Proceduri stocate în Transact-SQL
Funcții definite de utilizator
View
Cursor
Trigger

Curs 4

Proceduri stocate

- Procedură stocată – secvență de instrucțiuni T-SQL salvată sub un nume pe server; execută o anumită sarcină și poate fi (re)utilizată de mai multe aplicații.
- Avantaje:
  - Deoarece ele sunt stocate pe server nu mai necesită transfer de informaţii din baza de date, ele fiind pe acelaşi calculator;
  - Se poate reutiliza codul
  - Performanță îmbunătățită
  - Securitate și tratarea erorilor (se pot trata erori în interiorul procedurii stocate)

Procedură stocată

- Sintaxa pentru crearea și execuția unei proceduri stocate:

```
CREATE PROCEDURE NumeProcedura (@p1 datatype1, @p2 datatype2,
...) AS
BEGIN
--secventa de comenzi SQL
END
GO

--executie
EXEC NumeProcedura

--sau
NumeProcedura
```

Următoarele instrucțiuni creează o procedură stocată fără parametri și o execută

```
CREATE PROCEDURE uspGetCourseTitle
AS
    SELECT Title
    FROM Course
GO

EXEC uspGetCourseTitle
GO
```

Următoarele instrucțiuni modifică procedura adăugându-i parametrii și o execută

```
ALTER PROCEDURE uspGetCourseTitle(@creditNr int)
AS
     SELECT Title
     FROM Course
     WHERE credits = @credits
GO

EXEC uspGetCourseTitle @creditNr = 5
GO
```

Variabile

- Declararea unei variabile:

```
DECLARE {{ @local_variable [AS] data_type } [
=value ] } [,...n]
```

- Asignarea unei valori sau schimbarea valorii unei variabile:

```
SET @local_variable {+= | -= | *= | /= | %= | &=
| ^= | |= } expression
```

Următoarele instrucțiuni creează și execută o procedură stocată cu parametru OUTPUT

```
ALTER PROCEDURE uspGetCourseTitle(@creditNr int, @Number int output)
      AS
            SELECT @Number = COUNT(*)
            FROM Courses
            WHERE credits = @creditNr
      GO
```

- Execuție:

```
      DECLARE @Nr int
      SET @Nr = 0
      exec uspGetCourseTitle 6, @Number=@Nr output
      print @Nr
```

RAISERROR

- generează un mesaj de eroare și inițiază procesarea erorilor pentru sesiune
- sintaxa:

```
RAISERROR ({msg_id| msg_str| @local_var} {,severity, state})
```

- severity :
  - Utilizatorul poate folosi severity Levels 0-18
  - Sys admin poate folosi și severity Levels 19-25
- state :
  - Un număr între 0 și 255 pentru a identifica unde a apărut eroarea

Exemplu procedură stocată cu RAISERROR

```
ALTER PROCEDURE uspGetCourseTitle(@creditNr int, @Number
int output)
     AS
     BEGIN
          SELECT @Number = COUNT(*)
          FROM Courses
          WHERE credits = @creditNr

          If @Number = 0
               RAISERROR ('no courses found', 10, 1)
     END
     GO
```

Procedură stocată

- Ștergerea unei proceduri:

```
DROP PROCEDURE <procedure-name>

DROP PROCEDURE uspGetCourseTitle
```

Instrucţiuni de control al execuţiei

- Ca şi în alte limbaje procedurale, ordinea de execuţie a instrucţiunilor unui program PL/SQL poate fi controlată prin mai multe instrucţiuni de control

```
IF conditie
{ instructiuni_sql}
[ ELSE
{instructiuni_sql} ]
```

Instrucţiuni de control al execuţiei

```
WHILE conditie
{ instructiuni_sql | BREAK | CONTINUE }
```

Instrucţiuni de control al execuţiei

```
BEGIN TRY
{ instructiuni_sql }
END TRY
BEGIN CATCH
[ { instructiuni_sql }
END CATCH
```

Funcții definite de utilizator/ User-defined Functions

- programatorul își poate defini propriile funcții, pe care le poate utiliza apoi în interogări SQL;
- 3 tipuri de funcții definite de utilizator în SQL Server:
  - *Scalare* – returnează o valoare;
  - *inline table-valued* – returnează un tabel; se utilizează de obicei în clauza FROM a unei interogări;
  - *multi-statement table-valued* – returnează un tabel; spre deosebire de funcțiile *inline table-valued,* o astfel de funcție poate conține mai mult de o instrucțiune.

