```
/****************/
/* SQL SERVER   */
/*    INDEX     */
/****************/
```

```
/**************/
/* SQL SERVER */
/**************/

---------------------------------------
-- 48 -- DERIVED TABLES AND CTE -- 48 --
---------------------------------------

-- given the following view

CREATE VIEW vWEmployeeCount
AS
SELECT DeptName, DepartmentId, COUNT(*) AS TotalEmployees
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId
GROUP BY DeptName, DepartmentId
```

```
SELECT DeptName, TotalEmployees FROM vWEmployeeCount
WHERE TotalEmployees >= 2

-- if we are creating a view just to do the select statement afterwards
-- is not the most efficient way
-- we can make use of other constructs


-- TEMPORARY TABLES --


SELECT DeptName, DepartmentId, COUNT(*) AS TotalEmployees
INTO #TempEmployeeCount
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId
GROUP BY DeptName, DepartmentId

SELECT DeptName, TotalEmployees
FROM #TempEmployeeCount
WHERE TotalEmployees >= 2

DROP TABLE #TempEmployeeCount

-- Local temp tables are visible only in the current session,
-- and can be shared between nested stored procedure calls.


-- TABLE VARIABLE --


DECLARE @tblEmployeeCount TABLE(DeptName NVARCHAR(20), DepartmentId INT,
TotalEmployees INT)

INSERT @tblEmployeeCount
SELECT DeptName, DepartmentId, COUNT(*) AS TotalEmployees
FROM tblEmployee
JOIN tblDepartment
ON tblEmployee.DepartmentId = tblDepartment.DeptId
GROUP BY DeptName, DepartmentId

SELECT DeptName, TotalEmployees
FROM @tblEmployeeCount
WHERE TotalEmployees >= 2

-- is also created in the temp db
-- you don't have to drop it like the temp table
-- DERIVED TABLE  --

SELECT DeptName, TotalEmployees
FROM
            (
                  SELECT DeptName, DepartmentId, COUNT(*) AS TotalEmployees
                  FROM tblEmployee
                  JOIN tblDepartment
                  ON tblEmployee.DepartmentId = tblDepartment.DeptId
                  GROUP BY DeptName, DepartmentId
            )
AS EmployeeCount -- this name can be treated as a table itself
WHERE TotalEmployees >= 2

-- EmployeeCount is the derived table


-- CTE (COMMON TABLE EXPRESSION)  --

WITH EmployeeCount(DeptName, DepartmentId, TotalEmployees)
AS
(
      SELECT DeptName, DepartmentId, COUNT(*) AS TotalEmployees
      FROM tblEmployee
      JOIN tblDepartment
      ON tblEmployee.DepartmentId = tblDepartment.DeptId
      GROUP BY DeptName, DepartmentId
)

SELECT DeptName, TotalEmployees
FROM EmployeeCount
WHERE TotalEmployees >= 2

-- EmployeeCount is the CTE Name
-- the select within parenthesis is assigned to teh CTE EmployeeCount
-- its very similar to the derived table:
-- it is not stored as an object and lasts only for the duration of the query


-------------------------------------------------
-- 49 -- COMMON TABLE EXPRESSIONS (CTE) -- 49 --
-------------------------------------------------

-- a CTE is a temporary result set, that can be referenced within a SELECT
-- INSERT, UPDATE OR DELETE statement that IMMEDIATELY follows the CTE
-- there cannot be anything in between, like another select statement
-- in between. The CTE will be available only to a SELECT, INSERT, UPDATE
-- or DELETE that IMMEDIATELY follows the CTE
```

```
WITH EmployeeCount(DepartmentId, TotalEmployees) -- columns that make up the
-- the CTE table, alternatively you could write WITH EmployeeCount
-- since the names of the columns in CTE query is the same
-- if you choose to spell out in the WITH statement the names of the
-- columns, keep in mind that:
-- the number of columns in the WITH statement must match
-- the number of columns in the CTE SELECT query, because they will be matched
AS
(
	SELECT DepartmentId, COUNT(*) as TotalEmployees
	FROM tblEmployee
	GROUP BY DepartmentId
)
-- all of this returns the Employee count CTE

SELECT DeptName, TotalEmployees
FROM tblDepartment -- which has DeptName
JOIN EmployeeCount -- which has TotalEmployees
ON tblDepartment.DeptId = EmployeeCount.DepartmentId
ORDER BY TotalEmployees
-- finally you use teh EmployeeCount CTE to join with a table tblDepartment

-- MULTIPLE CTEs with only one WITH CLAUSE

WITH EmployeesCountBy_Payroll_IT_Dept(DepartmentName, Total)
AS
(
	SELECT DeptName, COUNT(Id) AS TotalEmployees
	FROM tblEmployee
	JOIN tblDepartment
	ON tblEmployee.DepartmentId = tblDepartment.DeptId
	WHERE DeptName IN ('Payroll', 'IT')
	GROUP BY DeptName
),
EmployeesCountBy_HR_Admin_Dept(DepartmentName, Total)
AS
(
	SELECT DeptName, COUNT(Id) AS TotalEmployees
	FROM tblEmployee
	JOIN tblDepartment
	ON tblEmployee.DepartmentId = tblDepartment.DeptId
	WHERE DeptName IN ('HR', 'Admin')
	GROUP BY DeptName
)

-- with the UNION keyword the two selects are treated
-- as one statement. for this reason, it does not
-- matter that we do HR Admin first and Payrroll IT second
```

```
SELECT * FROM EmployeesCountBy_HR_Admin_Dept
UNION
SELECT * FROM EmployeesCountBy_Payroll_IT_Dept


---------------------------------------------------------
-- 50 -- UPDATABLE COMMON TABLE EXPRESSIONS (CTE) -- 50 --
---------------------------------------------------------

-- On some circumstances you can update CTEs

--- CTE AFFECTING ONE UNDERLING TABLE: ALLOWED ---

WITH Employees_Name_Gender
AS
(
	SELECT Id, Name, Gender
	FROM tblEmployee
)

SELECT * FROM Employees_Name_Gender

-- we update the CTE
-- we are updating the CTE, and it updates the table tblEmployee
WITH Employees_Name_Gender
AS
(
	SELECT Id, Name, Gender
	FROM tblEmployee
)

UPDATE Employees_Name_Gender
SET Gender = 'Female' WHERE Id = 1

-- if a CTE is creadte on a one base table, the it is possible
-- to update the CTE and the underlying table

-- let's check
SELECT * FROM tblEmployee


--- CTE based on two underlying tables: update allowed ---
--- only if it affects one of the tables                    ---

-- but when CTE is based on two tables
-- update is allowed only when UPDATE affects only one table
WITH EmployeesByDepartment
AS
(
	SELECT Id, Name, Gender, DeptName
	FROM tblEmployee
	JOIN tblDepartment
	ON tblDepartment.DeptId = tblEmployee.DepartmentId
)
UPDATE EmployeesByDepartment SET Gender = 'Male' WHERE Id = 1

SELECT * FROM tblEmployee
-- Johns gender changed from female to male


--- CTE based on two underlying table, update affect both tables ---
--- not allowed ---

-- update is not allowed only when UPDATE affects both tables
WITH EmployeesByDepartment
AS
(
	SELECT Id, Name, Gender, DeptName
	FROM tblEmployee
	JOIN tblDepartment
	ON tblDepartment.DeptId = tblEmployee.DepartmentId
)
UPDATE EmployeesByDepartment SET Gender = 'Female', DeptName = 'IT'
WHERE Id = 1
-- update is not allowed because gender comes from tblEmployee
-- but DeptName is coming from tblDepartment


--- CTE based on two underlying tables: update affects one table ---
--- but may have repercussions on a different table ---

-- CTE is based on two tables, and update affects only one table
-- but then after the update statement it may also change a different
-- table that references one of the fields in the original table being
-- affected
-- so it may work but not as expected.
