Performance-Tuning in VBA und Formularen

Josef Pötzl
www.joposol.com

AEK 7
25. – 26. Sept. 2004

Inhalt

Einleitung

Wie kann man eine Access-Anwendung schneller machen?
Diese Frage, die sich jeder Access-Entwickler einmal stellt, wird anhand von ausgewählten Beispielen behandelt. Ein wichtiger Punkt bei jeder Optimierung ist der Optimierungsgrad. Performance-Tests sollen diesen erkennbar machen.

Testablauf

Viele Messungen bewegen sich im Mikrosekundenbereich für eine einzelne Anweisung. Aus diesem Grund wurden die Anweisungen in eine Testschleife eingebettet.

Beispiel-Code

```
Stoppuhr.StartT
For j = 1 To conLoopX
For i = 1 To conLoopA
    S = Left(Wert(i), 3)
Next i
Next j
Stoppuhr.StopT
Laufzeit = Stoppuhr.Time / (conLoopX * conLoopA)
```

`Stoppuhr` ist ein Objekt, das von einer VBA-Klasse abgeleitet wurde, welche die Funktion `timeGetTime` aus der `winmm.dll` verwendet.

`conLoopA` dient als innere Schleife für die Variation der Testparameter, welche im obigen Beispiel im `Wert()` gespeichert sind.

Um eine höhere Durchlaufzeit zu erhalten, wird bei Bedarf die innere Schleife mit `conLoopX` mehrfach wiederholt.

Testumgebung

Netzwerk:100MBit

File-Server:
- Celeron 466 MHz, 512 MB RAM, kein Raid-System
- SuSE Linux 8.0 mit Kernel 2.4,
- Samba 2.28,

Client:
- Notebook Acer Travelmate 630 P4 1,8 GHz, 512 MB RAM,
- WindowsXP SP1,
- OfficeXP SP3

Grundkonfiguration der Access-Anwendung:
- Frontend-/Backend-Trennung
- Testtabelle für Datenzugriff mit ≈ 130.000 Datensätzen
- Anm.: Es wurden großteils indizierte Felder für die Datenfilterung verwendet.

VBA

String-Funktionen

$-Funktionen

Verwende die „$“-Variante von Textfunktionen, wenn Du als Rückgabe einen String erwartest.

Diesen Hinweis liest man immer wieder auf vielen Webseiten und in Büchern, die sich mit Access befassen. Doch welchen Performance-Gewinn bringt die Verwendung dieser $-Funktionen wirklich?

Test-Varianten

```
a) S = Left(Wert(i), 3)
b) S = Left(Wert(i), 3)
```

Messergebnis

(Laufzeit je Einzelaufruf)

	Ta Funktion()	Tb Funktion$()	Differenz Tb - Ta	Tb – Ta Ta
Left	0,525 µs	0,260 µs	- 0,000265 ms	- 50 %
Right	0,507 µs	0,254 µs	- 0,000253 ms	- 50 %
Mid	0,616 µs	0,326 µs	- 0,000290 ms	- 47 %
Trim	0,543 µs	0,273 µs	- 0,000270 ms	- 50 %
Format	2,569 µs	2,484 µs	- 0,000085 ms	- 3 %

Betrachtet man nur die Prozentangaben, dann laufen die $-Funktionen in etwa doppelt so schnell wie die Standardfunktionen ab. In absoluten Zahlen betrachtet, hält sich der Performance-Zuwachs in Grenzen.

Leere Zeichenfolge

Verwende zum Zuweisen eines Leerstrings vbNullString statt "".

Wenn eine Stringvariable den Wert eines Leerstrings erhalten soll, dann gibt es diese zwei Möglichkeiten.

Test-Varianten

```
a) S = ""
b) S = vbNullString
```

Der Testablauf basiert auf demselben Schema wie im vorigen Beispiel.

Messergebnis

Ta S = ″″	Tb S = vbNullString	Differenz Ta - Tb	Tb – Ta Ta
0,194 µs	0,047 µs	- 0,000146 ms	- 76 %

Dieser Performance-Gewinn bewegt sich in ähnlichen Grenzen wie bei den $-String-Funktionen. Der relative Vergleich fällt bei diesem Tipp etwas höher aus: Variante b ist 4x so schnell wie Variante a.

Textlänge

Überprüfe auf Leerstring mit Len(...) statt dem Vergleich mit vbNullString.

Die Länge eines Strings kann mit VBA schneller ermittelt werden als bei einem Vergleich mit einem Leerstring. Dies hängt von der VBA-internen Behandlung der Stringvariablen ab.

Test-Varianten

a) S > vbNullString
b) Len(S) > 0

Messergebnis

Ta S > vbNullString	Tb Len(S) > 0	Differenz Tb - Ta	Tb – Ta Ta
0,149 µs	0,071 µs	- 0,000078 ms	- 52 %

Textvergleich

Führe Textvergleiche mit StrComp(...) durch.

Für Textvergleiche muss man darauf achten, ob Groß- und Kleinschreibung berücksichtigt werden sollen. Will man diese nicht beachten, gibt es folgende Möglichkeiten:

Test-Varianten

```
a) UCase(S) = ″ABC″
b) S = ″abc″ 'mit Option Compare Text im Modulkopf
c) StrComp(S, "abc", vbTextCompare)
```

Messergebnis

Ta UCase(S) = ″ABC″	Tb S = ″abc″	Tc StrComp(…)	Differenz Tc - Ta	Tc – Ta Ta
1,415 µs	0,420 µs	0,417 µs	~0,001 ms	- 70 %

Die Variante a ist die langsamste Methode. b und c sind in etwa gleich schnell. UCase ist dabei um mehr als das 3-fache langsamer.

Objekte

Early Binding

Vermeide den Datentyp Object und verwende stattdessen den spezifischen Typ des jeweiligen Objektes.

Test-Varianten

a) Late Binding: Objekttypen wie *Object* oder *Control*

b) Early Binding: Objekttyp des jeweiligen Objektes (z.B. für ein Textfeld: *TextBox* statt allg. *Control*)

Messergebnis

Ta Late Binding	Tb Early Binding	Differenz Tb - Ta	Tb – Ta Ta
DAO.Recordset			

`Dim rst As Object`
`Set rst = <rst>`
`x = rst.Fields.Count`
	`Dim rst As DAO.Recordset`
	`Set rst = <rst>`
	`x = rst.Fields.Count`
		- 0,0005 ms	- 87 %

Ta = 0,562 µs	Tb = 0,074 µs		
Formularsteuerelement (Textfeld)			

`Dim txt As Control`
`Set txt = Me!Textfeld`
`x = txt.Name`
	`Dim txt As TextBox`
	`Set txt = Me!Textfeld`
	`x = txt.Name`
		- 0,0090 ms	- 40 %

Ta = 22,3 µs	Tb = 13,3 µs		

Beachte!

Early Binding kann aber auch Probleme verursachen. Man denke nur an die Verweisproblematik. ➔ Nur anwenden, wenn die spezifischen Objekttypen in den Standardverweisen enthalten sind.

Objekt-Zugriff

Speichere Objektreferenzen für mehrfache Verwendung ab.

Besonders bei der Verwendung von Objekten lassen sich Performance-Gewinne erzielen, wenn man gezielt mit Objektreferenzen arbeitet.
Am Beispiel der Recordset-Auswertung lässt sich dies gut darstellen.

