Funcții definite de utilizator. View-uri. Triggere

Seminar 4

Funcții definite de către utilizator

- Microsoft SQL Server oferă posibilitatea de a crea funcții care pot fi mai apoi folosite în interogări
- Funcțiile definite de utilizator pot avea parametri de intrare și returnează o valoare
- În Microsoft SQL Server sunt disponibile trei tipuri de funcții definite de utilizator:
    - Funcții **scalare**
    - Funcții **inline table-valued**
    - Funcții **multi-statement table-valued**

Funcții definite de către utilizator

- Funcțiile scalare returnează o singură valoare
- Sintaxa pentru crearea unei funcții scalare:

```
CREATE FUNCTION scalar_function_name(@param1
datatype1, @param2 datatype2)

RETURNS datatype AS

BEGIN

 -- SQL Statements

RETURN value;

END;
```

Funcții definite de către utilizator

- Sintaxa pentru modificarea unei funcții scalare:

```
ALTER FUNCTION scalar_function_name(@param1 datatype1,
@param2 datatype2)
RETURNS datatype AS
BEGIN
-- SQL Statements
RETURN value;
END;
```

- Sintaxa pentru ștergerea unei funcții scalare:

```
DROP FUNCTION scalar_function_name;
```

Funcții definite de către utilizator

- Funcție care returnează numărul de cursuri care au un anumit număr de credite:

```
CREATE FUNCTION ufNrCrediteCursuri(@nrcredite INT)
RETURNS INT AS
BEGIN
DECLARE @nrcursuri INT=0;
SELECT @nrcursuri=COUNT(*) FROM Cursuri WHERE
nrcredite=@nrcredite;
RETURN @nrcursuri;
END;
----------------------------------------------
PRINT dbo.ufNrCrediteCursuri(6);
```

Funcții definite de către utilizator

- Funcțiile definite de utilizator de **tip inline table-valued** returnează un tabel în locul unei singure valori
- Pot fi folosite oriunde poate fi folosit un tabel, de obicei în clauza FROM a unei interogări
- O funcție definită de utilizator de tip **inline table-valued** conține o singură instrucțiune SQL
- O funcție definită de utilizator de tipul **multi-statement table-valued** returnează un tabel și conține mai multe instrucțiuni SQL, spre deosebire de o funcție **inline table-valued** care conține o singură instrucțiune SQL

Funcții definite de către utilizator

- Crearea unei funcții care primește ca parametru numărul de credite și returnează numele cursurilor cu acel număr de credite:

```
CREATE FUNCTION ufNumeCursuri(@nrcredite INT)
RETURNS TABLE
AS
RETURN SELECT nume FROM Cursuri WHERE
nrcredite=@nrcredite;
----------------------------------------
SELECT * FROM dbo.ufNumeCursuri(6);
```

Funcții definite de către utilizator

```
CREATE FUNCTION ufPersoaneLocalitate(@localitate NVARCHAR(30))
RETURNS @PersoaneLocalitate TABLE (nume NVARCHAR(40), prenume
NVARCHAR(40)) AS
BEGIN
INSERT INTO @PersoaneLocalitate (nume, prenume) SELECT nume,
prenume FROM Persoane WHERE localitate=@localitate;
IF(@@ROWCOUNT=0)
      INSERT INTO @PersoaneLocalitate (nume, prenume) VALUES
      (N'Nicio persoană din această localitate',N'');
RETURN;
END;
```

Funcții definite de către utilizator

- Funcția **multi-statement table-valued** definită anterior primește ca parametru o valoare ce reprezintă localitatea și returnează un tabel cu numele și prenumele persoanelor care au localitatea egală cu valoarea transmisă ca parametru
- În cazul în care nu este returnată nicio înregistrare care să corespundă localității transmise ca parametru, în variabila de tip tabel se va insera o înregistrare care conține un mesaj corespunzător
- Exemplu de apel al funcției:

```
SELECT * FROM dbo.ufPersoaneLocalitate(N'Sibiu');
```

