Seminar 4. Functions. Views. System tables

Transact-SQL User Defined Functions

User defined functions allow developers to define their own functions to be used in SQL queries. There are three types of user defined functions in MS SQL Server:

- Scalar;
- Inline Table-Valued;
- Multi-Statement Table-Valued.

Scalar functions
Scalar functions return a single value (it does not matter what type it is, as long as it is only a single value).
The biggest drawback of scalar functions is that they may not be automatically inlined. For a scalar function that operates on multiple rows, SQL Server will execute the function once for every row in the result set and this can have a huge performance impact.

```
CREATE FUNCTION ufGetCourseNumber (@credits INT)
RETURNS INT AS
BEGIN
      DECLARE @Return INT
      SET @Return = 0

      SELECT @Return = COUNT(*)
      FROM Courses
      WHERE Credits = @credits

      RETURN @Return
END
-------------------------------------------------
PRINT dbo.ufGetCourseNumber(6)
```

Inline Table-Valued Functions
A table-valued user defined function returns a table instead of a single value. It can be used anywhere a table can be used – typically in the FROM clause of a query.

```
CREATE FUNCTION ufGetCourseNames (@credits INT)
RETURNS TABLE
AS
      RETURN
            SELECT CName
            FROM Courses
            WHERE Credits = @credits
----------------------------------------
SELECT * FROM dbo.ufGetCourseNames(6)
```

Multi-statement Table-Valued Functions
The difference between a multi-statement table-valued function and an inline table-valued function is that the first one contains more than one statement in the function body.

```
CREATE FUNCTION GetAuthorsByState ( @state CHAR(2) )
RETURNS @AuthorsByState table (au_id VARCHAR(11),
            au_fname VARCHAR(20))
AS
BEGIN

      INSERT INTO @AuthorsByState
      SELECT  au_id, au_fname FROM Authors
      WHERE state = @state

      IF @@ROWCOUNT = 0
      BEGIN
            INSERT INTO @AuthorsByState
            VALUES ('','No Authors Found')
      END

RETURN
END
GO
--------------------------------------------
SELECT * FROM dbo.GetAuthorsByState('CA')
```

Transact-SQL Views

A view creates a *virtual table* that represents the data in one or more tables in an alternative way.
A view contains rows and columns, just like a real table and can reference a maximum of 1,024 columns.
The SQL CREATE VIEW statement has the following syntax:

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name(s)
WHERE condition
```

System Tables

The information about all the objects (tables, fields, indexes, stored procedures, user defined functions, views, etc) created in a database is stored in special tables known as system tables.
System tables should not be altered directly by any user, they being maintained exclusively by the server.
Examples of system tables:
sys.objects - contains one row for each object (constraint, stored procedure, view, etc) created within a database.
sys.columns - contains one row for every column of an object that has columns, e.g., tables, views.
sys.sql_modules - returns a row for each object that is an SQL language-defined module.

Triggers

= special types of stored procedures that automatically execute when a DML or DDL statement is executed

- cannot be executed directly
- DML statements: INSERT, UPDATE, DELETE
- DDL statements: CREATE_DATABASE, DROP_LOGIN, UPDATE_STATISTICS, DROP_TRIGGER, ALTER_TABLE

```
CREATE TRIGGER <trigger_name>
ON { table | view }
[ WITH <dml_trigger_option> [ ,...n ] ]
{ FOR | AFTER | INSTEAD OF }
{ [INSERT] [,] [UPDATE] [,] [DELETE] }
[ WITH APPEND ] [ NOT FOR REPLICATION ] AS
{ sql_statement [;] [ ,...n ] | EXTERNAL NAME <method specifier [;] > }
```

Moment of execution:

- FOR
- AFTER
- INSTEAD OF

If multiple triggers are defined for the same action they are executed in random order
When a trigger is executed 2 special tables named *inserted* and *deleted* are available

Example:

```
CREATE TRIGGER [dbo].[On_Product_Insert]
ON [dbo].[Products]
FOR INSERT
AS
BEGIN
  SET NOCOUNT ON;
  INSERT INTO LogBuys (Name, Date, Quantity)
  SELECT Name, GETDATE(), Quantity
  FROM inserted
END

