Module 11: Using Table Expressions

Contents:

Module Overview

Previously in this course, you learned about using subqueries as an expression that returned results to an outer calling query. Like subqueries, table expressions are query expressions, but table expressions extend this idea by allowing you to name them and work with the results as you would with data in any valid relational table. Microsoft® SQL Server® 2016 supports four types of table expressions: derived tables, common table expressions (CTEs), views, and inline table-valued functions (TVFs). In this module, you will learn to work with these forms of table expressions and how to use them to help create a modular approach to writing queries.

Objectives

After completing this module, you will be able to:

- Create simple views and write queries against them.
- Create simple inline TVFs and write queries against them.
- Write queries that use derived tables.
- Write queries that use CTEs.

> **Note:** Some of the examples used in this module have been adapted from samples published in *Microsoft SQL Server 2008 T-SQL Fundamentals* (Microsoft Press 2009).

Lesson 1: Using Views

The lifespans of some table expressions are limited to the query in which they are defined and invoked. Views and TVFs, however, can be persistently stored in a database and reused. A view is a table expression whose definition is stored in a SQL Server database. Like derived tables and CTEs, views are defined with SELECT statements.

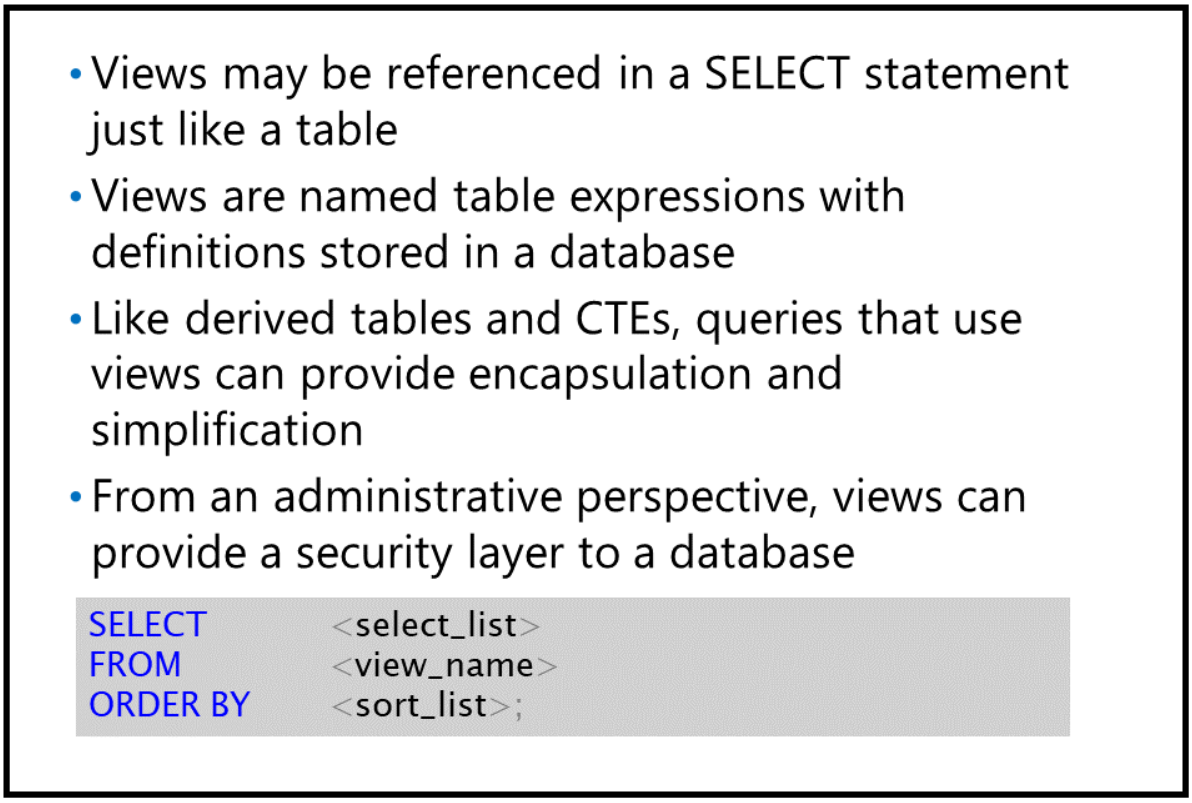
This provides not only the benefits of modularity and encapsulation possible with derived table and CTEs, but also adds reusability, in addition to extra security beyond that provided with query-scoped table expressions.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that return results from views.
- Create simple views.

Writing Queries That Return Results from Views



A view is a named table expression whose definition is stored as metadata in a SQL Server database. Views can be used as a source for queries in much the same way as tables themselves. However, views do not persistently store data; the definition of the view is unpacked at runtime and the source objects are queried.

> **Note:** In an indexed view, data is materialized in the view. Indexed views are beyond the scope of this course.

Querying a View Syntax

```
SELECT <select_list>
FROM <view_name>
ORDER BY <sort_list>;
```

Note that an ORDER BY clause is used in this sample syntax to emphasize the point that, as a table expression, there is no sort order included in the definition of a view. This will be discussed later in this lesson.

Querying a View Example

```
SELECT custid, ordermonth, qty
FROM Sales.CustOrders;
```

The partial results are indistinguishable from any other table-based query:

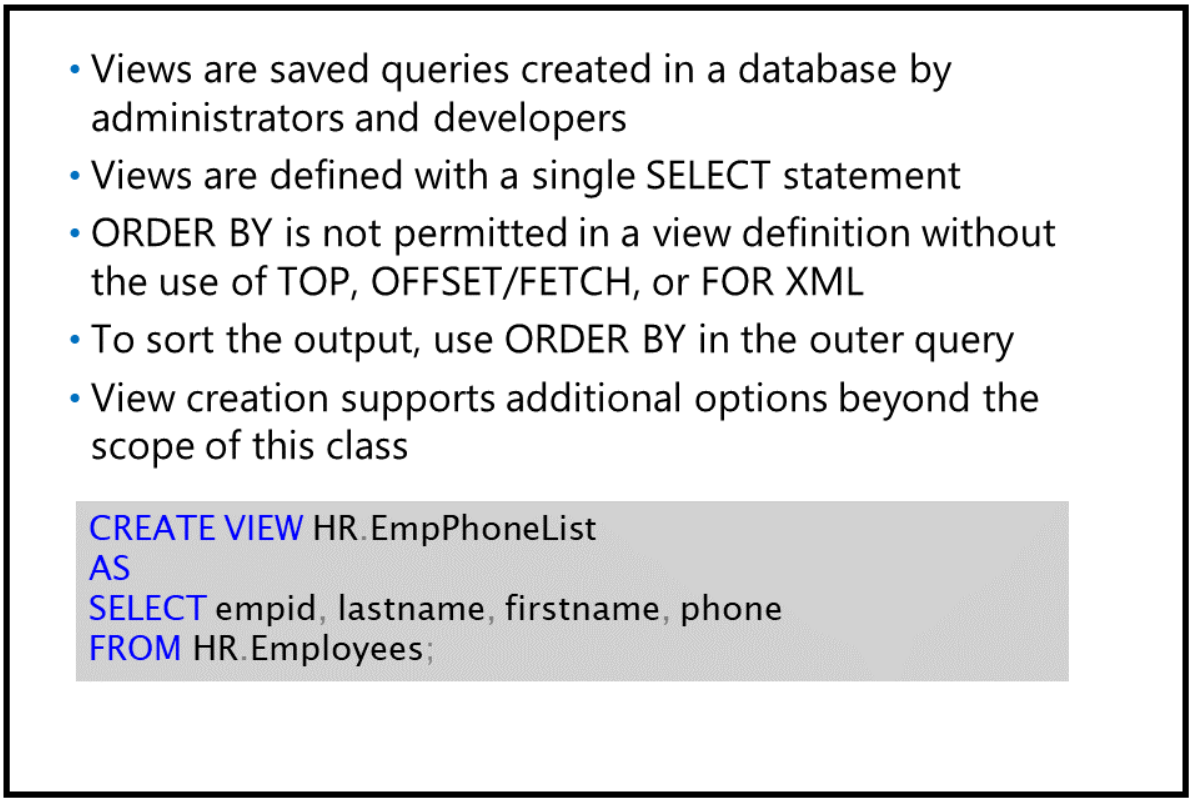
custid	ordermonth	qty
7	2006-07-01 00:00:00.000	50
13	2006-07-01 00:00:00.000	11
14	2006-07-01 00:00:00.000	57

