HAUFE.
Ausgabe 06/2019

ACCESS IM UNTERNEHMEN

ARCHIV IM SQL SERVER

Statten Sie Tabellen im SQL Server so aus, dass alle Änderungen archiviert werden (ab S. 5)

In diesem Heft:

VON ACCESS ZU WORDPRESS

Übertragen Sie Artikel direkt aus der Datenbank in eine Wordpress-Webseite.

SEITE 43

VERWEISE IM GRIFF MIT VBA

Verwalten Sie die Verweise eines VBA-Projekts mit VBA.

SEITE 23

FLEXIBLE ADRESSEN

Fügen Sie beliebig viele Adressen zu Kunden hinzu und legen Sie diese als Versand- oder Rechnungsadresse fest.

SEITE 30

Gut archiviert ist halb gewonnen

Als ich neulich meine Bestelldatenbank von einer reinen Access-Anwendung in ein Access-Frontend mit SQL Server-Backend umgewandelt habe, lief zunächst alles reibungslos. Nach kurzer Zeit stellte ich dann fest, dass die gewohnten Backups von Datensätzen bestimmter Tabellen vor dem Ändern oder Löschen nicht mehr angelegt wurden. Was ist da passiert?

Die Antwort war schnell gefunden: Unter Access gibt es die sogenannten Datenmakros, mit denen man zum Beispiel festlegen kann, dass ein Datensatz vor einer Änderung oder vor dem Löschen noch in eine Archivtabelle geschrieben wird. So kann man nach versehentlichen Änderungen oder Löschvorgängen immer noch auf die vorherige Version der Datensätze zugreifen.

Beim Migrieren auf den SQL Server mit dem dazu vorgesehenen Tool, dem SQL Server Migration Assistant, wurden die Datenmakros allerdings nicht berücksichtigt und folgerichtig nur die Tabellen inklusive Daten auf den SQL Server übertragen. So musste ich also nochmal Hand anlegen und die gewünschte Funktion diesmal in Form von Triggern programmieren. Trigger werden auch bei Änderungen von Datensätzen ausgelöst und sind so das passende Gegenstück zu den Datenmakros von reinen Access-Tabellen.

Dummerweise hatte ich eine Reihe von Tabellen mit diesen Datenmakros ausgestattet und ich wollte das nicht nochmal von Hand erledigen. Also habe ich mir ein paar Routinen geschrieben, bei denen ich nur noch in einem Formular die Tabellen der SQL Server-Datenbank auswählen musste, die ich mit Triggern zum Archivieren versehen wollte. Danach reichte ein Mausklick, um sowohl die Archivtabelle anzulegen, welche alle notwendigen Felder enthält, als auch die Tabelle mit den Produktivdaten mit dem benötigten Trigger auszustatten. Diese Lösung möchte ich Ihnen natürlich nicht vorenthalten: Sie finden die Grundlagen dazu im Beitrag **Datenhistorie per Trigger** (ab Seite 5) und die passende Lösung unter dem Titel **Datenhistorie-Trigger schnell anlegen** (ab Seite 11).

Außerdem lernen Sie in dieser Ausgabe die Unterschiede zwischen der **Like**- und der **Alike**-Funktion kennen, mit denen Sie in verschiedenen Versionen der SQL Server-Syntax arbeiten können und die je nach Version unterschiedlich wirken (siehe **Like, Alike, *, % und die Kompatibilität** ab Seite 2).

Im Beitrag **Verweise per VBA verwalten** zeigen wir Ihnen, wie Sie die Verweise auf Bibliotheken in einem VBA-Projekt mit VBA einsehen und verwalten können – siehe ab Seite 23.

Das Verwalten von Kunden und Adressen ist immer wieder eine Herausforderung. Im Beitrag **Flexible Adressen** gehen wir ab Seite 30 einmal weg von der Kundentabelle, welche die Eingabe genau einer Liefer- und einer Rechnungsadresse erlaubt. Stattdessen können Sie mit der Lösung aus diesem Beitrag beliebig viele Adressen je Kunde verwalten – von denen Sie jeweils eine als Lieferadresse und eine als Rechnungsadresse auswählen können.

Außerdem zeigen wir, wie Sie Artikel aus Access in das Blogsystem Wordpress übertragen (**Von Access nach Wordpress**, ab Seite 46), wie Sie sich Backends nur für die Zeit der Bearbeitung auf den Rechner holen (**Backend nur zum Bearbeiten holen**, ab Seite 61) und liefern ab Seite 68 den letzten Teil der Ticketverwaltung.

Und nun: Viel Spaß beim Lesen!

Ihr André Minhorst

Like, Alike, *, % und die Kompatibilität

Es gibt Dinge, die gehen an einem vorbei, ohne dass man es merkt. Das war bei mir der Fall mit den Thema der Überschrift: Ich arbeite größtenteils mit Access-Datenbanken und weiß, dass dort das Sternchen (*) als Platzhalter für beliebige Zeichen beim Einsatz des Like-Operators zum Einsatz kommt. Und wenn ich mal mit dem SQL Server arbeite, dann weiß ich, dass dort stattdessen das Prozentzeichen gefragt ist (%). Neulich erhielt ich aber eine Datenbank von einem Leser, der sich wunderte, warum eine Lösung bei ihm schlicht nicht arbeitete. Nach einigen Experimenten und einer Internetrecherche war die Lösung gefunden: Der Leser betrieb die Datenbank im SQL-92-Kompatibiltätsmodus, was bedeutet, dass die Platzhalter des SQL Servers in LIKE-Vergleichen zum Einsatz kommen. Was es damit auf sich hat und wie Sie damit umgehen, erfahren Sie im vorliegenden Beitrag.

Standardmäßig arbeitet Access mit Access-SQL, was beispielsweise heißt, dass als Platzhalter für beliebig viele Zeichen in einem **LIKE**-Ausdruck das Sternchen zum Einsatz kommt und als Platzhalter für genau ein Zeichen das Prozentzeichen.

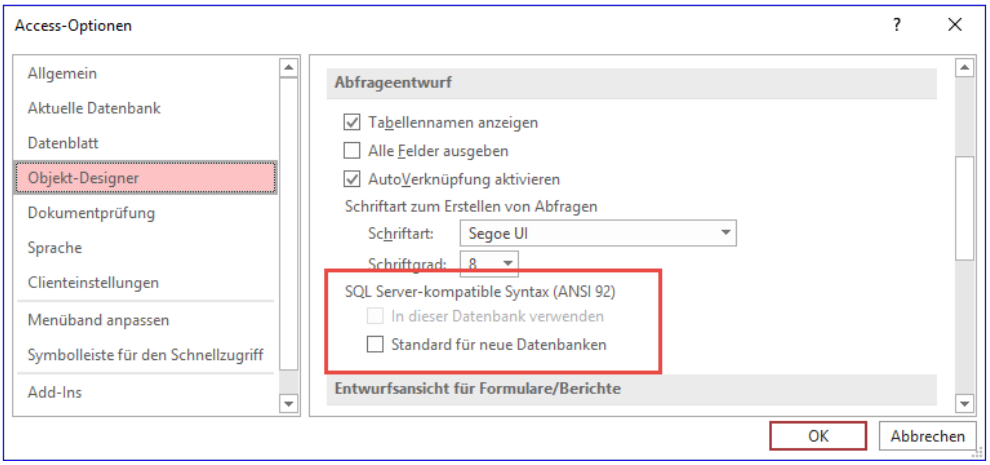

Bild 1: Einstellungen für die SQL-Syntax

Im SQL Server ist das anders: Hier setzt man für beliebig viele Zeichen das Prozentzeichen als Platzhalter (**%**) und für ein einziges Zeichen den Unterstrich (_).

SQL Server-kompatible Syntax

Allerdings kann man bereits seit Access 2010 in den Access-Optionen einstellen, in welchem Modus künftig neu erstellte Datenbanken arbeiten sollen. Dazu öffnen Sie die Access-Optionen und wechseln zum Bereich **Objekt-DesignerlAbfrageentwurflSQL Server-kompatible Syntax (ANSI92)**. Die dortige Option **Standard für neue Datenbanken** ist üblicherweise nicht aktiviert (siehe Bild 1). Dementsprechend ist auch die nicht direkt bearbeitbare Eigenschaft **In dieser Datenbank verwenden** normalerweise nicht aktiviert.

Erst wenn wir die Eigenschaft **Standard für neue Datenbanken** in den Access-Optionen einer beliebigen Datenbank (oder auch in einer Access-Instanz ohne geöffnete Datenbank – dazu nach dem Öffnen die Escape-Taste drücken, um den **Datei öffnen**-Bereich auszublenden) aktivieren, wird eine anschließend neu erstellte Datenbank im SQL Server-kompatiblen Modus geöffnet.

Wenn Sie nun eine neue Datenbank erstellen und den Optionen-Dialog erneut öffnen, sind beide Optionen aktiviert. Und Sie können die Option **In dieser Datenbank verwenden** sogar direkt einstellen (siehe Bild 2).

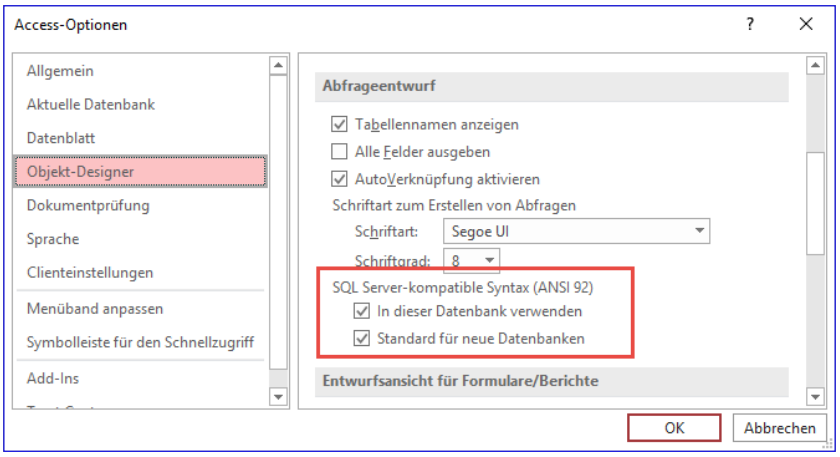

Bild 2: Aktivierte SQL Server-kompatible Syntax

Was ändert sich nun durch diese Einstellung? Beim Experimentieren mit dem **Like**-Operator im Entwurf einer Abfrage geben wir testweise den folgenden Ausdruck ein:

```
Like 'C*'
```

Nach Abschluss der Eingabe wird **Like** in den Operator **ALike** umgewandelt (siehe Bild 3). Wenn wir nun in die Datenblattansicht der Abfrage wechseln, erhalten wir allerdings ein leeres Datenblatt. Die **ALike**-Funktion arbeitet also scheinbar nicht mit den von Access-SQL bekannten Platzhaltern.

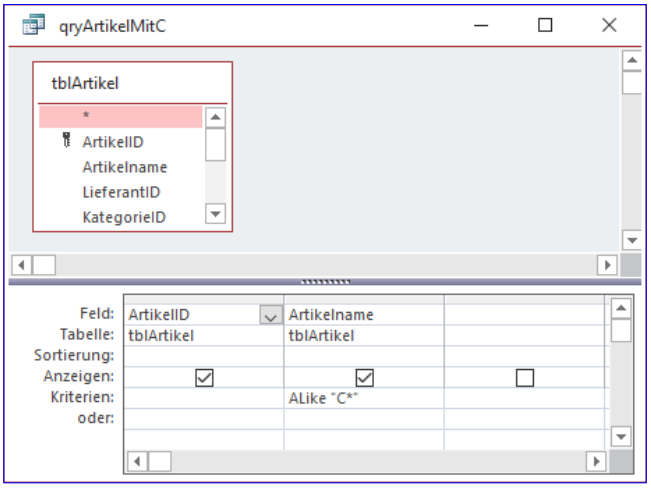

Bild 3: Umwandlung von **Like** in **ALike**

Was ist aber nun der Vorteil von **ALike**, wenn es nun schon automatisch von Access für **Like** eingesetzt wird? Der Vorteil ist, dass Sie mit **ALike** die mit SQL-92 kompatiblen Platzhalter auch unter Access einsetzen können. Sie können also mit der gleichen Abfrage sowohl auf die Daten aus Access-Tabellen zugreifen als auch direkt auf SQL Server-Tabellen.

In Datenbanken, für die **In dieser Datenbank verwenden** als Wert der Option **SQL Server-kompatible Syntax (ANSI-92)** verwendet wird, können Sie das Prozentzeichen und den Unterstrich nur verwenden, wenn Sie direkt auf SQL Server-Datenbanken und kompatible zugreifen.

Platzhalter im Code

Wenn Sie nun Code programmieren, der den **Like**-Operator mit dem Sternchen als Platzhalter verwenden soll, funktioniert dieser nur in Datenbanken wie erwartet, bei denen **In dieser Datenbank verwenden** für die Option **SQL Server-kompatible Syntax (ANSI-92)** nicht aktiviert ist.

Wenn Sie diesen Code dann in eine Datenbank übertragen, bei der die SQL Server-kompatible Syntax aktiviert ist, funktioniert dieser nicht mehr wie erwartet.

Was tun – sowohl in Abfragen als auch im SQL-Code unter VBA? Uns fehlen leider Daten, wieviele Datenbanken tatsächlich mit der SQL Server-kompatiblen Syntax betrieben werden. Wer das tut, sollte wissen, warum er das tut und wie es sich auswirkt. So sollte ein Programmierer einer solchen Datenbank sich darüber bewusst sein, dass es zu Problemen kommen kann, wenn er Code oder Objekte

Datenhistorie per Trigger

Wer seine Access-Tabellen von einem Access-Backend in ein SQL Server-Backend übertragen hat, dürfte zunächst keinen Unterschied beim Zugriff auf die Daten bemerken. Spannend wird es, wenn Sie unter Access jedoch die sogenannten Datenmakros verwendet haben, um automatisch auf Änderungen in den Daten zu reagieren und beispielsweise Kopien geänderter oder gelöschter Datensätze in einer Archivtabelle angelegt haben. Beim Migrieren nach SQL Server werden zwar auch die Archivtabellen erstellt, aber die Datenmakros bleiben außen vor. Damit keine Daten verloren gehen, zeigen wir in diesem Beitrag, wie Sie die Tabellen mit Triggern ausstatten, um die gewünschten Daten zu archivieren.

Voraussetzungen

Für den Zugriff auf den SQL Server und speziell für das Ändern des Datenmodells, zu dem auch die Trigger gehören, kann man das SQL Server Management Studio verwenden. Wir nutzen zwar gern die Tools, die wir im Beitrag **SQL Server-Tools** vorgestellt haben (siehe **www.access-im-unternehmen.de/1061**) – aber im Rahmen des vorliegenden Beitrags wollen wir die Aufgabe zunächst im SQL Server Management Studio erledigen. Später erfahren Sie, wie wir dann doch noch unsere Tools zum Anlegen von Triggern und Archivtabellen nutzen können.

Ziel des Beitrags

Das Ziel ist, für die Tabellen, für die es notwendig ist, Trigger zu definieren, die beim Löschen oder Ändern von Datensätzen einer Tabelle die zuletzt verwendete Version des jeweiligen Datensatzes zu archivieren. Wenn Sie also einen Datensatz löschen, soll dieser vor dem Löschen in eine als Archivtabelle angelegte Kopie der Tabelle mit dem Originaldatensatz kopiert werden. Wenn ein Datensatz geändert wird, soll ebenfalls eine Kopie des Originaldatensatzes in der Archivtabelle gespeichert werden.

In beiden Fällen stellt sich die Frage: Welche Daten speichern wir zusätzlich zu den zu archivierenden Daten in der Archivtabelle und wie sieht diese dann aus? Wir wollen die Kopien geänderter Datensätze auf jeden Fall den neuen Versionen der Datensätze zuordnen können. Daher sollte die Tabelle mit den archivierten Datensätzen ein Feld zum Aufnehmen des Primärschlüsselwertes des Originaldatensatzes enthalten. Außerdem wollen wir noch speichern, wann der Datensatz geändert oder gelöscht wurde und natürlich, ob er nun geändert oder gelöscht wurde. Da es mehrere Versionen des gleichen Datensatzes in der Archivtabelle geben kann, können wir also nicht das Primärschlüsselwert des Originaldatensatzes als Primärschlüsselwert des archivierten Datensatzes verwenden, sondern müssen für die Archivtabelle ein eigenes Primärschlüsselfeld vorsehen.

Wir benötigen also alle Felder der Originaltabelle, ein neues Primärschlüsselfeld sowie jeweils ein Feld für die Angabe des Datums und eines für die Aktion, welche für das Archivieren des Datensatzes verantwortlich ist – also Löschen oder Bearbeiten.

Das Primärschlüsselfeld wollen wir immer **ArchivID** nennen und die beiden anderen Felder sollen **Archivdatum** und **Archivart** heißen.

Aber ist das die beste Lösung? Wir können alternativ auch zwei Felder hinzufügen, die **Loeschdatum** und **Aenderungsdatum** heißen und je nach Vorgang das eine oder andere Feld füllen. Wir entscheiden uns für die zuletzt vorgeschlagene Variante und wollen diese beiden Felder zu den Archivtabellen hinzufügen. Die Archivtabellen

müssen wir natürlich auch noch entsprechend benennen. In diesem Fall wollen wir einfach den Namen der zu archivierenden Tabelle nutzen und dieser das Suffix **_Archiv** anhängen. Für die Tabelle **tblKunden** entsteht so beispielsweise die Tabelle **tblKunden_Archiv**.

Archivtabelle im SQL Server Management Studio anlegen

In Visual Studio legen wir die gewünschte Tabelle am schnellsten wie folgt an:

- Öffnen Sie SQL Server Management Studio.
- Navigieren Sie zu der gewünschten Tabelle in der entsprechenden Datenbank.
- Wählen Sie für die Tabelle den Kontextmenü-Eintrag **Skript für Tabelle alsICREATE inNeues Abfrage-Editor-Fenster** aus (siehe Bild 1).

