3 Mit automatischen Primärschlüsseln leben

Fragestellungen

- Kann ADO.NET mit Primärschlüsseln umgehen, die vom Datenbanksystem automatisch vergeben werden?
- Wie kommt ein automatisch vergebener PK-Wert für einen neuen Datensatz zurück in eine `DataRow`?
- Gibt es Unterschiede in der Behandlung von automatischen Primärschlüsseln bei MS Access und SQL Server?
- Können automatische PKs auch als Fremdschlüssel benutzt werden?

3.1 Die guten alten Zeiten

In Kapitel 2 über objekt-relationales Mapping (ORM) nimmt die Diskussion von Objektidentitäten einen breiten Raum ein: Jedes Objekt, das Sie in einer Datenbank speichern, muss eine eigene persistente Identität (Objekt-ID, OID) bekommen. Diese OID bildet dann den Primärschlüssel (PK) der Tabelle, in der Sie die Zustände der Objekte einer Klasse speichern. OIDs sind unveränderlich und gehören nicht zu den Geschäftsdaten einer Klasse.

Diese Eigenschaften sind für einen PK eigentlich nicht neu. Auch in Ihren bisherigen Datenbankschemata haben Sie PKs sicher schon oft so ausgelegt. Die Unveränderlichkeit eines PK macht es einfacher, die referentielle Integrität von relationalen Daten zu gewährleisten: Wenn sich ein PK nach Zuweisung nicht mehr ändern kann, dann fällt auch kein Aufwand dafür an, Änderungen an andere Datensätze weiterzureichen, bei denen er als Fremdschlüssel (FK) eingetragen ist.

Und dass wirklich keine Änderungen an einem PK stattfinden, gewährleisten Sie am besten, indem Sie keine Tabellenspalte mit Geschäftsdaten (z.B. Kunden- oder Produktnummer) als PK benutzen, sondern Tabellen mit einer speziellen PK-Spalte ausstatten (*Surrogatschlüssel*).

Für einen solchen PK stellt sich dann natürlich die Frage, woher seine Werte kommen. Als Antwort darauf bieten alle relationalen Datenbanksysteme (RDBMS) spezielle Datentypen für Spalten, deren Werte automatisch vergeben werden sollen; bei MS Access sind das die *AutoNumber*-, bei MS SQL Server die *identity*-Spalten. Allgemein kann man diese Art Spalten als *Autoincrement-Spalten* bezeichnen, weil ihnen immer ein ganzzahliger Datentyp zugrunde liegt und die Spaltenwerte ausgehend von einem Startwert (*Seed*) für jeden neuen Datensatz um einen festen Betrag (*Increment*, *Step*, üblicherweise 1) hochgezählt werden. Das RDBMS sorgt dafür, dass auch bei gleichzeitigem Zugriff von vielen Clients jeder Wert eindeutig ist.

Für die Cursor-basierte Datenbankprogrammierung war diese Philosophie völlig ausreichend und bequem. Als Beispiel hier zwei Tabellen, die jeweils mit einem Autoincrement-PK ausgestattet sind und in einer 1:n-Beziehung stehen:

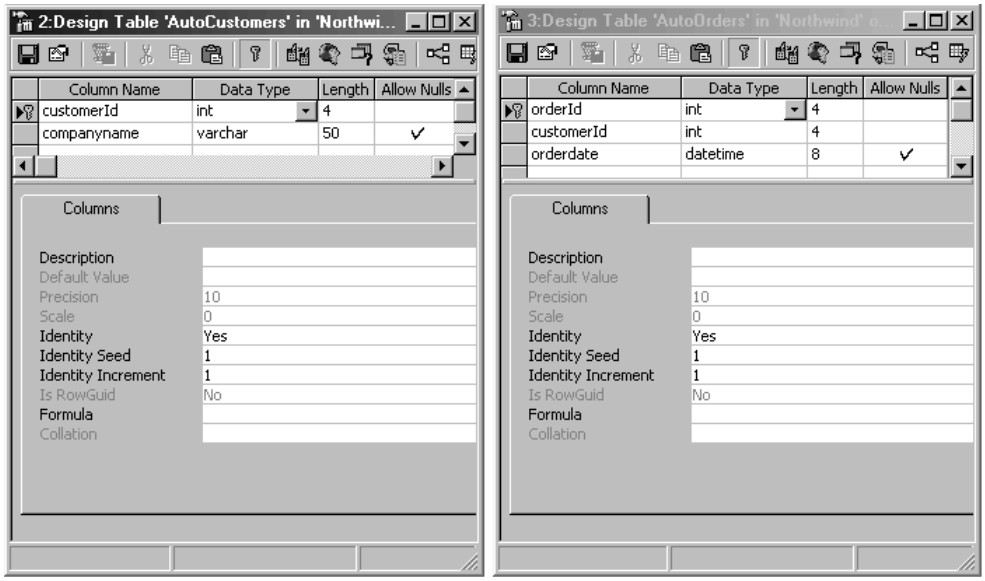

Abbildung 3.1: Zwei Tabellen mit Autoincrement-PK in einer MS SQL Server-Datenbank, die in einer 1:n-Beziehung zueinander stehen.

Wenn Sie ein Cursor-basiertes API wie ADO 2.x benutzen und in beiden Tabellen einen neuen Datensatz anlegen wollen, dann können Sie das sehr einfach tun:

```
Dim conn As New ADODB.Connection()
conn.Open("...")

Dim rsCust As New ADODB.Recordset()
rsCust.CursorLocation = ADODB.CursorLocationEnum.adUseServer
rsCust.Open("select * from AutoCustomers", conn, _
            ADODB.CursorTypeEnum.adOpenKeyset, _
            ADODB.LockTypeEnum.adLockOptimistic)

Dim rsOrd As New ADODB.Recordset()
rsOrd.CursorLocation = ADODB.CursorLocationEnum.adUseServer
rsOrd.Open("select * from AutoOrders", conn, _
           ADODB.CursorTypeEnum.adOpenKeyset, _
           ADODB.LockTypeEnum.adLockOptimistic)

rsCust.AddNew()
rsCust.Fields("companyname").Value = ...
rsCust.Update()

rsOrd.AddNew()
rsOrd.Fields("customerid").Value = rsCust.Fields("customerid").Value
rsOrd.Fields("orderdate").Value = ...
rsOrd.Update()
```

Dem Autoincrement-PK (Spalte `customerId` bzw. `orderId`) weisen Sie dabei keinen Wert zu. Er wird vom Datenbanksystem automatisch vergeben und steht spätestens nach dem Aufruf von `Update()` auf dem jeweiligen Cursor zur Verfügung.[1]

Das ist entscheidend dafür, dass der PK-Wert eines Parent-Datensatzes (hier: `rsCust.Fields("customerid")`) sofort in einen neuen Child-Datensatz als FK (in diesem Fall: `rsOrd.Fields("customerid")`) eingetragen werden kann!

In der guten alten Welt der Cursor-basierten APIs können also auch Beziehungen zwischen Datensätzen, die auf einem Autoincrement-PK beruhen, verlässlich und endgültig hergestellt werden. Bei ADO.NET ist das nicht der Fall.

3.2 Schöne neue Welt?

Natürlich können Sie auch in ADO.NET Tabellen mit einem Autoincrement-PK ausstatten:

```
Dim ds As New DataSet()
With ds.Tables.Add("AutoCustomers")
    With .Columns.Add("customerId", GetType(Integer))
        .AutoIncrement = True
        .AutoIncrementSeed = 0
        .AutoIncrementStep = 1
        .ReadOnly = True
    End With
    .PrimaryKey = New DataColumn() {.Columns("customerid")}

    .Columns.Add("companyname", GetType(String))
End With
```

Sie wählen dafür einen Zahlendatentyp als Grunddatentyp für die PK-Spalte und setzen dann die `AutoIncrement...`-Eigenschaften der Spalte.

Neue Datensätze anzulegen ist dann genauso einfach wie bei einem Cursor-basierten API:

```
With ds.Tables("AutoCustomers")
    Dim rCust As DataRow
    rCust = .NewRow
    rCust("companyname") = "Peter´s Inn"
    .Rows.Add(rCust)

    rCust = .NewRow
    rCust("companyname") = "Paul´s Bar"
    .Rows.Add(rCust)
End With
```

[1] Bei MS Access-Datenbanken z. B. auch schon sofort nach `AddNew()`.

