

9 T-SQL-Grundlagen

In gespeicherten Prozeduren, Triggern und benutzerdefinierten Funktionen kommen Sie teilweise mit den üblichen SQL-Anweisungen wie *SELECT*, *UPDATE*, *CREATE* et cetera aus. Gelegentlich werden Sie jedoch Geschäftslogik auf den SQL Server auslagern wollen, weil dieser spezielle Aufgaben einfach schneller erledigen kann als wenn Sie dies von Access aus durchführen. Nun beherrscht der SQL Server kein VBA, und so müssen Sie gespeicherte Prozeduren, Trigger und Co. mit T-SQL-Befehlen und -Strukturen programmieren. Dieses Kapitel liefert die notwendigen Grundlagen für die Beispiele der entsprechenden Kapitel. Dabei bewegen wir uns vorerst auf SQL Server-Ebene – die Interaktion mit Access, etwa zum Übergeben von Parametern aus Access heraus an eine gespeicherte Prozedur, besprechen wir in den folgenden Kapiteln.

Einige Möglichkeiten in T-SQL

Die Befehle von T-SQL bieten eine Reihe Möglichkeiten, die wir uns in den folgenden Abschnitten ansehen. Dazu gehören die folgenden:

- » Eingabe- und Ausgabeparameter nutzen
- » Variablen, temporäre Tabellen und *Table*-Variablen für Zwischenergebnisse verwenden
- » Programmfluss steuern
- » Anweisungen in Schleifen wiederholt ausführen
- » Daten hinzufügen, ändern und löschen
- » Systemwerte abfragen
- » Fehler behandeln

9.1 Grundlegende Informationen

Zum Einstieg einige wichtige Hinweise zum Umgang mit diesem Kapitel und den enthaltenen Techniken.

9.1.1 T-SQL-Skripte erstellen und testen

Vielleicht möchten Sie die hier abgebildeten Beispiele direkt ausprobieren. Alles was Sie dazu benötigen, ist das SQL Server Management Studio mit seinen Abfragefenstern. Ein neues Abfragefenster erhalten Sie mit der Tastenkombination *ALT + N*. Dabei sollten Sie vorher die Datenbank markieren, in der Sie das T-SQL-Skript ausführen möchten. Durch die Markierung wird das Abfragefenster mit der Verbindung zu dieser Datenbank geöffnet. Sie können diese Zuordnung

Kapitel 9 T-SQL-Grundlagen

natürlich noch nachträglich ändern. Dazu wählen Sie entweder in der Symbolleiste die Datenbank über die Auswahlliste aus (siehe Abbildung 9.1) oder Sie geben am Anfang Ihres T-SQL-Skripts die folgende Anweisung ein:

```
USE <Datenbank>;  
GO
```

Die *USE*-Anweisung wechselt zu der angegebenen Datenbank.

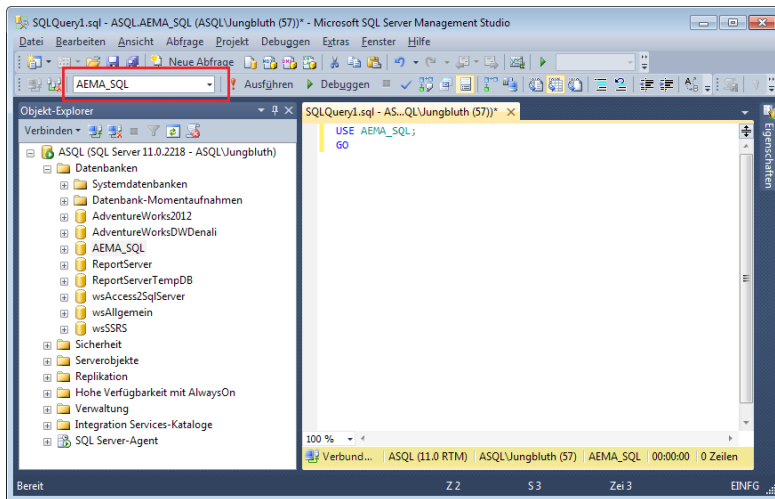


Abbildung 9.1: Die Auswahl der Datenbank für die Abfrage

Nachdem die Datenbank festgelegt ist, geben Sie im Abfragefenster die einzelnen SQL-Anweisungen zu Ihrem T-SQL-Skript ein. Dabei können Sie einzelne Anweisungen wie auch das komplette Skript mit *F5* ausführen. Das Ergebnis wird dann im unteren Bereich des Abfragefensters ausgegeben (siehe Abbildung 9.22). Mehr zu den Möglichkeiten, Abfragen zu erstellen und auszuführen, lesen Sie im Kapitel »SQL Server Management Studio«, Seite 135.

9.1.2 SELECT und PRINT

Mit *SELECT* ermitteln Sie nicht nur Daten aus einer oder mehreren Tabellen. Sie können mit *SELECT* auch Konstanten oder die Inhalte von Variablen ausgeben. Die Ergebnisse einer *SELECT*-Anweisung werden immer an den Client zurückgegeben. Im SQL Server Management Studio landen diese in der Registerkarte *Ergebnisse*. Bei einem Aufruf von Access heraus – beispielsweise über eine *Pass-Through*-Abfrage – werden die ermittelten Daten an Access übergeben.

Der *PRINT*-Befehl gibt lediglich Meldungen aus. Diese sehen Sie nach der Ausführung in der Registerkarte *Meldungen*. Bei einem längeren T-SQL-Skript, das viele Verarbeitungsschritte durchführt und erst am Ende ein Ergebnis liefert, lassen sich mit *PRINT* Informationen zu ein-

zelen Verarbeitungsschritten ausgeben – etwa Meldungen über die gerade eben ausgeführte SQL-Anweisung ergänzt mit der Anzahl der verarbeiteten Datensätze.

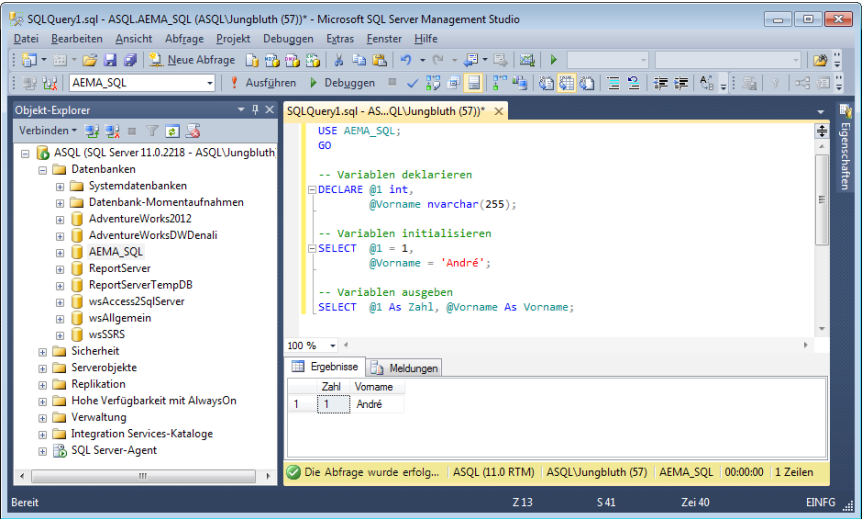


Abbildung 9.2: Ausführen von T-SQL-Skripten im Abfragefenster

SELECT liefert also immer Daten, während *PRINT* lediglich Meldungen ausgibt. Die folgenden beiden Abbildungen zeigen dies deutlich. Obwohl beide Anweisungen denselben Inhalt liefern, handelt es sich in Abbildung 9.3 um Daten und in Abbildung 9.4 lediglich um eine Meldung.

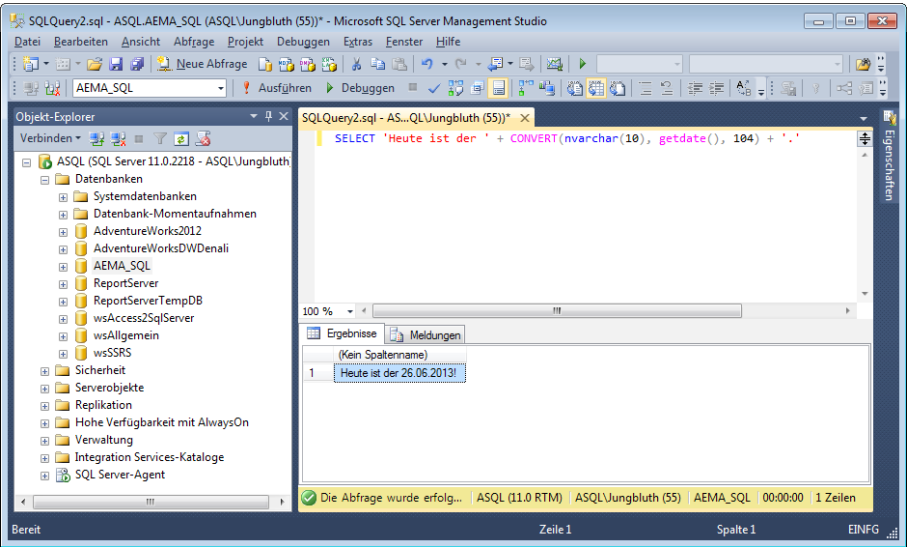


Abbildung 9.3: Ausgabe in das Meldungen-Fenster

Kapitel 9 T-SQL-Grundlagen

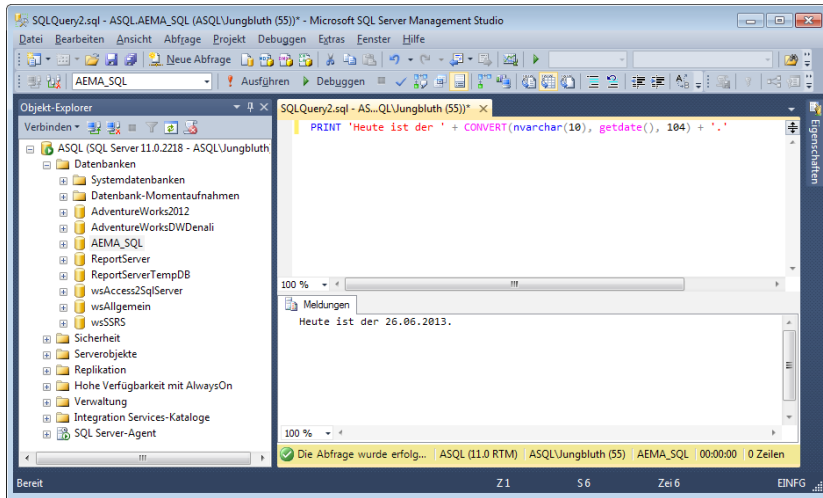


Abbildung 9.4: Ausgabe in das Ergebnisse-Fenster

9.1.3 Zeichenfolgen

Sie haben es vielleicht schon in den beiden vorherigen Abbildungen erkannt: Zeichenfolgen verkettet man unter T-SQL mit dem Plus-Operator (+). Und Literale werden ausschließlich in Hochkommata eingefasst, nicht in Anführungszeichen. Achtung: Wenn eines der per + verketteten Elemente den Wert *NULL* hat, wird der komplette Ausdruck zu *NULL*. Abhilfe schafft hier der seit SQL Server 2012 verfügbare Befehl *CONCAT*. Dieser ignoriert nur das Element mit dem *NULL*-Wert und verkettet alle anderen. In den Versionen vor 2012 müssen Sie die einzelnen Elemente in einer *CASE*-Anweisung auf mögliche *NULL*-Werte prüfen (siehe Abbildung 9.5).

Zeichenketten auf mehrere Zeilen aufteilen

Wenn Sie eine Zeichenkette im SQL-Code zur besseren Lesbarkeit aufteilen wollen, fügen Sie einfach ein Backslash-Zeichen am Ende des ersten Teils ein und fahren Sie in der folgenden Zeile mit dem Rest fort:

```
PRINT 'Dieser Zweizeiler wird in \  
einer Zeile ausgegeben.'
```

9.1.4 Datum

Beim Arbeiten mit einem Datumswert ist das Eingabeformat des Datums entscheidend – und das ist wiederum abhängig von der Standardsprache der SQL Server-Instanz. Ist diese *Deutsch*, arbeiten Sie mit dem deutschen Datumsformat, also Tag-Monat-Jahr. Steht die Standardsprache auf *Englisch*, werden Datumswerte im englischen Format (Monat-Tag-Jahr) abgelegt. Da kann schnell mal aus dem 06.07.2013 der 07.06.2013 werden.

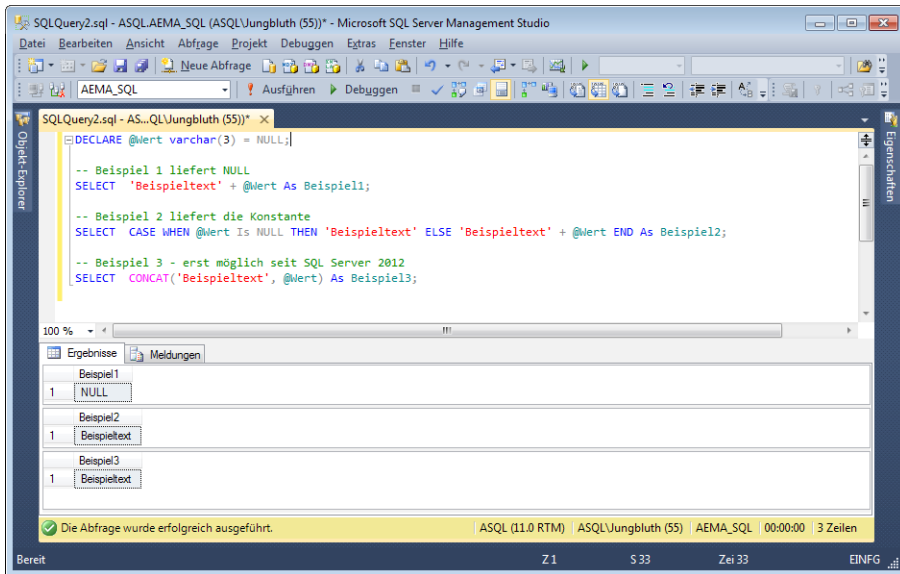


Abbildung 9.5: Verkettung von Zeichenfolgen mit *NULL*-Werten

Arbeiten Sie in einer SQL Server-Instanz mit der Standardsprache *Englisch*, müssen Sie entweder bei einer SQL-Anweisung mit Datumswerten das Datum formatieren oder aber Sie ändern das Datumsformat für das gesamte T-SQL-Skript. Letzteres übernimmt folgender T-SQL-Befehl:

```
SET DATEFORMAT dmy;
```

Welche möglichen Datumsformate Ihnen zur Verfügung stehen, sehen Sie nach Ausführen dieser Systemprozedur:

```
EXEC sp_helplanguage;
```

Zum Formatieren eines Datums in das entsprechende Format stehen Ihnen seit SQL Server 2012 zwei Möglichkeiten zur Verfügung: *CONVERT* und *FORMAT*. Bis SQL Server 2012 war eine Formatierung des Datums nur über eine Konvertierung des Werts mit dem T-SQL-Befehl *CONVERT* möglich. Folgende Anweisung konvertiert das Datum des aktuellen Zeitpunkts in das ISO-Format:

```
SELECT CONVERT(nvarchar(10), GETDATE(), 112) As Datum_ISO;
```

Die Art der Formatierung wird durch den Parameter *Style* festgelegt. In diesem Beispiel ist dies der Wert *112*. Um nur die Uhrzeit aus einem Datumswert zu ermitteln, geben Sie den Wert *108* ein.

```
SELECT CONVERT(nvarchar(8), GETDATE(), 108) As Uhrzeit;
```

Die verfügbaren Werte für den Parameter *Style* entnehmen Sie bitte der SQL Server-Hilfe unter dem Stichwort *CONVERT* und *CAST*.

Kapitel 9 T-SQL-Grundlagen

Mit *FORMAT* definieren Sie das Ausgabeformat ähnlich wie in VBA: Sie übergeben den Wert und das gewünschte Format. Ergänzend können Sie hier noch den Ländercode angeben. Dieser kann ausschlaggebend für das Ergebnis sein. So bleibt der Datumswert *06.07.2013* bei einer Formatierung mit dem Ländercode *de-de* weiterhin der 6. Juli 2013. Ändern Sie den Ländercode in *en-us*, wird das Datum als 7. Juni 2013 interpretiert (siehe Abbildung 9.6).

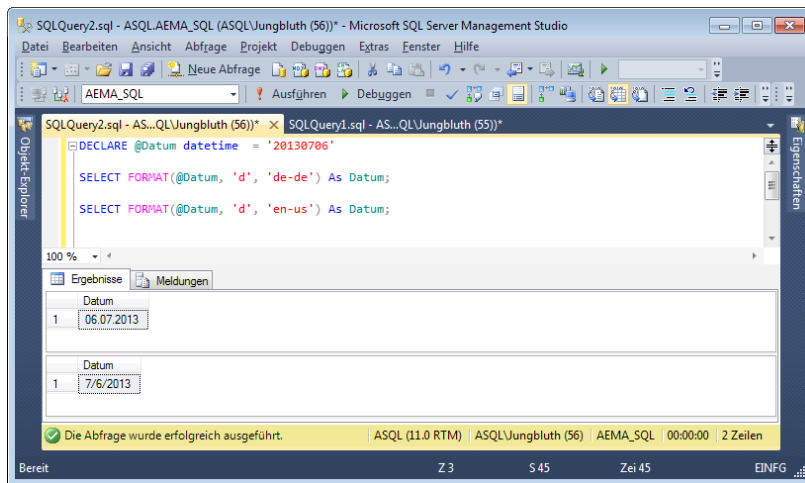


Abbildung 9.6: Formatierungen von Datumswerten

FORMAT und *CONVERT* formatieren Datumswerte nicht nur zur Ausgabe von Daten. Gerade in einer *WHERE*-Bedingung ist das korrekte Format des Datumswerts entscheidend für das Ergebnis der *SELECT*-Anweisung. Sie müssen also das Format des Datumswerts in der *WHERE*-Bedingung dem der Datenbank anpassen.

Es geht auch einfacher: Sie verwenden in *WHERE*-Bedingungen das ISO-Format. Mit dem ISO-Format sind Sie auf der sicheren Seite. Folgende *SELECT*-Anweisung wird immer korrekt als der 01.08.2012 interpretiert:

```
SELECT BestellungID, KundeID, Bestelldatum FROM dbo.tblBestellungen  
WHERE Bestelldatum = '20120801'
```

9.1.5 Das Semikolon

Anweisungen, die nicht Teil einer Struktur wie *IF...ELSE*, *BEGIN...END* et cetera sind, sollten Sie mit einem Semikolon abschließen. Dies führt zwar bei den meisten T-SQL-Befehlen nicht zu einem Fehler, ist jedoch SQL-Standard.

Bei einigen der neueren T-SQL-Befehle wie *MERGE* und bei der Verwendung einer *Common Table Expression* ist das Semikolon bereits Bestandteil der Syntax. Am besten gewöhnen Sie sich das Semikolon direkt an.

9.1.6 Code kommentieren

Wenn Sie Kommentare in T-SQL-Anweisungen einfügen möchten, können Sie dies auf zwei Arten erledigen:

- » Um nur eine Zeile als Kommentar zu kennzeichnen, stellen Sie dieser zwei Minuszeichen voran (--). Diese Art des Kommentars können Sie auch innerhalb einer Zeile nutzen, um zu verhindern, dass Bestandteile der SQL-Anweisung (beispielsweise die *WHERE*-Bedingung) ausgeführt werden.
- » Möchten Sie mehrere aufeinanderfolgende Zeichen als Kommentar kennzeichnen, fügen Sie vor der ersten Zeile die Zeichenfolge */** und hinter der letzten Zeile des Kommentars (am Ende der Zeile oder auch zu Beginn der folgenden Zeile) die Zeichenfolge **/* ein.

Beispiele:

```
-- Dies ist ein einzeiliger Kommentar.
SELECT * FROM dbo.tblArtikel -- WHERE WarengruppeID = 18;
/*
Dies ist ein
mehrzeiliger Kommentar.
*/
```

9.2 Variablen und Parameter

Genau wie jede andere Programmiersprache erlaubt auch T-SQL den Einsatz von Variablen. Im Vergleich etwa zu VBA sind Gültigkeitsbereich und Lebensdauer von Variablen sehr begrenzt:

Eine Variable ist maximal innerhalb der gespeicherten Prozedur, dem Trigger oder der Funktion gültig. Genau genommen kann dies noch weiter eingeschränkt werden – und zwar auf den sogenannten Batch. Darauf kommen wir später zu sprechen. Da der Gültigkeitsbereich einer Variablen derart eingeschränkt ist, nehmen wir in diesen Abschnitt gleich noch die Parameter hinzu. Parameter sind Variablen, die beim Aufruf einer gespeicherten Prozedur oder einer Funktion von der aufrufenden Instanz gefüllt werden können.

Auch hier gibt es eine Einschränkung gegenüber VBA: Einem Parameter können Sie unter T-SQL nur einen Wert zuweisen, aber keinen Verweis auf einen Wert. Das heißt, dass sich Änderungen an einem Parameter innerhalb einer gespeicherten Prozedur oder Funktion keinesfalls auf den Wert der Variablen in der aufrufenden Instanz auswirken.

9.2.1 Variablen deklarieren

Eine Variable muss zunächst deklariert werden. Dies geschieht mit der Anweisung *DECLARE*, die zwei Parameter erwartet: den mit führendem *@*-Symbol ausgestatteten Variablennamen und den Datentyp.

Kapitel 9 T-SQL-Grundlagen

Der Datentyp entspricht den beim Erstellen von Feldern verwendeten Datentypen, also beispielsweise *int* oder *nvarchar(255)*. Eine Laufvariable deklarieren Sie also etwa wie in folgendem Beispiel:

```
DECLARE @i int;
```

Sie können mit einer *DECLARE*-Anweisung gleich mehrere Variablen deklarieren:

```
DECLARE @i int, @j int;
```

Eine Variable zum Speichern einer Zeichenkette deklarieren Sie so:

```
DECLARE @Vorname nvarchar(255);
```

Die Namen von Variablen und Parametern dürfen keine Leerstellen oder Sonderzeichen enthalten – mit Ausnahme des Unterstrichs.

9.2.2 Variablen füllen

Eine Variable wird mit der *SET*-Anweisung gefüllt. Im Falle der soeben deklarierten Variablen *@i* sieht das so aus:

```
SET @i = 1;
```

Eine Textvariable füllen Sie, indem Sie den Text in Hochkommata einfassen. Anführungszeichen (") sind nicht zulässig!

```
SET @Vorname = 'André';
```

Sie können Variablen in neueren SQL Server-Versionen (ab SQL Server 2008) auch direkt bei der Deklaration mit einem Wert füllen:

```
DECLARE @k int = 10;
```

Möchten Sie mehrere Variablen initialisieren, verwenden Sie anstelle einzelner *SET*-Anweisungen einfach eine *SELECT*-Anweisung:

```
SELECT @i = 1, @Vorname = 'André';
```

Variable aus Abfrage füllen

Variablen lassen sich auch mit dem Ergebnis einer Abfrage füllen. Für die Abfrage gelten die gleichen Regeln wie für eine mit der *IN*-Klausel von SQL verwendete Abfrage: Sie muss in Klammern eingefasst werden und darf nur einen einzigen Datensatz mit einem einzigen Feld zurückliefern. Im folgenden Beispiel liefert die Abfrage den kleinsten Wert für das Feld *BankID* der Tabelle *tblBanken*:

```
DECLARE @BankID int;  
SET @BankID = (SELECT TOP 1 BankID FROM dbo.tblBanken ORDER BY BankID);  
SELECT @BankID;
```


Die *TOP 1*-Klausel sorgt dafür, dass die Abfrage nur einen Datensatz liefert, und da nur ein Feld als Ergebnismenge angegeben ist, gibt die Abfrage auch nur einen Wert zurück.

Variable in Abfrage füllen

Es gibt noch eine alternative Schreibweise, bei der die Variable direkt in die Abfrage integriert wird:

```
DECLARE @BankID int;
SELECT TOP 1 @BankID = BankID FROM dbo.tblBanken ORDER BY BankID;
SELECT @BankID;
```

Achten Sie darauf, dass die Abfrage nur einen Datensatz liefert. Enthält das Abfrageergebnis mehrere Datensätze, wird der Variablen der Wert des letzten Datensatzes zugewiesen. Ein anderer, viel interessanterer Aspekt bei dieser Variante ist, dass Sie mehrere Werte gleichzeitig in Variablen schreiben können. Die folgende Anweisungsfolge gibt die drei Werte des gefundenen Datensatzes zwar nur im Ergebnisfenster aus, aber natürlich können Sie diese Werte auch auf ganz andere Art und Weise weiterverarbeiten:

```
DECLARE @BankID int;
DECLARE @Bank varchar(255);
DECLARE @BLZ varchar(8);
SELECT TOP 1 @BankID = BankID, @Bank = Bank, @BLZ = BLZ FROM dbo.tblBanken ORDER BY
BankID;
SELECT @BankID, @Bank, @BLZ;
```

Variable mit Funktionswert füllen

T-SQL bietet eine ganze Reihe nützlicher eingebauter Funktionen – zum Beispiel zum Ermitteln des aktuellen Zeitpunkts oder des aktuell angemeldeten Benutzers.

Auch damit können Sie Variablen füllen – hier mit dem aktuellen Zeitpunkt:

```
DECLARE @AktuellesDatum datetime;
SET @AktuellesDatum = GETDATE();
SELECT @AktuellesDatum;
```

9.2.3 Werte von Variablen ausgeben

Während Sie mit T-SQL programmieren, möchten Sie vielleicht zwischenzeitlich den aktuellen Wert einer Variablen ausgeben lassen. Dies erledigen Sie mit der *SELECT*-Anweisung:

```
SELECT @i As i, @Vorname As Vorname;
```

Soll die Ausgabe lediglich eine Information sein, verwenden Sie anstelle *SELECT* die Ihnen bereits bekannte *PRINT*-Anweisung. Der Wert der Variablen sehen Sie dann in der Registerkarte *Meldungen*.

```
PRINT 'I: ' + CAST(@i As nvarchar(5)) + ' Vorname: ' + @Vorname;
```

9.2.4 Variablen ohne Wertzuweisung

Wenn Sie einer Variablen keinen Wert zuweisen, enthält diese den Wert *NULL*. Dies gilt, im Gegensatz zu VBA, für alle Datentypen. Folgende Anweisungen liefern also den Wert *NULL* zurück:

```
DECLARE @j int;  
SELECT @j;
```

9.2.5 Gültigkeitsbereich

Weiter oben haben wir bereits erwähnt, dass eine Variable maximal innerhalb einer gespeicherten Prozedur, eines Triggers oder einer Funktion gültig ist. Es gibt eine Einschränkung: Das Schlüsselwort *GO* löscht alle Variablen (siehe Abbildung 9.7).

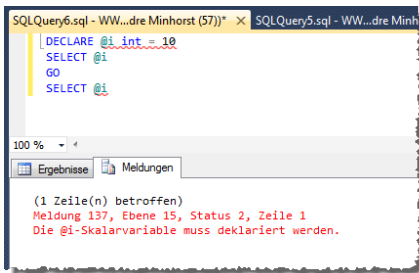


Abbildung 9.7: *GO* leert alle Variablen.

Das ist natürlich nicht der Sinn dieses Schlüsselworts. *GO* ist vielmehr dazu gedacht, den Code in verschiedene Batches zu unterteilen, die einzeln vom SQL Server abgearbeitet werden. Wozu braucht man das? Manche Anweisungen sind nur als erste Anweisung eines Batches zulässig. Dazu gehören die Anweisungen zum Erstellen von gespeicherten Prozeduren oder Sichten wie *CREATE PROC* oder *CREATE VIEW*.

9.2.6 Variablen einsetzen

Variablen können Sie beispielsweise in Auswahlabfragen einsetzen: Sie füllen die Variable mit einem Wert und setzen dann die Variable als Vergleichskriterium in einer *SELECT*-Abfrage ein:

```
DECLARE @BankID int;  
SET @BankID = 12;  
SELECT * FROM dbo.tblBanken WHERE BankID = @BankID;
```

9.2.7 Parameter

Parameter können Sie in gespeicherten Prozeduren oder in Funktionen einsetzen. Sie werden genau wie Variablen deklariert – mit dem Unterschied, dass die *DECLARE*-Anweisung entfällt.

Um den Kapiteln »Gespeicherte Prozeduren«, Seite 257, und »Funktionen«, Seite 273, vorzugreifen: Beim SQL Server legen Sie gespeicherte Prozeduren und Funktionen nicht einfach in einem Modul an, sondern Sie erstellen diese mit einer *CREATE*-Anweisung, ändern sie mit der *ALTER*-Anweisung und verwenden zum Löschen die *DROP*-Anweisung.

Übergabeparameter

Die Parameterliste wird in Klammern hinter dem Namen der gespeicherten Prozedur oder der Funktion angegeben. Das folgende Beispiel erstellt eine gespeicherte Prozedur. Die Parameterdefinition sehen Sie in der ersten Zeile:

```
CREATE PROC dbo.spBankNachID(@BankID int)
AS
SELECT * FROM tblBanken WHERE BankID = @BankID;
```

Um die gespeicherte Prozedur mit dem gewünschten Parameter auszuführen, verwenden Sie folgende Anweisung:

```
EXEC dbo.spBankNachID 210023;
```

Dies liefert die Werte aller Felder der Tabelle *tblBanken* für den per Parameter angegebenen Datensatz. Genau wie Variablen können Sie auch Parameter direkt bei der Deklaration vorbelegen:

```
CREATE PROC dbo.spAnrede(@AnredeID int = 1)
AS
SELECT Anrede FROM dbo.tblAnreden WHERE AnredeID = @AnredeID;
```

Rufen Sie die gespeicherte Prozedur ohne Parameter auf, liefert diese den Wert der Tabelle *tblAnreden* mit dem Wert *1* im Feld *AnredeID* zurück:

```
EXEC dbo.spAnrede;
```

Wenn Sie hingegen keinen Standardwert angeben und beim Aufruf keinen Parameter zuweisen, löst dies einen Fehler aus.