Test-Varianten

a)

```
With rst
    While Not .EOF
        X = !X
        Y = !Y
        Z = !Z
        .MoveNext
    Wend
End With
```

b)

```
With rst
    While Not .EOF
        X = .Fields("X")
        Y = .Fields("Y")
        Z = .Fields("Z")
        .MoveNext
    Wend
End With
```

c)

```
With rst
    While Not .EOF
        X = .Fields(0)
        Y = .Fields(1)
        Z = .Fields(2)
        .MoveNext
    Wend
End With
```

d)

```
Dim fldX As DAO.Field, _
    fldY As DAO.Field, _
    fldZ As DAO.Field
With rst
    Set fldX = .Fields(0)
    Set fldY = .Fields(1)
    Set fldZ = .Fields(2)
    While Not .EOF
        X = fldX.Value
        Y = fldY.Value
        Z = fldZ.Value
        .MoveNext
    Wend
End With
```

Messergebnis

Variante	Laufzeit je Datensatz	Differenz Tx - Ta	Tx – Ta / Ta
a	23,0 ms	-	-
b	22,7 ms	- 0,3 ms	- 1 %
c	13,2 ms	- 9,8 ms	- 43 %
d	7,5 ms	- 15,5 ms	- 67 %

Bei der Verwendung der Feldnamen macht es keinen Unterschied ob `rst!Feldname` bzw. `rst.Fields(Feldname)` verwendet wird. Den ersten Performance-Gewinn erzielt man mit `rst.Fields(Feldindex)`. Diese Optimierung bringt bereits einen Gewinn wenn nur ein Datensatz auswertet wird.

Sobald mehrer Datensätze ausgewertet werden sollen, ist es von Vorteil, die einzelnen Feldreferenzen abzuspeichern, wie in Variante d gezeigt.

Me-Schlüsselwort

Verwende Me statt Forms!FormularName für Formular-Referenzen innerhalb desselben Formulars.

In einem Formular- oder Berichtsmodul kann man für den Zugriff auf das Objekt, in dem sich der Code befindet, das Me-Schlüsselwort verwenden.

Beispiel: Zugriff auf ein Steuerelement

Test-Varianten

a) Forms!FormularName!SteuerelementName
b) Me!Steuerelementname

Messergebnis

Ta Forms!FormularName	Tb Me	Differenz Tb -Ta	Tb – Ta Ta
0,0215 ms	0,0134 ms	- 0,008 ms	- 38 %

Auch hier ist der Zeitgewinn nicht besonders groß. Dieser Tipp hat einen weiteren Vorteil: Der Code wird unabhängig vom Formularnamen!

Analog zu Formularen kann man dies natürlich auch bei Berichten anwenden.

Datenzugriff

Es gibt in Access zwei Möglichkeiten auf die Daten per VBA zuzugreifen:

- DAO
- ADO

Welche Methode die schnellere ist, hängt sehr stark vom jeweiligen Optimierungsgrad ab. Verwendet man die „Standard-Methoden“ wie sie in der Access-Hilfe dargestellt werden, so ist der Zugriff auf eine *mdb-Datei* mit DAO schneller. Sobald auf einen SQL-Server zugegriffen wird, ist ADO die schnellere Variante.

Genauere Vergleiche sind im Punkt *DAO vs. ADO* ersichtlich.

Ausgangsbasis

```
Dim dbs As DAO.Database
Dim rst As DAO.Recordset
Set dbs = CurrentDb
Set rst = dbs.OpenRecordset(…)
With rst
    …
    .close
End With
Set rst = Nothing
Set dbs = Nothing
```

CurrentDb

```
Optimierungspunkt: Set dbs = CurrentDb
```

Verwende statt CurrentDb die Lösung von Michael Kaplan

Bei Verwendung von CurrentDb wird jedes Mal eine neue Instanz eines Datenbankobjektes erzeugt. Dies benötigt natürlich Zeit. Abhilfe könnte die Verwendung von DBEngine(0)(0) bringen. Diese Methode wird jedoch von Microsoft nicht empfohlen. Eine weitere Möglichkeit stammt von Michael Kaplan.

```
Private m_DaoDB As DAO.Database
Public Property Get CurrentDbC() As DAO.Database
    If (m_DaoDB Is Nothing) Then
        Set m_DaoDB = CurrentDb
    End If
    Set CurrentDbC = m_DaoDB
End Property
```

Bei dieser Lösung wir die einmalig erzeugte Instanz des Datenbankobjektes in der Objektvariablen zwischengespeichert. Durch den Zugriff über die Eigenschaft CurrentDbC ist sichergestellt, dass immer eine vorhandene Objektreferenz zurückgegeben wird.

Test-Varianten

a) CurrentDb
b) DBEngine(0)(0)
c) CurrentDbC (Ersatz für CurrentDb)

Messergebnis

Set dbs = …	Dauer Tx	Differenz Tx - Ta	Tb – Ta / Ta
a) CurrentDb	0,846 ms	-	-
b) DBEngine(0)(0)	0,005 ms	- 0,841 ms	- 99,4 %
c) CurrentDbC	0,002 ms	- 0,844 ms	- 99,8 %

DBEngine und CurrentDbC sind als gleich schnell zu betrachten. Dabei ist zu berücksichtigen, dass beim ersten Aufruf von CurrentDbC einmal CurrentDb verwendet wird. Diese Zeit für den einmaligen Aufruf von nicht einmal einer Millisekunde ist vernachlässigbar.

Persistentes Recordset

Erstelle eine offene Verbindung zu einem Backend im Netzwerk durch ein persistentes Recordset.

Dieser Tipp ist besonders bei Mehrbenutzerbetrieb mit einem mdb-Backend auf einem Netzlaufwerk interessant.

Testaufbau

```
Set dbs = CurrentDb
Set rst = dbs.OpenRecordset("SELECT * FROM Tab Where Feld=…")
With rst
    X = .Fields(0)
    .close
End With
Set rst = Nothing
Set dbs = Nothing
```

Test-Varianten

a) ohne persistentes Recordset
b) mit persistentem Recordset

Messergebnis

Persistentes Recordset	Ta ohne pers. R.	Tb mit pers. R.	Differenz Tb - Ta	Tb – Ta / Ta
Lokales BE	22 ms	12 ms	- 10 ms	- 45 %
BE auf Netzlaufwerk	101 ms	41 ms	- 60 ms	- 59 %

Datensätze filtern

Verwende zum Filtern von Datensätzen die Where-Anweisung in SQL.

Wenn man bestimmte Datensätze aus den Tabellen auslesen möchte, so gibt es grundsätzlich 3 Methoden dies durchzuführen.

Test-Varianten

a) Den Filter mit einer SQL-Anweisung anwenden. (Select … Where <Filter>)
b) Die Datensätze per FindFirst (DAO) bzw. Find (ADO) abfragen.
c) Eine Parameterabfrage erstellen, bei der man die gewünschten Filterwerte als Parameter übergibt.

