Lab Manual

T2761, Querying Data with Transact-SQL - Continuation Course

Contents

Lab 1: Using Table Expressions

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You have been given a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. Because of advanced business requests, you will have to learn how to create and query different query expressions that represent a valid relational table.

Objectives

After completing this lab, you will be able to:

- Write queries that use views.
- Write queries that use derived tables.
- Write queries that use CTEs.
- Write queries that use inline TVFs.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Views

Scenario

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve All Products for a Specific Category
3. Write a SELECT Statement Against the Created View
4. Try to Use an ORDER BY Clause in the Created View
5. Add a Calculated Column to the View
6. Remove the Production.ProductsBeverages View

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab11\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve All Products for a Specific Category

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab11\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to return the productid, productname, supplierid, unitprice, and discontinued columns from the Production.Products table. Filter the results to include only products that belong to the category Beverages (categoryid equals 1).
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab11\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.
4. Modify the T-SQL code to include the following supplied T-SQL statement. Put this statement before the SELECT clause:

```
CREATE VIEW Production.ProductsBeverages AS
```

5. Execute the complete T-SQL statement. This will create an object view named ProductsBeverages under the Production schema.

▶ Task 3: Write a SELECT Statement Against the Created View

1. Write a SELECT statement to return the productid and productname columns from the Production.ProductsBeverages view. Filter the results to include only products where supplierid equals 1.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab11\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Try to Use an ORDER BY Clause in the Created View

1. The IT department has written a T-SQL statement that adds an ORDER BY clause to the view created in task 1:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1
ORDER BY productname;
```

2. Execute the provided code. What happened? What is the error message? Why did the query fail?

▶ Task 5: Add a Calculated Column to the View

1. The IT department has written a T-SQL statement that adds an additional calculated column to the view created in task 1:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT
productid, productname, supplierid, unitprice, discontinued,
CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END
FROM Production.Products
WHERE categoryid = 1;
```

2. Execute the provided query. What happened? What is the error message? Why did the query fail?
3. Apply the changes needed to get the T-SQL statement to execute properly.

▶ Task 6: Remove the Production.ProductsBeverages View

1. Remove the created view by executing the provided T-SQL statement:

```
IF OBJECT_ID(N'Production.ProductsBeverages', N'V') IS NOT NULL
DROP VIEW Production.ProductsBeverages;
```

2. Execute this code exactly as written inside a query window.

Results: After this exercise, you should know how to use a view in T-SQL statements.

Exercise 2: Writing Queries That Use Derived Tables

Scenario

The sales department would like to compare the sales amounts between the ordered year and the previous year to calculate the growth percentage. To prepare such a report, you will learn how to use derived tables inside T-SQL statements.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement Against a Derived Table
2. Write a SELECT Statement to Calculate the Total and Average Sales Amount
3. Write a SELECT Statement to Retrieve the Sales Growth Percentage

▶ Task 1: Write a SELECT Statement Against a Derived Table

1. Open the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against a derived table and retrieve the productid and productname columns. Filter the results to include only the rows in which the pricetype column value is equal to high. Use the SELECT statement from exercise 1, task 4, as the inner query that defines the derived table. Do not forget to use an alias for the derived table. (You can use the alias “p”.)
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab11\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Calculate the Total and Average Sales Amount

1. Write a SELECT statement to retrieve the custid column and two calculated columns: totalsalesamount, which returns the total sales amount per customer, and avgsalesamount, which returns the average sales amount of orders per customer. To correctly calculate the average sales amount of orders per customer, you should first calculate the total sales amount per order. You cando so by defining a derived table based on a query that joins the Sales.Orders and Sales.OrderDetails tables. You can use the custid and orderid columns from the Sales.Orders table and the qty and unitprice columns from the Sales.OrderDetails table.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Growth Percentage

1. Write a SELECT statement to retrieve the following columns:
   - orderyear, representing the year of the order date.
   - curtotalsales, representing the total sales amount for the current order year.
   - prevtotalsales, representing the total sales amount for the previous order year.
   - percentgrowth, representing the percentage of sales growth in the current order year compared to the previous order year.
2. You will have to write a T-SQL statement using two derived tables. To get the order year and total sales columns for each SELECT statement, you can query an already existing view named Sales.OrderValues. The val column represents the sales amount.
3. Do not forget that the order year 2006 does not have a previous order year in the database, but it should still be retrieved by the query.
4. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should be able to use derived tables in T-SQL statements.

Exercise 3: Writing Queries That Use CTEs

Scenario

The sales department needs an additional report showing the sales growth over the years for each customer. You could use your existing knowledge of derived tables and views, but instead you will practice how to use a CTE.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses a CTE

2. Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

3. Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

▶ Task 1: Write a SELECT Statement That Uses a CTE

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement like the one in exercise 2, task 1, but use a CTE instead of a derived table. Use inline column aliasing in the CTE query and name the CTE **ProductBeverages**.

3. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. Write a SELECT statement against Sales.OrderValues to retrieve each customer's ID and total sales amount for the year 2008. Define a CTE named c2008 based on this query, using the external aliasing form to name the CTE columns custid and salesamt2008. Join the Sales.Customers table and the c2008 CTE, returning the custid and contactname columns from the Sales.Customers table and the salesamt2008 column from the c2008 CTE.
2. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table. Also retrieve the following calculated columns:
   - salesamt2008, representing the total sales amount for the year 2008.
   - salesamt2007, representing the total sales amount for the year 2007.
   - percentgrowth, representing the percentage of sales growth between the year 2007 and 2008.
2. If percentgrowth is NULL, then display the value 0.
3. You can use the CTE from the previous task and add another one for the year 2007. Then join both of them with the Sales.Customers table. Order the result by the percentgrowth column.
4. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

Results: After this exercise, you should have an understanding of how to use a CTE in a T-SQL statement.

Exercise 4: Writing Queries That Use Inline TVFs

Scenario

You have learned how to write a SELECT statement against a view. However, since a view does not support parameters, you will now use an inline TVF to retrieve data as a relational table based on an input parameter.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer
2. Write a SELECT Statement Against the Inline TVF
3. Write a SELECT Statement to Retrieve the Top Three Products Based on the Total Sales Value for a Specific Customer
4. Using Inline TVFs, Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year
5. Remove the Created Inline TVFs

▶ Task 1: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. Open the T-SQL script **81 - Lab Exercise 4.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.OrderValues view and retrieve the custid and totalsalesamount columns as a total of the val column. Filter the results to include orders only for the year 2007.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.
4. Define an inline TVF using the following function header and add your previous query after the RETURN clause:

```
CREATE FUNCTION dbo.fnGetSalesByCustomer
(@orderyear AS INT) RETURNS TABLE
AS
RETURN
```

5. Modify the query by replacing the constant year value 2007 in the WHERE clause with the parameter @orderyear.
6. Highlight the complete code and execute it. This will create an inline TVF named dbo.fnGetSalesByCustomer.

▶ Task 2: Write a SELECT Statement Against the Inline TVF

1. Write a SELECT statement to retrieve the custid and totalsalesamount columns from the dbo.fnGetSalesByCustomer inline TVF. Use the value 2007 for the needed parameter.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Top Three Products Based on the Total Sales Value for a Specific Customer

1. In this task, you will query the Production.Products and Sales.OrderDetails tables. Write a SELECT statement that retrieves the top three sold products based on the total sales value for the customer with ID 1. Return the productid and productname columns from the Production.Products table. Use the qty and unitprice columns from the Sales.OrderDetails table to compute each order line's value, and return the sum of all values per product, naming the resulting column totalsalesamount. Filter the results to include only the rows where the custid value is equal to 1.
2. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\84 - Lab Exercise 4 - Task 3_1 Result.txt.
3. Create an inline TVF based on the following function header, using the previous SELECT statement. Replace the constant custid value 1 in the query with the function's input parameter @custid:

```
CREATE FUNCTION dbo.fnGetTop3ProductsForCustomer
(@custid AS INT) RETURNS TABLE
AS
RETURN
```

4. Highlight the complete code and execute it. This will create an inline TVF named dbo.fnGetTop3ProductsForCustomer that accepts a parameter for the customer ID.
5. Test the created inline TVF by writing a SELECT statement against it and use the value 1 for the customer ID parameter. Retrieve the productid, productname, and totalsalesamount columns, and use the alias "p" for the inline TVF.

Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\85 - Lab Exercise 4 - Task 3_2 Result.txt.

▶ Task 4: Using Inline TVFs, Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. Write a SELECT statement to retrieve the same result as in exercise 3, task 3, but use the created TVF in task 2 (dbo.fnGetSalesByCustomer).
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab11\Solution\86 - Lab Exercise 4 - Task 4 Result.txt.

▶ Task 5: Remove the Created Inline TVFs

1. Remove the created inline TVFs by executing the provided T-SQL statement:

```
IF OBJECT_ID('dbo.fnGetSalesByCustomer') IS NOT NULL
DROP FUNCTION dbo.fnGetSalesByCustomer; IF
OBJECT_ID('dbo.fnGetTop3ProductsForCustomer') IS NOT NULL
DROP FUNCTION dbo.fnGetTop3ProductsForCustomer;
```

2. Execute this code exactly as written inside a query window.

Results: After this exercise, you should know how to use inline TVFs in T-SQL statements.

Lab Answer Key 1: Using Table Expressions

Exercise 1: Writing Queries That Use Views

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd.**
2. In the **D:\Labfiles\Lab11\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement to Retrieve All Products for a Specific Category

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab11\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
6. In the query pane, type the following query after the **Task 1** description:

```
SELECT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1;
```

7. Highlight the written query, and click **Execute**.
8. Modify the query to include the provided CREATE VIEW statement. The query should look like this:

```
CREATE VIEW Production.ProductsBeverages AS
SELECT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1;
```

9. Highlight the modified query, and click **Execute**.

▶ Task 3: Write a SELECT Statement Against the Created View

1. In the query pane, type the following query after the **Task 2** description:

```
SELECT
productid, productname
FROM Production.ProductsBeverages
WHERE supplierid = 1;
```

2. Highlight the written query, and click **Execute**.

▶ Task 4: Try to Use an ORDER BY Clause in the Created View

1. Highlight the provided T-SQL statement under the **Task 3** description, and then click **Execute**.
2. Observe the error message:

```
The ORDER BY clause is invalid in views, inline functions, derived tables,
subqueries, and common table expressions, unless TOP, OFFSET or FOR XML is also
specified.
```

Why did the query fail? It failed because the view is supposed to represent a relation, and a relation has no order. You can only use the ORDER BY clause in the view if you specify the TOP, OFFSET, or FOR XML option. The reason you can use ORDER BY in special cases is that it serves a meaning other than presentation ordering to these special cases.

▶ Task 5: Add a Calculated Column to the View

1. Highlight the provided T-SQL statement under the **Task 4** description, and then click **Execute**.
2. Observe the error message:

```
Create View or Function failed because no column name was specified for column 6.
```

Why did the query fail? It failed because each column must have a unique name. In the provided T-SQL statement, the last column does not have a name.

3. Modify the T-SQL statement to include the column name pricetype. The query should look like this:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT
   productid, productname, supplierid, unitprice, discontinued,
   CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END AS pricetype
FROM Production.Products
WHERE categoryid = 1;
```

4. Highlight the written query, and click **Execute**.

▶ Task 6: Remove the Production.ProductsBeverages View

- Highlight the provided T-SQL statement under the **Task 5** description and click **Execute**.

Results: After this exercise, you should know how to use a view in T-SQL statements.

Exercise 2: Writing Queries That Use Derived Tables

▶ Task 1: Write a SELECT Statement Against a Derived Table

1. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, type the following query after the **Task 1** description:

```
SELECT
p.productid, p.productname
FROM
(
SELECT
productid, productname, supplierid, unitprice, discontinued,
CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END AS pricetype
FROM Production.Products
WHERE categoryid = 1
) AS p
WHERE p.pricetype = N'high';
```

4. Highlight the written query, and click **Execute**.

▶ Task 2: Write a SELECT Statement to Calculate the Total and Average Sales Amount

1. In the query pane, type the following query after the **Task 2** description:

```
SELECT
c.custid,
SUM(c.totalsalesamountperorder) AS totalsalesamount,
AVG(c.totalsalesamountperorder) AS avgsalesamount
FROM
(
SELECT
o.custid, o.orderid, SUM(d.unitprice * d.qty) AS totalsalesamountperorder
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails d ON d.orderid = o.orderid
GROUP BY o.custid, o.orderid
) AS c
GROUP BY c.custid;
```

2. Highlight the written query, and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Growth Percentage

1. In the query pane, type the following query after the **Task 3** description:

```
SELECT
cy.orderyear,
cy.totalsalesamount AS curtotalsales,
py.totalsalesamount AS prevtotalsales,
(cy.totalsalesamount - py.totalsalesamount) / py.totalsalesamount * 100. AS
percentgrowth
FROM
(
SELECT
YEAR(orderdate) AS orderyear, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
GROUP BY YEAR(orderdate)
) AS cy
LEFT OUTER JOIN
(
SELECT
YEAR(orderdate) AS orderyear, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
GROUP BY YEAR(orderdate)
) AS py ON cy.orderyear = py.orderyear + 1
ORDER BY cy.orderyear;
```

2. Highlight the written query, and click **Execute**.

Results: After this exercise, you should be able to use derived tables in T-SQL statements.

Exercise 3: Writing Queries That Use CTEs

▶ Task 1: Write a SELECT Statement That Uses a CTE

1. In Solution Explorer, double-click **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, type the following query after the **Task 1** description:

```
WITH ProductsBeverages AS
(
SELECT
productid, productname, supplierid, unitprice, discontinued,
CASE WHEN unitprice > 100. THEN N'high' ELSE N'normal' END AS pricetype
FROM Production.Products
WHERE categoryid = 1
)
SELECT
productid, productname
FROM ProductsBeverages
WHERE pricetype = N'high';
```

4. Highlight the written query, and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. In the query pane, type the following query after the **Task 2** description:

```
WITH c2008 (custid, salesamt2008) AS
(
SELECT
custid, SUM(val)
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2008
GROUP BY custid
)
SELECT
c.custid, c.contactname, c2008.salesamt2008
FROM Sales.Customers AS c
LEFT OUTER JOIN c2008 ON c.custid = c2008.custid;
```

2. Highlight the written query, and click **Execute**.

▶ Task 3: Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. In the query pane, type the following query after the **Task 3** description:

```
WITH c2008 (custid, salesamt2008) AS
(
SELECT
custid, SUM(val)
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2008
GROUP BY custid
),
c2007 (custid, salesamt2007) AS
(
SELECT
custid, SUM(val)
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2007
GROUP BY custid
)
SELECT
c.custid, c.contactname,
c2008.salesamt2008,
c2007.salesamt2007,
COALESCE((c2008.salesamt2008 - c2007.salesamt2007) / c2007.salesamt2007 * 100., 0) AS
percentgrowth
FROM Sales.Customers AS c
LEFT OUTER JOIN c2008 ON c.custid = c2008.custid
LEFT OUTER JOIN c2007 ON c.custid = c2007.custid
ORDER BY percentgrowth DESC;
```

2. Highlight the written query, and click **Execute**.

Results: After this exercise, you should have an understanding of how to use a CTE in a T-SQL statement.

Exercise 4: Writing Queries That Use Inline TVFs

▶ Task 1: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer

1. In Solution Explorer, double-click **81 - Lab Exercise 4.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, type the following query after the **Task 1** description:

```
SELECT
custid, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
WHERE YEAR(orderdate) = 2007
GROUP BY custid;
```

4. Highlight the written query, and click **Execute**.
5. Create an inline TVF using the provided code. Add the previous query, putting it after the function's RETURN clause. In the query, replace the order date of 2007 with the function's input parameter @orderyear. The resulting T-SQL statement should look like this:

```
CREATE FUNCTION dbo.fnGetSalesByCustomer
(@orderyear AS INT) RETURNS TABLE
AS
RETURN
SELECT
custid, SUM(val) AS totalsalesamount
FROM Sales.OrderValues
WHERE YEAR(orderdate) = @orderyear
GROUP BY custid;
```

   This T-SQL statement will create an inline TVF named dbo.fnGetSalesByCustomer.

6. Highlight the written T-SQL statement, and click **Execute**.

▶ Task 2: Write a SELECT Statement Against the Inline TVF

1. In the query pane, type the following query after the **Task 2** description:

```
SELECT
custid, totalsalesamount
FROM dbo.fnGetSalesByCustomer(2007);
```

2. Highlight the written query, and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve the Top Three Products Based on the Total Sales Value for a Specific Customer

1. In the query pane, type the following query after the **Task 3** description:

```
SELECT TOP(3)
d.productid,
MAX(p.productname) AS productname,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
WHERE custid = 1
GROUP BY d.productid
ORDER BY totalsalesamount DESC;
```

2. Highlight the written query, and click **Execute**.

3. Create an inline TVF using the provided code. Add the previous query, putting it after the function's RETURN clause. In the query, replace the constant custid value of 1 with the function's input parameter @custid. The resulting T-SQL statement should look like this:

```
CREATE FUNCTION dbo.fnGetTop3ProductsForCustomer
(@custid AS INT) RETURNS TABLE
AS
RETURN
SELECT TOP(3)
d.productid,
MAX(p.productname) AS productname,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
WHERE custid = @custid
GROUP BY d.productid
ORDER BY totalsalesamount DESC;
```

4. To test the inline TVF, add the following query after the CREATE FUNCTION and GO statement:

```
SELECT
p.productid,
p.productname,
p.totalsalesamount
FROM dbo.fnGetTop3ProductsForCustomer(1) AS p;
```

5. Highlight the CREATE FUNCTION statement and the written query, and click **Execute**.

▶ Task 4: Using Inline TVFs, Write a SELECT Statement to Compare the Total Sales Amount for Each Customer Over the Previous Year

1. In the query pane, type the following query after the **Task 4** description:

```
SELECT
c.custid, c.contactname,
c2008.totalsalesamount AS salesamt2008,
c2007.totalsalesamount AS salesamt2007,
COALESCE((c2008.totalsalesamount - c2007.totalsalesamount) / c2007.totalsalesamount *
100., 0) AS percentgrowth
FROM Sales.Customers AS c
LEFT OUTER JOIN dbo.fnGetSalesByCustomer(2007) AS c2007 ON c.custid = c2007.custid
LEFT OUTER JOIN dbo.fnGetSalesByCustomer(2008) AS c2008 ON c.custid = c2008.custid;
```

2. Highlight the written query, and click **Execute**.

▶ Task 5: Remove the Created Inline TVFs

- Highlight the provided T-SQL statement under the **Task 5** description and click **Execute**.

Results: After this exercise, you should know how to use inline TVFs in T-SQL statements.

Lab 2: Using Set Operators

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. Because of the complex business requirements, you will need to prepare combined results from multiple queries using set operators.

Objectives

After completing this lab, you will be able to:

- Write queries that use the UNION and UNION ALL operators.
- Write queries that use the CROSS APPLY and OUTER APPLY operators.
- Write queries that use the EXCEPT and INTERSECT operators.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use UNION Set Operators and UNION ALL Multi-Set Operators

Scenario

The marketing department needs some additional information regarding segmentation of products and customers. It would like to have a report, based on multiple queries, which is presented as one result. You will use the UNION operator to write different SELECT statements, and then merge them together into one result.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement to Retrieve Specific Products

3. Write a SELECT Statement to Retrieve All Products with a Total Sales Amount of More Than $50,000

4. Merge the Results from Task 1 and Task 2

5. Write a SELECT Statement to Retrieve the Top 10 Customers by Sales Amount for January 2008 and February 2008

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.

2. Run **Setup.cmd** in the **D:\Labfiles\Lab12\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve Specific Products

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab12\Starter\Project\Project.ssmssln** and the T-SQL **script 51 - Lab Exercise 1.sql**. Ensure that you are connected to the **TSQL** database.

2. Write a SELECT statement to return the **productid** and **productname** columns from the **Production.Products** table. Filter the results to include only products that have a categoryid value 4.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\Solution\52 - Lab Exercise 1 - Task 1 Result.txt. Remember the number of rows in the results.

▶ Task 3: Write a SELECT Statement to Retrieve All Products with a Total Sales Amount of More Than $50,000

1. Write a SELECT statement to return the **productid** and **productname** columns from the **Production.Products** table. Filter the results to include only products that have a total sales amount of more than $50,000. For the total sales amount, you will need to query the **Sales.OrderDetails** table and aggregate all order line values (qty * unitprice) for each product.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\Solution\53 - Lab Exercise 1 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 4: Merge the Results from Task 1 and Task 2

1. Write a SELECT statement that uses the UNION operator to retrieve the **productid** and **productname** columns from the T-SQL statements in task 1 and task 2.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\Solution\54 - Lab Exercise 1 - Task 3_1 Result.txt.
3. What is the total number of rows in the results? If you compare this number with an aggregate value of the number of rows from tasks 1 and 2, is there any difference?
4. Copy the T-SQL statement and modify it to use the UNION ALL operator.
5. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\Solution\55 - Lab Exercise 1 - Task 3_2 Result.txt.
6. What is the total number of rows in the result? What is the difference between the UNION and UNION ALL operators?

▶ Task 5: Write a SELECT Statement to Retrieve the Top 10 Customers by Sales Amount for January 2008 and February 2008

1. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table. Display the top 10 customers by sales amount for January 2008 and display the top 10 customers by sales amount for February 2008. (Hint: write two SELECT statements, eachjoining Sales.Customers and Sales.OrderValues, and use the appropriate set operator.)
2. Execute the T-SQL code and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\Solution\56 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should know how to use the UNION and UNION ALL set operators in T-SQL statements.

Exercise 2: Writing Queries That Use the CROSS APPLY and OUTER APPLY Operators

Scenario

The sales department needs a more advanced analysis of buying behavior. Staff want to find out the top three products, based on sales revenue, for each customer. Use the APPLY operator to achieve this result.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Last Two Orders for Each Product

2. Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Top Three Products, Based on Sales Revenue, for Each Customer

3. Use the OUTER APPLY Operator

4. Analyze the OUTER APPLY Operator

5. Remove the TVF Created for This Lab

▶ Task 1: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Last Two Orders for Each Product

1. Open the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **productid** and **productname** columns from the **Production.Products** table. In addition, for each product, retrieve the last two rows from the **Sales.OrderDetails** table based on orderid number.
3. Use the CROSS APPLY operator and a correlated subquery. Order the result by the column **productid**.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab12\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Top Three Products, Based on Sales Revenue, for Each Customer

1. Execute the provided T-SQL code to create the inline TVF fnGetTop3ProductsForCustomer:

```
DROP FUNCTION IF EXISTS dbo.fnGetTop3ProductsForCustomer;
GO
CREATE FUNCTION dbo.fnGetTop3ProductsForCustomer
(@custid AS INT) RETURNS TABLE
AS
RETURN
SELECT TOP(3)
d.productid,
p.productname,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
WHERE custid = @custid
GROUP BY d.productid, p.productname
ORDER BY totalsalesamount DESC;
```

2. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table. Use the CROSS APPLY operator with the dbo.fnGetTop3ProductsForCustomer function to retrieve **productid**, **productname**, and **totalsalesamount** columns for each customer.

3. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab12\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 3: Use the OUTER APPLY Operator

1. Copy the T-SQL statement from the previous task and modify it by replacing the CROSS APPLY operator with the OUTER APPLY operator.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\Solution\64 - Lab Exercise 2 - Task 3 Result.txt. Notice that more rows are returned than in the previous task.

▶ Task 4: Analyze the OUTER APPLY Operator

1. Copy the T-SQL statement from the previous task and modify it by filtering the results to show only customers without products. (Hint: in a WHERE clause, look for any column returned by the inline TVF that is NULL.)
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.
3. What is the difference between the CROSS APPLY and OUTER APPLY operators?

▶ Task 5: Remove the TVF Created for This Lab

1. Remove the created inline TVF by executing the provided T-SQL code:

```
DROP FUNCTION IF EXISTS dbo.fnGetTop3ProductsForCustomer;
```

2. Execute this code exactly as written inside a query window.

Results: After this exercise, you should be able to use the CROSS APPLY and OUTER APPLY operators in your T-SQL statements.

Exercise 3: Writing Queries That Use the EXCEPT and INTERSECT Operators

Scenario

The marketing department was satisfied with the results from exercise 1, but the staff now need to see specific rows from one result set that are not present in the other result set. You will have to write different queries using the EXCEPT and INTERSECT operators.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Return All Customers Who Bought More Than 20 Distinct Products
2. Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More Than 20 Distinct Products
3. Write a SELECT Statement to Retrieve Customers Who Spent More Than $10,000
4. Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators
5. Change the Operator Precedence

▶ Task 1: Write a SELECT Statement to Return All Customers Who Bought More Than 20 Distinct Products

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **custid** column from the **Sales.Orders** table. Filter the results to include only customers who bought more than 20 different products (based on the **productid** column from the **Sales.OrderDetails** table).
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More Than 20 Distinct Products

1. Write a SELECT statement to retrieve the **custid** column from the **Sales.Orders** table. Filter the results to include only customers from the country USA and exclude all customers from the previous (task 1) result. (Hint: use the EXCEPT operator and the previous query.)
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Spent More Than $10,000

1. Write a SELECT statement to retrieve the **custid** column from the **Sales.Orders** table. Filter only customers who have a total sales value greater than $10,000. Calculate the sales value using the **qty** and **unitprice** columns from the **Sales.OrderDetails** table.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators

1. Copy the T-SQL statement from task 2. Add the INTERSECT operator at the end of the statement. After the INTERSECT operator, add the T-SQL statement from task 3.
2. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab12\Solution\75 - Lab Exercise 3 - Task 4 Result.txt. Notice the total number of rows in the results.
3. In business terms, can you explain which customers are part of the result?

▶ Task 5: Change the Operator Precedence

1. Copy the T-SQL statement from the previous task and add parentheses around the first two SELECT statements (from the beginning until the INTERSECT operator).
2. Execute the T-SQL statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab12\Solution\76 - Lab Exercise 3 - Task 5 Result.txt. Notice the total number of rows in the results.
3. Are the results different to the results from task 4? Please explain why.
4. What is the precedence among the set operators?
5. Close SQL Server Management Studio, without saving any changes.

Results: After this exercise, you should have an understanding of how to use the EXCEPT and INTERSECT operators in T-SQL statements.

Lab Answer Key 2: Using Set Operators

Exercise 1: Writing Queries That Use UNION Set Operators and UNION ALL Multi-Set Operators

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab12\Starter** folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, wait for the script to finish, and then press any key.

▶ Task 2: Write a SELECT Statement to Retrieve Specific Products

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows® authentication.
2. On the **File** menu, point to **Open**, and then click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab12\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
6. In the query pane, after the **Task 1** description, type the following query:

```
SELECT
productid, productname
FROM Production.Products
WHERE categoryid = 4;
```

7. Highlight the written query, and click **Execute**. Observe that the query retrieved 10 rows.

▶ Task 3: Write a SELECT Statement to Retrieve All Products with a Total Sales Amount of More Than $50,000

1. In the query pane, after the **Task 2** description, type the following query:

```
SELECT
d.productid, p.productname
FROM Sales.OrderDetails d
INNER JOIN Production.Products p ON p.productid = d.productid
GROUP BY d.productid, p.productname
HAVING SUM(d.qty * d.unitprice) > 50000;
```

2. Highlight the written query, and click **Execute**. Observe that the query retrieved four rows.

▶ Task 4: Merge the Results from Task 1 and Task 2

1. In the query pane, after the **Task 3** description, type the following query:

```
SELECT
productid, productname
FROM Production.Products
WHERE categoryid = 4
UNION
SELECT
d.productid, p.productname
FROM Sales.OrderDetails d
INNER JOIN Production.Products p ON p.productid = d.productid
GROUP BY d.productid, p.productname
HAVING SUM(d.qty * d.unitprice) > 50000;
```

2. Highlight the written query, and click **Execute**.
3. Observe the result. What is the total number of rows in the result? If you compare this number with an aggregate value of the number of rows from tasks 1 and 2, is there any difference? The total number of rows retrieved by the query is 12. This is two rows less than the aggregate value of rows from the query in task 1 (10 rows) and task 2 (four rows).
4. Highlight the previous query, and on the **Edit** menu, click **Copy**.
5. In the query window, click the line after the written T-SQL statement, and on the **Edit** menu, click **Paste**.
6. Modify the T-SQL statement by replacing the UNION operator with the UNION ALL operator. The query should look like this:

```
SELECT
productid, productname
FROM Production.Products
WHERE categoryid = 4
UNION ALL
SELECT
d.productid, p.productname
FROM Sales.OrderDetails d
INNER JOIN Production.Products p ON p.productid = d.productid
GROUP BY d.productid, p.productname
HAVING SUM(d.qty * d.unitprice) > 50000;
```

7. Highlight the modified query, and click **Execute**.
8. Observe the result. What is the total number of rows in the result? What is the difference between the UNION and UNION ALL operators? The total number of rows retrieved by the query is 14. It is the same as the aggregate value of rows from the queries in tasks 1 and 2. This is because UNION ALL is a multi-set operator that returns all rows that appear in any of the inputs, without really comparing rows and without eliminating duplicates. The UNION set operator removes the duplicate rows and the result consists of only distinct rows.
9. So, when should you use either UNION ALL or UNION when unifying two inputs? If a potential exists for duplicates and you need to return them, use UNION ALL. If a potential exists for duplicates but you need to return distinct rows, use UNION. If no potential exists for duplicates when unifying the two inputs, UNION and UNION ALL are logically equivalent. However, in such a case, using UNION ALL is recommended because it removes the overhead of SQL Server checking for duplicates.

▶ Task 5: Write a SELECT Statement to Retrieve the Top 10 Customers by Sales Amount for January 2008 and February 2008

1. In the query pane, after the **Task 4** description, type the following query:

```
SELECT
c1.custid, c1.contactname
FROM
(
SELECT TOP (10)
o.custid, c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20080201'
GROUP BY o.custid, c.contactname
ORDER BY SUM(o.val) DESC
) AS c1
UNION
SELECT c2.custid, c2.contactname
FROM
(
SELECT TOP (10)
o.custid, c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.orderdate >= '20080201' AND o.orderdate < '20080301'
GROUP BY o.custid, c.contactname
ORDER BY SUM(o.val) DESC
) AS c2;
```

2. Highlight the written query, and click **Execute**.

Results: After this exercise, you should know how to use the UNION and UNION ALL set operators in T-SQL statements.

Exercise 2: Writing Queries That Use the CROSS APPLY and OUTER APPLY Operators

▶ Task 1: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Last Two Orders for Each Product

1. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
SELECT
p.productid, p.productname, o.orderid
FROM Production.Products AS p
CROSS APPLY
(
SELECT TOP(2)
d.orderid
FROM Sales.OrderDetails AS d
WHERE d.productid = p.productid
ORDER BY d.orderid DESC
) o
ORDER BY p.productid;
```

4. Highlight the written query, and click **Execute**.

▶ Task 2: Write a SELECT Statement That Uses the CROSS APPLY Operator to Retrieve the Top Three Products, Based on Sales Revenue, for Each Customer

1. Highlight the provided T-SQL code after the **Task 2** description, and click **Execute**.
2. In the query pane, type the following query after the provided T-SQL code:

```
SELECT
c.custid, c.contactname, p.productid, p.productname, p.totalsalesamount
FROM Sales.Customers AS c
CROSS APPLY dbo.fnGetTop3ProductsForCustomer (c.custid) AS p
ORDER BY c.custid;
```

   Tip: you can make the inline TVF (dbo.fnGetTop3ProductsForCustomer) more flexible by making the number of top rows to return an argument instead of fixing the number to three in the function's code.

3. Highlight the written query, and click **Execute**. Note that the query retrieves 265 rows.

▶ Task 3: Use the OUTER APPLY Operator

1. Highlight the previous query in **Task 2**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement by replacing the CROSS APPLY operator with the OUTER APPLY operator. The query should look like this:

```
SELECT
c.custid, c.contactname, p.productid, p.productname, p.totalsalesamount
FROM Sales.Customers AS c
OUTER APPLY dbo.fnGetTop3ProductsForCustomer (c.custid) AS p
ORDER BY c.custid;
```

4. Highlight the modified query, and click **Execute**.

5. Notice that the query retrieved 267 rows, which is two more rows than the previous query. Observe the result to see two rows with NULL in the columns from the inline TVF.

▶ Task 4: Analyze the OUTER APPLY Operator

1. Highlight the previous query in **Task 3**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 4** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement to search for a null productid. The query should look like this:

```
SELECT
c.custid, c.contactname, p.productid, p.productname, p.totalsalesamount
FROM Sales.Customers AS c
OUTER APPLY dbo.fnGetTop3ProductsForCustomer (c.custid) AS p
WHERE p.productid IS NULL;
```

4. Highlight the modified query, and click **Execute**.
5. Notice that the query now retrieves the two rows that do not occur in the CROSS APPLY query in Task 2.

▶ Task 5: Remove the TVF Created for This Lab

- Highlight the provided T-SQL statement after the **Task 5** description, and click **Execute**.

Results: After this exercise, you should be able to use the CROSS APPLY and OUTER APPLY operators in your T-SQL statements.

Exercise 3: Writing Queries That Use the EXCEPT and INTERSECT Operators

▶ Task 1: Write a SELECT Statement to Return All Customers Who Bought More Than 20 Distinct Products

1. In Solution Explorer, double-click **71 - Lab Exercise 3.sql**.
2. In the query pane, highlight the statement **USE TSQL;**, and then click **Execute**.
3. In the query pane, after the **Task 1** description, type the following query:

```
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20;
```

4. Highlight the written query, and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve All Customers from the USA, Except Those Who Bought More Than 20 Distinct Products

1. In the query pane, after the **Task 2** description, type the following query:

```
SELECT
custid
FROM Sales.Customers
WHERE country = 'USA'
EXCEPT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20;
```

2. Highlight the written query, and click **Execute**.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Spent More Than $10,000

1. In the query pane, after the **Task 3** description, type the following query:

```
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000;
```

2. Highlight the written query, and click **Execute**.

▶ Task 4: Write a SELECT Statement That Uses the EXCEPT and INTERSECT Operators

1. Highlight the query from **Task 2**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 4** description, and on the **Edit** menu, click **Paste**.
3. Modify the first SELECT statement so that it selects all customers—not just those from the USA—and include the INTERSECT operator, adding the query from **Task 3**. The query should look like this:

```
SELECT
c.custid
FROM Sales.Customers AS c
EXCEPT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20
INTERSECT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000;
```

4. Highlight the modified query, and click **Execute**.

5. Observe that the total number of rows is 59. In business terms, can you explain which customers are part of the result? Because the INTERSECT operator is evaluated before the EXCEPT operator, the result consists of all customers, except those who bought more than 20 different products and spent more than $10,000.

▶ Task 5: Change the Operator Precedence

1. Highlight the previous query in **Task 4**, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 5** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement by adding a set of parentheses around the first two SELECT statements. The query should look like this:

```
(
SELECT
c.custid
FROM Sales.Customers AS c
EXCEPT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT d.productid) > 20
)
INTERSECT
SELECT
o.custid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING SUM(d.qty * d.unitprice) > 10000;
```

4. Highlight the provided T-SQL statement, and click **Execute**.
5. Observe that the total number of rows is nine. Is that different to the result of the query in task 4? Yes, because when you added the parentheses, the SQL Server engine first evaluated the EXCEPT operation, and then the INTERSECT operation. In business terms, this query retrieved all customers who did not buy more than 20 distinct products, and who spent more than $10,000.
6. What is the precedence among the set operators? SQL defines the following precedence among the set operations: INTERSECT precedes UNION and EXCEPT, while UNION and EXCEPT are considered equal. In a query that contains multiple set operations, INTERSECT operations are evaluated first, and then operations with the same precedence are evaluated, based on appearance order. Remember that set operations in parentheses are always processed first.
7. Close SQL Server Management Studio, without saving any changes.

Results: After this exercise, you should have an understanding of how to use the EXCEPT and INTERSECT operators in T-SQL statements.

Lab 3: Using Window Ranking, Offset, and Aggregate Functions

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. To fill these requests, you will need to calculate ranking values, as well as the difference between two consecutive rows, and running totals. You will use window functions to achieve these calculations.

Objectives

After completing this lab, you will be able to:

- Write queries that use ranking functions.
- Write queries that use offset functions.
- Write queries that use window aggregation functions.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Ranking Functions

Scenario

The sales department would like to rank orders by their values for each customer. You will provide the report by using the RANK function. You will also practice how to add a calculated column to display the row number in the SELECT clause.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

3. Add an Additional Column Using the RANK Function

4. Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

5. Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

6. Filter Only Orders with the Top Two Ranks

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab13\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab13\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the orderid, orderdate, and val columns in addition to a calculated column named rowno from the view Sales.OrderValues. Use the ROW_NUMBER function to return rowno. Order the row numbers by the orderdate column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

▶ Task 3: Add an Additional Column Using the RANK Function

1. Copy the previous T-SQL statement and modify it by including an additional column named rankno. To create rankno, use the RANK function, with the rank order based on the orderdate column.
2. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\53 - Lab Exercise 1 - Task 2 Result.txt. Notice the different values in the rowno and rankno columns for some of the rows.
3. What is the difference between the RANK and ROW_NUMBER functions?

▶ Task 4: Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

1. Write a SELECT statement to retrieve the orderid, orderdate, custid, and val columns, as well as a calculated column named orderrankno from the Sales.OrderValues view. The orderrankno column should display the rank per each customer independently, based on val ordering in descending order.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

1. Write a SELECT statement to retrieve the custid and val columns from the Sales.OrderValues view. Add two calculated columns:
   - orderyear as a year of the orderdate column.
   - orderrankno as a rank number, partitioned by the customer and order year, and ordered by the order value in descending order.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\55 - Lab Exercise 1 - Task 4 Result.txt.

▶ Task 6: Filter Only Orders with the Top Two Ranks

1. Copy the previous query and modify it to filter only orders with the first two ranks based on the orderrankno column.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\ Solution\56 - Lab Exercise 1 - Task 5 Result.txt.

Results: After this exercise, you should know how to use ranking functions in T-SQL statements.

Exercise 2: Writing Queries That Use Offset Functions

Scenario

You need to provide separate reports to analyze the difference between two consecutive rows. This will enable business users to analyze growth and trends.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)
2. Add a Column to Display the Running Sales Total
3. Analyze the Sales Information for the Year 2007

▶ Task 1: Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Define a CTE named OrderRows based on a query that retrieves the orderid, orderdate, and val columns from the Sales.OrderValues view. Add a calculated column named rowno using the ROW_NUMBER function, ordering by the orderdate and orderid columns.
3. Write a SELECT statement against the CTE and use the LEFT JOIN with the same CTE to retrieve the current row and the previous row based on the rowno column. Return the orderid, orderdate, and val columns for the current row and the val column from the previous row as prevval. Add a calculated column named diffprev to show the difference between the current val and previous val.
4. Execute the T-SQL code and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab13\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Add a Column to Display the Running Sales Total

1. Write a SELECT statement that uses the LAG function to achieve the same results as the query in the previous task. The query should not define a CTE.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Analyze the Sales Information for the Year 2007

1. Define a CTE named SalesMonth2007 that creates two columns: monthno (the month number of the orderdate column) and val (aggregated val column). Filter the results to include only the order year 2007 and group by monthno.
2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
   - **avglast3months**. This column should contain the average sales amount for the last three months before the current month, using a window aggregate function. You can assume that there are no missing months.
   - **ytdval**. This column should contain the cumulative sales value up to the current month.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\63 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should be able to use the offset functions in your T-SQL statements.

Exercise 3: Writing Queries That Use Window Aggregate Functions

Scenario

To better understand the cumulative sales value of a customer through time and to provide the sales analyst with a year-to-date analysis, you will have to write different SELECT statements that use the window aggregate functions.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

2. Add a Column to Display the Running Sales Total

3. Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

▶ Task 1: Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid, orderid, orderdate, and val columns from the Sales.OrderValues view. Add a calculated column named percoftotalcust containing a percentage value of each order sales amount compared to the total sales amount for that customer.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Add a Column to Display the Running Sales Total

1. Copy the previous SELECT statement and modify it by adding a new calculated column named runval. This column should contain a running sales total for each customer based on order date, using orderid as the tiebreaker.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

1. Copy the SalesMonth2007 CTE in the final task in exercise 2. Write a SELECT statement to retrieve the monthno and val columns. Add two calculated columns:
   - **avglast3months**. This column should contain the average sales amount for the last three months before the current month using a window aggregate function. You can assume that there are nomissing months.
   - **ytdval**. This column should contain the cumulative sales value up to the current month.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab13\ Solution\74 - Lab Exercise 3 - Task 3 Result.txt.
3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have a basic understanding of how to use window aggregate functions in T-SQL statements.

Lab Answer Key 3: Using Window Ranking, Offset, and AggregateFunctions

Exercise 1: Writing Queries That Use Ranking Functions

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab13\Starter** folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement That Uses the ROW_NUMBER Function to Create a Calculated Column

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab13\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**.
5. When the query window opens, highlight the statement **USE TSQL**; and click **Execute**.
6. In the query pane, type the following query after the **Task 1** description:

```
SELECT
orderid,
orderdate,
val,
ROW_NUMBER() OVER (ORDER BY orderdate) AS rowno
FROM Sales.OrderValues;
```

7. Highlight the written query and click **Execute**.

▶ Task 3: Add an Additional Column Using the RANK Function

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the **Task 2** description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement by adding an additional calculated column. The query should look like this:

```
SELECT
orderid,
orderdate,
val,
ROW_NUMBER() OVER (ORDER BY orderdate) AS rowno,
RANK() OVER (ORDER BY orderdate) AS rankno
FROM Sales.OrderValues;
```

4. Highlight the written query and click **Execute**.
5. Observe the results. What is the difference between the RANK and ROW_NUMBER functions? The ROW_NUMBER function provides unique sequential integer values within the partition. The RANK function assigns the same ranking value to rows with the same values in the specified sort columns when the ORDER BY list is not unique. Also, the RANK function skips the next number if there is a tie in the ranking value.

▶ Task 4: Write A SELECT Statement to Calculate a Rank, Partitioned by Customer and Ordered by the Order Value

1. In the query pane, type the following query after the **Task 3** description:

```
SELECT
orderid,
orderdate,
custid,
val,
RANK() OVER (PARTITION BY custid ORDER BY val DESC) AS orderrankno FROM
Sales.OrderValues;
```

2. Highlight the written query and click **Execute**.

▶ Task 5: Write a SELECT Statement to Rank Orders, Partitioned by Customer and Order Year, and Ordered by the Order Value

1. In the query pane, type the following query after the **Task 4** description:

```
SELECT
custid,
val,
YEAR(orderdate) as orderyear,
RANK() OVER (PARTITION BY custid, YEAR(orderdate) ORDER BY val DESC) AS orderrankno
FROM Sales.OrderValues;
```

2. Highlight the written query and click **Execute**.

▶ Task 6: Filter Only Orders with the Top Two Ranks

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the **Task 5** description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL statement to look like this:

```
SELECT
s.custid,
s.orderyear,
s.orderrankno,
s.val
FROM
(
SELECT
custid,
val,
YEAR(orderdate) as orderyear,
RANK() OVER (PARTITION BY custid, YEAR(orderdate) ORDER BY val DESC) AS orderrankno
FROM Sales.OrderValues
) AS s
WHERE s.orderrankno <= 2;
```

4. Highlight the written query and click **Execute**.

Results: After this exercise, you should know how to use ranking functions in T-SQL statements.

