Querying Data Transact-SQL labs+answer key

Lab 1: Working with SQL Server Tools

Scenario

The Adventure Works Cycles Bicycle Manufacturing Company has adopted SQL Server as its relational database management system. You need to retrieve business data from several SQL Server databases. In the lab, you will begin to explore the new environment, and become acquainted with the tools for querying SQL Server.

Objectives

After completing this lab you will be able to:

- Use SQL Server Management Studio.
- Create and organize T-SQL scripts.

Estimated Time: 30 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Working with SQL Server Management Studio

Scenario

You have been tasked with writing queries for SQL Server. Initially, you want to become familiar with the development environment. You have therefore decided to explore SQL Server Management Studio and configure the editor for your use.

The main tasks for this exercise are as follows:

1. Open Microsoft SQL Server Management Studio

2. Configure the Editor Settings

▶ Task 1: Open Microsoft SQL Server Management Studio

1. Ensure that the **MT17B-WS2016-NAT**, **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. Start SSMS but do not connect to an instance of SQL Server.
3. Close the Object Explorer and Solution Explorer windows.
4. Using the **View** menu, show the Object Explorer and Solution Explorer windows in SSMS.

▶ Task 2: Configure the Editor Settings

1. With SSMS running, open the **Tools** menu and choose **Options**. Change the text editor font size to **14** points.
2. Change additional settings in **Options**:
    - Disable IntelliSense.
    - Change the tab indent to 6 spaces.
    - Include column headers when copying the result set from the grid. Select **Query Results**, **SQL Server**, **Results to Grid**. Select **Include column headers when copying or saving the results**.

Results: After this exercise, you should have opened SSMS and configured editor settings.

Exercise 2: Creating and Organizing T-SQL Scripts

Scenario

You have decided to organize your T-SQL scripts in a project folder. In this lab, you will practice how to create a project and add query files to it.

The main tasks for this exercise are as follows:

1. Create a Project
2. Add an Additional Query File
3. Reopen the Created Project

▶ Task 1: Create a Project

1. Create a new project called **MyFirstProject** and save it in the folder **D:\Labfiles\Lab01\Starter**.
2. Add a new query called **MyFirstQueryFile.sql** to **MyFirstProject**.
3. Save the project and the query file by clicking **Save All**.

▶ Task 2: Add an Additional Query File

1. Add an additional query file called **MySecondQueryFile.sql** to the project you created.
2. Open File Explorer, navigate to the **MyFirstProject** folder to check that your second query file is in your project folder.
3. In SSMS, use the Solution Explorer pane to remove **MySecondQueryFile.sql** from your project by choosing the **Remove** option. (Not the Delete option.)
4. In File Explorer, check to see whether the second query file is still in the project folder.
5. In SSMS, remove **MyFirstQueryFile.sql** by choosing **Delete**.
6. To see the difference, check in File Explorer.

▶ Task 3: Reopen the Created Project

1. Save the project, close SSMS, reopen SSMS, and open the project **MyFirstProject**.
2. Drag **MySecondQueryFile.sql** from File Explorer to the Queries folder beneath **MyFirstProject** in SSMS Solution Explorer.
3. Save the project.

Results: After this lab exercise, you will have a basic understanding of how to create a project in SSMS and add T-SQL query files to it.

Module Review and Takeaways

In this module, you have learned how to:

- Describe the architecture of SQL Server.
- Describe the different editions of SQL Server.
- Work with SSMS.

Review Question(s)

Question: Can a SQL Server database be stored across multiple instances?

Question: If no T-SQL code is selected in a query window, which code lines will be run when you click the Execute button?

Question: What does a SQL Server Management Studio solution contain?

Lab 2: Introduction to T-SQL Querying

Scenario

You are an Adventure Works business analyst, who will be writing reports against corporate databases stored in SQL Server. To help you become more comfortable with SQL Server querying, the Adventure Works IT department has provided some common queries to run against their databases. You will review and execute these queries.

Objectives

After completing this lab, you will be able to:

- Execute basic SELECT statements.
- Execute queries that filter data.
- Execute queries that sort data.

Estimated Time: 30 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Executing Basic SELECT Statements

Scenario

The T-SQL script provided by the IT department includes a SELECT statement that retrieves all rows from the HR.Employees table—this includes the firstname, lastname, city, and country columns. You will execute the T-SQL script against the TSQL database.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Execute the T-SQL Script
3. Execute a Part of the T-SQL Script

▶ 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\Lab02\Starter** folder as Administrator.

▶ Task 2: Execute the T-SQL Script

1. Open the project file **D:\Labfiles\Lab02\Starter\Project\Project.ssmssln**.
2. Connect to the **MIA-SQL** database using Windows authentication.
3. Open the T-SQL script **51 - Lab Exercise 1.sql**.
4. Execute the whole script.
5. Observe the result and the database context.
6. Which database is selected in the Available Databases box?

▶ Task 3: Execute a Part of the T-SQL Script

1. Highlight the SELECT statement in the T-SQL script under the **Task 2** description and click **Execute**.
2. Observe the result. You should get the same result as in **Task 2**.

Note: One way to highlight a portion of code is to hold down the Alt key while drawing a rectangle around it with your mouse. The code inside the drawn rectangle will be selected. Try it.

3. Close all open windows.

Results: After this exercise, you should know how to open the T-SQL script and execute the whole script or just a specific statement inside it.

Exercise 2: Executing Queries That Filter Data Using Predicates

Scenario

The next T-SQL script is very similar to the first one. The SELECT statement retrieves the same columns from the HR.Employees table, but uses a predicate in the WHERE clause to retrieve only rows with the value "USA" in the country column.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Script
2. Change the Database Context with the GUI
3. Change the Database Context with T-SQL

▶ Task 1: Execute the T-SQL Script

1. Open the project file **D:\Labfiles\Lab02\Starter\Project\Project.ssmssln** and the T-SQL script **61 - Lab Exercise 2.sql**. Execute the whole script.
2. There is an error. What is the error message? Why do you think this happened?

▶ Task 2: Change the Database Context with the GUI

1. Apply the needed changes to the script so that it will run without an error. (Hint: you do not need to change any T-SQL information to fix the error.) Test the changes by executing the whole script.
2. Observe the result. Notice that the result has fewer rows than the result in exercise 1, task 2.

▶ Task 3: Change the Database Context with T-SQL

1. Comments in T-SQL scripts can be written inside the line by specifying --. The text after the two hyphens will be ignored by SQL Server. You can also specify a comment as a block starting with **/*** and ending with ***/**. The text in between is treated as a block comment and is ignored by SQL Server.
2. Uncomment the following statements:

```
USE TSQL;
GO
```

3. Save and close the T-SQL script. Re-open the T-SQL script **61 - Lab Exercise 2.sql**. Execute the whole script.
4. Why did the script execute with no errors?

5. Observe the result and notice the database context in the Available Databases box.

Note: SSMS supplies keyboard shortcuts and two buttons so you can quickly comment and uncomment code.
The keyboard shortcuts are CTRL+K then CTRL+C to comment, and CTRL+K then CTRL+U to uncomment.
Or you can use these buttons on the toolbar.

Results: After this exercise, you should have a basic understanding of database context and how to change it.

Exercise 3: Executing Queries That Sort Data Using ORDER BY

Scenario

The last T-SQL script provided by the IT department has a comment: "This SELECT statement returns first name, last name, city, and country/region information for all employees from the USA, ordered by last name."

The main tasks for this exercise are as follows:

1. Execute the Initial T-SQL Script

2. Uncomment the Needed T-SQL Statements and Execute Them

▶ Task 1: Execute the Initial T-SQL Script

1. Open the T-SQL script **71 - Lab Exercise 3.sql**, and execute the whole script.
2. Observe the results. Why is the result window empty?

▶ Task 2: Uncomment the Needed T-SQL Statements and Execute Them

1. Observe that, before the USE statement, there are the characters -- which means that the USE statement is treated as a comment. There is also a block comment around the whole T-SQL SELECT statement. Uncomment both statements.
2. First, execute the USE statement, and then execute the SELECT clause.
3. Observe the results. Notice that the results have the same rows as in exercise 1, task 2, but they are sorted by the lastname column.

Note: What changes would you make to change the sort order to descending?

Results: After this exercise, you should have an understanding of how comments can be specified inside T-SQL scripts. You will also have an appreciation of how to order the results of a query.

Module Review and Takeaways

In this module, you have learned how to describe:

- The elements of T-SQL and their role in writing queries.
- The use of sets in SQL Server.
- The use of predicate logic in SQL Server.
- The logical order of operations in SELECT statements.

Review Question(s)

Question: Which category of T-SQL statements concerns querying and modifying data?

Question: What are some examples of aggregate functions supported by T-SQL?

Question: Which SELECT statement element will be processed before a WHERE clause?

Lab 3: Writing Basic SELECT Statements

Scenario

As a business analyst for Adventure Works, you will be writing reports using corporate databases stored in SQL Server. You can use your set of business requirements for data to write basic T-SQL queries to retrieve the specified data from the databases.

Objectives

After completing this lab, you will be able to:

- Write simple SELECT statements.
- Eliminate duplicate rows by using the DISTINCT keyword.
- Use table and column aliases.
- Use a simple CASE expression.

Estimated Time: 40 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Simple SELECT Statements

Scenario

As a business analyst, you want a better understanding of your corporate data. Usually, the best approach for an initial project is to get an overview of the main tables and columns, so you can better understand different business requirements. After an initial overview, you will provide a report for the marketing department, whose staff want to send invitation letters for a new campaign. You will use the TSQL sample database.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. View All the Tables in the ADVENTUREWORKS Database in Object Explorer

3. Write a Simple SELECT Statement That Returns All Rows and Columns from a Table

4. Write a SELECT Statement That Returns Specific Columns

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **MT17B-WS2016-NAT**, **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\Lab03\Starter** folder as Administrator.

▶ Task 2: View All the Tables in the ADVENTUREWORKS Database in Object Explorer

1. Using SSMS, connect to MIA-SQL using Windows® authentication (if you are connecting to an on-premises instance of SQL Server) or SQL Server authentication.
2. In Object Explorer, expand the **TSQL** database and expand the **Tables** folder.
3. Look at the names of the tables in the Sales schema.

▶ Task 3: Write a Simple SELECT Statement That Returns All Rows and Columns from a Table

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and the T-SQL script Lab Exercise 1.sql. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement that will return all rows and all columns from the **Sales.Customers** table.

Note: You can use drag-and-drop functionality to move items like table and column names from Object Explorer to the query window. Write the same SELECT statement using the drag-and-drop functionality.

3. You can use drag-and-drop functionality to move items like table and column names from Object Explorer to the query window. Write the same SELECT statement using the drag-and-drop functionality.

▶ Task 4: Write a SELECT Statement That Returns Specific Columns

1. Expand the **Sales.Customers** table in Object Explorer and expand the **Columns** folder. Observe all columns in the table.
2. Write a SELECT statement to return the **contactname**, **address**, **postalcode**, **city**, and **country** columns from the **Sales.Customers** table.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 1 - Task 3 Result.txt.
4. What is the number of rows affected by the last query? (Tip: Because you are issuing a SELECT statement against the whole table, the number of rows will be the same as that for the whole **Sales.Customers** table.)

Results: After this exercise, you should know how to create simple SELECT statements to analyze existing tables.

Exercise 2: Eliminating Duplicates Using DISTINCT

Scenario

After supplying the marketing department with a list of all customers for a new campaign, you are asked to provide a list of all the countries that the customers come from.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Includes a Specific Column
2. Write a SELECT Statement That Uses the DISTINCT Clause

▶ Task 1: Write a SELECT Statement That Includes a Specific Column

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and T-SQL script Lab Exercise 2.sql. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement against the **Sales.Customers** table showing only the **country** column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the DISTINCT Clause

1. Copy the SELECT statement in task 1 and modify it to return only distinct values.
2. Execute the written statement and compare the results that you achieved with the desired results shown in file D:\Labfiles\Lab03\Solution\Lab Exercise 2 - Task 2 Result.txt.
3. How many rows did the query in task 1 return?
4. How many rows did the query in task 2 return?
5. Under which circumstances do the following queries against the **Sales.Customers** table return the same result?

```
SELECT city, region FROM Sales.Customers;
SELECT DISTINCT city, region FROM Sales.Customers;
```

6. Is the DISTINCT clause being applied to all columns specified in the query or just the first column?

Results: After this exercise, you should understand how to return only the different (distinct) rows in the result set of a query.

Exercise 3: Using Table and Column Aliases

Scenario

After receiving the initial list of customers, the marketing department would like to have column titles that are more readable and a list of all products in the TSQL database.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses a Table Alias
2. Write a SELECT Statement That Uses Column Aliases
3. Write a SELECT Statement That Uses Table and Column Aliases
4. Analyze and Correct the Query

▶ Task 1: Write a SELECT Statement That Uses a Table Alias

1. Open the project file D:\Labfiles\Lab03\Starter\Project\Project.ssmssln and T-SQL script Lab Exercise 3.sql, and ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to return the **contactname** and **contacttitle** columns from the **Sales.Customers** table, assigning **"C"** as the table alias. Use the table alias **C** to prefix the names of the two needed columns in the SELECT list. The benefit of using table aliases will become clearer in future modules, when topics such as joins and subqueries are introduced.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses Column Aliases

1. Write a SELECT statement to return the **contactname**, **contacttitle**, and **companyname** columns. Assign these with the aliases **Name**, **Title**, and **Company Name**, respectively, to return more human-friendly column titles for reporting purposes.

2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 3 - Task 2 Result.txt. Notice specifically the titles of the columns in the desired output.

▶ Task 3: Write a SELECT Statement That Uses Table and Column Aliases

1. Write a query to display the **productname** column from the **Production.Products** table using **"P"** as the table alias and **Product Name** as the column alias.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Analyze and Correct the Query

1. A developer has written a query to retrieve two columns (**city** and **region**) from the **Sales.Customers** table. When the query is executed, it returns only one column. Your task is to analyze the query, correct it to return two columns, and explain why the query returned only one.

```
SELECT city country
FROM Sales.Customers;
```

2. Execute the query exactly as written inside a query window and observe the result.
3. Correct the query to return the **city** and **country** columns from the **Sales.Customers** table.

   Why did the query return only one column? What was the title of the column in the output? What is the best practice to avoid such errors when using aliases for columns?

Results: After this exercise, you will know how to use aliases for table and column names.

Exercise 4: Using a Simple CASE Expression

Scenario

Your company has a long list of products and the members of the marketing department would like to have product category information in their reports. They have supplied you with a document containing the following mapping between the product category IDs and their names:

categoryid	categoryname
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

They have an active marketing campaign, and would like to include product category information in their reports.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement

2. Write a SELECT Statement That Uses a CASE Expression

3. Write a SELECT Statement That Uses a CASE Expression to Differentiate Campaign-Focused Products

▶ Task 1: Write a SELECT Statement

1. Open the project file **D:\Labfiles\Lab03\Starter\Project\Project.ssmssln** and T-SQL script Lab Exercise 4.sql. Ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to display the **categoryid** and **productname** columns from the **Production.Products** table.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses a CASE Expression

1. Enhance the SELECT statement in task 1 by adding a CASE expression that generates a result column named **categoryname**. The new column should hold the translation of the category ID to its respective category name, based on the mapping table supplied earlier. Use the value "Other" for any category IDs not found in the mapping table.
2. Execute the written statement and compare the results that you achieved with the desired output shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement That Uses a CASE Expression to Differentiate Campaign-Focused Products

1. Modify the SELECT statement in task 2 by adding a new column named **iscampaign**. This will show the description "Campaign Products" for the categories Beverages, Produce, and Seafood, and the description "Non-Campaign Products" for all other categories.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab03\Solution\Lab Exercise 4 - Task 3 Result.txt.

Results: After this exercise, you should know how to use CASE expressions to write simple conditional logic.

Module Review and Takeaways

In this module, you have learned how to:

- Write simple SELECT statements.
- Eliminate duplicates using the DISTINCT clause.
- Use table and column aliases.
- Write simple CASE expressions.

 **Best Practice:** Terminate all T-SQL statements with a semicolon. This will make your code more readable, avoid certain parsing errors, and protect your code against changes in future versions of SQL Server.
Consider standardizing your code on the AS keyword for labeling column and table aliases. This will make it easier to read and avoids accidental aliases.

Review Question(s)

Question: Why is the use of SELECT * not a recommended practice?

Real-world Issues and Scenarios

You can create a column alias without using the AS keyword, something you are likely to see in code samples online, or written by developers you work with. While the T-SQL engine will parse this without issue, there is a problem when a comma is omitted between column names—the first column will take the name of the second column as its alias. Not only will the column have a misleading name, but you will also have one column too few in your result set. Always use the AS keyword to avoid this problem.

Lab 4: Querying Multiple Tables

Scenario

You are an Adventure Works business analyst who 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. You notice that the data is stored in separate tables, so you will need to write queries using various join operations.

Objectives

After completing this lab, you will be able to:

- Write queries that use inner joins.
- Write queries that use multiple-table inner joins.
- Write queries that use self joins.
- Write queries that use outer joins
- Write queries that use cross joins.

Estimated Time: 50 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Inner Joins

Scenario

You no longer need the supplied mapping information between categoryid and categoryname because you now have the Production.Categories table with the needed mapping rows. Write a SELECT statement using an inner join to retrieve the productname column from the Production.Products table and the categoryname column from the Production.Categories table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement That Uses an Inner Join

▶ 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\Lab04\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement That Uses an Inner Join

1. Open the project file **D:\Labfiles\Lab04\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 that will return the **productname** column from the **Production.Products** table (use table alias "p") and the **categoryname** column from the **Production.Categories** table (use table alias "c") using an inner join.

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file **D:\Labfiles\Lab04\Solution\52 - Lab Exercise 1 - Task 2 Result.txt**.

