Module 1: Table Expressions

Exercise 1: Derived Table – Filter Customers by Name Length

Use the AdventureWorksLT database.

1. 1 Your task is to retrieve the firstname,lastname and modifieddate for the ten most recently modified customers in the table SalesLT.customer. Write a query to retrieve them.

1.2 Let the query be a derived table and use an outer query to sort the customers first by Lastname, then by Firstname

1.3 Give adequate Swedish column aliases in the inner query

1.3 Give adequate Swedish column aliases in the outer query. If the inner and outer queries conflict, the outer aliases will be used

Exercise 2: CTE – Average Price by Color

Use the AdventureWorks database

2. 1 Write a query that retrieves the year of Modifieddate YEAR(Modifieddate) with the alias orderyear and CustomerID from the Sales.SalesOrderHeader table. Present the query as a Common Table Expression with the name CTE_year

2.2 Write a query against the CTE which retrieves the orderyear and number of unique customers in that year with the alias cust_count by grouping on orderyear

Exercise 3: View – Reusable Product Averages

Use the AdventureWorks database

Create a view that uses the tables Sales.SalesTerritory and Sales.SalesorderHeader. The name of the view should be salesterritory

The view should group by territoryname and year of the orderdate and include the name of the territory, the year of the Orderdate YEAR(Orderdate) and Sum of Totaldue cast as an Integer CAST(SUM(TotalDue as INT))

Exercise 4: Inline Table-Valued Function – Customers with Long First Names

Use the AdventureWorksLT database.

Suppose this filtering of customers with long names becomes a recurring requirement. Instead of repeating the same logic, you decide to encapsulate it in an inline table-valued function.

Create an Inline-Table Valued Function that receives an integer and returns CustomerID, Firstname and Lastname from the SalesLT.Customer table for all customers longer than the integer provided. Test with the integer 10. To get the length of a column, use the function LEN(Columnname)

Module 2: Set Operators

Exercise 1: Union, Except, Intersect

Use the AdventureWorks and AdventureWorksDW database.

Your task is to get some information from the tables AdventureWorks.Person.Person and AdventureworksDW.dbo.DimEmployee. Change two of the firstnames in DimEmployee to 'Pelle' and 'Kalle'

1.1

Retrieve all full name, that is Firstname, MiddleName and Lastname from both tables, filtering duplicates

1.2

Retrieve all full name, that is Firstname, MiddleName and Lastname from both tables, NOT filtering duplicates

1.3

Retrieve all full names which exist in both tables

1.4

Retrieve all names which exist in AdventureworksDW.dbo.DimEmployee BUT NOT in AdventureWorks.Person.Person. That would be the added names Pelle and Kalle

Exercise 2: Cross Apply and Outer Apply

2.1

Use the AdventureWorksLT database.

Write an Inline Table-Valued Function that takes the Customerid INT as parameter and returns the latest order (with the highest SalesOrderID) for a customer from the table SalesLT.SalesorderHeader. Return SalesOrderID, CustomerID and TotalDue. Test with CustomerID 29847.

2.2

Select Customerid, Companyname, LastName and Firstname for all customers from the SalesLT.Customer table and combine with all columns in the newly created function to show the last order

2.3

Not all Customers were included in 2.2. Rewrite the query to include all customers, also customers without orders

Module 3: Windowing

Exercise 1: Aggregate Windowing Functions

Use the AdventureWorks database

1.1

Create a view called vw_Territory_orders by joining the tables Sales.Territory and Sales.SalesOrderHeader. Group by Name and Year(Orderdate) and include the sum of Freight cast as an integer. Name the columns Territory, OrderYear and Freight

1.2

Select all columns from the view vw_Territory_orders and include the total freight by using the OVER() function. Let the name of this column be TotalFreight

1.3

Select all columns from the view vw_Territory_orders and include the total freight per territory by using the OVER() function with PARTITION BY Territory. Let the name of this column be FreightPerTerritory. Check if the sums are correct.

1.4

Select all columns from the view vw_Territory_orders and include the total freight per year by using the OVER() function with PARTITION BY OrderYear. Let the name of this column be FreightPerYear. Check if the sums are correct.

1.5

Now, we want to find accumulated sums. Select all columns from the view vw_Territory_orders. Create a new column with an over function that partitions by Territory and orders by OrderYear. Name the column FreightYTD.

