AUSGABE 04/2026

Access, SQL und Cloud AUTOMATION

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT ACCESS, SQL SERVER UND CLOUD-AUTOMATIONEN MIT VBA UND CO.



IN DIESEM HEFT:

PDFS IN SQL SERVER SPEICHERN

Lerne, wie Du PDF-Dokumente in einem varbinary(max)-Feld speicherst und anzeigst.

SEITE 56

T-SQL FÜR ACCESS-ENTWICKLER

Lerne, wie Du Jet-SQL-Abfragen in T-SQL umsetzt und an den SQL Server schickst.

SEITE 23

STRINGFUNKTIONEN IN VBA

Lerne die wichtigsten Funktionen für Zeichenketten kennen und setze sie in der Praxis ein.

SEITE 4

André Minhorst Verlag

Nutze unsere Lernplattform!



Seit einige Ausgaben stellen wir unsere Artikel auch auf unserer exklusiven Lernplattform zur Verfügung. Wir haben mittlerweile einen Account für Dich angelegt. Du solltest eine entsprechende E-Mail mit einem Link zur Registrierung erhalten haben, wenn Du nicht ohnehin schon ein anderes Produkt auf unserer LearningSuite nutzen solltest.

Unsere Lernplattform bietet Dir einige praktische Features:

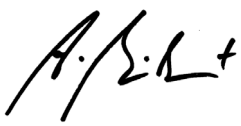
- Als Erstes kannst Du natürlich alle Artikel, die wir im Magazin veröffentlichen, in unserer Lernplattform nachlesen – und Du findest dort auch die Downloads der Beispieldateien sowie der PDFs.

- Außerdem findest Du hier eine Community, wo Du Dich mit uns und den anderen Lesern austauschen und wo Du Fragen zu den Artikeln stellen kannst.

- Schließlich haben wir eine KI bereitgestellt, die im Gegensatz zu anderen KIs speziell auf dem Wissen unserer Inhalte arbeitet. Das heißt, Du bekommst passende Antworten auf Deine Fragen und zusätzlich verweist die KI auf Fachartikel, die speziell zu Deiner Frage passen.

Im Bild rechts siehst Du, wie die Nutzung des Access-GOATs, unserer Access-KI, funktioniert.

Viel Spaß beim Erkunden der neuen Lernplattform!



André Minhorst

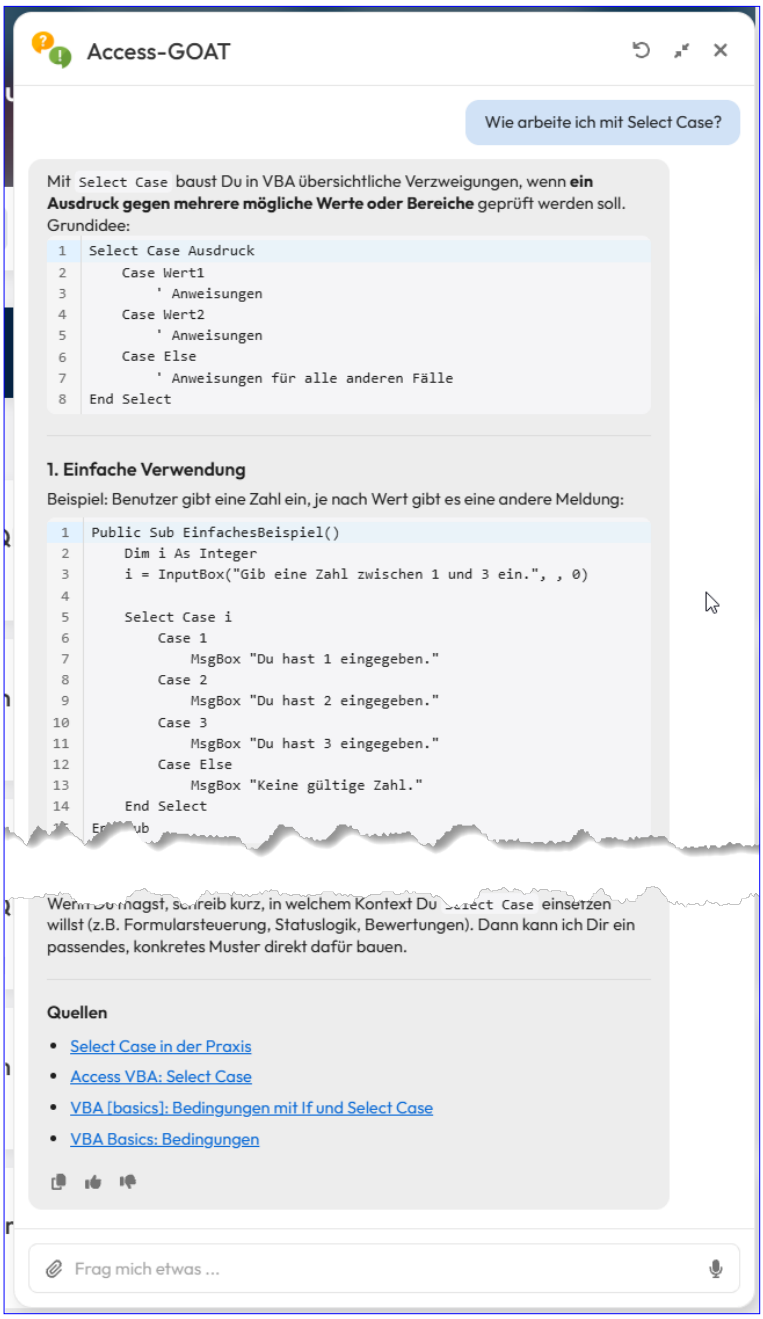


Bild 1: Den Access-GOAT nutzen

VBA Basics: Strings und Stringfunktionen

Kaum eine VBA-Prozedur kommt ohne Zeichenketten aus. Wir zerlegen Dateinamen, setzen SQL-Anweisungen zusammen, bereinigen Importdaten oder formatieren Ausgaben – und greifen dabei ständig zu Left, Mid, InStr und Replace. Trotzdem wurden diese Funktionen in unserer VBA-Reihe bisher nie vollständig vorgestellt. Das holen wir hier nach: zuerst eine kompakte Übersicht aller Stringfunktionen, dann praxisnahe Lösungen für typische Aufgaben – vom CSV-Parsing über Pattern-Matching bis zur sicheren Zusammensetzung von SQL-Strings. Die VBA-Grundlagen setzen wir voraus; hier geht es um Techniken, die aus fragilen Einzeilern robuste Lösungen machen.

Alle Stringfunktionen auf einen Blick

VBA bietet rund 30 Funktionen und Operatoren für die Arbeit mit Zeichenketten. Die folgende Liste zeigt sie gruppiert nach Aufgabe. Die meisten existieren in zwei Varianten: ohne Dollarzeichen (Ergebnis **Variant**) und mit Dollarzeichen (Ergebnis **String**). Die **$**-Variante ist schneller und wirft bei **Null** einen Laufzeitfehler statt **Null** stillschweigend weiterzureichen. Für produktiven Code empfehlen wir grundsätzlich die **$**-Form.

Teilstrings extrahieren – mit den folgenden Funktionen holen wir Teilzeichenketten von links, recht und auch aus der Mitte:

- **Left/Left$**: liefert die ersten n Zeichen von links
- **Right/Right$**: liefert die letzten n Zeichen von rechts
- **Mid/Mid$**: liefert ab Position p optional n Zeichen; ohne n den gesamten Rest

Suchen – Du suchst die Position einer bestimmten Zeichenkette von vorn oder hinten? Hier sind die geeigneten Funktionen:

- **InStr**: Position des ersten Vorkommens eines Suchstrings (von links)
- **InStrRev**: Position des letzten Vorkommens (von rechts)

Ersetzen – auch das gelingt schnell mit einer VBA-Funktion:

- **Replace**: ersetzt alle (oder n) Vorkommen eines Teilstrings

Aufteilen und Zusammenfügen – diese beiden Funktionen trennen Texte an bestimmten Zeichen und schieben sie in ein Array und umgekehrt:

- **Split**: zerlegt einen String an einem Trennzeichen in ein Array
- **Join**: fügt ein Array mit einem Trennzeichen zu einem String zusammen

Bereinigen – Leerzeichen vorn, hinten oder an beiden Seiten entfernen gelingt mit diesen Funktionen:

- **Trim/Trim$**: entfernt Leerzeichen am Anfang und Ende
- **LTrim/LTrim$**: entfernt Leerzeichen nur am Anfang
- **RTrim/RTrim$**: entfernt Leerzeichen nur am Ende

Groß-/Kleinschreibung – wer bequem zwischen beiden umwandeln möchte, hat diese Möglichkeiten:

- **UCase/UCase$**: wandelt in Grossbuchstaben um
- **LCase/LCase$**: wandelt in Kleinbuchstaben um
- **StrConv**: vielseitig: Gross, Klein, Proper Case, Unicode-Konvertierung

Vergleichen und Analysieren – dazu können wir die folgenden Funktionen nutzen:

- **StrComp**: vergleicht zwei Strings mit wählbarer Methode (binär oder textuell); liefert -1, 0 oder 1
- **Like**: Operator für Pattern-Matching mit Platzhaltern (*, **?**, #, **[...]**)

Länge und Zeichencodes – kein Problem mit diesen Funktionen:

- **Len**: Anzahl der Zeichen (bei **Null**: liefert **Null**)
- **Asc**: liefert den Zeichencode des ersten Zeichens
- **Chr/Chr$**: liefert das Zeichen zu einem Code

Erzeugen und Füllen – zum Erzeugen von beliebig vielen Zeichen in einer Zeichenkette:

- **String/String$**: wiederholt ein einzelnes Zeichen n-mal
- **Space/Space$**: erzeugt n Leerzeichen

Formatieren – nur eine Funktion, aber sehr vielseitig einsetzbar:

- **Format/Format$**: formatiert Zahlen, Datumsangaben und Strings nach einer Formatmaske

Konvertieren – für die folgenden Spezialfälle:

- **CStr**: wandelt einen beliebigen Wert in einen String um
- **Val**: liest eine Zahl vom Anfang eines Strings
- **Str/Str$**: wandelt eine Zahl in einen String (mit führendem Leerzeichen)

Verkettung – mit unterschiedlicher Verarbeitung von **Null**:

- **&**: Verkettungsoperator; wandelt **Null** in Leerstring um
- **+**: alternativer Operator; vererbt **Null** an das Ergebnis

Sonstiges:

- **StrReverse**: dreht die Zeichenfolge um
- **Nz**: Access-spezifisch; ersetzt **Null** durch einen Vorgabewert

String, Variant und das Dollarzeichen

Warum gibt es die meisten Funktionen doppelt – einmal als **Left** und einmal als **Left$**? Der Grund ist einfach: Die Variante ohne Dollarzeichen gibt einen **Variant** zurück, die Variante mit Dollarzeichen einen **String**.

Das klingt nach Erbsenzählerei, hat aber eine handfeste Folge – die **$**-Variante ist schneller, weil keine Typumwandlung anfällt, und sie ist ehrlicher, weil sie bei einem **Null**-Wert einen Fehler auslöst, statt ihn stillschweigend weiterzureichen.

Für Fortgeschrittene ergibt sich daraus eine klare Regel: Wenn die Eingabe garantiert kein **Null** enthält (etwa

bei **String**-Variablen oder bereits geprüften Werten), nehmen wir **Left$**, **Mid$**, **Trim$** und so weiter. Nur wenn wir bewusst mit **Null** aus Datenbankfeldern arbeiten, greifen wir zur **Variant**-Variante – und sichern uns vorher mit **Nz** ab.

Was Null mit Texten anstellt

Importdaten sind voller leerer Felder, und in Access heisst leer nun einmal **Null**. Das Verhalten ist tückisch: **Len(Null)** liefert nicht **0**, sondern selbst **Null**.

Eine Prüfung wie **If Len(varText) > 0 Then** ist damit wirkungslos – der Vergleich ergibt **Null**, und **If** behandelt das wie **False**. Noch gefährlicher: Der Verkettungsoperator **&** wandelt **Null** still in einen Leerstring um, während + das gesamte Ergebnis zu **Null** macht.

Die Lösung in Access ist die Funktion **Nz**. Sie ersetzt **Null** durch einen Vorgabewert. Für unsere Zwecke lautet die Standard-Absicherung:

```
Dim strSicher As String
strSicher = Nz(rst!Feldname, vbNullString)
```

Ab jetzt können wir gefahrlos mit den **$**-Funktionen arbeiten. vbNullString ist gleichbedeutend mit einer leeren Zeichenkette, wir könnten also auch nutzen:

```
strSicher = Nz(rst!Feldname, "")
```

Zerlegen mit Left, Right und Mid

Die drei Klassiker schneiden Teilstücke aus einer Zeichenkette heraus. **Left$(strText, 5)** liefert die ersten fünf Zeichen, **Right$(strText, 5)** die letzten fünf und **Mid$(strText, 3, 4)** vier Zeichen ab Position **3**. Zwei Details lohnen sich zu merken: Die Zählung beginnt bei **1**, und das dritte Argument von **Mid** ist optional – lässt man es weg, bekommt man den gesamten Rest ab der Startposition.

