Dynamische Ausführung/ Dynamic Execution. Cursors

Transact-SQL Server Stored Procedures / Datenbankprozeduren

- Eine gespeicherte Prozedur entspricht einer oder mehreren Transact-SQL-Anweisungen, die als eigenständiger Befehl benutzt werden kann
- Syntax für eine einfache gespeicherte Prozedur:

```
CREATE PROCEDURE <Name> AS
    -- Sequenz von SQL Befehele
GO
```

- Eine gespeicherte Prozedur wird mit EXEC ausgeführt:

```
EXEC <Name>
```

- oder nur

```
<Name>
```

Datenbankprozeduren - Beispiel

```
CREATE PROCEDURE getKursTitel
AS
    SELECT Titel
    FROM Kurse
GO
```

- Ausgeführt:

```
EXEC getKursTitel
```

Gespeicherte Prozedur mit **Parameter**

```
ALTER PROCEDURE getKursTitel(@Kredite int)
AS
    SELECT Titel
    FROM Kurse
    WHERE ECTS = @Kredite
GO
```

- Ausgeführt:

```
EXEC getKursTitel 6
```

Gespeicherte Prozedur mit Parameter und **Output Parameter**

```
ALTER PROCEDURE getKursTitel(@Kredite int, @Number int output)
        AS
                SELECT @Number = COUNT(*)
                FROM Kurse
                WHERE ECTS = @Kredite
        GO
```

- Wie wird das ausgeführt um den Wert des Output Parameters herauszufinden?

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

RAISERROR

- Syntax:

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

- severity :
  - der Benutzer kann severity Levels 0-18 benutzen
  - der sys admin kann auch severity Levels 19-25 benutzen
- state :
  - ein Integer zwischen 0 und 255
  - benutzt um herauszufinden wo genau ein Error vorgekommen ist

Gespeicherte Prozedur mit **RAISEERROR** Aussage

```
ALTER PROCEDURE getKursTitel(@Kredite int, @Number int
output)
      AS
      BEGIN
            SELECT @Number = COUNT(*)
            FROM Kurse
            WHERE ECTS = @Kredite

            If @Number = 0
                  RAISERROR ('Keine Kurse gefunden', 10, 1)
      END
      GO
```

Dynamische Execution

- Syntax:

```
EXEC (command)
```

- Beispiel:

```
EXEC ('SELECT * FROM Studenten WHERE age>20')
GO
```

- oder

```
DECLARE @query as varchar(MAX)
SET @query = 'SELECT * FROM Studenten WHERE age>20'
EXEC(@query)
GO
```

EXEC vs. sp_executesql

- Nachteile von EXEC:
  - kann schlecht für Performance sein (es kann sein dass SQL Server für dynamisches SQL den Ausführungsplan jedesmal neu erstellen muss)
  - potentielle Sicherheit Probleme (SQL Injection)
- Gespeicherte Prozedur **sp_executesql**:
  - manchmal viel schneller als EXEC
  - verhindert SQL Injection
  - sie können Parameter nur dort verwenden, an denen die SQL Syntax dies auch zulässt → es dürfen keine Parameter für Spalten oder Tabellennamen verwenden können
  - wenn dynamisches SQL regelmäßig verwendet wird, ist **sp_executesql** die bessere Wahl, da der Abfrageplan/ Ausführungsplan wiederverwendet werden kann

sp_executesql - Beispiel

```
DECLARE @query as nchar(50)
SET @query = 'SELECT * FROM Studenten WHERE age>@Nr'
EXECUTE sp_executesql @query, N'@Nr int', @Nr=20;
```

Bemerkung!