View

- Un **view** este un tabel virtual bazat pe result set-ul unei interogări
- Conține înregistrări și coloane ca un tabel real
- Un **view** nu stochează date, stochează definiția unei interogări
- Cu ajutorul unui **view** putem prezenta date din mai multe tabele ca și cum ar veni din același tabel
- De fiecare dată când un **view** este interogat, motorul bazei de date va recrea datele folosind **instrucțiunea SELECT** specificată la crearea **view-ului**, astfel că un **view** va prezenta întotdeauna **date actualizate**
- **Numele coloanelor** dintr-un **view** trebuie să fie **unice** (în cazul în care avem două coloane cu același nume provenind din tabele diferite, putem folosi un **alias** pentru una dintre ele)

View

- Sintaxa pentru crearea unui view:

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

- Sintaxa pentru modificarea unui view:

```
ALTER VIEW view_name AS
SELECT column_name(s) FROM table_name;
```

- Sintaxa pentru ștergerea unui view:

```
DROP VIEW view_name;
```

View

– Crearea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
CREATE VIEW vw_Produse AS
SELECT P.nume, P.preț,
C.nume AS categorie
FROM Produse AS P
INNER JOIN Categorii AS C
ON P.id_cat=C.id_cat;
```

View

– Modificarea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
ALTER VIEW vw_Produse AS
SELECT P.nume, P.preț, P.cantitate,
C.nume AS categorie
FROM Produse AS P
INNER JOIN Categorii AS C
ON P.id_cat=C.id_cat;
```

View

- Interogarea unui view care conține date din două tabele, *Categorii* și *Produse*:

```
SELECT nume, preț, cantitate, categorie
FROM vw_Produse;
SAU
SELECT * FROM vw_Produse;
```

- Exemplu de ștergere a unui view:

```
DROP VIEW vw_Produse;
```

View

- Nu se poate folosi clauza ORDER BY în definiția unui view (decât dacă se specifică în definiția view-ului clauza TOP, OFFSET sau FOR XML)
- Dacă dorim să ordonăm înregistrările din result set, putem folosi clauza ORDER BY atunci când interogăm view-ul
- Pentru a afișa definiția unui view, putem folosi funcția **OBJECT_DEFINITION** sau procedura stocată **sp_helptext**:

```
PRINT OBJECT_DEFINITION (OBJECT_ID('schema_name.view_name');

EXEC sp_helptext 'schema_name.view_name';
```

View

- Se pot insera date într-un view doar dacă inserarea afectează un singur base table (în cazul în care view-ul conține date din mai multe tabele)
- Se pot actualiza date într-un view doar dacă actualizarea afectează un singur base table (în cazul în care view-ul conține date din mai multe tabele)
- Se pot șterge date dintr-un view doar dacă view-ul conține date dintr-un singur tabel
- Operațiunile de inserare într-un view sunt posibile doar dacă view-ul expune toate coloanele care nu permit valori NULL
- Numărul maxim de coloane pe care le poate avea un view este 1024

Tabele sistem

- Tabelele sistem sunt niște tabele speciale care conțin informații despre toate obiectele create într-o bază de date, cum ar fi:
  - Tabele
  - Coloane
  - Proceduri stocate
  - Trigger-e
  - View-uri
  - Funcții definite de către utilizator
  - Indecși

Tabele sistem

- Tabelele sistem sunt gestionate de către server (nu se recomandă modificarea lor direct de către utilizator)
- Exemple:

  sys.objects – conține câte o înregistrare pentru fiecare obiect creat în baza de date, cum ar fi: procedură stocată, trigger, tabel, constrângere

  sys.columns – conține câte o înregistrare pentru fiecare coloană a unui obiect care are coloane, cum ar fi: tabel, view, funcție definită de către utilizator care returnează un tabel

