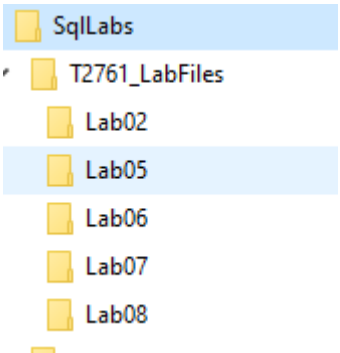
# Lab Manual

## T2761, Querying Data with Transact-SQL - Continuation Course

# Lab environment overview

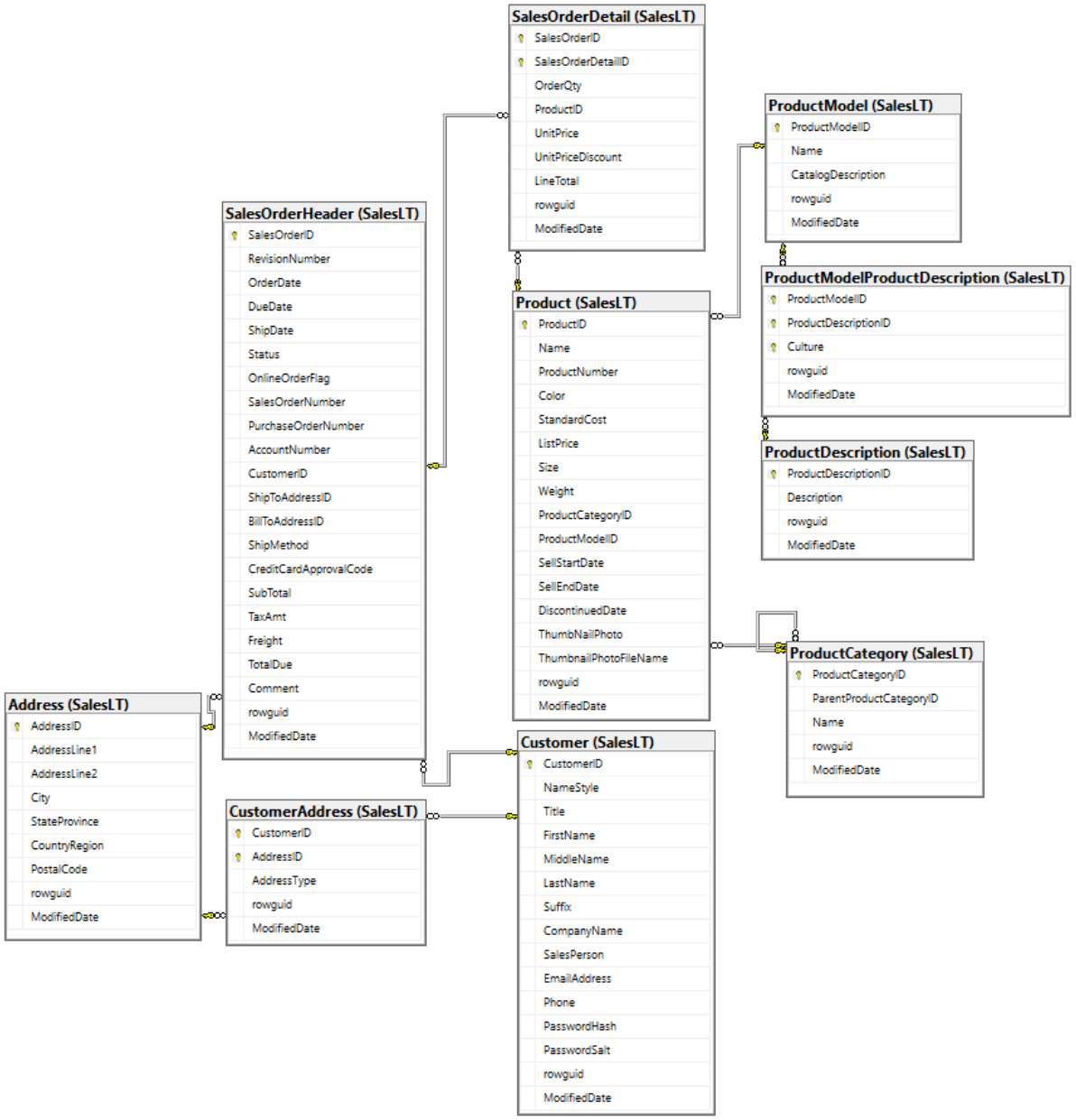
- Your machine name is ***NORTH***.
- If not already done for you, log in to Windows using
  - ***Student***
  - ***myS3cret***
- Copy the lab files to your virtual machine. You find them here: https://karaszi.com/training
  - Use a web browser in the above virtual machine
  - If Virsoft is used: to paste text from the host into the virtual machine, position the cursor, right-click the notepad at the top left and select "paste…"
- Extract the files, so you in the root of your C: drive end up with a folder structure looking like below




