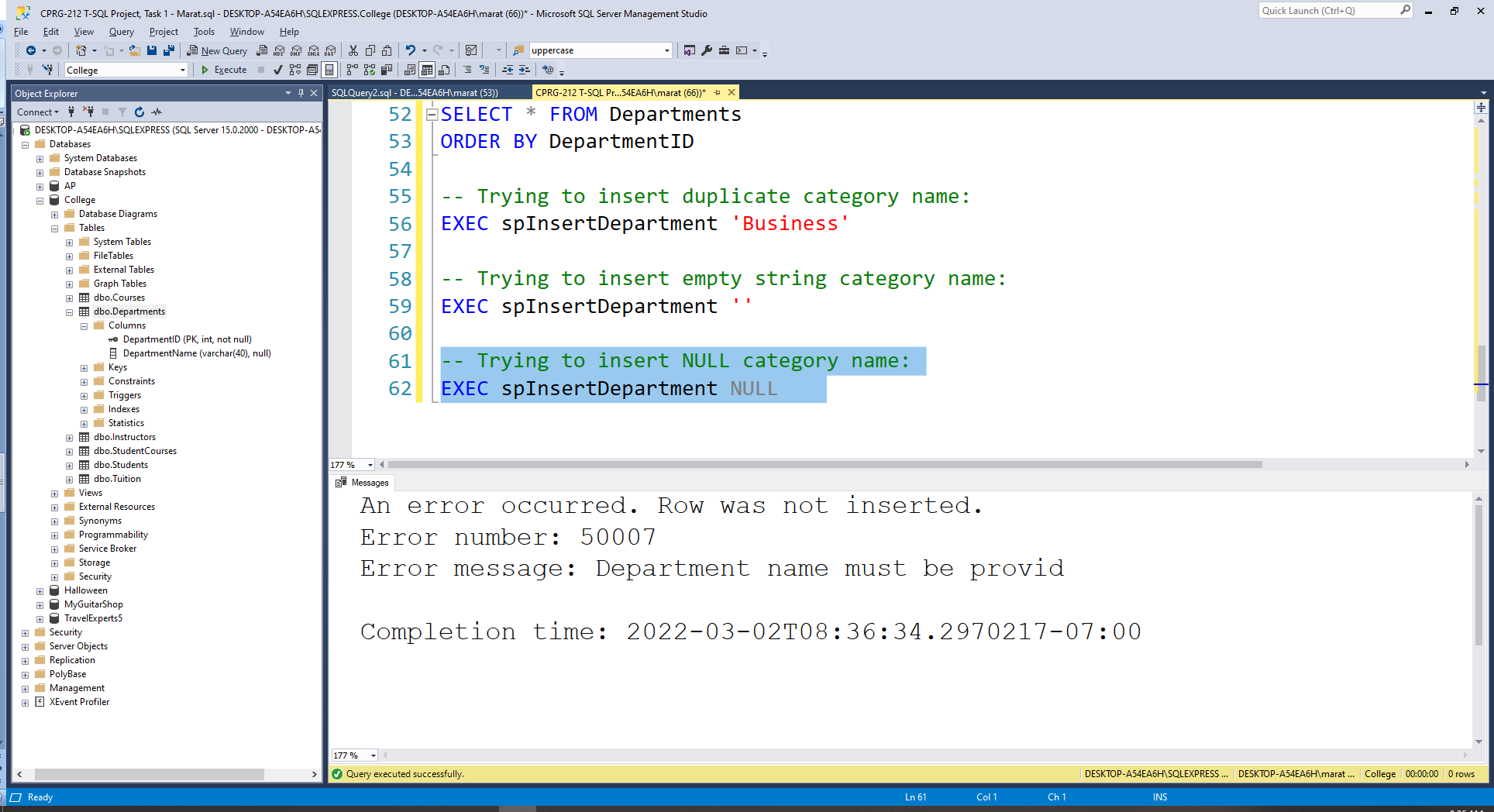
-- Trying to insert empty string category name:

EXEC spInsertDepartment ''



-- Trying to insert NULL category name:

EXEC spInsertDepartment NULL



1. Create a function named fnStudentUnits that calculates the sum of course units of a student. This function accepts one parameter, the student ID, and returns an integer value that is the sum of the course units for the student. You can find courses that the student takes in StudentCourses table, and CourseUnits for each course in the Courses tables.

If the student does not exist or has no courses, this function should return 0.

Code three tests: 1)for a student who has courses, 2) for student who does not have courses, and 3) a non-existing studentID. For each test,display the value of the student ID that was passed to the function and the result returned by the function. Also, run a supportive SELECT query or queries that prove the test results are correct.

Paste here code of your function, statements for the three tests, select query or queries, and screenshots with the results.

USE College;

GO

IF OBJECT\_ID('fnStudentUnits') IS NOT NULL

DROP FUNCTION fnStudentUnits;

GO

CREATE OR ALTER FUNCTION fnStudentUnits(@StudentID INT)

RETURNS INT

AS

BEGIN

DECLARE @Result INT

-- Checking first if a given StudentID exists:

IF (SELECT StudentID FROM Students WHERE StudentID = @StudentID) IS NULL

SET @Result = 0 -- returning 0 for a non-existing StudentID

ELSE -- it means that the given StudentID exists (this side works

-- both for students with courses and without courses)

BEGIN

SELECT @Result = SUM(Courses.CourseUnits)

