15 dbExpress

Mit den Komponenten von *dbExpress* können Sie auf verschiedene SQL-Server zugreifen:

- In der Professional-Version haben Sie die DLLs für den Zugriff auf *InterBase* und *MySQL*.
- In der Enterprise-Version können Sie zusätzlich noch auf *Oracle* und *DB2* zugreifen.

Im Gegensatz zur BDE ist *dbExpress* für Cross-Plattform-Entwicklung geeignet – die Komponenten gibt es nicht nur bei der VCL, sondern auch bei CLX, und somit können Sie aus demselben Code Windows- und Linux-Anwendungen erstellen. Bei der Programmierung der Treiber hat sich Borland folglich auch auf die Datenbanksysteme beschränkt, die sowohl unter Windows als auch unter Linux verfügbar sind.

Einfache Installation

dbExpress zeichnet sich durch eine einfache Installation aus:

- Wenn im Betrieb zwischen verschiedenen Datenbanksystemen hin- und hergeschaltet werden muss oder während der Entwicklung noch nicht feststeht, welches Datenbanksystem verwendet werden soll, dann müssen DLLs für alle in Frage kommenden Datenbanksysteme installiert werden, darüber hinaus noch zwei Ini-Dateien.
- In der Regel wird eine Client-Server-Datenbankanwendung nur mit einem Datenbanksystem zusammenarbeiten und dies ist auch schon während der Entwicklung bekannt. In diesem Fall muss nur eine DLL an eine Stelle kopiert werden, an der sie vom Programm gefunden wird – am einfachsten in das Verzeichnis, in dem auch die *exe*-Datei liegt.
- In jedem Fall muss der Client der verwendeten Datenbanksysteme auf den Rechnern installiert sein.
- Die Möglichkeit, auf die externe DLL zu verzichten und stattdessen eine Unit einzubinden, ist vorgesehen, führt aber bei mir auf dem Rechner zu einem *Internen Fehler* bei der Kompilierung.

Der langen Rede kurzer Sinn: Die Installation beschränkt sich in der Regel darauf, eine DLL mitzukopieren.

15.1 Erste Schritte mit dbExpress

dbExpress stellt unidirektionale, nicht editierbare Datenmengen zur Verfügung – für Entwickler, die bislang mit der BDE gearbeitet haben, ist dies eine ziemliche Umstellung. Wir werden uns nun ansehen, was man mit dbExpress tun darf und was nicht.

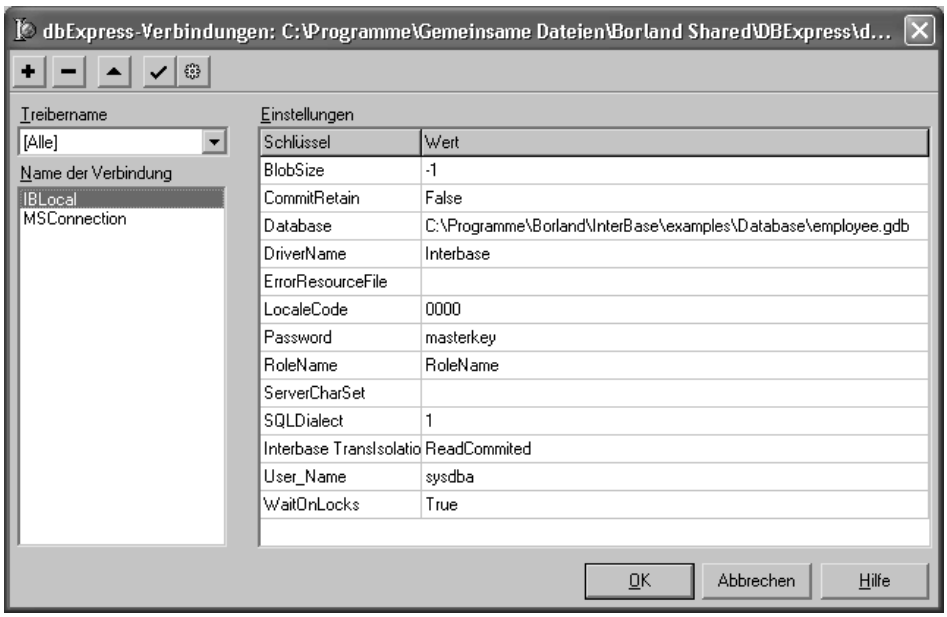
Damit alle Leser dieses Beispiel nachvollziehen können, arbeiten wir mit der Professional-Version und InterBase.

