Data Management Using Microsoft SQL Server

Session: 10

Using Views, Stored Procedures, and Querying Metadata

Objectives

- Define views
- Describe the technique to create, alter, and drop views
- Define stored procedures
- Explain the types of stored procedures
- Describe the procedure to create, alter, and execute stored procedures
- Describe nested stored procedures
- Describe querying SQL Server metadata
  - System Catalog views and functions
  - Querying Dynamic Management Objects

Introduction

- An SQL Server database has two main categories of objects:
  - those that store data.
  - those that access, manipulate, or provide access to data.

- Views and stored procedures belong to this latter category.

Views

- A view is a virtual table that is made up of selected columns from one or more tables.
- The tables from which the view is created are referred to as base tables.
- These base tables can be from different databases.
- A view can also include columns from other views created in the same or a different database.
- A view can have a maximum of 1024 columns.
- The data inside the view comes from the base tables that are referenced in the view definition.
- The rows and columns of views are created dynamically when the view is referenced.

Creating Views 1-3

A view is created using the `CREATE VIEW` statement and it can be created only in the current database.

SQL Server verifies the existence of objects that are referenced in the view definition.

- The syntax used to create a view is as follows:

Syntax:

```
CREATE VIEW <view_name>
AS <select_statement>
```

where,

`view_name`: specifies the name of the view.

`select_statement`: specifies the `SELECT` statement that defines the view.

Creating Views 2-3

- Following code snippet creates a view from the **Product** table to display only the product id, product number, name, and safety stock level of products.

```
CREATE VIEW vwProductInfo AS
SELECT ProductID, ProductNumber, Name, SafetyStockLevel
FROM Production.Product;
GO
```

- The code in the following code snippet is used to display the details of the **vwProductInfo** view.

```
SELECT * FROM vwProductInfo
```

Creating Views 3-3

- The result will show the specified columns of all the products from the **`Product`** table.
- The following figure shows a part of the output.

	ProductID	ProductNumber	Name	SafetyStockLevel
1	1	AR-5381	Adjustable Race	1000
2	2	BA-8327	Bearing Ball	1000
3	3	BE-2349	BB Ball Bearing	800
4	4	BE-2908	Headset Ball Bearings	800
5	316	BL-2036	Blade	800
6	317	CA-5965	LL Crankarm	500
7	318	CA-6738	ML Crankarm	500
8	319	CA-7457	HL Crankarm	500
9	320	CB-2903	Chainring Bolts	1000
10	321	CN-6137	Chainring Nut	1000

Creating Views Using JOIN Keyword 1-5

The JOIN keyword can also be used to create views.

- The syntax used to create a view with the JOIN keyword is as follows:

Syntax:

```
CREATE VIEW <view_name>
AS
SELECT * FROM table_name1
JOIN table_name2
ON table_name1.column_name = table_name2.column_name
```

where,

view_name: specifies the name of the view.

table_name1: specifies the name of first table.

JOIN: specifies that two tables are joined using JOIN keyword.

table_name2: specifies the name of the second table.

Creating Views Using JOIN Keyword 2-5

- Following code snippet creates a view named **vwPersonDetails** with specific columns from the Person and Employee tables.
- The JOIN and ON keywords join the two tables based on BusinessEntityID column.

```
CREATE VIEW vwPersonDetails
AS
SELECT
p.Title
,p.[FirstName]
,p.[MiddleName]
,p.[LastName]
,e.[JobTitle]
FROM [HumanResources].[Employee] e
INNER JOIN [Person].[Person] p
ON p.[BusinessEntityID] = e.[BusinessEntityID]
GO
```

- This view will contain the columns Title, FirstName, MiddleName, and LastName from the Person table and JobTitle from the Employee table.

Creating Views Using JOIN Keyword 3-5

- The following figure displays the output.

	Title	FirstName	MiddleName	LastName	JobTitle
1	NULL	Ken	J	Sánchez	Chief Executive Officer
2	NULL	Terri	Lee	Duffy	Vice President of Engineering
3	NULL	Roberto	NULL	Tamburello	Engineering Manager
4	NULL	Rob	NULL	Walters	Senior Tool Designer
5	Ms.	Gail	A	Erickson	Design Engineer
6	Mr.	Jossef	H	Goldberg	Design Engineer
7	NULL	Dylan	A	Miller	Research and Development Manager
8	NULL	Diane	L	Margheim	Research and Development Engineer
9	NULL	Gigi	N	Matthew	Research and Development Engineer
10	NULL	Michael	NULL	Raheem	Research and Development Manager

- As shown in the figure, all the rows may not have values for the `Title` or `MiddleName` columns - some may have `NULL` in them.
- A person seeing this output may not be able to comprehend the meaning of the `NULL` values.

Creating Views Using JOIN Keyword 4-5

- The following code snippet replaces all the NULL values in the output with a null string using the COALESCE() function.

```
CREATE VIEW vwPersonDetails
AS
SELECT
COALESCE(p.Title, ' ') AS Title
,p.[FirstName]
,COALESCE(p.MiddleName, ' ') AS MiddleName
,p.[LastName]
,e.[JobTitle]
FROM [HumanResources].[Employee] e
INNER JOIN [Person].[Person] p
ON p.[BusinessEntityID] = e.[BusinessEntityID]
GO
```

Creating Views Using JOIN Keyword 5-5

- When this view is queried with a `SELECT` statement, the output will be as shown in the following figure:

	Title	FirstName	MiddleName	LastName	JobTitle
1		Ken	J	Sánchez	Chief Executive Officer
2		Terri	Lee	Duffy	Vice President of Engineering
3		Roberto		Tamburello	Engineering Manager
4		Rob		Walters	Senior Tool Designer
5	Ms.	Gail	A	Erickson	Design Engineer
6	Mr.	Jossef	H	Goldberg	Design Engineer
7		Dylan	A	Miller	Research and Development Manager
8		Diane	L	Margheim	Research and Development Engineer
9		Gigi	N	Matthew	Research and Development Engineer
10		Michael		Raheem	Research and Development Manager

Guidelines and Restrictions on Views 1-2

- Before creating a view, the following guidelines and restrictions should be considered:

  - A view can be created using the `CREATE VIEW` command.
  - View names must be unique and cannot be the same as the table names in the schema.
  - A view cannot be created on temporary tables.
  - A view cannot have a full-text index.
  - A view cannot contain the `DEFAULT` definition.
  - The `CREATE VIEW` statement can include the `ORDER BY` clause only if the `TOP` keyword is used.
  - Views cannot reference more than 1024 columns.
  - The `CREATE VIEW` statement cannot include the `INTO` keyword.
  - The `CREATE VIEW` statement cannot be combined with other Transact-SQL statements in a single batch.