- Use ***SQL Server Management Studio (SSMS) or Azure Data Studio (ADS)*** to do the labs, whichever you prefer.
- Login to the SQL Server named “North” using Windows authentication.
  - (There is a default SQL Server instance on the machine and there is a Windows login in your SQL Server for your Windows account, which is a sysadmin.)
- You will be using the database named **AdventureworksLT** and the tables you will use are in the SalesLT schema, unless other is specified.

- You can always revert/reset your databases to "default".
  - There are three bat files in the C:\SqlLabs folder that does this (RESTORE), one for each database.
    - You might need to download these files first, from https://files.karaszi.com/rot/courses/RestoreLabDatabases.zip
  - Note: **Run as Administrator**
- The lab answers are *not* designed to be used independently. Use the lab instructions, and check the answers when you get stuck, etc.

# Diagram of the AdventureworksLT database



# Lab 1: Table expressions

## Ex 1. Derived table

1. Write a query that uses the SalesLT.Customer table and returns the following columns:
    a. CustomerID
    b. FirstName
    c. LastName
    d. LettersFirstName (use "LEN(FirstName)" to calculate this)
2. Now use above query as a derived table and use a WHERE clause in the outer query to return only the Customers where FirstName has more than 10 letters.
    a. *This is a very simple example, and you probably wouldn't use a derived table for such a simple example. The point is to get the syntax elements and concept. There will be other exercises with similar simplifications.*

## Ex 2. Common Table Expression

1. Write a query that uses the SalesLT.Product table. Group over the Color column and average over the ListPrice and the Weight columns. The query should return the following columns:
    a. Color
    b. AvgListPrice
    c. AvgWeight
2. Modify above query so you get 'None' instead of NULL for products that doesn't have a color.
3. Modify the query so the result is ordered by AvgListPrice
4. Use above query as a Common Table Expression and in the outer query limit so you only see the rows where AvgWeight > 1000. (You can also do this without a subquery, using the HAVING clause, but we don't want to over-complicate our examples.)
5. You might have ran into a problem above. Go over the limitations for Table Expressions and work around that problem.

## Ex 3. View

1. Create a view based on the query in step 3 in Ex 2 above. Use the name "vProds" for the view. Try to have the ORDER BY in the view definition.
2. Try to workaround the ORDER BY "limitation" by using TOP(100) PERCENT in the view definition. Were you able to create the view?
3. Now query the view. Were the rows sorted by AvgListPrice?
    - Don't try to outsmart SQL Server… 😊
4. Use CREATE OR ALTER to modify the view, remove the ORDER BY.
5. Query above view and sort the result based on AvgListPrice.

# Lab 1 answer suggestions

## Ex 1. Derived table

```
--Step 1
SELECT
 c.CustomerID
,c.FirstName
,c.LastName
,LEN(c.FirstName) AS LettersFirstName
FROM SalesLT.Customer AS c


--Step 2
SELECT * FROM
(      SELECT
       c.CustomerID
       ,c.FirstName
       ,c.LastName
       ,LEN(c.FirstName) AS LettersFirstName
       FROM SalesLT.Customer AS c
) AS t
WHERE t.LettersFirstName > 10
```

## Ex 2. Common Table Expression

```
--Step 1
SELECT
 Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color

--Step 2
SELECT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color

--Step 3
SELECT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color
ORDER BY AvgListPrice

--Step 4
--Try to use above in a CTE. This doesn't compile due to ORDER BY in CTE
;WITH grpProds AS
(
SELECT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color
ORDER BY AvgListPrice
)
SELECT *
FROM grpProds
WHERE AvgWeight > 1000

--Step 5
--Use ORDER BY in the outer query instead
;WITH grpProds AS
(
SELECT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color
)
SELECT *
FROM grpProds
WHERE AvgWeight > 1000
ORDER BY AvgListPrice
```

