



# Quản Trị Dữ Liệu Với Microsoft SQL Server

## Chương: 10

Sử dụng Views, Stored Procedures,  
và Truy vấn Metadata



# Mục tiêu

- Định nghĩa view
- Mô tả kỹ thuật tạo, sửa đổi( alter), và xóa view
- Định nghĩa thủ tục lưu (stored procedures)
- Giải thích về các loại thủ tục
- Mô tả các bước thủ tục để tạo, sửa đổi, và thực thi thủ tục lưu
- Mô tả thủ tục lồng nhau
- Mô tả việc truy vấn thông tin metadata của SQL Server metadata
  - Các hàm hệ thống và view danh mục (Catalog views) hệ thống
  - Truy vấn Dynamic Management Objects



# Giới thiệu

- Một csdl SQL Server có hai danh mục đối tượng chính:
  - Các đối tượng lưu trữ dữ liệu.
  - Các đối tượng truy xuất, thao tác, hoặc cung cấp truy xuất đến dữ liệu.
- View và thủ tục (stored procedures) thuộc về danh mục thứ hai.

# Views

View là một bảng ảo, được tạo ra bởi các cột được lấy từ một hoặc nhiều bảng(table) khác nhau

Các bảng mà view được tạo từ đó được gọi là các bảng cơ sở.

Những bảng này có thể trong cùng csdl hoặc từ các csdl khác.

View cũng có thể bao gồm các cột lấy từ một View khác trong cùng csdl hoặc từ csdl khác.

Một View có thể có tối đa 1024 cột.

Dữ liệu bên trong view được lấy từ các bảng cơ sở, là các bảng được tham chiếu trong phần định nghĩa của view.

Các dòng và các cột của view được tạo động khi view được tham chiếu.

# Tạo View 1-3

Có thể tạo một view trong csdl hiện tại bằng việc dùng lệnh `CREATE VIEW`.

Người dùng chỉ có thể tạo view với các cột lấy từ bảng cơ sở hoặc từ các view khác nếu người dùng có quyền truy cập đến các bảng và view đó.

SQL Server kiểm tra sự tồn tại của các đối tượng được tham chiếu trong phần định nghĩa của view.

➤ Cú pháp sử dụng để tạo view như sau:

## Cú pháp:

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

Trong đó,

`view_name`: chỉ ra tên của view.

`select_statement`: chỉ ra câu lệnh `SELECT` định nghĩa view.



## Tạo View 2-3

- Đoạn code sau tạo view từ bảng **Product**, chỉ hiển thị các cột product id, product number, name, và cột SafetyStockLevel.

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

- Trong đoạn code dưới đây được sử dụng để hiển thị chi tiết của view **vwProductInfo**.

```
SELECT * FROM vwProductInfo
```

## Tạo View 3-3

- Kết quả sẽ đưa ra các cột được chỉ ra của tất cả sản phẩm từ bảng **Product**.
- Hình dưới đây trình bày một phần của kết quả.

	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	Chaining Bolts	1000
10	321	CN-6137	Chaining Nut	1000

# Tạo view sử dụng từ khóa JOIN 1-5

Từ khóa JOIN được sử dụng để tạo view.

Câu lệnh CREATE VIEW được sử dụng cùng với từ khóa JOIN để tạo một view sử dụng các cột từ nhiều bảng.

➤ Cú pháp được sử dụng để tạo một view với từ khóa JOIN như sau:

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

Trong đó,

view\_name: chỉ ra tên của view.

table\_name1: chỉ tên của bảng thứ nhất.

JOIN: chỉ ra rằng hai bảng được ghép nối bằng từ khóa JOIN.

table\_name2: chỉ ra tên của bảng thứ hai.



## Tạo view sử dụng từ khóa JOIN 2-5

- Đoạn code sau đây tạo view có tên **vwPersonDetails** với các cột được chỉ ra từ bảng **Person** và bảng **Employee**.
- Từ khóa **JOIN** và **ON** ghép nối dữ liệu hai bảng dựa trên cột **BusinessEntityID**.

```
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
```

- Đây là view có chứa các cột **Title**, **FirstName**, **MiddleName**, và **LastName** từ bảng **Person** và cột **JobTitle** từ bảng **Employee**.

# Tạo view sử dụng từ khóa JOIN 3-5

- Hình sau cho thấy kết quả.

	Title	FirstName	MiddleName	LastName	Job Title
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

- Như hình trên , tất cả các dòng không có giá trị cho cột Title hoặc MiddleName - có thể có NULL trong chúng.
- Một người nhìn vào kết quả này có lẽ không thể A person seeing this output may not be able to comprehend the meaning of the NULL values.

## Tạo view sử dụng từ khóa JOIN 4-5

- Đoạn code sau sử dụng hàm COALESCE () để thay thế tất cả các giá trị NULL trong kết quả bằng chuỗi rỗng(null string).

```
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
```

## Tạo view sử dụng từ khóa JOIN 5-5

- Khi view được truy vấn bằng câu lệnh `SELECT`, kết quả sẽ như hình dưới đây:

	Title	FirstName	MiddleName	LastName	Job Title
1		Ken	J	Sánchez	Chief Executive Officer
2		Teri	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

# Các nguyên tắc và giới hạn trên View 1-2

➤ Trước khi tạo view, các nguyên tắc và giới hạn sau nên được xem xét:

Một view được tạo bằng việc sử dụng câu lệnh `CREATE VIEW`.

Tên view phải là duy nhất, không thể trùng với tên các bảng khác trong cùng lược đồ.

View không thể tạo trên các bảng tạm.

View không thể có full-text index.

View không chứa định nghĩa `DEFAULT`.

Câu lệnh `CREATE VIEW` chỉ có thể bao gồm mệnh đề `ORDER BY` nếu như có từ khóa `TOP` được sử dụng.

