**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 though Departments 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 name has to be unique. The procedure should throw an error with 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.

CREATE OR ALTER PROC spInsertDepartment

@DepartmentName VARCHAR(40) = NULL

AS

-- verify departmentname is present

IF @DepartmentName IS NULL

THROW 50001, 'Department Name must be provided.', 0

-- validate DepartmentName is unique

IF @DepartmentName = (SELECT DepartmentName FROM Departments

WHERE DepartmentName = @DepartmentName)

THROW 50002, 'Department Name must be unique.', 1

--Validation passed so insert data

INSERT Departments

VALUES (@DepartmentName)

1)

EXEC spInsertDepartment

@DepartmentName = NULL

Graphical user interface, text, application

Description automatically generated

2)

EXEC spInsertDepartment

@DepartmentName = 'Computer Science'

Table

Description automatically generated

3)

EXEC spInsertDepartment

@DepartmentName = 'Computer Science'

Graphical user interface, text, application

Description automatically generated

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 student ID. 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.

CREATE OR ALTER FUNCTION fnStudentUnits (@StudentID INT)

RETURNS INT

BEGIN

-- create results variable for easy if/else

DECLARE @Result INT;

-- if student doesn't exist

IF (NOT EXISTS (SELECT StudentID FROM Students

WHERE StudentID = @StudentID))

SELECT @Result = 0;

-- or if student hasn't taken any courses

ELSE IF (NOT EXISTS (SELECT CourseID FROM StudentCourses

JOIN Students ON StudentCourses.StudentID = Students.StudentID

WHERE Students.StudentID = @StudentID))

-- return 0

SELECT @Result = 0;

-- student exists and has courses

ELSE

SELECT @Result = SUM(CourseUnits)

FROM StudentCourses JOIN Courses ON StudentCourses.CourseID = Courses.CourseID

WHERE StudentID = @StudentID

GROUP BY StudentCourses.StudentID

RETURN @Result

END

GO

1)

DECLARE @UnitsCount INT = dbo.fnStudentUnits(5);

PRINT 'Total Course units = ' + Convert(VARCHAR, @UnitsCount)

Graphical user interface, text, application, email

Description automatically generated

2)

DECLARE @UnitsCount INT = dbo.fnStudentUnits(1);

PRINT 'Total Course units = ' + Convert(VARCHAR, @UnitsCount)

Graphical user interface, text, application, email

Description automatically generated

3)

DECLARE @UnitsCount INT = dbo.fnStudentUnits(50);

PRINT 'Total Course units = ' + Convert(VARCHAR, @UnitsCount)

Graphical user interface, text, application, email

Description automatically generated

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 of FullTimeCost, PartTimeCost, and PerUnitCost from table Tuition.

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

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 both tests and select query or queries, and screenshots with the results.

CREATE OR ALTER FUNCTION fnTuition (@StudentID INT)

RETURNS MONEY

BEGIN

-- declare variables

DECLARE @Result MONEY;

DECLARE @FullTimeCost MONEY;

DECLARE @PartTimeCost MONEY;

DECLARE @PerUnitCost MONEY;

-- set required variables

SET @FullTimeCost = (SELECT FullTimeCost FROM Tuition)

SET @PartTimeCost = (SELECT PartTimeCost FROM Tuition)

SET @PerUnitCost = (SELECT PerUnitCost FROM Tuition)

-- if student ID doesn't exist, or does not have courses fnStudentUnits will return 0

-- therefore if 0 is returned, student doesnt exist, or has no courses, set result to 0

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

SELECT @Result = 0;

-- student exists and has enough course units for fulltime status

ELSE IF (dbo.fnStudentUnits(@StudentID) >= 9)

SELECT @Result = (@FullTimeCost + dbo.fnStudentUnits(@StudentID) \* @PerUnitCost)

-- student exists, has courses, but not enough for full time status

ELSE

SELECT @Result = (@PartTimeCost + dbo.fnStudentUnits(@StudentID) \* @PerUnitCost)

RETURN @Result

END1)

DECLARE @TuitionCost MONEY = dbo.fnTuition(5);

PRINT 'Total Tuition costs = ' + FORMAT(@TuitionCost, 'c'

Graphical user interface, text, application

Description automatically generated

SELECT (1250 + dbo.fnStudentUnits(5) \* 62.5)

2)

DECLARE @TuitionCost MONEY = dbo.fnTuition(10);

PRINT 'Total Tuition costs = ' + FORMAT(@TuitionCost, 'c')

Graphical user interface, text, application, email

Description automatically generated

SELECT (750 + dbo.fnStudentUnits(10) \* 62.5)

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 the salary 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.