Die Werte für die Autoincrement-Spalte beginnen dann beim Seed-Wert und wachsen jeweils um den Step-Wert:

```
<NewDataSet>
  <AutoCustomers>
    <customerId>0</customerId>
    <companyname>Peter´s Inn</companyname>
  </AutoCustomers>
  <AutoCustomers>
    <customerId>1</customerId>
    <companyname>Paul´s Bar</companyname>
  </AutoCustomers>
</NewDataSet>
```

3.2.1 Autoincrement-Spalten im Datenbankschema

Wenn Sie Daten aus einer Datenbanktabelle laden wollen, die einen Autoincrement-PK hat, müssen Sie natürlich die `AutoIncrement...`-Eigenschaften der Spalte nicht selbst setzen:

```
Dim ds As New DataSet()
Dim conn As New SqlConnection("...")
Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
adap.MissingSchemaAction = MissingSchemaAction.AddWithKey
adap.Fill(ds, "AutoCustomers")
```

Der `DataAdapter`, mit dem Sie die ausgewählten Daten laden, legt selbst die Zieltabelle mit allen Spalten an.

In der Standardeinstellung lässt er Autoincrement-Einstellungen und Primärschlüsseldefinition jedoch außen vor. Sie müssen ihn explizit anweisen, auch diese Angaben in das `DataSet`-Schema zu übernehmen, indem Sie vor dem Aufruf von `Fill()` die Eigenschaft `MissingSchemaAction` auf `AddWithKey` setzen.[2]

Wo immer Sie lediglich `Fill()` zum Laden von Daten benutzen, sollten Sie sich zur Angewohnheit machen, `MissingSchemaAction` so zu belegen. Auch wenn Sie keinen Autoincrement-PK benutzen, können Sie dann zumindest sicher sein, dass im `DataSet` ein Primärschlüssel-Constraint hinterlegt ist.

Falls Sie das Schema einer Tabelle allerdings nicht »in einem Rutsch« mit den Daten laden wollen, sondern getrennt davon, können Sie `FillSchema()` aufrufen:

```
adap.FillSchema(ds, SchemaType.Source, "AutoCustomers")
```

[2] Warum das beim SQL Server Managed Provider notwendig ist, der im Schema hinter einem `SqlDataReader` – mittels dessen ein `SqlDataAdapter` eine `DataTable` aufbaut und füllt – ohnehin Schlüssel- und Autoincrement-Informationen transportiert, ist allerdings nicht ganz einsichtig.
Ein `OleDbDataAdapter` hingegen müsste intern einen `OleDbDataReader` ausdrücklich über die Angabe von `CommandBehavior.KeyInfo` beim Aufruf von `ExecuteReader()` auf einem `OleDbCommand` öffnen (z.B. beim Zugriff auf eine MS Access-Datenbank), um diese Informationen zu erhalten. Das mag intern zu aufwändig sein, als dass Microsoft es als Standardverhalten hätte implementieren wollen.

`FillSchema()` lädt selbstständig alle Schlüssel- und Autoincrement-Einstellungen.

3.2.2 Beziehungen über Autoincrement-PKs herstellen

Einzelne Tabellen mit einem Autoincrement-PK zu bearbeiten, ist also zunächst kein Problem. Wie sieht es aber mit mehreren Tabellen aus, die über einen Autoincrement-PK in Beziehung stehen? Am besten versuchen Sie, das obige ADO 2.x Szenario mit ADO.NET-Mitteln nachzustellen.

Zunächst laden Sie die Daten der Parent-Tabelle in ein `DataSet`:

```
Dim ds As New DataSet()
Dim conn As New SqlConnection("...")

Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
adap.MissingSchemaAction = MissingSchemaAction.AddWithKey
adap.Fill(ds, "customers")
```

Danach fügen Sie die Datensätze der Child-Tabelle hinzu:[3]

```
adap.SelectCommand.CommandText = "select * from AutoOrders"
adap.Fill(ds, "orders")
```

Und abschließend bauen Sie eine Beziehung zwischen beiden auf:

```
ds.Relations.Add("custord", ds.Tables("customers").Columns("customerid"), _
                            ds.Tables("orders").Columns("customerid"), _
                            True)
```

Jetzt sind Sie bereit, Datensätze in beiden Tabellen anzulegen:

```
Dim rCust, rOrd As DataRow
With ds.Tables("customers")
    rCust = .NewRow
    rCust("companyname") = ...
    .Rows.Add(rCust)
End With

With ds.Tables("orders")
    rOrd = .NewRow
    rOrd("customerid") = rCust("customerid")
    rOrd("orderdate") = ...
    .Rows.Add(rOrd)
End With
```

[3] Nur der Einfachheit halber werden hier weder Parent- noch Child-Daten eingeschränkt. In Ihren Anwendungen sollten Sie natürlich niemals `select`-Anweisungen der Form `select * from tabelle` benutzen.

Die ADO.NET-Implementation des 1:n-Szenarios unterscheidet sich prinzipiell nicht von der in ADO 2.x. Zuerst legen Sie einen Parent-Datensatz an (`rCust`), anschließend den Child-Datensatz (`rOrd`), dessen FK-Feld (`customerId`) den Wert des Parent-PK (`rCust("customerid")`) erhält.

Anschließend können Sie sofort über den Parent-Datensatz auf seine Children zugreifen, z.B.:

```
rCust.GetChildRows("custord")(0)("orderdate")
```

Die `DataRelation` zwischen den beiden Tabellen macht es möglich.

Die schöne neue Welt von ADO.NET scheint also in Ordnung zu sein. Warum fordern dann die ORM-Regeln, keine Autoincrement-Spalten als PKs einzusetzen? Die Probleme kommen mit dem Speichern!

3.3 Autoincrement-PKs speichern

3.3.1 Das Problem: Datenhaltung in einem Cache

Solange Sie Datensätze mit einem Autoincrement-PK dort erzeugen und benutzen, wo die Daten gehalten werden, ist alles in Ordnung. Bei ADO 2.x ist das die Datenbank selbst, also sogar das Speichermedium, bei ADO.NET ist das eine `DataTable` bzw. ein `DataSet`, also ein Cache.

Der Cache hält die Daten und vergibt auch die Autoincrement-PK-Werte. Innerhalb des Cache können die Daten damit aufeinander Bezug nehmen und bleiben konsistent. Der Cache ist jedoch nicht der endgültige Speicherort für Ihre Datensätze. Sie wollen sie früher oder später in einer Datenbank speichern.

Wie aber können Sie einen Autoincrement-Wert, der vom Cache vergeben wurde, in einer Datenbank speichern, die für sich beansprucht, ausschließlich selbst Autoincrement-Werte zuzuweisen? Die simple Antwort lautet: gar nicht.

Überlegen Sie einmal was passiert, wenn Sie einen Datensatz an eine Tabelle mit einem Autoincrement-PK anfügen und speichern:

```
Dim ds As New DataSet()
Dim conn As New SqlConnection("...")
Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
adap.MissingSchemaAction = MissingSchemaAction.AddWithKey
adap.Fill(ds, "AutoCustomers")

With ds.Tables("AutoCustomers")
    Dim rCust As DataRow
    rCust = .NewRow
    rCust("companyname") = "Peter´s Inn"
    .Rows.Add(rCust)
End With
```

Das DataSet hat nun z.B. folgenden Inhalt (falls die Tabelle keine Sätze mehr enthielt):

```
<NewDataSet>
  <AutoCustomers>
    <customerId>0</customerId>
    <companyname>Peter´s Inn</companyname>
  </AutoCustomers>
</NewDataSet>
```

Der neue Datensatz hat 0 als Wert für den Autoincrement-PK zugewiesen bekommen. Jetzt speichern Sie ihn:

```
Dim cb As New SqlCommandBuilder(adap)
adap.Update(ds, "AutoCustomers")
```

Dafür erzeugt der SqlCommandBuilder eine SQL insert-Anweisung ähnlich der folgenden:

```
insert into AutoCustomers (companyname) values ("Peter's Inn")
```

Die PK-Spalte taucht darin nicht auf! Der SqlCommandBuilder weiß, dass eine Autoincrement-Spalte beim Einfügen oder Ändern eines Datensatzes nicht zugewiesen werden darf. Das RDBMS vergibt beim Einfügen ja selbst einen Wert, der dann unveränderlich ist.