Trigger

- Trigger-ul este un **tip special de procedură stocată** care se execută automat atunci când un anumit eveniment DML sau DDL are loc în baza de date
- Nu se poate executa în mod direct
- Evenimente DML:
  - INSERT
  - UPDATE
  - DELETE
- Evenimente DDL:
  - CREATE
  - ALTER
  - DROP
- Fiecare trigger (DML) aparține unui singur tabel

Trigger

- Sintaxa:

```
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 [ ; ] > }
```

Trigger

- Momentul execuției unui trigger
  - FOR, AFTER (se pot defini mai multe trigger-e de acest tip) – trigger-ul se execută după ce s-a executat evenimentul declanșator
  - INSTEAD OF – trigger-ul se execută în locul evenimentului declanșator
- Dacă se definesc mai multe trigger-e pentru aceeași acțiune (eveniment), ele se execută în ordine aleatorie
- Când se execută un trigger, sunt disponibile două tabele speciale, numite **inserted** și **deleted**

Trigger - Exemplu

```
CREATE TRIGGER [dbo].[La_introducere_produs]
ON [dbo].[Produse]
FOR INSERT
AS
BEGIN
SET NOCOUNT ON;
INSERT INTO Arhivă_Cumpărare (nume, dată, cantitate)
SELECT nume, GETDATE(), cantitate FROM inserted;
END;
```

Trigger - Exemplu

```
CREATE TRIGGER [dbo].[La_ștergere_produs]
ON [dbo].[Produse]
FOR DELETE
AS
BEGIN
SET NOCOUNT ON;
INSERT INTO Arhivă_Vânzare (nume, dată, cantitate)
SELECT nume, GETDATE(), cantitate FROM deleted;
END;
```

Trigger - Exemplu

```
CREATE TRIGGER [dbo].[La_actualizare_produs]

ON [dbo].[Produse]

FOR UPDATE AS

BEGIN

SET NOCOUNT ON;

INSERT INTO Arhivă_Vânzare (nume, dată, cantitate) SELECT d.nume,
GETDATE(), d.cantitate-i.cantitate FROM deleted d INNER JOIN
inserted i ON d.cod_p=i.cod_p WHERE i.cantitate<d.cantitate;

INSERT INTO Arhivă_Cumpărare (nume, dată, cantitate) SELECT i.nume,
GETDATE(), i.cantitate-d.cantitate from deleted d INNER JOIN
inserted i on d.cod_p=i.cod_p WHERE i.cantitate>d.cantitate;

END;
```

SET NOCOUNT

- SET NOCOUNT ON – oprește returnarea mesajului cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată
- SET NOCOUNT OFF – mesajul cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată va fi returnat ca parte din result set
- Variabila globală @@ROWCOUNT va fi modificată întotdeauna
- Dacă NOCOUNT este setat pe ON, performanța procedurilor stocate care conțin bucle Transact-SQL sau instrucțiuni care nu returnează multe date se va îmbunătăți (traficul pe rețea este redus)