FROM Students JOIN StudentCourses

ON Students.StudentID = StudentCourses.StudentID

JOIN Courses

ON Courses.CourseID = StudentCourses.CourseID

WHERE Students.StudentID = @StudentID

END

IF @Result IS NULL -- It means this student does not have courses

SET @Result = 0 -- Assigning 0 units to students without courses

RETURN @Result

END

GO

-- Supportive query to prove that test results are correct

-- Among StudentID = 1, 5, 999 only StudentID = 5 has courses,

-- StudentID = 1 exists but doesn't have courses, S

SELECT Students.StudentID, Courses.CourseID, Courses.CourseUnits

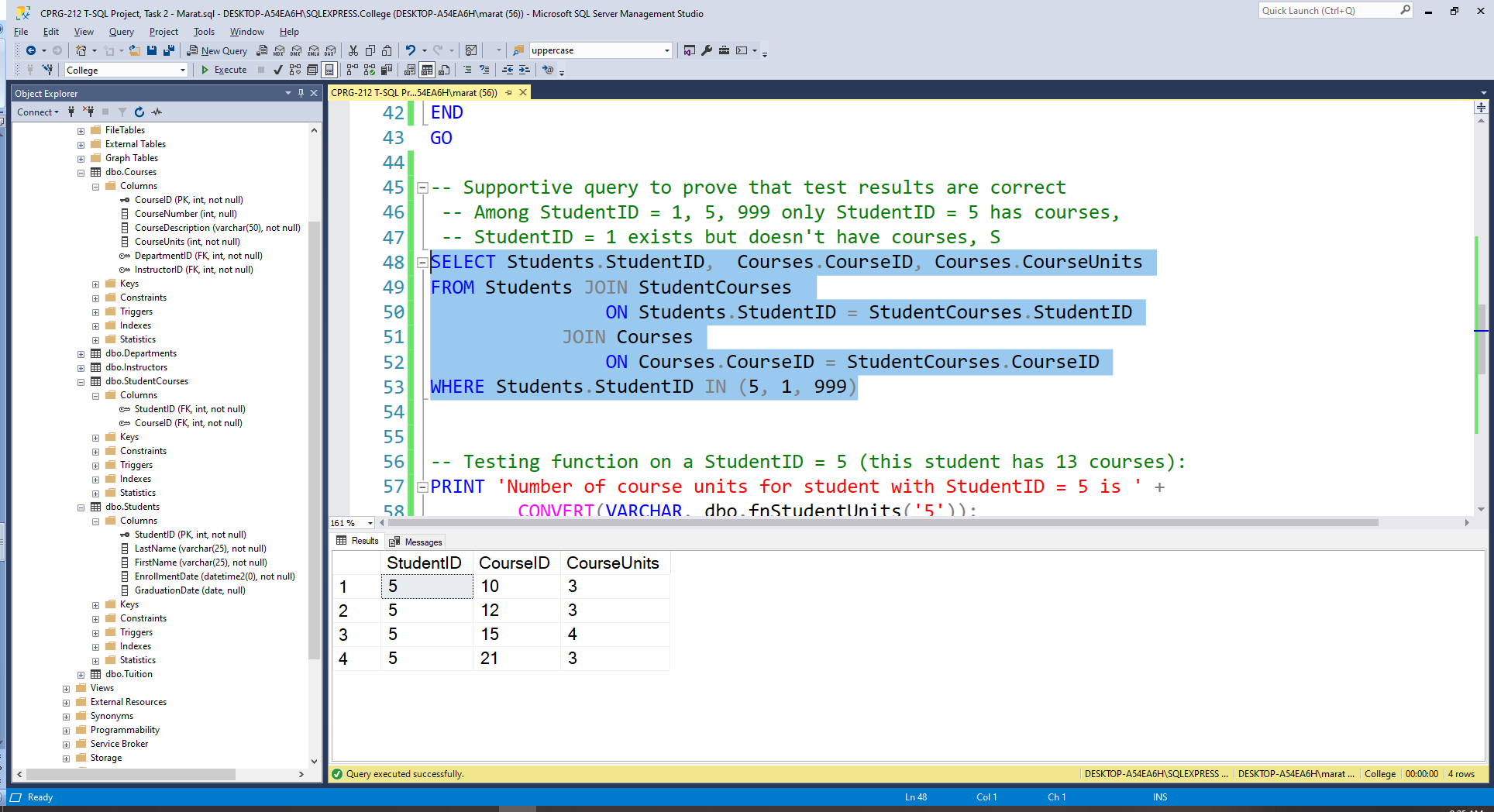
FROM Students JOIN StudentCourses

ON Students.StudentID = StudentCourses.StudentID

JOIN Courses

ON Courses.CourseID = StudentCourses.CourseID

WHERE Students.StudentID IN (5, 1, 999)



-- Testing function on a StudentID = 5 (this student has 13 courses):

PRINT 'Number of course units for student with StudentID = 5 is ' +

CONVERT(VARCHAR, dbo.fnStudentUnits('5'));

GO

-- Testing function on a StudentID = 1 (this student does not have courses):

PRINT 'Number of course units for student with StudentID = 1 is ' +

CONVERT(VARCHAR, dbo.fnStudentUnits('1'));

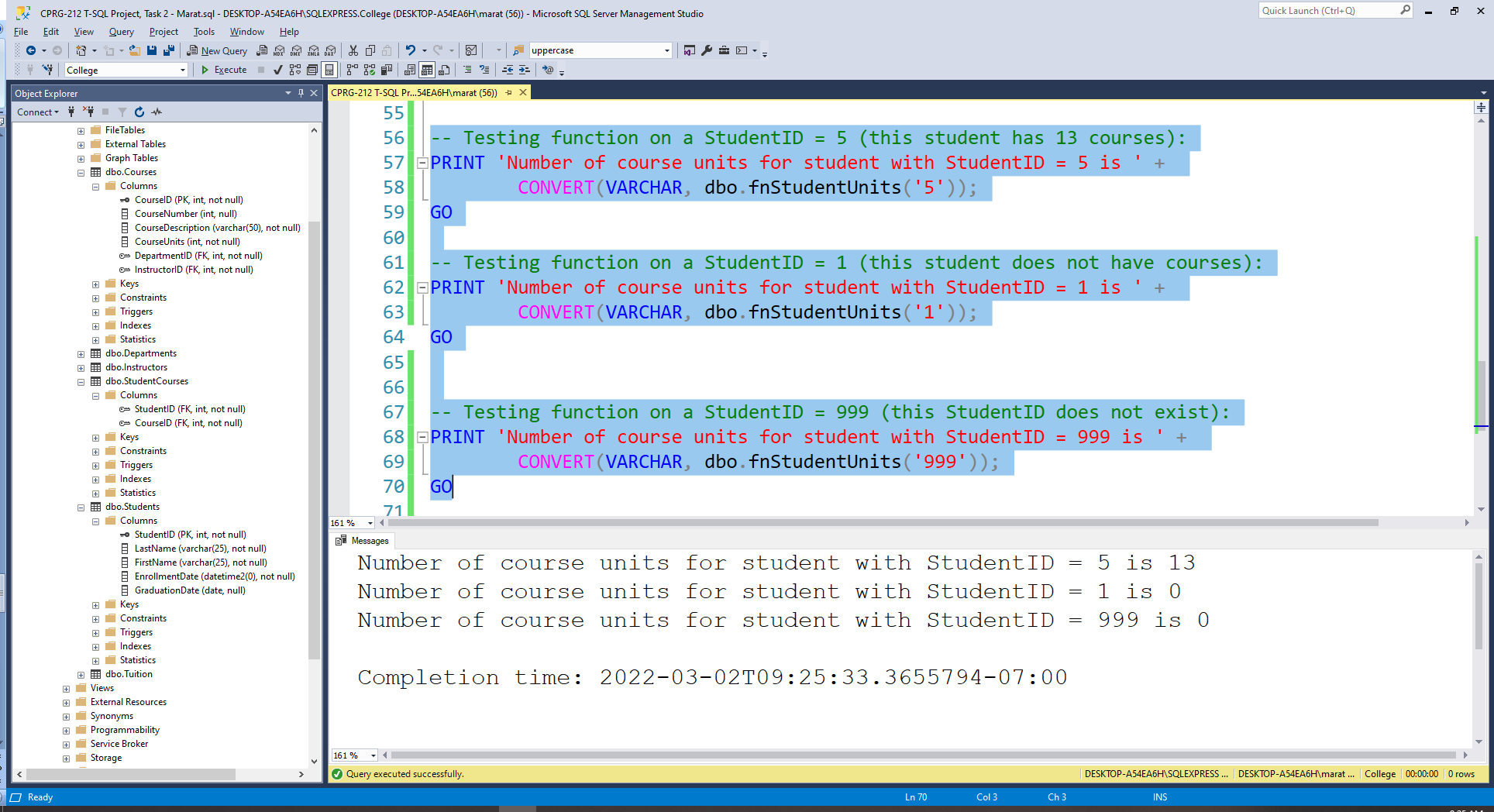
GO

-- Testing function on a StudentID = 999 (this StudentID does not exist):