Rückgabeparameter

Gespeicherte Prozeduren können auch einzelne Werte zurückgeben. Diese werden in der Parameterliste mit dem Schlüsselwort *OUTPUT* hinter dem Datentyp gekennzeichnet – also beispielsweise wie folgt. Die Prozedur füllt den Parameter *@Rueckgabe* und gibt diesen zurück:

```
CREATE PROC dbo.spRueckgabewert(@Rueckgabe int OUTPUT)
AS
SET @Rueckgabe = 100;
```

Vor dem Aufruf der Prozedur deklarieren Sie eine Variable zum Aufnehmen des Rückgabewertes. Diese Variable übergeben Sie mit der Kennzeichnung *OUTPUT* an die gespeicherte Prozedur. Die *SELECT*-Anweisung gibt schließlich den zurückgegebenen Wert aus:

```
DECLARE @Ergebnis int;  
EXEC dbo.spRueckgabewert @Ergebnis OUTPUT;  
SELECT @Ergebnis;
```

9.3 Temporäre Tabellen

Manchmal möchten Sie vielleicht komplexere Datenstrukturen in Variablen zwischenspeichern – zum Beispiel einen oder mehrere komplette Datensätze. Hierzu bietet Ihnen SQL Server zwei Möglichkeiten: die temporären Tabellen und die *Table*-Variablen. Eine temporäre Tabelle erstellen Sie wie eine normale Tabelle – mit einem kleinen Unterschied: Der Name der temporären Tabelle beginnt mit dem Raute-Zeichen (#). Im folgenden Beispiel erstellen wir eine temporäre Tabelle mit den gleichen Feldern wie die Tabelle *tblAnreden*:

```
CREATE TABLE #tblAnreden(  
    AnredeID int, Anrede nvarchar(10)  
);
```

Diese Tabelle wird nicht in der Liste der Tabellen der Datenbank angezeigt. Wo aber erscheint sie dann? Bevor wir dies untersuchen, legen wir noch eine weitere Tabelle an – diesmal mit zwei Raute-Zeichen vor dem Tabellennamen. Dies bedeutet, dass das Objekt als globale temporäre Tabelle angelegt werden soll:

```
CREATE TABLE ##tblAnreden(  
    AnredeID int, Anrede nvarchar(10)  
);
```

Die beiden temporären Tabellen finden sich in der Systemdatenbank *tempdb* (siehe Abbildung 9.8). Die Bezeichnung der lokalen temporären Tabelle wird noch durch einige Unterstriche (_) und eine Nummer ergänzt. Diese Maßnahme ergreift der SQL Server, damit jede Verbindung eine eigene lokale temporäre Tabelle dieses Namens anlegen kann. Sie können so beispielsweise eine solche Tabelle direkt im SQL Server Management Studio erzeugen und eine weitere in einer Verbindung, die Sie von einer Access-Datenbank aus erstellt haben. Der Gültigkeitsbereich dieser beiden Tabellenarten lässt sich leicht experimentell prüfen. Führen Sie einmal im SQL Server Management Studio einen Datensatz zur globalen temporären Tabelle hinzu:

```
INSERT INTO ##tblAnreden(AnredeID, Anrede) VALUES(3,N'Firma');
```

Lesen Sie dann den Inhalt dieser temporären Tabelle in Access mittels einer Pass-Through-Abfrage mit folgender Anweisung aus:

```
SELECT * FROM ##tblAnreden;
```

Das Ergebnis ist der zuvor im SQL Server Management Studio angelegte Datensatz. Wenn Sie die gleichen Schritte für die temporäre lokale Tabelle durchführen, finden Sie die angelegten Datensätze nur in der jeweils gleichen Verbindung vor.

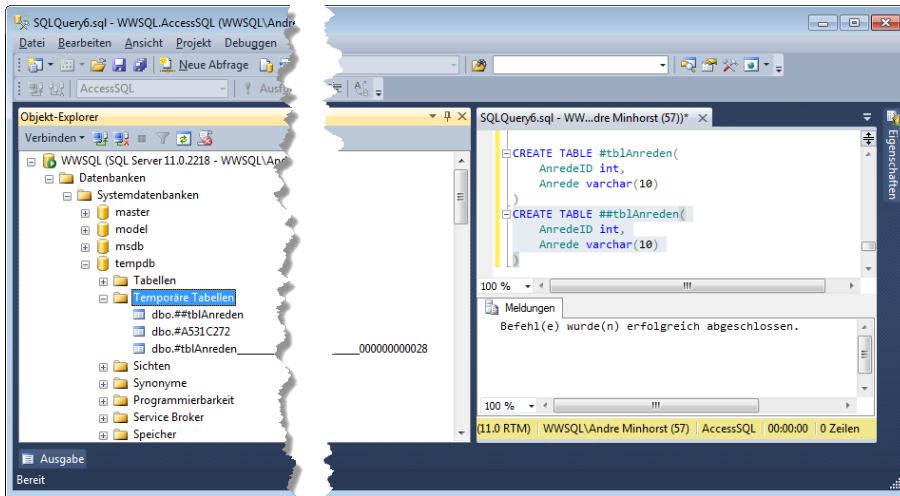


Abbildung 9.8: Speicherort für lokale und globale temporäre Tabellen

Temporäre Tabelle auf Basis bestehender Tabelle

Sie können eine temporäre Tabelle auch komfortabel mit einer *SELECT...INTO*-Anweisung erstellen. Dabei kopieren Sie einen oder mehrere Datensätze der Quelltable in die temporäre Zieltabelle. Dies sieht wie folgt aus:

```
SELECT * INTO #tblKunden FROM dbo.tblKunden;
```

Mit der *SELECT...INTO*-Anweisung ist es auch möglich, Teile einer Tabelle oder Ergebnisse von *SELECT*-Anweisungen mit Tabellenverknüpfungen zwischenzuspeichern.

```
SELECT Kundenbezeichnung, Liefer_Ort
INTO #Kunden
FROM dbo.tblBestellungen B INNER JOIN dbo.tblKundenBase K ON B.KundeID = K.KundeID
WHERE Liefer_Ort = 'Duisburg';
```

Letztendlich ist das Ergebnis ähnlich dem einer Tabellenerstellungsabfrage in Access. Spalten und Inhalt der Zieltabelle entsprechen der *SELECT*-Anweisung; Definitionen von Primärschlüssel, Einschränkungen und anderen Eigenschaften der Quelltable werden nicht übernommen.

9.4 Table-Variablen

Die *Table-Variable* ist der temporären Tabelle recht ähnlich, wird aber nicht als Tabelle in der Datenbank *tempdb* angelegt. Vielmehr handelt es sich um eine Variable, die einen Satz von Variablen beziehungsweise Feldern beliebigen Datentyps aufnehmen kann. Dies ist vergleichbar

Kapitel 9 T-SQL-Grundlagen

mit dem Type-Konstrukt aus VBA – nur viel mächtiger, da mit fast allen Möglichkeiten einer Tabelle ausgestattet. Die Deklaration einer solchen Table-Variable sieht fast genauso aus wie die Erstellung einer herkömmlichen oder einer temporären Tabelle:

```
DECLARE @Tablevariable table(  
    AnredeID int,  
    Anrede varchar(255)  
);
```

Sie füllen die *Table-Variable* dann wie eine herkömmliche Tabelle – beispielsweise mit den Datensätzen der Tabelle *tblAnreden*:

```
INSERT INTO @Tablevariable SELECT * FROM dbo.tblAnreden;
```

Schließlich greifen Sie wie gewohnt auf die Datensätze zu:

```
SELECT * FROM @Tablevariable;
```

Im Gegensatz zur temporären Tabelle lässt sich eine *Table-Variable* nicht direkt aus einer anderen Tabelle per *SELECT...INTO* erstellen. Außerdem ist im Vergleich zur temporären Tabelle der Gültigkeitsbereich auf die aktuelle gespeicherte Prozedur, die Funktion oder den Trigger begrenzt. Seit SQL Server 2008 ist es jedoch möglich, eine *Table-Variable* über einen *Tabellenwertparameter* (im Original *Table Valued Parameter*) an weitere Routinen zu übergeben.

Sollten Sie eine *Table-Variable* mit anderen Tabellen verknüpfen wollen, müssen Sie die *Table-Variable* mit einem Alias versehen:

```
SELECT tblKunden.Vorname, tblKunden.Nachname, t1.Anrede  
FROM dbo.tblKunden INNER JOIN @TableAnreden AS t1 ON dbo.tblKunden.AnredeID =  
t1.AnredeID;
```

9.5 Ablaufsteuerung

Vorgänge, die Sie unter VBA mit Strukturen wie *If...Then*, *Select Case*, *Do While*, *For...Next* erledigen, können Sie größtenteils auch unter T-SQL abbilden – wenn auch teilweise unter erschwerten Bedingungen, da der Befehlsumfang etwas geringer ist. So müssen Sie eine *For...Next*-Schleife mit einem *WHILE*-Konstrukt mit Laufvariable nachbilden.

Die folgenden Abschnitte zeigen die diesbezüglichen Möglichkeiten unter T-SQL.

9.5.1 IF...ELSE

Das *IF...ELSE*-Konstrukt unter T-SQL bildet einen Teil der von VBA bekannten *If...Then...Else...Elseif*-Struktur ab. Der Unterschied ist, dass es zwar einen *ELSE*-Teil gibt, aber kein *ELSEIF*. Sie können also mit der Haupt-*IF...ELSE*-Anweisung nur einen Ausdruck auf die Werte *TRUE* oder *FALSE* prüfen. Ein einfaches Beispiel sieht wie folgt aus:

```
DECLARE @Zahl int;
SET @Zahl = 11;
IF @Zahl < 10
    PRINT 'Zahl ist kleiner als 10';
ELSE
    PRINT 'Zahl ist größer gleich 10';
```

Wenn Sie mehr als die beiden Werte *True* oder *False* prüfen möchten, müssen Sie weitere *IF...ELSE*-Strukturen einbetten:

```
DECLARE @Zahl int;
SET @Zahl = 11;
IF @Zahl < 10
    PRINT 'Zahl ist kleiner als 10';
ELSE
    IF @Zahl = 10
        PRINT 'Zahl ist gleich 10';
    ELSE
        PRINT 'Zahl ist größer als 10';
```

In den Vergleichsausdrücken des *IF...ELSE*-Konstrukts können Sie wiederum alle möglichen Ausdrücke einsetzen – beispielsweise das Ergebnis einer Abfrage. Diese darf jedoch wiederum nur einen Wert zurückgeben. Folgendes Beispiel prüft, ob das Feld *Anrede* in der Tabelle *tblAnreden* mit dem Wert *1* im Feld *AnredeID* den Wert *Herr* enthält:

```
IF (SELECT Anrede FROM dbo.tblAnreden WHERE AnredeID = 1) = 'Herr'
    PRINT 'Alles in Ordnung.';
ELSE
    PRINT 'Ups!';
```

Achten Sie aber darauf, dass die Abfrage im Vergleichsausdruck keinen Fehler liefert. In einem solchen Fall ist das Ergebnis des Vergleichsausdrucks *FALSE*, wodurch der *ELSE*-Part der *IF*-Anweisung ausgeführt wird. Um diese zu vermeiden, prüfen Sie im *ELSE*-Part zunächst die Systemvariable *@@ERROR*. Nur wenn diese den Wert 0 enthält, soll der *ELSE*-Part ausgeführt werden. Auf die Systemvariable *@@ERROR* kommen wir später nochmal zurück.

```
IF (SELECT Anrede FROM dbo.tblAnreden WHERE AnredeID = 1) = 'Herr'
    PRINT 'Alles in Ordnung.';
ELSE
    IF @@ERROR = 0
        PRINT 'Ups!';
```

9.5.2 BEGIN...END

Möchten Sie innerhalb eines Konstruktes wie *IF...ELSE* oder auch den nachfolgend vorgestellten Anweisungen mehr als einen Befehl unterbringen, müssen Sie diese in die Anweisungen *BEGIN* und *END* einfassen.

```
IF (SELECT Anrede FROM tblAnreden WHERE AnredeID = 1) = 'Herr'
BEGIN
```

Kapitel 9 T-SQL-Grundlagen

```
        PRINT 'Alles in Ordnung.';
        PRINT 'Hoffentlich.';
END
PRINT 'Egal wie, ab hier geht es weiter';
```

9.5.3 CASE und IIF

Wenn Sie einen Wert in Abhängigkeit des Ergebnisses verschiedener Ausdrücke ermitteln möchten, verwenden Sie die *CASE*-Bedingung. Im Gegensatz zu *IF...ELSE* können Sie hier auch mehr als einen Vergleichsausdruck angeben. Dafür liefert *CASE* nur einen korrespondierenden Wert zurück und erlaubt nicht je nach Bedingung weitere Anweisungen auszuführen. *CASE* können Sie auch in Abfragen einsetzen – mehr dazu im Kapitel »Auswahl- und Aktionsabfragen mit T-SQL«, Seite 225. Im folgenden Beispiel prüft *CASE*, ob der in *@i* gespeicherte Wert entweder 1, 2 oder einem anderen Wert entspricht, und weist der Variablen *@Ergebnis* den Ausdruck *Eins*, *Zwei* oder *Andere Zahl* zu:

```
DECLARE @i int, @Ergebnis varchar(255);
SET @i = 3;
SET @Ergebnis = CASE
    WHEN @i = 1 THEN 'Eins'
    WHEN @i = 2 THEN 'Zwei'
    ELSE 'Andere Zahl'
END
PRINT @Ergebnis;
```

Seit SQL Server 2012 gibt es mit *IIF* noch einen weiteren T-SQL-Befehl für eine Fallunterscheidung. Die Syntax in T-SQL ist die gleiche wie in Access, wobei auch hier mehrere *IIF*-Anweisungen verschachtelt werden können:

```
DECLARE @i int, @Ergebnis varchar(255);
SET @i = 3;
SET @Ergebnis = IIF(@i = 1, 'Eins', IIF(@i = 2, 'Zwei', 'Andere Zahl'))
PRINT @Ergebnis;
```

9.5.4 WHILE

Unter T-SQL ist *WHILE* die einzige Möglichkeit, eine Schleife zu programmieren. Die *WHILE*-Schleife fragt einen im Kopf angegebenen Ausdruck ab. Liefert dieser den Wert *TRUE*, werden die Befehle der *WHILE*-Schleife ausgeführt, ansonsten die erste Anweisung nach der *WHILE*-Schleife.

For...Next nachbilden

Mithilfe einer Laufvariablen können Sie mit *WHILE* einen *For...Next*-Ersatz programmieren. Das sieht beispielsweise wie folgt aus, wenn Sie die Zahlen von 1 bis 10 ausgeben möchten:

```
DECLARE @i int;
SET @i = 0;
```



```
WHILE @i<10
BEGIN
    SET @i = @i + 1;
    PRINT @i;
END
```

Vergessen Sie nicht, die Variable *@i* mit dem Wert *0* zu initialisieren. Anderenfalls würde *@i* den Wert *NULL* enthalten und die Bedingung *@i<10* den Wert *False* liefern. Ein praktischer Einsatzzweck ist das Erstellen einiger Beispieldaten. Die folgenden Anweisungen fügen beispielsweise zehn Datensätze zur Tabelle *tblKunden* hinzu:

```
DECLARE @i int;
SET @i = 0;
WHILE @i < 10
BEGIN
    SET @i = @i + 1;
    INSERT INTO dbo.tblKunden(Vorname, Nachname) VALUES('Vorname' + CAST(@i AS
        varchar(2)), 'Nachname' + CAST(@i AS varchar(2)));
END
```

Sie müssen keine Zählervariable in der Abbruchbedingung der Schleife verwenden, sondern können jeden anderen Ausdruck einsetzen. Im folgenden Beispiel fügt die *INSERT*-Anweisung so lange Datensätze zur Tabelle *tblKunden* hinzu, bis diese 20 Datensätze enthält. Den notwendigen Vergleichswert liefert eine *SELECT*-Unterabfrage:

```
DECLARE @i int;
SET @i = 0;
WHILE (SELECT COUNT(*) FROM dbo.tblKunden) < 20
BEGIN
    SET @i = @i + 1;
    INSERT INTO dbo.tblKunden(Vorname, Nachname) VALUES('Vorname' + CAST(@i AS
        varchar(2)), 'Nachname' + CAST(@i AS varchar(2)));
END
```

Inkrementieren leicht gemacht

Wenn Sie innerhalb einer Schleife jeweils den gleichen Zahlenwert zum Wert einer Variablen addieren oder subtrahieren möchten, können Sie dies auf die klassische Weise tun:

```
SET @i = @i + 1;
SET @j = @j - 1;
```

Sie können ab SQL Server 2008 jedoch auch folgende Schreibweise verwenden:

```
SET @j+=1;
SET @j-=1;
```

Folgendes gibt also beispielsweise den Wert *12* aus:

```
DECLARE @i int = 10;
SET @i+=1;
```

```
SET @i+=1;  
SELECT @i;
```

Schleifendurchlauf abbrechen

Mit dem Schlüsselwort *BREAK* können Sie eine Schleife abbrechen, bevor die Abbruchbedingung im Kopf der Schleife eintritt. *BREAK* fügen Sie an beliebiger Stelle innerhalb der zu durchlaufenden Befehle ein. Beim Erreichen von *BREAK* wird die Schleifenverarbeitung direkt abgebrochen und eventuell nachfolgende SQL-Anweisungen nicht mehr ausgeführt. Typischerweise werden Sie die *BREAK*-Anweisung innerhalb einer *IF...ELSE*-Bedingung verwenden, damit die Schleife nur bei bestimmten Ereignissen beendet wird. Technisch können Sie mit dieser Anweisung etwa eine Schleife programmieren, die auf jeden Fall mindestens einmal durchlaufen werden soll. In diesem Fall würden Sie die Abbruchbedingung im Kopf der Schleife so formulieren, dass diese immer den Wert *TRUE* liefert und die eigentliche Abbruchbedingung im Fuß der Bedingung unterbringen. Dies sieht dann beispielsweise wie folgt aus:

```
DECLARE @i int;  
SET @i=0;  
WHILE 1=1  
BEGIN  
    SET @i+=1;  
    PRINT @i;  
    IF @i = 10  
        BREAK;  
END
```

Schleifendurchlauf neu starten

Wie bei *BREAK* wird auch bei *CONTINUE* die Schleifenverarbeitung unterbrochen und die nachfolgenden SQL-Anweisungen der Schleife nicht mehr ausgeführt. Nur mit dem Unterschied, dass *CONTINUE* die Schleifenverarbeitung mit der ersten SQL-Anweisung wieder aufnimmt. Im folgenden Beispiel wird der Zähler *@i* bis 20 hochgezählt. Die *PRINT*-Ausgabe erfolgt jedoch erst ab dem zehnten Durchlauf.

```
DECLARE @i int = 10  
SET @i=0;  
WHILE @i < 20  
BEGIN  
    SET @i+=1;  
    IF @i < 10  
        CONTINUE;  
    PRINT @i;  
END
```

9.6 Batches

Ein Batch ist eine Abfolge von einem oder mehreren T-SQL-Befehlen, die in einem Rutsch über die aktuelle Verbindung zur Ausführung zum SQL Server geschickt werden.

9.6.1 Batchende mit GO markieren

Ein Batch von Anweisungen wird durch das *GO*-Schlüsselwort beendet. *GO* ist kein SQL-Befehl, sondern nur eine Kennzeichnung für das Ende eines Batches. Sie können auch ein beliebiges anderes Wort definieren.

Diese Einstellung nehmen Sie in den Optionen des SQL Server Management Studios vor. Wählen Sie dort den Menübefehl *Extras/Optionen* aus, um den Dialog aus Abbildung 9.9 zu öffnen. Hier finden Sie die Option *Batchtrennzeichen*, die Sie selbst anpassen können.

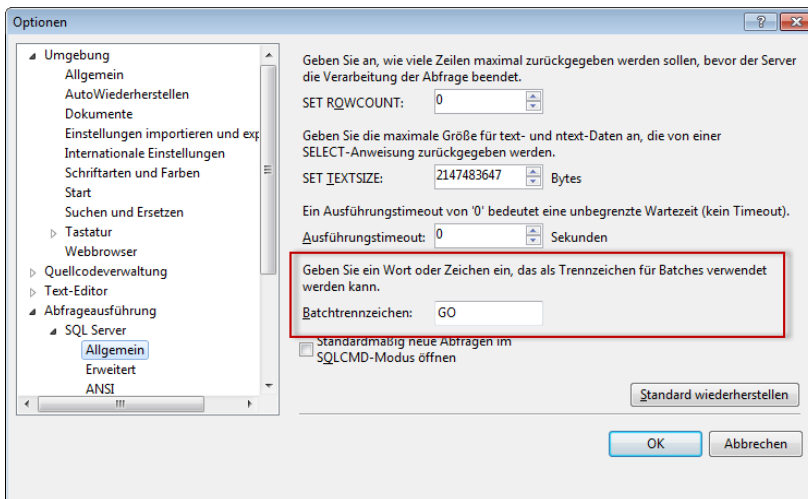


Abbildung 9.9: Option zum Festlegen eines Batchtrennzeichens

Bezüglich eines Batches müssen Sie Folgendes beachten:

- » Lokale Variablen gelten nur innerhalb eines Batches, sie werden also beim Erreichen des Batch-Endes beziehungsweise des *GO*-Schlüsselwortes gelöscht.
- » Manche Anweisungen dürfen nur die erste Position im Batch einnehmen – beispielsweise die *CREATE PROC*-Anweisung.

9.6.2 Batch mehrfach ausführen

Auch wenn das *GO*-Schlüsselwort kein T-SQL-Befehl, sondern ein austauschbares Schlüsselwort ist, so verfügt es dennoch über einen Parameter. Mit einer Zahl können Sie angeben, wie oft der vorherige Batch ausgeführt werden soll. Wenn Sie also 10 gleiche Datensätze in eine Tabelle schreiben möchten, gelingt dies am schnellsten wie folgt:

```
INSERT INTO dbo.tblKunden(Vorname, Nachname) VALUES('Vorname', 'Nachname');
GO 10
```

9.6.3 Aktuellen Batch beenden

Einen Batch beenden Sie vorzeitig mit der *RETURN*-Anweisung. In den meisten Fällen werden Sie diese abhängig von einer Bedingung, also in einer *IF*-Anweisung, verwenden. Die gleiche Anweisung verwenden Sie, wenn Sie eine gespeicherte Prozedur verlassen und einen Statuscode zurückgeben möchten. Mehr dazu erfahren Sie in »Der RETURN-Wert einer gespeicherten Prozedur«, Seite 267.

9.6.4 WAITFOR

Die *WAITFOR*-Anweisung ist geeignet, um die Ausführung des aktuellen Batches um eine bestimmte Zeit zu unterbrechen oder anzugeben, um welche Uhrzeit diese fortgesetzt werden soll. Möchten Sie also etwa eine spezielle Aktion erst in fünf Sekunden durchführen, stellen Sie dieser die folgende Anweisung voran:

```
WAITFOR DELAY '0:00:05';  
PRINT 'Fünf Sekunden sind vorüber.'
```

Wenn Sie die folgende Anweisung erst zu einer bestimmten Uhrzeit ausführen lassen wollen, verwenden Sie diese Variante:

```
WAITFOR TIME '13:23:30';  
PRINT 'Es ist jetzt 13:23:30 Uhr.'
```

9.7 CURSOR

Als Access-Entwickler kennt man das folgende Beispiel, mit dem sich die Datensätze eines Recordsets durchlaufen lassen:

```
Dim db As DAO.Database  
Dim rst As DAO.Recordset  
Set db = Currentdb  
Set rst = db.OpenRecordset("SELECT * FROM tblKunden", dbOpenDynaset)  
Do While Not rst.EOF  
    'etwas mit den Feldern des Recordsets machen  
    rst.MoveNext  
Loop
```

Falls Sie gern mit solchen Codeschnipseln gearbeitet haben, gibt es gute Nachrichten: Auch unter T-SQL können Sie die Ergebnisse von Abfragen in ähnlicher Weise durchlaufen.

9.7.1 Nachteil Performance

Die schlechte Nachricht ist: Genau wie das Durchlaufen eines Recordsets unter VBA ist auch der Einsatz eines Cursors nicht die performanteste Möglichkeit, um eine Aktion auf Basis der

betroffenen Daten durchzuführen. Die gute Nachricht ist: In den meisten Fällen lässt sich das Durchlaufen der Recordsets unter VBA genau wie unter T-SQL verhindern und durch entsprechende SQL-Abfragen ersetzen. Unter VBA gibt es nur wenige echte Einsatzzwecke für *Do While*-Schleifen in Zusammenhang mit Recordsets – zum Beispiel das Füllen von Steuerelementen wie dem *TreeView*- oder dem *ListView*-Steuerelement. Unter T-SQL sollte es ebenso wenig sinnvolle Einsatzmöglichkeiten geben, daher versuchen Sie immer, den Einsatz eines Cursors zu vermeiden.

9.7.2 Einsatz eines Cursors

Dennoch wollen wir kurz beschreiben, was Sie mit einem Cursor anstellen können. Für einen *CURSOR* definieren Sie zunächst eine Variable und legen dann mit dem *FOR*-Schlüsselwort fest, welche Daten der Cursor enthalten soll. Diese ermitteln Sie mit einer entsprechenden *SELECT*-Auswahlabfrage über eine oder mehrere Tabellen.

Nach dem Zuweisen der Datenherkunft öffnen Sie den Cursor mit der *OPEN*-Methode und greifen dann mit verschiedenen Befehlen auf die enthaltenen Datensätze zu – beispielsweise zum Durchlaufen der einzelnen Datensätze des Recordsets in einer Schleife oder durch den gezielten Zugriff etwa auf das erste, letzte oder ein anderes Element.

Der Zugriff erfolgt über die *FETCH*-Anweisung, die letztlich eine Ergebnismenge wie eine *SELECT*-Anweisung erzeugt. Sie kann jedoch noch mehr – nämlich die in den Feldern enthaltenen Daten in Variablen speichern und die Werte somit für die weitere Verarbeitung zur Verfügung stellen. Die folgende Befehlsfolge entspricht einer sehr einfachen Anwendung eines Cursors:

```
DECLARE curKunden SCROLL CURSOR
FOR SELECT * FROM dbo.tb1Kunden;
OPEN curKunden;
FETCH ABSOLUTE 1 FROM curKunden;
CLOSE curKunden;
DEALLOCATE curKunden;
```

Die erste Anweisung deklariert den Cursor mit der Bezeichnung *curKunden* unter Verwendung des Schlüsselworts *SCROLL* (dies ist Voraussetzung für den nachfolgend verwendeten absoluten Zugriff auf einen der Datensätze). An dieser Stelle der erste wichtige Hinweis: Die für einen Cursor verwendete Variable darf nicht mit dem @-Zeichen beginnen!

Die *FOR*-Zeile leitet die *SELECT*-Anweisung ein, welche die Daten für den Cursor liefert. Erst die *OPEN*-Zeile füllt den Cursor mit den angegebenen Daten. Schließlich gibt die *FETCH*-Zeile mit dem Parameter *ABSOLUTE 1* den ersten Datensatz des Cursors *curKunden* aus (siehe Abbildung 9.10). Zur Freigabe des Speichers und des Cursornamens für eine eventuelle Neuerstellung unter gleichem Namen verwenden Sie die *DEALLOCATE*-Methode mit dem Namen des Cursors als Parameter. Einen Cursor braucht man schon selten, aber ganz bestimmt nicht für den im vorherigen Beispiel verwendeten Zweck – nämlich um Daten zu liefern. Meist verwenden Sie einen Cursor, um dessen Daten in Variablen zu schreiben und etwas mit diesen Variablen zu erledigen.

Kapitel 9 T-SQL-Grundlagen

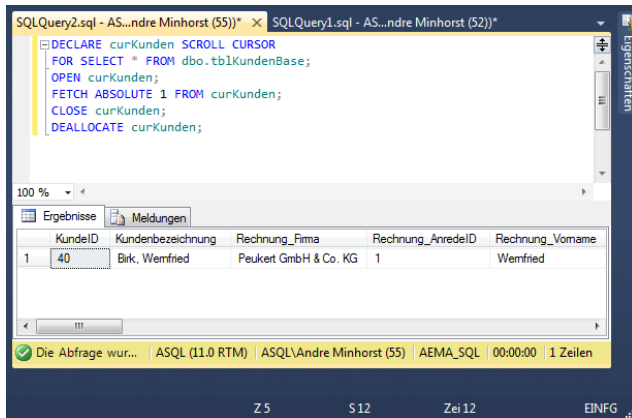


Abbildung 9.10: Ergebnis einer FETCH-Anweisung

9.7.3 Cursor per Schleife durchlaufen

Im zweiten Beispiel durchlaufen wir alle Datensätze der Tabelle *tblAnreden* in einer Schleife, schreiben den Wert des Felds *Anrede* in eine Variable und geben deren Inhalt im Meldungsfenster aus – stellvertretend für eine alternative Aktion, die den Einsatz eines Cursors erfordert:

```
DECLARE @Anrede varchar(255);
DECLARE curAnreden CURSOR
FOR SELECT Anrede FROM dbo.tblAnreden;
OPEN curAnreden;
FETCH NEXT FROM curAnreden INTO @Anrede;
WHILE @@FETCH_STATUS = 0
BEGIN
    PRINT @Anrede
    FETCH NEXT FROM curAnreden INTO @Anrede;
END
CLOSE curAnreden;
DEALLOCATE curAnreden;
```

Die erste Zeile deklariert zunächst eine Variable zum Speichern der Anrede (*@Anrede*), die zweite den Cursor. Diesmal verwenden wir nicht das Schlüsselwort *SCROLL* – wir wollen ja mit der Anweisung *FETCH NEXT* vorwärts durch die Datensätze navigieren. Der Rest der Anweisung gibt das Feld *Anrede* der Tabelle *tblAnreden* als Inhalt des Cursors an. Nach dem Öffnen des Cursors ermittelt die Anweisung *FETCH NEXT* den ersten Datensatz und schreibt seinen Wert in die Variable *@Anrede*. Hierbei ist wichtig, dass Anzahl, Typ und Reihenfolge der hinter dem Schlüsselwort *INTO* angegebenen Variablen genau mit den Feldern des Cursors übereinstimmen müssen.

Wir wissen nun noch nicht, ob der Zeiger auf einen Datensatz des Cursors zeigt. Dies ermitteln wir mit der Variablen *@@FETCH_STATUS*, die folgende Werte annehmen kann:

- » 0: Datensatz gefunden
- » -1: Kein Datensatz gefunden oder Zeiger vor oder hinter dem ersten beziehungsweise letzten Element
- » -2: Datensatz nicht gefunden

Den `@@FETCH_STATUS` prüfen wir direkt als Bedingung einer *WHILE*-Schleife. Wurde der Zeiger auf einem Datensatz platziert (`@@FETCH_STATUS = 0`), ist die Bedingung erfüllt. In diesem Fall kann die Ausführung der Anweisungen innerhalb der Schleife beginnen. Die Variable `@Anrede` ist gefüllt und wird per *PRINT* ausgegeben.

Noch innerhalb dieses Schleifendurchlaufs verschiebt *FETCH NEXT* den Datensatzzeiger auf das nächste Element und schreibt den Inhalt des Feldes *Anrede* in die Variable `@Anrede`. Die Schleife prüft mit `@@FETCH_STATUS = 0`, ob erneut ein Datensatz gefunden wurde, und führt gegebenenfalls nochmals die zwischen *BEGIN* und *END* enthaltenen Anweisungen aus. Sollte `@@FETCH_STATUS` einen anderen Wert als 0 enthalten, wurden alle Datensätze durchlaufen und die Schleife kann verlassen werden. Das Ergebnis sehen Sie in Abbildung 9.11.