The apparent similarity between a table and a view provides an important benefit—an application can be written to use views instead of the underlying tables, shielding the application from changes to the tables. Providing the view continues to present the same structure to the calling application, the application will receive consistent results. Views can be considered an application programming interface (API) to a database for purposes of retrieving data.

Administrators can also use views as a security layer, granting users permissions to select from a view without providing permissions on the underlying source tables.

> **Additional Reading:** For more information on database security, go to course 20764C: *Administering a SQL Database Infrastructure*.

Creating Simple Views



To use a view in your queries, it must be created by a database developer or administrator with appropriate permission in the database. While coverage of database security is beyond the scope of this course, you will have permission to create views in the lab database.

To store a view definition, use the CREATE VIEW T-SQL statement to name and store a single SELECT statement. Note that the ORDER BY clause is not permitted in a view definition unless the view uses a TOP, OFFSET/FETCH, or FOR XML element.

CREATE VIEW Syntax

```
CREATE VIEW <schema_name.view_name> [<column_alias_list>]
[WITH <view_options>]
AS select_statement;
```

> **Note:** This lesson covers the basics of creating views for the purposes of discussion about querying them only. For more information on views and view options, go to course 20762B: *Developing Microsoft SQL Server Databases*.

CREATE VIEW Example

```
CREATE VIEW Sales.CustOrders
AS
SELECT
  O.custid,
```

```
  DATEADD(month, DATEDIFF(month, 0, O.orderdate), 0) AS ordermonth,
  SUM(OD.qty) AS qty
FROM Sales.Orders AS O
  JOIN Sales.OrderDetails AS OD
    ON OD.orderid = O.orderid
GROUP BY custid, DATEADD(month, DATEDIFF(month, 0, O.orderdate), 0);
```

You can query system metadata by querying system catalog views such as sys.views, which you will learn about in a later module.

Querying a View Example

```
SELECT custid, ordermonth, qty
FROM Sales.CustOrders;
```

Demonstration: Using Views

In this demonstration, you will see how to create views.

Demonstration Steps

Create Views

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 **D:\Demofiles\Mod11\Setup.cmd** as an administrator.
3. In the **User Account Control** dialog box, click **Yes**.
4. At the command prompt, type **y**, press Enter, and then wait for the script to finish.
5. Start SQL Server Management Studio and connect to the **MIA-SQL** database engine instance using Windows authentication.
6. Open the **Demo.ssmssln** solution in the **D:\Demofiles\Mod11\Demo** folder.
7. Open the **11 - Demonstration A.sql** script file.
8. Select the code under the comment **Step 1**, and then click **Execute**.
9. Select the code under the comment **Step 2**, and then click **Execute**.
10. Select the code under the comment **Step 3**, and then click **Execute**.
11. Select the code under the comment **Step 4**, and then click **Execute**.
12. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Discovery

Your DBAs want to grant access to Sales users on the Customers table in the Sales database. However, they also need to prevent Sales users from reading values in the Customers.Relationship column. How can they set up this access?

Show solution Reset

Create a view that queries the Customers table but does not include the Relationship column. Grant access to the view and not to the table.

Lesson 2: Using Inline TVFs

An inline TVF is a form of table expression with several properties in common with views. Like a view, the definition of a TVF is stored as a persistent object in a database. Also like a view, an inline TVF encapsulates a single SELECT statement, returning a virtual table to the calling query. A primary distinction between a view and an inline TVF is that the latter can accept input parameters and refer to them in the embedded SELECT statement.

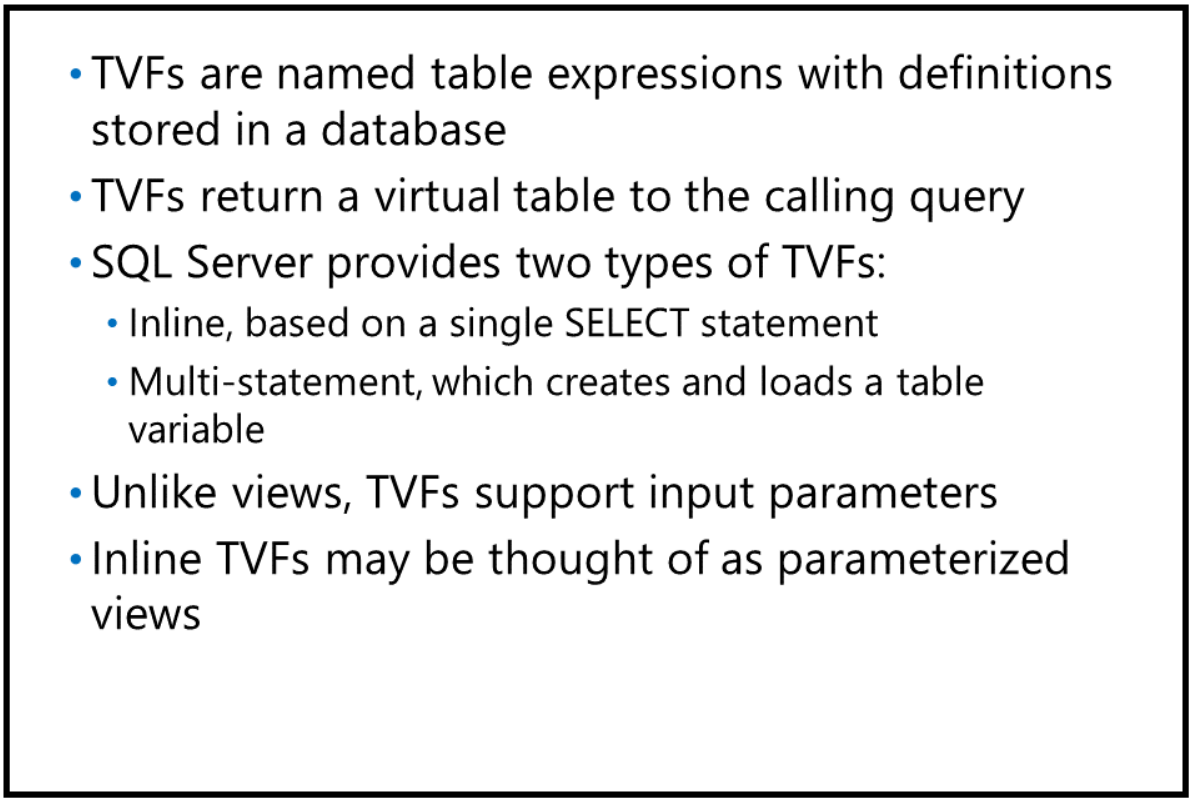
In this lesson, you will learn how to create basic inline TVFs and write queries that return results from them.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the structure and usage of inline TVFs.
- Use the CREATE FUNCTION statement to create simple inline TVFs.
- Write queries that return results from inline TVFs.