PRINT 'Number of course units for student with StudentID = 999 is ' +

CONVERT(VARCHAR, dbo.fnStudentUnits('999'));

GO



1. Create a function named fnTuition that calculates the tuition for a student. This function accepts one parameter, the student ID, and it calls the fnStudentUnits function that you created in task 2. The tuition value for the student calculated according to the following pseudocode:

**if(student does not exist)** or (student units = 0)

**tuition = 0**

else if (student units >= 9)

tuition = (full time cost) +(student units) \* (per unit cost)

else

tuition = (part time cost) +(student units) \* (per unit cost)

Retrieve values ofFullTimeCost, PartTimeCost, and PerUnitCost from table Tuition.

**~~If there is no student with the ID passed to the function, the function should return -1.~~**

(There was a contradiction between two statements, and Jolanta clarified in MS Teams chat that function should return 0 if StudentID does not exist)

Code two tests: 1) a student who has <9 student units, and 2) for a student who has >= 9 student units. For each test, display StudentID and the result returned by the function. Also, run supportive SELECT query or queries that prove the results to be correct.

Paste here code of your function, statements for bothtests and select queryor queries, and screenshots with the results.

USE College;

GO

IF OBJECT\_ID('fnTuition') IS NOT NULL

DROP FUNCTION fnTuition;

GO

CREATE OR ALTER FUNCTION fnTuition(@StudentID INT)

RETURNS MONEY

AS

BEGIN

DECLARE @Tuition MONEY

-- Checking first if a given StudentID exists using task 2 function:

IF (dbo.fnStudentUnits(@StudentID) = 0)

-- the task 2 function returns 0 both for non-existing StudentIDs

-- and students without courses, for all of them Tuition = 0:

SET @Tuition = 0

ELSE -- it means that the given StudentID exists (this side works

-- both for students with courses and without courses)

BEGIN

IF (dbo.fnStudentUnits(@StudentID) >= 9) -- it's full-time

SELECT @Tuition = (SELECT FullTimeCost FROM Tuition) +

dbo.fnStudentUnits(@StudentID)\*(SELECT PerUnitCost FROM Tuition)

ELSE -- it's part time

SELECT @Tuition = (SELECT PartTimeCost FROM Tuition) +

dbo.fnStudentUnits(@StudentID)\*(SELECT PerUnitCost FROM Tuition)

END

IF @Tuition IS NULL -- It means this student does not have courses

SET @Tuition = 0 -- Assigning 0 units to students without courses

RETURN @Tuition

END

GO

-- Testing function on a StudentID = 5 (this full-time student has 13 units):

PRINT 'Tuition for the student with StudentID = 5 (full time, 13 units) is $' +

CONVERT(VARCHAR, dbo.fnTuition('5'));

GO

-- Testing function on a StudentID = 10 (this part-time student has 7 units):

PRINT 'Tuition for the student with StudentID = 10 (part-time, 7 units) is $' +

CONVERT(VARCHAR, dbo.fnTuition('10'));

GO

-- Testing function on a StudentID = 1 (exists, bot no units):

PRINT 'Tuition for the student with StudentID = 1 (no units) is $' +

CONVERT(VARCHAR, dbo.fnTuition('1'));

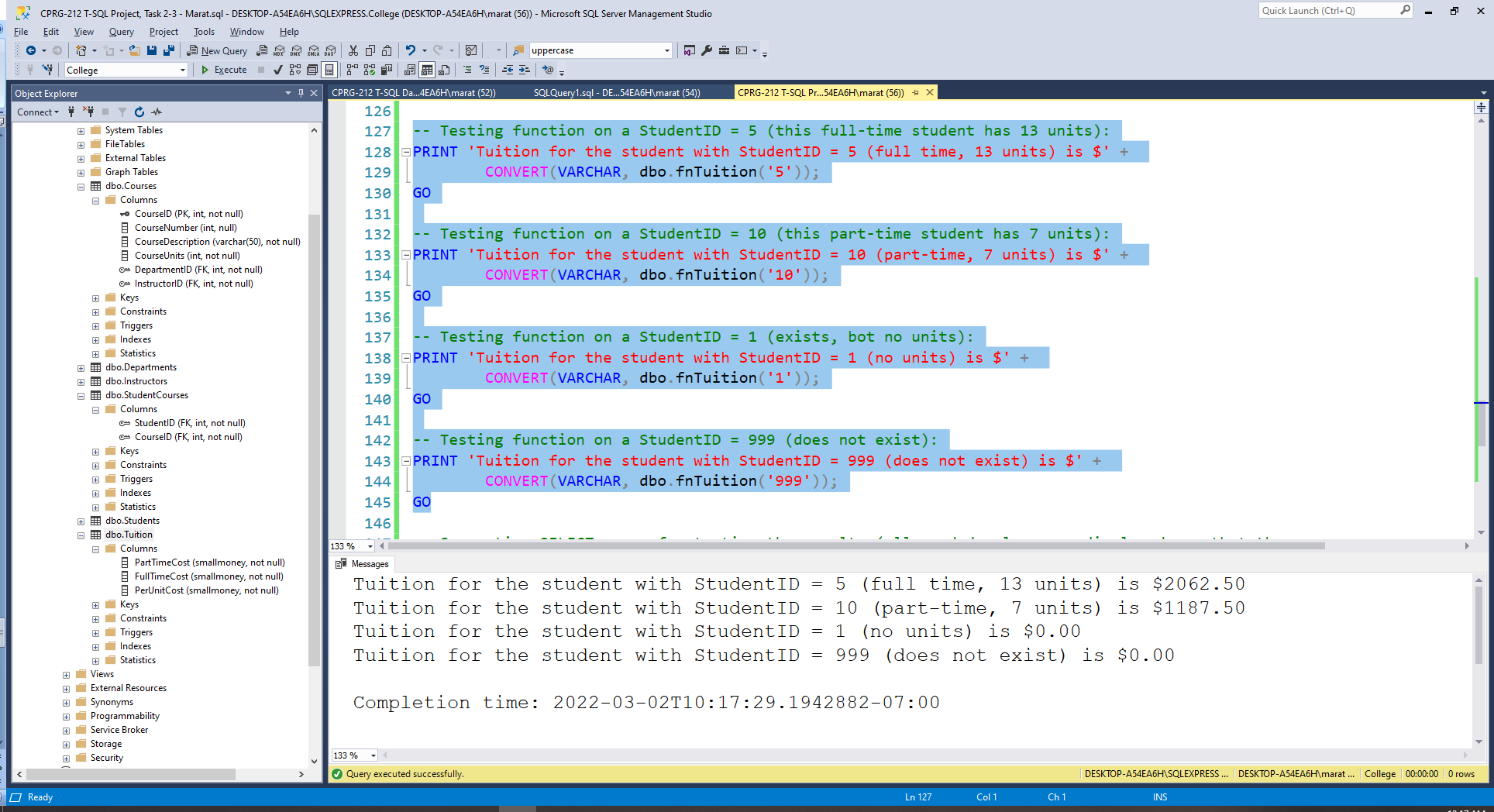
GO

-- Testing function on a StudentID = 999 (does not exist):

