MYSQL–Datenbanken mit Delphi ohne Fremdkomponenten

MySQL mit Delphi

Michael Puff
mail@michael-puff.de

2010-03-26

Inhaltsverzeichnis

1 Vorwort

Wer eine Delphi Personal Version von Borland einsetzt, wird schon festgestellt haben, dass die Datenbank-Komponenten fehlen. Selbst Komponentensammlungen von anderen Anbietern setzen meist auf `TDataSource` und Konsorten auf, so dass auch diese Komponenten nicht eingesetzt werden können. Wer aber trotzdem nicht auf den Komfort einer SQL Datenbank verzichten möchte, der kann auf den MySQL-Datenbank-Server[1] zurückgreifen. Allerdings hat man die datensensitiven Kontrolls nicht zur Verfügung, aber es geht auch ohne, wenn auch nicht ganz so komfortabel.

In diesem Tutorial will ich, am Beispiel einer kleinen Adressdatenbank, eine erste Einführung in die Arbeit mit dem mySQL–Server und der Datenbankabfragesprache SQL geben. SQL– und Datenbank–Kenntnisse werden nicht vorausgesetzt – lediglich Grundkenntnisse in Delphi wären von Vorteil. Dabei werden wir in unserer Datenbank nur eine Tabelle haben mit den gängisten Datentypen und die wichtigsten Operationen mit dieser Datenbank ausführen: eine Datenbank anlegen, eine Tabelle mit Felder anlegen, eine Tabelle wieder löschen, Datensätze einfügen, Datensätze löschen, Datensätze editieren und Datensätze suchen und filtern. Als Tool zum Verwalten der Datenebanken, empfehle ich das Programm von EMS[2]: EMS MySQL Manager[3]. Die Lite Version gibt es als Freeware auf der Homepage von EMS zum Download.

[1] http://dev.mysql.com/
[2] http://sqlmanager.net/
[3] http://sqlmanager.net/products/mysql/manager/

2 Was wir brauchen – Vorbereitungen

Um mit dem mySQL Server arbeiten zu können, brauchen wir erstmal den Server selber. Runterladen kann man ihn sich auf der Homepage des mySQL-Servers[1]. Ich empfehle die vollständige Windows Installation, die man hier[2] findet. Wird der Server nicht in das Verzeichnis `C:\mysql` installiert, ist es nötig noch eine Konfigurationsdatei im Windowsordner abzulegen, dies muss man eventuell von Hand machen, sollte nach dem Setup nicht gleich der Konfigurationsassitent starten. Bei mir befindet sich der Server und seine Daten im Verzeichnis `C:\Programme\MySQL\MySQL Server 4.1` entsprechend sieht die Konfigurationsdatei aus:

```
[mysqld]
basedir=C:/Programme/MySQL/MySQL Server 4.1/
datadir=C:/Programme/MySQL/MySQL Server 4.1/data/
[WinMySQLAdmin]
Server=C:/Programme/MySQL/MySQL Server 4.1/bin/mysqld-nt.exe!
```

Nach der Installation sollten sich im bin-Verzeichnis die Server-Anwendungsdateien befinden. Davon installieren wir die mysqld-nt.exe als Dienst für Windows. Wir geben dazu in der Konsole folgendes ein:

```
mysqld-nt --install
```

Danach sollte sich der mySQL Dienst in der Liste der lokalen Dienste wiederfinden. Den Server hätten wir also erstmal installiert und kongiguriert. Fehlen noch die für Delphi nötigen Header-Übersetzungen der C-API-Funktionen für den mySQL-Server.

Um nun unter Delphi den mySQL–Server einsetzten zu können brauchen wir noch die C-Header-Übersetzungen der C–Headerdateien. Diese bfindet sich mit in dem Archiv mit dem Delphi Demo–Programm: MySQL_mit_Delphi_Demos.zip. Diese Unit wird in unser Delphi-Projekt einfach eingebunden. Damit das ganze aber funktioniert, wird noch Client-Library libmysql.dll benötigt. Diese befindet sich im Verzeichnis
`C:\Programme\MySQL\MySQL Server 4.1\lib\opt` und muss sich im Suchpfad von Windows befinden. Also entweder im Anwendungsverzeichnis selber oder zum Beispiel im Windows-Ordner.

Damit wären unsere Vorbereitungen soweit abgeschlossen und wir können uns der eigentlichen Programmierung zu wenden.

