1. **Create a database with two tables: Persons (Id (PK), FirstName, LastName, SSN) and Accounts (Id (PK), PersonId (FK), Balance). Insert few records for testing. Write a stored procedure that selects the full names of all persons.**

CREATE DATABASE BankAccount

BEGIN TRAN

CREATE TABLE Persons(

PersonID int IDENTITY,

FirstName nvarchar(50) NOT NULL,

LastName nvarchar(50) NOT NULL,

SSN nvarchar(50) NOT NULL,

CONSTRAINT PK\_Persons PRIMARY KEY(PersonID)

)

GO

COMMIT TRAN

BEGIN TRAN

CREATE TABLE Accounts(

AccountID int IDENTITY,

PersonID int NOT NULL,

Balance money NOT NULL,

CONSTRAINT PK\_Accounts PRIMARY KEY(AccountID),

CONSTRAINT FK\_Accounts\_Persons FOREIGN KEY(PersonID)

REFERENCES Persons(PersonID)

)

GO

COMMIT TRAN

BEGIN TRAN

INSERT INTO Persons

(FirstName, LastName, SSN)

VALUES('Peter', 'Petrov', '123456PP')

INSERT INTO Persons

(FirstName, LastName, SSN)

VALUES('Ivan', 'Ivanov', '123456II')

INSERT INTO Persons

(FirstName, LastName, SSN)

VALUES('Maria', 'Dimitrova', '123456MD')

INSERT INTO Persons(FirstName, LastName, SSN)

VALUES('Penka', 'Ivanova', '123456PI')

COMMIT TRAN

BEGIN TRAN

INSERT INTO Accounts(PersonID, Balance)

VALUES(1, 1000)

INSERT INTO Accounts(PersonID, Balance)

VALUES(2, 500)

INSERT INTO Accounts(PersonID, Balance)

VALUES(3, 2000)

INSERT INTO Accounts(PersonID, Balance)

VALUES(4, 1500)

COMMIT TRAN

USE BankAccount

GO

CREATE PROC SelectPersonFullname

AS

SELECT FirstName + ' ' + LastName

FROM Persons

1. **Create a stored procedure that accepts a number as a parameter and returns all persons who have more money in their accounts than the supplied number.**

CREATE PROC usp\_PersonsHavingMoreMoneyThan(@moneyAmount int = 1000)

AS

SELECT FirstName + ' ' + LastName

FROM Persons p

JOIN Accounts a

ON p.PersonID = a.PersonID

WHERE a.Balance > @moneyAmount

1. **Create a function that accepts as parameters – sum, yearly interest rate and number of months. It should calculate and return the new sum. Write a SELECT to test whether the function works as expected.**

CREATE FUNCTION ufn\_CalculateSumAfterInterest(

@sum money,

@yearlyInterestRate money,

@numberOfMonths money

)

RETURNS money

AS

BEGIN

DECLARE @newSum money

SET @newSum = @sum + @sum \* @yearlyInterestRAte / @numberOfMonths

RETURN @newSum

END

GO

SELECT dbo.ufn\_CalculateSumAfterInterest(1000, 0.05, 6)

AS [Sum after interest]

GO

1. **Create a stored procedure that uses the function from the previous example to give an interest to a person's account for one month. It should take the AccountId and the interest rate as parameters.**

CREATE PROCEDURE dbo.usp\_GiveOneMonthInterest(@AccountId INT, @InterestRate FLOAT)

AS

DECLARE @AccBalance FLOAT, @RESULT FLOAT

SELECT @AccBalance = a.balance

FROM Accounts a

WHERE a.AccountID = @AccountId

SELECT dbo.ufn\_CalculateSumAfterInterest(@AccBalance,@InterestRate,1)

GO

EXEC usp\_GiveOneMonthInterest 1, 0.05

1. **Add two more stored procedures WithdrawMoney( AccountId, money) and DepositMoney (AccountId, money) that operate in transactions.**

CREATE PROC usp\_WithdrawMoney(@AccountID int, @moneyAmount money)