View không thể tham chiếu hơn 1024 cột.

Câu lệnh `CREATE VIEW` không thể bao gồm từ khóa `INTO`.

Câu lệnh `CREATE VIEW` không thể kết hợp với các lệnh Transact-SQL khác trong cùng một khối (batch).

# Các nguyên tắc và giới hạn trên View 2-2

- Đoạn code sau sử dụng lại code trước đó với mệnh đề ORDER BY.

```
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
```

- Từ khóa TOP hiển thị tên của 10 nhân viên đầu tiên được sắp xếp thứ tự tăng dần(z->z) theo tên.

# INSERT với Views 1-5

**Cột có thuộc tính `IDENTITY`.**

Câu lệnh `INSERT` được sử dụng để thêm các dòng mới tới bảng hoặc view. Giá trị của cột được cung cấp tự động nếu:

**Cột có giá trị mặc định được chỉ ra.**

**Cột có kiểu dữ liệu `timestamp` .**

**Cột chấp nhận các giá trị `null`.**

**Cột là cột tính toán.**

- Khi sử dụng câu lệnh `INSERT` trên view, nếu vi phạm(violated) bất kỳ quy tắc(rules), bản ghi sẽ không được chèn.

# INSERT với Views 2-5

- Trong ví dụ sau, dữ liệu được chèn vào thông qua xem, nhưng việc chèn không xảy ra khi view được tạo ra từ hai bảng cơ sở.
- Trước tiên, tạo bảng **Employee\_Personal\_Details** như trong đoạn code dưới đây

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

- Sau đó tạo bảng **Employee\_Salary\_Details** như trong đoạn code dưới đây:

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



# INSERT với Views 3-5

- Đoạn code sau đây tạo view **vwEmployee\_Details** sử dụng các cột từ hai bảng **Employee\_Personal\_Details** và **Employee\_Salary\_Details** bằng cách ghép nối hai bảng thông qua cột **EmpID**.

```
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
```

- Đoạn code sau sử dụng lệnh **INSERT** để chèn dữ liệu thông qua view **vwEmployee\_Details**.

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

- Tuy nhiên, dữ liệu không được chèn khi view được tạo từ hai bảng cơ sở.
- Thông điệp lỗi được hiển thị như sau khi câu lệnh **INSERT** được thực thi.  
'Msg 4405, Level 16, State 1, Line 1  
View or function 'vwEmployee\_Details' is not updatable  
because the modification affects multiple base tables.'

# INSERT với Views 4-5

- Các giá trị có thể chèn vào các cột có kiểu dữ liệu do người dùng định nghĩa (user-defined data type) bằng cách:

Chỉ ra một giá trị kiểu người dùng định nghĩa.

Gọi hàm do người dùng định nghĩa có trả về một giá trị kiểu người dùng định nghĩa.

- Các quy tắc sau cần được tuân thủ khi sử dụng câu lệnh `INSERT`:

Câu lệnh `INSERT` phải chỉ ra giá trị cho tất cả các cột trong view mà ở bảng phía dưới không cho phép các giá trị null và không có định nghĩa `DEFAULT`.

Khi có self-join(ghép nối đệ qui) với cùng một view hoặc một bảng cơ sở, câu lệnh `INSERT` không làm việc.

# INSERT với Views 5-5

- Đoạn code dưới đây tạo view **vwEmpDetails** sử dụng bảng **Employee\_Personal\_Details**.

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

- Bảng **Employee\_Personal\_Details** có chứa cột **LastName** không cho phép chèn vào giá trị null.
- Đoạn code dưới đây cố thử chèn các giá trị vào view **vwEmpDetails**.

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

- Việc chèn này không được phép do view có chứa cột **LastName** từ bảng cơ sở và cột này không cho phép null.

# UPDATE với Views 1-5

- Câu lệnh UPDATE có thể được sử dụng để thay đổi dữ liệu trên view.
- Việc cập nhật trên view cũng sẽ cập nhật đến các bảng phía dưới.
- Đoạn code dưới đây tạo bảng có tên **Product\_Details**.

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

- Giả sử có một số bản ghi đã được thêm vào bảng như hình dưới đây:

	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 với Views 2-5

- Đoạn dưới đây tạo một view dựa trên bảng **Product\_Details**.

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

- Đoạn code dưới đây cập nhật view để thay đổi tất cả đơn giá (rates) của DVD writers thành 3000.

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

- Kết quả của đoạn này không ảnh hưởng tác động trên view **vwProduct\_Details**, mà còn tác động đến bảng bên dưới, là bảng mà view được tạo từ nó.
- Hình sau đây cho thấy dữ liệu trong bảng được cập nhật tự động do 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 với Views 3-5

- Kiểu dữ liệu có giá trị lớn gồm có `varchar(max)`, `nvarchar(max)`, và `varbinary(max)`.
- Để cập nhật dữ liệu có kiểu dữ liệu lớn, mệnh đề `.WRITE` được sử dụng.
- Mệnh đề `.WRITE` chỉ ra rằng một đoạn(section) của giá trị trong cột được sửa đổi.
- Không thể sử dụng mệnh đề `.WRITE` để cập nhật giá trị `NULL` trong cột.
- Tương tự, không thể sử dụng để gán giá trị cho cột là `NULL`.

## Cú pháp:

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

Trong đó,

`column_name`: chỉ ra tên của cột có kiểu dữ liệu lớn.

`Expression`: chỉ ra giá trị được copy tới cột.

`@Offset`: chỉ ra điểm bắt đầu của giá trị trong cột mà `Expression` sẽ được ghi tại đó .

`@Length`: chỉ ra độ dài của đoạn(section) trong cột.