Variante a

```
Dim strSQL As String, _
    rst As DAO.Recordset
strSQL = "SELECT * FROM <Tabelle>" & _
         " WHERE Zahlenfeld = " & <Zahlenwert>
set rst = CurrentDbC.OpenRecordset(strSQL)
With rst
    X = .Fields(0)
    …
    .close
End With
set rst = Nothing
```

Variante b (DAO)

```
Dim strFilter As String, _
    rst As DAO.Recordset
strFilter = "Zahlenfeld = " & <Zahlenwert>
set rst = CurrentDbC.OpenRecordset(<Tabelle oder Abfrage>, _
                                   dbOpenDynaset)
With rst
    .FindFirst strFilter
    X = .Fields(0)
    …
    .close
End With
set rst = Nothing
```

Variante c

```
Dim qdf as DAO.QueryDef, _
    rst as DAO.Recordset
Set qdf = CurrentDbc.QueryDefs(<Parametrabfrage>)
qdf.Parameters(0) = <1. Parameterwert>
set rst = qdf.OpenRecordset
With rst
    X = .Fields(0)
    …
    .close
End With
set rst = Nothing
qdf.close
set qdf = Nothing
```

Messergebnis

Anm.: Die Absolutwerte beziehen sich auf das Auswerten eines einzigen Datensatzes.

Test I: Nur den ersten Datensatz auswerten, der einem bestimmten Kriterium entspricht.

Lokales Backend	Laufzeit Tx	Differenz Tx - Ta	$\frac{Tx - Ta}{Ta}$
a) SQL	18,6 ms	-	-
b) Find	18,5 ms	- 0,1 ms	- 0,5 %
c) Parameter	31,5 ms	+ 12,9 ms	+ 69 %

Backend im Netz	Laufzeit Tx	Differenz Tx - Ta	$\frac{Tx - Ta}{Ta}$
a) SQL	81,7 ms	-	-
b) Find	82,0 ms	+ 0,3 ms	- 0,4 %
c) Parameter	131,1 ms	+ 49,4 ms	+ 60 %

Vergleicht man Variante a (Vorfilterung per SQL) mit der Anwendung der Find-Methode auf eine per Recordset geöffnete Tabelle, so stellt man fest, dass beide Varianten ungefähr gleich schnell ablaufen. Die Parameterabfrage ist hier eindeutig unterlegen.

Test II: Alle Datensatz auswerten, die einem bestimmten Kriterium entsprechen.

Lokales Backend	Laufzeit Tx	Differenz Tx - Ta	$\frac{Tx - Ta}{Ta}$
a) SQL	96,8 ms	-	-
b) Find	232,0 ms	+ 135,2 ms	+ 140 %
c) Parameter	114,3 ms	+ 17,4 ms	+ 18 %

Backend im Netz	Laufzeit Tx	Differenz Tx - Ta	$\frac{Tx - Ta}{Ta}$
a) SQL	616,0 ms	-	-
b) Find	898,8 ms	+ 282,8 ms	+ 46 %
c) Parameter	686,0 ms	+ 111,4 ms	+ 18 %

Anm.: Es wurden im Schnitt 2000 Datensätze je Abfrage ausgelesen.

Die Find-Methode ist hier eindeutig unterlegen.

Bei diesem Test wurde nur auf eine Tabelle zugegriffen. Sobald mehrere Tabellen miteinander verknüpft sind, entstehen andere Vergleichswerte.

Test III: Den ersten Datensatz auswerten, der einem bestimmten Kriterium entspricht. Wobei sich der Datensatz aus 2 verknüpften Tabellen ergibt.

Lokales Backend	Laufzeit Tx	Differenz Tx - Ta	$\frac{Tx - Ta}{Ta}$
a) SQL	18,5 ms	-	-
b) Find	24,1 ms	+ 5,6 ms	+ 30 %
c) Parameter	17,1 ms	- 1,4 ms	- 8 %

Backend im Netz	Laufzeit Tx	Differenz Tx - Ta	$\frac{Tx - Ta}{Ta}$
a) SQL	109,8 ms	-	-
b) Find	117,3 ms	+ 7,5 ms	+ 7 %
c) Parameter	109,4 ms	- 0,4 ms	- 0,4 %

Sobald die Abfrage-Syntax komplexer wird, werden Parameterabfragen zu einer interessanten Alternative zur reinen SQL-Text-Variante.

Snapshot

Öffne Abfragen mit wenigen Datensätzen als Snapshot.

Test-Varianten

a) OpenRecordset(strSQL, dbOpenDynaset)

b) OpenRecordset(strSQL, dbOpenSnapshot)

Der Parameter dbOpenSnapshot kann die Auswertung beschleunigen. Man muss aber auf die Anzahl der Datensätze achten, da bei einer großen Anzahl das Öffnen lange dauern

kann. Wenn nur ein Feld von einem Datensatz gelesen werden soll, dann bringt dbOpenSnapshot auf jeden Fall einen Performance-Gewinn.

Messergebnis

Open-Parameter	Ta Dynaset	Tb Snapshot	Differenz Tb - Ta	Tb – Ta Ta
Lokales BE	12,9 ms	9,8 ms	- 3,1 ms	- 24 %
BE auf Netzlaufwerk	56,3 ms	49,4 ms	- 6,9 ms	- 12 %

Anm.: Bei diesem Test wurde ein einzelnes Feld eines vorgefilterten Datensatzes (Filter auf Primärschlüssel) ausgelesen.

Domain-Funktionen

Verwende statt den in Access vorhandenen Domain-Funktionen (DLookup, DCount,...) die Recordset-Ersatzfunktionen.

Ersatzfunktion für DLookup

```
Private Function RstLookup( _
            ByVal sFieldName As String, _
            ByVal sSource As String, _
            Optional ByVal sCriteria As String=vbNullString _
        ) As Variant

    Dim rst As DAO.Recordset
        strSQL As String

    strSQL = "SELECT " & sFieldName & _
               " FROM " & sSource

    If sCriteria > vbNullString Then
        strSQL = strSQL & " WHERE " & sCriteria
    End If

    Set rst = CurrentDbC.OpenRecordset(strSQL, dbOpenSnapshot)
    With rst
        If .EOF Then
            RstLookup = Null
    Else
        RstLookup = .Fields(0)
    End If
    .Close
    End With
    Set rst = Nothing
End Function
```

Test-Varianten

a) DLookup

b) Ersatzfunktion

Messergebnis

DLookup	Ta Original	Tb Ersatzfunktion	Differenz Tb - Ta	Tb – Ta Ta
Lokales BE	44 ms	29 ms	- 15 ms	- 34 %
BE auf Netzlaufwerk	318 ms	171 ms	- 147 ms	- 46 %

DAO vs. ADO

Die oben angeführten Optimierungsvorschläge kann man in ähnlicher Form bei einem ADO-Datenzugriff anwenden.

Test-Varianten

a) DAO
b) ADO

Standard-Methode

DAO

```
Set dbs = CurrentDb

Set rst =
        dbs.OpenRecordset(…)
With rst
    ...
End With
...
```

ADO

```
Set conn =
CurrentProject.Connection
Set rst = New ADODB.Recordset
rst.Open <SQL>, conn, …
With rst
    ...
End With
...
```

Vergleicht man diese beiden Varianten, so ist bei einem Zugriff auf ein mdb-Backend die DAO-Methode schneller.

Messergebnis Standardmethoden

Standard	Ta DAO	Tb ADO	Differenz Tb - Ta	Tb – Ta Ta
Lokales BE	14,4 ms	20,0 ms	+ 5,6 ms	+ 39 %
Netzwerk	86,1 ms	126,3 ms	+ 40,2 ms	+ 47 %

Man kann bei ADO statt *CurrentProject.Connection* den Connection-String manuell setzen. Auf diese Art läuft der Datenzugriff nicht über das Frontend, sondern es wird direkt auf das Backend zugegriffen.

```
Set conn = New ADODB.Connection
conn.open "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=<BE>"
```

Anm.:
<BE> = Backenddatei – Diese kann man z.B. aus einer verknüpften Tabelle ermitteln.
Sobald die Verbindung auf die BE-Datei eingestellt ist, müssen die Abfragen, die in Verbindung mit dieser Variante verwendet werden, im Backend liegen!