Exemplu funcție scalară

```
CREATE FUNCTION ufGetCoursesNr (@creditNr int)
RETURNS int AS
BEGIN
      DECLARE @nr int
      SET @nr = 0

      SELECT @nr = COUNT(*)
      FROM Courses
      WHERE credits = @creditNr

      RETURN @nr
END
----------------------------
print dbo.ufGetCoursesNr(6)
```

Funcție scalară

- **Obs.** Un dezavantaj important al funcțiilor scalare:
  - O funcție scalară care se folosește la mai multe tupluri se executa de către SQL Server pentru fiecare tuplu în parte → pot apărea probleme de eficiență
- Exemplu:

```
select *
from table1
where column1 = ufGetSomeFunctionValue([param])
```

- `ufGetSomeFunctionValue` -> se execută pentru fiecare rând din table1
- De fapt e asemănător cu un sub-select.

Funcție scalar – verificare de parametru

```
CREATE FUNCTION ufCheckTitle(
     @title varchar(50)
)
RETURNS BIT
AS
BEGIN
      IF patindex('%[0-9]%', @title) = 0
          RETURN 1
     RETURN 0
END
GO
```

Exemplu funcție *inline table-valued*

```
CREATE FUNCTION ufGetCourseTitle (@creditNr int)
RETURNS TABLE
AS
     RETURN
          SELECT Title
          FROM Courses
          WHERE credits = @creditNr
---------------------------------
select * from dbo.ufGetCourseTitle(6)
```

Exemplu funcție *multi-statement table-valued*:

```
CREATE FUNCTION GetStudentsByCourse
(@CourseId nchar(10))
RETURNS @StudentsByCourse table
(MatrNr varchar(11),
 LastName varchar(20),
 FirstName varchar(20))
AS
BEGIN
 INSERT INTO @StudentsByCourse
 SELECT S.MatrNr, S.LastName,
        S.FirstName
 FROM Students S, Enrolled E
 WHERE S.MatrNr = E.MatrNr
 AND E.CourseId = @CourseId
 IF @@ROWCOUNT = 0
 BEGIN
   INSERT INTO @StudentsByCourse
   VALUES (‘’,’no students’)
 END
RETURN
END
GO

-------------------------------
select *
from GetStudentsByCourse('Alg1')
```

Funcții de sistem / Built-in Functions

Funcții care sunt deja definite si pot fi folosite:

- `PATINDEX ( '%pattern%' , expression )` – returnează poziția de început a primei apariții a pattern-ului în șirul de caractere sau 0 dacă nu s-a găsit pattern-ul

- `LEN ( string_expression )` – returnează numărul de caractere

- `SUBSTRING ( expression ,start , length )` – returnează subșirul din *expression* de lungime *length* care începe pe poziția *start*

Funcții de sistem / Built-in Functions

- `LOWER ( character_expression )` – returnează șirul în care toate literele mari au fost convertite la litere mici

- `UPPER ( character_expression )` – returnează șirul în care toate literele mici au fost convertite la litere mari

- `CONCAT ( string_value1, string_value2 [, string_valueN ] )` – concatenează șirurile de caractere date și returnează șirul rezultat

Funcții de sistem / Built-in Functions

- `GETDATE ( )` – returnează data și ora curentă

- `ISNUMERIC ( expression )` – determină dacă expresia dată este un tip de date numeric valid (returnează 1 dacă este valid sau 0 dacă nu este)

- Pentru mai multe detalii: https://docs.microsoft.com/en-us/sql/t-sql/functions

Variabile globale

- SQL Server folosește variabile globale care conțin informații despre server, instrucțiunile rulate, etc.
- Valorile acestor variabile sunt gestionate de către server
- Numele variabilelor are prefixul @@
- @@ROWCOUNT – numărul de rânduri afectate de ultima instrucțiune (SELECT, INSERT, etc.)
- @@IDENTITY – conține valoarea câmpului IDENTITY al ultimei înregistrări inserate
- @@ERROR – codul de eroare al ultimei erori (0 – nu a fost eroare)
- @@SERVERNAME
- @@VERSION

Cursoare

- În anumite situații se dorește procesarea rând cu rând a unui tabel-rezultat; deschiderea unui cursor pe un tabel-rezultat permite procesarea acestuia rând cu rând;
- Un cursor e mai încet decât operații set-based, dar mai rapizi decât să scriem noi un WHILE-LOOP !!
- Folosim cursor doar atunci când chiar avem nevoie să parcurgem tabelul linie cu linie.