Ein typischer Praxisfall: Aus einem Dateinamen wie **Kunden_2026-07-15_Export.csv** wollen wir das Datum extrahieren. Dafür kombinieren wir **InStrRev** mit **Mid$**.

Die Funktion aus Listing 1 nutzt gleich mehrere Techniken auf einmal: **Split** zerlegt den Basisnamen an den

```
Public Function DateiDatumExtrahieren(ByVal strDateiname As String) As Variant

    Dim varTeile As Variant
    Dim lngIndex As Long

    DateiDatumExtrahieren = Null
    varTeile = Split(DateiOhnePfad(strDateiname), "_")

    For lngIndex = LBound(varTeile) To UBound(varTeile)
        If varTeile(lngIndex) Like "####-##-##" Then
            DateiDatumExtrahieren = DateSerial(CLng(Left$(varTeile(lngIndex), 4)), _
                CLng(Mid$(varTeile(lngIndex), 6, 2)), CLng(Mid$(varTeile(lngIndex), 9, 2)))
            Exit For
        End If
    Next lngIndex

End Function
```

Listing 1: Datum aus einem Dateinamen extrahieren mit **Split**, **Like** und **Mid$**

Unterstrichen, der **Like**-Operator prüft jedes Stück auf das ISO-Datumsmuster, und **Left$** sowie **Mid$** schneiden Jahr, Monat und Tag heraus.

Suchen mit InStr und InStrRev

InStr liefert die Position, an der eine Zeichenfolge zum ersten Mal vorkommt, und **0**, wenn sie nicht vorkommt. Die Funktion kennt zwei Aufrufformen: Ohne Startposition heisst es **InStr(strText, strSuche)**, mit Startposition dagegen **InStr(lngStart, strText, strSuche)** – die Startposition rutscht also nach vorn und schiebt die anderen Argumente eine Stelle weiter.

Wer das Vergleichsargument (**vbBinaryCompare** oder **vbTextCompare**) angeben will, muss zwingend die lange Form benutzen.

InStrRev sucht dieselbe Zeichenfolge von hinten. Für Dateipfade ist das genau richtig, denn dort interessiert uns immer der letzte Backslash. Allerdings dreht **InStrRev** die Argumentreihenfolge wieder um – hier heisst es **InStrRev(strText, strSuche)**:

```
Public Function DateiOhnePfad( _
        ByVal strPfad As String) As String

    Dim lngPos As Long

    lngPos = InStrRev(strPfad, "\")

    If lngPos = 0 Then
        DateiOhnePfad = strPfad
    Else
        DateiOhnePfad = Mid$(strPfad, lngPos + 1)
    End If

End Function
```

Die Funktion sucht den letzten Backslash und gibt alles danach zurück. Kommt kein Backslash vor, ist der gesamte String bereits der Dateiname.

Ersetzen mit Replace

Replace ersetzt alle Vorkommen eines Teilstrings durch einen anderen. Die vollständige Signatur lautet **Replace(strText, strSuche, strErsatz, lngStart, lngAnzahl, lngVergleich)**.

Die letzten drei Argumente sind optional. Allerdings hat **Replace** eine Eigenheit, die viele übersehen: Gibt man **lngStart** an, liefert die Funktion nicht den vollständigen String zurück, sondern nur den Teil ab **lngStart**. Das überrascht und ist selten gewünscht.

Ein klassischer Einsatz ist die Bereinigung von Sonderzeichen für SQL-Strings – einfache Anführungszeichen müssen verdoppelt werden, damit sie keinen Syntaxfehler auslösen:

```
Public Function SQLSafe(ByVal strWert As String) As String
    SQLSafe = Replace(strWert, "'", "''")
End Function
```

So können wir gefahrlos schreiben: **strSQL = "WHERE Nachname = '" & SQLSafe(strName) & "'"**

Aufteilen und Zusammenfügen mit Split und Join

Split zerlegt einen String an einem Trennzeichen und liefert ein nullbasiertes **Variant**-Array. **Join** macht das Gegenteil – es fügt die Elemente eines Arrays mit einem Trennzeichen wieder zusammen. Zusammen bilden sie ein mächtiges Paar für Daten-Transformationen:

```
Public Function EinLeerzeichen(ByVal strText As String) _
        As String

    Dim varTeile As Variant
    Dim lngI As Long
    Dim lngZiel As Long

    varTeile = Split(strText, " ")
```

Select Case in der Praxis

Wer VBA-Bedingungen schreibt, greift fast automatisch zu If…Then…ElseIf. Das funktioniert, wird aber bei mehr als zwei, drei Zweigen schnell unübersichtlich – und fehleranfällig, weil jede Zeile den Prüfausdruck wiederholt. Select Case ist die Alternative: kompakter, lesbarer und in vielen Fällen auch schneller. Dieser Beitrag zeigt die Grundform, alle Varianten der Case-Klausel und drei Praxismuster, die über das hinausgehen, was die meisten VBA-Entwickler kennen.

Die Grundform

Select Case prüft einen Ausdruck gegen eine Reihe von Werten. Trifft ein **Case**-Zweig zu, führt VBA die darin enthaltenen Anweisungen aus und springt danach direkt zu **End Select** – es gibt kein Durchfallen wie in manchen anderen Sprachen:

```
Select Case intWochentag
    Case 1
        strName = "Montag"
    Case 2
        strName = "Dienstag"
    Case 3
        strName = "Mittwoch"
    Case Else
        strName = "(anderer Tag)"
End Select
```

Der Ausdruck hinter **Select Case** wird genau einmal ausgewertet. Das ist ein Vorteil gegenüber einer **If…ElseIf**-Kette, in der wir denselben Ausdruck in jeder Zeile wiederholen müssten.

Bei aufwendigen Funktionsaufrufen spart das nicht nur Tipparbeit, sondern auch Rechenzeit.

Case Else als Sicherheitsnetz

Der **Case Else**-Zweig fängt alles auf, was keiner der vorangehenden **Case**-Klauseln entspricht. Er ist optional, aber in der Praxis fast immer empfehlenswert – er macht explizit, was passiert, wenn ein unerwarteter Wert auftaucht. Das gilt besonders für Eingaben von Benutzern oder Werte aus Datenbanken, die sich jederzeit ändern können.

Wer sicher ist, dass alle möglichen Werte abgedeckt sind, kann **Case Else** als Wachposten nutzen – mit einer Fehlermeldung, die auf den Programmierfehler hinweist:

```
Case Else
    Err.Raise vbObjectError + 1001, _
        "StatusText", "Unbekannter Status: " & intStatus
```

Mehrere Werte in einer Case-Klausel

Eine **Case**-Klausel kann mehrere Werte aufnehmen, durch Komma getrennt. Das spart Zweige, wenn dieselbe Aktion für verschiedene Werte gelten soll:

```
Select Case intWochentag
    Case 1, 2, 3, 4, 5
        strTyp = "Werktag"
    Case 6, 7
        strTyp = "Wochenende"
End Select
```

Die Werte müssen nicht aufeinanderfolgend sein. **Case 1, 3, 5, 7** ist genauso gültig. Bei aufeinanderfolgenden Werten gibt es allerdings eine elegantere Lösung – den **To**-Bereich.

Bereiche mit To

Mit **Case 1 To 5** prüfen wir, ob der Wert zwischen **1** und **5** liegt – beide Grenzen eingeschlossen. Das funk-

tioniert auch mit Strings und Datumsangaben. Das Wochentagbeispiel von eben wird damit noch kompakter:

```
Select Case intWochentag
    Case 1 To 5
        strTyp = "Werktag"
    Case 6 To 7
        strTyp = "Wochenende"
End Select
```

Bereiche, Einzelwerte und **Is**-Vergleiche lassen sich in derselben **Case**-Klausel mischen. Ein Beispiel: **Case 1 To 5, 10, Is > 100** trifft auf die Werte **1** bis **5**, auf **10** und auf alles über **100** zu.

Vergleiche mit Is

Manchmal brauchen wir keine exakten Werte, sondern Schwellenwerte. Dafür gibt es das Schlüsselwort **Is** in Kombination mit einem Vergleichsoperator. Ein typisches Beispiel ist die Bewertung einer Punktzahl:

```
Public Function Bewertung(ByVal lngPunkte As Long) As String

    Select Case lngPunkte
        Case Is >= 90
            Bewertung = "sehr gut"
        Case Is >= 75
            Bewertung = "gut"
        Case Is >= 50
            Bewertung = "befriedigend"
        Case Is >= 25
            Bewertung = "ausreichend"
        Case Else
            Bewertung = "mangelhaft"
    End Select

End Function
```

Wichtig ist die Reihenfolge: VBA prüft die **Case**-Klauseln von oben nach unten und nimmt den ersten Treffer. Wer **Case Is >= 25** an den Anfang stellt, bekommt für 95 Punkte nur „ausreichend“, weil diese Klausel bereits zutrifft. Die Reihenfolge muss also vom engsten zum weitesten Kriterium laufen.

Select Case mit Strings

Select Case funktioniert nicht nur mit Zahlen. Strings werden nach den Regeln verglichen, die **Option Compare** vorgibt – in Access-Modulen also standardmässig ohne Berücksichtigung der Gross-/Kleinschreibung:

```
Public Function DateiTyp( _
        ByVal strErweiterung As String) As String

    Select Case LCase$(strErweiterung)
        Case "accdb", "accde", "mdb"
            DateiTyp = "Access-Datenbank"
        Case "xlsx", "xlsm", "xls"
            DateiTyp = "Excel-Arbeitsmappe"
        Case "docx", "docm", "doc"
            DateiTyp = "Word-Dokument"
        Case "pdf"
            DateiTyp = "PDF-Dokument"
        Case "csv", "txt"
            DateiTyp = "Textdatei"
        Case Else
            DateiTyp = "Unbekannt"
    End Select

End Function
```

In diesem Listing sichern wir mit **LCase$** ab, dass der Vergleich unabhängig von der Schreibweise funktioniert – auch in Modulen mit **Option Compare Binary**. Mit **To** lassen sich auch String-Bereiche prüfen: **Case "A" To "M"** trifft auf alle Strings zu, die mit einem Buchstaben zwischen A und M beginnen.

Das Pattern: Select Case True

Das mächtigste – und am wenigsten bekannte – Muster ist **Select Case True**. Hier prüft VBA nicht einen

VBA Basics: Nz, IIf, Switch und Choose

Mit If...Then und Select Case entscheiden wir, welche Anweisungen VBA ausführt. Oft brauchen wir aber einen Wert, der von einer Bedingung abhängt, etwa einen Rabattsatz oder die Vorgabe für ein leeres Feld. Dafür stehen die Funktionen Nz, IIf, Switch und Choose zur Verfügung. Sie werten alle Parameter aus und liefern in bestimmten Fällen Null, was zu Fehlern führen kann. Wie die Funktionen arbeiten und wann wir besser If...Then verwenden, liest Du in diesem Artikel.

Werte statt Anweisungen

Die Anweisungen **If...Then** und **Select Case** haben wir im Artikel **VBA Basics: Bedingungen (www.vbentwickler.de/336)** beschrieben. Sie führen je nach Bedingung unterschiedliche Anweisungen aus. Die Funktionen in diesem Artikel liefern dagegen einen Wert zurück. Wir setzen sie dort ein, wo ein Wert erwartet wird, zum Beispiel rechts von einer Zuweisung oder als Parameter eines anderen Aufrufs.

Die Beispiele findest Du im Modul **mdlBedingungsfunktionenBeispiele** der Datenbank **VBABasics_Bedingungsfunktionen.accdb**, zum Ausprobieren dient das Formular **frmBedingungsfunktionen**.