## Ex 3. View

```
--Step 1
--Try to use ORDER BY in the view definition
CREATE VIEW vProds
AS
SELECT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color
ORDER BY AvgListPrice
GO

--Step 2
--Use ORDER BY in the view definition, add "the TOP trick"
CREATE VIEW vProds
AS
SELECT TOP(100) PERCENT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color
ORDER BY AvgListPrice
GO

--Step 3
--Query the view. The result isn't sorted,
--since a view behaves like a table and tables are not sorted.
SELECT * FROM vProds
GO

--Step 4
--Remove the useless ORDER BY in the view definition.
CREATE OR ALTER VIEW vProds
AS
SELECT TOP(100) PERCENT
 ISNULL(Color, 'None') AS Color
,AVG(ListPrice) AS AvgListPrice
,AVG(Weight) AS AvgWeight
FROM SalesLT.Product
GROUP BY Color
GO


--Step 5
--Query the view
SELECT *
FROM vProds
ORDER BY AvgListPrice
GO
```

# Lab 2: Set operators

**NOTE**: Before doing the lab, run below bat file (**double-click** or r**ight-click, Open**).

C:\SqlLabs\T2761\Lab02**Setup.bat**

## Ex 1. UNION, INTERSECT and EXCEPT

You have a design with separate tables for products, one table for each year when the product was introduced (as per the SellStartDate). The tables are named

- Product2002
- Product2005
- Product2006
- Product2007

1. Perform SELECT * from each table so you familiar yourself with the data. Note the SellStartDate column.
2. Combine all products in one result. Should we remove duplicates?
3. Return all combinations of color and size that we have in ***both*** the 2006 and 2007 tables.
4. Return the combination of color and size that were have in the 2006 table ***but not*** in the 2007 table.

## Ex 2. Using APPLY

You have a table function that returns the 3 most recent orders containing a certain product. You pass on the product ID as a parameter to this function. The name of the function is dbo.RecentordersIncludingProduct.

1. Select from this function, and pass in 715 as product id. Return all columns and all rows.
2. Now you want to select from the product table, and for each product "join" to or "call" above function and pass in the ProductID column from the Product table into the function. Sort the data by product Name, OrderDate DESC. Return the following columns:
    a. Name
    b. Listprice
    c. SalesOrderID
    d. OrderDate
    e. SubTotal
3. If time permits, script out the source code for the table function and modify the query in above Step 2 so it uses a derived table instead of the table function.

# Lab 2 answer suggestions

## Ex 1. UNION, INTERSECT and EXCEPT

```
--Step 1
SELECT * FROM Product2002
SELECT * FROM Product2005
SELECT * FROM Product2006
SELECT * FROM Product2007

--Step 2
SELECT * FROM Product2002
UNION ALL
SELECT * FROM Product2005
UNION ALL
SELECT * FROM Product2006
UNION ALL
SELECT * FROM Product2007
```

Should we remove duplicates?

1. From a performance perspective: no. The cost of sorting the data to identify and removing duplicates is neglectable with so few rows as we have in these tables.
2. From a data perspective: it depends. Our logic is very simple. But imagine if we don't want to return the SellStartDate column? I.e., we are interested in produces regardless of whenever it was sold. For our data, a product might have been sold 2005, but not 2006, and then again sold in 2007. Should that be included?
3. Carefully consider the semantics of what we want to do when deciding these things.

```
--Step 3
SELECT Color, Size FROM Product2006
INTERSECT
SELECT Color, Size FROM Product2007

--Step 4
SELECT Color, Size FROM Product2006
EXCEPT
SELECT Color, Size FROM Product2007
```

## Ex 2. Using APPLY

```
--Step 1
SELECT * FROM dbo.RecentordersIncludingProduct(715)

--Step 2
SELECT p.Name, p.ListPrice, o.SalesOrderID, o.OrderDate, o.SubTotal
FROM SalesLT.Product AS p
CROSS APPLY dbo.RecentordersIncludingProduct(p.ProductID) AS o
ORDER BY name, OrderDate DESC

--Step 3
SELECT p.Name, p.ListPrice, o.SalesOrderID, o.OrderDate, o.SubTotal
FROM SalesLT.Product AS p
CROSS APPLY (
  SELECT TOP(3) h.SalesOrderID, h.OrderDate, h.CustomerID,
                h.SubTotal, h.TaxAmt, h.Freight
  FROM SalesLT.SalesOrderHeader AS h
  WHERE
  EXISTS(
       SELECT *
       FROM SalesLT.SalesOrderDetail AS d
       WHERE d.SalesOrderID = h.SalesOrderID AND d.ProductID = p.ProductID
       )
  ORDER BY OrderDate DESC
  ) AS o
ORDER BY name, OrderDate DESC
```