Guidelines and Restrictions on Views 2-2

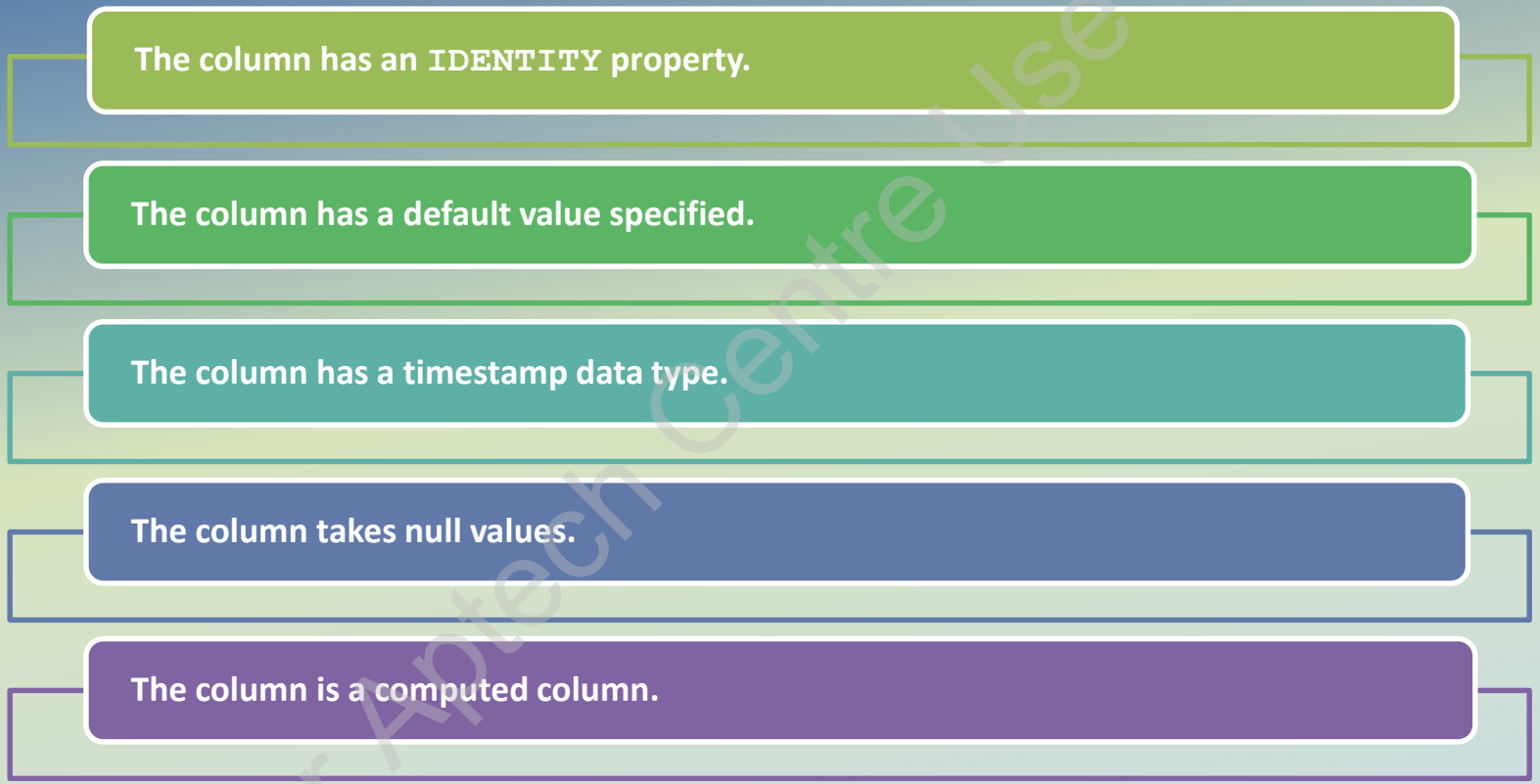
- Following code snippet reuses the code given earlier with an `ORDER BY` clause.

```
CREATE VIEW vwSortedPersonDetails
AS
SELECT TOP 10
COALESCE(p.Title, ' ') AS Title
,p.[FirstName]
,COALESCE(p.MiddleName, ' ') AS MiddleName
,p.[LastName]
,e.[JobTitle]
FROM [HumanResources].[Employee] e
INNER JOIN [Person].[Person] p
ON p.[BusinessEntityID] = e.[BusinessEntityID]
ORDER BY p.FirstName
GO
--Retrieve records from the view
SELECT * FROM vwSortedPersonDetails
```

- The `TOP` keyword displays the name of the first ten employees with their first names in ascending order.

INSERT with Views 1-5

- The `INSERT` statement is used to add a new row to a table or a view. The value of column is automatically provided if:



- While using the `INSERT` statement on a view, if any rules are violated, the record is not inserted.

INSERT with Views 2-5

- In the following example, when data is inserted through the view, the insertion does not take place as the view is created from two base tables.
- First, create a table **Employee_Personal_Details** as shown in the following code snippet:

```
CREATE TABLE Employee_Personal_Details
(
EmpID int NOT NULL,
FirstName varchar(30) NOT NULL,
LastName varchar(30) NOT NULL,
Address varchar(30)
)
```

- Then, create a table **Employee_Salary_Details** as shown in the following code snippet:

```
CREATE TABLE Employee_Salary_Details
(
EmpID
int NOT NULL,
Designation varchar(30),
Salary int NOT NULL
)
```

INSERT with Views 3-5

- Following code snippet creates a view **vwEmployee_Details** using columns from the **Employee_Personal_Details** and **Employee_Salary_Details** tables by joining the two tables on the **EmpID** column.

```
CREATE VIEW vwEmployee_Details
AS
SELECT e1.EmpID, FirstName, LastName, Designation, Salary
FROM Employee_Personal_Details e1
JOIN Employee_Salary_Details e2
ON e1.EmpID = e2.EmpID
```

- Following code snippet uses the INSERT statement to insert data through the view **vwEmployee_Details**.

```
INSERT INTO vwEmployee_Details VALUES (2,'Jack','Wilson','Software
Developer',16000)
```

- However, the data is not inserted as the view is created from two base tables.
- The following error message is displayed when the INSERT statement is executed.

```
'Msg 4405, Level 16, State 1, Line 1
View or function 'vEmployee_Details' is not updatable
because the modification affects multiple base tables.'
```

INSERT with Views 4-5

- Values can be inserted into user-defined data type columns by:
  - Specifying a value of the user-defined type.
  - Calling a user-defined function that returns a value of the user-defined type.
- The following rules and guidelines must be followed when using the `INSERT` statement:
  - The `INSERT` statement must specify values for all columns in a view in the underlying table that do not allow null values and have no `DEFAULT` definitions.
  - When there is a self-join with the same view or base table, the `INSERT` statement does not work.

INSERT with Views 5-5

- Following code snippet creates a view **vwEmpDetails** using **Employee_Personal_Details** table.

```
CREATE VIEW vwEmpDetails
AS
SELECT FirstName, Address
FROM Employee_Personal_Details
GO
```

- The **Employee_Personal_Details** table contains a column named **LastName** that does not allow null values to be inserted.
- Following code snippet attempts to insert values into the **vwEmpDetails** view.

```
INSERT INTO vwEmpDetails VALUES ('Jack','NYC')
```

- This insert is not allowed as the view does not contain the **LastName** column from the base table and that column does not allow null values.

UPDATE with Views 1-5

- The `UPDATE` statement can be used to change the data in a view.
- Updating a view also updates the underlying table.
- Following code snippet creates a table named **`Product_Details`**.

```
CREATE TABLE Product_Details
(
ProductID int,
ProductName varchar(30),
Rate money
)
```

- Assume some records are added in the table as shown in the following figure:

	ProductID	ProductName	Rate
1	5	DVD Writer	2250.00
2	4	DVD Writer	1250.00
3	6	DVD Writer	1250.00
4	2	External Hard Drive	4250.00
5	3	External Hard Drive	4250.00

UPDATE with Views 2-5

- Following code snippet creates a view based on the **Product_Details** table.

```
CREATE VIEW vwProduct_Details
AS
SELECT
ProductName, Rate FROM Product_Details
```

- Following code snippet updates the view to change all rates of DVD writers to 3000.

```
UPDATE vwProduct_Details
SET Rate=3000
WHERE ProductName='DVD Writer'
```

- The outcome of this code affects not only the view, **vwProduct_Details**, but also the underlying table from which the view was created.
- Following figure shows the updated table which was automatically updated because of the view.

	ProductID	ProductName	Rate
1	5	DVD Writer	3000.00
2	4	DVD Writer	3000.00
3	6	DVD Writer	3000.00
4	2	External Hard Drive	4250.00
5	3	External Hard Drive	4250.00

UPDATE with Views 3-5

- Large value data types include `varchar(max)`, `nvarchar(max)`, and `varbinary(max)`.
- To update data having large value data types, the `.WRITE` clause is used.
- The `.WRITE` clause specifies that a section of the value in a column is to be modified.
- The `.WRITE` clause cannot be used to update a `NULL` value in a column.
- Also, it cannot be used to set a column value to `NULL`.

