Kapitel 23

Serverseitige Programmierung

In diesem Kapitel:

Für viele Entwickler stellt gerade die Programmierung von Datenbankobjekten mithilfe der .NET-CLR ein spannendes Thema dar. In diesem Kapitel werden die wichtigsten Objekte vorgestellt, die Sie für SQL
Server 2005 erstellen können. Bei der Programmierung sollten dabei immer die Sicherheit und Robustheit des Codes im Vordergrund stehen.

Funktionen

Das einfachste SQL-Objekt, das Sie mit .NET programmieren können, ist die Funktion. Da können Sie sich so richtig austoben. Sie können alle .NET-Vorteile gnadenlos ausspielen! Wenn Sie es wünschen, dann können Sie sogar eigene Klassen (in externen DLLs) mit einbinden! Dabei ist jedoch immer etwas Umsicht gefordert. Man sollte eins nie vergessen: SQL Server ist immer noch eine Datenbank! Sie müssen sich darüber im Klaren sein, dass eine Funktion mehrfach parallel ausgeführt wird. Je nachdem, wie stark die Datenbank frequentiert wird, sollte eine Funktion immer möglichst schlank gehalten sein. Rechenintensive Prozeduren haben an dieser Stelle ebenso wenig zu suchen wie die Verwendung von globalen Variablen. Wer schon einmal Webanwendungen programmiert hat, wird diese Art der *stateless*-Programmierung kennen. Für komplexere und aufwändigere Methoden sollte man deshalb lieber einen eigenständigen Applikationsserver in Erwägung ziehen, um nicht die Datenbank zusätzlich zu belasten.

Grundlagen

Eine SQL-Funktion besteht aus einer Methode, die mit oder ohne Parameter definiert werden kann. Als Rückgabewerte kommen entweder skalare Einzelwerte oder ganze Tabellen in Frage. Eine Methode kann zusätzlich mit Attributen versehen werden. Damit kann man festlegen, wie SQL Server die Funktion verwenden kann, und welche internen Optimierungen (z. B. Zwischenspeichern von Ergebnissen) er durchführen darf.

Attributname	Mögliche Werte	Beschreibung
IsDeterministic	True False	Sind die Rückgabewerte der Funktion deterministisch, also unabhängig von Zeit und Ort immer gleich?
DataAccess	DataAccessKind.None DataAccessKind.Read	Aktiviert den Zugriff auf die Datenbank.
SystemDataAccess	SystemDataAccessKind.None SystemDataAccessKind.Read	Aktiviert den Zugriff auf Systemtabellen.
IsPrecise	True False	Ist der Rückgabewert numerisch vergleichbar?

Tabelle 23.1 Attribute der CLR-Funktionen

Es werden grundsätzlich zwei Arten von Funktionen unterschieden:

- *Skalarfunktionen*, die einen numerischen Wert zurückgeben. Dabei kann jeder Datentyp als Rückgabewert in Frage kommen, mit folgenden Ausnahmen: *RowVersion*, *Text*, *NText*, *Image*.
- *Funktionen mit Tabellenrückgabe*, die, wie der Name schon sagt, eine Tabelle als Rückgabewert besitzen.

Wir werden uns im Folgenden die beiden Arten genauer anschauen.

Nach der Kompilierung der Methoden und Registrierung der Assembly in SQL Server kann die Methode mit folgender Syntax in SQL Server als Funktion angelegt werden:

```
CREATE FUNCTION [ schema_name. ] function_name
( { @parameter_name [AS] [ type_schema_name. ] parameter_data_type
        [ = default ] }
    [ ,...n ]
)
RETURNS { return_data_type | TABLE <clr_table_type_definition> }
    [ WITH <clr_function_option> [ ,...n ] ]
    [ AS ] EXTERNAL NAME assembly_name.class_name.method_name
[ ; ]
```