Writing Queries That Use Inline TVFs



Inline TVFs are named table expressions whose definitions are stored persistently in a database that can be queried in much the same way as a view. This enables reuse and centralized management of code in a way that is not possible for derived tables and CTEs as query-scoped table expressions.

> **Note:** SQL Server supports several types of user-defined functions. In addition to inline TVFs, users can create scalar functions, multi-statement TVFs, and functions written in the .NET Common Language Runtime (CLR). For more information on these functions, go to course 20762B: *Developing Microsoft SQL Server Databases*.

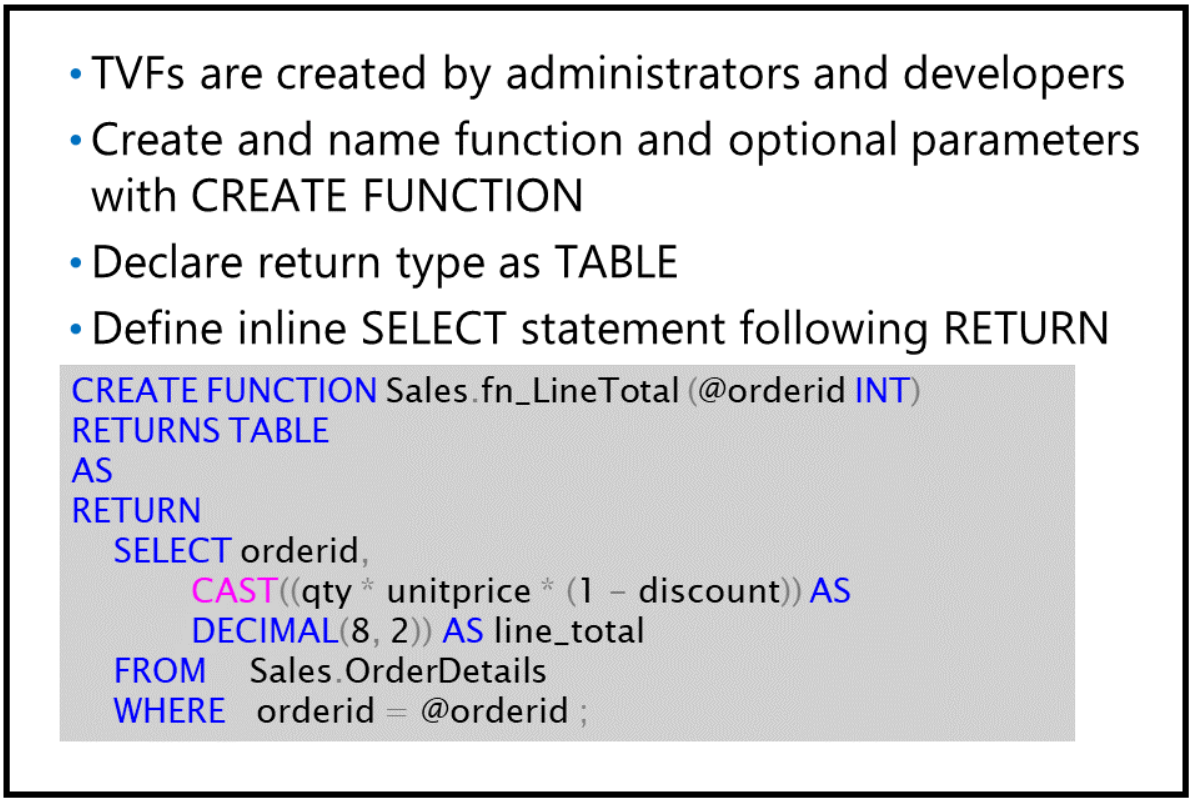
One of the key distinctions between views and inline TVFs is that the latter can accept input parameters. Therefore, you may think of inline TVFs conceptually as parameterized views and choose to use them in place of views when flexibility of input is preferred.

Additional reading can be found in Microsoft Docs:

CREATE FUNCTION (Transact-SQL)

http://go.microsoft.com/fwlink/?LinkID=402772

Creating Simple Inline TVFs



To use inline TVFs in your queries, they must be created by a database developer or administrator with appropriate permission in the database. While coverage of database security is beyond the scope of this course, you will have permission to create TVFs in the lab database.

To store an inline TVF view definition:

- Use the CREATE FUNCTION T-SQL statement to name and store a single SELECT statement with optional parameters.
- Use RETURNS TABLE to identify this function as a TVF.
- Enclose the SELECT statement inside parentheses following the RETURN keyword to make this an inline function.

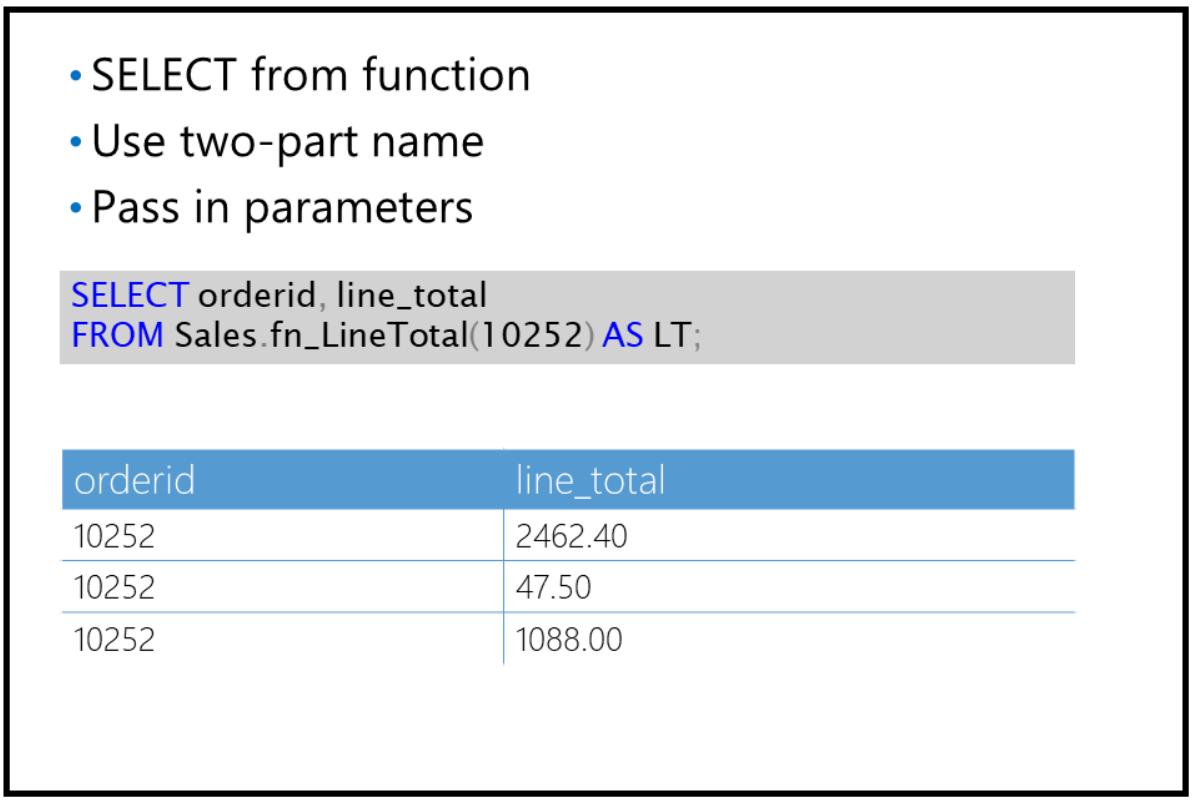
CREATE FUNCTION Syntax for Inline Table-Valued Functions

```
CREATE FUNCTION <schema.name>
(@<parameter_name> AS <data_type>, ...)
RETURNS TABLE
AS
RETURN (<SELECT_expression>);
```

