ADVANCED DATABASE

View, stored procedure, function, and trigger

Dr. NGUYEN Hoang Ha

Email: nguyen-hoang.ha@usth.edu.vn

Agenda

- View
- SP
- Function
- Trigger

VIEW

View

- Definition: a virtual relation based on the result-set of a SELECT statement
- Syntax:

  CREATE VIEW view_name **AS**

  SELECT column_name(s)

  FROM table_name

  WHERE condition
- Uses:
  - Restrict data access
  - Hide sensitive data
    - Names of tables and columns
  - Simplify data
  - Reuse complex queries

Example

WITH SCHEMABINDING
Avoid removing dependent objects

```
ALTER VIEW Partners WITH SCHEMABINDING AS

SELECT CustomerID PartnerID, CompanyName, 'C' AS [Type]

FROM dbo.Customers

UNION

SELECT CAST(SupplierID AS nvarchar) PartnerID, CompanyName,
'S' AS [Type]

FROM dbo.Suppliers
```

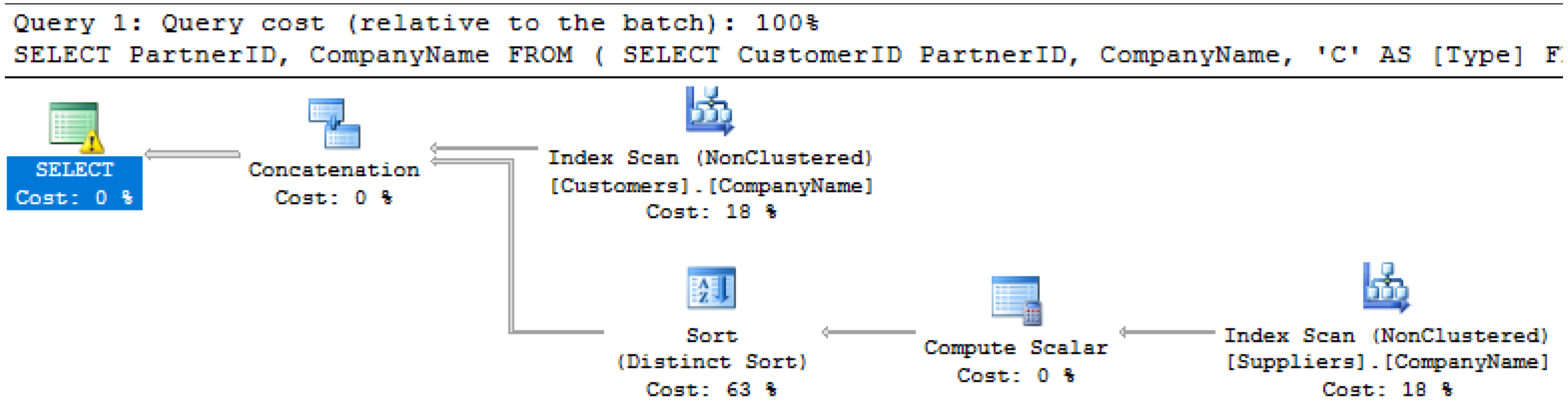
What happens when querying a view ?

```sql
ALTER VIEW Partners WITH SCHEMABINDING AS
SELECT CustomerID PartnerID, CompanyName, 'C' AS [Type]
FROM dbo.Customers
UNION
SELECT CAST(SupplierID AS nvarchar) PartnerID, CompanyName, 'S' AS [Type]
FROM dbo.Suppliers
```