Welchen customerId-Wert erhält der neue Datensatz aber? Das ist ungewiss und hängt davon ab, welchen Stand der tabelleninterne Zähler in der Datenbank hat. In jedem Fall entspricht der Autoincrement-Wert des PK in der Tabelle kaum dem, den das DataSet zugewiesen hat.

Die Daten im Cache und die in der Datenbank entsprechen damit nach dem Speichern nicht mehr einander, sie laufen auseinander. Das mag kein Problem sein, wenn Sie das DataSet nach dem Speichern nicht mehr benutzen.

Aber was passiert, wenn Sie den gerade angefügten Datensatz nach dem Speichern noch einmal verändern und wieder speichern wollen? Intern hat er den PK-Wert 0, in der Datenbank ist er aber z.B. 56. Ein nochmaliger Aufruf von Update() würde die Veränderung ungefähr wie folgt an das Datenbanksystem melden:

```
update AutoCustomers set companyname = "..." where customerid=0 ...
```

Es würde versucht, den neuen Datensatz über seinen PK zu lokalisieren. Das aber schlüge fehl, weil im DataSet nicht der PK steht, den die Datenbank beim vormaligen insert vergeben hat! Was tun?

3.3.2 Die Lösung: Synchronisation von Cache und Datenbank

Damit Sie mit neuen Datensätzen in einem DataSet vernünftig weiterarbeiten können, deren PK eine Autoincrement-Spalte ist, müssen Sie den vom Datenbanksystem vergebenen PK-Wert nach dem Einfügen zurück an das DataSet melden. Nur dann ist der Datensatz bei einem späteren Speichern auch in der Datenbank auffindbar. Sie müssen Datenbank und DataSet also *synchronisieren*.

Bei ADO 2.x war das nicht nötig, weil neue Datensätze unmittelbar in der Datenbank gespeichert wurden. Der zugewiesene Autoincrement-PK-Wert kam damit schon vom RDBMS und war endgültig. Cursor und Datenbankinhalt entsprachen einander immer. Mit einem Cursor haben Sie also immer auf Live-Daten gearbeitet.[4]

Mit einem `DataSet` hingegen arbeiten Sie immer auf Kopien von Daten. Und die entfernen sich vom Original in der Datenbank durch jede Änderung an ihnen bzw. in der Datenbank. Immer auf Datenkopien in einem Cache zu arbeiten, der nicht mit der Datenquelle verbunden ist, das macht das entscheidend Neue von ADO.NET gegenüber bisherigen Datenzugriffs-APIs aus. Ein anderes Programmiermodell als bisher ist erforderlich. Und symptomatisch dafür sind die Schwierigkeiten mit Autoincrement-Spalten.

Die notwendige Synchronisation von `DataSet` und Datenbank erfordert nicht nur Kodierungsaufwand, sondern kostet vor allem Performance: Eine »normale« Änderung, die Sie per SQL `insert/update/delete` an ein Datenbanksystem schicken, muss den Weg nur in eine Richtung zurücklegen. Eine `insert`-Anweisung für einen Datensatz mit Autoincrement-Spalte hingegen erfordert, dass auch noch ein Wert zurücktransportiert wird, der ins `DataSet` einzupflegen ist – und erfordert darüber hinaus womöglich einen zweiten Weg zum Datenbanksystem, um diesen Wert zu ermitteln.

Sie werden jetzt verstehen, warum die ORM-Regeln fordern, OIDs nicht als Autoincrement-PKs zu realisieren. Autoincrement-PKs sind umständlich und imperformant.

Aber die Verhältnisse in der Realität entsprechen natürlich nicht immer den Forderungen der Theorie. Sie werden leider immer wieder über Autoincrement-PKs in Datenbanken stolpern, die Sie nicht den ORM-Forderungen anpassen können. Sie müssen sich daher der Notwendigkeit zur Synchronisation zwischen `DataSet` und Datenbank stellen.

Dafür aber brauchen Sie die Unterstützung des Datenbanksystems, denn wie sollen Sie den Wert einer Spalte für einen Datensatz ermitteln, den Sie nicht identifizieren können, weil Sie seinen PK nicht kennen?

3.3.3 Autoincrement-PK synchronisieren mit MS SQL Server

Das Problem der Synchronisationsveränderungen in einer Datenbank mit ihren Clients ist natürlich nicht so neu, wie es vielleicht scheinen mag. Es tritt immer da auf, wo Sie neue Datensätze nicht direkt mittels eines Cursors anlegen.

Sobald Sie einem Datenbanksystem einen Befehl zur Datenanlage schicken, halten Sie dessen Ergebnis nicht unmittelbar in der Hand. Das gilt für eine einzelne `insert`-Anweisung – aber auch für eine *Stored Procedure*.

Mit einer Stored Procedure haben Sie aber die Chance, Informationen vom Datenbanksystem zurückgemeldet zu bekommen. Sie können z.B. die folgende Stored Procedure definieren, um einen neuen Datensatz anzulegen:

[4] Der ADO 2.x-clientseitige Cursor sei an dieser Stelle ausgenommen.

```
CREATE PROCEDURE dbo.AutoCustomersInsert
       @companyname varchar(50),
       @newCustomerId int OUT
AS
       ...
       set @newCustomerId = ...
```

Stored Procedures – die ja Funktionen in Hochsprachen wie C# oder VB .NET entsprechen, allerdings im Datenbanksystem laufen – kennen nicht nur Eingabeparameter (hier: `@companyname`), sondern auch Ausgabeparameter (hier: `@newCustomerId`) und Rückgabewerte. Die Stored Procedure könnte Ihnen also den vom Datenbanksystem erzeugten Autoincrement-PK-Wert per Ausgabeparameter zurückliefern.

Dafür müssten Sie diese dem `SqlDataAdapter` als Kommando für die Anlage neuer Datensätze mitgeben. Auf den `SqlCommandBuilder` könnten Sie verzichten:

```
Dim cmdIns As New SqlCommand("AutoCustomersInsert", conn)
cmdIns.CommandType = CommandType.StoredProcedure
cmdIns.Parameters.Add("@companyname", SqlDbType.VarChar, 50, "companyname")
cmdIns.Parameters.Add("@newCustomerId", SqlDbType.Int, 0, _
                      "customerid").Direction = ParameterDirection.Output
adap.InsertCommand = cmdIns
adap.Update(ds, "AutoCustomers")
```

Für die Synchronisation wäre dann nur noch notwendig, dass `adap` auch den Wert von `@newCustomerId` in den neuen Datensatz im `DataSet` einträgt. Tatsächlich tut das der `SqlDataAdapter` automatisch, weil bei `cmdIns` vermerkt ist, dass er Ausgabeparameter auslesen und ihre Werte in die zugehörigen Spalten (hier: `customerId`) des Datensatzes eintragen soll, der das Kommando ausgelöst hat.[5]

Dieses Verhalten steuern Sie über die `UpdateRowSource`-Eigenschaft eines `Command`-Objektes. Für den hiesigen Zweck könnten Sie diese z. B. wie folgt explizit setzen:

```
cmdIns.UpdatedRowSource = UpdateRowSource.OutputParameters
```

Das ist aber nicht nötig, weil ihr Default-Wert `UpdateRowSource.Both`[6] ist und bedeutet, dass ein `DataAdapter` sowohl Ausgabeparameter als auch den ersten Datensatz einer eventuell im Kommando ausgeführten `select`-Anweisung in das `DataSet` einarbeitet (siehe unten).

[5] Diese Veränderung an einer `DataRow` wird vorgenommen, bevor deren Änderungen intern mit `AcceptChanges()` akzeptiert werden. Sie müssen also nicht fürchten, am Ende mit immer noch als verändert gekennzeichneten Daten dazustehen.

[6] `UpdateRowSource.OutputParameters or UpdateRowSource.FirstReturnedRecord.`

Zugriff auf den zuletzt erzeugten Autoincrement-Wert

Mit der Stored Procedure statt einer automatisch vom `SqlCommandBuilder` erzeugten `insert`-Anweisung ist die Voraussetzung für eine Synchronisation von Datenbank und `DataSet` geschaffen. Der Ausgabeparameter aktualisiert den neuen Datensatz mit Informationen aus der Datenbank.