Messergebnis bei direktem Zugriff auf die BE-Datei per ADO

ADO: direkter Backendzugriff	Ta DAO	Tb ADO BE	Differenz Tb - Ta	Tb – Ta / Ta
Lokales BE	14,4 ms	15,5 ms	+ 1,1 ms	+ 7,6 %
Netzwerk	86,1 ms	74,4 ms	- 11,7 ms	- 13,6 %

Wenn man dies nun analog zu CurrentDbC in einer Property-Prozedur ausprogrammiert, so erhält man unter ADO eine ähnliche Zugriffsmethode wie bei DAO.

```
Private m_AdoConn As ADODB.Connection
Private CurrentConnectionString As String
Public Property Get CurrentConnection() As ADODB.Connection
    If (m_AdoConn Is Nothing) Then
        If Len(CurrentConnectionString) = 0 Then
            'keine Verküpfung
            Set p_AdoConn = CurrentProject.Connection
        Else
            Set p_AdoConn = New ADODB.Connection
            p_AdoConn.Open CurrentConnectionString
        End If
    End If
    Set CurrentConnection = m_AdoConn
End Property
```

Anm.: *CurrentConnectionString* dient zum Einstellen des Connection-Strings auf das BE und kann entweder als Konstante verwendet werden, oder man schreibt sich eine Prozedur die diese Variable entsprechend befüllt.

Die folgenden Testwerte beruhen auf einer Messung mit persistetem Recordset unter Verwendung von CurrentDbC bei DAO und CurrentConnection bei ADO.

Messergebnis mit CurrentDbC- und CurrentConnection-Methode

CurrentDbC / CurrentConnection	Ta DAO	Tb ADO BE	Differenz Tb - Ta	Tb – Ta / Ta
Lokales BE	10,1 ms	6,5 ms	- 3,6 ms	- 36 %
Netzwerk	49,4 ms	23,4 ms	- 26,0ms	- 53 %

Nun stellt sich die Frage: Warum soll man nicht auch per DAO direkt auf das Backend zugreifen?

```
Private m_DaoDbBE As DAO.Database
Public Property Get CurrentDbBE() As DAO.Database
    If (m_DaoDbBE Is Nothing) Then
        If Len(CurrentBackendFile) = 0 Then
            m_DaoDbBE = CurrentDb
        else
            Set m_DaoDbBE = _
               DBEngine(0).OpenDatabase(CurrentBackendFile)
        End If
    End If
    Set CurrentDbBE = m_DaoDbBE
End Property
```

Anm.: *CurrentBackendFile* beinhaltet den Pfad zum Backend.

Vergleicht man nun diese Methode mit der obigen ADO-Variante so ist die DAO-Variante wieder schneller.

Messergebnis bei direktem Zugriff auf die BE-Datei per ADO und DAO

DAO + ADO mit Backendzugriff	Ta DAO BE	Tb ADO BE	Differenz Tb - Ta	Tb – Ta / Ta
Lokales BE	4,5 ms	6,5 ms	+ 2,0 ms	+ 44 %
Netzwerk	5,3 ms	23,4 ms	+ 18,1 ms	+ 341 %

Eines darf bei der CurrentDbBE-Variante und bei der CurrentConnection-Variante nicht vergessen werden: die Verbindungen müssen wieder geschlossen werden!

Seek-Methode

Verwende *Seek* um einzelne Datensätze einer Tabelle auszuwerten.

Wenn man den direkten Datenzugriff auf die Backend-Datei wählt, so besteht die Möglichkeit, die Seek-Methode zum Auswerten einzelner Datensätze einer Tabelle anzuwenden. Diese Methode kann z.B. als eine „DLookup-Ersatzfunktion“ nur für Tabellen verwendet werden. Dabei ist zu beachten, dass *Seek* nur in Verbindung mit indizierten Feldern möglich ist.

Test-Varianten

a) Filterung direkt mit SQL-Anweisung

```
strSQL = "SELECT <FeldName> FROM <Tabelle>" & _
         " Where id=" & <Wert>
Set rst = CurrentDbBE.OpenRecordset(strSQL, _
                                    dbOpenSnapshot)
vTestValue = rst.Fields(0)
rst.Close
Set rst = Nothing
```

b) Seek-Methode

```
strTabName = <Tabelle>
Set rst = CurrentDbBE.OpenRecordset(strTabName, _
                                    dbOpenTable)
rst.Index = "PrimaryKey" 'Anm.: id = PK
rst.Seek "=", <Wert>
vTestValue = rst.Fields(<FeldName>)
rst.Close
Set rst = Nothing
```

Messergebnis

Ta SQL	Tb Seek	Differenz Tb - Ta	Tb – Ta / Ta
3,8 ms	0,4 ms	- 3,4 ms	- 90 %

Formulare

Der vorige Teil behandelte grundsätzliche Optimierungsmöglichkeiten in VBA. Diese sollte man durchaus berücksichtigen. Der eigentliche Performance-Gewinn für den Anwender entsteht in Verbindung mit der Benutzeroberfläche.

Dabei ist eines zu beachten: Es muss für den Anwender nicht immer die schnellste Variante den Eindruck der schnellsten Laufzeit machen. Dem Anwender kann durch kleine Tricks ein „schneller“ Ablauf vorgetäuscht werden.

So kann das Anzeigen eines PopUp-Formulars mit einem Fortschrittsbalken durchaus einen Laufzeitvorteil im Vergleich zur alleinigen Anzeige einer Sanduhr vortäuschen, obwohl der Code mit diesem Formular sicher eine längere Laufzeit benötigt. Der Trick besteht hier einfach darin, den Anwender mit dem Betrachten des Fortschritts zu beschäftigen, statt eine wenig aussagekräftige Sanduhr anzuzeigen.

Für die nun folgenden Tipps werden keine Messwerte mehr angeführt, da diese nur für einen bestimmten Formularaufbau gelten. Da Formulare immer unterschiedlich gestaltet sind, kann man diese Vorschläge nur am konkreten Formular testen, und auf diese Art herausfinden, welche der vorgeschlagenen Varianten die jeweils geeignete Methode ist.

Formulare öffnen

Der erste wichtige Ansatzpunkt für die Beschleunigung der Benutzerschnittstelle ist das Öffnen von Formularen.

Die Zeit bis zum Anzeigen hängt vor allem von folgenden Punkten ab:

- Datenherkunft
- Datenmenge
- Anzahl und Art der Steuerelemente

Besonders bei oft verwendeten Formularen kann man mit einem kleinen Trick die „Öffnungszeit“ beschleunigen: Die Formulare nicht schließen, sondern nur ausblenden und bei Bedarf wieder einblenden.

Datenherkunft

Grundsätzlich unterscheidet man zwischen

- ungebundenen Formularen und
- gebundenen Formularen.