Inline Table-Valued Function Example

```
CREATE FUNCTION Production.TopNProducts
(@t AS INT)
RETURNS TABLE
AS
RETURN
        (SELECT TOP (@t) productid, productname, unitprice
                FROM Production.Products
                ORDER BY unitprice DESC);
```

Retrieving from Inline TVFs



After creating an inline TVF, you can invoke it by selecting from it, as you would a view. If there is an argument, you need to enclose it in parentheses. Multiple arguments need to be separated by commas.

Querying an Inline TVF

```
SELECT * FROM Production.TopNProducts(3)
```

The results:

productid	productname	unitprice
38	Product QDOMO	263.50
29	Product VJXYN	123.79

productid	productname	unitprice
9	Product AOZBW	97.00

(3 row(s) affected)

> **Note:** You use a two-part name when calling a user-defined function.

Demonstration: Inline TVFs

In this demonstration, you will see how to create inline TVFs.

Demonstration Steps

Create Inline TVFs

1. In Solution Explorer, open the **21 - Demonstration B.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Select the best answer

From the following statements, select the one that is true of TVFs but not true of Views.

- Stored persistently in the database.
- Can accept input parameters.
- Can be referred to in a FROM clause, like a table.
- Does not store data in the database but queries the database whenever it is called.

Check answer Show solution Reset

One of the main differences between Views and TVFs is that TVFs can accept input parameters, while Views cannot.

Lesson 3: Using Derived Tables

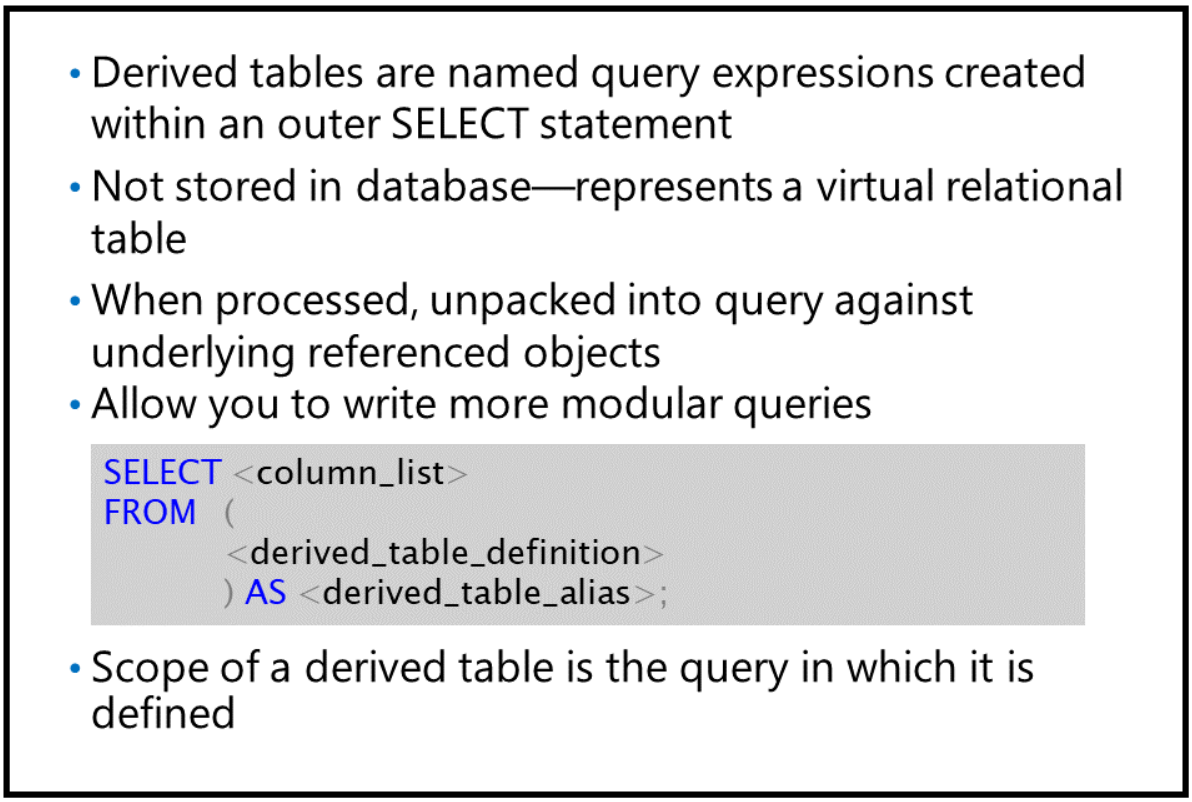
In this lesson, you will learn how to write queries that create derived tables in the FROM clause of an outer query. You will also learn how to return results from the table expression defined in the derived table.

Lesson Objectives

After completing this lesson, you will be able to:

- Write queries that create and retrieve results from derived tables.
- Describe how to provide aliases for column names in derived tables.
- Pass arguments to derived tables.
- Describe nesting and reuse behavior in derived tables.

Writing Queries with Derived Tables



Earlier in this course, you learned about subqueries, which are queries nested within other SELECT statements. Like subqueries, you create derived tables in the FROM clause of an outer SELECT statement. Unlike subqueries, you write derived tables using a named expression that is logically equivalent to a table and may be referenced as a table elsewhere in the outer query. Derived tables allow you to write T-SQL statements that are more modular, helping you break down complex queries into more manageable parts. Using derived tables in your queries can also provide workarounds for some of the restrictions imposed by the logical order of query processing, such as the use of column aliases.

Derived Table Syntax

```
SELECT <outer query column list>
FROM (SELECT <inner query column list>
        FROM <table source>) AS <derived table alias>
```

The following example uses a derived table to retrieve information about orders placed per year by distinct customers. The inner query builds a set of orders and places it into the derived table's derived year. The outer query operates on the derived table and summarizes the results.

