Transaktionale Erweiterungen der SQL Syntax (TSQL)

Transact SQL ist eine prozedurale Erweiterung von SQL, da nicht alle Aufgaben eines Datenbankservers mittels des lediglich deskriptiven SQL beschrieben werden können. Ein einfacher Anwendungsfall, bei dem einfaches SQL nicht ausreichend ist, ist zum Beispiel: „Erhöhe den Preis eines jeden 2. Artikels in Reihenfolge der Artikelnummer um 5 Prozent“.

Prozedurale Sprachen

Sprachen, die Schritt für Schritt dem Prozessor Anweisungen erteilen. Sie beschreiben den Arbeitsablauf, um ein bestimmtes Ergebnis zu erreichen. (z.B. C#, VB.NET,…)

Deskriptive Sprachen

Beschreiben nicht die Arbeitsabläufe, um ein Ergebnis zu erreichen, sondern das Ergebnis selbst. Wie das Ergebnis erreicht wird, ist dem Anwender der Sprache egal. (z.B. SQL)

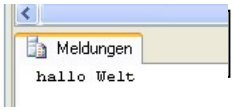
Grundlegende TSQL Strukturen

In diesem Abschnitt werden Schritt für Schritt die transaktionalen Strukturen erläutert. Generell ist anzumerken, dass die transaktionalen Erweiterungen so wie SQL selbst nicht casesensitiv sind.

Bildschirmausgabe im Mgmt Studio

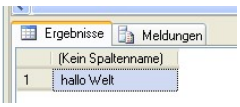
Eine einfache Bildschirmausgabe in der Konsole wird mit dem Befehl PRINT erreicht.

```
Print 'Hallo Welt'
```



Eine Ausgabe mittels SELECT Statement erzeugt immer ein Ergebnisset, das bei Bedarf an einen eventuellen Anwendungsclient weitergegeben werden kann. Das ist mittels PRINT nicht möglich.

```
SELECT 'Hallo Welt'
```



Stapeltrennzeichen GO

GO ist keine SQL-Anweisung. Die GO-Anweisung trennt eine Anweisungsfolge in einzelne Abschnitte (Stapel), die dann einzeln zum Server gesendet werden. Es wird nur im Management Studio verwendet und nicht zum Server gesendet. Man kann es gegen einen beliebigen Text austauschen (ist aber aus kompatibilitätsgründen für Skripte nicht zu empfehlen) unter: „`Extras->Optionen->Abfrageausführung->SQL Server`“

Kommentare

Es gibt in SQL Server 2 Arten von Kommentaren

```
--das ist ein einzeiliger Kommentar (mittels doppeltem --;

/*
Das ist ein Kommentar
Der auch mehrere Zeilen umfassen kann
*/
```

Lokale Variable

Wie in anderen Sprachen auch, verfügt SQL Server über die Möglichkeit, Variable zu verwenden. Es gelten folgende Prämissen:

- Variablen müssen deklariert werden
- Variablennamen müssen mit **@** beginnen
- Gültigkeitsbereich einer Variablen ist der Stapel, in dem sie deklariert wurde.
- Variablen sind nicht anfangsinitialisert

Deklaration von Variablen:

```
--SYNTAX:
      DECLARE Variablenname Datentyp

--Beispiele:
      DECLARE @zahl integer
      DECLARE @wert integer, @erg int = 5, @text varchar(20)
```

Einer Variablen können auch Werte zugewiesen werden:

```
--SYNTAX:
      SET Variablenname = Wert
      --Oder:
      SELECT Variablenname = Wert

--Beispiele:
      DECLARE @wert integer, @text varchar(20)
      SELECT @zahl = 42
      SELECT @text = 'Hallo Welt'
```

```
DECLARE @wert int
Select @wert = 4
GO


DECLARE @Anzahl int
SELECT @Anzahl = count(*)              -- SELECT zur Zuweisung
      FROM T
      WHERE stadt = 'Paris'
SELECT @anzahl                         -- SELECT zur Ausgabe


--alternativ mit SET (ist komplizierter):
SET @Anzahl = SELECT count(*)
                  FROM authors
                  WHERE city = ,Paris'
SELECT @Anzahl
```

