Netz-Weise
Lernen von den Besten.

Transact SQL

von Holger Voges

Version 1.0

Freundallee 13 a
30173 Hannover
www.netz-weise.de

Inhalt

Transact-SQL Grundlagen

SQL-Server verwendet einen eigenen SQL-Dialekt namens Transact-SQL, der aber zu großen Teilen ANSI-SQL-Konform ist – ANSI-SQL ist der Industriestandard für die SQL-Abfragesprache. Jeder Hersteller hat seinen eigenen SQL-Dialekt, der sich mehr oder weniger stark vom Standard unterscheidet.

SQL (Structured Query Language) ist in 3 Bereiche aufgeteilt, die DML (Data Manipulation Language), die DCL (Data Control Language) und die DDL (Data Definition Language).

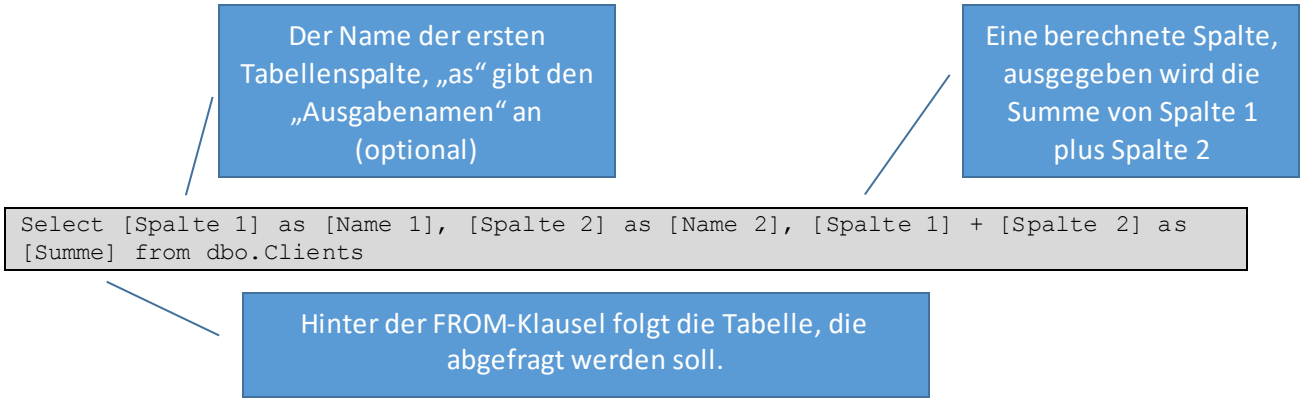
DML – Data Manipulation Language

Mit der DML können Daten vom Datenbankserver abgefragt werden. Die wesentlichen Befehle lauten:

Statement	
Select	Abfragen von Daten
Insert	Einfügen von Datensätzen
Update	Aktualisieren von Datensätzen
Delete	Löschen von Datensätzen
Merge	Eine Kombination von Select und Insert/Update

Select

Mit dem Select-Statement werden Daten aus Tabellen oder Sichten abgerufen. Die Standard-Syntax lautet:



Dem Select folgen die Spalten einer Tabelle, die abgefragt werden sollen. Die Spalten können über das *„AS“* hinter dem Spaltennamen in der Ausgabe umbenannt werden. Man kann auch berechnete Spalten erzeugen, indem man mehrere Spalten miteinander verrechnet [Spalte 1] + [Spalte 2]. Hinter dem *FROM*-Statement folgt die Tabelle, aus der die Daten geliefert werden sollen.
Auffällig ist hier der zweiteilige Name dbo.Rechnungen. Jedes Datenbankobjekt kann über einen **4-teiligen Namen** referenziert werden: Server.Datenbank.Schema.Tabelle. Der erste Abschnitt gibt den Servernamen an, der zweite Teil den Datenbanknamen, der 3. Teil das Schema (quasi ein Ordner für Tabellen), und der 4. Teil gibt den Objektnamen an. Das Standardschema für Objekte, wenn kein Schema angegeben wird, lautet dbo (database owner).
Daten in relationalen Datenbanken sind normalerweise normalisiert, was bedeutet, dass man versucht, Redundanzen in der Datenbank zu vermeiden. Dazu werden Daten, die für sich allein e stehen können, in einzelne Tabellen zusammengefasst. Will man Beispielsweise Daten für Bestellungen verwalten, legt man nicht alle Daten (Wer hat bestellt, was hat er bestellt, wie teuer war die Bestellung, wann war die Bestellung, wer hat geliefert usw.) in einer Tabelle ab, sondern versucht die Daten zu identifizieren, die als eigene Einheit (Entitäten) abgebildet werden können. Im Beispiel könnte man die Bestellungen-Tabelle also aufteilen in:

- Kunden (1 Tabelle)
- Produkte (1 Tabelle)
- Bestelldetails (1 Tabelle)
- Mitarbeiter (1 Tabelle)

usw.. Dieser Vorgang nennt sich „normalisieren der Daten“. Eine ausführliche Beschreibung zur Normalisierung finden Sie bei Wikipedia:
https://de.wikipedia.org/wiki/Normalisierung_(Datenbank)

Datensätze eingrenzen mit TOP und OFFSET-FETCH

Um die Anzahl der Datensätze, die der SQL-Server zurückliefert, einzugrenzen, kann man das TOP-Statement verwenden, oder alternativ ab SQL-Server 2012 auch OFFSET-FETCH.

TOP ist dazu da, die Menge der zurückgegebenen Datensätze zu begrenzen. Haben Sie eine Tabelle mit vielen Millionen Datensätzen, können Sie die Rückgabe einfach auf einige wenige begrenzen, indem Sie das TOP-Statement hinter den Select setzen:

```
SELECT TOP 5 * FROM [Production].[vProductAndDescription]
```

Die Zahl hinter *TOP* gibt an, wie viele Datensätze der SQL-Server zurückgeben soll. Dies senkt die Last auf dem Server erheblich, da der SQL-Server tatsächlich nur die ersten 5 Datensätze liest und danach den Lesezugriff beendet. Wenn Sie einen Prozentualen Teil an Daten bekommen wollen, verwenden Sie mit dem *TOP* das Schlüsselwort *Percent*:

```
SELECT TOP 5 PERCENT * FROM [Production].[vProductAndDescription]
```

Benötigen Sie die Daten in einer bestimmten Sortierung, können Sie die TOP-Klausel mit dem *Order BY* verwenden:

```
SELECT TOP 5 * FROM [Production].[vProductAndDescription]
ORDER BY ProductID DESC
```

Wenn Sie alle Rückgabewerte benötigen, die an der letzten Stelle identisch sind, gibt es noch das Schlüsselwort *WITH TIES*. Es arbeitet ähnlich der Medaillenvergabe bei den Olympischen Spielen: Werden mehrere Athleten mit den gleichen Medaillen ausgezeichnet, gibt es einfach mehrere Sieger. Wenn 5 Athleten mit der gleichen Sprintzeit einlaufen, gibt es statt 3 Medaillen 5. Genauso arbeitet *WITH TIES*. Haben Sie die Ausgabe auf 3 Datensätze beschränkt, aber der 3, 4 und 5 Ausgabewert sind identisch (in Verbindung mit Order BY!), werden alle 5 Datensätze ausgegeben.

```
SELECT TOP 10 WITH TIES * FROM
[ADVENTUREWORKS2012].[Production].[vProductAndDescription]
ORDER BY ProductModel
```

Seit SQL-Server 2012 gibt es eine Alternative zu TOP: *OFFSET-FETCH*. *OFFSET-FETCH* ist ein wenig komplizierter zu verwenden als *TOP*, allerdings ANSI-SQL Konform und auch leistungsfähiger. Er ist Bestandteil der *ORDER BY*-Klausel und kann auch nur mit dieser eingesetzt werden.

```
SELECT * FROM [Production].[vProductAndDescription]
ORDER BY ProductModel
OFFSET 0 ROWS FETCH NEXT 10 ROWS ONLY;
```

OFFSET bestimmt, ab welcher Zeile Daten zurückgegeben werden sollen und *FETCH* bestimmt, wie viele Zeilen zurückgegeben werden sollen. *FETCH* ist dabei OPTIONAL, *OFFSET* allerdings nicht! Wollen Sie z.B. Werte erst ab dem 50. Datensatz anzeigen, verwenden Sie:

```
SELECT * FROM [Production].[vProductAndDescription]
```

```
ORDER BY ProductModel
OFFSET 50 ROWS ;
```

OFFSET-FETCH unterstützt allerdings weder Percent noch WITH TIES.