- Wenn Parameter auch für Spalten oder Tabellennamen verwendet werden müssen, funktioniert das nur mit EXEC, aber aufpassen:
  - brauchen wir das unbedingt?
  - welcher Benutzer hat das Recht das Schema einer Tabelle ändern? (SQL Injection)
- Wenn die Abfrage als Verkettung von Strings aufgebaut wird, passt auf wo man Leerzeichen braucht.
- Ihr könnt überprüfen ob die Abfrage korrekt aufgebaut ist indem ihr den String anzeigt mit **PRINT**

Beispiel

```
DECLARE @atr1 as varchar(10)
SET @atr1 = 'MatrNr'
DECLARE @query as varchar(MAX)
SET @query = 'SELECT '+ @atr1 + ' FROM Studenten
             WHERE age>20'
PRINT @query
EXEC(@query)
GO
```

Prozedur mit dynamischer Execution - Beispiel

Eine Prozedur, die eine Indexstruktur löscht:

```
create procedure deleteIndex (@tableName varchar(10),
  @indexName varchar(10))
   as
      begin
      declare @sqlQuery as varchar(MAX)
      set @sqlQuery = 'drop index ' +@tableName +'.'+ @indexName

      print(@sqlQuery)
      exec(@sqlQuery)
      end
 go
```

Transact-SQL User Defined Functions / Benutzerdefinierte Funktionen

- Benutzerdefinierte Funktionen erlauben den Users neue Funktionen zu definieren, die dann in den Abfragen benutzt werden können.
- von benutzerdefinierten Funktionen können kein dynamisches SQL bzw. keine temporären Tabellen verwendet werden
- SET-Anweisungen sind in einer benutzerdefinierten Funktionen nicht zulässig
- eine benutzerdefinierte Funktion darf eine andere benutzerdefinierte Funktion aufrufen

Benutzerdefinierte Funktionen

- 3 Kategorien:
  - Skalarfunktionen
  - Tabellenwertfunktionen:
    - Inline Tabellenwertfunktionen
    - Multi-valued Tabellenwertfunktionen

Skalarfunktionen

- Skalarfunktionen geben einen einzelnen Datenwert zurück
- Der Datentyp des Rückgabewerts ist in der RETURNS-Klausel anzugeben, der Wert selbst durch eine RETURN-Anweisung.
- Komplexe Maßnahmen müssen in BEGIN...END eingeschlossen werden; eine Funktion kann aber auch nur aus der RETURN-Anweisung bestehen

Skalarfunktionen - Beispiel

```
CREATE FUNCTION ufGetKurseNr (@Kredite int)
RETURNS int AS
BEGIN
	DECLARE @Return int
	SET @Return = 0
	SELECT @Return = COUNT(*)
	FROM Kurse
	WHERE ECTS = @Kredite

	RETURN @Return
END
-----------------------------
print dbo.ufGetKurseNr(6)
```

Skalarfunktionen

- **Bem.** Wichtiges **Nachteil** der Skalarfunktion:
  - Eine Skalarfunktion die auf mehreren Zeilen angewendet wird, wird vom SQL Server einmal für jede Zeile ausgeführt. Das kann ein großes Problem für die Effizienz sein.
- z.B.

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

- `ufGetSomeFunctionValue` -> wird für jede Zeile aus table1 ausgeführt
- Eigentlich ist das wie eine verschachtelte SELECT Abfrage.

Tabellenwertfunktionen

- Tabellenwertfunktionen geben eine Tabelle zurück anstatt einen einzigen Wert
- Tabellenwertfunktionen können leistungsfähige Alternativen zu Sichten sein
- Eine Tabellenwertfunktion kann überall dort verwendet werden, wo Tabellen- oder Sichtausdrücke in Transact-SQL-Abfragen zulässig sind
- Eine benutzerdefinierte Tabellenwertfunktion kann auch gespeicherte Prozeduren ersetzen, die ein einzelnes Resultset zurückgeben
- Die RETURNS-Klausel definiert einen lokalen Rückgabevariablennamen für die Tabelle, die von der Funktion zurückgegeben wird

Inline Tabellenwertfunktion

```
CREATE FUNCTION ufGetKurseTitel (@Kredite int)
RETURNS TABLE
AS
     RETURN
          SELECT Titel
          FROM Kurse
          WHERE ECTS = @Kredite
--------------------------------
select * from dbo.ufGetKurseTitel(6)
```