[1] http://dev.mysql.com/
[2] http://dev.mysql.com/downloads/mysql/4.1.html

3 Mit dem Server verbinden

Als aller erstes muss man sich mit dem Datenbank-Server verbinden. Der MySQL-Server stellt dazu die Funktion `mysql_real_connect` zur Verfügung. Parameter siehe Tabelle 3.1, Seite 5.

Parameter	Datentyp	Bedeutung
Datenbankbeschreiber	PMYSQL	MYSQL-Struktur.
Host	PChar	Der Wert von Host kann entweder ein Hostname oder eine IP-Adresse sein. Wenn Host NULL oder die Zeichenkette «localhost» ist, wird eine Verbindung zum lokalen Host angenommen.
User	PChar	Benutzer der Datenbank.
Password	PChar	Passwort des Benutzers.
Datenbankname	PChar	Name der Datenbank, zu der verbunden werden soll. Wird ein leerstring angegeben, wird nur zum Server verbunden.
Port	Cardinal	Port-Nummer über den die TCP/IP Verbindung hergestellt werden soll.
Unix-Socket	PChar	Legt den Socket bzw. die named pipe fest, wenn dieser Parameter nicht leer ist.
Flag	Cardinal	Ist üblicherweise 0. Er kann aber auch eine Kombination von Flags wie unter anderem CLIENT_COMPRESS oder CLIENT_SSL enthalten.

Tab. 3.1: Paramter mysql_real_connect

Bevor die Funktion `mysql_real_connect` aufgerufen werden kann, muss der Datenbankbeschreiber erst mit `mysql_init` initialisiert werden.

Eine Funktion zum Verbinden mit dem Datenbankserver könnte nun so aussehen:

```
function Connect(Descriptor: PMYSQL; const Host, User, PW, DB: string; Port:
  Integer): PMYSQL;
begin
  result := mysql_real_connect(Descriptor, PChar(Host), PChar(User), PChar(PW),
    PChar(DB), PORT, nil, 0);
end;
```

Bzw. vollständig mit `mysql_init`:

```
Descriptor := mysql_init(nil);
Descriptor := Connect(Descriptor, HOST, USER, PW, '', PORT);
if Assigned(Descriptor) then
begin
...
...
```

Dieser Code befindet sich in der Demo Adress-Datenbank-Anwendung im OnPaint-Ereignis der Form, man kann ihn natürlich auch in das OnCreate-Ereigniss schreiben. Um entwaige Fehler mit zu loggen, habe ich mir eine Prozedur geschrieben, die immer den aktuellen Vorgang in ein Memo schreibt. Dies kann bei der Fehlersuche ganz nützlich sein, zeigt aber auch, was gerade passiert. Die Variable `Descriptor` ist global, da wir sie auch bei anderen Funktionsaufrufen benötigen.

Habe wird erstmal eine Verbindung aufgebaut, können wir diverse Informationen abfragen (siehe Tabelle 3.2, Seite 6):

Funktion	**Beschreibung**
mysql_get_server_info	Gibt eine Zeichenkette zurück, die die Server-Versionsnummer bezeichnet.
mysql_get_host_info	Gibt eine Zeichenkette zurück, die den Typ der benutzten Verbindung beschreibt, inklusive des Server-Hostnamens.
mysql_get_client_info	Gibt eine Zeichenkette zurück, die die Version der Client-Bibliothek bezeichnet.
mysql_get_proto_info	Gibt die Protokollversion zurück, die von der aktuellen Verbindung benutzt wird.
mysql_character_set_name	Gibt den vorgabemäßigen Zeichensatz für die aktuelle Verbindung zurück.

Tab. 3.2: Informationen über eine MySQL-DB erhalten

Getrennt wird eine Verbindung mit dem Server mit der Funktion `mysql_close`. Als Parameter wird der Datenbankbeschreiber erwartet, der unsere geöffnete Verbindung zur Datenbank beschreibt.