- Instrucțiuni Transact-SQL pentru declararea, popularea și obținerea datelor:
  - DECLARE CURSOR – declararea cursorului; se precizează instrucțiunea SELECT care va produce tabelul-rezultat;
  - OPEN – popularea cursorului; se execută instrucțiunea SELECT din instrucțiunea DECLARE CURSOR;

Cursoare

- Instrucțiuni Transact-SQL pentru declararea, popularea și obținerea datelor:
  - FETCH – obținerea rândurilor individuale din tabelul-rezultat; de obicei, se execută de mai multe ori (cel puțin o dată pentru fiecare rând din tabelul-rezultat);
  - dacă este necesar, se utilizează instrucțiuni UPDATE sau DELETE pentru a modifica rândul; acest pas este opțional;
  - CLOSE – închiderea cursorului; eliberează resurse cum ar fi tabelul-rezultat; cursorul rămâne declarat (se poate folosi instrucțiunea OPEN pentru a-l redeschide);
  - DEALLOCATE – eliberarea, în totalitate, a tuturor resurselor alocate cursorului, inclusiv a numelui acestuia; după dealocare, pentru reconstruirea cursorului trebuie utilizată instrucțiunea DECLARE.

Cursoare

- **Obs**. Într-o procedură nu e obligatoriu să fie închis (`CLOSE`) și eliberat (`DEALLOCATE`) manual cursorul pentru că acest lucru se întâmplă automat la finalul procedurii. Dar dacă îl folosim în afara unei proceduri trebuie să facem acei pași.

- **Sintaxa pentru OPEN, CLOSE, DEALLOCATE**

```
{OPEN| CLOSE| DEALLOCATE} {{[GLOBAL] cursor_name}
| @cursor_variable_name}
```

Exemplu. `OPEN Students_Cursor`

Cursoare

- Sintaxa extinsă Transact-SQL

```
DECLARE nume_cursor CURSOR [ LOCAL | GLOBAL ]
[ FORWARD_ONLY | SCROLL ]
[ STATIC | KEYSET | DYNAMIC | FAST_FORWARD ]
[ READ_ONLY | SCROLL_LOCKS | OPTIMISTIC ]
[ TYPE_WARNING ]
FOR instructiune_select
[ FOR UPDATE [ OF nume_coloana[ ,...n ] ] ]
[;]
```

Cursor Exemplu

```
DECLARE @ProductID          INT,
    @ProductName      VARCHAR(50),
    @Price            MONEY
DECLARE cursorProducts CURSOR FOR
  SELECT ProductID, ProductName, Price FROM Products
FOR READ ONLY
OPEN cursorProducts
FETCH cursorProducts INTO @ProductID, @ProductName, @Price
WHILE @@FETCH_STATUS = 0
BEGIN
       PRINT @ProductName + ' ' + CAST(@Price AS VARCHAR(10))
       -- code for processing @ProductID,@ProductName, @Price
FETCH cursorProducts INTO @ProductID, @ProductName, @Price
END
CLOSE cursorProducts
DEALLOCATE cursorProducts
```

FETCH

- Instrucțiunea FETCH are mai multe variante:
  - FETCH FIRST
  - FETCH NEXT (default)
  - FETCH PRIOR
  - FETCH LAST
  - FETCH ABSOLUTE { n | @nvar}
    - Dacă n sau @nvar este un număr **pozitiv,** atunci se ia al n-lea tuplu de la începutul tabelului
    - Dacă n sau @nvar este un număr **negativ,** atunci se ia al n-lea tuplu de la sfârșitul tabelului
    - Dacă n sau @nvar este **0,** atunci nu returnează nimic
  - FETCH RELATIVE { n | @nvar}
    - Asemănător cu FETCH ABSOLUTE, n sau @nvar pot fi numere pozitive sau negative, dar de data aceasta nu se începe numărătoarea de la începutul sau sfârșitul tabelului, ci de la rândul curent (relativ la rândul curent)
    - Dacă n sau @nvar este **0,** atunci returnează rândul curent (dacă există)

FETCH

- FETCH NEXT e singura variantă care poate fi folosită în cursorul declarat anterior
- Ca să putem folosi toate variantele de FETCH trebuie să declarăm cursorul de tip SCROLL

Cursor Exemplu

```
DECLARE @ProductID INT,   @ProductName  VARCHAR(50),   @Price MONEY
DECLARE cursorProducts CURSOR SCROLL FOR
  SELECT ProductID, ProductName, Price FROM Products
FOR READ ONLY
OPEN cursorProducts

FETCH LAST FROM cursorProducts; -- returnează ultimul rând din cursor

FETCH PRIOR FROM cursorProducts; -- returnează rândul anterior față de rândul curent

FETCH ABSOLUTE 2 FROM cursorProducts; -- returnează al doilea rând din cursor

FETCH RELATIVE 3 FROM cursorProducts; -- returnează al treilea rând după rândul curent

FETCH RELATIVE -2 FROM cursorProducts; -- returnează al doilea rând înainte de rândul curent

CLOSE cursorProducts
DEALLOCATE cursorProducts
```