Syntax:

```
column_name .WRITE (expression, @Offset, @Length)
```

where,

`column_name`: specifies the name of the large value data-type column.
`Expression`: specifies the value that is copied to the column.
`@Offset`: specifies the starting point in the value of the column at which the expression is written.
`@Length`: specifies the length of the section in the column.

- `@Offset` and `@Length` are specified in bytes for `varbinary` and `varchar` data types and in characters for the `nvarchar` data type.

UPDATE with Views 4-5

- Assume that the table **Product_Details** is modified to include a column **Description** having data type `nvarchar(max)`.
- The following code snippet creates a view based on this table, having the columns **ProductName**, **Description**, and **Rate**.

```
CREATE VIEW vwProduct_Details
AS
SELECT
ProductName,
Description,
Rate FROM Product_Details
```

- Following code snippet uses the `UPDATE` statement on the view **vwProduct_Details**.
- The `.WRITE` clause is used to change the value of Internal in the **Description** column to External.

```
UPDATE vwProduct_Details
SET Description .WRITE(N'Ex',0,2)
WHERE ProductName='Portable Hard Drive'
```

- All the rows in the view that had 'Portable Hard Drive' as product name will be updated with `External` instead of Internal in the **Description** column.

UPDATE with Views 5-5

- Following figure shows a sample output of the view after the updation.

	ProductName	Description	Rate
1	Hard Disk Drive	Internal 120 GB	3570.00
2	Portable Hard Drive	External Drive 500 GB	5580.00
3	Portable Hard Drive	External Drive 500 GB	5580.00
4	Hard Disk Drive	Internal 120 GB	3570.00
5	Portable Hard Drive	External Drive 500 GB	5580.00

- The following rules and guidelines must be followed when using the `UPDATE` statement:

  - The value of a column with an `IDENTITY` property cannot be updated.
  - Records cannot be updated if the base table contains a `TIMESTAMP` column.
  - While updating a row, if a constraint or rule is violated, the statement is terminated, an error is returned, and no records are updated.
  - When there is a self-join with the same view or base table, the `UPDATE` statement does not work.

DELETE with Views 1-2

- Rows can be deleted from the view using the `DELETE` statement.
- When rows are deleted from a view, corresponding rows are deleted from the base table.
- For example, consider a view **vwCustDetails** that lists the account information of different customers.
- When a customer closes the account, the details of this customer need to be deleted.
- The syntax used to delete data from a view is as follows:

Syntax:

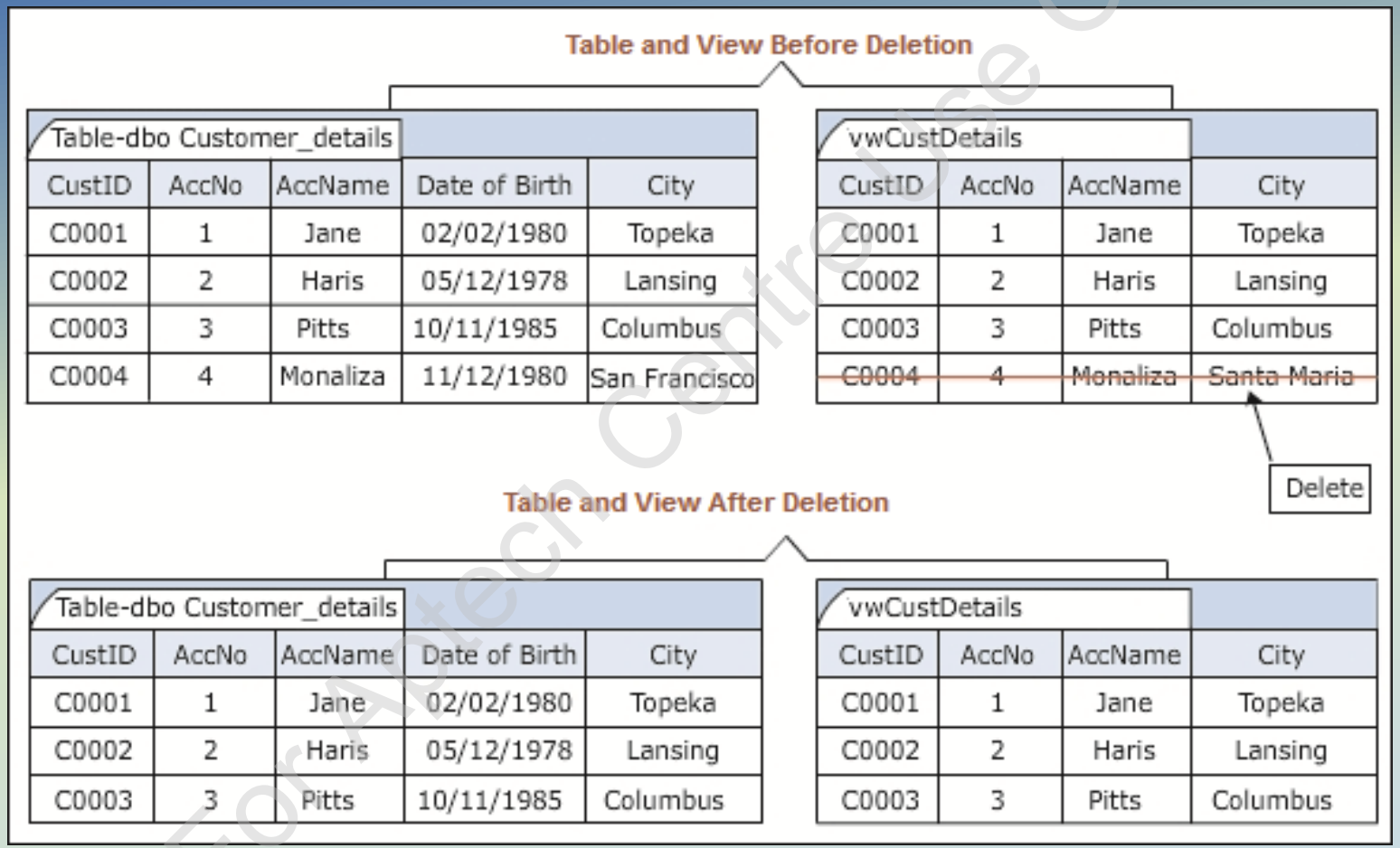
```
DELETE FROM <view_name>
WHERE <search_condition>
```

- Assume that a table named **Customer_Details** and a view **vwCustDetails** based on the table are created.
- Following code snippet is used to delete the record from the view **vwCustDetails** that has **CustID** C0004.

```
DELETE FROM vwCustDetails WHERE CustID='C0004'
```

DELETE with Views 2-2

- Following figure depicts the logic of deleting from views.



Altering Views 1-2

Besides modifying the data within a view, users can also modify the definition of a view.

The `ALTER VIEW` statement modifies an existing view without having to reorganize its permissions and other properties.

Views are often altered when a user requests for additional information or makes changes in the underlying table definition.