PRINT 'Tuition for the student with StudentID = 999 (does not exist) is $' +

CONVERT(VARCHAR, dbo.fnTuition('999'));

GO



-- Supportive SELECT query for testing the results (all needed values are displayed, so that the

-- result can be easily tested checked a calculator: 1250+62.5\*13 = 2062.5 & 750+62.5\*7=1187.5):

SELECT StudentCourses.StudentID,

CASE WHEN (SUM(Courses.CourseUnits) >= 9)

THEN CONCAT('Full-Time Fee: $', (SELECT FullTimeCost FROM Tuition))

ELSE CONCAT('Part-Time Fee: $', (SELECT PartTimeCost FROM Tuition)) END AS CommentOnFee,

(SELECT PerUnitCost FROM Tuition) AS PerUnitCost,

SUM(Courses.CourseUnits) AS CourseUnits#,

CASE WHEN (SUM(Courses.CourseUnits) >= 9)

THEN (SELECT FullTimeCost FROM Tuition)+(SELECT PerUnitCost FROM Tuition)\*SUM(Courses.CourseUnits)

ELSE (SELECT PartTimeCost FROM Tuition)+(SELECT PerUnitCost FROM Tuition)\*SUM(Courses.CourseUnits)

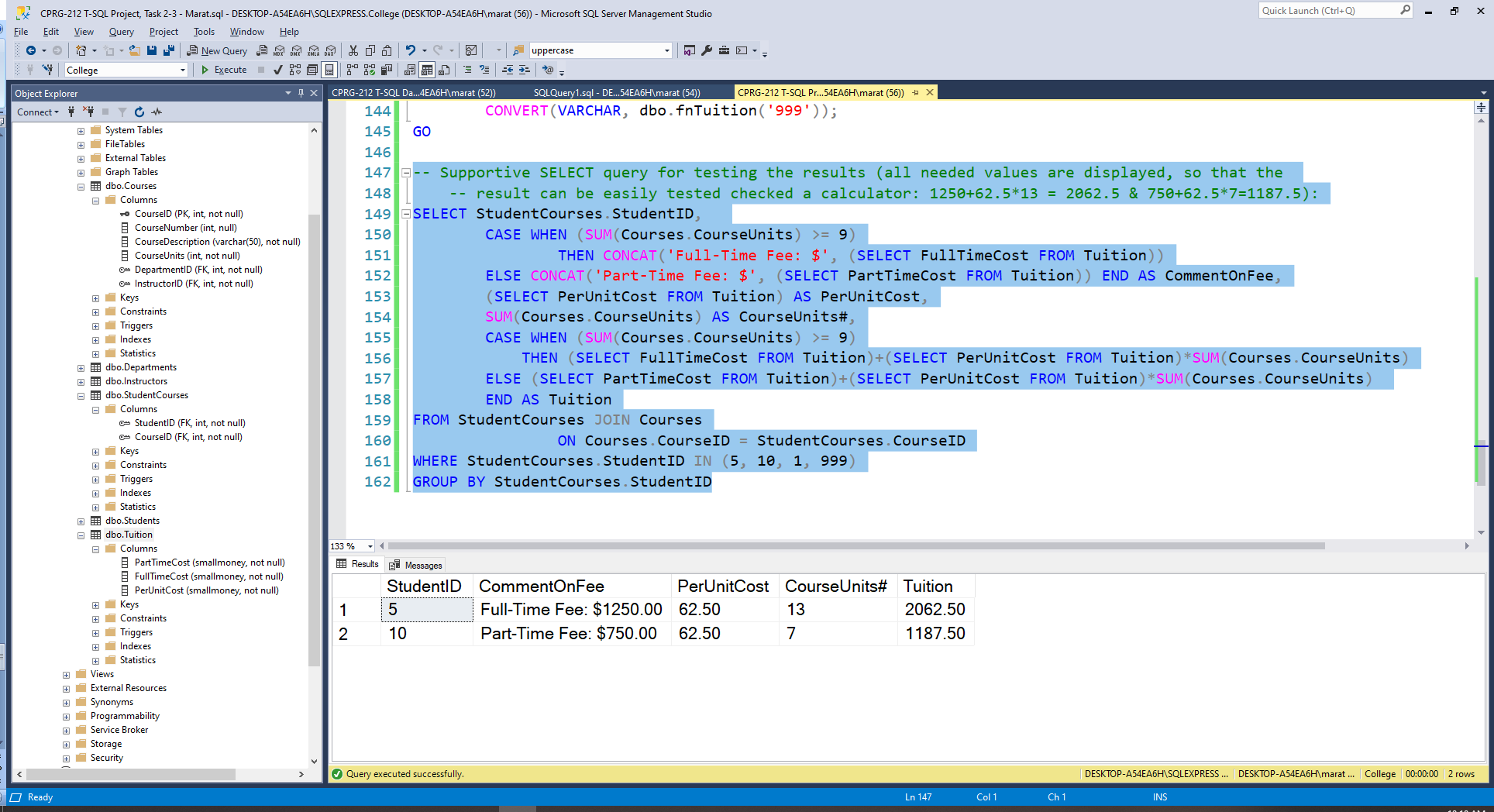
END AS Tuition

FROM StudentCourses JOIN Courses

ON Courses.CourseID = StudentCourses.CourseID

WHERE StudentCourses.StudentID IN (5, 10, 1, 999)

GROUP BY StudentCourses.StudentID



1. Create a trigger named InstructorInsertSalaryTR that fires when a new row is added to the Instructors table.

Throw an error when multiple rows are inserted.

When there is only one row inserted, validate that the AnnualSalary value is positive (strictly greater than zero) and less than or equal to 120000. Throw an error with appropriate message if the salary value is negative or too big.

Also, if the salary value is between 0 and 10000, assume that there was a mistake of entering monthly salary instead of annual salary, and multiply thesalary value by 12. For example, if the new value of the salary is 5000, it should be changed to 60000.

No need to validate any other data from the inserted row.

Test the trigger with appropriate INSERT statements. There should be four cases: 1) with negative salary, 2) with positive salary <= 10000, 3) with salary greater that 10000 and less than or equal to 120000, and 4) with salary > 120000.

Paste here code of your trigger, statements for the tests, and screenshots with results.