1.6

In the OVER clause in 1.5, add
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

The results should be identical

Exercise 2: Ranking Functions

Use the AdventureWorks database

2.1

Use the table Production.Product. Create a result set with all the product names and an additional columns ordering the products in alphabetical order, 1,2, 3 etc. Name the columns ProductName and RowNumber

2.2

Use the table Production.Product. Create a result set with the name and listprice of each product and a new column containing the ranking of the listprice in descending order. Name the new column PriceRank

2.3

The table Production.Product contains 504 rows. In addition to the columns name and Listprice, add a new column in the result set called PriceLevel where 1 is used for the 168 most expensive products, 2 for the next 168 products and 3 for the 168 least expensive products. Use the function NTILE

Exercise 3: Offset Functions

Use the AdventureWorks database

If not already created, create the view vw_territory_orders using this code:

```
CREATE VIEW vw_territory_orders AS
SELECT
        name AS Territory
        ,YEAR(orderdate) AS OrderYear
        ,CAST(SUM(freight) AS INT) AS Freight
FROM
        Sales.SalesTerritory AS st INNER JOIN Sales.SalesOrderHeader AS
soh
        ON st.TerritoryID=soh.TerritoryID
GROUP BY
        name
        ,YEAR(OrderDate)
GO
```

3.1

Select all columns from the view vw_territory_orders and add an additional column called FreightLastYear containing the freight from the previous year. Partition by Territory and order by OrderYear. If there is no previous year, use NULL.

3.2

Add another column to the result set in 3.1 called Difference. The column should contain the increase or decrease in Freight compared to the last year

3.3

We want the difference between the years in percent. Create a Common Table Expression called CTE containing the query in 3.2 and select all columns from CTE.

3.4

Using the Common Table Expression in 3.3, add another column called DiffProc containing the Difference*100.0 /FreightLastYear

Module 4: Pivoting and Grouping Sets

Exercise 1: Grouping Sets

Use the AdventureWorks database

1.1

Use the tables Sales.Territory and Sales.SalesOrderHeader. Group the result set by the name of the Territory and the Year of the OrderDate. Show the sum of Freight using the column alias SumFreight

1.2

Modifying 1.1 using CUBE, include the total sum of Freight and the sum for each territory and each year

1.3

Modify the query above using GROUPING SETS to only display the total for the table, the total per orderyear and the total per territory.

Exercise 2: Pivoting

Use the AdventureWorks database

If not already created, create the view vw_territory_orders using this code:

```
CREATE VIEW vw_territory_orders AS
SELECT
        name AS Territory
        ,YEAR(orderdate) AS OrderYear
        ,CAST(SUM(freight) AS INT) AS Freight
FROM
        Sales.SalesTerritory AS st INNER JOIN Sales.SalesOrderHeader AS
soh
        ON st.TerritoryID=soh.TerritoryID
GROUP BY
        name
        ,YEAR(OrderDate)
GO
```

2.1

We want to create a pivot table with the territories as rows and the OrderYears as columns, showing the Sum of Freight as values in the table.

To find out the orderyears run the query:

```
SELECT DISTINCT OrderYear FROM vw_territory_orders
```

The syntax of pivoting is:

```
SELECT (<Columns To Display>)

FROM (<SELECT columns to include in the query>) AS D

PIVOT(SUM(<Value Column>) FOR <ColumnName IN ([col1],[col2],[col3])) AS pvt
```

In this case:

<Columns To Display> are all, that is *

<columns to include in the query> are also all columns, that is
SELECT * FROM vw_territory_orders

<Value Column> is Freight

<ColumnName IN ([col1],[col2],[col3]> is OrderYear IN [2011],[2012],[2013],[2014]

Create the pivoting using this template:

```
SELECT  Category, [2006],[2007],[2008]
FROM (SELECT  Category, Qty, Orderyear FROM Sales.CategoryQtyYear) AS D
PIVOT(SUM(QTY) FOR orderyear IN ([2006],[2007],[2008])) AS pvt
```

Module 5: Stored Procedures

Exercise 1: Create a stored procedure

Use the AdventureWorks database

1.1

Create a stored procedure called proc_black_products which shows all black products from the table Production.Product

1.2

Execute the procedure proc_black_products

1.3

Create a procedure called proc_products_with_color with the input parameter @color which shows all products with that color