Aber wie weisen Sie dem Ausgabeparameter der Stored Procedure den Autoincrement-Wert für den Datensatz zu, den sie anlegt? MS SQL Server bietet dafür mehrere Wege:[7]

- `@@IDENTITY`: Liefert den letzten über eine Datenbankverbindung erzeugten Autoincrement-Wert nach einer `insert`-, `select into`- oder Massenkopier-Anweisung (bulk copy). Falls mehrere Datensätze erzeugt und mehrere Autoincrement-Werte generiert wurden, enthält `@@IDENTITY` den letzten. Das gilt auch, falls eine Anweisung dazu führt, dass ein Trigger ausgeführt wird, innerhalb dessen Datensätze mit Autoincrement-Werten erzeugt werden. Wenn Sie `@@IDENTITY` nach einer solchen Anweisung auslesen, finden Sie darin den zuletzt von einer Trigger-Anweisung generierten Autoincrement-Wert.
- `SCOPE_IDENTITY()`: Wie `@@IDENTITY`, allerdings beschränkt sich die Funktion auf Autoincrement-Werte, die innerhalb desselben Bereichs (*Scope*) erzeugt wurden. Bereiche sind: Stored Procedure, Trigger, Funktion und Batch.
- `IDENT_CURRENT(`*tabellenname*`)`: Liefert den zuletzt für eine Tabelle erzeugten Autoincrement-Wert zurück – unabhängig davon, in welchem Bereich oder über welche Datenbankverbindung er erzeugt wurde.

Die Stored Procedure zum Anlegen eines neuen Satzes lässt sich damit wie folgt komplettieren:

```
CREATE PROCEDURE dbo.AutoCustomersInsert
       @companyname varchar(50),
       @newCustomerId int OUT
AS
       insert into AutoCustomers (companyname) values(@companyname)
       set @newCustomerId = SCOPE_IDENTITY()
```

Sie setzen zwar wieder eine `insert`-Anweisung ab – so wie sie auch vom `SqlCommandBuilder` generiert worden wäre –, aber anschließend holen Sie sofort den dabei erzeugten Autoincrement-Wert über `SCOPE_IDENTITY()` ab und weisen ihn dem Ausgabeparameter zu.

Da Sie `SCOPE_IDENTITY()` statt `@@IDENTITY` benutzen, können Sie sicher sein, dass Sie den PK-Wert des vom darüber stehenden `insert` erzeugten Datensatzes bekommen, auch wenn die Anweisung einen Trigger ausgelöst haben sollte, welcher wiederum Autoincrement-Werte generiert hat.

Die Synchronisation zwischen Datenbank und `DataSet` ist damit erreicht.

[7] Nähere Informationen finden Sie in der MS SQL Server-Dokumentation.

Es geht auch ohne Stored Procedure

Die Hilfsmittel des MS SQL Server auf der einen Seite und die Fähigkeiten eines `SqlDataAdapters` auf der anderen Seite, stellen Sie sich an dieser Stelle vielleicht die Frage, ob für die Synchronisation immer eine Stored Procedure nötig ist? Sie bringt zwar bestimmt einen Performancegewinn, kommt aber auch mit erhöhtem Entwicklungs- und Pflegeaufwand daher.[8]

Wenn Sie sich an die möglichen Einstellungen für die `UpdateRowSource`-Eigenschaft eines `Command`-Objektes erinnern, haben Sie die Antwort auf Ihre Frage aber schon in der Hand. Ein `DataAdapter` kann nicht nur Ausgabeparameter einer Stored Procedure in den Datensatz einarbeiten, der ein Kommando ausgelöst hat, sondern auch den Inhalt eines Datensatzes, der zurückgegeben wird.

Sie könnten die Stored Procedure also auch wie folgt formulieren:

```
CREATE PROCEDURE dbo.AutoCustomersInsert
       @companyname varchar(50)
AS
       insert into AutoCustomers (companyname) values(@companyname)
       select customerId = SCOPE_IDENTITY()
```

Als Ergebnis liefert sie jetzt einen Datensatz, als hätten Sie selbst über eine `select`-Anweisung einen `DataReader` geöffnet. Der DataAdapter bemerkt das, weil `UpdateRowSource` per Default auch auf `FirstReturnedRecord` eingestellt ist, durchläuft die Spalten (hier: `customerId`) dieses Datensatzes und kopiert die Werte in die korrespondierenden Spalten des auslösenden Datensatzes im `DataSet`. Beim Zusammenbau des `InsertCommand` für den `SqlDataAdapter` entfällt dadurch natürlich die Definition des Ausgabeparameters.

In der Stored Procedure stehen jetzt zwei SQL-Anweisungen und ihre Fähigkeit zur Rückgabe von Parametern wird nicht genutzt. Damit ist die Stored Procedure überflüssig, denn der MS SQL Server Managed Provider akzeptiert SQL-Batch-Kommandos, d.h. Sie können mit einem `SqlCommand`-Aufruf mehrere SQL-Anweisungen absetzen. Also können Sie auch die beiden obigen Anweisungen in einem Batch zusammenfassen:

```
insert into AutoCustomers ...; select customerId=SCOPE_IDENTITY()
```

Es genügt, beide Anweisungen getrennt durch ein ; einem `SqlCommand` zuzuweisen:

```
Dim cmdIns As New SqlCommand( _
    "insert into AutoCustomers (companyname) values(@companyname);" & _
    "select customerId=SCOPE_IDENTITY()", conn)
cmdIns.Parameters.Add("@companyname", SqlDbType.VarChar, 50, "companyname")
adap.InsertCommand = cmdIns
adap.Update(ds, "AutoCustomers")
```

[8] Anwendungscode ist jetzt an zwei Stellen zu pflegen: in der Geschäftslogikkomponente einer Anwendung und in der Datenbank. Gerade bei einem potenziellen Wechsel des Datenbanksystems mag das ungünstig sein. Sie sollten vor dem Einsatz von Stored Procedures daher immer Performancegewinne gegen Codetransparenz und Pflegefreundlichkeit/Portabilität abwägen.

Das ist der unaufwändigste Weg zur Synchronisation von Datenbank und `DataSet`, wenn Sie Autoincrement-PKs verwenden müssen.

Synchronisation mit Veränderungen durch Trigger

Durch die Möglichkeit, den Inhalt eines am Ende eines Kommandos selektierten Datensatzes automatisch in ein `DataSet` einfließen zu lassen, haben Sie natürlich die Chance, es auch mit anderen automatischen datenbankinternen Veränderungen zu synchronisieren. Sie sind dabei nicht auf Datenbanksystemvariablen/-funktionen angewiesen. Die einzige Einschränkung besteht darin, dass Sie lediglich die aktuell gespeicherte `DataRow` anpassen lassen können.

Falls bei einer Änderung an einem Datensatz oder einer Neuanlage also z. B. Spalten automatisch durch einen Trigger gesetzt werden, können Sie auch den Datensatz erneut selektieren. Denken Sie z. B. an die automatische Vergabe einer Kundennummer durch Geschäftslogik in einem Trigger. Nehmen Sie als Beispiel die folgende Tabelle:

```
Kunden(kdId, kdNr, name, str, plz, ort)
```

Bei Neuanlage eines Kunden könnte die Spalte `kdNr` z. B. aufgrund einer internen Tabelle und der PLZ zusammengesetzt werden. Das Kommando zum Einfügen neuer Kunden würde dann so aussehen:

```
insert into Kunden (name, str, ...) values(@name, @str, ...);
select kdId, kdNr where kdId=SCOPE_IDENTIY()
```

`KdId` und `kdNr` fehlen beim Einfügen und werden anschließend durch Auswahl des neuen Satzes mit einer `select`-Anweisung ermittelt und an den `SqlDataAdapter` zurückgeliefert.