Tabellen zusammenfügen mit JOIN

Um Daten aus normalisierten Tabellen jetzt wieder miteinander zu verknüpfen, stellt SQL das Schlüsselwort JOIN (=Zusammenfügen) zur Verfügung. Join kennt Cross Joins, Inner Joins und Outer Joins, wobei der am häufigsten genutzte Join der „Inner Join" ist.

Alle 3 Typen basieren auf dem Cross-Join. Ein Cross-Join erzeugt ein Kartesisches- oder Kreuz-Produkt zwischen Tabellen, was nichts anderes bedeutet, als dass jede Zeile der einen Tabelle mit jeder Zeile der anderen Tabelle einmal kombiniert wird. Hat man also zwei Tabellen mit jeweils 3 Datensätzen, bekommt man als Ergebnis 3x3 Datensätzen ausgegeben, nämlich jede Zeile der einen mit jeder Zeile der anderen Tabelle kombiniert.

```
CREATE TABLE JoinMe
(
ID VARCHAR(50),
Vorname VARCHAR(50),
Nachname VARCHAR(50)
)

INSERT INTO Joinme VALUES (1,'Holger','Voges')
INSERT INTO Joinme VALUES (2,'Julia','Voges')
INSERT INTO Joinme VALUES (3,'Gudrun','Voges')

SELECT Links.ID, Links.Vorname,Rechts.Vorname FROM JoinMe AS Links
CROSS JOIN JoinMe AS Rechts
```

Das Ergebnis sieht so aus:

```
ID      Vorname       Vorname
1       Holger        Holger
2       Julia         Holger
3       Gudrun        Holger
1       Holger        Julia
2       Julia         Julia
3       Gudrun        Julia
1       Holger        Gudrun
2       Julia         Gudrun
3       Gudrun        Gudrun
```

In diesem Fall haben wir auch gleich noch einen Spezialfall verwendet, nämlich den Self-Join, bei dem eine Tabelle mit sich selbst verknüpft wird.

Beim Inner Join werden mind. 2 Tabellen miteinander verknüpft, indem man dem SQL-Server eine Spalte angibt, die in beiden Tabellen die gleichen Daten verwaltet, z.B. eine Kundennummer (wenn man alle Bestellungen eines Kunden einsehen möchte), eine Bestellnummer (wenn man alle Artikel einer Bestellung sehen möchte) usw. Die JOIN-Klausel wird der FROM hintenangestellt, zusammen mit der Spalte, über die die Tabellen „zusammengeklebt" werden sollen.

```
USE KAV
SELECT  ho.tmLastInfoUpdate
       ,ho.strWinHostName
       ,ho.nStatus
       ,ag.tmUpdate
```

```
FROM    hosts AS ho
        INNER JOIN admgroups AS ag
ON ho.ngroup = ag.nID
```

tmLastInfoUpdate	strWinHostName	nStatus	tmUpdate
2013-04-15 11:38:42.247	SERVER	29	2013-03-30 21:24:20.027
2013-04-15 11:33:32.823	3GNGEL243	0	2013-02-01 10:03:19.020
2013-04-15 11:33:32.823	3GNGEL30	0	2013-02-01 10:03:19.020
2013-04-15 11:33:32.823	3GNGEL385	0	2013-02-01 10:03:19.020
2013-04-15 11:33:32.823	3ANGEL827	0	2013-02-01 10:03:19.020

In diesem Beispiel werden die beiden Tabellen admgroups und host zusammengefügt. Dabei vergleicht SQL-Server die erste Tabelle (hosts) zeilenweise mit der zweiten Tabelle (admgroups). Tatsächlich führt SQL-Server intern 2 Schritte durch. Erst wird über beide Tabellen ein Cross-Join erzeugt, also ein kartesisches Produkt. Im zweiten Schritt wird der Vergleich hinter dem ON verwendet, um das kartesische Produkt wieder zu filtern, in diesem Fall, indem nur die Spalten ausgegeben werden, die auf den beiden Spalten hinter dem on die gleichen Werte besitzen. Im Übrigen kann man hinter dem ON prinzipiell beliebige Vergleichsoperatoren verwenden, also auch z.B. <> oder Like.

Ein Outer join liefert neben den Werten des inner join auch die Werte zurück, für die in der einen Tabelle ein Wert existiert, in der Vergleichstabelle jedoch KEIN EINZIGER Vergleichswert. Dies ist wichtig, wenn man Datensätze finden möchte, die in der einen Tabelle existieren, in der anderen aber nicht. Ein klassisches Beispiel wäre z.B. eine Bestelldatenbank, in der man alle Kundendatensätze finden möchte, die noch niemals eine Bestellung aufgegeben haben. Der Outer Join kennt den Right Outer Join und den Left outer Join. Prinzipiell sind die beiden Kommandos austauschbar, da Right und Left nur angibt, ob die Datensätze aus der Rechten oder der Linken Tabelle wieder angefügt werden sollen.

```
USE KAV
SELECT  ho.tmLastInfoUpdate
       ,ho.strWinHostName
       ,ho.nStatus
       ,ag.tmUpdate
FROM    hosts AS ho
        LEFT OUTER JOIN admgroups AS ag ON ho.ngroup = ag.nID
WHERE   ag.nid IS NULL
```

Beim Outer Join wird zuerst ein Cross-Join durchgeführt, dann werden mit einem Inner Join die Spalten gefiltert, die nicht dem Join-Filter entsprechen, und dann werden die Spalten der einen Tabelle wieder vollständig eingefügt. Je nachdem, ob man einen Right-oder Left-Join durchgeführt hat, sind das die Datensätze der rechten oder linken Tabelle.

Insert

Mit dem Insert-Befehl werden neue Datensätze eingefügt. Insert benötigt mindestens die Tabelle und die Daten, die in die Tabelle eingefügt werden sollen

```
INSERT  INTO dbo.Clients
VALUES  ( 'pc1', 'pc2' )
INSERT  INTO dbo.Clients
VALUES  ( 'pc1', 'pc2' ), ( 'pc3', 'pc4' ), ( 'pc5', 'pc6' )
```

Wenn nicht alle Spalten eingefügt werden sollen, oder die Spalten nicht in der gleichen Reihenfolge eingefügt werden, wie sie in der Tabelle vorkommen, kann man die Spalten hinter dem Tabellennamen in Klammern angeben:

```
INSERT  INTO dbo.Clients ( [Name 2] )
VALUES  ( 'pc1' ), ( 'pc3' ), ( 'pc5' )
```

Um eine große Menge von Daten von einer Tabelle in eine andere zu übertragen, kann man einen Select mit einem Insert verbinden:

```
INSERT INTO dbo.Clients ( [NAME 1] )
       SELECT Computername FROM dbo.Hosts
```

http://technet.microsoft.com/de-de/library/ms174335.aspx

Update

Zum Einfügen von Daten wird der Update-Befehl verwendet. *Update* ist Datensatzbasiert, ändert also grundsätzlich einen Satz von Daten und nicht Einzeldaten. Wird kein Filter (where) verwendet, ändert update also jede Zeile innerhalb einer Tabelle.

```
UPDATE SalesLT.SalesOrderDetail
       SET UnitPrice = UnitPrice * 1.1
```

Um einzelne Datensätze zu ändern, müssen mit der *Where*-Klausel die zu ändernden Datensätze eingeschränkt werden:

```
UPDATE SalesLT.SalesOrderDetail
       SET UnitPrice = UnitPrice * 1.1
       WHERE OrderQty > 10
```

Die *Where*-Klausel filtert erst alle Datensätze, die in der Spalte OrderQty einen Wert größer 10 haben, und ändert dann nur diese Datensätze.