-- for example we want to change the department name of a record,
-- but other records sharing that department may be changed as well
WITH EmployeesByDepartment
AS
(
	SELECT Id, Name, Gender, DeptName
	FROM tblEmployee
	JOIN tblDepartment
	ON tblDepartment.DeptId = tblEmployee.DepartmentId
)
UPDATE EmployeesByDepartment SET DeptName = 'IT'
WHERE Id = 1
```

```
--------------------------------------------------------
-- 51 -- RECURSIVE COMMON TABLE EXPRESSIONS (CTE) -- 51 --
--------------------------------------------------------

-- when you have a self referencing table
-- you perform SELF JOIN

-- a CTE that references itself is a recursive CTE

WITH EmployeesCTE(Employee, Name, MangerId, [Level]) -- level is a key
-- word so it needs to be wrapped in square brackets
AS
(
        SELECT EmployeeId, Name, ManagerId, 1
        FROM tblEmployee
        WHERE ManagerId IS NULL

        UNION ALL

        SELECT tblEmployee.EmployeeId, tblEmployee.Name,
        tblEmployee.MangerId, EmployeesCTE.[Level] + 1
        FROM tblEmployee
        JOIN EmployeesCTE
        ON tblEmployee.ManagerId = EmployeesCTE.EmployeeId
)

-- here is the self join:
SELECT ExpCTE.Name AS Employee, ISNULL(MgrCTE.Name, 'President') AS Manager,
EmpCTE.[Level]
FROM EmployeeCTE EmpCTE
LEFT JOIN EmpolyeesCTE MgrCTE
ON ExpCTE.ManagerId = MgrCTE.EmployeeId



-----------------------------------------
-- 52 -- DATABASE NORMALIZATION -- 52 --
-----------------------------------------

-- Normalization is the process of organizing data to minimize
-- data redundancy - data duplication, to ensure data consistency
-- for example if in a table you have a deparment head of it sector
-- and sector and head are separate columns, but for all entries
-- in IT department the department head is the same.

-- without normalization, you have wasted disk space
-- data can be inconsistent, dml queries come become slow (insert
```

```sql
-- update, delete)

-- There are 6 Normal Forms, but most tables are in the 3rd normal
-- form.

-- Advantages of Normalization: faster dml queries, saved disk space
-- less risk of data inconsistency

--- FIRST NORMAL FORM 1NF ---

--1. The DATA IN EACH COLUMN should be ATOMIC
        -- no values separated by comma, for example
        -- Employee column has value Sam, Mike, Shane
        -- you could solve this creating three columns
        -- Employee 1, Employee 2, Employee 3
--2. The table does not contain REPEATING COLUMN GROUPS
        -- Employee 1, Employee 2, Employee 3
        -- These are problematic because some rows may contain
        -- less than three employees, which means wasted disk space
--3. Identify each record uniquely by using PRIMARY KEY



----------------------------------------------
-- 53 -- DATABASE NORMALIZATION (CONT.)-- 53 --
----------------------------------------------

--- SECOND NORMAL FORM 2NF ----

--1. The table meets all conditions of 1NF
--2. Move redundant data to a separate table
        -- If Department Name always coincides with Department Head
        -- and location will always be the same, this is redundant data
        -- So move it to a separate table
        -- The problem here is disk space wastage
--3. Create relationships between tables using foreign keys
        -- your main table will have a primary key and foreign key
        -- that foreign key will be a primary key on a different table

--- THIRD NORMAL FORM 3NF ----

--1. The table meets all conditions of 1NF and 2NF
--2. It does not contain columns(attributes) that are not
        -- fully dependent on the primary key (they depend on another column)

        -- For example if the column AnnualSalary depends on
        -- the column MonthlySalary. You should instead compute
        -- the Annual Salary in your query
        -- Solution: Remove Annual Salary
```

```
	-- Department Head is also dependent on Department Name
	-- whenever the Department Name is IT, the department head
	-- will always be Mike
	-- Solution: Move Department Name and Department Head into a
	-- separate table

----------------------
-- 54 -- PIVOT -- 54 --
----------------------

-- PIVOT can turn unique values in one column into multiple
-- columns in the output, thereby rotating the table

SELECT SalesCountry, SalesAgent, SUM(SalesAmount) AS Total
FROM tblProductSales
GROUP BY SalesCountry, SalesAgent
ORDER BY SalesCountry, SalesAgent

-- nwo we use pivot
SELECT SalesAgent, India, US, UK -- India, US and UK were values of
--SalesCountry
FROM
(
	SELECT SalesAgent, SalesCountry, SalesAmout -- we do this
	-- so that the id column is not selected, which would mess up
	-- our results
	FROM tblProductSales
) AS SourceTable
PIVOT
(
	-- the pivot operator will have an aggregate function, in this case
	-- it is SUM()
	SUM(SalesAmout) -- the values that will appear in each row under each
	-- of the new columns India, US, UK
	FOR SalesCountry -- the column with rows to be converted into columns
	IN ([India],[US],[UK]) -- values to be converted into columns
) AS PivotTable



-------------------------------------------------
-- 55 -- HANDLING ERRORS IN SQL SERVER 2000 -- 55 --
-------------------------------------------------

--@@ prefixed are functions but are known as global variables

-- given a stored procedure
CREATE PROCEDURE spSellProduct 1, 10
```

```
@ProductId INT
@QuantityToSell INT
AS
BEGIN
    -- check the stock available, for the product we want to sell
    DELCARE @StockAvailable INT
    SELECT @StockAvailable = QtyAvailable
    FROM tblProduct
    WHERE ProductId = @ProductId

    -- throw an error to the calling application, if enough stock is not
    -- available
    IF(@StockAvailable < @QuantityToSell)
        BEGIN
            RAISERROR("Not enough stock available", 16, 1)
            -- 16 is the severity level, which indicates the user
            -- can correct the error: reducing the QuantityToSell
            -- 1 is the error state
        END
    -- If enough stock available
    ELSE
        BEGIN
            BEGIN TRAN
            -- First reduce the quantity available
            UPDATE tblProduct SET QtyAvailable = (QtyAvailable - @QuantityToSell)
            WHERE ProductId = ProductId

            DELCARE @MaxPorductSalesId INT
            -- Calculate MAX ProductSalesId
            SELECT @MaxPorductSalesId = CASE
                                                WHEN
MAX(ProductSalesId) IS NULL
                                                THEN 0
                                                ELSE MAX(ProductSalesId)
                                                END
                                                FROM tblProductSales
            -- Increment @MaxProductSalesId by 1, so we don't get a primary key
            -- violation
            SET @MaxProductSalesId = @MaxProductSalesId + 1
            INSERT INTO tblProductSales VALUES(@MaxProductSalesId, @ProductId,
@QuantityToSell)
            COMMIT TRAN
        END
END


-- now intentionally we comment the
            SET @MaxProductSalesId = @MaxProductSalesId + 1
-- line... when we try to insert, we get a primary key violation
```

```
EXECUTE spSellProduct 1, 10
-- our data will become inconsistent
-- the quantity available has been reduced, but no rew row was added
-- to prevent this, we should check for errors
-- using the @@Error

ALTER PROCEDURE spSellProduct 1, 10
@ProductId INT
@QuantityToSell INT
AS
BEGIN
    -- check the stock available, for the product we want to sell
    DELCARE @StockAvailable INT
    SELECT @StockAvailable = QtyAvailable
    FROM tblProduct
    WHERE ProductId = @ProductId

    -- throw an error to the calling application, if enough stock is not
    -- available
    IF(@StockAvailable < @QuantityToSell)
        BEGIN
            RAISERROR("Not enough stock available", 16, 1)
            -- 16 is the severity level, which indicates the user
            -- can correct the error: reducing the QuantityToSell
            -- 1 is the error state
        END
    -- If enough stock available
    ELSE
        BEGIN
            BEGIN TRAN
            -- First reduce the quantity available
            UPDATE tblProduct SET QtyAvailable = (QtyAvailable - @QuantityToSell)
            WHERE ProductId = ProductId

            DELCARE @MaxPorductSalesId INT
            -- Calculate MAX ProductSalesId
            SELECT @MaxPorductSalesId = CASE
                                                        WHEN
MAX(ProductSalesId) IS NULL
                                                        THEN 0
                                                        ELSE MAX(ProductSalesId)