Datentyp Table als lokale Variable

```
Declare @erg_tab                 --Definition einer
      table (tnr varchar(2),     --Tabellenvariable zur
            Stadt varchar (6))   --Speicherung von
                                 --Zwischenergebnissen

Insert into @erg_tab
      Select TNr, Stadt
      From T
      Where preis > @nr

Select * from @erg_tab           -- Ausgabe der Ergebnistabelle
GO
```

Ablaufsteuerung

Zur Ablaufsteuerung können wie üblich Ausdrücke eingesetzt werden. Folgende Operatoren kommen zur Verwendung:

- Arithmetische Operatoren: +, -, *, /, %
- Vergleichsoperatoren: =, >, <, >=, <=, <>, !=
- Logische Operatoren: NOT, AND, OR
- Zeichenverkettungsoperator: +

TSQL - Blöcke:

```
--Syntax:

      BEGIN
            Anweisungen
      END
```

Verzweigung (IF…ELSE):

```
--Syntax:

      IF Boolscher_Ausdruck
            TSQL-Anweisung / Anweisungsblock
      [ELSE TSQL-Anweisung / Anweisungsblock]

--IF Verschachtelung möglich
```

Beispiele:

```
--Wenn der Jahresumsatz der Buchnummer (title_id) größer als 5000
ist, gib eine Meldung aus:

IF (SELECT ytd_sales FROM titles WHERE title_id ='PC1035') > 5000
      PRINT 'Jahresumsatz für PC1035 ist größer als 5000!'


--Wenn der Jahresumsatz (ytd_sales) der Buchnummer (title_id)
größer als 9000 ist, dann gib den tatsächlichen Umsatz und eine
Meldung aus. Sonst gib den Preis und eine Meldung aus.

IF (SELECT ytd_sales FROM titles WHERE title_id ='PC1035') > 9000
      BEGIN
            PRINT 'Jahresumsatz für PC1035 ist größer als 9000!'
            SELECT ytd_sales FROM titles WHERE title_id ='PC1035'
      END
ELSE
      BEGIN
            PRINT 'Jahresumsatz für PC1035 ist kleiner gleich
            9000!'
            SELECT price FROM titles WHERE title_id ='PC1035'
      END
```

IF EXISTS

```
--Syntax:

      IF EXISTS (Abfrage)
            TSQL-Anweisung / Anweisungsblock
      [ELSE TSQL-Anweisung / Anweisungsblock]

--Beispiel:
      IF EXISTS (SELECT * FROM PRODUCT WHERE PRODUCT_ID=123)
            PRINT 'Datensatz existiert'
      ELSE
            PRINT 'Datensatz existiert nicht'
```

WHILE

```
--Syntax:

      WHILE Boolescher Ausdruck
            TSQL-Anweisung / Anweisungsblock
            [BREAK] TSQL Anweisung / Anweisungsblock
            [CONTINUE]

--Beispiel: Solange der Mittelwert der Tantiemen (royalty) kleiner
als 20 ist, sollen die Tantiemenwerte um 5% erhöht werden.

      WHILE (SELECT AVG(royalty) FROM roysched) < 20
            Update roysched SET royalty = royalty *1.05
```

RETURN

(unbedingter Abbruch in Anweisungsgruppe) hat gleiche Bedeutung wie BREAK steht, aber nicht in Zusammenhang mit einer WHILE Schleife, sondern einer Anweisungsgruppe. Abarbeitung wird beendet und wahlweise ein Rückgabeparameter geliefert.

GOTO Marke

Sprung zu der Marke (marke:), die sich vor einer Anweisung innerhalb der Anweisungsgruppe befinden muss.

WAITFOR

setzt eine Verzögerung oder legt für die Ausführung eine bestimmte Zeit fest

Fallunterscheidung

Innerhalb einer deskriptiven SQL Anweisung kann auch eine Fallunterscheidung implementiert werden.

```
--Syntax   CASE:

      CASE Spaltenname
            WHEN Vergleichswert THEN Rückgabewert
            ELSE else-Rückgabewert
      END

--Beispiele:

      SELECT Productnumber, Color,
            CASE Color
                  WHEN ,black' THEN ,schwarz'
                  WHEN ,red'   THEN ,rot'
            END
            As Farbe, ListPrice
      FROM Product;

      SELECT Productnumber, Weight,
            CASE
                  WHEN Weight < 100  THEN 'leicht'
                  WHEN Weight < 1000 THEN 'mittel'
                  WHEN Weight < 8000 THEN 'schwer'
                  ELSE '-'
            END
            AS Gewicht
      FROM Product;
```

Systemfunktionen & weitere Funktionalitäten

Wie üblich gibt es für spezielle Aufgaben serverseitige Funktionen.