4 Anlegen einer Datenbank

Die bisherigen Funktionen sind abhängig vom verwendeten Datenbank-Server. Benutzen wor einen anderen Server können sie entsprechend anders aussehen. Das Prinzip bleibt aber das gleiche. Eine Datenbank wird über ein SQL-Query, eine Abfrage, angelegt und ist weitgehend Server unabhängig. Allerdings haben einige Herstellen die herkömmliche SQL-Syntax erweitert bzw. eine für ihre Server spezifische Syntax implementiert, die nicht unbedingt kompatible sein muss. Die grundlegenden SQL-Befehle sollten allerdings kompatibel sein, so dass die hier vorkommenden SQL-Befehle entsprechend übertragen werden können. Alle weiteren direkten Datenbankoperationen werden mittels eines Querys ausgeführt.

Die SQL-Syntax zum Anlegen einer Datenbank lautet:

```
CREATE DATABASE <name>
```

Wobei `name` den Namen der Datenbank bezeichnet.

4.1 Ausführen eines Querys

Querys werden bei mySQL mittels der Funktion `mysql_real_query` ausgeführt (Tabelle 4.1, Seite 7).

Parameter	Datentyp	Bedeutung
Datenbankbeschreiber	PMYSQL	MYSQL-Struktur.
Query	PChar	Die Zeichenkette des Querys.
length	Cardinal	Länge der Zeichenkette des Querys.

Tab. 4.1: Parameter mysql_real_query

Der Rückgabewert ist entweder 0, wenn kein Fehler oder ungleich 0, wenn ein Fehler aufgetreten ist. Der Rückgabewert sagt allerdings nichts über die Art des Fehlers aus, der aufgetreten ist. Um eine aussagekräftige Fehlermeldung zu erhalten, kann man folgende MySQL-Funktionen aufrufen (Tabelle 4.2, Seite 8):

Beide Funktionen erwarten als Parameter den Datenbankbeschreiber, der gerade geöffneten Serververbindung. Konkret im Quellcode könnte das nun so aussehen:

```
var
  query: String;
  ErrorCode: Integer;
begin
```

Funktion	Beschreibung
mysql_error	Gibt eine Zeichenkette zurück, die den zu letzt aufgetretenen Fehler beschreibt.
mysql_errno	Gibt den Fehlercode, der zu letzt aufgerufenen Funktion zurück.

Tab. 4.2: MySQL Error Funktionen

```
  query := 'CREATE DATABASE' + ' ' + DBNAME;
  ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
DBNAME ist hier eine Konstante, die den Namen der Datenbank bezeichnet:
const
  DBNAME       = 'AdressDB';
```

Ich habe mir zum Ausführen von Querys eine extra Funktion geschrieben: ExecQuery:

```
function ExecQuery(const Datenbank, query: string; var Cols: TCols; var Rows:
  TRows): Boolean;
```

Im Moment ist für uns aber erst mal nur die Zeile von Interesse, in der der Query ausgeführt wird. Alles weiter werde ich im Verlauf des Tutorials erläutern.

5 Anlegen einer Tabelle

Eine Datenbank ist im eigentlichen Sinne nur ein Container, ein Container für Tabellen, die die eigentlichen Daten enthalten. Das heißt eine Datenbank kann mehrere Tabellen enthalten. Diese können untereinander verknüpft sein. So könnte man sich eine Datenbank vorstellen die in einer Tabelle Lieferanten und ihre Adressen verwaltet und in einer zweiten Tabelle werden die Produkte dieser Lieferanten aufgenommen. Dies macht in sofern Sinn, als dass ein Lieferant mehrere Produkte liefert und so zu jedem Produkt nicht noch die einzelnen Adressdaten des Lieferanten gespeichert werden müssen, sondern dass diese über eine gemeinsame Spalte in beiden Tabellen mit einander verknüpft werden. So wird vermieden, dass Daten mehrfach in einer Tabelle abgelegt werden. Man spricht dann auch von einer normalisierten Datenbank. Wir wollen es aber erstmal für den Einstieg einfach halten und nur mit einer Tabelle arbeiten.

Eine Tabelle besteht aus sogenannten Feldern oder auch Spalten genannt. Diese Felder definieren, was wir in unserer Tabelle ablegen können. Da wir eine Adressdatenbank erstellen wollen, habe ich folgende Felder gewählt (Siehe Tabelle 5.1, Seite 9):