                                                        END
                                                        FROM tblProductSales
            -- Increment @MaxProductSalesId by 1, so we don't get a primary key
            -- violation
            SET @MaxProductSalesId = @MaxProductSalesId + 1
            INSERT INTO tblProductSales VALUES(@MaxProductSalesId, @ProductId,
@QuantityToSell)
            /*******************/
            --CHECK FOR ERRORS--
            /*******************/
            IF (@@Error <> 0)
            BEGIN
                ROLLBACK TRAN
                PRINT 'Transaction Rolledback'
            END
            ELSE
            BEGIN
                COMMIT TRAN
                PRINT 'Transaction Commmitted'
            END
        END
END

-- so long the line is not commented, Transaction will be Commited
EXECUTE spSellProduct 1, 10

SELECT * FROM tblProduct
SELECT * FROM tblProductSales

-- @@Error is cleared and reset on each statement execution
INSERT INTO tblProduct VALUES (2, 'Mobile Phone', 1500, 100)
IF (@@ERROR <> 0)
    PRINT 'Error Occurred'
ELSE
    PRINT 'No Errors'
-- if a record with id 2 exists, well get an error
-- we are checking immediately after the statement where the error
-- occurred, we must do so for @@Error to report the error

-- However, if I'm not checking immediately after the statement
-- execution, because we have a select in between, the @@Error
-- will have been cleared and then used for the select statemnt
INSERT INTO tblProduct VALUES (2, 'Mobile Phone', 1500, 100)
SELECT * FROM tblProduct
IF (@@ERROR <> 0)
    PRINT 'Error Occurred'
ELSE
    PRINT 'No Errors'

-- An alternative is to store @@Error into a variable
DECLARE @Error INT
INSERT INTO tblProduct VALUES (2, 'Mobile Phone', 1500, 100)
SET @Error = @@Error
SELECT * FROM tblProduct
IF (@Error <> 0) -- the local variable contains the value of the @@Error
-- corresponding to the insert statement
    PRINT 'Error Occurred'
```

```
ELSE
    PRINT 'No Errors'


--------------------------------------------------------------------
-- 56-- ERROR HANDLING IN SQL SERVER 2005 AND LATER VERSIONS-- 56 --
--------------------------------------------------------------------

-- In 2005, TRY CATCH block was introduced

-- Let's create a working stored procedure for updating the two
-- tables, increasing the stock of the product in one, and inserting
-- a row with the new purchase in another

CREATE PROCEDURE spSellProduct
@ProductId INT
@QuantityToSell INT
AS
BEGIN
    -- check the stock available, for the product we want to sell
    DELCARE @StockAvailable INT
    SELECT @StockAvailable = QtyAvailable
    FROM tblProduct
    WHERE ProductId = @ProductId

    -- throw an error to the calling application, if enough stock is not
    -- available
    IF(@StockAvailable < @QuantityToSell)
        BEGIN
            RAISERROR("Not enough stock available", 16, 1)
            -- 16 is the severity level, which indicates the user
            -- can correct the error: reducing the QuantityToSell
            -- 1 is the error state
        END
    -- If enough stock available
    ELSE
        BEGIN
            BEGIN TRAN
            -- First reduce the quantity available
            UPDATE tblProduct SET QtyAvailable = (QtyAvailable - @QuantityToSell)
            WHERE ProductId = @ProductId

            DELCARE @MaxPorductSalesId INT
            -- Calculate MAX ProductSalesId
            SELECT @MaxPorductSalesId = CASE
                                                    WHEN
MAX(ProductSalesId) IS NULL
                                                    THEN 0
                                                    ELSE MAX(ProductSalesId)
```

```
                                        END
                                        FROM tblProductSales
            -- Increment @MaxProductSalesId by 1, so we don't get a primary key
            -- violation
            SET @MaxProductSalesId = @MaxProductSalesId + 1
            INSERT INTO tblProductSales VALUES(@MaxProductSalesId, @ProductId,
@QuantityToSell)
            COMMIT TRAN
        END
END

-- so long the line is not commented, Transaction will be Commited
EXECUTE spSellProduct 2, 10

SELECT * FROM tblProduct
SELECT * FROM tblProductSales

-- lets sell 10 more items, BUT let's comment the line
-- that increments the primary key value by 1 such that we
-- get a primary key violation error.

ALTER PROCEDURE spSellProduct
@ProductId INT
@QuantityToSell INT
AS
BEGIN
    -- check the stock available, for the product we want to sell
    DELCARE @StockAvailable INT
    SELECT @StockAvailable = QtyAvailable
    FROM tblProduct
    WHERE ProductId = @ProductId

    -- throw an error to the calling application, if enough stock is not
    -- available
    IF(@StockAvailable < @QuantityToSell)
        BEGIN
            RAISERROR("Not enough stock available", 16, 1)
            -- 16 is the severity level, which indicates the user
            -- can correct the error: reducing the QuantityToSell
            -- 1 is the error state
        END
    -- If enough stock available
    ELSE
        BEGIN
            BEGIN TRAN
            -- First reduce the quantity available
            UPDATE tblProduct SET QtyAvailable = (QtyAvailable - @QuantityToSell)
            WHERE ProductId = @ProductId
            DELCARE @MaxPorductSalesId INT
            -- Calculate MAX ProductSalesId
            SELECT @MaxPorductSalesId = CASE
                                        WHEN
MAX(ProductSalesId) IS NULL
                                        THEN 0
                                        ELSE MAX(ProductSalesId)
                                        END
                                        FROM tblProductSales
            -- Let's comment the line below to get a primary key violation
            --SET @MaxProductSalesId = @MaxProductSalesId + 1
            INSERT INTO tblProductSales VALUES(@MaxProductSalesId, @ProductId,
@QuantityToSell)
            COMMIT TRAN
        END
END

EXECUTE spSellProduct 2, 10

-- this will alter the first table (tblProduct) and not the second (tblProductSales)
-- because of the primary key violation

SELECT * FROM tblProduct
SELECT * FROM tblProductSales

-- let's use error handling

ALTER PROCEDURE spSellProduct
@ProductId INT
@QuantityToSell INT
AS
BEGIN
    DELCARE @StockAvailable INT
    SELECT @StockAvailable = QtyAvailable
    FROM tblProduct
    WHERE ProductId = @ProductId

    IF(@StockAvailable < @QuantityToSell)
        BEGIN
            RAISERROR("Not enough stock available", 16, 1)
]       END
    ELSE
        BEGIN
            BEGIN TRY
                BEGIN TRAN
                -- First reduce the quantity available
                UPDATE tblProduct SET QtyAvailable = (QtyAvailable -
@QuantityToSell)
                WHERE ProductId = @ProductId
                DELCARE @MaxPorductSalesId INT
                -- Calculate MAX ProductSalesId
                SELECT @MaxPorductSalesId = CASE
                                                            WHEN
MAX(ProductSalesId) IS NULL
                                                            THEN 0
                                                            ELSE
MAX(ProductSalesId)
                                                            END
                                                            FROM
tblProductSales
                -- Let's comment the line below to get a primary key violation
                --SET @MaxProductSalesId = @MaxProductSalesId + 1
                INSERT INTO tblProductSales VALUES(@MaxProductSalesId,
@ProductId, @QuantityToSell)
                COMMIT TRAN
            END TRY
            BEGIN CATCH
                ROLLBACK TRANSACTION
                SELECT
                    ERROR_NUMBER() AS ErrorNumber,
                    ERROR_MESSAGE() AS ErrorMessage,
                    ERROR_PROCEDURE() AS ErrorProcedure,
                    ERROR_STATE() AS ErrorState,
                    ERROR_SEVERITY() AS ErrorSeverity,
                    ERROR_LINE() AS ErrorLine,
            END CATCH

        END
END

EXECUTE spSellProduct 2, 10

-- the catch block will execute because of primary key violation
-- and the 6 error functions will return error information
-- Error functions will only work only in the context of a catch block
-- Otherwise they will return NULL

-- Errors trapped by a CATCH block are not returned to the calling
-- application. If any part of the error information must be returned
-- to the application, the code in the CATCH block must do so by using
-- the RAISEERROR() function.