15.1.1 Zugriff auf eine Tabelle

Für unser erstes Beispiel wollen wir auf die Tabelle *projects* aus der Datenbank *employee.gdb* zugreifen.

Einrichten der Datenbankverbindung

Zunächst stellen wir eine Verbindung zur Datenbank her. Dazu legen wir die Komponente *TSQLConnection* auf ein Formular und rufen den Komponenteneditor auf:

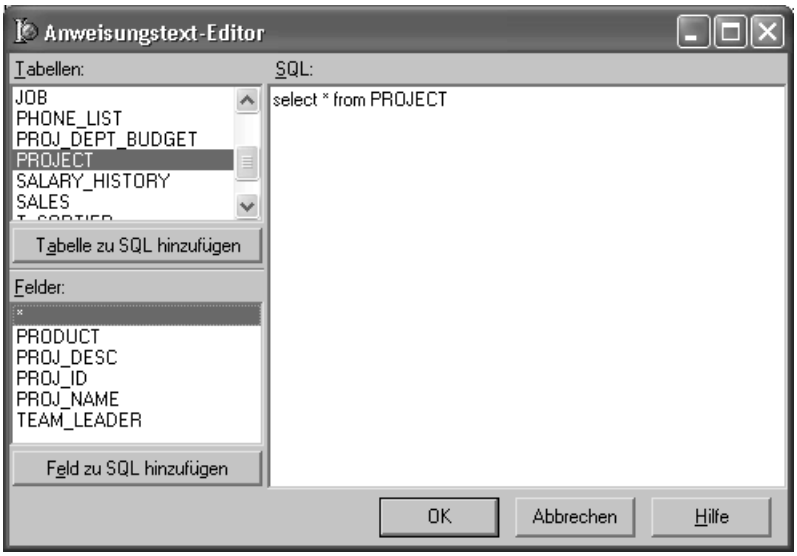


Hier stellen wir unter *Database* den Dateinamen der Datenbank inklusive Pfad ein. Des Weiteren soll der Benutzername und das Password angegeben werden.

Nun schließen wir den Komponenteneditor. Um den lästigen Login-Dialogen zu entgehen, stellen wir *LoginPrompt* auf *false*. Anschließend stellen wir die Verbindung dadurch her, dass *Connected* auf *true* gesetzt wird.

Zugriff auf die Tabelle

Für den Zugriff auf die Tabelle nehmen wir *TSQLDataSet*. Zunächst setzen wir *SQLConnection* auf *SQLConnection1*. Die Eigenschaft *CommandType* belassen wir auf *ctQuery* und öffnen dann den Eigenschaftseditor von *CommandText*:



Hier greifen wir nun zunächst auf die komplette Tabelle *project* zu.

An die Komponente *SQLDataSet1* hängen wir eine *TDataSource*-Instanz, daran wieder eine *TDBNavigator*- und zwei *TDBText*-Komponenten – auf *TDBGrid* soll zunächst verzichtet werden.