Datumsfunktionen:
`Dateadd, datediff, datename, day, month, year`
`getdate()` → liefert aktuelles Datum mit Uhrzeit.

Zeichenfolgenfunktionen:
`len, lower, upper, ltrim, rtrim, usw.`

Systemvariable zur Statusabfrage:
@@ERROR -> Fehlernummer der letzten SQL Anweisung
@@ROWCOUNT -> Anzahl der von der letzten Anweisung betroffenen Zeilen

COALESCE
Verhindert, dass ein Ausdruck aus welchem Grund auch immer den Wert „NULL“ annimmt.

```
--Syntax:
      COALESCE (expression[,…n])

--Beispiel:
      select coalesce(avg(menge),0) from product
      --sollte der Durchschnitt NULL sein, dann wird 0 eingesetzt.
```

Konvertierungsfunkionen
CAST Funktion:

```
--Syntax:
      CAST (Wert AS Datentyp)

--Beispiel:
      Select * from T
      Print CAST (@@ROWCOUNT as CHAR(3))
```

CONVERT Funktion:

```
--Syntax:
      CONVERT (Datentyp, Wert)

--Beispiel:
      select * from T
      Print CONVERT(CHAR(3), @@ROWCOUNT)
```

Am Server gespeicherte Prozeduren („Stored Procedures“)

Stored Procedures sind Funktionen oder Prozeduren, die möglicherweise große Gruppierungen von SQL Befehlen enthalten und prozedural ausgeführt werden. Am Server können Prozeduren hinterlegt und aufgerufen werden.

```
Create Procedure              --Prozedurkopf
STP1                          --Variablennamen beginnen mit @
(                             --Referenzparameter mit OUTPUT
@param1 datatype,
@param2 datatype              --Nach AS: Prozedurrumpf
)                             --SET NOCOUNT ON verhindert die
AS                            --Rückmeldung der betroffenen Zeilen
/*set nocount on*/           --nach jedem SELECT, INSERT, …
                              --ist für Applikationsclients
                              --erforderlich
```

Ein Beispiel:

```
Create procedure ueb1 @nr int       --@nr int ist ein Parameter
As
      Declare @erg int              --Variablendefinition

      Select @erg = coalesce(sum(preis),0)
      from t
      Where preis > @nr

      Select @erg                   -- Ausgabe
GO

--Aufruf vom SQl Mgmt Studio:
Exec ueb1 20
```

Vorteile der Verwendung von Stored Procedures

- Die SQL Anweisung muss nur ein Mal am Server gespeichert werden, kann aber von mehreren Anwendungen verwendet werden.
- Aufeinander folgende Befehle laufen vollständig am DB Server ab (Zwischenergebnisse werden nicht zwischen Client und Server hin- und hergeschickt.)
- STPs können unabhängig von der Clientanwendung geändert werden, nur Name und Parameter müssen gleich bleiben.
- Schnellere Ausführung, da nach erster Ausführung am DB-Server der „optimierte Abfrageplan“ im Prozedurcache gespeichert wird.
- Berechtigungen können mittels STPs serverseitig auf Datensatzebene geprüft werden.

Stored Procedures löschen

```
Drop Procedure name
```

Stored Procedures ausführen

```
[ [ EXEC [ UTE ] ]
      {
            [ @return_status = ]
                  { procedure_name [ ;number ] |
                  @procedure_name_var
      }
      [ [ @parameter = ] { value | @variable [ OUTPUT
            ] | [ DEFAULT ] ]
      [ ,...n ]
      [ WITH RECOMPILE ]
```

Wie bei create procedure, lediglich **`@return_status`**: eine optionale, ganzzahlige Variable,
die den Rückgabewert der Prozedur speichert.