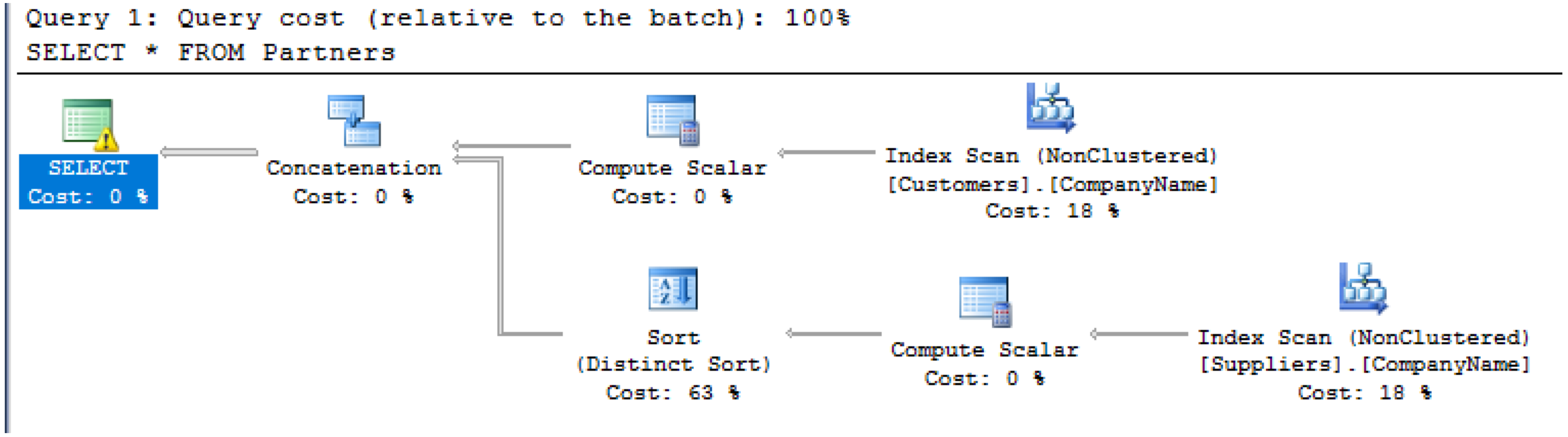
```sql
SELECT PartnerID, CompanyName
FROM Partners
WHERE CompanyName LIKE 'A%'
ORDER BY CompanyName
```

```sql
SELECT PartnerID, CompanyName
FROM (
    SELECT CustomerID PartnerID, CompanyName, 'C' AS [Type]
    FROM Customers
    UNION
    SELECT CAST(SupplierID AS nvarchar) PartnerID, CompanyName, 'S' AS [Type]
    FROM Suppliers) AS S
WHERE CompanyName LIKE 'A%'
ORDER BY CompanyName
```

Analyze query with Execution Plan



Query 1: Query cost (relative to the batch): 100%
SELECT PartnerID, CompanyName FROM (SELECT CustomerID PartnerID, CompanyName, 'C' AS [Type] F



Query 1: Query cost (relative to the batch): 100%
SELECT * FROM Partners

Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand – slower at runtime
  - Always up to date
- Materialized views
  - Used in data warehouses
  - Pre-computed offline – faster at runtime
  - May have stale data

Performance tuning

Modify data of views

- Modify a view → modify base tables
- Restrictions:
  - View contains joins between multiple tables → only INSERT and UPDATE one table, can't DELETE rows
  - Views based on UNION, GROUP BY, DISTINCT → can't modify
  - Can't UPDATE text and image columns

Modifiable views - INSERT

- Define view

```
CREATE VIEW CustomersParis AS
SELECT CompanyName, ContactName, Phone, City
FROM Customers
WHERE City = 'Paris'
```

- What happen?

```
INSERT INTO CustomersParis (CompanyName, ContactName)
VALUES ('Techmaster', 'Peter Pan')
```

- How to solve?

```
ALTER VIEW CustomersParis AS
SELECT CustomerID, CompanyName, ContactName, Phone, City
FROM Customers
WHERE City = 'Paris'
WITH CHECK OPTION
GO
INSERT INTO vwCustomersParis (CustomerID, CompanyName, ContactName, City)
VALUES ('TMVN', 'Techmaster', 'Peter Pan', 'Paris')
```

Modifiable views - UPDATE

- Join-based view – update only one side

```sql
CREATE VIEW vwCategoriesProducts AS
SELECT Categories.CategoryName, Products.ProductID,
       Products.ProductName
FROM Products INNER JOIN Categories
 ON Products.CategoryID = Categories.CategoryID
```

✓

```sql
UPDATE vwCategoriesProducts
SET ProductName = 'Chay'
WHERE ProductID = 1
```

✓

```sql
UPDATE vwCategoriesProducts
SET CategoryName = 'Drinks'
WHERE ProductID = 1
```