Gebundene Formulare benötigen normalerweise längere Zeit zum Öffnen, da die Daten abgefragt werden müssen. Man kann das Formular z.B. zum Öffnen ungebunden halten und die Datenbindung erst nach dem Öffnen herstellen. So erreicht man ein schnelles Öffnen. Die Datenbindung benötigt zwar auch noch eine gewisse Zeit, aber der Anwender sieht zumindest während dieser Zeit schon das Formular vor sich.

Übliche Konfiguration

Unter der Formulareigenschaft Datenherkunft wird die benötigte Tabelle, Abfrage oder SQL-Anweisung eingetragen.

Datenherkunft später einstellen

Die Eigenschaft Datenherkunft wird vorerst einmal leer gelassen. Erst nach dem Öffnen wird die Datenherkunft eingestellt. Einen guten Anzeige-Effekt bei sehr aufwändigen Formularen erzielt man durch das Verwenden einer Timer-Anweisung.

Nach einem Timerintervall von 1 ms wird folgenden Code ausgeführt:

```
Private Sub Form_Timer()
    Me.TimerInterval = 0
    Me.RecordSource = <Datenherkunft>
End Sub
```

Alternativ zu *Me.RecordSource* kann *Me.Recordset* verwendet werden. Diese Methode kann mit der im vorigen Abschnitt beschriebenen VBA-Methoden zur Beschleunigung des Datenzugriffes kombiniert werden.

Bei „normalen“ Formularen werden diese Methoden keinen besonderen Effekt erzielen, da das Öffnen der gebundenen Formulare so schnell geht, dass man den Zeitunterschied für die Anzeige nicht merkt. Die angeführte Methode zeigt in etwas abgewandelter Form und bei Verwendung von Unterformularen einen klar erkennbaren Effekt.

Datensätze filtern

Oft werden nur bestimmte Datensätze benötigt. In so einem Fall sollte man nicht alle Datensätze laden und dann per Find oder ähnlichen Anweisungen die jeweiligen Datensätze auffinden, sondern die benötigten Daten schon ab Start herausfiltern.

Dies kann durch folgende Methoden erfolgen:

a) Filter an den Where-Parameter der OpenForm-Methode übergeben

b) Filter nach dem Öffnen eines ungebundenen Formulars mit in die Datenherkunft einbinden

Filter an den Where-Parameter der OpenForm-Methode übergeben

Diese von Access standardmäßig eingebaute Methode ist für die meisten Formulare völlig ausreichend. Sie kann aber nur in Verbindung mit den im Frontend verknüpften Tabellen verwendet werden.
Beispiel:

```
DoCmd.OpenForm <FormName>, , , <Filter>
```

Filter bei anfangs ungebundenem Formular anwenden

Für diese Methode kann man den Filter z.B. als OpenArg-Parameter an ein ungebundenes Formular übergeben.

```
DoCmd.OpenForm <FormName>, , , , , , <Filter>
```

Anschließend wird per VBA die Datenherkunft über eine Hilfsfunktion im Formularmodul gesetzt.

```
Function setFilter(ByVal sFilter As String)

    Dim strSQL As String

    strSQL = [SQL-Anweisung noch ohne Where]
    If len(sFilter) > 0 Then
        strSQL = strSQL & " WHERE " & sFilter
    End If
    'Falls Sortiert werden soll:
    strSQL = strSQL & " Order By ..."
    Me.RecordSource = strSQL

End Function
```

Oder:

```
Function setFilter(ByVal sFilter As String)
    Dim strSQL As String
    strSQL = [SQL-Anweisung noch ohne Where]
    If len(sFilter) > 0 Then
        strSQL = strSQL & " WHERE " & sFilter
    End If
    'Falls Sortiert werden soll:
    strSQL = strSQL & " Order By ..."
    Set Me.Recordset = CurrentDbc.OpenRecordset(strSQL)
End Function
```

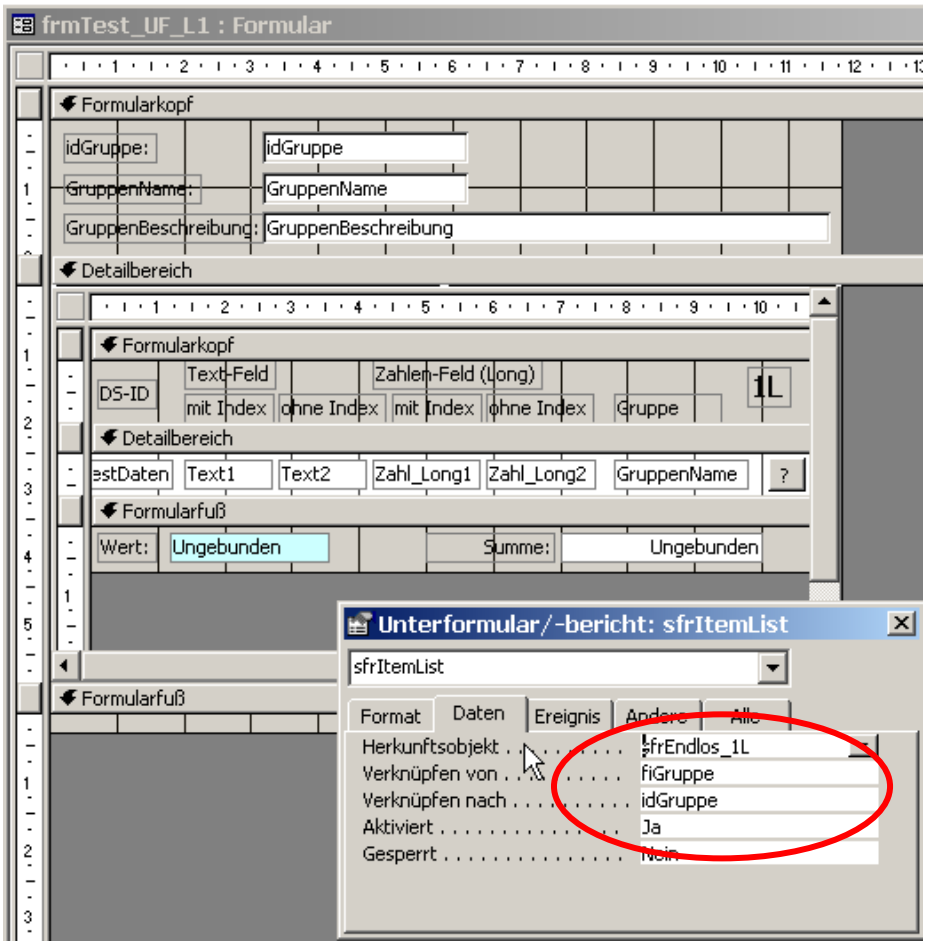
Wenn man diese Funktion dann als *Public* deklariert, kann darauf von außen zugegriffen werden.