USE College;

GO

IF OBJECT\_ID('InstructorInsertSalaryTR') IS NOT NULL

DROP TRIGGER InstructorInsertSalaryTR;

GO

-- Creating the trigger:

CREATE OR ALTER TRIGGER InstructorInsertSalaryTR

ON Instructors

AFTER INSERT

AS

BEGIN TRY

BEGIN TRANSACTION

-- Throwing an error when multiple rows are inserted:

-- LastName is required in the DB, therefore, we can count by it:

IF (SELECT COUNT(LastName) FROM Inserted) > 1

THROW 50007, 'Please insert 1 row only', 1;

-- Validating that the AnnualSalary value is positive

-- (strictly greater than zero) and less than or equal to 120000:

IF (SELECT AnnualSalary FROM Inserted) <= 0

OR (SELECT AnnualSalary FROM Inserted) > 120000

THROW 50007, 'AnnualSalary must be $0-120K!', 1;

-- if the salary value is between 0 and 10000, assume that there was a mistake of

-- entering monthly salary instead of annual salary, and multiply the

-- salary value by 12:

IF (SELECT AnnualSalary FROM Inserted) > 0

AND (SELECT AnnualSalary FROM Inserted) <= 10000

BEGIN

UPDATE Instructors

SET AnnualSalary = 12 \* (SELECT AnnualSalary FROM Inserted)

WHERE InstructorID = (SELECT InstructorID FROM Inserted);

-- Feedback/warning to a user about magic happening behind the scenes:

PRINT 'Trigger was fired and the inserted AnnualSalary value was multiplied by 12'

END

COMMIT TRANSACTION

END TRY

BEGIN CATCH

ROLLBACK TRANSACTION

PRINT 'Error ' + CONVERT(VARCHAR, Error\_Number()) + ': ' + Error\_Message();

PRINT 'The InstructorInsertSalaryTR trigger was executed:'

PRINT 'Transaction was rolled back - row insert was cancelled.'

END CATCH;

GO -- End of the trigger

-- Testing:

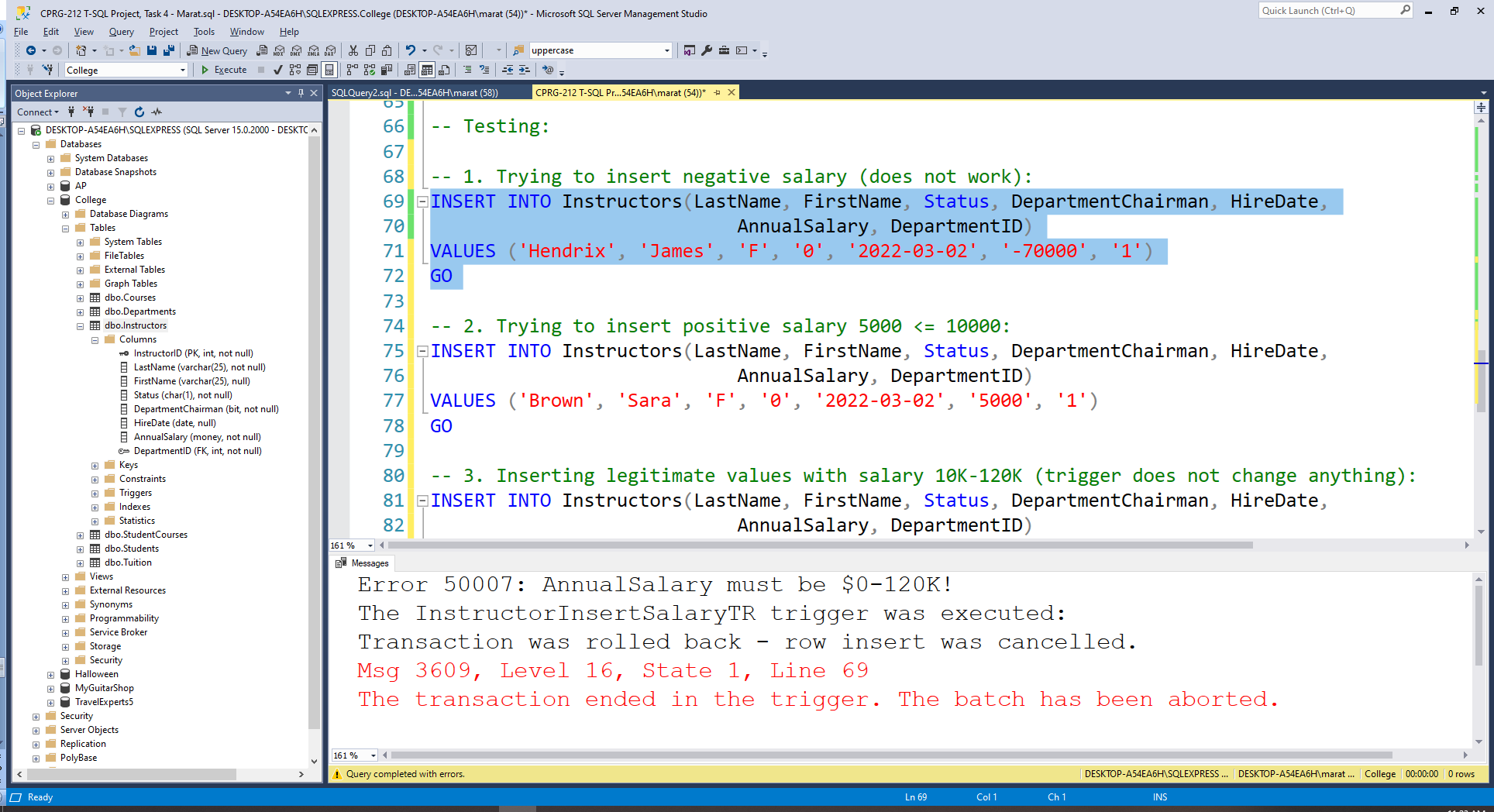
-- 1. Trying to insert negative salary (does not work):

INSERT INTO Instructors(LastName, FirstName, Status, DepartmentChairman, HireDate,

AnnualSalary, DepartmentID)

VALUES ('Hendrix', 'James', 'F', '0', '2022-03-02', '-70000', '1')