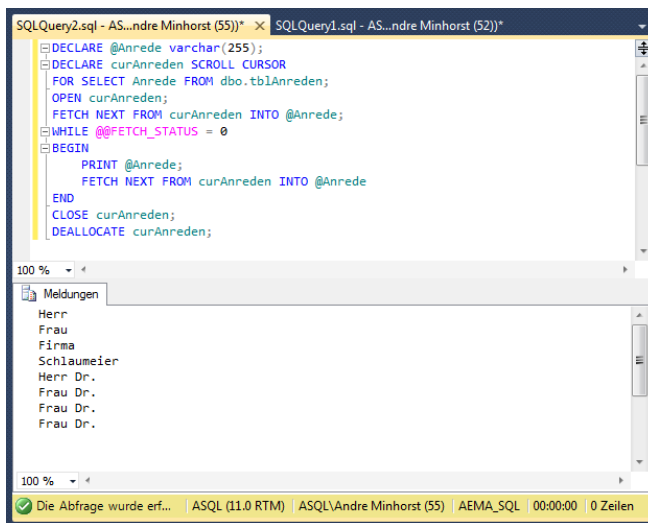


Abbildung 9.11: Fetch mit Schleife

9.7.4 Weitere Sprungpunkte

Wenn Sie den Cursor mit dem Schlüsselwort *SCROLL* erstellen, können Sie mit *FETCH* noch weitere Schlüsselwörter verwenden, um verschiedene Positionen anzuspringen:

- » *FETCH LAST*: Zeiger auf letzten Datensatz verschieben
- » *FETCH PRIOR*: Zeiger auf den vorherigen Datensatz verschieben (wie *FETCH RELATIVE -1*)

- » **FETCH ABSOLUTE <Zahl>**: Zeiger auf den Datensatz mit der mit <Zahl> angegebenen absoluten Position verschieben, wobei <Zahl> nur positive Werte größer 1 annehmen darf
- » **FETCH RELATIVE <Zahl>**: Zeiger um die in <Zahl> angegebene Anzahl Datensätze nach oben oder unten verschieben, wobei <Zahl> positive und negative Werte annehmen darf

Das folgende Beispiel zeigt die Verwendung der Schlüsselwörter und gibt jeweils den Wert des Feldes *KundeID* aus:

```
DECLARE @KundeID int;
DECLARE curKunden SCROLL CURSOR
FOR SELECT KundeID FROM dbo.tblKunden;
OPEN curKunden;
FETCH ABSOLUTE 1 FROM curKunden INTO @KundeID;
PRINT @KundeID;
FETCH RELATIVE 4 FROM curKunden INTO @KundeID;
PRINT @KundeID;
FETCH RELATIVE -2 FROM curKunden INTO @KundeID;
PRINT @KundeID;
FETCH LAST FROM curKunden INTO @KundeID;
PRINT @KundeID;
FETCH PRIOR FROM curKunden INTO @KundeID;
PRINT @KundeID;
DEALLOCATE curKunden;
```

9.8 Datenmanipulation

T-SQL bietet die üblichen Befehle zur Datenmanipulation, die Sie eventuell bereits vom Umgang mit Access-Abfragen kennen. Es gibt jedoch auch ein paar zusätzliche Befehle – hier sind die wichtigsten Informationen dazu.

9.8.1 Datensätze einfügen mit INSERT INTO

Mit der **INSERT INTO**-Anweisung fügen Sie Daten an eine bestehende Tabelle an. Diese Tabelle wird gleich als erster Parameter hinter der **INSERT INTO**-Anweisung angegeben. Dahinter folgt ein Ausdruck, der die einzufügenden Daten beschreibt:

```
INSERT INTO <Tabelle> <Anzufügende Daten>
```

Als <Tabelle> geben Sie den Namen der Tabelle an, an die die Daten angefügt werden sollen. Für den Ausdruck <Anzufügende Daten> gibt es zwei Möglichkeiten:

- » Angabe einer **SELECT**-Anweisung
- » Direkte Angabe der Felder und der einzufügenden Werte

Im Gegensatz zu Access-SQL erlaubt T-SQL den Einsatz von Unterabfragen in **INSERT INTO**-Statements. Das gilt auch für die übrigen Aktionsabfragen.

Anzufügende Daten per SELECT-Anweisung angeben

Wenn Sie eine *SELECT*-Anweisung als Datenherkunft für das Anfügen von Daten an eine Tabelle verwenden, müssen Sie darauf achten, dass die Datentypen der anzufügenden Felder mit den Zielfeldern übereinstimmen und dass alle Felder, die eine Eingabe erfordern, auch gefüllt werden. Die folgende Abfrage kopiert alle Datensätze der Tabelle *tblBestellungen* in die Tabelle *tblBestellungenArchiv*, deren Bestelldatum vor dem angegebenen Datum liegt:

```
INSERT INTO dbo.tblBestellungenArchiv(BestellungID, KundeID, Bestelldatum, Bemerkungen,
RechnungAm, Rechnungsdatei, Zahlungsziel, BezahlAm, UmsatzID, RechnungsversandartID,
StorniertAm)
SELECT BestellungID, KundeID, Bestelldatum, Bemerkungen, RechnungAm, Rechnungsdatei,
Zahlungsziel, BezahlAm, UmsatzID, RechnungsversandartID, StorniertAm
FROM dbo.tblBestellungen
WHERE Bestelldatum < '20120801';
```

Eine kürzere Fassung dieses SQL-Ausdrucks wäre folgende:

```
INSERT INTO dbo.tblBestellungenArchiv
SELECT BestellungID, KundeID, Bestelldatum, Bemerkungen, RechnungAm, Rechnungsdatei,
Zahlungsziel, BezahlAm, UmsatzID, RechnungsversandartID, StorniertAm
FROM dbo.tblBestellungen
WHERE Bestelldatum < '20120801';
```

Dabei müssen die Reihenfolge der Felder der Definition der Tabelle entsprechen und es müssen alle Felder gefüllt werden. Und nun die kürzeste Version, bei der die Anzahl und Reihenfolge zwischen Quell- und Zieltabelle exakt übereinstimmen müssen:

```
INSERT INTO dbo.tblBestellungenArchiv
SELECT dbo.tblBestellungen.*
FROM dbo.tblBestellungen
WHERE Bestelldatum < '20120801';
```

Anzufügende Daten direkt angeben

Die erste und längste Variante des oben genannten Beispiels ist auch Grundlage für das Anfügen eines Datensatzes, dessen Daten nicht aus einer Tabelle ermittelt werden:

```
INSERT INTO dbo.tblBestellungen(KundeID, Bestelldatum, Bemerkungen, RechnungAm,
Rechnungsdatei, Zahlungsziel, BezahlAm, UmsatzID, RechnungsversandartID)
VALUES (74, '20130622', 'Per INSERT angelegte Bestellung', '20130622', 'C:\Daten\
Buchprojekte\Test.pdf', '20130722', '20130715', 737, 2);
```

Die einzufügenden Werte geben Sie in diesem Fall direkt in der *VALUES*-Liste an. Beachten Sie, dass Sie genau die Reihenfolge einhalten, die durch die Feldliste im *INSERT INTO*-Abschnitt angegeben ist, und dass die Syntax für die Datentypen korrekt ist: Texte in Hochkommata, Datumswerte als 'yyyymmdd' et cetera. In diesem Fall ist *BestellungID* ein Feld mit Autowert (*IDENTITY*). Dies bedeutet, dass der SQL Server diesen Wert automatisch vergibt und Sie ihn nicht direkt angeben können. Dies ist ein Unterschied zur Access-SQL-Syntax. Seit SQL Server 2008 ist es erlaubt, mehr als einen Datensatz über die *VALUES*-Liste an eine Tabelle anzufügen. Hierbei

trennen Sie die einzelnen Datensätze durch ein Komma. Folgendes Beispiel legt zwei Datensätze in der Tabelle *tblBestellungen* an.

```
INSERT INTO dbo.tblBestellungen (KundeID, Bestelldatum, Bemerkungen, RechnungAm,
Rechnungsdatei, Zahlungsziel, BezahlAm, UmsatzID, RechnungsversandartID)
VALUES (43, '20130622', 'Per INSERT angelegte Bestellung', '20130622', 'C:\Daten\
Buchprojekte\Test.pdf', '20130722', '20130715', 737, 2),
(40, '20130620', 'Per INSERT angelegte Bestellung', '20130628', 'C:\Daten\Buchprojekte\
Test.pdf', '20130801', '20130722', 737, 2);
```

9.8.2 Datensätze einfügen mit SELECT INTO

Mit der *SELECT INTO*-Anweisung können Sie das Erstellen einer neuen Tabelle und das Anfügen von Daten in einem Schritt erledigen. Die Syntax dieses Abfragetyps ist fast identisch mit der für normale *SELECT*-Anweisungen – mit der Ausnahme, dass Sie mit dem *INTO*-Schlüsselwort noch den Namen der Tabelle angeben, die SQL Server erstellen und mit den Daten des Abfrageergebnisses füllen soll. Den *INTO*-Abschnitt platzieren Sie einfach zwischen den *SELECT*- und den *FROM*-Abschnitt einer Abfrage.

Das folgende Beispiel zeigt, wie Sie die Tabelle zum Archivieren von Bestellungen erzeugen und gleichzeitig alle Bestellungen einfügen, deren Bestelldatum vor dem angegebenen Datum liegt:

```
SELECT dbo.tblBestellungen.*
INTO dbo.tblBestellungenArchiv
FROM dbo.tblBestellungen
WHERE tblBestellungen.Bestelldatum < '20120801';
```

Durch die *SELECT INTO*-Anweisung erhält die neue Tabelle dieselbe Spaltendefinition wie die Ausgangstabelle. Weitere Eigenschaften der Ausgangstabelle wie Primärschlüssel, Einschränkungen und andere werden jedoch nicht übernommen.

9.8.3 Autowert des zuletzt hinzugefügten Datensatzes

Fügen Sie einen einzelnen Datensatz per *INSERT INTO* in eine Tabelle mit einer *IDENTITY*-Spalte ein, möchten Sie möglicherweise dessen Autowert (*IDENTITY*) erfahren. Dies gelingt mit folgender Abfrage, wenn Sie diese direkt nach der Anfügeabfrage ausführen:

```
SELECT SCOPE_IDENTITY();
```

9.8.4 Datensätze aktualisieren mit UPDATE

Das Aktualisieren von Daten mit SQL erfolgt über die *UPDATE*-Anweisung:

```
UPDATE <Tabelle>
SET <Feldname1> = <Wert1>[, <Feldname2> = <Wert2>][, ...]
[WHERE <Kriterium>]
```

Dabei können Sie als *<Tabelle>* beliebige Tabellen angeben. *<Feldname1>*, *<Feldname2>* ... müssen in *<Tabelle>* enthalten sein. *<Wert1>*, *<Wert2>* ... sind die Werte, die den Feldern *<Feldname1>*, *<Feldname2>* ... zugewiesen werden sollen. Mit *<Kriterium>* legen Sie einen Ausdruck fest, der die von der Aktualisierung betroffenen Datensätze einschränkt. Die Abfrage des folgenden Beispiels fügt einen aus Nachname und Vorname bestehenden Ausdruck in das Feld *Kundenbezeichnung* der Tabelle *tblKundenBase* ein, wenn dieses noch keinen Wert oder eine leere Zeichenkette enthält:

```
UPDATE dbo.tblKundenBase
SET Kundenbezeichnung = IIF(Rechnung_Vorname Is Null, Rechnung_Nachname, Rechnung_
Nachname + ', ' + Rechnung_Vorname)
WHERE Kundenbezeichnung IS NULL OR Kundenbezeichnung = '';
```

Die Grundlage für die *UPDATE*-Anweisung kann auch aus einer Tabellenverknüpfung stammen. Folgende Abfrage setzt bei allen Kunden, zu denen in der Tabelle *tblEmailAdressen* eine E-Mail-Adresse hinterlegt ist, das Kennzeichen der Spalte *Newsletter* auf 1.

```
UPDATE dbo.tblKundenBase
SET Newsletter = 1
FROM dbo.tblKundenBase INNER JOIN dbo.tblEmailAdressen
ON dbo.tblKundenBase.EmailadresseID = dbo.tblEmailAdressen.EmailAdresseID;
```

9.8.5 Datensätze löschen mit DELETE

Löschabfragen lassen sich in T-SQL mit dem *DELETE*-Schlüsselwort ausführen. Sie haben die folgende Syntax:

```
DELETE [<Zieltabelle>] FROM <Tabelle> [WHERE <Kriterien>]
```

<Zieltabelle> und *<Tabelle>* sind identisch, wenn unter *<Tabelle>* nur eine Tabelle angegeben wird. In diesem Fall kann *<Zieltabelle>* weggelassen werden. *<Zieltabelle>* müssen Sie nur angeben, wenn *<Tabelle>* einen aus mehreren verknüpften Tabellen bestehenden Ausdruck enthält. Ansonsten reicht die einfache Form:

```
DELETE FROM <Tabelle> [WHERE Kriterien]
```

Die folgende Löschabfrage bezieht sich auf eine einzige Tabelle und löscht den Datensatz mit dem Primärschlüsselwert 123:

```
DELETE FROM dbo.tblBestellungen WHERE BestellungID = 123;
```

Wenn Sie alle Datensätze dieser Tabelle löschen möchten, verwenden Sie folgende Abfrage:

```
DELETE FROM dbo.tblBestellungen;
```

Dieses Beispiel macht die Angabe der Zieltabelle direkt hinter der *DELETE*-Anweisung nötig:

```
DELETE dbo.tblKundenBase
FROM dbo.tblKundenBase INNER JOIN dbo.tblEmailAdressen
```

Kapitel 9 T-SQL-Grundlagen

```
ON tblKundenBase.EmailadresseID = tblEmailAdressen.EmailadresseID  
WHERE tblEmailAdressen.EmailAdresse = 'andre@minhorst.com';
```

Hierbei werden alle Datensätze der Tabelle *tblKundenBase* gelöscht, die einer bestimmten E-Mail-Adresse aus der Tabelle *tblEmailAdressen* zugewiesen sind.

Unter T-SQL ist, im Gegensatz zu Access-SQL, auch der Einsatz einer Unterabfrage möglich, der das Gleiche bewirkt:

```
DELETE FROM dbo.tblKundenBase  
WHERE tblKundenBase.EmailadresseID IN (  
    SELECT tblEmailAdressen.EmailadresseID  
    FROM dbo.tblEmailAdressen  
    WHERE EmailAdresse = 'andre@minhorst.com'  
);
```

9.8.6 TRUNCATE TABLE

Möchten Sie alle Datensätze einer Tabelle löschen, können Sie dies mit der *TRUNCATE TABLE*-Anweisung erledigen. Diese erwartet lediglich den Namen der betroffenen Tabelle als Parameter:

```
TRUNCATE TABLE <Tabellenname>
```

Aus den folgenden und weiteren Gründen ist *TRUNCATE TABLE* schneller als *DELETE*, wenn es um das Leeren einer kompletten Tabelle geht:

- » *TRUNCATE TABLE* löscht nicht wie *DELETE* jeden einzelnen Datensatz, sondern lediglich die Zuordnungen der Datenseiten einer Tabelle.
- » Im Transaktionsprotokoll werden nicht die einzelnen Datensätze, sondern lediglich die aufgehobenen Zuordnungen der Datenseiten protokolliert.
- » *TRUNCATE TABLE* löst keine Trigger aus. Soll das Löschen etwa per Trigger protokolliert werden, verwenden Sie lieber die *DELETE*-Anweisung.

Ein weiterer Vorteil ist, dass ein eventuell vorhandener Autowert (*IDENTITY*) mit *TRUNCATE TABLE* auf den Startwert zurückgesetzt wird, mit *DELETE* jedoch nicht.

Zum Ausführen von *TRUNCATE TABLE* muss der jeweilige Benutzer die entsprechenden Rechte zum Ändern der Tabellenstruktur besitzen. Ein einfaches Recht zum Löschen der Daten der Tabelle reicht hier nicht aus. Es gibt bei der Verwendung von *TRUNCATE TABLE* noch einiges zu beachten – mehr in der SQL Server-Hilfe unter dem Stichwort *TRUNCATE TABLE*.

9.8.7 MERGE

Neu im SQL Server ab Version 2008 und ohne Pendant unter Access ist die *MERGE*-Anweisung. Die *MERGE*-Anweisung erlaubt es, Lösch-, Anfüge- und Aktualisierungsabfragen einfach zu kom-

binieren. Um zu verstehen, welche Möglichkeiten sie bietet, schauen wir uns drei Beispielaufufe einer gespeicherten Prozedur an, welche die *MERGE*-Anweisung verwendet.

Der erste Anweisung soll einen neuen Datensatz zur Tabelle *tblAnreden* hinzufügen, die ja die Felder *AnredeID*, *Anrede* und *Briefanrede* enthält. Die aufgerufene gespeicherte Prozedur erwartet vier Parameter:

- » *AnredeID*: Primärschlüssel, sofern vorhanden
- » *Anrede*: zu speichernde Anrede
- » *Briefanrede*: zu speichernde Briefanrede
- » *Delete*: BIT-Wert, der angibt, ob der Datensatz gelöscht werden soll, falls vorhanden

Der Aufruf zum Anlegen eines neuen Datensatzes sieht wie folgt aus – wobei der Primärschlüsselwert den Wert *NULL* erhält und *Delete* den Wert *0* (für *False* – beim SQL Server entspricht *False* dem Wert *0* und *True* dem Wert *1*):

```
EXEC dbo.spMergeAnrede NULL, 'Herr Dr.', 'Sehr geehrter Herr Dr.', 0
```

Der Aufruf dieser Anweisung wirkt sich wie in Abbildung 9.12 aus. Als Ergebnis liefert die gespeicherte Prozedur den Wert des Primärschlüsselfeldes des neu hinzugefügten Datensatzes zurück.

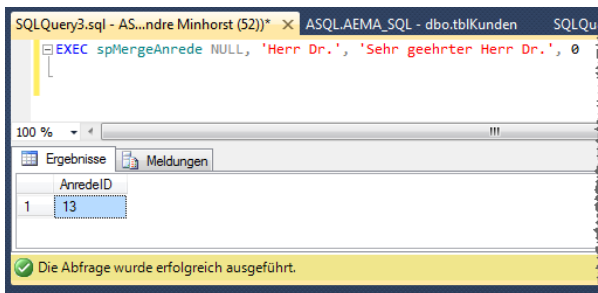


Abbildung 9.12: Hinzufügen eines Datensatzes mit *MERGE*

Im zweiten Beispiel soll ein vorhandener Datensatz geändert werden. Dazu geben wir den Primärschlüsselwert des zu ändernden Datensatzes an und neue Werte für die Felder *Anrede* und *Briefanrede*:

```
EXEC spMergeAnrede 13, 'Frau Dr.', 'Sehr geehrte Frau Dr.', 0
```

Auch dieser Aufruf liefert wieder den Wert des Feldes *AnredeID* des bearbeiteten Datensatzes zurück. Wenn Sie hingegen einen beliebigen Wert für das Primärschlüsselfeld übergeben, der ungleich *NULL* ist, legt die gespeicherte Prozedur den Datensatz ebenfalls an – allerdings mit dem durch die Autowert-Funktion beziehungsweise *IDENTITY* ermittelten neuen Primärschlüsselwert.

Kapitel 9 T-SQL-Grundlagen

Schließlich fehlt noch das Löschen. Übergeben Sie der gespeicherten Prozedur den Wert 1 für den letzten Parameter und einen vorhandenen Primärschlüsselwert für den ersten Parameter, wird der gefundene Datensatz gelöscht:

```
EXEC spMergeAnrede 13, 'Frau Dr.', 'Sehr geehrte Frau Dr.', 1
```

Die gespeicherte Prozedur liefert nach dem Löschen den Wert vom Feld *AnredeID* des eben gelöschten Datensatzes.

Gespeicherte Prozedur mit MERGE-Anweisung

Schauen wir uns an, wie die gespeicherte Prozedur *spMergeAnrede* erstellt wird und aufgebaut ist:

```
CREATE PROCEDURE dbo.spMergeAnrede(  
    @AnredeID INT = NULL,  
    @Anrede NVARCHAR(255),  
    @Briefanrede NVARCHAR(255),  
    @Delete BIT)  
AS  
DECLARE @result TABLE(id INT);  
SET NOCOUNT ON;  
MERGE dbo.tblAnreden AS b USING (  
    SELECT @AnredeID, @Anrede, @Briefanrede  
) AS source (AnredeID, Anrede, Briefanrede) ON b.AnredeID = source.AnredeID  
WHEN MATCHED AND @Delete = 1 THEN DELETE  
WHEN MATCHED THEN  
    UPDATE SET Anrede = source.Anrede, Briefanrede = source.Briefanrede  
WHEN NOT MATCHED And @Delete = 0 THEN  
    INSERT (Anrede, Briefanrede) VALUES(source.Anrede, source.Briefanrede)  
OUTPUT COALESCE(inserted.AnredeID, deleted.AnredeID) INTO @result;  
SELECT id AS AnredeID FROM @result;  
GO
```

Als Erstes deklariert die *CREATE PROCEDURE*-Anweisung die vier Parameter *@AnredeID*, *@Anrede*, *@Briefanrede* und *@Delete* sowie die *Table-Variable* *@result* als Rückgabewert. Dann beginnt die eigentliche *MERGE*-Anweisung. Diese führt zwei Tabellen zusammen: Die Erste ist die zu bearbeitende und tatsächlich in der Datenbank vorhandene Tabelle *tblAnreden*. Sie wird mit dem Namen *b* als Alias angegeben (weitere Referenzen wie *b.AnredeID* beziehen sich also auf die zu ändernde Tabelle). Die zweite Tabelle heißt *source* und besteht aus den drei Feldern *AnredeID*, *Anrede* und *Briefanrede*, wobei der einzige Datensatz dieser Tabelle mit den in den Parametern *@AnredeID*, *@Anrede* und *@Briefanrede* übergebenen Werten gefüllt wird.

Der *ON*-Teil mit dem Ausdruck *b.AnredeID = source.AnredeID* gibt an, dass im Folgenden nur solche Datensätze als vorhanden (*MATCHED*) behandelt werden, bei denen die in den beiden angegebenen Feldern enthaltenen Datensätze übereinstimmen.

In den *WHEN*-Klauseln werden verschiedene Zustände geprüft. Die Erste prüft beispielsweise, ob *MATCHED* zutrifft (die übergebene *AnredeID* stimmt mit dem Wert des gleichnamigen Feldes

eines der vorhandenen Datensätze überein) und gleichzeitig der Parameter *@Delete* den Wert 1 hat. In diesem Fall wird die *DELETE*-Anweisung aufgerufen, was bewirkt, dass der vorhandene Datensatz mit dem Wert *@AnredeID* im Feld *AnredeID* gelöscht wird.

Die Zweite prüft auch, ob *MATCHED* wahr ist. Hierhin gelangt die Prozedur nur, wenn *@Delete* nicht den Wert 1 hat. Dies bedeutet: Es ist ein Datensatz mit dem Wert *@AnredeID* im Feld *Anrede* vorhanden. Dieser soll dann mit den hinter der *UPDATE*-Klausel angegebenen Werten aktualisiert werden:

```
WHEN MATCHED THEN UPDATE SET Anrede = source.Anrede, Briefanrede = source.Briefanrede
```

Ist *MATCHED* nicht wahr, wird die letzte *WHEN*-Anweisung aufgerufen. Es ist noch kein Datensatz mit dem Wert *@AnredeID* im Feld *AnredeID* vorhanden, also wird ein neuer Datensatz angelegt. Die Felder *Anrede* und *Briefanrede* werden dann mit den Werten aus *source.Anrede* und *source.Briefanrede* gefüllt, welche wiederum zuvor mit den Werten der Parameter *@Anrede* und *@Briefanrede* versehen wurden.

Das Einfügen darf natürlich nur erfolgen, wenn die Prozedur nicht zum Löschen eines Datensatzes aufgerufen wurde. Schließlich könnte bei einem solchen Aufruf eine *AnredeID* übergeben werden, die nicht in der Tabelle existiert. In einem solchen Fall würde *NOT MATCHED* ebenfalls den Wert *True* liefern und somit den Datensatz anlegen. Aus diesem Grund ist hier *NOT MATCHED* mit der Prüfung *@Delete=0* ergänzt:

```
WHEN NOT MATCHED AND @Delete = 0 THEN  
INSERT (Anrede, Briefanrede) VALUES(source.Anrede, source.Briefanrede)
```

Am Schluss liefert *OUTPUT* die *AnredeID*, über die das Matching stattgefunden hat. Die *AnredeID* steht nach einem *INSERT* in der Spalte *inserted.AnredeID*, nach einem *DELETE* in der Spalte *deleted.AnredeID* und nach einem *UPDATE* in beiden Spalten, wobei *inserted.AnredeID* den Wert nach der Änderung und *deleted.AnredeID* den Wert vor der Änderung enthält. Die Funktion *COALESCE* prüft beide Felder und speichert den Wert des ersten Felds, das keinen *NULL*-Wert enthält, in der *Table-Variable @result*. Anschließend wird der Inhalt der *Table-Variable* ausgegeben. Ein praktisches Beispiel für den Aufruf dieser gespeicherten Prozedur und somit der *MERGE*-Anweisung finden Sie in Kapitel »Formulare und Berichte«, Seite 345.

Tabellen abgleichen mit MERGE

Während dies ein Beispiel für den Einsatz der *MERGE*-Anweisung zusammen mit Access ist, können Sie diese Anweisung auch zum Abgleichen der Daten zweier Tabellen verwenden – beispielsweise die aktuellen Daten Ihrer lokalen Datenbank mit den Daten aus der Datenbank Ihres Onlineshops. Ein Beispiel für den Abgleich zweier Tabellen sieht so aus:

```
CREATE PROCEDURE dbo.spMERGEArtikel AS  
MERGE INTO dbo.tblArtikel  
USING dbo.tblArtikelShop  
ON (dbo.tblArtikel.ArtikelID = dbo.tblArtikelShop.ArtikelID)
```

Kapitel 9 T-SQL-Grundlagen

```
WHEN MATCHED THEN
    UPDATE SET dbo.tblArtikel.Artikelname = dbo.tblArtikelShop.Artikelname
WHEN NOT MATCHED THEN
    INSERT (ArtikelID, Artikelname) VALUES (dbo.tblArtikelShop.ArtikelID, dbo.tblArtikelShop.Artikelname)
WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Die *MERGE*-Anweisung vergleicht die Tabellen *tblArtikel* und *tblArtikelShop*. Existiert zu einer *ArtikelID* je ein Datensatz in *tblArtikel* und in *tblArtikelShop*, wird der Artikelname aus der Tabelle *tblArtikelShop* in die Tabelle *tblArtikel* übertragen.

Ist die *ArtikelID* aus *tblArtikelShop* noch nicht in *tblArtikel* vorhanden, wird der Artikeldatensatz in der Tabelle *tblArtikel* angelegt. Wenn eine *ArtikelID* nur in der Tabelle *tblArtikel*, aber nicht in *tblArtikelShop* vorhanden ist, wird der Datensatz aus *tblArtikel* gelöscht.

9.9 Systemwerte abfragen

SQL Server bietet verschiedene Systemvariablen an, auf die Sie per Abfrage zugreifen können. Einige beinhalten Informationen zur Konfiguration des SQL Servers, andere wiederum Informationen zu Ausführungen von SQL-Anweisungen. Folgende Auflistung beinhaltet einige der interessantesten Systemvariablen:

- » *@@Servername*: liefert den Namen der aktuellen SQL Server-Instanz
- » *@@Version*: liefert Informationen über die installierte SQL Server-Version
- » *@@Language*: liefert die aktuelle Spracheinstellung
- » *@@Rowcount*: liefert die Anzahl der von einer Abfrage betroffenen Datensätze
- » *@@Error*: liefert die Fehlernummer der letzten Anweisung

9.10 Fehlerbehandlung

Mit der Systemvariablen *@@Error* kennen Sie bereits eine Möglichkeit zur Fehlerbehandlung. *@@Error* enthält nach jeder SQL-Anweisung die Fehlernummer zu dem Fehler, der durch die Anweisung ausgelöst wurde. Gab es keinen Fehler, ist der Wert von *@@Error* 0.

Für eine Fehlerbehandlung müssen Sie also nach jeder SQL-Anweisung prüfen, ob der Wert von *@@Error* größer 0 ist. Beachten Sie dabei, dass der Wert von *@@Error* nach *jeder* SQL-Anweisung neu gesetzt wird (siehe Abbildung 9.13). Die Fehlermeldung zu einer Fehlernummer ermitteln Sie mit der folgenden Anweisung:

```
SELECT [Text] FROM sys.messages WHERE Language_ID= 1033 AND Message_ID = <Fehlernummer>;
```

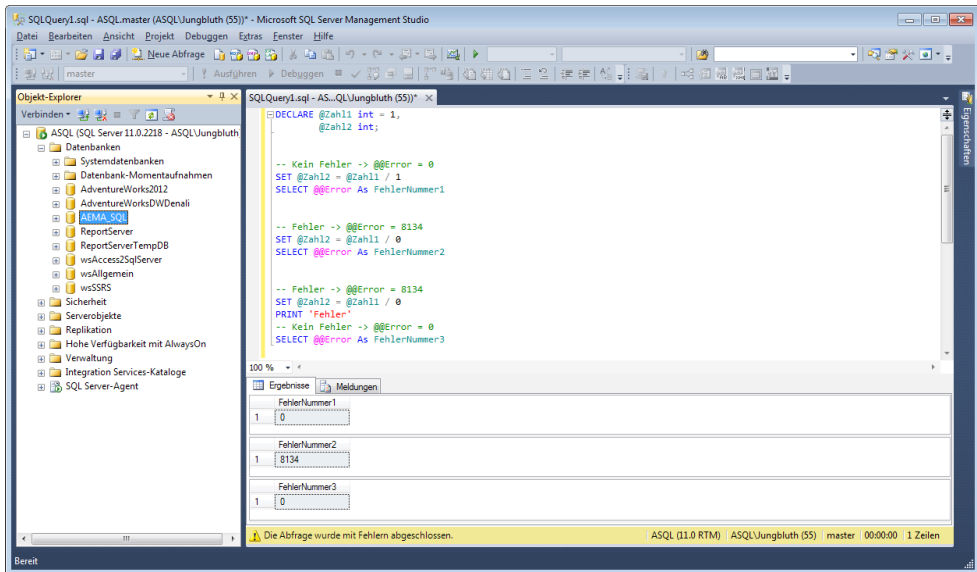



Abbildung 9.13: Die Systemvariable @@Error

@@Error war bis zum SQL Server 2008 die einzige Möglichkeit für eine Fehlerbehandlung. Seit SQL Server 2008 wurde dies mit den Befehlen *BEGIN TRY ... END TRY* und *BEGIN CATCH .. END CATCH* ergänzt. Dabei ist im *TRY*-Block die eigentliche Logik enthalten und im *CATCH*-Block die Fehlerbehandlung. Der *CATCH*-Block bietet Ihnen die folgenden Funktionen, die Ihnen weitere Informationen zum Fehler liefern.

- » *Error_Number()*: die Fehlernummer
- » *Error_Message()*: die Fehlermeldung
- » *Error_Line()*: die Zeilennummer, in der der Fehler eingetreten ist
- » *Error_Procedure()*: der Name der gespeicherten Prozedur oder des Triggers, in dem der Fehler eingetreten ist
- » *Error_State()*: der Status zur Fehlernummer. Pro Fehlernummer kann es mehrere Status geben, wobei jeder Status für eine andere Ausprägung des Fehlers steht.
- » *Error_Severity()*: der Schweregrad des Fehlers

Mit der *TRY/CATCH*-Variante sind Sie in der Lage, eine einheitliche Fehlerbehandlung in Ihrem T-SQL-Skript zu etablieren. Das Auswerten von @@Error nach jeder SQL-Anweisung ist hierbei nicht mehr notwendig. Folgendes Beispiel zeigt die Verwendung von *TRY/CATCH*.