✗

```sql
UPDATE vwCategoriesProducts
SET ProductName = 'Chay', CategoryName = 'Drinks'
WHERE ProductID = 1
```

Modifiable views - DELETE

- Define view

```
CREATE VIEW CustomersParis AS
SELECT CustomerID, CompanyName, ContactName, Phone,
City
FROM Customers
WHERE City = 'Paris'
```

- Run query

```
DELETE FROM  CustomersParis
WHERE CustomerID = 'TMVN'
```

→ Data in base table deleted

Ensuring the data consistency of view

- Using WITH CHECK OPTION

```
CREATE VIEW CustomersParis AS
SELECT CompanyName, ContactName, Phone, City
FROM Customers
WHERE City = 'Paris‘
WITH CHECK OPTION
```

- Try

```
UPDATE CustomersParis
SET City = 'Lyon'
```

```
INSERT INTO CustomersParis (CompanyName, ContactName)
VALUES ('Techmaster', 'Peter Pan')
```

STORED PROCEDURE

Stored Procedure (SP)

- SP is a collection of T-SQL statements that SQL Server compiles into a single execution plan.
- SP is stored in cache area of memory when it is first executed so that it can be used repeatedly, not need recompiled
- Parameters:
  - Input
  - Output

SP Syntax

[ENCRYPTION]
[RECOMPILE]
[EXECUTE AS username]

```
CREATE [ OR ALTER ] { PROC | PROCEDURE }
    [schema_name.] procedure_name
    [ { @parameter [ type_schema_name. ] data_type }
        [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
    ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS
{ [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
```

```
DROP PROC [schema_name.] procedure_name
```

Stored Procedure vs. SQL Statement

SQL Statement

First Time
- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

Second Time
- *Check syntax*
- *Compile*
- *Execute*
- *Return data*

Stored Procedure

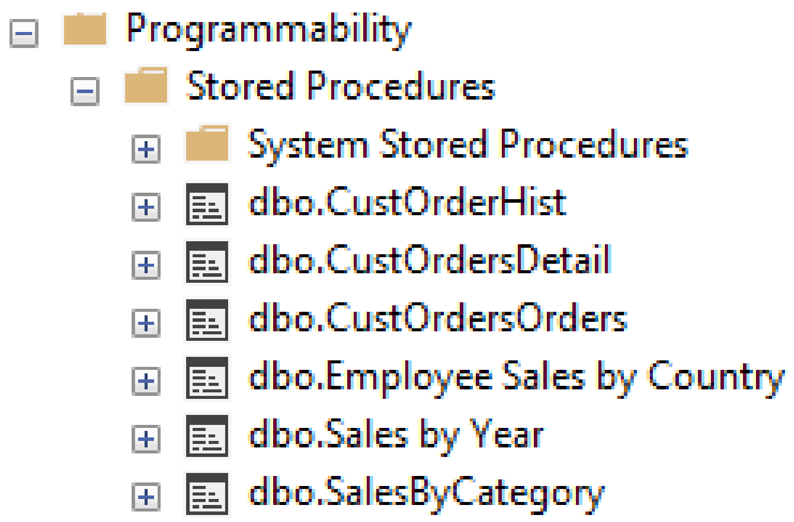
Creating
- *Check syntax*
- *Compile*

First Time
- *Be loaded*
- *Execute*
- *Return data*

Second Time
- *Execute*
- *Return data*

Types of SP

- System stored procedure:
  - Name begins with sp_
  - Created in master database
  - For application in any database
  - Often used by sysadmins
- Local stored procedure:
  - Defined in the local database



Executing a SP

- EXEC pr_GetTopProducts
- With parameters
  - By Name:

```
EXEC pr_GetTopProducts
  @StartID = 1, @EndID = 10
```

  - By Position:

```
EXEC pr_GetTopProducts 1, 10
```

  - Leveraging Default values

```
EXEC pr_GetTopProducts @EndID=10
```

  - Place parameters with default values at the end of the list for flexibility of use

Output parameters

- Used to send non-recordset information back to client
- Example: returning identity field