4. Which column did you specify as a predicate in the ON clause of the join? Why?
5. Let us say that there is a new row in the **Production.Categories** table and this new product category does not have any products associated with it in the **Production.Products** table. Would this row be included in the result of the SELECT statement written in task 1? Please explain.

Results: After this exercise, you should know how to use an inner join between two tables.

Exercise 2: Writing Queries That Use Multiple-Table Inner Joins

Scenario

The sales department would like a report of all customers who placed at least one order, with detailed information about each one. A developer prepared an initial SELECT statement that retrieves the custid and contactname columns from the Sales.Customers table and the orderid column from the Sales.Orders table. You should observe the supplied statement and add additional information from the Sales.OrderDetails table.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Statement
2. Apply the Needed Changes and Execute the T-SQL Statement
3. Change the Table Aliases
4. Add an Additional Table and Columns

▶ Task 1: Execute the T-SQL Statement

1. Open the T-SQL script **61 - Lab Exercise 2.sql**. Ensure that you are connected to the TSQL database.
2. The developer has written this query:

```
SELECT
custid, contactname, orderid
FROM Sales.Customers
INNER join Sales.Orders ON Customers.custid = Orders.custid;
```

   Execute the query exactly as written inside a query window and observe the result.

3. An error is shown. What is the error message? Why do you think this happened?

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Notice that there are full source table names written as table aliases.
2. Apply the needed changes to the SELECT statement so that it will run without an error. Test the changes by executing the T-SQL statement.
3. Observe and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\62 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Change the Table Aliases

1. Copy the T-SQL statement from task 2 and modify it to use the table aliases “c” for the Sales.Customers table and “o” for the Sales.Orders table.
2. Execute the written statement and compare the results with those in task 2.
3. Change the prefix of the columns in the SELECT statement with full source table names and execute the statement.

4. There is an error. Why?
5. Change the SELECT statement to use the table aliases written at the beginning of the task.

▶ Task 4: Add an Additional Table and Columns

1. Copy the T-SQL statement from task 3 and modify it to include three additional columns from the Sales.OrderDetails table: productid, qty, and unitprice.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\63 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should have a better understanding of why aliases are important and how to do a multiple-table join.

Exercise 3: Writing Queries That Use Self Joins

Scenario

The HR department would like a report showing employees and their managers. They want to see the lastname, firstname, and title columns from the HR.Employees table for each employee, and the same columns for the employee's manager.

The main tasks for this exercise are as follows:

1. Write a Basic SELECT Statement
2. Write a Query That Uses a Self Join

▶ Task 1: Write a Basic SELECT Statement

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the TSQL database.
2. To better understand the needed tasks, you will first write a SELECT statement against the HR.Employees table showing the empid, lastname, firstname, title, and mgrid columns.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\72 - Lab Exercise 3 - Task 1 Result.txt. Notice the values in the mgrid column. The mgrid column is in a relationship with the empid column. This is called a self-referencing relationship.

▶ Task 2: Write a Query That Uses a Self Join

1. Copy the SELECT statement from task 1 and modify it to include additional columns for the manager information (lastname, firstname) using a self join. Assign the aliases mgrlastname and mgrfirstname respectively, to distinguish the manager names from the employee names.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\73 - Lab Exercise 3 - Task 2 Result.txt. Notice the number of rows returned.
3. Is it mandatory to use table aliases when writing a statement with a self join? Can you use a full source table name as an alias? Please explain.
4. Why did you get fewer rows in the T-SQL statement under task 2 compared to task 1?

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use self joins.

Exercise 4: Writing Queries That Use Outer Joins

Scenario

The sales department was satisfied with the report you produced in exercise 2. Now sales staff would like to change the report to show all customers, even if they did not have any orders, and still include order information for the customers who did. You need to write a SELECT statement to retrieve all rows from Sales.Customers (columns custid and contactname) and the orderid column from the table Sales.Orders.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses an Outer Join

▶ Task 1: Write a SELECT Statement That Uses an Outer Join

1. Open the project file **D:\Labfiles\Lab04\Starter\Project\Project.ssmssln** and the T-SQL script **81 - Lab Exercise 4.sql**. Ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the custid and contactname columns from the Sales.Customers table and the orderid column from the Sales.Orders table. The statement should retrieve all rows from the Sales.Customers table.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.
4. Notice the values in the column orderid. Are there any missing values (marked as NULL)? Why?

Results: After this exercise, you should have a basic understanding of how to write T-SQL statements that use outer joins.

Exercise 5: Writing Queries That Use Cross Joins

Scenario

The HR department would like to prepare a personalized calendar for each employee. The IT department supplied you with T-SQL code that will generate a table with all dates for the current year. Your job is to write a SELECT statement that would return all rows in this new calendar date table for each row in the HR.Employees table.

The main tasks for this exercise are as follows:

1. Execute the T-SQL Statement
2. Write a SELECT Statement That Uses a Cross Join
3. Drop the HR.Calendar Table

▶ Task 1: Execute the T-SQL Statement

1. Open the T-SQL script **91 - Lab Exercise 5.sql**. Ensure that you are connected to the TSQL database.
2. Execute the T-SQL code under task 1. Don't worry if you do not understand the provided T-SQL code, as it is used here to give a more realistic example for a cross join in the next task.

▶ Task 2: Write a SELECT Statement That Uses a Cross Join

1. Write a SELECT statement to retrieve the empid, firstname, and lastname columns from the HR.Employees table and the calendardate column from the HR.Calendar table.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab04\Solution\92 - Lab Exercise 5 - Task 2 Result.txt.

Note: The dates from the query might not exactly match the solution file.

3. How many rows are returned by the query? There are nine rows in the HR.Employees table. Try to calculate the total number of rows in the HR.Calendar table.

▶ Task 3: Drop the HR.Calendar Table

- Execute the provided T-SQL statement to remove the HR.Calendar table.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use cross joins.

Module Review and Takeaways

In this module, you have learned how to:

- Describe how multiple tables may be queried in a SELECT statement using joins.
- Write queries that use inner joins.
- Write queries that use outer joins.
- Write queries that use self joins and cross joins.

 **Best Practice:** Table aliases should always be defined when joining tables. Joins should be expressed using SQL-92 syntax, with JOIN and ON keywords.

Review Question(s)

Question: How does an inner join differ from an outer join?

Question: Which join types include a logical Cartesian product?

Question: Can a table be joined to itself?

Lab 5: Sorting and Filtering Data

Scenario

You are an Adventure Works business analyst who will be writing reports using corporate databases stored in SQL Server. You have been provided with a set of data business requirements and will write T-SQL queries to retrieve the specified data from the databases. You will need to retrieve only some of the available data, and return it to your reports in a specified order.

Objectives

After completing this lab, you will be able to:

- Write queries that filter data using a WHERE clause.
- Write queries that sort data using an ORDER BY clause.
- Write queries that filter data using the TOP option.
- Write queries that filter data using an OFFSET-FETCH clause.

Estimated Time: 60 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Write Queries that Filter Data Using a WHERE Clause

Scenario

The marketing department is working on several campaigns for existing customers and staff need to obtain different lists of customers, depending on several business rules. Based on these rules, you will write the SELECT statements to retrieve the needed rows from the Sales.Customers table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement Using a WHERE Clause
3. Write a SELECT Statement Using an IN Predicate in the WHERE Clause
4. Write a SELECT Statement Using a LIKE Predicate in the WHERE Clause
5. Observe the T-SQL Statement Provided by the IT Department
6. Write a SELECT Statement to Retrieve Customers Without Orders

▶ 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\Lab05\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement Using a WHERE Clause

1. Open the project file **D:\Labfiles\Lab05\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 that will return the **custid**, **companyname**, **contactname**, **address**, **city**, **country**, and **phone** columns from the **Sales.Customers** table. Filter the results to include only the customers from the country Brazil.

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

▶ Task 3: Write a SELECT Statement Using an IN Predicate in the WHERE Clause

1. Write a SELECT statement that will return the **custid**, **companyname**, **contactname**, **address**, **city**, **country**, and **phone** columns from the **Sales.Customers** table. Filter the results to include only customers from the countries Brazil, UK, and USA.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Write a SELECT Statement Using a LIKE Predicate in the WHERE Clause

1. Write a SELECT statement that will return the **custid**, **companyname**, **contactname**, **address**, **city**, **country**, and **phone** columns from the **Sales.Customers** table. Filter the results to include only the customers with a contact name starting with the letter A.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Observe the T-SQL Statement Provided by the IT Department

1. The IT department has written a T-SQL statement that retrieves the **custid** and **companyname** columns from the **Sales.Customers** table and the **orderid** column from the **Sales.Orders** table:

```
SELECT
c.custid, c.companyname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid AND c.city = N'Paris';
```

2. Execute the query and notice two things: first, the query retrieves all the rows from the **Sales.Customers** table. Second, there is a comparison operator in the ON clause, specifying that the **city** column should be equal to the value 'Paris'.
3. Copy the provided T-SQL statement and modify it to have a comparison operator for the **city** column in the WHERE clause. Execute the query.
4. Compare the results that you achieved with the desired results shown in the files D:\Labfiles\Lab05\Solution\55 - Lab Exercise 1 - Task 4a Result.txt and D:\Labfiles\Lab05\Solution\56 - Lab Exercise 1 - Task 4b Result.txt.
5. Is the result the same as in the first T-SQL statement? Why? What is the difference between specifying the predicate in the ON clause and in the WHERE clause?

▶ Task 6: Write a SELECT Statement to Retrieve Customers Without Orders

1. Write a T-SQL statement to retrieve customers from the **Sales.Customers** table that do not have matching orders in the **Sales.Orders** table. Matching customers with orders is based on a comparison between the customer's and the order's custid values. Retrieve the **custid** and **companyname** columns from the **Sales.Customers** table. (Hint: Use a T-SQL statement similar to the one in the previous task.)
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\57 - Lab Exercise 1 - Task 5 Result.txt.

Results: After this exercise, you should be able to filter rows of data from one or more tables by using WHERE predicates with logical operators.

Exercise 2: Write Queries that Sort Data Using an ORDER BY Clause

Scenario

The sales department would like a report showing all the orders with some customer information. An additional request is that the result be sorted by the order dates and the customer IDs. From previous modules, remember that the order of the rows in the output of a query without an ORDER BY clause is not guaranteed. Because of this, you will have to write a SELECT statement that uses an ORDER BY clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement Using an ORDER BY Clause
2. Apply the Needed Changes and Execute the T-SQL Statement
3. Order the Result by the firstname Column

▶ Task 1: Write a SELECT Statement Using an ORDER BY Clause

1. Open the T-SQL script **61 - Lab Exercise 2.sql**, and ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table and the **orderid** and **orderdate** columns from the **Sales.Orders** table. Filter the results to include only orders placed on or after April 1, 2008 (filter the **orderdate** column), then sort the result by orderdate in descending order and custid in ascending order.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab05\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Someone took your T-SQL statement from lab 4 and added the following WHERE clause:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
WHERE mgrlastname = 'Buck';
```

2. Execute the query exactly as written inside a query window and observe the result.
3. There is an error. What is the error message? Why do you think this happened? (Tip: Remember the logical processing order of the query.)
4. Apply the needed changes to the SELECT statement so that it will run without an error. Test the changes by executing the T-SQL statement.
5. Observe and compare the results that you achieved with the recommended results shown in the D:\Labfiles\Lab05\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Order the Result by the firstname Column

1. Copy the existing T-SQL statement from task 2 and modify it so that the result will return all employees and be ordered by the manager's first name. First, try to use the source column name, and then the alias column name.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab05\Solution\64 - Lab Exercise 2 - Task 3a and 3b Result.txt.
3. Why were you able to use a source column or alias column name?

Results: After this exercise, you should know how to use an ORDER BY clause.

Exercise 3: Write Queries that Filter Data Using the TOP Option

Scenario

The sales department wants to have some additional reports that show the last invoiced orders and the top 10 percent of the most expensive products being sold.

The main tasks for this exercise are as follows:

1. Writing Queries That Filter Data Using the TOP Clause
2. Use the OFFSET-FETCH Clause to Implement the Same Task
3. Write a SELECT Statement to Retrieve the Most Expensive Products

▶ Task 1: Writing Queries That Filter Data Using the TOP Clause

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.Orders** table, and retrieve the **orderid** and **orderdate** columns. Retrieve the 20 most recent orders, ordered by orderdate.
3. Execute the written statement and compare the results that you achieved with the recommended result shown in the file 72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Use the OFFSET-FETCH Clause to Implement the Same Task

1. Write a SELECT statement to retrieve the same result as in task 1, but use the OFFSET-FETCH clause.
2. Execute the written statement and compare the results that you achieved with the results from task 1.
3. Compare the results that you achieved with the recommended result shown in the file 73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Most Expensive Products

1. Write a SELECT statement to retrieve the **productname** and **unitprice** columns from the **Production.Products** table.
2. Execute the T-SQL statement and notice the number of the rows returned.
3. Modify the SELECT statement to include only the top 10 percent of products based on unitprice ordering.
4. Execute the written statement and compare the results that you achieved with the recommended result shown in the file 74 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows returned.
5. Is it possible to implement this task with the OFFSET-FETCH clause?

Results: After this exercise, you should have an understanding of how to apply the TOP option in the SELECT clause of a T-SQL statement.

Exercise 4: Write Queries that Filter Data Using the OFFSET-FETCH Clause

Scenario

In this exercise, you will implement a paging solution for displaying rows from the **Sales.Orders** table because the total number of rows is high. In each page of a report, the user should only see 20 rows.

The main tasks for this exercise are as follows:

1. OFFSET-FETCH Clause to Fetch the First 20 Rows
2. Use the OFFSET-FETCH Clause to Skip the First 20 Rows
3. Write a Generic Form of the OFFSET-FETCH Clause for Paging

▶ Task 1: OFFSET-FETCH Clause to Fetch the First 20 Rows

1. Open the T-SQL script **81 - Lab Exercise 4.sql**, and ensure that you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **custid**, **orderid**, and **orderdate** columns from the **Sales.Orders** table. Order the rows by orderdate and ordered, and then retrieve the first 20 rows.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab05\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Use the OFFSET-FETCH Clause to Skip the First 20 Rows

1. Copy the SELECT statement in task 1 and modify the OFFSET-FETCH clause to skip the first 20 rows and fetch the next 20.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab05\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a Generic Form of the OFFSET-FETCH Clause for Paging

- You are given the parameters @pagenum for the requested page number and @pagesize for the requested page size. Can you work out how to write a generic form of the OFFSET-FETCH clause using those parameters? (Don't worry about not being familiar with those parameters yet.)

Results: After this exercise, you will be able to use OFFSET-FETCH to work page-by-page through a result set returned by a SELECT statement.

Question: What is the difference between filtering using the TOP option, and filtering using the WHERE clause?

Module Review and Takeaways

In this module, you have learned how to enhance a query to limit the number of rows that the query returns, and control the order in which the rows are displayed.

Review Question(s)

Question: Does the physical order of rows in a SQL Server table guarantee any sort order in queries using the table?

Question: You have the following query:

```
SELECT p.PartNumber, p.ProductName, o.Quantity
FROM Sales.Products AS p
LEFT OUTER JOIN Sales.OrderItems AS o
ON p.ID = o.ProductID
ORDER BY o.Quantity ASC
```

You have one new product that has yet to receive any orders. Will this product appear at the top or the bottom of the results?

Lab 6: Working with SQL Server 2016 Data Types

Scenario

You are an Adventure Works business analyst who will be writing reports using corporate databases stored in SQL Server 2016. 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. You will need to retrieve and convert character, and date and time data into various formats.

Objectives

After completing this lab, you will be able to:

- Write queries that return date and time data.
- Write queries that use date and time functions.
- Write queries that return character data.
- Write queries that use character functions.

Estimated Time: 90 Minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Return Date and Time Data

Scenario

Before you start using different date and time functions in business scenarios, you should practice on sample data.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve Information About the Current Date
3. Write a SELECT Statement to Return the Date Data Type
4. Write a SELECT Statement That Uses Different Date and Time Functions
5. Write a SELECT Statement to Show Whether a Table of Strings Can Be Used as Dates

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **MT17B-WS2016-NAT**, **20761C-MIA-DC**, and **20761C-MIA-SQL** virtual machines are 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\Lab06\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve Information About the Current Date

1. Open the project file **D:\Labfiles\Lab06\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 columns that contain:
   - The current date and time. Use the alias currentdatetime.
   - Just the current date. Use the alias currentdate.

- Just the current time. Use the alias currenttime.
- Just the current year. Use the alias currentyear.
- Just the current month number. Use the alias currentmonth.
- Just the current day of month number. Use the alias currentday.
- Just the current week number in the year. Use the alias currentweeknumber.
- The name of the current month based on the currentdatetime column. Use the alias currentmonthname.

3. Execute the written statement and compare the results achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\52 - Lab Exercise 1 - Task 1 Result.txt. Your results will be different because of the current date and time value.
4. Can you use the alias currentdatetime as the source in the second column calculation (currentdate)? Please explain.

▶ Task 3: Write a SELECT Statement to Return the Date Data Type

- Write December 11, 2015 as a column with a data type of date. Use the different possibilities inside the T-SQL language (cast, convert, specific function, and so on) and use the alias somedate.

▶ Task 4: Write a SELECT Statement That Uses Different Date and Time Functions

1. Write a SELECT statement to return columns that contain:
   - A date and time value that is three months from the current date and time. Use the alias threemonths.
   - The number of days between the current date and the first column (threemonths). Use the alias diffdays.
   - The number of weeks between April 4, 1992, and September 16, 2011. Use the alias diffweeks.
   - The first day in the current month, based on the current date and time. Use the alias firstday.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\53 - Lab Exercise 1 - Task 3 Result.txt. Some results will be different because of the current date and time value.

▶ Task 5: Write a SELECT Statement to Show Whether a Table of Strings Can Be Used as Dates

1. The IT department has written a T-SQL statement that creates and populates a table named Sales.Somedates.
2. Execute the provided T-SQL statement.
3. Write a SELECT statement against the Sales.Somedates table and retrieve the isitdate column. Add a new column named converteddate with a new date data type value, based on the column isitdate. If the isitdate column cannot be converted to a date data type for a specific row, return a NULL.
4. Execute the written statement and compare the results achieved with the desired results shown in the file D:\Labfiles\Lab06\Solution\54 - Lab Exercise 1 - Task 4 Result.txt.

   Answer the following questions:
   - What is the difference between the SYSDATETIME and CURRENT_TIMESTAMP functions?
   - What is a language-neutral format for the DATE type?

Results: After this exercise, you should be able to retrieve date and time data using T-SQL.

Exercise 2: Writing Queries That Use Date and Time Functions

Scenario

The sales department wants to have different reports that focus on data during specific time frames. The sales staff would like to analyze distinct customers, distinct products, and orders placed near the end of the month. You should write the SELECT statements using the different date and time functions.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve Customers with Orders in a Given Month
2. Write a SELECT Statement to Calculate the First and Last Day of the Month
3. Write a SELECT Statement to Retrieve the Orders Placed in the Last Five Days of the Ordered Month
4. Write a SELECT Statement to Retrieve All Distinct Products Sold in the First 10 Weeks of the Year 2007

▶ Task 1: Write a SELECT Statement to Retrieve Customers with Orders in a Given Month

1. In Solution Explorer, open the T-SQL script **61 - Lab Exercise 2.sql**.
2. Write a SELECT statement to retrieve distinct values for the custid column from the Sales.Orders table. Filter the results to include only orders placed in February 2008.
3. Execute the written statement and compare your results with the desired results shown in the file 62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Calculate the First and Last Day of the Month

1. Write a SELECT statement with these columns:
   - Current date and time.
   - First date of the current month.
   - Last date of the current month.
2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. The results will differ because they rely on the current date.

▶ Task 3: Write a SELECT Statement to Retrieve the Orders Placed in the Last Five Days of the Ordered Month

1. Write a SELECT statement against the Sales.Orders table and retrieve the orderid, custid, and orderdate columns. Filter the results to include only orders placed in the last five days of the order month.
2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement to Retrieve All Distinct Products Sold in the First 10 Weeks of the Year 2007

1. Write a SELECT statement against the Sales.Orders and Sales.OrderDetails tables and retrieve all the distinct values for the productid column. Filter the results to include only orders placed in the first 10 weeks of the year 2007.

2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.

Results: After this exercise, you should know how to use the date and time functions.

Exercise 3: Writing Queries That Return Character Data

Scenario

Members of the marketing department would like to have a more condensed version of a report for when they talk with customers. They want the information that currently exists in two columns displayed in a single column.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Concatenate Two Columns

2. Add an Additional Column to the Concatenated String Which Might Contain NULL

3. Write a SELECT Statement to Retrieve Customer Contacts Based on the First Character in the Contact Name

▶ Task 1: Write a SELECT Statement to Concatenate Two Columns

1. Open the T-SQL script **71 - Lab Exercise 3.sql**, and ensure that you are connected to the TSQL database.

2. Write a SELECT statement against the Sales.Customers table and retrieve the contactname and city columns. Concatenate both columns so that the new column looks like this:

```
Allen, Michael (city: Berlin)
```

3. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Add an Additional Column to the Concatenated String Which Might Contain NULL

1. Copy the T-SQL statement in task 1 and modify it to extend the calculated column with new information from the region column. For concatenation purposes, treat a NULL in the region column as an empty string. When the region is NULL, the modified column should look like this:

```
Allen, Michael (city: Berlin, region: )
```

   When the region is not NULL, the modified column should look like this:

```
Richardson, Shawn (city: Sao Paulo, region: SP)
```

2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve Customer Contacts Based on the First Character in the Contact Name

1. Write a SELECT statement to retrieve the contactname and contacttitle columns from the Sales.Customers table. Return only rows where the first character in the contact name is 'A' through 'G'.

2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\74 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows returned.

Results: After this exercise, you should have an understanding of how to concatenate character data.

Exercise 4: Writing Queries That Use Character Functions

Scenario

The marketing department want to address customers by their first and last names. In the Sales.Customers table, there is only one column named contactname—it has both elements separated by a comma. You will have to prepare a report to show the first and last names separately.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement That Uses the SUBSTRING Function
2. Write a Query to Retrieve the Contact's First Name Using SUBSTRING
3. Write a SELECT Statement to Format the Customer ID
4. Challenge: Write a SELECT Statement to Return the Number of Character Occurrences

▶ Task 1: Write a SELECT Statement That Uses the SUBSTRING Function

1. Open the T-SQL script **81 - Lab Exercise 4.sql**, and ensure that you are connected to the TSQL database.
2. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table. Based on this column, add a calculated column named lastname, which should consist of all the characters before the comma.
3. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

▶ Task 2: Write a Query to Retrieve the Contact's First Name Using SUBSTRING

1. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table and replace the comma in the contact name with an empty string. Based on this column, add a calculated column named firstname, which should consist of all the characters after the comma.
2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Format the Customer ID

1. Write a SELECT statement to retrieve the custid column from the Sales.Customers table. Add a new calculated column to create a string representation of the custid as a fixed-width (six characters) customer code, prefixed with the letter C and leading zeros. For example, the custid value 1 should look like C00001.
2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\84 - Lab Exercise 4 - Task 3 Result.txt.

▶ Task 4: Challenge: Write a SELECT Statement to Return the Number of Character Occurrences

1. Write a SELECT statement to retrieve the contactname column from the Sales.Customers table. Add a calculated column, which should count the number of occurrences of the character 'a' inside the contact name. (Hint: use the string functions REPLACE and LEN.) Order the result from highest to lowest occurrence.
2. Execute the written statement and compare your results with the recommended results shown in the file D:\Labfiles\Lab06\Solution\85 - Lab Exercise 4 - Task 4 Result.txt.
3. Close SQL Server Management Studio without saving any files.

Results: After this exercise, you should have an understanding of how to use the character functions.

Module Review and Takeaways

In this module, you have learned how to:

- Describe SQL Server data types, type precedence, and type conversions.
- Write queries using numeric data types.
- Write queries using character data types.
- Write queries using date and time data types.

Review Question(s)

Question: Will SQL Server be able to successfully and implicitly convert an **int** data type to a **varchar**?

Question: What data type is suitable for storing Boolean flag information, such as TRUE or FALSE?

Question: What logical operators are useful for retrieving ranges of date and time values?

Lab 7: Using DML to Modify Data

Scenario

You are a database developer for Adventure Works and need to create DML statements to update data in the database to support the website development team. The team need T-SQL statements that they can use to carry out updates to data, based on actions performed on the website. You will supply template DML statements that they can modify to their specific requirements.

Objectives

After completing this lab, you will be able to:

- Insert records.
- Update and delete records.

Estimated Time: 30 Minutes

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

User Name: **ADVENTUREWORKS\STUDENT**

Password: **Pa55w.rd**

Exercise 1: Inserting Records with DML

Scenario

You need to add a new employee to the TempDB.Hr.Employee table and test the required T-SQL code. You can then pass the T-SQL code to the human resources system's web developers, who are creating a web form to simplify this task. You also want to add all potential customers to the Customers table to consolidate those records.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

The exercises will be performed within the TempDB database so that none of the real data is affected. Two scripts are used to set up the environment for the lab—both are included in the project for the lab, along with a sample solution for each exercise.

If you need to start again, open and execute the cleanup script, followed by the setup script; you have a clean environment and can try again.

2. Insert a Row

3. Insert a Row with a SELECT Statement As the Data Provider

▶ Task 1: Prepare the Lab Environment

The exercises will be performed within the TempDB database so that none of the real data is affected. Two scripts are used to set up the environment for the lab—both are included in the project for the lab, along with a sample solution for each exercise. If you need to start again, open and execute the cleanup script, followed by the setup script; you have a clean environment and can try again.

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\Lab07\Starter** folder as Administrator.

▶ Task 2: Insert a Row

1. Open the project file **D:\Labfiles\Lab07\Starter\Project\Project.ssmssln** and execute the **01 setup.sql** query.
2. Write an INSERT statement to add a record to the Employees table within the TempDB.HR.Employees table, with the following values:
   - Title: Sales Representative
   - Titleofcourtesy: Mr
   - FirstName: Laurence
   - Lastname: Grider
   - Hiredate: 04/04/2013
   - Birthdate: 10/25/1975
   - Address: 1234 1st Ave. S.E.
   - City: Seattle
   - Country: USA
   - Phone: (206)555-0105

▶ Task 3: Insert a Row with a SELECT Statement As the Data Provider

- Write an INSERT statement to add all the records from the PotentialCustomers table to the Customers table.

Results: After successfully completing this exercise, you will have one new employee and three new customers.

Exercise 2: Update and Delete Records Using DML

Scenario

You want to update the use of contact titles in the database to match the most commonly-used term in the company—making searches more straightforward. You also want to remove the three potential customers who have been added to the Customers table.

The main tasks for this exercise are as follows:

1. Update Rows
2. Delete Rows

▶ Task 1: Update Rows

- Write an UPDATE statement to update all the records in the Customers table that have a city of **'**Berlin' and a contacttitle of 'Sales Representative' to have a contacttitle of **'**Sales Consultant'.

▶ Task 2: Delete Rows

- Write a DELETE statement to delete all the records in the PotentialCustomers table which have the contactname of 'Taylor, Maurice', 'Mallit, Ken', or 'Tiano, Mike', as these records have now been added to the Customers table.

Results: After successfully completing this exercise, you will have updated all the records in the Customers table that have a city of Berlin and a contacttitle of Sales Representative, to now have a contacttitle of Sales Consultant. You will also have deleted the three records in the PotentialCustomers table, which have already been added to the Customers table.

Question: What attributes of the source columns are transferred to a table created with a SELECT INTO query?

Question: The presence of which constraint prevents TRUNCATE TABLE from executing successfully?

Module Review and Takeaways

In this module, you have learned how to:

- Write T-SQL statements that insert column values into rows within the tables.
- Write T-SQL statements that modify values in columns, within rows, within tables.
- Write T-SQL statements that remove existing rows from tables.
- Appreciate the importance of the WHERE clause when using data modification language (DML).
- Appreciate T-SQL statements that automatically generate values for columns and how this affects you when using DML.
- Understand the use of the MERGE statement to compare and contrast two tables and direct different DML statements, based on their content comparisons.

Common Issues and Troubleshooting Tips

Common Issue	Troubleshooting Tip
You are partway through the exercises and want to start again from the beginning. You run the setup script within the solution and receive lots of error messages. This might occur if you have tried to execute the setup script without running the cleanup script to remove any changes you might have made during the lab.	

Lab 8: Using Built-in Functions

Scenario

You are an Adventure Works business analyst, who 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. You will need to retrieve the data, convert it, and then check for missing values.

Objectives

After completing this lab, you will be able to:

- Write queries that include conversion functions.
- Write queries that use logical functions.
- Write queries that test for nullability.

Estimated Time: 40 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Conversion Functions

Scenario

You have been asked to write the following reports for these departments:

1. **Sales**. The product name and unit price for each product within an easy to read string.
2. **Marketing**. The order id, order date, shipping date, and shipping region for each order after 4/1/2007.
3. **IT**. Convert all Sales phone number information into integers.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement that Uses the CAST or CONVERT Function

3. Write a SELECT Statement to Filter Rows Based on Specific Date Information

4. Write a SELECT Statement to Convert the Phone Number Information to an Integer Value

▶ 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\Lab08\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement that Uses the CAST or CONVERT Function

1. Open the project file **D:\Labfiles\Lab08\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 against the **Production.Products** table to retrieve a calculated column named **productdesc**. The calculated column should be based on the **productname** and **unitprice** columns and look like this:

```
Results: The unit price for the Product HHYDP is 18.00 $.
```

3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.
4. Did you use the CAST or the CONVERT function? Which one do you think is more appropriate to use?

▶ Task 3: Write a SELECT Statement to Filter Rows Based on Specific Date Information

1. The US marketing department has supplied you with a start date of "**4/1/2007**" (using US English form, read as "**April 1, 2007**") and an end date of "**11/30/2007**" (using US English form, read as "**November 30, 2007**").
2. Write a SELECT statement against the **Sales.Orders** table to retrieve the **orderid**, **orderdate**, **shippeddate**, and **shipregion** columns. Filter the result to include only rows with the order date between the specified start date and end date, and have more than 30 days between the shipped date and order date. Also check the **shipregion** column for missing values. If there is a missing value, then return the value "**No region**".
3. In this SELECT statement, you can use the CONVERT function with a style parameter or the PARSE function.
4. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Write a SELECT Statement to Convert the Phone Number Information to an Integer Value

1. The IT department would like to convert all the information about phone numbers in the **Sales.Customers** table to integer values. The IT staff indicated that all hyphens, parentheses, and spaces have to be removed before the conversion to an integer data type.
2. Write a SELECT statement to implement the requirement of the IT department. Replace all the specified characters in the phone column of the **Sales.Customers** table, and then convert the column from the nvarchar datatype to the int datatype. The T-SQL statement must not fail if there is a conversion error—it should return a NULL. (Hint: first try writing a T-SQL statement using the CONVERT function, and then compare it with the TRY_CONVERT function.) Use the alias **phoneasint** for this calculated column.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\54 - Lab Exercise 3 - Task 3 Result.txt.

Results: After this exercise, you should be able to use conversion functions.

Exercise 2: Writing Queries That Use Logical Functions

Scenario

The sales department would like to have different reports regarding the segmentation of customers and specific order lines. You will add a new calculated column to show the target group for the segmentation.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Mark Specific Customers Based on Their Country and Contact Title
2. Modify the T-SQL Statement to Mark Different Customers
3. Create Four Groups of Customers

▶ Task 1: Write a SELECT Statement to Mark Specific Customers Based on Their Country and Contact Title

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 the **Sales.Customers** table and retrieve the **custid** and **contactname** columns. Add a calculated column named **segmentgroup**, using a logical function IIF with the value "**Target group**" for customers that are from Mexico and have the value "**Owner**" in the contact title. Use the value "**Other**" for the rest of the customers.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab08\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Modify the T-SQL Statement to Mark Different Customers

1. Modify the T-SQL statement from task 1 to change the calculated column to show the value "**Target group**" for all customers without a missing value in the region attribute or with the value "**Owner**" in the contact title attribute.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab08\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Create Four Groups of Customers

1. Write a SELECT statement against the **Sales.Customers** table and retrieve the **custid** and **contactname** columns. Add a calculated column named **segmentgroup** using the logical function CHOOSE with four possible descriptions ("**Group One**", "**Group Two**", "**Group Three**", "**Group Four**"). Use the modulo operator on the column **custid**. (Use the expression **custid % 4 + 1** to determine the target group.)
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab08\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

Results: After this exercise, you should know how to use the logical functions.

Exercise 3: Writing Queries That Test for Nullability

Scenario

The sales department would like to have additional segmentation of customers. Some columns that you should retrieve contain missing values, and you will have to change the NULL to some more meaningful information for the business users.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Customer Fax Information

2. Write a Filter for a Variable That Could Be a Null

3. Write a SELECT Statement to Return All the Customers That Do Not Have a Two-Character Abbreviation for the Region

▶ Task 1: Write a SELECT Statement to Retrieve the Customer Fax Information

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 **contactname** and fax columns from the **Sales.Customers** table. If there is a missing value in the fax column, return the value "**No information**".
3. Write two solutions, one using the COALESCE function and the other using the ISNULL function.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab08\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.
5. What is the difference between the ISNULL and COALESCE functions?

▶ Task 2: Write a Filter for a Variable That Could Be a Null

- Update the provided T-SQL statement with a WHERE clause to filter the region column using the provided variable **@region**, which can have a value or a NULL. Test the solution using both provided variable declaration cases:

```
DECLARE @region AS NVARCHAR(30) = NULL;
SELECT
custid, region
FROM Sales.Customers;
GO
DECLARE @region AS NVARCHAR(30) = N'WA';
SELECT
custid, region
FROM Sales.Customers;
```

▶ Task 3: Write a SELECT Statement to Return All the Customers That Do Not Have a Two-Character Abbreviation for the Region

1. Write a SELECT statement to retrieve the **contactname**, **city**, and **region** columns from the **Sales.Customers** table. Return only rows that do not have two characters in the region column, including those with an inapplicable region (where the region is NULL).
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab08\Solution\73 - Lab Exercise 3 - Task 3 Result.txt. Notice the number of rows returned.

Results: After this exercise, you should have an understanding of how to test for nullability.

Module Review and Takeaways

In this module, you have learned how to:

- Write queries with built-in scalar functions.
- Use conversion functions.
- Use logical functions.
- Use functions that work with NULL.

Best Practice: When possible, use standards-based functions, such as CAST or COALESCE, rather than SQL Server-specific functions like NULLIF or CONVERT.

- Consider the impact of functions in a WHERE clause on query performance.

Review Question(s)

Question: Which function should you use to convert from an int to a nchar(8)?

Question: Which function will return a NULL, rather than an error message, if it cannot convert a string to a date?

Question: What is the name for a function that returns a single value?

Lab 9: Grouping and Aggregating Data

Scenario

You are an Adventure Works business analyst, who 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 it from the databases. You will need to perform calculations upon groups of data and filter according to the results.

Objectives

After completing this lab, you will be able to:

- Write queries that use the GROUP BY clause.
- Write queries that use aggregate functions.
- Write queries that use distinct aggregate functions.
- Write queries that filter groups with the HAVING clause.

Estimated Time: 60 Minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use the GROUP BY Clause

Scenario

The sales department want to create additional upsell opportunities from existing customers. The staff need to analyze different groups of customers and product categories, depending on several business rules. Based on these rules, you will write SELECT statements to retrieve the needed rows from the Sales.Customers table.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment

2. Write a SELECT Statement to Retrieve Different Groups of Customers

3. Add an Additional Column From the Sales.Customers Table

4. Write a SELECT Statement to Retrieve the Customers with Orders for Each Year

5. Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

▶ 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\Lab09\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve Different Groups of Customers

1. Open the project file **D:\Labfiles\Lab09\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 that will return groups of customers who made a purchase. The SELECT clause should include the **custid** column from the **Sales.Orders** table, and the **contactname** column from the **Sales.Customers** table. Group both columns and filter only the orders from the sales employee whose empid equals five.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\52 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 3: Add an Additional Column From the Sales.Customers Table

1. Copy the T-SQL statement in task 1 and modify it to include the **city** column from the **Sales.Customers** table in the SELECT clause.
2. Execute the query.
3. You will get an error. What is the error message? Why?
4. Correct the query so that it will execute properly.
5. Execute the query and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\53 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement to Retrieve the Customers with Orders for Each Year

1. Write a SELECT statement that will return groups of rows based on the **custid** column and a calculated column **orderyear** representing the order year based on the **orderdate** column from the **Sales.Orders** table. Filter the results to include only the orders from the sales employee whose empid equals five.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\54 - Lab Exercise 1 - Task 4 Result.txt.

▶ Task 5: Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

1. Write a SELECT statement to retrieve groups of rows based on the **categoryname** column in the **Production.Categories** table. Filter the results to include only the product categories that were ordered in the year 2008.
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\55 - Lab Exercise 1 - Task 5 Result.txt.

Results: After this exercise, you should be able to use the GROUP BY clause in the T-SQL statement.

Exercise 2: Writing Queries That Use Aggregate Functions

Scenario

The marketing department wants to launch a new campaign, so the staff need to gain a better insight into the existing customers' buying behavior. You should create different sales reports, based on the total and average sales amount per year and per customer.

The main tasks for this exercise are as follows:

1. Write a SELECT statement to Retrieve the Total Sales Amount Per Order

2. Add Additional Columns

3. Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

4. Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

▶ Task 1: Write a SELECT statement to Retrieve the Total Sales Amount Per Order

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 **orderid** column from the **Sales.Orders** table and the total sales amount per orderid. (Hint: multiply the **qty** and **unitprice** columns from the **Sales.OrderDetails** table.) Use the alias **salesamount** for the calculated column. Sort the result by the total sales amount in descending order.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab09\Solution\62 - Lab Exercise 2 - Task 1 Result.txt.

▶ Task 2: Add Additional Columns

1. Copy the T-SQL statement in task 1 and modify it to include the total number of order lines for each order and the average order line sales amount value within the order. Use the aliases **nooforderlines** and **avgsalesamountperorderline**, respectively.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\63 - Lab Exercise 2 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

1. Write a select statement to retrieve the total sales amount for each month. The SELECT clause should include a calculated column named **yearmonthno** (YYYYMM notation), based on the **orderdate** column in the **Sales.Orders** table and a total sales amount (multiply the **qty** and **unitprice** columns from the **Sales.OrderDetails** table). Order the result by the **yearmonthno** calculated column.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab09\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

1. Write a select statement to retrieve all the customers (including those who did not place any orders) and their total sales amount, maximum sales amount per order line, and number of order lines.
2. The SELECT clause should include the **custid** and **contactname** columns from the **Sales.Customers** table and four calculated columns based on appropriate aggregate functions:
   a. **totalsalesamount**, representing the total sales amount per order
   b. **maxsalesamountperorderline**, representing the maximum sales amount per order line

c. **numberofrows**, representing the number of rows (use * in the COUNT function)

d. **numberoforderlines**, representing the number of order lines (use the **orderid** column in the COUNT function)

3. Order the result by the **totalsalesamount** column.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\65 - Lab Exercise 2 - Task 4 Result.txt.
5. Notice that the **custid** 22 and 57 rows have a NULL in the columns with the SUM and MAX aggregate functions. What are their values in the **COUNT** columns? Why are they different?

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

Scenario

The marketing department want to have some additional reports that display the number of customers who made any order in a specific time period and the number of customers based on the first letter in the contact name.

The main tasks for this exercise are as follows:

1. Modify a SELECT Statement to Retrieve the Number of Customers
2. Write a SELECT Statement to Analyze Segments of Customers
3. Write a SELECT Statement to Retrieve Additional Sales Statistics

▶ Task 1: Modify a SELECT Statement to Retrieve the Number of Customers

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Ensure that you are connected to the **TSQL** database.
2. A junior analyst prepared a T-SQL statement to retrieve the number of orders and the number of customers for each order year. Observe the provided T-SQL statement and execute it:

```
SELECT
YEAR(orderdate) AS orderyear,
COUNT(orderid) AS nooforders,
COUNT(custid) AS noofcustomers
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

3. Observe the results. Notice that the number of orders is the same as the number of customers. Why?
4. Amend the T-SQL statement to show the correct number of customers who placed an order for each year.
5. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Analyze Segments of Customers

1. Write a SELECT statement to retrieve the number of customers based on the first letter of the values in the **contactname** column from the **Sales.Customers** table. Add an additional column to show the total number of orders placed by each group of customers. Use the aliases **firstletter**, **noofcustomers** and **nooforders**. Order the result by the **firstletter** column.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve Additional Sales Statistics

1. Copy the T-SQL statement in exercise 1, task 5, and modify to include the following information about each product category—total sales amount, number of orders, and average sales amount per order. Use the aliases **totalsalesamount, nooforders**, and **avgsalesamountperorder**, respectively.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

Results: After this exercise, you should have an understanding of how to apply a DISTINCT aggregate function.

Exercise 4: Writing Queries That Filter Groups with the HAVING Clause

Scenario

The sales and marketing departments were satisfied with the reports you provided to analyze customers' behavior. Now they would like to have the results filtered, based on the total sales amount and number of orders. So, in the final exercise, you will learn how to filter the result, based on aggregated functions, and learn when to use the WHERE and HAVING clauses.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Top 10 Customers
2. Write a SELECT Statement to Retrieve Specific Orders
3. Apply Additional Filtering
4. Retrieve the Customers with More Than 25 Orders

▶ Task 1: Write a SELECT Statement to Retrieve the Top 10 Customers

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 to retrieve the top 10 customers (by total sales amount) who spent more than $10,000. Display the **custid** column from the **Orders** table and a calculated column that contains the total sales amount, based on the **qty** and **unitprice** columns from the **Sales.OrderDetails** table. Use the alias **totalsalesamount** for the calculated column.
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\82 - Lab Exercise 4 - Task 1 Result.txt.

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

1. Write a SELECT statement against the **Sales.Orders** and **Sales.OrderDetails** tables, and display the **empid** column and a calculated column representing the total sales amount. Filter the results to group only the rows with an order year 2008.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\83 - Lab Exercise 4 - Task 2 Result.txt.

▶ Task 3: Apply Additional Filtering

1. Copy the T-SQL statement in task 2 and modify it to apply an additional filter to retrieve only the rows that have a sales amount higher than $10,000.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\84 - Lab Exercise 4 - Task 3_1 Result.txt.

3. Apply an additional filter to show only employees with empid equal to 3.
4. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab09\Solution\85 - Lab Exercise 4 - Task 3_2 Result.txt.
5. Did you apply the predicate logic in the WHERE clause or the HAVING clause? Which do you think is better? Why?

▶ Task 4: Retrieve the Customers with More Than 25 Orders

1. Write a SELECT statement to retrieve all customers who placed more than 25 orders and add information about the date of the last order and the total sales amount. Display the **custid** column from the **Sales.Orders** table and two calculated columns— **lastorderdate** based on the **orderdate** column, and **totalsalesamount** based on the **qty** and **unitprice** columns in the **Sales.OrderDetails** table.
2. Execute the written statement and compare the results that you achieved with the recommended result shown in the file D:\Labfiles\Lab09\Solution\86 - Lab Exercise 4 - Task 4 Result.txt.
3. Close SQL Server Management Studio without saving any files.

Results: After this exercise, you should have an understanding of how to use the HAVING clause.

Module Review and Takeaways

In this lesson, you have learned how to:

- List the built-in aggregate functions provided by SQL Server.
- Write queries that use aggregate functions in a SELECT list to summarize all the rows in an input set.
- Describe the use of the DISTINCT option in aggregate functions.
- Write queries using aggregate functions that handle the presence of NULLs in source data.

Review Question(s)

Question: What is the difference between the COUNT function and the COUNT_BIG function?

Question: Can a GROUP BY clause include more than one column?

Question: In a query, can a WHERE clause and a HAVING clause filter on the same column?

Lab 10: Using Subqueries

Scenario

As a business analyst for Adventure Works, you are writing reports using corporate databases stored in SQL Server. You have been handed a set of business requirements for data and will write T-SQL queries to retrieve the specified data from the databases. Due to the complexity of some of the requests, you will need to embed subqueries into your queries to return results in a single query.

Objectives

After completing this lab, you will be able to:

- Write queries that use subqueries.
- Write queries that use scalar and multiresult set subqueries.
- Write queries that use correlated subqueries and the EXISTS predicate.

Estimated Time: 60 minutes

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

User name: **AdventureWorks\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Self-Contained Subqueries

Scenario

The sales department needs some advanced reports to analyze sales orders. You will write different SELECT statements that use self-contained subqueries.

The main tasks for this exercise are as follows:

1. Prepare the Lab Environment
2. Write a SELECT Statement to Retrieve the Last Order Date
3. Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date
4. Observe the T-SQL Statement Provided by the IT Department
5. Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

▶ 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\Lab10\Starter** folder as Administrator.

▶ Task 2: Write a SELECT Statement to Retrieve the Last Order Date

1. Open the project file **D:\Labfiles\Lab10\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 maximum order date from the table **Sales.Orders**.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\52 - Lab Exercise 1 - Task 1 Result.txt.

▶ Task 3: Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date

1. Write a SELECT statement to return the **orderid**, **orderdate**, **empid**, and **custid** columns from the **Sales.Orders** table. Filter the results to include only orders where the date order equals the last order date. (Hint: use the query in task 1 as a self-contained subquery.)
2. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\53 - Lab Exercise 1 - Task 2 Result.txt.

▶ Task 4: Observe the T-SQL Statement Provided by the IT Department

1. The IT department has written a T-SQL statement that retrieves the orders for all customers whose contact name starts with a letter I:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid =
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'I%'
);
```

2. Execute the query and observe the result.
3. Modify the query to filter customers whose contact name starts with a letter B.
4. Execute the query. What happened? What is the error message? Why did the query fail?
5. Apply the needed changes to the T-SQL statement so that it will run without an error.
6. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\54 - Lab Exercise 1 - Task 3 Result.txt.

▶ Task 5: Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

1. Write a SELECT statement to retrieve the **orderid** column from the **Sales.Orders** table and the following calculated columns:
   - **totalsalesamount** (based on the **qty** and **unitprice** columns in the **Sales.OrderDetails** table).
   - **salespctoftotal** (percentage of the total sales amount for each order divided by the total sales amount for all orders in a specific period).
2. Filter the results to include only orders placed in May 2008.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\55 - Lab Exercise 1 - Task 4 Result.txt.

Results: After this exercise, you should be able to use self-contained subqueries in T-SQL statements.

Exercise 2: Writing Queries That Use Scalar and Multiresult Subqueries

Scenario

The marketing department would like to prepare materials for different groups of products and customers, based on historic sales information. You have to prepare different SELECT statements that use a subquery in the WHERE clause.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve Specific Products
2. Write a SELECT Statement to Retrieve Those Customers Without Orders
3. Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

▶ Task 1: Write a SELECT Statement to Retrieve Specific Products

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. Filter the results to include only products that were sold in high quantities (more than 100) for a specific order line.
3. Execute the written statement and compare the results that you achieved with the desired results shown in the file D:\Labfiles\Lab10\Solution\62 - Lab Exercise 2 -Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement to Retrieve Those Customers Without Orders

1. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table. Filter the results to include only those customers who do not have any placed orders.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\63 - Lab Exercise 2 - Task 2 Result.txt. Remember the number of rows in the results.

▶ Task 3: Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

1. The IT department has written a T-SQL statement that inserts an additional row in the **Sales.Orders** table. This row has a NULL in the **custid** column:

```
INSERT INTO Sales.Orders (
custid, empid, orderdate, requireddate, shippeddate, shipperid, freight,
shipname, shipaddress, shipcity, shipregion, shippostalcode, shipcountry)
VALUES
(NULL, 1, '20111231', '20111231', '20111231', 1, 0,
'ShipOne', 'ShipAddress', 'ShipCity', 'RA', '1000', 'USA')
```

2. Execute this query exactly as written inside a query window.
3. Copy the T-SQL statement you wrote in task 2 and execute it.
4. Observe the result. How many rows are in the result? Why?
5. Modify the T-SQL statement to retrieve the same number of rows as in task 2. (Hint: you have to remove the rows with an unknown value in the **custid** column.)
6. Execute the modified statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\64 - Lab Exercise 2 - Task 3 Result.txt.

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

Exercise 3: Writing Queries That Use Correlated Subqueries and an EXISTS Predicate

Scenario

The sales department would like to have some additional reports to display different analyses of existing customers. Because the requests are complex, you will need to use correlated subqueries.

The main tasks for this exercise are as follows:

1. Write a SELECT Statement to Retrieve the Last Order Date for Each Customer
2. Write a SELECT Statement That Uses the EXISTS Predicate to Retrieve Those Customers Without Orders
3. Write a SELECT Statement to Retrieve Customers Who Bought Expensive Products
4. Write a SELECT Statement to Display the Total Sales Amount and the Running Total Sales Amount for Each Order Year
5. Clean the Sales.Customers Table

▶ Task 1: Write a SELECT Statement to Retrieve the Last Order Date for Each Customer

1. Open the T-SQL script **71 - Lab Exercise 3.sql**. Make sure you are connected to the **TSQL** database.
2. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table. Add a calculated column named **lastorderdate** that contains the last order date from the **Sales.Orders** table for each customer. (Hint: you have to use a correlated subquery.)
3. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\72 - Lab Exercise 3 - Task 1 Result.txt.

▶ Task 2: Write a SELECT Statement That Uses the EXISTS Predicate to Retrieve Those Customers Without Orders

1. Write a SELECT statement to retrieve all customers that do not have any orders in the **Sales.Orders** table, similar to the request in exercise 2, task 3. However, this time use the EXISTS predicate to filter the results to include only those customers without an order. Also, you do not need to explicitly check that the **custid** column in the **Sales.Orders** table is not NULL.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\73 - Lab Exercise 3 - Task 2 Result.txt.
3. Why didn't you need to check for a NULL?

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Bought Expensive Products

1. Write a SELECT statement to retrieve the **custid** and **contactname** columns from the **Sales.Customers** table. Filter the results to include only customers who placed an order on or after April 1, 2008, and ordered a product with a price higher than $100.
2. Execute the written statement and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\74 - Lab Exercise 3 - Task 3 Result.txt.

▶ Task 4: Write a SELECT Statement to Display the Total Sales Amount and the Running Total Sales Amount for Each Order Year

1. Running aggregates accumulate values over time. Write a SELECT statement to retrieve the following information for each year:
   - The order year.
   - The total sales amount.
   - The running total sales amount over the years. That is, for each year, return the sum of sales amount up to that year. So, for example, for the earliest year (2006), return the total sales amount; for the next year (2007), return the sum of the total sales amount for the previous year and 2007.
2. The SELECT statement should have three calculated columns:
   - **orderyear**, representing the order year. This column should be based on the **orderyear** column from the **Sales.Orders** table.
   - **totalsales**, representing the total sales amount for each year. This column should be based on the **qty** and **unitprice** columns from the **Sales.OrderDetails** table.
   - **runsales**, representing the running sales amount. This column should use a correlated subquery.
3. Execute the T-SQL code and compare the results that you achieved with the recommended results shown in the file D:\Labfiles\Lab10\Solution\75 - Lab Exercise 3 - Task 4 Result.txt.

▶ Task 5: Clean the Sales.Customers Table

1. Delete the row added in exercise 2 using the provided SQL statement:

```
DELETE Sales.Orders
WHERE custid IS NULL;
```

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

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

Module Review and Takeaways

In this module, you have learned how to:

- Describe the uses for queries that are nested within other queries.
- Write self-contained subqueries that return scalar or multi-valued results.
- Write correlated subqueries that return scalar or multi-valued results.
- Use the EXISTS predicate to efficiently check for the existence of rows in a subquery.

Review Question(s)

Question: Can a correlated subquery return a multi-valued set?

Question: What type of subquery may be rewritten as a JOIN?

Question: Which columns should appear in the SELECT list of a subquery following the EXISTS predicate?

Lab 11: 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.

Estimated Time: 90 minutes

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

User name: **ADVENTUREWORKS\Student**

Password: **Pa55w.rd**

Exercise 1: Writing Queries That Use Views

Scenario

In the last 10 modules, you had to prepare many different T-SQL statements to support different business requirements. Because some of them used a similar table and column structure, you would like to have them reusable. You will learn how to use one of two persistent table expressions—a view.

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?
3. Modify the supplied T-SQL statement by including the TOP (100) PERCENT option. The query should look like this:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT TOP(100) PERCENT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1
ORDER BY productname;
```

4. Execute the modified T-SQL statement. By applying the needed changes, you have altered the existing view. Notice that you are still using the ORDER BY clause.
5. If you write a query against the modified Production.ProductsBeverages view, is it guaranteed that the retrieved rows will be sorted by productname? Please explain.

▶ 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 can do 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.

6. 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 12: 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.

Estimated Time: 60 minutes

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, each joining 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 13: 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.

Estimated Time: 60 minutes

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 no missing 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.