Messung der Ausführungsgeschwindigkeit

Nun wollen wir wissen, wie schnell die Abfrage ausgeführt wird, und natürlich auch, wie schnell zum nächsten Datensatz gescrollt wird. Mit diesen Messungen wollen wir *TSQLDataSet* und *TSQLQuery* vergleichen, des Weiteren die BDE-Komponente *TQuery*.

```
procedure TForm1.btnOpenClick(Sender: TObject);
var
  c, t1, t2: int64;
```

```
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  SQLDataSet1.Open;
  QueryPerformanceCounter(t2);
  Label2.Caption := IntToStr(1000000 * (t2 - t1) div c) + ' µs';
end;
```

Mit der Windows-API-Funktion *QueryPerformanceCounter* kann man sehr genaue Zeitmessungen durchführen, so dass wir hier das Ergebnis auf die µs (eine millionstel Sekunde) genau ausgeben können. Das Zeichen µ können Sie in Delphi nicht eingeben, zumindest nicht bei der Standard-Tastaturbelegung. Erzeugen Sie es in einem anderen Programm und kopieren Sie es dann über die Zwischenablage.

Um den Einfluss externer Faktoren zu minimieren, wurden Client und Server auf demselben Rechner ausgeführt. Damit die Daten bereits im Cache liegen – schließlich wollen wir die Verbindung prüfen und nicht den Server –, wurde zunächst eine Anwendung einmal gestartet, die Abfrage geöffnet und ein paar Scroll-Vorgänge durchgeführt. Anschließend wurde die Anwendung geschlossen, erneut gestartet und die dann gemessenen Werte sind die hier veröffentlichten.

Der erste Scroll-Vorgang nach dem Aufruf von *Open* dauert vergleichsweise lange (beispielsweise 757 µs), die dann folgenden Aufrufe von *Next* werden in etwa alle gleich schnell ausgeführt (bei diesem Beispiel etwa 115 µs). Der erste Scroll-Vorgang, der nach dem letzten Datensatz erfolgt, dauert dann wieder lange (780 µs), anschließend weiß das System, dass es nichts mehr zu Scrollen gibt, der Aufruf von Next ist dann in 8 µs erledigt. Die hier veröffentlichten Zahlen sind ein Mittelwert des zweiten, dritten und vierten Scroll-Vorgangs (115 µs).

Bei der Abfrage über alle vorhandenen Spalten dauerte das Öffnen 27 168 µs, das Scrollen 115 µs. Würde *TSQLQuery* anstelle von *TSQLDataSet* verwendet, dann würde das Öffnen bei 27 397 µs liegen und das Scrollen ebenfalls bei 115 µs – die Unterschiede sind geringer als die Streuung der Messwerte, ich habe jedoch den Eindruck gewonnen, dass *TSQLQuery* stets einen Hauch länger braucht.

Würden wir nun mit *TQuery* vergleichen, dann dauert dort das Öffnen 4 323 µs und das Scrollen 349 µs. Das heißt im Klartext, dass *TQuery* das Ergebnis der Abfrage etwa sechsmal so schnell liefert wie *TSQLDataSet* und *TSQLQuery* – dafür geht bei Letzteren das Scrollen etwa dreimal schneller.

Der erste Gedanke könnte nun sein, dass durch das Jokerzeichen * in der SQL-Anweisung die Memo-Spalte mit in die Abfrage aufgenommen wird – dies ist zwar nicht der Fall, aber wir wollen es sicherheitshalber ausschließen:

```
SELECT proj_id, proj_name
  FROM project
```

Einen Hauch schneller wird das Öffnen der Abfrage dadurch (26 862 statt 27 168 µs), und auch das Scrollen geht etwas schneller (110 statt 115 µs), aber ein großer Fortschritt ist auch das nicht.

Persistente TField-Instanzen anlegen

Nun legen wir persistente TField-Instanzen an. Dabei ist es egal, ob wir *TSQLDataSet* oder *TSQLQuery* verwenden, und ob wir die benötigten Spalten einzeln aufzählen oder wir im SELECT-Statement das Jokerzeichen verwenden – der Beschleunigungseffekt ist in etwa derselbe: Statt 26 862 µs dauert das Öffnen der Abfrage nun nur noch 2 884 µs.

Bei *TQuery* fällt die Beschleunigung deutlich geringer aus, die Abfrage ist in 4 273 µs geöffnet.

Wir wollen nun der Frage nachgehen, warum das Anlegen persistenter TField-Instanzen das Öffnen der Datenmenge um etwa den Faktor neun beschleunigt.