# Lab 3: Windowing

## Ex 1. Using ranking functions

1. Return all columns and all rows from the Products table. Sort the result over the ProductID column. Also generate a unique incrementing value in the result as a column named Rank_, starting with 1 (i.e.: 1, 2, 3, 4…).
2. Return below columns from the Products table. Sort the result over the Name column. Also return a calculated column named PriceRank, which ranks the products based on ListPrice (1 would be the lowest price). Two products having the same price should have the same rank, and if there are for instance two rows ranked as 1, then the following should rank 3.
    - Name
    - ProductNumber
    - Color
    - ListPrice
    - Size
    - Weight
    - PriceRank
3. If time permits: Use above query as starting point. Join to the ProductCategory and also return the column Name from that table as CategoryName in the result. Add one more ranking column to rank the price within each product category, and name that column PriceRankCategory

## Ex 2. Generate running totals

1. Use the ***Adventureworks*** database for this exercise.
2. Create a query to return below columns. It should join the Person.Person and Sales.SalesOrderHeader tables (on Person. BusinessEntityID = SalesOrderHeader `SalesPersonID`). Sort the result on the OrderDate column.
    a. FirstName
    b. LastName
    c. SalesOrderID
    d. OrderDate
    e. SubTotal
3. Use above query and add a running total over the SubTotal column. Name that column as RunningTotal. The running total should be based/sorted on the OrderDate column. We can have several order on the same OrderDate, and they should have the same value in the RunningTotal column.
4. If time permits. Tweak above query to do the running total within each year (re-start for each new year).

# Lab 3 answer suggestions

## Ex 1. Using ranking functions

```
--Step 1
SELECT  ROW_NUMBER() OVER(ORDER BY ProductID) AS Rank_, *
FROM SalesLT.Product
ORDER BY ProductID

--Step 2
SELECT
 p.Name
,p.ProductNumber
,p.Color
,p.ListPrice
,p.Size
,p.Weight
,RANK() OVER(ORDER BY ListPrice) AS PriceRank
FROM SalesLT.Product AS p
ORDER BY Name

--Step 3
SELECT
 p.Name
,p.ProductNumber
,p.Color
,p.ListPrice
,p.Size
,p.Weight
,c.Name AS CategoryName
,RANK() OVER(ORDER BY ListPrice) AS PriceRank
,RANK() OVER(PARTITION BY c.name ORDER BY ListPrice) AS PriceRankCategory
FROM SalesLT.Product AS p
 INNER JOIN SalesLT.ProductCategory AS c ON p.ProductCategoryID = c.ProductCategoryID
ORDER BY Name
```

## Ex 2. Generate running totals

```
--Step 2
SELECT p.FirstName, p.LastName, h.SalesOrderID, h.OrderDate, h.SubTotal
FROM Sales.SalesOrderHeader AS h
 INNER JOIN Person.Person AS p ON p.BusinessEntityID = h.SalesPersonID
ORDER BY OrderDate

--Step 3
SELECT p.FirstName, p.LastName, h.SalesOrderID, h.OrderDate
,h.SubTotal
,SUM(h.SubTotal) OVER(ORDER BY h.OrderDate
  RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal
FROM Sales.SalesOrderHeader AS h
 INNER JOIN Person.Person AS p ON p.BusinessEntityID = h.SalesPersonID
ORDER BY OrderDate

--Step 4
SELECT p.FirstName, p.LastName, h.SalesOrderID, h.OrderDate
,h.SubTotal
,SUM(h.SubTotal) OVER(PARTITION BY DATEPART(year, OrderDate)
  ORDER BY h.OrderDate
  RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS RunningTotal
FROM Sales.SalesOrderHeader AS h
 INNER JOIN Person.Person AS p ON p.BusinessEntityID = h.SalesPersonID
ORDER BY OrderDate
```