Module Review and Takeaways

In this module, you have learned how to:

- Describe the benefits of using window functions.
- Restrict window functions to rows defined in an OVER clause, including partitions and frames.
- Write queries that use window functions to operate on a window of rows and return ranking, aggregation, and offset comparison results.

Review Question(s)

Question: What results will be returned by a ROW_NUMBER function if there is no ORDER BY clause in the query?

Question: Which ranking function would you use to return the values 1,1,3? Which would return 1,1,2?

Question: Can a window frame extend beyond the boundaries of the window partition defined in the same OVER() clause?

Lab 14: 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 the GROUPING SETS, CUBE, and ROLLUP subclauses.

Estimated Time: 60 minutes

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 and product 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 with an 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 15: 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.

Estimated Time: 30 minutes

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 16: 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.

Estimated Time: 45 minutes

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 an empid 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 17: 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.

Estimated Time: 30 minutes

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.

Module Review and Takeaways

In this module, you have learned how to:

- Implement T-SQL error handling.
- Implement structured exception handling.

Review Question(s)

Question: Which error types cannot by caught by structured exception handling?

Question: Can TRY/CATCH blocks be nested?

Question: How can you use THROW outside of a CATCH block?

Lab 18: 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.

Estimated Time: 30 minutes

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.

Module Review and Takeaways

In this module, you have learned how to:

- Describe transactions and the differences between batches and transactions.
- Describe batches and how they are handled by SQL Server.
- Create and manage transactions with transaction control language (TCL) statements.
- Use SET XACT_ABORT to define SQL Server's handling of transactions outside TRY/CATCH blocks.

Review Question(s)

Question: What happens to a nested transaction when the outer transaction is rolled back?

Question: When a runtime error occurs in a transaction and SET XACT_ABORT is ON, is the transaction always automatically rolled back?

Lab 1: Working with SQL Server Tools

Exercise 1: Working with SQL Server Management Studio

▶ Task 1: Open Microsoft SQL Server Management Studio

1. Ensure that the **MT17B-WS2016-NAT**, **20761C-MIA-DC** and **20761C-MIA-SQL** virtual machines are running.
2. Log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
3. Start SQL Server Management Studio.
4. In the **Connect to Server** dialog box, click **Cancel**.
5. Close the Object Explorer window by clicking the **Close** icon.
6. Close the Solution Explorer window by clicking the **Close** icon.
7. To open the Object Explorer pane, on the **View** menu, click **Object Explorer** (or press F8).
8. To open the Solution Explorer pane, on the **View** menu, click **Solution Explorer** (or press Ctrl+Alt+L).

▶ Task 2: Configure the Editor Settings

1. In SQL Server Management Studio, on the **Tools** menu, click **Options**.
2. In the **Options** dialog box, expand the **Environment** option, and then click **Fonts and Colors**.
3. In the **Show settings for** list, click **Text Editor**.
4. In the **Size** box, set the font size to **14**.
5. In the left pane, expand **Text Editor**, expand **Transact-SQL**, and then click **IntelliSense**.
6. In the **Transact-SQL IntelliSense Settings** section, clear the **Enable IntelliSense** check box.
7. In the left pane, under **Transact-SQL**, click **Tabs**, and then change the **Tab size** to **6**.
8. In the left pane, expand **Query Results**, expand **SQL Server**, and then click **Results to Grid**.
9. Select the **Include column headers when copying or saving the results** check box, and then click **OK**.

Results: After this exercise, you should have opened SSMS and configured editor settings.

Exercise 2: Creating and Organizing T-SQL Scripts

▶ Task 1: Create a Project

1. On the **File** menu, point to **New**, and then click **Project**.
2. In the **New Project** dialog box, click **SQL Server Scripts**.
3. In the **Name** box, type **MyFirstProject**.
4. In the **Location** box, type **D:\Labfiles\Lab01\Starter**, and then click **OK** to create the new project.
5. In Solution Explorer, under **MyFirstProject**, right-click the **Queries** folder, and then click **New Query**.
6. In the **Connect to Database Engine** dialog box, click **Cancel**.
7. Under the **Queries** folder, right-click **SQLQuery1.sql**, click **Rename**, type **MyFirstQueryFile**, and then press Enter.
8. On the **File** menu, click **Save All**.

▶ Task 2: Add an Additional Query File

1. In Solution Explorer, right-click the **Queries** folder, and then click **New Query**.
2. In the **Connect to Database Engine** dialog box, click **Cancel**.
3. In the **Queries** folder, right-click **SQLQuery1.sql**, click **Rename**, type **MySecondQueryFile**, and then press Enter.
4. On the taskbar, click **File Explorer**.
5. In File Explorer, navigate to the **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject** folder to see where the files have been created.
6. In SQL Server Management Studio, in Solution Explorer, right-click **MySecondQueryFile.sql**, and then click **Remove**.
7. In the **Microsoft SQL Server Management Studio** dialog box, click **Remove**.
8. In File Explorer, press F5 to refresh, notice that the file **MySecondQueryFile.sql** is still there.
9. In SQL Server Management Studio Solution, in Solution Explorer, right-click **MyFirstQueryFile.sql**, and then click **Remove**.
10. In the **Microsoft SQL Server Management Studio** dialog box, click **Delete**.
11. In File Explorer, press F5 to refresh, notice that the **MyFirstQueryFile.sql** file has been deleted.

▶ Task 3: Reopen the Created Project

1. In SQL Server Management Studio Solution, on the **File** menu, click **Save All**.
2. On the **File** menu, click **Exit** to close the project and SQL Server Management Studio.
3. Open SQL Server Management Studio.
4. In the **Connect to Server** dialog box, click **Cancel**.
5. On the **File** menu, point to **Open**, and then click **Project/Solution**.
6. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab01\Starter\MyFirstProject** folder, click **MyFirstProject.ssmssln**, and then click **Open**.
7. In File Explorer, navigate to the **D:\Labfiles\Lab01\Starter\MyFirstProject\MyFirstProject** folder.
8. Drag the **MySecondQueryFile.sql** file to the **Queries** folder in **Solution Explorer**.

9. On the **File** menu, click **Save All**.

Results: After this lab exercise, you will have a basic understanding of how to create a project in SSMS and add T-SQL query files to it.

Lab 2: Introduction to T-SQL Querying

Exercise 1: Executing Basic SELECT Statements

▶ 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\Lab02\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.
4. Press any key to close the command window.

▶ Task 2: Execute the T-SQL Script

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, ensure Windows Authentication is selected, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, browse to the **D:\Labfiles\Lab02\Starter\Project** folder, and then double-click **Project.ssmssln**.
5. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
6. When the query window opens, click **Execute**. You will notice that the TSQL database is selected in the Available Databases box. The Available Databases box displays the current database context under which the T-SQL script will run. This information is also visible on the status bar.

▶ Task 3: Execute a Part of the T-SQL Script

1. Highlight the following text under the **Task 2** description:

```
SELECT firstname
       ,lastname
       ,city
       ,country
FROM HR.Employees;
```

 **Note:** To highlight it, move the pointer over the statement while pressing the left mouse button or use the arrow keys to move the pointer while pressing the Shift key.

2. Click **Execute**. It is very important to understand that you can highlight a specific part of the code inside the T-SQL script, and execute only that part. If you click **Execute** without selecting any part of the code, the whole T-SQL script will be executed. If you highlight a specific part of the code by mistake, the SQL Server will attempt to run only that part.
3. On the **File** menu, click **Close**.
4. Close SQL Server Management Studio, without saving any changes.

Results: After this exercise, you should know how to open the T-SQL script and execute the whole script or just a specific statement inside it.

Exercise 2: Executing Queries That Filter Data Using Predicates

▶ Task 1: Execute the T-SQL Script

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Options**.
3. On the **Connection Properties** tab, in the **Connect to database** list, ensure **<default>** is selected, and then click **Connect**.
4. On the **File** menu, point to **Open**, and then click **Project/Solution**.
5. In the **Open Project** dialog box, browse to the **D:\Labfiles\Lab02\Starter\Project** folder, and then double-click **Project.ssmssln**.
6. In Solution Explorer, expand **Queries**, and then double-click **61 - Lab Exercise 2.sql**.
7. In the query pane, click **Execute**.
8. Notice that you get the error message:

```
Msg 208, Level 16, State 1, Line 18
Invalid object name 'HR.Employees'.
```

   Why do you think this happened? This error is very common when you are beginning to learn T-SQL.

9. The message tells you that SQL Server could not find the object HR.Employees. This is because the current database context is set to the master database (look at the Available Databases box where the current database is displayed), but the IT department supplied T-SQL scripts to be run against the TSQL database. So you need to change the database context from master to TSQL. You will learn how to change the database context in the next task.

▶ Task 2: Change the Database Context with the GUI

1. In the **Available Databases** list, click **TSQL** to change the database context.
2. Click **Execute**.
3. Notice that the result from the **SELECT** statement returns fewer rows than the one in exercise 1. That is because it has a predicate in the **WHERE** clause to filter out all rows that do not have the value **USA** in the country column. Only rows for which the logical expression evaluates to **TRUE** are returned by the **WHERE** phase to the subsequent logical query processing phase.

▶ Task 3: Change the Database Context with T-SQL

1. In the script **61 - Lab Exercise 2.sql**, find the lines:

```
--USE TSQL;
--Go
```

2. Delete the first two characters, so that the line looks like this:

```
USE TSQL;
GO
```

3. By deleting these two characters, you have removed the comment mark. Now the line will not be ignored by SQL Server.
4. On the **File** menu, click **Save 61 - Lab Exercise 2.sql**.
5. In the **Save File As** dialog box, click **Save**.
6. In the **Confirm Save As** dialog box, click **Yes**
7. On the **File** menu, click **Close**. This will close the T-SQL script.
8. In Solution Explorer, double-click **61 - Lab Exercise 2.sql**.
9. Click **Execute**.
10. Observe the results. Why did the script execute with no errors? The script now includes the uncommented *USE TSQL;* statement. When you execute the whole T-SQL script, the USE statement sets the database context to the **TSQL** database. The next statement in the T-SQL script, the **SELECT**, then executes against the **TSQL** database.
11. On the **File** menu, click **Close**.

Results: After this exercise, you should have a basic understanding of database context and how to change it.

Exercise 3: Executing Queries That Sort Data Using ORDER BY

▶ Task 1: Execute the Initial T-SQL Script

1. In Solution Explorer, double-click **71 - Lab Exercise 3.sql**.
2. Click **Execute**.
3. Notice that the result window is empty. All the statements inside the T-SQL script are commented out, so SQL Server ignores them.

▶ Task 2: Uncomment the Needed T-SQL Statements and Execute Them

1. Locate the line:

```
--USE TSQL;
```

2. Delete the two characters before the USE statement. The line should now look like this:

```
USE TSQL;
```

3. Locate the block comment start element **/*** after the **Task 1** description and delete it.
4. Locate the block comment end element */ and delete it.
   Any text residing within a block starting with /* and ending with */ is treated as a block comment and is ignored by SQL Server.
5. Highlight the statement:

```
USE TSQL;
```

6. Click **Execute**. The database context is now set to the **TSQL** database.

7. Highlight the statement:

```
SELECT
firstname, lastname, city, country
FROM HR.Employees
WHERE country = 'USA'
ORDER BY lastname;
```

8. Click **Execute**.
9. Observe the result and notice that the rows are sorted by the lastname column in ascending order.

Results: After this exercise, you should have an understanding of how comments can be specified inside T-SQL scripts. You will also have an appreciation of how to order the results of a query.

Lab 3: Writing Basic SELECT Statements

Exercise 1: Writing Simple SELECT Statements

▶ Task 1: Prepare the Lab Environment

Ensure that the **MT17B-WS2016-NAT**, **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**.

1. In the **D:\Labfiles\Lab03\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
2. In the **User Account Control** dialog box, click **Yes**.
3. When the script has finished, press Enter.

▶ Task 2: View All the Tables in the ADVENTUREWORKS Database in Object Explorer

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Options**.
3. Under **Connection Properties**, in the **Connect to database** list, click **<Browse server>**.
4. In the **Browse for Databases** dialog box, click **Yes**.
5. In the **Browse Server for Databases** dialog box, under **User Databases**, click **TSQL**, and then click **OK**.
6. In the **Connect to Server** dialog box, on the **Login** tab, in the **Authentication** list, click **Windows Authentication**, and then click **Connect**.
7. In Object Explorer, under **MIA-SQL**, expand **Databases**, expand **TSQL**, and then expand **Tables**.
8. Under **Tables**, notice that there are four table objects in the Sales schema:
   - Sales.Customers
   - Sales.OrderDetails
   - Sales.Orders
   - Sales.Shippers

▶ Task 3: Write a Simple SELECT Statement That Returns All Rows and Columns from a Table

1. On the **File** menu, point to **Open**, and then click **Project/Solution**.
2. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab03\Starter\Project** folder, and then double-click **Project.ssmssln**.
3. In Solution Explorer, expand **Queries**, and then double-click **Lab Exercise 1.sql**.
4. In the query window, highlight the statement **USE TSQL;**, and then click **Execute**.

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

```
SELECT *
FROM Sales.Customers;
```

6. Highlight the query you typed, and click **Execute**.
7. In the query pane, type the following code after the first query:

```
SELECT *
FROM
```

8. In Object Explorer, under **MIA-SQL**, under **Databases**, under **TSQL**, under **Tables**, click **Sales.Customers**.
9. Drag the selected table into the query pane, after the FROM clause. Add a semicolon to the end of the SELECT statement. Your finished query should look like this:

```
SELECT *
FROM [Sales].[Customers];
```

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

▶ Task 4: Write a SELECT Statement That Returns Specific Columns

1. In Object Explorer, expand **Sales.Customers**, expand **Columns** and observe all the columns in the **Sales.Customers** table.
2. In the query pane, after the **Task 3** description, type the following query:

```
SELECT contactname, address, postalcode, city, country
FROM Sales.Customers;
```

3. Highlight the written query, and click **Execute**.
4. Observe the result. How many rows are affected by the last query? There are multiple ways to answer this question using SQL Server Management Studio. One way is to select the previous query and click **Execute**. The total number of rows affected by the executed query is written in the Results pane under the **Messages** tab:

```
(91 row(s) affected)
```

Another way is to look at the status bar displayed below the Results pane. On the left side of the status bar, there is a message stating: "Query executed successfully." On the right side, the total number of rows affected by the current query is displayed (91 rows).

Results: After this exercise, you should know how to create simple SELECT statements to analyze existing tables.

Exercise 2: Eliminating Duplicates Using DISTINCT

▶ Task 1: Write a SELECT Statement That Includes a Specific Column

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

```
SELECT    country
FROM Sales.Customers;
```

4. Highlight the written query, and click **Execute**.
5. Observe that you have multiple rows with the same values. This occurs because the **Sales.Customers** table has multiple rows with the same value for the country column.

▶ Task 2: Write a SELECT Statement That Uses the DISTINCT Clause

1. Highlight the previous query, and then on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 2** description.
3. On the **Edit** menu, click **Paste**. You have now copied the previous query to the same query window after the task 2 description.
4. Modify the query by typing **DISTINCT** after the SELECT clause. Your query should look like this:

```
SELECT DISTINCT country
FROM Sales.Customers;
```

5. Highlight the written query, and click **Execute**.
6. Observe the result and answer these questions:

   How many rows did the query in task 1 return?

   *To answer this question, you can highlight the query written under the task 1 description, click **Execute**, and read the Results pane. (If you forgot how to access this pane, look at task 4 in exercise 1.) The number of rows affected by the query is 91.*

   How many rows did the query in task 2 return?

   *To answer this question, you can highlight the query written under the task 2 description, click **Execute**, and read the Results pane. The number of rows affected by the query is 21. This means that there are 21 distinct values for the **country** column in the **Sales.Customers** table.*

   Under which circumstances do the following queries against the **Sales.Customers** table return the same result?

```
SELECT city, region FROM Sales.Customers;
SELECT DISTINCT city, region FROM Sales.Customers;
```

   If all combinations of values in the city and region columns in the Sales.Customers table are unique, both queries would return the same number of rows. If they are not unique, the first query would return more rows than the second one with the DISTINCT clause.

   Is the DISTINCT clause applied to all columns specified in the query—or just the first column?

   The DISTINCT clause is always applied to all columns specified in the SELECT list. It is very important to remember that the DISTINCT clause does not just apply to the first column in the list.

Results: After this exercise, you should understand how to return only the different (distinct) rows in the result set of a query.

Exercise 3: Using Table and Column Aliases

▶ Task 1: Write a SELECT Statement That Uses a Table Alias

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

```
SELECT    c.contactname, c.contacttitle
FROM Sales.Customers AS c;
```

   Tip: To use the IntelliSense feature when entering column names in a SELECT statement, you can use keyboard shortcuts. To enable IntelliSense, press Ctrl+Q+I. To list all the alias members, position your pointer after the alias and dot (for example, after "c.") and press Ctrl+J.

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

▶ Task 2: Write a SELECT Statement That Uses Column Aliases

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

```
SELECT    c.contactname AS Name, c.contacttitle AS Title, c.companyname AS [Company
Name]
FROM Sales.Customers AS c;
```

   Observe that the column alias **[Company Name]** is enclosed in square brackets. Column names and aliases with embedded spaces or reserved keywords must be delimited. This example uses square brackets as the delimiter, but you can also use the ANSI SQL standard delimiter of double quotes, as in "Company Name".

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

▶ Task 3: Write a SELECT Statement That Uses Table and Column Aliases

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

```
SELECT p.productname AS [Product Name]
FROM Production.Products AS p;
```

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

▶ Task 4: Analyze and Correct the Query

1. Highlight the written query under the **Task 4** description, and click **Execute**.
2. Observe the result. Note that only one column is retrieved. The problem is that the developer forgot to add a comma after the first column name, so SQL Server treated the second word after the first column name as an alias. For this reason, it is best practice to always use AS when specifying aliases—then it is easier to spot such errors.