Exercise 2: Writing Queries That Use Offset Functions

▶ Task 1: Write a SELECT Statement to Retrieve the Next Row Using a Common Table Expression (CTE)

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the **Task 1** description:

```
WITH OrderRows AS
(
SELECT
orderid,
orderdate,
ROW_NUMBER() OVER (ORDER BY orderdate, orderid) AS rowno,
val
FROM Sales.OrderValues
)
SELECT
o.orderid,
o.orderdate,
o.val,
o2.val as prevval,
o.val - o2.val as diffprev
FROM OrderRows AS o
LEFT OUTER JOIN OrderRows AS o2 ON o.rowno = o2.rowno + 1;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Add a Column to Display the Running Sales Total

1. In the query pane, type the following query after the **Task 2** description:

```
SELECT
orderid,
orderdate,
val,
LAG(val) OVER (ORDER BY orderdate, orderid) AS prevval,
val - LAG(val) OVER (ORDER BY orderdate, orderid) AS diffprev
FROM Sales.OrderValues;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Analyze the Sales Information for the Year 2007

1. In the query pane, type the following query after the **Task 3** description:

```
WITH SalesMonth2007 AS
(
SELECT
MONTH(orderdate) AS monthno,
SUM(val) AS val
FROM Sales.OrderValues
WHERE orderdate >= '20070101' AND orderdate < '20080101'
GROUP BY MONTH(orderdate)
)
SELECT
monthno,
val,
(LAG(val, 1, 0) OVER (ORDER BY monthno) + LAG(val, 2, 0) OVER (ORDER BY monthno) +
LAG(val, 3, 0) OVER (ORDER BY monthno)) / 3 AS avglast3months,
val - FIRST_VALUE(val) OVER (ORDER BY monthno ROWS UNBOUNDED PRECEDING) AS
diffjanuary,
LEAD(val) OVER (ORDER BY monthno) AS nextval
FROM SalesMonth2007;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to use the offset functions in your T-SQL statements.

Exercise 3: Writing Queries That Use Window Aggregate Functions

▶ Task 1: Write a SELECT Statement to Display the Contribution of Each Customer's Order Compared to That Customer's Total Purchase

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the **Task 1** description:

```
SELECT
custid,
orderid,
orderdate,
val,
100. * val / SUM(val) OVER (PARTITION BY custid) AS percoftotalcust
FROM Sales.OrderValues
ORDER BY custid, percoftotalcust DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Add a Column to Display the Running Sales Total

1. Highlight the previous query. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the **Task 2** description. On the toolbar, click **Edit** and then **Paste**.

3. Modify the T-SQL statement by adding an additional calculated column. The query should look like this:

```
SELECT
custid,
orderid,
orderdate,
val,
100. * val / SUM(val) OVER (PARTITION BY custid) AS percoftotalcust,
SUM(val) OVER (PARTITION BY custid
ORDER BY orderdate, orderid
ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS runval
FROM Sales.OrderValues;
```

4. Highlight the written query and click **Execute**.

▶ Task 3: Analyze the Year-to-Date Sales Amount and Average Sales Amount for the Last Three Months

1. In the query pane, type the following query after the **Task 3** description:

```
WITH SalesMonth2007 AS
(
SELECT
MONTH(orderdate) AS monthno,
SUM(val) AS val
FROM Sales.OrderValues
WHERE orderdate >= '20070101' AND orderdate < '20080101'
GROUP BY MONTH(orderdate)
)
SELECT
monthno,
val,
AVG(val) OVER (ORDER BY monthno ROWS BETWEEN 3 PRECEDING AND CURRENT ROW) AS
avglast3months,
SUM(val) OVER (ORDER BY monthno ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS
ytdval
FROM SalesMonth2007;
```

2. Highlight the written query and click **Execute**.
3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have a basic understanding of how to use window aggregate functions in T-SQL statements.

Lab 4: Pivoting and Grouping Sets

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You have been given a set of business requirements for data and you will write T-SQL queries to retrieve the specified data from the databases. The business requests are analytical in nature. To fulfill those requests, you will need to provide crosstab reports and multiple aggregates based on different granularities. Therefore, you will need to use pivoting techniques and grouping sets in your T-SQL code.

Objectives

After completing this lab, you will be able to:

Write queries that use the PIVOT operator.

Write queries that use the UNPIVOT operator.

Write queries that use GROUPING SETS, CUBE, and ROLLUP

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use the PIVOT Operator

Scenario

The sales department would like to have a crosstab report, displaying the number of customers for each customer group and country. They would like to display each customer group as a new column. You will write different SELECT statements using the PIVOT operator to achieve the required result.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve the Number of Customers for a Specific Customer Group
3. Specify the Grouping Element for the PIVOT Operator
4. Use a Common Table Expression (CTE) to Specify the Grouping Element for the PIVOT Operator
5. Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer and Product Category

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab14\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve the Number of Customers for a Specific Customer Group

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab14\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.

2. The IT department has given you T-SQL code to generate a view named Sales.CustGroups, which contains three pieces of information about customers—their IDs, the countries in which they are located, and the customer group in which they have been placed. Customers are placed into one of three predefined groups (A, B, or C).

3. Execute the provided T-SQL code:

```
CREATE VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
country
FROM Sales.Customers;
```

4. Write a SELECT statement that will return the custid, custgroup, and country columns from the newly created Sales.CustGroups view.

5. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\Solution\52 - Lab Exercise 1 - Task 1_1 Result.txt.

6. Modify the SELECT statement. Begin by retrieving the column country then use the PIVOT operator to retrieve three columns based on the possible values of the custgroup column (values A, B, and C), showing the number of customers in each group.

7. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\53 - Lab Exercise 1 - Task 1_2 Result.txt.

▶ Task 3: Specify the Grouping Element for the PIVOT Operator

1. The IT department has provided T-SQL code to add two new columns—city and contactname—to the Sales.CustGroups view. Execute the provided T-SQL code:

```
ALTER VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
country,
city,
contactname
FROM Sales.Customers;
```

2. Copy the last SELECT statement in task 1 and execute it.

3. Is this result the same as that from the query in task 1? Is the number of rows retrieved the same?

4. To better understand the reason for the different results, modify the copied SELECT statement to include the new city and contactname columns.

5. Execute the modified statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\54 - Lab Exercise 1 - Task 2 Result.txt.

6. Notice that this query returned the same number of rows as the previous SELECT statement. Why did you get the same result with and without specifying the grouping columns for the PIVOT operator?

▶ Task 4: Use a Common Table Expression (CTE) to Specify the Grouping Element for the PIVOT Operator

1. Define a CTE named PivotCustGroups based on a query that retrieves the custid, country, and custgroup columns from the Sales.CustGroups view. Write a SELECT statement against the CTE, using a PIVOT operator to retrieve the same result as in task 1.
2. Execute the written T-SQL code and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\55 - Lab Exercise 1 - Task 3 Result.txt.
3. Is this result the same as the one returned by the last query in task 1? Can you explain why?
4. Why do you think it is beneficial to use the CTE when using the PIVOT operator?

▶ Task 5: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer and Product Category

1. For each customer, write a SELECT statement to retrieve the total sales amount for all product categories, displaying each as a separate column. Here is how to accomplish this task:
   - Create a CTE named SalesByCategory to retrieve the custid column from the Sales.Orders table as a calculated column, based on the qty and unitprice columns and the categoryname column from the table Production.Categories. Filter the result to include only orders in the year 2008.
   - You will need to JOIN tables Sales.Orders, Sales.OrderDetails, Production.Products, and Production.Categories.
   - Write a SELECT statement against the CTE that returns a row for each customer (custid) and a column for each product category, with the total sales amount for the current customer andproduct category.
   - Display the following product categories: Beverages, Condiments, Confections, [Dairy Products], [Grains/Cereals], [Meat/Poultry], Produce, and Seafood.
2. Execute the complete T-SQL code (the CTE and the SELECT statement).
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\56 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should be able to use the PIVOT operator in T-SQL statements.

Exercise 2: Writing Queries That Use the UNPIVOT Operator

Scenario

You will now create multiple rows by turning columns into rows.

The main tasks for this exercise are as follows:

1. Create and Query the Sales.PivotCustGroups View
2. Write a SELECT Statement to Retrieve a Row for Each Country and Customer Group
3. Remove the Created Views

▶ Task 1: Create and Query the Sales.PivotCustGroups View

1. Open the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the TSQL database.
2. Execute the provided T-SQL code to generate the Sales.PivotCustGroups view:

```
CREATE VIEW Sales.PivotCustGroups AS
WITH PivotCustGroups AS
(
SELECT
custid,
country,
custgroup
FROM Sales.CustGroups
)
SELECT
country,
p.A,
p.B,
p.C
FROM PivotCustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

3. Write a SELECT statement to retrieve the country, A, B, and C columns from the Sales.PivotCustGroups view.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab14\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve a Row for Each Country and Customer Group

1. Write a SELECT statement against the Sales.PivotCustGroups view that returns the following:
   - A row for each country and customer group.
   - The column country.
   - Two new columns—custgroup and numberofcustomers. The custgroup column should hold the names of the source columns A, B, and C as character strings, and the numberofcustomers column should hold their values (that is, number of customers).
2. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Remove the Created Views

1. Remove the created views by executing the provided T-SQL code:

```
DROP VIEW  Sales.CustGroups;
DROP VIEW Sales.PivotCustGroups;
```

2. Execute this code exactly as written, inside a query window.

Results: After this exercise, you should know how to use the UNPIVOT operator in your T-SQL statements.

Exercise 3: Writing Queries That Use the GROUPING SETS, CUBE, and ROLLUP Subclauses

Scenario

You have to prepare SELECT statements to retrieve a unified result set with aggregated data for different combinations of columns. First, you have to retrieve the number of customers for all possible combinations of the country and city columns. Instead of using multiple T-SQL statements with a GROUP BY clause and then unifying them with the UNION ALL operator, you will use a more elegant solution using the GROUPING SETS subclause of the GROUP BY clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses the GROUPING SETS Subclause to Return the Number of Customers for Different Grouping Sets

2. Write a SELECT Statement That Uses the CUBE Subclause to Retrieve Grouping Sets Based on Yearly, Monthly, and Daily Sales Values

3. Write the Same SELECT Statement Using the ROLLUP Subclause

4. Analyze the Total Sales Value by Year and Month

▶ Task 1: Write a SELECT Statement That Uses the GROUPING SETS Subclause to Return the Number of Customers for Different Grouping Sets

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement against the Sales.Customers table and retrieve the country column, the city column, and a calculated column noofcustomers as a count of customers. Retrieve multiple grouping sets based on the country and city columns, the country column, the city column, and a column withan empty grouping set.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the CUBE Subclause to Retrieve Grouping Sets Based on Yearly, Monthly, and Daily Sales Values

1. Write a SELECT statement against the view Sales.OrderValues and retrieve these columns:
   - Year of the orderdate column as orderyear.
   - Month of the orderdate column as ordermonth.
   - Day of the orderdate column as orderday.
   - Total sales value using the val column as salesvalue.
   - Return all possible grouping sets based on the orderyear, ordermonth, and orderday columns.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\73 - Lab Exercise 3 - Task 2 Result.txt. Notice the total number of rows in your results.

▶ Task 3: Write the Same SELECT Statement Using the ROLLUP Subclause

1. Copy the previous query and modify it to use the ROLLUP subclause instead of the CUBE subclause.
2. Execute the modified query and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\74 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows in your results.

3. What is the difference between the ROLLUP and CUBE subclauses?
4. Which is the more appropriate subclause to use in this example?

▶ Task 4: Analyze the Total Sales Value by Year and Month

1. Write a SELECT statement against the Sales.OrderValues view and retrieve these columns:
   - Calculated column with the alias groupid (use the GROUPING_ID function with the order year and order month as the input parameters).
   - Year of the orderdate column as orderyear.
   - Month of the orderdate column as ordermonth.
   - Total sales value using the val column as salesvalue.
   - Since year and month form a hierarchy, return all interesting grouping sets based on the orderyear and ordermonth columns and sort the result by groupid, orderyear, and ordermonth.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab14\75 - Lab Exercise 3 - Task 4 Result.txt.
3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have an understanding of how to use the GROUPING SETS, CUBE, and ROLLUP subclauses in T-SQL statements.

Lab Answer Key 4: Pivoting and Grouping Sets

Exercise 1: Writing Queries That Use the PIVOT Operator

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab14\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Write a SELECT Statement to Retrieve the Number of Customers for a Specific Customer Group

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab14\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, double-click the query **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
6. Highlight the following provided T-SQL code:

```
CREATE VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
Country
FROM Sales.Customers;
```

7. Click **Execute**. This code creates a view named Sales.CustGroups.
8. In the query pane, type the following query after the provided T-SQL code:

```
SELECT
custid,
custgroup,
country
FROM Sales.CustGroups;
```

9. Highlight the written query and click **Execute**.
10. Modify the written T-SQL code by applying the PIVOT operator. The query should look like this:

```
SELECT
country,
p.A,
p.B,
p.C
FROM Sales.CustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

11. Highlight the written query and click **Execute**.

▶ Task 3: Specify the Grouping Element for the PIVOT Operator

1. Highlight the following provided T-SQL code after the **Task 2** description:

```
ALTER VIEW Sales.CustGroups AS
SELECT
custid,
CHOOSE(custid % 3 + 1, N'A', N'B', N'C') AS custgroup,
country,
city,
contactname
FROM Sales.Customers;
```

2. Click **Execute**. This code modifies the view by adding two additional columns.
3. Highlight the last query in task 1. On the toolbar, click **Edit** and then **Copy**.
4. In the query window, click the line after the provided T-SQL code. On the toolbar, click **Edit** and then **Paste**. The query should look like this:

```
SELECT
country,
p.A,
p.B,
p.C
FROM Sales.CustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

5. Highlight the copied query and click **Execute**.
6. Observe the result. Is this result the same as that from the query in task 1? The result is not the same. More rows were returned after you modified the view.
7. Modify the copied T-SQL statement to include additional columns from the view. The query should look like this:

```
SELECT
country,
city,
contactname,
p.A,
p.B,
p.C
FROM Sales.CustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

8. Highlight the written query and click **Execute**.
9. Notice that you received the same result as the previous query. Why did you get the same number of rows? The PIVOT operator assumes that all the columns except the aggregation and spreading elements are part of the grouping columns.