Parameter	**Datentyp**	**Bedeutung**
id	eindeutige fortlaufende Datensatznummer	int NOT NULL AUTO_INCREMENT
name	Familienname	varchar(20)
vorname	Vorname	varchar(20)
strsse	Strasse	varchar(55)
plz	Postleitzahl	int
ort	Wohnort	varchar(50)
telefon1	Festnetznummer	varchar(17)
telefon2	Handynummer	varchar(17)
fax	Faxnummer	varchar(17)
email1	erste E–Mail Adresse	varchar(50)
email2	zweite E–Mail Adresse	varchar(50)
url	Homepage	varchar(50)
gebdat	Geburtsdatum	date
firma	Firma	varchar(25)
ts	Timestamp	timestamp

Tab. 5.1: Felder Tabelle Adress-Datenbank

Auf ein Feld der Tabelle möchte ich noch mal etwas genauer eingehen. Es handelt sich um das Feld `id`. Dieses hat noch zwei zusätzliche Attribute bekommen: `NOT NULL` und `AUTO_INCREMEMNT`. `NOT NULL` bedeutet, dass es keinen Datensatz geben darf, in dem

dieses Feld leer ist. Der Grund ist der, dass mit Hilfe dieses Feldes jeder Datensatz eindeutig identifiziert werden soll. Wozu wir das brauchen, sehen wir, wenn wir Datensätze editieren oder löschen wollen, spätestens dann müssen wir nämlich einen Datensatz eindeutig identifizieren können. Wird nichts weiter angegeben, dann darf dieses Feld auch leer sein. Damit wir uns nicht selber darum kümmern müssen, habe ich dem Feld noch zusätzlich das Attribut `AUTO_INCREMENT` gegeben. Das heißt, die Datenbank sorgt selbständig dafür, dass dort immer ein Wert enthalten ist, der um eins höher ist, als der letzte vergebene Wert für dieses Feld, somit sollte eine Eindeutigkeit gewährleistet sein.

Bei der Wahl der Datentypen und der Größe der Felder habe ich erstmal nicht auf die Optimierung geachtet. Bei groß Datenbanken, die optimale Wahl der Datentypen und Größe der Felder sehr zur Performance beitragen.

5.1 Verfügbare SQL Datentypen

Die Datentypen und ihre Bedeutung kann man aus der folgenden Tabelle entnehmen (Siehe Tabelle 5.2, Seite 11):

5.2 Schlüssel und Indizes

Beschäftigt man sich mit Datenbanken wird man unweigerlich von Schlüssel, primär Schlüssel, sekundär Schlüssel und Indizes hören. Was ein Schlüssel bzw. Index ist, will ich hier kurz erläutern und versuchen zu erklären, wozu sie da sind und warum man sie benötigt.

Die Begriffe Schlüssel und Index sind eigentlich Synonyme und bezeichnen somit das gleiche. Ich werde im weiteren Verlauf den Begriff Index benutzen.

Indiziert man eine Spalte, wird diese separat, sortiert abgelegt und verwaltet. Das hat zur Folge das Abfragen schneller und effizienter berabeitet werden können. Ein kleines Beispiel:Nehmen wir an ich will alle Datensätze mit der ID zwischen 5 und 10 haben. Man kann davon ausgehen, dass diese in einer unsortierten Tabelle über die ganze Tabelle verstreut vorliegen. Stellt man sich jetzt vor dass jeder Datensatz als eine Karteikarte ais vorliegt, kann man sich leicht vorstellen, dass Aufgabe die Karteikarten mit den IDs zwischen 5 und 10 rauszusuchen recht mühsam ist. Sind die Karteikarten aber nach den IDs sortiert, erleichtert einem das die Arbeit ungemein. Man sollte also die Felder einer Tabelle indiziere, die am häufigsten sortiert bzw. abgefragt werden. Bei unserer Adressdatenbank wären das die Felder `name` und `vorname`. Also sollte auf genau diese Felder ein Index gesetzt werden. Hinzukommt, dass man über die ID eines Datensatzes auf den selbigen zugreift. Also sollte auch dieses Feld indiziert werden. Ein großer Nachteil von Indizes ist die Tatsache, dass sämtliche Datenänderungen langsamer werden (Die Sortierung muss ja neu aufgebaut werden). Man muss also immer abwiegen, ob ein Index auf Feld Sinn macht.