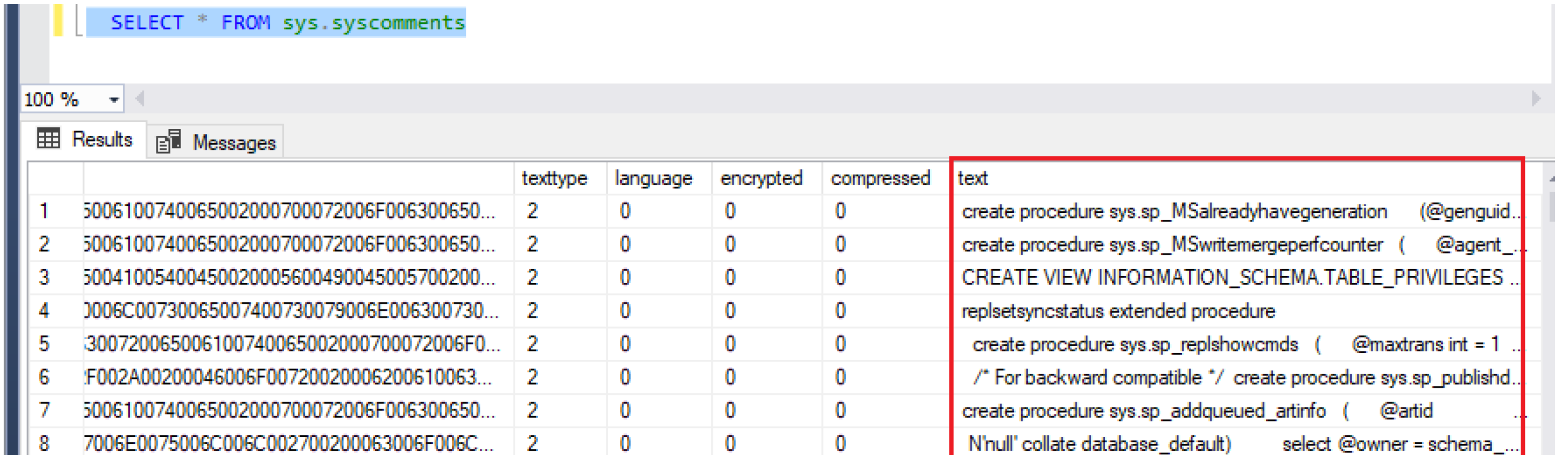
```
CREATE PROC InsertSuppliers
@CompanyName nvarchar(40), @returnID int OUTPUT
AS
INSERT INTO Suppliers(CompanyName) VALUES (@CompanyName)
SET @returnID = @@IDENTITY

GO

DECLARE @ID int
EXEC InsertSuppliers @CompanyName = 'NewTech', @returnID = @ID OUTPUT
SELECT @ID
```

Encrypting stored procedures

- When the stored procedures created, the text for them is saved in the *SysComments* table.
- If the stored procedures are created with the "WITH ENCRYPTION" then the text in *SysComments* is not directly readable
- "WITH ENCRYPTION" is a common practice for software vendors



```
SELECT * FROM sys.syscomments
```