Derived Table Example

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

The results:

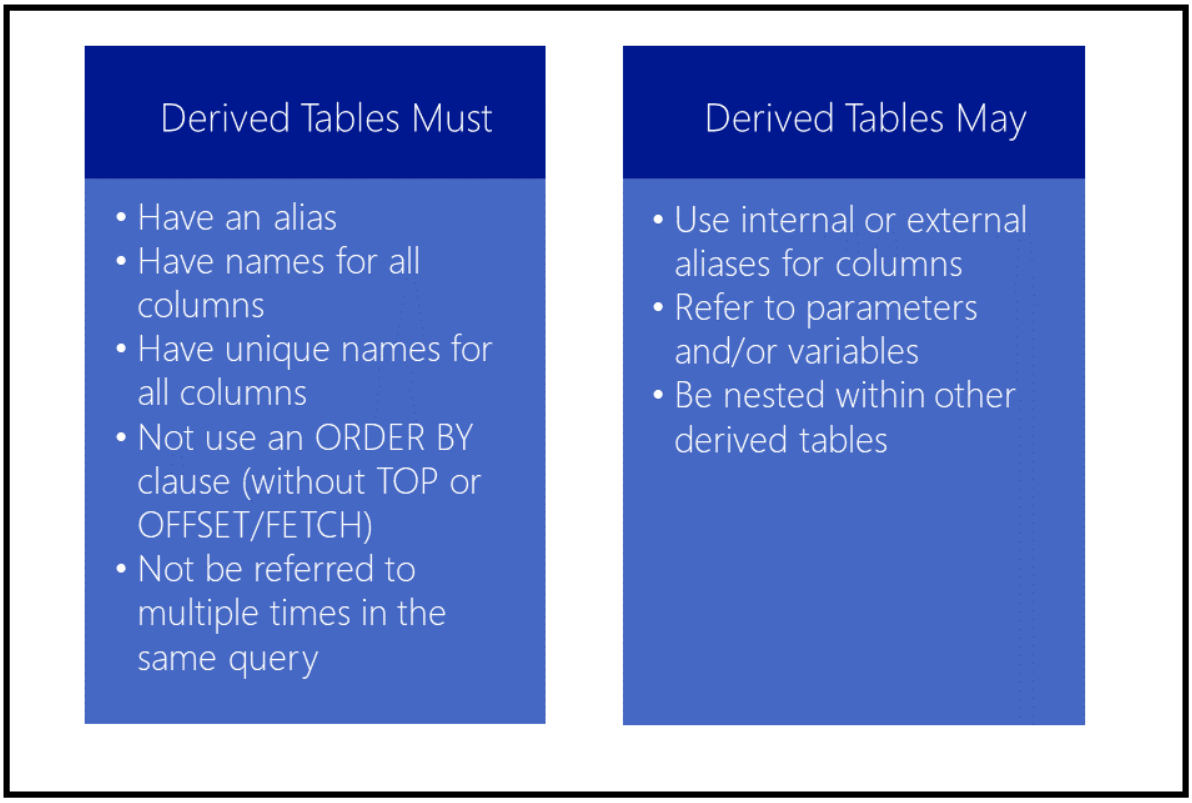
orderyear	cust_count
2006	67
2007	86
2008	81

(3 row(s) affected)

When writing queries that use derived tables, consider the following:

- Derived tables are not stored in the database. Therefore, no special security privileges are required to write queries using derived tables, other than the rights to select from the source objects.
- A derived table is created at the time of execution of the outer query and goes out of scope when the outer query ends.
- Derived tables do not necessarily have an impact on performance, compared to the same query expressed differently. When the query is processed, the statement is unpacked and evaluated against the underlying database objects.

Guidelines for Derived Tables

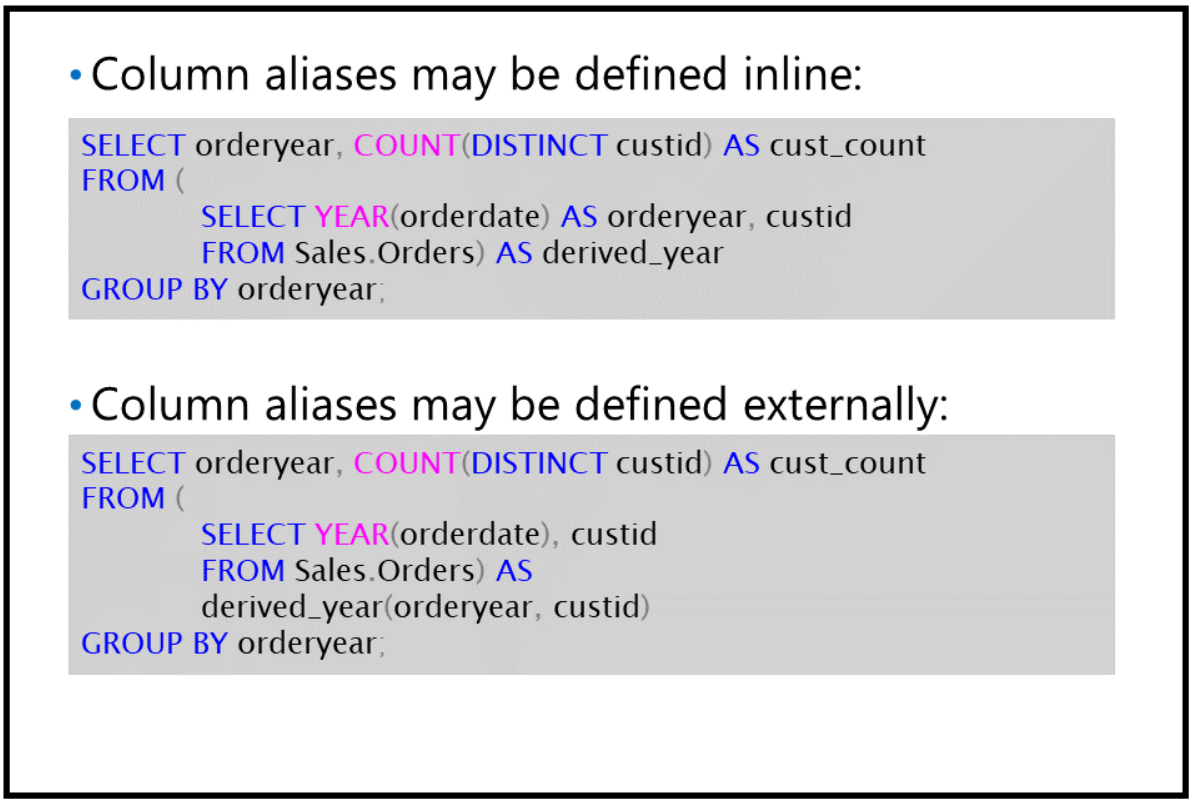


When writing queries that use derived tables, keep the following guidelines in mind:

- The nested SELECT statement that defines the derived table must have an alias assigned to it. The outer query will use the alias in its SELECT statement in much the same way you refer to aliased tables joined in a FROM clause.
- All columns referenced in the derived table's SELECT clause should be assigned aliases, a best practice that is not always required in T-SQL. Each alias must be unique within the expression. The column aliases may be declared inline with the columns or externally to the clause. You will see examples of this in the next topic.
- The SELECT statement that defines the derived table expression may not use an ORDER BY clause, unless it also includes a TOP operator, an OFFSET/FETCH clause, or a FOR XML clause. As a result, there is no sort order provided by the derived table. You sort the results in the outer query.
- The SELECT statement that defines the derived table may be written to accept arguments in the form of local variables. If the SELECT statement is embedded in a stored procedure, the arguments may be written as parameters for the procedure. You will see examples of this later in the module.
- Derived table expressions that are nested within an outer query can contain other derived table expressions. Nesting is permitted, but it is not recommended due to increased complexity and reduced readability.
- A derived table may not be referred to multiple times within an outer query. If you need to manipulate the same results, you will need to define the derived table expression every time, such as on each side of a JOIN operator.

