Datenbankentwicklung mit PureBasic

Datenbanken stellen heutzutage wichtige Informationsquellen für viele Bereiche der Wirtschaft, Verwaltung aber auch im eigenen Haushalt dar. In Datenbanken werden Daten integriert gesammelt und verarbeitet. Ein Datenbankmanagementsystem sorgt für den Zugriff auf die eigentliche Datensammlung und stellt Methoden zur Verfügung, um die Datenbank in einem konsistenten Zustand zu erhalten. Konsistenz ist die wichtigste der Anforderungen, die ein Datenbanksystem erfüllen muss und bedeutet nichts anderes als dass die Daten richtig und vollständig sind. Eine große Gefahrenquelle stellt das mehrfache Vorhandensein des gleichen Datums dar (Redundanz). Wenn nun beispielsweise eine Änderung an einer Stelle erfolgt und an einer anderen ausbleibt, dann entstehen Inkonsistenzen, da an zwei Stellen, wo gleiche Daten vermutet werden, unterschiedliche Ausprägungen stehen. Die Folge ist, das Informationen verloren gehen, denn wer mag später noch feststellen, welche Ausprägung die richtige ist. Also bevor ihr loslegt mit Datenbanken, erstellt erst ein vernünftiges Datenbankkonzept. Die Methoden der Modellierung sollen an dieser Stelle nicht Gegenstand sein und werden vielleicht später einmal ausgeformt.

Paul Leischows MDB_Lib

Die UserLibrary basiert auf der ODBC-Technik und vereinfacht das Erstellen und Verwenden von Access-Datenbanken.

Anlegen einer neuen Datenbank:

```
If InitDatabase()
  If MDB_Create("d:\test")
    MessageRequester("Fertig", "Datenbank angelegt.")
  EndIf
EndIf
```

Das Anlegen der neuen Datenbank reicht aber nicht aus. Nun muss diese auch geöffnet werden:

```
...
handle.l = MDB_Connect("d:", "test", "", "")
If handle
  ;Datenbankoperationen
  CloseDatabase(handle)
  MDB_Disconnect("test")
EndIf
...
```

Die Datenbankoperationen können ganz normal mit den Befehlen

```
Ergebnis.l = DatabaseQuery(Request.s) und
Ergebnis.l = DatabaseUpdate(Request.s)
```

ausgeführt werden.

Wenn jeweils das Ergebnis gleich 0 ist, dann ist während der Abfrage ein Fehler aufgetreten, den man per

```
Fehler.s = DatabaseError()
```

auswerten kann. Während bei Update lediglich mitgeteilt wird, ob eine Operation erfolgreich ausgeführt wurde oder nicht, liefert eine Query stets ein Set von Zeilen einer Relation, egal ob dieses leer, nur eine Zeile oder mehrere Zeilen enthält. Das abschließende Beispiel soll die einfache Handhabung mit dieser UserLibrary zeigen:

```
DefType.s sql, zwName, zwGroesse, name
DefType.l handle, id
DefType.b i
DefType.f gross



If InitDatabase()
  If MDB_Create("d:\test")
    ;erfolgreich angelegt
    ;Datentabellen anlegen
    handle = MDB_Connect("d:", "test", "", "")
    If handle
      sql = "CREATE TABLE Zwerg(ZwergID AUTOINCREMENT, Name CHAR(30), Groesse
FLOAT);"
      If DatabaseUpdate(sql) = 0
        MessageRequester("Fehler", "Bei Anlegen der Datentabelle trat ein Fehler
auf.")
        DeleteFile("d:\test.mdb")
      EndIf
      Restore Zwerge
      For i = 1 To 4
        Read zwName
        Read zwGroesse
        sql = "INSERT INTO Zwerg (Name, Groesse) VALUES ('" + zwName + "', " +
zwGroesse + ")"
        ;Der Autowert ID wird automatisch hochgezählt.
        If DatabaseUpdate(sql) = 0
          MessageRequester("Fehler", "Beim Einfügen eines Wertes trat ein Fehler
auf." + #CRLF$ + DatabaseError())
        EndIf
      Next i
      CloseDatabase(handle)
    EndIf
  EndIf
  handle = MDB_Connect("d:", "test", "", "")
  If handle
```

```
    ;
    sql = "SELECT * FROM Zwerg WHERE Groesse < 1.44 ORDER BY Groesse desc"
    If DatabaseQuery(sql)
      While NextDatabaseRow()
        id = GetDatabaseLong(0)
        name = GetDatabaseString(1)
        gross = GetDatabaseFloat(2)
        MessageRequester("Datenzeile", Str(id)+ #TAB$ + name + #TAB$ +
StrF(gross, 2))
      Wend
    EndIf
    CloseDatabase(handle)
  EndIf
  MDB_Disconnect("test")
EndIf
End



DataSection
Zwerge:
;Ein Zwerg hat je einen Namen und eine Größe.
;Die Größe ist als float-formatierter String definiert. Das ist wichtig für den
Update-Befehl.
Data.s "Heinz", 1.24", "Gerd", "1.11", "Willi", "1.18", "Kuni", "1.44"
EndDataSection
```