Nz: Null durch einen Wert ersetzen

Ein Feld, in das nichts eingegeben wurde und das keinen Standardwert hat, enthält den Wert **Null**. Rechnen wir mit diesem Wert, ist auch das Ergebnis **Null**:

```
Debug.Print 100 * (1 - Null)
```

Die Ausgabe lautet **Null**. Enthält das Feld **Rabatt** einer Bestellung den Wert **Null**, ist auch der Preis nach Rabatt **Null**. Die Funktion **Nz** ersetzt **Null** durch den Wert, den wir als zweiten Parameter angeben. Wir setzen **Nz** in der Funktion **PreisMitRabatt** ein. Sie erwartet den Preis als **Currency** und den Rabatt als **Variant**, damit wir auch **Null** übergeben können:

```
Public Function PreisMitRabatt(curPreis As Currency, _
        varRabatt As Variant) As Currency
    PreisMitRabatt = curPreis * (1 - Nz(varRabatt, 0))
End Function
```

Mit **Null** als Rabatt erhalten wir den Preis unverändert, mit dem Rabatt **0,1** (im Code **0.1**) den um zehn Prozent verminderten Preis:

```
Debug.Print PreisMitRabatt(100, Null)
Debug.Print PreisMitRabatt(100, 0.1)
```

Die Ausgaben lauten **100** und **90**. Der zweite Parameter von **Nz** ist optional. Fehlt er, liefert **Nz** für **Null** einen Wert vom Datentyp **Variant** mit dem Untertyp **Empty**, der in Berechnungen als **0** und in Zeichenketten als leerer Text gilt. Der Ausdruck **Nz(Null) + 5** ergibt daher **5** und **Nz(Null) & "x"** ergibt **x**.

Nz ersetzt nur **Null**. Ein leerer Text ist nicht **Null**, deshalb gibt **Nz("", "x")** den leeren Text zurück.

Nz gehört zu Access und nicht zu VBA. In Excel, Word oder Outlook steht die Funktion ohne einen Verweis auf die Access-Bibliothek nicht zur Verfügung. Dort prüfen wir mit **IsNull**, ob ein Wert **Null** ist.

IIf: Einen von zwei Werten auswählen

IIf erwartet drei Parameter: die Bedingung, den Wert für **True** und den Wert für **False**. Als Beispiel berechnen wir den Anteil eines Teils an einer Summe. Bei der Summe **0** soll das Ergebnis ebenfalls **0** lauten, damit keine Division durch **0** stattfindet. Zuerst versuchen wir es mit **IIf**:

```
Public Function AnteilIIf(dblTeil As Double, _
        dblSumme As Double) As Double
    AnteilIIf = IIf(dblSumme = 0, 0, dblTeil / dblSumme)
End Function
```

Der erste Aufruf liefert **0,25**, der zweite löst einen Fehler aus:

```
Debug.Print AnteilIIf(1, 4)
Debug.Print AnteilIIf(5, 0)
```

Der zweite Aufruf löst den Laufzeitfehler **11** mit der Meldung **Division durch Null** aus, obwohl wir die Division durch die Bedingung ausschließen wollen. VBA wertet nämlich beim Aufruf einer Funktion zuerst die Parameter aus. Bei **IIf** gehören dazu beide Werte, auch der Wert, den wir gar nicht benötigen.

Die Funktion **AnteilIf** verwendet deshalb die Anweisung **If...Then**, bei der VBA nur den passenden Zweig ausführt:

```
Public Function AnteilIf(dblTeil As Double, _
        dblSumme As Double) As Double
    If dblSumme = 0 Then
        AnteilIf = 0
    Else
        AnteilIf = dblTeil / dblSumme
    End If
End Function
```

AnteilIf(5, 0) liefert **0** und **AnteilIf(1, 4)** liefert **0,25**. **IIf** setzen wir daher ein, wenn die Berechnung beider Werte keinen Fehler auslösen kann, etwa bei Konstanten oder einfachen Feldwerten. Kann einer der Werte einen Fehler auslösen, verwenden wir **If...Then**.

Switch: Mehrere Bedingungen nacheinander prüfen

Switch erwartet mehrere Paare aus einer Bedingung und einem Wert. Die Funktion liefert den Wert des ersten Paares, dessen Bedingung **True** ergibt. Als Beispiel ermitteln wir mit der Funktion **Staffelrabatt** den Rabattsatz anhand der Bestellsumme. Die Funktion gibt einen Wert vom Datentyp **Variant** zurück, weil **Switch** auch **Null** liefern kann:

```
Public Function Staffelrabatt( _
        curSumme As Currency) As Variant
    Staffelrabatt = Switch(curSumme >= 1000, 0.1, _
        curSumme >= 500, 0.05, _
        curSumme >= 0, 0)
End Function
```

Ab einer Summe von **1000** erhalten wir den Rabattsatz **0,1**, ab **500** den Satz **0,05** und darunter **0**. **Switch** prüft die Bedingungen von links nach rechts.

Die Reihenfolge bestimmt deshalb das Ergebnis, denn mit dem Vergleichsoperator Größer-gleich (>=) trifft jede Summe ab **0** zu. Stünde **curSumme >= 0** am Anfang, erhielten Summen ab **0** den Rabattsatz **0**.

Trifft keine Bedingung zu, liefert **Switch** den Wert **Null**. Bei der Summe **-1** ist das der Fall. Die Zuweisung dieses Ergebnisses an eine Variable vom Typ **Double** oder **String** löst den Fehler **94** mit der Meldung **Unzulässige Verwendung von Null** aus. Mit **Nz** erhalten wir stattdessen **0**:

```
Debug.Print Nz(Staffelrabatt(-1), 0)
```

Wie bei **IIf** wertet VBA vor dem Aufruf alle Bedingungen und alle Werte aus. Der Ausdruck **Switch(True, 1, True, 1 / 0)** löst deshalb trotz des ersten Treffers den Fehler **11** aus.

Choose: Einen Wert nach Position auswählen

Choose erwartet einen Index und eine Liste von Werten und liefert den Wert an der Position des Index. Die erste Position hat den Index **1**. Die Funktion **Lieferart**

VBA Basics: Auflistungen mit Collections

Arrays haben eine feste Grösse – oder müssen mit ReDim Preserve umständlich vergrössert werden. Sobald wir eine Liste brauchen, die während der Laufzeit wächst und schrumpft, ist die Collection das Mittel der Wahl. Sie gehört zur VBA-Laufzeitbibliothek, steht ohne zusätzlichen Verweis in jedem Projekt zur Verfügung und lässt sich mit For Each durchlaufen. Dieser Beitrag zeigt, was die Collection kann, wo ihre Grenzen liegen und wie wir sie in der Praxis einsetzen.

Was ist eine Collection?

Eine **Collection** ist eine geordnete Liste von Elementen. Jedes Element kann einen beliebigen Datentyp haben – Strings, Zahlen, Objekte oder sogar weitere Collections. Die Reihenfolge der Elemente bleibt erhalten, und jedes Element bekommt automatisch einen numerischen Index, der bei **1** beginnt.

Eine neue **Collection** legen wir so an:

```
Dim col As New Collection
```

Mit **New** wird die Instanz automatisch beim ersten Zugriff auf die Variable erzeugt – nicht bereits in der Deklarationszeile selbst. Alternativ können wir **Dim col As Collection** schreiben und später **Set col = New Collection** aufrufen – das ist nützlich, wenn wir die Collection erst unter bestimmten Bedingungen brauchen.

Die Member der Collection

Die **Collection** ist bewusst schlank. Sie bietet genau vier Member:

- **Add Item, [Key], [Before], [After]:** fügt ein Element hinzu. **Item** ist der Wert, **Key** ein optionaler, eindeutiger String-Schlüssel. Mit **Before** oder **After** steuern wir die Einfügeposition.

- **Item(Index):** gibt das Element an Position **Index** zurück (1-basiert) oder das Element mit dem angegebenen Schlüssel. **Item** ist die Standardeigenschaft – **col(3)** ist gleichbedeutend mit **col.Item(3)**.

- **Count:** liefert die Anzahl der Elemente.

- **Remove(Index):** entfernt das Element an der angegebenen Position oder mit dem angegebenen Schlüssel. Die Indizes der nachfolgenden Elemente verschieben sich automatisch.

Was auffällt: Es gibt kein **Exists**, kein **Replace** und kein **Clear** oder **RemoveAll**.

Wer die Collection leeren will, kann zwar alle Elemente rückwärts mit **Remove** löschen, schneller und sauberer ist aber eine neue Instanz: **Set col = New Collection**.

```
Public Sub CollectionFuellen()

    Dim col As New Collection

    col.Add "Berlin"
    col.Add "Hamburg"
    col.Add "München"

    Debug.Print col(1)       ' -> Berlin
    Debug.Print col(2)       ' -> Hamburg
    Debug.Print col.Count    ' -> 3

End Sub
```

Listing 1: Elemente hinzufügen und per Index abrufen

Elemente hinzufügen

Im einfachsten Fall fügen wir Elemente ohne Schlüssel hinzu. Sie landen am Ende der Liste und sind über ihren Index erreichbar (siehe Listing 1).

Arbeiten mit Schlüsseln

Geben wir beim **Add** einen Schlüssel an, können wir später über diesen Schlüssel auf das Element zugreifen – unabhängig von seiner Position. Der Schlüssel muss ein **String** sein und innerhalb der Collection eindeutig. Beachte die Reihenfolge: Zuerst der Wert, dann der Schlüssel – das ist anders als beim **Dictionary**, wo der Schlüssel an erster Stelle steht.

```
Public Sub CollectionMitKeys()

    Dim col As New Collection

    ' Add Wert, Schluessel
    col.Add "Berlin", "BE"
    col.Add "Hamburg", "HH"
    col.Add "München", "BY"

    ' Zugriff per Schluessel
    Debug.Print col("HH")    ' -> Hamburg

    ' Zugriff per Index geht weiterhin
    Debug.Print col(1)       ' -> Berlin

    ' Entfernen per Schluessel
    col.Remove "HH"
    Debug.Print col.Count    ' -> 2

End Sub
```

Listing 2: Collection mit Schlüsseln

Ein wichtiges Detail: Nach dem **Remove** in Listing 2 verschiebt sich der Index. **col(2)** liefert jetzt **"München"**, nicht mehr **"Hamburg"**. Wer über den Index auf Elemente zugreift und gleichzeitig entfernt, muss die Schleife rückwärts laufen lassen – dazu gleich mehr.

Einfügeposition steuern

Mit den Argumenten **Before** und **After** können wir Elemente an einer bestimmten Position einfügen statt immer am Ende. Das ist ein Vorteil, den das **Dictionary** nicht bietet:

```
Dim col As New Collection

col.Add "Zweiter"
col.Add "Erster", Before:=1
col.Add "Dritter", After:=2

' Ergebnis: Erster, Zweiter, Dritter
```

Durchlaufen mit For Each

Die **Collection** lässt sich mit **For Each** durchlaufen. Die Laufvariable muss als **Variant** deklariert sein, wenn die Collection Werte enthält, oder als der jeweilige Objekttyp, wenn sie Objekte enthält:

```
' Werte durchlaufen
Dim varItem As Variant
For Each varItem In col
    Debug.Print varItem
Next varItem

' Objekte durchlaufen
Dim frm As Form
For Each frm In Forms
    Debug.Print frm.Name
Next frm
```

Alternativ können wir eine **For…Next**-Schleife mit dem numerischen Index verwenden.

Das ist nötig, wenn wir während des Durchlaufs Elemente entfernen wollen – **For Each** verträgt das nicht:

T-SQL, Teil 1: Von Jet-SQL zu T-SQL

Die Abfrage läuft in Access seit Jahren fehlerfrei – bis die Tabellen auf den SQL Server umziehen. Plötzlich liefert Format im Gruppierungsausdruck eine Fehlermeldung, Like "M*" findet keinen einzigen Datensatz, und ein Aufruf von Nz endet mit der Meldung, der Server kenne diese Funktion nicht. Wer Access mit einem SQL Server als Backend betreibt, kommt früher oder später nicht mehr um T-SQL herum, die Abfragesprache des Servers. Was sich dabei gegenüber dem SQL von Access ändert und wie wir T-SQL aus Access und aus VBA heraus ausführen, zeigt der erste Teil dieser Reihe.

Beispieldatenbank

Als Beispiel dient durch die ganze Reihe eine kleine Bestellverwaltung: Kunden bestellen bei einem Händler für Lebensmittel, Büroartikel und Technik, Mitarbeiter betreuen die Bestellungen, Lieferanten beliefern das Sortiment.