- `@Offset` và `@Length` được chỉ bằng byte cho kiểu dữ liệu `varbinary` và kiểu dữ liệu `varchar` và bằng kí tự cho kiểu dữ liệu `nvarchar`.

# UPDATE với Views 4-5

- Giả sử bảng **Product\_Details** được chỉ sửa ở cột **Description**, là cột có kiểu dữ liệu `nvarchar(max)`.
- Đoạn code sau tạo một view dựa trên bảng này, có các cột **ProductName**, **Description**, và **Rate**.

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

- Đoạn code sau sử dụng câu lệnh **UPDATE** trên view **vwProduct\_Details**.
- Mệnh đề **.WRITE** được sử dụng để thay đổi **Internal** trong cột **Description** thành **External**.

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

- Tất cả các dòng trong view có tên **ProductName** là 'Portable Hard Drive' sẽ được cập nhật thành **External** thay cho **Internal** trong cột **Description**.

# UPDATE với Views 5-5

- Hình dưới đây cho thấy kết quả sau khi view được cập nhật.

	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

- Các nguyên tắc sau cần được tuân thủ khi sử dụng câu lệnh UPDATE:

Không thể cập nhật giá trị của cột có thuộc tính **IDENTITY**.

Các bản ghi không thể cập nhật nếu bảng cơ sở có chứa cột **TIMESTAMP**.

Trong khi đang cập nhật dòng, nếu constraint hoặc rule bị vi phạm, câu lệnh bị kết thúc, một lỗi được trả về và không có bản ghi nào được cập nhật.

Khi có một ghép nối đệ quy (self-join) với cùng view hoặc bảng cơ sở, câu lệnh **UPDATE** không làm việc.



# DELETE với Views 1-2

- Có thể dùng câu lệnh `DELETE` để xóa các dòng dữ liệu từ view.
- Khi các dòng được xóa khỏi view, các dòng tương ứng trong bảng cơ sở cũng được xóa.
- Ví dụ, xem xét view **vwCustDetails** liệt kê thông tin tài khoản của nhiều khách hàng khác nhau.
- Khi khách hàng đóng tài khoản, chi tiết của khách hàng cần được xóa.
- Cú pháp được sử dụng để xóa dữ liệu từ view như sau:

## Cú pháp:

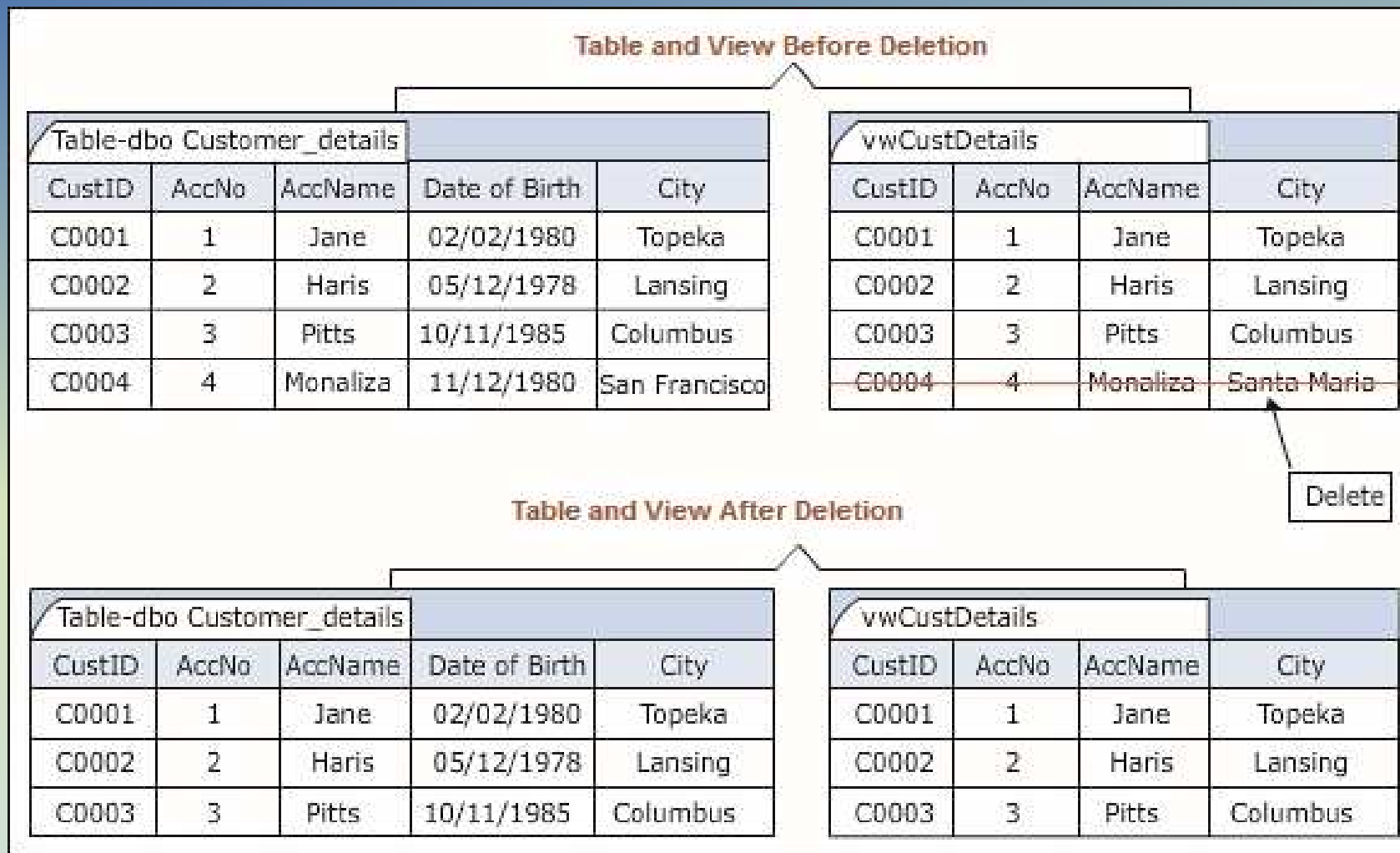
```
DELETE FROM <view_name>  
WHERE <search_condition>
```

- Giả sử một bảng có tên **Customer\_Details**, và view có tên **vwCustDetails** được tạo dựa trên bảng này.
- Đoạn code sau đây được sử dụng để xóa các bản ghi có **CustID** là C0004 khỏi view **vwCustDetails**.