▶ Task 4: Use a Common Table Expression (CTE) to Specify the Grouping Element for the PIVOT Operator

1. In the query pane, type the following query after the **Task 3** description:

```
WITH PivotCustGroups AS
(
SELECT
custid,
country,
custgroup
FROM Sales.CustGroups
)
SELECT
country,
p.A,
p.B,
p.C
FROM PivotCustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

2. Highlight the written query and click **Execute**.
3. Observe the result. Is it the same as the result of the last query in task 1? Can you explain why? The result is the same. In this task, the CTE has provided three possible columns to the PIVOT operator. In task 1, the view also provided three columns to the PIVOT operator.
4. Why do you think it is beneficial to use a CTE when using the PIVOT operator? When using the PIVOT operator, you cannot directly specify the grouping element because SQL Server automatically assumes that all columns should be used as grouping elements, with the exception of the spreading and aggregation elements. With a CTE, you can specify the exact columns and therefore control that columns use for the grouping.

▶ Task 5: Write a SELECT Statement to Retrieve the Total Sales Amount for Each Customer and Product Category

1. In the query pane, type the following query after the **Task 4** description:

```
WITH SalesByCategory AS
(
SELECT
o.custid,
d.qty * d.unitprice AS salesvalue,
c.categoryname
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON o.orderid = d.orderid
INNER JOIN Production.Products AS p ON p.productid = d.productid
INNER JOIN Production.Categories AS c ON c.categoryid = p.categoryid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
)
SELECT
custid,
p.Beverages,
p.Condiments,
p.Confections,
p.[Dairy Products],
p.[Grains/Cereals],
p.[Meat/Poultry],
p.Produce,
p.Seafood
FROM SalesByCategory
PIVOT (SUM(salesvalue) FOR categoryname
```

```
IN (Beverages, Condiments, Confections, [Dairy Products], [Grains/Cereals],
[Meat/Poultry], Produce, Seafood)) AS p;
```

2. Highlight the written query and click **Execute**.

Results: After this exercise, you should be able to use the PIVOT operator in T-SQL statements.

Exercise 2: Writing Queries That Use the UNPIVOT Operator

▶ Task 1: Create and Query the Sales.PivotCustGroups View

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the following provided T-SQL code:

```
CREATE VIEW Sales.PivotCustGroups AS
WITH PivotCustGroups AS
(
SELECT
custid,
country,
custgroup
FROM Sales.CustGroups
)
SELECT
country,
p.A,
p.B,
p.C
FROM PivotCustGroups
PIVOT (COUNT(custid) FOR custgroup IN (A, B, C)) AS p;
```

4. Click **Execute**. This code creates a view named Sales.PivotCustGroups.
5. In the query pane, type the following query after the provided T-SQL code:

```
SELECT
country, A, B, C
FROM Sales.PivotCustGroups;
```

6. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement to Retrieve a Row for Each Country and Customer Group

1. In the query pane, type the following query after the **Task 2** descriptions:

```
SELECT
custgroup,
country,
numberofcustomers
FROM Sales.PivotCustGroups
UNPIVOT (numberofcustomers FOR custgroup IN (A, B, C)) AS p;
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Remove the Created Views

- Highlight the provided T-SQL statement after the **Task 3** description and click **Execute**.

Results: After this exercise, you should know how to use the UNPIVOT operator in your T-SQL statements.

Exercise 3: Writing Queries That Use the GROUPING SETS, CUBE, and ROLLUP Subclauses

▶ Task 1: Write a SELECT Statement That Uses the GROUPING SETS Subclause to Return the Number of Customers for Different Grouping Sets

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following query after the **Task 1** description:

```
SELECT
country,
city,
COUNT(custid) AS noofcustomers
FROM Sales.Customers
GROUP BY
GROUPING SETS
(
(country, city),
(country),
(city),
()
);
```

4. Highlight the written query and click **Execute**.

▶ Task 2: Write a SELECT Statement That Uses the CUBE Subclause to Retrieve Grouping Sets Based on Yearly, Monthly, and Daily Sales Values

1. In the query pane, type the following query after the **Task 2** description:

```
SELECT
YEAR(orderdate) AS orderyear,
MONTH(orderdate) AS ordermonth,
DAY(orderdate) AS orderday,
SUM(val) AS salesvalue
FROM Sales.OrderValues
GROUP BY
CUBE (YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Write the Same SELECT Statement Using the ROLLUP Subclause

1. In the query pane, type the following query after the **Task 3** description:

```
SELECT
YEAR(orderdate) AS orderyear,
MONTH(orderdate) AS ordermonth,
DAY(orderdate) AS orderday,
SUM(val) AS salesvalue
FROM Sales.OrderValues
GROUP BY
ROLLUP (YEAR(orderdate), MONTH(orderdate), DAY(orderdate));
```

2. Highlight the written query and click **Execute**.

3. Observe the result. What is the difference between the ROLLUP and CUBE subclauses of the GROUP BY clause? Like the CUBE subclause, the ROLLUP subclause provides an abbreviated way to define multiple grouping sets. However, unlike CUBE, ROLLUP doesn't produce all possible grouping sets that can be defined based on the input members—it produces a subset of those. ROLLUP assumes a hierarchy among the input members and produces all grouping sets that make sense, considering the hierarchy. In other words, while CUBE(a, b, c) produces all eight possible grouping sets out of the three input members, ROLLUP(a, b, c) produces only four grouping sets, assuming the hierarchy a>b>c. ROLLUP(a, b, c) is the equivalent of specifying GROUPING SETS((a, b, c), (a, b), (a), ()).

   Which is the more appropriate subclause to use in this example? Since year, month, and day form a hierarchy, the ROLLUP clause is more suitable. There is probably not much interest in showing aggregates for a month irrespective of year, but the other way around is interesting.

▶ Task 4: Analyze the Total Sales Value by Year and Month

1. In the query pane, type the following query after the **Task 4** description:

```
SELECT
GROUPING_ID(YEAR(orderdate), MONTH(orderdate)) as groupid,
YEAR(orderdate) AS orderyear,
MONTH(orderdate) AS ordermonth,
SUM(val) AS salesvalue
FROM Sales.OrderValues
GROUP BY
ROLLUP (YEAR(orderdate), MONTH(orderdate))
ORDER BY groupid, orderyear, ordermonth;
```

2. Highlight the written query and click **Execute**.

3. Close SQL Server Management Studio without saving any changes.

Results: After this exercise, you should have an understanding of how to use the GROUPING SETS, CUBE, and ROLLUP subclauses in T-SQL statements.

Lab 5: Executing Stored Procedures

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You have been given a set of business requirements for data and will write T-SQL queries to retrieve the specified data from the databases. You have learned that some of the data can only be accessed via stored procedures instead of directly querying the tables. Additionally, some of the procedures require parameters in order to interact with them.

Objectives

After completing this lab, you will be able to:

- Use the EXECUTE statement to invoke stored procedures.
- Pass parameters to stored procedures.
- Execute system stored procedures.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Using the EXECUTE Statement to Invoke Stored Procedures

Scenario

The IT department has supplied T-SQL code to create a stored procedure to retrieve the top 10 customers by the total sales amount. You will practice how to execute a stored procedure.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Create and Execute a Stored Procedure
3. Modify the Stored Procedure and Execute It

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab15\Starter** folder as Administrator.

▶ Task 2: Create and Execute a Stored Procedure

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab15\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.

2. Execute the provided T-SQL code to create the stored procedure Sales.GetTopCustomers:

```
CREATE PROCEDURE Sales.GetTopCustomers AS
SELECT TOP(10)
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC;
```

3. Write a T-SQL statement to execute the created procedure.
4. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.
5. What is the difference between the previous T-SQL code and this one?
6. If some applications are using the stored procedure from task 1, would they still work properly after the changes you have applied in task 2?

▶ Task 3: Modify the Stored Procedure and Execute It

1. The IT department has changed the stored procedure from task 1 and supplied you with T-SQL code to apply the required changes. Execute the provided T-SQL code:

```
ALTER PROCEDURE Sales.GetTopCustomers AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Write a T-SQL statement to execute the modified stored procedure.
3. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

Results: After this exercise, you should be able to invoke a stored procedure using the EXECUTE statement.

Exercise 2: Passing Parameters to Stored Procedures

Scenario

The IT department supplied you with additional modifications of the stored procedure in task 1. The modified stored procedure lets you pass parameters that specify the order year and number of customers to retrieve. You will practice how to execute the stored procedure with a parameter.

The main tasks for this exercise are as follows:

1. Execute a Stored Procedure with a Parameter for Order Year
2. Modify the Stored Procedure to Have a Default Value for the Parameter
3. Pass Multiple Parameters to the Stored Procedure
4. Return the Result from a Stored Procedure Using the OUTPUT Clause

▶ Task 1: Execute a Stored Procedure with a Parameter for Order Year

1. Open the SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the TSQL database.
2. Execute the provided T-SQL code to modify the Sales.GetTopCustomers stored procedure to include a parameter for order year (@orderyear):

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

3. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for the year 2007.
4. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\62 - Lab Exercise 2 - Task 1_1 Result.txt.
5. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for the year 2008.
6. Execute the T-SQL statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab15\Solution\63 - Lab Exercise 2 - Task 1_2 Result.txt.
7. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure without a parameter.
8. Execute the T-SQL statement. What happened? What is the error message?
9. If an application was designed to use the exercise 1 version of the stored procedure, would the modification made to the stored procedure in this exercise impact the usability of that application? Please explain.

▶ Task 2: Modify the Stored Procedure to Have a Default Value for the Parameter

1. Execute the provided T-SQL code to modify the Sales.GetTopCustomers stored procedure:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure without a parameter.

3. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\64 - Lab Exercise 2 - Task 2 Result.txt.

4. If an application was designed to use the Exercise 1 version of the stored procedure, would the change made to the stored procedure in this task impact the usability of that application? How does this change influence the design of future applications?

▶ Task 3: Pass Multiple Parameters to the Stored Procedure

1. Execute the provided T-SQL code to add the parameter @n to the Sales.GetTopCustomers stored procedure. You use this parameter to specify how many customers you want retrieved. The default value is 10.

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL,
@n int = 10
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT @n ROWS ONLY;
```

2. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure without any parameters.

3. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\65 - Lab Exercise 2 - Task 3_1 Result.txt.

4. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for order year 2008 and five customers.

5. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\66 - Lab Exercise 2 - Task 3_2 Result.txt.

6. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure for the order year 2007.

7. Execute the T-SQL statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab15\Solution\67 - Lab Exercise 2 - Task 3_3 Result.txt.
8. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure to retrieve 20 customers.
9. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\68 - Lab Exercise 2 - Task 3_4 Result.txt.
10. Do the applications using the stored procedure need to be changed because another parameter was added?

▶ Task 4: Return the Result from a Stored Procedure Using the OUTPUT Clause

1. Execute the provided T-SQL code to modify the Sales.GetTopCustomers stored procedure to return the customer contact name based on a specified position in a ranking of total sales, which is provided by the parameter @customerpos. The procedure also includes a new parameter named @customername, which has an OUTPUT option:

```
ALTER PROCEDURE Sales.GetTopCustomers
@customerpos int = 1,
@customername nvarchar(30) OUTPUT
AS
SET @customername = (
SELECT
c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY SUM(o.val) DESC
OFFSET @customerpos - 1 ROWS FETCH NEXT 1 ROW ONLY
);
```

2. The IT department also supplied you with T-SQL code to declare the new variable @outcustomername. You will use this variable as an output parameter for the stored procedure.
3. DECLARE @outcustomername nvarchar(30);
4. Write an EXECUTE statement to invoke the Sales.GetTopCustomers stored procedure and retrieve the first customer.
5. Write a SELECT statement to retrieve the value of the output parameter @outcustomername.
6. Execute the batch of T-SQL code consisting of the provided DECLARE statement, the written EXECUTE statement, and the written SELECT statement.
7. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\69 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should know how to invoke stored procedures that have parameters.

Exercise 3: Executing System Stored Procedures

Scenario

In the previous module, you learned how to query the system catalog. Now you will practice how to execute some of the most commonly used system-stored procedures to retrieve information about tables and columns.

The main tasks for this exercise are as follows:

1. Execute the Stored Procedure sys.sp_help
2. Execute the Stored Procedure sys.sp_helptext
3. Execute the Stored Procedure sys.sp_columns
4. Drop the Created Stored Procedure

▶ Task 1: Execute the Stored Procedure sys.sp_help

1. Open the SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write an EXECUTE statement to invoke the sys.sp_help stored procedure without a parameter.
3. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\72 - Lab Exercise 3 - Task 1_1 Result.txt.
4. Write an EXECUTE statement to invoke the sys.sp_help stored procedure for a specific table by passing the parameter Sales.Customers.
5. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\73 - Lab Exercise 3 - Task 1_2 Result.txt.

▶ Task 2: Execute the Stored Procedure sys.sp_helptext

1. Write an EXECUTE statement to invoke the sys.sp_helptext stored procedure, passing the Sales.GetTopCustomers stored procedure as a parameter.
2. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\74 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Execute the Stored Procedure sys.sp_columns

1. Write an EXECUTE statement to invoke the sys.sp_columns stored procedure for the table Sales.Customers. You will have to pass two parameters: @table_name and @table_owner.
2. Execute the T-SQL statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab15\Solution\75 - Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Drop the Created Stored Procedure

- Execute the provided T-SQL statement to remove the Sales.GetTopCustomers stored procedure:

```
DROP PROCEDURE Sales.GetTopCustomers;
```

Results: After this exercise, you should have a basic knowledge of invoking different system-stored procedures.

Lab Answer Key 5: Executing Stored Procedures

Exercise 1: Using the EXECUTE Statement to Invoke Stored Procedures

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab15\Starter** folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish.

▶ Task 2: Create and Execute a Stored Procedure

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab15\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute** on the toolbar.
6. Highlight the following T-SQL code under the **Task 1** description:

```
CREATE PROCEDURE Sales.GetTopCustomers AS
SELECT TOP(10)
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC;
```

7. Click **Execute**. You have created a stored procedure named Sales.GetTopCustomers.
8. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXECUTE Sales.GetTopCustomers;
```

9. Highlight the written T-SQL code and click **Execute**. You have executed the stored procedure.

▶ Task 3: Modify the Stored Procedure and Execute It

1. Highlight the following T-SQL code after the **Task 2** description:

```
ALTER PROCEDURE Sales.GetTopCustomers AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure.
3. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXECUTE Sales.GetTopCustomers;
```

4. Highlight the written T-SQL code and click **Execute**. You have executed the modified stored procedure.
5. Compare both the code and the result of the two versions of the stored procedure. What is the difference between them? In the modified version, the TOP option has been replaced with theOFFSET-FETCH option. Despite this change, the result is the same.

   If some applications had been using the stored procedure in task 1, would they still work properly after the change you applied in task 2? Yes, since the result from the stored procedure is still the same. This demonstrates the huge benefit of using stored procedures as an additional layer between the database and the application/middle tier. Even if you change the underlying T-SQL code, the application would work properly without any changes. There are also other benefits of using stored procedures in terms of performance (for example, caching and reuse of plans) and security (for example, preventing SQL injections).

Results: After this exercise, you should be able to invoke a stored procedure using the EXECUTE statement.

