Project : Student Functionality Requirement Students,

Topic : Courses and Tutors

The following data model is designed to hold information relating to Students, Student Courses and Instructors who tutor students.

For this scenario, we need to define the following entities:

- Student Information
- Courses
- Student Courses  (enrollment)
- Employees (instructors)
- Student Contacts (Contact between the Student and the Instructor)
- Contact Types (Tutor, Test support,etc..)

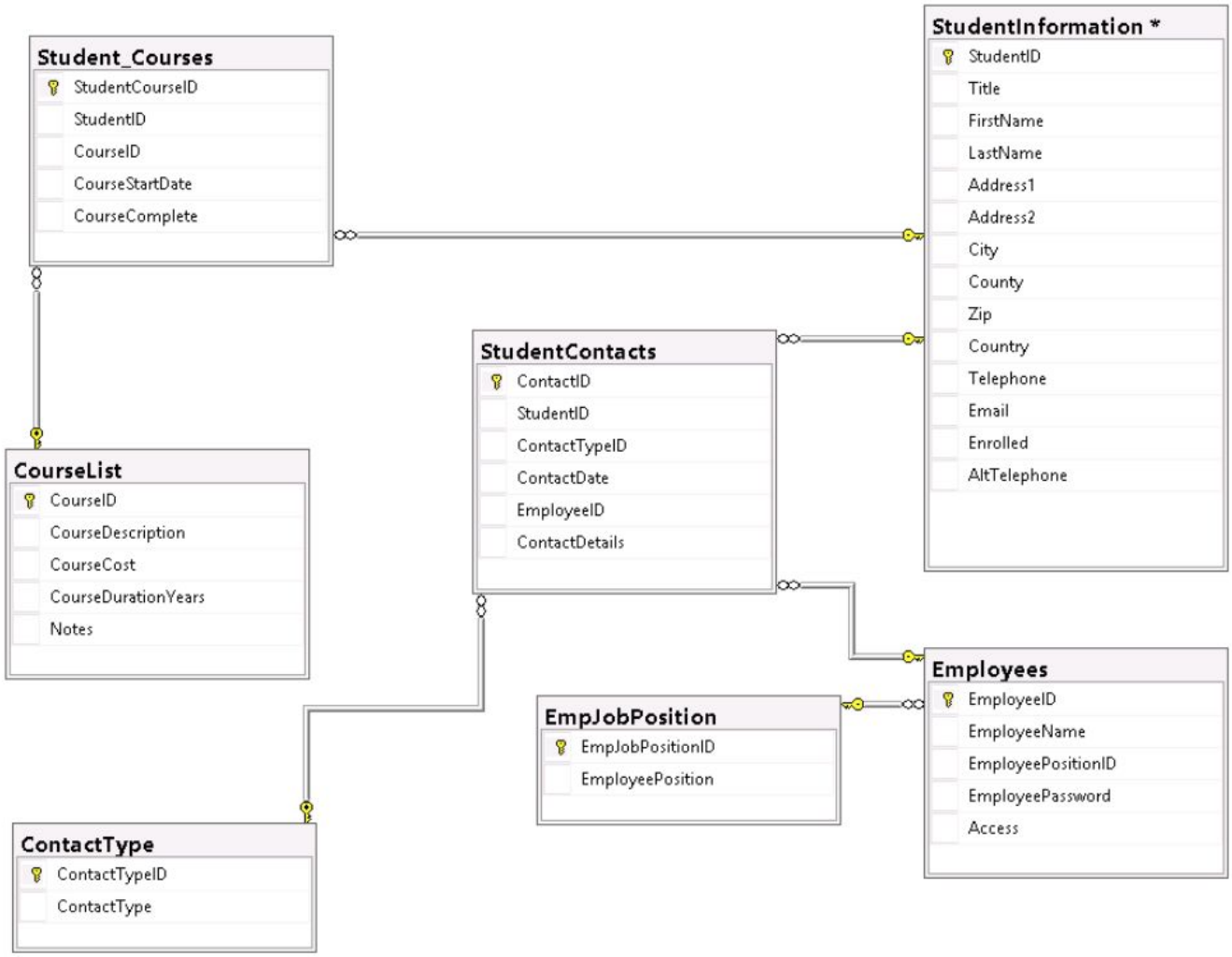
The entities are based on the ER diagram below and use the following rules to determine the table relationships.