15.1.2 Belauschen der Datenbankverbindung

Wir wollen nun protokollieren, welche Anweisungen der Client-DLL von InterBase aufgerufen werden. In der Enterprise-Version von Delphi gibt es dafür die Komponente *TSQLMonitor*. Damit aber auch die Leser, welche die Professional-Version verwenden, dieses Beispiel nachvollziehen können, werden wir stattdessen eine Callback-Funktion verwenden.

```
function TraceCallback(CallType: TRACECat;
  CBInfo: Pointer): CBRType; stdcall;
var
  s: string;
  rec: pSQLTRACEDesc;
begin
  rec := CBInfo;
  s := rec.pszTrace;
  Form1.Memo1.Lines.Add(s);
  result := cbrUSEDEF;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  SQLConnection1.SetTraceCallbackEvent(TraceCallback, 1);
  SQLConnection1.Connected := true;
end;
```

Diese Callback-Funktion belegt ihrerseits eine Verbindung zur Datenbank. Wenn Sie *SQLConnection1* schon zur Entwurfszeit verbunden haben, dann sind nach dem Programmstart drei Datenbankverbindungen offen – oder auch nicht: Denn wenn Sie als Besitzer der Professional-Version nur eine 2-Benutzer-Lizenz haben, lässt sich das Programm nicht mehr starten. Deshalb setzen wir die Eigenschaft *Connected* von *SQLConnection1* auf *false* und verbinden erst zur Laufzeit.

Die Details der Callback-Verbindung sollen uns hier nicht interessieren – wenn Sie so etwas für eigene Projekte benötigen, dann schreiben Sie hier einfach den Code ab.

```
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  SQLConnection1.SetTraceCallbackEvent(nil, 0);
end;
```

Beim Schließen des Programms wird die Protokollfunktion wieder aufgehoben. (Normalerweise würde man alles, was man mit *FormCreate* erzeugt hat, in *FormDestroy* wieder freigeben. In diesem Fall soll die Callback-Funktion jedoch so früh wie möglich »abgeklemmt« werden, so dass wir hier *FormClose* verwenden.)

Für Vergleiche wollen wir uns jetzt noch die Zeilen in *Memo1* zählen lassen:

```
procedure TForm1.Memo1Change(Sender: TObject);
var
  i: integer;
begin
  i := Memo1.Lines.Count;
  if i = 1
    then Label1.Caption := '1 Zeile'
    else Label1.Caption := IntToStr(i) + ' Zeilen';
end;
```

Nach dem Programmstart sehen wir nun, wie die Verbindung zur Datenbank hergestellt wurde:

```
INTERBASE - isc_attach_database
```

Nun Öffnen wir *SQLDataSet1*:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT proj_id, proj_name
  FROM project
```

```
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

Zunächst wird Speicher für die Anweisung alloziert und dann eine Transaktion gestartet. Anschließend wird das SQL-Statement zum Server übertragen und vorbereitet. Mit *isc_dsql_describe_bind* werden die Parameter übertragen – in diesem Fall haben wir keine –, danach wird die Anweisung ausgeführt. Mit *isc_dsql_fetch* wird ein (in diesem Fall der erste) Datensatz geholt.

Wird nun gescrollt, dann werden weitere Aufrufe von *isc_dsql_fetch* abgesetzt, so lange, bis der Server meldet, dass keine weiteren Datensätze mehr vorhanden sind. Dies wird von der Komponente registriert, so dass weitere Aufrufe von *Next* keine Aktion mehr auslösen und damit sehr schnell ausgeführt werden.