Typ	Beschreibung
TINYINT	-128 .. 127
TINYINT UNSIGNED	0 .. 255
INT	-2.147.483.648 .. 2.147.483.647
INT UNSIGNED	0 .. 4.294.967.295
BIGINT	-3402823e+31 .. 3402823e+31
DECIMAL(length,dec)	Kommazahl der Länge length und mit dec Dezimalstellen; die Länge beträgt: Stellen vor dem Komma + 1 Stelle für Komma + Stellen nach dem Komma
VARCHAR(NUM) [BINARY]	Zeichenkette mit max NUM Stellen (1<= NUM <=255). Alle Leerstellen am Ende werden gelöscht. Solange nicht BINARY angegeben wurde, wird bei Vergleichen nicht auf Groß-/Kleinschreibung geachtet.
TEXT	Text mit einer max. Länge von 65535 Zeichen
MEDIUMTEXT	Text mit einer max. Länge von 16.777.216 Zeichen
TIME	Zeit; Format: HH:MM:SS, HHMMSS, HHMM oder HH
DATE	Datum; Format: YYYY-MM-DD, wobei - jedes nicht numerische Zeichen sein kann
TIMESTAMP	Setzt einen Datumswert beim Einfügen/Updaten einzelner Felder automatisch auf das Systemdatum. Format: YYYYMMDDHHMMSS. Wenn mehrere Felder den Typ TIMESTAMP haben, wird immer nur das erste automatisch geändert!

Tab. 5.2: MySQL Datentypen

5.3 CREATE TABLE

Kommen wir nun zum eigentlichen anlegen der Tabelle. Dies geschieht auch wieder über einen Query. Wie man unter mySQL einen Query ausführt, habe ich schon unter Punkt 5.1 erläutert. Deswegen beschränke ich mich im Folgenden nur noch auf die SQL Syntax:

```
CREATE TABLE <tabellenname>(<feldname> <datentyp> <weitere Attribute>, <feldname>
    <datentyp> <weitere Attribute>, ...)
```

Indizes werden mit dem SQL-Schlüsselwort `KEY` oder `INDEX` definiert. Dann folgt der Name des Indexes und Klammern das zu indizierende Feld:

```
PRIMARY KEY(id), KEY idx_name (name), KEY idx_vorname (vorname)!.
```

Für unsere Datenbank würde der Query jetzt vollständig so aussehen:

```
CREATE TABLE Kontakte(id INT NOT NULL AUTO_INCREMENT, name varchar(20), vorname
   varchar(20), strasse varchar(55), plz int, ort varchar(50),
```

```
telefon1 varchar(17), telefon2 varchar(17), fax varchar(17), email1 varchar(50),
    email2 varchar(50), url varchar(50), gebdat date,
firma varchar(25), ts timestamp, PRIMARY KEY(id), KEY idx_name (name), KEY
    idx_vorname (vorname))
```

5.4 Löschen einer Tabelle

Gelöscht wird eine Tabelle mit dem SQL-Befehl `DROP TABLE <tabellenname>`.

6 Datensätze einfügen, editieren und löschen

6.1 Einen Datensatz einfügen

Auch das Einfügen eines Datensatzes geschieht wieder über ein Query:

```
INSERT INTO <tabellenname>(<feldname>, <feldname>,...) VALUES('<wert>', '<wert>',
    ...)
```

Man gibt also die Tabelle an, in die man einen Datensatz einfügen will und dann in Klammern die betroffenen Felder als Parameter und dann als Parameter für das Schlüsselwort VALUES die entsprechenden Werte. das war eigentlich schon fast alles. Allerdings hat der Query keine Ahnung davon, mit welche Datenbank wir arbeiten wollen. Also müssen wir sie vorher auswählen. Dies geschieht mit der mySQL-Funktion `mysql_select_db`. Diese Funktion erwartet zwei Parameter. Als ersten wieder den Beschreiber vom Datentyp PSQL und als zweiten den Datenbanknamen. Der Rückgabewert ist wieder 0 oder ungleich 0 im Fehlerfall.

In dem Demo zu diesem Tutorial habe ich das wieder in eine separate Funktion ausgelagert:

```
function Insert(Kontakt: TKontakt): Boolean;
var
  query        : string;
  ErrorCode    : Integer;
begin
  ErrorCode := mysql_select_db(Descriptor, DBNAME);
  if ErrorCode = 0 then
  begin
    with Kontakt do
    begin
      query := 'INSERT INTO Kontakte(Name, Vorname, Strasse, Plz, Ort, ' +
        'Telefon1, Telefon2, EMail1, Gebdat) VALUES('
        + QuotedStr(Name)
        + SEP + QuotedStr(Vorname)
        + SEP + QuotedStr(Strasse)
        + SEP + IntToStr(PLZ)
        + SEP + QuotedStr(Ort)
        + SEP + QuotedStr(Telefon1)
        + SEP + QuotedStr(Telefon2)
        + SEP + QuotedStr(EMail1)
        + SEP + QuotedStr(GebDat)
        + ')';
    end;
```

```
    ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
  end;
  result := ErrorCode = 0;
end;
```