Altering Views 2-2

- The syntax used to alter a view is as follows:

Syntax:

```
ALTER VIEW <view_name>
AS <select_statement>
```

- Following code snippet alters the view, **vwProductInfo** to include the **ReOrderPoint** column.

```
ALTER VIEW vwProductInfo AS
SELECT ProductID, ProductNumber, Name, SafetyStockLevel, ReOrderPoint
FROM Production.Product;
GO
```

Dropping Views

- A view can be removed from the database using the `DROP VIEW` statement.
- When a view is dropped, the data in the base tables remains unaffected.
- The definition of the view and other information associated with the view is deleted from the system catalog.
- All permissions for the view are also deleted.
- The syntax used to drop a view is as follows:

Syntax:

```
DROP VIEW <view_name>
```

- Following code snippet deletes the view, **vwProductInfo**.

```
DROP VIEW vwProductInfo
```

Definition of a View

- The definition of a view helps to understand how its data is derived from the source tables.
- The `sp_helptext` stored procedure displays view related information when the name of the view is given as its parameter.
- The syntax used to view the definition information of a view is as follows:

Syntax:

```
sp_helptext <view_name>
```

- Following code snippet displays information about the view, **`vwProductPrice`**.

```
EXEC sp_helptext vwProductPrice
```

- The execution of the code will display the definition about the view as shown in the following figure.

	Text
1	CREATE VIEW vwProductPrice AS
2	SELECT ProductID, ProductNumber, Name, SafetySto...
3	FROM Production.Product;

Creating a View Using Built-in Functions

- When functions are used, the derived column must include the column name in the `CREATE VIEW` statement.
- Consider the view that was created in the following code snippet to make use of the `AVG()` function.

```
CREATE VIEW vwProduct_Details
AS
SELECT
ProductName,
AVG(Rate) AS AverageRate
FROM Product_Details
GROUP BY ProductName
```

- Here, the `AVG()` function calculates the average rate of similar products by using a `GROUP BY` clause.
- Following figure shows the result when the view is queried.

	ProductName	AverageRate
1	Hard Disk Drive	3570.00
2	Portable Hard Drive	5580.00

CHECK OPTION 1-2

The CHECK OPTION is used to enforce domain integrity; it checks the definition of the view to see that the WHERE conditions in the SELECT statement is not violated.

- The syntax used to create a view using the CHECK OPTION is as follows:

Syntax:

```
CREATE VIEW <view_name>
AS select_statement [ WITH CHECK OPTION ]
```

where,

WITH CHECK OPTION: specifies that the modified data in the view continues to satisfy the view definition.

CHECK OPTION 2-2

- Following code snippet re-creates the view **vwProductInfo** having SafetyStockLevel less than or equal to 1000:

```
CREATE VIEW vwProductInfo AS
SELECT ProductID, ProductNumber,Name,SafetyStockLevel,
ReOrderPoint
FROM Production.Product
WHERE SafetyStockLevel <=1000
WITH CHECK OPTION;
GO
```

- In the following code snippet, the UPDATE statement is used to modify the view **vwProductInfo** by changing the value of the SafetyStockLevel column for the product having id 321 to 2500.

```
UPDATE vwProductInfo SET SafetyStockLevel= 2500
WHERE ProductID=321
```

- The UPDATE statement fails to execute as it violates the view definition, which specifies that SafetyStockLevel must be less than or equal to 1000.
- Thus, no rows are affected in the view **vwProductInfo**.

SCHEMABINDING Option 1-2

A view can be bound to the schema of the base table using the SCHEMABINDING option.

The view definition must be first modified or deleted to remove dependencies on the table that is to be modified.

- The syntax used to create a view with the SCHEMABINDING option is as follows:

Syntax:

```
CREATE VIEW <view_name> WITH SCHEMABINDING
AS <select_statement>
```

where,

view_name: specifies the name of the view.

SCHEMABINDING Option 2-2

`WITH SCHEMABINDING`: specifies that the view must be bound to a schema.
`select_statement`: specifies the `SELECT` statement that defines the view.

- Following code snippet creates a view **`vwNewProductInfo`** with `SCHEMABINDING` option to bind the view to the `Production` schema, which is the schema of the table `Product`.

```
CREATE VIEW vwNewProductInfo
WITH SCHEMABINDING AS
SELECT ProductID, ProductNumber, Name, SafetyStockLevel
FROM Production.Product;
GO
```

Using sp_refreshview 1-3

- The `sp_refreshview` stored procedure updates the metadata for the view.
- If the `sp_refreshview` procedure is not executed, the metadata of the view is not updated to reflect the changes in the base tables.
- This results in the generation of unexpected results when the view is queried.
- The `sp_refreshview` stored procedure returns code value zero if the execution is successful or returns a non-zero number in case the execution has failed.
- The syntax used to run the `sp_refreshview` stored procedure is as follows:

Syntax:

```
sp_refreshview '<view_name>'
```

- Following code snippet creates a table **Customers** with the **CustID**, **CustName**, and **Address** columns.

```
CREATE TABLE Customers
(
CustID int,
CustName varchar(50),
Address varchar(60)
)
```

Using sp_refreshview 2-3

- Following code snippet creates a view **vwCustomers** based on the table **Customers**.

```
CREATE VIEW vwCustomers
AS
SELECT * FROM Customers
```

- Following code snippet executes the SELECT query on the view.

```
SELECT * FROM vwCustomers
```

- The output will show three columns, **CustID**, **CustName**, and **Address**.
- Following code snippet uses the ALTER TABLE statement to add a column Age to the table **Customers**.

```
ALTER TABLE Customers ADD Age int
```

- Following code snippet executes the SELECT query on the view.

```
SELECT * FROM vwCustomers
```

Using sp_refreshview 3-3

- The updated column **Age** is not seen in the view.
- To resolve this, the `sp_refreshview` stored procedure must be executed on the view **vwCustomers** as shown in the following code snippet:

```
EXEC sp_refreshview 'vwCustomers'
```

- When a `SELECT` query is run again on the view, the column **Age** is seen in the output.
- This is because the `sp_refreshview` procedure refreshes the metadata for the view **vwCustomers**.
- Consider the schema-bound view that is dependent on the **Production.Product** table.
- Following code snippet tries to modify the data type of **ProductID** column in the **Production.Product** table from `int` to `varchar(7)`.

```
ALTER TABLE Production.Product ALTER COLUMN ProductID varchar(7)
```

- An error message is displayed as the table is schema-bound to the **vwNewProductInfo** view and hence, cannot be altered such that it violates the view definition of the view.

Stored Procedures

A stored procedure is a group of Transact-SQL statements that act as a single block of code that performs a specific task.

A stored procedure may also be a reference to a .NET Framework Common Language Runtime (CLR) method.

Stored procedures can accept values in the form of input parameters and return output values as defined by the output parameters.

- The advantages of using stored procedures are as follows:
  - **Improved Security**
  - **Precompiled Execution**
  - **Reduced Client/Server Traffic**
  - **Reuse of Code**

Types of Stored Procedures 1-3

- SQL Server supports the following types of stored procedures:

User-Defined Stored Procedures

- User-defined stored procedures are also known as custom stored procedures.
- These procedures are used for reusing Transact-SQL statements for performing repetitive tasks.
- There are two types of user-defined stored procedures, the Transact-SQL stored procedures and the Common Language Runtime (CLR) stored procedures.
- Transact-SQL stored procedures consist of Transact-SQL statements whereas the CLR stored procedures are based on the .NET framework CLR methods.
- Both the stored procedures can take and return user-defined parameters.

Types of Stored Procedures 2-3

Extended Stored Procedures

- Extended stored procedures help SQL Server in interacting with the operating system.
- Extended stored procedures are not resident objects of SQL Server.
- They are procedures that are implemented as dynamic-link libraries (DLL) executed outside the SQL Server environment.
- The application interacting with SQL Server calls the DLL at run-time. The DLL is dynamically loaded and run by SQL Server.
- SQL Server allots space to run the extended stored procedures.
- Extended stored procedures use the '`xp`' prefix.
- Tasks that are complicated or cannot be executed using Transact-SQL statements are performed using extended stored procedures.