Optimistisches Sperren mit Zeitstempeln

Ein sehr typischer Fall einer Veränderung nicht nur bei Neuanlage, sondern auch bei Modifikation von Daten ist eine Zeitstempelspalte (*Timestamp*). Ein Timestamp wird vom Datenbanksystem bei jeder Veränderung eines Datensatzes neu gesetzt. Durch den Vergleich des Zeitstempels im Datensatz mit dem in einer `DataRow` beim Speichern können Sie daher feststellen, ob sich der Datensatz verändert hat, seitdem er in die `DataRow` geladen wurde. Wenn Sie eine solche Prüfung beim Speichern durchführen, wird das *optimistisches Sperren* (Optimistic Locking) genannt.[9]

Die ADO.NET `CommandBuilder` arbeiten auch nach diesem Verfahren – allerdings prüfen sie auf Veränderungen im Datensatz, indem Sie die Inhalte der Spalten mit denen in der `DataRow` vergleichen, die dort verändert wurden. Der Unterschied wird an einem Beispiel am besten deutlich. Nehmen Sie folgende Tabelle:

```
Kunden(kdId, name, str, plz, ort)
```

[9] Es wird dabei zwar der Datensatz nicht wirklich gesperrt, aber Sie vermeiden dennoch das Überschreiben von Änderungen anderer Benutzer, weil Ihre Daten nur gespeichert werden, wenn der Datensatz seit dem Laden unverändert ist.

Wenn Sie nun z. B. `str` und `plz` in einem `Kunden`-Datensatz in einem `DataSet` verändert haben, dann erzeugt ein `CommandBuilder` ein solches `update`-Kommando:

```
update Kunden set str=@str, plz=@plz where kdId=@kdId and
                                           str=@str_orig and plz=@plz_orig
```

`@str` und `@plz` stehen für die neuen Werte der Spalten, `@str_orig` und `@plz_orig` für die Werte, die ursprünglich aus der Datenbank geladen wurden. Nur wenn die Spalten in der Datenbank unverändert sind, überschreibt das Kommando sie mit den neuen Werten, da es sonst den Datensatz gar nicht lokalisieren kann.

Wenn Sie in der Tabelle jedoch eine Zeitstempelspalte definiert haben[10]

```
Kunden(kdId, name, str, plz, ort, zeitstempel:timestamp)
```

kann das `update`-Kommando einfacher ausfallen:

```
update Kunden set str=@str, plz=@plz where kdId=@kdId and
                                           zeitstempel=@zeitstempel
```

Statt je nach Umfang der Veränderungen eine variable Anzahl von Spalten mit ihren Originalwerten zu vergleichen, genügt der Vergleich des Zeitstempels.[11]

Da sich der Zeitstempel durch das `update`-Kommando jedoch in der Datenbank ändert, muss die zugehörige `DataRow` anschließend mit dem Datensatz in der Datenbank synchronisiert werden! Das gilt auch für die Neuanlage von Sätzen.

Wenn Sie optimistisches Sperren mit Zeitstempeln selbst implementieren wollen, können Sie nicht mehr mit der `SqlCommandBuilder`-Klasse arbeiten, sondern müssen alle Kommandos selbst definieren:

```
Dim cmdIns As New SqlCommand( _
     "insert into Kunden (name, str, ...) values(@name, @str, ...);" & _
     "select kdId=SCOPE_IDENTITY(), zeitstempel=@@DBTS", conn)
cmdIns.Parameters.Add("@name", SqlDbType.VarChar, 50, "name")
...
adap.InsertCommand = cmdIns

Dim cmdUpd As New SqlCommand( _
```

[10] Spaltentyp `timestamp` bei MS SQL Server.

[11] Je nachdem wie Sie es betrachten, kann das ein Vor- oder Nachteil sein. Beim optimistischen Sperren mit Zeitstempel wird jede weitere Veränderung an einem Datensatz abgelehnt, sobald er aufgrund einer Modifikation einen neuen Zeitstempel erhalten hat und daher von der `update`-Anweisung nicht gefunden wird. Beim optimistischen Sperren mit Spaltenvergleich sind dagegen gleichzeitige Modifikationen durch mehrere Benutzer möglich, solange sich die Mengen der veränderten Spalten nicht schneiden. Dieses Verfahren lässt mehr Raum für Gleichzeitigkeit, kann aber zu unerwarteten Inkonsistenzen zwischen Cache bzw. Datenanzeige und Datenbank führen: Ein Anwender könnte z. B. einen Spalteninhalt in einem Formular nach Abspeicherung von Änderungen sehen, der tatsächlich nicht mehr dem Spaltenwert im Datensatz in der Datenbank entspricht. Die Speicherung wäre ja fehlerfrei erfolgt, solange dieser Spalteninhalt nicht verändert wurde.

```
     "update Kunden set name=@name, ... where kdId=@kdId and " & _
     "zeitstempel=@zeitstempel; select zeitstempel=@@DBTS", conn)
cmdUpd.Parameters.Add("@name", SqlDbType.VarChar, 50, "name")
...
cmdUpd.Parameters.Add("@zeitstempel", SqlDbType.Timestamp, 0, "zeitstempel")
adap.UpdateCommand = cmdUpd

Dim cmdDel As New SqlCommand( _
     "delete Kunden where kdId=@kdId and zeitstempel=@zeitstempel", conn)
cmdDel.Parameters.Add("@kdId", SqlDbType.Int, 0, "kdId")
cmdDel.Parameters.Add("@zeitstempel", SqlDbType.Timestamp, 0, "zeitstempel")
adap.DeleteCommand = cmdDel
```

`@@DBTS` enthält den zuletzt in einer Datenbank vergebenen Zeitstempel, den Sie bei neuen und veränderten Datensätzen zur Synchronisation zurück in die gerade gespeicherte `DataRow` schreiben müssen. Wie `@@IDENTITY` gilt er jedoch sehr global und es kann durch Trigger während der Ausführung der SQL-Anweisung passieren, dass er sich nicht mehr auf den gerade veränderten Datensatz bezieht. Sie können daher alternativ den aktuellen Zustand des gespeicherten Satzes auch erneut selektieren, z.B.

```
update Kunden set ... where kdId=@kdId and zeitstempel=@zeitstempel;
select zeitstempel where kdId=@kdId
```

In jedem Fall jedoch müssen Sie den Zeitstempel zur Auswahl eines Datensatzes dem PK beistellen. Das ist für die Modifikation und das Löschen notwendig.

3.3.4 Autoincrement-PK synchronisieren mit MS Access 2000

Um einen Autoincrement-PK im `DataSet` mit dem Wert in der Datenbank zu synchronisieren, sind zwei Voraussetzungen zu erfüllen: Zum einen muss das Datenbanksystem Zugriff auf den relevanten Autoincrement-Wert geben, zum anderen muss ein Rücktransport über Ausgabeparameter oder Resultatsdatensatz möglich sein.

Die erste Voraussetzung erfüllt MS Access seit der Version 2000.[12] MS Access bietet wie MS SQL Server die Systemvariable `@@IDENTITY`.[13] Sie können also nach einem

```
insert AutoCustomers (companyname) values(@companyname)
```

eine Abfrage absetzen, um den erzeugten Autoincrement-PK-Wert zu ermitteln:

```
select @@IDENTITY
```

Leider kennt MS Access jedoch weder Stored Procedures mit Ausgabeparametern, noch akzeptiert der OLE DB Provider, der über den OLE DB Managed Provider angesprochen wird, Stapelkommandos. Sie können also *nicht* beide Anweisungen *en bloc* absetzen:

[12] Es reicht nicht aus, den Jet 4.0 OLE DB Provider z.B. auf eine MS Access 98-Datenbank anzusetzen.

[13] Andere Datenbanksysteme bieten ähnliche Funktionalitäten, z.B. `LAST_INSERT_ID()` (MySQL) oder `SELECT SequenceName.CURRVAL FROM DUAL` (Oracle).

```
insert AutoCustomers (companyname) values(@companyname);
select customerId = @@IDENTITY
```

Sie müssen sie vielmehr nacheinander getrennt ausführen.

Die `Update()`-Funktion eines `OleDbDataAdapter` speichert allerdings potenziell sehr viele Datensätze mit einem Aufruf. Es stellt sich daher die Frage, wie Sie an den Autoincrement-PK jedes einzelnen neuen Datensatzes kommen, sofort nachdem er gespeichert wurde.