GO



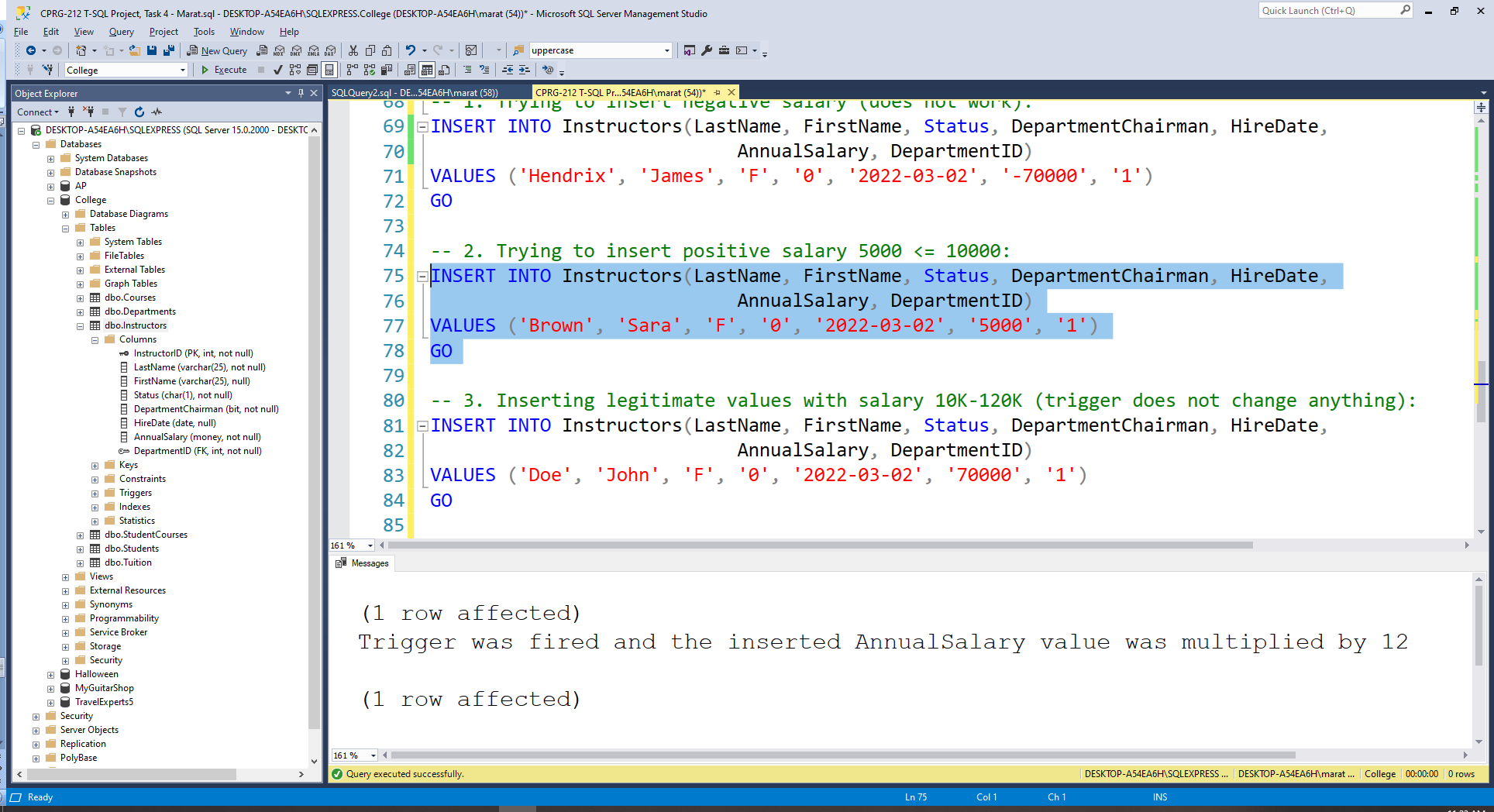
-- 2. Trying to insert positive salary 5000 <= 10000:

INSERT INTO Instructors(LastName, FirstName, Status, DepartmentChairman, HireDate,

AnnualSalary, DepartmentID)

VALUES ('Brown', 'Sara', 'F', '0', '2022-03-02', '5000', '1')

GO



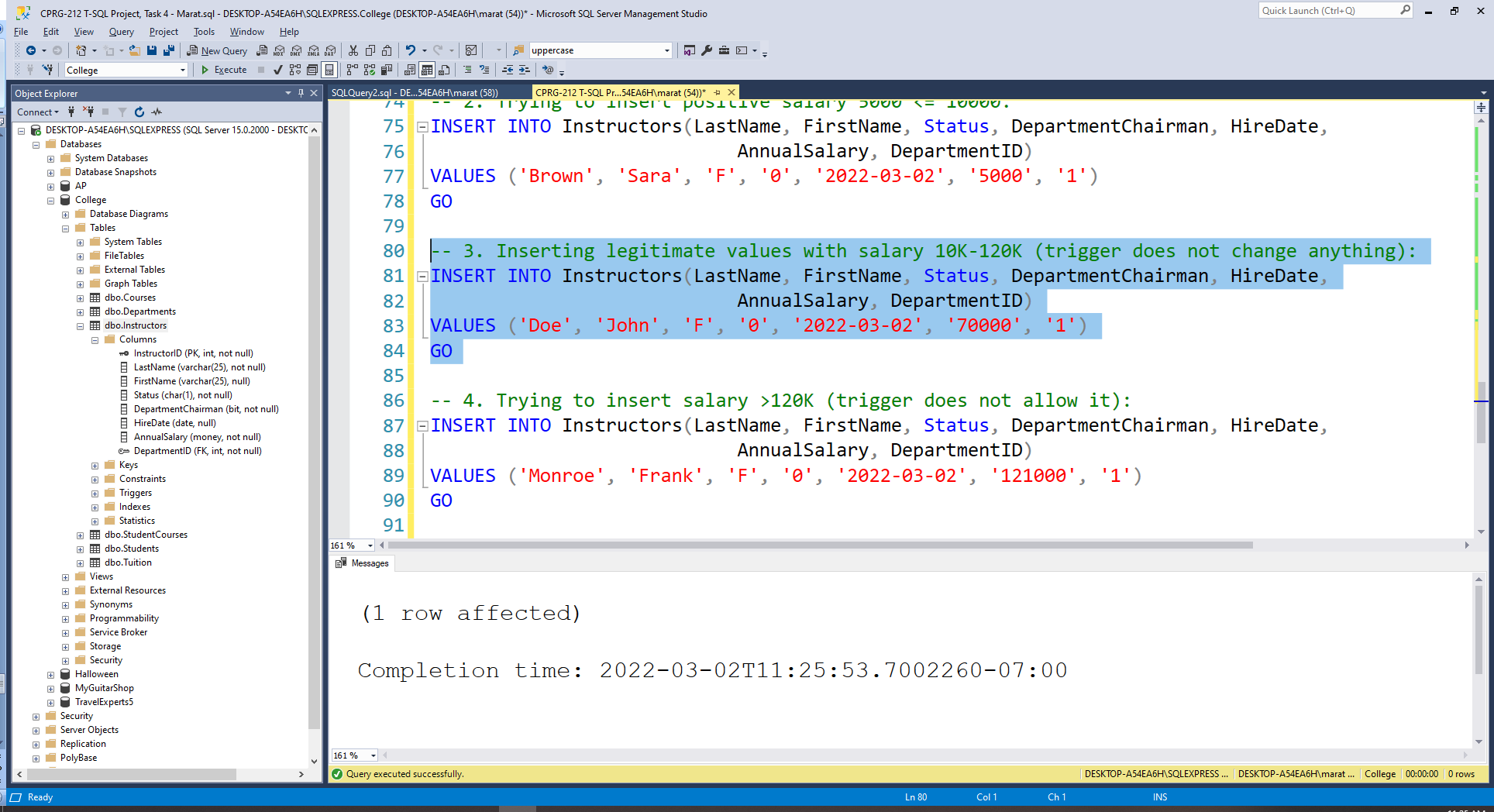
-- 3. Inserting legitimate values with salary 10K-120K (trigger does not change anything):