Cursor

- @@FETCH_STATUS – statusul ultimei instrucțiuni FETCH
    - 0 – instrucțiunea FETCH s-a executat cu succes
    - < 0 – la instrucțiunea FETCH a apărut o eroare

View-uri

- un view creează un tabel virtual care reprezintă datele din unul sau mai multe tabele într-o manieră alternativă; conținutul tabelului virtual (coloane și rânduri) este definit de o interogare;

- sintaxa:

```
CREATE VIEW nume_view
AS instructiune_select
```

Exemplu view

```
CREATE VIEW OptionalCourses
AS
   SELECT C.Title, C.credits
   FROM Courses C
   WHERE C.optional = 1
GO

SELECT * FROM OptionalCourses
```

Triggere

- Tip special de proceduri stocate care se execută automat când se execută o instrucțiune DML (INSERT, UPDATE, DELETE) sau DDL;
- Pot fi folosite pentru a verifica anumite constrângeri pe baza de date
- Când se execută un trigger, se pot accesa două tabele speciale numite *inserted* și *deleted*.

Triggere

- Trigger FOR(AFTER) – triggerul este declanșat doar când toate operațiile precizate în instrucțiunea declanșatoare s-au executat cu succes;

- Trigger INSTEAD OF – triggerul se execută în locul acțiunii declanșatoare;

Trigger

```
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 [;] > }
```

Momentul în care se execută trigger-ul

Instrucțiunile care declanșează trigger-ul

Acțiunea trigger-ului

Exemplu Trigger

```
CREATE TRIGGER [dbo].[On_Product_Insert]
   ON [dbo].[Products]
   AFTER INSERT
AS
BEGIN
      SET NOCOUNT ON;

      insert into LogBuys (ProductName, Date, Quantity)
      select ProductName, GETDATE(), Quantity
      from inserted
END
```

Exemplu Trigger

```
CREATE TRIGGER [dbo].[On_Product_Delete]
   ON [dbo].[Products]
   AFTER DELETE
AS
BEGIN
      SET NOCOUNT ON;

      insert into LogSells (ProductName, Date, Quantity)
      select ProductName, GETDATE(), Quantity
      from deleted
END
```

Exemplu Trigger

```
ALTER TRIGGER [dbo].[On_Product_Update]
   ON [dbo].[Products]
   AFTER UPDATE
AS
BEGIN
      SET NOCOUNT ON;
      insert into LogSells (ProductName, Date, Quantity)
      select d.ProductName, GETDATE(), d.Quantity - i.Quantity
      from deleted d inner join inserted i on d.ProductId=i.ProductId
      where i.Quantity < d.Quantity

      insert into LogBuys (ProductName, Date, Quantity)
      select i.ProductName, GETDATE(), i.Quantity - d.Quantity
      from deleted d inner join inserted i on d.ProductId=i.ProductId
      where i.Quantity > d.Quantity
END
```

Trigger

- Ștergerea unui trigger:

```
DROP TRIGGER <trigger-name>
```

SET NOCOUNT

- SET NOCOUNT ON – oprește returnarea mesajului cu numărul de înregistrări afectate de către ultima instrucțiune Transact-SQL sau procedură stocată

- Variabila globală @@ROWCOUNT va fi actualizată în continuare

Tabele/View-uri de sistem

- Toate informațiile despre structura bazei de date (tabele, indecși, constrângeri, funcții, proceduri, etc.) se stocheza tot într-o structură relațională de către server
- Pentru acest lucru se folosesc tabele de sistem (gestionate de server, nu ar trebui modificate de către utilizator)
- Exemple:
  - Sys.objects – conține câte un tuplu pentru fiecare obiect (tabele, indecși, constrângeri, funcții, proceduri, etc.)
  - Sys.columns – conține câte un tuplu pentru fiecare coloana din fiecare tabel sau view; conține câte un rând pentru fiecare parametru al procedurilor stocate

Tabele/View-uri de sistem - Exemplu

```
SELECT SCHEMA_NAME(schema_id) AS schema_name ,
          name AS table_name
FROM sys.tables
WHERE OBJECTPROPERTY(object_id,'TableHasPrimaryKey') = 0
ORDER BY schema_name, table_name;
GO
```

Trigger – Exercițiu

- Pentru tabelul Students (MatrNr, LastName, FirstName, groupId) vrem să avem următoarea constrângere la inserarea valorilor:

  100 <= groupId <= 1000

- Definiți această constrângere:
  a) Ca și constrângere de integritate
  b) Cu ajutorul unui trigger, astfel încât înregistrările care nu respectă regula să nu fie inserate în tabel