Exercise 2: Passing Parameters to Stored Procedures

▶ Task 1: Execute a Stored Procedure with a Parameter for Order Year

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight the following T-SQL code under the **Task 1** description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

4. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure to accept the parameter @orderyear. Notice that the modified stored procedure uses a predicate in the WHERE clause that isn't a search argument. This predicate was used to keep things simple. The best practice is to avoid such filtering because it does not allow efficient use of indexing. A better approach would be to use the DATETIMEFROMPARTS function to provide a search argument for orderdate:

```
WHERE o.orderdate >= DATETIMEFROMPARTS(@orderyear, 1, 1, 0, 0, 0, 0)
AND o.orderdate < DATETIMEFROMPARTS(@orderyear + 1, 1, 1, 0, 0, 0, 0)
```

5. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2007;
```

Notice that you are passing the parameter by name as this is considered the best practice. There is also support for passing parameters by position. For example, the following EXECUTE statement would retrieve the same result as the T-SQL code you just typed:

```
EXECUTE Sales.GetTopCustomers 2007;
```

6. Highlight the written T-SQL code and click **Execute**.
7. After the previous T-SQL code, type the following T-SQL code to execute the stored procedure for the order year 2008:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2008;
```

8. Highlight the written T-SQL code and click **Execute**.
9. After the previous T-SQL code, type the following T-SQL code to execute the stored procedure without specifying a parameter:

```
EXECUTE Sales.GetTopCustomers;
```

10. Highlight the written T-SQL code and click **Execute**.
11. Observe the error message:

Procedure or function 'GetTopCustomers' expects parameter '@orderyear', which was not supplied. This error message is telling you that the @orderyear parameter was not supplied.

12. Suppose that an application named MyCustomers is using the exercise 1 version of the stored procedure. Would the modification made to the stored procedure in this exercise impact the usability of the GetCustomerInfo application? Yes. The exercise 1 version of the stored procedure did not need a parameter, whereas the version in this exercise does not work without a parameter. To avoid problems, you can add a default parameter to the stored procedure. That way, the MyCustomers application does not have to be changed to support the @orderyear parameter.

▶ Task 2: Modify the Stored Procedure to Have a Default Value for the Parameter

1. Highlight the following T-SQL code under the **Task 2** description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

2. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure to have a default value (NULL) for the @orderyear parameter. You have also included an additional logical expression to the WHERE clause.

3. In the query pane, type the following T-SQL code after the previous one:

```
EXECUTE Sales.GetTopCustomers;
```

This code tests the modified stored procedure by executing it without specifying a parameter.

4. Highlight the written query and click **Execute**.
5. Observe the result. How do the changes to the stored procedure in task 2 influence the MyCustomers application and the design of future applications? The changes enable the MyCustomers application to use the modified stored procedure, and no changes need to be made to the application. The changes add new possibilities for future applications because the modified stored procedure accepts the order year as a parameter.

▶ Task 3: Pass Multiple Parameters to the Stored Procedure

1. Highlight the following T-SQL code under the **Task 3** description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@orderyear int = NULL,
@n int = 10
AS
SELECT
c.custid,
c.contactname,
SUM(o.val) AS salesvalue
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE YEAR(o.orderdate) = @orderyear OR @orderyear IS NULL
GROUP BY c.custid, c.contactname
ORDER BY salesvalue DESC
OFFSET 0 ROWS FETCH NEXT @n ROWS ONLY;
```

2. Click **Execute**. You have modified the Sales.GetTopCustomers stored procedure to have an additional parameter named @n. You can use this parameter to specify how many customers to retrieve. The default value is 10.
3. After the previous T-SQL code, type the following T-SQL code to execute the modified stored procedure:

```
EXECUTE Sales.GetTopCustomers;
```

4. Highlight the written query and click **Execute**.
5. After the previous T-SQL code, type the following T-SQL code to retrieve the top five customers for the year 2008:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2008, @n = 5;
```

6. Highlight the written query and click **Execute**.
7. After the previous T-SQL code, type the following T-SQL code to retrieve the top 10 customers for the year 2007:

```
EXECUTE Sales.GetTopCustomers @orderyear = 2007;
```

8. Highlight the written query and click **Execute**.
9. After the previous T-SQL code, type the following T-SQL code to retrieve the top 20 customers:

```
EXECUTE Sales.GetTopCustomers @n = 20;
```

10. Highlight the written query and click **Execute**.
11. Do the applications using the stored procedure need to be changed because another parameter was added? No changes need to be made to the application.

▶ Task 4: Return the Result from a Stored Procedure Using the OUTPUT Clause

1. Highlight the following T-SQL code under the **Task 4** description:

```
ALTER PROCEDURE Sales.GetTopCustomers
@customerpos int = 1,
@customername nvarchar(30) OUTPUT
AS
SET @customername = (
SELECT
c.contactname
FROM Sales.OrderValues AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
GROUP BY c.custid, c.contactname
ORDER BY SUM(o.val) DESC
OFFSET @customerpos - 1 ROWS FETCH NEXT 1 ROW ONLY
);
```

2. Click **Execute**.
3. Find the following DECLARE statement in the provided code:

```
DECLARE @outcustomername nvarchar(30);
```

   This statement declares a parameter named @outcustomername.

4. After the DECLARE statement, add code that uses the OUTPUT clause to return the stored procedure's result as a variable named @outcustomername. Your code, together with the provided DECLARE statement, should look like this:

```
DECLARE @outcustomername nvarchar(30);
EXECUTE Sales.GetTopCustomers @customerpos = 1, @customername = @outcustomername
OUTPUT;
SELECT @outcustomername AS customername;
```

5. Highlight all three T-SQL statements and click **Execute**.

Results: After this exercise, you should know how to invoke stored procedures that have parameters.

Exercise 3: Executing System Stored Procedures

▶ Task 1: Execute the Stored Procedure sys.sp_help

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the **Task 1** description:

```
EXEC sys.sp_help;
```

4. Highlight the written query and click **Execute**.
5. In the query pane, type the following T-SQL code after the previous T-SQL code:

```
EXEC sys.sp_help N'Sales.Customers';
```

6. Highlight the written query and click **Execute**.

▶ Task 2: Execute the Stored Procedure sys.sp_helptext

1. In the query pane, type the following T-SQL code after the **Task 2** description:

```
EXEC sys.sp_helptext N'Sales.GetTopCustomers';
```

2. Highlight the written query and click **Execute**.

▶ Task 3: Execute the Stored Procedure sys.sp_columns

1. In the query pane, type the following T-SQL code after the **Task 3** description:

```
EXEC sys.sp_columns @table_name = N'Customers', @table_owner = N'Sales';
```

2. Highlight the written query and click **Execute**.

▶ Task 4: Drop the Created Stored Procedure

- Highlight the provided T-SQL statement under the **Task 4** description and click **Execute**.

Results: After this exercise, you should have a basic knowledge of invoking different system-stored procedures.

Lab 6: Programming with T-SQL

Scenario

As a junior database developer for Adventure Works, you have so far focused on writing reports using corporate databases stored in SQL Server. To prepare for upcoming tasks, you will be working with some basic T-SQL programming objects.

Objectives

After completing this lab, you will be able to:

- Declare variables and delimit batches.
- Use control of flow elements.
- Use variables with a dynamic SQL statement.
- Use synonyms.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Declaring Variables and Delimiting Batches

Scenario

You will practice how to declare variables, retrieve their values, and use them in a SELECT statement to return specific employee information.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Declare a Variable and Retrieve the Value
3. Set the Variable Value Using a SELECT Statement
4. Use a Variable in the WHERE Clause
5. Use Variables with Batches

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab16\Starter** folder as Administrator.

▶ Task 2: Declare a Variable and Retrieve the Value

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab16\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.
2. Write T-SQL code that will create a variable called @num as an int data type. Set the value of the variable to 5 and display it using the alias mynumber. Execute the T-SQL code.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\52 - Lab Exercise 1 - Task 1_1 Result.txt.

4. Write the batch delimiter GO after the written T-SQL code. In addition, write new T-SQL code that defines two variables, @num1 and @num2, both as an int data type. Set the values to 4 and 6 respectively. Write a SELECT statement to retrieve the sum of both variables using the alias totalnum. Execute the T-SQL code.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\53 - Lab Exercise 1 - Task 1_2 Result.txt.

▶ Task 3: Set the Variable Value Using a SELECT Statement

1. Write T-SQL code that defines the variable @empname as an nvarchar(30) data type.
2. Set the value by executing a SELECT statement against the HR.Employees table. Compute a value that concatenates the firstname and lastname column values. Add a space between the two column values and filter the results to return the employee whose empid value is equal to 1.
3. Return the @empname variable's value using the alias employee.
4. Execute the T-SQL code.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\54 - Lab Exercise 1 - Task 2Result.txt.
6. What would happen if the SELECT statement was returning more than one row?

▶ Task 4: Use a Variable in the WHERE Clause

1. Copy the T-SQL code from task 2 and modify it by defining an additional variable named @empid with an int data type. Set the variable's value to 5. In the WHERE clause, modify the SELECT statement to use the newly-created variable as a value for the column empid.
2. Execute the modified T-SQL code.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\55 - Lab Exercise 1 - Task 3 Result.txt.
4. Change the @empid variable's value from 5 to 2 and execute the modified T-SQL code to observe the changes.

▶ Task 5: Use Variables with Batches

1. Copy the T-SQL code from task 3 and modify it by adding the batch delimiter GO before the statement:

```
SELECT @empname AS employee;
```

2. Execute the modified T-SQL code.
3. What happened? What is the error message? Can you explain why the batch delimiter caused an error?

Results: After this exercise, you should know how to declare and use variables in T-SQL code.

Exercise 2: Using Control-of-Flow Elements

Scenario

You would like to include conditional logic in your T-SQL code to control the flow of elements by setting different values to a variable using the IF statement.

The main tasks for this exercise are as follows:

1. Write Basic Conditional Logic
2. Check the Employee Birthdate
3. Create and Execute a Stored Procedure
4. Execute a Loop Using the WHILE Statement
5. Remove the Stored Procedure

▶ Task 1: Write Basic Conditional Logic

1. Open the SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the TSQL database.
2. Write T-SQL code that defines the variable @result as an nvarchar(20) data type and the variable @i as an int data type. Set the value of the @i variable to 8. Write an IF statement that implements the following logic:
   - For @i variable values less than 5, set the value of the @result variable to "Less than 5".
   - For @i variable values between 5 and 10, set the value of the @result variable to "Between 5 and 10".
   - For all @i variable values over 10, set the value of the @result variable to "More than 10".
   - For other @i variable values, set the value of the @result variable to "Unknown".
3. At the end of the T-SQL code, write a SELECT statement to retrieve the value of the @result variable using the alias result. Highlight the complete T-SQL code and execute it.
4. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.
5. Copy the T-SQL code and modify it by replacing the IF statement with a CASE expression to get the same result.

▶ Task 2: Check the Employee Birthdate

1. Write T-SQL code that declares two variables: @birthdate (data type date) and @cmpdate (data type date).
2. Set the value of the @birthdate variable by writing a SELECT statement against the HR.Employees table and retrieving the column birthdate. Filter the results to include only the employee with anempid equal to 5.
3. Set the @cmpdate variable to the value January 1, 1970.
4. Write an IF conditional statement by comparing the @birthdate and @cmpdate variable values. If @birthdate is less than @cmpdate, use the PRINT statement to print the message "The person selected was born before January 1, 1970". Otherwise, print the message "The person selected was born on or after January 1, 1970".
5. Execute the T-SQL code.

6. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. This is a simple example for the purpose of this exercise. Typically, a different statement block would execute in each case.

▶ Task 3: Create and Execute a Stored Procedure

1. The IT department has provided T-SQL code that encapsulates the previous task in a stored procedure named Sales.CheckPersonBirthDate. It has two parameters: @empid, which you use to specify an employee id, and @cmpdate, which you use as a comparison date. Execute the provided T-SQL code:

```
CREATE PROCEDURE Sales.CheckPersonBirthDate
@empid int,
@cmpdate date
AS
DECLARE
@birthdate date;
SET @birthdate = (SELECT birthdate FROM HR.Employees WHERE empid = @empid);
IF @birthdate < @cmpdate
PRINT 'The person selected was born before ' + FORMAT(@cmpdate, 'MMMM d, yyyy', 'en-
US')
ELSE
PRINT 'The person selected was born on or after ' + FORMAT(@cmpdate, 'MMMM d, yyyy',
'en-US');
```

2. Write an EXECUTE statement to invoke the Sales.CheckPersonBirthDate stored procedure using the parameters of 3 for @empid and January 1, 1990, for @cmpdate. Execute the T-SQL code.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

▶ Task 4: Execute a Loop Using the WHILE Statement

1. Write T-SQL code to loop 10 times, displaying the current loop information on each occasion.
2. Define the @i variable as an int data type. Write a WHILE statement to execute while the @i variable value is less than or equal to 10. Inside the loop statement, write a PRINT statement to display the value of the @i variable using the alias loopid. Add T-SQL code to increment the @i variable value by 1.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.

▶ Task 5: Remove the Stored Procedure

- Execute the provided T-SQL code under the task 5 description to remove the created stored procedure.

Results: After this exercise, you should know how to control the flow of the elements inside the T-SQL code.

Exercise 3: Using Variables in a Dynamic SQL Statement

Scenario

You will practice how to invoke dynamic SQL code and how to pass variables to it.

The main tasks for this exercise are as follows:

1. Write a Dynamic SQL Statement That Does Not Use a Parameter

2. Write a Dynamic SQL Statement That Uses a Parameter

▶ Task 1: Write a Dynamic SQL Statement That Does Not Use a Parameter

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. Write T-SQL code that defines the variable @SQLstr as nvarchar(200) data type. Set the value of the variable to a SELECT statement that retrieves the empid, firstname, and lastname columns in the HR.Employees table.
3. Write an EXECUTE statement to invoke the written dynamic SQL statement inside the @SQLstr variable. Execute the T-SQL code.
4. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a Dynamic SQL Statement That Uses a Parameter

1. Copy the previous T-SQL code and modify it to include in the dynamic batch stored in @SQLstr, a filter in which empid is equal to a parameter named @empid. In the calling batch, define a variable named @SQLparam as nvarchar(100). This variable will hold the definition of the @empid parameter. This means setting the value of the @SQLparam variable to @empid int.
2. Write an EXECUTE statement that uses sp_executesql to invoke the code in the @SQLstr variable, passing the parameter definition stored in the @SQLparam variable to sp_executesql. Assign the value 5 to the @empid parameter in the current execution.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

Results: After this exercise, you should have a basic knowledge of generating and invoking dynamic SQL statements.

Exercise 4: Using Synonyms

Scenario

You will practice how to create a synonym for a table inside the AdventureWorks2008R2 database and how to write a query against it.

The main tasks for this exercise are as follows:

1. Create and Use a Synonym for a Table

2. Drop the Synonym

▶ Task 1: Create and Use a Synonym for a Table

1. Open the T-SQL script **81 - Lab Exercise 4.sql**. Ensure that you are connected to the TSQL database.

2. Write T-SQL code to create a synonym named dbo.Person for the Person.Person table in the AdventureWorks database. Execute the written statement.
3. Write a SELECT statement against the dbo.Person synonym and retrieve the FirstName and LastName columns. Execute the SELECT statement.
4. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab16\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Drop the Synonym

- Execute the provided T-SQL code under the task 2 description to remove the synonym.