Die Antwort bietet der `RowUpdated`-Event des `OleDbDataAdapter`. Er wird für jede veränderte `DataRow` gefeuert, nachdem ein Kommando zu ihrer Speicherung an die Datenbank abgesetzt wurde.

```
Dim ds As New DataSet()
Dim conn As New OleDbConnection("...")
Dim adap As New OleDbDataAdapter("select * from AutoCustomers", conn)
adap.MissingSchemaAction = MissingSchemaAction.AddWithKey
adap.Fill(ds, "customers")
...
Dim cb As New OleDbCommandBuilder(adap)
AddHandler adap.RowUpdated, _
           New OleDbRowUpdatedEventHandler(AddressOf OnCustRowUpdated)
adap.Update(ds, "customers")
```

Sie laden Ihre Daten wie gewohnt und können die Aktualisierungskommandos für einen `OleDbDataAdapter` von einem `OleDbCommandBuilder` erzeugen lassen. Sie müssen nur vor Aufruf von `Update()` eine Event-Handler-Routine für `RowUpdated` registriert haben.

Der `OleDbDataAdapter` ruft den Event-Handler für jeden angefügten, modifizierten und gelöschten Datensatz und übergibt ihn (`args.Row`) zusammen mit Statusinformationen (`args.Status`) und dem benutzten Aktualisierungskommando (`args.Command`):

```
Sub OnCustRowUpdated(ByVal sender As Object, _
                     ByVal args As OleDbRowUpdatedEventArgs)
    If args.Status = UpdateStatus.Continue AndAlso _
       args.StatementType = StatementType.Insert Then

        Dim cmdId As New OleDbCommand("SELECT @@IDENTITY", args.Command.Connection)

        args.Row("customerId") = CInt(cmdId.ExecuteScalar())
    End If
End Sub
```

Im Event-Handler prüfen Sie dann zunächst, ob ein Fehler beim Speichern aufgetreten ist oder ob es weitergehen kann (`UpdateStatus.Continue`). Nur wenn alles in Ordnung ist und ein neuer Datensatz angefügt wurde (`StatementType.Insert`), ermitteln Sie den Autoincrement-PK über `@@IDENTITY`. Sie können dafür die Datenbankverbindung des gerade gelaufenen Aktualisierungskommandos benutzen.

Die Synchronisation mit MS Access 2000-Datenbanken ist durch den Zwang zur Definition eines Event-Handlers also etwas umständlicher als mit MS SQL Server-Datenbanken. Vor allem aber ist sie imperformanter, weil Aktualisierungskommando und Rückgabedaten nicht mit demselben Datenbankaufruf transportiert werden können. Es sind in jedem Fall zwei *Roundtrips* zur Datenbank erforderlich.

3.4 Autoincrement-PKs in verteilten Anwendungen speichern

Autoincrement-PKs erzeugen ein Synchronisationsproblem. Sie schicken ein Änderungskommando an die Datenbank und müssen Daten von dort in einen Cache integrieren. Beide Aufgaben erledigt mit ein wenig Hilfestellung ein `DataAdapter` für Sie. Das funktioniert, weil er sich dort befindet, wo der Cache ist.

Wie ist es aber, wenn sich Cache, d.h. `DataSet`, und `DataAdapter` an verschiedenen Orten befinden? Das ist nämlich in einer verteilten Anwendung oft der Fall.[14] Schauen Sie sich die folgende Abbildung an:

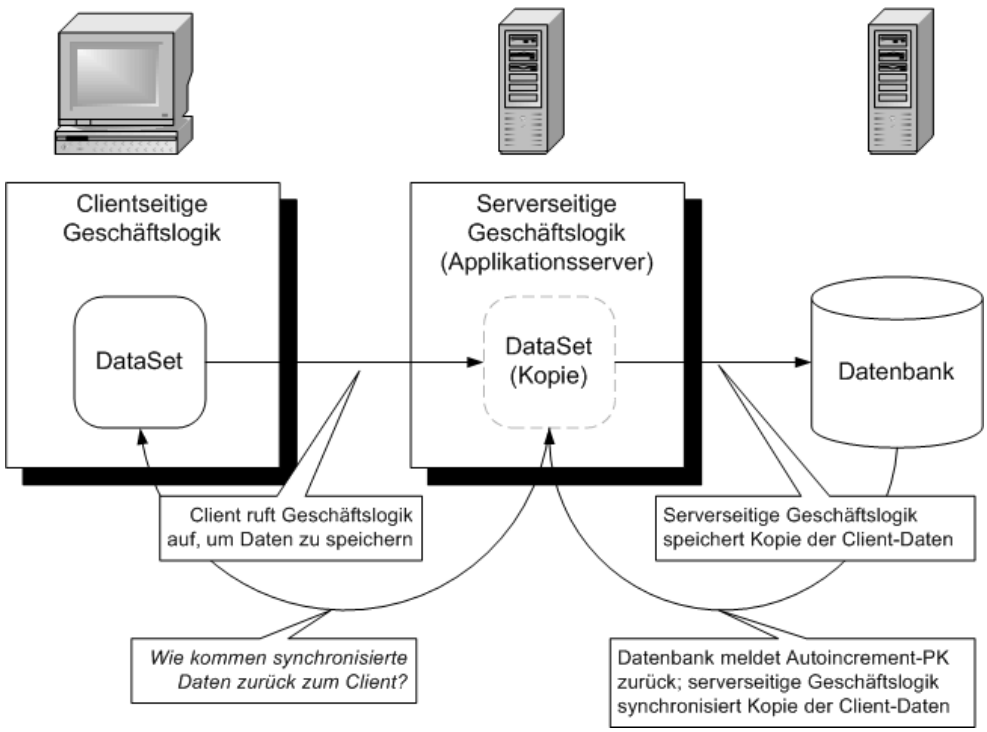

Abbildung 3.2: Datentransport zur Speicherung in einer verteilten Anwendung

[14] Die Betrachtungen hier behandeln nur den schwierigeren Fall einer verteilten Anwendung, bei der Änderungen an einem `DataSet` potenziell häufiger gespeichert werden müssen. Dieser Fall tritt natürlich vor allem bei Windows Forms-basierenden Rich-Clients auf, ist aber auch in anderen Szenarien möglich.

Der Client hat sich von der serverseitigen Geschäftslogik ein `DataSet` beschafft und fügt darin neue Datensätze mit Autoincrement-PKs an. Hier eine Funktion in einer serverseitigen Geschäftslogikkomponente (*Business Logic API*, BAPI) als Beispiel:

```
Public Class BAPI
    Inherits MarshalByRefObject

    Function Load() As DataSet
        Dim ds As New DataSet()
        Dim conn As New SqlConnection("...")
        Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
        adap.MissingSchemaAction = MissingSchemaAction.AddWithKey
        adap.Fill(ds, "customers")
        Return ds
    End Function

    Sub Store(ds As DataSet)
        ...
    End Sub
End Class
```

Diese Klasse können Sie mit den Mitteln von .NET Remoting sehr einfach in einem Prozess (*Host*) so veröffentlichen, dass sie von Client-Anwendungen auf anderen Rechnern instanziert und aufgerufen werden kann:

```
ChannelServices.RegisterChannel(New Tcp.TcpServerChannel(8000))
RemotingConfiguration.ApplicationName = "AutoNumberHost"
RemotingConfiguration.RegisterWellKnownServiceType( _
                                        GetType(BAPI), _
                                        "BAPI", _
                                        WellKnownObjectMode.SingleCall)
```

Für einen Client ist es dann nahezu transparent, ob die Instanz, die er von der Klasse `BAPI` erzeugt, lokal in seinem Prozess läuft oder in einem Host-Prozess auf einem anderen Rechner im Netzwerk. Für die Methodenaufrufe macht es formal keinen Unterschied:

```
ChannelServices.RegisterChannel(New Tcp.TcpClientChannel())
Dim s As BAPI = Activator.GetObject(_
                            GetType(BAPI), _
                            "tcp://localhost:8000/autonumberhost/bapi")
Dim ds As DataSet = s.Load()
```

Mit einem Aufruf von `Load()` verschafft sich der Client ein lokales `DataSet`. Wie es gebildet wird, verbirgt die Geschäftslogik-Funktion. Natürlich wartet der Client schon darauf, denn er will ja neue Datensätze erzeugen:

```
Dim rCust As DataRow
With ds.Tables("customers")
    rCust = .NewRow
```

```
    rCust("companyname") = ...
    .Rows.Add(rCust)
End With
```

Und diese neuen Datensätze später an die Geschäftslogik zum Speichern übergeben:

```
s.Store(ds)
```

Wie das Speichern genau vor sich geht, ist für den Client dabei wieder unwesentlich. Im einfachsten Fall sähe die Speicherlogik wie folgt aus:

```
Public Class BAPI
    Inherits MarshalByRefObject
    ...
    Sub Store(ByVal ds As DataSet)
        Dim conn As New SqlConnection("...")
        Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
        Dim cb as New SqlCommandBuilder(adap)

        Dim cmdIns As New SqlCommand( _
               "insert into AutoCustomers (companyname) " & _
                                        "values(@companyname);" & _
               "select customerId=SCOPE_IDENTITY()", conn)
        cmdIns.Parameters.Add("@companyname", SqlDbType.VarChar, 50, "companyname")
        adap.InsertCommand = cmdIns

        adap.Update(ds, "customers")
    End Sub
End Class
```

Sie entspräche also dem oben entwickelten Muster für Code zur Neuanlage von Datensätzen mit Autoincrement-PKs. Selbstverständlich wäre `ds` am Ende von `Store()` also korrekt mit der Datenbank synchronisiert.