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

# DELETE với Views 2-2

- Hình dưới đây minh họa logic của việc xóa dữ liệu từ view.



# Sửa View (Altering Views) 1-2

Bên cạnh việc chỉnh sửa dữ liệu bên trong view, người dùng cũng có thể chỉnh sửa định nghĩa của view.

Một view có thểA view can được chỉnh sửa bằng cách xóa đi sau đó tạo lại hoặc thực thi câu lệnh `ALTER VIEW`.

Câu lệnh `ALTER VIEW` giúp chỉnh sửa view tồn tại mà không cần phải phân quyền hoặc thiết lập lại các thuộc tính khác của nó.

`ALTER VIEW` có thể được áp dụng cho indexed views; tuy nhiên nó xóa vô điều kiện tất cả các index trên view.

View thường được chỉnh sửa khi người dùng cần bổ sung thông tin hoặc thay đổi các bảng cơ sở trong phần định nghĩa.

## Sửa View (Altering Views) 2-2

- Cú pháp được sử dụng để sửa view:

### Cú pháp:

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

- Đoạn code sau đây sửa view **vwProductInfo**, view có thêm cột **ReOrderPoint**.

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

# Xóa View

- Một view có thể xóa khỏi csdl bằng lệnh `DROP VIEW`.
- Khi xóa view, dữ liệu trong bảng cơ sở không bị ảnh hưởng(vẫn còn).
- Định nghĩa của view và thông tin khác gắn kết với view bị xóa khỏi system catalog.
- Tất cả các quyền hạn gán cho view cũng được xóa.
- Cú pháp được sử dụng để xóa view như sau:

## Cú pháp:

```
DROP VIEW <view_name>
```

- Đoạn code dưới đây xóa view **vwProductInfo**.

```
DROP VIEW vwProductInfo
```

# Định nghĩa của View

- Định nghĩa của view giúp để hiểu dữ liệu được lấy từ các bảng nguồn như thế nào.
- Thủ tục `sp_helptext` hiển thị thông tin có liên quan đến view khi tên của view được chỉ ra cho tham số của nó.
- Cú pháp được sử dụng để xem thông tin định nghĩa của view như sau:

## Cú pháp:

```
sp_helptext <view_name>
```

- Đoạn code dưới đây hiển thị thông tin về view **vwProductPrice**.

```
EXEC sp_helptext vwProductPrice
```

- Sau khi thực thi đoạn code, thông tin định nghĩa của view được hiển thị như hình dưới đây:

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

# Tạo view có sử dụng hàm có sẵn

- Khi các hàm được sử dụng, cột được dẫn xuất này phải bao gồm cả tên trong câu lệnh CREATE VIEW.
- Xem xét view có sử dụng hàm AVG ( ) được tạo trong đoạn code dưới đây:

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

- Ở đây, hàm AVG ( ) tính giá trị đơn giá(rate) trung bình của mỗi sản phẩm bằng mệnh đề GROUP BY.
- Hình dưới đây cho thấy kết quả khi view được truy vấn.

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

# Mệnh đề CHECK OPTION 1-2

Tùy chọn CHECK OPTION được sử dụng để đảm bảo toàn vẹn miền giá trị (domain integrity); Nó kiểm tra các giá trị khi cập nhật trên view phải thỏa mãn điều kiện được chỉ ra ở mệnh đề WHERE trong câu lệnh SELECT.

Mệnh đề WITH CHECK OPTION đảm bảo tất cả các câu lệnh chỉnh sửa được thực thi đối với view phải tuân thủ điều kiện được thiết lập bên trong câu lệnh SELECT.

- Cú pháp tạo view có sử dụng CHECK OPTION như sau:

## Cú pháp:

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

trong đó,

WITH CHECK OPTION: chỉ ra rằng dữ liệu chỉnh sửa trong view tiếp tục thỏa mãn định nghĩa của view.



## Mệnh đề CHECK OPTION 2-2

- Đoạn code sau tạo lại view **vwProductInfo** có `SafetyStockLevel` nhỏ hơn 1000:

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

- Trong đoạn code sau, câu lệnh `UPDATE` được sử dụng để chỉnh sửa view **vwProductInfo** bằng cách thay đổi giá trị của cột `SafetyStockLevel` cho sản phẩm (product) có id là 321 thành 2500.

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

- Câu lệnh `UPDATE` bị lỗi thực thi khi nó vi phạm định nghĩa của view, vì giá trị `SafetyStockLevel` chỉ ra phải nhỏ hơn 1000.
- Do vậy không có dòng nào trong view **vwProductInfo** bị ảnh hưởng.

# Tùy chọn SCHEMABINDING 1-2

Một view có thể được ràng buộc tới lược đồ của bảng cơ sở bằng tùy chọn SCHEMABINDING.

Khi tùy chọn SCHEMABINDING được chỉ ra, các bảng cơ sở sẽ không thể chỉnh sửa được nữa, vì nó sẽ làm ảnh hưởng tới định nghĩa của view.

Nếu muốn chỉnh sửa được bảng, trước tiên cần phải chỉnh sửa hoặc xóa bỏ sự phụ thuộc giữa view với bảng.