Parameterübergabe an Stored Procedure
Eingabeparameter (Werteparameter):
Parameter mit vorangestelltem @

Beispiel für Prozedurkopf:

```
Create Procedure Uebung @Eingabewert char(20)
```

Aufruf der Prozedur:

```
execute uebung 'Stringkonstante'
```

```
--Wertübergabe:
EXECUTE test @p2 = 'Hallo'    -- Reihenfolge egal

EXECUTE test 4, 'Hallo'       -- Reihenfolge nicht egal

--Standardwert angeben:
Create Procedure test @p = 5  --Standardwert = 5

EXEC test         -- p hat den Wert 5
EXEC test 20      -- p hat den Wert 20
```

Ausgabeparameter (Referenzparameter)
Parameter mit vorangestelltem **@** und Schlüsselwort **OUTPUT**

```
--Prozedurkopf:
Create Procedure uebung1 @ein char(20), @aus int OUTPUT

--Aufruf der Prozedur:
DECLARE @Wert int
Execute uebung1 'Hallo', @wert OUTPUT
Print convert (varchar(6), @wert)
```

Eine STP aus einer anderen STP aufrufen:

```
Create procedure STP2
AS
EXECUTE STP1
```

CURSOR

Ein Datenbankcursor lässt sich mit dem Cursor einer Textverarbeitung vergleichen.
Er dient dazu, eine Ergebnismenge Zeile für Zeile durchzugehen und einzeln zu bearbeiten

Anwendung:

- Man wählt eine Gruppe von Daten aus
- Man scrollt durch die Gruppe der Datensätze (auch recordset genannt)
- Man untersucht und bearbeitet die einzelne Datenzeile, auf die der Cursor gerade zeigt.

Die Ergebnismenge eines Cursors wird aus der Ergebnismenge einer SELECT Abfrage gebildet. Wenn eine Anwendung oder Prozedur wiederholt auf eine Gruppe von Datensätzen zugreifen muss, geht es schneller, einen Cursor einmalig zu erzeugen und ihn mehrmals wiederzuverwenden als wiederholt die Abfrage der Datenbank auszuführen.

Arbeiten mit Cursor

1. Erzeugen Sie den Cursor:
```
declare cursorname cursor
for Selectanweisung
[for {read only | update [of Spaltenliste]}]
```

2. Öffnen Sie den Cursor für die Verwendung innerhalb der Prozedur oder Anwendung
```
open Cursorname
```

3. Holen Sie die Daten eines Datensatzes zeilenweise, bis das Ende der Datensätze im Cursor erreicht ist.
```
FETCH[ [ NEXT | PRIOR | FIRST | LAST | ABSOLUTE { n | @nvar } |
RELATIVE { n | @nvar }]FROM] { { [ GLOBAL ] cursor_name } |
@cursor_variable_name }[ INTO @variable_name [ ,...n ] ]
```

4. Status des Cursors abfragen
   - Die globale Server Variable ***@@fetch_status***
     i. Hat den Wert 0 wenn OK
     ii. Hat den Wert 1 wenn Fetch Anweisung fehlerhaft
     iii. Hat den Wert 2 wenn keine Daten innerhalb der Ergebnismenge

   - Die globale Server Variable ***@@rowcount*** beinhaltet die Anzahl der Ergebnisdatensätze im Cursor

5. Schließen Sie den Cursor, wenn Sie die Arbeit damit beendet haben.
```
close cursorname
```

6. Geben Sie den vom Cursor belegten Speicherplatz frei um ihn vollständig zu verwerfen.
```
deallocate cursorname
```

Gültigkeitsbereich eines Cursors

- **Sitzung:** Eine Sitzung beginnt, wenn sich ein Benutzer anmeldet. Wenn sich der Benutzer an einen SQL Server angemeldet und dann einen Cursor erzeugt hat, existiert Cursorname solange, bis sich der Benutzer abmeldet. Der Benutzer kann Cursorname während der aktuellen Sitzung nicht wiederverwenden.

- **Gespeicherte Prozedur:** Ein Cursor, der innerhalb einer gespeicherten Prozedur erzeugt wird, ist nur während der Ausführung der gespeicherten Prozedur zugänglich. sobald die gespeicherte Prozedur verlassen wird, ist Cursorname nicht mehr gültig.