Results: After this exercise, you should know how to create and use a synonym.

Lab Answer Key 6: Programming with T-SQL

Exercise 1: Declaring Variables and Delimiting Batches

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab16\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. Wait for the script to finish then press any key to continue

▶ Task 2: Declare a Variable and Retrieve the Value

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab16\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and then double-click the query **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
6. In the query pane, type the following T-SQL code after the **Task 1** description:

```
DECLARE @num int = 5;
SELECT @num AS mynumber;
```

7. Highlight the written T-SQL code and click **Execute**.
8. In the query pane, type the following T-SQL code after the previous one:

```
DECLARE
@num1 int,
@num2 int;
SET @num1 = 4;
SET @num2 = 6;
SELECT @num1 + @num2 AS totalnum;
```

9. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Set the Variable Value Using a SELECT Statement

1. In the query pane, type the following T-SQL code after the **Task 2** description:

```
DECLARE @empname nvarchar(30);
SET @empname = (SELECT firstname + N' ' + lastname FROM HR.Employees WHERE empid =
1);
SELECT @empname AS employee;
```

2. Highlight the written T-SQL code and click **Execute**.

3. Observe the result. What would happen if the SELECT statement was returning more than one row?You would get an error because the SET statement requires you to use a scalar subquery to pull data from a table. Remember that a scalar subquery fails at runtime if it returns more than one value.

▶ Task 4: Use a Variable in the WHERE Clause

1. In the query pane, type the following T-SQL code after the **Task 3** description:

```
DECLARE
@empname nvarchar(30),
@empid int;
SET @empid = 5;
SET @empname = (SELECT firstname + N' ' + lastname FROM HR.Employees WHERE empid =
@empid);
SELECT @empname AS employee;
```

2. Highlight the written T-SQL code and click **Execute**.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab16\Solution\55 - Lab Exercise 1 - Task 3 Result.txt.
4. Change the @empid variable's value from 5 to 2 and execute the modified T-SQL code to observe the changes.

▶ Task 5: Use Variables with Batches

1. Highlight the T-SQL code in **Task 3**. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the **Task 4** description. On the toolbar, click **Edit** and then **Paste**.
3. In the code you just copied, add the batch delimiter GO before this statement:

```
SELECT @empname AS employee;
```

4. Make sure your T-SQL code looks like this:

```
DECLARE
@empname nvarchar(30),
@empid int;
SET @empid = 5;
SET @empname = (SELECT firstname + N' ' + lastname FROM HR.Employees WHERE empid =
@empid)
GO
SELECT @empname AS employee;
```

5. Highlight the written T-SQL code and click **Execute**.
6. Observe the error:

   Must declare the scalar variable "@empname".
   Can you explain why the batch delimiter caused an error? Variables are local to the batch in which they are defined. If you try to refer to a variable that was defined in another batch, you get an error saying that the variable was not defined. Also, keep in mind that GO is a client command, not a server T-SQL command.

Results: After this exercise, you should know how to declare and use variables in T-SQL code.

Exercise 2: Using Control-of-Flow Elements

▶ Task 1: Write Basic Conditional Logic

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the **Task 1** description:

```
DECLARE
@i int = 8,
@result nvarchar(20);
IF @i < 5
SET @result = N'Less than 5'
ELSE IF @i <= 10
SET @result = N'Between 5 and 10'
ELSE if @i > 10
SET @result = N'More than 10'
ELSE
SET @result = N'Unknown';
SELECT @result AS result;
```

4. Highlight the written T-SQL code and click **Execute**.
5. In the query pane, type the following T-SQL code:

```
DECLARE
@i int = 8,
@result nvarchar(20);
SET @result =
CASE
WHEN @i < 5 THEN
N'Less than 5'
WHEN @i <= 10 THEN
N'Between 5 and 10'
WHEN @i > 10 THEN
N'More than 10'
ELSE
N'Unknown'
END;
SELECT @result AS result;
```

   This code uses a CASE expression and only one SET expression to get the same result as the previous T-SQL code. Remember to use a CASE expression when it is a matter of returning an expression. However, if you need to execute multiple statements, you cannot replace IF with CASE.

6. Highlight the written T-SQL code and click **Execute**.

▶ Task 2: Check the Employee Birthdate

1. In the query pane, type the following T-SQL code after the **Task 2** description:

```
DECLARE
@birthdate date,
@cmpdate date;
SET @birthdate = (SELECT birthdate FROM HR.Employees WHERE empid = 5);
SET @cmpdate = '19700101';
IF @birthdate < @cmpdate
PRINT 'The person selected was born before January 1, 1970'
ELSE
PRINT 'The person selected was born on or after January 1, 1970';
```

2. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Create and Execute a Stored Procedure

1. Highlight the following T-SQL code under the **Task 3** description:

```
CREATE PROCEDURE Sales.CheckPersonBirthDate
@empid int,
@cmpdate date
AS
DECLARE
@birthdate date;
SET @birthdate = (SELECT birthdate FROM HR.Employees WHERE empid = @empid);
IF @birthdate < @cmpdate
PRINT 'The person selected was born before ' + FORMAT(@cmpdate, 'MMMM d, yyyy', 'en-
US');
ELSE
PRINT 'The person selected was born on or after ' + FORMAT(@cmpdate, 'MMMM d, yyyy',
'en-US');
```

2. Click **Execute**. You have created a stored procedure named Sales.CheckPersonBirthDate. It has two parameters: @empid, which you use to specify an employee ID, and @cmpdate, which you use as a comparison date.

3. In the query pane, type the following T-SQL code after the provided T-SQL code:

```
EXECUTE Sales.CheckPersonBirthDate @empid = 3, @cmpdate = '19900101';
```

4. Highlight the written T-SQL code and click **Execute**.

▶ Task 4: Execute a Loop Using the WHILE Statement

1. In the query pane, type the following T-SQL code after the **Task 4** description:

```
DECLARE @i int = 1;
WHILE @i <= 10
BEGIN
PRINT @i;
SET @i = @i + 1;
END;
```

2. Highlight the written T-SQL code and click **Execute**.

▶ Task 5: Remove the Stored Procedure

1. Highlight the following T-SQL code under the **Task 5** description:

```
DROP PROCEDURE Sales.CheckPersonBirthDate;
```

2. Click **Execute**.

Results: After this exercise, you should know how to control the flow of the elements inside the T-SQL code.

Exercise 3: Using Variables in a Dynamic SQL Statement

▶ Task 1: Write a Dynamic SQL Statement That Does Not Use a Parameter

1. In Solution Explorer, double-click the query **71 - Lab Exercise 3.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the **Task 1** description:

```
DECLARE @SQLstr nvarchar(200);
SET @SQLstr = N'SELECT empid, firstname, lastname FROM HR.Employees';
EXECUTE sys.sp_executesql @statement = @SQLstr;
```

4. Highlight the written T-SQL code and click **Execute**.

▶ Task 2: Write a Dynamic SQL Statement That Uses a Parameter

1. Highlight the T-SQL code in **Task 1**. On the toolbar, click **Edit** and then **Copy**.
2. In the query window, click the line after the **Task 2** description. On the toolbar, click **Edit** and then **Paste**.
3. Modify the T-SQL code to look like this:

```
DECLARE
@SQLstr nvarchar(200),
@SQLparam nvarchar(100);
SET @SQLstr = N'SELECT empid, firstname, lastname FROM HR.Employees WHERE empid =
@empid';
SET @SQLparam = N'@empid int';
EXECUTE sys.sp_executesql @statement = @SQLstr, @params = @SQLparam, @empid = 5;
```

4. Highlight the written T-SQL code and click **Execute**.

Results: After this exercise, you should have a basic knowledge of generating and invoking dynamic SQL statements.

Exercise 4: Using Synonyms

▶ Task 1: Create and Use a Synonym for a Table

1. In Solution Explorer, double-click the query **81 - Lab Exercise 4.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. In the query pane, type the following T-SQL code after the **Task 1** description:

```
CREATE SYNONYM dbo.Person
FOR AdventureWorks.Person.Person;
```

4. Highlight the written T-SQL code and click **Execute**. You have created a synonym named dbo.Person.
5. In the query pane, type the following SELECT statement after the previous T-SQL code:

```
SELECT FirstName, LastName
FROM dbo.Person;
```

6. Highlight the written query and click **Execute**.

▶ Task 2: Drop the Synonym

1. Highlight the following T-SQL code under the **Task 2** description:

```
DROP SYNONYM dbo.Person;
```

2. Click **Execute**.

Results: After this exercise, you should know how to create and use a synonym.

Lab 7: Implementing Error Handling

Scenario

As a junior database developer for Adventure Works, you will be creating stored procedures using corporate databases stored in SQL Server 2012. To create more robust procedures, you will be implementing error handling in your code.

Objectives

After completing this lab, you will be able to:

- Redirect errors with TRY/CATCH.
- Use THROW to pass an error message to a client.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Redirecting Errors with TRY/CATCH

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a Basic TRY/CATCH Construct
3. Display an Error Number and an Error Message
4. Add Conditional Logic to a CATCH Block
5. Execute a Stored Procedure in the CATCH Block

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab17\Starter** folder as Administrator.

▶ Task 2: Write a Basic TRY/CATCH Construct

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. Open the project file **D:\Labfiles\Lab17\Starter\Project\Project.ssmssln** and the T-SQL script **51 - Lab Exercise 1.sql**. Ensure that you are connected to the TSQL database.
3. Execute the provided SELECT statement:

```
SELECT CAST(N'Some text' AS int);
```

4. Notice that you get an error. Write a TRY/CATCH construct by placing the SELECT statement in a TRY block. In the CATCH block, use the PRINT command to display the text "Error". Execute the T-SQL code.

▶ Task 3: Display an Error Number and an Error Message

1. The IT department has provided T-SQL code that looks like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
END CATCH;
```

2. Execute the provided T-SQL code. Notice that nothing happens although, based on the @num variable's value, you should get an error because of the division by zero. Why didn't you get an error?
3. Modify the CATCH block by adding two PRINT statements. The first statement should display the error number by using the ERROR_NUMBER function. The second statement should display the error message by using the ERROR_MESSAGE function. Also, include a label in each printed message, such as "Error Number:" for the first message and "Error Message:" for the second one.
4. Execute and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 2_1 Result.txt.
5. Change the value of the @num variable from 0 to A and execute the T-SQL code.
6. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 2_2 Result.txt.
7. Change the value of the @num variable from A to 1000000000 and execute the T-SQL code.
8. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 2_3 Result.txt.

▶ Task 4: Add Conditional Logic to a CATCH Block

1. Modify the T-SQL code by including an IF statement in the CATCH block before the added PRINT statements. The IF statement should check to see whether the error number is equal to 245 or 8114. If this condition is true, display the message "Handling conversion error…" using a PRINT statement. If this condition is not true, the message "Handling non-conversion error…" should be displayed.
2. Set the value of the @num variable to A and execute the T-SQL code.
3. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\ Exercise 1 - Task 3_1 Result.txt.
4. Change the value of the @num variable to 0 and execute the T-SQL code.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab17\Solution\Exercise 1 - Task 3_2 Result.txt.

▶ Task 5: Execute a Stored Procedure in the CATCH Block

1. The IT department has given you T-SQL code to create a stored procedure named dbo.GetErrorInfo to display different information about the error. Execute the provided T-SQL code:

```
CREATE PROCEDURE dbo.GetErrorInfo AS
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS varchar(10));
PRINT 'Error State: ' + CAST(ERROR_STATE() AS varchar(10));
PRINT 'Error Line: ' + CAST(ERROR_LINE() AS varchar(10));
PRINT 'Error Proc: ' + COALESCE(ERROR_PROCEDURE(), 'Not within procedure');
```

2. Modify the TRY/CATCH code by writing an EXECUTE statement in the CATCH block to invoke the stored procedure dbo.GetErrorInfo.
3. Execute the TRY/CATCH T-SQL code.

Results: After this exercise, you should be able to capture and handle errors using a TRY/CATCH construct.

Exercise 2: Using THROW to Pass an Error Message Back to a Client

Scenario

You will practice how to pass an error message using the THROW statement, and how to send custom error messages.

The main tasks for this exercise are as follows:

1. Rethrow the Existing Error Back to a Client
2. Add an Error Handling Routine
3. Add a Different Error Handling Routine
4. Remove the Stored Procedure

▶ Task 1: Rethrow the Existing Error Back to a Client

1. Open the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the TSQL2012 database.
2. Modify the code to include the THROW statement in the CATCH block after the EXECUTE statement. Execute the T-SQL code.

▶ Task 2: Add an Error Handling Routine

1. Modify the T-SQL code by replacing a THROW statement with an IF statement. Write a condition to compare the error number to the value 8134. If this condition is true, the message "Handling division by zero…" should be displayed. Otherwise, display the message "Throwing original error" and add a THROW statement.
2. Execute the T-SQL code.

▶ Task 3: Add a Different Error Handling Routine

1. The IT department has given you T-SQL code to create a new variable named @msg and set its value:

```
DECLARE @msg AS varchar(2048);
SET @msg = 'You are doing the module 17 on ' + FORMAT(CURRENT_TIMESTAMP, 'MMMM d,
yyyy', 'en-US') + '. It''s not an error but it means that you are near the final
module!';
```

2. Write a THROW statement and specify the message ID of 50001 for the first argument, the @msg variable for the second argument, and the value 1 for the third argument. Highlight the complete T-SQL code and execute it.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab17\Solution\Exercise 2 - Task 3 Result.txt.

▶ Task 4: Remove the Stored Procedure

- Execute the provided T-SQL code to remove the stored procedure dbo.GetErrorInfo.

Results: After this exercise, you should know how to throw an error to pass messages back to a client.

Lab Answer Key 7: Implementing Error Handling

Exercise 1: Redirecting Errors with TRY/CATCH

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab17\Starter** folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**,
4. At the command prompt, type y, and then press **Enter**.
5. Press any key to continue.

▶ Task 2: Write a Basic TRY/CATCH Construct

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab17\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries** and then double-click the **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
6. Highlight the following SELECT statement under the **Task 1** description:

```
SELECT CAST(N'Some text' AS int);
```

7. Click **Execute**. Notice the conversion error.
8. Write a TRY/CATCH construct. Your T-SQL code should look like this:

```
BEGIN TRY
SELECT CAST(N'Some text' AS int);
END TRY
BEGIN CATCH
PRINT 'Error';
END CATCH;
```

9. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Display an Error Number and an Error Message

1. Highlight the following T-SQL code under the **Task 2** description:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
END CATCH;
```

2. Click **Execute**. Notice that you did not get an error because you used the TRY/CATCH construct.

3. Modify the T-SQL code by adding two PRINT statements. The T-SQL code should look like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

4. Highlight the T-SQL code and click **Execute**.
5. Change the value of the @num variable to look like this:

```
DECLARE @num varchar(20) = 'A';
```