Multi-valued Tabellenwertfunktion

- Im Vergleich zu der Inline Tabellenwertfunktion enthält die multi-valued Tabellenwertfunktion mehrere Anweisungen.
- die Transact-SQL-Anweisungen sind in einem BEGIN…END-Block enthalten

Multi-valued Tabellenwertfunktion

```
CREATE FUNCTION
GetStudentenByKurs (@KursId
nchar(10))
RETURNS @StudentenByKurs table
(MatrNr varchar(11),
 Name varchar(20),
 Vorname varchar(20))
AS
BEGIN
 INSERT INTO @StudentenByKurs
 SELECT S.MatrNr, S.Name,
        S.Vorname
 FROM Studenten S, Enrolled E
 WHERE S.MatrNr = E.MatrNr
 AND E.KursId = @KursId
 IF @@ROWCOUNT = 0
 BEGIN
   INSERT INTO @StudentenByKurs
   VALUES (‘’,’Keine Studenten
                gefunden’)
 END
RETURN
END
GO
```

```
select * from
dbo.GetStudentenByKurs('Alg1')
```

Globale Variablen

- SQL Server hat viele globale Variablen:
  - Der Server verwaltet die Werte der globalen Variablen
  - alle globalen Variablen enthalten Informationen über den Server oder das aktuelle Session
  - globale Variablen haben den Prefix @@

Globale Variablen - Beispiele

- @@ERROR – enthält den Error Code der letzten Fehlermeldung; 0 = kein Error
- @@IDENTITY – enthält den Identity field value für die letzte Zeile auf die eine Operation (insert, select, update, delete) durchgeführt wurde
- @@ROWCOUNT – enthält die Anzahl der betroffenen Zeilen bei dem letzten SELECT, INSERT, UPDATE oder DELETE Anweisung
- @@SERVERNAME
- @@SPID – enthält den server process ID für das aktuelle Prozess
- @@SQLSTATUS – enthält den Completion Status (als Integer) des letzten SQL Anweisung; 0 = successful completion, 1 = failure, 2 = no (more) data available
- @@VERSION – enthält Daten über die Server Installation: Versionnummer, Datum, Zeit

System Tabellen

- die Informationen über alle Objekte (Tabellen, Indexstrukturen, Datenbankprozeduren, benutzerdefinierte Funktionen, Sichten, usw.) die in der Datenbank erstellt wurden, werden in System Tabellen gespeichert
- System Tabellen werden von dem Server verwaltet und sollten von keinem Benutzer direkt geändert werden
- Beispiele:
  - *Sys.objects* - enthält eine Zeile für jedes Objekt (Integritätsbedingung, Gespeicherte Prozedur, ...)
  - *Sys.columns* – enthält **eine Zeile für jede Spalte** aus jeder Tabelle oder Sicht und eine Zeile **für jedes Parameter** in einer gespeicherten Prozedur

Beispiel

```
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
```

Transact-SQL Cursors

- Die SQL Anweisungen erzeugen ein vollständiges Resultset.
- Manchmal ist es jedoch effizienter (vor allem in interaktiven und online Applications) wenn die Ergebnisse zeilenweise verarbeitet werden.
- Das Öffnen eines Cursors auf einen Resultset ermöglicht das zeilenweise Verarbeiten des Resultsets.
- Ein Cursor ist also eine Entität, die einem Resultset zugeordnet wird, sodass diese die Position für jede Zeile bestimmen kann.
- Nachdem der Cursor auf eine Zeile positioniert wird, können Operationen auf diese Zeile oder auf einen Block, der mit dieser Zeile anfängt, durchgeführt werden.

Transact-SQL Cursors

- **Bemerkung!** Wenn der Cursor dieselbe Operation für jede Zeile durchführt, dann ist eine Mengen-basierte Operation effizienter.
- Cursoren erweitern also die Verarbeitung des Resultsets mit folgenden Operationen:
  - Positionierung auf eine bestimmte in dem Resultset
  - ruft eine Zeile oder einen Block von Zeilen (der mit dieser Zeile anfängt) ab
  - bietet Zugriff zu den Daten in dem Resultset für gespeicherte Prozeduren, Triggers oder Scripts
- Transact-SQL Cursoren werden mit DECLARE CURSOR definiert und werden dann in Transact-SQL scripts, Prozeduren und Triggers benutzt.