# Lab 4: Advanced grouping and pivoting data

## Ex 1. Performing independent aggregations

1. Write a query that joins the Products, SalesOrderHeader and SalesOrderDetails tables. Return all columns and all rows.
2. Modify above query to group over the Product.Size column and SUM over the SalesOrderHeader.SubTotal column. Also COUNT number of rows in each group. The query should return the following columns:
    a. Size
    b. CountRows
    c. SaleSum
3. Do a similar query to Step 2 above, but instead group over the Color column. The query should return the following columns:
    a. Color
    b. CountRows
    c. SaleSum
4. Now return the combination of above two queries. Use GROUPING SETS, so you get independent groupings. The query should return the following columns:
    a. Size
    b. Color
    c. CountRows
    d. SaleSum
5. Advanced, if time permits: Modify the above query so that for the rows where grouping doesn't apply, the columns for Size and Color should be the string '[ALL]' instead of NULL. Hint: use the GOUPING aggregate function in a CASE expression for this. Also sort so that the aggregates for the color comes first (alphabetically) and then the aggregates for size (alphabetically).

## Ex 2. Pivot the result from a query

You will use the Products table and average the ListPrice for each Color. Each color will have a separate column (one column for Red, one for Blue, etc). You will have one row for each sets of aggregates. See below:

| SellStartDate | Black | Blue | Grey | Multi | Red |
|---|---|---|---|---|---|
| 2002-06-01 00:00:00.000 | 1431,50 | NULL | NULL | NULL | 1431 |
| 2005-07-01 00:00:00.000 | 1227,464 | 34,99 | NULL | 41,79 | 1274 |
| 2006-07-01 00:00:00.000 | 692,2771 | NULL | 125,00 | 89,99 | 2443 |
| 2007-07-01 00:00:00.000 | 357,79 | 959,2268 | NULL | NULL | NULL |

1. Create a query that returns the Color, ListPrice and SellStartDate columns. This will be used as a subquery for your PIVOT in the subsequent steps.
2. Create a query to give you what colors exists in the Products table.
3. Use PIVOT to get above result. Use the query from Step 1 as base for the PIVOT (either as a derived table or a CTE). Ignore the NULL colors.

# Lab 4: answer suggestions

## Ex 1. Performing independent aggregations

```
--Step 1
--Join the three tables
SELECT *
FROM SalesLT.SalesOrderHeader AS h
INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = d.SalesOrderID
INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID

--Step 2
--Sum sales per Size
SELECT p.Size, COUNT(*) AS CountRows, SUM(h.SubTotal) AS SaleSum
FROM SalesLT.SalesOrderHeader AS h
INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = d.SalesOrderID
INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID
GROUP BY p.Size

--Step 3
--Sum sales per Color
SELECT p.Color, COUNT(*) AS CountRows, SUM(h.SubTotal) AS SaleSum
FROM SalesLT.SalesOrderHeader AS h
INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = d.SalesOrderID
INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID
GROUP BY p.Color

--Step 4
--Sum sales per size and Color independently
SELECT p.Size, p.Color, COUNT(*) AS CountRows, SUM(h.SubTotal) AS SaleSum
FROM SalesLT.SalesOrderHeader AS h
INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = d.SalesOrderID
INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID
GROUP BY GROUPING SETS((p.Size), (p.Color))


--Step 5
--Replace NULL where grouping doesn't apply with '[ALL]'
--Also sort color aggregates first and then size aggregates
SELECT
 CASE WHEN GROUPING(p.Size) = 0 THEN p.Size ELSE '[ALL]' END AS Size
,CASE WHEN GROUPING(p.Color) = 0 THEN p.Color ELSE '[ALL]' END AS Color
,COUNT(*) AS CountRows
,SUM(h.SubTotal) AS SaleSum
FROM SalesLT.SalesOrderHeader AS h
INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = d.SalesOrderID
INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID
GROUP BY GROUPING SETS((p.Size), (p.Color))
ORDER BY GROUPING(p.Color), Color, GROUPING(p.Size), Size
```

## Ex 2. Pivot the result from a query

```
--Step 1
SELECT p.Color, p.ListPrice, p.SellStartDate
FROM SalesLT.Product AS p

--Step 2
--What colors exists
SELECT DISTINCT p.Color FROM SalesLT.Product AS p


--Step 3
--Perform the pivoting
SELECT *
FROM(
SELECT p.Color, p.ListPrice, p.SellStartDate FROM SalesLT.Product AS p
) AS i
PIVOT (AVG(i.ListPrice) FOR i.Color IN(Black, Blue, Grey, Multi, Red, Silver,
[Silver/Black], White, Yellow)) AS pvt
```