Listing 23.1 Syntax der CREATE FUNCTION – Anweisung für CLR-Funktionen

Bei der Programmierung von Funktionen sind zunächst zwei Faktoren besonders zu berücksichtigen. Der eine ist die korrekte Verwendung der SQL-Datentypen. Im Einführungskapitel zur .NET-Programmierung finden Sie eine übersichtliche Tabelle, in der die entsprechenden SQL-Datentypen den normalen .NET-Datentypen gegenübergestellt sind. Die Schnittstellen der Funktion nach außen (also die Inputparameter und die Rückgabewerte) sollten immer SQL-Datentypen sein, z. B. *SqlInt32* statt *Int32*. Zwar würde der Code kompilieren und sich auch auf dem SQL-Server registrieren lassen, aber wir handeln uns spätestens beim Umgang mit *NULL*-Werten große Probleme ein. Das ist dann auch der zweite wichtige Punkt, denn wir brauchen einen robusten Umgang mit diesen *NULL*-Werten. Deshalb sollten alle Funktionen, die *NULL*-Werte zulassen (oder auch nicht) immer erst einmal alle Parameter auf *NULL* abprüfen und dann die entsprechenden Konsequenzen ziehen. Ob Sie innerhalb der Funktion die Datentypen konvertieren, liegt an der geplanten Verarbeitung innerhalb des Codes. Normalerweise kostet das Konvertieren immer extra Zeit, deshalb sollte man nach Möglichkeit darauf verzichten. Trotzdem ist es nicht verkehrt, bei umfangreichen Logiken (mit vielen Schleifen) die Ausführungszeit im Auge zu behalten. Gerade spezielle SQL-Datentypen wie *SqlDecimal*, die einen größeren Wertebereich als der .NET-*Decimal*-Typ haben, brauchen mehr Ressourcen und damit Verarbeitungszeit. Dann ist es unter Umständen sinnvoller, die Daten intern in einen .NET-Datentyp zu konvertieren und dann damit weiterzurechnen, natürlich nur, wenn man sich über den nötigen Wertebereich im Klaren ist.

Einfache Skalarfunktion ohne Parameter

Hier zunächst einmal ein einfaches Beispiel für eine Funktion:

```
[SqlFunction]
        public static SqlString MitarbeiterDerWoche()
        {
```

```
                return "Alexander Köller";
        }
```

Listing 23.2 Einfache Funktion

Auf den ersten Blick scheint es kein Problem mit den Datentypen zu geben, da die Funktion schon idealerweise einen *SqlString* zurückgibt. Der Code ist also schlank und robust.

Das Registrieren der Assembly und der Funktion sieht dann so aus:

```
CREATE ASSEMBLY TestFunction from 'C:\Beispiele\TestFunction.dll' WITH PERMISSION_SET = SAFE
CREATE FUNCTION MitarbeiterDerWoche() RETURNS Nvarchar(50)
AS
EXTERNAL NAME TestFunction.UserDefinedFunctions.MitarbeiterDerWoche
-- Aufruf der Funktion
select dbo.MitarbeiterDerWoche()
```

Listing 23.3 Registrierung und Aufruf einer Funktion

Wie Sie sehen, ist eine einfache Funktion ohne große Umstände zu programmieren. Beim Erstellen ist aber auf jeden Fall zu beachten, den vollständigen Namen in der Form *Assemblyname.Namensraum.Funktionsname* beim Definieren des externen Namens zu berücksichtigen.

Skalarfunktionen mit Parametern

In den seltensten Fällen kommt man mit einfachen Funktionen ohne Parameter aus. Interessanter wird die Sache deswegen, wenn wir Parameter dazu nehmen. Als Beispiel dient diesmal eine Funktion, die den Wert des Eingabeparameters verdoppelt:

```
 [SqlFunction]
        public static Int32 Verdoppeln(Int32 wert)
        {
                return (wert+wert);
        }
```

Listing 23.4 Funktion mit Parametern

Bei der Registrierung der Funktion ist nun der Parameter mit zu berücksichtigen. Das sieht dann so aus:

```
CREATE FUNCTION Verdoppeln(@Wert int) RETURNS int
AS
EXTERNAL NAME TestFunction.UserDefinedFunctions.Verdoppeln
```

Listing 23.5 Registrierung einer Funktion mit Parametern

Der Aufruf der Funktionen in SQL Server ist analog wie die beliebiger anderer Funktionen vorzunehmen, der Parameter wird dabei in Klammern übergeben:

```
select dbo.Verdoppeln(12)
```

Listing 23.6 Funktionsaufruf

Datentyp-Problematik

Auf den ersten Blick funktioniert bei der Verdoppeln-Methode alles korrekt. Das ist ein gutes Beispiel dafür, wie man aber nicht programmieren sollte. Beim Ausführen der folgenden Anweisung auf der Datenbank sieht es dann schon nicht mehr so schön aus:

```
select dbo.Verdoppeln(NULL)
-- folgende Meldung wird zurückgegeben:
Meldung 6569, Ebene 16, Status 1, Zeile 2
Fehler bei 'Verdoppeln', weil der 1-Parameter nicht NULL sein darf und kein Ausgabeparameter ist.
```