Diese Funktion erwartet als Parameter einen Record, der als Felder die Felder der Tabelle enthält:

```
type
  TKontakt = record
    ID: Integer;
    Name: string;
    Vorname: string;
    Strasse: string;
    Plz: Integer;
    Ort: string;
    Land: string;
    Telefon1: string;
    Telefon2: string;
    Fax: string;
    EMail1: string;
    EMail2: string;
    URL: string;
    Gebdat: string;
    Firma: string;
    Position: string;
  end;
```

Wie wir sehen, wird zu erst die Datenbank ausgewählt und dann der Query ausgeführt. Wichtig ist dabei noch, dass auch Datums-Werte als Zeichenketten übergeben werden, unabhängig davon dass dieses Feld in der Tabelle als date-Datentyp deklariert wurde.

6.2 Einen Datensatz editieren

Die allgemeine Syntax einen Datensatz zu editieren lautet:

```
UPDATE <tabellenname> SET <feldname>='<wert>' WHERE <feldname>='<wert>'
```

Wichtig hier bei ist die `WHERE`-Klausel mit der wir den SQL-Ausdruck nur auf einem bestimmten Datensatz anwenden. Würden wir dies weglassen, wären alle Datensätze betroffen. Günstiger weise nimmt man hier ein Feld, welches einen Datensatz eindeutig identifiziert. Man kann auch Felder der Tabelle kombinieren, um einen Datensatz eindeutig zu identifizieren. Um mir die Arbeit nicht unnötig schwer zu machen, habe ich deswegen in der Tabelle das Feld id definiert, welches einen Datensatz eindeutig identifiziert.

Im Quellcode könnte eine Routine, um einen Datensatz in eine Tabelle einzufügen nun so aussehen:

```
function UpdateRecord(ID: Integer; Kontakt: TKontakt): Boolean;
var
  query        : string;
```

```
  ErrorCode    : Integer;
begin
  log(Format('Datensatz %d ändern', [ID]));
  ErrorCode := mysql_select_db(Descriptor, DBNAME);
  if ErrorCode = 0 then
  begin
    with Kontakt do
      query := 'UPDATE Kontakte SET name=' + QuotedStr(Name) +
        SEP + 'vorname=' + QuotedStr(Vorname) +
        SEP + 'strasse=' + QuotedStr(Strasse) +
        SEP + 'plz=' + QuotedStr(IntToStr(PLZ)) +
        SEP + 'ort=' + QuotedStr(Ort) +
        SEP + 'telefon1=' + QuotedStr(Telefon1) +
        SEP + 'telefon2=' + QuotedStr(Telefon2) +
        SEP + 'email1=' + QuotedStr(EMail1) +
        ' WHERE ID =' + IntToStr(ID) + '';
    ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
  end;
  result := ErrorCode = 0;
end;
```

6.3 Einen Datensatz löschen

Das Löschen eines Datensatzes gestaltet sich entsprechend trivial:

```
DELETE FROM <tabellenname> WHERE <feldname>='<wert>'
```

Die `WHERE`-Klausel ist auch hier wieder von Bedeutung, sonst werden nämlich alle Datensatze gelöscht. Womit wir auch schon wissen, wie man alle Datensätze löscht.

7 Datensätze filtern

Jede noch so umfangreiche Datenbank wäre wertlos und überflüssig, wenn sie nicht die Möglichkeit bieten würde gezielt nach Informationen zu suchen und / oder die enthaltenen Informationen nach bestimmten Kriterien zu filtern. Denn was will man mit Informationen, wenn kein gezielter Zugriff möglich ist? Um Datensätze auszuwählen und zu filtern, bietet SQL nun sehr umfangreiche Möglichkeiten.