Daraufhin erscheint ein neues Abfragefenster, das den SQL-Code zum Erstellen der Tabelle anzeigt, hier **tblKunden**. Hier nehmen Sie einige Änderungen vor. Die erste ist, dass wir den Tabellennamen in **tblKunden_Archiv** ändern. Danach fügen wir das Feld **ArchivID** als neuen Primärschlüsselwert hinzu und gestalten das Feld **KundeID** als normales **Integer**-Feld. Außerdem legen wir am Ende noch die beiden Felder **Loeschdatum** und **Aenderungsdatum** an, jeweils mit dem Datentyp **DATETIME**. Außerdem ändern wir noch den Namen des Primärschlüsselfeldes für den **CONSTRAINT**-Abschnitt. Der SQL-Code sieht dann in gekürzter Form wie folgt aus:

```
CREATE TABLE [dbo].[tblKunden_Archiv](
        [ArchivID] [int] IDENTITY(1,1) NOT NULL,
        [KundeID] [int] NULL,
        [KundenCode] [nvarchar](5) NULL,
        [Firma] [nvarchar](40) NOT NULL,
        [AnredeID] [int] NULL,
        [Vorname] [nvarchar](255) NULL,
```

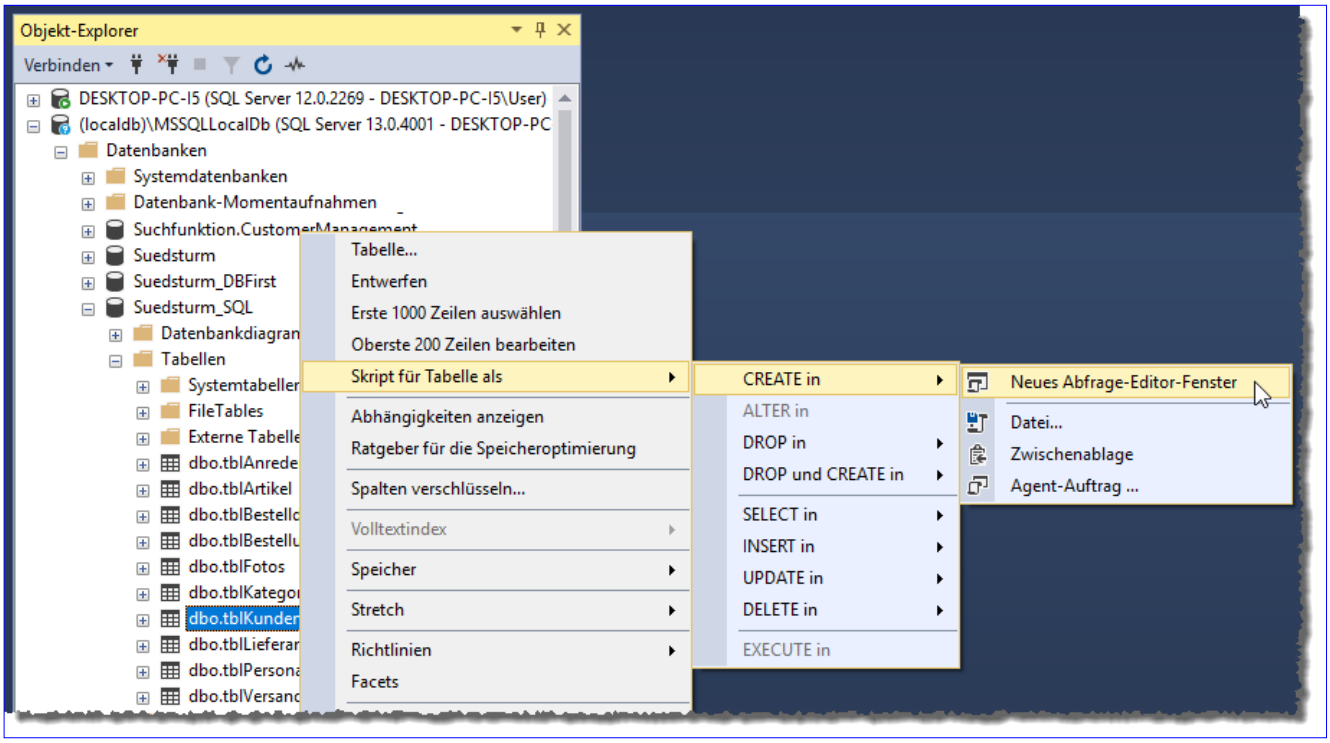

Bild 1: Archivtabelle anlegen

```
        [Nachname] [nvarchar](255) NULL,
        [Position] [nvarchar](30) NULL,
        [PLZ] [nvarchar](10) NULL,
        [Strasse] [nvarchar](255) NULL,
        [Ort] [nvarchar](15) NULL,
        [Region] [nvarchar](15) NULL,
        [Land] [nvarchar](15) NULL,
        [Telefon] [nvarchar](24) NULL,
        [Telefax] [nvarchar](24) NULL,
        [Loeschdatum] [datetime] NULL,
        [Aenderungsdatum] [datetime] NULL
 CONSTRAINT [tblKunden_Archiv$PrimaryKey] PRIMARY KEY
CLUSTERED
(
        [ArchivID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IG-
NORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS
= ON) ON [PRIMARY]
) ON [PRIMARY]
```

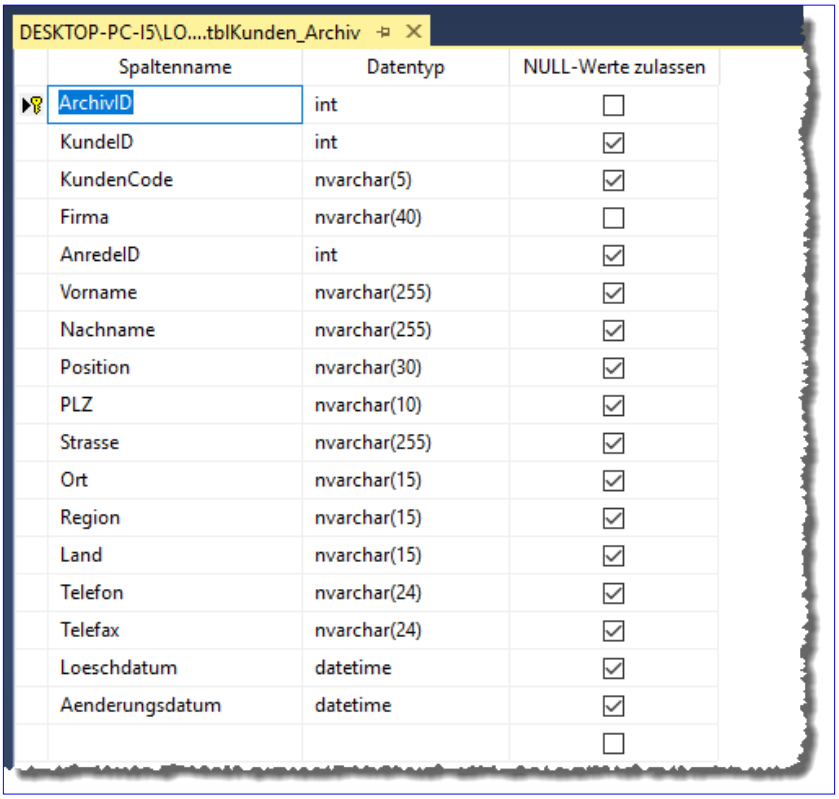

Bild 2: Entwurf der neuen Archivtabelle

Nur diesen Teil wollen wir dann auch ausführen. Löschen Sie also den Rest des automatisch erzeugten Skriptes oder markieren Sie nur diesen Teil und führen Sie diesen mit **F5** aus.

Nach einer Aktualisierung des Knotens **Tabellen** im SQL Server Management Studio erscheint die Tabelle **tblKunden_Archiv** dann in der Liste der Tabellen und der Entwurf sieht wie in Bild 2 aus.

Trigger im SQL Server Management Studio anlegen

Nun wollen wir die notwendigen Trigger für die Tabelle **tblKunden** anlegen, die dafür sorgen, dass ein zu löschender oder zu ändernder Datensatz vor dem jeweiligen Vorgang in die Tabelle **tblKunden_Archiv** gesichert wird. Wenn man das Kontextmenü der Tabelle **tblKunden** durchsieht, findet man dort keine Möglichkeit, einen Trigger anzulegen. Allerdings gibt es unterhalb des Elements **tblKunden** ein weiteres Element namens **Trigger**, dessen Kontextmenü den Eintrag **Neuer Trigger...** liefert (siehe Bild 3).

Trigger beim Aktualisieren

Dies öffnet ein neues Abfragefenster mit einer Vorlage für einen neuen Trigger. Den hier abgebildeten Code ersetzen wir durch die von uns gewünschte Version. Dabei wollen wir in einer **INSERT INTO**-Abfrage die Inhalte aller Felder des aktualisierten Datensatzes der Tabelle **tblKunden** in die entsprechenden Felder der Tabelle **tblKunden_Archiv** schreiben. Außerdem soll das Feld **Aenderungsdatum** mit dem aktuellen Datum plus Uhrzeit gefüllt werden. Dazu ändern wir die **CREATE TRIGGER**-Anweisung wie folgt ab:

```
CREATE TRIGGER dbo.tblKunden_Update
    ON  dbo.tblKunden
```

Datenhistorie-Trigger schnell anlegen

Im Beitrag »Datenhistorie per Trigger« haben wir gezeigt, welche Anweisungen nötig sind, um eine Archivtabelle anzulegen und der Originaltabelle einen Trigger hinzuzufügen, der beim Ändern oder Löschen eines Datensatzes die vorherige Version in einer Archivtabelle speichert. Im vorliegenden Beitrag wollen wir eine VBA-Prozedur entwickeln, mit der Sie zu einer Tabelle mit einem einfachen Aufruf die Archivtabelle und den Trigger in einem Rutsch anlegen können. Damit sichern Sie Ihre Daten bei Änderungen auch für mehrere Tabellen ganz schnell ab.

Voraussetzungen

In diesem Beitrag kommen die bereits im Beitrag **Datenhistorie per Trigger** (siehe **www.access-im-unternehmen.de/1211**) erwähnten Access-SQL-Tools zum Einsatz. Diese haben wir wiederum bereits im Beitrag **SQL Server-Tools** vorgestellt (siehe **www.access-im-unternehmen.de/1061**).

Ziel des Beitrags

Das Ziel ist es nun, eine Prozedur zu entwickeln, der Sie nur den Namen der Tabelle übergeben, für welche eine Archivtabelle erstellt werden soll und die einen Trigger erhalten soll, der beim Ändern oder Löschen von Datensätzen aus dieser Tabelle die vorherige Version des Datensatzes in der Archivtabelle speichert.

Dazu haben wir zunächst alle Elemente der SQL Server-Tools in die Beispieldatenbank integriert.

Prozedur zum Erstellen der Archivtabelle

Im oben genannten Beitrag **Datenhistorie per Trigger** haben wir eine Archivtabelle mit einer SQL-Anweisung wie der folgenden erstellt:

```
CREATE TABLE [dbo].[tblKunden_Archiv](
    [ArchivID] [int] IDENTITY(1,1) NOT NULL,
    [KundeID] [int] NULL,
    ...
    [Loeschdatum] [datetime] NULL,
    [Aenderungsdatum] [datetime] NULL
CONSTRAINT [tblKunden_Archiv$PrimaryKey] PRIMARY KEY
CLUSTERED
(
        [ArchivID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

Diese Anweisung wollen wir nun parametrisieren, damit wir diese für beliebige Tabellennamen erstellen können. Dazu benötigen wir etwas Wissen über die Tabelle, für die wir die Archivtabelle anlegen wollen. Da wir nicht davon ausgehen können, dass es eine per ODBC erstellte Verknüpfung auf diese Tabelle gibt, wollen wir die Felder und ihre Eigenschaften direkt vom SQL Server beziehen. Um die Feldnamen und die Datentypen zu ermitteln, können wir die folgende Abfrage nutzen, die wir direkt an den SQL Server absetzen:

```
SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = 'tblKunden'
```

Im SQL Server sieht das Ergebnis für die Tabelle **tblKunden** wie in Bild 1 aus.

Funktion zum Einlesen der Feldliste für CREATE TABLE

Wir benötigen eine Funktion, mit der wir die in der Archivtabelle anzulegenden Felder ermitteln.

Ergebnisse | Meldungen

	TABLE_CATALOG	TABLE_SCHEMA	TABLE_NAME	COLUMN_NAME	ORDINAL_POSITION	COLUMN_DEFAULT	IS_NULLABLE	DATA_TYPE	CHARACTER_MAXIMUM_LENGTH
1	Suedsturm_SQL	dbo	tblKunden	KundeID	1	NULL	NO	int	NULL
2	Suedsturm_SQL	dbo	tblKunden	KundenCode	2	NULL	YES	nvarchar	5
3	Suedsturm_SQL	dbo	tblKunden	Firma	3	NULL	NO	nvarchar	40
4	Suedsturm_SQL	dbo	tblKunden	AnredeID	4	NULL	YES	int	NULL
5	Suedsturm_SQL	dbo	tblKunden	Vorname	5	NULL	YES	nvarchar	255
6	Suedsturm_SQL	dbo	tblKunden	Nachname	6	NULL	YES	nvarchar	255
7	Suedsturm_SQL	dbo	tblKunden	Position	7	NULL	YES	nvarchar	30
8	Suedsturm_SQL	dbo	tblKunden	PLZ	8	NULL	YES	nvarchar	10
9	Suedsturm_SQL	dbo	tblKunden	Strasse	9	NULL	YES	nvarchar	255
10	Suedsturm_SQL	dbo	tblKunden	Ort	10	NULL	YES	nvarchar	15
11	Suedsturm_SQL	dbo	tblKunden	Region	11	NULL	YES	nvarchar	15
12	Suedsturm_SQL	dbo	tblKunden	Land	12	NULL	YES	nvarchar	15
13	Suedsturm_SQL	dbo	tblKunden	Telefon	13	NULL	YES	nvarchar	24
14	Suedsturm_SQL	dbo	tblKunden	Telefax	14	NULL	YES	nvarchar	24

Bild 1: Ausgabe der Abfrage der Eigenschaften der Felder einer Tabelle

```
Public Function FeldlisteZusammenstellen(strTabelle As String, strVerbindungszeichenfolge As String) As String
    Dim rst As DAO.Recordset
    Dim strFeldliste As String
    Set rst = mdlToolsSQLServer.SQLRecordset("SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = '" _
        & strTabelle & "'", strVerbindungszeichenfolge)
    strFeldliste = "    [ArchivID] [int] IDENTITY(1,1) NOT NULL," & vbCrLf
    Do While Not rst.EOF
        If Not rst!DATA_TYPE = "Timestamp" Then
            strFeldliste = strFeldliste & "    [" & rst!COLUMN_NAME & "]"
            strFeldliste = strFeldliste & " [" & rst!DATA_TYPE & "]"
            If Not IsNull(rst!Character_maximum_Length) Then
                If Not rst!Character_maximum_Length = -1 Then
                    strFeldliste = strFeldliste & "(" & rst!Character_maximum_Length & ")"
                Else
                    strFeldliste = strFeldliste & "(max)"
                End If
            End If
            If rst!IS_NULLABLE = "YES" Then
                strFeldliste = strFeldliste & " NULL"
            Else
                strFeldliste = strFeldliste & " NOT NULL"
            End If
            strFeldliste = strFeldliste & "," & vbCrLf
        End If
        rst.MoveNext
    Loop
    strFeldliste = strFeldliste & "    [Loeschdatum] [datetime] NULL," & vbCrLf
    strFeldliste = strFeldliste & "    [Aenderungsdatum] [datetime] NULL"
    FeldlisteZusammenstellen = strFeldliste
End Function
```

Listing 1: Funktion zum Ermitteln der Feldliste

Diese heißt **FeldlisteZusammenstellen** und sieht wie in Listing 1 aus.

Die Funktion nimmt den Namen der zu untersuchenden Tabelle sowie die Verbindungszeichenfolge der SQL Server-Datenbank als Parameter entgegen und liefert einen String als Ergebnis zurück. Die Funktion verwendet die Funktion **SQLRecordset** des Moduls **mdlToolsSQLServer**, um die Datensätze der Tabelle **INFORMATION_SCHEMA.COLUMNS** zurückzuliefern.

Dabei wollen wir nach dem Feld **TABLE_NAME** filtern, das die Datensätze auf diejenigen einschränken soll, deren Tabellenname dem Wert des Parameters **strTabelle** entspricht. Auch diese Funktion nimmt wieder die Verbindungszeichenfolge als Parameter entgegen.

Danach stellt die Funktion **FeldlisteZusammenstellen** in der Variablen **strFeldliste** die Liste der anzulegenden Felder für die Archivtabelle zusammen. Bevor wir die Datensätze des Recordsets **rst** durchlaufen, fügen wir dort bereits als ersten Eintrag den folgenden hinzu:

```
[ArchivID] [int] IDENTITY(1,1) NOT NULL,
```

Dies entspricht dem Primärschlüsselfeld für die Archivtabelle. Das Komma am Ende deutet bereits an, dass nun weitere Felder folgen. Diese stellen wir in einer **Do While**-Schleife über alle Datensätze der Tabelle zusammen. Ein Feld wird nur hinzugefügt, wenn es nicht den Datentyp **Timestamp** hat, was wir in einer **If...Then**-Bedingung prüfen.

Ist das nicht der Fall, berücksichtigen wir als Erstes den Inhalt des Feldes **COLUMN_NAME** und damit den Feldnamen des anzulegenden Feldes, den wir in eckige Klammern einfassen.

Dann folgt der Datentyp, den wir aus dem Feld **DATA_TYPE** des Recordsets entnehmen und den wir durch ein Leerzeichen vom Feldnamen getrennt ebenfalls in eckige Klammern einfassen. Dann prüfen wir, ob das Feld **CHARACTER_MAXIMUM_LENGTH** einen Wert enthält. Dieses liefert die Anzahl der Zeichen für Textfelder. Liegt ein Wert vor, fügen wir diesen in runde Klammern eingefasst zum Ausdruck in **strFeldListe** ein. Aber es kann auch sein, dass der Wert **-1** beträgt. Dieser tritt auf, wenn der eigentliche Wert **max** lautet, was auf ein langes Textfeld hinweist. In diesem Fall wollen wir nicht **-1**, sondern **max** in runden Klammern zur Felddefinition hinzufügen.

Schließlich folgt das Feld **ISNULLABLE**, das mit den Werten **YES** und **NO** angibt, ob das Feld Nullwerte aufnehmen darf oder nicht. Im Falle von **YES** fügen wir **NULL** zur Felddefinition hinzu, anderenfalls **NOT NULL**. Anschließend hängen wir noch ein abschließendes Komma an die Beschreibung dieses Feldes an und gehen dann in die nächste Runde der **Do While**-Schleife.

Sind alle Felder durchlaufen, fehlen noch die beiden Felder **Loeschdatum** und **Aenderungsdatum**, die beide mit dem Datentyp **datetime** angelegt werden sollen und Nullwerte enthalten dürfen.

Jeder Felddefinition fügen wir außerdem noch einen Zeilenumbruch hinzu, sodass das Ergebnis der Funktion beispielsweise für die Tabelle **tblKunden** wie folgt aussieht:

```
[ArchivID] [int] IDENTITY(1,1) NOT NULL,
 [KundeID] [int] NOT NULL,
 [KundenCode] [nvarchar](5) NULL,
 [Firma] [nvarchar](40) NOT NULL,
 [AnredeID] [int] NULL,
 [Vorname] [nvarchar](255) NULL,
 [Nachname] [nvarchar](255) NULL,
 [Position] [nvarchar](30) NULL,
 [PLZ] [nvarchar](10) NULL,
 [Strasse] [nvarchar](255) NULL,
 [Ort] [nvarchar](15) NULL,
 [Region] [nvarchar](15) NULL,
 [Land] [nvarchar](15) NULL,
 [Telefon] [nvarchar](24) NULL,
 [Telefax] [nvarchar](24) NULL,
```

```
Public Function CreateTableZusammenstellen(strTabelle As String, strVerbindungszeichenfolge As String) As String
    Dim strCreateTable As String
    strCreateTable = "CREATE TABLE [dbo].[" & strTabelle & "_Archiv](" & vbCrLf
    strCreateTable = strCreateTable & FeldlisteZusammenstellen(strTabelle, strVerbindungszeichenfolge) & vbCrLf
    strCreateTable = strCreateTable & "CONSTRAINT [" & strTabelle & "_Archiv$PrimaryKey] PRIMARY KEY CLUSTERED" _
        & vbCrLf
    strCreateTable = strCreateTable & "(" & vbCrLf
    strCreateTable = strCreateTable & "    [ArchivID] Asc" & vbCrLf
    strCreateTable = strCreateTable & ")WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, " _
        & "ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]" & vbCrLf
    strCreateTable = strCreateTable & ") ON [PRIMARY]"
    CreateTableZusammenstellen = strCreateTable
End Function
```

Listing 2: Funktion zum Ermitteln der Definition der Archivtabelle

```
    [Loeschdatum] [datetime] NULL,
    [Aenderungsdatum] [datetime] NULL
```

Diese Zeichenkette wird dann als Ergebnis der Funktion zurückgegeben. Den Aufruf der Funktion zum Testen können Sie so gestalten:

```
Public Sub FieldlisteCreate_Test()
    Dim strVerbindungszeichenfolge As String
    strVerbindungszeichenfolge = _
        VerbindungszeichenfolgeNachID(10)
    Debug.Print FieldlistCreate("tblKunden", _
        strVerbindungszeichenfolge)
End Sub
```

Hier nutzen wir die Funktion **Verbindungszeichenfolge-NachID**, um die Verbindungszeichenfolge zu ermitteln. Die SQL Server-Tools enthalten eine Tabelle namens **tblVerbindungszeichenfolgen**, welche die Informationen für die Verbindungszeichenfolgen speichert.

Mit dem Formular **frmVerbindungszeichenfolgen** können Sie die Verbindungszeichenfolge zusammenstellen und auch testen. Merken Sie sich die ID der dabei gespeicherten Verbindungszeichenfolge und übergeben Sie diese der Funktion **VerbindungszeichenfolgeNachID** als Parameter.

Funktion CreateTableZusammenstellen

Die Funktion aus Listing 2 stellt die aus der Funktion **FeldlisteZusammenstellen** ermittelten Felder mit einigen weiteren Zeilen zu der kompletten **CREATE TABLE**-Anweisung zusammen. Diese Funktion erwartet die gleichen Parameter wie die Funktion **FeldlisteZusammenstellen**.

Archivtabelle erstellen

Mit dem Ergebnis können wir nun bereits die Archivtabelle erstellen. Während wir die hier zusammengestellte **CREATE TABLE**-Anweisung normalerweise im SQL Server Management Studio ausführen würden, wollen wir dies auch von Access aus erledigen. Dazu liefern die SQL Server Tools auch eine Funktion. Diese heißt **SQLAktionsabfrageOhneErgebnis** und erwartet die auszuführende Abfrage sowie die Verbindungszeichenfolge als Parameter und einen optionalen Parameter für die Rückgabe einer eventuellen Fehlermeldung.

Diese Funktion rufen wir von einer anderen Funktion namens **ArchivtabelleErstellen** auf, die Sie in Listing 3 finden. Diese ermittelt zunächst die Verbindungszeichenfolge und mit der Funktion **CreateTableZusammenstellen** die SQL-Anweisung zum Erstellen der Archivtabelle. Dann ruft sie die Funktion **SQLAktionsabfrageOhneErgebnis** auf und speichert das Ergebnis in der Variablen **lngErgebnis**. Außerdem verwendet sie als Rückgabeparameter die

```
Public Function ArchivtabelleErstellen() As Boolean
    Dim strTabelle As String
    Dim strCreateTable As String
    Dim strVerbindungszeichenfolge As String
    Dim strFehlermeldung As String
    Dim lngErgebnis As Long
    strTabelle = "tblKunden"
    strVerbindungszeichenfolge = VerbindungszeichenfolgeNachID(10)
    strCreateTable = CREATETABLEZusammenstellen(strTabelle, strVerbindungszeichenfolge)
    lngErgebnis = SQLAktionsabfrageOhneErgebnis(strCreateTable, strVerbindungszeichenfolge, strFehlermeldung)
    If Not lngErgebnis = 0 Then
        Debug.Print lngErgebnis, strFehlermeldung
    Else
        ArchivtabelleErstellen = True
    End If
End Function
```

Listing 3: Funktion zum Anlegen der Archivtabelle

Variable **strFehlermeldung**. Liefert **SQLAktionsabfrageOhneErgebnis** einen Wert ungleich **0**, können Sie der Variablen **strFehlermeldung** den Text der Meldung des aufgetretenen Fehlers entnehmen.

Ob das Ausführen der Abfrage erfolgreich war, prüfen wir in einer **If...Then**-Bedingung, in der wir gegebenenfalls die Fehlermeldung im Direktfenster ausgeben. Außerdem wird der Rückgabewert der Funktion **ArchivtabelleErstellen** nur dann auf **True** eingestellt, wenn kein Fehler aufgetreten ist. Ein Fehler tritt beispielsweise auf, wenn wir eine Archivtabelle erstellen wollen, die bereits vorhanden ist. Die Meldung lautet dann:

```
[Microsoft][SQL Server Native Client 11.0][SQL Server]
There is already an object named 'tblKunden_Archiv' in the
database.
```

Prozedur zum Erstellen des Triggers

Nun wollen wir die Funktion zum Erstellen des Triggers zusammenstellen. Der Trigger soll, am Beispiel der Tabelle **tblKunden**, gekürzt wie folgt aussehen:

```
CREATE TRIGGER [dbo].[tblKunden_DeleteUpdate]
    ON  [dbo].[tblKunden]
    AFTER DELETE, UPDATE
AS
BEGIN
    SET NOCOUNT ON;
    INSERT INTO dbo.tblKunden_Archiv([KundeID],
        [KundenCode], ...) SELECT [KundeID],
        [KundenCode], ... FROM deleted;
    IF EXISTS(SELECT 1 FROM inserted)
        UPDATE dbo.tblKunden_Archiv
            SET Aenderungsdatum = GETDATE()
            WHERE tblKunden_Archiv.ArchivID = @@IDENTITY
    ELSE
        UPDATE dbo.tblKunden_Archiv
            SET Loeschdatum = GETDATE()
            WHERE tblKunden_Archiv.ArchivID = @@IDENTITY
END
```

Dazu benötigen wir zunächst einmal wieder eine Liste aller Felder der Tabelle. Diese lesen wir mithilfe der Funktion **FeldlisteTriggerZusammenstellen** aus Listing 4 ein. Die Funktion arbeitet grundsätzlich wie die bereits weiter oben vorgestellte Funktion **FeldlisteZusammenstellen**, mit der wir die Felddefinitionen für die **CREATE TABLE**-Anweisung zusammengestellt haben. In diesem Fall stellen wir allerdings lediglich eine kommaseparierte Liste aller

```
Public Function FeldlisteTriggerZusammenstellen(strTabelle As String, strVerbindungszeichenfolge As String) As String
    Dim rst As DAO.Recordset
    Dim strFeldliste As String
    Set rst = mdlToolsSQLServer.SQLRecordset("SELECT * FROM INFORMATION_SCHEMA.COLUMNS WHERE TABLE_NAME = '" _
        & strTabelle & "'", strVerbindungszeichenfolge)
    Do While Not rst.EOF
        If Not rst!DATA_TYPE = "timestamp" Then
            strFeldliste = strFeldliste & ", [" & rst!COLUMN_NAME & "]"
        End If
        rst.MoveNext
    Loop
    strFeldliste = Mid(strFeldliste, 3)
    FeldlisteTriggerZusammenstellen = strFeldliste
End Function
```

Listing 4: Funktion zum Zusammenstellen der Feldliste für den Trigger

Felder der per Parameter übergebenen Tabelle zusammen. Die Feldnamen sollen dabei wieder in eckige Klammern eingefasst werden. Nach dem Durchlaufen aller Datensätze des Recordsets mit den Feldern der zu untersuchenden Tabelle, bei dem wir jedem Feld ein Komma vorangestellt haben, wird das Komma vor dem ersten Feld entfernt. Das Ergebnis dieser Funktion lautet am Beispiel der Tabelle **tblKunden** wie folgt:

```
[KundeID], [KundenCode], [Firma], [AnredeID], [Vorname],
[Nachname], [Position], [PLZ], [Strasse], [Ort], [Region],
[Land], [Telefon], [Telefax]
```

```
Public Function CREATETRIGGERZusammenstellen(strTabelle As String, strVerbindungszeichenfolge As String)
    Dim strCreateTrigger As String
    Dim strFeldliste As String
    strFeldliste = FeldlisteTriggerZusammenstellen(strTabelle, strVerbindungszeichenfolge)
    strCreateTrigger = "CREATE TRIGGER [dbo].[" & strTabelle & "_DeleteUpdate]" & vbCrLf
    strCreateTrigger = strCreateTrigger & "    ON  [dbo].[" & strTabelle & "]" & vbCrLf
    strCreateTrigger = strCreateTrigger & "    AFTER Delete, Update" & vbCrLf
    strCreateTrigger = strCreateTrigger & "AS" & vbCrLf
    strCreateTrigger = strCreateTrigger & "BEGIN" & vbCrLf
    strCreateTrigger = strCreateTrigger & "    SET NOCOUNT ON;" & vbCrLf
    strCreateTrigger = strCreateTrigger & "    INSERT INTO dbo." & strTabelle & "_Archiv(" & strFeldliste _
        & ") SELECT " & strFeldliste & " FROM deleted;" & vbCrLf
    strCreateTrigger = strCreateTrigger & "    IF EXISTS(SELECT 1 FROM inserted)" & vbCrLf
    strCreateTrigger = strCreateTrigger & "        UPDATE dbo." & strTabelle & "_Archiv SET Aenderungsdatum = " _
        & "GETDATE() WHERE " & strTabelle & "_Archiv.ArchivID = @@IDENTITY" & vbCrLf
    strCreateTrigger = strCreateTrigger & "    Else" & vbCrLf
    strCreateTrigger = strCreateTrigger & "        UPDATE dbo." & strTabelle & "_Archiv SET Loeschdatum = " _
        & "GETDATE() WHERE " & strTabelle & "_Archiv.ArchivID = @@IDENTITY" & vbCrLf
    strCreateTrigger = strCreateTrigger & "End" & vbCrLf
    CREATETRIGGERZusammenstellen = strCreateTrigger
End Function
```

Listing 5: Funktion zum Zusammenstellen der Abfrage zum Zusammenstellen der **CREATE TRIGGER**-Anweisung

Verweise per VBA verwalten

Wenn Sie den Umfang der Objekte, Methoden, Eigenschaften und Ereignisse unter VBA erweitern wollen, geht kein Weg am Verweise-Dialog des VBA-Editors zum Hinzufügen neuer Verweise vorbei. Hier legen Sie fest, welche zusätzlichen Bibliotheken neben den eingebauten Bibliotheken noch ihre Elemente für die Programmierung im aktuellen VBA-Projekt bereitstellen sollen. In diesem Beitrag werfen wir einen kurzen Blick auf den Verweise-Dialog, aber vor allem schauen wir uns an, wie Sie per VBA auf die enthaltenen Einträge zugreifen und diese verwalten.

Wenn Sie vom Access-Fenster aus mit einer geöffneten Datenbankdatei eine der Tastenkombinationen **Strg + G** oder **Alt + F11** betätigen, erscheint der VBA-Editor. Hier öffnen Sie mit dem Menübefehl **Extras|Verweise** den **Verweise**-Dialog. Dieser enthält standardmäßig, also für eine neu unter Access 2016 angelegte Access-Datenbank, die Einträge wie in Bild 1.

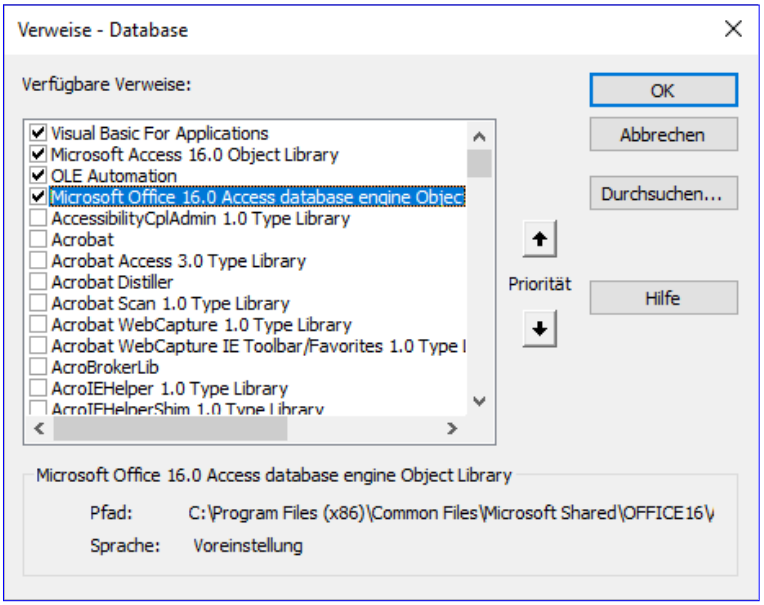

Bild 1: Der **Verweise**-Dialog von Access

Sie können Verweise hinzufügen, indem Sie diese in der Liste auffinden und das Kontrollkästchen des gewünschten Eintrags aktivieren. Um einen Verweis zu entfernen, entfernen Sie einfach den passenden Haken. Sie können auch Bibliotheken hinzufügen, die nicht in der Liste angezeigt werden. Dazu klicken Sie auf die Schaltfläche **Durchsuchen...**, mit der Sie einen Dialog zum Auswählen der hinzuzufügenden Datei öffnen.

Die References-Auflistung

Es gibt zwei interessante Elemente, die wir uns in diesem Beitrag ansehen werden: die **References**-Auflistung und das **Reference**-Objekt.

Die **References**-Auflistung liefert die folgenden Eigenschaften, Methoden und Ereignisse:

- **AddFromFile**: Fügt einen Verweis über die Angabe des Pfades zu der Bibliotheksdatei zur Verweisliste hinzu.
- **AddFromGuid**: Fügt einen Verweis über die eindeutige GUID des Verweises hinzu.
- **Count**: Liefert die Anzahl der aktuell vom Projekt referenzierten Verweise.
- **Item**: Erlaubt das Referenzieren eines **Reference**-Objekts über den Index. Sie können allerdings auch über die Eigenschaft **Name** eines **Reference**-Objekts auf das Element verweisen, zum Beispiel über **vba**.
- **ItemAdded**: Ereignis, das beim Hinzufügen eines Verweises ausgelöst wird.

- **ItemRemoved**: Ereignis, das beim Entfernen eines Verweises ausgelöst wird.
- **Parent**: Verweis auf das übergeordnete Objekt der **References**-Auflistung.
- **Remove**: Entfernen eines als **Reference**-Objekt angegebenen Verweises.

Das Reference-Objekt

Die **References**-Auflistung enthält Objekte des Typs **Reference**. Diese Objekte haben die folgenden Eigenschaften:

- **BuiltIn**: Boolean-Eigenschaft, die angibt, ob es sich um einen eingebauten Verweis handelt – also um einen solchen, den Sie nicht entfernen können.
- **Collection**: Liefert einen Verweis auf die **References**-Auflistung, in der sich das **Reference**-Element befindet.
- **FullPath**: Gibt den kompletten Pfad zur referenzierten Bibliotheksdatei an.
- **Guid**: Gibt die eindeutige GUID zur Identifizierung der referenzierten Bibliothek an.
- **IsBroken**: Gibt an, ob die im Verweis vorhandene Bibliotheksdatei vorhanden ist oder nicht.
- **Kind**: Gibt an, ob es sich um einen Verweis auf eine Bibliothek oder auf ein Visual Basic-Projekt handelt.
- **Major**: Gibt die Hauptversionsnummer der referenzierten Bibliothek an.
- **Minor**: Gibt die Unterversion der referenzierten Bibliothek an.
- **Name**: Liefert den Namen der Bibliothek, den Sie im VBA-Editor zum Referenzieren der Elemente dieser Bibliothek verwenden können.

Durchlaufen der vorhandenen Verweise

Um die im **Verweise**-Dialog angezeigten und somit eingebundenen Bibliotheken auszugeben, können Sie die **References**-Auflistung durchlaufen und dabei die Eigenschaften der **Reference**-Objekte ausgeben. Das können Sie sowohl mit einer **For...Next**-Schleife als auch mit einer **For Each**-Schleife erledigen. Mit einer **For...Next**-Schleife verwenden Sie eine Zählervariable und greifen über die **Item**-Eigenschaft auf die Verweise zu:

```
Public Sub VerweisePerForNext()
    Dim i As Integer
    Dim ref As Reference
    For i = 1 To References.Count
        Set ref = References.Item(i)
        Debug.Print ref.Name
    Next i
End Sub
```

Dies liefert im Direktbereich des VBA-Editors für die standardmäßig vorhandenen Bibliotheken die folgende Ausgabe:

```
VBA
Access
stdole
DAO
```

In einer **For Each**-Schleife weisen Sie die **Reference**-Objekte direkt der in der Schleife verwendeten Variablen zu:

```
Public Sub VerweisePerForEach()
    Dim ref As Reference
    For Each ref In References
        Debug.Print ref.Name
    Next ref
End Sub
```

Eigenschaften von Verweisen ausgeben

Mit der folgenden Prozedur geben wir die Werte aller Eigenschaften des mit der Variablen **ref** referenzierten

Reference-Elements (außer **Collection**) im Direktbereich des VBA-Editors aus:

```
Public Sub Verweiseigenschaften()
    Dim ref As Reference
    For Each ref In References
        Debug.Print "BuildIn:  " & ref.BuiltIn
        Debug.Print "FullPath: " & ref.FullPath
        Debug.Print "GUID:     " & ref.Guid
        Debug.Print "IsBroken: " & ref.IsBroken
        Debug.Print "Kind:     " & ref.Kind
        Debug.Print "Major:    " & ref.Major
        Debug.Print "Minor:    " & ref.Minor
        Debug.Print "Name:     " & ref.Name
    Next ref
End Sub
```

Für die Bibliothek **Visual Basic for Applications** erhalten wir beispielsweise die folgende Ausgabe:

```
BuildIn:  Wahr
FullPath: C:\Program Files (x86)\Common Files\Microsoft
Shared\VBA\VBA7.1\VBE7.DLL
GUID:     {000204EF-0000-0000-C000-000000000046}
IsBroken: Falsch
Kind:     0
Major:    4
Minor:    2
Name:     VBA
```

Verweise hinzufügen

Zum Hinzufügen von Verweisen gibt es zwei Methoden. Die erste ist das Hinzufügen über den Dateinamen mit **AddFromFile**, die zweite das Hinzufügen über die GUID mit **AddFromGUID**. Um die beiden Methoden auszuprobieren, fügen wir zunächst einen Verweis auf eine Bibliothek über den **Verweise**-Dialog hinzu, und zwar die Bibliothek **Microsoft Office x.0 Object Library**. Diese finden Sie in aktuelleren Access-Versionen nicht unter diesem Namen in der Liste der Verweise, sondern unter der Bezeichnung **Office** (siehe Bild 2).

Wenn Sie den Verweis allerdings erst einmal hinzugefügt haben und den **Verweise**-Dialog erneut öffnen, erscheint die richtige Bezeichnung in der Liste der Verweise, nämlich im Falle von Access 2016 **Microsoft Office 16.0 Object Library**.

Um die beiden Methoden **AddFromFile** und **AddFromGuid** auszuprobieren, lassen wir uns den Dateinamen und die GUID dieses Verweises im Direktbereich ausgeben – das erledigen wir mit der Prozedur **Verweiseigenschaften**, die wir weiter oben bereits vorgestellt haben.

Die Ergebnisse lauten:

```
FullPath: C:\Program Files (x86)\Microsoft Office\root\
VFS\ProgramFilesCommonX86\Microsoft Shared\Office16\MSO.
DLL
GUID:     {2DF8D04C-5BFA-101B-BDE5-00AA0044DE52}
```

Diese beiden Informationen nutzen wir nun als Parameter der Methoden **AddFromFile** und **AddFromGuid**. Vorher

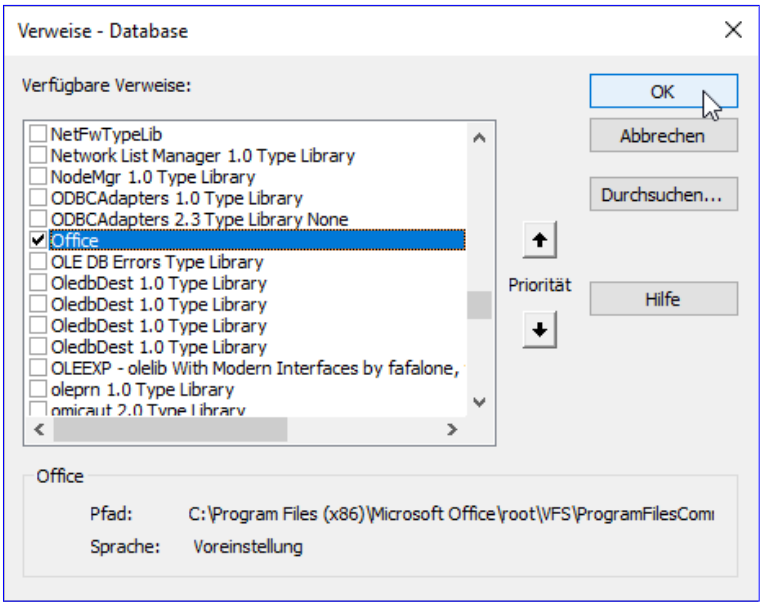

Bild 2: Hinzufügen des Verweises auf die Office-Bibliothek

Flexible Adressen

Das Thema Adressverwaltung kann man auf viele verschiedene Arten lösen. In vielen Fällen reicht eine einfache Tabelle, die eine Adresse speichert. Reden wir über Adressen in einer Kundenverwaltung, kann es auch vorkommen, dass der Kunde unterschiedliche Adressen für Lieferungen und Rechnungen angeben möchte. Solche Daten speichert man entweder in einer einzigen Tabelle je Kunde oder man trägt die Liefer- und Rechnungsadresse in eigene Tabellen ein, die man dann per 1:1-Beziehung mit dem Kundendatensatz verknüpft. Wir gehen noch einen Schritt weiter und wollen mit der hier vorgestellten Lösung die Möglichkeit bieten, nicht nur zwei, sondern beliebig viele Adressen je Kunde anzulegen – die dann flexibel als Rechnung- oder Lieferadresse festgelegt werden können.

Eigentlich ist es kein Problem, die Adressdaten eines Kunden oder einer Person in einem Datensatz einer Tabelle zu speichern. Sie legen dann Felder für ein oder zwei Adressen an, hier Liefer- und Rechnungsadresse, und speichern den Datensatz. Aus dieser Tabelle können Sie dann ganz einfach die Lieferadresse und die Rechnungsadresse entnehmen.

Nachteile herkömmlicher Kundentabellen

Aber was, wenn der Kunde vorübergehend unter einer anderen Lieferadresse erreichbar ist? Oder wenn er eine Rechnung zwischenzeitlich an seinen Arbeitgeber schicken lassen will, statt zu sich nach Hause?

Dann müssen Sie entweder zur klassischen Methode greifen und die Rechnung oder das Paket von Hand beschriften oder Sie ändern die Rechnungs- oder Lieferadresse vorübergehend, um diese anschließend wieder in den alten Zustand zu versetzen. Aber wie stellen Sie sicher, dass Sie dann nicht vergessen, die vorübergehende Adresse wieder herauszulöschen? Und wo speichern Sie in der Zwischenzeit die üblicherweise zu verwendende Adresse?

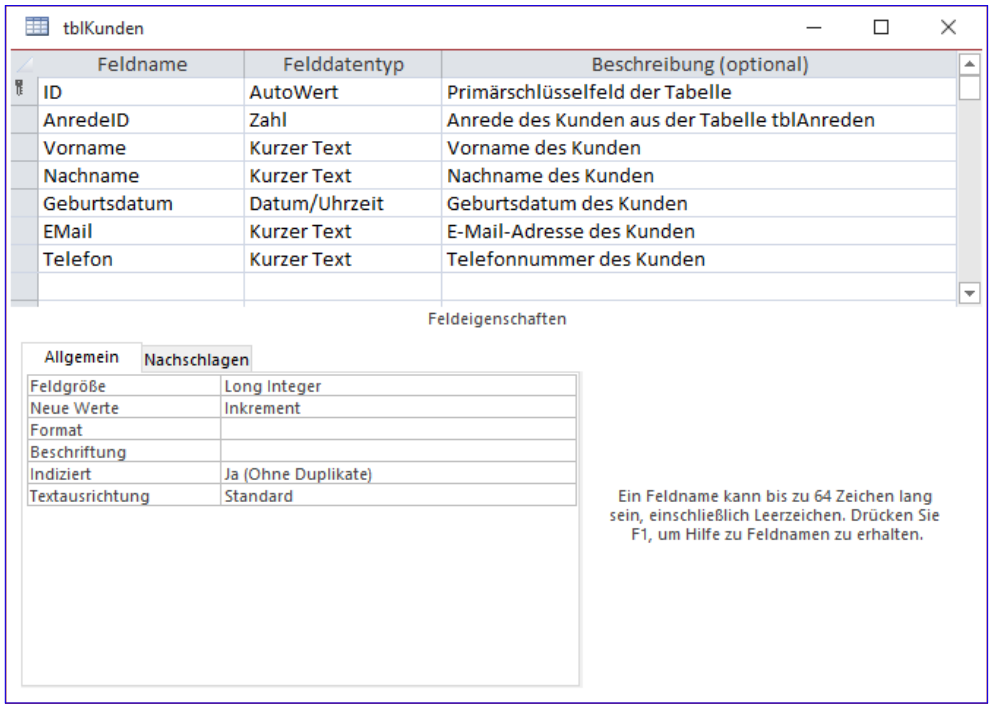

Bild 1: Entwurf der Tabelle **tblKunden**

Solche Fragen stellen sich auch, wenn Sie mit der hier vorgestellten Lösung arbeiten, aber sie macht das Leben deutlich einfacher.

Datenmodell für flexible Adressen

Im Datenmodell für unsere Lösung finden wir zunächst die übliche Tabelle etwa namens **tblKunden**. Diese enthält einige Basisdaten wie etwa die Anrede, den Vor- und den Nachnamen des Kunden sowie das Geburtsdatum. Wir haben beispielsweise auch noch die E-Mail-Adresse und die Telefonnummer des Kunden erfasst. Sie können nach Bedarf noch weitere organisatorische Daten erfassen, wie etwa das Datum, an dem der Kundendatensatz erstellt wurde.

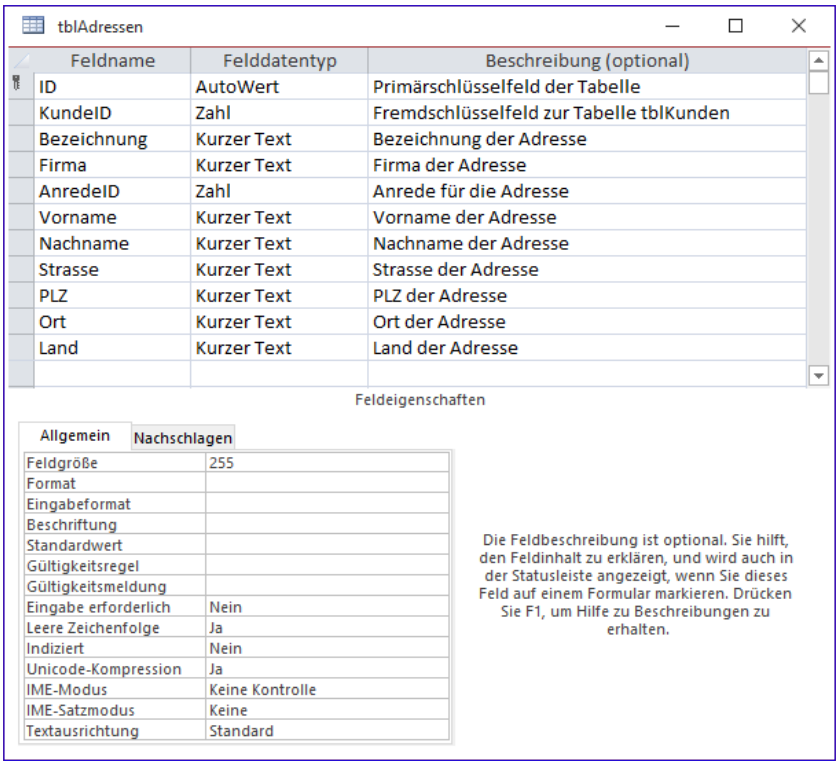

Bild 2: Entwurf der Tabelle **tblAdresse**

Diese Tabelle sieht vorerst wie in Bild 1 aus. Das Feld **AnredeID** ist ein Fremdschlüsselfeld zur Tabelle **tblAnreden**, welche wir hier nicht abbilden. Diese Tabelle besteht lediglich aus den Feldern **ID** und **Anrede**.

Die neue Tabelle zum Speichern von Liefer- und Rechnungsadressen heißt schlicht **tblAdressen**. Sie sieht im Entwurf wie in Bild 2 aus. Sie enthält nochmals die Felder **AnredeID**, **Vorname** und **Nachname** – das deshalb, weil ja auch einmal eine ganz andere Person eine Lieferung oder eine Rechnung erhalten können soll. Außerdem finden Sie hier die üblichen Felder zum Erfassen einer Adresse wie **Firma**, **Strasse**, **PLZ**, **Ort** und **Land**.

Interessant wird es beim Feld **KundeID**. Dabei handelt es sich um ein Fremdschlüsselfeld, mit dem Sie einen der Datensätze der Tabelle **tblKunden** auswählen können. Wir haben es als Nachschlagefeld ausgelegt.

Außerdem finden Sie hier das Feld **Bezeichnung**. Damit soll der Benutzer eine individuelle Bezeichnung für die Adresse angeben können, damit er diese später leicht identifizieren kann. Es handelt sich jedoch um kein Pflichtfeld.

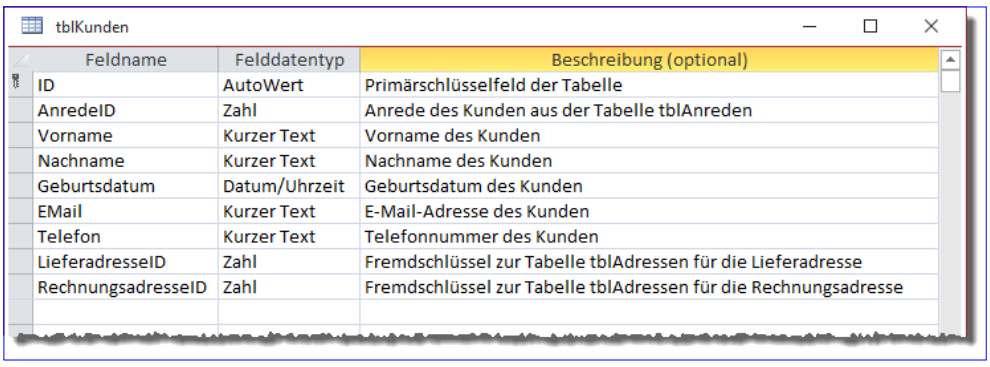

Bild 3: Entwurf der Tabelle **tblAdresse** mit Feldern zur Auswahl von Rechnungs- und Lieferadresse

Von der Adresse zum Kunden zur Adresse

Nun können wir zwar bereits für eine Adresse festlegen, zu welchem Kunden diese gehört, und auf diese Weise einem Kunden keine, eine oder auch mehrere Adressen zuweisen. Aber woher wissen wir, welche Adresse die Lieferadresse des Kunden ist und welche die Rechnungsadresse?

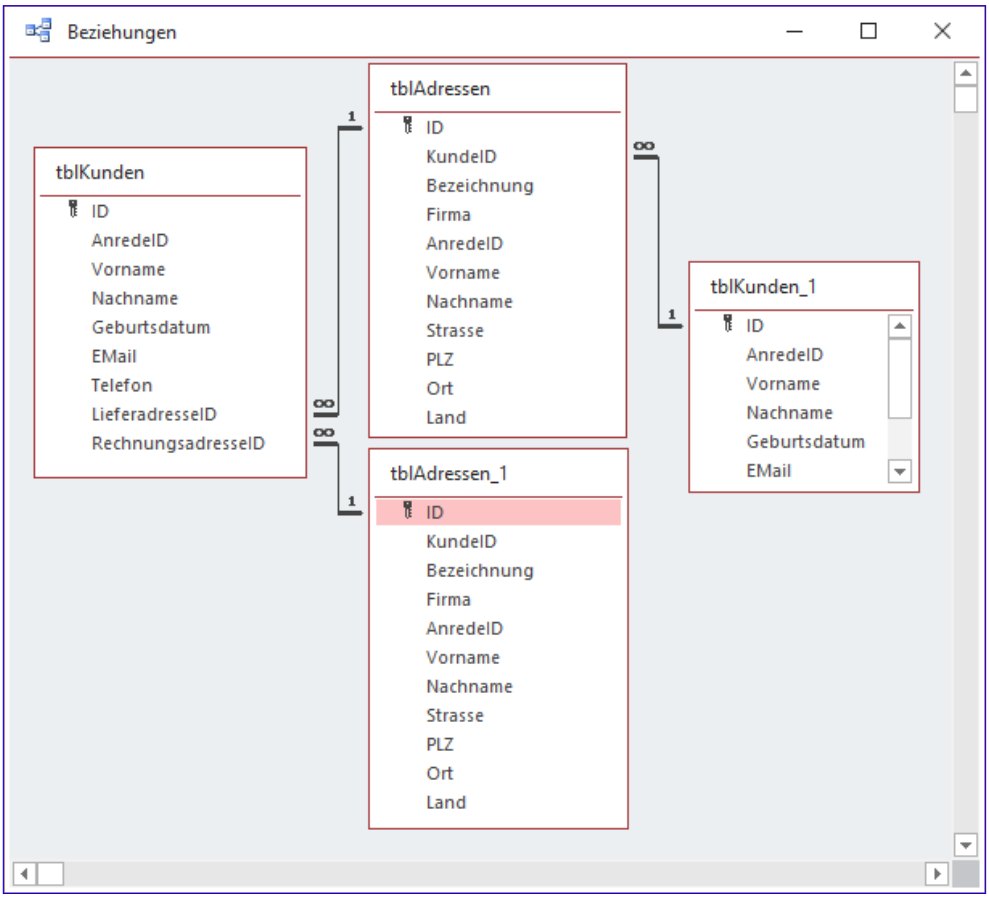

Bild 4: Beziehungen zwischen den Tabellen

Dazu müssen wir der Tabelle **tblKunden** noch weitere Felder hinzufügen, was wir wie in Bild 3 erledigen. Das erste Feld heißt **LieferadresseID** und ist ein Fremdschlüsselfeld zur Auswahl eines der Datensätze der Tabelle **tblAdressen**. Das zweite Feld namens **RechnungsadresseID** dient der Auswahl der für den Rechnungsversand zu verwendenden Adresse.

Beziehungen zwischen den Tabellen

Wir haben nun verschiedene Beziehungen festgelegt, welche die Tabellen **tblKunden** und **tblAdressen** in beide Richtungen miteinander verknüpfen. Die Tabelle **tblAdressen** wird über das Fremdschlüsselfeld **KundeID** mit je einem Datensatz der Tabelle **tblKunden** verknüpft. Die Tabelle **tblKunden** wiederum wird über kein, eines oder zwei der Felder **LieferadresseID** und **RechnungsadresseID** mit der Tabelle **tblAdressen** verknüpft (siehe Bild 4). Um die Beziehungen zwischen den ID-Feldern der Tabelle **tblAdressen** und den beiden Fremdschlüsselfeldern **LieferadresseID** und **RechnungsadresseID** im Beziehungen-Fenster abzubilden, haben wir die Tabelle **tblAdressen** unter dem Namen **tblAdressen_1** ein zweites Mal hinzugefügt. Die Beziehung zwischen dem Feld **KundeID** der hier unter **tblAdressen_1** abgebildeten Tabelle und der Tabelle **tblKunden** lässt sich aus technischen Gründen nicht abbilden, sie ist aber wie für die Tabelle **tblAdressen** dargestellt vorhanden.

Reihenfolge beim Anlegen der Daten

Wenn Sie einen Datensatz in die Tabelle **tblAdressen** eingeben, müssen Sie auch gleich über das Feld **KundeID** festlegen, zu welchem Kunden die Adresse gehört. Daher muss zu diesem Zeitpunkt bereits ein Kundendatensatz vorhanden sein. Die Reihenfolge lautet also: Erst Kundendatensatz anlegen, dann die Adressdatensätze. Um sicherzustellen, dass kein Adressdatensatz gespeichert werden kann, ohne dass ein Kundendatensatz referenziert wurde, stellen wir die Eigenschaft **Eingabe erforderlich** für das Feld **KundeID** der Tabelle **tblAdressen** auf **Ja** ein.

Dies sollten Sie erledigen, bevor Sie Daten in die Tabelle **tblAdressen** eingegeben haben, deren Fremdschlüsselfeld **KundeID** noch leer ist. Datensätze in der Tabelle **tblAdressen** ohne referenzierten Kundendatensatz sind ein Problem, denn sie befinden sich quasi im luftleeren Raum – sie sind keinem Kunden zugeordnet und so nicht zu gebrauchen.

Löschweitergabe bei den Verknüpfungen

Wenn wir mit Verknüpfungen arbeiten, müssen wir uns auch ansehen, welche Regeln wir für das Löschen von Daten festlegen. Das ist einfach: Wenn ein Kundendatensatz aus der Tabelle **tblKunden** gelöscht wird, sollen natürlich auch die mit diesem Kunden verknüpften Adressdatensätze aus der Tabelle **tblAdressen** gelöscht werden. Die Frage ist nur: Über welche Beziehung können wir das festlegen? Die korrekte Antwort ist: Über die Beziehung der Felder **LieferadresseID** und **RechnungsadresseID** der Tabelle **tblKunden** zum Feld **ID** der Tabelle **tblAdressen**.

Im Fenster **Beziehungen bearbeiten** für diese Beziehung stellen wir die Eigenschaft **Löschweitergabe an verwandte Datensätze** zusätzlich zu **Mit referentieller Integrität** ein (siehe Bild 5).

Damit werden nun beim Löschen eines Kunden aus der Tabelle **tblKunden** gleichzeitig alle mit diesem Kundendatensatz verknüpften Adressen gelöscht.

Auswahl der Rechnungs- und Lieferadresse

Wenn wir nun einige Datensätze zur Tabelle **tblKunden** hinzugefügt haben und in der Tabelle **tblAdressen** Adressen angelegt haben, die mit den Einträgen der Tabelle **tblKunden** verknüpft sind, können wir für die beiden Fremdschlüsselfelder **LieferadresseID** und **RechnungsadresseID** die passenden Einträge aus der Tabelle **tblAdressen** auswählen.

Sobald wir zum Beispiel das Nachschlagefeld **LieferadresseID** der Tabelle **tblKunden** aufklappen, finden wir dort

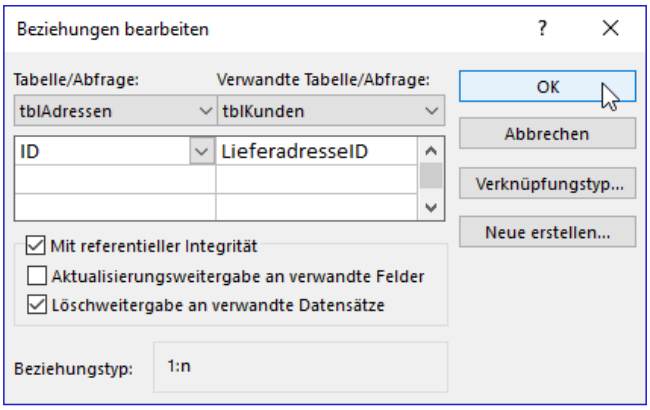

Bild 5: Beziehung zwischen den Tabellen **tblKunden** und **tblAdressen**

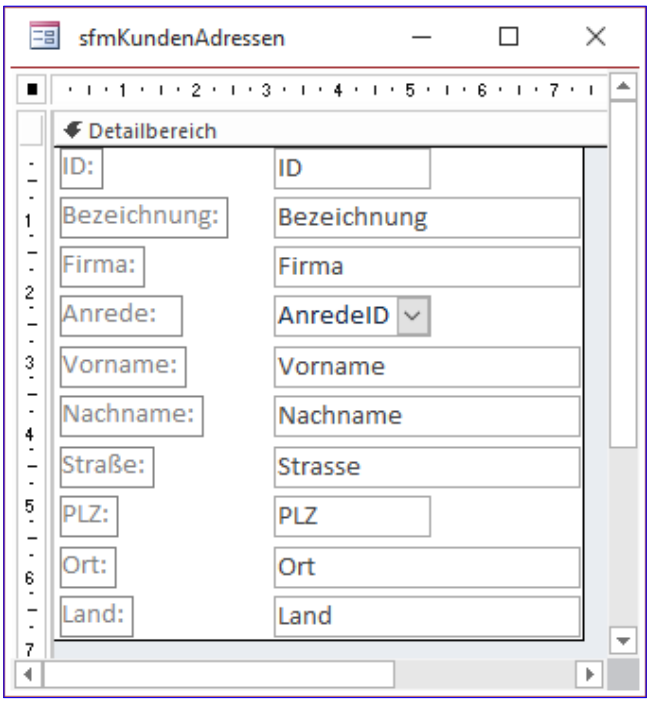

Bild 6: Das Formular **sfmKundenAdressen** in der Entwurfsansicht

nicht nur die Einträge der Tabelle **tblAdressen** vor, die wir dem aktuellen Kunden über das Feld **KundeID** zugewiesen haben, sondern alle Adressen. Wir müssten hier also noch einen Filter einbauen, der nur die zu diesem Kunden passenden Adressen liefert.

Das ist nun allerdings keine Aufgabe mehr, die wir im Datenmodell erledigen können – dazu benötigen wir entsprechende Formulare.

Formulare zur Eingabe von Kunden und Adressen

Wir starten mit einem einfachen Formular namens **frmKunden**, das über die Eigenschaft **Datensatzquelle** an die Tabelle **tblKunden** gebunden ist.

Außerdem ziehen wir alle Felder dieser Tabelle über die Feldliste in den Entwurf des Formulars und ordnen die Felder wie in Bild 6 an.

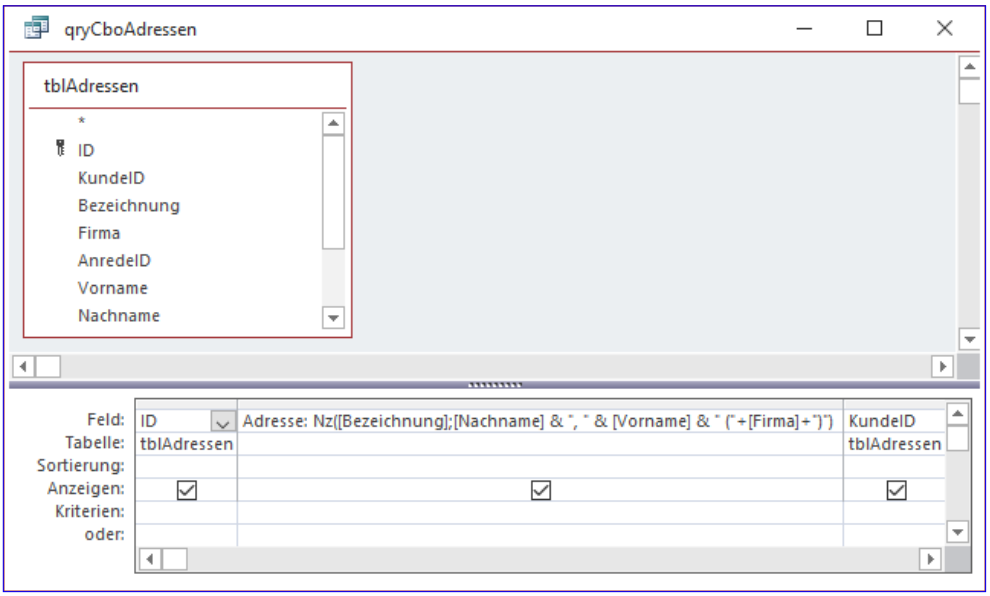

Bild 7: Datensatzherkunft für die Kombinationsfelder zur Auswahl der Liefer- und Rechnungsadresse

Damit haben wir zwar nun die Möglichkeit, über die beiden Kombinationsfelder, die an die Felder **LieferadresseID** und **RechnungsadresseID** gebunden sind, aus allen vorhandenen Adressen auszuwählen.

Aber genau das wollten wir ja verhindern und die dort zur Auswahl angebotenen Datensätze auf die zum Kunden gehörenden Adressen reduzieren.

Dazu passen wir nun die Datensatzherkunft der beiden Kombinationsfelder an, die wir übrigens vorher in **cboLieferadresseID** und **cboRechnungsadresseID** umbenennen.

Dann fügen wir dem ersten der beiden Kombinationsfelder eine Abfrage hinzu, die wir mit dem Abfrage-Generator wie in Bild 7 zusammenstellen.

Wir fügen der Abfrage zunächst das Primärschlüsselfeld der Tabelle hinzu. In der zweiten Spalte legen wir ein Feld namens **Adresse** an, das folgenden Ausdruck, bestehend aus verschiedenen Feldern der Tabelle, liefern soll. Aber nur, wenn das Feld **Bezeichnung**, das der Benutzer für jede Adresse vergeben kann, leer ist:

```
Adresse: Nz([Bezeichnung];[Nachname] & ", " & [Vorname] & " ("+[Firma]+")")
```

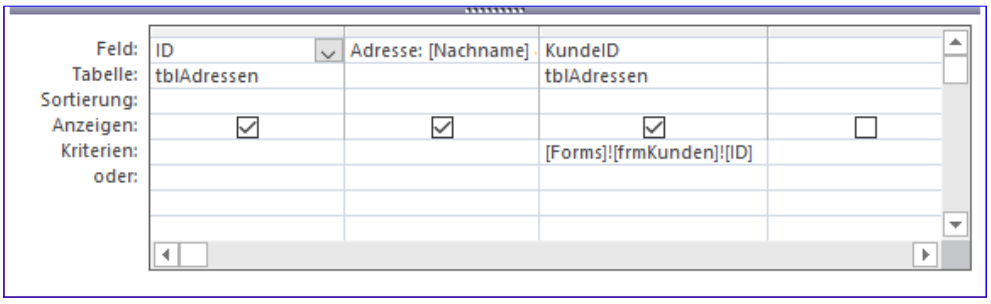

Bild 8: Hinzufügen eines Kriteriums, das nur die zum Kunden gehörenden Adressen liefert

Wir stellen hier einen Ausdruck zusammen, der zunächst prüft, ob das Feld **Bezeichnung** einen Wert enthält. Falls ja, liefert der Ausdruck diesen Wert zurück. Anderenfalls ist das Ergebnis ein Ausdruck mit dem Format **<Nachname>, <Vorname>**

(<Firma>). Für den Fall, dass das Feld **Firma** einmal keinen Wert enthält, haben wir die öffnende und schließende Klammer und das Feld **Firma** mit dem Plus-Operator (**+**) statt mit dem Kaufmanns-Und (**&**) verknüpft. Bei Verwendung des Plus-Operators werden alle verknüpften Elemente zu **Null**, wenn nur eines der Elemente den Wert **Null** hat. Sprich: Wenn **Firma** leer ist, fallen auch die beiden Klammern weg und es werden nur der Nachname und der Vorname der Adresse ausgegeben.

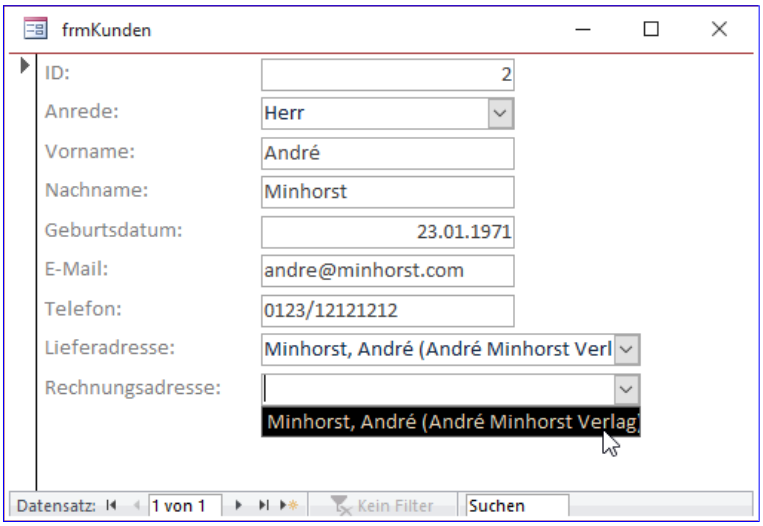

Bild 9: Auswahl einer Rechnungsadresse

Da wir diese Abfrage für beide Kombinationsfelder nutzen können, speichern wir sie gleich unter dem Namen **qryCboAdressen**, sodass sie auch im Navigationsbereich angezeigt wird.

Danach stellen wir diese Abfrage auch als Datensatzherkunft des zweiten Kombinationsfeldes ein. Schließlich müssen wir noch die beiden Eigenschaften **Spaltenanzahl** und **Spaltenbreiten** auf die Werte **2** und **0cm** einstellen, damit nur der aus den Feldern zusammengestellte Ausdruck angezeigt wird, nicht jedoch das Primärschlüsselfeld.

Nun haben wir allerdings immer noch nicht die angezeigten Einträge nach dem aktuell angezeigten Kunden gefiltert. Das erledigen wir, indem wir der Abfrage noch ein Vergleichskriterium für das Feld **KundeID** zuweisen (siehe Bild 8). Dabei beziehen wir uns auf das entsprechende Feld im Formular:

```
[Forms]![frmKunden]![ID]
```

Damit erreichen wir, wenn wir bereits einen Kunden und eine Adresse für diesen Kunden über die Datenblattansicht der Tabellen angelegt haben, zumindest schon einmal die Ansicht aus Bild 9.

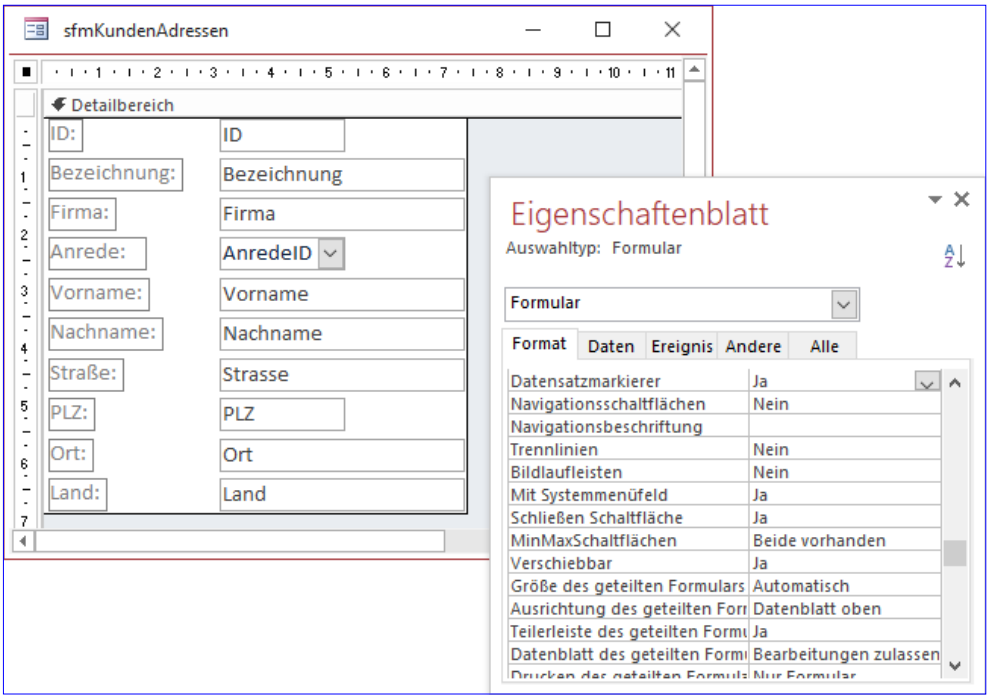

Bild 10: Das Unterformular zur Anzeige der Adressen in der Entwurfsansicht

Adressen vollständig anzeigen

Nun wollen wir dem Benutzer die Adressen nicht

Von Access nach Wordpress

Wordpress ist das gängigste System für die Erstellung und Pflege eines Blogs. Mittlerweile gibt es so viele Erweiterungen, dass ein Blog nur noch eine von vielen Möglichkeiten ist, eine Webseite über die Wordpress-Plattform zu bauen. Da für Access im Unternehmen der Umzug auf ein moderneres System anstand, haben wir uns für Wordpress entschieden. Es bietet gleichzeitig viele Möglichkeiten und ist dennoch recht einfach zu bedienen. Das gilt allerdings auch nur, wenn Sie keine besonderen Anforderungen haben – was in unserem Fall anders aussieht: Immerhin sollen mehr als 1.000 Beiträge samt Bildern und Downloads übertragen werden. Und es darf auch nicht jeder alle Beiträge vollständig betrachten, sondern nur über eine Benutzerverwaltung erfasste Abonnenten. Wie wir das realisiert haben, zeigen wir in diesem Beitrag.

Aus Alt mach Neu

Am Anfang stand die Erkenntnis, dass das im Jahre 2003 aufgesetzte Typo3-System zwar noch klaglos seinen Dienst tut, aber leider nicht der Server, auf dem das Content-Management-System läuft. Bei einem 16 Jahre alten Gerät muss man sich freuen, dass es überhaupt noch läuft. Leider wurden keine regelmäßigen Updates gemacht, sodass es nur mit erheblichem Aufwand möglich gewesen wäre, das System nochmal auf eine aktuellere Version zu übertragen. Also musste ein neues System her, und zwar Wordpress.

Zwei Wordpress-Varianten

Es gibt zwei Wordpress-Varianten: **Wordpress.com** bietet die Software inklusive Hosting an. Sie melden dort eine neue Wordpress-Seite an und bekommen die komplette Infrastruktur gestellt. Dafür gibt es allerdings auch Einschränkungen: Sie können nicht nach Lust und Laune auf Datenbank, Dateien und sonstige Elemente zugreifen, was unseren Spieltrieb und auch die Flexibilität erheblich einschränkte.

Die zweite Wordpress-Variante findet sich auf **Wordpress.org** zum Download und kann auf Ihrem eigenen oder gemieteten Webserver installiert werden. Einige Anbieter bieten auch Pakete an, auf denen alle Voraussetzungen für den Einsatz von Wordpress vorliegen. Hier können Sie je nach den durch den Hoster angebotenen Möglichkeiten auf alle notwendigen Bereiche zugreifen, also auf die Datenbank, das Dateisystem und mehr. Diese Variante haben wir gewählt. Als Hoster nutzen wir **aixpro.de**.

Voraussetzungen

Schon unter Verwendung des in die Jahre gekommen Typo3-Systems sah der Workflow für die Veröffentlichung der Beiträge wie folgt aus: Die Beiträge wurden in InDesign geschrieben und gesetzt.

Von dort haben wir die PDFs für den Drucker erstellt und auch die für den Downloadbereich für registrierte Abonnenten. Außerdem haben wir daraus per VBA eine HTML-Version erstellt, die wir in einer Access-Datenbank gespeichert haben.

Es gibt eine Tabelle für die Beiträge namens **tblBeitraege**, eine Tabelle für die Kategorien (**tblKategorien**) sowie eine Verknüpfungstabelle, damit wir den Kategorien die Beiträge zuweisen konnten (**tblBeitraegeKategorien**). Dies sind die wesentlichen Tabellen, die wir für die Übertragung der Inhalte in das Wordpress-System benötigen. In der Beispieldatenbank haben wir diese Tabellen mit ein paar Beispielartikeln im HTML-Format untergebracht.

Wichtig ist noch zu wissen, dass die HTML-Version mit Verweisen auf Bilder versehen sind, die sich in einem speziellen Unterverzeichnis befinden. Dieses kann man also so, wie es ist, auf den Webserver legen.

Und auch die Beispieldatenbanken und sonstige Downloaddateien liegen in einer bestimmten Verzeichnisstruktur vor, die man problemlos so auf den Server übertragen kann. Hier sind nur die Verlinkungen in den Beiträgen noch anzupassen.

Eine weitere Voraussetzung ist natürlich das Vorhandensein einer Wordpress-Installation, auf deren SQL-Datenbank Sie auf irgendeinen Weg Zugriff haben – mehr dazu weiter unten.

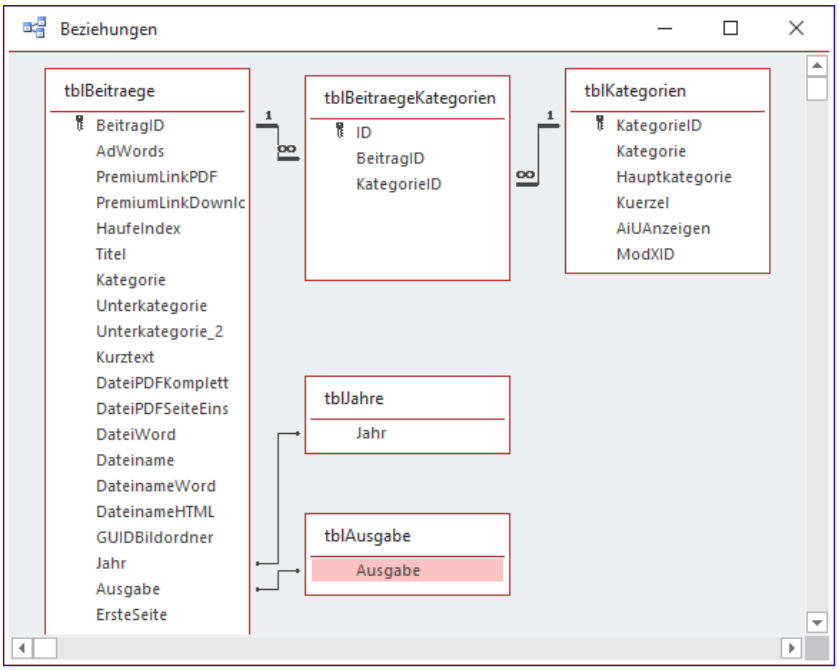

Bild 1: Datenmodell der Access-Anwendung

Ins HTML-Format

Wenn Sie Beiträge aus einer Datenbank in eine Wordpress-Anwendung übertragen wollen, ist es sinnvoll, wenn diese Daten bereits im HTML-Format in der Tabelle **tblBeitraege** gespeichert sind. Aber woher überhaupt kommen die Beiträge, die Sie mit der Datenbank verwalten und nach Wordpress überführen wollen? In meinem Fall kommen diese, wie oben beschrieben, aus InDesign. InDesign bietet die Möglichkeit, Dateien in das HTML-Format zu exportieren. Das Ergebnis dieses Exports war jedoch für meine Zwecke nicht passend, sodass ich den Export selbst programmiert habe.

Da vermutlich nicht viele Entwickler mit InDesign arbeiten werden, hier nur eine kurze Zusammenfassung: InDesign bietet eine VBA-Bibliothek für den VBA-gesteuerten Zugriff auf das Objektmodell. Auf dieser Basis habe ich einen Satz von Routinen geschrieben, die das InDesign-Dokument Absatz für Absatz durchlaufen und prüfen, welches Absatzformat vorliegt, und einen entsprechenden HTML-Absatz daraus erstellen. Aus einem Absatz mit dem Absatzformat Fliesstext wird dann etwa ein Text, der von den entsprechenden HTML-Tags eingefasst wird, beispielsweise ** ... **.

Die Programmierung geht jedoch über diese recht einfache Vorgehensweise hinaus, denn die Texte enthalten auch Verweise auf Bilder oder Listings in Kästen. Diese werden von den Routinen identifiziert und entsprechend behandelt. Bei einem Bild, das im InDesign-Satz unabhängig vom Fließtext in einem Kasten an beliebiger Stelle im Dokument platziert ist, landet etwa das ****-Element direkt hinter dem ** ... **-Absatz, der im Originaldokument auf das Bild verweist.

Gleiches gilt für Kästen mit Listings. Die Bilder im InDesign-Dokument werden ähnlich wie in Word über Verknüpfungen angezeigt. Diese Verknüpfungen können wir auch über VBA auslesen und so den Pfad der angezeigten Bilder ermitteln.

Die Bilder aus dem Originaldokument kopiere ich dann ebenfalls per VBA in ein Verzeichnis, das alle Bilddateien für das entsprechende Dokument enthält.

Damit erhalte ich also für jeden Artikel einen Datensatz in der Tabelle **tblBeitrage**, der den Titel des Artikels in einem Feld enthält, außerdem die Ausgabe, das Erscheinungsjahr und den HTML-Code in einem Memofeld.

Die Kategorie des Artikels füge ich dann selbst über die Benutzeroberfläche hinzu. Und auch die Ausgabe und das Erscheinungsjahr wähle ich im Formular für die importierten Artikel aus. Damit ergibt sich dann das Datenmodell aus Bild 1.

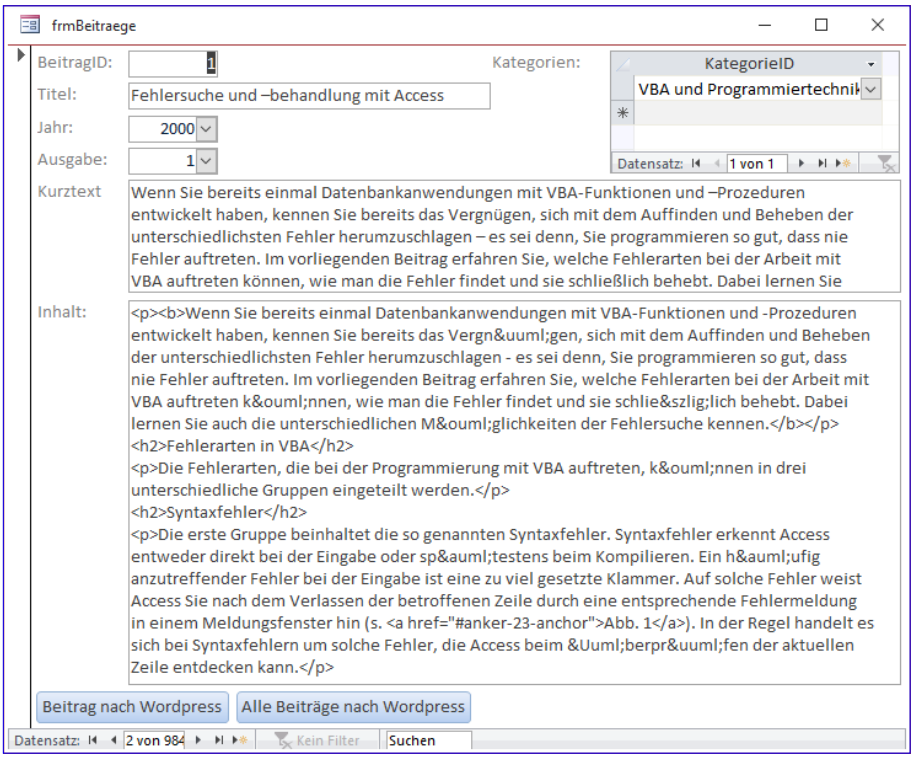

Bild 2: Formular mit den Beiträgen und den Schaltflächen zum Übertragen nach Wordpress

Formular zur Anzeige der Beiträge

Um die Prozeduren zum Erstellen der Wordpress-Daten aufrufen zu können, erstellen wir uns noch ein Formular namens **frmBeitraege**, mit dem wir die einzelnen Datensätze der Tabelle **tblBeitraege** anzeigen (siehe Bild 2). Aus dieser Tabelle zeigen wir dabei die Werte der Felder **BeitragID**, **Titel**, **Jahr**, **Ausgabe**, **Kurztext** und **Inhalt** an.

Außerdem liefert ein Unterformular namens **sfmKategorien** die Daten der Verknüpfungstabelle **tblBeitraegeKategorien**, genau genommen die des Feldes **KategorieID**. Damit können wir die zugeordneten Kategorien auswählen.

Unten im Formular legen wir zwei Schaltflächen an. Die erste heißt **cmdBeitragNachWordpress**. Sie soll den aktuell angezeigten Beitrag behandeln. Die zweite namens **cmdAlleNachWordpress** soll sich um alle Beiträge kümmern, die sich in der Tabelle **tblBeitraege** befinden.

Was aber heißt »behandeln« oder »kümmern« in diesem Fall? Das müssen wir noch klären.

Übertragen der Artikel nach Wordpress

Es gibt verschiedene Möglichkeiten, wie wir die HTML-Artikel nach Wordpress übertragen. Wordpress hat logischerweise ebenfalls eine Datenbank, welche die enthaltenen Artikel und weitere Daten speichert. Theoretisch könnten wir die Tabellen dieser meist unter MySQL verwalteten Datenbank in unsere Access-Datenbank einbinden und so direkt auf diese Tabellen zugreifen. Aus Sicherheitsgründen bieten die meisten Provider dies jedoch nicht an.

Das Übertragen von Daten aus den Tabellen einer lokalen Tabelle in die verknüpften Tabellen einer Datenbank auf einem Webserver ist aber auch nicht besonders herausfordernd.

Also gehen wir direkt den alternativen Weg, der von den meisten Providern angeboten werden sollte: die Möglichkeit, SQL-Skripte auszuführen. Das können Sie, wenn Sie kompletten Zugriff auf den Server etwa über eine Konsole

haben, erledigen, indem Sie die Datei mit den entsprechenden SQL-Anweisungen auf den Server kopieren und diese dort in der Konsole aufrufen.

Unser Provider **Aixpro** stellt uns eine komfortable Benutzeroberfläche für die Verwaltung unserer Webseiten zur Verfügung stellt.

Die Benutzeroberfläche heißt **Plesk** und bietet die Möglichkeit, die Webseiten zu administrieren, Dateien zu verwalten und ein Tool für den Zugriff auf die SQL-Datenbanken zu öffnen, nämlich **phpMyAdmin**.

Diese Tools sind wirklich sehr hilfreich, wenn es um die Verwaltung von Webanwendungen geht. phpMyAdmin bietet die Möglichkeit, SQL-Skripte als Datei hochzuladen und diese direkt in der aktuell angezeigten Datenbank auszuführen.

Unser Ziel ist es also auf jeden Fall, eine Datei mit den auszuführenden SQL-Anweisungen zu erstellen, die wir dann auf dem Webserver laden und dort ausführen können. Dies ist dann auch der kleinste gemeinsame Nenner für die verschiedenen Konfigurationsmöglichkeiten für eine Wordpress-Installation.

Bild 3: Datenmodell von Wordpress

Die relevanten Tabellen von Wordpress

Wordpress hat in der Basisinstallation, also ohne zusätzliche Plugins, ein recht überschaubares Datenmodell. Die für uns interessanten Tabellen befinden sich auf der

Übersicht des Datenmodells, die wir der Webseite **https://codex.wordpress.org** entnommen haben, im markierten Bereich (siehe Bild 3).

Die Tabelle **wp_posts** enthält die einzelnen Artikel. Das Feld **postTitle** nimmt dabei den Titel auf, das Feld **post_content** (im Screenshot **post_content_filtered** benannt) nimmt den Artikel selbst auf. Die Kategorien können wir in der Tabelle **wp_Terms** hinterlegen. Die Tabelle **wp_term_taxonomy** weist den Kategorien aus **wp_terms** eine Funktion zu, nämlich Kategorie, Link oder Tag, und definiert die Hierarchie der Einträge der Tabelle **wp_Terms** – das ist für die Erschaffung einer Kategorien-Hierarchie wichtig.

Die Hierarchie wird über das Feld **parent** erzeugt, das auf das Primärschlüsselfeld **term_taxonomy_id** der gleichen Tabelle verweist. Die Tabelle **wp_term_relationsships** verknüpft wiederum die Einträge der Tabelle **wp_term_taxonomy** und die Artikel aus der Tabelle **wp_posts** miteinander. Für uns ist diese Tabelle also wichtig, weil wir über diese die Artikel den Kategorien zuweisen können, die wir in der Tabelle **wp_terms** definiert haben.

Zu beachten

Bevor wir die Daten einfach von Access in die Wordpress-Tabellen übertragen, ist etwas unter Umständen sehr wichtiges zu beachten – zumindest war das bei meinem Projekt der Fall. Die Artikel existierten ja zuvor bereits auf einer anderen Webseite, in diesem Fall einer auf Basis des CMS-Systems Typo3. Hier sind die Links auf die Artikel wie folgt aufgebaut:

```
http://www.access-im-unternehmen.de/index1.
php?id=300&BeitragID=1
```

Dabei war die einzige Variable die Zahl für den Parameter **BeitragID**. Wenn man eine Webseite neu aufbaut, die bei Google bereits ein gutes Ranking erreicht hat, sollte man sicherstellen, dass die Adresse, unter welcher die gerankten Artikel erreichbar sind, gleich bleibt. Natürlich gibt es technische Möglichkeiten, dafür zu sorgen, dass wenn der Benutzer einen Google-Link auf die Seite anklickt, auch die auf dem neuen Server unter einer anderen Adresse gespeicherte Seite angezeigt wird. Unter Wordpress sehen die Links normalerweise wie folgt aus:

```
http://www.access-im-unternehmen.de/?p=1
```

Man kann nun auf dem Server bestimmte Regeln einrichten, nach denen etwa der obige, alte Link in den neuen Link umbenannt wird – das ist allerdings ein Thema, das wir in diesem Beitrag nicht mehr anreißen wollen.

Für uns ist nur interessant, dass es zum Definieren einer solchen Regel einfach ist, wenn wir den dynamischen Teil der alten Adresse, hier also den Parameter **BeitragID**, auslesen und diesen dann in Form des Parameters **p** für den Link zur neuen Webseite nutzen können.

Mein Plan war also zunächst, die Artikel in der Tabelle **wp_posts** genauso zu nummerieren wie in der Tabelle **tblBeitraege**. So wäre es dann am einfachsten, per URL auf die Artikel zuzugreifen.

Das hat auch zu Beginn gut geklappt, nämlich als die Wordpress-Datenbank und speziell die Tabelle **wp_posts** noch komplett leer war. Dann allerdings habe ich zum Beispiel einen Beitrag bearbeitet oder Menüeinträge hinzugefügt.

Und sowohl für die Bearbeitungsstände von Wordpress-Artikeln als auch für Menüeinträge werden neue Datensätze in der Tabelle **wp_posts** angelegt. Ich hatte zwar nun die Datensätze mit den Primärschlüsselwerten bis etwa 1.200 mit Beiträgen gefüllt und jeder dieser Artikel hatte den seiner ID entsprechenden Primärschlüsselwert.

Die angelegten Menüs und die Bearbeitungsstände der editierten Artikel haben dann allerdings die Primärschlüsselwerte ab 1.200 in Beschlag genommen – die nächsten Werte wie 1.201, 1.202 und so weiter waren also belegt.

Neue Beiträge hätte ich dann also mit Lücken in den ID-Werten hinzufügen müssen. Das war erstens ein Problem, weil ich die Beitrag-IDs zumindest einigermaßen lückenlos halten wollte und zweitens die Beiträge ja bereits weit vor der Veröffentlichung in der Artikeldatenbank angelegt wurden und somit bereits feste ID-Werte erhalten haben. Also sollten diese dann auch unter den vergebenen ID-Werten in der Tabelle **wp_posts** landen.

Es musste also eine alternative Lösung her, die zukunftssicher war und gleichzeitig die IDs der Beiträge im Primärschlüsselfeld enthielt. Also habe ich mich entschieden, einfach eine recht große Zahl zu den IDs hinzuzuaddieren – in diesem Fall 55.000.000 – und somit für alle Zeiten genügend freie IDs für alle sonstigen Zwecke vorzuhalten. Der Beitrag mit der ID **1.201** würde also in der Tabelle **wp_posts** unter dem Primärschlüsselwert **55.001.201** abgespeichert werden.

Sie mögen jetzt einwerfen, dass man dann nicht mehr so einfach über einer URL wie **www.access-im-unternehmen.de/1201** auf den Beitrag zugreifen kann. Das hätte natürlich gravierende Folgen, denn wir haben diese Nummern in jedem Beitrag in den gedruckten Heften angegeben, damit die Leser darunter die Downloads mit den Beispieldatenbanken zu dem jeweiligen Beitrag beziehen können.

Das ist aber auch kein Problem: Wir haben ja bereits beschrieben, dass wir auf dem Server definieren können, dass URLs geändert werden können. Und so lässt sich auch aus **www.access-im-unternehmen.de/1201** relativ leicht **www.access-im-unternehmen.de/?p=1201** erzeugen. Der Leser soll dann also sowohl über den im Heft angegebenen Link **www.access-im-unternehmen.de/1201** als auch über die bei Google gespeicherten URLs wie **www.access-im-unternehmen.de/index1.php?id=300&BeitragID=1201** auf die Beiträge zugreifen können. Und das können wir durch entsprechende Regeln auch erreichen, wenn die ID eigentlich **55001201** lautet.

Löschen und neu anlegen

Gerade in der Testphase wird es oft vorkommen, dass Sie die Datensätze in der Wordpress-Datenbank löschen und neu anlegen wollen. Wir haben dies in den folgenden Prozeduren berücksichtigt.

Da wir nicht nur für die Beiträge Primärschlüsselwerte in einem bestimmten Bereich verwenden wollen (ab 55.000.001), sondern auch für die Kategorien in der Tabelle **wp_terms** (ab 88.000.001) und für die ebenfalls dort definierten Jahre (ab 99.001) und die untergeordneten Ausgaben (ab 99.000.001), können wir die Datensätze jeweils auch wieder entfernen, damit wir sie anschließend jederzeit wieder reproduzieren können.

Das ist aber nicht der alleinige Grund dafür, dass wir auch Kategorien, Ausgaben und Jahre mit solch hohen Werten gefüllt werden sollen. Der nächste Grund ist, dass wir später noch eine Menüstruktur auf Basis der Kategorien, Ausgaben und Jahre aufbauen wollen. Und damit wir diese nicht immer wieder neu erstellen müssen, wenn wir die Kategorien, Ausgaben und Jahre neu hinzugefügt haben, verwenden wir für diese genau wie für die Beiträge immer die gleichen ID-Werte.

Beiträge und Kategorien nur per Access

Damit dies funktioniert, dürfen Sie Änderungen an Beiträgen und Kategorien natürlich nur über die Datenbank durchführen. Jede Änderung, die Sie über den Editor von Wordpress durchgeführt haben, wird ja sonst bei einem erneuten Einspielen der Daten der Access-Datenbank wieder überschrieben.

Prozedur zum Übertragen der Beiträge

Beginnen wir mit der Prozedur, die durch die beiden Ereignisprozeduren des Formulars aufgerufen wird. Diese beiden Ereignisprozeduren sehen wie folgt aus:

```
Private Sub cmdAlleNachWordpress_Click()
    WordpressExport
End Sub
```

```
Public Sub WordpressExport(Optional lngBeitragID As Long)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim strSQL As String, strSQLDelete As String, strDateiname As String
    Dim strInhalt As String, strTitel As String
    Dim strKurztext As String, strWhere As String
    strDateiname = CurrentProject.Path & "\SQLBeitraege.sql"
    On Error Resume Next
    Kill strDateiname
    On Error GoTo 0
    Set db = CurrentDb
    If Not lngBeitragID = 0 Then
        strWhere = "tblBeitraege.BeitragID = " & lngBeitragID & " AND NOT " & cstrFeldInhalt & " IS NULL"
    Else
        strWhere = "NOT " & cstrFeldInhalt & " IS NULL"
    End If
    Set rst = db.OpenRecordset("SELECT * FROM tblBeitraege WHERE " & strWhere, dbOpenDynaset)
    Open strDateiname For Append As #1
    Do While Not rst.EOF
        strSQLDelete = "DELETE FROM wp_posts WHERE ID = " & 55000000 + rst!BeitragID & ";" & vbCrLf
        DoEvents
        Print #1, strSQLDelete
        rst.MoveNext
    Loop
    rst.MoveFirst
    Do While Not rst.EOF
        strInhalt = rst(cstrFeldInhalt)
        strTitel = rst!Titel
        strTitel = Ersetzen(strTitel)
        strKurztext = rst!Kurztext
        strKurztext = Ersetzen(strKurztext)
        strSQL = "INSERT INTO wp_posts(ID, post_author, post_date, post_date_gmt, post_content, post_title, " _
            & "post_excerpt, post_status, comment_status, ping_status, post_name, post_parent, guid, menu_order, " _
            & "post_type, comment_count) VALUES(" & 55000000 + rst!BeitragID & ", 1, '" _
            & format(DateSerial(rst!Jahr, rst!Ausgabe * 2, 1), "yyyy-mm-dd hh:nn:ss") & "', '" & format(Now, _
            "yyyy-mm-dd hh:nn:ss") & "', '" & strInhalt & "', '" & strTitel & "', '" & strKurztext _
            & "', 'publish', 'open', 'open', '" & Replace(Replace(rst!Titel, " ", "_"), "'", "''") & "', " _
            & "0, 'http://access-im-unternehmen.aix-dev.de/aiu/?p=" & rst!BeitragID & "', 0, 'post', 0);" & vbCrLf
        Print #1, strSQL
        DoEvents
        rst.MoveNext
    Loop
    Print #1, KategorienArtikel(strWhere)
    Print #1, KategorienAusgaben(strWhere)
    Close #1
End Sub
```

Listing 1: Prozedur zum Erstellen des SQL-Skripts für die Beiträge

Backendkopie zum Bearbeiten holen

Ein Kunde hatte die Anforderung, dass das Backend grundsätzlich auf dem Server liegen sollte und der Benutzer vom Frontend auf seinem Rechner per Netzwerk vom Homeoffice darauf zugreift. Wegen instabiler Internetverbindung soll das Backend aber während der Bearbeitung besser auf den lokalen Rechner kopiert werden und nach dem Schließen der Anwendung wieder zurück auf den Server. Wie man dies realisieren kann und was dabei zu beachten ist, zeigen wir in diesem Beitrag.

Voraussetzung

Wir gehen an dieser Stelle davon aus, dass das Verzeichnis, in dem sich das Backend auf dem Server befindet, als Netzlaufwerk in den lokalen Rechner eingebunden ist, sodass das Backend mit den üblichen VBA-Methoden zum Kopieren von Dateien auf den lokalen Rechner und zurück zum Server kopiert werden kann.

Planung

Naiv betrachtet könnte man die Aufgabe herunterbrechen in folgende Schritte:

- Beim Öffnen der Anwendung wird das Backend auf den lokalen Rechner kopiert und die Tabellen werden in das Frontend eingebunden.
- Bei Schließen der Anwendung wird das Backend zurück auf den Server kopiert.

Das setzt jedoch voraus, dass das Fronted immer ordnungsgemäß geschlossen wird und niemals abstürzt. Wenn das Frontend einmal abstürzt, würde es beim erneuten Start das Backend erneut auf den lokalen Rechner kopieren und die Tabellen einbinden. Das wird zum Problem, sobald der Benutzer vor dem Absturz bereits Daten geändert hat.

Beim Starten wird nämlich dann die alte Version des Backends vom Server geladen und eingebunden und die vor dem Absturz durchgeführten Änderungen sind nicht mehr verfügbar.

Dem müssen wir Rechnung tragen, indem wir beispielsweise in einer Optionen-Tabelle festhalten, wann das Backend vom Server auf den lokalen Rechner kopiert wurde und umgekehrt.

Dann können wir beim Start der Anwendung prüfen, ob das letzte Kopieren auf den Server vor oder nach dem letzten Kopieren vom Server auf den lokalen Rechner stattgefunden hat und damit ermitteln, ob die Anwendung beendet wurde, ohne dass das Backend auf den Server kopiert wurde.

Erster Schritt: Backend erstellen

Sollten Sie eine solche Lösung ausgehend von einer Datenbank erstellen, deren Tabellen und Benutzeroberfläche noch in einer einzigen Datenbankdatei gespeichert sind, müssen Sie zunächst ein Backend erstellen und die Tabellen, die später auf dem Server liegen sollen, in das Backend verschieben. Danach löschen Sie diese Tabellen aus dem Frontend und erstellen entsprechende Verknüpfungen auf die Tabellen des Backends. Als Beispiel verwenden wir die Datenbank zum flexiblen Verwalten von Adressen aus dem Beitrag **Flexible Adressen** (**www.access-im-unternehmen.de/1214**). Diese enthält eine überschaubare Anzahl von Tabellen.

Aufteilen der Datenbank

Das Aufteilen der Datenbank lassen wir bequem von dem dafür vorgesehenen Assistenten erledigen. Diesen starten Sie über den Ribbon-Befehl **Datenbanktools|Daten verschieben|Access-Datenbank** (siehe Bild 1).

Auf der ersten Seite des so gestarteten Assistenten klicken Sie auf die Schaltfläche **Datenbank aufteilen** und wählen im nun erscheinenden Dialog **Back-End-Datenbank erstellen** den Pfad zu der zu erstellenden Backend-Datenbank aus. Wir geben hier einfach den Namen der aufzuteilenden Datenbank mit dem Suffix **_BE** an, also **BackendKopieren_BE.accdb** (siehe Bild 2).

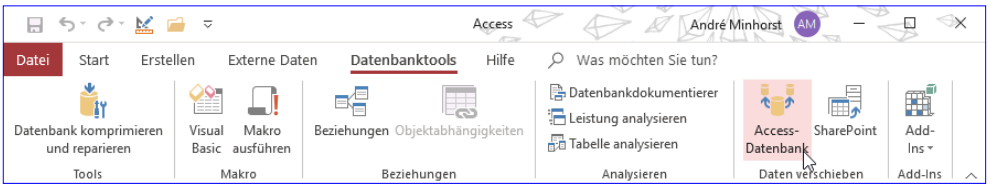

Bild 1: Aufteilen einer Datenbank

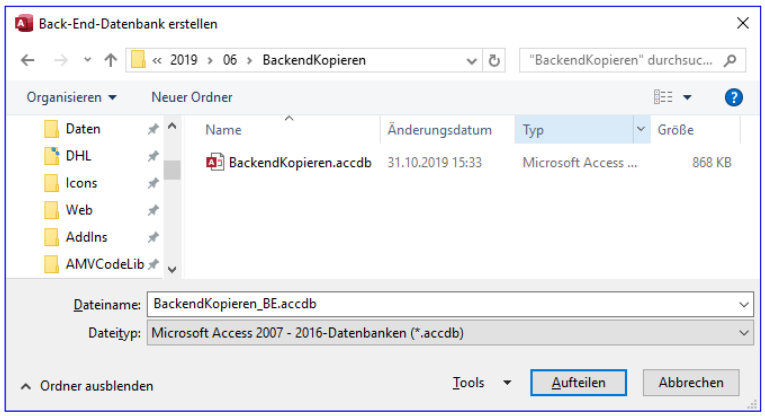

Bild 2: Auswahl der Zieldatenbank

Bei der hier verwendeten Anzahl Tabellen schließt der Assistent den Vorgang recht zügig ab.

Die Änderungen im Frontend sind überschaubar: Statt der bisher vorhandenen lokalen Tabellen finden sich im Navigationsbereich nun die gleichen Tabellen, allerdings mit dem Verknüpfungssymbol (siehe Bild 3).

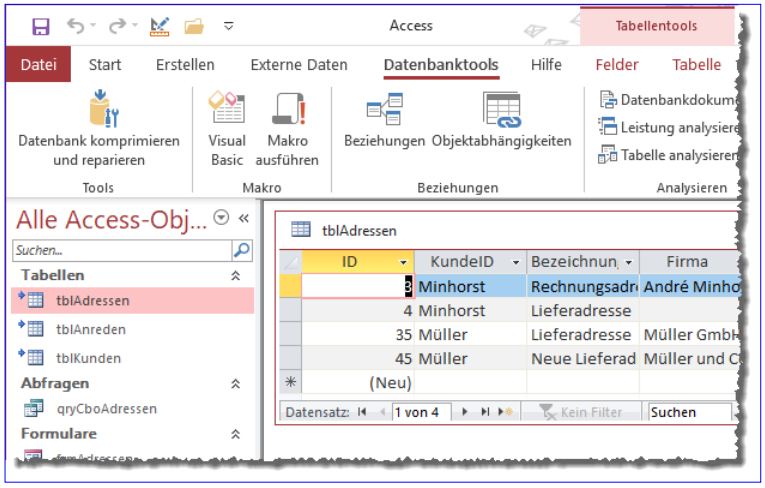

Bild 3: Das Frontend nach der Aufteilung

Wenn Sie diese öffnen, zeigen Sie die Daten allerdings genau so an, als ob Sie eine lokale Tabelle verwenden. Auch das Bearbeiten der enthaltenen Daten geschieht wie gewohnt.

Verknüpfungen löschen und neu erstellen oder aktualisieren?

Nun stellt sich die Frage: Wollen wir die Verknüpfungen löschen, bevor wir die Backend-Datenbank, die ja aktuell im gleichen Verzeichnis wie die Frontend-Datenbank liegt, auf den Server verschieben und diese neu anlegen, wenn das Backend beim nächsten Öffnen des Datenbankfrontends vom Server geholt wird? Oder reicht es aus, die Verknüpfungen zu aktualisieren? Wenn Sie das Frontend immer vom gleichen Verzeichnis aus starten und sich das Backend ebenfalls immer in diesem Verzeichnis befindet, ist das grundsätzlich nicht nötig, wenn nur das Backend ausgetauscht wird. Allerdings kann es ja einmal vorkommen, dass der Benutzer die Frontend-Datenbank auf einen anderen Rechner kopiert oder sich der Verzeichnisname ändert. Dann müssen die Verknüpfungen doch aktualisiert werden.

Wir begnügen uns mit dem Aktualisieren aller in der Frontend-Datenbank enthaltenen Verknüpfungen – so kann man auch einmal eine neue Tabelle zum Backend hinzufügen und braucht diese nur manuell zu verknüpfen, damit diese Verknüpfung bei folgenden Starts der Anwen-