Ohne persistente TField-Instanzen

Nun wollen wir die Sache ohne persistente TField-Instanzen wiederholen:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from PROJECT
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME,
    B.RDB$FIELD_NAME, B.RDB$FIELD_POSITION, '', 0, A.RDB$INDEX_TYPE,
    '', A.RDB$UNIQUE_FLAG, C.RDB$CONSTRAINT_NAME,
    C.RDB$CONSTRAINT_TYPE
  FROM RDB$INDICES A, RDB$INDEX_SEGMENTS B
    FULL OUTER JOIN RDB$RELATION_CONSTRAINTS C
      ON A.RDB$RELATION_NAME = C.RDB$RELATION_NAME
        AND C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY'
  WHERE (A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG IS NULL)
    AND (A.RDB$INDEX_NAME = B.RDB$INDEX_NAME)
    AND (A.RDB$RELATION_NAME = UPPER('PROJECT'))
  ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
```

```
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
```

Zunächst geht es genauso los wie mit den persistenten *TField*-Instanzen. Nach dem Aufruf von *isc_dsql_execute* wird jedoch eine weitere Abfrage ausgeführt, die den JOIN einiger Systemtabellen beinhaltet (die Formatierung hier stammt von mir ...). Mit diesem JOIN werden die Indexinformationen der Abfrage ermittelt. Diese würden benötigt, wenn an unserer Datenmenge über einen Provider eine *TClientDataSet*-Komponente hängen würde.

Den Verzicht, diese Indexinformationen zu ermitteln, könnte man auch dadurch bewirken, dass man *NoMetadata* auf *true* setzt.

BLOB-Felder

Die Tabelle *project* – aus diesem Grunde habe ich Sie hier verwendet – hat eine BLOB-Spalte namens *proj_desc*, die wir bislang nicht angezeigt haben. Dies wollen wir nun tun. Fügen Sie dafür eine *TDBMemo*-Instanz ein und setzen die Eigenschaften entsprechend. Für die Abfrage verwenden wir wieder das Joker-Zeichen.

Zunächst wollen wir eine Zeitmessung durchführen. Dazu legen wir die Zuweisung der Callback-Funktion vorübergehend still. Das Öffnen der Abfrage dauert nun 11 091 µs (statt 3 007 µs), das Scrollen 845 µs (statt 115 µs). Beim Öffnen und bei jedem Scrollen werden zusätzlich die folgenden Anweisungen ausgeführt:

```
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
```

```
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_open_blob
INTERBASE - isc_blob_info
INTERBASE - isc_get_segment
INTERBASE - isc_get_segment
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
```

Dadurch wird der Vorgang natürlich alles andere als beschleunigt.

Aufruf von First

Fügen Sie einen *TDBNavigator* hinzu, verbinden Sie diesen mit *DataSource1* und starten das Programm. Nach dem Öffnen der Datenmenge und einem Aufruf von *Next* rufen Sie (über den DBNavigator) *First* auf (den einen Aufruf von *Next* benötigen wir, damit der Button *First* überhaupt verfügbar geschalten wird). Hier erfolgen dann die folgenden DLL-Aufrufe:

```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

Die Transaktion wird also bestätigt, dann wird die Anweisung erneut aufgerufen. Der erneute Aufruf von *isc_dsql_execute* legt den Verdacht nahe, dass zwischenzeitlich eingefügte Datensätze dann auch angezeigt werden. Wir werden das gleich experimentell ermitteln und fügen dafür eine neue *TSQLQuery*-Instanz ein, der wir folgendes SQL-Statement zuweisen:

```
INSERT INTO project (proj_id, proj_name)
  VALUES (:id, :name)
```

Mit den folgenden Anweisungen wird ein Datensatz eingefügt:

```
with SQLQuery2 do
begin
  ParamByName('id').AsString := 'TEST';
  ParamByName('name').AsString := 'Dies ist ein TEST';
  ExecSQL;
end;
```

Nach einem Aufruf von *First* wird dieser Datensatz dann mit angezeigt. Im Gegensatz zu *TQuery* ist also ein erneutes Öffnen der Datenmenge nicht erforderlich – der Aufruf von *First* wird deutlich schneller ausgeführt als der von *Open*, weil die Anweisung nicht mehr interpretiert werden muss. (Dasselbe könnte man erreichen, wenn man die Abfrage explizit vorbereiten würde.)