Die Datenbank heißt **Bestellverwaltung_TSQL** und besteht aus diesen Tabellen:

- **tblKategorien:** die Warengruppen als zweistufige Hierarchie, über das Fremdschlüsselfeld **OberKategorieID** auf sich selbst verwiesen
- **tblLieferanten:** sieben Lieferanten aus Deutschland, Österreich und der Schweiz
- **tblProdukte:** dreißig Produkte mit Preis, Lagerbestand, dem Ja/Nein-Feld **Auslaufartikel** und einer optionalen Beschreibung
- **tblKunden:** zweiunddreißig Kunden, teils Firmen, teils Privatpersonen, mit leeren Feldern und Sonderzeichen in den Namen
- **tblMitarbeiter:** acht Mitarbeiter, über **VorgesetzterID** ebenfalls hierarchisch verknüpft
- **tblBestellstatus:** die fünf Bearbeitungsstufen einer Bestellung

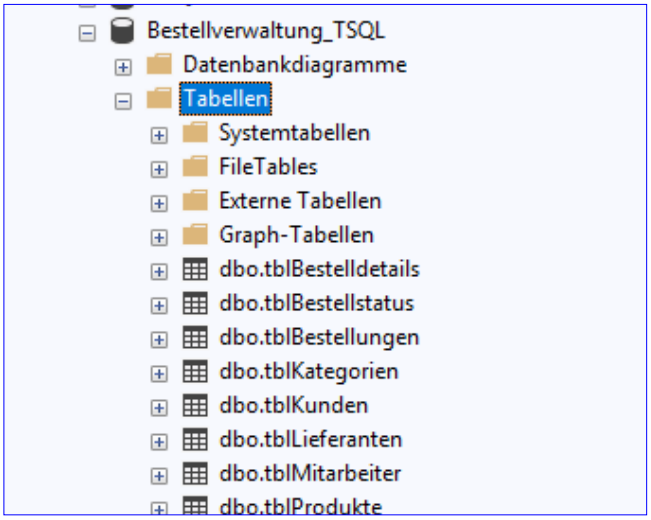


Bild 1: Die Tabellen der Datenbank **Bestellverwaltung_TSQL** im Objekt-Explorer von **SSMS**

- **tblBestellungen:** 330 Bestellungen von Januar 2024 bis September 2026
- **tblBestelldetails:** die 825 Positionen der Bestellungen

Die Namen folgen einer festen Konvention: Tabellen beginnen mit **tbl**, der Primärschlüssel einer Tabelle heißt wie die Tabelle in der Einzahl mit angehängtem **ID** (etwa **ProduktID** in **tblProdukte**), und Fremdschlüsselfelder tragen denselben Namen wie das Feld, auf das sie verweisen (**KategorieID** in **tblProdukte** verweist auf **tblKategorien**).

```
SELECT Format([b].[Bestelldatum],"yyyy-mm") AS Monat,
       Sum([d].[Menge]*[d].[Einzelpreis]*(1-[d].[Rabatt])) AS Umsatz
FROM dbo_tblBestellungen AS b INNER JOIN dbo_tblBestelldetails AS d
     ON b.BestellungID = d.BestellungID
WHERE b.Bestelldatum >= #1/1/2026# AND b.Bestelldatum < #1/1/2027#
      AND b.BestellstatusID <> 5
GROUP BY Format([b].[Bestelldatum],"yyyy-mm")
ORDER BY Format([b].[Bestelldatum],"yyyy-mm");
```

Listing 1: Die Monatsumsätze 2026 in Access-SQL

Das Skript **Bestellverwaltung_TSQL.sql** aus dem Download legt die Datenbank samt Daten an. Wir führen es in **SQL Server Management Studio** (kurz SSMS) auf einer lokalen **SQL Server Express**-Instanz aus. Es löscht eine vorhandene Datenbank gleichen Namens und erzeugt sie neu. Danach zeigt der Objekt-Explorer die acht Tabellen (siehe Bild 1).

Das Access-Frontend **SQLServer_EinstiegVonJetSQLZuTSQL.accdb** enthält die acht Tabellen als Verknüpfungen mit den Namen **dbo_tblKategorien**, **dbo_tblKunden** und so weiter. Wie man solche Verknüpfungen von Hand oder per VBA anlegt, haben wir im Beitrag **SQL Server: Tabellen per VBA verknüpfen (www.vbentwickler.de/485)** gezeigt. Die VBA-Beispiele dieses Beitrags stehen im Modul **mdlTSQLEinstiegBeispiele**.



Bild 2: Access öffnet die Abfrage **qryMonatsumsatzJet** auf den verknüpften Tabellen nicht

Ein Szenario: Monatsumsätze

Der Vertrieb möchte wissen, wie sich der Umsatz im Jahr 2026 auf die einzelnen Monate verteilt. Stornierte Bestellungen sollen nicht zählen. In Access liegt die Lösung auf der Hand: eine Auswahlabfrage auf den verknüpften Tabellen, die den Umsatz je Monat summiert. Sie steht als **qryMonatsumsatzJet** im Frontend (siehe Listing 1).

Der Umsatz einer Position ist **Menge** mal **Einzelpreis** abzüglich **Rabatt**. Die Bestellungen mit der **BestellstatusID** 5 sind storniert und fallen heraus. Gegen reine Access-Tabellen würde die Abfrage sofort laufen. Auf den verknüpften SQL Server-Tabellen öffnet sie sich dagegen mit einer Fehlermeldung (siehe Bild 2).

Woran liegt das? Das Feld **Bestelldatum** hat im Server den Datentyp **datetime2(0)**. In unserer Beispielkonfiguration bindet Access es als **Datum/Uhrzeit, erweitert** ein, und mit diesem Typ kommt **Format** im Gruppierungsausdruck nicht zurecht. Ob Access den Typ so anbindet, lässt sich in den Optionen der aktuellen Datenbank einstellen. Ersetzen wir **Format** durch **Year** und **Month**, läuft die Abfrage.

Auch die Summen bergen Überraschungen. Die Summe von **Menge*Einzelpreis*(1-Rabatt)** über alle Positionen ohne Gruppierung scheitert mit der Meldung **Staffelung des Dezimalwerts führte zu Datenkürzung**, und erst mit **CCur** um den Ausdruck herum liefert Access ein Ergebnis. Solche Umwege kosten Zeit.

Die Alternative ist, die Auswertung dem Server zu überlassen. Dort ist sie eine einzige Anweisung in der Sprache des Servers, in

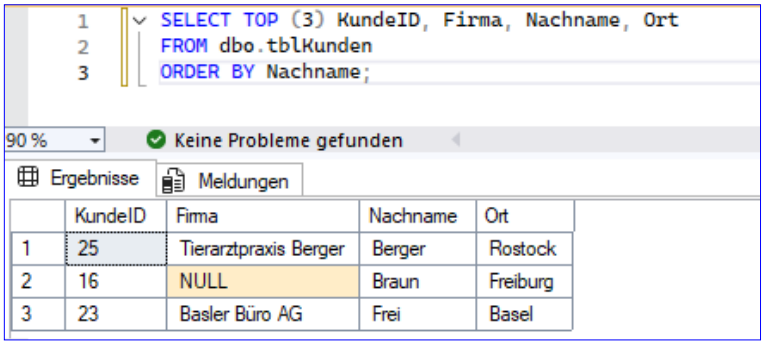


Bild 3: Das erste **SELECT** im Abfragefenster von **SSMS**, darunter das Ergebnis

T-SQL. Dafür müssen wir zunächst klären, wie Access eine Abfrage verarbeitet.

Wie Access Abfragen verarbeitet

Eine Access-Abfrage auf verknüpfte Tabellen schickt Access nicht einfach an den Server. Die Datenbank-Engine von Access verarbeitet sie selbst. Sie übersetzt so viel wie möglich über den ODBC-Treiber in T-SQL und erledigt den Rest lokal, nachdem sie die benötigten Daten vom Server geholt hat. Was dabei schiefgehen kann, haben wir im Szenario gesehen.

Der Umzug der Tabellen allein macht Access-Funktionen wie **Nz** nicht unbrauchbar. Probleme entstehen erst, wenn eine Anweisung direkt auf dem Server ausgeführt wird, denn dort gelten die Regeln von T-SQL. Dafür gibt es in Access die **Pass-Through-Abfrage**, die ihren Text unverändert an den Server reicht. Sie lernen wir weiter unten kennen. Vorher werfen wir einen Blick auf das Werkzeug, mit dem wir T-SQL am bequemsten ausprobieren.

SQL Server Management Studio

SSMS ist die Verwaltungsoberfläche des SQL Servers. Wir starten es und melden uns an der Instanz **.SQLEXPRESS** mit der Windows-Authentifizierung an. Links erscheint der **Objekt-Explorer** mit den Datenbanken der Instanz, darunter **Bestellverwaltung_TSQL**.

Ein Abfragefenster öffnen wir mit **Neue Abfrage**. Wichtig ist die Datenbank, gegen die die Anweisung läuft: Sie steht in der Auswahlliste links neben dem Ausführen-Schalter. Alternativ stellen wir der Anweisung ein **USE Bestellverwaltung_TSQL** voran.

Als erste Abfrage holen wir die ersten drei Kunden, alphabetisch nach Nachnamen sortiert:

```
SELECT TOP (3) KundeID, Firma, Nachname, Ort
FROM dbo.tblKunden
ORDER BY Nachname;
```

Fast alles daran kennen wir aus Access. Neu ist **dbo**: Das ist das Schema, zu dem die Tabelle gehört. Tabellen, Sichten und gespeicherte Prozeduren liegen im SQL Server in einem Schema, und wenn wir nichts anderes angelegt haben, heißt es **dbo**. Das Semikolon am Ende beendet die Anweisung.

Mit **F5** oder dem Schalter **Ausführen** läuft die Anweisung, und das Ergebnis erscheint im Ergebnisbereich unter dem Abfragefenster (siehe Bild 3).

In SSMS-Skripten steht häufig die Zeile **GO**. Sie ist kein T-SQL-Befehl, sondern ein Trennzeichen für SSMS und das Kommandozeilenwerkzeug **sqlcmd**. Es teilt ein Skript in einzelne Stapel (Batches), die nacheinander an den Server gehen.

Das Skript der Beispieldatenbank nutzt es zum Beispiel, damit die Datenbank angelegt ist, bevor das erste **CREATE TABLE** an sie gerichtet wird. In Pass-Through-Abfragen gehört **GO** nicht hinein.

Zeichenketten

Der auffälligste Unterschied zeigt sich bei einem Textkriterium. In Access fassen wir Text in doppelte Anführungszeichen ein. Die Abfrage nach den Kunden aus Köln lautet in der SQL-Ansicht einer Access-Abfrage so:

```
SELECT KundeID, Vorname, Nachname, Ort
FROM dbo_tblKunden
WHERE Ort = "Köln";
```

In T-SQL stehen Zeichenketten in einfachen Anführungszeichen:

```
SELECT KundeID, Vorname, Nachname, Ort
FROM dbo.tblKunden
WHERE Ort = N'Köln';
```

Vor dem Text steht hier ein **N**. Es kennzeichnet das Literal als Unicode-Text (**nvarchar**) und verhindert, dass ein Zeichen durch die Zeichensatzumsetzung verloren geht. Wir setzen es vor jeden Text, der Umlaute oder andere Sonderzeichen enthalten kann.

Die doppelten Anführungszeichen sind in T-SQL nicht verboten, aber sie bedeuten etwas anderes: Bei eingeschaltetem **QUOTED_IDENTIFIER** – das ist die Voreinstellung in SSMS und beim ODBC-Treiber – kennzeichnen sie einen Objektnamen, so wie bei Access die eckigen Klammern. Ein **"Köln"** sucht der Server deshalb als Spaltennamen. Wie die Fehlermeldung dazu aussieht, sehen wir im Abschnitt über die Fehlermeldungen des Servers. Kommt ein Apostroph im Text vor, verdoppeln wir ihn. Der Kunde Sean O'Brien findet sich so:

```
SELECT KundeID, Vorname, Nachname
FROM dbo.tblKunden
WHERE Nachname = N'O''Brien';
```

Platzhalter in LIKE

Auch die Platzhalter ändern sich. In Access suchen wir mit * nach beliebig vielen und mit **?** nach genau einem Zeichen. Die Kunden, deren Nachname mit S beginnt, finden wir dort so:

```
SELECT Nachname FROM dbo_tblKunden
WHERE Nachname Like "S*";
```

In T-SQL heißen die Platzhalter **%** für beliebig viele Zeichen und _ für genau ein Zeichen:

```
SELECT Nachname FROM dbo.tblKunden
WHERE Nachname LIKE N'S%';
```

Der Platzhalter * löst in T-SQL keinen Fehler aus, er ist dort ein gewöhnliches Zeichen. Die Abfrage **WHERE Nachname LIKE N'M*'** sucht daher nach dem wörtlichen Nachnamen M* und liefert keine Datensätze, statt Meier und Müller zu finden. Wer Abfragen aus Access übernimmt, sollte jedes **Like** auf diese Platzhalter hin prüfen.

Datumswerte

In Access schreiben wir Datumsliterale in Rautezeichen und im amerikanischen Format: **#9/15/2026#**. T-SQL kennt diese Schreibweise nicht. Der Server beantwortet sie mit der Meldung **Falsche Syntax in der Nähe von "#"**.