Delete

Delete löscht Datensätze aus einer Tabelle. Auch *Delete* arbeitet Datensatzbasiert, löscht also ohne *Where*-Klausel alle Datensätze einer Tabelle:

```
DELETE FROM SalesLT.SalesOrderDetail
```

Sollten nicht alle Datensätze gelöscht werden, muss wieder erst mit einer Where-Klausel gefiltert werden:

```
DELETE FROM SalesLT.SalesOrderDetail
WHERE ModifiedDate < GETDATE()-28
```

http://msdn.microsoft.com/de-de/library/ms189835.aspx

Soll eine komplette Tabelle gelöscht werden, empfiehlt es sich, Truncate Table zu verwenden. Truncate Table löscht eine Tabelle unter Umgehung des Transaktionsprotokolls (es wird nicht jeder Datensatz einzeln gelöscht), was bei einer großen Tabelle einen deutlichen Performance-Vorteil bedeutet:

```
TRUNCATE TABLE SalesLT.SalesOrderDetail
```

Um Änderungen rückgängig zu machen – TSQL löscht und überschreibt Daten ohne Nachfragen – ist es sinnvoll, Änderungen in einer Transaktion zu verpacken:

```
BEGIN TRANSACTION
DELETE FROM SalesLT.SalesOrderDetail
WHERE ModifiedDate < GETDATE()-28
SELECT * FROM SalesLT.SalesOrderDetail
COMMIT
```

Mit Hilfe von ROLLBACK TRANSACTION kann, bevor der Commit durchgeführt wird, die Transaktion wieder rückgängig gemacht werden.

DCL – Data Control Language

Mit der DCL werden die Berechtigungen auf dem Datenbankserver gesetzt. Die wesentlichen Befehle lauten:

Statement	
Grant	Vergeben von Berechtigungen
Deny	Verweigern von Berechtigungen
Revoke	Entfernen von Berechtigungen

Die Berechtigungsvergabe findet am einfachsten über die GUI statt.

DDL – Data Definition Language

Mit der DDL werden neue Strukturen wie Tabellen aber auch Datenbanken angelegt. Die wesentlichen Befehle lauten:

Statement	
Create	Anlegen eines Datenbankobjekts
Drop	Löschen eines Datenbankobjekts
Alter	Verändern von Datenbankobjekten

Um neue Objekte wie Tabellen, Sichten, gespeicherte Prozeduren usw. anzulegen, werden die Befehle der DDL verwendet. Drop löscht Objekte (Drop Table, Drop Database, Drop View), Create legt neue Objekte an (Create Table, Create Trigger, Create Procedure).

Sichten (Views)

Sichten machen es möglich, eine Select-Abfrage auf dem Datenbankserver zu hinterlegen und dann abzufragen wie eine Tabelle. Eine Sicht unterscheidet sich beim Aufrufen für den Anwender nicht von einer Tabelle, aber im Gegensatz zu dieser existiert eine Sicht nur im Hauptspeicher – einzige

Ausnahme hiervon sind Sichten, auf die ein Index gelegt wurde. Der Index liegt physikalisch in der Datenbankdatei – man spricht auch von materialisierten Sichten.

Für Sichten gibt es verschiedene Anwendungsfälle. Sichten können zum einen komplexe Abfragen vereinfachen, indem statt der Abfrage die Sicht aufgerufen wird. Das kann vor allem bei Abfragen mit vielen Joins hilfreich sein. Sichten können außerdem die Menge an Daten reduzieren, die vom Client zum Server übertragen werden müssen. Hat ein Benutzer nur Zugriff auf eine Sicht, aber nicht auf die darunterliegenden Daten, kann man auch die Tabellenstrukturen verbergen. Um den Code der Sicht zu verbergen, kann man sie außerdem verschlüsseln. Die Abfrage hinter der Sicht ist dann für den Benutzer nicht einsehbar. Sichten können auch verwendet werden, um den Datenzugriff für Benutzer einzuschränken (mehr dazu finden Sie u.a. in den Books online unter Besitzketten).

Das Erstellen einer Sicht ist denkbar einfach. Stellen Sie vor das Select-Statement einfach das Statement "Create View *Sichtname* AS":

```
CREATE VIEW NoSales
AS
SELECT P.[BusinessEntityID]
      ,[Title]
      ,[FirstName]
      ,[MiddleName]
      ,[LastName]
      ,[EmailPromotion]
        ,[SalesQuota]
      ,[Bonus]
      ,[CommissionPct]
      ,[SalesYTD]
      ,[SalesLastYear]
  FROM [Person].[Person] AS P
  INNER JOIN  [Sales].[SalesPerson] AS SP
  ON P.BusinessEntityID = SP.BusinessEntityID
  WHERE SalesLastYear <= 0
```

Um eine Sicht zu verschlüsseln, verwenden Sie das Schlüsselwort „with encryption“:

```
CREATE VIEW NoSales
WITH ENCRYPTION
AS
SELECT P.[BusinessEntityID]
      ,[Title]
      ,[FirstName]
      ,[MiddleName]
      ,[LastName]
      ,[EmailPromotion]
  FROM [Person].[Person]
```

Einige Dinge gibt es bei Sichten zu beachten. Wenn Sie eine Sicht erstellen, sollten alle Spalten mit Namen angegeben werden. SQL-Server speichert beim Erstellen der View die Spalten, auf die die View referenziert. Wird die der View zugrunde liegende Tabelle geändert, ändert sich die View nicht, obwohl Sie Wildcards verwendet haben! Um Verwirrungen zu vermeiden, vermeiden Sie daher auch den *!

Ein *ORDER BY* darf nur in Verbindung mit dem TOP bzw. OFFSET FETCH verwendet werden.

Wenn Sie in eine Sicht einfügen wollen, muss die Sicht mit der Option „*With Schemabinding*“ erstellt worden sein. Sie stellt sicher, dass eine Tabelle nicht verändert werden kann, solange die Sicht auf sie referenziert.

Funktionen

Funktionen sind wiederverwendbarer SQL-Code, der sich in einem Select-Statement aufrufen lässt. SQL-Server unterscheidet dabei eine Reihe von unterschiedlichen Funktionstypen. Die wichtigsten Funktionen sind:

Aggregatfunktionen

Aggregatfunktionen können aus mehreren Feldern eine Spalte ein Aggregat, also eine Zusammenfassung, bilden. Klassische Aggregatfunktionen, die regelmäßig verwendet werden, sind z.B. die SUM- oder die MAX-Funktion. Möchten Sie z.B. den größten Umsatz aus einer Spalte filtern, wenden Sie einfach MAX() auf die Spalte an:

```
SELECT MAX(TotalDue) AS BiggestSale FROM [AdventureWorks2012].
[Sales].[SalesOrderHeader]

BiggestSale
---------------------
187487,825

(1 row(s) affected)
```

Die Aggregatfunktion durchsucht alle Felder und gibt den größten Wert zurück. Da die Suche über alle Felder geht, kann eine Aggregatfunktion im Select-Statement immer nur mit Spalten zusammen stehen, über die gruppiert wird, da jetzt keine Einzelwerte mehr dargestellt werden können. Beim Gruppieren werden alle Datensätze, die den gleichen Werte in der zu gruppierenden Spalte haben, zusammengefasst. Die Aggregatfunktion wird dann jeweils über alle Elemente einer Gruppe gebildet. Interessant beim Umsatz wäre ja z.B. auch, was der jeweils größte Umsatz jedes Kunden war:

```
SELECT CustomerID, MAX(TotalDue) AS BiggestSale FROM [Sales].[SalesOrderHeader]
GROUP BY CustomerID

Customerid  BiggestSale
----------- ---------------------
14324       2535,964
22814       5,514
11407       59,659
28387       645,2869
19897       659,6408
…
```

Die Gruppen werden hier über die Kundennummer gebildet. Es werden also alle Datensätze, die vom gleichen Kunden stammen, gemeinsam verwaltet und dann über jeden Kunden jeweils sein größter Umsatz angezeigt. Genauso interessant könnte auch der größte Umsatz pro Verkäufer sein. Die Spalte, in der die Verkäufer-ID gespeichert ist, heißt „SalesPersonID“. Wie würde die Abfrage lauten?