ALTER TRIGGER [dbo].[On_Product_Update]
ON [dbo].[Products]
FOR UPDATE
AS
BEGIN
  SET NOCOUNT ON;
  INSERT INTO LogSells (Name, Date, Quantity)
  SELECT d.Name, GETDATE(), d.Quantity-i.Quantity
  FROM deleted d INNER JOIN inserted i ON d.ID=i.ID
  WHERE i.Quantity < d.Quantity
  INSERT INTO LogBuys (Name, Date, Quantity)
  SELECT i.Name, GETDATE(), i.Quantity-d.Quantity
 FROM deleted d INNER JOIN inserted i ON d.ID = i.ID
 WHERE  i.Quantity > d.Quantity
END
```

SET NOCOUNT ON/OFF

- ON - the count is not returned.
- OFF - the count is returned
- @@ROWCOUNT is always updated.

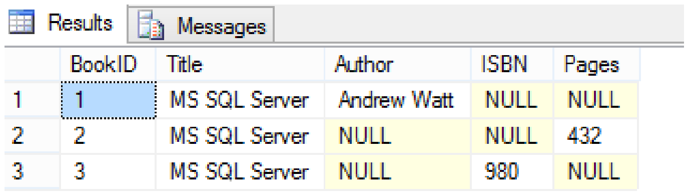
MERGE statement

MERGE – gives the ability to compare rows in a source and a destination table
INSERT, UPDATE or DELETE commands could be performed based on the result of this comparison (i.e., can perform I/U/D on a target table based on the results of a join with a source table)

MERGE general syntax:

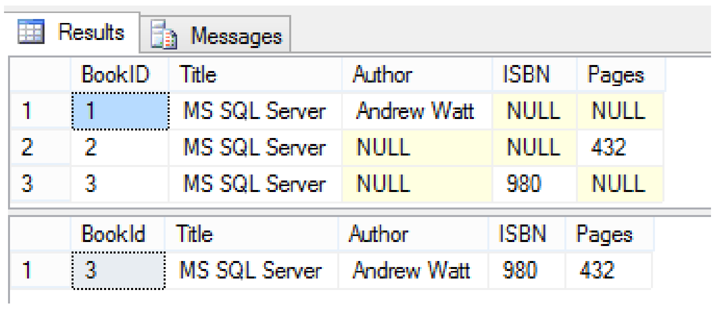
```
Merge Table definition as Target
Using ( Table Source ) as Source
ON (
Search Terms
)
WHEN MATCHED THEN
  UPDATE SET
    or
  DELETE
WHEN NOT MATCHED BY TARGET/SOURCE THEN
  INSERT / (UPDATE | DELETE)
```

MERGE sample



	BookID	Title	Author	ISBN	Pages
1	1	MS SQL Server	Andrew Watt	NULL	NULL
2	2	MS SQL Server	NULL	NULL	432
3	3	MS SQL Server	NULL	980	NULL

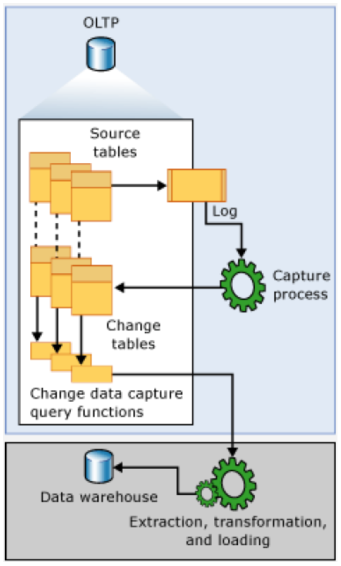
```
MERGE Books
USING
  (SELECT MAX(BookId) BookId, Title, MAX(Author) Author, MAX(ISBN) ISBN,
MAX(Pages) Pages
  FROM Books
  GROUP BY Title
  ) MergeData
ON Books.BookId = MergeData.BookId
WHEN MATCHED THEN
UPDATE SET Books.Title = MergeData.Title,
  Books.Author = MergeData.Author,
  Books.ISBN = MergeData.ISBN,
  Books.Pages = MergeData.Pages
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```



	BookID	Title	Author	ISBN	Pages
1	1	MS SQL Server	Andrew Watt	NULL	NULL
2	2	MS SQL Server	NULL	NULL	432
3	3	MS SQL Server	NULL	980	NULL

	BookId	Title	Author	ISBN	Pages
1	3	MS SQL Server	Andrew Watt	980	432

Change Data Capture


OLTP
Source tables
Log
Capture process
Change tables
Change data capture query functions
Data warehouse
Extraction, transformation, and loading

- provides information about DML changes on a table and a database
- introduced in SQL Server 2008
- *sys.sp_cdc_enable_db* – explicitly enables CDC for the db
- *sys.sp_cdc_enable_table* - tracked tables

Enables data archiving and capturing without additional programming (like triggers)
Tracks changes in user created tables
Captured data is stored in relational tables => can easily be queried using SQL
Apply CDC features on a table => a mirror table is created, contains the same columns as the tracked table + metadata describing the changes