# Lab 5: Using stored procedures

**NOTE**: Before doing the lab, run below bat file (**double-click** or r**ight-click, Open**).

C:\SqlLabs\T2761\Lab05**Setup.bat**

## Ex 1. Executing a stored procedure

1. You have a stored procedure named GetCustomersCountry. Execute it, without passing any parameters to it.
2. Note the error message. Determine what the error is. Look into the source code for the procedure, if needed.
3. Now execute the procedure so it doesn't fail.

## Ex 2. Write your own stored procedure

1. Determine the data type for the ListPrice column for the Product table.
2. Write a stored procedure to list all products with a price which is higher than a certain value (to be passed in as a parameter to the procedure). Make sure that the parameter is of the same data type as the ListPrice column. Return whichever columns you feel like. Name the procedure GetExpensiveProducts.
3. Test your stored procedure.

# Lab 5 answer suggestions

## Ex 1. Executing a stored procedure

```
--Step 1
EXEC GetCustomersCountry
GO

--Step 2
EXEC GetCustomersCountry @City = 'Seattle'
```

## Ex 2. Write your own stored procedure

```
--Step 1
--Determine the data types for the columns
EXEC sp_help 'SalesLT.Product'
GO

--Step 2
CREATE OR ALTER PROC GetExpensiveProducts
@ListPrice money
AS
SELECT p.Name, p.ListPrice, p.Color
FROM SalesLT.Product AS p
WHERE p.ListPrice > @ListPrice
ORDER BY ListPrice
GO

--Step 3
EXEC GetExpensiveProducts @ListPrice = 3000
```

# Lab 6: T-SQL procedural language elements

**NOTE**: Before doing the lab, run below bat file (**double-click** or r**ight-click, Open**).

C:\SqlLabs\T2761\Lab06**Setup.bat**

## Ex 1. Using T-SQL variables

1. You will assign the value from the Name column in the Product table to a variable and then PRINT that variable.
    a. Declare the variable, using the datatype for that column in the table.
    b. Use an in-line variable assignment in a SELECT to your variable from the Name column. Use a WHERE clause for ProductID = 680.
    c. PRINT the contents of the variable.
2. Use the same as in Step 1 above, but instead of using a ProductId, you will in the WHERE clause have Color = 'Red'.
    a. Does this feel right?
    b. Why not?
3. Instead of using in-line variable assignment in Step 2 above, assign the value to the variable using a scalar subquery.
    a. Is this better than above?
    b. Why?

## Ex 2. Using IF

You have a stored procedure named GetExpensiveProducts. This requires a parameter. You want to return something nicer than the SQL Server error message if the caller doesn't pass a parameter.

1. Execute the procedure without passing a value for the parameter. Note the error message.
2. Execute the procedure and pass the value 3000 for the parameter, just to have a successful execution.
3. Modify the procedure:
    a. Define a default value for the parameter, NULL.
    b. In the beginning of the proc, use IF to check if the parameter is NULL.
        i. If so, PRINT some descripting message and exit. (*If this was for real, we would use THROW or RAISERROR, but that is the topic for next module. Feel free to try those instead of PRINT if you have the time and knowledge.*)
        ii. If not, then perform the SELECT statement.
4. Try the procedure again as per step 1 and 2 above.

# Lab 6 answer suggestions

## Ex 1. Using T-SQL variables

```
--Step 1
DECLARE @ProductName nvarchar(50)

SELECT @ProductName = Name
FROM SalesLT.Product AS p
WHERE p.ProductID = 680

PRINT @ProductName
GO

--Step 2
DECLARE @ProductName nvarchar(50)

SELECT @ProductName = Name
FROM SalesLT.Product AS p
WHERE p.Color = 'Red'

PRINT @ProductName
GO
```

*Does this feel right? Why not?*

*No, it hopefully doesn't feel right, because several products have the color Red, but only one of them are assigned to the variable. It could be any of those products, it is not deterministic which one.*

```
--Step 3
DECLARE @ProductName nvarchar(50)

SET @ProductName = (SELECT Name FROM SalesLT.Product AS p WHERE p.Color = 'Red')

PRINT @ProductName
```