-- Also remember that the TRY CATCH block cannot be used in a user
-- defined function.

-- SHORTCUT : CTRL G, to go to a line number.
-- Also make sure when you get a line number error, in an error message
```

```sql
-- the line number refers no to the line number that appears on your
-- screen in SQL Server Management studio necessarily as it appears in
-- your screen. Rather it refers to the line number that is numbered
-- starting at the line number where the statement that you are executing
-- begins.


------------------------------
-- 57 -- TRANSACTIONS -- 57 --
------------------------------

-- A transaction is a group of commands that change the data stored
-- in a database. A transaction is treated as a single unit, either
-- all of the commands succeed, or none of them. If one of the commands
-- in the transaction fails, all of the commands fails, and any data
-- that way modified in the database is rolled back. In this way,
-- transaction maintain the integrity of data in a database.

-- Transaction processing follows these steps:
-- 1. Begin a transaction
-- 2. Process database commands
-- 3. Check for errors
--    If errors occurred, rollback the transaction
--    If there are no errors, commit the transaction

BEGIN TRANSACTION
UPDATE tblProduct SET QtyAvailable = 300
WHERE ProductId = 1

-- If you execute this, when you do a select statement
-- you will not see the changes because by default sql server
-- does not read uncommitted data, and this transaction is still uncommited

SET TRANSACTION ISOLATION LEVEL READ uncommitted

-- In this way, changing the defaults, wow you can see the uncommitted data

-- You can rollback this uncommitted transaction

ROLLBACK TRANSACTION

-- Until you commit the transaction, the change is not permanent

BEGIN TRANSACTION
UPDATE tblProduct SET QtyAvailable = 300
WHERE ProductId = 1
COMMIT TRANSACTION

-- Now the change is made permanent to the database
-- Let's see how to use transactions with error handling

CREATE PROCEDURE spUpdateAddress
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
            UPDATE tblMailingAddress SET City = 'LONDON'
            WHERE AddressId = 1 AND EmployeeNumber = 101

            UPDATE tblPhysicalAddress SET City = 'LONDON'
            WHERE AddressId = 1 AND EmployeeNumber = 101
        COMMIT TRANSACTION
        PRINT "Transaction Committed"
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION
    END CATCH
END

-- Knowing that City column lenght is only 10 characters
-- the following update (the second) will produce an error
-- and the transaction will be rolled back.

ALTER PROCEDURE spUpdateAddress
AS
BEGIN
    BEGIN TRY
        BEGIN TRANSACTION
            UPDATE tblMailingAddress SET City = 'LONDON2'
            WHERE AddressId = 1 AND EmployeeNumber = 101

            UPDATE tblPhysicalAddress SET City = 'LONDON LONDON'
            WHERE AddressId = 1 AND EmployeeNumber = 101
        COMMIT TRANSACTION
        PRINT "Transaction Committed"
    END TRY
    BEGIN CATCH
        ROLLBACK TRANSACTION
        PRINT "Transaction Rolled Back"
    END CATCH
END


-- The statement above will throw an exception and the catch
-- block will fire, rolling back the transaction and printing
-- "Transaction Rolled Back"
```

```sql
-------------------------------------------
-- 58 -- TRANSACTIONS AND ACID TESTS -- 58 --
-------------------------------------------

-- A transaction is a group of database commands that are treated
-- as a single unit. A successful transaction must pass the "ACID"
-- test, that is, it must be:

-- ATOMIC: All statements in the transaction are either completed
-- successfully or they were all rolled back. The task that the set
-- of the operations represents is either accomplished or not, but in
-- any case is not left half done.

-- CONSISTENT: All data touched by the transaction is left in a logically
-- consistent state. For example, if stock available numbers are decremented
-- from tblProductTable, the, there has to be a related entry in
-- tblProductSales table. The inventory can't just dissapear.

-- ISOLATED: The transaction must affect data without interferring with
-- other concurrent transactions, or being interfered with by them.
-- This prevents transactions from making changes to data besed on uncommited
-- information, for example, changes to a record that are subsequently
-- rolled back. Most databases use locking to maintain transaction isolation.

-- For example, the following set of commands will result in
-- uncommited changes,

BEGIN TRANSACTION
UPDATE tblProduct SET QtyAvailable = 350
WHERE ProductId = 1

-- that could be interefered with by a select statement, which will not
-- show the dat

SELECT * FROM tblProduct

-- it will not be able to return data because the transaction is being
-- processed.

SELECT * FROM tblProduct
WHERE ProductId = 2

-- but this will execute ok, because row 1 is being locked by the database

-- if the user were able to access the data, the user could make decisions
-- based on incomplete or un updated information.

-- DURABLE: Once a change is made, it is permanent. If a system error or
-- power failure occurs before a set of commands is complete, those commands
-- are undone and the data is restored to its original state once the system
-- begins running again.


----------------------------
-- 59 -- SUBQUERIES -- 59 --
----------------------------

-- First let's create two tables

CREATE TABLE tblProducts
{
	[Id] INT IDENTITY PRIMARY KEY,
	[Name] NVARCHAR(50),
	[Description] NVARCHAR(250)
}

CREATE TABLE tblProductSales
{
	[Id] INT IDENTITY PRIMARY KEY,
	ProductId INT FOREIGN KEY REFERENCES tblProducts[Id],
	UnitPrice INT,
	QuantitySold INT
}

-- let's insert some information

INSERT INTO tblProducts VALUES ('TV', '52 inch black color LCD TV')
INSERT INTO tblProducts VALUES ('Laptop', 'Very thin black color ACER laptop')
INSERT INTO tblProducts VALUES ('Desktop', 'HP high performance desktop')

INSERT INTO tblProductSales VALUES (3, 450, 5)
INSERT INTO tblProductSales VALUES (2, 250, 7)
INSERT INTO tblProductSales VALUES (3, 450, 4)
INSERT INTO tblProductSales VALUES (3, 450, 9)

}

-- let's examine the data

SELECT * FROM tblProducts
SELECT * FROM tblProductSales

-- let's select the id, name and description of all products
-- that we have not sold at least once
-- so we are selecting only distinct values in tblProductSales
-- table and we want those records where the Id does not appear
-- in the table that records the sales we've made
```

```
SELECT Id, Name, [Description]
FROM tblProducts
WHERE Id NOT IN (SELECT DISTINCT ProductId FROM tblProductSales)

-- the subquery is also called, the inner query.
-- Subqueries are easily replaceable with JOINS

-- let's rewrite it using a JOIN statement

SELECT Id, Name, [Description]
FROM tblProducts
LEFT JOIN tblProductSales
ON tblProducts.Id = tblProductSales.ProductId
WHERE tblProductSales.ProductId IS NULL
-- this query will return the same as the subquery above


-- Write a query to retrieve the name of the product and the
-- total quantity of the product sold.

SELECT Name, (SELECT SUM(QuantitySold)
                  FROM tblProductSales
                  WHERE ProductId = tblProducts.Id
                  AS QtySold)
FROM tblProducts
ORDER BY Name

-- which can be rewritten in JOIN form as
SELECT Name, SUM(QuantitySold AS QtySold)
FROM tblProducts
LEFT JOIN tblProductSales
ON tblProducts.Id = tblProductSales.ProductId
GROUP BY Name

-- A subquery is simply a sellect statement that returns a single
-- value and can be nested inside a SELECT, UPDATE, INSERT or DELETE
-- statement. It is also possible to nest a subquery inside another
-- subquery. According to MSDN, subqueries can be nested up to 32 levels.

-- Subqueries are always enclosed in parenthesis are are also called
-- inner queries, and the query containing the subquery is called the
-- outter query. The columns from a table that is present only inside a
-- subquery, cannot be used in the SELECT list of the outer query.



------------------------------------
-- 60 -- CORRELATED SUBQUERY -- 60 --
```

```
-----------------------------------

-- If a subquery is dependent on the outer query for it's value, then
-- it is called a correlated subquery.