Khi sử dụng tùy chọn SCHEMABINDING trong view, tên của các đối tượng được chỉ ra trong câu lệnh SELECT cần phải kèm cùng với tên của lược đồ.

➤ Cú pháp được sử dụng để tạo view với tùy chọn SCHEMABINDING như sau:

## Cú pháp:

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

trong đó,

view\_name: chỉ ra tên của view.



## Tùy chọn SCHEMABINDING 2-2

WITH SCHEMABINDING: chỉ ra rằng view view được gắn với lược đồ.  
select\_statement: chỉ ra câu lệnh SELECT định nghĩa view.

- Đoạn code dưới đây tạo view **vwNewProductInfo** với tùy chọn SCHEMABINDING để ràng buộc view với lược đồ Production, đây là lược đồ của bảng Product.

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

# Sử dụng sp\_refreshview 1-3

- Thủ tục lưu sp\_refreshview cập nhật metadata(thông tin mô tả) cho view.
- Nếu thủ tục sp\_refreshview không được thực thi, metadata của view không được cập nhật để phản ánh(reflect) các sự thay đổi của các bảng cơ sở.
- Điều này dẫn đến việc tạo ra kết quả không mong muốn(unexpected) khi view được truy vấn.
- Thủ tục lưu sp\_refreshview được trả về giá trị mã là 0 nếu thực thi thành công, hoặc trả về số khác 0 trong trường hợp thực thi có lỗi.
- Cú pháp được sử dụng để chạy thủ tục sp\_refreshview như sau:

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

- Đoạn code dưới đây tạo bảng **Customers** với các cột **CustID**, **CustName**, và **Address**.

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



## Sử dụng sp\_refreshview 2-3

- Đoạn code dưới đây tạo view **vwCustomers** dựa trên bảng **Customers**.

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

- Đoạn code sau thực thi truy vấn SELECT trên view.

```
SELECT * FROM vwCustomers
```

- Kết quả hiển thị ra ba cột: **CustID**, **CustName**, và **Address**.
- Đoạn code sau sử dụng câu lệnh ALTER TABLE để thêm cột Age tới bảng **Customers**.

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

- Đoạn code sau thực thi câu truy vấn SELECT trên view.

```
SELECT * FROM vwCustomers
```

## Sử dụng sp\_refreshview 3-3

- Cột **Age** được cập nhật nhưng không hiển thị trên view.
- Để giải quyết điều này, thủ tục lưu `sp_refreshview` phải được thực thi trên view **vwCustomers** như được trình bày trong đoạn code dưới đây:

```
EXEC sp_refreshview 'vwCustomers'
```

- Khi truy vấn `SELECT` được chạy lại trên view, cột **Age** được nhìn thấy trong kết quả.
- Điều này là vì thủ tục `sp_refreshview` làm tươi(refreshes) metadata cho view **vwCustomers**.
- Xem xét view ràng buộc với lược đồ được phụ thuộc vào bảng **Production.Product**
- Đoạn code sau thử thay đổi(alter) kiểu dữ liệu của cột **ProductID** trong bảng **Production.Product** từ kiểu `int` sang `varchar(7)`.

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

- Một thông báo lỗi được hiển thị khi bảng được ràng buộc lược đồ với view **vwNewProductInfo**, do vậy không thể sửa đổi được vì nó vi phạm định nghĩa của view.

# Thủ tục lưu (Stored Procedure)

Stored Procedure là một nhóm nhiều câu lệnh Transact-SQL có vai trò như một khối mã lệnh thực hiện một tác vụ cụ thể, đã được biên dịch và lưu trữ trong SQL Server dưới một cái tên nào đó và được xử lý như một đơn vị

Một thủ tục cũng có thể tham chiếu tới phương thức .NET Framework Common Language Runtime(CLR).

Stored procedure được dùng cho những công việc được thực hiện nhiều lần

Stored procedure có thể nhận nhiều giá trị qua các tham số đầu vào và trả về các giá trị kết quả qua các tham số ra.

➤ Sử dụng stored procedure có một số lợi ích sau:

Improved Security

Precompiled Execution

Reduced Client/Server Traffic

Reuse of Code

# Các loại thủ tục lưu 1-3

- SQL Server hỗ trợ các loại thủ tục lưu sau:

## Thủ tục lưu do người dùng định nghĩa (User-Defined Stored Procedures)

Thủ tục lưu do người dùng định nghĩa hay còn gọi là thủ tục lưu tùy biến

Các thủ tục lưu này được dùng để tái sử dụng các câu lệnh Transact-SQL của các bài toán được thực hiện có tính lặp đi lặp lại.

Có hai loại thủ tục do người dùng định nghĩa là: thủ tục lưu Transact-SQL (Transact-SQL stored procedure) và thủ tục lưu CLR (Common Language Runtime (CLR) ).

Thủ tục lưu Transact-SQL gồm có các câu lệnh Transact-SQL trong khi thủ tục lưu CLR dựa trên các phương thức .NET framework CLR.

Cả hai thủ tục lưu có thể nhận và về các tham số do người dùng định nghĩa Both the stored procedures can take and return user-defined parameters.



# Các loại thủ tục lưu 2-3

## Thủ tục lưu mở rộng (Extended Stored Procedure)

Thủ tục nội tại mở rộng giúp SQL Server tương tác với hệ điều hành

Thủ tục lưu mở rộng không phải là đối tượng nội trú của SQL Server.

Chúng là các thủ tục được thực thi ở dạng các thư viện liên kết động [dynamic-link libraries (DLL)] được thực thi bên ngoài môi trường SQL Server.

Ứng dụng tương tác với SQL Server gọi DLL lúc thực thi. DLL được tải(load) động và chạy bởi SQL Server.

SQL Server được cấp phát không gian để chạy các thủ tục lưu mở rộng.