3. Correct the query by adding a comma after the first column name. The corrected query should look like this:

```
SELECT city, country
FROM Sales.Customers;
```

Results: After this exercise, you will know how to use aliases for table and column names.

Exercise 4: Using a Simple CASE Expression

▶ Task 1: Write a SELECT Statement

1. In Solution Explorer, double-click **Lab Exercise 4.sql**.
2. In the query window, 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.categoryid, p.productname
FROM Production.Products AS p;
```

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

▶ Task 2: Write a SELECT Statement That Uses a CASE Expression

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

```
SELECT p.categoryid, p.productname,
CASE
WHEN p.categoryid = 1 THEN 'Beverages'
WHEN p.categoryid = 2 THEN 'Condiments'
WHEN p.categoryid = 3 THEN 'Confections'
WHEN p.categoryid = 4 THEN 'Dairy Products'
WHEN p.categoryid = 5 THEN 'Grains/Cereals'
WHEN p.categoryid = 6 THEN 'Meat/Poultry'
WHEN p.categoryid = 7 THEN 'Produce'
WHEN p.categoryid = 8 THEN 'Seafood'
ELSE 'Other'
END AS categoryname
FROM Production.Products AS p;
```

   This query uses a CASE expression to add a new column. Note that, when you have a dynamic list of possible values, you usually store them in a separate table. However, for this example, a static list of values is being supplied.

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

▶ Task 3: Write a SELECT Statement That Uses a CASE Expression to Differentiate Campaign-Focused Products

1. Highlight the previous query, and then on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3** description.
3. On the **Edit** menu, click **Paste**. You have now copied the previous query to the same query window after the task 3 description.

4. Add a new column using an additional CASE expression. Your query should look like this:

```
SELECT    p.categoryid, p.productname,
   CASE
          WHEN p.categoryid = 1 THEN 'Beverages'
          WHEN p.categoryid = 2 THEN 'Condiments'
WHEN p.categoryid = 3 THEN 'Confections'
          WHEN p.categoryid = 4 THEN 'Dairy Products'
          WHEN p.categoryid = 5 THEN 'Grains/Cereals'
          WHEN p.categoryid = 6 THEN 'Meat/Poultry'
          WHEN p.categoryid = 7 THEN 'Produce'
          WHEN p.categoryid = 8 THEN 'Seafood'
          ELSE 'Other'
   END AS categoryname,
   CASE
          WHEN p.categoryid IN (1, 7, 8) THEN 'Campaign Products'
          ELSE 'Non-Campaign Products'
   END AS iscampaign
FROM Production.Products AS p;
```

5. Highlight the written query, and click **Execute**.
6. In the result, observe that the first CASE expression uses the simple form, whereas the second uses the searched form.

Results: After this exercise, you should know how to use CASE expressions to write simple conditional logic.

Lab 4: Querying Multiple Tables

Exercise 1: Writing Queries That Use Inner Joins

▶ 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\Lab04\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**, press Enter, wait for the script to finish, and then press any key.

▶ Task 2: Write a SELECT Statement That Uses an Inner Join

1. On the taskbar, click **Microsoft SQL Server Management Studio**.
2. In the **Connect to Server** dialog box, in the **Server name** box, type **MIA-SQL**, and then click **Connect**.
3. On the **File** menu, point to **Open**, and then click **Project/Solution**.
4. In the **Open Project** dialog box, browse to the **D:\Labfiles\Lab04\Starter\Project** folder, and then double-click **Project.ssmssln**.
5. In Solution Explorer, expand **Queries**, and then double-click **51 - Lab Exercise 1.sql**.
6. In the query window, highlight the statement **USE TSQL;**, and click **Execute**.
7. In the query pane, after the **Task 1** description, type the following query:

```
SELECT
p.productname, c.categoryname
FROM Production.Products AS p
INNER JOIN Production.Categories AS c ON p.categoryid = c.categoryid;
```

8. Highlight the written query, and click **Execute**.
9. Observe the result and answer these questions:
   - Which column did you specify as a predicate in the ON clause of the join? Why?

     In this query, the categoryid column is the predicate. By intuition, most people would say that this is the predicate because the column exists in both input tables. By the way, using the same name for columns that contain the same data but in different tables is a good practice in data modeling. Another possibility is to check for referential integrity through primary and foreign key information using SQL Server Management Studio. If there are no primary or foreign key constraints, you will have to acquire information about the data model from the developer.
   - Let us say that there is a new row in the Production.Categories table and this new product category does not have any products associated with it in the Production.Products table. Would this row be included in the result of the SELECT statement written under the task 1 description?

     No, because an inner join retrieves only the matching rows based on the predicate from both input tables. Since the new value for the categoryid is not present in the categoryid column in the

Production.Products table, there would be no matching rows in the result of the SELECT statement.

Results: After this exercise, you should know how to use an inner join between two tables.

Exercise 2: Writing Queries That Use Multiple-Table Inner Joins

▶ Task 1: Execute the T-SQL Statement

1. In Solution Explorer, double-click the **61 - Lab Exercise 2.sql** query.
2. In the query window, highlight the statement **USE TSQL;**, and then click **Execute**.
3. Under the **Task 1** description, highlight the written query, and click **Execute**.
4. Observe the error message:

   Ambiguous column name 'custid'.

5. This error occurred because the custid column appears in both tables; you have to specify from which table you would like to retrieve the column values.

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Highlight the previous query, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 2** description, and on the **Edit** menu, click **Paste**.
3. Add the column prefix **Customers** to the existing query so that it looks like this:

```
SELECT
Customers.custid, contactname, orderid
FROM Sales.Customers
INNER JOIN Sales.Orders ON Customers.custid = Orders.custid;
```

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

▶ Task 3: Change the Table Aliases

1. Highlight the previous query, 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 to use table aliases. Your query should look like this:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

4. Highlight the written query and click **Execute**.
5. Compare these results with the **Task 2** results.
6. Modify the **T-SQL** statement to include a full source table name as the column prefix. Your query should now look like this:

```
SELECT
Customers.custid, Customers.contactname, Orders.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

7. Highlight the written query and click **Execute**.
8. Observe the error messages:

```
Msg 4104, Level 16, State 1, Line 57
The multipart identifier "Customers.custid" could not be bound.
Msg 4104, Level 16, State 1, Line 57
The multipart identifier "Customer.contactname" could not be bound.
Msg 4104, Level 16, State 1, Line 57
The multipart identifier "Orders.orderid" could not be bound.
```

   You received these error messages as, because you are using a different table alias, the full source table name you are referencing as a column prefix no longer exists. Remember that the SELECT clause is evaluated after the FROM clause, so you must use the table aliases when specifying columns in the SELECT clause.

9. Modify the SELECT statement so that it uses the correct table aliases. Your query should look like this:

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

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

▶ Task 4: Add an Additional Table and Columns

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

```
SELECT
c.custid, c.contactname, o.orderid, d.productid, d.qty, d.unitprice
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid;
```

2. Highlight the written query, and click **Execute**.
3. Observe the result. Remember that, when you have a multiple-table inner join, the logical query processing is different from the physical implementation. In this case, it means that you cannot guarantee the order in which the SQL Server optimizer will process the tables. For example, you cannot guarantee that the Sales.Customers table will be joined first with the Sales.Orders table, and then with the Sales.OrderDetails table.

Results: After this exercise, you should have a better understanding of why aliases are important and how to do a multiple-table join.

Exercise 3: Writing Queries That Use Self Joins

▶ Task 1: Write a Basic SELECT Statement

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

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid
FROM HR.Employees AS e;
```

4. Highlight the written query and click **Execute**.
5. Observe that the query retrieved nine rows.

▶ Task 2: Write a Query That Uses a Self Join

1. Highlight the previous query, and on the **Edit** menu, click **Copy**.
2. In the query window, after the **Task 2** description, click the line, and on the **Edit** menu, click **Paste**.
3. Modify the query by adding a self join to get information about the managers. The query should look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid;
```

4. Highlight the written query and click **Execute**.
5. Observe that the query retrieved eight rows and answer these questions:
   - Is it mandatory to use table aliases when writing a statement with a self join? Can you use a full source table name as an alias?

   You must use table aliases. You cannot use the full source table name as an alias when referencing both input tables. Eventually, you could use a full source table name as an alias for one input table and another alias for the second input table.

   - Why did you get fewer rows in the result from the T-SQL statement under the task 2 description, compared to the result from the T-SQL statement under the task 1 description?
6. In task 2's T-SQL statement, the inner join used an ON clause based on manager information (column mgrid). The employee who is the CEO has a missing value in the mgrid column, so this row is not included in the result.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use self joins.

Exercise 4: Writing Queries That Use Outer Joins

▶ Task 1: Write a SELECT Statement That Uses an Outer Join

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

```
SELECT
c.custid, c.contactname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid;
```

4. Highlight the written query and click **Execute**.
5. Inspect the result. Notice that the custid 22 and custid 57 rows have a missing value in the orderid column. This is because there are no rows in the Sales.Orders table for these two values of the custid column. In business terms, this means that there are currently no orders for these two customers.

Results: After this exercise, you should have a basic understanding of how to write T-SQL statements that use outer joins.

Exercise 5: Writing Queries That Use Cross Joins

▶ Task 1: Execute the T-SQL Statement

1. In Solution Explorer, double-click the **91 - Lab Exercise 5.sql** query.
2. In the query window, highlight the statement **USE TSQL;**, and then click **Execute**.
3. Under the **Task 1** description, highlight the T-SQL code and click **Execute**. Don't worry if you do not understand the provided T-SQL code, as it is used here to provide a more realistic example for a cross join in the next task.

▶ Task 2: Write a SELECT Statement That Uses a Cross Join

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

```
SELECT
e.empid, e.firstname, e.lastname, c.calendardate
FROM HR.Employees AS e
CROSS JOIN HR.Calendar AS c;
```

2. Highlight the written query and click **Execute**.
3. Observe that the query retrieved 3,294 rows and that there are nine rows in the HR.Employees table. Because a cross join produces a Cartesian product of both inputs, it means that there are 366 (3,294/9) rows in the HR.Calendar table.

▶ Task 3: Drop the HR.Calendar Table

- Under the **Task 3** description, highlight the written query and click **Execute**.

Results: After this exercise, you should have an understanding of how to write T-SQL statements that use cross joins.

Lab 5: Sorting and Filtering Data

Exercise 1: Write Queries that Filter Data Using a WHERE Clause

▶ 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\Lab05\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.
4. At the command prompt, when prompted, press any key.

▶ Task 2: Write a SELECT Statement Using a WHERE Clause

1. Start SQL Server Management Studio and connect to the MIA-SQL database engine instance 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\Lab05\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
custid, companyname, contactname, address, city, country, phone
FROM Sales.Customers
WHERE
country = N'Brazil';
```

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

   Note the use of the "N" prefix for the character literal 'Brazil'. This prefix is used because the country column is a Unicode data type. When expressing a Unicode character literal, you need to specify the character "N" (for National) as a prefix. If the "N" is omitted, then the query may still run successfully. However, the safest way is to include the "N" every time, to ensure the results are predictable. You will learn more about data types in the next module.

▶ Task 3: Write a SELECT Statement Using an IN Predicate in the WHERE Clause

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

```
SELECT
custid, companyname, contactname, address, city, country, phone
FROM Sales.Customers
WHERE
country IN (N'Brazil', N'UK', N'USA');
```

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

▶ Task 4: Write a SELECT Statement Using a LIKE Predicate in the WHERE Clause

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

```
SELECT
custid, companyname, contactname, address, city, country, phone
FROM Sales.Customers
WHERE
contactname LIKE N'A%';
```

2. Remember that the percent sign (%) wildcard represents a string of any size (including an empty string), whereas the underscore (_) wildcard represents a single character.

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

▶ Task 5: Observe the T-SQL Statement Provided by the IT Department

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

2. Highlight the provided T-SQL statement, and on the toolbar, click **Edit**, and then click **Copy**.

3. In the query window, click the line after the **Task 4b** description, and on the toolbar, click **Edit**, and then click **Paste**.

4. Modify the query so that it looks like this:

```
SELECT
c.custid, c.companyname, o.orderid
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE
c.city = N'Paris';
```

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

6. Observe the result. Is it the same as that of the first SQL statement?

   The result is not the same. When you specify the predicate in the ON clause, the left outer join preserves all the rows from the left table (**Sales.Customers**) and adds only the matching rows from the right table (**Sales.Orders**), based on the predicate in the ON clause. This means that all the customers will show up in the output, but only the ones from Paris will have matching orders. When you specify the predicate in the WHERE clause, the query will filter only the Paris customers. So be aware that, when you use an outer join, the result of a query where the predicate is specified in the ON clause can differ from the result of a query in which the predicate is specified in the WHERE clause. (When using an INNER JOIN, the results are always the same.) This is because the ON predicate is matching—it defines which rows from the nonpreserved side to match to those from the preserved side. The WHERE predicate is a filtering predicate—if a row from either side doesn't satisfy the WHERE predicate, the row is filtered out.

▶ Task 6: Write a SELECT Statement to Retrieve Customers Without Orders

1. In the query pane, type the following query after the **Task 5** description:

```
SELECT
c.custid, c.companyname
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE o.custid IS NULL;
```

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

   It is important to note that, when you are looking for a NULL, you should use the IS NULL operator, not the equality operator. The equality operator will always return UNKNOWN when you compare something to a NULL. It will even return UNKNOWN when you compare two NULLs.
   The choice of which attribute to filter from the nonpreserved side of the join is also important. You should choose an attribute that can have a NULL only when the row is an outer row (for example, a NULL originating from the base table). For this purpose, three cases are safe to consider:

   - A primary key column. A primary key column cannot be NULL. Therefore, a NULL in such a column can only mean that the row is an outer row.
   - A join column. If a row has a NULL in the join column, it is filtered out by the second phase of the join. So a NULL in such a column can only mean that it is an outer row.
   - A column defined as NOT NULL. A NULL in a column that is defined as NOT NULL can only mean that the row is an outer row.

Results: After this exercise, you should be able to filter rows of data from one or more tables by using WHERE predicates with logical operators.

Exercise 2: Write Queries that Sort Data Using an ORDER BY Clause

▶ Task 1: Write a SELECT Statement Using an ORDER BY Clause

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

```
SELECT
c.custid, c.contactname, o.orderid, o.orderdate
FROM Sales.Customers AS c
INNER JOIN Sales.Orders AS o ON c.custid = o.custid
WHERE
o.orderdate >= '20080401'
ORDER BY
o.orderdate DESC, c.custid ASC;
```

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

   Notice the date filter. It uses a literal (constant) of a date. SQL Server recognizes “20080401” as a character string literal, and not as a date and time literal. However, because the expression involves two operands of different types, one needs to be implicitly converted to the other’s type. In this example, the character string literal is converted to the column’s data type (DATETIME) because character strings are considered lower in terms of data type precedence—with respect to date and time data types. Data type precedence and working with date values are covered in detail in the next module.
   Also notice that the character string literal follows the format “yyyymmdd”. Using this format is a best practice because SQL Server knows how to convert it to the correct date, regardless of the language settings.

▶ Task 2: Apply the Needed Changes and Execute the T-SQL Statement

1. Highlight the written query under the **Task 2** description, and click **Execute**.
2. Observe the error message:

```
Invalid column name 'mgrlastname'.
```

3. This error occurred because the WHERE clause is evaluated before the SELECT clause and, at that time, the column did not have an alias. To fix this problem, you must use the source column name with the appropriate table alias. Modify the T-SQL statement to look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
WHERE
m.lastname = N'Buck';
```

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

▶ Task 3: Order the Result by the firstname Column

1. Highlight the previous query, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 3a** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement to remove the WHERE clause, and add an ORDER BY clause that uses the source column name of m.firstname. Your query should look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
ORDER BY
m.firstname;
```

4. Highlight the written query and click **Execute**.
5. Highlight the previous query, and on the **Edit** menu, click **Copy**.
6. In the query window, click the line after the **Task 3b** description, and on the **Edit** menu, click **Paste**.
7. Modify the ORDER BY clause so that it uses the alias for the same column (mgrfirstname). Your query should look like this:

```
SELECT
e.empid, e.lastname, e.firstname, e.title, e.mgrid,
m.lastname AS mgrlastname, m.firstname AS mgrfirstname
FROM HR.Employees AS e
INNER JOIN HR.Employees AS m ON e.mgrid = m.empid
ORDER BY
mgrfirstname;
```

8. Highlight the written query and click **Execute**.
9. Compare the results for Task 3a and 3b.
10. Why were you equally able to use a source column name or an alias column name?

Results: After this exercise, you should know how to use an ORDER BY clause.

Exercise 3: Write Queries that Filter Data Using the TOP Option

▶ Task 1: Writing Queries That Filter Data Using the TOP Clause

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:

```
SELECT TOP (20)
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC;
```

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

▶ Task 2: Use the OFFSET-FETCH Clause to Implement the Same Task

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

```
SELECT
orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate DESC
OFFSET 0 ROWS FETCH FIRST 20 ROWS ONLY;
```

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

   Remember that the OFFSET-FETCH clause was a new functionality in SQL Server 2012 and will not work in earlier versions. Unlike the TOP clause, the OFFSET-FETCH clause must be used with the ORDER BY clause.

▶ Task 3: Write a SELECT Statement to Retrieve the Most Expensive Products

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

```
SELECT TOP (10) PERCENT
productname, unitprice
FROM Production.Products
ORDER BY unitprice DESC;
```

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

   Implementing this task with the OFFSET-FETCH clause is possible but not easy because, unlike TOP, OFFSET-FETCH does not support a PERCENT option.

Results: After this exercise, you should have an understanding of how to apply the TOP option in the SELECT clause of a T-SQL statement.

Exercise 4: Write Queries that Filter Data Using the OFFSET-FETCH Clause

▶ Task 1: OFFSET-FETCH Clause to Fetch the First 20 Rows

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, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 0 ROWS FETCH FIRST 20 ROWS ONLY;
```

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

▶ Task 2: Use the OFFSET-FETCH Clause to Skip the First 20 Rows

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

```
SELECT
custid, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET 20 ROWS FETCH NEXT 20 ROWS ONLY;
```

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

▶ Task 3: Write a Generic Form of the OFFSET-FETCH Clause for Paging

1. The correct code is:

```
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY
```

2. To test the above expression, type the following query after the **Task 3** description:

```
DECLARE @pagenum int,
          @pagesize int;
SET @pagenum = 3;
SET @pagesize = 10;
SELECT
custid, orderid, orderdate
FROM Sales.Orders
ORDER BY orderdate, orderid
OFFSET (@pagenum - 1) * @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;
```

3. Highlight the query and click **Execute**.
4. Compare your results with the recommended results in file **D:\Labfiles\Lab05\Solution\84 - Lab Exercise 4- Task 3 Result**. Try changing the values for **@pagenum** and/or **@pagesize**, highlight the whole query (including the DECLARE and SET statements) and then click **Execute**.