Skalarfunktionen

Skalare sind ein Bezeichner für Einzelwerte. Eine Skalarfunktion zeichnet sich dadurch aus, dass sie nur genau einen Rückgabewert erzeugt. Skalarfunktionen werden in der Spaltenauswahl des Select-Statements aufgerufen und können dabei z.B. Berechnungen mit Spaltenwerten durchführen. Dies könnte z.B. die automatische Berechnung eines Rabattes sein. Anders als die Aggregatfunktion wird die Skalarfunktion aber auf jeden einzelnen Datensatz angewendet. Es gibt eine ganze Reihe von eingebauten Skalarfunktionen zur Bearbeitung von Texten (String-Funktionen), Mathematischen Funktionen, Datums-Funktionen, Konvertierungsfunktionen usw., von denen einige wichtige weiter unten ausführlicher beschrieben werden. Eine Auflistung aller SQL-Server Funktionen finden Sie in der MSDN-Library:
https://msdn.microsoft.com/de-de/library/ms174318(v=sql.120).aspx

Um eigene Skalarfunktionen zu erstellen, verwenden Sie das Statement Create Function:

```
CREATE FUNCTION [dbo].[ufnConvertError](@ErrorCode [tinyint])
RETURNS [nvarchar](16)
AS
BEGIN
    DECLARE @ret [nvarchar](16);

    SET @ret =
        CASE @ErrorCode
            WHEN 0 THEN N'Success'
            WHEN 1 THEN N'Type Mismatch'
            WHEN 2 THEN N'Undefined Error'
            ELSE N'** Invalid **'
        END;

    RETURN @ret
END;
```

Die Parameter werden im Header definiert und übergeben

Der Rückgabewert muss bereits im Header definiert werden

Der Rückgabewert wird über ret an den Aufrufer zurückgegeben

Das Statement beginnt mit *Create Function*, gefolgt vom Namen der Funktion. Da Funktionen einen oder mehrere Übergabeparameter haben können, werden die zu definierenden Parameter in Klammern hinter dem Namen der Funktion angegeben. Achten Sie darauf, dass Sie zur Deklaration jeweils den Namen des Parameters und den Datentypen angeben müssen. Den Parameter können Sie mit genau dem Namen, den Sie hier angegeben haben, dann im Code der Funktion verwenden. Er enthält den Wert, der der Funktion übergeben wurde. Der Parameter kann auch Standardwerte zugewiesen bekommen, die dann mit einem Gleichheitszeichen direkt zugewiesen werden:

```
CREATE FUNCTION [dbo].[ufnConvertError](@ErrorCode [tinyint] = 0)
[…]
```

Das *Returns*-Statement gibt an, was für einen Datentyp die Funktion zurückliefert – zur Erinnerung, eine Funktion kann immer nur einen Rückgabewert erzeugen! In unserem Beispiel wird ein Text von maximal 16 Zeichen zurück geliefert. Returns definiert hierbei nicht die Rückgabevariable, sondern nur den Datentyp!

Hinter dem *AS* folgt der auszuführende Code. Der Codeblock bei einer Prozedurdefinition muss immer mit einem *Begin* und einem *End* markiert werden, damit SQL-Server weiß, wann die Funktionsdefinition endet, da hinter dem Begin beliebiger SQL-Code stehen kann. Return gibt schließlich den Rückgabewert an den Aufrufer zurück. Achten Sie darauf dass die Variable, die Sie zurückgeben, auch den Datentyp hat, den Sie mit dem RETURNS-Statement im Kopf der Funktion angegeben haben.

Unsere kleine Beispielfunktion tut also nichts anderes, als einen Übergabewert zu überprüfen (der Übergabewert darf nicht größer als 255 sein, da der Datentyp auf Tinyint gesetzt wurde), und je nach Wert einen Fehlercode auszugeben. Der Aufruft folgt im *Select*-Statement und könnte z.B. den Inhalt einer Tabelle konvertieren.

```
create table Errorcodes
(
      Errordate smalldatetime NOT NULL,
      Errorcode Tinyint NOT NULL
)

Insert into Errorcodes Values (getdate(),1);
Insert into Errorcodes Values (getdate(),0);
Insert into Errorcodes Values (getdate(),3);
Insert into Errorcodes Values (getdate(),10);
Insert into Errorcodes Values (getdate(),2);

Select Errordate, dbo.ufnConvertError(Errorcode) from Errorcodes
```

Dieses kleine Beispiel erzeugt eine kleine Tabelle mit einem Datumswert und einem dazugehörigen Error-Code. Um die Skalarfunktion zu verwenden, wird sie wie eine eingebaute Funktion direkt im Select-Statement aufgerufen. Der Übergabewert wird über die Spalte Errorcode direkt in die Funktion weitergegeben.

Tabellenwert-Funktionen

Tabellenwert-Funktionen oder Inline Table Valued Functions sind Funktionen, die als Rückgabewert eine Tabelle liefern. Auch für eine Tabellenwert-Funktion gilt aber, dass die Funktion nur einen Rückgabewert erzeugen kann, der in diesem Fall aber eben eine Tabelle ist.

Tabellenwertfunktionen werden auch über das Select-Statement aufgerufen, allerdings stehen sie als Datenquelle hinter dem FROM-Statement, werden also wie eine Tabelle behandelt. Man kann sich eine Tabellenwertfunktion insofern auch wie eine parametrisierte Sicht vorstellen, da Tabellenwertfunktionen wie Skalarfunktionen Parameter übergeben bekommen können. Die Definition einer Tabellenwert-Funktion ist grundsätzlich ähnlich einer Skalarfunktion, nur wird als Rückgabewert eine Tabelle angegeben und als Code für die Funktion darf nur ein Select-Statement verwendet werden.

```
CREATE FUNCTION Production.ProductAndDescription (@CultureID AS nvarchar(6))
RETURNS TABLE
AS
RETURN
	SELECT
		p.[ProductID]
		,p.[Name]
		,pm.[Name] AS [ProductModel]
		,pmx.[CultureID]
		,pd.[Description]
	FROM [Production].[Product] p
		INNER JOIN [Production].[ProductModel] pm
		ON p.[ProductModelID] = pm.[ProductModelID]
		INNER JOIN [Production].[ProductModelProductDescriptionCulture] pmx
		ON pm.[ProductModelID] = pmx.[ProductModelID]
		INNER JOIN [Production].[ProductDescription] pd
		ON pmx.[ProductDescriptionID] = pd.[ProductDescriptionID]
	WHERE pmx.[CultureID] = @CultureID
GO
```

Die Tabellenwertfunktion ProductAndDescription ist von der Deklaration der Skalarwertfunktion ähnlich. Der erste Unterschied ist, dass als Rückgabewerte hinter dem Returns-Statement Table angegeben wird.

Hinter der AS steht nun kein Begin mehr, da die Funktion, genau wie eine Sicht, nur ein Select-Statement enthalten darf. Dafür wird direkt hinter dem AS das Schlüsselwort Returns angegeben, gefolgt vom Select-Statement.

Die Funktion kann wie eine Sicht verwendet werden, allerdings mit Parametern:

```
select Name, CultureID, Description from
[Adventureworks2012].[Production].[ProductAndDescription]('en')
```

Gespeicherte Prozeduren (Stored Procedures)

Gespeicherte Prozeduren sind auf dem SQL-Server hinterlegter SQL-Code, der über den Namen der gespeicherten Prozedur aufgerufen werden kann. Man kann sich eine gespeicherte Prozedur wie ein SQL-Programm vorstellen, das auf dem Server gespeichert und vom Client nur aufgerufen wird. Dies hat mehrere Vorteile:

- Der SQL-Code kann an zentraler Stelle auf dem Server gewartet und ausgetauscht werden. Eine Änderung an der Client-Software ist nicht notwendig.
- Der Code läuft oft schneller, da er auf dem Server "kompiliert" wird[1]
- Je nach Komplexität der Anfrage wird die Menge an Daten beim Aufrufen reduziert, da nur der Prozedurname aufgerufen werden muss und nicht das Script
- Der SQL-Code bleibt auf dem Server. Hier kann er vor dem Zugriff und vor Veränderungen geschützt werden
- Gespeicherte Prozeduren erlauben über eine Reihe von Sicherheitsfunktionen eine besser Berechtigungssteuerung.

Wir greifen hier wieder das Beispiel unserer Tabellenwertfunktion auf. Die Funktion besteht aus einem Select-Statement. Da eine gespeicherte Prozedur beliebigen Code aufnehmen kann, können wir das Select-Statement auch in eine gespeichert Prozedur verpacken:

```
CREATE Procedure Production.uspProductAndDescription
@CultureID AS nvarchar(6)
AS
BEGIN
	SELECT
		p.[ProductID]
		,p.[Name]
		,pm.[Name] AS [ProductModel]
		,pmx.[CultureID]
		,pd.[Description]
	FROM [Production].[Product] p
		INNER JOIN [Production].[ProductModel] pm
		ON p.[ProductModelID] = pm.[ProductModelID]
		INNER JOIN [Production].[ProductModelProductDescriptionCulture] pmx
		ON pm.[ProductModelID] = pmx.[ProductModelID]
		INNER JOIN [Production].[ProductDescription] pd
		ON pmx.[ProductDescriptionID] = pd.[ProductDescriptionID]
	WHERE pmx.[CultureID] = @CultureID
END
GO
```

Wir müssen nicht viel ändern. Das Statement zum Erstellen einer gespeicherten Prozedur lautet *Create Procedure*, gefolgt vom Prozedurnamen. Der Prozedurname fängt hier mit einem usp an, was für user defined Stored Procedure steht. SQL-Server eigene Prozeduren fangen mit einem sp an, und oft sieht man benutzerdefinierte Funktionen, die ebenfalls mit diesem Kürzel angelegt werden. Das ist bis auf Einzelfälle nicht korrekt! Wenn der SQL-Server eine gespeicherte Prozedur aufruft, die mit sp anfängt, geht er davon aus, dass es sich um eine Systemprozedur handelt und sucht diese automatisch in der Master-Datenbank! Verwenden Sie daher das Kürzel sp nicht für Ihre eigenen Prozeduren, es sei denn, das ist das gewünschte Verhalten.