CREATE OR ALTER TRIGGER InstructorInsertSalaryTR

ON Instructors

INSTEAD OF INSERT

AS

SELECT \* FROM inserted

-- declare needed variables

DECLARE @LastName VARCHAR(25),

@FirstName VARCHAR(25),

@Status CHAR(1),

@DepartmentChairman BIT,

@HireDate DATE,

@AnnualSalary MONEY,

@DepartmentID INT

-- ensure only 1 row inserted

DECLARE @Count INT;

SELECT @Count = COUNT(\*) FROM Inserted;

IF @Count > 1

THROW 50000, 'Insert limited to one row at a time', 0;

-- ensure required values are set

SELECT @LastName = LastName,

@FirstName = FirstName,

@Status = Status,

@DepartmentChairman = DepartmentChairman,

@HireDate = HireDate,

@AnnualSalary = AnnualSalary,

@DepartmentID = DepartmentID

FROM inserted;

-- validate salary range

IF @AnnualSalary < 0

THROW 50001, 'Salary must be greater than 0', 0;

IF @AnnualSalary >= 120000

THROW 50001, 'Salary cannot be greater or equal to 120000', 0;

IF @AnnualSalary > 0 AND @AnnualSalary < 10000

SET @AnnualSalary = @AnnualSalary \* 12;

-- insert all data

INSERT INTO Instructors(LastName, FirstName, Status, DepartmentChairman, HireDate, AnnualSalary, DepartmentID)

VALUES (@LastName, @FirstName, @Status, @DepartmentChairman, @HireDate, @AnnualSalary, @DepartmentID)1)

INSERT INTO Instructors

VALUES ('Doe', 'John', 'F', 1, GetDate(), -100, 1)

Graphical user interface, application

Description automatically generated

2)

INSERT INTO Instructors

VALUES ('Doe', 'John', 'F', 1, GetDate(), 5000, 1)

Graphical user interface, text, application

Description automatically generated

3)

INSERT INTO Instructors

VALUES ('Anderson', 'Thomas', 'F', 1, GetDate(), 100000, 1)

Graphical user interface, application

Description automatically generated

4)

INSERT INTO Instructors

VALUES ('Anderson', 'Thomas', 'F', 1, GetDate(), 200000, 1)

Graphical user interface, text

Description automatically generated with medium confidence

**NOTE: originally the code I was using would still insert values into table even if there were errors. After finishing the question I performed a SELECT \* FROM Instructors and noticed they were being added. I changed the code and updated the triggers code in this file. Just wanted an explanation for the jumping of InstructorIDs.**

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 code and a screenshot with the results it produces.

CREATE OR ALTER FUNCTION fnInstructorCourseCount(@InstructorID INT)

RETURNS INT

BEGIN

DECLARE @Result INT;

SELECT @Result = COUNT(InstructorID) FROM Courses WHERE InstructorID = @InstructorID

RETURN @Result

END

GO

SELECT Instructors.InstructorID, LastName, FirstName, dbo.fnInstructorCourseCount(Instructors.InstructorID) AS [# of Courses],

-- determine number of courses Instructor is teaching and change note accordingly

CASE WHEN (dbo.fnInstructorCourseCount(Instructors.InstructorID) = 0) -- no courses

THEN 'On leave'

WHEN (dbo.fnInstructorCourseCount(Instructors.InstructorID) = 1) -- 1 course

THEN 'Available for another course'

ELSE -- 2 or more courses

'' END AS Note

FROM Instructors LEFT JOIN Courses ON Instructors.InstructorID = Courses.InstructorID

GROUP BY Instructors.InstructorID, LastName, FirstName

Table

Description automatically generated

# 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** |