Unidirektionale Datenmenge

Wenn wir schon einen DBNavigator auf dem Formular liegen haben, dann wollen wir auch mal den Aufruf von *Prior* und *Last* testen – allerdings mit mäßigem Erfolg:

Die Komponenten von dbExpress liefern eine unidirektionale Datenmenge, und diese heißt so, weil man nur in eine Richtung navigieren kann: nach hinten. Genau genommen ist noch nicht einmal der Aufruf von *First* möglich, die Entwickler von Borland haben diese Anweisung – wie wir gerade gesehen haben – aber so »hingebogen«, dass die Abfrage neu ausgeführt wird und somit der Cursor auf dem ersten Datensatz steht.

Auch der Versuch, an *DataSource1* ein *DBGrid* zu hängen, wird scheitern. Aufrufe von *Locate* oder *Filtered* ohnehin. Die Schnelligkeit und »Schlankheit« von dbExpress haben halt ihren Preis: geringen Komfort.

15.1.3 TSQLClientDataSet

Um die Datensätze in einem DBGrid anzeigen zu können, könnte man nun an *SQLDataSet1* einen *TDataSetProvider* und daran ein *TClientDataSet* hängen. Borland hat uns aber hier ein wenig Arbeit abgenommen und stellt uns die Komponente *TSQLClientDataSet* zur Verfügung.

Statt *SQLConnection* heißt die Verbindung zu *SQLConnection1* nun *DBConnection* – wir sind ja flexibel. Ansonsten setzen Sie *CommandText* wieder auf *select * from PROJECT*.

Das Öffnen dauert nun wesentlich länger (49 636 µs), das Scrollen auch (1 861 µs). Das Scrollen liegt am DBGrid, bei *TQuery* kämen Sie auf ähnliche Zeiten.

Beim Öffnen der Datenmenge gibt es zwei »Bremsen«:

- Es werden die Systemtabellen abgefragt – da helfen hier auch keine persistenten TField-Instanzen und *NoMetadata* gibt es hier nicht.
- Es werden alle Datensätze auf den Client übertragen, und zwar inklusive aller BLOBS, obwohl diese gar nicht angezeigt werden. (Da hilft es auch nichts, die Spalte aus der Spaltenliste von *DBGrid1* und/oder aus der Liste der persistenten TField-Instanzen zu löschen).

Für das erste Problem habe ich noch keine Lösung gefunden, gegen das zweite kann man jedoch etwas tun:

- Um das Übertragen der Memos zu vermeiden, ändert man entweder das SQL-Statement entsprechend, oder man setzt die Option *poFetchBlobsOnDemand* – Letzteres hat den Vorteil, dass man sie problemlos nachladen kann.
- Mit der Eigenschaft *PacketRecords* kann man einstellen, wie viele Datensätze jeweils übertragen werden. Per Voreinstellung steht diese Eigenschaft auf *-1*, somit werden alle Datensätze übertragen. Würde man *PacketRecords* auf 3 setzen, dann würden jeweils nur drei Datensätze übertragen: Beim Öffnen der Datenmenge die ersten drei, und wenn auf einen weiteren Datensatz gescrollt wird, die nächsten drei. Gerade bei großen Datenmengen macht es viel Sinn, *PacketRecords* auf einen »handlichen« Wert zu setzen.

Datensätze ändern

TSQLClientDataSet hat nicht nur den Vorteil, ein DBGrid verwenden zu können, wir können auch Datensätze einfügen, ändern und löschen. Die folgenden Ausführungen beziehen sich auf alle drei Möglichkeiten der Bearbeitung, deswegen wollen wir uns hier nur den Update-Fall ansehen.