INSERT INTO Instructors(LastName, FirstName, Status, DepartmentChairman, HireDate,

AnnualSalary, DepartmentID)

VALUES ('Doe', 'John', 'F', '0', '2022-03-02', '70000', '1')

GO



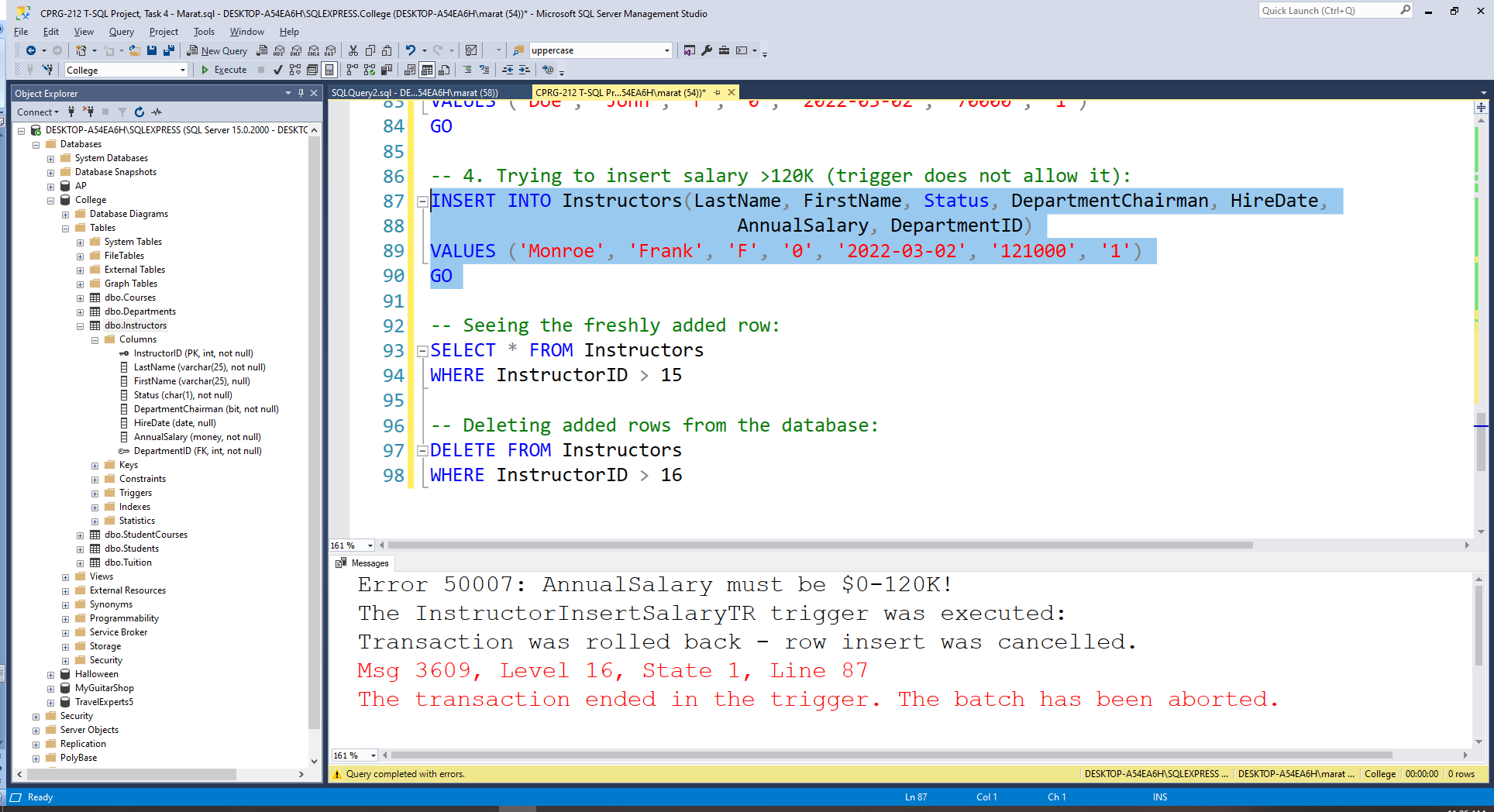
-- 4. Trying to insert salary >120K (trigger does not allow it):

INSERT INTO Instructors(LastName, FirstName, Status, DepartmentChairman, HireDate,

AnnualSalary, DepartmentID)

VALUES ('Monroe', 'Frank', 'F', '0', '2022-03-02', '121000', '1')