Ein Datum steht in T-SQL als Zeichenkette in einfachen Anführungszeichen. Damit es unabhängig von den Spracheinstellungen des Servers richtig gelesen wird, schreiben wir es als **'20260915'**, also Jahr, Monat und Tag ohne Trennzeichen.

Die zweite Falle betrifft die Uhrzeit. Die Spalte **Bestelldatum** in **tblBestellungen** hat den Datentyp **datetime2(0)** und speichert Datum und Uhrzeit. Die Bestellung vom 15. September 2026 ist um 11:30:00 Uhr eingegangen. Wir zählen die Bestellungen dieses Tages zunächst mit einem Vergleich auf Gleichheit:

```
SELECT COUNT(*) FROM dbo.tblBestellungen
WHERE Bestelldatum = '20260915';
```

Das Ergebnis ist 0, denn **'20260915'** bedeutet Mitternacht, und die Bestellung stammt von 11:30 Uhr. Wir brauchen einen Zeitraum, der den ganzen Tag umfasst:

T-SQL, Teil 2: SELECT in der Praxis

Die Montagsauswertung für den Vertrieb entsteht in Access im Abfrageentwurf: Positionen, Kunden, Umsätze und einige Gruppierungen. Zieht das Backend auf den SQL Server um, laufen Teile davon nicht mehr oder liefern in unserer Konfiguration andere Zahlen. Ein TOP 1 liefert dann eine Zeile statt zweier, ein Durchschnitt verliert seine Nachkommastellen, und eine Summe kommt als Text in Access an. Wer SELECT in T-SQL schreibt, kennt viel davon aus Access. Umdenken müssen wir an einigen Stellen, und einige davon zeigen keine Fehlermeldung, sondern nur andere Ergebnisse. Wie wir die Abfragen schreiben, die im Alltag gebraucht werden, und worauf wir dabei achten, zeigt dieser zweite Teil der Reihe.

Die Auswertungen des Vertriebs

Als Szenario dient wieder die Bestellverwaltung aus **T-SQL, Teil 1: Von Jet-SQL zu T-SQL**. Der Vertrieb möchte wöchentlich sechs Fragen beantwortet haben:

- Welche Positionen hatten den höchsten Betrag?
- Welche Kunden haben noch keine Bestellung aufgegeben?
- Wie viel Umsatz bringen die einzelnen Warengruppen?
- Welche Kunden haben die meisten Bestellungen?
- In welche Klasse fallen die Kunden nach ihrem Umsatz?
- Wie lässt sich die Kundenliste seitenweise anzeigen, damit nicht die gesamte Liste vom Server geholt wird?

Die Beispiele laufen gegen die Datenbank **Bestellverwaltung_TSQL** mit den 330 Bestellungen, 32 Kunden und 825 Positionen aus Teil 1. Wer die Daten verändert hat, führt das Skript **Bestellverwaltung_TSQL.sql** erneut aus. Es legt die Datenbank neu an und löscht dabei eine vorhandene gleichen Namens. Die Hilfsfunktionen **PassThroughOeffnen**, **PassThroughWert**, **SqlText** und **SqlDatum** stammen aus dem Modul **mdlTSQLEinstiegBeispiele** aus Teil 1 und liegen auch im Frontend **TSQL_SELECTInDerPraxis.accdb** bereit. Die neuen VBA-Funktionen dieses Beitrags stehen im Modul **mdlSelectBeispiele**.

Die größten Positionen

Wir beginnen mit den fünf Positionen mit dem höchsten Betrag. Der Betrag einer Position ist **Menge** mal **Einzelpreis** mal (1 minus **Rabatt**). Der **Rabatt** steht als Anteil in der Tabelle, 0,05 entspricht fünf Prozent. Berechnete Spalten benennen wir mit einem Alias, den wir wie in Access mit dem Schlüsselwort **AS** angeben. Die Positionen stehen in **tblBestelldetails**, den Namen der

```
SELECT TOP (5)
    d.BestellungID,
    p.Produkt,
    d.Menge,
    d.Einzelpreis,
    d.Rabatt,
    CAST(d.Menge * d.Einzelpreis * (1 - d.Rabatt)
        AS decimal(10,2)) AS Zeilenbetrag
FROM dbo.tblBestelldetails AS d
    INNER JOIN dbo.tblProdukte AS p ON p.ProduktID = d.ProduktID
ORDER BY Zeilenbetrag DESC, d.BestelldetailID;
```

Listing 1: Die fünf größten Bestellpositionen

Produkte holen wir aus **tblProdukte** (siehe Listing 1).

	BestellungID	Produkt	Menge	Einzelpreis	Rabatt	Zeilenbetrag
1	238	Laptoptasche 15"	10	39.90	0.00	399.00
2	197	Laptoptasche 15"	10	39.90	0.05	379.05
3	226	Laptoptasche 15"	9	39.90	0.00	359.10
4	267	Laptoptasche 15"	9	39.90	0.00	359.10
5	135	Schreibtischlampe LED	10	34.90	0.00	349.00

Bild 1: Die fünf größten Positionen im Ergebnisraster von **SSMS**

Das Ergebnis sind die fünf Positionen mit den Beträgen 399,00 Euro bis 349,00 Euro (siehe Bild 1). **TOP (5)** begrenzt die Ausgabe auf fünf Zeilen. Die **ORDER BY**-Klausel sortiert nach dem Alias **Zeilenbetrag**. Als zweites Sortierkriterium dient **d.BestelldetailID**, damit die Reihenfolge bei gleichen Beträgen festliegt.

Der **CAST** auf **decimal(10,2)** rundet den Betrag auf zwei Nachkommastellen. Bei Aggregatfunktionen hat er eine weitere Aufgabe, die im Abschnitt **Dezimalzahlen in Access** folgt.

Aliase und die Reihenfolge der Auswertung

In **ORDER BY** durften wir den Alias **Zeilenbetrag** verwenden. In **WHERE** funktioniert das nicht. Wir versuchen es mit den Positionen, deren Betrag über 300 Euro liegt:

```
SELECT Menge * Einzelpreis * (1 - Rabatt) AS Betrag
FROM dbo.tblBestelldetails
WHERE Betrag > 300;
```

Der Server meldet **Ungültiger Spaltenname "Betrag"**. Woran liegt das? Er wertet eine Abfrage logisch in einer anderen Reihenfolge aus, als wir die Klauseln hinschreiben. Der tatsächliche Ausführungsplan kann davon noch einmal abweichen. Die logische Reihenfolge lautet:

- **FROM/JOIN:** welche Tabellen verknüpft werden
- **WHERE:** welche Zeilen übrig bleiben
- **GROUP BY:** wie die Zeilen zusammengefasst werden
- **HAVING:** welche Gruppen übrig bleiben
- **SELECT:** welche Spalten entstehen, und damit erst hier auch die Aliase
- **ORDER BY:** wie das Ergebnis sortiert wird, hier sind die Aliase schon bekannt
- **TOP/OFFSET:** wie viele Zeilen davon ausgegeben werden

Ein Alias aus der **SELECT**-Liste ist in **WHERE**, **GROUP BY** und **HAVING** also noch nicht bekannt. In Access ist das nicht anders. Dort fragt die Abfrage allerdings nach einem Parameterwert, statt einen Fehler zu melden. Wir wiederholen deshalb den Ausdruck in **WHERE** oder kapseln die Abfrage als Unterabfrage, davon handelt der nächste Teil.

Auch **DISTINCT** kennen wir aus Access. Es entfernt doppelte Zeilen aus dem Ergebnis. Die Länder der Kunden stehen so nur einmal in der Liste:

```
SELECT DISTINCT Land
FROM dbo.tblKunden
ORDER BY Land;
```

Die Abfrage liefert drei Zeilen, obwohl **tblKunden** 32 Datensätze enthält.

TOP und Gleichstände

Die Frage nach den Kunden mit den meisten Bestellungen führt zu **TOP**. In Access schreiben wir die Abfrage mit **TOP 1** und einer absteigenden Sortierung

```
SELECT TOP (1) WITH TIES
    k.KundeID, k.Nachname, COUNT(*) AS Bestellungen
FROM dbo.tblKunden AS k
     INNER JOIN dbo.tblBestellungen AS b ON b.KundeID = k.KundeID
GROUP BY k.KundeID, k.Nachname
ORDER BY COUNT(*) DESC;
```

Listing 2: WITH TIES liefert die Gleichplatzierten

nach der Zahl der Bestellungen. In unseren Tests lieferte sie zwei Zeilen, weil zwei Kunden jeweils 19 Bestellungen haben. Access gab bei diesem Gleichstand beide Kunden zurück.

In T-SQL liefert **TOP (1)** ohne weitere Angabe höchstens eine Zeile. Welcher der beiden Kunden das ist, steht ohne ein weiteres Sortierkriterium nicht fest. Wer Gleichstände wie Access behandeln möchte, verwendet **WITH TIES** (siehe Listing 2).

Das Ergebnis sind die Kunden Schröder und Vogel mit je 19 Bestellungen. **WITH TIES** setzt eine **ORDER BY**-Klausel voraus, und diese Klausel bestimmt, was als Gleichstand zählt. Hängen wir dort **k.KundeID** an, sind die beiden nicht mehr gleich platziert, und **WITH TIES** liefert nur eine Zeile.

Statt einer Anzahl lässt sich mit **TOP (n) PERCENT** auch ein Prozentsatz der Zeilen angeben.

Die Kundenliste seitenweise holen

Die Kundenliste soll ein Formular seitenweise zeigen. **TOP** liefert nur die ersten Zeilen und kennt keine Seiten. Dafür gibt es **OFFSET** und **FETCH**: **OFFSET** überspringt die angegebene Zahl von Zeilen, **FETCH NEXT** holt die folgenden.

Die zweite Seite mit zehn Kunden je Seite lautet so:

```
SELECT KundeID, Nachname, Ort
FROM dbo.tblKunden
ORDER BY Nachname, KundeID
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY;
```

Das Ergebnis sind die Kunden Köhler bis Neumann. **OFFSET** gehört zu **ORDER BY** und ist ohne diese Klausel nicht erlaubt. Die Sortierung muss außerdem eindeutig sein, sonst können Datensätze beim Blättern doppelt oder gar nicht erscheinen. Haben zwei Kunden denselben Nachnamen, entscheidet **KundeID** als zweites Kriterium, auf welcher Seite sie stehen.

Das setzt voraus, dass sich die Daten zwischen den Seitenaufrufen nicht verändern. Neue, gelöschte oder umsortierte Datensätze können die Seitengrenzen verschieben.

In VBA setzen wir die Anweisung aus der Seitenzahl und der Zahl der Zeilen je Seite zusammen. Das erledigt die Funktion **KundenSeite**. Sie setzt die Seitenzahl auf 1, wenn sie kleiner ist, und löst bei einer Seitengröße unter 1 einen Fehler aus, weil **FETCH NEXT** mindestens eine Zeile verlangt (siehe Listing 3).

```
Public Function KundenSeite(lngSeite As Long, lngProSeite As Long) As DAO.Recordset
    Dim strSQL As String
    If lngSeite < 1 Then lngSeite = 1
    If lngProSeite < 1 Then
        Err.Raise 5, "KundenSeite", "Mindestens 1 Zeile je Seite"
    End If
    strSQL = "SELECT KundeID, Nachname, Ort FROM dbo.tblKunden " & _
             "ORDER BY Nachname, KundeID " & _
             "OFFSET " & (lngSeite - 1) * lngProSeite & " ROWS " & _
             "FETCH NEXT " & lngProSeite & " ROWS ONLY"
    Set KundenSeite = PassThroughOeffnen(strSQL)
End Function
```

Listing 3: KundenSeite holt eine Seite der Kundenliste

Der Ausdruck **(lngSeite - 1) * lngProSeite** liefert den Wert für **OFFSET**. Die Funktion **PassThrough-Oeffnen** aus Teil 1 schickt den Text als temporäre Pass-Through-Abfrage an den Server und gibt das Recordset zurück.

Zum Ausprobieren gibt die Prozedur **KundenSeite-Ausgeben** eine Seite im Direktbereich aus (siehe Listing 4).

Der Aufruf **KundenSeiteAusgeben 2, 10** liefert die zehn Kunden der zweiten Seite (siehe Bild 2). Seite 4 enthält die letzten beiden Kunden, Seite 5 ist leer.

```
KundenSeiteAusgeben 2, 10
 18             Köhler          Kiel
 13             Krause          Bremen
 26             Lange           Kassel
 6              Lehmann         Nürnberg
 11             Lindner         Regensburg
 27             Maier           Düsseldorf
 3              Meier           Stuttgart
 22             Meyer           Tübingen
 1              Müller          München
 12             Neumann         Hannover
```

Bild 2: Die zweite Seite der Kundenliste, ausgegeben von **KundenSeiteAusgeben** im Direktbereich

JOIN ohne Klammern

Bei Verknüpfungen unterscheiden sich Access und T-SQL deutlich. Der Abfrageentwurf von Access setzt um die bisherigen Verknüpfungen eine Klammer, sobald ein weiterer **INNER JOIN** hinzukommt. Ohne diese Klammern meldet die Datenbank-Engine einen Syntaxfehler.