> **Note:** You will see examples of multiple usage of the same derived table expression in a query in the demonstration for this lesson.

Using Aliases for Column Names in Derived Tables



To create aliases, you can use one of two methods—inline or external.

Alias Syntax

```
SELECT <outer query column list>
FROM (SELECT <col1> AS <alias>, <col2> AS <alias>...
        FROM <table_source>);
```

Alias Example

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders) AS derived_year
GROUP BY orderyear;
```

A partial result for the inner query displays the following:

orderyear	custid
2006	85
2006	79
2006	34

The inner results are passed to the outer query, which operates on the derived table's orderyear and custid columns:

orderyear	cust_count
2006	67
2007	86
2008	81

Declared Aliases with Derived Tables Syntax

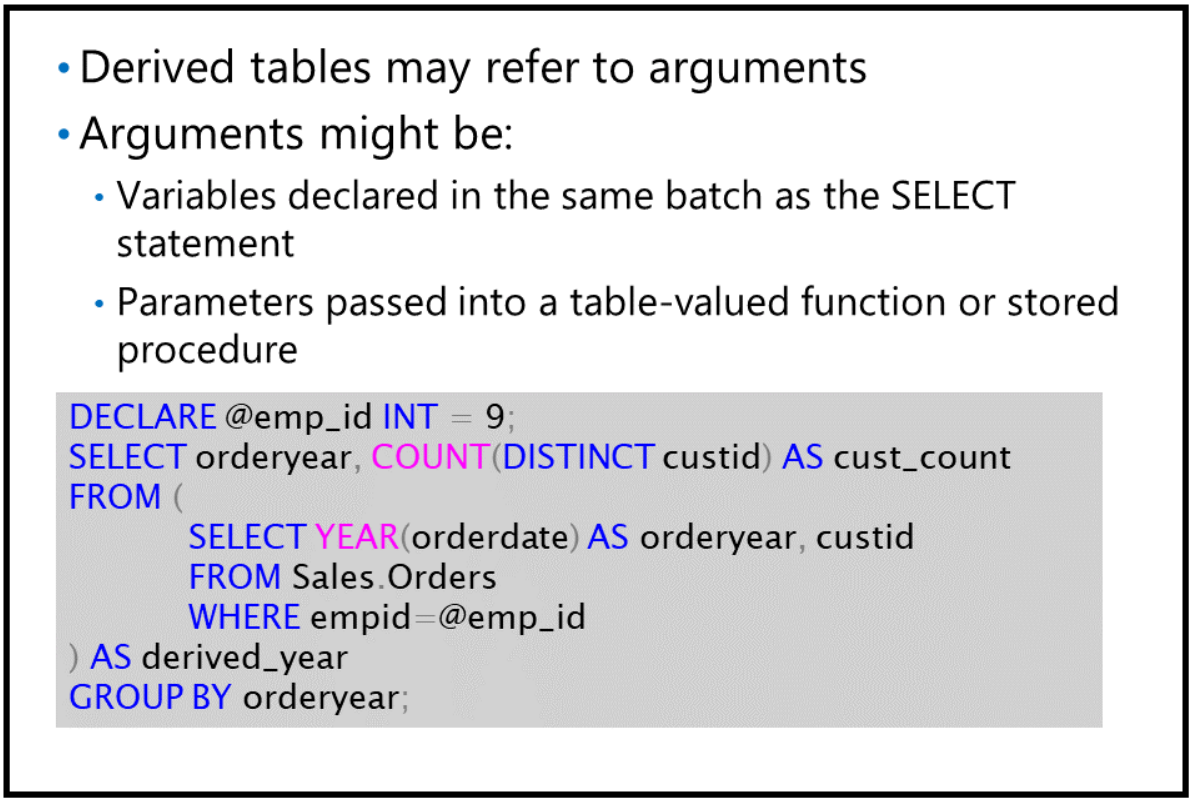
```
SELECT <outer query column list>
FROM (SELECT <col1>, <col2>..
        FROM <table_source>) AS <derived_table_alias>(<col1_alias>, <col2_alias>);
```

Declared Aliases with Derived Tables Example

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (SELECT YEAR(orderdate), custid
        FROM Sales.Orders) AS derived_year(orderyear, custid)
GROUP BY orderyear;
```

> **Note:** When using external aliases, if the inner query is executed separately, the aliases will not be returned to the outer query. For ease of testing and readability, it is recommended that you use inline rather than external aliases.

Passing Arguments to Derived Tables



Derived tables in SQL Server can accept arguments passed in from a calling routine, such as a T-SQL batch, function, or a stored procedure. Derived tables can be written with local variables serving as placeholders in their code. At runtime, the placeholders can be replaced with values supplied in the batch or with values passed as parameters to the stored procedure that invoked the query. This will allow your code to be reused more flexibly than rewriting the same query with different values each time.

> **Note:** The use of parameters in functions and stored procedures will be covered later in this course. This lesson focuses on writing table expressions that can accept arguments.

Passing Arguments to Derived Tables

```
DECLARE @emp_id INT = 9; --declare and assign the variable
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
        SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders
        WHERE empid=@emp_id --use the variable to pass a value to the derived table query
) AS derived_year
GROUP BY orderyear;
GO
```

The results:

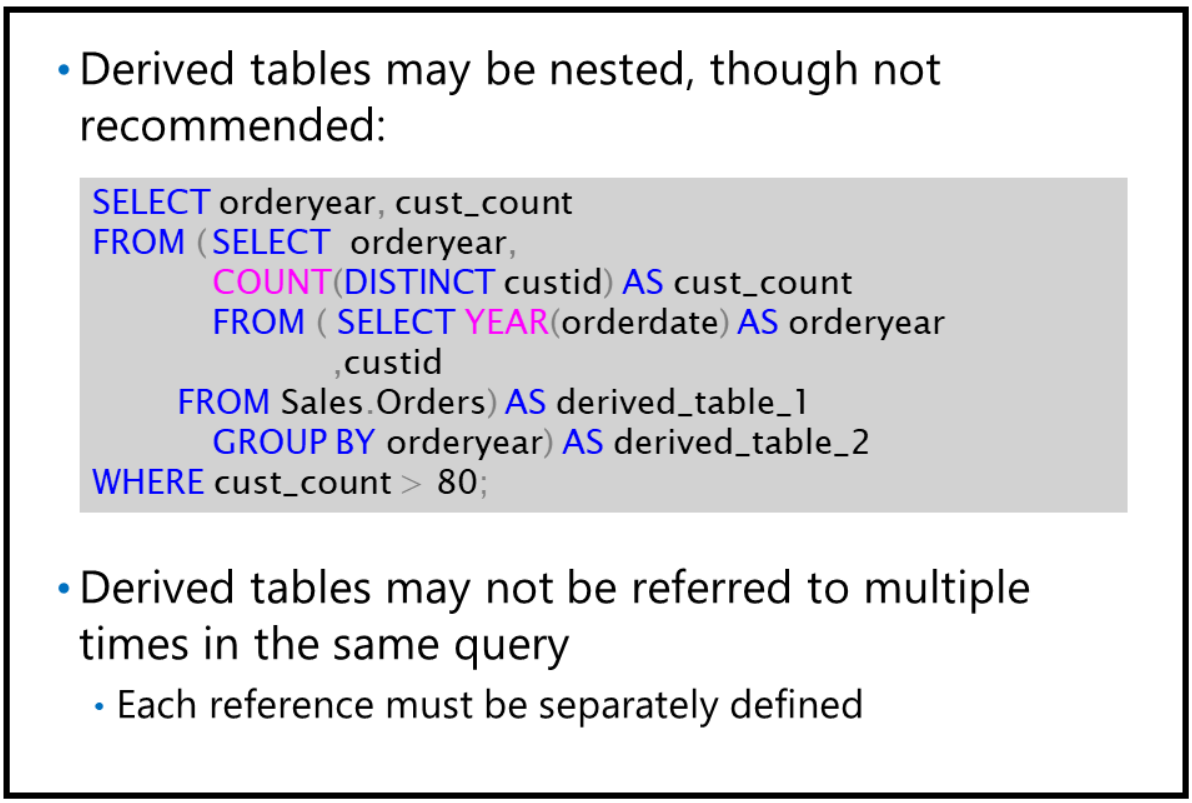
orderyear	cust_count

orderyear	cust_count
2006	5
2007	16
2008	16

(3 row(s) affected)

> **Note:** You will learn more about declaring variables, executing T-SQL code in batches, and working with stored procedures later in this class.

Nesting and Reusing Derived Tables



Since a derived table is itself a complete query expression, that query can refer to a derived table expression. This creates a nesting scenario, which while possible, is not recommended for reasons of code maintenance and readability.

Nested Derived Tables

```
SELECT orderyear, cust_count
FROM  (
        SELECT  orderyear, COUNT(DISTINCT custid) AS cust_count
        FROM (
                SELECT YEAR(orderdate) AS orderyear ,custid
        FROM Sales.Orders) AS derived_table_1
```

```
        GROUP BY orderyear) AS derived_table_2
WHERE cust_count > 80;
```

Logically, the innermost query is processed first, returning these partial results as derived_table_1:

orderyear	custid
2006	85
2006	79
2006	34

Next, the middle query runs, grouping and aggregating the results into derived_table_2:

orderyear	cust_count
2006	67
2007	86
2008	81

Finally, the outer query runs, filtering the output:

orderyear	cust_count
2007	86
2008	81

As you can see, while is possible to nest derived tables, it does add complexity.

While nesting derived tables is possible, references to the same derived table from multiple clauses of an outer query can be challenging. Since the table expression is defined in the FROM clause, subsequent phases of the query can see it, but it cannot be referenced elsewhere in the same FROM clause.

For example, a derived table defined in a FROM clause may be referenced in a WHERE clause, but not in a JOIN in the same FROM clause that defines it. The derived table must be defined separately, and multiple copies of the code maintained. For an alternative approach that allows reuse without maintaining separate copies of the derived table definition, see the CTE discussion later in this module.

Question: How could you rewrite the previous example to eliminate one level of nesting?

Demonstration: Using Derived Tables

In this demonstration, you will see how to write queries that create derived tables.

Demonstration Steps

Write Queries that Create Derived Tables

1. In Solution Explorer, open the **31 - Demonstration C.sql** script file.

2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Select the code under the comment **Step 4**, and then click **Execute**.
6. Keep SQL Server Management Studio open for the next demonstration.

Check Your Knowledge

Discovery

You are troubleshooting the following query, which returns an error:

```
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM (
    SELECT YEAR(orderdate) AS orderyear, custid
    FROM Sales.Orders
    WHERE empid = 354
  ORDER BY YEAR(orderdate)
) AS derived_year
GROUP BY orderyear;
```

How can you resolve the error?

Show solution Reset

Remove the ORDER BY clause from the derive table query. ORDER BY clauses are not permitted within derived tables. You can move the ORDER BY clause after the GROUP BY clause in the outer query.

Lesson 4: Using CTEs

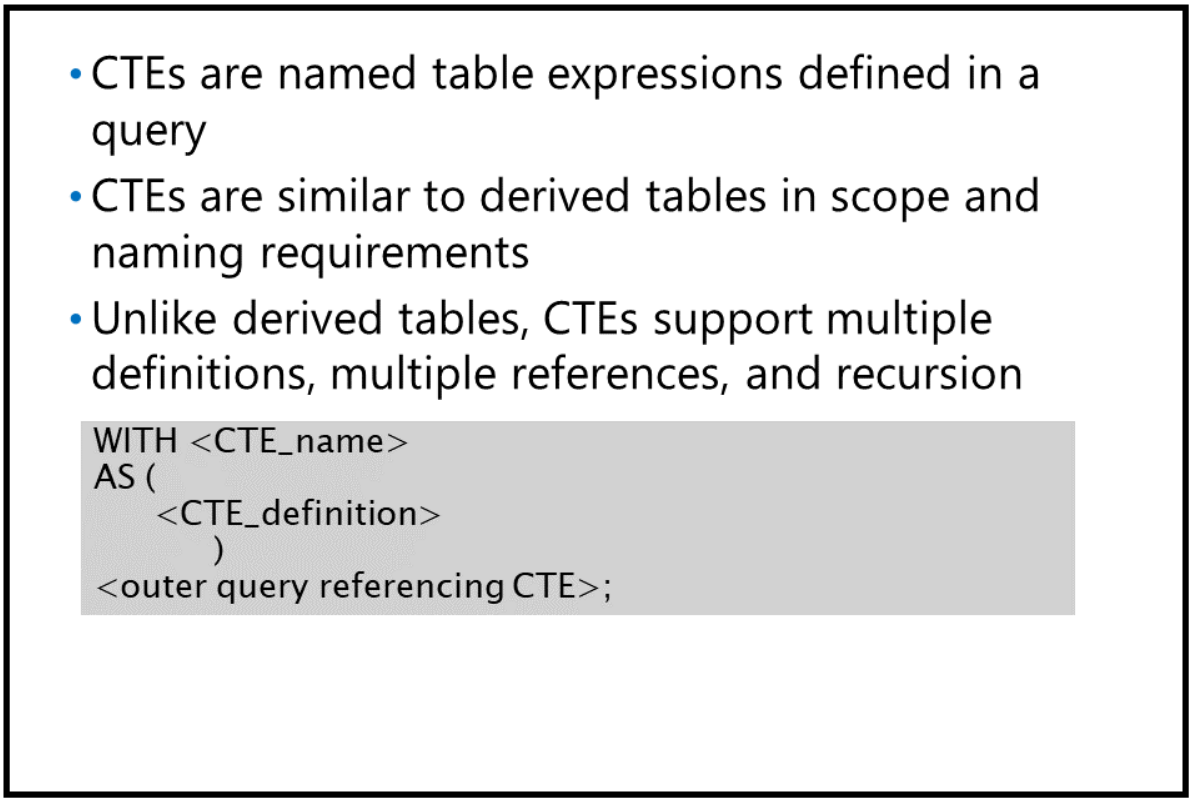
Another form of table expression provided by SQL Server is the CTE. Similar in some ways to derived tables, CTEs provide a mechanism for defining a subquery that may then be used elsewhere in a query. Unlike a derived table, a CTE is defined at the beginning of a query and may be referenced multiple times in the outer query.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the use of CTEs.
- Write queries that create CTEs and return results from the table expression.
- Describe how a CTE can be reused multiple times by the same outer query.