Parameter werden bei gespeicherten Prozeduren nicht in Klammern angegeben, sondern als Einzelwerte, und zwar Komma-separiert. Hinter der Parameter-Deklaration folgen ein AS und ein *Begin*. Der komplette SQL-Block der gespeicherten Prozedur muss von *Begin* und *End* eingeschlossen

[1] Was in diesem Fall bedeutet, dass der Ausführungsplan auf dem SQL-Server gecached wird, was die folgenden Ausführungen beschleunigt

sein, damit SQL-Server weiß, wo die Prozedurdeklaration endet – es kann ja beliebiger Code in einer gespeicherten Prozedur aufgerufen werden.

Das Statement ist zuerst einmal dasselbe wie in unserer Prozedur. Der Aufruf sieht allerdings deutlich anders aus. Gespeicherte Prozeduren werden wie Programme "ausgeführt", indem man den Exec-Befehl vor den Prozedurnamen stellt:

```
EXEC Production.uspProductAndDescription @Cultureid='en'
```

Außerdem werden die Parameter immer mit Namen und Argument übergeben, hier also *@CultureID='en'*. Die Ausgabe ist identisch mit der Ausgabe der Tabellenwertfunktion, solange sie nicht nach einzelnen Spalten filtern wollen. Das geht nur mit dem Befehl *Select*, der aber gespeicherte Prozeduren nicht aufrufen kann. Ein weiteres Manko ist, dass die Ausgabe der gespeicherten Prozedur nur über Umwege weiterverarbeitet werden kann. Im Select kann man mit dem Schlüsselwort *INTO* Daten direkt in eine neue Tabelle weiterleiten, mit dem *INSERT* direkt in eine bestehende Tabelle einfügen oder die Daten einfach in eine Variable kopieren. All das ist mit einer gespeicherten Prozedur nicht möglich.

Die gespeicherte Prozedur kann aber auch für beliebigen anderen Code verwendet werden. Möchten Sie z.B. Daten in eine Tabelle einfügen, aber die eingefügten Werte vorher überprüfen, kann eine gespeicherte Prozedur sehr nützlich sein – sie kann ja jeglichen Code enthalten:

```
ALTER PROCEDURE uspInsertErrors
@date SMALLDATETIME = Null,
@Errorcode TINYINT = 2
AS
BEGIN
	IF @errorcode NOT BETWEEN -1 AND 3
	BEGIN
		PRINT 'Ungültiger ErrorCode';
		RETURN;
	END
	IF @date IS NULL
		SET @date = getdate();
	INSERT INTO [dbo].[Errorcodes] VALUES (@date,@ErrorCode);

End
```

Die Prozedur übernimmt 2 Parameter, *@date* für das Datum und *@Errorcode*. Der Code besteht zum einen aus einer Überprüfung des eingegebenen Errorcodes. Liegt der Code außerhalb des definierten Bereichs 0-3 wird eine Fehlermeldung zurückgegeben und die Prozedur wird mit *RETURN* abgebrochen. Die Überprüfung findet über das *IF*-Statement statt. Da der Code, der hinter dem *IF* ausgeführt werden soll, aus mehr als einer Zeile besteht, wird der Code wieder mit *Begin* und *End* umschlossen.

Die zweite *IF*-Überprüfung testet, ob das übergebene Datum *NULL* ist. Dies ist ein kleiner Kunstgriff. Die Prozedur soll immer das aktuelle Datum in die Spalte Date eintragen, wenn kein Wert an die Prozedur übergeben wird. Das machen wir normalerweise mit einem Standardwert, der im Prozedurkopf hinter der Variablen angegeben wird – wie bei der Variablen *@Errorcode*. Dummerweise kann man als Standardwerte aber keine Funktionen angeben, so dass der *getdate()* nicht im Prozedurkopf aufgerufen werden kann. Also weisen wir als Standardwert *NULL* (also leer) zu und überprüfen mit dem *IF*, ob ein Wert übergeben wurde (der Standardwert also überschrieben ist) oder die Variable immer noch leer ist. Ist sie leer, weist der *IF* der Variablen einen neuen Wert zu. In diesem Fall steht kein *Begin-End* um den *IF*-Block. SQL-Server geht dann automatisch davon aus, dass nur der nächste Befehl zum *IF* gehört und der *IF* danach beendet ist.

Aufrufen kann man die Prozedur jetzt auf verschiedene Weisen:

```
EXEC uspInsertErrors
EXEC uspInsertErrors @Errorcode = 15

DECLARE @Datum SMALLDATETIME
SELECT @Datum = getdate()-1
EXEC uspInsertErrors @Errorcode = 1, @date = @Datum
```

Unterabfragen (Subqueries)

Unterabfragen sind Select-Statements (inneres Statement), die in einem (äußeren) Select-Statement als Datenquelle dienen. Sie werden oft dafür benutzt, um Code zu verkürzen, denn Sie vermeiden, dass man die Ergebnisse einer Abfrage, die nur als Zwischenergebnis genutzt werden soll, erst speichern muss. Eine Unterabfrage kann dabei einen einzelnen Wert (Skalar), mehrere Werte aus einer Spalte oder eine ganze Tabelle zurück liefern. Wenn eine Unterabfrage unabhängig von der äußeren Abfrage ist, also auch alleine ausgeführt werden kann, spricht man von einer eigenständigen Unterabfrage. Eine Unterabfrage kann aber auch die Ergebnisse einer äuße ren Abfrage als Argumente übernehmen. Man spricht dann von einer korrelierten Unterabfrage.

Unterabfragen können dabei an verschiedenen Stellen verwendet werden:

Direkt in der Spaltenauswahl eines Select. Die Unterabfrage darf dann nur einen einzelnen Wert zurück liefern. Ein Beispiel für die Anwendung einer Unterabfrage in einem Select wäre z.B. die Erzeugung laufender Aggregate, die später behandelt werden – s.u.

In der FROM-Klausel des Select-Statements kann die Unterabfrage als Datenquelle für den Select verwendet werden. Genau genommen handelt es sich hier nicht um eine Unterabfrage, sondern um eine sogenannte abgeleitete Tabelle.

Im Where-Statement kann die Unterabfrage zur Filterung von Daten verwendet werden.

```
SELECT orderid, orderdate, empid, custid
FROM Sales.Orders
WHERE orderid = (SELECT MAX(O.orderid)
FROM Sales.Orders AS O);
```

Hier wird das Select-Statement verwendet, um den größten Wert (Skalarfunktion MAX) der Spalte OrderID zu ermitteln. Da nur ein Wert zurückgeliefert wird, handelt es sich um eine Skalare Unterabfrage, und da die Abfrage auch alleine ausgeführt werden kann, ist sie eigenständig.