Wie man sieht, muss nachdem dem CloseDatabase(handle) die Access-Datenbank-Verbindung getrennt werden. Das geschieht mittels MDB_Disconnect("test"). Das Unterlassen führt zur Abspeicherung einer permanenten Datenquelle im ODBC-Dialog.

Folgende Datentypen können in Access verarbeitet werden (innerhalb der Datenbank):

BINARY, BIT, TINYINT (=.b), MONEY, DATETIME, UNIQUEIDENTIFIER, REAL (=.f), FLOAT (=.d), SMALLINT (=.w), INTEGER (=.l), DECIMAL, TEXT, IMAGE, CHAR(n) (=.s)

Die Angabe des Datentyps erfolgt immer immer hinter dem Feldnamen. Der Typ AUTOINCREMENT ist immer vom Typ INTEGER und wird von der Datenbank automatisch hochgezählt.

Was ist nun das besondere an der Access-Datenbank?

Mal angenommen, eine Abfrage wird öfter benötigt. Dann kann man diese Abfrage unter einem Namen in der Datenbank abspeichern und später aufrufen. Unsere Zwergabfrage, die uns alle Zwerge listen soll, die kleiner sind als der größte Zwerg lautet dann (mit Berücksichtigung auf noch später eingefügte Zwerge):

```
sql = "SELECT * FROM Zwerg z WHERE EXISTS (SELECT * FROM Zwerg x WHERE x.ZwergID
<> z.ZwergID and z.Groesse < x.Groesse) ORDER BY z.Groesse desc"
```

Die Abfrage wird jetzt als Sicht gespeichert:

```
sql = "CREATE VIEW Kleinste as " + sql
If DatabaseUpdate(sql)
      MessageRequester("View angelegt", "Die Sicht wurde erfolgreich angelegt.")
    Else
      MessageRequester("Fehler", DatabaseError())
EndIf
```

Die Sicht wird dann ganz bequem über

```
sql = "SELECT * FROM Kleinste"
If DatabaseQuery(sql)
  While NextDatabaseRow()
    id = GetDatabaseLong(0)
    name = GetDatabaseString(1)
    gross = GetDatabaseFloat(2)
    MessageRequester("Datenzeile", Str(id)+ #TAB$ + name + #TAB$ + StrF(gross,
2))
  Wend
Else
  MessageRequester("Fehler", DatabaseError())
EndIf
```

aufgerufen und liefert das gewohnte Ergebnis. Ein nachträglicher Eintrag des Zwergs "Günter" mit einer Größe von "1.54" wird dann automatisch beim nächsten Aufruf der Sicht berücksichtigt.

```
If DatabaseUpdate("INSERT INTO Zwerg (Name, Groesse) VALUES ('Günter', 1.54)")
    If DatabaseQuery(sql)
      While NextDatabaseRow()
        id = GetDatabaseLong(0)
        name = GetDatabaseString(1)
        gross = GetDatabaseFloat(2)
        MessageRequester("Datenzeile", Str(id)+ #TAB$ + name + #TAB$ +
StrF(gross, 2))
      Wend
    Else
      MessageRequester("Fehler", DatabaseError())
    EndIf
EndIf
```

Dieser Code muss vor dem CloseDatabase(handle) eingefügt werden.