Die Abfrage, die die Positionen über fünf Tabellen zählt, sieht in der SQL-Ansicht so aus (siehe Listing 5).

Lassen wir die Klammern weg, meldet Access den Fehler 3075, dessen Text mit **Syntaxfehler (fehlender Operator) in Abfrageausdruck** beginnt.

T-SQL braucht keine Klammern. Die Verknüpfungen stehen hintereinander, und sie beziehen sich auf die Tabellen davor (siehe Listing 6).

Der Wert **Anzahl** ist wie in der Access-Fassung 825. Das Schlüsselwort **INNER** dürfen wir weglassen, **JOIN** allein bedeutet dasselbe. In diesem Beitrag steht es ausgeschrieben, damit der Unterschied zu den äußeren Verknüpfungen erkennbar bleibt.

Kunden ohne Bestellung

Für die Frage nach den Kunden ohne Bestellung brauchen wir den **LEFT JOIN**. Er behält die Zeilen der linken Tabelle, auch wenn in der rechten nichts dazu passt. Ausgeschrieben heißt er **LEFT OUTER JOIN**. Die Spalten der rechten Tabelle sind für solche Zeilen **NULL**, und danach fragen wir in **WHERE** (siehe Listing 7).

```
Public Sub KundenSeiteAusgeben(lngSeite As Long, lngProSeite As Long)
    Dim rst As DAO.Recordset
    Set rst = KundenSeite(lngSeite, lngProSeite)
    Do While Not rst.EOF
        Debug.Print rst!KundeID, rst!Nachname, rst!Ort
        rst.MoveNext
    Loop
    rst.Close
End Sub
```

Listing 4: **KundenSeiteAusgeben** zeigt eine Seite der Kundenliste an

```
SELECT Count(*) AS Anzahl
FROM (((dbo_tblBestelldetails AS d
  INNER JOIN dbo_tblBestellungen AS b ON d.BestellungID = b.BestellungID)
  INNER JOIN dbo_tblKunden AS k ON b.KundeID = k.KundeID)
  INNER JOIN dbo_tblProdukte AS p ON d.ProduktID = p.ProduktID)
  INNER JOIN dbo_tblKategorien AS c ON p.KategorieID = c.KategorieID;
```

Listing 5: Fünf Tabellen in Access-SQL mit Klammern

Die Abfrage liefert drei Kunden: Peters, Kleinschmidt und Zimmermann. Im Frontend liegt sie als Pass-Through-Abfrage **ptKundenOhneBestellung** vor (siehe Bild 3).

```
SELECT COUNT(*) AS Anzahl
FROM dbo.tblBestelldetails AS d
     INNER JOIN dbo.tblBestellungen AS b ON b.BestellungID = d.BestellungID
     INNER JOIN dbo.tblKunden AS k ON k.KundeID = b.KundeID
     INNER JOIN dbo.tblProdukte AS p ON p.ProduktID = d.ProduktID
     INNER JOIN dbo.tblKategorien AS c ON c.KategorieID = p.KategorieID;
```

Listing 6: Dieselbe Verknüpfung in T-SQL, ohne Klammern

Für den Einsatz in VBA genügt oft die Zahl dieser Kunden. Die Funktion **KundenOhneBestellungZaehlen** schickt dieselbe Bedingung mit **COUNT(*)** an den Server und holt den Wert mit **PassThroughWert** (siehe Listing 8).

```
SELECT k.KundeID, k.Nachname, k.Ort
FROM dbo.tblKunden AS k
     LEFT JOIN dbo.tblBestellungen AS b ON b.KundeID = k.KundeID
WHERE b.BestellungID IS NULL
ORDER BY k.KundeID;
```

Listing 7: Kunden ohne Bestellung per **LEFT JOIN**

Im Direktbereich liefert **?KundenOhneBestellungZaehlen()** die 3.

KundeID	Nachname	Ort
30	Peters	Lübeck
31	Kleinschmidt	Bremen
32	Zimmermann	Erfurt

Bild 3: Die Pass-Through-Abfrage **ptKundenOhneBestellung** in der Datenblattansicht

Bedingungen im ON oder im WHERE

Bei einem **LEFT JOIN** entscheidet der Ort einer Bedingung über das Ergebnis. Das zeigt ein Beispiel: Wir wollen für die Kunden die Zahl der Bestellungen seit Jahresbeginn 2026, auch für Kunden ohne eine solche Bestellung. Im ersten Versuch steht die Bedingung im **ON** (siehe Listing 9).

Das Ergebnis hat 32 Zeilen, also einen Eintrag je Kunde. Wer 2026 nicht bestellt hat, steht mit einer 0 in der Liste. Im zweiten Versuch verschieben wir die Bedingung in die **WHERE**-Klausel (siehe Listing 10).

```
Public Function KundenOhneBestellungZaehlen() As Long
    Dim strSQL As String
    strSQL = "SELECT COUNT(*) FROM dbo.tblKunden k " & _
             "LEFT JOIN dbo.tblBestellungen b ON b.KundeID = k.KundeID " & _
             "WHERE b.BestellungID IS NULL"
    KundenOhneBestellungZaehlen = Nz(PassThroughWert(strSQL), 0)
End Function
```

Listing 8: **KundenOhneBestellungZaehlen** zählt die Kunden ohne Bestellung

Jetzt sind es 27 Zeilen. Woran liegt das? **WHERE** wirkt auf das Ergebnis der Verknüpfung. Für die drei Kunden ohne Bestellung ist **Bestelldatum** dort **NULL**, und ein Vergleich mit **NULL** ist weder wahr noch falsch. Die Zeilen fallen heraus. Dasselbe gilt für die zwei Kunden, deren Bestellungen vor 2026 liegen. Allgemein entfernt ein **WHERE**-Kriterium auf der rechten Tabelle, das **NULL** ausschließt, auch die Kunden ohne passende Bestellung. Die Ab-

```
SELECT k.KundeID, COUNT(b.BestellungID) AS Bestellungen
FROM dbo.tblKunden AS k
     LEFT JOIN dbo.tblBestellungen AS b
          ON b.KundeID = k.KundeID AND b.Bestelldatum >= '20260101'
GROUP BY k.KundeID;
```

Listing 9: Die Bedingung steht im **ON**

```
SELECT m.Vorname + N' ' + m.Nachname AS Mitarbeiter,
       m.Funktion,
       ISNULL(v.Vorname + N' ' + v.Nachname, N'(oberste Ebene)') AS Vorgesetzter
FROM dbo.tblMitarbeiter AS m
     LEFT JOIN dbo.tblMitarbeiter AS v ON v.MitarbeiterID = m.VorgesetzterID
ORDER BY m.MitarbeiterID;
```

Listing 11: Selbstverknüpfung: Mitarbeiter und ihre Vorgesetzten

frage verhält sich dann hinsichtlich dieser Zeilen wie ein **INNER JOIN**. Steht die Bedingung im **ON**, bleiben die Zeilen der linken Tabelle erhalten.

Selbstverknüpfung

Eine Tabelle lässt sich auch mit sich selbst verknüpfen. In **tblMitarbeiter** zeigt das Fremdschlüsselfeld **VorgesetzterID** auf **MitarbeiterID** derselben Tabelle. Um zu einem Mitarbeiter den Namen des Vorgesetzten zu bekommen, verwenden wir die Tabelle zweimal unter verschiedenen Aliasen. Mit **LEFT JOIN** bleibt die Geschäftsführung ohne Vorgesetzten im Ergebnis (siehe Listing 11).

Das Ergebnis hat acht Zeilen. Bei Claudia Hoffmann steht **(oberste Ebene)**, denn für sie ist **VorgesetzterID** leer und die rechte Seite des **LEFT JOIN NULL**.

Weitere Verknüpfungsarten brauchen wir seltener. **RIGHT JOIN** ist ein **LEFT JOIN** mit vertauschten Tabellen und lässt sich umschreiben. **FULL OUTER JOIN** behält die Zeilen beider Seiten und eignet sich für den Abgleich zweier Tabellen, etwa eines Imports gegen den Bestand. **CROSS JOIN** bildet das Kreuzprodukt beider Tabellen und kommt im nächsten Teil vor.

```
SELECT k.KundeID, COUNT(b.BestellungID) AS Bestellungen
FROM dbo.tblKunden AS k
     LEFT JOIN dbo.tblBestellungen AS b ON b.KundeID = k.KundeID
WHERE b.Bestelldatum >= '20260101'
GROUP BY k.KundeID;
```

Listing 10: Dieselbe Bedingung im **WHERE**

Umsatz je Warengruppe

Nun zum Umsatz der einzelnen Warengruppen. Beim Gruppieren gelten in T-SQL dieselben Grundregeln wie in Access. Eine Spalte der **SELECT**-Liste steht entweder in **GROUP BY** oder in einer Aggregatfunktion wie **SUM**, **COUNT**, **AVG**, **MIN** oder **MAX**. Im Abfrageentwurf von Access nehmen wir die Zuordnung über die Zeile **Funktion** vor. In T-SQL schreiben wir **GROUP BY** und die Aggregatfunktionen hin.

Wir werten das Jahr 2025 aus, ohne stornierte Bestellungen, und behalten nur Gruppen mit mehr als 500 Euro Umsatz (siehe Listing 12).

Das Ergebnis zeigt sieben Warengruppen, die größte ist **Zubehör** mit 7.205,65 Euro.

Der Unterschied zwischen **WHERE** und **HAVING** ergibt sich aus der Auswertungsreihenfolge. **WHERE** wirkt auf die einzelnen Zeilen vor dem Gruppieren, **HAVING** auf die fertigen Gruppen. Eine Bedingung auf eine Aggregatfunktion in **WHERE** führt zu einer Fehlermeldung. Eine Bedingung auf ein Gruppierungsfeld gehört in **WHERE**, weil dann weniger Zeilen in die Gruppierung gehen.

COUNT: Zeilen und Werte

Bei **COUNT** kommt es darauf an, was gezählt wird. **COUNT(*)** zählt Zeilen, **COUNT(Spalte)** zählt die Zeilen, in denen die Spalte nicht **NULL** ist. Eine leere Zeichenkette ist nicht **NULL** und wird mitgezählt.

Die folgende Abfrage zählt in **tblBestellungen** 330 Bestellungen, davon 14 mit einer **Bemerkung** und 312 mit einem **Lieferdatum**:

SQL Server: datetime und datetime2 in Access

Wir verknüpfen eine Tabelle des SQL Servers mit Access, öffnen den Direktbereich und lesen mit DLookup ein Datum aus der ersten Zeile. Bei einem Feld liefert der Aufruf den Wert, bei einem anderen scheitert derselbe Aufruf mit der Meldung Typen unverträglich. Beide Felder enthalten ein Datum mit Uhrzeit. Der Unterschied liegt im Datentyp: Der SQL Server kennt mehrere Datentypen für Datum und Uhrzeit, und Access bindet sie je nach Einstellung unterschiedlich ein. Welche Konstellationen ohne Probleme laufen, welche Fehler auftreten und wie wir die Anbindung einrichten, zeigt dieser Beitrag.

Beispieldatenbank

Die Beispiele dieses Beitrags laufen gegen die SQL Server-Datenbank **DateTimeTest**. Das Skript **DateTimeTest.sql** legt sie auf einer lokalen Instanz an, in unserem Fall **.SQLEXPRESS**.

Es löscht eine vorhandene Datenbank gleichen Namens und erzeugt sie neu.

Die Datenbank enthält zwei Tabellen und eine Sicht. Die Tabelle **tblZeitstempel** hat neben **ZeitstempelID** und **Bezeichnung** je eine Spalte für jeden Datentyp, den wir untersuchen wollen:

- **ZDate:** Datentyp **date**
- **ZSmallDateTime:** Datentyp **smalldatetime**
- **ZDateTime:** Datentyp **datetime**
- **ZDateTime2_0:** Datentyp **datetime2(0)**
- **ZDateTime2_3:** Datentyp **datetime2(3)**
- **ZDateTime2_7:** Datentyp **datetime2(7)**

Sie enthält drei Zeilen. Die erste enthält den 2. Oktober 2026 um 8:30:15 Uhr. Bei **ZDate** fehlt die Uhrzeit, **ZSmallDateTime** steht auf 8:30:00, weil der Typ nur Minuten speichert. **ZDateTime** und **ZDateTime2_3** tragen die Sekundenbruchteile **.123**, **ZDateTime2_7** die Sekundenbruchteile **.1234567**.