Mehrwertige Unterabfragen werden oft mit dem "IN"-Operator verwendet.

```
SELECT SalesOrderID
FROM Sales.SalesOrderHeader
WHERE SalesPersonID IN
(SELECT E.[BusinessEntityID]
FROM [HumanResources].[Employee] AS E
WHERE E.LoginID LIKE N'adventure-works\D%');
```

Korrelierte Unterabfragen

Eine korrelierte Unterabfrage ist eine verschachtelte Unterabfrage, bei der die Ausgabe des äußeren Select-Statements in der Unterabfrage zur weiteren Filterung verwendet werden kann. Hierbei ist wichtig, dass sowohl die äußere als auch die innere Abfrage mit einem Alias versehen werden, damit auf die einzelnen Spalten sowohl der inneren als auch der äußeren Abfrage referenziert werden kann.

```
SELECT custid, orderid, orderdate, empid
FROM Sales.Orders AS O1
WHERE orderid =
(SELECT MAX(O2.orderid)
FROM Sales.Orders AS O2
WHERE O2.custid = O1.custid);
```

Die Abfrage sieht ein wenig kompliziert aus, da die Abfrage in 3 Schritten aufgelöst wird:

1. Zuerst wird die äußere Abfrage ausgeführt und es werden die Bestelldaten ausgelesen. Die äußere Abfrage wird mit einem Alias o1 versehen.
2. Nun wird die innere Abfrage ausgeführt. Da die äußere Abfrage bereits fertig ist, kann auf die Spalte custid über das alias o1.custid zugegriffen werden. Die innere Abfrage filtert die jeweils größte Bestellnummer für den jeweiligen Kunden.
3. Das Ergebnis, also der Maximalwert, wird als Filter in der Where-Klausel verwendet, um die äußere Abfrage zu filtern.

Wichtig ist hier zu verstehen, dass die Abfragen in einzelnen Schritten abgearbeitet werden, also erst der Select der äußeren Abfrage, dann die innere Abfrage, und schließlich die Where-Bedingung.

Laufende Aggregate

Mit Hilfe der korrelierten Unterabfragen kann man nun im Select-Statement laufende Aggregate berechnen, also z.B. das aufaddieren einzelner Ergebnisse einer Abfrage. Hierfür ein Beispiel:

Eine Tabelle beinhaltet die Spalten Bestelljahr und Bestellmenge. Man kann die Daten über folgende Abfrage ausgeben:

```
SELECT orderyear, qty
FROM Sales.OrderTotalsByYear;
```

Will man die Bestellmengen der einzelnen Jahre in einer dritten Spalte aufaddieren, um jeweils die Summe der vorgehenden Jahre auszugeben, kann man folgende Unterabfrage verwenden:

```
SELECT orderyear, qty,
(SELECT SUM(O2.qty)
FROM Sales.OrderTotalsByYear AS O2
WHERE O2.orderyear <= O1.orderyear) AS runqty
FROM Sales.OrderTotalsByYear AS O1
ORDER BY orderyear;
```

Die Unterabfrage

```
(SELECT SUM(O2.qty)
FROM Sales.OrderTotalsByYear AS O2
WHERE O2.orderyear <= O1.orderyear) AS runqty
```

summiert alle Spalten aus der Abfrage, die kleiner oder gleich dem Bestelljahr sind und erzeugt daraus eine neue Spalte runqty.

Common Table Expressions (CTE)

Common Table Expessions wurden mit SQL Server 2005 eingeführt. Sie bringen im eigentlichen Sinne keine neue Funktionalität mit, sondern sollen die Verwendung von Unterabfragen erleichtern. Das Problem bei Unterabfragen liegt darin, dass Sie nur im Speicher des SQL Server vorliegen und man

nicht über Namen aus sie zugreifen kann. Will man beispielsweise mehrfach auf die gleichen Datenquellen zugreifen, muss man jedes Mal die komplette Unterabfrage erneut eingeben. Mit einer CTE kann man eine Unterabfrage mit einem Namen versehen, und dann einfach auf die CTE zugreifen, statt jedes Mal wieder die Unterabfrage komplett einzutippen. Man erzeugt also quasi im Hauptspeicher eine benannte temporäre Tabelle aus einer Unterabfrage, und kann dann auf diese zugreifen.

Bei der Definition von CTEs sind ein paar Besonderheiten zu beachten. Die CTE ist Bestandteil einer Select-Abfrage, wird aber vor dem Select mit dem With-Statement gestartet. Da With aber normalerweise Optionen in verschiedenen Statements definiert, muss der Befehl vor der CTE immer mit einem Semikolon abgeschlossen werden, da SQL-Server sonst nicht erkennt, dass es sich um eine CTE handelt. Wie bei abgeleiteten Tabellen müssen in der CTE außerdem alle abgefragten Spalten explizit angegeben werde, ein * ist nicht erlaubt, und die Reihenfolge der Ausgabe darf nicht über einen "Order by" erzwungen werden.

```
Declare @empid as int = 282;

WITH C AS
(
SELECT YEAR(orderdate) AS orderyear, CustomerID
FROM Sales.SalesOrderHeader
WHERE SalesPersonID = @empid
)
SELECT orderyear, COUNT(DISTINCT CustomerID) AS numcusts
FROM C
GROUP BY orderyear;


orderyear   numcusts
2005        29
2006        47
2007        36
2008        34
```

CTE-Definition

Mit der ersten Zeile wird ein Parameter deklariert. Wichtig ist, dass die Deklaration der Variablen mit einem Semikolon abgeschlossen wird.

Die Definition der CTE startet mit dem WITH-Statement, gefolgt vom Namen der CTE (das Tabellenalias, auf das später auf die CTE zugegriffen wird) und einem AS. Die CTE wird dann wie eine Unterabfrage in Klammern angegeben. Da die CTE erzeugt wird, bevor das folgende Select-Statement ausgeführt wird, kann im Select-Statement über den Namen direkt über die CTE zugegriffen werden, als würde es sich um eine Tabelle handeln.

Achtung! Die CTE ist Bestandteil des Select-Statements und kann nicht einzeln verwendet werden. Nach Abschluss des Select ist die CTE nicht mehr verfügbar!

Um mehrere CTEs in einem Select zu verwenden, kann man einfach mehrere WITH-Deklarationen hintereinander mit Kommas trennen:

```
WITH C1 AS
(
SELECT YEAR(orderdate) AS orderyear, CustomerID
FROM Sales.SalesOrderHeader
),
C2 AS
(
SELECT orderyear, COUNT(DISTINCT CustomerID) AS numcusts
FROM C1
GROUP BY orderyear
)
```

```
SELECT orderyear, numcusts
FROM C2
WHERE numcusts > 1000;
```

Der Apply-Operator

Den *JOIN*-Operator haben Sie weiter oben bereits kennen gelernt. Er ist dafür da, mehrere Tabellen miteinander zu verknüpfen. Das klappt mit statischen Tabellen hervorragend. Möchten Sie aber z.B. eine Tabelle mit einer Funktion verknüpfen, können Sie *JOIN* nicht verwenden. Hier kommt der APPLY-Operator ins Spiel, den Microsoft mit SQL-Server 2005 implementiert hat. Er kann das Ergebnis einer Tabelle mit einer Tabellenwertfunktion zusammenfügen, als würde es sich um eine Tabelle handeln. Dazu ein kleines Beispiel.

Wir erstellen zuerst eine Tabellenwertfunktion fn_sales, die zu einem Mitarbeiter die 3 höchsten Umsätze ausgibt.

```
CREATE FUNCTION Adventureworks2012.[dbo].[fn_sales] (@SalesPersonID int)
RETURNS TABLE
AS
RETURN
(
  SELECT TOP 3      SalesPersonID,
                          ROUND(TotalDue, 2) AS SalesAmount
  FROM Sales.SalesOrderHeader
  WHERE SalesPersonID = @SalesPersonID
  ORDER BY TotalDue DESC
)
```

Die SalespersonID wird als Parameter übergeben. Ruft man Die Funktion mit einer beliebigen Mitarbeiternummer auf, erhält man max. 3 Rückgabewerte:

```
select * from FN_Sales(280)

SalesPersonID SalesAmount
------------- ---------------------
280           105493.63
280           103227.00
280           99023.67

(3 row(s) affected)
```

Will man die Ergebnistabelle mit einer Tabelle verknüpfen, verwendet man Apply. In diesem Beispiel fügen wir die Sicht vSalesPerson mit der Funktion zusammen, um für jeden Mitarbeiter die 3 höchsten Verkäufe zur Sicht hinzuzufügen.

```
SELECT sp.FirstName + ' ' + sp.LastName AS FullName,
            fn.SalesAmount
FROM Sales.vSalesPerson AS sp
CROSS APPLY fn_sales(sp.BusinessEntityID) AS fn
ORDER BY sp.LastName, fn.SalesAmount DESC
```

CROSS APPLY wird wie INNER JOIN aufgerufen. Sowohl die Tabelle als auch die Funktion werden mit einem Alias referenziert, um die Spalten anzugeben, die im Select ausgewählt werden. Anstatt wie beim JOIN eine Spalte anzugeben, über die die Tabellen zusammengefügt werden sollen, wird beim APPLY die Spalte der ersten Tabelle, die als Input für die Funktion verwendet werden soll , als Parameter für die Funktion verwendet. Die Funktion wird also für jeden Datensatz der Tabelle einmal angewendet.