Marshaling von DataSet-Objekten

Das Problem für den Client ist jedoch, dass `ds` innerhalb von `Store()` *nicht* das `DataSet ds` ist, mit dem er selbst arbeitet. Es handelt sich vielmehr um eine Kopie! Diese Kopie wurde vom .NET Remoting Framework automatisch beim Aufruf von `s.Store(ds)` erzeugt.

Alle Daten, die an eine entfernte Methode[15] übergeben werden oder von dort zurückkehren, müssen *gemarshalt* werden.[16] Für ein `DataSet` bedeutet das die Serialisierung auf dem Weg zum Geschäftslogikserver bei Aufruf von `Store()` und die dortige Deserialisierung.[17] Serialisierung und Deserialisierung zusammen erzeugen immer eine Kopie des Ausgangsobjektes.

Was hinter den Kulissen des Aufrufs der entfernten Geschäftslogik in etwa passiert, zeigt die folgende »Simulation«:

```
Dim f As New SoapFormatter()
Dim sDataSet As New MemoryStream()
f.Serialize(sDataSet, ds)
sDataSet.Seek(0, SeekOrigin.Begin)
s.Store(sDataSet)
```

Der Client serialisiert sein `DataSet ds` in einen `Stream` und übergibt diesen an die `Store()`-Routine. Diese deserialisiert ihn sofort nach Erhalt in ein eigenes, lokales `DataSet` und fertigt somit eine Kopie des Ausgangsobjektes an:

```
Sub Store(ByVal sDataSet As Stream)
    Dim f As New SoapFormatter()
    Dim ds As DataSet = f.Deserialize(sDataSet)
    ...
End Sub
```

Für client- und serverseitige Geschäftslogik ist all dies transparent – aber deshalb nicht minder relevant. Als erste Konsequenz ergibt sich, dass zwischen Client und Server nur möglichst wenige Daten ausgetauscht werden sollten, weil Marshaling und Übertragung der Daten über einen .NET Remoting-Kanal auf womöglich einen anderen Rechner sehr aufwändig sind: Wenn `Load()` z. B. 100 Datensätze zum Client zurückliefert, der Client in dieser Menge einen Satz löscht, einen ändert und einen hinzufügt, dann sollten beim Aufruf von `Store()` nicht wieder 100 Datensätze an den Server übertragen werden, sondern nur die drei veränderten:

```
s.Store(ds.GetChanges())
```

Durch den Aufruf von `GetChanges()` auf `ds` erreichen Sie genau das. Die Funktion erzeugt ein neues `DataSet`, welches nur geänderte Datensätze aus dem Original enthält. Schon `GetChanges()` erzeugt also eine Kopie der Daten – die dann wiederum als Kopie zum Server übertragen werden.

---

[15] »Entfernt« (remote) bedeutet hier: Das Objekt, dessen Methode aufgerufen wird, liegt in einer anderen *Application Domain* (AppDomain) als der des Aufrufers. Ob diese AppDomain im selben oder einem anderen Prozess auf demselben oder einem anderen Rechner existiert, ist unerheblich.

[16] Mehr Informationen zum Thema .NET Remoting finden Sie in [Rammer02], [Westphal02a] oder [Willers02].

[17] Beim Aufruf von `Load()` läuft das Marshaling umkehrt ab: Das im Business-API erzeugte `DataSet` wird für die Rückgabe auf dem Server serialisiert und bei Ankunft im Client deserialisiert.

Eine weitere Konsequenz des `DataSet`-Marshaling als Kopie ist, dass die im serverseitigen `DataSet` synchronisierten Daten explizit zurück zum Client transportiert werden müssen. `Store()` muss also sein `DataSet` zurückliefern:

```
Function Store(ByVal ds as DataSet) as DataSet
    ...
    adap.Update(ds, "customers")
    return ds
End Function
```

Synchronisierte Daten einmischen

Wenn `Store()` sein `ds` als Funktionsresultat herausreicht, hat der Client prinzipiell die korrekt synchronisierten Daten in der Hand – allerdings in einem separaten `DataSet`. Wie kommen die von der Datenbank erzeugten Autoincrement-PK-Werte nun aber von diesem `DataSet` in das ursprüngliche `ds` im Client?

ADO.NET hat solche Anforderungen vorhergesehen und bietet daher bei `DataSet`-Objekten die Methode `Merge()` an. Mit ihr lassen sich die Tabellen und Datensätze zweier `DataSet`-Objekte zusammenführen. Das gilt sowohl für ihre Schemata als auch für die Datensätze. Nicht vorhandene Tabellen und Spalten werden im Ziel-`DataSet` (auf welchem Sie `Merge()` aufrufen) erzeugt, veränderte Datensätze der Quelle werden entsprechend ihrem Änderungsstatus in die Ziel-Tabellen übernommen. Um Datensätze im Ziel zu überschreiben, muss natürlich ein PK definiert sein, damit für einen Quelldatensatz der anzupassende Zieldatensatz lokalisierbar ist.[18]

Für den Client ergibt sich damit folgendes Vorgehen beim Speichern von Änderungen:

```
Dim dsSynchronized As DataSet
dsSynchronized = s.Store(ds.GetChanges())
ds.Merge(dsSynchronized, False)
ds.AcceptChanges()
```

`Merge()` mischt die synchronisierten Daten vom Server ein, indem es die in `ds` noch existierenden Änderungen überschreibt (`False`). Anschließend muss der Client die Änderungen am `DataSet` explizit akzeptieren, da es ja nicht direkt an einen `DataAdapter` zur Speicherung übergeben wurde, sondern nur die zweifache Kopie seiner geänderten Datensätze.[19] Würden die Änderungen nicht akzeptiert, würden sie auch beim nächsten Aufruf von `Store()` wieder mit zur Speicherung übergeben und resultierten in Fehlern (z. B. weil ein schon gelöschter Datensatz nochmals gelöscht werden soll).

[18] Mehr Informationen zu `Merge()` finden Sie in der Online-Hilfe des .NET Framework SDK und z. B. [Beauchemin02].

[19] `DataAdapter` akzeptieren die Änderungen erfolgreich gespeicherter `DataRow`-Objekte automatisch in der `Update()`-Methode.

Änderungen als Änderungen erhalten

`Merge()` macht die Synchronisation mit vom Server zurückgelieferten Daten sehr einfach. Voraussetzung dafür ist jedoch, dass die einzumischenden Daten noch als Änderungen gekennzeichnet sind. Ein auf dem Client hinzugefügter Datensatz, der an die serverseitige Geschäftslogik übertragen, dort gespeichert und mit der Datenbank synchronisiert wurde, muss nach seiner Rückkehr immer noch als hinzugefügt erkennbar sein. Ist er das, überschreibt `Merge()` sein Original mit ihm und erreicht damit wiederum eine Synchronisation. Ist er es jedoch nicht, fügt `Merge()` ihn an, denn sein neuer PK-Wert aus der Datenbank entspricht ja nicht dem temporären, im clientseitigen `DataSet` noch vergebenen Autoincrement-Wert.[20]

Unglücklicherweise enthält nun das von `Store()` zurückgelieferte `DataSet` ohne weiteren Eingriff natürlich nur Datensätze, deren Änderungen von `Update()` akzeptiert wurden. Damit `Merge()` erfolgreich sein kann, müssen Sie in `Store()` verhindern, dass für den hiesigen Zweck zumindest die erfolgreich gespeicherten Änderungen an neuen Datensätzen *nicht* akzeptiert werden.