Thủ tục nội mở rộng có tên được bắt đầu với 'xp'.

Các công việc (Tasks)phức tạp hoặc không thể thực thi bằng câu lệnh Transact-SQL thì sẽ được thực thi bằng thủ tục lưu mở rộng.

# Các loại thủ tục lưu 3-3

## Thủ tục lưu hệ thống (System Stored Procedures)

Thủ tục lưu hệ thống được sử dụng phổ biến cho việc tương tác với các bảng hệ thống và cho công việc thực hiện các bài toán quản trị như cập nhật các bảng hệ thống.

Thủ tục lưu hệ thống được bắt đầu với tiền tố 'sp\_'. Các thủ tục này được đặt trong cơ sở dữ liệu Resource.

Có thể nhìn thấy các thủ tục này trong lược đồ sys của mọi csdl hệ thống và người dùng định nghĩa.

Thủ tục nội tại hệ thống cho phép các quyền hạn GRANT, DENY, và REVOKE.

Một thủ tục lưu hệ thống là một tập các câu lệnh Transact-SQL được biên dịch trước được thực thi như một đơn vị .

Thủ tục lưu hệ thống được sử dụng trong hoạt động quản trị csdl.

Khi tham chiếu một thủ tục hệ thống, bộ định danh lược đồ sys được sử dụng. Thủ tục lưu hệ thống được sở hữu bởi người quản trị csdl (database administrator).

# Phân loại thủ tục lưu hệ thống 1-2

## Thủ tục lưu danh mục (Catalog Stored Procedures)

- Tất cả thông tin về các bảng trong csdl người dùng được lưu trữ trong tập các bảng được gọi là danh mục hệ thống (system catalog).
- Thông tin từ bảng danh mục hệ thống có thể được truy xuất bằng các thủ tục danh mục (catalog procedures).
- Ví dụ thủ tục lưu catalog `sp_tables` hiển thị danh sách tất cả các bảng trong csdl hiện tại.

## Thủ tục lưu bảo mật (Security Stored Procedures)

- Thủ tục lưu bảo mật được sử dụng để quản lý an ninh của csdl.
- Ví dụ, thủ tục lưu bảo mật `sp_changedbowner` được sử dụng để thay đổi chủ sở hữu của csdl hiện tại.

## Thủ tục lưu Cursor (Cursor Stored Procedures)

- Thủ tục lưu Cursor được sử dụng thực thi các chức năng của một cursor.
- Ví dụ, thủ tục cursor `sp_cursor_list` liệt kê tất cả các cursor được mở bởi kết nối và mô tả các thuộc tính của chúng.



# Phân loại thủ tục lưu hệ thống 2-2

## Distributed Query Stored Procedures

- Thủ tục lưu phân tán được sử dụng trong việc quản lý các truy vấn phân tán.
- Ví dụ, thủ tục lưu truy vấn phân tán `sp_indexes` trả về thông tin index cho bảng từ xa(remote table) được chỉ ra.

## 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.

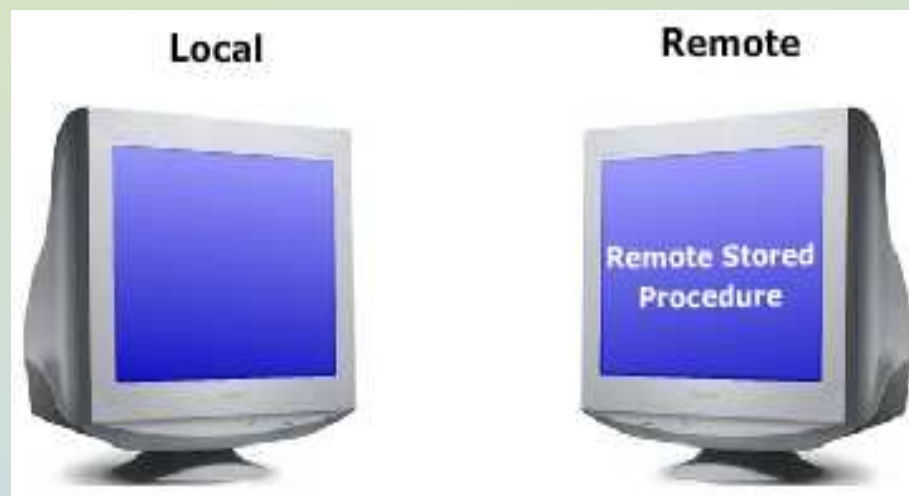
# Phân loại thủ tục tạm thời (Temporary)

- Thủ tục được tạo ra để sử dụng trong một phiên làm việc (session) được gọi là thủ tục tạm.
- Các thủ tục tạm được tạo ra thường được lưu trong csdl `tempdb`.
- CSLD hệ thống `tempdb` là tài nguyên toàn cục sẵn có cho tất cả người dùng có kết nối tới một thể hiện (instance).
- SQL Server cho phép tạo hai loại thủ tục tạm là cục bộ và toàn cục.
- Bảng liệt kê sự biệt giữa hai thủ tục cục bộ và toàn cục.

Thủ tục tạm thời cục bộ	Thủ tục tạm thời toàn cục
Chỉ có người tạo mới có thể nhìn thấy và sử dụng.	Tất cả người dùng đều có thể nhìn thấy và sử dụng.
Tự động bị xóa khi phiên làm việc kết thúc.	Bị xóa khi tất cả các phiên làm việc kết thúc.
Tên thủ tục được bắt đầu bằng #.	Tên thủ tục được bắt đầu bằng ##.