-- This is NON CORRELATED SUBQUERY

SELECT Id, Name, [Description]
FROM tblProducts
WHERE Id NOT IN (SELECT DISTINCT ProductId
                                    FROM tblProductSales)

-- because the subquery can be run independent of the outer query

-- This is a CORRELATED SUBQUERY
SELECT Name, (SELECT SUM(QuantitySold)
                FROM tblProductSales
                           WHERE ProductId = tblProducts.Id)
FROM tblProducts
-- the subquery depends on the outter query for its value
-- in this case it depends on ProductId

-----------------------------------------------------------------------------
-- 61 -- CREATING A LARGE TABLE WITH RANDOM DATA FOR PERFORMANCE TESTING --
61 --
-----------------------------------------------------------------------------

-- We first check if tables exists, and if they do, we drop them (so
-- we can create them)

-- information_schema_tables is a table that contains a list of
-- tables present in the sample database - if we execute the query
-- in the context of the sample database.

IF (EXISTS (SELECT *
        FROM information_schema_tables
        WHERE table_name = 'tblProductSales'))
BEGIN
        DROP TABLE tblProductSales
END

IF (EXISTS (SELECT *
                        FROM information_schema_tables
                        WHERE table_name = 'tblProducts'))
BEGIN
        DROP TABLE tblProducts
END
```

```
-- Now we create the tables
CREATE TABLE tblProducts
(
	[Id] INT IDENTITY PRIMARY KEY,
	[Name] NVARCHAR(50),
	[Description] NVARCHAR(250)

)

CREATE TABLE tblProductSales
(
	Id INT IDENTITY PRIMARY KEY,
	ProductId INT FOREIGN KEY REFERENCES tblProducts(Id),
	UnitPrice INT,
	QtySold INT
)

-- Now we insert sample data in tblProducts table
Declare @Id INT
Set @Id = 1

WHILE (@Id <= 1000)
BEGIN
	INSERT INTO tblProducts VALUES ('Product = ' + CAST(@Id AS NVARCHAR(20)),
	'Product = ' + CAST(@Id AS NVARCHAR(20)) + ' Description')
	-- above we specify the product name and product description columns
	-- since we defined the id column to be an identity column, we don't
	-- need to worry about providing a value.
	PRINT @Id
	SET @Id = @Id + 1
END

-- Now we insert sample data into the tbllProductSales table

-- First, declare and set variables to generate a random ProductId between
-- 1 and 100000
DECLARE @UpperLimitForProductId INT
DECLARE @LowerLimitForProductId INT

SET @LowerLimitForProductId = 1
SET @UpperLimitForProductId = 8500

-- Declare and set variables to generate a
-- random UnitPrice between 1 and 100
DECLARE @UpperLimitForUnitPrice INT
DECLARE @LowerLimitForUnitPrice INT

SET @LowerLimitForUnitPrice = 1
SET @UpperLimitForUnitPrice = 100
```

```sql
-- Declare and set variables to generate a
-- random QuantitySold between 1 and 10
DECLARE @UpperLimitForQuantitySold INT
DECLARE @LowerLimitForQuantitySold INT

SET @LowerLimitForQuantitySold = 1
SET @UpperLimitForQuantitySold = 10



DECLARE @Counter INT
SET @Counter = 1

WHILE(@Counter <= 15000)
BEGIN
	-- we get rid of decimals with ROUND
	SELECT @RandomProductId = ROUND(((@UpperLimitForProductId -
@LowerLimitForProductId) * RAND() + @LowerLimitForProductId), 0)
	SELECT @RandomUnitPrice = ROUND(((@UpperLimitForUnitPrice -
@LowerLimitForUnitPrice) * RAND() + @LowerLimitForUnitPrice), 0)
	SELECT @RandomQuantitySold = ROUND(((@UpperLimitForQuantitySold -
@LowerLimitForQuantitySold) * RAND() + @LowerLimitForQuantitySold), 0)

	INSERT INTO tblProductSales
	VALUES (@RandomProductId, @RandomUnitPrice, @RandomQuantitySold)

	PRINT @Counter
	SET @Counter = @Counter + 1
END

-- in the end we'll get 10000 products in tblProducts and 15000 records in
-- tblProductSales

------------------------------------------------
-- 62 -- SUBQUERIES VS JOINS: PERFORMANCE -- 62 --
------------------------------------------------


-- According to MSDN, in sql server, in most cases, there is usually no performance difference
-- between queries that uses sub-queries and equivalent queries using joins.
-- For example, on my machine I have
-- 400,000 records in tblProducts table
-- 600,000 records in tblProductSales tables

-- The following query, returns, the list of products that we have sold atleast once. This query
-- is formed using sub-queries. When I execute this query I get 306,199 rows in 6 seconds
Select Id, Name, Description
from tblProducts
```

```
where ID IN
(
 Select ProductId from tblProductSales
)


-- At this stage please clean the query and execution plan cache using the following T-SQL command.
CHECKPOINT;
GO
DBCC DROPCLEANBUFFERS; -- Clears query cache
Go
DBCC FREEPROCCACHE; -- Clears execution plan cache
GO

-- Now, run the query that is formed using joins. Notice that I get the exact same 306,199 rows in 6
seconds.

Select distinct tblProducts.Id, Name, Description
from tblProducts
inner join tblProductSales
on tblProducts.Id = tblProductSales.ProductId

-- According to MSDN, in some cases where existence must be checked, a join produces better
performance.
-- Otherwise, the nested query must be processed for each result of the outer query. In such cases, a join
-- approach would yield better results.

-- The following query returns the products that we have not sold at least once. This query is formed
using
-- sub-queries. When I execute this query I get 93,801 rows in 3 seconds

Select Id, Name, [Description]
from tblProducts
where Not Exists(Select * from tblProductSales where ProductId = tblProducts.Id)

-- At this stage please clean the query and execution plan cache using the following T-SQL command.
CHECKPOINT;
GO
DBCC DROPCLEANBUFFERS; -- Clears query cache
Go
DBCC FREEPROCCACHE; -- Clears execution plan cache
GO


-- When I execute the below equivalent query, that uses joins, I get the exact same 93,801 rows in 3
seconds.

Select tblProducts.Id, Name, [Description]
from tblProducts
```

```
left join tblProductSales
on tblProducts.Id = tblProductSales.ProductId
where tblProductSales.ProductId IS NULL

-- In general joins work faster than sub-queries, but in reality it all depends on the execution plan that is
-- generated by SQL Server. It does not matter how we have written the query, SQL Server will always
transform
-- it on an execution plan. If sql server generates the same plan from both queries, we will get the same
result.

-- I would say, rather than going by theory, turn on client statistics and execution plan to see the
performance
-- of each option, and then make a decision. To get client statistics and execution plan details upon
executing
-- a query, simply press the respective icons in the menu bar in SQL server.


/****************************************************
*63****************** CURSORS *******************63*
*****************************************************/

-- The following update query , updates a set of rows that matches
-- the condition in the where clause at the same time.