100 %

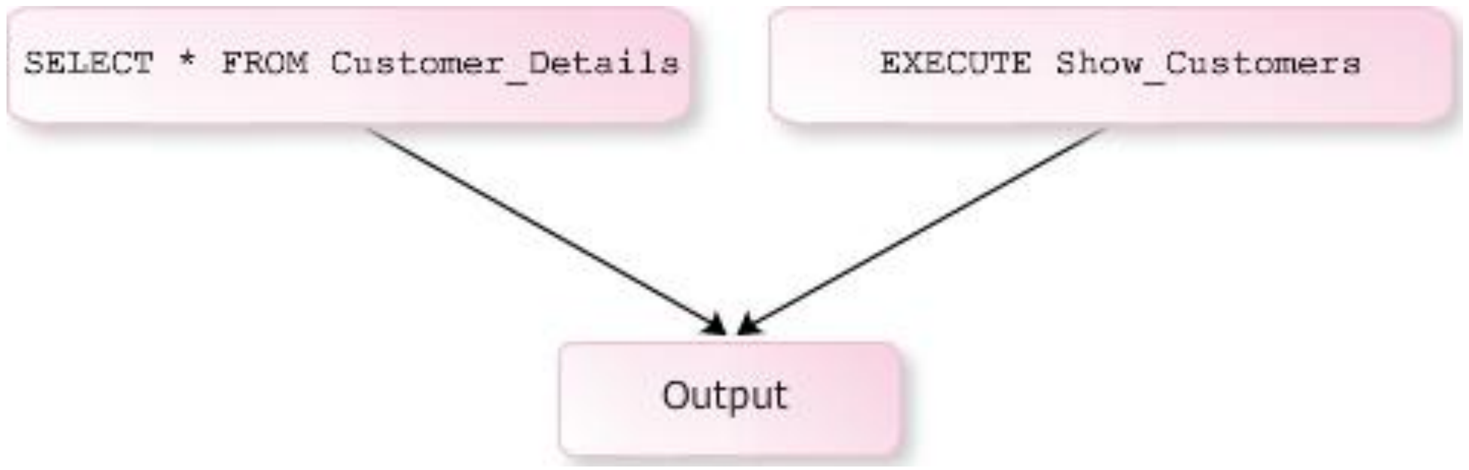
Results | Messages

		texttype	language	encrypted	compressed	text
1	50061007400650020007000072006F006300650...	2	0	0	0	create procedure sys.sp_MSalreadyhavegeneration (@genguid..
2	50061007400650020007000072006F006300650...	2	0	0	0	create procedure sys.sp_MSwritemergeperfcounter (@agent_..
3	500410054004500200056004900450057002000...	2	0	0	0	CREATE VIEW INFORMATION_SCHEMA.TABLE_PRIVILEGES ..
4	0006C00730065007400730079006E006300730...	2	0	0	0	replsetsyncstatus extended procedure
5	300720065006100740065002000700072006F0...	2	0	0	0	create procedure sys.sp_replshowcmds (@maxtrans int = 1 ..
6	F002A00200046006F00720020006200610063...	2	0	0	0	/* For backward compatible */ create procedure sys.sp_publishd..
7	50061007400650020007000072006F006300650...	2	0	0	0	create procedure sys.sp_addqueued_artinfo (@artid ..
8	7006E0075006C006C00270020006300F006C...	2	0	0	0	N'null' collate database_default) select @owner = schema_...

Advantages of SP

- Security
- Code reuse, modular programming
- Performance
- Reduce traffic

Example: Reduced traffic



- Each time Client wants to execute the statement "SELECT * FROM customer_details", it must send this statement to the Server.
- Of course, we see that, the length of that statement is longer than the length of "Show_Customers"

Control of flow – SQL Programming

- Still somewhat limited compared to other languages
  - WHILE
  - IF ELSE
  - BEGIN END block
  - CASE
  - WAITFOR
  - CONTINUE/BREAK

Variables

- Declare a variable:

```
DECLARE @limit money
DECLARE @min_range int, @hi_range int
```

- Assign a value into a variable:

```
SET @min_range = 0, @hi_range = 100
SET @limit = $10
```

- Assign a value into a variable in SQL statement:

```
SELECT @price = price FROM titles
WHERE  title_id = 'PC2091'
```

Control of Flow

BEGIN…END

IF…ELSE

CASE … WHEN

 RETURN [n]

WHILE

PRINT

CASE … WHEN

```
CASE input_expression
   WHEN  when_expression  THEN result_expression
   [WHEN when_expression  THEN result_expression…n]
   [ELSE else_result_expression ]
END
```

Example:

```
SELECT CASE payterms
   WHEN 'Net 30' THEN 'Payable 30 days  after invoice'
   WHEN 'Net 60' THEN 'Payable 60 days after invoice'
   WHEN 'On invoice' THEN 'Payable upon  receipt of invoice'
   ELSE 'None'
END as Payment_Terms FROM sales ORDER BY payterms
```

RETURN [n]

- Exits unconditionally of Trigger, Procedure or Function and return a value (if any).

```
USE AdventureWorks2012;
GO
CREATE PROCEDURE checkstate @param varchar(11)
AS
IF (SELECT StateProvince FROM Person.vAdditionalContactInfo WHERE
       ContactID = @param) = 'WA'
    RETURN 1
ELSE
    RETURN 2;
```