Writing Queries with CTEs



CTEs are named expressions defined in a query. Like subqueries and derived tables, CTEs provide a means to break down query problems into smaller, more modular units.

When writing queries with CTEs, consider the following guidelines:

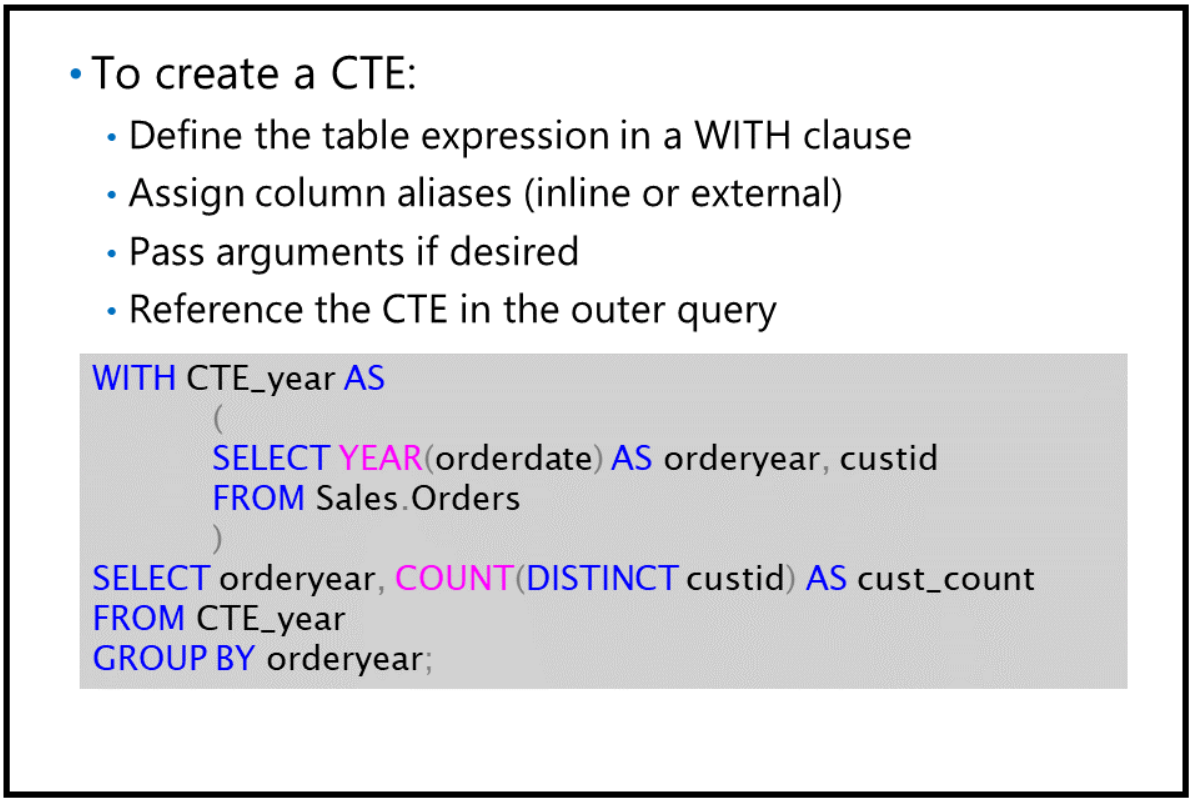
- Like derived tables, CTEs are limited in scope to the execution of the outer query. When the outer query ends, so does the CTE's lifetime.
- CTEs require a name for the table expression, in addition to unique names for each of the columns referenced in the CTE's SELECT clause.
- CTEs may use inline or external aliases for columns.
- Unlike a derived table, a CTE may be referenced multiple times in the same query with one definition. Multiple CTEs may also be defined in the same WITH clause.
- CTEs support recursion, in which the expression is defined with a reference to itself. Recursive CTEs are beyond the scope of this course.

For additional reading on recursive CTEs, see the SQL Server Technical Documentation:

Recursive Queries Using Common Table Expressions

http://go.microsoft.com/fwlink/?LinkID=402773

Creating Queries with Common Table Expressions



CTE Syntax

```
WITH    <CTE_name>
AS      ( <CTE_definition> )
```

CTE Example

```
WITH CTE_year --name the CTE
AS -- define the subquery
(
        SELECT YEAR(orderdate) AS orderyear, custid
        FROM Sales.Orders
)
SELECT orderyear, COUNT(DISTINCT custid) AS cust_count
FROM CTE_year --reference the CTE in the outer query
GROUP BY orderyear;
```

The results:

orderyear	cust_count
2006	67
2007	86

orderyear	cust_count
2008	81

(3 row(s) affected)

Demonstration: Using CTEs

In this demonstration, you will see how to write queries that create CTEs.

Demonstration Steps

Write Queries that Create CTEs

1. In Solution Explorer, open the **41 - Demonstration D.sql** script file.
2. Select the code under the comment **Step 1**, and then click **Execute**.
3. Select the code under the comment **Step 2**, and then click **Execute**.
4. Select the code under the comment **Step 3**, and then click **Execute**.
5. Close SQL Server Management Studio without saving any files.

Check Your Knowledge

Select the best answer

Which of the following features is required for a CTE query?

- The query must have a WITH … AS clause.
- The query must include a GROUP BY clause.
- The query must include a CREATE FUNCTION statement.
- The query must include a nested derived query.

Check answer Show solution Reset

CTEs can be identified by the WITH … AS clause—this defines the CTE so that it can be reused in the subsequent SELECT query.

Lab: 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.

Lab Setup

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

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Result: 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

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Result: 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

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Result: 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

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

Detailed Steps ▲

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.

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

Module Review and Takeaways

In this module, you have learned how to:

- Create simple views and write queries against them.
- Create simple inline TVFs and write queries against them.
- Write queries that use derived tables.
- Write queries that use CTEs.

Review Question(s)

Check Your Knowledge

Discovery

When would you use a CTE rather than a derived table for a query?

Show solution Reset

CTEs may be written once, referenced multiple times in a query.

Check Your Knowledge

Discovery

Which table expressions allow variables to be passed in as parameters to the expression?

Show solution Reset

Table-valued functions.