Types of Stored Procedures 3-3

System Stored Procedures

- System stored procedures are commonly used for interacting with system tables and performing administrative tasks such as updating system tables.
- The system stored procedures are prefixed with '`sp_`'. These procedures are located in the Resource database.
- These procedures can be seen in the sys schema of every system and user-defined database.
- System stored procedures allow `GRANT`, `DENY`, and `REVOKE` permissions.
- A system stored procedure is a set of pre-compiled Transact-SQL statements executed as a single unit.
- System procedures are used in database administrative and informational activities.
- When referencing a system stored procedure, the sys schema identifier is used. System stored procedures are owned by the database administrator.

Classification of System Stored Procedures 1-2

Catalog Stored Procedures

- All information about tables in the user database is stored in a set of tables called the system catalog.
- Information from the system catalog can be accessed using catalog procedures.
- For example, the `sp_tables` catalog stored procedure displays the list of all the tables in the current database.

Security Stored Procedures

- Security stored procedures are used to manage the security of the database.
- For example, the `sp_changedbowner` security stored procedure is used to change the owner of the current database.

Cursor Stored Procedures

- Cursor procedures are used to implement the functionality of a cursor.
- For example, the `sp_cursor_list` cursor stored procedure lists all the cursors opened by the connection and describes their attributes.

Classification of System Stored Procedures 2-2

Distributed Query Stored Procedures

- Distributed stored procedures are used in the management of distributed queries.
- For example, the `sp_indexes` distributed query stored procedure returns index information for the specified remote table.

Database Mail and SQL Mail Stored Procedures

- Database Mail and SQL Mail stored procedures are used to perform e-mail operations from within the SQL Server.
- For example, the `sp_send_dbmail` database mail stored procedure sends e-mail messages to specified recipients.
- The message may include a query resultset or file attachments or both.

Temporary Stored Procedures

- Stored procedures created for temporary use within a session are called temporary stored procedures.
- These procedures are stored in the `tempdb` database.
- The `tempdb` system database is a global resource available to all users connected to an instance of SQL Server.
- SQL Server supports two types of temporary stored procedures namely, local and global.
- The table lists the differences between the two types of stored procedures.

Local Temporary Procedure	Global Temporary Procedure
Visible only to the user that created it	Visible to all
Dropped at the end of the current session	Dropped at the end of the last session
Local Temporary Procedure	Global Temporary Procedure
Can only be used by its owner	Can be used by any user
Uses the # prefix before the procedure name	Uses the ## prefix before the procedure name

Remote Stored Procedures

Stored procedures that run on remote SQL Servers are known as remote stored procedures.

When a remote stored procedure is executed from a local instance of SQL Server to a client computer, a statement abort error might be encountered.

Extended Stored Procedures

Extended stored procedures are used to perform tasks that are unable to be performed using standard Transact-SQL statements.

- The syntax used to execute an extended stored procedure is as follows:

Syntax:

```
EXECUTE <procedure_name>
```

- Following code snippet executes the extended stored procedure `xp_fileexist` to check whether the `MyTest.txt` file exists or not.

```
EXECUTE xp_fileexist 'c:\MyTest.txt'
```

Custom or User-defined Stored Procedures 1-3

In SQL Server, users are allowed to create customized or user-defined stored procedures for performance of various tasks.

- The syntax used to create a custom stored procedure is as follows:

Syntax:

```
CREATE { PROC | PROCEDURE } procedure_name
[ { @parameter data_type } ]
AS <sql_statement>
```

where,

`procedure_name`: specifies the name of the procedure.

`@parameter`: specifies the input/output parameters in the procedure.

`data_type`: specifies the data types of the parameters.

`sql_statement`: specifies one or more Transact-SQL statements to be included in the procedure.

Custom or User-defined Stored Procedures 2-3

- Following code snippet creates and then executes a custom stored procedure, **`uspGetCustTerritory`**, which will display the details of customers such as customer id, territory id, and territory name.

```
CREATE PROCEDURE uspGetCustTerritory
AS
SELECT TOP 10 CustomerID, Customer.TerritoryID,
Sales.SalesTerritory.Name
FROM Sales.Customer JOIN Sales.SalesTerritory ON
Sales.Customer.TerritoryID = Sales.SalesTerritory.TerritoryID
```

- The following code snippet executes the stored procedure using the `EXEC` command.

```
EXEC uspGetCustTerritory
```

Custom or User-defined Stored Procedures 3-3

- The output is shown in the following figure:

	CustomerID	TerritoryID	Name
1	15	9	Australia
2	33	9	Australia
3	51	9	Australia
4	69	9	Australia
5	87	9	Australia
6	105	9	Australia
7	123	9	Australia
8	141	9	Australia
9	159	9	Australia
10	177	9	Australia

Using Parameters

The real advantage of a stored procedure comes into picture only when one or more parameters are used with it.

- This data transfer is done using parameters. Parameters are of two types that are as follows:

Input Parameters	• Input parameters allow the calling program to pass values to a stored procedure. • These values are accepted into variables defined in the stored procedure.
Output Parameters	• Output parameters allow a stored procedure to pass values back to the calling program. • These values are accepted into variables by the calling program.

Input Parameters 1-2

- Values are passed from the calling program to the stored procedure and these values are accepted into the input parameters of the stored procedure.
- The input parameters are defined at the time of creation of the stored procedure.
- The values passed to input parameters could be either constants or variables.
- These values are passed to the procedure at the time of calling the procedure.
- The stored procedure performs the specified tasks using these values.
- The syntax used to create a stored procedure is as follows:

Syntax:

```
CREATE PROCEDURE <procedure_name>
@parameter <data_type>
AS <sql_statement>
```

where,

`data_type`: specifies the system defined data type.

- The syntax used to execute a stored procedure and pass values as input parameters is as follows:

Syntax:

```
EXECUTE <procedure_name> <parameters>
```

Input Parameters 2-2

- Following code snippet creates a stored procedure, **uspGetSales** with a parameter territory to accept the name of a territory and display the sales details and salesperson id for that territory.
- Then, the code executes the stored procedure with Northwest being passed as the input parameter.

```
CREATE PROCEDURE uspGetSales
@territory varchar(40)
AS
SELECT BusinessEntityID, B.SalesYTD, B.SalesLastYear
FROM Sales.SalesPerson A
JOIN Sales.SalesTerritory B
ON A.TerritoryID = B.TerritoryID
WHERE B.Name = @territory;
--Execute the stored procedure
EXEC uspGetSales 'Northwest'
```

- The output is shown in the following figure:

	BusinessEntityID	SalesYTD	SalesLastYear
1	280	7887186.7882	3298694.4938
2	283	7887186.7882	3298694.4938
3	284	7887186.7882	3298694.4938

Output Parameters 1-3

Output parameters are defined at the time of creation of the procedure.

Also, the calling statement has to have a variable specified with the `OUTPUT` keyword to accept the output from the called procedure.

- The following syntax is used to pass output parameters in a stored procedure and then, execute the stored procedure with the `OUTPUT` parameter.

Syntax:

```
EXECUTE <procedure_name> <parameters>
```

Output Parameters 2-3

- Following code snippet creates a stored procedure, `uspGetTotalSales` with input parameter `@territory` to accept the name of a territory and output parameter `@sum` to display the sum of sales year to date in that territory.