Results: After this exercise, you will be able to use OFFSET-FETCH to work page-by-page through a result set returned by a SELECT statement.

Lab 6: Working with SQL Server 2016 Data Types

Exercise 1: Writing Queries That Return Date and Time Data

▶ Task 1: Prepare the Lab Environment

1. Ensure that the **MT17B-WS2016-NAT**, **20761C-MIA-DC**, and **20761C-MIA-SQL** virtual machines are running, and then log on to **20761C-MIA-SQL** as **ADVENTUREWORKS\Student** with the password **Pa55w.rd**.
2. In the **D:\Labfiles\Lab06\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, and when prompted, press any key.

▶ Task 2: Write a SELECT Statement to Retrieve Information About the Current Date

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\Lab06\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
CURRENT_TIMESTAMP AS currentdatetime,
CAST(CURRENT_TIMESTAMP AS DATE) AS currentdate,
CAST(CURRENT_TIMESTAMP AS TIME) AS currenttime,
YEAR(CURRENT_TIMESTAMP) AS currentyear,
MONTH(CURRENT_TIMESTAMP) AS currentmonth,
DAY(CURRENT_TIMESTAMP) AS currentday,
DATEPART(week, CURRENT_TIMESTAMP) AS currentweeknumber,
DATENAME(month, CURRENT_TIMESTAMP) AS currentmonthname;
```

   This query uses the CURRENT_TIMESTAMP function to return the current date and time. You can also use the SYSDATETIME function to get a more precise time element, compared to the CURRENT_TIMESTAMP function.
   Note that you cannot use the alias currentdatetime as the source in the second column calculation because SQL Server supports a concept called all-at-once operations. This means that all expressions appearing in the same logical query processing phase are evaluated as if they occurred at the same point in time. This concept explains why, for example, you cannot refer to column aliases assigned in the SELECT clause within the same SELECT clause, even if it seems intuitive that you should be able to.

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

▶ Task 3: Write a SELECT Statement to Return the Date Data Type

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

```
SELECT DATEFROMPARTS(2015, 12, 11) AS somedate;
SELECT CAST('20151211' AS DATE) AS somedate;
SELECT CONVERT(DATE, '12/11/2015', 101) AS somedate;
```

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

▶ Task 4: Write a SELECT Statement That Uses Different Date and Time Functions

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

```
SELECT
DATEADD(month, 3, CURRENT_TIMESTAMP) AS threemonths,
DATEDIFF(day, CURRENT_TIMESTAMP, DATEADD(month, 3, CURRENT_TIMESTAMP)) AS diffdays,
DATEDIFF(week, '19920404', '20110916') AS diffweeks,
DATEADD(day, 1, EOMONTH(CURRENT_TIMESTAMP,-1)) AS firstday;
```

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

▶ Task 5: Write a SELECT Statement to Show Whether a Table of Strings Can Be Used as Dates

1. Under the **Task 4** description, highlight the written query, and click **Execute**.
2. In the query pane, type the following queries after the **Task 4** description:

```
SELECT
isitdate,
CASE WHEN ISDATE(isitdate) = 1 THEN CONVERT(DATE, isitdate) ELSE NULL END AS
converteddate
FROM Sales.Somedates;
--Uses the TRY_CONVERT function:
SELECT
isitdate,
TRY_CONVERT(DATE, isitdate) AS converteddate
FROM Sales.Somedates;
```

   The second query uses the TRY_CONVERT function. This function returns a value cast to the specified data type if the casting succeeds; otherwise, it returns NULL. Don't worry if you do not recognize the type conversion functions, as they will be covered in the next module.

3. Highlight the written queries, and click **Execute**.
4. Observe the result and answer these questions:
   - What is the difference between the SYSDATETIME and CURRENT_TIMESTAMP functions?

     There are two main differences. First, the SYSDATETIME function provides a more precise time element compared to the CURRENT_TIMESTAMP function. Second, the SYSDATETIME function returns the data type **datetime2**(7), whereas the CURRENT_TIMESTAMP returns the data type **datetime**.

   - What is a language-neutral format for the data type date?

     You can use the formats 'YYYYMMDD' or 'YYYY-MM-DD'.

Results: After this exercise, you should be able to retrieve date and time data using T-SQL.

Exercise 2: Writing Queries That Use Date and Time Functions

▶ Task 1: Write a SELECT Statement to Retrieve Customers with Orders in a Given Month

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 DISTINCT
custid
FROM Sales.Orders
WHERE
YEAR(orderdate) = 2008
AND MONTH(orderdate) = 2;
```

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

   Note that, as a performance enhancement, you could also write a query that uses a range format that would utilize an index on Sales.Orders.orderdate. The query would then look like this:

```
SELECT DISTINCT
custid
FROM Sales.Orders
WHERE
orderdate >= '20080201'
AND orderdate < '20080301';
```

▶ Task 2: Write a SELECT Statement to Calculate the First and Last Day of the Month

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

```
SELECT
CURRENT_TIMESTAMP AS currentdate,
DATEADD (day, 1, EOMONTH(CURRENT_TIMESTAMP, -1)) AS firstofmonth,
EOMONTH(CURRENT_TIMESTAMP) AS endofmonth;
```

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

   This query uses the EOMONTH function, which was added in SQL Server 2012.

▶ Task 3: Write a SELECT Statement to Retrieve the Orders Placed in the Last Five Days of the Ordered Month

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

```
SELECT
orderid, custid, orderdate
FROM Sales.Orders
WHERE
DATEDIFF(
day,
orderdate,
EOMONTH(orderdate)
) < 5;
```

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

▶ Task 4: Write a SELECT Statement to Retrieve All Distinct Products Sold in the First 10 Weeks of the Year 2007

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

```
SELECT DISTINCT
d.productid
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE
DATEPART(week, orderdate) <= 10
AND YEAR(orderdate) = 2007;
```

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

Results: After this exercise, you should know how to use the date and time functions.

Exercise 3: Writing Queries That Return Character Data

▶ Task 1: Write a SELECT Statement to Concatenate Two Columns

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
CONCAT(contactname, N' (city: ', city, N')') AS contactwithcity
FROM Sales.Customers;
```

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

   An alternate way to write this query would be to use the + (plus) operator:

```
SELECT
contactname + N' (city: ' + city + N')' AS contactwithcity
```

5. FROM Sales.Customers;

▶ Task 2: Add an Additional Column to the Concatenated String Which Might Contain NULL

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

```
SELECT
CONCAT(contactname, N' (city: ', city,  N', region: ', region, N')') AS fullcontact
FROM Sales.Customers;
```

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

   An alternative way to write this query would be to use the + (plus) operator, which requires the COALESCE function to replace a NULL with an empty string. Later modules will include more examples of how to handle NULL.

```
SELECT
contactname + N' (city: ' + city + N', region: ' + COALESCE(region, '') + N')' AS
fullcontact
FROM Sales.Customers;
```

▶ Task 3: Write a SELECT Statement to Retrieve Customer Contacts Based on the First Character in the Contact Name

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

```
SELECT contactname, contacttitle
FROM Sales.Customers
WHERE contactname LIKE N'[A-G]%'
ORDER BY contactname;
```

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

Results: After this exercise, you should have an understanding of how to concatenate character data.

Exercise 4: Writing Queries That Use Character Functions

▶ Task 1: Write a SELECT Statement That Uses the SUBSTRING Function

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, after the **Task 1** description, type the following query:

```
SELECT
contactname,
SUBSTRING(contactname, 0, CHARINDEX(N',', contactname)) AS lastname
FROM Sales.Customers;
```

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

▶ Task 2: Write a Query to Retrieve the Contact's First Name Using SUBSTRING

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

```
SELECT
REPLACE(contactname, ',', '') AS newcontactname,
SUBSTRING(contactname, CHARINDEX(N',', contactname)+1, LEN(contactname)-
CHARINDEX(N',', contactname)+1) AS firstname
FROM Sales.Customers;
```

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

▶ Task 3: Write a SELECT Statement to Format the Customer ID

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

```
SELECT
custid,
N'C' + RIGHT(REPLICATE('0', 5) + CAST(custid AS VARCHAR(5)), 5) AS custnewid
FROM Sales.Customers;
```

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

   An alternative way to write this query would be to use the FORMAT function. The query would then look like this:

```
SELECT custid,
FORMAT(custid, N'\C00000') AS custnewid
FROM Sales.Customers;
```

▶ Task 4: Challenge: Write a SELECT Statement to Return the Number of Character Occurrences

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

```
SELECT
contactname,
LEN(contactname) - LEN(REPLACE(contactname, 'a', '')) AS numberofa
FROM Sales.Customers
ORDER BY numberofa DESC;
```

   This elegant solution first returns the number of characters in the contact name, and then subtracts the number of characters in the contact name without the character 'a'. The result is stored in a new column named numberofa.

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

Results: After this exercise, you should have an understanding of how to use the character functions.

Lab 7: Using DML to Modify Data

Exercise 1: Inserting Records with DML

▶ Task 1: Prepare the Lab Environment

The exercises will be performed within the TempDB database so that none of the real data is affected. Two scripts are used to set up the environment for the lab—both are included in the project for the lab, along with a sample solution for each exercise.

If you need to start again, open and execute the cleanup script, followed by the setup script; you have a clean environment and can try again.

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\Lab07\Starter** folder, right-click **Setup.cmd**, and then click **Run as administrator**.
3. In the **User Account Control** dialog box, click **Yes**.
4. When the script has finished, press Enter.

▶ Task 2: Insert a Row

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine using Windows authentication.
2. In the **File** menu, point to **Open**, and click **Project/Solution**.
3. In the **Open Project** window, open the project **D:\Labfiles\Lab07\Starter\Project\Project.ssmssln**.
4. In Solution Explorer, expand **Queries**, and double-click **01 Setup.sql**.
5. On the toolbar, click **Execute**. When the script has executed, you should get the messages indicating (9 row(s) affected), (88 row(s) affected) and (3 row(s) affected).
6. Close the **01 Setup.sql** pane and open a new query window by clicking the **New Query** icon.
7. When the query window opens, type **USE TempDB**, followed by **GO** on the next line, and then, on the toolbar, click **Execute**.
8. Below the GO statement in the open window, type the following query:

```
INSERT INTO HR.Employees
(
  Title
, titleofcourtesy
, FirstName
, Lastname
, hiredate
, birthdate
, address
, city
, country
, phone
)
VALUES
(
  'Sales Representative'
```

```
, 'Mr'
, 'Laurence'
, 'Grider'
, '04/04/2013'
, '10/25/1975'
, '1234 1st Ave. S.E. '
, 'Seattle'
, 'USA'
, '(206)555-0105'
);
```

9. Click **Execute**.
10. Make sure the row has been inserted, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

▶ Task 3: Insert a Row with a SELECT Statement As the Data Provider

1. Click **New Query**.
2. In the query pane, type the following query:

```
USE TempDB
GO
INSERT INTO Sales.Customers
(
  Companyname
, contactname
, contacttitle
, address
, city
, region
, postalcode
, country
, phone
, fax
)
SELECT
  Companyname
, contactname
, contacttitle
, address
, city
, region
, postalcode
, country
, phone
, fax
FROM dbo.PotentialCustomers;
```

3. Click **Execute**.
4. Make sure that the rows have been inserted, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

   How could you have checked the data as it was transferred by the query? Remember the OUTPUT command? If not, you can look at the exercise solution labeled **42 - Lab Exercise 1b Solution.sql**.

Results: After successfully completing this exercise, you will have one new employee and three new customers.

Exercise 2: Update and Delete Records Using DML

▶ Task 1: Update Rows

1. Click **New Query**.
2. In the query pane, type the following query:

```
Use TempDB
GO
UPDATE Sales.Customers
SET contacttitle='Sales Consultant'
WHERE city='Berlin'AND contacttitle='Sales Representative';
```

3. Click **Execute**.
4. Make sure the rows have been modified, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

   How could you have checked the data as it was transferred by the query? Remember the OUTPUT command? If not, you can look at the exercise solution labeled **43 - Lab Exercise 2 Solution.sql**.

▶ Task 2: Delete Rows

1. Click **New Query**.
2. In the query pane, type the following query:

```
USE TempDB
GO
DELETE FROM dbo.PotentialCustomers
WHERE contactname
IN( 'Taylor, Maurice','Mallit, Ken', 'Tiano, Mike' );
```

3. Click **Execute**.
4. Make sure the rows have been deleted, and then close the window. You will be asked if you want to save the query—you can choose where to save it and what to call it.

   How could you have checked the data as it was transferred by the query? Remember the OUTPUT command? If not, you can look at the exercise solution labeled **44 - Lab Exercise 2b Solution.sql**.

Results: After successfully completing this exercise, you will have updated all the records in the Customers table that have a city of Berlin and a contacttitle of Sales Representative, to now have a contacttitle of Sales Consultant. You will also have deleted the three records in the PotentialCustomers table, which have already been added to the Customers table.

Lab 8: Using Built-in Functions

Exercise 1: Writing Queries That Use Conversion 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\Lab08\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. Wait for the script to finish, and press Enter.

▶ Task 2: Write a SELECT Statement that Uses the CAST or CONVERT Function

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\Lab08\Starter\Project** folder, and then double-click **Project.ssmssln**.
4. In Solution Explorer, expand the **Queries** folder, 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 N'The unit price for the ' + productname + N' is ' + CAST(unitprice AS
NVARCHAR(10)) + N' $.' AS productdesc
FROM Production.Products;
```

   This query uses the CAST function rather than the CONVERT function. It is better to use the CAST function because it is an ANSI SQL standard. You should use the CONVERT function only when you need to apply a specific style during a conversion.

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

▶ Task 3: Write a SELECT Statement to Filter Rows Based on Specific Date Information

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

```
SELECT orderid, orderdate, shippeddate, COALESCE(shipregion, 'No region') AS
shipregion
FROM Sales.Orders
WHERE
orderdate >= CONVERT(DATETIME, '4/1/2007', 101)
AND orderdate <= CONVERT(DATETIME, '11/30/2007', 101)
AND shippeddate > DATEADD(DAY, 30, orderdate);
```

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

3. Note that you could also write a solution using the PARSE function. The query would look like this:

```
SELECT orderid, orderdate, shippeddate, COALESCE(shipregion, 'No region') AS
shipregion
FROM Sales.Orders
WHERE
orderdate >= PARSE('4/1/2007' AS DATETIME USING 'en-US')
AND orderdate <= PARSE('11/30/2007' AS DATETIME USING 'en-US')
AND shippeddate > DATEADD(DAY, 30, orderdate);
```

▶ Task 4: Write a SELECT Statement to Convert the Phone Number Information to an Integer Value

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

```
SELECT
CONVERT(INT, REPLACE(REPLACE(REPLACE(REPLACE(phone, N'-', N''), N'(', ''), N')', ''),
' ', '')) AS phonenoasint
FROM Sales.Customers;
```

   This query is trying to use the CONVERT function to convert phone numbers that include characters, such as hyphens and parentheses, into an integer value.

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

   Observe the error message:

```
Conversion failed when converting the nvarchar value '67.89.01.23' to data type int.
```

   Because you want to retrieve rows without conversion errors and have a NULL for those that produce a conversion error, you can use the TRY_CONVERT function.

3. Modify the query to use the TRY_CONVERT function. The query should look like this:

```
SELECT
TRY_CONVERT(INT, REPLACE(REPLACE(REPLACE(REPLACE(phone, N'-', N''), N'(', ''), N')',
''), ' ', '')) AS phonenoasint
FROM Sales.Customers;
```

4. Highlight the written query, and click **Execute**. Observe the result. The rows that could not be converted have a NULL.

Results: After this exercise, you should be able to use conversion functions.