- **Trigger:** Ein innerhalb eines Triggers erzeugter Cursor unterliegt den gleichen Einschränkungen wie ein Cursor, der innerhalb einer gespeicherten Prozedur erzeugt wird.

Beispiel für die Verwendung eines Cursors

```
create procedure probe_cursor @grenze integer
as

set nocount on                --Unterdrückung der Messageausgabe

declare @lnr char(2)
declare @anzahl integer
select @anzahl=0

declare probecursor cursor for     --Cursor erzeugen
   select distinct lnr from lt where menge>@grenze

open probecursor                   --Cursor öffnen

fetch probecursor into @lnr        --1. Datensatz einlesen

while @@FETCH_STATUS=0             --Wiederholungsstruktur
                                   --mit Abbruchbed. Fetch_status

Begin                              --Beginn TSQL Block

      print @lnr

      select @anzahl=@anzahl+1

      fetch probecursor into @lnr  --nächsten Datensatz lesen

end                                --ENDE TSQL Block

close probecursor                  --Probecursor schliessen

deallocate probecursor             --Speicher freigeben

select @anzahl Anzahl              --Ergebnis ausgeben
go



--Aufruf der STP:
probe cursor 200
```

Überlegen Sie, was die STP „Probecursor“ macht.

Fehlerbehandlung

Der SQL Server stellt selbstverständlich auch Methoden zur Fehlerbehandlung zur Verfügung.

Jede Fehlermeldung besteht aus:

- **Fehlernummer** (bis 50.000 reserviert für Systemfehler, ab 50.001 für benutzerdefinierte Fehlermeldungen)

- **Ebene** (0-10 -> kein schwerwiegender Fehler,
  11-16 -> vom Benutzer behebbarer Fehler
  17-19 -> vom Benutzer nicht behebbarer Fehler
  20-24 -> schwerwiegender Fehler)

- Status

- Zeile

- Text

Funktionen zur Fehlerbehandlung:

```
ERROR_NUMBER()
ERROR_MESSAGE()
ERROR_SEVERITY()
ERROR_STATE()
```

Erstellung einer benutzerdefinierten Fehlermeldung

Eine benutzerdefinierte Fehlermeldung kann fix im System hinterlegt warden und mittels Fehlernummer explizit aufgerufen warden. Diese Vorgangsweise empfiehlt sich, wenn die gleiche Fehlermeldung mehrfach verwendet werden muss.

Die Fehlermeldung muss zuerst im US_Englisch hinterlegt werden. Sie wird in der Tabelle `sys.messages` eingetragen.

```
--Syntax:
      exec sp_addmessage fehlernummer, schweregrad, fehlertext,
                         sprache, protokollierung

--Beispiel:

exec sp_addmessage 50001,16,'No record found', 'us_english',
                   'FALSE'

exec sp_addmessage 50001,16,'Kein Datensatz gefunden', 'German,
```

Löschen einer benutzerdefinierten Fehlermeldung

```
--Syntax:
      exec sp_dropmessage fehlernummer, sprache
```

Fehlerauslösung mit raiserror

```
--Syntax:
      raiserror (fehlernummer, schweregrad, status)

--Beispiele:

      raiserror (50001,16,2);

      raiserror ('Mehrere Datensätze gefunden! ',16,2);
            --kann auch ohne vorher angelegte Fehlermeldung
            --verwendet werden

      raiserror ('Ungültige Produktnummer %s für Produkt %s',
                  16, 1, @Produktname, @Name);

      raiserror ('falsche Anzahl %d ',16, 1, @Anzahl);
```

TRY und CATCH

```
--Syntax:
      BEGIN TRY
            TSQL Anweisung / Anweisungsblock
      END TRY

      BEGIN CATCH
            TSQL Anweisung / Anweisungsblock
      END CATCH

--Beispiel:

      BEGIN TRY
            BEGIN Transaction
            IF EXISTS (Select * from product where product_number=8)
                  Select * from product where product_number=8
            ELSE
                  raiserror (50001,16,2);
            COMMIT Transaction
      END TRY


      BEGIN CATCH
            print error_message()
            rollback transaction
      END CATCH
```