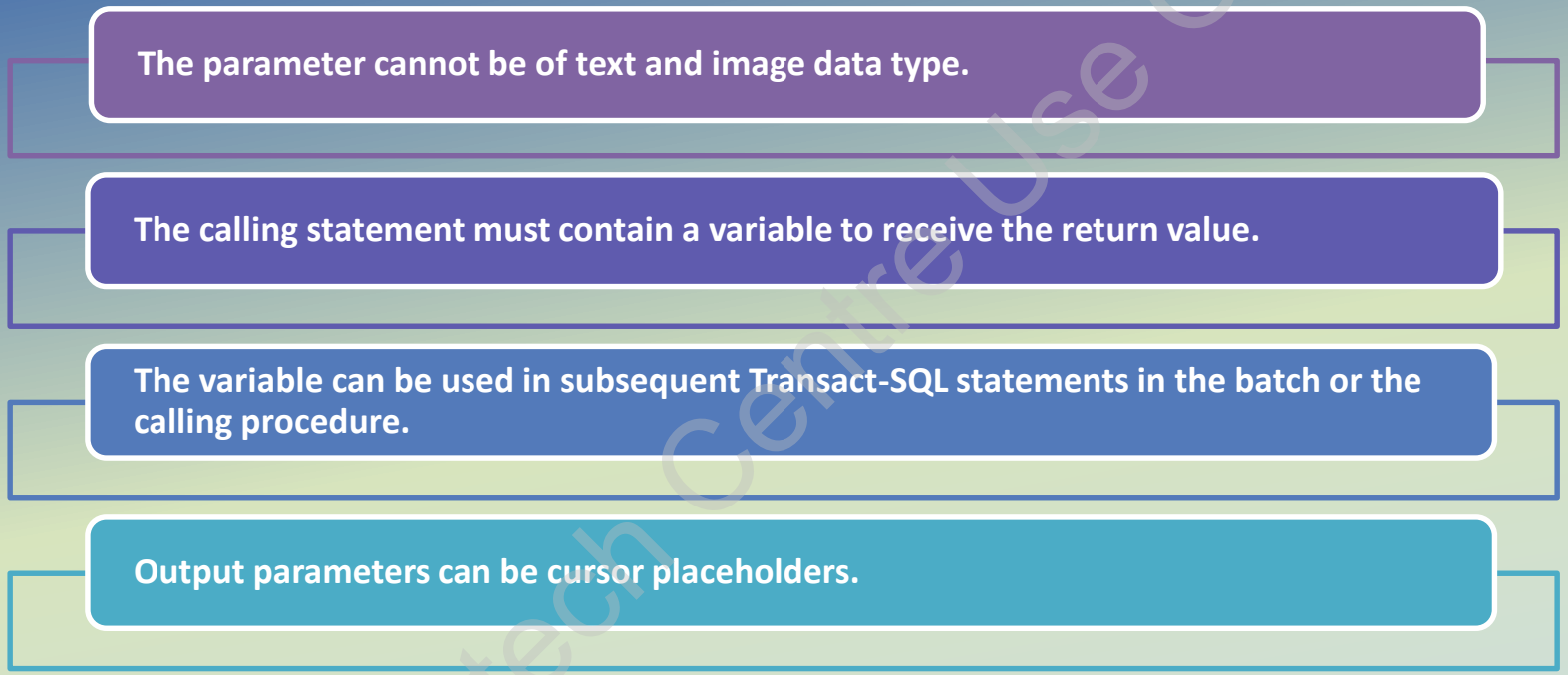
```
CREATE PROCEDURE uspGetTotalSales
@territory varchar(40), @sum int OUTPUT
AS
SELECT @sum= SUM(B.SalesYTD)
FROM Sales.SalesPerson A
JOIN Sales.SalesTerritory B
ON A.TerritoryID = B.TerritoryID
WHERE B.Name = @territory
```

- Following code snippet declares a variable `sumsales` to accept the output of the procedure `uspGetTotalSales`.

```
DECLARE @sumsales money;
EXEC uspGetTotalSales 'Northwest', @sum = @sum OUTPUT;
PRINT 'The year-to-date sales figure for this territory is ' +
convert(varchar(100),@sumsales);
GO
```

Output Parameters 3-3

- OUTPUT parameters have the following characteristics:



- The OUTPUT clause returns information from each row on which the INSERT, UPDATE, and DELETE statements have been executed.
- This clause is useful to retrieve the value of an identity or computed column after an INSERT or UPDATE operation.

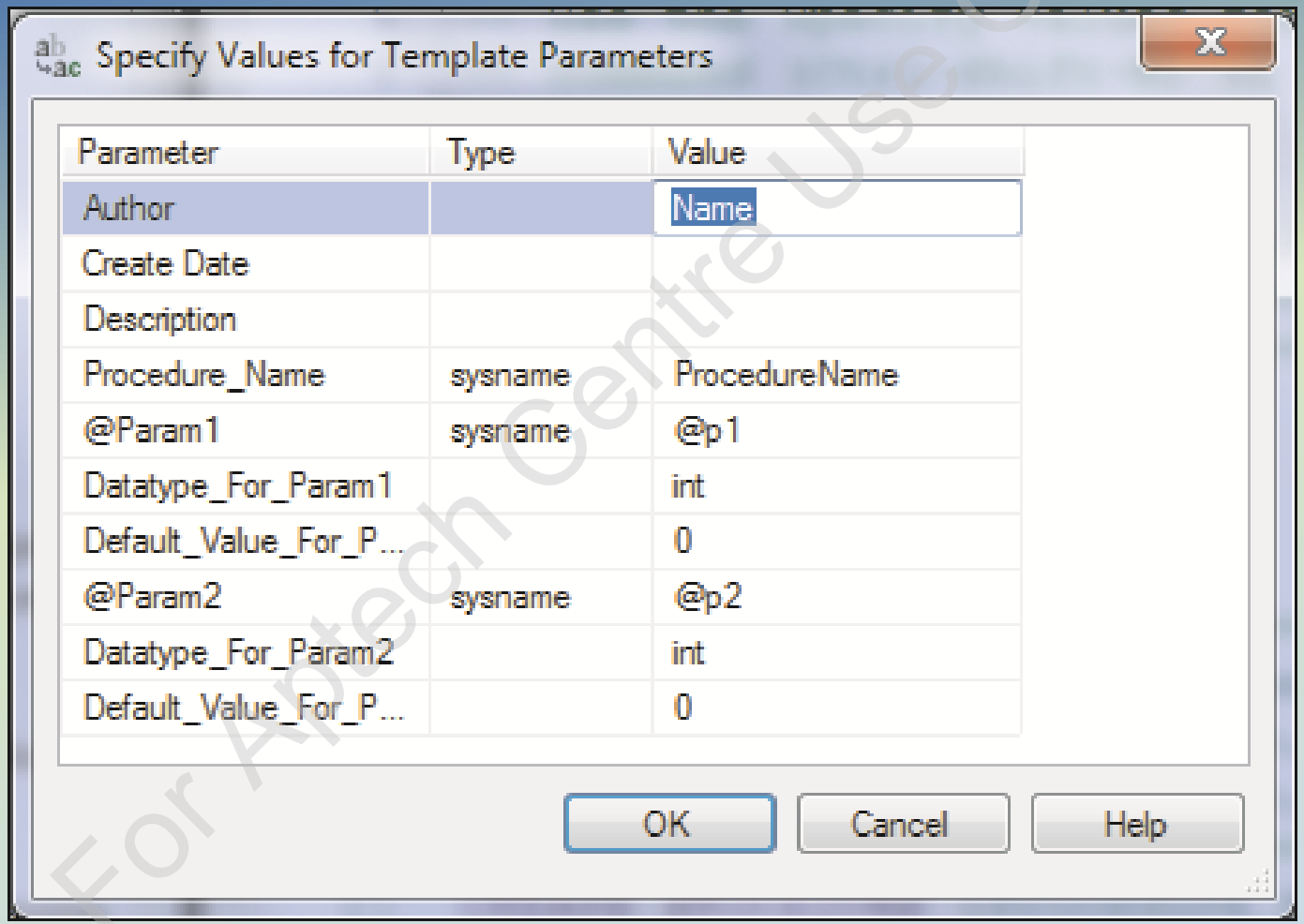
Using SSMS to Create Stored Procedures 1-5

- You can also create a user-defined stored procedure using SSMS by performing the following steps:

1. Launch **Object Explorer**.
2. In **Object Explorer**, connect to an instance of Database Engine.
3. After successfully connecting to the instance, expand that instance.
4. Expand **Databases** and then, expand the **AdventureWorks2012** database.
5. Expand **Programmability**, right-click **Stored Procedures**, and then, click **New Stored Procedure**.
6. On the **Query** menu, click **Specify Values for Template Parameters**. The **Specify Values for Template Parameters** dialog box is displayed.