SQLite3-Datenbanken

Die Lösung mit SQLite3 setzt zunächst das Einbinden und Ausliefern der SQLite3.dll voraus. Die jeweils aktuelle Version kann von http://www.sqlite.org beschafft werden. SQLite ist eine in C geschriebene Mini-Database-Engine und beinhaltet viele Elemente des SQL-92-Standards. Insbesondere wird das Transaktionenkonzept (ACID-Prinzip) unterstützt. Ein weiterer Vorteil ist, dass SQLite OpenSource ist und ohne Gebühren verwendet werden kann (auch z. B. auf einem Webserver, der keine MySQL-DB besitzt). Eine UserLibrary für den Einsatz der SQLite3.dll ist in den PureBasic OpenSource Libraries (PBOSL) enthalten. Also als erstes die SQLite3.dll zusammen mit den PBOSL holen und anschließend die PBOSL installieren. Die SQLite3.dll wird in das Projektverzeichnis kopiert. Die UserLibrary sorgt für den Zugriff auf die SQLite3.dll - die SQLite3.dll wiederrum organisiert die Datenbank.

Mit dem Befehl

```
Ergebnis.l = SQLite3_InitLib(Pfad.s)
```

wird die dll in die Datei eingebunden, aber nur als Verweis auf die externe dll (deshalb muss sie auch mit ausgeliefert werden). Eine andere Möglichkeit ist das vollständige Einbinden mittels

```
Ergebnis.l = SQLite3_InitLibFromMemory(?Sqlite3)
...

DataSection
Sqlite3:
IncludeBinary "sqlite3.dll"
EndDataSection
```

Der Vorteil bei der zweiten Variante ist, dass die dll bereits Bestandteil des Programms geworden ist, also nicht extra ausgeliefert werden muss. Dadurch wird das Programm natürlich größer. Das spielt aber in der Regel bei Einzelbenutzer zunächst nur eine geringe Rolle. Allerdings wird bei mehreren Datenbankanwendungen, die die sqlite3.dll benutzen, das Speicherproblem größer.

Das Anlegen einer Datenbank (eine Datendatei) erfolgt mit dem intuitiven Befehl

```
handle.l = SQLite3_CreateDatabase(Database.s, Overwrite.b)
```

Für Database ist der vollständige Pfad anzugeben. Overwrite ist eigentlich ein Boolean-Wert und lässt also Nein (0) oder Ja (1) zu. Das Handle wird später benötigt, weil auch mehrere Datenbanken zur gleichen Zeit geöffnet sein könnten. Wenn eine Datenbank lediglich geöffnet werden soll, dann wird der Befehl

```
handle.l = SQLite3_OpenDatenbase(Database.s)
```

verwendet. Datenbanken anlegen und öffnen reicht aber nicht, schließlich soll ja irgendwas gespeichert oder abgerufen werden. Da die Datenbank noch leer ist, soll jetzt die Zwergentabelle angelegt werden. Der SQL-String ist ähnlich dem der Access-Datenbank. Leider kennt SQLite nicht

den AUTOINCREMENT, aber Rettung naht. Durch die Angabe INTEGER PRIMARY KEY kann das Problem behoben werden. Der Befehl

```
Ergebnis.l = SQLite3_Execute(sql.s, handle.l)
```