Der *CROSS APPLY* verhält sich auch in der Ausgabe wie ein *INNER JOIN*. Nur die Datensätze, für die ein Wert in der Spalte existiert, die an die Funktion übergeben wird, wird auch ein Datensatz generiert. Würde in unserem Beispiel also ein Datensatz existieren, bei dem die Spalte BusinessEntityID keinen Wert enthält, wird der komplette Datensatz nicht mehr im Ergebnis angezeigt. Genau wie beim *LEFT* bzw. *RIGHT OUTER JOIN* kann man mit dem *OUTER APPLY* SQL-Server dazu zwingen, auch diese Datensätze auszugeben.

Fehlerbehandlung in SQL-Scripten

SQL-Server stellt grundsätzlich 2 Möglichkeiten zur Verfügung, um Fehler zu erkennen. Zum einen speichert SQL-Server den jeweils letzten Fehler, der aufgetreten ist, in der Systemvariablen *@@ERROR*. Dummerweise beinhaltet *@@ERROR* tatsächlich immer den Fehlercode des letzten Statements. Daher ist es wichtig, nach Ausführung eines Befehls, dessen Ausführungserfolg überprüft werden soll, den Inhalt von *@@ERROR* immer erst in einer anderen Variablen zu sichern.

```
DECLARE @ErrorVar INT

RAISERROR(N'Message', 16, 1);
-- Save the error number before @@ERROR is reset by
-- the IF statement.
SET @ErrorVar = @@ERROR
IF @ErrorVar <> 0
-- This PRINT statement correctly prints 'Error = 50000'.
    PRINT N'Error = ' + CAST(@ErrorVar AS NVARCHAR(8));
GO
```

Mit RAISERROR wird ein Fehler generiert. Das Ergebnis des Kommandos wird im nächsten Statement in einer Variablen gespeichert. Danach kann der Inhalt von @ErrorVar überprüft werden.

Eine einfachere Fehlerbehandlung erlaubt *TRY…CATCH*. Mit *TRY…CATCH* ist es möglich, ein Statement über *TRY* aufzurufen und nur im Falle eines Fehlers den *CATCH*-Block abzuarbeiten, der auf den Fehler reagieren kann. Tritt kein Fehler auf, wird der *CATCH*-Block nicht angesprungen.

```
BEGIN TRY
    -- Generate a divide-by-zero error.
    SELECT 1/0;
END TRY
BEGIN CATCH
    SELECT
        ERROR_NUMBER() AS ErrorNumber
        ,ERROR_SEVERITY() AS ErrorSeverity
        ,ERROR_STATE() AS ErrorState
        ,ERROR_PROCEDURE() AS ErrorProcedure
        ,ERROR_LINE() AS ErrorLine
        ,ERROR_MESSAGE() AS ErrorMessage;
END CATCH;
GO

ErrorNumber ErrorSeverity ErrorState  ErrorProcedure  ErrorLine   ErrorMessage
8134        16            1           NULL            3           Divide by zero
error encountered.

(1 row(s) affected)
```

Der Try-Block wird über BEGIN TRY gestartet. Hinter dem TRY folgt das auszuführende Script. Im Beispiel wird immer ein Fehler durch eine Division durch Null erzeugt. Der TRY-BLOCK endet mit einem END TRY.

Direkt auf den TRY-Block folgt der BEGIN CATCH Block. Er gibt die sämtliche Fehlerinformationen aus. Hierfür stehen eine ganze Reihe von Systemfunktionen zur Verfügung.

Eine ausführliche Beschreibung zum TRY-CATCH finden Sie bei Technet:
https://technet.microsoft.com/en-us/library/ms175976(v=sql.110).aspx

Funktionsreferenz

String-Funktionen

Um die Ausgabe von Spalten, die das Select-Statement abfragt, zu bearbeiten, stehen Funktionen zur Verfügung. Strings sind Zeichenketten und können in SQL-Server entweder als 16Bit-Unicode oder als ASCII-Wert gespeichert werden. Während Unicode quasi jedes Buchstabenzeichen jeder Sprache kodieren kann, muss bei ASCII explizit die Codetabelle angegeben werden. Allerdings ist der Nachteil von Unicode, dass zur Speicherung des gleichen Textes die doppelte Menge an Speicherplatz verwendet wird (16 Bit pro Zeichen statt 8). Um Text in Unicode zu speichern, muss die Spalte, in der der Text abgelegt wird, vom Datentyp nchar oder nvarchar sein. Gibt man einen Text in einem Script ein und will Unicode erzwingen, muss man dem Text ein N voran stellen. Da alle Zeichenfolgen in SQL-Server von einfachen Anführungszeichen umschlossen werden, würde ein Text in Unicode also so aussehen:

```
N'Dies ist ein Unicode-Text'
```

Zur Manipulation von Strings gibt es eine ganze Reihe von vordefinierten Funktionen, die im Folgenden kurz beschrieben werden.

Strings verknüpfen

Strings lassen sich mit dem „+“-Zeichen oder ab SQL Server 2012 mit Concat() Verknüpfen

```
SELECT empid, firstname + N' ' + lastname AS fullname
FROM HR.Employees;
```

COALESCE()

COALESECE() nimmt mehrere Werte als Input und gibt den ersten Wert aus, der nicht NULL ist.

COALESCE (expression [,...n])

```
SELECT custid, country, region, city,
country + COALESCE( N',' + region, N'') + N',' + city AS location
FROM Sales.Customers;
```

[ab SQL 2012] CONCAT()

CONCAT (string_value1, string_value2 [, string_valueN])

Mit Concat können Strings addiert werden. NULL-Werte werden durch einen Leerstring ersetzt.

```
SELECT custid, country, region, city,
CONCAT(country, N',' + region, N',' + city) AS location
FROM Sales.Customers;
```

SUBSTRING()

Liefert aus einem übergebenen String einen Teilstring zurück.

SUBSTRING(string, start, length)

```
SELECT SUBSTRING('abcde', 1, 3);
```

Quotename

Umschließt (quoted) den angegebenen String mit dem Quote-Character. Bei Klammern wird automatisch mit öffnender und schließender Klammer gequotet.

QUOTENAME ('character_string' [, 'quote_character'])

```
SELECT QUOTENAME('Ein Text','[')
```

Left(), Right()

Vereinfachte Form von Substring, liefert von Links (Textanfang) oder Rechts (Textende) eine definierte Anzahl von Zeichen zurück.

LEFT(string, n), RIGHT(string, n)

```
SELECT RIGHT('abcde', 3);
```

LEN(), DATALENGTH()

Liefert die Anzahl von Zeichen zurück. Während LEN() nur die Anzahl der Zeichen zurückgibt, gibt DATALENGTH() die Anzahl an Bytes zurück.

LEN (string)

```
SELECT LEN(N'abcde');
```

CHARINDEX()

Gibt den Index (Nummer) des ersten Auftretens eines Zeichens in einem String zurück.

CHARINDEX(substring, string[, start_pos])

```
SELECT CHARINDEX(' ','Das 4 Zeichen dieses Textes ist das erste Space');
```

PATINDEX()

PAT steht für Pattern und gibt das erste Zeichen des Auftretens eines Musters in einem Text zurück

PATINDEX(pattern, string)

```
SELECT PATINDEX('%[1-9]%', 'abcd123efgh');
```

Als Pattern werden die gleichen Platzhalter wie im LIKE-Statement verwendet. % steht hier für beliebige Zeichen, die Klammern [] stehen für genau ein Zeichen. 0-9 sind die Zeichen, die für das Muster wahr ergeben, in diesem Fall also alle Ziffern von 0-9.

REPLACE()

REPLACE(string, substring1, substring2)

Ersetzt ein Zeichenmuster durch ein anderes

```
SELECT REPLACE('Ein Text mit Leerzeichen', ' ', '');
```

Im Beispiel wird das Leerzeichen durch Nichts ersetzt, die Leerzeichen werden also entfernt.

Durch dieses Verfahren kann auch die Häufigkeit eines Zeichens in einem Text ermittelt werden

```
SELECT Len('Ein Text mit Leerzeichen') - LEN(REPLACE('Ein Text mit Leerzeichen', '
', ''));
```

REPLICATE()

REPLICATE(string, n)

Repliziert einen String n mal.

```
SELECT REPLICATE('123', 3)
```

STUFF()

Ersetzt den Teil eines Textes durch einen anderen String.