```
DECLARE @Zahl int;
BEGIN TRY
```

Kapitel 9 T-SQL-Grundlagen

```
        SET @Zahl = 1 / 0
    END TRY
    BEGIN CATCH
        SELECT ERROR_NUMBER() as FehlerNr, ERROR_MESSAGE() As Fehler
    END CATCH;
```

@@Error und *TRY/CATCH* haben eines gemeinsam: Beide Fehlerbehandlungen sind unwirksam, liegt der Schweregrad des Fehlers bei 20 oder höher.

Fehler mit einem Schweregrad von 20 oder höher führen immer zum direkten Abbruch des T-SQL-Skripts, der gespeicherten Prozedur oder des Triggers. Eine Fehlerbehandlung ist hier nicht möglich.

Fehler auslösen

Hin und wieder kann es notwendig sein, einen Fehler manuell auszulösen. SQL Server stellt hierzu die Befehle *RAISERROR* und (seit SQL Server 2012) *THROW* zur Verfügung. *RAISERROR* belegt den Wert der Systemvariablen *@@Error* und gibt den angegebenen Text als Fehlermeldung aus. Die Syntax ist wie folgt:

```
RAISERROR ('Achtung, Fehler', 16, 1);
```

THROW löst tatsächlich einen Fehler aus, weshalb hier immer eine Fehlernummer angegeben werden muss. Die Fehlernummer kann dabei frei erfunden sein.

```
THROW 99999999, 'Achtung, Fehler', 1;
```

Möchten Sie bei einer *TRY/CATCH*-Fehlerbehandlung im *CATCH*-Block den Fehler erneut auslösen, müssen Sie bei Verwendung von *RAISERROR* den Fehler anhand der oben beschriebenen Funktionen erzeugen.

```
DECLARE @Zahl int;
BEGIN TRY
    SET @Zahl = 1 / 0
END TRY
BEGIN CATCH
    DECLARE @Fehlermeldung nvarchar(4000) = ERROR_MESSAGE();
    DECLARE @Schweregrad int = ERROR_SEVERITY();
    DECLARE @Status int = ERROR_STATE();
    RAISERROR (@Fehlermeldung, @Schweregrad, @Status);
END CATCH;
```

Ab SQL Server 2012 wird im *CATCH*-Block der Fehler einfach mittels *THROW* erneut ausgelöst.

```
DECLARE @Zahl int;
BEGIN TRY
    SET @Zahl = 1 / 0
END TRY
BEGIN CATCH
    THROW;
END CATCH;
```

9.11 NULL-Werte

Zum Prüfen und Bearbeiten von NULL-Werten bietet SQL Server einige Hilfsfunktionen. Da wäre als Erstes der Operator *Is NULL* zu nennen, mit dem der Wert einer Variablen oder eine Spalte auf *NULL* geprüft wird.

```
DECLARE @Zahl1 int;
IF @Zahl1 Is NULL
BEGIN
    PRINT 'Kein Wert'
END;
```

Solche Prüfungen sollten Sie immer mit *Is Null* oder *Is Not Null* ausführen. Eine Prüfung mit einem Gleichheitszeichen (*Wert = NULL*) liefert als Ergebnis *UNKNOWN*. Ob *UNKNOWN* nun als *True* oder *False* interpretiert wird, ist nicht definiert.

Die Funktion *ISNULL* arbeitet genauso wie das von Access-Ausdrücken bekannte *IsNull*. Es erwartet zwei Parameter: den auf *NULL* zu prüfenden Wert und den Wert, durch den ein eventuell vorhandener *NULL*-Wert ersetzt werden soll. Die folgende Abfrage liefert beispielsweise immer den Wert *0*, wenn das Feld *Einzelpreis* den Wert *NULL* enthält:

```
SELECT Artikelname, ISNULL(Einzelpreis, 0) As Einzelpreis FROM dbo.tblArtikel;
```

Es gibt jedoch einen entscheidenden Unterschied zu der *IsNull*-Variante von Access: Der zweite Parameter muss beim SQL Server explizit angegeben werden, sonst wird ein Fehler ausgelöst. Die *IsNull*-Funktion von Access gibt für Zahlenfelder automatisch *0* und für Textfelder eine leere Zeichenkette ("") zurück.

Abschließend sei noch die Funktion *COALESCE* erwähnt, die Sie bereits von der Beschreibung des T-SQL-Befehls *MERGE* kennen. *COALESCE* überprüft die übergebenen Parameterwerte auf *NULL* und gibt den ersten Wert zurück, der nicht *NULL* ist. Sind alle Werte *NULL*, ist das Ergebnis von *COALESCE* ebenfalls *NULL*. Folgende Anweisung liefert als Ergebnis den Wert *2* der Variablen *@Zahl2*, da diese Variable der erste Parameter mit einem Wert ungleich *NULL* ist.

```
DECLARE @Zahl1 int, @Zahl2 int, @Zahl3 int;
SELECT @Zahl2 = 2, @Zahl3 = 3;
SELECT COALESCE(@Zahl1, @Zahl2, @Zahl3) As Zahl;
```

Wie geht es weiter?

Mit den Erkenntnissen dieses Kapitels sind Sie für das Aufbauen von gespeicherten Prozeduren, benutzerdefinierten Funktionen und Triggern gewappnet. In den folgenden Kapitel lernen Sie diese Objekttypen kennen, die mit T-SQL programmiert werden.

10 Gespeicherte Prozeduren

Neben den Sichten können Sie auch mit gespeicherten Prozeduren auf die Daten einer SQL Server-Datenbank zugreifen und sich das ermittelte Ergebnis ausgeben lassen. Dabei sind gespeicherte Prozeduren gegenüber Sichten wesentlich flexibler: Sie erlauben zum Beispiel den Einsatz von Parametern und bieten zudem die Möglichkeit, Daten hinzuzufügen, zu ändern und zu löschen. Dabei können gespeicherte Prozeduren eine oder mehrere SQL-Anweisungen oder auch weitere Befehle und sogar Strukturen wie Bedingungen oder Variablen enthalten.

Wie wir bereits weiter vorn im Buch erläutert haben, sind Access-Abfragen, die sich auf Tabellen einer SQL Server-Datenbank beziehen, meist sehr viel langsamer als solche Abfragen, die direkt im SQL Server ausgeführt werden. Deshalb verwenden Sie besser Sichten und gespeicherte Prozeduren statt Access-Abfragen.

Im Gegensatz zu Sichten, die Sie genau wie die Tabellen einer SQL Server-Datenbank mit einer Access-Datenbank verknüpfen können, ist zu gespeicherten Prozeduren keine Verknüpfung möglich – zumindest keine direkte. Wenn Sie eine gespeicherte Prozedur aufrufen und das Ergebnis in Access anzeigen oder weiterverwenden möchten, müssen Sie eine Pass-Through-Abfrage anlegen, welche den Aufruf der gespeicherten Prozedur und eventuell benötigte Parameter enthält.

10.1 Vorteile gespeicherter Prozeduren

Sie können über eingebundene Tabellen auf eine SQL Server-Datenbank zugreifen oder Sie schreiben die gewünschte SQL-Anweisung in eine Pass-Through-Abfrage und senden diese zur Ausführung an den SQL Server. Der beste Weg ist es jedoch, die SQL-Anweisungen in Form einer gespeicherten Prozedur in der SQL Server-Datenbank zu speichern und diese dann über eine Pass-Through-Abfrage aufzurufen. Warum die Abarbeitung direkt im SQL Server schneller erfolgt als bei verknüpften Tabellen, haben wir bereits im Kapitel »Performance analysieren«, Seite 105, besprochen. Warum aber soll man SQL-Anweisungen in gespeicherte Prozeduren schreiben und diese nicht ad-hoc aufrufen? Dies und weitere Vor- wie auch Nachteile von gespeicherten Prozeduren klären die folgenden Abschnitte.

10.1.1 Geschwindigkeit

Grundsätzlich ist es möglich, beim Einsatz einer Access-Anwendung mit SQL Server-Backend komplett auf gespeicherte Prozeduren zu verzichten. Sie könnten die SQL-Abfragen oder auch komplette Abfolgen von SQL-Anweisungen in einer Pass-Through-Abfrage speichern und diese dann zum SQL Server senden, damit dieser die Anweisungen ausführt. Das Ausführen einer gespeicherten Prozedur, die dieselbe Anweisung enthält, ist jedoch wesentlich schneller als die

Ad-hoc-Ausführung von SQL-Anweisungen über eine Pass-Through-Abfrage. Schauen wir uns an, warum dies so ist:

Die Ad-hoc-Ausführung von SQL-Anweisungen unterscheidet sich auf den ersten Blick nicht von der Ausführung einer gespeicherten Prozedur oder den anderen SQL Server-Objekten Sichten, Funktionen und Triggern. Bei allen erfolgt zunächst eine Syntaxprüfung des enthaltenen SQL und eine Existenzprüfung der dort verwendeten Tabellen, Sichten et cetera samt deren Spalten. Anschließend erstellt der Abfrageoptimierer den Ausführungsplan und speichert diesen im Prozedurcache. Zur Erinnerung: der Ausführungsplan legt fest, in welcher Reihenfolge etwa die Tabellen und Felder einer SQL-Anweisung gelesen werden und wie dabei die Indizes zu berücksichtigen sind. Im Anschluss daran folgt die Ad-hoc-Ausführung der SQL-Anweisung beziehungsweise des SQL Server-Objekts anhand des eben erstellten Ausführungsplans.

Ab der zweiten Ausführung entfällt das Erstellen des Ausführungsplans, denn der bereits im Prozedurcache gespeicherte wird einfach wiederverwendet. Da die Arbeit des Abfrageoptimierers in diesem ganzen Vorgang die zeitintensivste ist, wird durch die Wiederverwendung viel Zeit gespart und eine bessere Gesamtperformance erreicht. Die vom Abfrageoptimierer erstellten Ausführungspläne und deren Wiederverwendung haben wir im Kapitel »FAQ«, Seite 19, kurz bereits beschrieben.

Soweit die kurze Wiederholung zur Vorgehensweise der Ausführung einer SQL-Anweisung und/oder eines SQL Server-Objekts. Bis jetzt ist kein Unterschied zwischen einer Ad-hoc-SQL-Anweisung und einer gespeicherten Prozedur zu erkennen. Aber es gibt einen, der in der Performance einen großen Unterschied ausmacht.

Der Abfrageoptimierer erstellt für eine Ad-hoc-SQL-Anweisung einen Ausführungsplan, der exakt auf diese SQL-Anweisung zutrifft. Führen Sie zum Beispiel die beiden folgenden SQL-Anweisungen einzeln aus, ergibt dies zwei Ausführungspläne:

```
SELECT Bestelldatum FROM dbo.tblBestellungen WHERE KundeID = 74;  
SELECT Bestelldatum FROM dbo.tblBestellungen WHERE KundeID = 96;
```

Hier greift der Vorteil und Nutzen des Ausführungsplans nur, wenn beide Abfragen mehrmals aufgerufen werden. Erstellen Sie eine gespeicherte Prozedur, die Ihnen ebenfalls das Bestelldatum liefert, wobei jedoch die *KundeID* als Parameter übergeben wird, erzeugt der Abfrageoptimierer nur einen Ausführungsplan – für die gespeicherte Prozedur. Dieser wird bei jedem Aufruf wiederverwendet, unabhängig von der übergebenen ID des Kunden.

Bei einer gespeicherten Prozedur ist somit nicht nur die Wiederverwendbarkeit des zugehörigen Ausführungsplans höher, der Prozedurcache enthält auch weitaus weniger Ausführungspläne.

10.1.2 Datenkonsistenz und Geschäftsregeln

Sie können Ihre Anwendung auch so konzipieren, dass Sie keine Tabellen einer SQL Server-Datenbank einbindet. An Stelle dessen verwenden Sie gespeicherte Prozeduren für die Datener-

mittlung, wie auch für das Hinzufügen, Ändern und Löschen von Daten. Dies ist ohne Weiteres möglich – *SELECT*-, *UPDATE*-, *INSERT*- und *DELETE*-Anweisungen lassen sich problemlos in gespeicherten Prozeduren abbilden, und dies sogar kombiniert. Nicht nur das: in gespeicherten Prozeduren können Sie richtig programmieren.

Es stehen Ihnen fast alle Möglichkeiten von T-SQL zur Verfügung, wie *IF...ELSE* und *WHILE* sowie die Verwendung von Variablen und Systemfunktionen (siehe Kapitel »T-SQL-Grundlagen«, Seite 221). Die Gewährleistung der Datenkonsistenz und die Einhaltung der Geschäftsregeln lassen sich also auch mit gespeicherten Prozeduren realisieren.

Was aber haben Sie nun davon? Gesetzt den Fall, dass Sie nicht nur mit einem Access-Frontend auf die Daten im SQL Server-Backend zugreifen, sondern auch noch über eine Webanwendung und vielleicht eine .NET-Desktop-Anwendung, müssen Sie die Geschäftsregeln in allen Frontends realisieren. Selbst die kleinste Änderung ist dann in allen Versionen der Benutzeroberfläche anzupassen. Definieren Sie die Geschäftsregeln hingegen im SQL Server-Backend in gespeicherten Prozeduren, brauchen Sie die Änderung nur dort durchzuführen und nicht in jedem einzelnen Frontend.

Angenommen, Sie entscheiden sich, in einer Tabelle keine Datensätze mehr zu löschen, sondern stattdessen die Datensätze als inaktiv zu kennzeichnen. Für diese Änderung müssten Sie in jeder Benutzeroberfläche den Löschvorgang durch eine entsprechende *UPDATE*-Anweisung austauschen. Liegt Ihre Logik für den Löschvorgang in einer gespeicherten Prozedur, ersetzen Sie nur dort den *DELETE*-Befehl durch einen *UPDATE*-Befehl.

Die Änderung an einer Stelle bedeutet nicht nur weniger Aufwand, auch die Fehleranfälligkeit ist geringer. Es gibt noch einen weiteren und nicht zu unterschätzenden Vorteil: Nach dem Speichern der gespeicherten Prozedur steht die Änderung direkt für alle Benutzeroberflächen zur Verfügung. Ein erneutes Verteilen der Access-Datenbanken, .NET-Applikationen, oder was auch immer die Benutzeroberfläche anbietet, entfällt.

Jetzt werden Sie möglicherweise einwerfen, dass Sie ja ausschließlich mit einem Access-Frontend arbeiten und dies auch in absehbarer Zeit nicht geändert werden soll. Dann erhalten Sie durch die Verlagerung der Geschäftsregeln in gespeicherte Prozeduren immer noch einen Vorteil: Sobald Sie nämlich selbst innerhalb eines einzigen Access-Frontends von mehreren Stellen aus etwa die Daten einer Tabelle ändern, müssen Sie an all diesen Stellen die Geschäftslogik implementieren. Liegt diese hingegen in einer gespeicherten Prozedur, brauchen Sie sich bei der Programmierung des Frontends keine Sorgen darüber zu machen.

Und damit der Benutzer nicht doch irgendeinen Weg findet, eine Tabelle per ODBC einzubinden oder per VBA darauf zuzugreifen, entziehen Sie dem Benutzer einfach die kompletten Rechte an dieser Tabelle!

Wenn ihm als einzige Möglichkeit zum Ändern eines Datensatzes eine entsprechende gespeicherte Prozedur vorliegt, werden die dazu definierten Geschäftsregeln in jedem Fall durchgesetzt. Diese Vorgehensweise lässt sich natürlich auch beim Hinzufügen und Löschen von Datensätzen

und sogar bei der Datenermittlung und Datenausgabe nutzen. Jegliche Aktion mit den Daten erfolgt über gespeicherte Prozeduren und somit über die dort definierte Geschäftslogik.

10.2 Nachteile von gespeicherten Prozeduren

Wenn denn überhaupt von Nachteilen die Rede sein kann, sind lediglich zwei Punkte zu erwähnen. Und beide betreffen die Verwendung von gespeicherten Prozeduren. Gespeicherte Prozeduren werden mit dem Befehl *EXECUTE* oder *EXEC* ausgeführt. *EXECUTE* wiederum lässt sich nicht in einer *SELECT*-Anweisung nutzen. Aus diesem Grund ist es nicht möglich, eine gespeicherte Prozedur in einer *SELECT*-Anweisung ähnlich einer Tabelle, Sicht oder Funktion anzusprechen.

Der zweite Punkt bezieht sich auf die Verwendung von gespeicherten Prozeduren in Access. Die von einer gespeicherten Prozedur gelieferten Daten können in Access nicht wie bei einer eingebundenen Tabelle direkt geändert werden. Dies liegt jedoch nicht an der gespeicherten Prozedur, sondern vielmehr an der Pass-Through-Abfrage. Daten, die Access über eine Pass-Through-Abfrage ermittelt, können Sie schlicht und einfach nicht ändern. Dabei ist es egal, ob Sie in der Pass-Through-Abfrage eine gespeicherte Prozedur oder eine SQL-Anweisung ausführen.

10.3 Gespeicherte Prozeduren erstellen

Gespeicherte Prozeduren verwenden die Programmiersprache *T-SQL*. Diese ist selbst verglichen mit VBA relativ übersichtlich. Für die Erstellung gespeicherter Prozeduren liefert diese Programmiersprache die folgenden interessanten Features:

- » Definition von Ein- und Ausgabeparametern mit oder ohne Standardwerte
- » Deklaration und Verwendung von Variablen
- » Abfrage von Systemwerten wie Anzahl geänderter Datensätze
- » Verwendung einfacher Strukturen wie *IF...ELSE*
- » Programmierung einfacher Schleifen
- » Speichern von Zwischenergebnissen in temporären Tabellen oder *TABLE*-Variablen
- » Verschachteln von gespeicherten Prozeduren
- » Einsatz von Sichten und benutzerdefinierten Funktionen innerhalb gespeicherter Prozeduren
- » Implementieren einer Fehlerbehandlung

Eine kleine Einführung in T-SQL finden Sie in Kapitel »T-SQL-Grundlagen«, Seite 221.

10.3.1 Anlegen einer gespeicherten Prozedur mit Vorlage

Das SQL Server Management Studio bietet für den Beginn eine recht gute Hilfe zum Anlegen einer gespeicherten Prozedur.

Dazu wählen Sie im Objekt-Explorer zunächst die entsprechende Datenbank aus (etwa die Beispieldatenbank *AEMA_SQL*) und dort im Kontextmenü des Elements *Programmierbarkeit*/*Gespeicherte Prozeduren* den Eintrag *Neue gespeicherte Prozedur ...* (siehe Abbildung 10.1).

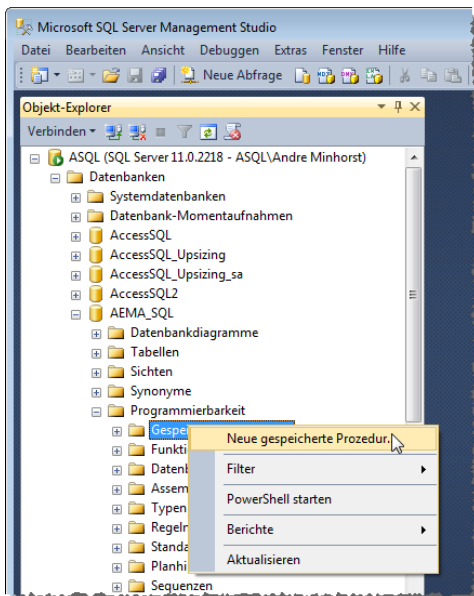
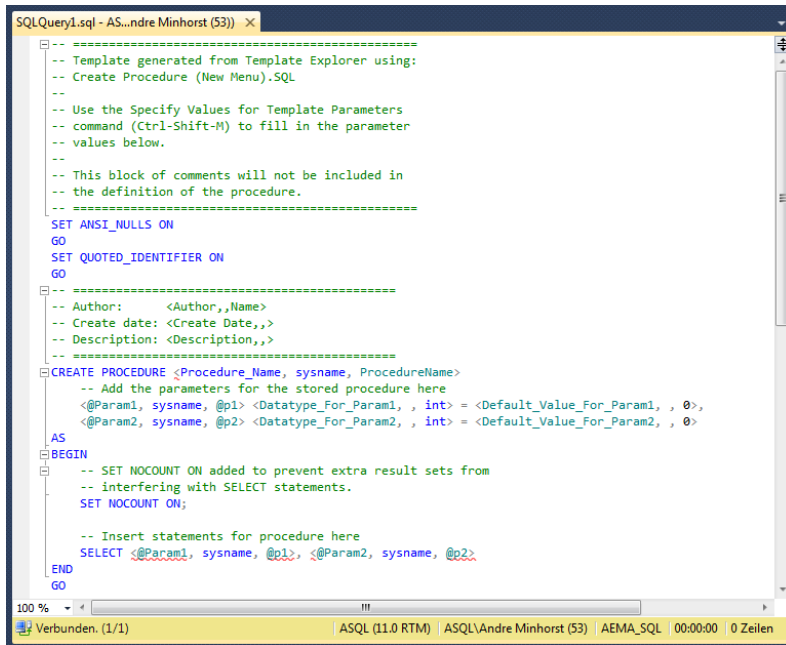


Abbildung 10.1: Anlegen des Grundgerüsts einer gespeicherten Prozedur

Dies öffnet ein neues Abfragefenster und bildet dort das Grundgerüst für eine neue gespeicherte Prozedur ab (siehe Abbildung 10.2). Der Code der Vorlage sieht auf den ersten Blick etwas verwirrend aus, aber wenn Sie erst mal die Kommentare entfernen, ist es nur noch halb so wild.

Hier wird direkt ein wesentlicher Unterschied zum VBA-Editor deutlich: Beim SQL Server legen Sie eine Prozedur nicht einfach in einem Modul an, sondern Sie verwenden dazu einen T-SQL-Befehl, in diesem Fall *CREATE PROCEDURE*.

Es gibt auch nirgends eine sichtbare Fassung der gespeicherten Prozedur – Sie können lediglich eine Anweisung generieren lassen, die den Code zum Erstellen oder Ändern der gespeicherten Prozedur bereitstellt. Der wiederum sieht fast genauso aus wie der, mit dem Sie die gespeicherte Prozedur einst erstellt haben.



```
-- =====
-- Template generated from Template Explorer using:
-- Create Procedure (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- =====
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =====
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =====
CREATE PROCEDURE <Procedure_Name, sysname>, <ProcedureName>
-- Add the parameters for the stored procedure here
    <@Param1, sysname>, <@p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>,
    <@Param2, sysname>, <@p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    -- Insert statements for procedure here
    SELECT <@Param1, sysname>, <@p1>, <@Param2, sysname>, <@p2>
END
GO
```

Abbildung 10.2: Vorlage für eine gespeicherte Prozedur

10.3.2 Neue gespeicherte Prozedur

Trotz dieser ungewohnten Vorgehensweise wollen wir uns ans Werk machen. Leeren wir das Abfragefenster und beginnen komplett von vorn. Der erste Teil der Anweisung zum Erstellen einer gespeicherten Prozedur lautet immer wie folgt und wird meist in der ersten Zeile abgebildet:

```
CREATE PROCEDURE <Schema>.<Prozedurname>
```

Es ist üblich, die Anweisung zum Erstellen einer gespeicherten Prozedur auf mehrere Zeilen aufzuteilen. Dies dient lediglich der Übersicht – Sie könnten die Anweisung auch in einer Zeile erfassen.

Die erste Zeile legt das Schema und den Namen der gespeicherten Prozedur fest. Der Name beginnt meist mit *sp* und darf nur alphanumerische Zeichen und den Unterstrich enthalten. Sie können natürlich auch ein anderes Präfix als *sp* verwenden, aber nicht *sp_*. Dies ist das Präfix der gespeicherten Prozeduren des Systems.

Das Problem mit diesem Präfix ist, dass der SQL Server beim Aufruf von gespeicherten Prozeduren mit dem Präfix *sp_* diese zunächst in der *master*-Datenbank und anschließend in den Systemobjekten der aktuellen Datenbank sucht, bevor er sie letztendlich bei den benutzerdefinierten Prozeduren findet. Dieser Suchvorgang kostet nur unnötig Zeit.

Den Platzhalter <Schema> ersetzen Sie mit dem Namen des Schemas, dem die gespeicherte Prozedur angehören soll. Wenn Sie Ihr Berechtigungskonzept nicht auf Schemata aufbauen wollen (siehe Kapitel »Sicherheit und Benutzerverwaltung«, Seite 395), geben Sie hier einfach immer *dbo* an. Dies gilt auch für die übrigen Datenbankobjekte. Die erste Zeile lautet dann also:

```
CREATE PROCEDURE dbo.spAlleArtikel
```

Nach der Benennung folgen die Parameter. Die Definition eines Parameters besteht aus den folgenden Elementen:

- » Name des Parameters, der immer mit *@* beginnt
- » Datentyp des Parameters – zum Beispiel *int* oder *nvarchar(255)*. Die Datentypen sind die gleichen, die auch bei der Definition von Tabellen zum Einsatz kommen (siehe Abschnitt »Datentypen von Access nach SQL Server«, Seite 164).
- » Schlüsselwort *OUTPUT*, falls es sich um einen Rückgabewert handelt
- » Standardwert, der mit einem Gleichheitszeichen zugewiesen wird. Ein Standardwert kennzeichnet den Parameter als optional, wodurch dieser beim Aufruf der gespeicherten Prozedur nicht zwingend angegeben werden muss.

Eine Prozedur kann komplett ohne Parameter auskommen, Sie können aber auch bis zu 1.024 Parameter definieren. Die Parameter werden durch Kommata voneinander getrennt und wiederum der Übersicht halber in jeweils eine eigene Zeile geschrieben.

Nach der Parameterliste folgt das Schlüsselwort *AS* und schließlich die eigentlichen Anweisungen der gespeicherten Prozedur. In einem ganz einfachen Fall sieht dies nun wie folgt aus:

```
CREATE PROCEDURE dbo.spAlleArtikel  
AS  
SELECT * FROM dbo.tblArtikel;
```

Um die Prozedur zu erstellen, betätigen Sie am einfachsten die Taste *F5*. Das Abfragefenster quittiert dies, sofern keine Fehler auftreten, wie in Abbildung 10.3.

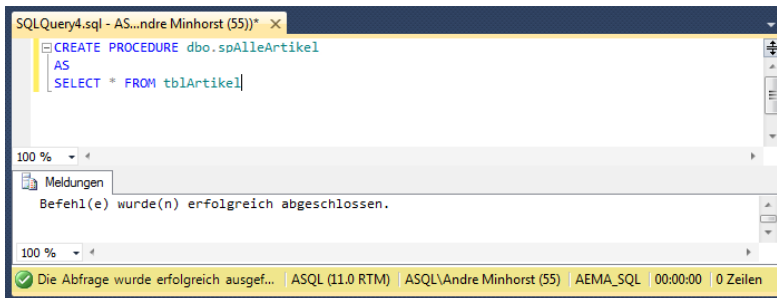


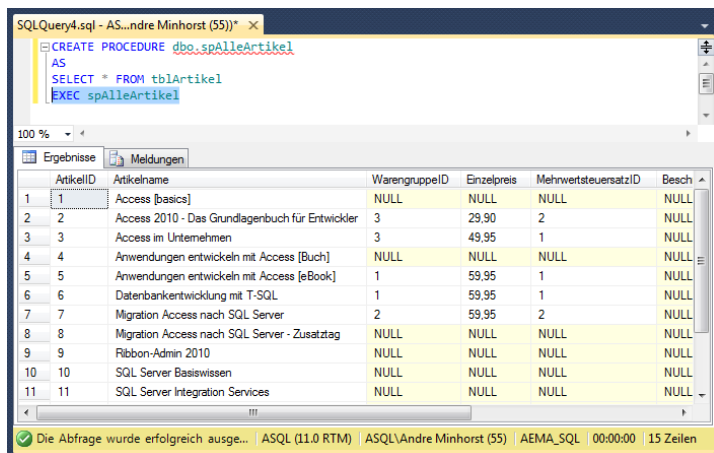
Abbildung 10.3: Erstellen einer ersten gespeicherten Prozedur

Kapitel 10 Gespeicherte Prozeduren

Um die soeben erstellte Prozedur auszuführen, tragen Sie den folgenden Befehl in ein neues oder in das aktuelle Abfragefenster ein:

```
EXEC dbo.spAlleArtikel;
```

Wenn Sie die Anweisung in ein Abfragefenster eingeben, das bereits andere Anweisungen enthält, markieren Sie die auszuführende Anweisung zuvor und betätigen Sie dann die Taste F5. Das Abfragefenster liefert das Resultat in der Datenblattansicht (siehe Abbildung 10.4).



ArtikelID	Artikelname	WarengruppeID	Einzelpreis	MehrwertsteuersatzID	Besch
1	Access [basics]	NULL	NULL	NULL	NULL
2	Access 2010 - Das Grundlagenbuch für Entwickler	3	29.90	2	NULL
3	Access im Unternehmen	3	49.95	1	NULL
4	Anwendungen entwickeln mit Access [Buch]	NULL	NULL	NULL	NULL
5	Anwendungen entwickeln mit Access [eBook]	1	59.95	1	NULL
6	Datenbankentwicklung mit T-SQL	1	59.95	1	NULL
7	Migration Access nach SQL Server	2	59.95	2	NULL
8	Migration Access nach SQL Server - Zusatztag	NULL	NULL	NULL	NULL
9	Ribbon-Admin 2010	NULL	NULL	NULL	NULL
10	SQL Server Basiswissen	NULL	NULL	NULL	NULL
11	SQL Server Integration Services	NULL	NULL	NULL	NULL

Abbildung 10.4: Aufruf der gespeicherten Prozedur über das Abfragefenster

10.3.3 Gespeicherte Prozedur mit Parametern

Möchten Sie der gespeicherten Prozedur Parameter übergeben, was im Großteil der Fälle geschehen wird, geben Sie diese vor dem AS-Schlüsselwort an. Wir erweitern die Logik der eben beschriebenen Prozedur so, dass diese nur einen bestimmten Artikel liefert:

```
CREATE PROC dbo.spArtikelNachID  
@ArtikelID int  
AS  
SELECT * FROM dbo.tblArtikel WHERE ArtikelID = @ArtikelID;
```

Den Parameter übergeben Sie nun wie folgt:

```
EXEC dbo.spArtikelNachID 2;
```

Die gespeicherte Prozedur liefert jetzt nur den gesuchten Artikel zurück.

In diesem Beispiel wird eine Ganzzahl als Parameter übergeben. Erwartet der Parameter jedoch eine Zeichenfolge, müssen Sie diese in Hochkommata angeben. Dies gilt auch für Parameter mit Datumswerten, hier ist die Zeichenfolge das Datum im ISO-Format (yyyy-mm-dd). Auch Dezimalzahlen übergeben Sie als Zeichenfolge, wobei Sie als Dezimaltrennzeichen den Punkt

verwenden. Im nächsten Beispiel legen wir eine gespeicherte Prozedur an, die zwei Parameter erwartet – ein Bestelldatum und die ID eines Kunden.