6. Highlight the T-SQL code and click **Execute**. Notice that you get a different error number and message.
7. Change the value of the @num variable to look like this:

```
DECLARE @num varchar(20) = ' 1000000000';
```

8. Highlight the T-SQL code and click **Execute**. Notice that you get a different error number and message.

▶ Task 4: Add Conditional Logic to a CATCH Block

1. Modify the T-SQL code in **Task 3** to look like this:

```
DECLARE @num varchar(20) = 'A';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
IF ERROR_NUMBER() IN (245, 8114)
BEGIN
PRINT 'Handling conversion error...'
END
ELSE
BEGIN
PRINT 'Handling non-conversion error...';
END;
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
END CATCH;
```

2. Highlight the written query and click **Execute**.
3. Change the value of the @num variable to look like this:

```
DECLARE @num varchar(20) = '0';
```

4. Highlight the T-SQL code and click **Execute**.

▶ Task 5: Execute a Stored Procedure in the CATCH Block

1. Highlight the following T-SQL code under the **Task 4** description:

```
CREATE PROCEDURE dbo.GetErrorInfo AS
PRINT 'Error Number: ' + CAST(ERROR_NUMBER() AS varchar(10));
PRINT 'Error Message: ' + ERROR_MESSAGE();
PRINT 'Error Severity: ' + CAST(ERROR_SEVERITY() AS varchar(10));
PRINT 'Error State: ' + CAST(ERROR_STATE() AS varchar(10));
 PRINT 'Error Line: ' + CAST(ERROR_LINE() AS varchar(10));
 PRINT 'Error Proc: ' + COALESCE(ERROR_PROCEDURE(), 'Not within procedure');
```

2. Click **Execute**. You have created a stored procedure named dbo.GetErrorInfo.
3. Modify the T-SQL code under **TRY/CATCH** to look like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
EXECUTE dbo.GetErrorInfo;
END CATCH;
```

4. Highlight the written **TRY/CATCH** T-SQL code and click **Execute**.

Results: After this exercise, you should be able to capture and handle errors using a TRY/CATCH construct.

Exercise 2: Using THROW to Pass an Error Message Back to a Client

▶ Task 1: Rethrow the Existing Error Back to a Client

1. In Solution Explorer, double-click the query **61 - Lab Exercise 2.sql**.
2. When the query window opens, highlight the statement **USE TSQL;** and click **Execute**.
3. Modify the T-SQL code under the **Task 1** description to look like this:

```
DECLARE @num varchar(20) = '0';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
EXECUTE dbo.GetErrorInfo; THROW;
END CATCH;
```

4. Highlight the written T-SQL code and click **Execute**.

▶ Task 2: Add an Error Handling Routine

1. Modify the T-SQL code under the **Task 2** description to look like this:

```
DECLARE @num varchar(20) = 'A';
BEGIN TRY
PRINT 5. / CAST(@num AS numeric(10,4));
END TRY
BEGIN CATCH
EXECUTE dbo.GetErrorInfo;
IF ERROR_NUMBER() = 8134
BEGIN
PRINT 'Handling devision by zero...';
END
ELSE
BEGIN
PRINT 'Throwing original error';
THROW;
END;
END CATCH;
```

2. Highlight the written T-SQL code and click **Execute**.

▶ Task 3: Add a Different Error Handling Routine

1. Find the following T-SQL code under the **Task 3** description:

```
DECLARE @msg AS varchar(2048);
SET @msg = 'You are doing the exercise for Module 17 on ' + FORMAT(CURRENT_TIMESTAMP,
'MMMM d, yyyy', 'en-US') + '. It''s not an error but it means that you are near the
final module!';
```

2. After the provided code, add a THROW statement. The completed T-SQL code should look like this:

```
DECLARE @msg AS varchar(2048);
SET @msg = 'You are doing the exercise for Module 17 on ' + FORMAT(CURRENT_TIMESTAMP,
'MMMM d, yyyy', 'en-US') + '. It''s not an error but it means that you are near the
final module!';
THROW 50001, @msg, 1;
```

3. Highlight the written T-SQL code and click **Execute**.

▶ Task 4: Remove the Stored Procedure

- Highlight the provided T-SQL statement under the **Task 4** description and click **Execute**.

Results: After this exercise, you should know how to throw an error to pass messages back to a client.

Lab 8: Implementing Transactions

Scenario

As a junior database developer for Adventure Works, you will be creating stored procedures using corporate databases stored in SQL Server. To create more robust procedures, you will be implementing transactions in your code.

Objectives

After completing this lab, you will be able to:

- Control transactions.
- Add error handling to a CATCH block.

Virtual machine: **20761C-MIA-SQL**

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Controlling Transactions with BEGIN, COMMIT, and ROLLBACK

Scenario

The IT department has supplied different examples of INSERT statements to practice executing multiple statements inside one transaction. You will practice how to start a transaction, commit or abort it, and return the database to its state before the transaction.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Commit a Transaction
3. Delete the Previously Inserted Rows from the HR.Employees Table
4. Open a Transaction and Use the ROLLBACK Statement
5. Clear the Modifications Against the HR.Employees Table

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Run **Setup.cmd** in the **D:\Labfiles\Lab18\Starter** folder as Administrator.

▶ Task 2: Commit a Transaction

1. In SQL Server Management Studio, open the project file **D:\Labfiles\Lab18\Starter\Project\Project.ssmssln** and the T-SQL script 51 - Lab Exercise 1.sql. Ensure that you are connected to the TSQL database.

2. The IT department has provided the following T-SQL code:

```
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'20110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
```

3. This code inserts two rows into the HR.Employees table. By default, SQL Server treats each individual statement as a transaction. In other words, by default, SQL Server automatically commits the transaction at the end of each individual statement. In this case, the default behavior would be two transactions because you have two INSERT statements. (Do not worry about the details of the INSERT statements because they are only meant to provide sample code for the transaction scenario.)
4. In this example, you would like to control the transaction and execute both INSERT statements inside one transaction.
5. Before the supplied T-SQL code, write a statement to open a transaction. After the supplied INSERT statements, write a statement to commit the transaction. Highlight all of the T-SQL code and execute it.
6. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\52 - Lab Exercise 1 - Task 1_1 Result.txt.
7. Write a SELECT statement to retrieve the empid, lastname, and firstname columns from the HR.Employees table. Order the employees by the empid column in a descending order. Execute the SELECT statement.
8. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\53 - Lab Exercise 1 - Task 1_2 Result.txt. Notice the two new rows in the result set.

▶ Task 3: Delete the Previously Inserted Rows from the HR.Employees Table

1. Execute the provided T-SQL code to delete rows inserted from the previous task:

```
DELETE HR.Employees
WHERE empid IN (10, 11);
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Note that this is cleanup code that will not be explained in this course.

▶ Task 4: Open a Transaction and Use the ROLLBACK Statement

1. The IT department has provided T-SQL code (which happens to the same code as in task 1). Before the provided T-SQL code, write a statement to start a transaction.
2. Highlight the written statement and the provided T-SQL code, and execute it.
3. Write a SELECT statement to retrieve the empid, lastname, and firstname columns from the HR.Employees table. Order the employees by the empid column.
4. Execute the written SELECT statement and notice the two new rows in the result set.
5. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\54 - Lab Exercise 1 - Task 3_1 Result.txt.

6. After the written SELECT statement, write a ROLLBACK statement to cancel the transaction. Only execute the ROLLBACK statement.
7. Highlight this and execute the written SELECT statement against the HR.Employees table again.
8. Observe and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab18\Solution\55 - Lab Exercise 1 - Task 3_2 Result.txt. Notice that the two new rows are no longer present in the table.

▶ Task 5: Clear the Modifications Against the HR.Employees Table

- Execute the provided T-SQL code:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

Results: After this exercise, you should be able to control a transaction using the BEGIN TRAN, COMMIT, and ROLLBACK statements.

Exercise 2: Adding Error Handling to a CATCH Block

Scenario

In the previous module, you learned how to add error handling to T-SQL code. Now you will practice how to properly control a transaction by testing to see if an error occurred.

The main tasks for this exercise are as follows:

1. Observe the Provided T-SQL Code
2. Delete the Previously Inserted Row in the HR.Employees Table
3. Abort Both INSERT Statements If an Error Occurs
4. Clear the Modifications Against the HR.Employees Table

▶ Task 1: Observe the Provided T-SQL Code

1. Open the T-SQL script 61 - Lab Exercise 2.sql. Ensure that you are connected to the TSQL database.
2. The IT department has provided T-SQL code that is similar to the code in the previous exercise:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
GO
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

3. Execute only the SELECT statement.
4. Observe and compare the results that you achieved with the desired results shown in the file 62 - Lab Exercise 2 - Task 1_1 result.txt. Notice the number of employees in the HR.Employees table.

5. Execute the part of the T-SQL code that starts with a BEGIN TRAN statement and ends with the COMMIT TRAN statement. You will get a conversion error in the second INSERT statement.
6. Again, execute only the SELECT statement.
7. Observe and compare the results that you achieved with the desired results shown in the file 63 - Lab Exercise 2 - Task 1_2 Result.txt. Notice that, although an error showed inside the transaction block, one new row was added to the HR.Employees table based on the first INSERT statement.

▶ Task 2: Delete the Previously Inserted Row in the HR.Employees Table

- Execute the provided T-SQL code to delete the row inserted from the previous task:

```
DELETE HR.Employees
WHERE empid IN (10, 11);DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

▶ Task 3: Abort Both INSERT Statements If an Error Occurs

1. Modify the provided T-SQL code to include a TRY/CATCH block that rolls back the entire transaction if any of the INSERT statements throws an error:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

2. In the CATCH block, include a PRINT statement that prints the message "Rollback the transaction…" if an error occurred and the message "Commit the transaction…" if no error occurred.
3. Execute the modified T-SQL code.
4. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab18\Solution\64 - Lab Exercise 2 - Task 3_1 Result.txt.
5. Write a SELECT statement against the HR.Employees table to see if any new rows were inserted (like you did in exercise 1). Execute the SELECT statement.
6. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab18\Solution\65 - Lab Exercise 2 - Task 3_2 Result.txt.

▶ Task 4: Clear the Modifications Against the HR.Employees Table

- Execute the provided T-SQL code:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

Results: After this exercise, you should have a basic understanding of how to control a transaction inside a TRY/CATCH block to efficiently handle possible errors.

Lab Answer Key 8: Implementing Transactions

Exercise 1: Controlling Transactions with BEGIN, COMMIT, and ROLLBACK

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are both running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab18\Starter** folder, right-click **Setup.cmd** and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**, and then wait for the script to finish then press any key to continue.

▶ Task 2: Commit a Transaction

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. On the **File** menu, click **Open** and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab18\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, in the **Queries** folder, double-click the query **51 - Lab Exercise 1.sql**.
5. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
6. Modify the T-SQL code under the **Task 1** description by adding the BEGIN TRAN and COMMIT TRAN statements. Your T-SQL code should look like this:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'20110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

   Highlight the written T-SQL code and click **Execute**.

7. In the query pane, type the following query after the previous T-SQL code:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
```

8. Highlight the written query and click **Execute**.

▶ Task 3: Delete the Previously Inserted Rows from the HR.Employees Table

1. Highlight the following T-SQL code under the **Task 2** description:

```
DELETE HR.Employees
WHERE empid IN (10, 11);
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

▶ Task 4: Open a Transaction and Use the ROLLBACK Statement

1. Modify the T-SQL code under the **Task 3** description by adding the BEGIN TRAN statement. Your T-SQL code should look like this:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'20110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
```

2. Highlight the written T-SQL code and click **Execute**.
3. In the query pane, type the following query after the previous T-SQL code:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
```

4. Highlight the written query and click **Execute**.
5. In the query pane, type the following statement after the SELECT statement:

```
ROLLBACK TRAN;
```

6. Highlight the written statement and click **Execute**.
7. Again, highlight the SELECT statement shown in **Step 3** and click **Execute**.

▶ Task 5: Clear the Modifications Against the HR.Employees Table

1. Highlight the following T-SQL code after the **Task 4** description:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

Results: After this exercise, you should be able to control a transaction using the BEGIN TRAN, COMMIT, and ROLLBACK statements.

Exercise 2: Adding Error Handling to a CATCH Block

▶ Task 1: Observe the Provided T-SQL Code

1. In Solution Explorer, double-click the **query 61 - Lab Exercise 2.sql**.
2. In the query window, highlight the statement **USE TSQL;** and click **Execute**.
3. Highlight only the following SELECT statement under the **Task 1** description:

```
SELECT empid, lastname, firstname
FROM HR.Employees
ORDER BY empid DESC;
```

4. Click **Execute**.
5. In the provided T-SQL code, highlight the code between the BEGIN TRAN and COMMIT TRAN statements. Your highlighted T-SQL code should look like this:

```
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
COMMIT TRAN;
```

6. Click **Execute**. Notice there is a conversion error in the second INSERT statement.
7. Again, highlight the SELECT statement shown in **Step 3** and click **Execute**.

▶ Task 2: Delete the Previously Inserted Row in the HR.Employees Table

1. Highlight the following T-SQL code under the **Task 2** description:

```
DELETE HR.Employees
WHERE empid IN (10, 11);
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

▶ Task 3: Abort Both INSERT Statements If an Error Occurs

1. Modify the T-SQL code under the **Task 3** description to look like this:

```
BEGIN TRY
BEGIN TRAN;
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Johnson', N'Test 1', N'Sales Manager', N'Mr.', '19700101', '20110101',
N'Some Address 18', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386) 113322', 2);
INSERT INTO HR.Employees (lastname, firstname, title, titleofcourtesy, birthdate,
hiredate, address, city, region, postalcode, country, phone, mgrid)
VALUES (N'Robertson', N'Test 2', N'Sales Representative', N'Mr.', '19850101',
'10110601', N'Some Address 22', N'Ljubljana', NULL, N'1000', N'Slovenia', N'(386)
553344', 10);
PRINT 'Commit the transaction...';
COMMIT TRAN;
END TRY
BEGIN CATCH
IF @@TRANCOUNT > 0
BEGIN
PRINT 'Rollback the transaction...';
ROLLBACK TRAN;
END
END CATCH;
```

2. Highlight the modified T-SQL code and click **Execute**.
3. In the query pane, type the following query after the modified T-SQL code:

```
SELECT empid, lastname, firstname
FROM HR.Employees ORDER BY empid DESC;
```

4. Highlight the written query and click **Execute**.

▶ Task 4: Clear the Modifications Against the HR.Employees Table

1. Highlight the following T-SQL code under the **Task 4** description:

```
DBCC CHECKIDENT ('HR.Employees', RESEED, 9);
```

2. Click **Execute**.

Results: After this exercise, you should have a basic understanding of how to control a transaction inside a TRY/CATCH block to efficiently handle possible errors.