Listing 23.7 Fehlerhafter Aufruf

Bei der Definition der Funktion wurde die Behandlung von *NULL*-Werten nicht berücksichtigt! Jetzt kann man natürlich auf die Idee kommen, die Datentypen entsprechend anzupassen. Die Funktion könnte dann wie folgt aussehen:

```
[SqlFunction]
        public static SqlInt32 Verdoppeln(SqlInt32 wert)
        {
                return (wert.value + wert.value);
        }
```

Der Aufruf der Funktion mit *NULL* als Parameter wirkt sich dafür noch verheerender aus:

```
Meldung 6522, Ebene 16, Status 2, Zeile 1
.NET Framework-Fehler beim Ausführen der benutzerdefinierten Routine oder des benutzerdefinierten
Aggregats 'Verdoppeln':
System.Data.SqlTypes.SqlNullValueException: Data is Null. This method or property cannot be called on
Null values.
System.Data.SqlTypes.SqlNullValueException:
   at System.Data.SqlTypes.SqlInt32.get_Value()
   at UserDefinedFunctions.Multiplizieren(SqlInt32 wert)
```

Um diesen recht einfachen Code nun *NULL-sicher* zu machen, sollte man also zunächst den Parameter auf *NULL* prüfen, bevor man damit arbeitet.

```
if (wert.IsNull)
   return wert;
else
{
   return (wert.Value * 2);
}
```

Listing 23.8 Abprüfen auf NULL-Parameter

HINWEIS Für die einfachere Verwendung von SQL-Datentypen wurden die Standardoperatoren überladen. Wenn man im obigen Beispiel statt

```
wert.value + wert.value
```

einfach nur

```
wert + wert
```

schreibt, so werden die *NULL*-Werte korrekt verarbeitet. In diesem Fall wird die Überladung des +-Operators für den *SqlInt32*-Datentyp aufgerufen, und dieser kann mit *NULL*-Werten so umgehen, dass das Ergebnis gleich *NULL* ist. Das funktioniert aber nur so lange, bis ein normaler .NET-Datentyp in die Operation verwickelt wird. Dann wird wieder eine Ausnahme ausgelöst.

Der Umgang mit SQL-Datentypen ist also alles andere als trivial. Ein weiterer Beweis dafür mag der Vergleich von zwei Werten bieten. In folgender Funktion werden die beiden Eingabeparameter miteinander verglichen.

```
[SqlFunction()]
    public static SqlString Vergleich(SqlInt32 wert1, SqlInt32 wert2)
    {
        if (wert1 == wert2)
            return ("Werte sind gleich!");
        if (wert1 > wert2)
            return ("Wert 1 ist größer als Wert 2!");
        else
            return ("Wert 1 ist kleiner als Wert 2!");
        //}
    }
```

Listing 23.9 CLR-Funktion mit Vergleich von Parametern

Wenn jetzt einer der Parameter gleich *NULL* ist, dann ist das Ergebnis immer »Wert 1 ist kleiner als Wert 2«! Natürlich lautet jetzt die Frage, warum das so ist. Ganz einfach: Jeder Vergleich mit mindestens einem *NULL*-Wert als Beteiligtem wird als »Falsch« ausgewertet. Deshalb landet die Programmausführung immer im zweiten *Else*-Zweig. Man kann also auch zwei *NULL*-Werte auf diese Weise nicht auf Gleichheit überprüfen. Es gibt aber dafür einen anderen Weg, zum Beispiel über die *CompareTo*-Methode.

```
[SqlFunction()]
    public static SqlString Vergleich2(SqlInt32 wert1, SqlInt32 wert2)
    {
        int i = wert1.CompareTo(wert2);
        switch (i)
        {
            case 0:
                return ("Werte sind gleich!");
            case 1:
                return ("Wert 1 ist größer als Wert 2!");
            case -1:
                return ("Wert 1 ist kleiner als Wert 2!");
        }
```

```
    }
```

Listing 23.10 CLR-Funktion mit Vergleich durch CompareTo

CompareTo liefert 0 bei Gleichheit, 1 wenn der erste Wert größer ist, und –1, wenn der zweite Wert größer ist. *NULL* ist dabei immer kleiner als jeder andere mit diesem Werttyp darstellbare Wert. Dieser Vergleich funktioniert auch bei zwei *NULL*-Parametern, die dann auch als gleich ausgewertet werden.