- A Student can enroll in one or many Courses
- A Course can have one or many Students enrolled in it.
- A Student can have zero, one, or many forms of contact with the Course Tutor
- An Employee (Tutor) can have many contacts
- A contact Type (Tutor, Test support,etc..) can have zero, one, or many contacts

The design allows ~

- a Student to enroll in one or multiple Courses,
- a Course allowing one or more Students enrolled in it.
- a student may be in contact with the Course Tutor many times using many different forms of contact.
- an instructor can connect with many contacts involving many Students

Entity Relationship Diagram (ERD)




Setting up the project

1. Make a copy of the FinalProject.sql template file (also linked in Canvas) to help guide you through the project.
    a. Download it as a text file and work on it locally (can still have the .sql extension)
    b. In Google Drive -> File->Download As -> Text File
2. Read through the A-I part (sections) below, and add your responses to the problems in your local copy of the project.sql file.

3. Ensure the ENTIRE project.sql runs without errors (use sql commenting if there are any problems you are unable to finish)
4. Upload your FinalProject.sql text file through Canvas in the Final Project Assignment.

*Notes on project.sql :*

Each part has some documentation (below and in the project.sql template) to describe the specific statements needed to answer each part. While the execution order of the script should remain sequential (e.g. Part C executes before B, which executes before A), the order in which you work on the script can happen in any order you want (e.g. if you want to start with part G, and it doesn't depend on something earlier in the script, go for it).

Also, the HINTS with test data are merely "examples", and are NOT REQUIRED in your response. They are there to help guide you. I'll be looking at how you constructed your logic for each of the Parts below instead of resulting data from each Part's query execution.

The total project is worth 200 points

Rubric:

- Part A & Part F are supplied 0 points
- No errors when executing the entire script - 15 points
- Part B - 30 points
- Part C - 50 points
- Part D - 15 points
- Part E - 30 points
- Part G - 30 points
- Part H - 15 points
- Part I - 15 points

<u>Part Task Descriptions</u>

- Part A - Creating the database
    - ***Use the provided template, no action required.***

- Part B -Define and execute **<u>usp_dropTables</u>**
    - *Create a Stored Procedure : usp_dropTables, which executes a series of DROP TABLE statements to remove all the tables created (from the ERD). To prevent errors in trying to drop a table that may not exist, use DROP TABLE IF EXISTS version of your statements.*
    - HINT: Looking at the ERD, CONSTRAINTS are implied. (Trying to drop a table that is a FK to another table will fail). The order in which you drop the tables is

important. When running the stored procedure and the script multiple times, it should run without errors.

    - HINT: test with *EXEC usp_dropTables;*

- Part C - Define and create the tables from the Entity Relationship Diagram
    - Write the CREATE TABLE statements for each of the tables in the Entity Relationship Diagram.
    - Integrate the PRIMARY KEY and FOREIGN KEY CONSTRAINTS in the CREATE TABLE statement itself.
        - Hint: Here's a reference to the statement format
        - https://docs.microsoft.com/en-us/sql/t-sql/statements/create-table-transact-sql or try Google for examples on specifying the PRIMARY and FOREIGN KEY CONSTRAINTS when the table is created.
    - General notes about Table and Entity Relationship Diagram
        - Many of the fields accept NULL, review the INSERT statements in PART F to determine the “NOT NULL” fields, as well as the implied field types.
        - For alpha-numeric data, use char() datatype.
            - Refer to the test examples and the INSERT statements (in Part F) to determine the length of each field (e.g. script should execute without the need to truncate data)
        - Use the following int IDENTITIES for the relevant PRIMARY KEYS in each of the Tables. Again, refer to PART F to help guide how the columns are declared.
            - StudentInformation. StudentID - starts at **100**, increments 1
            - CourseList.CourseID - starts at **10**, increments 1
            - Employees.EmployeeID - starts at **1000**, increments 1
            - StudentContacts.ContactID -starts at **10000**, increments 1
            - StudentCourseID, EmpJobPositionID, ContactTypeID all start at **1**, increments 1

- Part D - Adding columns, constraints, and indexes
    - Modify the table structures and constraints for the following scenarios:
        - Prevent duplicate records in Student_Courses.
            - Specifically, consider a duplicate as a matching of both StudentIDs and CourseIDs (e.g. Composite Key needs to be unique)
        - Add a new column to the StudentInformation table called CreatedDateTime. It should default to the current timestamp when a record is added to the table.
        - Remove the AltTelephone column from the StudentInformation table
        - Add an Index called IX_LastName on the StudentInformation table.