führt die Abfrage aus. Das Befüllen erfolgt wieder mit dem gewöhnlichen INSERT INTO:

```
DefType.s sql, zwName, zwGroesse
DefType.l handle, i

If SQLite3_InitLibFromMemory(?Sqlite)
  MessageRequester("Meldung", "Die DLL wurde vollständig eingebunden.")
  handle = SQLite3_CreateDatabase("d:\test.db", 1)
  MessageRequester("Meldung", "Datenbank angelegt.")
  sql = "CREATE TABLE Zwerg(ZwergID INTEGER PRIMARY KEY, Name CHAR(30), Groesse FLOAT);"
  If SQLite3_Execute(sql, handle)
    MessageRequester("Meldung", "Tabelle angelegt.")
    Restore Zwerge
    For i = 1 To 4
      Read zwName
      Read zwGroesse
      sql = "INSERT INTO Zwerg (Name, Groesse) VALUES ('" + zwName + "', " + zwGroesse + ")"
      ;Der Autowert ID wird automatisch hochgezählt.
      If SQLite3_Execute(sql, handle) = #False
        MessageRequester("Fehler", "Beim Einfügen eines Wertes trat ein Fehler auf." + #CRLF$ + SQLite3_GetLastMessage())
      EndIf
    Next i
  Else
    MessageRequester("Fehler", SQLite3_GetLastMessage())
  EndIf
  SQLite3_CloseDatabase(handle)
EndIf

DataSection
Sqlite:
IncludeBinary "sqlite3.dll"
Zwerge:
;Ein Zwerg hat je einen Namen und eine Größe.
;Die Größe ist als float-formatierter String definiert. Das ist wichtig für den Update-Befehl.
Data.s "Heinz", "1.24", "Gerd", "1.11", "Willi", "1.18", "Kuni", "1.44"
EndDataSection
```

Falls ein Fehler aufgetreten ist, kann dieser mit SQLite3_GetLastMessage() ausgewertet werden.
Jetzt kommt der weitaus wichtigere Teil, die Abfrage von Daten. Zuvor muss wieder eine Abfrage in SQL-Notation geschrieben werden. Dieser SQL-String wird zusammen mit dem Handle der Datenbank an den Befehl

```
Ergebnis.l = SQLite3_GetRecordSet(sql.s, handle.l, @rs)
```

übergeben. Der dritte Parameter bezeichnet einen vorher definiertes RecordSet. Das RecordSet ist eine spezielle Struktur, die neben den eigentlichen Werten zusätzliche Informationen enthält, wie BOF (Flag = 1, wenn Zeiger auf Anfang), EOF (Flag = 1, wenn Zeiger auf Ende), Handle (Handle des RecordSets), Rows (Anzahl der Zeilen), Cols (Anzahl der Spalten), CurrentPos (aktuelle Position des RecordSet-Zeigers) und sValue (aktueller Wert an Zeigerposition). Was nicht fehlen darf ist die Definition einer Variablen vom Typ s_Recordset.

```
Structure s_Recordset
  BOF.l
  EOF.l
  handle.l
  Rows.l
  Cols.l
  CurrentPos.l
  sValue.s
EndStructure


DefType.s sql, id, name, gross
DefType.l handle, i
rs.s_Recordset


If SQLite3_InitLibFromMemory(?Sqlite)
  MessageRequester("Meldung", "Die DLL wurde vollständig eingebunden.")
  handle = SQLite3_OpenDatabase("d:\test.db")
  sql = "SELECT * FROM Zwerg WHERE Groesse < 1.44 ORDER BY Groesse desc"
  If SQLite3_GetRecordset(sql, handle, @rs)
    If rs\handle
      While rs\EOF = 0
        If SQLite3_GetRecordsetValueByIndex(0, @rs)
          id = rs\sValue
        EndIf
        If SQLite3_GetRecordsetValueByIndex(1, @rs)
          name = rs\sValue
        EndIf
        If SQLite3_GetRecordsetValueByIndex(2, @rs)
          gross = rs\sValue
```

```
        EndIf
        MessageRequester("1. DL: Datenzeile", id+ #TAB$ + name + #TAB$ + gross)
        SQLite3_RecordsetMoveNext(@rs)
      Wend
      SQLite3_ReleaseRecordset(@rs)
    EndIf
  EndIf
  SQLite3_CloseDatabase(handle)
EndIf


DataSection
Sqlite:
IncludeBinary "sqlite3.dll"
EndDataSection
```