Datenzugriff innerhalb einer Funktion

Bisher haben die Funktionen nicht auf Daten in unserer Datenbank zugegriffen. Nun versuchen wir einen Zugriff auf die Datenbank innerhalb unserer Funktion zu integrieren.

WICHTIG Um einen Datenzugriff innerhalb der Funktion zu ermöglichen, der die bestehende Verbindung (und damit die Berechtigungen) des Clients verwendet, greift man auf das SqlContext-Objekt zurück. Es ist nicht möglich, innerhalb einer CLR-Funktion eine zusätzliche eigene Verbindung zur Datenbank zu etablieren!

```
 [SqlFunction(DataAccess = DataAccessKind.Read)]
public static SqlInt32 Verdoppeln2(SqlInt32 wert)
{
  using(SqlConnection conn = new SqlConnection("context connection=true"))
  {
        conn.Open();
        SqlCommand cmd = new SqlCommand("SELECT Zahl FROM Kunde WHERE ID = 2", conn);
        int a = (int)cmd.ExecuteScalar();
        return (a*wert);
  }
}
```

Listing 23.11 Funktion mit Datenzugriff

Vielleicht ist Ihnen das Attribut aufgefallen, das der Funktion vorangestellt ist. Würden Sie die Funktion ohne dieses Attribut im SQL Server registrieren, dann würde das zwar funktionieren, jedoch der Aufruf mit der Ausnahme

```
System.InvalidOperationException: Data access is not allowed in this context.  Either the context is
a function or method not marked with DataAccessKind.Read or SystemDataAccessKind.Read, is a callback
to obtain data from FillRow method of a Table Valued Function, or is a UDT validation method.
```

abbrechen. Eine aussagekräftige Fehlermeldung! Damit also der Datenzugriff für eine Funktion möglich sein soll, muss man darauf achten, den Zugriff auf die Daten explizit zu gestatten. Dazu setzt man das Attribut *DataAccess* auf den Wert *DataAccessKind.Read*. Wenn zusätzlich der Zugriff auf Systemtabellen erlaubt werden soll, so ist das Attribut *SystemDataAccess* mit *SystemDataAccess.Read* zu verwenden.

Wird die Funktion für eine berechnete Spalte verwendet, so ist das Attribut *IsDeterministic* auf *True* zu setzen, wenn die Spalte indiziert werden soll. Das sollte eine Garantie dafür sein, dass sich der Wert der Funktion nicht ändert. Leider darf dann nicht gleichzeitig ein Zugriff auf die Daten erfolgen.

Natürlich ist auch der Zugriff auf externe Ressourcen möglich, dann muss das *PERMISSION_SET* der Assembly auf *EXTERNAL_ACCESS* oder *UNSAFE* stehen.

Der Zugriff auf eine XML-Konfigurationsdatei sieht dann zum Beispiel wie folgt aus:

```
[SqlFunction]
public static SqlInt32 TestXML(SqlInt32 wert)
{
        System.Xml.XmlDocument doc = new XmlDocument();
        doc.Load(@"C:\Austausch\Test.xml");
        XmlNode no = doc.SelectSingleNode(@"//Test");
        Int32 i = Int32.Parse(no.FirstChild.Value);
        return wert * i;
}
```

Listing 23.12 Funktion mit Zugriff auf XML-Datei

Bei der Registrierung der Funktion in SQL Server muss dann das *PERMISSION_SET* festgelegt werden:

```
CREATE ASSEMBLY TestFunction from 'C:\Beispiele\TestFunction.dll' WITH PERMISSION_SET =
EXTERNAL_ACCESS
CREATE FUNCTION TestXML (@Wert int) RETURNS int
AS
EXTERNAL NAME TestFunction.UserDefinedFunctions.TestXML
SELECT dbo.TestXML(2)
```

Listing 23.13 Registrierung und Aufruf der skalaren Funktion

Wie Sie sehen können, haben wir viele Möglichkeiten mit Funktionen zu arbeiten. Im Datenbankbereich ist es aber häufig nötig, komplette Tabellen an den Client zurückzugeben.