# Thủ tục nội tại từ xa (remote)

- Thủ tục nội tại chạy trên SQL Server của một máy tính xa được gọi là remote stored procedure.
- Remote stored procedure chỉ có thể dùng được khi server ở xa cho phép truy cập từ xa (remote access).
- Khi một thủ tục lưu từ xa được thực thi từ một thể hiện SQL Server cục bộ tới một máy tính client, một lệnh hủy bỏ lỗi có thể gặp phải.
- Khi một lỗi như vậy xảy ra, những lệnh gây ra lỗi được chấm dứt nhưng các thủ tục từ xa vẫn tiếp tục được thực hiện.



# Thủ tục nội tại mở rộng

Thủ tục nội tại mở rộng (extended SP) là các thủ tục thực thi các mã lệnh không nằm trong SQL Server, nghĩa là không được viết bởi các lệnh T-SQL.

Các thủ tục nội tại mở rộng có tên bắt đầu bằng tiền tố 'xp\_', và chúng được chứa trong lược đồ dbo của csdl master

- Cú pháp thực thi được sử dụng như sau:

```
EXECUTE <procedure_name>
```

- Ví dụ: Dùng thủ tục nội tại mở rộng xp\_fileexist để kiểm tra xem tập tin MyTest.txt có tồn tại hay không

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

# Thủ tục nội tại do người dùng định nghĩa 1-3

Trong SQL Server, người dùng được phép tạo thủ tục lưu theo ý mình để thực hiện nhiều bài toán khác nhau.

Cần phải có quyền CREATE PROCEDURE và ALTER để có thể tạo và sửa thủ tục .

- Cú pháp tạo thủ tục người dùng định nghĩa:

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

Trong đó,

procedure\_name: chỉ ra tên của thủ tục.

@parameter: chỉ ra tham số vào/ra của thủ tục.

data\_type: chỉ ra kiểu dữ liệu của tham số.

sql\_statement: là một hay nhiều câu lệnh Transact-SQL.





# Thủ tục nội tại do người dùng định nghĩa 2-3

- Đoạn code dưới đây tạo một thủ tục người dùng định nghĩa với tên **uspGetCustTerritory**, thủ tục sẽ hiển thị chi tiết khách hàng gồm customer id, territory id, và 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
```

- Đoạn code thực thi thủ tục bằng câu lệnh EXEC

```
EXEC uspGetCustTerritory
```

# Thủ tục nội tại do người dùng định nghĩa 3-3

➤ Kết quả khi thực thi thủ tục như hình sau:

	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

# Tham số của thủ tục

- Dữ liệu có thể được truyền từ chương trình gọi đến thủ tục được gọi bằng việc dùng các tham số.
- Tham số được chia thành hai loại
  - **Tham số vào:** cho phép chương trình gọi truyền các giá trị tới một thủ tục. Các giá trị này được gán cho các biến đã được định nghĩa bên trong thủ tục.
  - **Tham số ra:** cho phép thủ tục truyền các giá trị ngược trở lại chương trình gọi. Các giá trị này được gán cho các biến ở chương trình gọi.

# Tham số vào (Input Parameters) 1-2

- Các giá trị được truyền vào từ chương trình gọi tới thủ tục, và được nhận(lưu) vào các tham số vào của thủ tục.
- Tham số vào được khai báo ngay lúc tạo thủ tục.
- Các giá trị truyền cho tham số vào có thể là các hằng và cũng có thể là các biến.
- Các giá trị này được truyền tại lúc gọi thủ tục.
- Thủ tục sử dụng các giá trị này để thực hiện các công việc đã được chỉ ra.
- Cú pháp tạo thủ tục với tham số vào như sau:

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

Trong đó,

data\_type: chỉ ra kiểu dữ liệu.

- Cú pháp thực thi thủ tục và truyền các giá trị cho tham số vào:

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

## Tham số vào (Input Parameters) 2-2

- Đoạn code sau đây tạo thủ tục với tên **uspGetSales** có một tham số để nhận vào tên của một khu vực (miền) là territory, thủ tục hiển thị chi tiết bán hàng và id người của người bán ứng với khu vực được truyền vào.
- Sau đó, là lệnh gọi thực thi thủ tục với giá trị truyền cho tham số vào territory là khu vực Northwest.

```
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'
```

- Kết quả được minh họa trong hình sau:

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

# Tham số ra (Output Parameter) 1-3

Tham số ra được khai báo ngay lúc tạo thủ tục.

Để chỉ ra một tham số là tham số ra, từ khóa OUTPUT được sử dụng để khai báo tham số.

Cũng tương tự, câu lệnh gọi thủ tục cũng phải có biến được chỉ ra với từ khóa OUTPUT để nhận kết quả từ thủ tục được gọi.

- Cú pháp gọi thủ tục có tham số ra OUTPUT.

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

## Tham số ra (Output Parameter) 2-3

- Đoạn code tạo một thủ tục `uspGetTotalSales` với tham số đầu vào `@territory` để nhận tên quốc gia và tham số đầu ra `@sum` để hiển thị tổng bán đến năm hiện tại của quốc gia được truyền vào

```
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
```

- Đoạn code sau khai báo biến `sumsales` để nhận kết quả của thủ tục `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
```

## Tham số ra (Output Parameter) 3-3

➤ Các tham số OUTPUT ra có các đặc điểm sau:

Kiểu dữ liệu của tham số không thể là kiểu text và image

Câu lệnh gọi thủ tục phải chứa một biến để nhận giá trị trả về

Biến có thể sử dụng trong các câu lệnh Transact-SQL tiếp theo trong cùng lô(batch) hoặc lớp gọi thủ tục

Các tham số ra có thể là các giữ chỗ cho cursor (các biến kiểu cursor)



# Sử dụng SSMS để tạo thủ tục 1-5

- Bạn có thể tạo thủ tục do người dùng định nghĩa bằng SSMS theo các bước dưới đây:

1

- Mở **Object Explorer**.