Die zweite Zeile hat den 15. September 2026 um 11:30:05 Uhr, die dritte den 1. Januar 2026 ohne Uhrzeit. Das Ergebnis in SQL Server Management Studio steht im Ergebnisraster (siehe Bild 1).

Die Tabelle **tblSchreibtest** hat die vier Spalten **ZDateTime**, **ZDateTime2_0**, **ZDateTime2_3** und **ZDateTime2_7**. Wir verwenden sie für die Schreibversuche. Die Sicht **vwZeitstempel** liefert dieselben Daten wie **tblZeitstempel**, wandelt aber die drei **datetime2**-Spalten mit **CAST** in **datetime** um.

Das Frontend **SQLServer_DateTimeUndDateTime2InAccess.accdb** enthält die Tabellen als Verknüpfungen.

Die Verknüpfungen mit dem Zusatz **Normal** im Namen verwenden die Einstellung, die wir weiter unten als Konstellation 3 beschreiben, die übrigen die Stan-

	ZeitstempelID	Bezeichnung	ZDate	ZSmallDateTime	ZDateTime	ZDateTime2_0	ZDateTime2_3	ZDateTime2_7
1	1	Mit Sekundenbruchteil	2026-10-02	2026-10-02 08:30:00	2026-10-02 08:30:15.123	2026-10-02 08:30:15	2026-10-02 08:30:15.123	2026-10-02 08:30:15.1234567
2	2	Ganze Sekunden	2026-09-15	2026-09-15 11:30:00	2026-09-15 11:30:05.000	2026-09-15 11:30:05	2026-09-15 11:30:05.000	2026-09-15 11:30:05.0000000
3	3	Ohne Uhrzeit	2026-01-01	2026-01-01 00:00:00	2026-01-01 00:00:00.000	2026-01-01 00:00:00	2026-01-01 00:00:00.000	2026-01-01 00:00:00.0000000

Bild 1: Die Tabelle **tblZeitstempel** im Ergebnisraster von **SSMS**



Bild 2: Die Option für **datetime2** in den Access-Optionen

dardeinstellung. Die VBA-Prozeduren stehen im Modul **mdlDateTimeBeispiele**, ein Testmodul prüft die Aussagen dieses Beitrags.

Datums- und Zeittypen im SQL Server

Für Datum und Uhrzeit bietet der SQL Server vier Datentypen an, die wir hier betrachten:

- **date:** nur das Datum, vom 1. Januar 0001 bis zum 31. Dezember 9999
- **smalldatetime:** Datum und Uhrzeit auf die Minute genau, vom 1. Januar 1900 bis zum 6. Juni 2079
- **datetime:** Datum und Uhrzeit, vom 1. Januar 1753 bis zum 31. Dezember 9999, in Schritten von etwa 3 Millisekunden
- **datetime2(n):** Datum und Uhrzeit, vom 1. Januar 0001 bis zum 31. Dezember 9999, mit **n** Nachkommastellen für die Sekunden, wobei **n** zwischen 0 und 7 liegt

Den Typ **datetime** gibt es seit den frühen Versionen des SQL Servers, **datetime2** ist neuer. Er hat einen größeren Bereich, eine höhere Genauigkeit und entspricht dem SQL-Standard. Die Dokumentation von Microsoft zu **datetime** empfiehlt für neue Arbeiten die Typen **date**, **time**, **datetime2** und **datetimeoffset**.

Für Access sieht das anders aus. Hier kommt es darauf an, wie Access die Typen einbindet, wenn wir eine Tabelle verknüpfen.

Wie Access die Typen einbindet

Access kennt für Datum und Uhrzeit zwei Felddatentypen. **Datum/Uhrzeit** speichert den Wert intern als Zahl vom Typ **Double**.

Der Teil vor dem Komma ist die Zahl der Tage seit dem 30. Dezember 1899, der Teil hinter dem Komma der Anteil des Tages. Daneben gibt es seit Access 2021 und Access in Microsoft 365 den Typ **Datum/Uhrzeit, erweitert**, der Sekundenbruchteile mit bis zu sieben Stellen speichert. Er ist nicht mit älteren Access-Versionen kompatibel.

Der erweiterte Typ wurde vor allem für die Anbindung des SQL Servers eingeführt. Er bildet den Typ **datetime2** nach. Ob Access ihn beim Verknüpfen verwendet, steuert eine Option der aktuellen Datenbank. Wir finden sie in den Access-Optionen unter **Aktuelle Datenbank** im Bereich **Datentypunterstützungs-Optionen**, sie heißt **Unterstützung des erweiterten Datentyps „Date Time Extended" (DateTime2) für mit ODBC verknüpfte/importierte Tabellen** (siehe Bild 2).

Die Option wirkt beim Verknüpfen oder Importieren. Eine bestehende Verknüpfung ändert sich nicht, wenn wir die Option später umschalten, bis wir sie neu verknüpfen. In unseren Tests verhielt sich eine neue Datenbank, in der wir die Option nie geändert hatten, wie bei eingeschalteter Option.

Was daraus folgt, sehen wir an den Verknüpfungen mit **DAO**. Die Funktion **FeldTypen** liefert zu jedem Feld

```
Public Function FeldTypen(strTabelle As String) As String
    Dim db As DAO.Database
    Dim fld As DAO.Field
    Set db = CurrentDb
    For Each fld In db.TableDefs(strTabelle).Fields
        FeldTypen = FeldTypen & fld.Name & "=" & fld.Type & " "
    Next fld
    FeldTypen = Trim(FeldTypen)
End Function
```

Listing 1: FeldTypen liefert die Felder einer Tabelle mit dem DAO-Datentyp

einer Tabelle den DAO-Datentyp (siehe Listing 1). Der Wert 8 steht für **Datum/Uhrzeit**, der Wert 26 für **Datum/Uhrzeit, erweitert**.

Für die Standardverknüpfung **dbo_tblZeitstempel** erhalten wir im Direktbereich diese Zeile, hier zur besseren Lesbarkeit umgebrochen:

```
?FeldTypen("dbo_tblZeitstempel")
ZeitstempelID=4 Bezeichnung=10 ZDate=8
ZSmallDateTime=8 ZDateTime=8
ZDateTime2_0=26 ZDateTime2_3=26
ZDateTime2_7=26
```

Die Typen **date**, **smalldatetime** und **datetime** kommen als **Datum/Uhrzeit** an, alle drei **datetime2**-Felder als **Datum/Uhrzeit, erweitert**.

Die Tabelle im Entwurf zeigt denselben Unterschied (siehe Bild 3).

Damit ergeben sich drei Konstellationen, die wir nacheinander durchgehen: **datetime**, **smalldatetime** und **date** in Access, **datetime2** bei eingeschalteter Option und **datetime2** bei ausgeschalteter Option.

Konstellation 1: datetime, smalldatetime und date

Die Typen **date**, **smalldatetime** und **datetime** verhalten sich in Access wie ein normales Datum/Uhrzeit-Feld. Das gilt in unseren Tests unabhängig von der Option. Ein Aufruf von **DLookup** auf das Feld **ZDateTime** liefert den Wert wie bei einer Access-Tabelle:

```
?DLookup("ZDateTime","dbo_tblZeitstempel")
02.10.2026 08:30:15
```

Der Wert in der Tabelle lautet 8:30:15.123 Uhr. Access zeigt die Sekundenbruchteile nicht an. **Format**, **CDate**, **Nz** und **DateAdd** arbeiten mit dem Feld wie bei einer Access-Tabelle.

Bei **ZDate** und **ZSmallDateTime** gelten in unseren Tests dieselben Ergebnisse. Der Aufruf ohne Kriterium liest den ersten Datensatz in der Reihenfolge der Tabelle.

Wer in einer Datenbank, auf die vor allem Access zugreift, keine Sekundenbruchteile braucht, kommt mit diesen Typen ohne Einstellungen aus.

Die Einschränkungen des Typs **datetime** sind die Schritte von etwa 3 Millisekunden und der Bereich ab dem Jahr 1753.

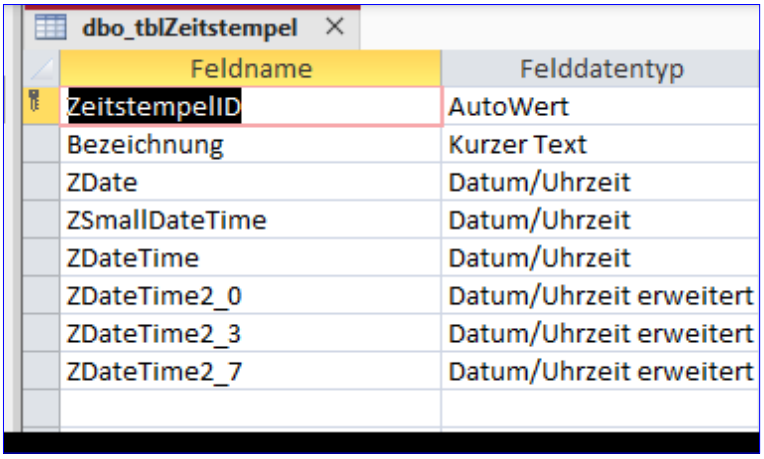


Bild 3: Die Felder der verknüpften Tabelle im Entwurf mit den Datentypen

```
Set rst = CurrentDb.OpenRecordset("SELECT ZDateTime2_7 FROM dbo_tblZeitstempel WHERE ZeitstempelID=1", 4, 512)
?rst.Fields(0).Value
10/02/2026 08:30:15.1234567
?CDate(Left(rst.Fields(0).Value, 19))
10.02.2026 08:30:15
```

Bild 5: **CDate** liest den Text des Feldes mit vertauschtem Tag und Monat

Konstellation 2: datetime2 bei eingeschalteter Option

Bei eingeschalteter Option, in unseren Tests die Standardeinstellung, kommt **datetime2** als **Datum/Uhrzeit, erweitert** an. Hier treten die Fehler auf. Wir gehen sie der Reihe nach durch, beginnend mit der Funktion aus der Einleitung.

Der Aufruf von **DLookup** auf ein **datetime2**-Feld liefert einen Fehler:

```
?DLookup("ZDateTime2_3","dbo_tblZeitstempel")
```

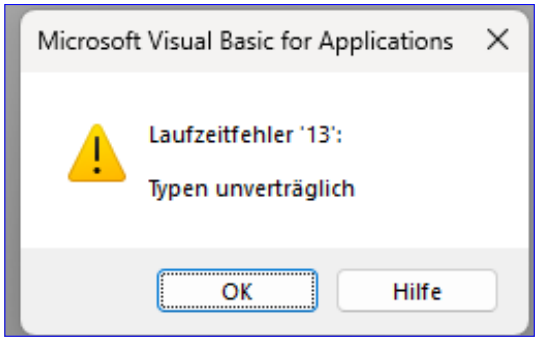


Bild 4: Der Aufruf von **DLookup** auf ein **datetime2**-Feld meldet **Typen unverträglich**

Es erscheint die Meldung **Laufzeitfehler 13: Typen unverträglich** (siehe Bild 4). Der Aufruf liefert einen Wert, den Access nicht ausgeben kann. Auch die Zuweisung an eine **Date**-Variable oder an eine **String**-Variable scheitert mit dem Fehler 13. Dasselbe gilt in unseren Tests für **DMax**, **DMin** und **DFirst**.

Eine Verwendung der Domänenfunktionen ist trotzdem möglich. Die Funktion **DCount** zählt Datensätze auch dann, wenn das Kriterium ein **datetime2**-Feld enthält. Der folgende Aufruf zählt die Zeilen vom 2. Oktober 2026 und liefert 1:

```
?DCount("*","dbo_tblZeitstempel", _
  "ZDateTime2_3>=#10/2/2026# AND " & _
  "ZDateTime2_3<#10/3/2026#")
```

Eine Datumsbedingung auf einem **datetime2**-Feld funktioniert also, nur der Wert des Feldes lässt sich nicht lesen.

Recordset und CDate bei datetime2

Mit einem Recordset lässt sich der Feldwert lesen. Er kommt allerdings als Text an. Die folgende Abfrage liest das Feld **ZDateTime2_7** der ersten Zeile:

```
Set rst = CurrentDb.OpenRecordset( _
  "SELECT ZDateTime2_7 " & _
  "FROM dbo_tblZeitstempel " & _
  "WHERE ZeitstempelID=1", _
  dbOpenSnapshot, dbSeeChanges)
?TypeName(rst.Fields(0).Value)
String
?rst.Fields(0).Value
10/02/2026 08:30:15.1234567
```

Der Text hat in unseren Tests mit deutschen Regionaleinstellungen das amerikanische Format Monat, Tag, Jahr, dazu sieben Nachkommastellen für die Sekunden.