PRINT

- Display message in SQL Query Analyze (Console)

```
USE AdventureWorks2008R2;
GO
IF (SELECT SUM(i.Quantity)
    FROM Production.ProductInventory i
    JOIN Production.Product p
    ON i.ProductID = p.ProductID
    WHERE Name = 'Hex Nut 17'
    ) < 1100
    PRINT N'There are less than 1100 units of Hex Nut 17 in stock.'
GO
```

TRY CATCH structure

```
CREATE PROCEDURE dbo.uspTryCatchTest
AS
BEGIN TRY
    SELECT 1/0
END TRY
BEGIN CATCH
    SELECT ERROR_NUMBER() AS ErrorNumber
     ,ERROR_SEVERITY() AS ErrorSeverity
     ,ERROR_STATE() AS ErrorState
     ,ERROR_PROCEDURE() AS ErrorProcedure
     ,ERROR_LINE() AS ErrorLine
     ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH
```

WHILE

- Repeats a statement (or block) while a specific condition is true

```
WHILE Boolean_expression
SQL_statement | block_of_statements
[BREAK] SQL_statement | block_of_statements [CONTINUE]
```

- Example:

```
WHILE (SELECT AVG(royalty) FROM roysched) < 25
BEGIN
    UPDATE roysched SET royalty = royalty * 1.05
    IF (SELECT MAX(royalty)FROM roysched) > 27  BREAK
    ELSE   CONTINUE
END
SELECT MAX(royalty) AS "MAX royalty"
FROM roysched
```

Cursor

```
DECLARE myCursor CURSOR
FOR SELECT TOP(10) ContactName FROM Customers
DECLARE @RowNo int,@ContactName nvarchar(30)
SET @RowNo=1
OPEN myCursor
FETCH NEXT FROM myCursor INTO @ContactName
PRINT  LEFT(CAST(@rowNo as varchar) + '      ',6)+'   '+
@ContactName
SET @RowNo=@RowNo+1
SET @ContactName=''
WHILE @@FETCH_STATUS=0
  BEGIN
        FETCH NEXT FROM myCursor INTO @ContactName
        PRINT + LEFT(CAST(@rowNo as varchar) + '      ',6)+'   '+
@ContactName
        SET @RowNo=@RowNo+1
        SET @ContactName=''
  END
CLOSE myCursor
DEALLOCATE myCursor
```

Basic Syntax

```
DECLARE demo_cursor CURSOR
READ_ONLY
FOR SELECT ProductID FROM Northwind..Products ORDER BY ProductID

DECLARE @ProductName nvarchar(50)

OPEN demo_cursor

FETCH NEXT FROM demo_cursor INTO @ProductName
WHILE (@@fetch_status <> -1)
BEGIN
    IF (@@fetch_status <> -2)
    BEGIN
      DECLARE @message varchar(100)
      SELECT @message = 'The product is: ' + @ProductName
      PRINT @message
    END
    FETCH NEXT FROM demo_cursor INTO @ProductName
END

CLOSE demo_cursor
DEALLOCATE demo_cursor
GO
```

USER DEFINED FUNCTIONS

Basic Syntax

```
CREATE FUNCTION dbo.fn_total(@param1
datatype)
RETURNS datatype2
AS
BEGIN
   DECLARE @localvar datatype2
   --populate @localvar here
   RETURN @localvar
END
```

Returned data types

- Scalar
  - Returns a single value
  - Evaluated for every row if used in select line
- Inline table values
  - Returns a variable of type table
  - Single select statement defines the table
- Multi-statement table valued

Example: Return a scalar value

```
CREATE FUNCTION FetchTotalOrders(@p_CustomerID nvarchar(10))
RETURNS INT
BEGIN
RETURN (SELECT COUNT(OrderID) FROM Orders
WHERE CustomerID = @p_CustomerID)
END

GO

SELECT dbo.FetchTotalOrders('ANTON')
```

Example: Return inline table value

```sql
CREATE FUNCTION CustomerPurchasedDetails (@p_CustomerID nvarchar(10))
RETURNS TABLE AS
RETURN (SELECT P.ProductName, P.UnitPrice
FROM Customers C INNER JOIN Orders O ON C.CustomerID = O.CustomerID
INNER JOIN [Order Details] OD ON O.OrderID = OD.OrderID
INNER JOIN Products P ON OD.ProductID = P.ProductID
WHERE C.CustomerID = @p_CustomerID)

GO

SELECT * FROM dbo.CustomerPurchasedDetails('ANTON')
```