UPDATE tblProductSales
SET UnitPrice = 50
WHERE ProductId = 101

-- however if there is ever a need to process the rows on a row
-- by row basis we use cursors which are bad for performance and
-- should normally be avoided. Most of the time cursors can be replaced
-- by Joins

-- There are four cursor types: Forward Only, Static, Keyset, and Dynamic

-- Suppose you have a table with 400,000 records in tblProducts and
-- 600,000 records in tblProductSales
-- Suppose you want to update the UNITPRICE column in tblProductSales table
-- based on the following criteria

-- 1. If the ProductName = 'Product - 55', Set Unit Price to 55
-- 2. If the ProductName = 'Product - 65', Set Unit Price to 65
-- 3. If the ProductName = 'Product - 100%', Set Unit Price to 1000

DELCARE @ProductId INT

-- Declare the cursor using the declare keyword
DECLARE ProductIdCursor CURSOR FOR
SELECT ProductId FROM tblProductSales
```

```
-- Open statement, executes the SELECT statment
-- and populates the result set
OPEN ProductIdCursor

-- Fetch the row from the result set into the variable
FETCH NEXT FROM ProductIdCursor INTO @ProductId

-- If the result set still has rows, @@FETCH_STATUS will be ZERO
WHILE(@@FETCH_STATUS = 0)
BEGIN
	DECLARE @ProductName NVARCHAR(50)
	SELECT @ProductName = Name
	FROM tblProducts
	WHERE Id = @ProductId

	IF(@ProductName = 'Product - 55')
	BEGIN
		UPDATE tblProductSales SET UnitPrice = 55
		WHERE ProductId = @ProductId
	END
	ELSE IF(@ProductName = 'Product - 65')
	BEGIN
		UPDATE tblProductSales SET UnitPrice = 65
		WHERE ProductId = @ProductId
	END
	ELSE IF(@ProductName like 'Product - 100%')
	BEGIN
		UPDATE tblProductSales SET UnitPrice = 1000
		WHERE ProductId = @ProductId
	END

	FETCH NEXT from ProductIdCursor INTO @ProductId
END

-- Release the row set
CLOSE ProductIdCursor
-- Deallocate, the resources associated with the cursor
DEALLOCATE ProductIdCursor


-- The cursor will loop thru each row in tblProductSales table. As there are
-- 600,000 rows, to be processed on a row-by-row basis, it takes around 40 to
-- 45 seconds on my machine.

-- To check if the rows have been correctly updated, please use the following query.

SELECT  Name, UnitPrice
FROM tblProducts
```

```
JOIN tblProductSales ON tblProducts.Id = tblProductSales.ProductId
where (Name='Product - 55' or Name='Product - 65' or Name like 'Product - 100%')


/***************************************************
*64****** REPLACING CURSORS USING JOINS *********64*
****************************************************/

-- The previous example using cursors took around 45 seconds on my machine.
-- In this query we will re write the example using joins.

UPDATE tblProductSales
SET UnitPrice =
	CASE
		WHEN Name = 'Product - 55' THEN 155
		WHEN Name = 'Product - 65' THEN 165
		WHEN Name LIKE 'Product - 100%' THEN 10001
	END
FROM tblProductSales
JOIN tblProducts
ON tblProducts.Id = tblProductSales.ProductId
WHERE Name = 'Product - 55' OR Name = 'Product - 65' OR
Name LIKE 'Product - 100%'


-- When I executed this query, on my machine it took less than a second.
-- Where as the same thing using a cursor took 45 seconds. Just imagine the
-- amount of impact cursors have on performance. Cursors should be used as
-- your last option. Most of the time cursors can be very easily replaced
-- using joins.

-- To check the result of the UPDATE statement, use the following query.
SELECT  Name, UnitPrice
FROM tblProducts
JOIN tblProductSales ON tblProducts.Id = tblProductSales.ProductId
WHERE (Name='Product - 55' or Name='Product - 65' or
Name like 'Product - 100%')


/*************************************************************************
*65****** LIST ALL TABLES IN SQL SERVER DATABASE USING QUERY *********65*
**************************************************************************/

-- We will discuss writing a transact sql query to list all the tables in
-- sql server database. This is a very common sql server interview question.

-- To write a query to do this, there are 3 system views we can use
-- 1. SYSOBJECTS -- Supported in SQL Server version 2000, 2005 and 2008
-- 2. SYS.TABLES -- Supported in SQL Server version 2005 and 2008
-- 3. INFORMATION_SCHEMA.TABLES -- Supported in SQL Server version 2005 and 2008.

-- Gets the list of tables only
SELECT * FROM SYSOBJECTS
WHERE XTYPE='U'
-- Gets the list of tables only
SELECT * FROM  SYS.TABLES
-- Gets the list of tables and views
SELECT * FROM INFORMATION_SCHEMA.TABLES

-- To get the list of different object types (XTYPE) in a database
SELECT DISTINCT XTYPE
FROM SYSOBJECTS

-- Executing the above queery on my sample database returned the following
-- values for XTYPE column from SYSOBJECTS
IT - Internal table
P - Stored procedure
PK - PRIMARY KEY constraint
S - System table
SQ - Service queue
U - User table
V - View

-- Please check the following MSDN link for all possible XTYPE column values
-- and what they represent.
-- http://msdn.microsoft.com/en-us/library/ms177596.aspx


/*************************************************************
*66****** WRITING RE RUNNABLE SQL SERVER SCRIPTS *********66*
**************************************************************/

-- What is a re-runnable sql script?
-- A re-runnable script is a script, that, when run more than, once will not
-- throw errors.

-- Let's understand writing re-runnable sql scripts with an example.
-- To create a table tblEmployee in Sample database, we will write the
-- following CREATE TABLE sql script.

USE [Sample]
CREATE TABLE tblEmployee
(
	ID int IDENTITY PRIMARY KEY,
	Name NVARCHAR(100),
	Gender NVARCHAR(10),
	DateOfBirth DATETIME
)
```

```
-- When you run this script once, the table tblEmployee gets created without
-- any errors. If you run the script again, you will get an error - There
-- is already an object named 'tblEmployee' in the database.

-- To make this script re-runnable
-- 1. Check for the existence of the table
-- 2. Create the table if it does not exist
-- 3. Else print a message stating, the table already exists

Use [Sample]
IF NOT EXISTS (
        SELECT * FROM information_schema.tables
        WHERE table_name = 'tblEmployee'
        )
BEGIN
        CREATE TABLE tblEmployee
        (
                ID int IDENTITY PRIMARY KEY,
                Name NVARCHAR(100),
                Gender NVARCHAR(10),
                DateOfBirth DATETIME
        )
        PRINT 'Table tblEmployee successfully created'
END
ELSE
BEGIN
        PRINT 'Table tblEmployee already exists'
END

-- The above script is re-runnable, and can be run any number of times.
-- If the table is not already created, the script will create the table,
-- else you will get a message stating - The table already exists. You will
-- never get a sql script error.

-- Sql server built-in function OBJECT_ID(), can also be used to check for
-- the existence of the table
IF OBJECT_ID('tblEmployee') IS NULL
BEGIN
  -- Create Table Script
  PRINT 'Table tblEmployee created'
END
ELSE
BEGIN
  PRINT 'Table tblEmployee already exists'
END

-- Depending on what we are trying to achieve, sometime we may need to drop
-- (if the table already exists) and re-create it. The sql script below, does
-- exactly the same thing.

Use [Sample]
IF OBJECT_ID('tblEmployee') IS NOT NULL
BEGIN
        DROP TABLE tblEmployee
END
CREATE TABLE tblEmployee
(
        ID int IDENTITY PRIMARY KEY,
        Name NVARCHAR(100),
        Gender NVARCHAR(10),
        DateOfBirth DATETIME
)

-- Let's look at another example. The following sql script adds column
-- "EmailAddress" to table tblEmployee. This script is not re-runnable
¨-- because, if the column exists we get a script error.
Use [Sample]
ALTER TABLE tblEmployee
ADD EmailAddress NVARCHAR(50)

-- To make this script re-runnable, check for the column existence
Use [Sample]
IF NOT EXISTS (
        SELECT * FROM INFORMATION_SCHEMA.COLUMNS
        WHERE COLUMN_NAME='EmailAddress' AND TABLE_NAME = 'tblEmployee'
        AND TABLE_SCHEMA='dbo'
        )
BEGIN
        ALTER TABLE tblEmployee
        ADD EmailAddress NVARCHAR(50)
END
ELSE
BEGIN
        PRINT 'Column EmailAddress already exists'
END

--- Col_length() function can also be used to check for the existence of
-- a column
IF col_length('tblEmployee','EmailAddress') IS NOT NULL
BEGIN
        PRINT 'Column already exists'
END
ELSE
BEGIN
        PRINT 'Column does not exist'
END
```

```sql
/*******************************************************************
*67****** ALTER TABLE COLUMNS WITHOUT DROPPING DB ********67*
********************************************************************/

-- We will be using table tblEmployee for this demo. Use the sql script below,
-- to create and populate this table with some sample data.