*Is this better than above? Why?*

*Yes, it is better. Since we have a logic flaw in our thinking (that only one product can be Red), it is better to get an error than incorrect data. We are now more likely to catch our mistake and correct it.*

## Ex 2. Using IF

```
--Step 1
EXEC GetExpensiveProducts

--Step 2
EXEC GetExpensiveProducts @ListPrice = 3000
GO

--Step 3
CREATE OR ALTER PROC GetExpensiveProducts
@ListPrice money = NULL
AS
IF @ListPrice IS NULL
  BEGIN
    PRINT 'You need to pass the price limit to this procedure'
       RETURN
  END
SELECT p.Name, p.ListPrice, p.Color
FROM SalesLT.Product AS p
WHERE p.ListPrice > @ListPrice
ORDER BY ListPrice
GO
```

# Lab 7: Understanding and handling errors

**NOTE**: Before doing the lab, run below bat file (**double-click** or r**ight-click, Open**).

C:\SqlLabs\T2761\Lab07**Setup.bat**

## Ex 1. Understanding an error message

1. Run below query:
```
SELECT * FROM SalesLT.Product WHERE Size = 44
```
2. Note the error message. Is it understandable?
3. Fix the problem and run the query again.

## Ex 2. Throwing an error

You have a stored procedure named GetExpensiveProducts. This currently does PRINT if no value is passed into the procedure.

1. Change the PRINT to RAISERROR. Use the value 1 for state. For severity, first use 1. Try it. Change the severity to 16 (which is what we normally use when we generate errors).
2. Now try THROW instead of RAISERROR. For error number, use 50001.

## Ex 3. If time permits: catching an error and logging it

You have a procedure named ProductsInOrders. This list products based on some nonsense calculation (the actual calculation is not relevant for the lab purposes).

1. Execute the procedure, passing in Red for the color parameter. This should succeed.
2. Do the same passing in Blue. You will get an error.
3. Your job is to add structured error handling, using TRY/CATCH.
4. In case of error, you will return a more user-friendly error (whatever you feel like) to the user.
5. Optionally, in the CATCH block, call the existing procedure dbo.uspLogError which logs the error to the dbo.ErrorLog table. You will have to investigate dbo.uspLogError regarding how to call it.

# Lab 7 answer suggestions

## Ex 1. Understanding an error message

The problem is that the Size column is a string column and there are size like for instance 'S', 'M' and 'L' in the table. The rules for implicit type conversion are that the string is attempted to be a number. And for instance, 'M' cannot be converted to a number, hence the error. Here is a correct version of the query:

```
--Step 3
SELECT * FROM SalesLT.Product WHERE Size = '44'
```

## Ex 2. Throwing an error

```
--Step 1
--Only the raiserror command shown below, not the full procedure.
RAISERROR('You need to pass the price limit to this procedure', 16, 1)

--Step 2
--Only the throw command shown below, not the full procedure.
;THROW 50001, 'You need to pass the price limit to this procedure', 1
```

## Ex 3. If time permits: catching an error and logging it

```
--Step 1
EXEC ProductsInOrders @Color = 'Red'


--Step 2
EXEC ProductsInOrders @Color = 'Blue'
GO


--Step 4
CREATE OR ALTER PROCEDURE ProductsInOrders
@Color varchar(30)
AS
BEGIN TRY
SET NOCOUNT ON
       SELECT
        p.Name AS ProductName
       ,SUM(h.TotalDue / (d.OrderQty * d.UnitPrice)) AS PartOfTotalOrder
       FROM SalesLT.SalesOrderHeader AS h
       INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = h.SalesOrderID
       INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID
       WHERE p.Color = @Color
       GROUP BY p.Name
END TRY

BEGIN CATCH
       ;THROW 50001, 'There was a problem in the ProductsInOrders procedure.', 1
END CATCH
GO

--Test
EXEC ProductsInOrders @Color = 'Red'
```

```
EXEC ProductsInOrders @Color = 'Blue'
GO


--Step 5
--Add logging using the dbo.uspLogError procedure
CREATE OR ALTER PROCEDURE ProductsInOrders
@Color varchar(30)
AS
BEGIN TRY
SET NOCOUNT ON
       SELECT
        p.Name AS ProductName
       ,SUM(h.TotalDue / (d.OrderQty * d.UnitPrice)) AS PartOfTotalOrder
       FROM SalesLT.SalesOrderHeader AS h
       INNER JOIN SalesLT.SalesOrderDetail AS d ON d.SalesOrderID = h.SalesOrderID
       INNER JOIN SalesLT.Product AS p ON p.ProductID = d.ProductID
       WHERE p.Color = @Color
       GROUP BY p.Name
END TRY

BEGIN CATCH
       EXEC dbo.uspLogError
       ;THROW 50001, 'There was a problem in the ProductsInOrders procedure.', 1
END CATCH
GO

--Test
EXEC ProductsInOrders @Color = 'Red'
EXEC ProductsInOrders @Color = 'Blue'
GO

--Was anything logged?
SELECT * FROM dbo.ErrorLog
```