AS

BEGIN TRAN

DECLARE @accountBalance money

SELECT @accountBalance = a.Balance - @moneyAmount

FROM Accounts a

WHERE a.AccountID = @AccountID

UPDATE Accounts

SET Balance = @accountBalance

WHERE AccountID = @AccountID

COMMIT TRAN

EXEC usp\_WithdrawMoney 1, 125

CREATE PROC usp\_DepositMoney(@AccountID int, @moneyToDeposit money)

AS

BEGIN TRAN

DECLARE @updatedBalance money

SELECT @updatedBalance = Balance + @moneyToDeposit

FROM Accounts a

WHERE a.AccountID = @AccountID

UPDATE Accounts

SET Balance = @updatedBalance

WHERE AccountID = @AccountID

COMMIT TRAN

EXEC usp\_DepositMoney 1, 130

1. **Create another table – Logs (LogID, AccountID, OldSum, NewSum). Add a trigger to the Accounts table that enters a new entry into the Logs table every time the sum on an account changes.**

BEGIN TRAN

CREATE TABLE Logs(

LogID int IDENTITY,

AccountID int NOT NULL,

OldSum money NOT NULL,

NewSum money NOT NULL,

CONSTRAINT PK\_Logs PRIMARY KEY(LogID),

CONSTRAINT FK\_Logs\_Accounts FOREIGN KEY(AccountID)

REFERENCES Accounts(AccountID)

)

GO

CREATE TRIGGER tr\_AccountUpdate ON Accounts FOR UPDATE

AS

DECLARE @AccountID int, @oldSum money, @newSum money

SELECT @oldSum = Balance

FROM deleted

SELECT @newSum = Balance, @AccountID = AccountID

FROM inserted

INSERT INTO Logs(AccountID, OldSum, NewSum)

VALUES(@AccountID, @oldSum, @newSum)

GO

COMMIT TRAN

1. **Define a function in the database TelerikAcademy that returns all Employee's names (first or middle or last name) and all town's names that are comprised of given set of letters. Example 'oistmiahf' will return 'Sofia', 'Smith', … but not 'Rob' and 'Guy'.**

USE TelerikAcademy

GO

CREATE FUNCTION ufn\_IsComposed(

@name nvarchar(50),

@letters nvarchar(50)

)

RETURNS BIT

AS

BEGIN

DECLARE @index int = 1,

@foundIndex int,

@currentLetter nvarchar(1),

@counter int,

@result bit

DECLARE @usedLetters table(LetterIndex int, Letter nvarchar(1))

SET @letters = LOWER(@letters)

WHILE(@index <= LEN(@name))

BEGIN

SET @currentLetter = LOWER(SUBSTRING(@name, @index, 1))

SET @foundIndex = CHARINDEX(@currentLetter, @letters)

IF(@foundIndex = 0)

BEGIN

SET @result = 0

BREAK

END

ELSE

BEGIN

IF(EXISTS(SELECT \* FROM @usedLetters

WHERE LetterIndex = @foundIndex))

BEGIN

SELECT TOP 1 @foundIndex = LetterIndex

FROM @usedLetters

WHERE Letter = @currentLetter

ORDER BY Letter DESC

SET @foundIndex = CHARINDEX(@currentLetter,

@letters, @foundIndex + 1)

IF(@foundIndex = 0)

BEGIN

SET @result = 0

BREAK

END

END

INSERT INTO @usedLetters

VALUES(@foundIndex, @currentLetter)

END

SET @index = @index + 1

END

SELECT @counter = COUNT(\*)

FROM @usedLetters

IF(@counter = LEN(@name))

BEGIN

SET @result = 1

END

ELSE

BEGIN

SET @result = 0

END

RETURN @result

END

GO

USE TelerikAcademy

GO

CREATE FUNCTION ufn\_GetComposedNames(

@letters nvarchar(50)

)

RETURNS TABLE

AS

RETURN(

(SELECT 'First Name: ' + e.FirstName AS Name

FROM Employees e

WHERE 1 = (SELECT dbo.ufn\_IsComposed(e.FirstName, @letters)))

UNION

(SELECT 'Middle Name: ' + e.MiddleName AS Name

FROM Employees e

WHERE 1 = (SELECT dbo.ufn\_IsComposed(e.MiddleName, @letters)))