1.4

Execute the procedure proc_products_with_color using the color blue

1.5

Alter the procedure proc_products_with_color so that it only returns products with a SellEndDate that is null

1.6

Execute the procedure proc_products_with_color using the color blue

Exercise 2: Dynamic SQL

Use the AdventureWorks database

1.1

Create a stored procedure where the user can pass the name of any table in the database as an input parameter with the name @tablename with the data type NVARCHAR(50)

The command should be a parameter with the name @sql with the data type NVARCHAR(200) and it should be run by the command EXEC sp_executesql

1.2

Test the procedure with the table name Production.Product

Module 6: Procedural T-SQL

Exercise 1: Batches

Use the AdventureWorks database

1.1

Run the following code:

```
DECLARE @FirstName NVARCHAR(50);

SET @FirstName = 'John';

GO

SELECT *

FROM Person.Person

WHERE FirstName = @FirstName;
```

Change the code so that all people named John are returned!

Exercise 2: IF…ELSE

2.1

Declare a variable @color and assign it the value 'Red'.Use IF...ELSE to print:

'Color is Red' if the color is 'Red'

'Color is NOT Red' otherwise

2.2

Use the same variable @color. If the color is 'Red', print two lines:

'Color is Red'

'This message is only for Red products'

Otherwise, print:

'Color is NOT Red'

Use BEGIN...END where needed so the logic is correct and clean.

Exercise 3: Synonyms

3.1

Create a synonym called MyProducts for the table Production.Product

Select the first five products from Production.Product using the synonym

Module 7: Understanding and Handling Errors

Exercise 1: Raising Errors

Use the AdventureWorks database

1.1

Using the RAISERROR function, raise an ad-hoc error with the following properties:

Message: Unknown Error. Call helpdesk!

Severity Level: 16

State: 1

WITH LOG: Yes, log the error

1.2

In Object Explorer, under Management and SQL Server Logs, open the current log and find the error raised

1.3

Using the THROW statement, throw and event with the following properties:

Message: This is not the intended behavior

Error Number: 50010

State: 1

Exercise 2: Try Catch

Use the AdventureWorks database

1.1

Use structured error handling. In the try block, try to update the Column Orderdate in the table Sales.SalesOrderDetail with the value ‘Hejsan’ for the row with SalesOrderDetailId 1.

In the catch block, select the function ERROR_MESSAGE() using the alias ErrorMessage.

After the catch block, write a select statement that retrieves the first five rows of the table Sales.SalesOrderDetail

Module 8: Protecting yourself using transactions

Exercise 1: TRY CATCH Without Transactions

Use the AdventureWorks database

1.1

You will try the following two inserts:

```
    -- This one works: using an existing BusinessEntityID

      INSERT INTO Person.PersonPhone (BusinessEntityID, PhoneNumber,
      PhoneNumberTypeID)

       VALUES (1, '555-1234', 1);


    -- This one fails: bad foreign key (BusinessEntityID doesn't exist)

      INSERT INTO Person.PersonPhone (BusinessEntityID, PhoneNumber,
      PhoneNumberTypeID)

      VALUES (999999, '555-9999', 1);
```

Enclose the two inserts in a try-block and use a catch-block to display an error message.

After the catch-block, select BusinessEntityID, PhoneNumber and ModifiedDate ordered by ModifiedDate DESC

Only one row was inserted. Edit the Person.PersonPhone table in Object Explorer and remove the inserted row (you may have to refresh the table first)

1.2

Now, in the TRY-block, start with BEGIN TRAN and before the PRINT clause, add COMMIT TRAN. If everything works, both inserts will be effectuated.

In the CATCH-block, begin the CATCH-block with ROLLBACK TRAN. This means that if an error occurs, both inserts will be rolled back.

Run the statements. Nothing will be inserted

1.3

Now, in the TRY-block, replace the inserts with these new ones:

```
    -- This one works: using an existing BusinessEntityID

      INSERT INTO Person.PersonPhone (BusinessEntityID, PhoneNumber,
      PhoneNumberTypeID)

       VALUES (1, '555-1234', 1);


    -- This one works: good foreign key (BusinessEntityID does exist)

      INSERT INTO Person.PersonPhone (BusinessEntityID, PhoneNumber,
      PhoneNumberTypeID)

      VALUES (2, '555-9999', 1);
```

Run the statement. Two new inserts have taken place. Run the statement again. This time, no more inserts are done