Steuerelemente

Unterformulare

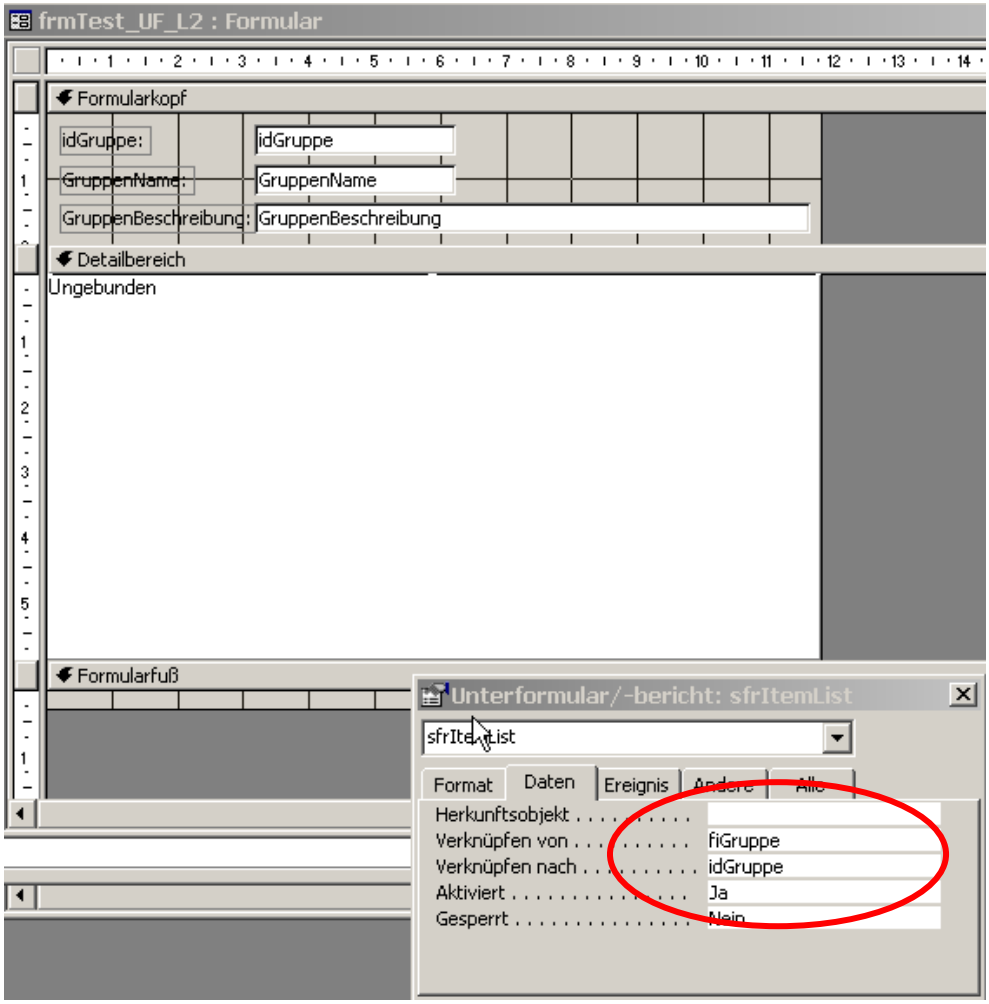
Die Verwendung von Unterformularen kann eine erhebliche Verzögerung des Anzeigevorgangs von Formularen bewirken, da zuerst die Unterformulare geladen werden und erst dann das Hauptformular angezeigt wird.

Beispiel eines verknüpften Unterformulars (Endlosformular)



In der Unterformular-Steuerelementeigenschaft Herkunftsobjekt ist das Unterformular *sfrEndlos_1L* eingetragen. Dieses wird über das Feld *idGruppe* mit dem Hauptformularfeld *fiGruppe* verknüpft. Wenn jetzt im Unterformular viele Datensätze geladen werden müssen, dauert das Anzeigen des Hauptformulars so lange, bis die erforderlichen Datensätze fertig geladen sind.

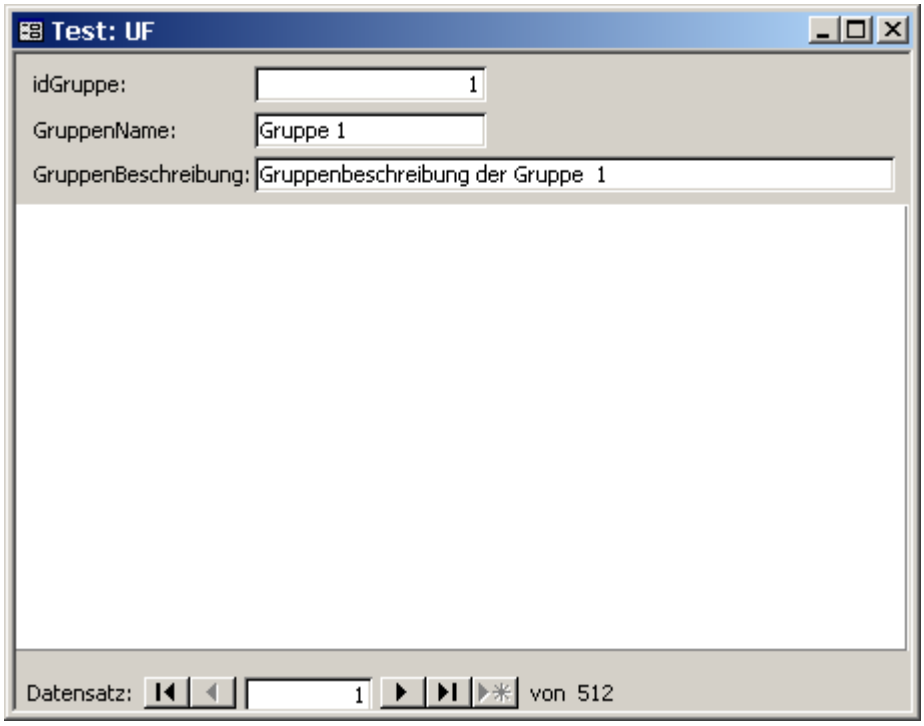
Damit der Ladevorgang des Hauptformulars schneller abläuft, kann man zuerst die Eigenschaft Herkunftsobjekt leeren und diese erst nach dem Laden des Hauptformulars setzen.



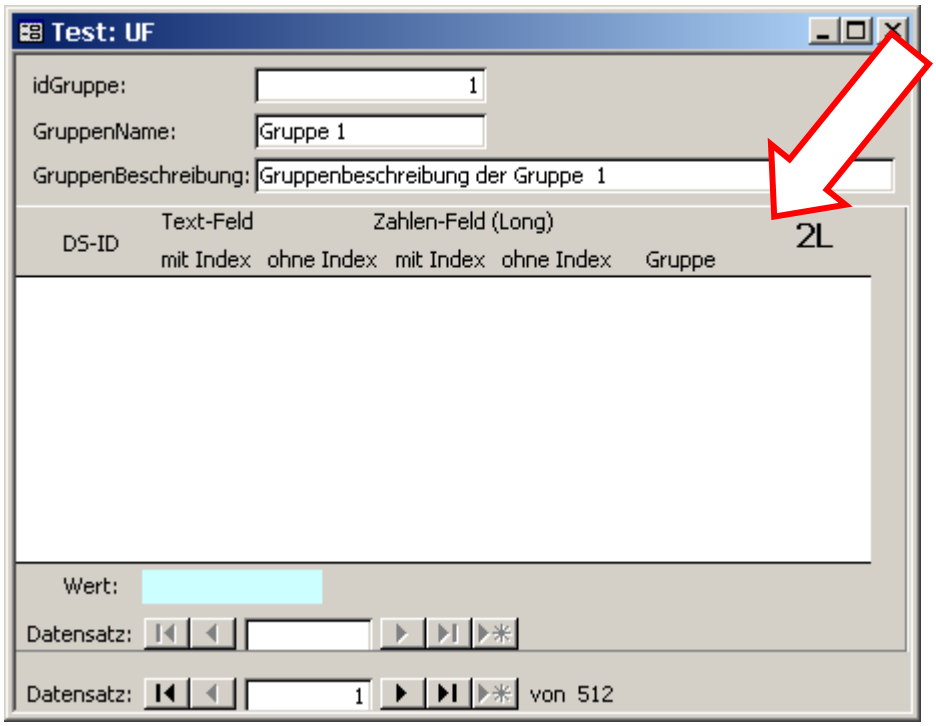
Das Herkunftsobjekt wird dann im Timer-Ereignis (Timerintervall = 1ms) gesetzt.

```
Private Sub Form_Timer()
    Me.TimerInterval = 0
    Me!sfrItemList.SourceObject = "sfrEndlos_1L"
End Sub
```

Diese Methode hat einen Schönheitsfehler: Beim Laden sieht der Anwender erstmal nur ein weißes Feld.