Dafür bieten `DataAdapter` jedoch keinen direkten Ansatzpunkt. Sie können nicht einfach über eine Eigenschaft wählen, ob und welche Änderungen während `Update()` akzeptiert werden sollen. Die Lösung bringt nur der Umweg über das schon vorgestellte `RowUpdate`-Ereignis:

```
Function Store(ByVal ds As DataSet) As DataSet
    Dim conn As New SqlConnection("...")

    Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
    AddHandler adap.RowUpdated, AddressOf OnRowUpdated

    Dim cb as New SqlCommandBuilder(adap)

    Dim cmdIns As New SqlCommand("...", conn)
    ...
    adap.InsertCommand = cmdIns

    adap.Update(ds, "customers")

    Return ds.GetChanges()
End Function
```

[20] Nur solange eine `DataRow` als verändert gekennzeichnet ist und im Autoincrement-PK der Originalwert vom aktuellen, synchronisierten abweicht, wird der Originalwert benutzt, um den korrespondierenden Datensatz im Ziel-`DataSet` zu finden.
Dieses Verhalten wird im Knowledgebase-Artikel *»PRB: Merge Method Creates Duplicate Records for AutoIncrement Fields (Q313540)«* als Problem formuliert, scheint aber doch eher ein Feature denn ein Bug zu sein. Das Verhalten von `Merge()` beim Vorhandensein zweier Werte für eine Spalte ist plausibel. Problematisch ist eher der umständliche Weg für den Erhalt beider Informationen.

```
Private Sub OnRowUpdated(ByVal sender As Object, _
                         ByVal e As SqlRowUpdatedEventArgs)
    If e.StatementType = StatementType.Insert Then _
            e.Status = UpdateStatus.SkipCurrentRow
End Sub
```

Wurde ein hinzugefügter Satz gespeichert, setzen Sie den Status für die Weiterverarbeitung auf `SkipCurrentRow` und weisen den `SqlDataAdapter` damit an, auf der zugrunde liegenden `DataRow` *nicht* `AcceptChanges()` aufzurufen. Auch nach erfolgreicher Synchronisation bleiben so die Änderungen als Änderungen gekennzeichnet und werden später als solche wieder zum Client zurückgeschickt.

3.5 Beziehungen auf der Basis von Autoincrement-PKs speichern

Der Umgang mit Beziehungen, die auf Autoincrement-PKs beruhen, ist – wie am Anfang des Kapitels gezeigt – solange einfach, wie sich die Parent- und Child-Datensätze nur im `DataSet` befinden. Am Schluss des Kapitels stellt sich nun die Frage, inwiefern sich das ändert, sobald die Datensätze auch gespeichert werden?

Als herausforderndes Beispiel mag gleich ein verteiltes Szenario dienen. Hier die erweiterte Version der `Load()`-Methode des obigen Geschäftslogik-API:

```
Function Load() As DataSet
    Dim ds As New DataSet()

    Dim conn As New SqlConnection("...")

    Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
    adap.MissingSchemaAction = MissingSchemaAction.AddWithKey
    adap.Fill(ds, "customers")

    adap.SelectCommand.CommandText = "select * from AutoOrders"
    adap.Fill(ds, "orders")

    ds.Relations.Add("custord", _
                     ds.Tables("customers").Columns("customerid"), _
                     ds.Tables("orders").Columns("customerid"), _
                     True)

    Return ds
End Function
```

Sie liefert jetzt sowohl Parent- wie Child-Datensätze in einem `DataSet` und verbunden über eine Relation zurück.[21]

Die Frage ist nun: Wenn der Client einen neuen Kunden hinzufügt, diesem sofort eine neue Bestellung zuordnet und beide zum Speichern an `Store()` schickt, wie gelangt dann der datenbankseitig vergebene Autoincrement-PK für den `AutoCustomers`-Satz in die Fremdschlüsselspalte (!) des `AutoOrders`-Satzes? Die Antwort: automatisch.

Der Eigenschaft `UpdateRule` des durch die Beziehungsdefinition erzeugten `ForeignKeyConstraint` auf `ds.Tables("orders")` hat den Default-Wert `Rule.Cascade`. Änderungen am Primärschlüssel, der zum Fremdschlüssel gehört, werden so automatisch an diesen weitergegeben (*Cascading Update*). Ein `DataSet` bietet also dieselben Funktionen zur Durchsetzung referentieller Integrität, wie Sie diese von Datenbanksystemen wie MS Access und MS SQL Server kennen.

```
Function Store(ByVal ds As DataSet) As DataSet
    Dim conn As New SqlConnection("...")

    Dim adap As New SqlDataAdapter("select * from AutoCustomers", conn)
    ...
    adap.Update(ds, "customers")

    cmdIns = New SqlCommand("insert into AutoOrders " & _
                            "(customerid, orderdate) " & _
                            "values(@customerid, @orderdate);" & _
                            "select orderid=SCOPE_IDENTITY()", conn)
    cmdIns.Parameters.Add("@customerid", SqlDbType.Int, 0, "customerid")
    cmdIns.Parameters.Add("@orderdate", SqlDbType.DateTime, 0, "orderdate")
    adap.InsertCommand = cmdIns

    adap.Update(ds, "orders")

    Return ds.GetChanges
End Function
```

[21] Beide Tabellen entsprechen dem am Kapitelanfang vorgestellten Schema.

Wenn dann die neuen Datensätze der Parent-Tabelle vor denen der Child-Tabelle gespeichert und damit auch synchronisiert werden, stehen die endgültigen Fremdschlüssel rechtzeitig zur Speicherung in den Child-Datensätzen. Sie müssen also keine Angst haben, dass Child-Datensätze mit temporären Fremdschlüsseln persistiert würden und insofern die referentielle Integrität ihrer Datenbank gefährdeten.

Es ist kein zusätzlicher Aufwand nötig, um Beziehungen auf der Basis von Autoincrement-PKs zu speichern. Wenn Sie für die Synchronisation der einzelnen Tabellen sorgen, übernimmt ADO.NET den Rest.[22]

[22] Der Artikel »*HOW TO: Update Parent-Child Data with an Identity Column from a Windows Forms Application by Using a Web Service (Q310350)*« in der Microsoft Knowledgebase weist korrekt darauf hin, dass der synchronisierte Fremdschlüsselwert nicht als verändert gekennzeichnet ist. Die `DataRowVersion.Original`- und `DataRowVersion.Current`-Werte der Spalte unterscheiden sich nicht. Das ist auf der einen Seite merkwürdig, da ja der vorherige temporäre Fremdschlüsselwert durch die kaskadierende Aktualisierung seines Parent-PK verändert wurde und beide Werte sich bei seinem synchronisierten Autoincrement-PK unterscheiden. Auf der anderen Seite ist es aber auch wiederum nicht merkwürdig, da es sich um einen neuen Datensatz handelt, der eigentlich keine Originalwerte kennt, und insofern aktueller und ursprünglicher Wert auch als gleich angesehen werden können.
Unabhängig davon jedoch, ob die Übereinstimmung beider Werte inkorrekt oder plausibel ist, treibt der Code im Knowledgebase-Artikel einigen Aufwand, um dem Fremdschlüssel seinen temporären Wert als Original unterzuschieben und den synchronisierten Wert als aktuellen einzutragen. Der hier gezeigte Code unterlässt diesen Aufwand – ohne Abstriche beim Erfolg. Der Fremdschlüssel wird sowohl korrekt gespeichert als auch bei der Synchronisation des von `Store()` zurückgelieferten `DataSet` korrekt in das `DataSet` des Clients übertragen.
Es sind auch keine Nachteile in einem nichtverteilten Szenario zu erwarten, falls der Child-Datensatz später einfach nach weiteren Änderungen erneut gespeichert werden sollte. Ein `SqlCommandBuilder` mag dann zwar das Fremdschlüsselfeld nicht als verändert erkennen und insofern auch nicht speichern wollen – allein das ist unerheblich, weil der Fremdschlüssel bereits korrekt in der Datenbank steht und zukünftig ohne Neuzuweisung unveränderlich ist.