Exercise 2: Writing Queries That Use Logical Functions

▶ Task 1: Write a SELECT Statement to Mark Specific Customers Based on Their Country and Contact Title

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
IIF(country = N'Mexico' AND contacttitle = N'Owner', N'Target group', N'Other') AS
segmentgroup, custid, contactname
FROM Sales.Customers;
```

   The IIF function was new in SQL Server 2012. It was added mainly to support migrations from Microsoft Access to SQL Server. You can use the CASE expression to achieve the same result.

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

▶ Task 2: Modify the T-SQL Statement to Mark Different Customers

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

```
SELECT
IIF(contacttitle = N'Owner' OR region IS NOT NULL, N'Target group', N'Other') AS
segmentgroup,  custid,  contactname
FROM Sales.Customers;
```

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

▶ Task 3: Create Four Groups of Customers

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

```
SELECT CHOOSE(custid % 4 + 1, N'Group One', N'Group Two', N'Group Three', N'Group
Four') AS segmentgroup, custid, contactname
FROM Sales.Customers;
```

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

Results: After this exercise, you should know how to use the logical functions.

Exercise 3: Writing Queries That Test for Nullability

▶ Task 1: Write a SELECT Statement to Retrieve the Customer Fax Information

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

```
SELECT contactname, COALESCE(fax, N'No information') AS faxinformation
FROM Sales.Customers;
```

   This query uses the COALESCE function to retrieve customers' fax information.

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

```
SELECT contactname, ISNULL(fax, N'No information') AS faxinformation
FROM Sales.Customers;
```

   This query uses the ISNULL function. What is the difference between the ISNULL and COALESCE functions? COALESCE is a standard ANSI SQL function and ISNULL is not. So you should use the COALESCE function.

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

▶ Task 2: Write a Filter for a Variable That Could Be a Null

1. Highlight the query provided under the **Task 2** description, and click **Execute**.
2. Modify the query so that it looks like this:

```
DECLARE @region AS NVARCHAR(30) = NULL;
SELECT
custid, region
FROM Sales.Customers
WHERE region = @region OR (region IS NULL AND @region IS NULL);
```

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

▶ Task 3: Write a SELECT Statement to Return All the Customers That Do Not Have a Two-Character Abbreviation for the Region

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

```
SELECT custid, contactname, city, region
FROM Sales.Customers
WHERE
region IS NULL
OR LEN(region) <> 2;
```

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

Results: After this exercise, you should have an understanding of how to test for nullability.

Lab 9: Grouping and Aggregating Data

Exercise 1: Writing Queries That Use the GROUP BY Clause

▶ 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\Lab09\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 Different Groups of Customers

1. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance 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\Lab09\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 click **Execute**.
6. In the query pane, type the following query after the **Task 2** description:

```
SELECT
o.custid, c.contactname
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname;
```

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

▶ Task 3: Add an Additional Column From the Sales.Customers Table

1. Highlight the previous query, 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 so that it adds an additional column. Your query should look like this:

```
SELECT
o.custid, c.contactname, c.city
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname;
```

4. Highlight the written query, and click **Execute**.
5. Observe the error message:

```
Column 'Sales.Customers.city' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.
```

Why did the query fail?

In a grouped query, there will be an error if you refer to an attribute that is not in the GROUP BY list (such as the city column) or not an input to an aggregate function in any clause that is processed after the GROUP BY clause.

6. Modify the SQL statement to include the city column in the GROUP BY clause. Your query should look like this:

```
SELECT
o.custid, c.contactname, c.city
FROM Sales.Orders AS o
INNER JOIN Sales.Customers AS c ON c.custid = o.custid
WHERE o.empid = 5
GROUP BY o.custid, c.contactname, c.city;
```

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

▶ Task 4: Write a SELECT Statement to Retrieve the Customers with Orders for Each Year

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

```
SELECT
custid, YEAR(orderdate) AS orderyear
FROM Sales.Orders
WHERE empid = 5
GROUP BY custid, YEAR(orderdate)
ORDER BY custid, orderyear;
```

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

▶ Task 5: Write a SELECT Statement to Retrieve Groups of Product Categories Sold in a Specific Year

1. In the query pane, type the following query after the **Task 5** description:

```
SELECT
c.categoryid, c.categoryname
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
INNER JOIN Production.Categories AS c ON c.categoryid = p.categoryid
WHERE orderdate >= '20080101' AND orderdate < '20090101'
GROUP BY c.categoryid, c.categoryname;
```

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

Note: Important note regarding the use of the DISTINCT clause:
In all the tasks in Exercise 1, you could use the DISTINCT clause in the SELECT clause as an alternative to using a grouped query. This is possible because aggregate functions are not being requested.

Results: After this exercise, you should be able to use the GROUP BY clause in the T-SQL statement.

Exercise 2: Writing Queries That Use Aggregate Functions

▶ Task 1: Write a SELECT statement to Retrieve the Total Sales Amount Per Order

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
o.orderid, o.orderdate, SUM(d.qty * d.unitprice) AS salesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.orderid, o.orderdate
ORDER BY salesamount DESC;
```

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

▶ Task 2: Add Additional Columns

1. Highlight the previous query, and on the **Edit** menu, click **Copy**.
2. In the query window, click the line after the **Task 2** description, and on the **Edit** menu, click **Paste**.
3. Modify the T-SQL statement so that it adds extra columns. Your query should look like this:

```
SELECT
o.orderid, o.orderdate,
SUM(d.qty * d.unitprice) AS salesamount,
COUNT(*) AS noofoderlines,
AVG(d.qty * d.unitprice) AS avgsalesamountperorderline
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.orderid, o.orderdate
ORDER BY salesamount DESC;
```

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

▶ Task 3: Write a SELECT Statement to Retrieve the Sales Amount Value Per Month

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

```
SELECT
YEAR(orderdate) * 100 + MONTH(orderdate) AS yearmonthno,
SUM(d.qty * d.unitprice) AS saleamountpermonth
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY YEAR(orderdate), MONTH(orderdate)
ORDER BY yearmonthno;
```

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

▶ Task 4: Write a SELECT Statement to List All Customers with the Total Sales Amount and Number of Order Lines Added

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

```
SELECT
c.custid, c.contactname,
SUM(d.qty * d.unitprice) AS totalsalesamount,
MAX(d.qty * d.unitprice) AS maxsalesamountperorderline,
COUNT(*) AS numberofrows,
COUNT(o.orderid) AS numberoforderlines
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON o.custid = c.custid
LEFT OUTER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY c.custid, c.contactname
ORDER BY totalsalesamount;
```

2. Highlight the written query, and click **Execute**.
3. Observe the result. Notice that the values in the **numberofrows** and **numberoforderlines** columns are different. Why? All aggregate functions ignore NULLs except COUNT(*), which is why you received the value 1 for the **numberofrows** column. When you used the **orderid** column in the COUNT function, you received the value 0 because the **orderid** is NULL for customers without an order.

Exercise 3: Writing Queries That Use Distinct Aggregate Functions

▶ Task 1: Modify a SELECT Statement to Retrieve the Number of Customers

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. Highlight the provided T-SQL statement after the **Task 1** description, and click **Execute**.
4. Observe the result. Notice that the number of orders is the same as the number of customers. Why? You are using the aggregate COUNT function on the **orderid** and **custid** columns and, because every order has a customer, the COUNT function returns the same value. It does not matter if there are multiple orders for the same customer, because you are not using a DISTINCT clause inside the aggregate function. To get the correct number of distinct customers, you can modify the provided T-SQL statement to include a DISTINCT clause.
5. Modify the provided T-SQL statement to include a DISTINCT clause. The query should look like this:

```
SELECT
YEAR(orderdate) AS orderyear,
COUNT(orderid) AS nooforders,
COUNT(DISTINCT custid) AS noofcustomers
FROM Sales.Orders
GROUP BY YEAR(orderdate);
```

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

▶ Task 2: Write a SELECT Statement to Analyze Segments of Customers

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

```
SELECT
SUBSTRING(c.contactname,1,1) AS firstletter,
COUNT(DISTINCT c.custid) AS noofcustomers,
COUNT(o.orderid) AS nooforders
FROM Sales.Customers AS c
LEFT OUTER JOIN Sales.Orders AS o ON o.custid = c.custid
GROUP BY SUBSTRING(c.contactname,1,1)
ORDER BY firstletter;
```

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

▶ Task 3: Write a SELECT Statement to Retrieve Additional Sales Statistics

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

```
SELECT
c.categoryid, c.categoryname,
SUM(d.qty * d.unitprice) AS totalsalesamount, COUNT(DISTINCT o.orderid) AS
nooforders,
SUM(d.qty * d.unitprice) / COUNT(DISTINCT o.orderid) AS avgsalesamountperorder
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
INNER JOIN Production.Categories AS c ON c.categoryid = p.categoryid
WHERE orderdate >= '20080101' AND orderdate < '20090101'
GROUP BY c.categoryid, c.categoryname;
```

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

Results: After this exercise, you should have an understanding of how to apply a DISTINCT aggregate function.

Exercise 4: Writing Queries That Filter Groups with the HAVING Clause

▶ Task 1: Write a SELECT Statement to Retrieve the Top 10 Customers

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 TOP (10)
o.custid,
SUM(d.qty * d.unitprice) AS totalsalesamount
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
ORDER BY totalsalesamount DESC;
```

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

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

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

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
GROUP BY o.orderid, o.empid;
```

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

▶ Task 3: Apply Additional Filtering

1. Highlight the previous query, 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 to apply additional filtering. Your query should look like this:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080101' AND o.orderdate < '20090101'
GROUP BY o.orderid, o.empid
HAVING SUM(d.qty * d.unitprice) >= 10000;
```

4. Highlight the written query, and click **Execute**.
5. Modify the T-SQL statement to include an additional filter to retrieve only orders handled by the employee whose ID is 3. Your query should look like this:

```
SELECT
o.orderid,
o.empid,
SUM(d.qty * d.unitprice) as totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE
o.orderdate >= '20080101' AND o.orderdate <= '20090101'
AND o.empid = 3
GROUP BY o.orderid, o.empid
HAVING SUM(d.qty * d.unitprice) >= 10000;
```

   In this query, the predicate logic is applied in the WHERE clause. You could also write the predicate logic inside the HAVING clause. Which do you think is better?

   Unlike with **orderdate** filtering, with **empid** filtering, the result is going to be correct either way because you are filtering by an element that appears in the GROUP BY list. Conceptually, it seems more intuitive to filter as early as possible. This query then applies the filtering in the WHERE clause because it will be logically applied before the GROUP BY clause. Do not forget, though, that the actual processing in the SQL Server engine could be different.

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

▶ Task 4: Retrieve the Customers with More Than 25 Orders

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

```
SELECT
o.custid,
MAX(orderdate) AS lastorderdate,
SUM(d.qty * d.unitprice) AS totalsalesamount
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY o.custid
HAVING COUNT(DISTINCT o.orderid) > 25;
```

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

Results: After this exercise, you should have an understanding of how to use the HAVING clause.

Lab 10: Using Subqueries

Exercise 1: Writing Queries That Use Self-Contained Subqueries

▶ 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\Lab10\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.
4. At the command prompt, press any key.

▶ Task 2: Write a SELECT Statement to Retrieve the Last Order Date

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 click **Project/Solution**.
3. In the **Open Project** dialog box, navigate to the **D:\Labfiles\Lab10\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 click **Execute**.
6. In the query pane, type the following query after the **Task 1** description:

```
SELECT MAX(orderdate) AS lastorderdate
FROM Sales.Orders;
```

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

▶ Task 3: Write a SELECT Statement to Retrieve All Orders Placed on the Last Order Date

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

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
orderdate = (SELECT MAX(orderdate) FROM Sales.Orders);
```

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

▶ Task 4: Observe the T-SQL Statement Provided by the IT Department

1. Highlight the provided T-SQL statement under the **Task 3** description, and click **Execute**.
2. Modify the query to filter customers whose contact name starts with the letter B. Your query should look like this:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid =
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'B%'
);
```

3. Highlight the written query, and click **Execute**.
4. Observe the error message:

```
Subquery returned more than 1 value. This is not permitted when the subquery follows
=, !=, <, <= , >, >= or when the subquery is used as an expression.
```

   Why did the query fail? It failed because the subquery returned more than one row. To fix this problem, you should replace the = operator with an IN operator.

5. Modify the query so that it uses the IN operator. Your query should look like this:

```
SELECT
orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE
custid IN
(
SELECT custid
FROM Sales.Customers
WHERE contactname LIKE N'B%'
);
```

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

▶ Task 5: Write A SELECT Statement to Analyze Each Order's Sales as a Percentage of the Total Sales Amount

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

```
SELECT
o.orderid,
SUM(d.qty * d.unitprice) AS totalsalesamount,
SUM(d.qty * d.unitprice) /
(
SELECT SUM(d.qty * d.unitprice)
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080501' AND orderdate < '20080601'
) * 100. AS salespctoftotal
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.orderdate >= '20080501' AND orderdate < '20080601'
GROUP BY o.orderid;
```

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

Results: After this exercise, you should be able to use self-contained subqueries in T-SQL statements.

Exercise 2: Writing Queries That Use Scalar and Multiresult Subqueries

▶ Task 1: Write a SELECT Statement to Retrieve Specific Products

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
productid, productname
FROM Production.Products
WHERE
productid IN
(
SELECT productid
FROM Sales.OrderDetails
WHERE qty > 100
);
```

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

▶ Task 2: Write a SELECT Statement to Retrieve Those Customers Without Orders

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

```
SELECT
custid, contactname
FROM Sales.Customers
WHERE custid NOT IN
(
SELECT custid
FROM Sales.Orders
);
```

2. Highlight the written query, and click **Execute**.
3. Observe the result. Notice there are two customers without an order.

▶ Task 3: Add a Row and Rerun the Query That Retrieves Those Customers Without Orders

1. Highlight the provided T-SQL statement under the **Task 3** description, and click **Execute**. This code inserts an additional row that has a NULL in the **custid** column of the **Sales.Orders** table.
2. Highlight the query in **Task 2**, and on the **Edit** menu, click **Copy**.
3. In the query window, under the **Task 3** description, click the line after the provided T-SQL statement, and on the **Edit** menu, click **Paste**.
4. Highlight the written query, and click **Execute**.

5. Notice that you have an empty result despite having two rows when you first ran the query in task 2. Why did you have an empty result this time? There is an issue with the NULL in the new row you added because the **custid** column is the only one that is part of the subquery. The IN operator supports three-valued logic (TRUE, FALSE, UNKNOWN). Before you apply the NOT operator, the logical meaning of UNKNOWN is that you can't tell for sure whether the customer ID appears in the set, because the NULL could represent that customer ID as well as anything else. As a more tangible example, consider the expression 22 NOT IN (1, 2, NULL). If you evaluate each individual expression in the parentheses to its truth value, you will get NOT (FALSE OR FALSE OR UNKNOWN), which translates to NOT UNKNOWN, which evaluates to UNKNOWN. The tricky part is that negating UNKNOWN with the NOT operator still yields UNKNOWN; and UNKNOWN is filtered out in a query filter. In short, when you use the NOT IN predicate against a subquery that returns at least one NULL, the outer query always returns an empty set.
6. To solve this problem, modify the T-SQL statement so that the subquery does not return NULLs. Your query should look like this:

```
SELECT
custid, contactname
FROM Sales.Customers
WHERE custid NOT IN
(
SELECT custid
FROM Sales.Orders
WHERE custid IS NOT NULL
);
```

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

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

Exercise 3: Writing Queries That Use Correlated Subqueries and an EXISTS Predicate

▶ Task 1: Write a SELECT Statement to Retrieve the Last Order Date for Each Customer

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:

```
SELECT
c.custid, c.contactname,
(
SELECT MAX(o.orderdate)
FROM Sales.Orders AS o
WHERE o.custid = c.custid
) AS lastorderdate
FROM Sales.Customers AS c;
```

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

▶ Task 2: Write a SELECT Statement That Uses the EXISTS Predicate to Retrieve Those Customers Without Orders

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

```
SELECT c.custid, c.contactname
FROM Sales.Customers AS c
WHERE
NOT EXISTS (SELECT * FROM Sales.Orders AS o WHERE o.custid = c.custid);
```

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

3. Notice that you achieved the same result as the modified query in exercise 2, task 3, but without a filter to exclude NULLs. Why didn't you need to explicitly filter out NULLs? The EXISTS predicate uses two-valued logic (TRUE, FALSE) and checks only if the rows specified in the correlated subquery exist. Another benefit of using the EXISTS predicate is better performance. The SQL Server engine knows it is enough to determine whether the subquery returns at least one row or none, so it doesn't need to process all qualifying rows.

▶ Task 3: Write a SELECT Statement to Retrieve Customers Who Bought Expensive Products

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

```
SELECT c.custid, c.contactname
FROM Sales.Customers AS c
WHERE
EXISTS (
SELECT *
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
WHERE o.custid = c.custid
AND d.unitprice > 100.
AND o.orderdate >= '20080401'
);
```

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

▶ Task 4: Write a SELECT Statement to Display the Total Sales Amount and the Running Total Sales Amount for Each Order Year

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

```
SELECT
YEAR(o.orderdate) as orderyear,
SUM(d.qty * d.unitprice) AS totalsales,
(
SELECT SUM(d2.qty * d2.unitprice)
FROM Sales.Orders AS o2
INNER JOIN Sales.OrderDetails AS d2 ON d2.orderid = o2.orderid
WHERE YEAR(o2.orderdate) <= YEAR(o.orderdate)
) AS runsales
FROM Sales.Orders AS o
INNER JOIN Sales.OrderDetails AS d ON d.orderid = o.orderid
GROUP BY YEAR(o.orderdate)
ORDER BY orderyear;
```

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

▶ Task 5: Clean the Sales.Customers Table

- Under the **Task 5** description, highlight the provided T-SQL statement, and then click **Execute**.

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

Lab 11: 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.

3. Modify the previous T-SQL statement by including the TOP (100) PERCENT option. The query should look like this:

```
ALTER VIEW Production.ProductsBeverages AS
SELECT TOP(100) PERCENT
productid, productname, supplierid, unitprice, discontinued
FROM Production.Products
WHERE categoryid = 1
ORDER BY productname;
```

4. Highlight the written query, and click **Execute**.
5. Observe the result. If you now write a query against the Production.ProductsBeverages view, is it guaranteed that the retrieved rows will be sorted by productname? If you do not specify the ORDER BY clause in the T-SQL statement against the view, there is no guarantee that the retrieved rows will be sorted. It is important to remember that any order of the rows in the output is considered valid, and no specific order is guaranteed. Therefore, when querying a table expression, you should not assume any order unless you specify an ORDER BY clause in the outer query.

▶ 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 12: 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 13: Using Window Ranking, Offset, and Aggregate Functions

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 14: 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 15: 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 the OFFSET-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 16: 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 17: 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 18: 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.