`SELECT` tut eigentlich genau das, was es schon aussagt, es selektiert. Und zwar selektiert es Datensätze aus der angegebenen Tabelle. Ein SELECT-Statement hat folgende allgemeine Syntax:

```
SELECT <feldname>, <feldname>, <...> FROM <tabellenname> WEITERE_SQL_ANWEISUNGEN
```

Wollen wir uns zum Beispiel nur alle Nachname und die Vornamen ausgeben lassen, so sieht das passende SQL-Statement so aus:

```
SELECE name, vorname FROM kontakte
```

Will man sich alle Spalten einer Tabelle ausgeben lassen, kann man anstatt alle Spaltennamen hinzuschreiben auch ein «*» als allgemeinen Platzhalter angeben:

```
SELECT * FROM kontakte
```

Allerdings hilft uns das auch erstmal nicht viel weiter, wenn wir einen bestimmten Datensatz suchen, da sich die Datenmenge nicht reduziert hat. Ein erster Schritt wäre wohl erst mal die Datensätze zu sortieren.

7.1 ORDER BY

Mit `ORDER BY` wird festgelegt, nach welcher Spalte bzw. welchen Spalten sortiert werden soll. Mit `ASC` werden die Zeilen aufsteigend, mit `DESC` absteigend sortiert. Ist nichts angegeben, wird aufsteigend sortiert. Hier ein einfaches Beispiel, Datensätze nach dem Nachnamen sortieren:

```
SELECT name, vorname FROM kontakte ORDER BY name
```

Will man nach mehreren Spalten gleichzeitig sortieren, gibt man die weiteren Spalten einfach durch ein Komma getrennt mit an:

```
SELECT name, vorname FROM kontakte ORDER BY name, ort
```

Womit wir schon einen Schritt weiter wären. In einer sortierten Tabelle hat man wem falls schon mal eine Chance gezielt etwas zu finden. Aber so ganz das Wahre ist es noch nicht.

7.2 WHERE

Wenn wir gezielt Datensätze suchen bzw,. herausfiltern wollen, erweitern wir unser SQL-Statement mit dem Zusatz `WHERE`:

```
SELECT * FROM <tabellenname> WHERE <feldname>='<wert>'
```

Wollen wir zum Beispiel alle Meiers aus unserer Adress-Datenbank haben, könnte das entsprechende SQL-Statement wie folgt aussehen:

```
SELECT * FROM kontakte WHERE name='meier'
```

Ausdrücke können auch mit AND, OR und NOT miteinander verknüpft werden. Desweiteren ist es möglich Platzhalter zu verwenden: «_» steht für ein beliebiges Zeichen und «%» für eine beliebige Zeichenkette. Auch kann man natürlich `WHERE` noch mit `ORDER BY` und weiteren SQL-Ausdrücken kombinieren.

7.2.1 LIKE

Immer dann, wenn man in Textfeldern im Suchmuster Platzhalter oder Jokerzeichen verwenden will, können die Vergleichsoperatoren nicht verwendet werden. Statt dessen muss man in diesen Fällen auf den Operator `LIKE` zurückgreifen. Sollen zum Beispiel alle Personen mit der Vorwahl «0561» gefunden werden, sähe dies so aus:

```
SELECT name, vorname, telefon1 FROM kontakte WHERE telefon1 LIKE '%0561%'
```

7.2.2 BETWEEN

Ein weiterer Operator ist BETWEEN. BETWEEN wählt alle Spalten aus die zwischen den oberen und unteren Wert liegen:

```
SELECT name, vorname, gebdat FROM kontakte WHERE gebdat BETWEEN '1980-01-01' and
    '2005-01-01'
```

Diese Abfrage liefert uns alle Personen, die zwischen 1. Januar 1980 und 1. Januar 2005 geboren wurden. Man beachte die Angabe des Datums: yyyy-mm-dd. So und nicht anders muss es angegeben werden, damit es der mySQL Server versteht.

7.2.3 IN

Der letzte Operator aus dieser Gruppe ist der IN-Operator. Er wird benutzt, wenn man nicht mit einem einzelnen Wert, sondern mit einer Wertemenge vergleichen will. Beispiel: Wir wollen alle Personen die entweder «Schmidt» oder «Meier» heißen. Mit dem Vergleichsoperator «=» und einer Oder–Verknüpfung wird das bei vielen Werten, die zu vergleichen sind, schnell recht unübersichtlich. Einfacher geht es mit dem IN–Operator:

```
SELECT name, vorname FROM kontakte WHERE name IN ('schmidt', 'meier')
```

8 Die Demo-Anwendung „AdressDBSQL“

Wie schon in der Einleitung erwähnt, basiert dieses Tutorial auf einer kleinen Adress-Datenbank als Demo. Ich will an dieser Stell noch mal kurz die beiden zentralen Routinen der Anwendung, `ExecQuery` und `FillGrid`, erläutern.