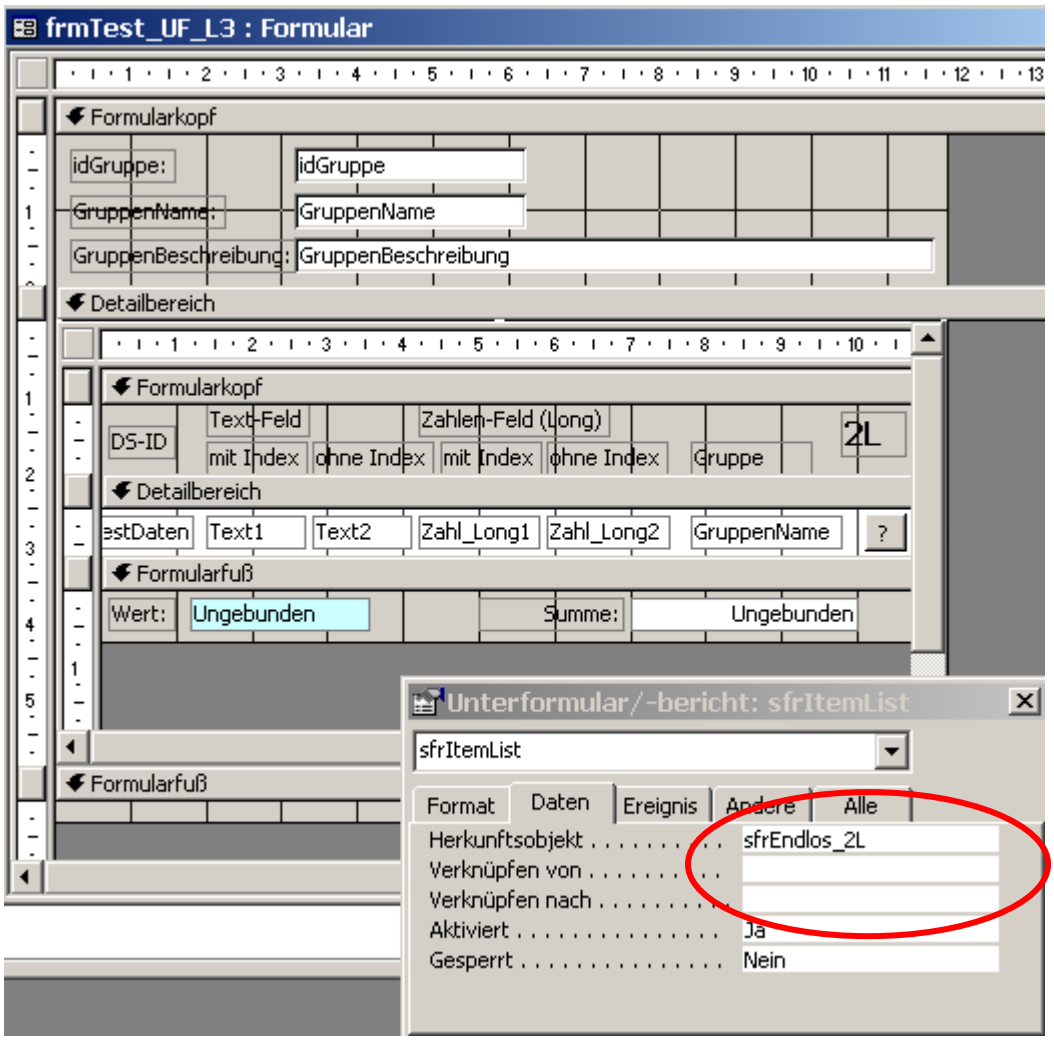


Die lässt sich vermeiden, indem man die Datenherkunft fix auf das Unterformular einstellt, das Unterformular selbst aber vorerst ungebunden lässt.



Kurz nach dem Laden sieht der Anwender nun kein weißes Feld mehr, sondern zumindest den Formularkopf und den Formularfuß des Unterformulars.

Bei dieser Variante ist es vorteilhaft, die Datenherkunft des Unterformulars mit den notwendigen Filtern zu versehen. Dafür bleiben die Eigenschaften „Verknüpfen von“ und „Verknüpfen nach“ leer.



Im Hauptformular ist zu beachten, dass die Filterung des Unterformulars mit dem aktuellen Datensatz des Hauptformulars übereinstimmt.

Code-Beispiel im Hauptformular:

```
Private m_FormIsLoaded As Boolean

Private Sub Form_Current()
    If m_FormIsLoaded Then 'bei Start umgehen
        Me!sfrItemList.Form.setFilter "fiGruppe=" & _
              Me!idGruppe
    End If
End Sub

Private Sub Form_Timer()
    Me.TimerInterval = 0
    Me!sfrItemList.Form.setFilter "fiGruppe=" & Me!idGruppe
    m_FormIsLoaded = True
End Sub
```

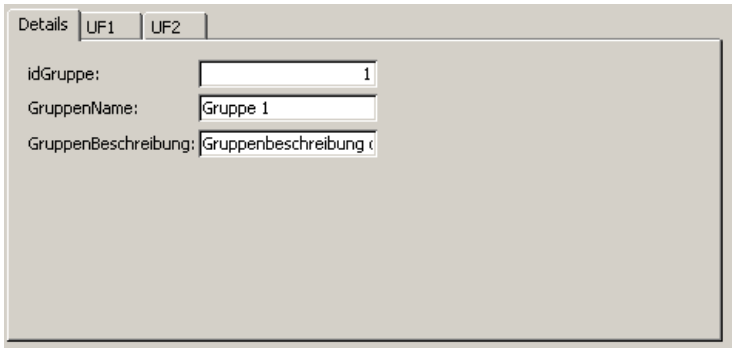
Für die Formular-Methode *setFilter* des Unterformulars kann man den Code verwenden, der zum Filtern von ungebundenen Formularen gezeigt wurde.

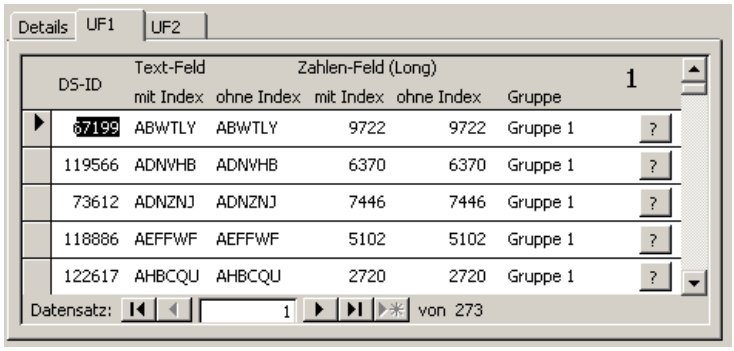
Kombinations- und Listenfelder

Ähnlich wie bei Unterformularen verhält es sich mit Kombinations- bzw. Listenfeldern. Bei diesen lassen sich im Ereignis „Beim Hingehen“ die vollständigen Daten nachladen. Vorher beschränkt man sich auf einen einzigen Datensatz, oder man verwendet - falls dies möglich ist - eine leere Datensatzherkunft.