STUFF(string, pos, delete_length, insertstring)

```
SELECT STUFF('Meine Freundin wird bald laut', 26, 1, 'Br');
```

UPPER(),LOWER()

UPPER(string), LOWER(string)

Wandelt alle Zeichen eines Strings in Klein-bzw. Großbuchstaben um

```
SELECT UPPER('ich bin ganz groß')
```

RTRIM(),LTRIM()

RTRIM(string), LTRIM(string)

Entfernt anführende bzw. am Ende stehende Leerzeichen

```
SELECT RTRIM('   Ein Text mit Leerzeichen')
```

[SQL 2012] Format()

Formatiert einen String nach .NET Formatierungsanweisungen.

FORMAT(input , format_string[, culture])

```
SELECT FORMAT(10.1,'C','de-de')
SELECT FORMAT(GETDATE(),'dd/MM/yyyy','de-de')
```

Eine Auflistung der Zahlenformatzeichenfolgen finden Sie unter "Standardmäßige Zahlenformatzeichenfolgen": https://msdn.microsoft.com/library/dwhawy9k.aspx

Eine Auflistung aller String-Funktionen finden Sie bei Microsoft in der MSDN-Library:

https://msdn.microsoft.com/de-de/library/ms181984.aspx

Arbeiten mit Datumswerten

SQL-Server kennt zur Speicherung von Datumswerten verschiedene Datentypen. Dies sind im Einzelnen:

Datentyp	Größe in Byte	Wertebereich	Genauigkeit
Datetime	8	1.1.1753-31.12.9999	3 1/3 ms
SmallDateTime	4	1.1.1900-6.06.2079	1 Minute
Date	3	1.1.0001-31.12.9999	1 Tag
Time	3-5		100 Nanosek
Datetime2	6-8	1.1.0001-31.12.9999	100 Nanosek
Datetimeoffset	8-10	1.1.0001-31.12.9999	100 Nanosek

Die Typen Time, Datetime2 und Datetimeoffset haben eine variable Größe, die von der Genauigkeit abhängt, die man benötigt. Die Genauigkeit wird mit der Definition des Datentyps in Klammern angegeben und legt die Menge der Stellen hinter dem Komma fest, der gespeichert werden kann. Datetime2(3) bedeutet z.B. 3 Stellen hinter dem Sekunden-Komma und legt fest, dass Millisekunden gespeichert werden können. Bis auf Datetime und Smalldatetime sind alle anderen Datentypen erst ab SQL 2008 unterstützt.

```
declare @zeit as datetime2(5)
select @zeit = getdate()
```

Die Systemzeit ausgeben

Zum Ausgeben der Systemzeit stehen eine Reihe von Funktionen zur Verfügung, die sich vor allem im Datentyp unterscheiden, den Sie ausgeben, und welche Zeitzone sie ausgeben.

Current Date and Time	Function Return Type	Description
GETDATE()	DATETIME	Current date and time
CURRENT_TIMESTAMP (ohne ()!)	DATETIME	Same as GETDATE but ANSI SQL–compliant
GETUTCDATE()	DATETIME	Current date and time in UTC
SYSDATETIME()	DATETIME2	Current date and time
SYSUTCDATETIME()	DATETIME2	Current date and time in UTC
SYSDATETIMEOFFSET()	DATETIMEOFFSET	Current date time including time zone

Datums-Funktionen

Auch für die Bearbeitung von Datumswerten stehen eine ganze Reihe von Funktionen zur Verfügung.

DateDiff()

DATEDIFF(part, dt_val1, dt_val2)

Datediff berechnet die Differenz zwischen zwei Datumswerten dt_val1 und dt_val2. Part gibt an, welches Intervall zur Berechnung verwendet werden soll (day, month, year)

```
SELECT DATEDIFF(day, '20000101', getdate());
```

DATEADD(part, n, dt_val)

Dateadd addiert oder subtrahiert zu einem gegebenen Datumwert eine Differenz dt_val. Part gibt an, welches Intervall zur Berechnung verwendet werden soll (day, month, year), n gibt die Anzahl der Einheiten an, die addiert oder subtrahiert werden sollen.

```
SELECT DATEADD(day, 365, '20000101');
```

DatePart()

Datepart liefert einen Teil des Datumswertes zurück, und zwar nach angegebenem Typ, also z.B. Tag, Monat oder Jahr.

DATEPART(part, dt_val)

```
SELECT DATEPART(WEEKDAY, '19720526')
6
```

YEAR(),MONTH(),DAY()

Die drei Funktionen Year, Month und Day bekommen einen Datumswert übergeben und geben dann nur den jeweiligen Teil des Datums aus. Sie sind eine verkürzte Spezialversion von Datepart.

```
SELECT Year('20000101');
2000
```

DATENAME()

Funktioniert ähnlich wie Datepart, gibt aber den Namen des ausgewählten Datumswerts zurück.

```
SELECT DATENAME(Month,'20000101');
JANUARY
```

ISDATE()

ISDATE übernimmt einen String und versucht, diesen in einen Datumswert zu konvertieren. Ist der Wert konvertierbar, gibt ISDATE() 1 zurück, ist der Wert nicht konvertierbar, liefert ISDATE 0 zurück.

```
SELECT ISDATE('20000101');
```

Umwandeln von Datentypen

Zum Umwandeln von Datentypen stehen 3 Funktionen zur Verfügung, nämlich Cast, Convert und Parse.

CAST()

Cast ist der einzige ANSI-SQL konforme Konvertierungsbefehl, und auch der einfachste. Cast übernimmt einen Übergabewert und legt über das AS-Schlüsselwort fest, in welchen Datentyp der Übergabewert konvertiert werden soll.

CAST(value AS datatype)

```
select CAST('20000101' as date);
```

Schlägt die Konvertierung fehl, weil der Wert nicht in den Zieldatentypen umgewandelt werden kann, gibt CAST eine Fehlermeldung aus.

Convert()

Convert funktioniert ähnlich wie Cast, kann aber einen dritten Parameter übernehmen, nämlich Culture-Settings oder Spracheinstellungen, die die Zielsprache festlegen. Damit kann man z.B. bei Währungs- oder Datumskonvertierungen gleich die richtige Ausgabe festlegen. Die Culture-Settings sind optional und können in books online für Convert nachgelesen werden.

CONVERT (datatype, value [, style_number])

```
select Convert(date, '20000130',102);
2000-01-30
```

Parse()

Parse unterscheidet sich von den vorhergehenden beiden dadurch, dass er .NET-Funktionen zur Datumskonvertierung nutzt. Er verwendet mehr Varianten als die beiden obigen Funktionen, um den eingegebenen Wert korrekt zu konvertieren, benötigt aber auch deutlich mehr Zeit.

PARSE (value AS datatype [USING culture])

```
SELECT PARSE('12/30/2000' AS DATETIME USING 'en-US');
2007-12-30 00:00:00.000
```

Try_CAST(),TRY_CONVERT(),TRY_PARSE()

Die 3 Try-Varianten von Cast, Convert und Parse unterscheiden sich nur dadurch von Ihren Pendants, dass Sie keinen Fehler ausgeben, wenn die Konvertierung fehlschlägt, sondern NULL.

Über den Autor

Holger Voges ist IT-Trainer und Consultant. Seine IT-Karriere begann mit einem Atari ST 512 Mitte der 80er Jahre. Seine ersten Erfahrungen mit großen Netzwerken hat er im Systembetrieb der Volkswagen Financial Services 1999 gewonnen. Ab dem Jahr 2000 war er dann als freiberuflicher IT-Trainer für verschiedene Schulungsunternehmen im Bereich Braunschweig und Hannover tätig, bis er 2002 mit 2 Mitstreitern sein erstes Schulungsunternehmen LayerDrei in Braunschweig gründete. Nach seinem Ausstieg bei LayerDrei war er dann mehrere Jahre als freiberuflicher Consultant vor allem im SQL-Server Umfeld u.a. für T-Home Entertain, e.on und Hewlett-Packard unterwegs. 2012 gründete er dann das Schulungsunternehmen Netz-Weise IT-Training.

Netz-Weise IT-Training hat sich auf Firmenschulungen im professionellen IT-Umfeld spezialisiert und bietet Schulungen u.a. im Bereich Microsoft, VMware, Linux und Oracle an.