CREATE TABLE tblEmployee
(
	ID INT PRIMARY KEY IDENTITY,
	Name NVARCHAR(50),
	Gender NVARCHAR(50),
	Salary NVARCHAR(50)
)


INSERT INTO tblEmployee VALUES('Sara Nani','Female','4500')
INSERT INTO tblEmployee VALUES('James Histo','Male','5300')
INSERT INTO tblEmployee VALUES('Mary Jane','Female','6200')
INSERT INTO tblEmployee VALUES('Paul Sensit','Male','4200')
INSERT INTO tblEmployee VALUES('Mike Jen','Male','5500')

-- The requirement is to group the salaries by gender. The output should
--be as shown below.
-- Total salary of employees grouped by gender

-- To achieve this we would write a sql query using GROUP BY as shown below.

SELECT Gender, SUM(Salary) AS Total
FROM tblEmployee
GROUP BY Gender

-- When you execute this query, we will get an error - Operand data type
-- nvarchar is invalid for sum operator. This is because, when we created
-- tblEmployee table, the "Salary" column was created using nvarchar datatype.
-- SQL server Sum() aggregate function can only be applied on numeric columns.
-- So, let's try to modify "Salary" column to use int datatype. Let's do it
-- using the designer.

1. Right click on "tblEmployee" table in "Object Explorer" window, and select
"Design"
2. Change the datatype from nvarchar(50) to int
3. Save the table

--At this point, you will get an error message - Saving changes is not permitted.
--The changes you have made require the following tables to be dropped and
-- re-created. You have either made changes to a table that can't be re-created
-- or enabled the option Prevent saving changes that require the table to be
-- re-created.

--Alter database table columns without dropping table

-- So, the obvious next question is, how to alter the database table definition
-- without the need to drop, re-create and again populate the table with data?

-- There are 2 options

--Option 1: Use a sql query to alter the column as shown below.
ALTER TABLE tblEmployee
ALTER COLUMN Salary INT

-- Option 2: Disable "Prevent saving changes that require table re-creation"
-- option in sql server 2008
1. Open Microsoft SQL Server Management Studio 2008
2. Click Tools, select Options
3. Expand Designers, and select "Table and Database Designers"
4. On the right hand side window, uncheck, Prevent saving changes that
require table re-creation
5. Click OK




/*********************************************************
*68****** OPTIONAL PARAMETERS IN STORED PROCEDURES *******68*
**********************************************************/

-- Parameters of a sql server stored procedure can be made optional by
--specifying default values.

--We wil be using table tblEmployee for this Demo.
CREATE TABLE tblEmployee
(
        Id INT IDENTITY PRIMARY KEY,
        Name NVARCHAR(50),
        Email NVARCHAR(50),
        Age INT,
        Gender NVARCHAR(50),
        HireDate DATE,
)


INSERT INTO tblEmployee VALUES
('Sara Nan','Sara.Nan@test.com',35,'Female','1999-04-04')
INSERT INTO tblEmployee VALUES
('James Histo','James.Histo@test.com',33,'Male','2008-07-13')
INSERT INTO tblEmployee VALUES
```

```
('Mary Jane','Mary.Jane@test.com',28,'Female','2005-11-11')
INSERT INTO tblEmployee VALUES
('Paul Sensit','Paul.Sensit@test.com',29,'Male','2007-10-23')

-- Name, Email, Age and Gender parameters of spSearchEmployees stored procedure
-- are optional. Notice that, we have set defaults for all the parameters, and
-- in the "WHERE" clause we are checking if the respective parameter IS NULL.

CREATE PROC spSearchEmployees
@Name NARCHAR(50) = NULL,
@Email NVARCHAR(50) = NULL,
@Age INT = NULL,
@Gender NVARCHAR(50) = NULL
AS
BEGIN
        SELECT * FROM tblEmployee
        WHERE (Name = @Name OR @Name IS NULL) AND
        (Email = @Email OR @Email IS NULL) AND
        (Age = @Age OR @Age IS NULL) AND
        (Gender = @Gender OR @Gender IS NULL)
END

--Testing the stored procedure
1. Execute spSearchEmployees
-- This command will return all the rows

2. Execute spSearchEmployees @Gender = 'Male'
-- Retruns only Male employees

3. Execute spSearchEmployees @Gender = 'Male', @Age = 29
-- Retruns Male employees whose age is 29

-- This stored procedure can be used by a search page that looks as shown below.
-- sql server stored procedure optional parameters
```

WebForm1.aspx:

```
<table style="font-family:Arial; border:1px solid black">
    <tr>
        <td colspan="4" style="border-bottom: 1px solid black">
            <b>Search Employees</b>
        </td>
    </tr>
    <tr>
        <td>
            <b>Name</b>
        </td>
        <td>
            <asp:TextBox ID="txtName" runat="server"></asp:TextBox>
        </td>
        <td>
            <b>Email</b>
        </td>
        <td>
            <asp:TextBox ID="txtEmail" runat="server"></asp:TextBox>
        </td>
    </tr>
    <tr>
        <td>
            <b>Age</b>
        </td>
        <td>
            <asp:TextBox ID="txtAge" runat="server"></asp:TextBox>
        </td>
        <td>
            <b>Gender</b>
        </td>
        <td>
            <asp:DropDownList ID="ddlGender" runat="server">
                <asp:ListItem Text="Any Gender" Value="-1"></asp:ListItem>
                <asp:ListItem Text="Male" Value="Male"></asp:ListItem>
                <asp:ListItem Text="Female" Value="Female"></asp:ListItem>
            </asp:DropDownList>
        </td>
    </tr>
    <tr>
        <td colspan="4">
            <asp:Button ID="btnSerach" runat="server" Text="Search"
                onclick="btnSerach_Click" />
        </td>
    </tr>
    <tr>
        <td colspan="4">
            <asp:GridView ID="gvEmployees" runat="server">
            </asp:GridView>
        </td>
    </tr>
</table>
```

WebForm1.aspx.cs:

```
public partial class WebForm1 : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            GetData();
        }
    }
    protected void btnSerach_Click(object sender, EventArgs e)
    {
        GetData();
    }

    private void GetData()
    {
        string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
        using (SqlConnection con = new SqlConnection(cs))
        {
            SqlCommand cmd = new SqlCommand("spSearchEmployees", con);
            cmd.CommandType = CommandType.StoredProcedure;

            AttachParameter(cmd, "@Name", txtName);
            AttachParameter(cmd, "@Email", txtEmail);
            AttachParameter(cmd, "@Age", txtAge);
            AttachParameter(cmd, "@Gender", ddlGender);

            con.Open();
            gvEmployees.DataSource = cmd.ExecuteReader();
            gvEmployees.DataBind();
        }
    }

    private void AttachParameter(SqlCommand command, string parameterName, Control control)
    {
        if (control is TextBox && ((TextBox)control).Text != string.Empty)
        {
            SqlParameter parameter = new SqlParameter(parameterName, ((TextBox)control).Text);
            command.Parameters.Add(parameter);
        }
        else if (control is DropDownList && ((DropDownList)control).SelectedValue != "-1")
        {
            SqlParameter parameter = new SqlParameter parameterName,
((DropDownList)control).SelectedValue);
            command.Parameters.Add(parameter);
        }
    }
}
```

-- Make sure you have the following using statements in your code-behind page

```
using System;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Data;
using System.Data.SqlClient;
using System.Configuration;
```

```
/*************************
*69****** MERGE *******69*
**************************/

-- What is the use of MERGE statement in SQL Server
-- Merge statement introduced in SQL Server 2008 allows us to perform Inserts,
-- Updates and Deletes in one statement. This means we no longer have to use
--multiple statements for performing Insert, Update and Delete.

--With merge statement we require 2 tables
--1. Source Table - Contains the changes that needs to be applied to the target
--table
--2. Target Table - The table that require changes (Inserts, Updates and
-- Deletes)

--The merge statement joins the target table to the source table by using a
--common column in both the tables. Based on how the rows match up as a result
--of the join, we can then perform insert, update, and delete on the target
--table.