```
CREATE PROCEDURE dbo.spBestellungenNachDatumUndKunde
    @Bestelldatum datetime2,
    @KundeId int
AS
SELECT RechnungAm, Zahlungsziel FROM dbo.tblBestellungen
WHERE Bestelldatum = @Bestelldatum AND KundeID = @KundeId;
```

Beim Aufruf dieser gespeicherten Prozedur übergeben Sie nun das Bestelldatum als Zeichenfolge im ISO-Format und, durch ein Komma getrennt, die ID des Kunden als Zahl:

```
EXEC dbo.spBestellungenNachDatumUndKunde '2012-08-12', 205;
```

10.3.4 Gespeicherte Prozedur mit optionalen Parametern

Weisen Sie einem Parameter einen Standardwert zu, verhält sich dieser wie ein optionaler Parameter in VBA. Dies bedeutet, dass beim Aufruf der gespeicherten Prozedur der Parameter nicht zwingend angegeben werden muss.

Bei folgender gespeicherten Prozedur ist der zweite Parameter mit einem Standardwert belegt und somit optional für den Aufruf.

```
CREATE PROC dbo.spArtikelNachWarengruppeUndPreis
    @WarengruppeID int, @Einzelpreis money = 0
AS
SELECT ArtikelID, Artikelname, Einzelpreis, WarengruppeID
FROM dbo.tblArtikel
WHERE WarengruppeID = @WarengruppeID AND Einzelpreis >= @Einzelpreis;
```

Rufen Sie gespeicherte Prozedur ohne den Parameter *@Einzelpreis* auf, gilt für den Parameter der Standardwert 0. Sie erhalten somit alle Artikel der angegebenen Warengruppe mit einem Einzelpreis größer 0 Euro – letztendlich also alle Artikel der gewünschten Warengruppe:

```
EXEC dbo.spArtikelNachWarengruppeUndPreis 3
```

Im nächsten Aufruf wird der optionale Parameter angegeben. Jetzt liefert die gespeicherte Prozedur die Artikel der Warengruppe 3 mit Preisen größer 19,95 Euro.

```
EXEC dbo.spArtikelNachWarengruppeUndPreis 3, '19.95'
```

10.3.5 Gespeicherte Prozedur mit Variablen

In gespeicherten Prozeduren können Sie lokale Variablen verwenden. Diese deklarieren Sie ähnlich wie die Parameter, allerdings erst hinter dem *AS*-Schlüsselwort und mit der Anweisung *DECLARE*. Auch hier geben Sie erst den Variablennamen (beginnend mit *@*) und dann den Datentyp ein.

Kapitel 10 Gespeicherte Prozeduren

Die Zuweisung erfolgt im einfachen Fall mit der *SET*-Anweisung. Im folgenden Beispiel füllen wir die Variable *@int1* mit dem Wert *1* und geben den Inhalt der Variablen als Meldung aus:

```
CREATE PROCEDURE dbo.spVariablen
AS
DECLARE @int1 int;
SET @int1 = 1;
PRINT @int1;
```

Nun folgt ein praxisnahes Beispiel für eine gespeicherte Prozedur mit einer Variablen: Das Ergebnis soll den Kunden liefern, der eine Bestellung mit einer bestimmten ID aufgegeben hat. Man könnte dies mit einer *INNER JOIN*-Abfrage erledigen, aber wir gehen einmal einen kleinen Umweg. Die folgende Prozedur erwartet als Parameter die *BestellungID* der Bestellung, zu welcher der Kunde ausgegeben werden soll. Sie deklariert eine lokale Variable namens *@KundeID*. Dieser weist die Prozedur mit einer *SELECT*-Abfrage den Wert des Feldes *KundeID* aus dem Datensatz der übergebenen *BestellungID* zu.

Schließlich liefert die letzte *SELECT*-Abfrage den passenden Kundendatensatz zurück:

```
CREATE PROCEDURE dbo.spKundeNachBestellungID
@BestellungID int
AS
DECLARE @KundeID int;
SELECT @KundeID = KundeID FROM dbo.tblBestellungen WHERE BestellungID = @BestellungID;
SELECT * FROM dbo.tblKundenBase WHERE KundeID = @KundeID;
```

Der Aufruf sieht wieder wie folgt aus, das Ergebnis ist ein einzelner Datensatz der Tabelle *tblKundenBase*:

```
EXEC dbo.spKundeNachBestellungID 140
```

10.3.6 Gespeicherte Prozedur mit Rückgabewert

Den Rückgabeparameter können Sie von Access aus nicht direkt nutzen. Ähnlich wie bei einem mit *ByRef* deklarierten Parameter unter VBA müssen Sie auch hier beim Aufruf eine Variable übergeben, die Sie anschließend auswerten. Wir schauen uns dies gleich anhand eines Beispiels an. Die erste gespeicherte Prozedur verwendet den Rückgabeparameter *@Rueckgabewert*. Dieser wird dem Schlüsselwort *OUTPUT* gekennzeichnet und in der Prozedur mit dem Zahlenwert *12345* gefüllt:

```
CREATE PROCEDURE dbo.spProcMitRueckgabewert
@Rueckgabewert int OUTPUT
AS
SET @Rueckgabewert = 12345;
```

Die zweite Prozedur deklariert die Variable *@Rueckgabe* und ruft die erste Prozedur *spProcMitRueckgabewert* auf, wobei dem Parameter *@Rueckgabewert* die Variable *@Rueckgabe* zugewiesen wird. Diese Zuweisung erhält noch das Schlüsselwort *OUTPUT* zur genaueren Kennzeich-

nung. Nach dem Aufruf der Prozedur wird das in *@Rueckgabe* gespeicherte Ergebnis in einer einfachen *SELECT*-Abfrage ausgegeben – und das ließe sich wiederum auch von Access aus per Pass-Through-Abfrage realisieren:

```
CREATE PROCEDURE dbo.spProcMitRueckgabewertAufrufen
AS
DECLARE @Rueckgabe int;
EXEC dbo.spProcMitRueckgabewert @Rueckgabewert = @Rueckgabe OUTPUT;
SELECT @Rueckgabe AS Rueckgabewert;
```

Der Aufruf dieser Prozedur erfolgt schließlich mit der folgenden Anweisung – wahlweise im Abfragefenster oder auch über eine Pass-Through-Abfrage von Access aus absetzbar:

```
EXEC dbo.spProcMitRueckgabewertAufrufen;
```

10.3.7 Der RETURN-Wert einer gespeicherten Prozedur

Die Anweisung *RETURN* beendet die Verarbeitung einer gespeicherten Prozedur. Sie können *RETURN* im Quellcode einer Prozedur an mehreren Stellen angeben, was meist in Abhängigkeit mit *IF*-Anweisungen geschehen wird. Erreicht die Prozedur bei der Ausführung eine *RETURN*-Anweisung, ist die Prozedur sofort beendet – eventuell nachfolgende SQL-Anweisungen werden nicht mehr verarbeitet. Folgendes Beispiel beendet die Ausführung der Prozedur, wenn der übergebene Wert kleiner 10 ist. Ist der Wert größer oder gleich 10, könnte die tatsächliche Verarbeitung der Prozedur erfolgen. Um das Beispiel kurz zu halten, geben wir an dieser Stelle nur den übergebenen Wert aus:

```
CREATE PROC dbo.spReturn
@Wert int
AS
IF @Wert < 10
BEGIN
    RETURN
END
-- Hier könnte nun die tatsächliche Verarbeitung folgen ...
SELECT @Wert As Wert;
```

Am Ende einer gespeicherten Prozedur müssen Sie die *RETURN*-Anweisung nicht angeben. Eine Prozedur wird natürlich auch ohne *RETURN* beendet, nachdem sie die letzte Anweisung ausgeführt hat. Und doch kann die Angabe von *RETURN* am Ende einer gespeicherten Prozedur sinnvoll sein, denn die *RETURN*-Anweisung beendet nicht nur die Prozedur, sie liefert auch einen Integer-Wert. Dieser Integer-Wert könnte zum Beispiel ein Kennzeichen sein, ob die Prozedur erfolgreich oder fehlerhaft ausgeführt wurde.

Im nächsten Beispiel erstellen wir eine gespeicherte Prozedur, die eine Fehlerbehandlung mit *TRY/CATCH* beinhaltet sowie einen davon abhängigen *RETURN*-Wert. Mehr zum Thema Fehlerbehandlung lesen Sie im Kapitel »T-SQL-Grundlagen«, Seite 221.

Kapitel 10 Gespeicherte Prozeduren

```
CREATE PROC dbo.spReturnwert
@Wert int
AS
DECLARE @Ergebnis float;
DECLARE @ReturnWert int = 0;
BEGIN TRY
    SET @Ergebnis = 10 / @Wert;
    SELECT @Ergebnis As Ergebnis;
END TRY
BEGIN CATCH
    SET @ReturnWert = ERROR_NUMBER();
END CATCH
RETURN @ReturnWert;
```

Die Prozedur erhält über den Parameter *@Wert* eine Zahl. Diese Zahl wird im *TRY*-Block zum Teilen der Zahl 10 verwendet. Das Ergebnis landet in der Variablen *@Ergebnis*, gefolgt von der Ausgabe dieser Variablen per *SELECT*. Ebenfalls ausgegeben wird der Wert der Variablen *@ReturnWert*, allerdings über die *RETURN*-Anweisung. Der Wert dieser Variablen ist standardmäßig 0, was den Erfolg der Aktion darstellt. Gibt es einen Fehler, erhält die Variable *@ReturnWert* im *CATCH*-Block die entsprechende Fehlernummer. In diesem Fall ist der *RETURN*-Wert der einzige Wert, den die Prozedur liefert. Den *RETURN*-Wert werten Sie dann in der aufrufenden Instanz wie folgt aus – hier am Beispiel eines SQL-Skripts:

```
DECLARE @Returnwert int;
EXEC @Returnwert = dbo.spReturnwert 1;
SELECT @Returnwert As Returnwert;
```

Das Ergebnis beinhaltet zwei Ausgaben: das Ergebnis der Division und den *RETURN*-Wert 0 für Erfolg (siehe Abbildung 10.5). Führen Sie dieselben Anweisungen erneut aus, wobei Sie der Prozedur jetzt den Wert 0 übergeben, erhalten Sie das Ergebnis aus .

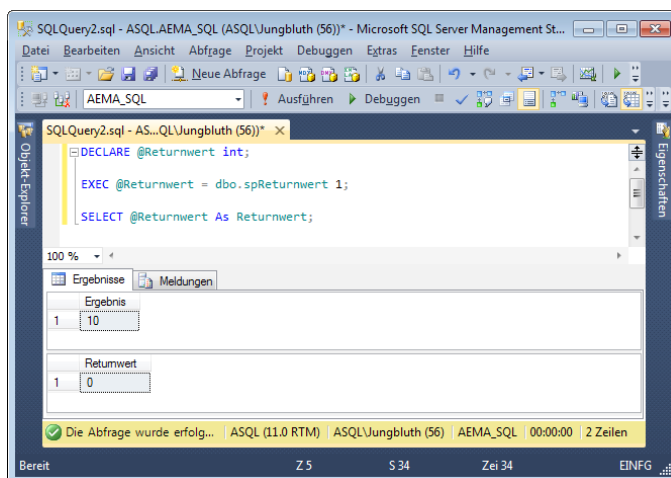


Abbildung 10.5: Die Ausgabe des *RETURN*-Werts

Es besteht nun lediglich aus dem *RETURN*-Wert mit der Fehlernummer des ausgelösten Fehlers. Übrigens liefert eine gespeicherte Prozedur immer einen *RETURN*-Wert, ob Sie nun die Anweisung *RETURN* in der Prozedur verwenden oder nicht.

Welcher Wert dabei ausgegeben wird? Im Erfolgsfall 0 und im Fehlerfall ein Wert ungleich 0. Sie können dies mit der eben erstellten Prozedur *spArtikelNachID* testen.

```
DECLARE @Returnwert int;
EXEC @Returnwert = dbo.spArtikelNachID 2;
SELECT @Returnwert As Returnwert;
```

Als Ausgabe erhalten Sie den Datensatz vom Artikel mit der ID 2 und einen *RETURN*-Wert 0 für die erfolgreiche Ausführung der Prozedur.

10.4 Gespeicherte Prozeduren verwalten

Ihre gespeicherten Prozeduren finden Sie im Objekt-Explorer des SQL Server Management Studios in der jeweiligen Datenbank unter *Programmierbarkeit/Gespeicherte Prozeduren*.

Sollten Sie dort eine Ihrer eben angelegten gespeicherten Prozeduren nicht direkt sehen, müssen Sie nicht nervös werden.

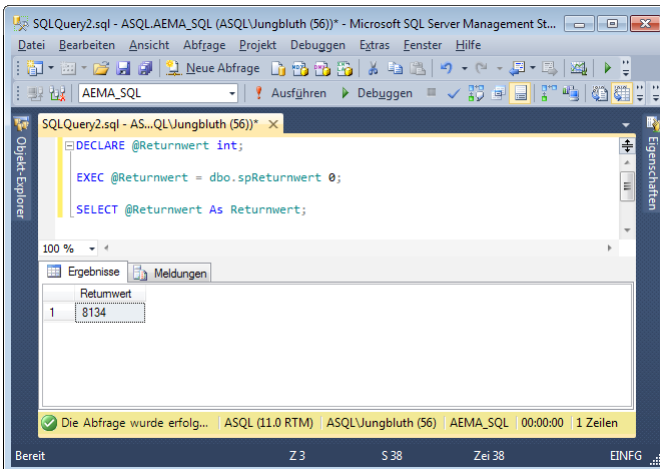


Abbildung 10.6: Der *RETURN*-Wert mit Fehlernummer

Auch das SQL Server Management Studio ist lediglich eine Client-Anwendung, die auf den SQL Server zugreift – und hier und da ist eine manuelle Aktualisierung der Anzeige erforderlich. Hierzu wählen Sie aus dem Kontextmenü des Elements *Gespeicherte Prozeduren* den Eintrag *Aktualisieren*. Anschließend sehen Sie dort auch Ihre neuen gespeicherten Prozeduren.

Gespeicherte Prozeduren ändern

Das Ändern einer gespeicherten Prozedur ist ähnlich der Neuanlage. Auch die Änderung findet in einem neuen Abfragefenster statt. Dieses zeigt Ihnen den Quellcode der gespeicherten Prozedur, nachdem Sie aus dem Kontextmenü der gespeicherten Prozedur den Eintrag *Ändern* gewählt haben. Der einzige Unterschied besteht darin, dass die tatsächliche Anweisung nun nicht mit *CREATE PROCEDURE*, sondern mit *ALTER PROCEDURE* beginnt (siehe Abbildung 10.7). Mit der Anweisung *ALTER* ändern Sie bereits existierende SQL Server-Objekte, wie *ALTER VIEW* für Sichten und *ALTER FUNCTION* für Funktionen.

Mag das Schlüsselwort *CREATE* noch leicht verständlich sein, wird *ALTER* als Übersetzung für *Ändern* eher selten verwendet. Hier eine kleine Eselsbrücke, wie Sie sich am Anfang den Befehl *ALTER* als Ändern-Befehl merken können: *Was ändert sich ständig? Ihr Alter*. Nun gibt es bei den eben angelegten gespeicherten Prozeduren auch tatsächlich etwas zu ändern. Es ist empfehlenswert, jede Prozedur mit der folgenden Anweisung zu ergänzen:

```
SET NOCOUNT ON;
```

Mit *SET NOCOUNT ON* deaktivieren Sie die Meldungen, wie viele Datensätze von der Prozedur verarbeitet wurden. Wenn Sie beispielsweise die Prozedur *spArtikelNachID* ausführen, sehen Sie in der Registerkarte *Meldungen* die folgende Ausgabe:

```
(1 Zeile(n) betroffen)
```

Diese Meldung erhalten Sie zu jeder SQL-Anweisung, die die gespeicherte Prozedur ausführt. Ebenso wie die Meldung in unserem Beispiel an die Client-Anwendung SQL Server Management Studio übergeben wird, erhält auch Ihre Access-Applikation diese Meldungen. Um diesen unnötigen Traffic zu sparen, schreiben Sie in jeder Ihrer gespeicherten Prozeduren als erste Anweisung *SET NOCOUNT ON*. Mit der folgenden Anweisung ergänzen Sie die gespeicherte Prozedur *spArtikelNachID* um *SET NOCOUNT ON*.

```
ALTER PROCEDURE dbo.spArtikelNachID
@ArtikelID int
AS
SET NOCOUNT ON;
SELECT * FROM dbo.tblArtikel WHERE ArtikelID = @ArtikelID;
```

Die Änderung bestätigen Sie wieder durch Ausführen der Abfrage mittels *F5*. Führen Sie nun die Prozedur erneut aus, werden Sie in der Registerkarte *Meldungen* die Meldung von eben nicht mehr sehen. Vielleicht finden Sie aber die Anzahl der betroffenen Datensätze interessant und möchten diese in Access weiterverarbeiten. Auch in diesem Fall sollten Sie die Meldungen ausschalten. Zur Ermittlung der Anzahl der betroffenen Datensätze verwenden Sie besser die Systemvariable *@@ROWCOUNT*. Diese liefert Ihnen die Anzahl nach jeder SQL-Anweisung.

Das folgende Beispiel erweitert die Prozedur *spAlleArtikel* um die Anzahl der ermittelten Datensätze, die dann über den *RETURN*-Wert zurückgegeben wird. Das Ergebnis der Ausführung dieser Prozedur sehen Sie in Abbildung 10.8.

```
ALTER PROCEDURE dbo.spAlleArtikel
AS
SET NOCOUNT ON;
SELECT * FROM dbo.tblArtikel;
RETURN @@ROWCOUNT;
```

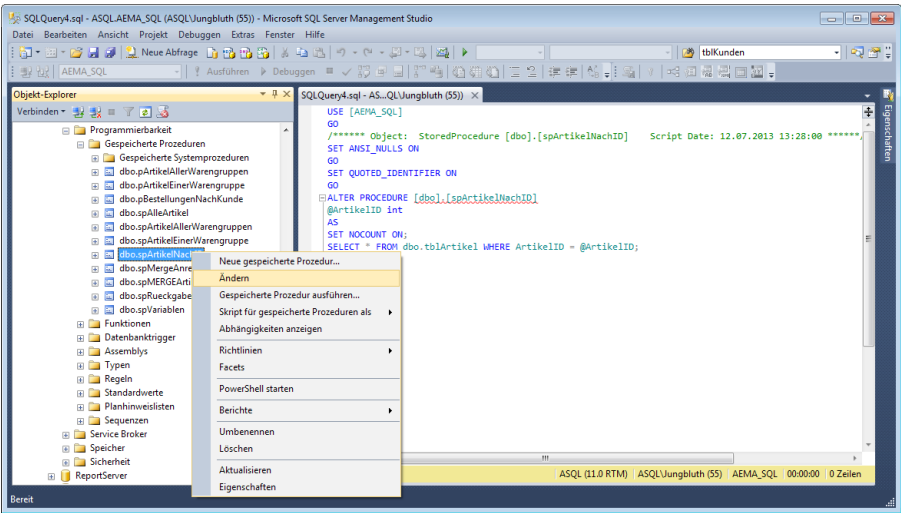


Abbildung 10.7: Das Ändern einer gespeicherten Prozedur

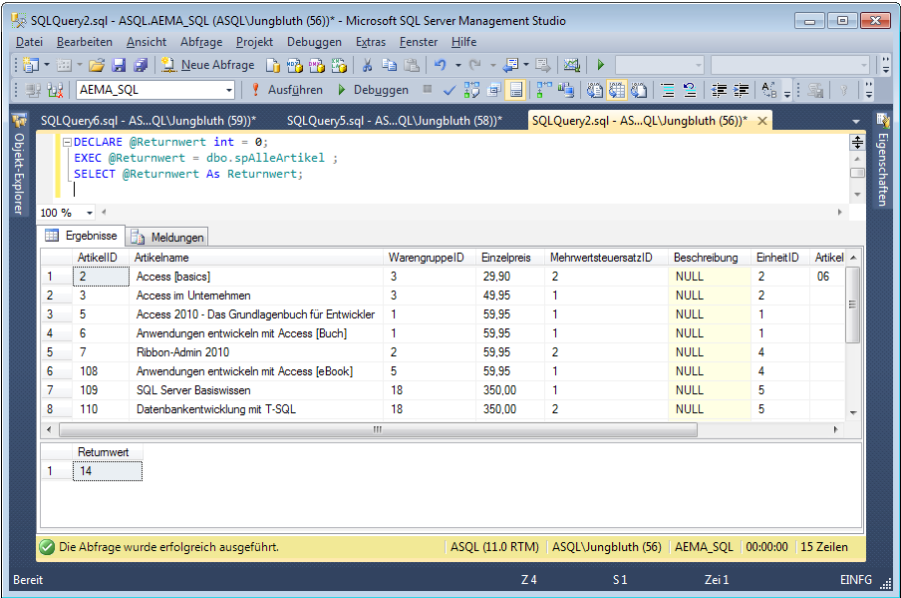


Abbildung 10.8: @@ROWCOUNT im Einsatz

11 Funktionen

Bestimmt enthält Ihre Access-Applikation einige VBA-Funktionen. Funktionen, in denen Sie wiederkehrende Funktionalität, wie Berechnungen, Datenermittlungen und Programmlogiken, ausgelagert haben, um diese dann an mehreren Stellen im VBA-Code oder in Access-Abfragen wiederzuverwenden. Auch SQL Server bietet Ihnen die Möglichkeit dieser modularen Entwicklungsmethode.

Mit den Funktionen im SQL Server lassen sich ebenfalls wiederkehrende Berechnungen, Programmlogiken und Datenermittlungen kapseln und in gespeicherten Prozeduren, Sichten und Triggern wiederverwenden. Funktionen werden wie die anderen SQL Server-Objekte in T-SQL programmiert. Sie unterstützen Parameter und liefern je nach Typ einen Wert oder eine Ergebnismenge. Insgesamt gibt es drei verschiedene Typen, die Sie in diesem Kapitel kennen lernen.

In Access lassen sich die Funktionen vom SQL Server nicht direkt verwenden. Zwar liefern zwei der drei Funktionstypen Ergebnismengen, dennoch können Sie diese Funktionen nicht wie Tabellen oder Sichten in Access einbinden. Es bleibt Ihnen eigentlich nur der Weg über eine Pass-Through-Abfrage; dieser ist aber nicht empfehlenswert.

Funktionen lassen sich nur in Verbindung mit SQL-Anweisungen nutzen – und wie Sie im Kapitel »Gespeicherte Prozeduren«, Seite 257, gelernt haben, sollten Sie aus Gründen der Performance in Pass-Through-Abfragen keine SQL-Anweisungen verwenden, sondern gespeicherte Prozeduren. Folglich nutzen Sie die Funktionen lediglich in Ihren gespeicherten Prozeduren, Sichten und Triggern und somit nur indirekt in Access.

11.1 Vorteile von Funktionen

Ein großer Vorteil der Funktionen ist die Möglichkeit der modularen Entwicklung. Wiederkehrende Berechnungen zum Beispiel müssen Sie nicht in jeder SQL-Anweisung angeben, in der Sie diese benötigen, sondern Sie definieren diese nur ein einziges Mal in einer Funktion. Diese Funktion verwenden Sie dann an den jeweiligen Stellen, an denen Sie sonst die Berechnung angegeben hätten. Ein gutes Beispiel für eine wiederkehrende Berechnung ist die Positionssumme einer Bestellung, die sich aus Einzelpreis, Menge und Rabatt ergibt. Oder Sie erstellen eine Funktion, die Ihnen aus dem übergebenen Nachnamen und Vornamen eine Zeichenfolge mit Vor- und Nachnamen erstellt. Eine solche Funktion lässt sich dann gleich für mehrere Zwecke verwenden, beispielsweise für die Namen der Mitarbeiter und für die der Ansprechpartner von Lieferanten und Kunden. Diese Art der Funktionen sind typische Beispiele für eine Skalarfunktion.

Nicht nur Berechnungen, sondern auch Datenermittlungen können von Funktionen übernommen werden. Die Tabellenwertfunktionen liefern Ergebnismengen und lassen sich dabei wie Sichten und Tabellen ansprechen. Benötigen Sie zum Beispiel eine Auflistung mit Kunden, die

einen Newsletter beziehen möchten, und eine weitere mit den Kunden, die den Newsletter nicht erhalten wollen, können Sie beide Ausgaben mit einer Tabellenwertfunktion realisieren. Diese liefert abhängig vom übergebenen Parameter die Kunden mit Newsletteranmeldung oder die Kunden ohne Newsletteranmeldung.

Die weiteren Vorteile von Funktionen sind schnell beschrieben: Funktionen bieten die gleichen Vorteile wie gespeicherte Prozeduren. Sie können mit Funktionen die Geschäftslogik im SQL Server abbilden und erhalten dabei eine bessere Performance, denn auch für die Funktionen werden Ausführungspläne erstellt, im Prozedurcache gespeichert und bei jedem Einsatz der Funktion wiederverwendet. Mehr zu den Ausführungsplänen lesen Sie im Kapitel »FAQ«, Seite 19.

11.2 Skalarfunktion

Die Skalarfunktion liefert – nomen es omen – einen einzelnen Wert. Bei diesem Wert kann es sich um eine Konstante handeln, einem Ergebnis einer einfachen *SELECT*-Anweisung, einer Berechnung anhand der übergebenen Parameter oder auch um das Ergebnis einer komplexen Logik, realisiert mit mehreren SQL-Anweisungen, Variablen und Programmstrukturen wie *IF...ELSE* und *WHILE*.

Nicht nur der Inhalt des Werts, auch dessen Datentyp wird über die Skalarfunktion definiert. Dabei können Sie fast alle Datentypen nutzen, die Ihnen auch bei der Definition einer Tabelle zur Verfügung stehen. Lediglich die veralteten Datentypen für binäre Werte (*Image*, *nText* und *Text*) wie auch *Table*-Variablen werden nicht unterstützt. Skalarfunktionen sind vielseitig einsetzbar:

- » Als Spalten in *SELECT*-Anweisungen
- » Als Filterkriterien in *WHERE*-Bedingungen
- » Als Werte in *INSERT*- und *UPDATE*-Anweisungen
- » Als Standardwerte für Spalten von Tabellen
- » Zur Prüfung von Einschränkungen von Spalten und Tabellen
- » Zur Initialisierung von Variablen
- » Zur Steuerung des Programmflusses in *IF*- und *WHILE*-Anweisungen

11.2.1 Skalarfunktion anlegen

Zum Erstellen einer Skalarfunktion können Sie wie bei den gespeicherten Prozeduren eine Vorlage verwenden. Sie erhalten diese Vorlage, indem Sie im Objekt-Explorer den Kontextmenüeintrag *Neue Skalarfunktion* des Elements *Programmierbarkeit/Funktionen/Skalarfunktionen* wählen. Leider ist diese Vorlage wie bei den gespeicherten Prozeduren eher verwirrend als hilfreich.

Am besten legen Sie eine Skalarfunktion in einem neuen Abfragefenster an. Dazu markieren Sie im Objekt-Explorer die Datenbank, in der die Skalarfunktion gespeichert werden soll, und wählen dort im Kontextmenü den Eintrag *Neue Abfrage* aus.

Im neuen Abfragefenster geben Sie dann den entsprechenden *CREATE FUNCTION*-Befehl an, ergänzt mit dem Schemanamen und dem Namen der Skalarfunktion. Den Typ der Skalarfunktion müssen Sie dabei nicht angeben. Dieser ergibt sich aus der weiteren Syntax.

Im folgenden Beispiel möchten wir eine Skalarfunktion erstellen, die das aktuelle Tagesdatum ausgibt. Dazu wird der Rückgabewert der Systemfunktion *GETDATE()*, die den aktuellen Zeitpunkt liefert, in den Datentyp *Date* konvertiert. Die Skalarfunktion nennen wir *sfAktuellesDatum*. Die erste Zeile unserer neuen Skalarfunktion lautet somit wie folgt:

```
CREATE FUNCTION dbo.sfAktuellesDatum
```

Um beim späteren Programmieren die Skalarfunktion als solche zu erkennen, empfiehlt sich bei der Namensvergabe die Verwendung eines entsprechenden Präfix – zum Beispiel *sf*.

Als Nächstes ist die Definition der Eingabeparameter an der Reihe. Die Angabe der Parameter erfolgt immer in runden Klammern. Dies gilt auch für Skalarfunktionen ohne Parameter. Ob Sie also in Ihrer Skalarfunktion Parameter verwenden oder nicht, die runden Klammern sind Pflicht. In unserem Beispiel verwenden wir keine Parameter.

Nach den Klammern folgt die Anweisung *RETURNS*, mit der Sie den Datentyp des Skalarwerts – des Rückgabewerts der Skalarfunktion – festlegen. Das Datum des aktuellen Zeitpunkts liefern wir im Datentyp *Date*:

```
RETURNS date
```

Nun kommt noch das Schlüsselwort *AS*, das Sie bereits von den gespeicherten Prozeduren kennen, und anschließend die Programmlogik zur Ermittlung des Skalarwerts innerhalb eines *BEGIN...END*-Blocks. In unserem Beispiel besteht die Logik lediglich aus der Konvertierung des Rückgabewerts der Systemfunktion *GETDATE()* in den Datentyp *Date*.

Das Ergebnis einer Skalarfunktion wird unabhängig vom Datentyp über *RETURN* ausgegeben. Da in unserem Beispiel die Programmlogik lediglich aus der Rückgabe eines konvertierten Werts besteht, verbinden wir die Konvertierung direkt mit der *RETURN*-Anweisung:

```
CREATE FUNCTION dbo.sfAktuellesDatum ()  
RETURNS date  
AS  
BEGIN  
    RETURN CAST(GETDATE() As Date);  
END
```

Diese recht überschaubare Skalarfunktion legen Sie nun mit der Taste *F5* an. Anschließend ist sie im Element *Programmierbarkeit/Funktionen/Skalarfunktionen* zu sehen. Sollte dies nicht der Fall sein, aktualisieren Sie die Ansicht über den Kontextmenüeintrag *Aktualisieren* des Elements

Kapitel 11 Funktionen

Skalarfunktionen. Die einfachste Verwendung dieser Skalarfunktion ist ein simpler Aufruf per **SELECT**:

```
SELECT dbo.sfAktuellesDatum() As Tagesdatum;
```

Beachten Sie, dass Sie beim Aufruf einer Skalarfunktion immer das Schema und die Klammern der Parameter angeben. Beide sind zwingend erforderlich, auch wenn Sie nur das Standard-schema *dbo* verwenden beziehungsweise keine Parameter definiert sind. Führen Sie nun die Skalarfunktion unter Berücksichtigung dieser beiden Syntaxvorschriften aus, erhalten Sie als Ergebnis das Datum von heute im ISO-Format (siehe Abbildung 11.1).

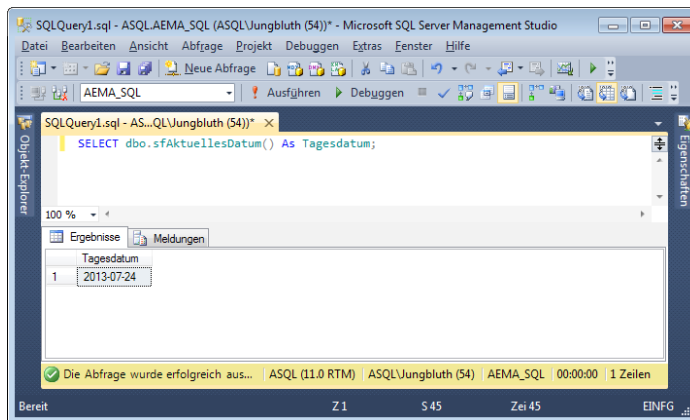


Abbildung 11.1: Eine Skalarfunktion im Einsatz

Die Skalarfunktion können Sie ebenso in einer **WHERE**-Bedingung verwenden. Folgende Abfrage liefert Ihnen die Bestellungen von heute:

```
SELECT * FROM dbo.tb1Bestellungen WHERE Bestelldatum = dbo.sfAktuellesDatum();
```

11.2.2 Skalarfunktion mit Parametern

Die Parameter einer Skalarfunktion werden direkt nach dem Funktionsnamen in den bereits erwähnten runden Klammern angegeben. Dabei wird jeder Parameter mit einem Datentyp deklariert, sein Name beginnt immer mit einem **@**-Zeichen und darf keine Sonderzeichen mit Ausnahme des Unterstrichs enthalten und mehrere Parameter sind mit einem Komma voneinander zu trennen.

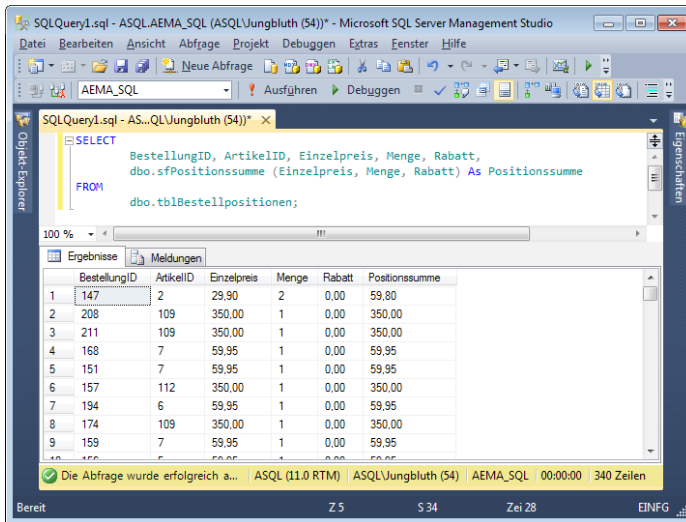
Im nächsten Beispiel soll eine Skalarfunktion die Summe einer Bestellposition ermitteln. Hierzu erhält die Skalarfunktion über Parameter die Werte *Einzelpreis*, *Menge* und *Rabatt*, berechnet die Positionssumme und gibt das Ergebnis im Datentyp *Money* aus:

```
CREATE FUNCTION dbo.sfPositionssumme (  
    @Einzelpreis money,
```



```
@Menge int,
@Rabatt money)
RETURNS money
AS
BEGIN
    RETURN (@Einzelpreis * @Menge) - @Rabatt;
END
```

Diese Skalarfunktion liefert Ihnen nun zu jeder Bestellposition die Positionssumme, wie in Abbildung 10.2 zu sehen ist.



BestellungID	ArtikelID	Einzelpreis	Menge	Rabatt	Positionssumme
147	2	29,90	2	0,00	59,80
208	109	350,00	1	0,00	350,00
211	109	350,00	1	0,00	350,00
168	7	59,95	1	0,00	59,95
151	7	59,95	1	0,00	59,95
157	112	350,00	1	0,00	350,00
194	6	59,95	1	0,00	59,95
174	109	350,00	1	0,00	350,00
159	7	59,95	1	0,00	59,95

Abbildung 11.2: Die Berechnung der Positionssumme durch eine Skalarfunktion

11.2.3 Skalarfunktion mit mehreren Anweisungen

Die bisher gezeigten Beispiele ermitteln den Skalarwert anhand einer einzigen Anweisung. Nun kann eine Skalarfunktion, wie oben bereits beschrieben, auch mehr als eine Anweisung enthalten. Auch diesen Fall möchten wir an einem Beispiel zeigen. Dieses Mal ermittelt die Skalarfunktion den Namen der Warengruppe zu einer übergebenen *WarengruppeID*:

```
CREATE FUNCTION dbo.sfWarengruppe (
@WarengruppeID int)
RETURNS nvarchar(255)
AS
BEGIN
    DECLARE @Warengruppe nvarchar(255);
    SELECT @Warengruppe = Warengruppe FROM dbo.tblWarengruppen
    WHERE WarengruppeID = @WarengruppeID;
    RETURN @Warengruppe;
END
```

Kapitel 11 Funktionen

Die Skalarfunktion *sfWarengruppe* empfängt über den Parameter *@WarengruppeID* einen Wert vom Datentyp *Integer* und liefert nach der Verarbeitung den Skalarwert im Datentyp *nVarChar(255)*.

Die im *BEGIN...END*-Block enthaltene Programmlogik startet mit der Definition der Variablen *@Warengruppe*. Da sowohl die Ermittlung wie auch die Ausgabe des Skalarwerts über diese Variable erfolgt, erhält sie den gleichen Datentyp wie die Skalarfunktion selbst – in diesem Fall *nVarChar(255)*.

Dann wird mit der im Parameter *@WarengruppeID* enthaltenen ID der entsprechende Datensatz in der Tabelle *tblWarengruppen* ermittelt und dabei der Wert der Spalte *Warengruppe* der Variablen *@Warengruppe* zugewiesen. Abschließend folgt die Ausgabe über die *RETURN*-Anweisung mit dem Wert der Variablen.

Verwenden Sie diese Skalarfunktion in der folgenden Abfrage, erhalten Sie neben den Informationen zum Artikel auch die Bezeichnung der Warengruppe.

```
SELECT ArtikelID, Artikelname, dbo.sfWarengruppe(WarengruppeID) As Warengruppe
FROM dbo.tblArtikel;
```

11.2.4 Skalarfunktion ändern

Den Quellcode einer Skalarfunktion ändern Sie in derselben Art und Weise, wie Sie ihn erstellt haben – mit einem T-SQL-Skript in einem Abfragefenster. Sie erhalten den Quellcode über den Eintrag *Ändern* aus dem Kontextmenü der Skalarfunktion.

Das Abfragefenster zeigt fast den gleichen Code, den Sie beim Erstellen der Funktion eingegeben haben. Da Sie aber über diesen Weg keine Funktion erstellen, sondern eine bestehende ändern möchten, enthält der Quellcode nun anstelle der Anweisung *CREATE FUNCTION* die Anweisung *ALTER FUNCTION*. Nachdem Sie den Quellcode der Skalarfunktion an die neuen Anforderungen angepasst haben, führen Sie das T-SQL-Skript mit der Taste *F5* aus, worauf die Skalarfunktion neu erstellt und gespeichert wird. Sie kennen diese Vorgehensweise bereits von den gespeicherten Prozeduren.

11.3 Tabellenwertfunktionen

Im Gegensatz zu Skalarfunktionen liefern Tabellenwertfunktionen keinen einzelnen Wert, sondern einen oder mehrere Datensätze. SQL Server stellt Ihnen zwei Typen von Tabellenwertfunktionen zur Verfügung. Beide sind mit einer Sicht vergleichbar: sie liefern auf Basis einer *SELECT*-Anweisung Daten einer oder mehrerer Tabellen, werden in SQL-Anweisungen wie eine Tabelle oder Sicht angesprochen und lassen sich dabei auch mit anderen Tabellen verknüpfen.

Anders als Sichten unterstützen Tabellenwertfunktionen jedoch Parameter und eine der beiden ist nicht einmal auf eine *SELECT*-Anweisung begrenzt:

- » Die *Tabellenwertfunktion mit mehreren Anweisungen* verrät ihre Möglichkeiten schon durch ihre Bezeichnung. Ähnlich einer Skalarfunktion können Sie mehrere SQL-Anweisungen sowie Variablen, temporäre Tabellen, Programmstrukturen et cetera zur Datenermittlung nutzen – nur dass das Ergebnis in diesem Fall ein oder mehrere Datensätze beinhaltet. Die Ausgabe der Datensätze erfolgt über eine *Table*-Variable, deren Definition zwingend erforderlich ist. Im Laufe der Datenermittlung werden dieser *Table*-Variablen Daten hinzugefügt sowie gegebenenfalls dort bereits enthaltene Daten geändert und gelöscht.
- » Die *Inline-Tabellenwertfunktion* erlaubt lediglich eine einzelne *SELECT*-Anweisung und ist somit der Beschreibung einer Sicht recht nahe. Die Definition einer *Table*-Variablen zur Ausgabe der Daten ist hier nicht notwendig, sie ergibt sich aus der *SELECT*-Anweisung der Funktion.

Beide Typen der Tabellenwertfunktionen sind im Objekt-Explorer im Element *Programmierbarkeit/Funktionen/Tabellenwertfunktionen* zusammengefasst. Hier können Sie auch neue Tabellenwertfunktionen anlegen und bestehende ändern.

Das Anlegen und Ändern einer Tabellenwertfunktion erfolgt wieder in einem Abfragefenster und unterscheidet sich nicht von dem einer Skalarfunktion. Lediglich die Syntax ist bei jedem Typ unterschiedlich.

11.3.1 Inline-Tabellenwertfunktion

Die einfachste Variante der Funktionen ist die *Inline-Tabellenwertfunktion*, besteht sie doch lediglich aus einer einzelnen *SELECT*-Anweisung. Dass es sich bei der Funktion um eine Inline-Tabellenwertfunktion handelt, ergibt sich aus der Syntax, die wir Ihnen nun an dem folgenden Beispiel zeigen:

```
CREATE FUNCTION dbo.ifNewsletterAbonnenten (
    @Aktiv bit)
RETURNS TABLE
AS
RETURN
(SELECT KundeID, Kundenbezeichnung FROM dbo.tblKunden WHERE Newsletter = @Aktiv);
```

Zum Anlegen einer Inline-Tabellenwertfunktion geben Sie wie bei der Skalarfunktion den Befehl *CREATE FUNCTION* an, gefolgt vom Namen der Funktion inklusive Schema und der Parameterdefinition. Bei der Namensvergabe wäre das Präfix *if* sinnvoll, um die Funktion beim späteren Gebrauch als Inline-Tabellenwertfunktion zu erkennen.

Die *RETURNS*-Anweisung beinhaltet den ersten Unterschied zur Syntax einer Skalarfunktion. Bei der Inline-Tabellenwertfunktion ist als Ausgabe lediglich der Datentyp *Table* zugelassen.

Das folgende Schlüsselwort *AS* leitet wie gewohnt den eigentlichen Quellcode der Funktion ein. Dieser besteht jedoch lediglich aus dem Befehl *RETURN* gefolgt von einer *SELECT*-Anweisung. Mehr benötigt eine Inline-Tabellenwertfunktion auch nicht.

Kapitel 11 Funktionen

Wie bei der Skalarfunktion ist bei der Inline-Tabellenwertfunktion der Befehl *RETURN* zuständig für die Ausgabe der ermittelten Daten. Und da eine Inline-Tabellenwertfunktion per Definition nur aus einer einzigen *SELECT*-Anweisung besteht, reicht das *RETURN* mit der in Klammern zugewiesenen *SELECT*-Anweisung völlig aus.

Die in diesem Beispiel verwendete Inline-Tabellenwertfunktion liefert je nach übergebenem Parameter alle Kunden, die einen Newsletter beziehen oder eben nicht.

Der folgende Aufruf zeigt die bereits erfassten Newsletter-Abonnenten:

```
SELECT KundeID, Kundenbezeichnung FROM dbo.ifNewsletterAbonnenten(1);
```

Und diese Abfrage liefert die Kunden, die man noch als Newsletter-Abonnenten gewinnen könnte:

```
SELECT KundeID, Kundenbezeichnung FROM dbo.ifNewsletterAbonnenten(0);
```

11.3.2 Tabellenwertfunktion mit mehreren Anweisungen

Kommen wir zur dritten Variante der Funktionen im SQL Server. Die *Tabellenwertfunktion mit mehreren Anweisungen* ist eine Mischung aus Skalarfunktion und Inline-Tabellenwertfunktion.

Sie liefert wie die Inline-Tabellenwertfunktion einen oder mehrere Datensätze, wobei die Datenermittlung wie bei der Skalarfunktion durch eine oder mehrere Anweisungen erfolgen kann. Auch hier stehen Ihnen wieder fast alle Möglichkeiten von T-SQL zur Verfügung. Auf die Ausnahmen kommen wir später noch zurück.

Die *Tabellenwertfunktion mit mehreren Anweisungen* definiert sich wie die anderen beiden Funktionen durch ihre eigene Syntax. Namensvergabe und Parameterdefinition sind zwar noch gleich, aber bereits bei der Definition der Rückgabe mit der Anweisung *RETURNS* gibt es den ersten Unterschied.

Als Datentyp ist hier lediglich eine *Table*-Variable erlaubt, die zudem an dieser Stelle benannt und definiert werden muss.

Es folgt wieder das Schlüsselwort *AS* und der *BEGIN...END*-Block mit der Logik zur Datenermittlung. Im *BEGIN...END*-Block füllen Sie nun mit den entsprechenden Anweisungen nach und nach die *Table*-Variable. Eventuell ändern oder löschen Sie abhängig von der Programmlogik auch wieder bereits dort gespeicherte Daten.

Ebenso können Sie zur Ermittlung des Ergebnisses Variablen, weitere *Table*-Variablen, temporäre Tabellen sowie Programmstrukturen wie *IF...ELSE* und Schleifen nutzen. Sind alle Daten beisammen, wird die Verarbeitung mit *RETURN* beendet und der Inhalt der *Table*-Variable ausgegeben.

Im folgenden Beispiel ermittelt die Tabellenwertfunktion den Umsatz eines Kunden für ein bestimmtes Jahr und das Jahr davor:

```

CREATE FUNCTION dbo.tfUmsatzKundeJahr (
    @KundeID int,
    @Jahr int)
RETURNS @tabAusgabe TABLE (
    KundeID int,
    Kundenbezeichnung nvarchar(255),
    Jahr int,
    UmsatzJahr money,
    UmsatzVorJahr money)
AS
BEGIN
    DECLARE @AnfangJahr datetime = CAST(@Jahr As char(4)) + '0101',
            @EndeJahr datetime = CAST(@Jahr As char(4)) + '1231',
            @UmsatzJahr money,
            @UmsatzVorJahr money;

    -- Summe Rechnungen zum Jahr aus Parameter @Jahr
    SELECT  @UmsatzJahr = Sum((Menge*Einzelpreis)-Rabatt)
    FROM    dbo.tblBestellungen INNER JOIN dbo.tblBestellpositionen
            ON dbo.tblBestellungen.BestellungID = dbo.tblBestellpositionen.
    BestellungID
    WHERE   dbo.tblBestellungen.KundeID = @KundeID
            AND RechnungAm BETWEEN @AnfangJahr AND @EndeJahr;

    -- Jahreswerte für Vorjahr erstellen
    SELECT  @AnfangJahr = CAST((@Jahr - 1) As char(4)) + '0101',
            @EndeJahr = CAST((@Jahr - 1) As char(4)) + '1231';

    -- Summe Rechnungen zum Vorjahr
    SELECT  @UmsatzVorJahr = Sum((Menge*Einzelpreis)-Rabatt)
    FROM    dbo.tblBestellungen INNER JOIN dbo.tblBestellpositionen
            ON dbo.tblBestellungen.BestellungID = dbo.tblBestellpositionen.
    BestellungID
    WHERE   dbo.tblBestellungen.KundeID = @KundeID
            AND RechnungAm BETWEEN @AnfangJahr AND @EndeJahr;

    -- Kunde ermitteln und mitsamt Summen in @tabAusgabe speichern
    INSERT INTO @tabAusgabe (KundeID, Kundenbezeichnung, Jahr, UmsatzJahr,
    UmsatzVorJahr)
        SELECT KundeID, Kundenbezeichnung, @Jahr, @UmsatzJahr, @UmsatzVorJahr
        FROM dbo.tblKunden WHERE KundeID = @KundeID;

    RETURN
END

```

Der Quellcode dieser Tabellenwertfunktion beginnt wie bei den vorherigen Funktionen mit dem Befehl *CREATE FUNCTION* ergänzt mit dem Schema und dem Funktionsnamen. Als Präfix für eine *Tabellenwertfunktion mit mehreren Anweisungen* empfiehlt sich das Kürzel *tf*.

Nach der Benennung folgt die Parameterdefinition. In diesem Fall gibt es zwei Parameter: den Parameter *@KundeID* für die ID des Kunden und den Parameter *@Jahr* für das Jahr, zu dem der Umsatz sowie der Umsatz des vorangegangenen Jahres ermittelt werden soll.

```

CREATE FUNCTION dbo.tfUmsatzKundeJahr (
    @KundeID int,
    @Jahr int)

```

Kapitel 11 Funktionen

Die Ausgabe wird eingeleitet mit der Anweisung *RETURNS*, welche an dieser Stelle nur eine *Table-Variable* zulässt. In unserem Beispiel ist es eine *Table-Variable* mit der Bezeichnung *@tabAusgabe* und den Spalten *KundeID*, *Kundenbezeichnung*, *Jahr*, *UmsatzJahr* und *UmsatzVorjahr*.

```
RETURNS @tabAusgabe TABLE (  
    KundeID int,  
    Kundenbezeichnung nvarchar(255),  
    Jahr int,  
    UmsatzJahr money,  
    UmsatzVorjahr money)
```

Nachdem die Ausgabe definiert ist, folgt das Schlüsselwort *AS* und in dem *BEGIN...END*-Block die eigentliche Verarbeitung der Tabellenwertfunktion. Diese beginnt mit der Deklaration der Variablen.

Die Variablen *@AnfangJahr* und *@EndeJahr* speichern den 01. Januar und den 31. Dezember von dem Jahr, das über den Parameter *@Jahr* übergeben wird.

Die beiden Variablen *@UmsatzJahr* und *@UmsatzVorJahr* sind für die jeweiligen Umsatzzahlen vorgesehen, deren Ermittlung in den nächsten Anweisungen stattfindet.

```
DECLARE @AnfangJahr datetime = CAST(@Jahr As char(4)) + '0101',  
        @EndeJahr datetime = CAST(@Jahr As char(4)) + '1231',  
        @UmsatzJahr money,  
        @UmsatzVorJahr money;
```

Die Ermittlung des Umsatzes für das Jahr aus dem Parameter *@Jahr* erfolgt über eine *SELECT*-Anweisung, welche die Variablen *@AnfangJahr* und *@EndeJahr* sowie den Parameter *@KundeID* als Filterkriterium nutzt und anhand dessen die Gesamtsumme der Bestellpositionen ermittelt.

Das Ergebnis wird dann in der Variablen *@UmsatzJahr* gespeichert.

```
SELECT @UmsatzJahr = Sum((Menge*Einzelpreis)-Rabatt)  
FROM dbo.tblBestellungen INNER JOIN dbo.tblBestellpositionen  
ON dbo.tblBestellungen.BestellungID = dbo.tblBestellpositionen.BestellungID  
WHERE dbo.tblBestellungen.KundeID = @KundeID  
AND RechnungAm BETWEEN @AnfangJahr AND @EndeJahr;
```

Um nun die Summe vom Vorjahr zu ermitteln, erhalten die Variablen *@AnfangJahr* und *@EndeJahr* den 01. Januar und den 31. Dezember des Vorjahres. Das Vorjahr ergibt sich aus dem im Parameter *@Jahr* enthaltenen Wert minus 1.

```
SELECT @AnfangJahr = CAST((@Jahr - 1) As char(4)) + '0101',  
        @EndeJahr = CAST((@Jahr - 1) As char(4)) + '1231';
```

Mit den neuen Werten in den Variablen erfolgt nun ein zweites Mal die Ermittlung des Umsatzes. Jetzt landet das Ergebnis in der Variablen *@UmsatzVorJahr*.

```

SELECT @UmsatzVorJahr = Sum((Menge*Einzelpreis)-Rabatt)
FROM dbo.tblBestellungen INNER JOIN dbo.tblBestellpositionen
ON dbo.tblBestellungen.BestellungID = dbo.tblBestellpositionen.BestellungID
WHERE  dbo.tblBestellungen.KundeID = @KundeID
AND RechnungAm BETWEEN @AnfangJahr AND @EndeJahr;

```

Die ermittelten Werte werden dann in die *Table-Variable* geschrieben, ergänzt mit der Kundenbezeichnung zur übergebenen *KundeID*.

```

INSERT INTO @tabAusgabe (KundeID, Kundenbezeichnung, Jahr, UmsatzJahr, UmsatzVorJahr)
SELECT KundeID, Kundenbezeichnung, @Jahr, @UmsatzJahr, @UmsatzVorJahr
FROM dbo.tblKunden WHERE KundeID = @KundeID;

```

Hiermit ist die Datensammlung bereits komplett, weshalb an dieser Stelle die Funktion mit der Anweisung *RETURN* abgeschlossen wird. Im Gegensatz zu den anderen Funktionen darf der *RETURN*-Anweisung weder ein Wert noch eine *SELECT*-Anweisung zugewiesen werden. An dieser Stelle beendet *RETURN* lediglich die Datensammlung in der *Table-Variable* und gibt deren Inhalt aus.

Mit der Ihnen bereits bekannten Taste *F5* legen Sie die Funktion *tfUmsatzKundeJahr* an.

Die neue *Tabellenwertfunktion mit mehreren Anweisungen* verwenden Sie wie eine Tabelle oder eine Sicht. Folgender Aufruf liefert Ihnen den Umsatz des Kunden mit der ID 96 für die Jahre 2012 und 2011:

```

SELECT KundeID, Kundenbezeichnung, Jahr, UmsatzJahr, UmsatzVorJahr
FROM dbo.tfUmsatzKundeJahr(96, 2012);

```

11.4 Limitationen

Die Skalarfunktionen und Tabellenwertfunktionen unterliegen einigen Einschränkungen:

- » Die Ausgabe von Meldungen mit *PRINT* oder *RAISERROR* wird nicht unterstützt.
- » *TRY...CATCH*-Anweisungen zur Fehlerbehandlung sind nicht erlaubt.
- » Das Hinzufügen, Ändern und Löschen von Daten ist nicht möglich, außer es handelt sich um *Table-Variablen* oder temporäre Tabellen, die in der Funktion erstellt wurden.
- » Der Befehl *EXECUTE* darf in einer Funktion nicht verwendet werden, was das Ausführen von gespeicherten Prozeduren verhindert.

Ergänzend dazu ist noch die Limitation bei den Parameterwerten einer Tabellenwertfunktion zu erwähnen.

Diese können nur Konstanten enthalten, eine Parameterübergabe mit Werten aus Spalten einer *SELECT*-Anweisung ist nicht möglich. In Abbildung 10.3 sehen Sie den Versuch einer solchen Abfrage und die entsprechende Fehlermeldung.

Kapitel 11 Funktionen

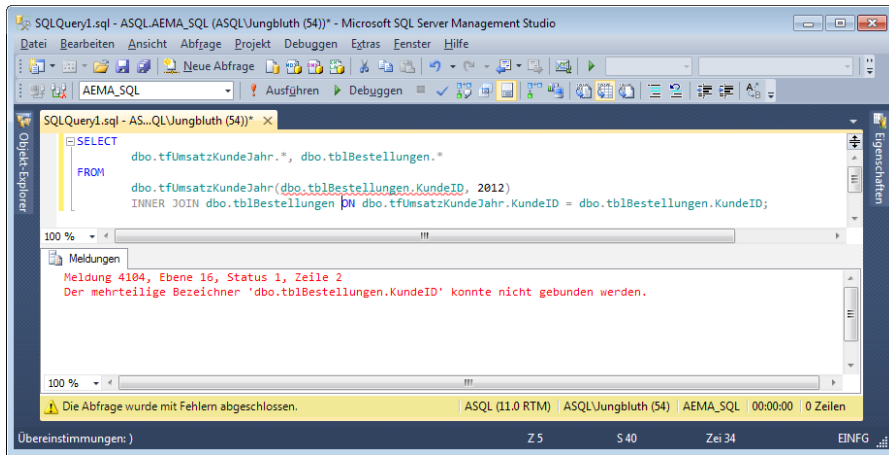


Abbildung 11.3: Falsche Verwendung einer Tabellenwertfunktion

11.5 Performance

So verführerisch der Einsatz von Funktionen auch sein mag, Sie sollten immer die Performance berücksichtigen. Was es zu beachten gibt, zeigen wir nun an zwei kleinen Beispielen. In Beispiel 1 verwenden wir unsere erste Skalarfunktion *sfAktuellesDatum*, um die aktuellen Bestellungen auszugeben:

```
SELECT * FROM dbo.tblBestellungen WHERE Bestelldatum = dbo.sfAktuellesDatum();
```

Bei dieser Anweisung wird die Skalarfunktion so oft ausgeführt, wie es Datensätze in der Tabelle *tblBestellungen* gibt. Je nach Anzahl der Datensätze ein nicht gerade schneller Vorgang – und ein unnötiger noch dazu, da sich das Ergebnis der Skalarfunktion nicht ändert. Es enthält immer das aktuelle Tagesdatum. Besser ist es, das Ergebnis der Skalarfunktion in eine Variable zu schreiben und diese als Filterkriterium zu verwenden:

```
DECLARE @Tagesdatum datetime;
SELECT @Tagesdatum = dbo.sfAktuellesDatum();
SELECT * FROM dbo.tblBestellungen WHERE Bestelldatum = @Tagesdatum;
```

Beispiel 2 geht noch einen Schritt weiter. Hier wird die Funktion *sfWarengruppe* nicht als Filterkriterium verwendet, sondern um den Inhalt der Vergleichsspalte an das Filterkriterium anzupassen:

```
SELECT ArtikelID, Artikelname, WarengruppeID FROM dbo.tblArtikel
WHERE dbo.sfWarengruppe(WarengruppeID) = 'Seminare';
```

Nun wird die Skalarfunktion für jeden Datensatz der Tabelle *tblArtikel* ausgeführt, um das Ergebnis mit der Konstanten *Seminare* vergleichen zu können. An dieser Stelle ist es sinnvoller,

auf den Einsatz der Skalarfunktion komplett zu verzichten und stattdessen eine Variable zu verwenden:

```
DECLARE @WarengruppeID int;  
SELECT @WarengruppeID = WarengruppeID  
FROM dbo.tblWarengruppen WHERE Warengruppe = 'Seminare';  
SELECT ArtikelID, Artikelname, WarengruppeID  
FROM dbo.tblArtikel WHERE WarengruppeID = @WarengruppeID;
```

Beide Beispiele zeigen, dass es sich lohnt, bei der Verwendung von Funktionen deren Folgen zu prüfen und gegebenenfalls den Code umzustellen. Ebenso kann ein Blick in den Quellcode der Funktion nicht schaden, bevor Sie diese einsetzen. Hier sollten Sie prüfen, ob der in der Funktion betriebene Aufwand zur Datenermittlung für den geplanten Einsatzzweck nicht kontraproduktiv ist. Beispielsweise werden Skalarfunktionen, die aufwendige Schleifenverarbeitungen zur Ermittlung des Skalarwerts enthalten, nicht gerade zu einer besseren Performance beitragen, wenn Sie diese als Spalte oder in der *WHERE*-Bedingung einer SQL-Anweisung verwenden, die viele Datensätze verarbeitet.

Mit den Funktionen haben Sie nun das zweite SQL Server-Objekt kennen gelernt, mit dem Sie die Geschäftslogik im SQL Server-Backend abbilden können. Bei der Migration der Logik von Access nach SQL Server lassen sich viele Access- beziehungsweise VBA-Funktionen als Skalarfunktionen umstellen, Access-Parameterabfragen als Inline-Tabellenwertfunktionen und aufwendigere Datenermittlungen, die in Access durch das sequenzielle Ausführen mehrerer Abfragen erfolgen, als Tabellenwertfunktionen mit mehreren Anweisungen.

12 Trigger

Trigger lassen sich mit gespeicherten Prozeduren vergleichen. Sie werden auf ähnliche Weise angelegt und geändert und können eine oder mehrere SQL-Anweisungen kombiniert mit T-SQL-Befehlen und Strukturen wie Bedingungen oder Variablen enthalten. Ebenso wie bei den gespeicherten Prozeduren erstellt der SQL Server bei der ersten Ausführung eines Triggers einen Ausführungsplan und speichert diesen im Prozedurcache, sodass bei nachfolgenden Aufrufen der gespeicherte Ausführungsplan wiederverwendet wird und die Ausführung des Triggers einen Performancevorteil erhält.

Damit enden die Gemeinsamkeiten aber auch schon. Denn im Gegensatz zu gespeicherten Prozeduren sind Trigger keine eigenständigen Objekte und lassen sich dementsprechend auch nicht manuell aufrufen. Ein Trigger ist vielmehr Bestandteil der Definition einer Tabelle und legt fest, welche Aktionen bei den verschiedenen Datenoperationen ausgelöst werden.

12.1 Funktionsweise von Triggern

Wer bislang nur mit Access programmiert hat, erhält ein besseres Verständnis von der Funktionsweise von Triggern, wenn er diese mit den Ereignisprozeduren von Formularen oder Steuerelementen vergleicht. In Access können Sie Ereignisprozeduren hinterlegen, die durch verschiedene Ereignisse eines Formulars oder Steuerelements ausgelöst werden – etwa beim Laden oder Schließen des Formulars, beim Anzeigen oder Löschen eines neuen Datensatzes oder beim Betätigen einer Schaltfläche. Ähnlich ist es mit den Tabellen einer SQL Server-Datenbank. Auch hier lassen sich Prozeduren definieren, die durch verschiedene Ereignisse ausgelöst werden.

Welche Ereignisse können dies sein? Die folgende Übersicht fasst die möglichen Trigger zusammen:

- » *Aktualisieren eines Datensatzes:* Wird ein Datensatz einer Tabelle geändert, löst dies den sogenannten Update-Trigger aus.
- » *Löschen eines Datensatzes:* Wird ein Datensatz einer Tabelle gelöscht, löst dies den Delete-Trigger aus.
- » *Anlegen eines Datensatzes:* Auch beim Anlegen eines Datensatzes wird ein Trigger ausgelöst, in diesem Fall der Insert-Trigger.

Dabei ist wichtig, dass es keine Rolle spielt, auf welchem Weg ein Datensatz der Tabelle hinzugefügt, geändert oder gelöscht wird – ob im Frontend, mittels einer gespeicherten Prozedur oder durch die direkte Eingabe einer entsprechenden Anweisung im SQL Server Management Studio. Existiert an der Tabelle für diese Aktionen ein Trigger, wird dieser auch ausgelöst.

12.2 Pro und Contra

Trigger sind wegen ihrer Unbestechlichkeit nahezu perfekt zur Gewährleistung der Datenkonsistenz und zur Einhaltung von Geschäftsregeln. Nachfolgend finden Sie einige praktische Einsatzzwecke:

- » *Durchsetzen referenzieller Integrität:* Referenzielle Integrität inklusive Lösch- und Aktualisierungsweitergabe sind fester Bestandteil des Funktionsumfangs vom SQL Server. Dennoch kann es sein, dass Sie zusätzliche Anforderungen an die Prüfung referenzieller Integrität oder an die Lösch- und Aktualisierungsweitergabe haben. Diese lassen sich mit Triggern realisieren.
- » *Definition von Standardwerten:* SQL Server bietet die Möglichkeit, Standardwerte für Felder auf Basis von Konstanten sowie einfacher T-SQL-Funktionen und eigener Skalarfunktionen zu definieren. Alternativ wäre dies auch mit Triggern möglich.
- » *Definition von Einschränkungen:* Um eine Datenmanipulation zu verhindern, die gegen definierte Regeln verstößt, bieten Tabellen die *Check Constraints* (zu deutsch Einschränkungen). Ähnlich der Standardwerte können Sie zur Prüfung der Eingaben Konstanten sowie einfache T-SQL-Funktionen und eigene Skalarfunktionen nutzen. Die Prüfung der Eingaben kann aber auch von Triggern übernommen werden.
- » *Protokollieren von Datenänderungen:* Jede Änderung an den Daten einer Tabelle lässt sich über Trigger protokollieren. Auf diese Weise wäre nachvollziehbar, wer wann welchen Datensatz hinzugefügt, geändert oder gelöscht hat.
- » *Redundante Datenhaltung:* Redundante Daten werden gerne zur Performance-Steigerung verwendet. Um zum Beispiel die Rechnungssumme nicht für jede Auswertung erneut aus den Summen der Rechnungspositionen ermitteln zu müssen, könnte diese ebenso gut im Rechnungskopf gespeichert werden. Es muss jedoch gewährleistet sein, dass bei jeder Änderung der Positionen die Summe im Rechnungskopf angepasst wird. Wenn schon Redundanz, dann sollte diese auch korrekt und konsistent sein. Diese Konsistenz könnte ein Trigger sicherstellen. Egal ob eine neue Position hinzugefügt, eine bestehende geändert oder gelöscht wird, der Trigger übernimmt die Übertragung der neuen Summe in den Rechnungskopf.

Dies sind nur ein paar Beispiele für die Verwendung von Triggern. Beispiele, die auch gleichzeitig gute Argumente für den Einsatz von Triggern liefern. Nun gibt es aber auch durchaus einige Argumente gegen die Verwendung von Triggern.

Nicht umsonst wird oft über das Für und Wider von Triggern diskutiert. Wir möchten Ihnen die Entscheidung überlassen und führen nun auch die Gegenargumente auf:

- » *Keine Transparenz:* Die Trigger einer Datenbank und deren Auswirkungen sind nicht einfach zu überblicken. Trigger sind fest mit einer Tabelle verbunden und deshalb auch nur dort zu

finden. Um die Trigger einer Datenbank zu sehen, müssen Sie sich also mühsam durch alle Tabellen klicken oder aber Sie fragen die entsprechenden Systemtabellen der Datenbank ab und erstellen sich so eine Aufstellung aller Trigger. Aber auch mit einer Übersicht aller Trigger bleiben deren Auswirkungen weiterhin im Verborgenen. Schließlich könnte das Hinzufügen eines Datensatzes in Tabelle A durch einen Trigger eine Datenänderung an Tabelle B auslösen, die wiederum einen Trigger auslöst, der in Tabelle C und in Tabelle D einen Wert ändert. Es wäre sogar möglich, dass eine dieser Änderungen einen anderen Datensatz in Tabelle A löscht, worauf dort erneut ein Trigger aktiv wird. Solche rekursiven Aufrufe sind bis zu einer Tiefe von 32 Ebenen möglich, zum Glück nicht ohne vorherige Aktivierung.

- » *Schlechte Performance:* Abhängig von der Programmierung eines Triggers führt dieser seine Verarbeitung zu jedem Datensatz aus, der gerade hinzugefügt, geändert oder gelöscht wird. Je aufwendiger die Verarbeitung des Triggers ist, umso schlechter wird die Performance der Datenverarbeitung. Dies macht sich gerade dann bemerkbar, wenn Sie mit einer Anweisung gleich mehrere Datensätze verarbeiten.
- » *Komplexe Programmierung:* Zwar verwenden Sie zur Programmierung von Triggern ebenfalls T-SQL, jedoch ist die Vorgehensweise und Logik hierbei etwas anders. Mehr dazu später im Abschnitt »Trigger erstellen«, Seite 292.
- » *Für das Frontend unsichtbar:* Eine Datenänderung über eine SQL-Anweisung im VBA oder direkt in einer eingebundenen Tabelle weist Sie in keiner Weise auf einen möglichen Trigger hin. Dies bedeutet, dass Sie im Quellcode des Frontends von den Aktionen eines Triggers nichts sehen. Bilden Sie hingegen die Logik des Triggers mit gespeicherten Prozeduren ab, sehen Sie im Quellcode den Aufruf der jeweiligen gespeicherten Prozeduren.
- » *Gespeicherte Prozeduren:* Jegliche Aktion, die ein Trigger vor oder nach einer Datenänderung ausführen soll, können Sie auch in einer gespeicherten Prozedur realisieren. Eingabeprüfungen zum Beispiel führen Sie in einer gespeicherten Prozedur in entsprechenden *IF*-Anweisungen aus, worauf in Abhängigkeit vom Ergebnis die Aktion ausgeführt wird oder eben nicht. Oder soll beispielsweise beim Ändern einer Rechnungsposition die Rechnungssumme im Kopf angepasst werden, sind der *INSERT*- und *UPDATE*-Befehl lediglich in einer Transaktion auszuführen. Gespeicherte Prozeduren bieten Triggern gegenüber einen entscheidenden Vorteil: Anstelle der verborgenen Aktionen eines Triggers, die zudem noch eine wilde Verkettung weiterer Trigger zur Folge haben kann, sehen Sie in einer gespeicherten Prozedur alle notwendigen SQL-Anweisungen – lesbar und einfach nachvollziehbar.

Natürlich kommt es bei der Entscheidung für den Einsatz von Triggern nicht nur auf die oben genannten Argumente an. Viel entscheidender sind die Umstände des Projekts. In einer bestehenden und bereits installierten Client-/Server-Applikation werden Sie sich nicht unbedingt für die Variante mit gespeicherten Prozeduren entscheiden. Dies würde entscheidende Änderungen am Frontend nach sich ziehen, müssten Sie dort doch die jeweiligen Aufrufe der gespeicherten Prozeduren programmieren. In einem solchen Fall lässt sich die Logik schneller mit Triggern abbilden, da Sie hierbei das Frontend nicht zu ändern brauchen. Nur als Tipp: Dokumentieren

Sie Ihre Trigger und deren Auswirkungen auf andere Tabellen und deren Trigger von Anfang an ausführlich. Das erspart Ihnen später Zeit bei einer Fehlersuche. Sind Sie jedoch noch am Anfang des Projekts, sollten Sie den gespeicherten Prozeduren den Vorzug geben und auf Trigger verzichten – und genau dies ist der Fall, wenn Sie nach der Migration einer Access-Datenbank zum SQL Server auch die Logik von Access nach SQL Server verlagern. Technisch spricht nichts dagegen, Trigger und gespeicherte Prozeduren gemeinsam zu verwenden. Sie sollten jedoch gerade bei dieser Konstellation den Tipp zur ausführlichen Dokumentation berücksichtigen.

12.3 Zwei Arten von Triggern

SQL Server bietet Ihnen zwei verschiedene Trigger an. Beide reagieren auf die Aktionen *INSERT*, *UPDATE* und *DELETE*, jedoch zu unterschiedlichen Zeitpunkten:

- » *INSTEAD OF-Trigger*: wird vor der Datenmanipulation ausgelöst und ersetzt die eigentliche Aktion.
- » *AFTER-Trigger*: wird nach der Datenmanipulation ausgelöst.

Um zu verstehen, wann die beiden Trigger ausgeführt werden, schauen wir uns einmal den gesamten Prozess einer Datenmanipulation am Beispiel einer *INSERT*-Anweisung an:

- » Erstellen der virtuellen Tabellen *inserted* und *deleted*: Mehr dazu im Anschluss.
- » Auslösen des *INSTEAD OF*-Triggers: Die Logik des Triggers wird verarbeitet. Beinhaltet diese beispielsweise eine Prüfung der Eingabe, könnte der Trigger bei einem negativen Ergebnis die *INSERT*-Anweisung bereits an dieser Stelle abbrechen.
- » Prüfen der Einschränkungen (Check Constraints) und der referenziellen Integrität: Schlägt eine der Prüfungen fehl, wird die Verarbeitung der *INSERT*-Anweisung abgebrochen.
- » Erzeugen von Standardwerten: Zu jeder Spalte, die nicht durch die *INSERT*-Anweisung einen Wert erhält, wird gemäß der Spaltendefinition der Standardwert erstellt.
- » Manipulation der Daten: Der Datensatz wird der Datenseite und somit der Tabelle hinzugefügt.
- » Auslösen des *AFTER*-Triggers: Die im Trigger enthaltene Logik wird verarbeitet. Gibt es hierbei einen Fehler, wird der komplette *INSERT*-Vorgang abgebrochen und die Tabelle auf den Zustand vor der Aktion zurückgesetzt – es wird ein *ROLLBACK* ausgeführt.
- » Bestätigen der Datenmanipulation: Die *INSERT*-Anweisung wird mit einem *COMMIT* bestätigt.

Interessant bei diesem Prozess ist der *AFTER*-Trigger. Obwohl dieser erst nach der eigentlichen Aktion ausgeführt wird, ist er dennoch ein fester Bestandteil dieser Aktion. Ein Fehler im *AFTER*-Trigger würde in unserem Beispiel die *INSERT*-Anweisung verhindern, auch wenn es beim eigentlichen Hinzufügen des Datensatzes kein Problem gibt. Ein Umstand, den es bei der

Programmierung von *AFTER*-Triggern zu beachten gilt. Dort sollten keine komplexen und fehleranfälligen Verarbeitungen erfolgen.

Die virtuellen Tabellen *inserted* und *deleted*

Kommen wir auf die virtuellen Tabellen *inserted* und *deleted* zurück. Beide Tabellen werden bei jeder Datenmanipulation erstellt und enthalten die davon betroffenen Daten.

Abhängig von der Aktion sind die Daten wie folgt auf die beiden Tabellen verteilt:

- » *INSERT*: Die Tabelle *inserted* beinhaltet die neuen Datensätze, die Tabelle *deleted* ist leer.
- » *UPDATE*: Die Tabelle *inserted* beinhaltet die neuen Werte der Datensätze, die Tabelle *deleted* die Werte vor der Änderung.
- » *DELETE*: Die Tabelle *inserted* ist leer und die Tabelle *deleted* beinhaltet die gelöschten Datensätze.

Im Trigger können Sie nun mit den Daten dieser beiden Tabellen arbeiten. Die Daten lassen sich per *SELECT* auswerten und dabei mit anderen Tabellen verknüpfen sowie aggregieren, sortieren und filtern. Nur das Ändern der Daten ist nicht möglich.

Ob Sie die Daten der beiden virtuellen Tabellen im Trigger überhaupt benötigen, ist abhängig von der dort abgebildeten Logik. Ein Trigger reagiert auf die Aktion, für die er erstellt wurde und nicht unbedingt auf die dabei geänderten Daten. Sie könnten zum Beispiel mit einem *AFTER*-Trigger lediglich das Löschen von Datensätzen protokollieren.

Dabei wird nur der Zeitpunkt und der Benutzer in einem Protokoll gespeichert, nicht aber die gelöschten Daten. Möchten Sie hingegen die entfernten Datensätze ebenfalls protokollieren, müssen Sie im Trigger die Daten der virtuellen Tabelle *deleted* auswerten und diese im Protokoll speichern. Wie Sie die Daten der virtuellen Tabellen im Trigger verwenden, sehen Sie in den kommenden Beispielen.

Mehrere Trigger an einer Tabelle

Eine Tabelle kann maximal einen *INSTEAD OF*-Trigger enthalten, jedoch mehrere *AFTER*-Trigger. Es wäre also beispielsweise möglich, einen *AFTER*-Trigger für die Aktionen *INSERT*, *UPDATE* und *DELETE* zur Protokollierung von Datenänderungen zu definieren sowie einen weiteren *AFTER*-Trigger für einen besonderen und eher seltenen Fall einer *UPDATE*-Anweisung. In einem solchen Fall würden beim Ändern eines Datensatzes beide Trigger ausgelöst.

Grundsätzlich gibt es für die Ausführung mehrerer *AFTER*-Trigger keine Reihenfolge, es sei denn Sie legen diese mit der Systemprozedur *sp_SetTriggerOrder* fest. Allerdings definieren Sie hiermit lediglich die beiden *AFTER*-Trigger, die als Erstes und als Letztes ausgeführt werden.

Die Ausführungsreihenfolge für alle anderen *AFTER*-Trigger lässt sich nicht beeinflussen. Haben Sie also an einer Tabelle vier *AFTER*-Trigger, ist nur die Ausführungsreihenfolge für den als ersten

Kapitel 12 Trigger

und den als letzten definierten Trigger festlegt. Die Ausführung der beiden anderen Trigger erfolgt willkürlich.

Grundsätzlich sollten Sie sich gut überlegen, ob Sie die Logik eines *AFTER*-Triggers in einem Trigger abbilden oder ob Sie diese auf mehrere Trigger aufteilen. Wie so oft, gilt auch hier der Tipp: *KISS* – Keep It Simple And Stupid.

12.4 Trigger erstellen

Trigger werden wie die anderen SQL Server-Objekte mit der *CREATE*-Anweisung angelegt – hier mit *CREATE TRIGGER*. Das SQL Server Management Studio bietet zudem wie bei gespeicherten Prozeduren und Funktionen eine Vorlage mit der Syntax zum Anlegen eines Triggers.

Um diese Vorlage zu erhalten, navigieren Sie im Objekt-Explorer zu der Tabelle, an der der Trigger angelegt werden soll. Dort wählen Sie im Kontextmenü des Elements *Trigger* den Eintrag *Neuer Trigger*. Die Vorlage wird dann in einem neuen Abfragefenster geöffnet (siehe Abbildung 12.1).

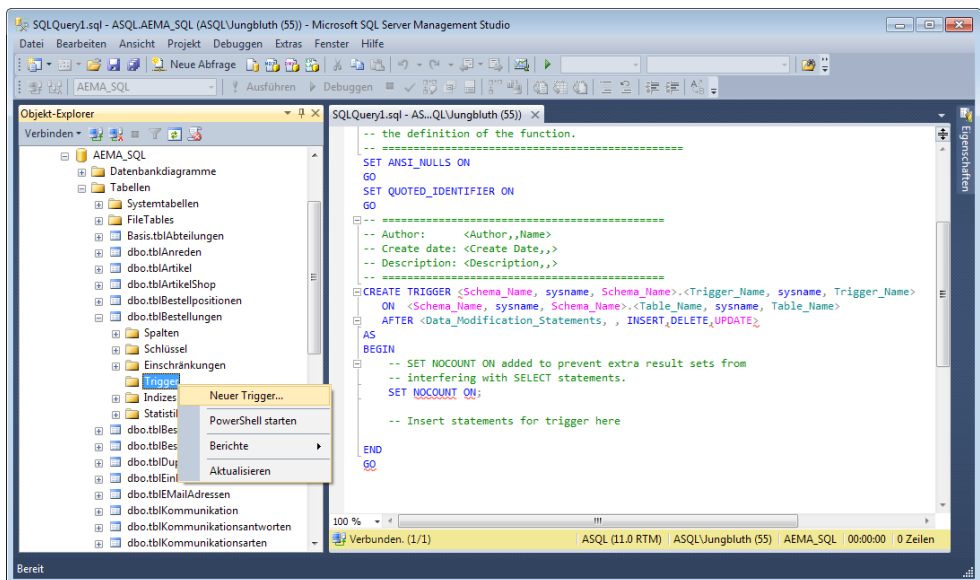


Abbildung 12.1: Einen neuen Trigger anlegen

Leider ist auch diese Vorlage nicht sonderlich hilfreich. Deshalb zeigen wir Ihnen in den folgenden Beispielen, wie Sie Trigger ohne die Vorlage anlegen.

Ein Trigger ist in seiner Struktur vielschichtiger als eine gespeicherte Prozedur, eine Sicht oder eine Funktion. Infolgedessen ist auch seine Syntax etwas komplexer. Bevor wir also zu den ei-

gentlichen Beispielen kommen, möchten wir Ihnen zunächst die Grundstruktur eines Triggers vorstellen.

Los geht es mit der *CREATE TRIGGER*-Anweisung. Es folgt der Name des Schemas, dem Sie den Trigger zuordnen möchten. Anschließend geben Sie dem Trigger einen Namen. Bei der Namensvergabe empfiehlt sich wieder die Verwendung eines Präfix, etwa *tr_*.

Ebenso empfehlenswert ist es, bereits im Triggernamen darauf hinzuweisen, zu welcher Tabelle der Trigger gehört, um welchen Typ es sich handelt und bei welchen Aktionen er ausgelöst wird. Hier ein Beispiel für den Namen eines Triggers, der als *AFTER*-Trigger auf *UPDATE*-Anweisungen an der Tabelle *tblBestellungen* reagiert:

```
dbo.tr_tblBestellungenAfterUpdate
```

Nach dem Schema und der Bezeichnung folgt mit dem Schlüsselwort *ON* die Zuweisung zur Tabelle. Anschließend wird die Art des Triggers mit *AFTER* oder *INSTEAD OF* angegeben und die Aktionen aufgelistet, bei denen der Trigger ausgelöst werden soll.

```
CREATE TRIGGER dbo.tr_tblBestellungenAfterUpdate  
ON dbo.tblBestellungen AFTER Update
```

Mit dem Schlüsselwort *AS* leiten Sie dann den eigentlichen Programmcode ein, der auch hier wieder in einem *BEGIN...END*-Block enthalten ist.

```
CREATE TRIGGER dbo.tr_tblBestellungenAfterUpdate  
ON dbo.tblBestellungen AFTER Update  
AS  
BEGIN  
    SET NOCOUNT ON;  
    -- Ihr Programmcode  
END
```

Im obigen Skript sehen Sie die Anweisung *SET NOCOUNT ON*. Diese geben Sie in jedem Trigger aus den gleichen Gründen wie bei den gespeicherten Prozeduren an.

Innerhalb des *BEGIN...END*-Blocks können Sie mit T-SQL, Variablen, *IF*-Anweisungen und Schleifen arbeiten. Dabei sollten Sie jedoch nie die Performance außer Acht lassen. Bedenken Sie, dass die hier abgebildete Logik bei jeder Aktion ausgelöst wird, für die der Trigger definiert ist.

Die Trigger-Programmierung bietet eine besondere Spezialität: die Funktion *UPDATE()*. Hiermit prüfen Sie, ob sich bei der aktuellen Datenverarbeitung der Wert einer bestimmten Spalte geändert hat. Zur Prüfung geben Sie dazu den Spaltennamen als Parameter an:

```
IF UPDATE(<spaltenname>)
```

Wurde der Inhalt der Spalte geändert, liefert die Funktion den Wert *True*, ansonsten *False*. Auf diese Weise können Sie die weitere Verarbeitung des Triggers abhängig von der Änderung einer bestimmten Spalte steuern. Im Abschnitt »*INSTEAD OF*-Trigger erstellen«, Seite 294, erstellen sehen Sie den Einsatz dieser Funktion.

Neben den gespeicherten Prozeduren sind Trigger die einzigen SQL Server-Objekte, mit denen Sie Daten in Tabellen hinzufügen, ändern und löschen können. Jedoch sollten Sie in einem Trigger die SQL-Befehle *INSERT*, *UPDATE* und *DELETE* vorsichtig einsetzen und vorab prüfen, ob diese Aktionen nicht weitere Trigger an den betroffenen Tabellen auslösen.

Dies gilt auch für gespeicherte Prozeduren, die Sie in Triggern verwenden. Auch hier sind vorher die Auswirkungen zu prüfen.

12.4.1 INSTEAD OF-Trigger erstellen

Ein *INSTEAD OF*-Trigger wird vor der eigentlichen Datenmanipulation ausgeführt. Er ist also prädestiniert für Prüfungen. Bei einem negativen Ergebnis der Prüfung kann im Trigger die eigentliche Aktion abgebrochen oder auch korrigiert werden.

Liefert die Prüfung ein positives Ergebnis, bedeutet dies aber nicht, dass der Trigger die tatsächliche Aktion ausführt. *INSTEAD OF* steht nun mal für *Anstelle dessen*, weshalb auch die im Trigger enthaltene Logik anstelle der eigentlichen SQL-Anweisung ausgeführt wird. Soll also im positiven Fall der Prüfung die tatsächliche Aktion verarbeitet werden, muss der Quellcode des Triggers die SQL-Anweisung der tatsächlichen Aktion auch enthalten.

Ein *INSTEAD OF*-Trigger kann die eigentliche SQL-Anweisung ebenso komplett mit einer anderen Verarbeitung ersetzen. So wäre es beispielsweise möglich, dass ein Trigger anstelle einer ursprünglichen *DELETE*-Anweisung eine *UPDATE*-Anweisung ausführt. Auf diese Weise könnten Sie das Löschen von Datensätzen vermeiden und stattdessen diese als inaktiv kennzeichnen.

Im Gegensatz zum *AFTER*-Trigger erweitert ein *INSTEAD OF*-Trigger also nicht den Vorgang der eigentlichen Datenverarbeitung, sondern ersetzt diesen durch seine eigene Logik. Für die Verarbeitung der Daten ist dann nicht mehr die ursprüngliche Aktion maßgebend, sondern die SQL-Anweisungen des *INSTEAD OF*-Triggers.

In unserem ersten Beispiel wollen wir dies nutzen und mit einem *INSTEAD OF*-Trigger das Löschen von Datensätzen in der Tabelle *tblRechnungen* vermeiden. Da hier eine bestimmte Datenverarbeitung kategorisch ausgeschlossen werden soll, ist dies ein klassischer Fall für einen *INSTEAD OF*-Trigger.

Für den neuen Trigger öffnen Sie zunächst ein neues Abfragefenster über den Eintrag *Neue Abfrage* aus dem Kontextmenü der Beispieldatenbank *AEMA_SQL*. Dort geben Sie nun die *CREATE TRIGGER*-Anweisung ein, gefolgt vom Schema und dem Namen für den Trigger.

```
CREATE TRIGGER dbo.tr_tblRechnungenInsteadOfDelete
```

Das es sich um einen Trigger an der Tabelle *tblRechnungen* handelt, der zudem auf die Aktion *DELETE* reagieren soll, definieren Sie mit den folgenden Parametern:

```
ON dbo.tblRechnungen INSTEAD OF Delete
```

Es folgt das Schlüsselwort *AS* und darauf die eigentliche Logik im *BEGIN...END*-Block. Die Logik beginnt wie bereits erwähnt immer mit *SET NOCOUNT ON*.

Da dieser Trigger letztendlich nur das Löschen von Datensätzen vermeiden soll, besteht der Quellcode lediglich aus einer Fehlermeldung, die mit dem T-SQL-Befehl *RAISERROR* ausgelöst wird.

Der komplette Trigger sieht folgendermaßen aus:

```
CREATE TRIGGER dbo.tr_tblRechnungenInsteadOfDelete
ON dbo.tblRechnungen INSTEAD OF Delete
AS
BEGIN
    SET NOCOUNT ON;
    RAISERROR('Das Löschen von Rechnungen ist nicht erlaubt!', 16, 1)
END;
```

Nachdem Sie den Quellcode im Abfragefenster eingegeben haben, führen Sie diesen mit der Taste *F5* aus. Der Trigger wird angelegt, wodurch die im Trigger enthaltenen SQL-Anweisungen nun ein fester Bestandteil der Tabelle sind.

Sie finden den Trigger im Element *Trigger* der Tabelle *tblRechnungen*. Möglicherweise müssen Sie die Anzeige im Objekt-Explorer zunächst aktualisieren (siehe Abbildung 12.2).

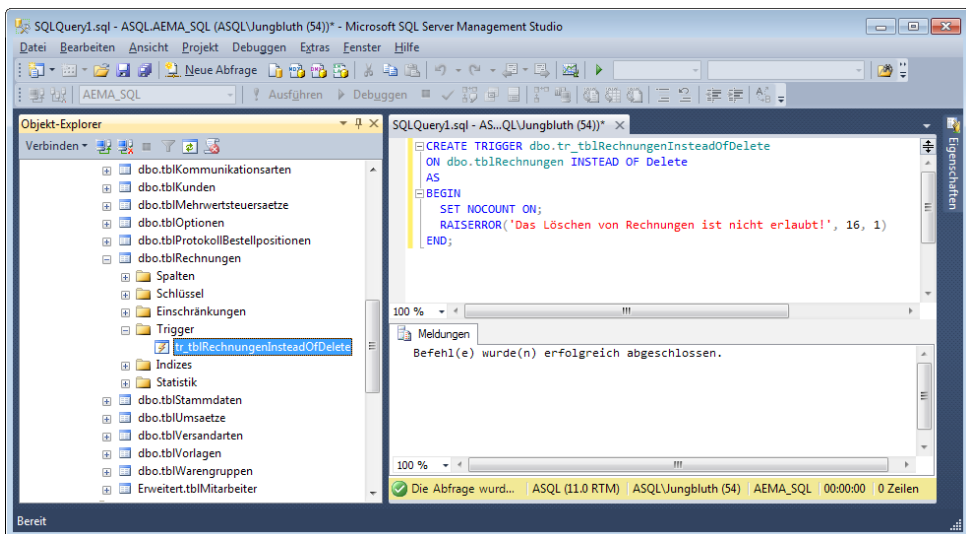


Abbildung 12.2: Der neue Trigger

Den Trigger können Sie in einem neuen Abfragefenster mit der folgenden *DELETE*-Anweisung testen:

```
DELETE FROM dbo.tblRechnungen WHERE RechnungID = 2;
```

Kapitel 12 Trigger

Als Ergebnis erhalten Sie die Meldung des Triggers, dass das Löschen von Rechnungen nicht erlaubt ist (siehe Abbildung 12.3). Diese Meldung wird auch ausgegeben, wenn Sie mehrere Rechnungen mit einer SQL-Anweisung löschen.

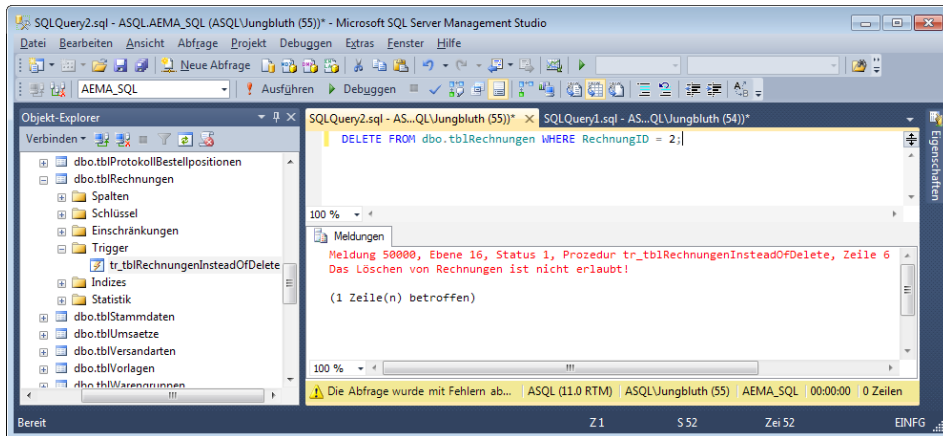


Abbildung 12.3: Der neue Trigger in Aktion

Dieses Beispiel zeigt nicht nur einen klassischen Einsatz eines *INSTEAD OF*-Triggers, Sie sehen hier auch, dass die Verarbeitung eines Triggers unabhängig von den verarbeiteten Datensätzen sein kann.

Im nächsten Beispiel wird der Trigger die durch die Aktion geänderten Daten verarbeiten. Dieser Trigger vermeidet in der Tabelle *tblBestellpositionen* die Eingabe von Datensätzen mit einem zu hohen Rabatt. Durch eine fiktive Geschäftsregel sind Rabatte von mehr als 10 € nicht erlaubt und sollen auf jeden Fall vermieden werden. Gibt es beim Hinzufügen oder Ändern von Datensätzen auch nur einen Datensatz mit einem Rabatt von mehr als 10 €, wird die Aktion direkt abgebrochen. Ein Fall für einen *INSTEAD OF*-Trigger. Dieses Beispiel zeigt Ihnen drei neue Aspekte bei der Programmierung eines Triggers:

- » Die Verwendung der Funktion *UPDATE()*
- » Der Bezug auf die verarbeiteten Datensätze
- » Die Besonderheit eines *INSTEAD OF*-Triggers

Beginnen wir von vorne und geben dem Trigger zunächst einen Namen, sowie die Zuordnung zur Tabelle und den Aktionen:

```
CREATE TRIGGER dbo.tr_tblBestellpositionenInsteadOfInsertUpdate  
ON dbo.tblBestellpositionen INSTEAD OF Insert, Update
```

Um die Rabattvergabe beim Neuanlegen wie auch beim Ändern eines Datensatzes zu prüfen, wird der Trigger für die Aktionen *INSERT* und *UPDATE* definiert. Bevor es danach an die tat-

sächliche Logik geht, ist wieder das Schlüsselwort *AS* und der *BEGIN...END*-Block anzugeben. Die eigentliche Verarbeitung des Triggers beginnt erneut mit der Anweisung *SET NOCOUNT ON*. Danach wird in einer *IF*-Anweisung mit der Funktion *UPDATE()* geprüft, ob bei der Datenmanipulation der Wert der Spalte *Rabatt* geändert wurde. Liefert dies den Wert *True*, folgt die weitere Verarbeitung der Datensätze mit geänderten Rabattwerten. Ist das Ergebnis der Prüfung jedoch *False*, bedeutet dies, dass die Spalte *Rabatt* nicht von der Datenmanipulation betroffen war.

Betrachten wir zunächst den Fall, dass der Inhalt der Spalte verändert wurde. Hier folgt dann in einer weiteren *IF*-Anweisung die Prüfung, ob einer der vergebenen Rabatte über 10 € liegt. Dazu wird in der Tabelle *inserted* der größte Wert der Spalte *Rabatt* ermittelt und verglichen. Die Tabelle *inserted* beinhaltet im Fall einer *INSERT*-Aktion die neuen Werte und im Fall einer *UPDATE*-Aktion die Werte der Änderung.

Liegt der höchste vergabene Rabatt über dem Grenzwert, wird mit der *RAISERROR*-Anweisung eine Meldung ausgegeben und die gesamte Datenverarbeitung per *RETURN* beendet.