UNION

(SELECT 'Last Name: ' + e.LastName AS Name

FROM Employees e

WHERE 1 = (SELECT dbo.ufn\_IsComposed(e.LastName, @letters)))

UNION

(SELECT 'Town Name: ' + t.Name AS Name

FROM Towns t

WHERE 1 = (SELECT dbo.ufn\_IsComposed(t.Name, @letters)))

)

GO

SELECT \*

FROM dbo.ufn\_GetComposedNames('oistmiahf')

GO

1. **Using database cursor write a T-SQL script that scans all employees and their addresses and prints all pairs of employees that live in the same town.**

USE TelerikAcademy

GO

DECLARE lineCursor CURSOR READ\_ONLY FOR

SELECT e1.FirstName, e1.LastName, t1.Name, e2.FirstName, e2.LastName

FROM Employees e1

INNER JOIN Addresses a1

ON a1.AddressID = e1.AddressID

INNER JOIN Towns t1

ON a1.TownID = t1.TownID,

Employees e2

INNER JOIN Addresses a2

ON e2.AddressID = a2.AddressID

INNER JOIN Towns t2

ON t2.TownID = a2.TownID

WHERE t1.TownID = t2.TownID and

e1.EmployeeID <> e2.EmployeeID

ORDER BY t1.Name, e1.FirstName, e2.FirstName

OPEN lineCursor

DECLARE @firstName1 nvarchar(50),

@lastName1 nvarchar(50),

@town nvarchar(50),

@firstName2 nvarchar(50),

@lastName2 nvarchar(50)

DECLARE @resultTable table(

FirstEmployee nvarchar(100),

Town nvarchar(500),

SecondEmployee nvarchar(100)

)

FETCH NEXT FROM lineCursor

INTO @firstName1, @lastName1, @town, @firstName2, @lastName2

WHILE @@FETCH\_STATUS = 0

BEGIN

INSERT INTO @resultTable

VALUES(@firstName1 + ' ' + @lastName1, @town,

@firstName2 + ' ' + @lastName2)

FETCH NEXT FROM lineCursor INTO @firstName1, @lastName1,

@town, @firstName2, @lastName2

END

CLOSE lineCursor

DEALLOCATE lineCursor

SELECT \* FROM @resultTable

GO

1. **Write a T-SQL script that shows for each town a list of all employees that live in it. Sample output:**

**Sofia -> Svetlin Nakov, Martin Kulov, George Denchev**

**Ottawa -> Jose Saraiva**

**…**

USE TelerikAcademy

GO

DECLARE lineCursor CURSOR READ\_ONLY FOR

SELECT t.Name AS TownName,

e.FirstName + ' ' + e.LastName AS EmployeesName

FROM Employees e

INNER JOIN Addresses a

ON e.AddressID = a.AddressID

INNER JOIN Towns t

ON t.TownID = a.TownID

OPEN lineCursor

DECLARE @employeesName nvarchar(50),

@townName nvarchar(50)

DECLARE @resultTable table(

TownName nvarchar(50),

EmployeesName nvarchar(4000)

)

FETCH NEXT FROM lineCursor INTO @townName, @employeesName

WHILE @@FETCH\_STATUS = 0

BEGIN

IF(EXISTS(SELECT \* FROM @resultTable WHERE TownName = @townName))

BEGIN

UPDATE @resultTable

SET EmployeesName = EmployeesName + ', ' + @employeesName

WHERE TownName = @townName

END

ELSE

BEGIN

INSERT INTO @resultTable

VALUES(@townName, @employeesName)

END

FETCH NEXT FROM lineCursor INTO @townName, @employeesName

END

CLOSE lineCursor

DEALLOCATE lineCursor

SELECT \* FROM @resultTable

ORDER BY TownName

GO

1. **Define a .NET aggregate function StrConcat that takes as input a sequence of strings and return a single string that consists of the input strings separated by ','. For example the following SQL statement should return a single string:**

**SELECT StrConcat(FirstName + ' ' + LastName)**

**FROM Employees**

DECLARE @name nvarchar(MAX);

SET @name = N'';

SELECT @name+=e.FirstName + N', '

FROM Employees e

SELECT LEFT(@name, LEN(@name) - 1);