Transact-SQL Cursors

- Wenn man mit einem Cursor arbeitet muss man folgendermaßen vorgehen:

1. Man muss die **DECLARE CURSOR** Anweisung benutzen um den Cursor zu definieren. Beim Definieren des Cursors muss man auch die SELECT-Anweisung angeben, die das Resultset des Cursors definiert
2. Man muss die **OPEN** Anweisung benutzen um den Cursor zu öffnen und aufzufüllen. D.h. die SELECT Anweisung embedded in der DECLARE Anweisung wird ausgeführt
3. Man muss die **FETCH** Anweisung benutzen um eine bestimmte Zeile aus einem Servercursor abzurufen. Eine FETCH Anweisung wird mehrmals ausgeführt (wenigstens einmal pro Zeile)

Transact-SQL Cursors

4. Wenn nötig, muss man die UPDATE oder DELETE Anweisungen benutzen um eine Zeile (auf die der Cursor positioniert ist) zu ändern (optional).
5. Man muss die **CLOSE** Anweisung benutzen um den geöffneten Cursor zu schließen. Mit dieser Anweisung wird die Zuordnung des aktuellen Resultsets zum Cursor aufgehoben und alle Cursorsperren für die Zeilen, auf die der Cursor positioniert ist, werden freigegeben.

   CLOSE sorgt dafür, dass die Daten für ein erneutes Öffnen verfügbar sind, jedoch sind das Abrufen und positionierte Aktualisieren von Daten erst dann zulässig, wenn der Cursor erneut geöffnet wird.

   Man kann CLOSE nur auf einem geöffneten Cursor anwenden.
6. Man muss die **DEALLOCATE** Anweisung benutzen um einen Cursorverweis zu entfernen. Nachdem die Zuordnung des letzten Cursorverweises aufgehoben wurde, werden alle Datenstrukturen, die den Cursor bilden, vom Server freigegeben. Nachdem ein Cursor freigegeben wurde, muss man wieder DECLARE benutzen, wenn man den Cursor wieder benutzen will.

Transact-SQL Cursors

- **Bem**. In einer gespeicherten Prozedur ist es nicht unbedingt nötig den Cursor zu schließen (`CLOSE`) und freizugeben (`DEALLOCATE`). Das wird automatisch gemacht wenn die Prozedur beendet wird.
- **Syntax von OPEN, CLOSE, DEALLOCATE**

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

Bsp. `OPEN Studenten_Cursor`

Transact-SQL Cursors

- **Syntax von DECLARE**

```
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] ] ]
```

Transact-SQL Cursors - Beispiel

```
DECLARE @ProductID          INT,
    @ProductName      VARCHAR(50),
    @ListPrice                MONEY
DECLARE cursorproducts CURSOR FOR
  SELECT ProductID, ProductName, ListPrice FROM Products.Product
FOR READ ONLY
OPEN cursorproducts
FETCH cursorproducts INTO @ProductID, @ProductName, @ListPrice
WHILE @@FETCH_STATUS = 0
BEGIN
…… -- code for processing @ProductID,@ProductName, @ListPrice
FETCH cursorproducts INTO @ProductID, @ProductName, @ListPrice
END
CLOSE cursorproducts
DEALLOCATE cursorproducts
```

Transact-SQL Cursors

- Cursoren können in vier Kategorien eingeteilt werden:
  - FORWARD-ONLY – innerhalb des Cursors können die Zeilen nur nacheinander von der ersten bis zur letzten gelesen werden; FAST-FORWARD gibt einen FORWARD-ONLY, READ ONLY Cursor
  - STATIC – definiert einen Cursor, der eine temporäre Kopie der Daten, die er verwendet, erzeugt
  - KEYSET – im Cursor ist die Mitgliedschaft und Reihenfolge der Zeilen fest, wenn der Cursor geöffnet wird. (Die Menge der Schlüssel, die die Zeilen eindeutig indetifizieren, wird in einer Tabelle in tempdb erstellt)
  - DYNAMIC – in dem Cursor werden alle in den Zeilen vorgenommenen Datenänderungen in seinem Resultset widerspiegelt
- Für mehrere Details über die Optionen:
  - https://msdn.microsoft.com/de-de/library/ms180169.aspx