Example: Multi-statement table valued

```
CREATE FUNCTION GetLastShipped(@CustomerID nchar(5))
RETURNS @CustomerOrder TABLE
            (SaleOrderID INT, CustomerID nchar(5), OrderDate  DATETIME,
            OrderQty        INT)
AS
BEGIN
    DECLARE @MaxDate DATETIME
    SELECT @MaxDate = MAX(OrderDate)
    FROM Orders
    WHERE CustomerID = @CustomerID
    INSERT @CustomerOrder
    SELECT a.OrderID, a.CustomerID, a.OrderDate, b.Quantity
    FROM Orders a INNER JOIN [Order Details] b
        ON a.OrderID = b.OrderID
    WHERE a.OrderDate = @MaxDate
        AND a.CustomerID = @CustomerID
    RETURN
END
GO

SELECT * FROM dbo.GetLastShipped('ALFKI')
```

Uses of Functions

- Can greatly simplify the select line
- Modular programming
- Can improve reliability of data by reducing the number of joins and encapsulating queries
- Reduce network traffic
- Faster execution

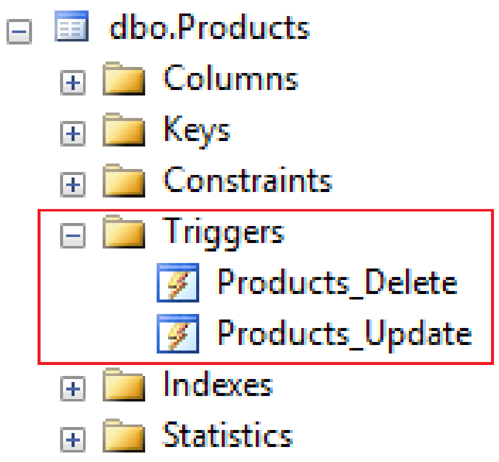
Function vs Stored Procedure

	Function	Stored procedure
Returned value	Required	Optional
Parameters	Only input	Input, output
Supported statements	Only SELECT, Not DML	SELECT, UPDATE, DELETE, INSERT…
Transactions	Not support	Support
Temporary table	Not support	Support
Call Function or SP?	Can't call SP, only Functions	Can call SPs and Functions

TRIGGERS

Trigger overview

- Definition: A trigger is a special SP executed automatically as part of a data modification (INSERT, UPDATE, or DELETE)
- Associated with a table
- Invoked automatically
- Cannot be called explicitly



Syntax

```
CREATE TRIGGER trigger_name
ON  <tablename>
<{FOR | AFTER}>
{[DELETE] [,] [INSERT] [,] [UPDATE]}
AS
SQL_Statement [...n]
```

Simplied Syntax

```
CREATE TRIGGER trg_one
ON tablename
FOR INSERT, UPDATE, DELETE
AS
BEGIN
    SELECT * FROM Inserted
    SELECT * FROM Deleted
END
```

Temporary table holding new records

Temporary table holding old, deleted, updated records

Uses of Triggers

- Maintenance of duplicate and derived data
- Ensure integrity
  - Complex column constraints
  - Cascading referential integrity
  - Inter-database referential integrity
- Complex defaults
- Logging/Auditing
- Maintaining de-normalized data

Trigger example

```
Use Northwind
GO
CREATE TRIGGER Cust_Delete_Only1 ON Customers
FOR DELETE
AS
IF (SELECT COUNT(*) FROM Deleted) > 1
BEGIN
   RAISERROR('You are not allowed to delete more than one customer at a
time.', 16, 1)
   ROLLBACK TRANSACTION
END
```

```
DELETE FROM Customers
WHERE CustomerID NOT IN (SELECT CustomerID FROM Orders)
```

Define a trigger preventing users from updating more than 2 records at a time?