```
IF UPDATE(Rabatt)
BEGIN
    IF (SELECT Max(Rabatt) FROM inserted) > 10
        BEGIN
            RAISERROR ('Es wurde ein Rabatt von mehr als 10 Euro vergeben!', 16, 1)
            RETURN
        END
END
END;
```

Soweit die Verarbeitung der Datensätze mit einem zu hohen Rabatt. Wurde jedoch weder die Spalte *Rabatt* verändert, noch ein Rabatt über der angegebenen Grenze vergeben, können die Datensätze der Tabelle hinzugefügt oder dort geändert werden.

Ein *INSTEAD OF*-Trigger führt aber per Definition nicht die ursprüngliche Aktion aus, sondern ersetzt diese durch seine eigene Verarbeitung. Also muss der Quellcode des Triggers die hierfür notwendigen *INSERT*- und *UPDATE*-Anweisungen enthalten. Wobei die *UPDATE*-Anweisung natürlich nur bei einer *UPDATE*-Aktion und die *INSERT*-Anweisung im Fall einer *INSERT*-Aktion ausgeführt werden darf.

Woher aber kommt die Information, ob der Trigger nun über eine *INSERT*-Aktion oder durch eine *UPDATE*-Aktion ausgelöst wurde? Die Antwort liefert uns in diesem Fall die virtuelle Tabelle *deleted*. Diese Tabelle enthält bei einer *UPDATE*-Aktion die Werte vor der Änderung und im Fall einer *INSERT*-Aktion ist sie leer. Wir müssen also lediglich prüfen, ob die Tabelle *deleted* Daten enthält. Ist dies der Fall, handelt es sich um eine *UPDATE*-Aktion, ansonsten um eine *INSERT*-Aktion:

```
IF (SELECT Count(*) FROM deleted) = 0
BEGIN
    INSERT dbo.tblBestellpositionen
        (BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
        SELECT BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt FROM
        inserted;
```

Kapitel 12 Trigger

```
END
ELSE
BEGIN
    UPDATE dbo.tblBestellpositionen
    SET BestellungID = inserted.BestellungID,
        ArtikelID = inserted.ArtikelID,
        Einzelpreis = inserted.Einzelpreis,
        Mehrwertsteuersatz = inserted.Mehrwertsteuersatz,
        Menge = inserted.Menge,
        Rabatt = inserted.Rabatt
    FROM inserted INNER JOIN dbo.tblBestellpositionen
        ON inserted.BestellpositionID = dbo.tblBestellpositionen.
        BestellpositionID;
END
```

Das Hinzufügen der neuen Datensätze erfolgt mit den Daten der virtuellen Tabelle *inserted*. Auch für die *UPDATE*-Anweisung sind die Daten der Tabelle *inserted* maßgebend. Diese werden hierfür mit den Daten der Tabelle *tblBestellpositionen* verknüpft. Mit diesen Anweisungen ist der Quellcode des Triggers auch schon komplett. Der gesamte Quellcode sieht wie folgt aus:

```
CREATE TRIGGER dbo.tr_tblBestellpositionenInsteadOfInsertUpdate
ON dbo.tblBestellpositionen INSTEAD OF Insert, Update
AS
BEGIN
    SET NOCOUNT ON;
    IF UPDATE(Rabatt)
    BEGIN
        IF (SELECT Max(Rabatt) FROM inserted) > 10
        BEGIN
            RAISERROR ('Es wurde ein Rabatt von mehr als 10 Euro vergeben!', 16, 1)
            RETURN
        END
    END
    IF (SELECT Count(*) FROM deleted) = 0
    BEGIN
        INSERT dbo.tblBestellpositionen
        (BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
        SELECT BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt FROM
        inserted;
    END
    ELSE
    BEGIN
        UPDATE dbo.tblBestellpositionen
        SET BestellungID = inserted.BestellungID,
            ArtikelID = inserted.ArtikelID,
            Einzelpreis = inserted.Einzelpreis,
            Mehrwertsteuersatz = inserted.Mehrwertsteuersatz,
            Menge = inserted.Menge,
            Rabatt = inserted.Rabatt
        FROM inserted INNER JOIN dbo.tblBestellpositionen
            ON inserted.BestellpositionID = dbo.tblBestellpositionen.BestellpositionID;
    END
END
```

Um den Trigger zu testen, verwenden Sie die folgende Anweisung:

```
UPDATE dbo.tblBestellpositionen SET Rabatt = 11 WHERE BestellpositionID = 279;
```

Die Änderung der Bestellposition mit der ID 279 wird mit der Fehlermeldung des Triggers abgewiesen. Ein Blick in die Tabelle zeigt, dass der neue Rabatt von 11 € dort nicht gespeichert wurde.

Als Nächstes sollen drei Datensätze auf einmal an die Tabelle angefügt werden, einer davon mit einem zu hohen Rabatt. Hierfür verwenden wir die seit SQL Server 2008 mögliche Syntax des *Table Row Constructors*.

```
INSERT INTO dbo.tblBestellpositionen
(BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
VALUES
(147, 108, '350.0', '0,19', 1, '0'),
(147, 109, '350.0', '0,19', 1, '5.0'),
(147, 110, '350.0', '0,19', 1, '11.0');
```

Auch der *INSERT*-Vorgang wurde abgebrochen und keiner der drei Datensätze in der Tabelle gespeichert. Den Grund hierfür liefert der letzte Datensatz, der einen Rabatt von 11 € aufweist. Den Hinweis hierzu zeigt die Meldung aus Abbildung 12.4.

Vielleicht finden Sie die Vorgehensweise des Triggers zu rigoros und Sie möchten wenigstens die Datensätze mit korrektem Rabatt speichern. Wobei der Benutzer natürlich darüber informiert werden sollte, wenn es Datensätze mit zu hohem Rabatt gibt und diese deshalb nicht angelegt oder geändert wurden. Auch diese Anforderung lässt sich mit einem *INSTEAD OF*-Trigger realisieren.

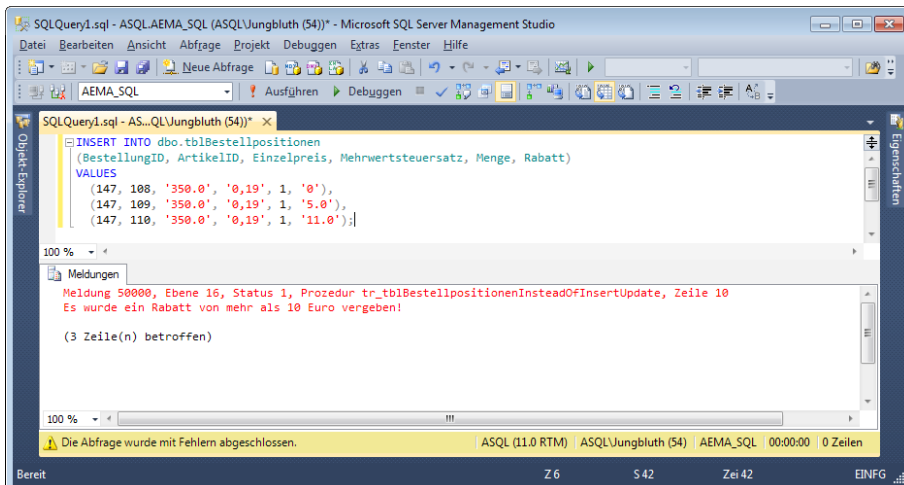


Abbildung 12.4: Die Eingabeverweigerung eines *INSTEAD OF*-Triggers

Die neue Logik unterscheidet sich nur an zwei Stellen von der ersten Variante. Der erste Unterschied besteht in der Verarbeitung der Datensätze mit zu hohem Rabatt. Liefert die Prüfung

Kapitel 12 Trigger

auf den höchsten vergebenen Rabatt bei dieser Version einen Wert größer 10 €, wird die Anzahl der Datensätze mit zu hohen Rabatten in der Tabelle *inserted* ermittelt und das Ergebnis in einer Variablen namens *@AnzahlZuHoherRabatt* gespeichert. Diese Variable ist die Basis für die darauf folgende Meldung, die mit der Anweisung *RAISERROR* ausgegeben wird.

```
IF (SELECT Max(Rabatt) FROM inserted) >= 10
BEGIN
    -- Speichern Anzahl Datensätze mit zu hohen Rabatt
    SELECT @AnzahlZuHoherRabatt = COUNT(*) FROM inserted WHERE Rabatt > 10;
    -- Ausgabe der Meldung
    SET @Meldung = N'Es wurden ' + CAST(@AnzahlZuHoherRabatt as nvarchar(5)) +
        N' Bestellposition(en) nicht gespeichert, da
diese einen zu hohen Rabatt aufweisen!';
    RAISERROR (@Meldung, 16, 1);
END
```

Den zweiten Unterschied gibt es bei der Verarbeitung der tatsächlichen SQL-Anweisung. Hier werden jetzt nur die Datensätze mit korrektem Rabatt per *INSERT* oder *UPDATE* verarbeitet. Da Sie an einer Tabelle nur einen *INSTEAD OF*-Trigger definieren können, müssen Sie die erste Variante mit dem neuen Quellcode überschreiben. Hierfür verwenden Sie wieder den *ALTER*-Befehl:

```
ALTER TRIGGER dbo.tr_tblBestellpositionenInsteadOfInsertUpdate
ON dbo.tblBestellpositionen INSTEAD OF Insert, Update
AS
BEGIN
    SET NOCOUNT ON;
    -- Variable für Anzahl Datensätze mit ungültigem Rabatt
    DECLARE @AnzahlZuHoherRabatt int = 0;
    -- Variable für die Meldung bei Datensätzen mit ungültigem Rabatt
    DECLARE @Meldung nvarchar(255);
    -- Rabatt geändert?
    IF UPDATE(Rabatt)
    BEGIN
        -- Ausgabe Datensätze mit zu hohen Rabatt
        IF (SELECT Max(Rabatt) FROM inserted) > 10
        BEGIN
            -- Speichern Anzahl Datensätze mit zu hohen Rabatt
            SELECT @AnzahlZuHoherRabatt = COUNT(*) FROM inserted WHERE Rabatt > 10;
            -- Ausgabe der Meldung
            SET @Meldung = N'Es wurden ' + CAST(@AnzahlZuHoherRabatt as nvarchar(5)) +
                N' Bestellposition(en) nicht gespeichert, da
diese einen zu hohen Rabatt aufweisen!';
            RAISERROR (@Meldung, 16, 1);
        END
    END
    -- INSERT oder UPDATE
    IF (SELECT Count(*) FROM deleted) = 0
    BEGIN
        -- Nur die mit guten Rabatt einfügen
        INSERT dbo.tblBestellpositionen
        (BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt)
        SELECT BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz, Menge, Rabatt
```



```

FROM inserted
  WHERE inserted.Rabatt <= 10;
END
ELSE
BEGIN
  -- Nur die mit gutem Rabatt ändern
  UPDATE dbo.tblBestellpositionen
  SET BestellungID = inserted.BestellungID, ArtikelID = inserted.ArtikelID,
      Einzelpreis = inserted.Einzelpreis, Mehrwertsteuersatz = inserted.
Mehrwertsteuersatz,
      Menge = inserted.Menge, Rabatt = inserted.Rabatt
  FROM inserted INNER JOIN dbo.tblBestellpositionen
      ON inserted.BestellpositionID = dbo.tblBestellpositionen.
BestellpositionID
  WHERE inserted.Rabatt <= 10;
END
END

```

Testen Sie diesen Trigger wieder mit der bereits eben verwendeten *INSERT*-Anweisung, werden Sie feststellen, dass nur ein Datensatz nicht im Ergebnis enthalten ist (siehe Abbildung 12.5 und Abbildung 12.6).

12.4.2 AFTER-Trigger erstellen

Der *AFTER*-Trigger wird nach der eigentlichen Datenmanipulation ausgeführt. Er beinhaltet in der Regel Ergänzungen zur Datenverarbeitung, beispielsweise das Speichern von Änderungsinformationen, bestehend aus dem Zeitpunkt der Änderung und dem Anmeldenamen des Benutzers, der die Änderung durchgeführt hat. Genau dieses Szenario wird unser Beispiel für den *AFTER*-Trigger sein. Eine neue Geschäftsregel schreibt vor, dass in der Tabelle *tblBestellpositionen* Informationen zur letzten Änderung zu speichern sind. Die Tabelle wird hierzu mit zwei neuen Spalten ergänzt:

- » *LetzteAenderungVon* vom Datentyp *nvarchar(255)* speichert den Anmeldenamen des Benutzers, der die Änderung durchgeführt hat.
- » *LetzteAenderungAm* vom Datentyp *datetime* speichert den Zeitpunkt der letzten Änderung.

Sie können die Tabelle *tblBestellpositionen* entweder manuell im Entwurfsmodus mit den beiden Spalten ergänzen oder aber Sie führen in einem neuen Abfragefenster diese Anweisung aus:

```

ALTER TABLE dbo.tblBestellpositionen
ADD LetzteAenderungVon nvarchar(255), LetzteAenderungAm datetime;

```

Die Werte zu den Änderungsinformationen liefert ein *AFTER*-Trigger. Dieser soll sowohl bei einer *INSERT*- wie auch bei einer *UPDATE*-Aktion die Änderungsinformationen in die beiden Spalten eintragen.

Öffnen Sie also ein neues Abfragefenster, um das Skript zum Erstellen eines *AFTER*-Triggers einzugeben. Das Skript beginnt wieder mit der Anweisung *CREATE TRIGGER*, der Angabe des

Kapitel 12 Trigger

Schemas und der Namensvergabe, gefolgt von der Zuweisung zur Tabelle. Als Typ geben Sie nun *AFTER* ein und ergänzen die Anweisung noch mit der Angabe der Aktionen, bei denen der *AFTER*-Trigger ausgelöst werden soll.

Das Schlüsselwort *AS* leitet wie gewohnt die Programmlogik ein, die auch hier in einem *BEGIN...END*-Block enthalten ist. Mal abgesehen von der Standardanweisung *SET NOCOUNT ON*, besteht die Logik dieses Triggers lediglich aus einer einzigen *UPDATE*-Anweisung.

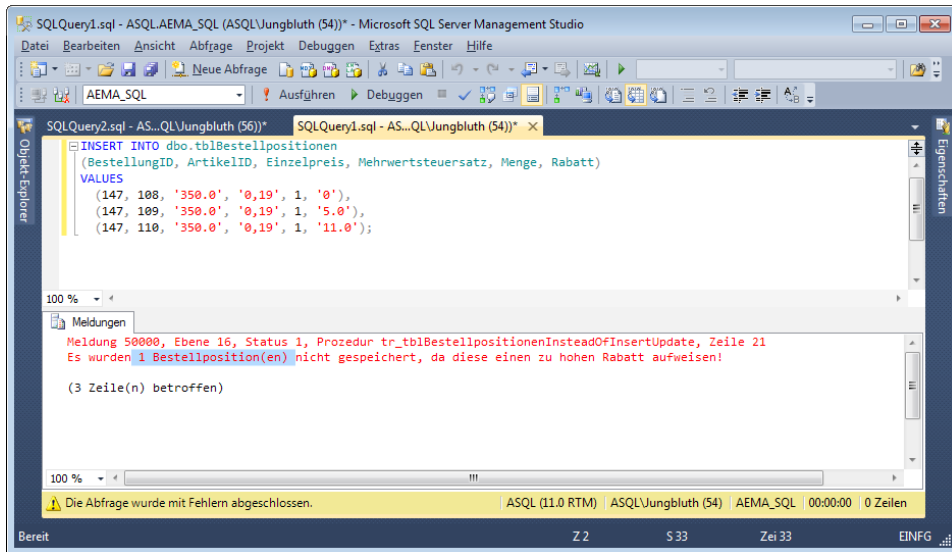
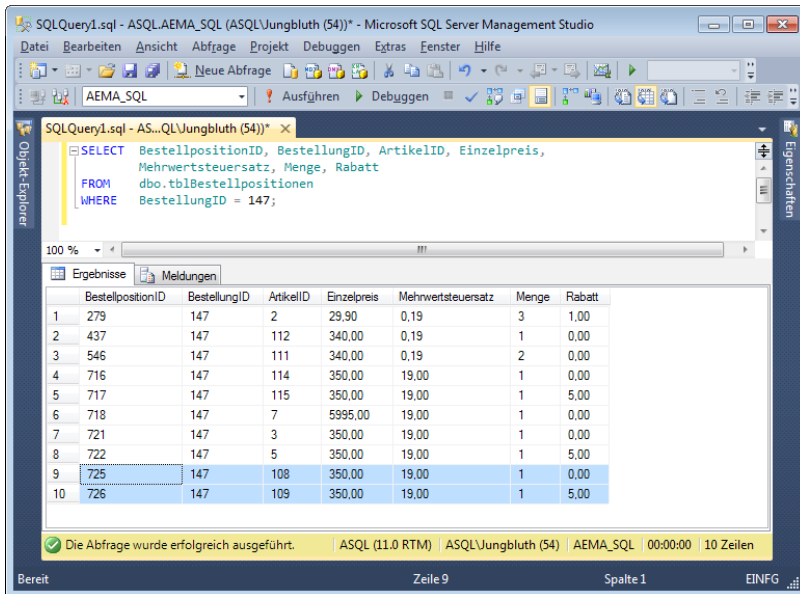


Abbildung 12.5: Die Schlechten ins Kröpfchen ...

Diese aktualisiert die verarbeiteten Datensätze der Tabelle *tblBestellpositionen* mit den Änderungsdaten, wobei die entsprechenden Datensätze über die virtuelle Tabelle *inserted* ermittelt werden.

```
CREATE TRIGGER dbo.tr_tblBestellpositionenAfterInsertUpdate
ON dbo.tblBestellpositionen AFTER Insert, Update
AS
BEGIN
    SET NOCOUNT ON;
    UPDATE dbo.tblBestellpositionen
    SET LetzteAenderungVon = SUSER_SNAME(), LetzteAenderungAm = GETDATE()
    FROM inserted INNER JOIN dbo.tblBestellpositionen
    ON inserted.BestellpositionID = dbo.tblBestellpositionen.
    BestellpositionID;
END
```

Den Zeitpunkt der letzten Änderung liefert Ihnen die bereits bekannte Systemfunktion *GETDATE()*. Auch die Information zum Benutzer ist das Ergebnis einer Systemfunktion: *SUSER_SNAME()* ermittelt den Anmeldenamen des angemeldeten Benutzers.



SQLQuery1.sql - AS...QLVungbluth (54)) - Microsoft SQL Server Management Studio

Objekt-Explorer

SQLQuery1.sql - AS...QLVungbluth (54))

```

SELECT BestellpositionID, BestellungID, ArtikelID, Einzelpreis,
Mehrwertsteuersatz, Menge, Rabatt
FROM
dbo.tblBestellpositionen
WHERE
BestellungID = 147;

```

100 %

Ergebnisse Meldungen

	BestellpositionID	BestellungID	ArtikelID	Einzelpreis	Mehrwertsteuersatz	Menge	Rabatt
1	279	147	2	29,90	0,19	3	1,00
2	437	147	112	340,00	0,19	1	0,00
3	546	147	111	340,00	0,19	2	0,00
4	716	147	114	350,00	19,00	1	0,00
5	717	147	115	350,00	19,00	1	5,00
6	718	147	7	5995,00	19,00	1	0,00
7	721	147	3	350,00	19,00	1	0,00
8	722	147	5	350,00	19,00	1	5,00
9	725	147	108	350,00	19,00	1	0,00
10	726	147	109	350,00	19,00	1	5,00

Die Abfrage wurde erfolgreich ausgeführt. ASQL (11.0 RTM) ASQLVungbluth (54) AEMA_SQL 00:00:00 10 Zeilen

Bereit Zeile 9 Spalte 1 EINFÜG

Abbildung 12.6: die Guten ins Töpfchen!

Drücken Sie nun die Taste *F5*, um das T-SQL-Skript auszuführen und den Trigger anzulegen. Anschließend geben Sie in einem neuen Abfragefenster diese SQL-Anweisungen ein:

```

UPDATE dbo.tblBestellpositionen SET Menge = 3 WHERE BestellpositionID = 279;
SELECT BestellpositionID, BestellungID, ArtikelID, Einzelpreis, Mehrwertsteuersatz,
Menge, Rabatt,
        LetzteAenderungVon, LetzteAenderungAm
FROM dbo.tblBestellpositionen WHERE BestellpositionID = 279;

```

Der Datensatz der Bestellposition mit der ID 279 wurde nicht nur mit der Menge 3 aktualisiert, sondern auch durch den Trigger mit den Änderungsdaten ergänzt (siehe Abbildung 12.7).

Dasselbe funktioniert mit der folgenden *INSERT*-Anweisung, wie Abbildung 12.8 beweist:

```

INSERT INTO tblBestellpositionen (BestellungID, ArtikelID, Einzelpreis,
Mehrwertsteuersatz, Menge)
VALUES (147, 110, '59.95', '0.19', 1);

```

Die Programmierung eines *AFTER*-Triggers ist nicht so aufwendig wie die eines *INSTEAD OF*-Triggers. Schließlich ersetzt er nicht die tatsächliche Aktion, sondern er ergänzt diese nur. In einem *INSTEAD OF*-Trigger sind Sie je nach Programmlogik gezwungen, die eigentliche Anweisung im Trigger nochmals zu programmieren – wie in unseren Beispielen zum *INSTEAD OF*-Trigger.

Wäre es da nicht sinnvoller, die Prüfungen vom *INSTEAD OF*-Trigger in den *AFTER*-Trigger zu verlagern? Ein *AFTER*-Trigger wird zwar nach der Datenverarbeitung ausgeführt, jedoch könnte

Kapitel 12 Trigger

bei einem negativen Ergebnis einer Prüfung der Befehl *ROLLBACK* die Aktion abbrechen und die Tabelle in den Zustand vor der Verarbeitung der Daten zurücksetzen. Das Ergebnis wäre gleich einem Abbruch in einem *INSTEAD OF*-Trigger: Die Tabelle hat denselben Zustand wie vor der Datenverarbeitung.

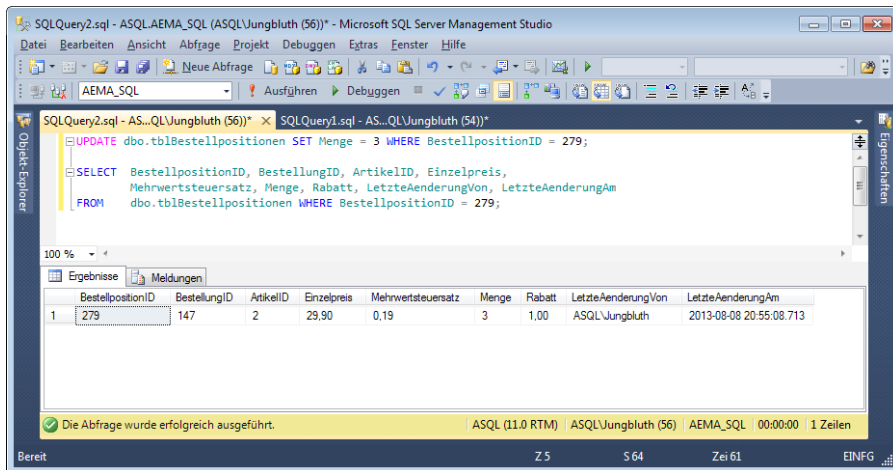


Abbildung 12.7: Der AFTER-Trigger bei einer UPDATE-Anweisung

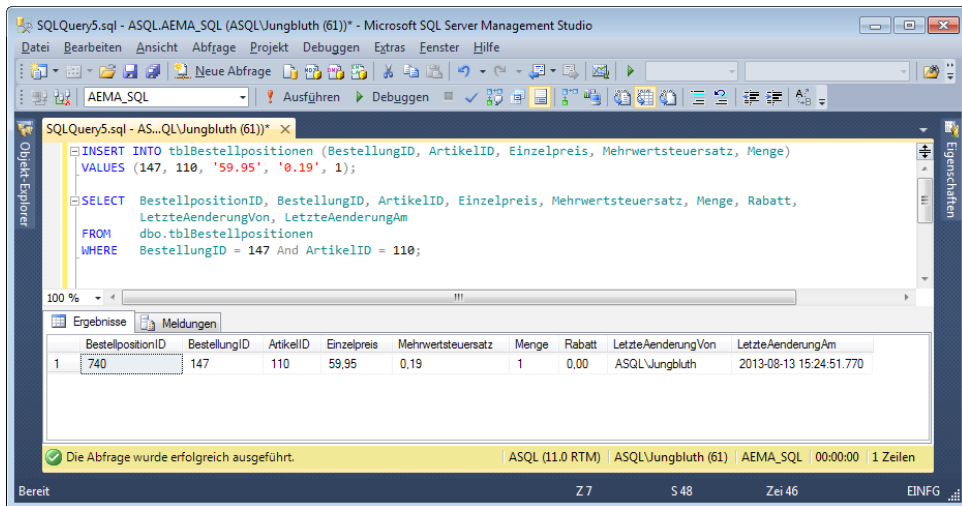


Abbildung 12.8: Der AFTER-Trigger bei einer INSERT-Anweisung

Beantworten wir die Frage anhand unseres *INSTEAD OF*-Triggers *tr_tblBestellpositionenInstead-OfInsertUpdate*. Die Logik von Variante 1 könnte durchaus ein *AFTER*-Trigger übernehmen. Ist der Rabatt zu hoch, sorgt ein *ROLLBACK* dafür, dass die Verarbeitung abgebrochen und die Ta-

belle in den Zustand vor der Datenverarbeitung gesetzt wird. Die zweite Variante des *INSTEAD OF*-Triggers lässt sich jedoch nicht mit einem *AFTER*-Trigger realisieren. Hierbei sollten wenigstens die Datensätze mit korrektem Rabatt in der Tabelle gespeichert werden. Ein *ROLLBACK* wirkt sich jedoch immer auf die gesamte Aktion aus, weshalb die Tabelle am Ende keine der verarbeiteten Datensätze enthält.

Nicht nur die Logik ist entscheidend, ob Sie die Prüfungen im *INSTEAD OF*-Trigger oder im *AFTER*-Trigger durchführen. Ein weiterer wichtiger Faktor ist die Performance. Es macht durchaus einen Unterschied, ob Sie die Gültigkeit einer Datenverarbeitung am Anfang durch einen *INSTEAD OF*-Trigger oder erst am Ende mit einem *AFTER*-Trigger prüfen, um dann gegebenenfalls die Verarbeitung abubrechen.

12.5 Trigger bei MERGE und TRUNCATE TABLE

Seit SQL Server 2008 gibt es die SQL-Anweisung *MERGE*, mit der Sie die Daten einer Quelle mit den Daten einer Zieltabelle abgleichen können. Je nach dem Stand der Daten werden hierbei in der Zieltabelle Daten gelöscht, geändert oder der Tabelle hinzugefügt – mehr dazu in Kapitel »T-SQL-Grundlagen«, Seite 221.

Bleibt die Frage, ob *MERGE*-Anweisungen auch Trigger auslösen können. Nicht direkt, denn der *MERGE*-Befehl selbst ändert keine Daten. Er steuert lediglich, für welche Datensätze ein *INSERT*, *UPDATE* oder *DELETE* ausgeführt wird. Haben Sie also beispielsweise eine Tabelle mit einem *UPDATE*-Trigger und Sie führen an dieser Tabelle einen *MERGE*-Befehl aus, der unter anderem Daten ändert, löst dies den *UPDATE*-Trigger der Tabelle aus.

Die gesamten Daten einer Tabelle können Sie mit der *DELETE*-Anweisung löschen oder auch per *TRUNCATE TABLE*. Dabei sollten Sie bedenken, dass ein *DELETE*-Trigger bei der Anweisung *TRUNCATE TABLE* nicht ausgelöst wird. Die Anweisung *TRUNCATE TABLE* löscht nicht wie *DELETE* die Daten einer Tabelle, sondern lediglich die Zuordnungen der Datenseite zu einer Tabelle. Mehr zu *TRUNCATE TABLE* lesen Sie im Kapitel »T-SQL-Grundlagen«, Seite 221.

Mit den Triggern wäre nun auch das letzte verfügbare SQL Server-Objekt beschrieben. Trigger mögen nicht unumstritten sein. Es gibt gute Gründe für und gegen den Einsatz von Triggern. Die Vor- und Nachteile haben Sie in diesem Kapitel kennengelernt. Die Entscheidung, ob Sie Trigger verwenden oder nicht, liegt nun bei ganz Ihnen.