Funcții definite de utilizator. View. Trigger. Cursoare

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 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, funcție definită de utilizator care returnează un tabel, view

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;
```

Clauza OUTPUT

- Cu ajutorul clauzei **OUTPUT** avem acces la înregistrările modificate, șterse sau adăugate
- În exemplul de mai jos se actualizează numele persoanei care are *cod_p*=5 din tabelul *Persoane* și se stochează în tabelul *ModificăriNumePersoane* valoarea din coloana *cod_p*, valoarea veche a numelui (*deleted.nume*), valoarea nouă a numelui (*inserted.nume*), data curentă (GETDATE()) și numele login-ului care a realizat modificarea (SUSER_SNAME()):

```
UPDATE Persoane SET nume='Pop' OUTPUT inserted.cod_p,
deleted.nume, inserted.nume, GETDATE(), SUSER_SNAME()
INTO ModificăriNumePersoane (cod_p, nume_vechi,
nume_nou, data_modificării, nume_login) WHERE cod_p=5;
```

Cursoare

- Sunt anumite situații în care procesarea unui result set este mai eficientă dacă se procesează pe rând fiecare înregistrare din result set
- Deschiderea unui cursor pe un result set permite procesarea result set-ului înregistrare cu înregistrare (se procesează o singură înregistrare la un moment dat)
- Cursoarele extind procesarea rezultatelor prin faptul că:

  - permit poziționarea la înregistrări specifice dintr-un result set

  - returnează o înregistrare sau un grup de înregistrări aflate la poziția curentă din result set

Cursoare

- suportă modificarea înregistrărilor aflate în poziția curentă în result set

- suportă diferite niveluri de vizibilitate a modificărilor făcute de către alți utilizatori asupra datelor din baza de date care fac parte din result set

- permit instrucțiunilor Transact-SQL din script-uri, proceduri stocate și trigger-e accesul la datele dintr-un result set

Cursoare

- Cursoarele Transact-SQL necesită anumite instrucțiuni pentru declarare, populare și extragere de date:

  - se folosește o instrucțiune **DECLARE CURSOR** pentru a declara cursorul și se specifică o instrucțiune **SELECT** care va produce result set-ul cursorului

  - se folosește o instrucțiune **OPEN** pentru a popula cursorul, care execută instrucțiunea **SELECT** încorporată în instrucțiunea **DECLARE CURSOR**

  - se folosește o instrucțiune **FETCH** pentru a extrage înregistrări individual din result set (de obicei **FETCH** se execută de multe ori, cel puțin o dată pentru fiecare înregistrare din result set)

Cursoare

- dacă este cazul, se folosește o instrucțiune **UPDATE** sau **DELETE** pentru a modifica înregistrarea (acest pas este opțional)

- se folosește o instrucțiune **CLOSE** pentru a închide cursorul și a elibera unele resurse (cum ar fi result set-ul cursorului și lock-urile de pe înregistrarea curentă)

- cursorul este încă declarat, deci poate fi deschis din nou folosind o instrucțiune **OPEN**

- se folosește o instrucțiune **DEALLOCATE** pentru a elimina referința cursorului din sesiunea curentă iar acest proces eliberează toate resursele alocate cursorului, inclusiv numele său (după acest pas, pentru a reconstrui cursorul este nevoie ca acesta să fie declarat din nou)

- cursoarele aflate în interiorul procedurilor stocate nu necesită închidere și eliminare, aceste instrucțiuni se execută automat când procedura stocată își încheie execuția

Cursoare

- Cursoarele Transact-SQL sunt extrem de eficiente atunci când sunt încorporate în proceduri stocate și trigger-e deoarece totul este compilat într-un singur plan de execuție pe server, deci nu există trafic pe rețea asociat cu returnarea înregistrărilor
- Operațiunea de a returna o înregistrare dintr-un cursor se numește **fetch**, iar în cazul cursoarelor Transact-SQL se folosește instrucțiunea **FETCH** pentru a returna înregistrări din result set-ul unui cursor
- Instrucțiunea **FETCH** suportă un număr de opțiuni care permit returnarea unor înregistrări specifice:

  - **FETCH FIRST** – returnează prima înregistrare din cursor

Cursoare

- **FETCH NEXT** – returnează înregistrarea care urmează după ultima înregistrare returnată

- **FETCH PRIOR** – returnează înregistrarea care se află înaintea ultimei înregistrări returnate

- **FETCH LAST** – returnează ultima înregistrare din cursor

- **FETCH ABSOLUTE n** – returnează a n-a înregistrare de la începutul cursorului dacă n este un număr pozitiv, iar dacă n este un număr negativ returnează înregistrarea care se află cu n înregistrări înaintea sfârșitului cursorului (dacă n este 0, nicio înregistrare nu este returnată)

Cursoare

- **FETCH RELATIVE n** – returnează a n-a înregistrare după ultima înregistrare returnată dacă n este pozitiv, iar dacă n este negativ returnează înregistrarea care se află înainte cu n înregistrări față de ultima înregistrare returnată (dacă n este 0, ultima înregistrare returnată va fi returnată din nou)

Comportamentul unui cursor poate fi specificat în două moduri:

– prin specificarea comportamentului cursoarelor folosind cuvintele cheie **SCROLL** și **INSENSITIVE** în instrucțiunea **DECLARE CURSOR** (SQL-92 standard)

– prin specificarea comportamentului unui cursor cu ajutorul tipurilor de cursoare

Cursoare

- de obicei API-urile pentru baze de date definesc comportamentul cursoarelor împărțindu-le în patru tipuri de cursoare: forward-only, static (uneori denumit snapshot sau insensitive), keyset-driven și dynamic

– Declararea unui cursor – sintaxa ISO:

```
DECLARE cursor_name [ INSENSITIVE ] [ SCROLL ] CURSOR
FOR select_statement
[ FOR { READ ONLY | UPDATE [ OF column_name [ ,...n ]
] } ]
```

Cursoare

- Declararea unui cursor – sintaxa Transact-SQL:

```
DECLARE cursor_name CURSOR [ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR select_statement
[ FOR UPDATE [ OF column_name [ ,...n ] ] ]
```

Cursoare - Exemplu

```
DECLARE @nume NVARCHAR(50), @prenume NVARCHAR(50), @oraș NVARCHAR(50);
DECLARE cursorpersoane CURSOR FAST_FORWARD FOR
SELECT prenume, nume, oraș FROM Persoane;
OPEN cursorpersoane;
FETCH NEXT FROM cursorpersoane INTO @prenume, @nume, @oraș;
WHILE @@FETCH_STATUS=0
        BEGIN
        PRINT @prenume+ N' '+@nume+ N' s-a născut în orașul '+@oraș;
        FETCH NEXT FROM cursorpersoane INTO @prenume, @nume, @oraș;
        END
CLOSE cursorpersoane;
DEALLOCATE cursorpersoane;
```