2

- Trong **Object Explorer**, kết nối tới thể hiện(instance) của Database Engine.

3

- Sau khi kết nối thành công tới thể hiện(instance), hãy mở thể hiện (click và dấu + ở phần đầu của thể hiện).

4

- Mở **Databases** và mở csdl **AdventureWorks2012** database.

5

- Mở **Programmability**, click chuột phải **Stored Procedures**, và sau đó nhấn **New Stored Procedure**.

6

- Trên menu **Query**, nhấn **Specify Values for Template Parameters**. Hộp thoại **Specify Values for Template Parameters** sẽ hiển thị.

# Sử dụng SSMS để tạo thủ tục 2-5

- Hộp thoại **Specify Values for Template Parameters** sẽ hiển thị như hình sau:

Parameter	Type	Value
Author		Name
Create Date		
Description		
Procedure_Name	sysname	ProcedureName
@Param1	sysname	@p1
Datatype_For_Param1		int
Default_Value_For_P...		0
@Param2	sysname	@p2
Datatype_For_Param2		int
Default_Value_For_P...		0

# Sử dụng SSMS để tạo thủ tục 3-5

7

- Trong hộp thoại **Specify Values for Template Parameters**, nhập vào các giá trị cho các tham số như trong bảng dưới đây :

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

- Sau khi nhập đầy đủ, bấm **OK**.

# Sử dụng SSMS để tạo thủ tục 4-5

9

- Trong trình soạn thảo truy vấn Query Editor, thay câu lệnh **SELECT** bằng câu lệnh sau:

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

10

- Để kiểm tra cú pháp, trên menu **Query**, nhấn **Parse**. Nếu có một thông báo lỗi được tra về, so sánh câu lệnh với thông tin và cần sửa cho đúng..

11

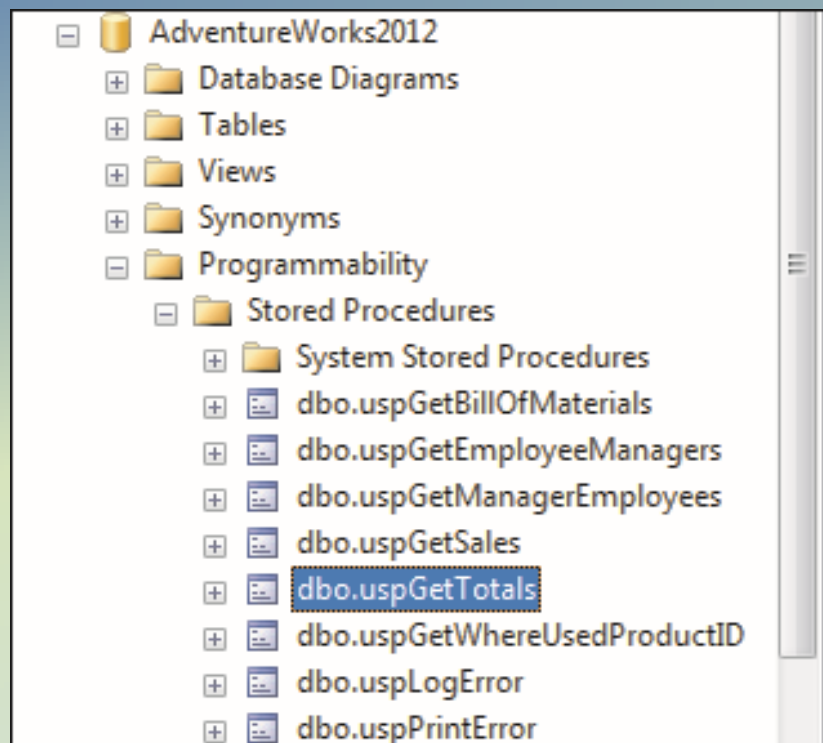
- Để tạo thủ tục, từ menu **Query**, nhấn **Execute**. Thủ tục được tạo là một đối tượng trong cơ sở dữ liệu.

12

- Để xem danh sách các thủ tục trong **Object Explorer**, nhấn chuột phải trên **Stored Procedures** và chọn **Refresh**.

# Sử dụng SSMS để tạo thủ tục 5-5

- Tên thủ tục sẽ được hiển thị trong cây **Object Explorer** như hình sau :



10

- Để chạy thủ tục trong **Object Explorer**, nhấn chuột phải vào tên thủ tục `uspGetTotals` và chọn **Execute Stored Procedure**.

11

- Trong cửa sổ **Execute Procedure**, nhập vào giá trị **Northwest** cho tham số `@territory`.



# Xem định nghĩa của thủ tục

- Thủ tục lưu hệ thống `sp_helptext` được sử dụng để xem phần định nghĩa của một thủ tục. Cần truyền cho nó tên của thủ tục cần xem
- Cú pháp sử dụng thủ tục `sp_helptext`

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

- Đoạn code sau minh họa xem phần định nghĩa của thủ tục **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.

When a stored procedure is altered, the permissions defined for the stored procedure remain the same even though the procedure definition is changed.

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.

The called procedures can in turn call other procedures.

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

The maximum level of nesting supported by SQL Server 2012 is 32.

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.

	NestLevel
1	1

	NestLevel With Execute
1	2

	NestLevel With sp_executesql
1	3



# 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:

## System Catalog Views


- 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

## Information Schema Views

- 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;
```

## System Metadata Functions

- 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.

DMVs and DMFs are collectively referred to as dynamic management objects.

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.

Both DMVs and DMFs return data in tabular format but the difference is that while a DMF normally accepts at least one parameter, a DMV does not accept parameters.

SQL Server 2012 provides nearly 200 dynamic management objects.

In order to query DMVs, it is required to have `VIEW SERVER STATE` or `VIEW DATABASE STATE` permission, depending on the scope of the DMV.

# 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.