Funktion mit Tabellenrückgabe

Funktionen mit Tabellenrückgabe sind Funktionen, die eine Tabelle (bzw. ein *SqlReader*-Objekt) als Rückgabewert besitzen. Dabei ist aber zu beachten, dass SQL Server keine Spalten vom Typ Timestamp, Char, Varchar oder Text unterstützt. Außerdem ist die Einschränkung *NOT NULL* nicht möglich.

Dazu ein einfaches Beispiel. Es soll eine einfache Tabelle mit einer ID und einer Name-Spalte zurückgegeben werden. Anders als bei einer Skalarfunktion, die nur einen einfachen Wert zurückliefert, werden hierbei die Daten nacheinander abgerufen, wie bei einem DB-Cursor. Dies wird über ein IEnumerable-Interface realisiert.

In diesem Beispiel werden folgende Komponenten verwendet:

1. Eine Klasse, die eine Zeile der Tabelle repräsentiert, mit den Spalten als Eigenschaften.
2. Eine Klasse die das *IEnumerable*-Interface implementiert. Dazu benötigt man folgende Methoden:
3. *public object Current* – Auslesen einer Zeile
4. *public bool MoveNext* – den nächsten Datensatz bereitstellen
5. *public void Reset* – Zurücksetzen vor den ersten Datensatz

6. Eine Basisklasse, die die Funktion repräsentiert, mit folgenden Methoden:
7. *public static IEnumerator Funktionsname* – Basisfunktion
8. *public static void FillRow(object obj, Parameter1, ...)* – Übernimmt das Füllen der Zeile, dabei steht der erste Parameter vom Typ *Object* für die *Zeile*, die von der *Current*-Methode geliefert wird. Der Rest sind die Output-Parameter.

Nun der Blick auf das Beispiel. Die eingebundenen Namensräume entsprechen dem Standard für SQL-Funktionen in Visual Studio. Für die zu erstellende Tabelle braucht man zusätzlich noch den Namensraum *System.Collections*:

```
using System;
using System.Data;
using System.Data.Sql;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;
using System.Collections;
```

Als nächstes definiert man eine Klasse, die dem Aufbau der zurückzugebenden Tabelle entspricht. Dazu wird eine Eigenschaft *ID* vom Datentyp *SqlInt32* und eine zweite Eigenschaft *Name* vom Typ *SqlString* benötigt. Hier werden die Eigenschaften durch einfache öffentliche Variablen realisiert. Zusätzlich wird eine Methode mit den beiden *Spalten* als Parameter definiert, die die Eigenschaftswerte entsprechend befüllt.

```
public partial class TestTab
{
        public SqlInt32 ID;
        public SqlString Name;
        public TestTab(SqlInt32 id, SqlString name)
        {
                this.ID = id;
                this.Name = name;
        }
};
```

Nun kommt der aufwändigere Teil der Programmierung. Der Enumerator muss implementiert werden. Enumeratoren haben die gleiche Funktion wie Cursor, d.h. sie ermöglichen das sequenzielle Abrufen von Werten. Zunächst werden lokale Variablen für die Datenübergabe definiert sowie eine kleine Methode, die die Initialisierung der Werte übernimmt.

```
public partial class TabLoader:IEnumerator
{
        private SqlInt32 id;
        private SqlString name;

        public TabLoader()
        {
                this.id = 0;
```

```
                this.name = "Schmidt";
        }
```

Anschließend wird die Methode *Current* implementiert, die die aktuelle *Zeile* der Tabelle zurückgibt.

```
public object Current
{
        get
        {
                return new TestTab(id, name + id.ToString());
        }
}
```

Fehlt nur noch die Methode *MoveNext* für das Abrufen nächsten *Zeile*, im Beispiel werden an dieser Stelle die Werte der neuen *Zeile* ermittelt. Ist die letzte *Zeile* erreicht, gibt die Methode *false* als Rückgabewert, ansonsten ein *true* zurück.

```
        public bool MoveNext()
        {
                this.id = this.id + 1;
                if (this.id < 20)
                        return true;
                else
                        return false;
        }
```

Zum Schluss wird noch eine Methode zum Zurücksetzten der Werte benötigt. Damit sind alle Methoden der Enumeration implementiert.

```
        public void Reset()
        {
                this.id = 0;
        }

};
```