INSERT-Trigger example

```
USE Northwind GO
CREATE TRIGGER Order_Insert
ON [Order Details]
FOR INSERT
AS
UPDATE P SET UnitsInStock = (P.UnitsInStock – I.Quantity)
FROM Products AS P INNER JOIN Inserted AS I ON P.ProductID = I.ProductID
```

Order Details

OrderID	*ProductID*	*UnitPrice*	*Quantity*	*Discount*
10522	10	31.00	7	0.2
10523	41	9.65	9	0.15
10524	7	30.00	24	0.0
10523	2	19.00	5	0.2

ProductID	*UnitsInStock*	…	…
1	15		
2	5		
3	65		
4	20		

```
INSERT [Order Details] VALUES
(10525, 2, 19.00, 5, 0.2)
```

inserted

10523	2	19.00	5	0.2

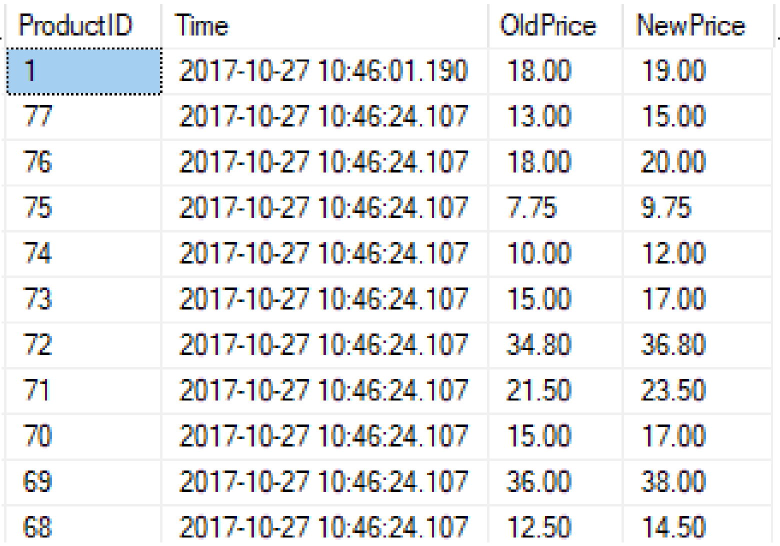
UPDATE-Trigger example

```
CREATE TABLE PriceTracking
(ProductID int, Time DateTime, OldPrice money, NewPrice money)

GO

CREATE TRIGGER Products_Update
ON Products FOR UPDATE
AS
INSERT INTO PriceTracking (ProductID, Time, OldPrice, NewPrice)
SELECT I.ProductID, GETDATE(), D.UnitPrice, I.UnitPrice
FROM inserted AS I INNER JOIN Deleted AS D ON I.ProductID = D.ProductID AND
I.UnitPrice <> D.UnitPrice
```

```
UPDATE Products
SET UnitPrice = UnitPrice + 2
```



ProductID	Time	OldPrice	NewPrice
1	2017-10-27 10:46:01.190	18.00	19.00
77	2017-10-27 10:46:24.107	13.00	15.00
76	2017-10-27 10:46:24.107	18.00	20.00
75	2017-10-27 10:46:24.107	7.75	9.75
74	2017-10-27 10:46:24.107	10.00	12.00
73	2017-10-27 10:46:24.107	15.00	17.00
72	2017-10-27 10:46:24.107	34.80	36.80
71	2017-10-27 10:46:24.107	21.50	23.50
70	2017-10-27 10:46:24.107	15.00	17.00
69	2017-10-27 10:46:24.107	36.00	38.00
68	2017-10-27 10:46:24.107	12.50	14.50

Enforcing integrity with Trigger

```
CREATE TRIGGER Products_Delete
ON Products FOR DELETE AS
IF (SELECT COUNT(*)
    FROM [Order Details] OD
    WHERE OD.ProductID = (SELECT ProductID FROM deleted)
    ) > 0
BEGIN
    PRINT 'Violate Foreign key reference. Rollback!!!'
    ROLLBACK TRAN
END
```

```
DELETE Products
WHERE ProductID = 11
```

Performance Considerations

- Triggers work quickly because the Inserted and Deleted tables are in cache
- Execution time is determined by:
  - Number of tables that are referenced
  - Number of rows that are affected
- Actions contained in triggers implicitly are part of a transaction