- Part E - Create and apply a Trigger called **trg_assignEmail**
    - Create a trigger on the StudentInfomation table : **trg_assignEmail**
        - When a new row is inserted into StudentInformation without the Email field specified, the trigger will fire and will automatically update the Email field of the record. The Email field will be automatically constructed using the following pattern **firstName.lastName@disney.com** *(e.g. Erik Kellener would be Erik.Kellener@disney.com)*
        - If the insert statement already contains an email address, the trigger does not update the Email field (e.g. ignores the trigger's action)
    - HINT: Use the following test cases
        - Case #1 Test when the email is specified.
            - *INSERT INTO StudentInformation (FirstName,LastName,Email) VALUES ('Porky', 'Pig','porky.pig@warnerbros.com');*
        - Case #2 Test when the email address is not specified.
            - *INSERT INTO StudentInformation (FirstName,LastName) VALUES ('Snow', 'White');*

- Part F - Populating sample data
    - ***Use the template, no action required.***

- Part G - Create and execute **usp_addQuickContacts**
    - Create a stored procedure that allows for quick adds of Student and Instructor contacts: usp_addQuickContacts.
    - The procedure will accept 4 parameters:
        - Student Email
        - EmployeeName
        - contactDetails
        - contactType
    - And performs an INSERT into the StudentsContacts table.
    - When inserting into StudentsContacts, the ContactDate field will automatically default to the current Date.
    - Additionally, upon calling the usp_addQuickContacts procedure,
        - If the contactType parameter value doesn't already exist in the ContactType table, it's first inserted as an additional contactType (e.g. append a new record) AND then used with an INSERT statement to the StudentContacts.
        - If the contactType parameter value does already exist in ContactType, it's corresponding ID is added as part of the StudentContacts INSERT statement.
    - Note: Assume parameters passed to the procedure are valid (e.g. all Student email addresses, and Employee names are correctly entered passed to the procedure)

- HINT: You'll want to initially establish the contactTypeID before moving onto the INSERT statement.
- HINT: Use these test cases to verify the desired output *(Note: Make sure the trg_assignEmail is created and applied before running these test cases)*
  - *EXEC usp_addQuickContacts 'minnie.mouse@disney.com','John Lasseter','Minnie getting Homework Support from John','Homework Support'*
  - *EXEC usp_addQuickContacts 'porky.pig@warnerbros.com','John Lasseter','Porky studying with John for Test prep','Test Prep'*

- Part H - Create and execute **usp_getCourseRosterByName**
  - Create a stored procedure: usp_getCourseRosterByName.
  - It takes 1 parameter, CourseDescription and returns the list of the student's FirstName, and LastName along with the CourseDescription they are enrolled in. (E.g. Student_Courses and CourseList tables are used)
  - Note: Use JOINS. Do not use multiple query statements AND subqueries in the procedure to form your answer.
  - HINT : Use this as a test example:
    - *EXEC usp_getCourseRosterByName 'Intermediate Math';*
    - Expected results:
      - Intermediate Math Mickey Mouse
      - Intermediate Math Minnie Mouse
      - Intermediate Math Donald Duck

- Part I Create and Select from vtutorContacts View
  - Create a view : vtutorContacts, which returns the results from StudentContacts displaying fields EmployeeName, StudentName, ContactDetails, and ContactDate where the contactType is 'Tutor'.
    - EmployeeName doesn't exist in StudentContacts, and may require a JOIN from another table.
    - StudentName doesn't exist, but should be in the form FirstName+' '+LastName. Ensuring both First and Last name are properly trimmed.