# Lab 8: Protecting yourself using transactions

**NOTE**: Before doing the lab, run below bat file (**double-click** or r**ight-click, Open**).

C:\SqlLabs\T2761\Lab08**Setup.bat**

## Ex 1. The Oops situation

You will recovery from an Oops situations, thanks to having foresight and transaction-protect your work.

1. Start a transaction
2. Delete all rows in the SalesLT.SalesOrderDetail table
3. Select (all rows and all columns) from the table. You will get 0 rows.
4. Open a new session/window and select (all rows and all columns) from that table. You will be blocked ("hanging").
5. Return to original session/window and rollback the transaction
6. Go to the session/window in Step 4 and note that the rows were returned.

## Ex 2. Adding transaction management to a stored procedure

You have a stored procedure which currently is without any type of error or transaction handling. The name of the procedure is dbo.AddOrder. Your task is to add both error handling and transaction management.

(To keep things simple, the procedure has hard-coded values for the INSERT statements.)

1. Script the source code for the procedure.
2. Modify the procedure to include both error and transaction handling. Optionally use the dbo.uspLogError procedure (that you might have used in the last lab) to log the error.

# Lab 8 answer suggestions

## Ex 1. The Oops situation

```
--Step 1
BEGIN TRAN

--Step 2
DELETE FROM SalesLT.SalesOrderDetail

--Step 3
SELECT * FROM SalesLT.SalesOrderDetail

--Step 4
--Do below from a different session/windows. Should be blocked.
SELECT * FROM SalesLT.SalesOrderDetail

--Step 5
--Do below from the original connection
ROLLBACK TRAN

--Step 6
--Return to the session/windows in Step 4. Data should now have been returned.
```

## Ex 2. Adding transaction management to a stored procedure

```
--Step 2
CREATE OR ALTER PROC dbo.AddOrder
AS
DECLARE @SalesOrderID int
SET NOCOUNT ON

BEGIN TRY
BEGIN TRAN
	INSERT INTO SalesLT.SalesOrderHeader
	(RevisionNumber, OrderDate, DueDate, ShipDate, Status,OnlineOrderFlag
	,PurchaseOrderNumber, AccountNumber, CustomerID, ShipToAddressID
	,BillToAddressID, ShipMethod, CreditCardApprovalCode, SubTotal, TaxAmt, Freight
	,Comment, rowguid, ModifiedDate)
	VALUES
	(2, GETDATE(), DATEADD(DAY, 30, GETDATE()), DATEADD(DAY, 5, GETDATE())
	,5, 0, 'PO5536545166', '10-4020-000277'
	,29638, 989, 989, 'CARGO TRANSPORT 5', NULL
	,2137.231, 170.9785, 53.4308
	,NULL, NEWID(), GETDATE())

	SET @SalesOrderID = SCOPE_IDENTITY()

	INSERT INTO SalesLT.SalesOrderDetail
	(SalesOrderID, OrderQty ,ProductID, UnitPrice, UnitPriceDiscount,
	 rowguid, ModifiedDate)
	VALUES(@SalesOrderID, 5, 982, 461.00, 0 ,NEWID(), GETDATE())

	INSERT INTO SalesLT.SalesOrderDetail
	(SalesOrderID, OrderQty ,ProductID, UnitPrice, UnitPriceDiscount,
	 rowguid, ModifiedDate)
	VALUES(@SalesOrderID, 2, 822, 356.00, 0 ,NEWID(), GETDATE())
COMMIT
END TRY

BEGIN CATCH
	EXEC dbo.uspLogError
	;THROW 50001, 'There was a problem in the AddOrder procedure.', 1

END CATCH

GO
```