```
Public Sub VerknuepfungenAktualisieren()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Set db = CurrentDb
    For Each tdf In db.TableDefs
        If Not Len(tdf.Connect) = 0 Then
            tdf.Connect = ";DATABASE=" & CurrentProject.Path & "\BackendKopieren_BE.accdb"
            tdf.RefreshLink
        End If
    Next tdf
End Sub
```

Listing 1: Verknüpfungen aktualisieren

dung in das Aktualisieren aller vorhandenen Verknüpfungen integriert wird.

Verknüpfungen per VBA aktualisieren

Das Aktualisieren der verknüpften Tabellen erledigen wir mit der Prozedur aus Listing 1. Diese deklariert ein **Database**- und ein **TableDef**-Objekt und weist der **Database**-Variablen **db** das **Database**-Objekt der aktuellen Datenbank zu. Dann durchläuft es alle **TableDef**-Objekte der Auflistung **TableDefs** des **Database**-Objekts und referenziert diese jeweils mit der Variablen **tdf**.

Dabei prüft sie zunächst anhand der Eigenschaft **Connect** des **TableDef**-Objekts, ob es sich bei der Tabelle um eine lokale oder eine verknüpfte Tabelle handelt. Enthält diese Eigenschaft einen Wert, handelt es sich um ein verknüpftes Objekt. In diesem Fall stellen wir die Eigenschaft auf einen Wert ein, der aus **;DATABASE=**, dem Pfad der Frontend-Datenbank, einem Backslash sowie dem angenommenen Namen der Backend-Datenbank besteht, also zum Beispiel folgenden Ausdruck:

```
;DATABASE=C:\...\BackendKopieren\BackendKopieren_BE.accdb
```

Danach aktualisieren wir die Verknüpfung mit der **RefreshLink**-Methode.

Aktualisieren der Verknüpfungen testen

Um die Verknüpfungen nach der Aktualisierung zu testen, verschieben Sie die beiden Datenbanken beispielsweise in ein Unterverzeichnis namens **Test**. Wenn Sie dann die Frontend-Datenbank öffnen und versuchen, eine der Tabellen über die Verknüpfung zu öffnen, erhalten Sie die Fehlermeldung aus Bild 4. Logisch: Die Verknüpfugn zeigt über die **Connect**-Eigenschaft noch auf den alten Speicherort der Backend-Datenbank.

Erst nachdem Sie nun die Prozedur **VerknuepfungenAktualisieren** aufrufen, funktioniert das Öffnen der verknüpften Tabellen wieder.

Übrigens reicht es nicht aus, Frontend und Backend einfach nur in ein anderes Verzeichnis zu kopieren, um den Fehler beim Zugriff auf eine der verknüpften Tabellen auszulösen. Genau genommen resultieren aus dieser Aktion oft Verwirrungen, weil man über das Frontend vermeintlich Daten der verknüpften Tabellen im neuen Verzeichnis vornimmt. Tatsächlich greift das Frontend aber, wenn Sie die Datenbankdateien nur kopieren und nicht verschieben (beziehungsweise die Datenbankdateien am

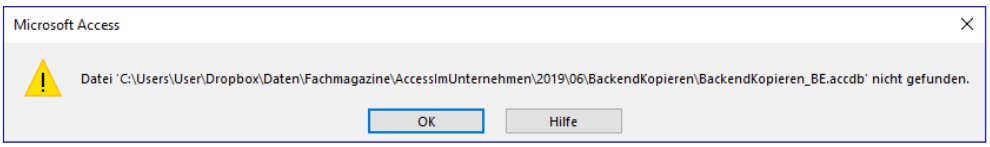

Bild 4: Fehler beim Versuch, eine verknüpfte Tabelle nach dem Verschieben zu öffnen

alten Speicherort löschen) noch auf das Backend im alten Speicherort zu.

Optionen für das Kopieren

Nun wollen wir das Backend bei jedem Öffnen der Anwendung neu vom Server in das lokale Verzeichnis kopieren und die Verknüpfung aktualisieren. Dazu muss die entsprechende Prozedur wissen, unter welchem Pfad sie auf die auf dem Server liegende Datei zugreifen kann. Dazu wollen wir eine Optionen-Tabelle namens **tblOptionen** anlegen. Neben einem Feld namens **Backendpfad** wollen wir noch weitere Informationen in dieser Tabelle speichern. Wir müssen nämlich in irgendeiner Form festhalten, ob das Backend zuletzt vom Server auf das lokale Verzeichnis oder vom lokalen Verzeichnis auf den Server kopiert wurde.

Wenn das Backend zuletzt vom lokalen Verzeichnis auf den Server kopiert wurde, ist das offensichtlich beim vorherigen Schließen der Anwendung geschehen. Wenn hingegen beim Öffnen des Frontends die Information vorliegt, dass der letzte Kopiervorgang vom Server zum lokalen Verzeichnis erfolgte, dann wurde das Frontend bei der letzten Verwendung offensichtlich nicht korrekt geschlossen.

In diesem Fall wollen wir prüfen, ob das auf dem Server oder das im lokalen Verzeichnis gespeicherte Backend ein aktuelleres Bearbeitungsdatum hat. Ist das Backend im lokalen Verzeichnis neuer, soll der Benutzer gefragt werden, ob das neuere Backend weiter genutzt werden soll oder ob das ältere Backend vom Server über das vorhandene Backend kopiert werden soll.

Wie speichern wir diese Information ab? Wir könnten zwei Felder definieren, die wir etwa **ZuletztVomServerKopiert** und **ZuletztZumServerVerschoben** nennen und die den Datentyp **Datum/Uhrzeit** erhalten. Oder wir verwenden einfach ein **Boolean**-Feld etwa namens **ZuletztVomServerKopiert**. Wenn es den Wert **True** enthält, wurde das Backend zuletzt vom Server in das lokale Verzeichnis kopiert, im Falle von **False** wurde es vom lokalen Verzeichnis zum Server verschoben.

Die Datumsangaben zu speichern, ist schon interessant, da wir dem Benutzer im Falle eines unterwarteten Schließens zusätzliche Informationen liefern können, wann das Backend zuletzt wohin kopiert oder verschoben wurde – also legen wir zwei Felder namens **ZuletztVomServerKopiert** und **ZuletztZumServerVerschoben** an.

Die Tabelle **tblOptionen** sieht damit wie in Bild 5 aus.

Verwalten der Optionen der Anwendung

Um den Pfad zur Backenddatenbank festlegen zu können, fügen wir der Datenbank ein Formular namens **frmOptionen** hinzu, das wir über die Eigenschaft **Datensatzquelle** an die Tabelle **tblOptionen** binden. Wir ziehen alle Felder dieser Tabelle aus der Feldliste in den Detailbereich des

tblOptionen

Feldname	Felddatentyp	Beschreibung (optional)
BackendPfad	Kurzer Text	Pfad des Backends auf dem Server
ZuletztVomServerKopiert	Datum/Uhrzeit	Datum und Uhrzeit, wann das BE zuletzt vom Server kopiert wurde.
ZuletztZumServerVerschoben	Datum/Uhrzeit	Datum und Uhrzeit, wann das BE zuletzt zum Server verschoben wurde.

Feldeigenschaften

Allgemein | Nachschlagen

Feldgröße	255
Format	
Eingabeformat	
Beschriftung	
Standardwert	
Gültigkeitsregel	
Gültigkeitsmeldung	
Eingabe erforderlich	Nein
Leere Zeichenfolge	Ja
Indiziert	Nein
Unicode-Kompression	Ja
IME-Modus	Keine Kontrolle
IME-Satzmodus	Keine
Textausrichtung	Standard

Ein Feldname kann bis zu 64 Zeichen lang sein, einschließlich Leerzeichen. Drücken Sie F1, um Hilfe zu Feldnamen zu erhalten.

Bild 5: Die Tabelle mit den Optionen der Anwendung

Ticketsystem, Teil VI

In den vorherigen Teilen dieser Beitragsreihe haben wir den Aufbau einiger Funktionen eines Ticketsystems beschrieben. Es fehlt noch der letzte Feinschliff: Wir wollen die erneuten Antworten von Kunden auf unsere als Antwort versendeten E-Mails automatisch in die Ticketverwaltung aufnehmen. Wie dies gelingt und wie Sie etwa nach Tickets filtern können oder per Ticketsystem erstellte E-Mails in Outlook einzusehen sind, zeigen wir in dieser letzten Folge der Beitragsreihe.

In diesem letzten Teil kümmern wir uns um drei noch offene Aufgaben:

- Zu den einzelnen per **Neue Aktion** hinzugefügten Aktionen soll es möglich sein, die entsprechenden E-Mails in Outlook anzuzeigen.
- Die Antworten des Benutzers sollen erkannt werden. Dazu soll der Benutzer die Markierung im Betreff (etwa **[Ticketx]** mit **x** für die Nummer des Tickets) in der Antwort beibehalten. Solche E-Mails sollen dann ebenfalls in den Ticketverwaltungs-Ordner von Outlook verschoben werden, wo sie dann direkt dem jeweiligen Ticket zugeordnet werden.
- Außerdem wollen wir das Formular **frmTickets** noch so anpassen, dass Sie nach Tickets filtern können – nach verschiedenen Kriterien wie dem Kunden, der Bezeichnung, dem Inhalt, dem Datum oder auch dem Status.

E-Mails zu den Aktionen einsehen

Im Formular **frmTicket** werden in einem Unterformular alle Aktionen angezeigt, die im Zusammenhang mit dem Ticket durchgeführt wurden. In der Tabelle **tblAktionen**, deren Datensätze in diesem Unterformular angezeigt werden, befindet sich bereits ein Feld namens **MailItemID**.

Dieses wird allerdings beim Verwenden einer Mail, die mit dieser Aktion zusammenhängt, noch nicht gefüllt. Dies müssten wir als Erstes bewerkstelligen, danach könnten wir dann ein Steuerelement hinzufügen, mit dem wir die entsprechende E-Mail öffnen können. Das wäre der einfache Weg, wenn man nur das Öffnen der E-Mail betrachtet – mit der **EntryID** könnte man dies einfach über die Methode **GetItemFromID** des **Namespace**-Objekts erledigen.

Allerdings ist es relativ kompliziert, per VBA an die **EntryID** der soeben versendeten E-Mail zu gelangen. Dazu deklarieren wir im Klassenmodul des Formulars **frmTicket** zunächst eine Variable, mit der wir später den Ordner mit den gesendeten Objekten referenzieren wollen, und zwar mit dem Schlüsselwort **WithEvents**:

```
Public WithEvents objFolderItems As Outlook.Items
```

Diese Variable füllen wir in der Prozedur, die durch das Anstoßen einer neuen Aktion ausgelöst wird. Damit stellen wir sicher, dass die Variable mit hoher Wahrscheinlichkeit auch noch nach dem Senden und damit beim Verschieben der E-Mail in den Ordner der gesendeten Elemente mit einem Verweis auf diesen Ordner gefüllt ist:

```
Private Sub cboNeueAktion_AfterUpdate()
    ...
    Set objFolderItems = _
        GetMAPI.GetDefaultFolder(olFolderSentMail).Items
    ...
End Sub
```

Warum der ganze Aufwand? Weil wir irgendwie an die soeben versendete E-Mail herankommen müssen. Und

das können wir, indem wir das Ereignis nutzen, das beim Verschieben einer E-Mail in den Ordner **Gesendete Elemente** ausgelöst wird.

Dazu legen wir dafür ein Ereignis an, indem wir im linken Kombinationsfeld des Codefensters des Klassenmoduls des Formulars den Eintrag **objFolderItems** Auswählen, wodurch im rechten Kombinationsfeld automatisch der Eintrag **ItemAdd** erscheint – und gleichzeitig wird die Ereignisprozedur mit der Signatur **objFolderItems_ItemAdd** im Code hinterlegt.

Diese haben wir wie in Listing 1 gefüllt. Dort ermitteln wir zunächst das mit dem Parameter **Item** ermittelte Element und weisen es einer Objektvariablen des Typs **MailItem** zu, damit wir seine Eigenschaften per IntelliSense nutzen können.

Dann führen wir eine **INSERT INTO**-Abfrage aus, mit der wir der Tabelle **tblMailItems** einen neuen Eintrag zuweisen. Dabei tragen wir nur den Wert der Eigenschaft **EntryID** des **MailItem**-Objekts in das Feld **EntryID** ein.

Den Primärschlüsselwert des neu angelegten Datensatzes ermitteln wir danach mit der Abfrage **SELECT @@IDENTITY** und tragen diesen dann für den Datensatz der Tabelle **tblAktionen** ein, in dessen Kontext die E-Mail verschickt wurde. Den dabei verwendeten Wert für das Feld **AktionID** haben wir dabei in einer modulweit deklarierten Variablen namens **lngNeuesteAktion** zwischengespeichert:

```
Dim lngNeuesteAktionID As Long
```

Diesen Wert weisen wir der Variablen in der Prozedur **cboNeueAktion** zu, nachdem wir den entsprechenden Eintrag zur Tabelle **tblAktionen** hinzugefügt haben.

Mehr Informationen dazu wollen wir dem Datensatz der Tabelle **tblMailItems** gar nicht hinzufügen – wir wollen nur die **EntryID** vorhalten, um gleich schnell per Mausklick unsere Antwort im E-Mail-Format öffnen zu können.

Wo wollen wir diesen Mausklick im Formular **frmTicket** platzieren? Am einfachsten wäre eine Schaltfläche, mit der wir den aktuellen Eintrag im Unterformular mit den Aktionen auslesen und die entsprechende E-Mail öffnen können. Diesen fügen wir wie in Bild 1 gezeigt ein.

Diese Schaltfläche soll die Ereignisprozedur aus Listing 2 auslösen. Diese prüft zunächst, ob aktuell ein Eintrag im Unterformular **sfmAktionen** markiert ist, dessen Feld **MailItemID** einen Wert enthält. Ist das der Fall, ermitteln wir den Wert des Feldes **EntryID** zu dem Datensatz der Tabelle **tblMailItems**, dessen Feld **MailItemID** dem

```
Private Sub objFolderItems_ItemAdd(ByVal Item As Object)
    Dim db As DAO.Database
    Dim objMailItem As Outlook.MailItem
    Dim lngMailItemID As Long
    Set db = CurrentDb
    Set objMailItem = Item
    With objMailItem
        db.Execute "INSERT INTO tblMailItems(EntryID) VALUES('" & objMailItem.EntryID & "')", dbFailOnError
        lngMailItemID = db.OpenRecordset("SELECT @@IDENTITY").Fields(0)
        db.Execute "UPDATE tblAktionen SET MailItemID = " & lngMailItemID & " WHERE AktionID = " _
            & lngNeuesteAktionID, dbFailOnError
    End With
End Sub
```

Listing 1: Speichern der **EntryID** einer frisch verschickten E-Mail

```
Private Sub cmdEMailAnzeigen_Click()
    Dim strEntryID As String
    Dim objMailitem As Outlook.MailItem
    If Not IsNull(Me!sfmAktionen.Form!MailitemID) Then
        strEntryID = DLookup("EntryID", "tblMailItems", "MailItemID = " & Me!sfmAktionen.Form!MailitemID)
        Set objMailitem = GetMAPI.GetItemFromID(strEntryID)
        objMailitem.Display
    Else
        MsgBox "Zu dieser Aktion ist keine E-Mail verfügbar."
    End If
End Sub
```

Listing 2: Anzeigen einer E-Mail, die im Rahmen einer Aktion versendet wurde

gleichnamigen Feld des aktuell markierten Datensatzes im Unterformular entspricht.

Dann weisen wir der Variablen **objMailItem** mit der **GetItemFromID**-Funktion das **MailItem**-Objekt zu, das zu dieser **EntryID** gehört und öffnen es mit der **Display**-Methode.

Auf diese Weise können Sie die an den Kunden oder an andere beteiligte Personen verschickten E-Mails über Outlook öffnen.

Antworten der Benutzer automatisch erkennen

Wenn der Benutzer auf eine der E-Mails antwortet, die sie ihm geschickt haben, sollte er diesen Teil in eckigen Klammern im E-Mail-Betreff beibehalten.

Wenn Sie eine solche E-Mail dann aus dem Posteingang in den Ordner der Ticketverwaltung ziehen, sollte automatisch erkannt werden, zu welchem Ticket die E-Mail gehört und diese sollte dann zu dem Ticket hinzugefügt werden.

Wir erinnern uns: Wenn eine E-Mail vom Posteingang in den als Ordner der Ticketverwaltung angegebenen Ordner gezogen wird, löst das die Ereignisprozedur **objItems_ItemAdd** aus. In dieser rufen wir dann die Prozedur **MailEinlesen** auf, in der wir die Informationen aus der E-Mail verarbeiten und diese dann in die Tabelle **tblMailItems**

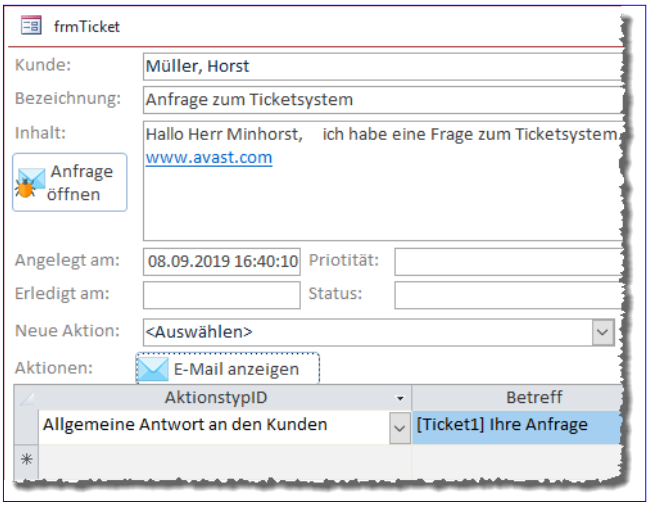

Bild 1: Anlegen der Prozedur, die beim Start von Outlook ausgelöst wird

eintragen. Wenn ein Benutzer eine E-Mail schickt, mit der er auf eine unserer Antworten antwortet, sollte diese im Betreff einen Text wie **[Ticketx]** enthalten, wobei **x** für die Ticketnummer steht.

Dies wollen wir in der Prozedur **MailEinlesen** nun vorab prüfen, um die Mail direkt als Antwort in einem vorhandenen Ticket einpflegen zu können.

Dazu erweitern wir die Prozedur **MailEinlesen** zunächst wie folgt:

```
Public Sub MailEinlesen(...)
    ...
```