Jetzt fehlt nur noch die eigentliche Implementierung der Funktion. Diese ist in der Klasse *TabAccess* realisiert. Über das Attribut *FillRowMethodName* wird die Methode definiert, die für das Füllen der nächsten Zeile aufgerufen wird, über *TableDefinition* werden die Spalten der Tabelle definiert.

```
public partial class TabAccess
{
```

```
        [SqlFunction(FillRowMethodName = "FillRow", TableDefinition = "ID int, Name
nvarchar(50)")]
        public static IEnumerator GetTab()
        {
                return new TabLoader();
        }
        public static void FillRow(object obj, out SqlInt32 id, out SqlString name)
        {
                if (obj != null)
                {

                        id = (SqlInt32)((TestTab)obj).ID;
                        name = (SqlString)((TestTab)obj).Name;
                }
                else
                {
                        id = SqlInt32.Null;
                        name = SqlString.Null;
                }
        }
};
```

Das hier ausführlich beschriebene Vorgehen bei einer Funktion mit Tabellenrückgabe ist auf den ersten Blick etwas umständlich, lässt sich aber bei Tabellen, deren Inhalt komplett neu erstellt wird, kaum umgehen. Wenn Sie jedoch die benötigten Zeilen für die Rückgabe direkt aus der Datenbank über eine *DataTable* abrufen, können Sie auch auf die Implementierung einer eigenen Klasse für den Enumerator verzichten, da das *DataTable*-Objekt selber schon *IEnumerable* implementiert, wie in Listing 23.14 zu sehen.

```
…
public partial class UDFunctions
{
[Microsoft.SqlServer.Server.SqlFunction(
    DataAccess=DataAccessKind.Read,
    FillRowMethodName = "FillDataRow",
    TableDefinition = "PersonID int, Name nvarchar(50)") ]
public static IEnumerable HoleKunden()
  {
   using (SqlConnection conn = new SqlConnection("context connection=true"))
   {
    SqlCommand cmd = conn.CreateCommand();

    cmd.CommandText = "SELECT ID, LastName FROM SalesDepartment.Customers";
    cmd.CommandType = CommandType.Text;
    conn.Open();

    DataTable dt = new DataTable();
    SqlDataAdapter da = new SqlDataAdapter(cmd);
    da.Fill(dt);
    return dt.Rows;
   }
  }

  public static void FillDataRow(Object obj, out int PersonID, out string Name)
```

```
  {
   DataRow row = (DataRow)obj;

   PersonID = (int)row[0];
   Name = row[1].ToString();
  }
 };
```

Listing 23.14 Einfache Funktion mit Tabellenrückgabe

ACHTUNG Leider ist es in der bisherigen Version des SQL Servers nicht möglich, direkt einen DataReader aus einer Context-Verbindung heraus durch eine Funktion mit Tabellenrückgabe direkt nach »außen« weiterzugeben. Die Datenströme innerhalb des SQL Servers und die äußeren sind strikt voneinander getrennt.

Zusammenfassung

Letztendlich ist die Erstellung einer Funktion mithilfe von .NET nicht unbedingt einfacher als die Programmierung einer klassischen SQL-Funktion mit T-SQL. Für T-SQL spricht, dass man auch Einschränkungen (wie z. B. Primary Keys) oder Indizes innerhalb der Funktion verwenden kann, man verwaltet also eine »richtige« temporäre Tabelle.

Dagegen sind .NET-Funktionen mehr ein Datenstrom. Die Daten können sofort geliefert werden, sobald die erste Zeile eintrifft, und werden *readonly* und *forwardonly* weitergegeben.

Wichtiger Grundsatz bei der Programmierung ist die korrekte Verwendung der Datentypen sowie in diesem Zusammenhang auch das Prüfen auf *NULL*-Werte, damit die Funktion möglichst robust und sicher wird.

Gespeicherte Prozeduren

Gespeicherte Prozeduren sind die wohl am häufigsten verwendeten Programmierobjekte des SQL Servers. Sie werden ähnlich der Funktionen entweder mit oder ohne Parameter aufgerufen und haben einen skalaren Wert oder eine Tabelle als Rückgabewert. Im Gegensatz zu Funktionen können sie aber zusätzlich auch DDL- oder DCL-Anweisungen aufrufen und Rückgabeparameter besitzen. Damit sind sie speziell für Abfragen durch Client-Anwendungen geeignet.

Grundlagen

Prozeduren werden als statische Methoden mit keinem oder einem *Integer* als Rückgabewert implementiert. Folgende Attribute können verwendet werden:

Attribut	Mögliche Werte	Beschreibung
Name	*Zeichenkette*	Name der gespeicherten Prozedur, nur für Visual Studio interessant. Der