Using SSMS to Create Stored Procedures 2-5

- This is shown in the following figure:



Using SSMS to Create Stored Procedures 3-5

7

- In the **Specify Values for Template Parameters** dialog box, enter the values for the parameters as shown in the following table:

Parameter	Value
Author	Your name
Create Date	Today's date
Description	Returns year to sales data for a territory
Procedure_Name	uspGetTotals
@Param1	@territory
@Datatype_For_Param1	varchar(50)
Default_Value_For_Param1	NULL
@Param2	
@Datatype_For_Param2	
Default_Value_For_Param2	

8

- After entering these details, click **OK**.

Using SSMS to Create Stored Procedures 4-5

9

- In the Query Editor, replace the `SELECT` statement with the following statement:

```
SELECT BusinessEntityID, B.SalesYTD, B.SalesLastYear
FROM Sales.SalesPerson A
JOIN Sales.SalesTerritory B
ON A.TerritoryID = B.TerritoryID
WHERE B.Name = @territory;
```

10

- To test the syntax, on the **Query** menu, click **Parse**. If an error message is returned, compare the statements with the information and correct as needed.

11

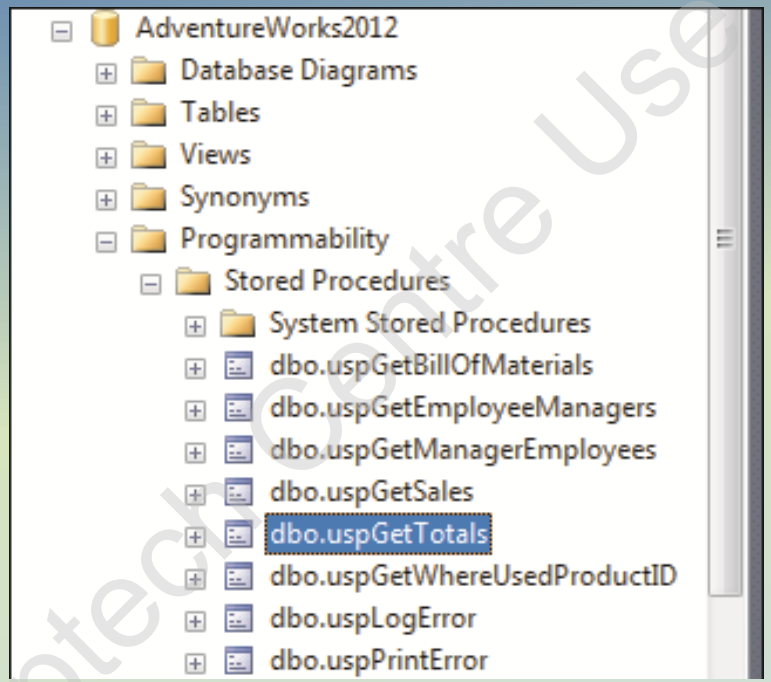
- To create the procedure, from the **Query** menu, click **Execute**. The procedure is created as an object in the database.

12

- To see the procedure listed in **Object Explorer**, right-click **Stored Procedures** and select **Refresh**.

Using SSMS to Create Stored Procedures 5-5

- The procedure name will be displayed in the **Object Explorer** tree as shown in the following figure:



10. To run the procedure, in **Object Explorer**, right-click the stored procedure name `uspGetTotals` and select **Execute Stored Procedure**.
11. In the **Execute Procedure** window, enter **Northwest** as the value for the parameter `@territory`.

Viewing Stored Procedure Definitions

- The definition of a stored procedure can be viewed using the sp_helptext system stored procedure.
- To view the definition, you must specify the name of the stored procedure as the parameter when executing sp_helptext.
- This definition is in the form of Transact-SQL statements.
- The Transact-SQL statements of the procedure definition include the `CREATE PROCEDURE` statement as well as the SQL statements that define the body of the procedure.
- The syntax used to view the definition of a stored procedure is as follows:

Syntax:

```
sp_helptext '<procedure_name>'
```

- Following code snippet displays the definition of the stored procedure named **`uspGetTotals`**.

```
EXEC sp_helptext uspGetTotals
```

Modifying and Dropping Stored Procedures 1-2

The permissions associated with the stored procedure are lost when a stored procedure is re-created.

A procedure can be altered using the `ALTER PROCEDURE` statement.

- The syntax used to modify a stored procedure is as follows:

Syntax:

```
ALTER PROCEDURE <procedure_name>
@parameter <data_type> [ OUTPUT ]
[ WITH { ENCRYPTION | RECOMPILE } ]
AS <sql_statement>
```

where,

`ENCRYPTION`: encrypts the stored procedure definition.

`RECOMPILE`: indicates that the procedure is compiled at run-time.

`sql_statement`: specifies the Transact-SQL statements to be included in the body of the procedure.

Modifying and Dropping Stored Procedures 2-2

- Following code snippet modifies the definition of the stored procedure named **uspGetTotals** to add a new column **CostYTD** to be retrieved from **Sales.SalesTerritory**.

```
ALTER PROCEDURE [dbo].[uspGetTotals]
@territory varchar = 40
AS
SELECT BusinessEntityID, B.SalesYTD, B.CostYTD, B.SalesLastYear
FROM Sales.SalesPerson A
JOIN Sales.SalesTerritory B
ON A.TerritoryID = B.TerritoryID
WHERE B.Name = @territory;
GO
```

Guidelines for Using ALTER PROCEDURE Statement

- When a stored procedure is created using options such as the `WITH ENCRYPTION` option, these options should also be included in the `ALTER PROCEDURE` statement.
- The `ALTER PROCEDURE` statement alters a single procedure.
  When a stored procedure calls other stored procedures, the nested stored procedures are not affected by altering the calling procedure.
- The creators of the stored procedure, members of the `sysadmin` server role and members of the `db_owner` and `db_ddladmin` fixed database roles have the permission to execute the `ALTER PROCEDURE` statement.
- It is recommended that you do not modify system stored procedures. If you need to change the functionality of a system stored procedure, then create a user-defined system stored procedure by copying the statements from an existing stored procedure and modify this user-defined procedure.

Dropping Stored Procedures

- Before dropping a stored procedure, execute the `sp_depends` system stored procedure to determine which objects depend on the procedure.
- A procedure is dropped using the `DROP PROCEDURE` statement.
- The syntax used to drop a stored procedure is as follows:

Syntax:

```
DROP PROCEDURE <procedure_name>
```

- Following code snippet drops the stored procedure, **`uspGetTotals`**.

```
DROP PROCEDURE uspGetTotals
```

Nested Stored Procedures 1-2

SQL Server 2012 enables stored procedures to be called inside other stored procedures.

This architecture of calling one procedure from another procedure is referred to as nested stored procedure architecture.

If a stored procedure attempts to access more than 64 databases, or more than two databases in the nesting architecture, there will be an error.

Nested Stored Procedures 2-2

- Following code snippet is used to create a stored procedure **NestedProcedure** that calls two other stored procedures that were created earlier.

```
CREATE PROCEDURE NestedProcedure
AS
BEGIN
EXEC uspGetCustTerritory
EXEC uspGetSales 'France'
END
```

- When the procedure **NestedProcedure** is executed, this procedure in turn invokes the **uspGetCustTerritory** and **uspGetSales** stored procedures and passes the value France as the input parameter to the **uspGetSales** stored procedure.