Weitere Navigationsbefehle innerhalb eines RecordSets sind

```
Ergebnis.l = SQLite3_RecordsetMoveFirst(@rs)
Ergebnis.l = SQLite3_RecordsetMoveLast(@rs)
Ergebnis.l = SQLite3_RecordsetMovePrevious(@rs)
```

Diese Befehle bewirken eine Änderung des Zeigers auf das erste, letzte oder vorherige Tupel. Mit dem SQLite3_ReleaseRecordset(@rs) wird der benutzte Speicherplatz wieder frei gegeben. Statt mit

```
SQLite3_GetRecordsetValueByIndex(Index.l. @rs)
```

kann der Zeiger über

```
SQLite3_GetRecordsetValueByName(Name.s, @rs)
```

gesetzt werden. Jetzt kann der Wert über rs\sValue ausgelesen werden.

Auch in SQLite können Sichten abgespeichert werden. Die Vorgehensweise ist entsprechend der Access-Datenbank.

```
sql = "CREATE VIEW Kleinste as SELECT * FROM Zwerg z WHERE EXISTS (SELECT * FROM
Zwerg x WHERE x.ZwergID <> z.ZwergID and z.Groesse < x.Groesse) ORDER BY
z.Groesse desc"
If SQLite3_Execute(sql, handle) = #False
  MessageRequester("Fehler", SQLite3_GetLastMessage())
EndIf
```

Transaktionenkonzept

Mehrfache Operationen, die ein Benutzer oder Anwendungsprogramm an das DBMS richtet, können zu Konsistenzproblemen führen. Das gleiche gilt, wenn Fehler beim Ausführen einer Abfrage auftreten oder die Hardware plötzlich versagt. Aus diesem Grunde wurde das Transaktionenkonzept entwickelt. Es besteht im wesentlichen aus dem ACID-Prinzip und soll ausgehend von einem vorher konsistenten Zustand nach Ausführung der Transaktion zu einem konsistenten Zustand führen. Hinter dem ACID-Prinzip stehen die Forderungen nach

- Unteilbarkeit (atomicy)
- Konsistenz (consistency)
- Unsichtbarkeit (isolation)
- Dauerhaftigkeit (durability)

Eine Transaktion ist eine Folge von Operationen und ist solange nicht sichtbar, soweit sie nicht abgeschlossen ist. Das Ergebnis der Transaktion ist dauerhaft und führt von einen konsistenten Zustand in einen nächsten konsistenten Zustand. Letztlich ist die Transaktion selbst wie ein Vorgang zu sehen und damit unteilbar. Aber genug Theorie - wann spielt es eine besondere Rolle? Gerade bei Mehrbenutzersystemen. Eine Transaktion wird mit dem SQL-Statement BEGIN TRANSACTION eingeleitet und mit COMMIT beendet. Für unsere Zwergentabelle müssen wir vor dem Einfügen des ersten Werts die Transaktion aufrufen.

```
IF SQLite3_Execute("BEGIN TRANSACTION", handle)
...
   SQLite3_Execute("COMMIT", handle)
EndIf
```

Wenn ein Fehler aufgetreten ist, dann kann mit dem ROLLBACK der letzte konsistente Zustand wiederhergestellt werden.

```
If SQLite3_Execute("BEGIN TRANSACTION", handle)
  For i = 1 To 4
    Read zwName
    Read zwGroesse
    sql = "INSERT INTO Zwerg (Name, Groesse) VALUES ('" + zwName + "', " +
zwGroesse + ")"
    ;Der Autowert ID wird automatisch hochgezählt.
    If SQLite3_Execute(sql, handle) = #False
      MessageRequester("Fehler", "Beim Einfügen eines Wertes trat ein Fehler
auf." + #CRLF$ + SQLite3_GetLastMessage())
      SQLite3_Execute("ROLLBACK", handle)
    EndIf
  Next i
  SQLite3_Execute("COMMIT", handle)
EndIf
```