Gemeint ist hier der 2. Oktober 2026. Die Funktion **CDate** kann diesen Text mit den Nachkommastellen nicht in ein Datum umwandeln und löst den Fehler 13 aus.

SQL Server: PDFs in varbinary(max)-Feld speichern

Rechnungen, Angebote und Lieferscheine liegen als PDF-Dateien auf einem Netzlaufwerk, während in der Datenbank nur der Pfad zur Datei steht. Wird eine Datei verschoben, umbenannt oder gelöscht, verweist der Pfad auf keine Datei mehr. Auch ein Backup der Datenbank enthält die Dokumente nicht. Der SQL Server kann die Dateien dagegen in der Datenbank speichern. Dafür nimmt der Datentyp varbinary(max) Binärdaten bis zu einer Größe von knapp 2 Gigabyte auf, auch ein PDF-Dokument. In diesem Beitrag zeigen wir, wie wir ein PDF-Dokument mit ADODB in ein Feld dieses Datentyps schreiben, wieder in eine Datei auf der Festplatte speichern und in einem Access-Formular im Edge-Steuerelement anzeigen.

Szenario und Beispieldatenbank

Zu den Datensätzen einer kleinen Dokumentenverwaltung soll jeweils ein PDF-Dokument gehören, zum Beispiel eine Rechnung, ein Angebot oder ein Lieferschein.

Die Beispieldatenbank **PDFInVarCharMax** enthält dazu eine Tabelle mit drei Feldern:

- **PDFID:** der Primärschlüssel vom Datentyp **int** mit der Eigenschaft **IDENTITY**, die dem AutoWert in Access entspricht
- **Dateiname:** der Name der Datei vom Datentyp **nvarchar(255)**
- **Dateiinhalt:** der Inhalt der Datei vom Datentyp **varbinary(max)**

Die Tabelle legen wir mit der folgenden Anweisung an, wobei **dbo** das Standardschema des SQL Servers ist. Das Skript **PDFInVarCharMax.sql** aus dem Download legt zusätzlich die Datenbank an.

Wir führen es im SQL Server Management Studio (SSMS) aus. Das Skript prüft vor jeder Anweisung, ob das Objekt schon existiert. Deshalb können wir es mehrmals ausführen.

```
CREATE TABLE dbo.tblPDFs (
    PDFID int IDENTITY(1, 1) NOT NULL
        CONSTRAINT PK_tblPDFs PRIMARY KEY,
    Dateiname nvarchar(255) NOT NULL,
    Dateiinhalt varbinary(max) NULL
);
```

Das Feld **Dateiinhalt** erlaubt den Wert **NULL**. Damit können wir auch Datensätze anlegen, die kein PDF-Dokument enthalten. Diesen Fall behandeln wir im Formular zur Anzeige der PDF-Dokumente gesondert.

Der Datentyp varbinary(max)

Der Datentyp **varbinary(n)** speichert höchstens 8.000 Byte. Geben wir statt **n** den Wert **max** an, nimmt das Feld bis zu 2.147.483.647 Byte auf, was knapp 2 Gigabyte entspricht.

Der SQL Server speichert die Datei als Folge von Bytes, ohne ihre Struktur zu kennen. Dass es sich um ein PDF-Dokument handelt, erkennen wir in der Tabelle nur am Dateinamen.

Die Länge eines Werts in Byte liefert die Funktion **DATALENGTH()**. Enthält das Feld den Wert **NULL**, liefert sie ebenfalls **NULL**. Die folgende Abfrage zeigt die Größe der gespeicherten Dateien:

```
Public Function Verbindung(Optional ByVal strDatenbank As String = cstrDatenbank) As String
    Verbindung = "Driver={ODBC Driver 18 for SQL Server};Server=" & cstrServer & _
        ";Database=" & strDatenbank & _
        ";Trusted_Connection=Yes;Encrypt=yes;TrustServerCertificate=yes;"
End Function
```

Listing 1: Verbindung liefert die Verbindungszeichenfolge

```
SELECT PDFID, Dateiname,
       DATALENGTH(Dateiinhalt) AS Groesse
FROM dbo.tblPDFs;
```

Ein Feld mit der Länge 0 Byte, geschrieben als **0x**, unterscheidet sich vom Wert **NULL**, der für einen fehlenden Wert steht.

Für die Anzeige im Formular behandeln wir beide Fälle gleich. Enthält das Feld keine Bytes, haben wir kein PDF-Dokument gespeichert.

Die Verbindung zum SQL Server

Die Beispiele verwenden **ADODB**. Dafür braucht das Projekt einen Verweis auf die **Microsoft ActiveX Data Objects 6.1 Library**. Die Funktionen und Prozeduren dieses Beitrags stehen im Modul **mdlPDFInVarBinaryMax**.

Am Anfang des Moduls legen wir den Server und die Datenbank als Konstanten fest. Hier tragen wir den Namen des eigenen Servers ein. In den Beispielen heißt er **amvDesktop2023**:

```
Private Const cstrServer As String = "amvDesktop2023"
Private Const cstrDatenbank _
    As String = "PDFInVarCharMax"
```

Die Funktion **Verbindung** setzt daraus die Verbindungszeichenfolge für den ODBC-Treiber zusammen (siehe Listing 1).

Mit **Trusted_Connection=Yes** verwenden wir die Windows-Anmeldung.

Der ODBC-Treiber 18 verschlüsselt die Verbindung standardmäßig, wobei **TrustServerCertificate=yes** die Prüfung des Serverzertifikats unterdrückt. Das genügt für die Beispiele. Auf einem Produktivserver installieren wir ein vertrauenswürdiges Zertifikat. Übergeben wir der Funktion keinen Datenbanknamen, verwendet sie die Konstante **cstrDatenbank**.

Die Datenbank und die Tabelle per VBA-Prozedur anlegen

Wer das Skript nicht im SSMS ausführen möchte, legt die Datenbank und die Tabelle per VBA-Prozedur an. Die Prozedur **DatenbankAnlegen** verbindet sich dazu mit der Systemdatenbank **master** und legt die Datenbank an (siehe Listing 2).

Anschließend öffnet sie eine Verbindung zu dieser Datenbank und legt die Tabelle an. Beide Anweisungen führt sie nur aus, falls das Objekt noch nicht existiert. Für die Datenbank braucht das Konto, mit dem wir uns anmelden, das Recht **CREATE DATABASE**. Die Prozedur setzt den Datenbanknamen unverändert in die Anweisung ein. Daher übergeben wir nur eigene, feste Namen.

Im Direktbereich rufen wir die Prozedur auf. Fehlen die Datenbank und die Tabelle, legt der Aufruf beide an. Existieren sie bereits, ändert er nichts:

```
DatenbankAnlegen
```

Ein PDF-Dokument speichern

Die Funktion **PDFInSQLSpeichern** liest eine Datei und schreibt sie in die Tabelle (siehe Listing 3). Zuerst

```
Public Sub DatenbankAnlegen(Optional ByVal strDatenbank As String = cstrDatenbank)
    Dim cnn As ADODB.Connection
    Set cnn = New ADODB.Connection
    cnn.Open Verbindung("master")
    cnn.Execute "IF DB_ID(N'" & strDatenbank & "') IS NULL " & _
        "CREATE DATABASE [" & strDatenbank & "]"
    cnn.Close
    cnn.Open Verbindung(strDatenbank)
    cnn.Execute "IF OBJECT_ID(N'dbo.tblPDFs', N'U') IS NULL " & _
        "CREATE TABLE dbo.tblPDFs (" & _
        "PDFID int IDENTITY(1, 1) NOT NULL CONSTRAINT PK_tblPDFs PRIMARY KEY, " & _
        "Dateiname nvarchar(255) NOT NULL, " & _
        "Dateiinhalt varbinary(max) NULL)"
    cnn.Close
End Sub
```

Listing 2: DatenbankAnlegen legt die Datenbank und die Tabelle an

prüft sie mit der Methode **FileExists** des Objekts **FileSystemObject**, ob die Datei existiert. Falls nicht, löst sie einen benutzerdefinierten Fehler aus (**vbObjectError + 1000**).

Danach öffnet sie einen **ADODB.Stream** vom Typ **adTypeBinary** und lädt die Datei mit **LoadFromFile**. Den gesamten Inhalt liest sie mit **Read** in eine Variant-Variable. Bei einer leeren Datei liefert **Read** den Wert **Null**. Dann löst die Funktion den benutzerdefinierten Fehler **vbObjectError + 1001** aus.

Den Dateinamen ermittelt die Funktion mit der Methode **GetFileName** aus dem Pfad. Die Anweisung **INSERT** führt sie über ein Objekt vom Typ **ADODB.Command** mit zwei Parametern aus. Der Parameter für den Dateinamen hat den Typ **adVarWChar**, der Parameter für den Inhalt den Typ **adLongVarBinary** mit der Länge des Byte-Arrays. Den Inhalt übergeben wir als Parameter, weil ein Hexadezimalliteral in der SQL-Anweisung bei großen Dateien unpraktisch wäre.

Die Anweisung beginnt mit **SET NOCOUNT ON**. Damit sendet der Server keine Meldung über die Zahl der geänderten Zeilen, und das Recordset enthält das Ergebnis des **SELECT**. Dieses **SELECT** gibt mit **SCOPE_IDENTITY()** die neue **PDFID** zurück. Die Funktion liefert sie als Ergebnis.

Mit dem optionalen Parameter **strConnection** können wir eine andere Verbindungszeichenfolge übergeben.

Übergeben wir keine, verwendet die Funktion die Zeichenfolge, die **Verbindung** liefert.

Im Direktbereich speichern wir die Datei **test.pdf** aus dem Ordner der Datenbank:

```
?PDFInSQLSpeichern(CurrentProject.Path & "\test.pdf")
 1
```

In einer neuen Tabelle erhält das Dokument die **PDFID** 1. Die Abfrage mit **DATALENGTH()** liefert für diesen Datensatz den Wert **155.264**, die Größe der Datei in Byte. Für die Bilder speichern wir auf demselben Weg die Datei **vbe_517.pdf** als zweiten Datensatz. Den dritten Datensatz legen wir ohne PDF-Dokument an. Das Feld **Dateiinhalt** bleibt dabei leer:

```
INSERT INTO dbo.tblPDFs (Dateiname)
VALUES (N'ohne_Inhalt.pdf');
```

Ein PDF-Dokument auf die Festplatte schreiben

Die Funktion **PDFAusSQLSpeichern** schreibt den Inhalt zurück in eine Datei (siehe Listing 4). Sie liest das Feld **Dateiinhalt** des Datensatzes mit der übergebenen

```
Public Function PDFInSQLSpeichern(ByVal strDateipfad As String, Optional ByVal strConnection As String) As Long
    Dim cnn As ADODB.Connection
    Dim cmd As ADODB.Command
    Dim rst As ADODB.Recordset
    Dim stm As ADODB.Stream
    Dim fso As Object
    Dim varData As Variant
    Dim strDateiname As String

    Set fso = CreateObject("Scripting.FileSystemObject")
    If Not fso.FileExists(strDateipfad) Then
        Err.Raise vbObjectError + 1000, "PDFInSQLSpeichern", "Die Datei " & strDateipfad & " existiert nicht."
    End If
    If Len(strConnection) = 0 Then strConnection = Verbindung()

    Set stm = New ADODB.Stream
    With stm
        .Type = adTypeBinary
        .Open
        .LoadFromFile strDateipfad
        varData = .Read
        .Close
    End With
    If IsNull(varData) Then
        Err.Raise vbObjectError + 1001, "PDFInSQLSpeichern", "Die Datei " & strDateipfad & " ist leer."
    End If

    strDateiname = fso.GetFileName(strDateipfad)

    Set cnn = New ADODB.Connection
    cnn.Open strConnection
    Set cmd = New ADODB.Command
    With cmd
        .ActiveConnection = cnn
        .CommandType = adCmdText
        .CommandText = "SET NOCOUNT ON; " _
            & "INSERT INTO dbo.tblPDFs (Dateiname, Dateiinhalt) VALUES (?, ?); " _
            & "SELECT CAST(SCOPE_IDENTITY() AS int);"
        .Parameters.Append .CreateParameter("pDateiname", adVarWChar, adParamInput, 255, strDateiname)
        .Parameters.Append .CreateParameter("pDateiinhalt", adLongVarBinary, adParamInput, UBound(varData) + 1, varData)
        Set rst = .Execute
    End With
    PDFInSQLSpeichern = rst.Fields(0).Value
    rst.Close
    cnn.Close
End Function
```

Listing 3: PDFInSQLSpeichern schreibt eine Datei in die Tabelle