@@NESTLEVEL Function 1-2

- The level of nesting of the current procedure can be determined using the `@@NESTLEVEL` function.
- When the `@@NESTLEVEL` function is executed within a Transact-SQL string, the value returned is the current nesting level + 1.
- If you use `sp_executesql` to execute the `@@NESTLEVEL` function, the value returned is the current nesting level + 2 (as another stored procedure, namely `sp_executesql`, gets added to the nesting chain).

Syntax:

```
@@NESTLEVEL
```

where,

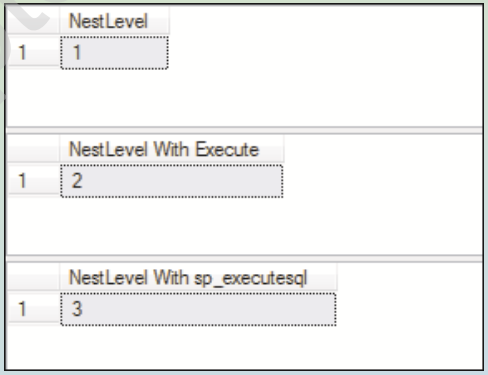
`@@NESTLEVEL`: Is a function that returns an integer value specifying the level of nesting.

@@NESTLEVEL Function 2-2

- Following code snippet creates and executes a procedure **Nest_Procedure** that executes the @@NESTLEVEL function to determine the level of nesting in three different scenarios.

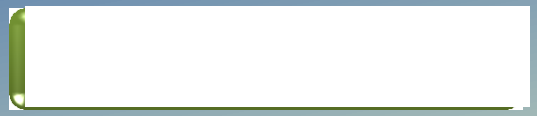
```
CREATE PROCEDURE Nest_Procedure
AS
SELECT @@NESTLEVEL AS NestLevel;
EXECUTE ('SELECT @@NESTLEVEL AS [NestLevel With Execute]');
EXECUTE sp_executesql N'SELECT @@NESTLEVEL AS [NestLevel With
sp_executesql]';
Code Snippet 44 executes the Nest_Procedure stored procedure.
Code Snippet 44:
EXECUTE Nest_Procedure
```

- Three outputs are displayed in the following figure for the three different methods used to call the @@NESTLEVEL function.



Querying System MetaData 1-4

- This metadata can be viewed using system views, which are predefined views of SQL Server.
- These views are grouped into several different schemas as follows:



- These contain information about the catalog in a SQL Server system.
- A catalog is similar to an inventory of objects.
- These views contain a wide range of metadata.
- Following code snippet retrieves a list of user tables and attributes from the system catalog view `sys.tables`.

```
SELECT name, object_id, type, type_desc
FROM sys.tables;
```

Querying System MetaData 2-4

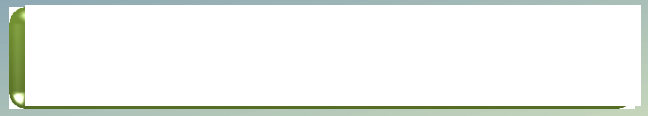
- These views are useful to third-party tools that may not be specific for SQL Server.
- Information schema views provide an internal, system table-independent view of the SQL Server metadata.
- Information schema views enable applications to work correctly although significant changes have been made to the underlying system tables.
- The points given in the following table will help to decide whether one should query SQL Server-specific system views or information schema views.

Information Schema Views	SQL Server System Views
They are stored in their own schema, INFORMATION_SCHEMA.	They appear in the sys schema.
They use standard terminology instead of SQL Server terms. For example, they use catalog instead of database and domain instead of user-defined data type.	They adhere to SQL Server terminology.
They may not expose all the metadata available to SQL Server's own catalog views. For example, sys.columns includes attributes for the identity property and computed column property, while INFORMATION_SCHEMA.columns does not.	They can expose all the metadata available to SQL Server's catalog views.

Querying System MetaData 3-4

- Following code snippet retrieves data from the `INFORMATION_SCHEMA.TABLES` view in the `AdventureWorks2012` database.

```
SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME, TABLE_TYPE
FROM INFORMATION_SCHEMA.TABLES;
```



- In addition to views, SQL Server provides a number of built-in functions that return metadata to a query.
- These include scalar functions and table-valued functions, which can return information about system settings, session options, and a wide range of objects.
- SQL Server metadata functions come in a variety of formats. Some appear similar to standard scalar functions, such as `ERROR_NUMBER()`.
- Others use special prefixes, such as `@@VERSION` or `$PARTITION`.

Querying System MetaData 4-4

- Following table shows some common system metadata functions.

Function Name	Description	Example
OBJECT_ID(<object_name>)	Returns the object ID of a database object.	OBJECT_ID('Sales.Customer')
OBJECT_NAME(<object_id>)	Returns the name corresponding to an object ID.	OBJECT_NAME(197575742)
@@ERROR	Returns 0 if the last statement succeeded; otherwise returns the error number.	@@ERROR
SERVERPROPERTY(<property >)	Returns the value of the specified server property.	SERVERPROPERTY('Collation')

- Following code snippet uses a `SELECT` statement to query a system metadata function.

```
SELECT SERVERPROPERTY('EDITION') AS EditionName;
```

Querying Dynamic Management Objects

Dynamic Management Views (DMVs) and Dynamic Management Functions (DMFs) are dynamic management objects that return server and database state information.

They provide useful insight into the working of software and can be used for examining the state of SQL Server instance, troubleshooting, and performance tuning.

SQL Server 2012 provides nearly 200 dynamic management objects.

Categorizing and Querying DMVs 1-3

- Following table lists the naming convention that helps organize the DMVs by function.

Naming Pattern	Description
db	database related
io	I/O statistics
Os	SQL Server Operating System Information
"tran"	transaction-related
"exec"	query execution-related metadata

- To query a dynamic management object, you use a `SELECT` statement as you would with any user-defined view or table-valued function.

Categorizing and Querying DMVs 2-3

- Following code snippet returns a list of current user connections from the `sys.dm_exec_sessions` view.

```
SELECT session_id, login_time, program_name
FROM sys.dm_exec_sessions
WHERE login_name ='sa' and is_user_process =1;
```

- `sys.dm_exec_sessions` is a server-scoped DMV that displays information about all active user connections and internal tasks.
- This information includes login user, current session setting, client version, client program name, client login time, and more.
- The `sys.dm_exec_sessions` can be used to identify a specific session and find information about it.

Categorizing and Querying DMVs 3-3

- Here, `is_user_process` is a column in the view that determines if the session is a system session or not.
- A value of 1 indicates that it is not a system session but rather a user session.
- The `program_name` column determines the name of client program that initiated the session.
- The `login_time` column establishes the time when the session began.
- The output of the code is shown in the following figure:

	session_id	login_time	program_name
1	51	2013-01-29 12:26:08.443	Microsoft SQL Server Management Studio
2	53	2013-01-29 12:26:20.247	Microsoft SQL Server Management Studio - Query

Summary 1-2

- A view is a virtual table that is made up of selected columns from one or more tables and is created using the CREATE VIEW command in SQL Server.
- Users can manipulate the data in views, such as inserting into views, modifying the data in views, and deleting from views.
- A stored procedure is a group of Transact-SQL statements that act as a single block of code that performs a specific task.
- SQL Server supports various types of stored procedures, such as User-Defined Stored Procedures, Extended Stored Procedures, and System Stored Procedures.
- System stored procedures can be classified into different categories such as Catalog Stored Procedures, Security Stored Procedures, and Cursor Stored Procedures.
- Input and output parameters can be used with stored procedures to pass and receive data from stored procedures.

Summary 2-2

- The properties of an object such as a table or a view are stored in special system tables and are referred to as metadata.
- DMVs and DMFs are dynamic management objects that return server and database state information.
- DMVs and DMFs are collectively referred to as dynamic management objects.