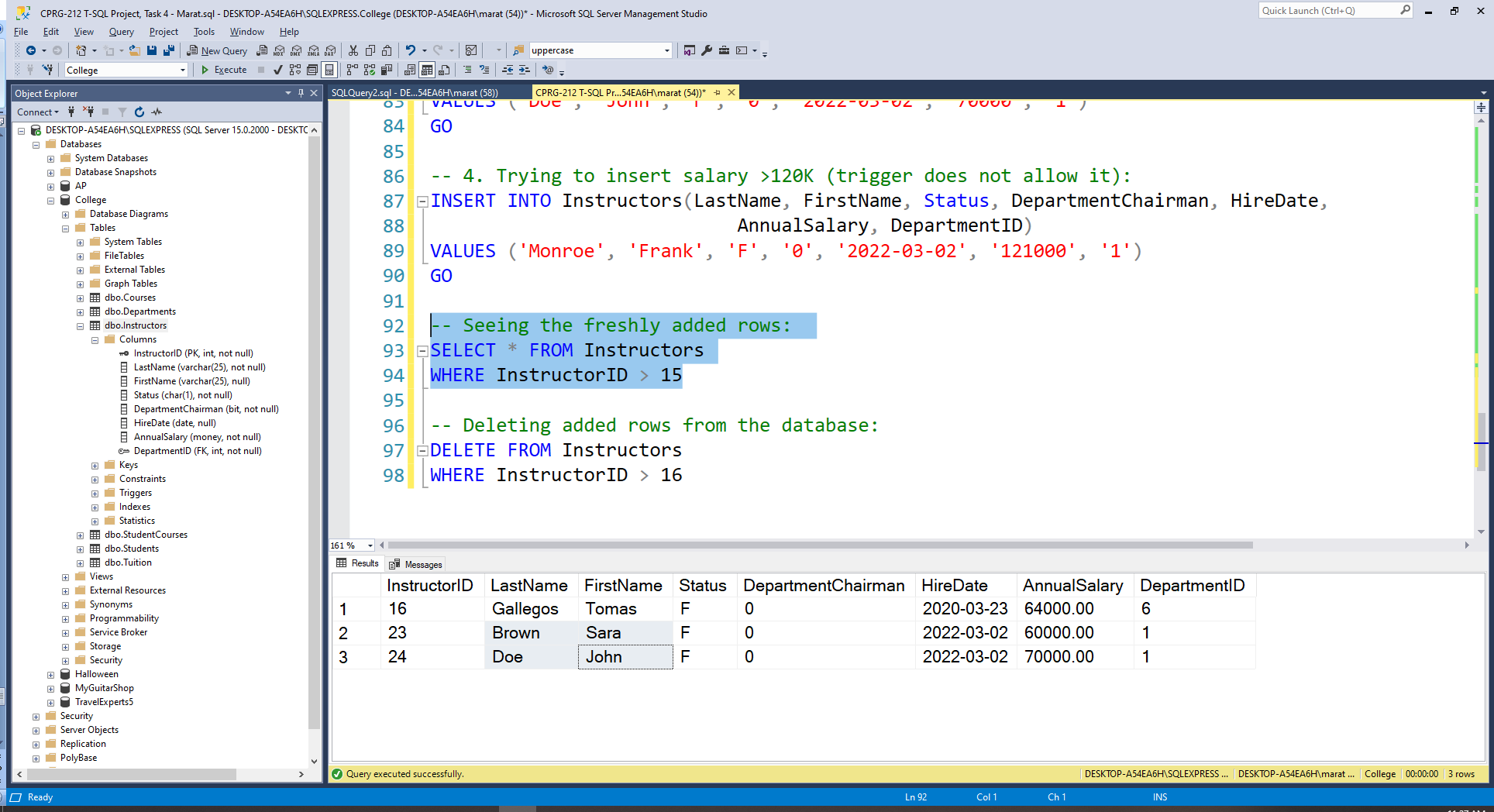
GO



-- Seeing the freshly added rows:

SELECT \* FROM Instructors

WHERE InstructorID > 15



1. Write a script that produces the following report:

For each instructor, display one line with InstructorID, last name, first name, how many courses the instructor teaches, and a note that is defined as follows:

* “On leave”, when instructor teaches no courses,
* “Available for another course”, when instructor teaches only one course, and
* Nothing otherwise

Instructors table contains data about instructors, and each course in the Courses table references InstructorID of an instructor who teaches the course.

The structure of the script is totally up to you, as long as it displays the desired report.

Paste here you script code of your codeand a screenshot with the results it produces.

USE College;

GO

SELECT Instructors.InstructorID, LastName, FirstName,

COUNT(CourseID) AS NumberOfCourses,

CASE WHEN COUNT(CourseID)=0 THEN 'On leave'

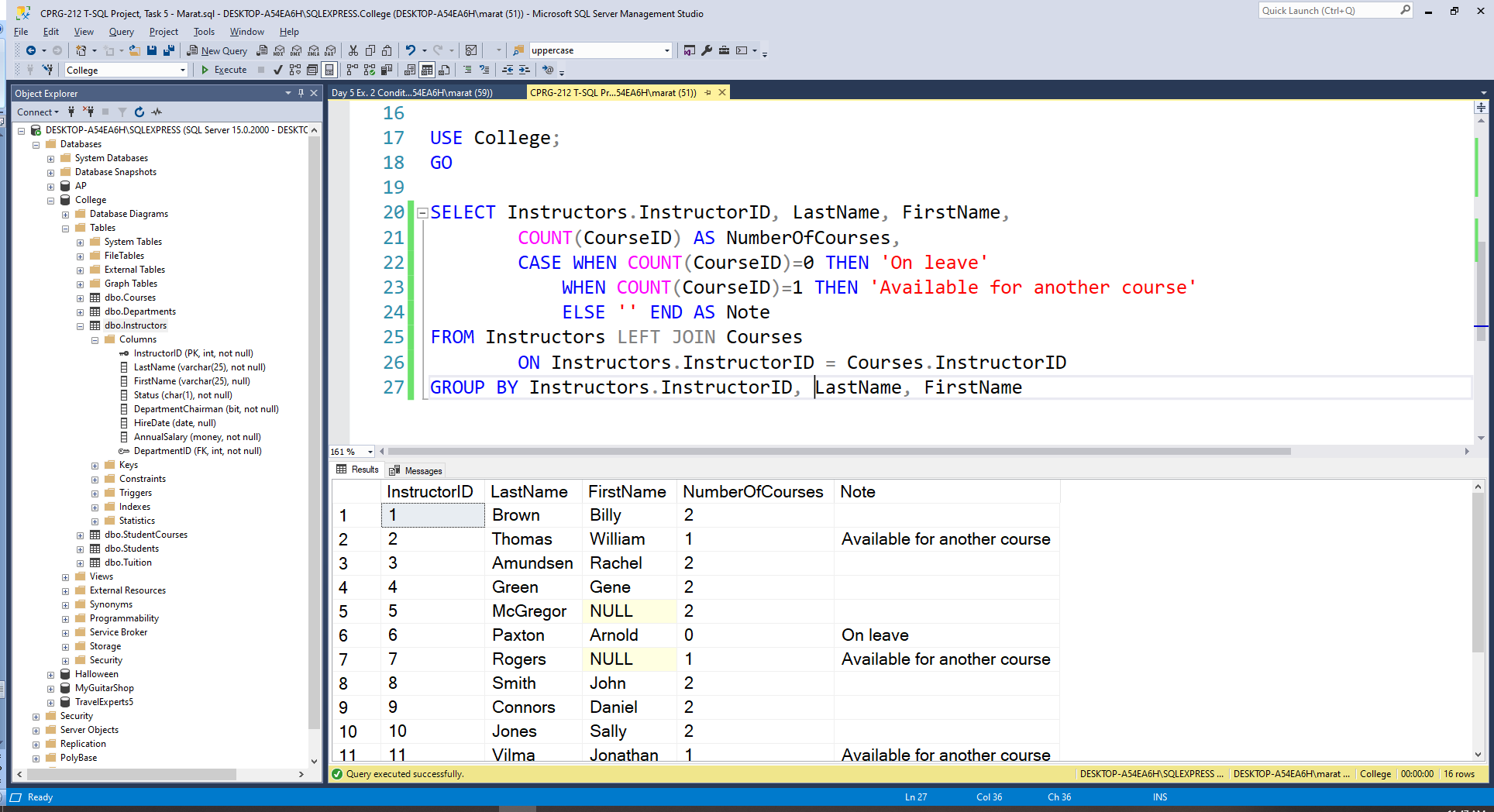
WHEN COUNT(CourseID)=1 THEN 'Available for another course'

ELSE '' END AS Note

FROM Instructors LEFT JOIN Courses

ON Instructors.InstructorID = Courses.InstructorID

GROUP BY Instructors.InstructorID, LastName, FirstName

****

# Submit

Submit on D2L this document with your code and results pasted in places indicated by yellow highlights.

# Marking Rubric

|  |  |
| --- | --- |
| **Marking Requirement** | **Marks** |
| Task 1:procedure code | 10 |
| Task 1: test statements | 6 |
| Task 1: result screenshots | 3 |
| Task 2:function code | 10 |
| Task 2:tests& support query or queries | 6 |
| Task 2:result screenshots | 3 |
| Task 3:function code | 15 |
| Task 3:2 tests & support query or queries | 6 |
| Task 3:result screenshots | 3 |
| Task 4:trigger code | 15 |
| Task 4: tests statements | 6 |
| Task 4:results screenshots | 4 |
| Task 5:script code | 10 |
| Task 5:results screenshot | 3 |
| **Total:** | **100** |