8.1 ExecQuery

Die Funktion `ExecQuery` führt eine Abfrage aus:

```
function ExecQuery(const Datenbank, query: string; var Cols: TCols; var Rows:
    TRows):
  Boolean;
var
  MySQLRes     : PMYSQL_RES;
  MySQLRow     : PMYSQL_ROW;
  AffectedRows : Int64;
  ColCount     : Cardinal;
  Field        : PMYSQL_FIELD;
  i            : Integer;
  j            : Integer;
  ErrorCode    : Integer;
begin
  // Datenbank auswählen
  ErrorCode := mysql_select_db(Descriptor, PChar(Datenbank));
  if ErrorCode = 0 then
  begin
    // Query ausführen
    ErrorCode := mysql_real_query(Descriptor, PChar(query), length(query));
    if ErrorCode = 0 then
    begin
      // Query speichern
      MySQLRes := mysql_store_result(Descriptor);
      if Assigned(MySQLRes) then
      begin
        // zurückgelieferte Anzahl der Spalten
        ColCount := mysql_num_fields(MySQLRes);
        SetLength(Cols, ColCount);
        // Spalten-Array füllen
        for i := 0 to ColCount - 1 do
        begin
          Field := mysql_fetch_field_direct(MySQLRes, i);
          Cols[i] := Field.Name;
        end;
        // Anzahl der betroffenen Zeilen ermitteln
        AffectedRows := mysql_affected_rows(Descriptor);
        SetLength(Rows, ColCount, AffectedRows);
```

```
        // Zeilen-array füllen
        for i := 0 to ColCount - 1 do
        begin
          for j := 0 to AffectedRows - 1 do
          begin
            MySQLRow := mysql_fetch_row(MySQLRes);
            Rows[i, j] := MySQLRow[i];
          end;
          mysql_real_query(Descriptor, PChar(query), length(query));
          MySQLRes := mysql_store_result(Descriptor);
        end;
        log(Format('Betroffene Zeile(n): %d',
          [mysql_affected_rows(Descriptor)]));
        // gespeicherte Abfrage wieder freigeben
        mysql_free_result(MySQLRes);
      end
    end
  end;
  result := ErrorCode = 0;
end;
```

Sie erwartet den Namen der Datenbank auf die die Abfrage ausgeführt werden soll, die Abfrage selber als String und dann die Spalten (Cols) und Zeilen (Rows), die die zurückgegeben Spalten bzw. die zurückgegebenen Zeilen beinhalten. Dabei ist der Parameter `Cols` vom Typ `TCols`, welcher ein eindimensionales dynamsiches String-Array ist und der Parameter `Rows` ist vom Typ `TRows`, welcher ein zweidimensionales dynamisches Array ist:

```
type
  TRows = array of array of string; // [Cols, Rows]
  TCols = array of string;
```

8.2 FillGrid

Die Prozedur FillGrid ist nun letztendlich dafür zuständig die Ergebnissmenge, die `ExecQuery` in den Parametern `Cols` und `Rows` zurückliefert, in einem StringGrid auszugeben.

```
procedure FillGrid(SG: TStringGrid; Cols: TCols; Rows: TRows);
var
  i, j         : Integer;
begin
  SG.ColCount := 0;
  SG.RowCount := 0;
  if Assigned(Rows) then
  begin
    // Wir brauchen eine Zeile mehr für die Spaltenüberschriften
    SG.RowCount := length(Rows[0]) + 1;
    SG.ColCount := length(Cols);
    SG.FixedRows := 0;
    // Spaltenüberschriften in die erste Zeile schreiben
    for i := 0 to length(Cols) - 1 do
    begin
      SG.Cols[i].Add(Cols[i]);
```

```
      SG.Cells[i, 0] := Cols[i];
    end;
    // zwei-dimensionales Zeilen-Array in den Zellen ausgeben
    for i := 0 to length(Cols) - 1 do
    begin
      for j := 0 to length(Rows[0]) - 1 do
      begin
        SG.Cells[i, j + 1] := Rows[i, j];
      end;
    end;
  end;
end;
```