--Merge statement syntax
MERGE [TARGET] AS T
USING [SOURCE] AS S
   ON [JOIN_CONDITIONS]
 WHEN MATCHED THEN
      [UPDATE STATEMENT]
 WHEN NOT MATCHED BY TARGET THEN
      [INSERT STATEMENT]
 WHEN NOT MATCHED BY SOURCE THEN
      [DELETE STATEMENT]

-- Example 1 : In the example below, INSERT, UPDATE and DELETE are all performed
-- in one statement
-- 1. When matching rows are found, StudentTarget table is UPDATED
--(i.e WHEN MATCHED)
-- 2. When the rows are present in StudentSource table but not in
-- StudentTarget table those rows are INSERTED into StudentTarget table
-- (i.e WHEN NOT MATCHED BY TARGET)
-- 3. When the rows are present in StudentTarget table but not in StudentSource
-- table those rows are DELETED from StudentTarget table (i.e WHEN NOT MATCHED
-- BY SOURCE)

----------------------------------
-- merge statement in sql server --
----------------------------------

CREATE TABLE StudentSource
```

```
(
    ID INT PRIMARY KEY,
    Name NVARCHAR(20)
)

GO

INSERT INTO StudentSource VALUES (1, 'Mike')
INSERT INTO StudentSource VALUES (2, 'Sara')

GO

CREATE TABLE StudentTarget
(
    ID INT PRIMARY KEY,
    Name NVARCHAR(20)
)

GO

INSERT INTO StudentTarget VALUES (1, 'Mike M')
INSERT INTO StudentTarget VALUES (3, 'John')

GO

MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;

-- Please Note : Merge statement should end with a semicolon, otherwise you
-- would get an error stating - A MERGE statement must be terminated by a
-- semi-colon (;)

-- In real time we mostly perform INSERTS and UPDATES. The rows that are
-- present in target table but not in source table are usually not deleted
-- from the target table.

-- Example 2 : In the example below, only INSERT and UPDATE is performed.
-- We are not deleting the rows that are present in the target table but not
-- in the source table.

------------------------------
-- merge in sql server 2008 --
```

```
------------------------------

TRUNCATE TABLE StudentSource
TRUNCATE TABLE StudentTarget
GO

INSERT INTO StudentSource VALUES (1, 'Mike')
INSERT INTO StudentSource VALUES (2, 'Sara')
GO

INSERT INTO StudentTarget VALUES (1, 'Mike M')
INSERT INTO StudentTarget VALUES (3, 'John')
GO

MERGE StudentTarget AS T
USING StudentSource AS S
ON T.ID = S.ID
WHEN MATCHED THEN
    UPDATE SET T.NAME = S.NAME
WHEN NOT MATCHED BY TARGET THEN
    INSERT (ID, NAME) VALUES(S.ID, S.NAME);

/*********************************************************************
*70************* CONCURRENT TRANSACTIONS: INTRO *******************70*
**********************************************************************/

-- What is a transaction
-- A transaction is a group of commands that change the data stored in a database. A transaction,
-- is treated as a single unit of work. A transaction ensures that, either all of the commands
-- succeed, or none of them. If one of the commands in the transaction fails, all of the commands
-- fail, and any data that was modified in the database is rolled back. In this way, transactions
-- maintain the integrity of data in a database.

-- sql server concurrent transactions

-- Example : The following transaction ensures that both the UPDATE statements succeed or both of
-- them fail if there is a problem with one UPDATE statement.

-- Transfer $100 from Mark to Mary Account
BEGIN TRY
   BEGIN TRANSACTION
       UPDATE Accounts SET Balance = Balance - 100 WHERE Id = 1
       UPDATE Accounts SET Balance = Balance + 100 WHERE Id = 2
   COMMIT TRANSACTION
   PRINT 'Transaction Committed'
END TRY
BEGIN CATCH
   ROLLBACK TRANSACTION
   PRINT 'Transaction Rolled back'
```

```
END CATCH

-- Databases are powerful systems and are potentially used by many users or applications at the same
-- time. Allowing concurrent transactions is essential for performance but may introduce concurrency
-- issues when two or more transactions are working with the same data at the same time.

-- Some of the common concurrency problems
-- Dirty Reads
-- Lost Updates
-- Nonrepeatable Reads
-- Phantom Reads

-- We will discuss what these problems are in detail with examples in our upcomning videos

-- One way to solve all these concurrency problems is by allowing only one user to execute,
-- only one transaction at any point in time. Imagine what could happen if you have a large
-- database with several users who want to execute several transactions. All the transactions
-- get queued and they may have to wait a long time before they could get a chance to execute
-- their transactions. So you are getting poor performance and the whole purpose of having a
-- powerful database system is defeated if you serialize access this way.

-- At this point you might be thinking, for best performance let us allow all transactions to
-- execute concurrently. The problem with this approach is that it may cause all sorts of
-- concurrency problems (i.e Dirty Reads, Lost Updates, Nonrepeatable Reads, Phantom Reads)
-- if two or more transactions work with the same data at the same time.

-- SQL Server provides different transaction isolation levels, to balance concurrency problems
-- and performance depending on our application needs.
-- Read Uncommitted
-- Read Committed
-- Repeatable Read
-- Snapshot
-- Serializable

-- The isolation level that you choose for your transaction, defines the degree to which one transaction
-- must be isolated from resource or data modifications made by other transactions. Depending on the
-- isolation level you have chosen you get varying degrees of performance and concurrency problems.
-- The table here has the list of isoltaion levels along with concurrency side effects.

-- Isolation Level          Dirty Reads         Lost Update         Nonrepeatable Reads
        Phantom Reads
-- Read Uncommitted         Yes                       Yes                       Yes
                            Yes
-- Read Committed           No                        Yes                       Yes
                            Yes
-- Repeatable Read    No                    No                  No
            Yes
-- Snapshot                       No                  No                        No
                            No
```

-- Serializable No No No No

-- If you choose the lowest isolation level (i.e Read Uncommitted), it increases the number of concurrent
-- transactions that can be executed at the same time, but the down side is you have all sorts of concurrency issues.
-- On the other hand if you choose the highest isolation level (i.e Serializable), you will have no concurrency side
-- effects, but the downside is that, this will reduce the number of concurrent transactions that can be executed
-- at the same time if those transactions work with same data.

/
************************************************************************************
*********
*71****** CONCURRENT TRANSACTION PROBLEMS: DIRTY READ WITH READ UNCOMMITTED  *************71*
************************************************************************************
**********/

-- A dirty read happens when one transaction is permitted to read data that has been modified by another transaction
-- that has not yet been committed. In most cases this would not cause a problem. However, if the first transaction
-- is rolled back after the second reads the data, the second transaction has dirty data that does not exist anymore.

-- SQL script to create table tblInventory

```
Create table tblInventory
(
   Id int identity primary key,
   Product nvarchar(100),
   ItemsInStock int
)
Go

Insert into tblInventory values ('iPhone', 10)
```

-- Table tblInventory dirty read problem in concurrency control

-- Dirty Read Example : In the example below, Transaction 1, updates the value of ItemsInStock to 9. Then it starts to

```
-- bill the customer. While Transaction 1 is still in progress, Transaction 2 starts and reads ItemsInStock value
-- which is 9 at the moment. At this point, Transaction 1 fails because of insufficient funds and is rolled back.
-- The ItemsInStock is reverted to the original value of 10, but Transaction 2 is working with a different value (i.e 10).

-- sql server dirty read example

-- Transaction 1 :
Begin Tran
Update tblInventory set ItemsInStock = 9 where Id=1

-- Billing the customer
Waitfor Delay '00:00:15'
-- Insufficient Funds. Rollback transaction

Rollback Transaction

-- Transaction 2 :
Set Transaction Isolation Level Read Uncommitted
Select * from tblInventory where Id=1

-- Read Uncommitted transaction isolation level is the only isolation level that has dirty read side effect.
-- This is the least restrictive of all the isolation levels. When this transaction isolation level is set, it is
-- possible to read uncommitted or dirty data. Another option to read dirty data is by using NOLOCK table hint.
-- Where the transaction isolation level is the default (Read Committed)
-- The query below is equivalent to the query in Transaction 2.

Set Transaction Isolation Level Read Committed -- this line to restore defaults we modified in Transacion 2 example above.
Select * from tblInventory (NOLOCK) where Id=1
```