Registersteuerelemente

Um ein Formular übersichtlich zu halten, werden gerne Registersteuerelemente verwendet. Befinden sich auf den am Anfang nicht sichtbaren Seiten Unterformulare oder Listenfelder, so sollten diese erst beim Wechsel auf die betreffende Seite geladen werden. Dies bewirkt nicht nur eine schnellere Ladezeit des Hauptformulars, sondern hat auch den Vorteil, dass das Unterformular nicht geladen wird, wenn der Anwender nicht auf die entsprechende Seite wechselt.





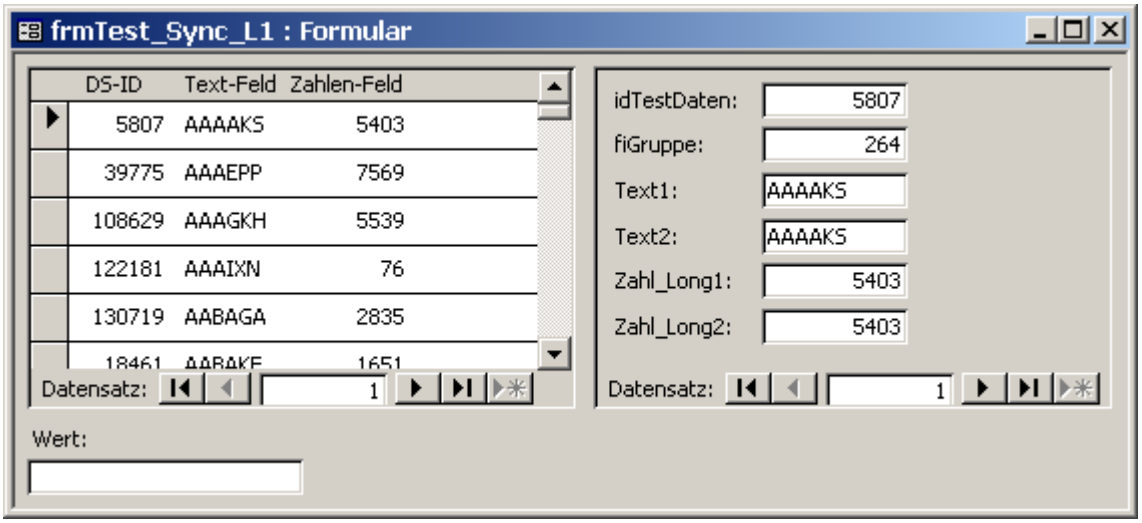
Code

Beim Ereignis „Bei Änderung“ des Registersteuerelements wird dieser Code verwendet:

```
Private Sub tabTest_Change()
'tabTest = Name des Registersteuerelements
    Select Case Me!tabTest
        Case 0 'Detail
            '…
        Case 1 'UF1 (sfr1)
            If Len(Me!sfr1.SourceObject) = 0 Then
               Me!sfr1.SourceObject = "NameUF1"
            End If
        Case 2 'UF2 (sfr2)
            If Len(Me!sfr2.SourceObject) = 0 Then
               Me!sfr2.SourceObject = " NameUF2"
            End If
    End Select
End Sub
```

Formulare synchronisieren

Um zwei Unterformulare – z.B. einmal ein Endlosformular und einmal eine Detailansicht des Datensatzes – zu synchronisieren, kann man sich der FindFirst-Methode bedienen. Damit dies von beiden Seiten funktioniert, ist entsprechender Code in beiden Formularen notwendig. Einfacher geht es, wenn man das Detail-Unterformular anfangs ungebunden lädt und beim Anzeigen des Hauptformulars das Recordset-Objekt mit dem Recordset-Objekt des Endlosformulars gleichsetzt.



Variante FindFirst

Die hier dargestellte Variante baut auf der Beschreibung in FAQ 4.3 (www.donkarl.com) auf.

Code in den Unterformularen

```
'Ereignis zum Auslösen der Prozedur im HF
Public Event SelectItem(ByVal lItemID As Long)

'Ereignis: Beim Anzeigen
Private Sub Form_Current()
    RaiseEvent SelectItem(Me!idTestDaten)
    'idTestDaten = Datensatzkennung
End Sub
```

Code im Hauptformular

```
'Formular-Objektvariablen für Ereigissteuerung
Private WithEvents frmEndlos As Form_frmTest_Sync_SF_Endlos_L1
Private WithEvents frmDetail As Form_frmTest_Sync_SF_Detail_L1

'Ereigis Beim Anzeigen
Private Sub Form_Current()
    'Objektvariable setzen
    Set frmEndlos = Me!sfrEndlos.Form
    Set frmDetail = Me!sfrDetail.Form
End Sub
```

```
'Ereignis SelectItem im Endlosformular
Private Sub frmEndlos_SelectItem(ByVal lItemID As Long)
    'Detailformular synchronisieren analog FAQ 4.3
    Dim rst As DAO.Recordset
    Set rst = frmDetail.RecordsetClone
    rst.FindFirst "idTestDaten=" & lItemID
    frmDetail.Bookmark = rst.Bookmark
    Set rst = Nothing
End Sub

'Ereignis SelectItem im Detailformular
Private Sub frmDetail_SelectItem(ByVal lItemID As Long)
    'Endlosformular synchronisieren analog FAQ 4.3
    Dim rst As DAO.Recordset
    Set rst = frmEndlos.RecordsetClone
    rst.FindFirst "idTestDaten=" & lItemID
    frmEndlos.Bookmark = rst.Bookmark
    Set rst = Nothing
End Sub
```

Variante Recordset

Code in den Unterformularen

Hier ist kein Code erforderlich.

Code im Hauptformular

```
'Ereignis: Beim Anzeigen
Private Sub Form_Current()
    'Recorset-Objekt synchronisieren
    Set Me!sfrDetail.Form.Recordset = _
                        Me!sfrEndlos.Form.Recordset
End Sub
```

Diese Variante hat einen Nachteil: Es wird auf die exakt gleiche Datenbasis zugegriffen. In der ersten Variante besteht die Möglichkeit, für die Unterformulare unterschiedliche Datenbasen zu verwenden.

Eines ist bei allen Optimierungsschritten zu beachten:

Keiner der vorgestellten Performance-Tipps kann die Optimierung der Datenstruktur ersetzen!

Quellenangaben

Paul Litwin, Ken Getz, Mike Gunderloy : Access 2002 Developer's Handbook Set
ISBN 0-7821-4011-4

Dan Haught (FMS): Making Access Faster - 168 Tips to Speed Up Your Apps! -
http://www.fmsinc.com/tpapers/faster/

Microsoft Knowledge Base: (300216) Keep a Jet 4.0 Database in Top Working Condition -
http://support.microsoft.com/support/kb/articles/Q300/2/16.asp

Granite Consulting: Microsoft Access Performance FAQ -
http://www.granite.ab.ca/access/performancefaq.htm