Change Data Capture

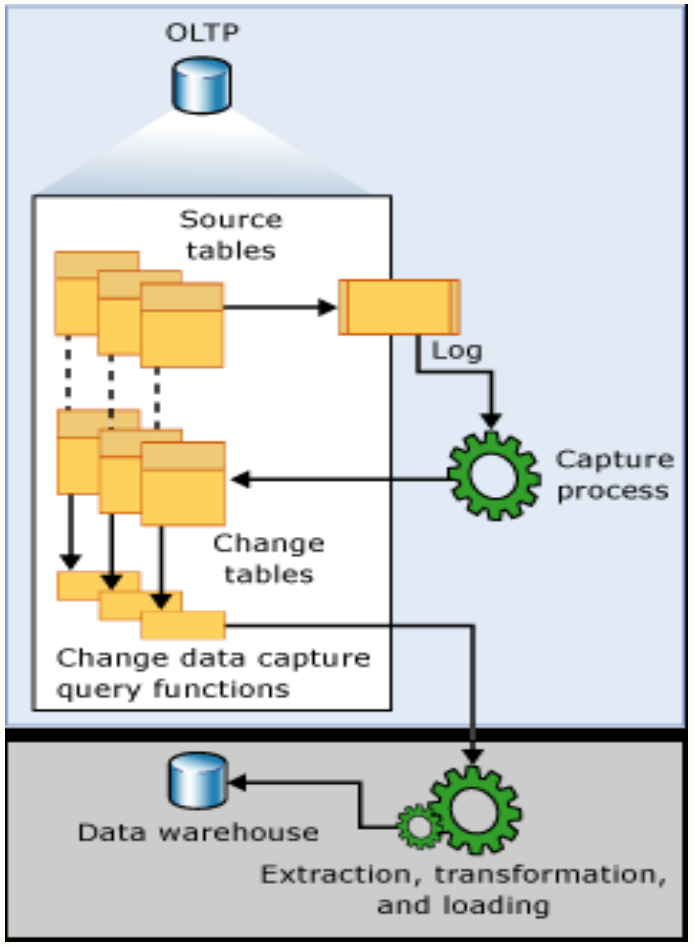
- **Change Data Capture** înregistrează comenzile INSERT, UPDATE și DELETE care sunt aplicate unui tabel
- Astfel, detaliile modificărilor DML asupra unui tabel devin disponibile într-un format relațional ușor de consumat
- Meta datele și informațiile despre coloane necesare pentru a aplica modificările unui mediu țintă sunt captate pentru înregistrările modificate și sunt stocate în **change tables** (aceste tabele reflectă structura tabelelor sursă)
- Pentru a putea monitoriza modificările DML care au loc într-un tabel, **Change Data Capture** trebuie activată mai întâi la nivelul bazei de date cu ajutorul procedurii stocate **sys.sp_cdc_enable_db**
- Monitorizarea unui tabel se poate activa cu ajutorul procedurii stocate **sys.sp_cdc_enable_table**

Change Data Capture

- Pentru a determina dacă **Change Data Capture** este activată pentru o anumită bază de date, se va executa interogarea de mai jos:

```
SELECT is_cdc_enabled FROM
sys.databases WHERE name=
'database_name';
```

- În momentul **activării Change Data Capture** se creează schema *cdc*, user-ul *cdc*, tabelele cu meta date și alte obiecte sistem



Instrucțiunea MERGE

- Instrucțiunea **MERGE** oferă posibilitatea de a compara înregistrări dintr-un tabel sursă cu înregistrări dintr-un tabel destinație
- Comenzile INSERT, UPDATE și DELETE pot fi executate pe baza rezultatului acestei comparații
- Tabelul *Filme*:

Cod_film	Titlu	An	Durata
1	IT	2017	NULL
2	IT	NULL	NULL
3	IT	NULL	135

Instrucțiunea MERGE

- Sintaxa:

```
MERGE Table_definition AS Target
USING (Source_table_name) as Source
ON (Search_Terms)
WHEN MATCHED THEN
UPDATE SET or DELETE
WHEN NOT MATCHED [BY TARGET] THEN INSERT
WHEN NOT MATCHED BY SOURCE THEN UPDATE SET or DELETE;
```

Instrucțiunea MERGE

- Exemplu:

```
MERGE Filme
USING (SELECT MAX(Cod_film) Cod_film, Titlu, MAX(An)
An, MAX(Durata) Durata FROM Filme GROUP BY Titlu)
MergeFilme ON Filme.Cod_film=MergeFilme.Cod_film
WHEN MATCHED THEN
UPDATE SET Filme.Titlu=MergeFilme.Titlu, Filme.An=
MergeFilme.An, Filme.Durata=MergeFilme.Durata
WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

Instrucțiunea MERGE

– Rezultat:

Cod_film	Titlu	An	Durata
3	IT	2017	135