Fetching und Scrolling

- Die FETCH Anweisung hat verschiedene Optionen die dem Benutzer erlauben bestimmte Zeilen anzurufen.
  - FETCH FIRST – gibt die erste Zeile im Cursor zurück und macht sie zur aktuellen Zeile
  - FETCH NEXT – gibt die Zeile, die nach der aktuellen Zeile folgt, zurück und macht diese zur aktuellen Zeile (ist FETCH NEXT der erste Datenabruf vom Cursor, dann wird die erste Zeile im Resultset zurückgegeben)
  - FETCH PRIOR – gibt die Zeile, die vor dem aktuellen Zeile ist, zurück (ist FETCH PRIOR der erste Datenabruf vom Cursor, dann wird keine Zeile zurückgegeben)
  - FETCH LAST – gibt die letzte Zeile im Cursor zurück und macht sie zur aktuellen Zeile

Fetching und Scrolling

- FETCH ABSOLUTE { n | @nvar} -
  - Wenn n oder @nvar eine **positive** Zahl ist, wird die n-te Zeile (ausgehend vom Anfang des Cursors) zurückgegeben.
  - Wenn n oder @nvar eine **negative** Zahlt ist, wird die n-te Zeile absteigend ausgehend von der letzten Zeile des Cursors zurückgegeben (n-te Zeile vor der Ende des Resultsets)
  - Wenn n oder @nvar gleich **0** ist, dann werden keine Zeilen zurückgegeben
  - Wenn wir einen Variable @nvar benutzen, dann muss @nvar *smallint*, *tinyint* oder *int* sein
- FETCH RELATIVE { n | @nvar}
  - Wenn n oder @nvar eine **positive** Zahl ist, wird die n-te Zeile nach der aktuellen Zeile zurückgegeben. Die zurückgegebende Zeile wird zur aktuellen Zeile.
  - Wenn n oder @nvar eine **negative** Zahl ist, wird die n-te Zeile vor der aktuellen Zeile zurückgegeben. Die zurückgegebende Zeile wird zur aktuellen Zeile.
  - Wenn n oder @nvar gleich **0** ist, wir die aktuelle Zeile zurückgegeben.
  - Wenn beim ersten Datenabruf vom Cursor FETCH RELATIVE auf n oder @nvar gleich 0 oder auf eine negative Zahl festgelegt wird, dann werden keine Zeilen zurückgegeben.

Fetching und Scrolling

- @@FETCH_STATUS – meldet den Status der letzten FETCH-Anweisung
  - mögliche Werte:
    - 0 – die FETCH-Anweisung war erfolgreich
    - -1 – die FETCH-Anweisung ist fehlgeschlagen, oder die Zeile war außerhalb des Resultsets
    - -2 – die abgerufene Zeile fehlt

- Für mehrere Details über FETCH:
  - https://msdn.microsoft.com/de-de/library/ms180152.aspx

Transact-SQL Cursoren in Applications

- Ein Verweis auf Cursornamen ist nur von anderen Transact-SQL Anweisungen aus möglich. Auf sie kann nicht von Datenbank-API (OLE DB, ODBC, ADO) Funktionen verwiesen werden.
- Die Zeilen des Cursors können nicht mithilfe von API-Funktionen oder API-Methoden abgerufen werden; ein Abrufender Zeilen mit FETCH-Anweisungen ist nur von Transact-SQL möglich
- Transact-SQL Cursoren sind aber sehr effizient wenn diese in gespeicherte Prozeduren und Triggers benutzt werden (diese erzeugen auch kein Network Traffic für die FETCH-Anweisungen)
- Wenn man Cursor Verarbeitung in einem Application braucht, dann kann man bestimmte Funktionen aus dem API benutzen die solche Cursor Verarbeitungen unterstützen anstatt von Transact-SQL Anweisungen