Damit die Änderungen auf den Server übertragen werden, verwenden wir eine *AfterPost*-Ereignisbehandlungsroutine:

```
procedure TForm1.SQLClientDataSet1AfterPost(DataSet: TDataSet);
begin
  SQLClientDataSet1.ApplyUpdates(0);
end;
```

Wir ändern nun einen Datensatz in der Spalte *proj_name*. Mit dem Aufruf von *ApplyUpdates* werden die folgenden Routinen der Client-DLL aufgerufen:

```
INTERBASE - isc_start_transaction
INTERBASE - isc_dsql_allocate_statement
```

```
update PROJECT  set
 PROJ_NAME = ?
where
 PROJ_ID = ? and
 PROJ_NAME = ? and
 TEAM_LEADER is null and
 PRODUCT = ?

INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_transaction
```

Zunächst wird eine Transaktion gestartet und Speicher für die Anweisung alloziert. Die UPDATE-Anweisung beinhaltet das, was verwendet wird, wenn die Spalte *proj_name* geändert wird und *UpdateMode* den Wert *upWhereAll* hat. Anschließend wird das Statement vorbereitet.

Mit *isc_dsql_sql_info* können Informationen über das Statement, insbesondere über Art und Anzahl der Parameter, erhalten werden. InterBase verwendet für Integer-Zahlen ein anderes Format als Windows, somit muss hier mit *isc_vax_integer* eine Umwandlung erfolgen. Mit *isc_dsql_describe_bind* werden Parameterinformationen besorgt, anschließend wird das Statement ausgeführt, der Speicher freigegeben und die Transaktion bestätigt.

So weit, so schön. Nun steht man bei Client-Server-Datenbanksystemen häufig vor dem Problem, dass man eine Datenmenge aktualisieren möchte, die aus einem JOIN resultiert. Dies sollte eigentlich dank der Eigenschaften *UpdateMode* und der *TField*-Eigenschaft *ProviderFlags* kein Problem sein. Stellen wir also *UpdateMode* auf *upWhereKeyOnly* und schauen zu, was passiert (es interessiert hier nur das SQL-Statement):

```
update PROJECT  set
  PROJ_NAME = ?
where
  PROJ_NAME = ? and
  PRODUCT = ?
```

Zunächst sind wir etwas verwundert, weil normalerweise die Spalte *id* (in diesem Fall *proj_id*) den Primärschlüssel bildet.

Sicherheitshalber schauen wir uns die Definition von *project* an, und es ist tatsächlich so, wie wir vermuten:

```
CREATE TABLE project
  (proj_id projno NOT NULL,
  proj_name VARCHAR(20) NOT NULL,
  proj_desc BLOB SUB_TYPE TEXT SEGMENT SIZE 800,
  team_leader empno,
  product prodtype,
  UNIQUE (proj_name),
  PRIMARY KEY (proj_id));
```

Obwohl *TSQLClientDataSet* beim Öffnen Indexinformationen aus den Systemtabellen liest, erkennt sie nicht den korrekten Schlüssel. Nun gut, gibt es noch die Eigenschaft *ProviderFlags*, wo wir es mit *pfInWhere* und *pfInKey* versuchen wollen. Allerdings – Auswirkungen: keine.

Die Komponente *TSQLClientDataSet* ist nicht von *TSQLDataSet*, sondern von *TClientDataSet* (genauer *TCustomClientDataSet*) abgeleitet. Erstellen wir persistente TField-Instanzen, so gehören diese zum ClientDataSet und nicht zum SQLDataSet. Aus diesem Grunde werden nicht nur die Indexinformationen abgefragt, wenn wir die Datenmenge öffnen, wir können auch nicht die Eigenschaft *ProviderFlags* setzen, die den Provider interessieren würde. Fazit: Da muss Borland noch mal nachbessern!

Einzelne Komponenten

In vielen Fällen werden Sie sich die Komponenten *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* einzeln auf das Formular (oder das Datenmodul) legen müssen. Dann legen Sie bei *TSQLDataSet* persistente TField-Instanzen an und setzen dort entsprechend die Eigenschaft *ProviderFlags*. Und siehe da, das Statement ist so, wie wir es brauchen:

```
update PROJECT  set
  PROJ_NAME = ?
where
  PROJ_ID = ?
```

Schließen des Handles

Am Rande: Wenn Sie mit *TSQLClientDataSet* oder mit *TDataSetProvider* alle Datensätze abrufen, wird hinterher die Transaktion beendet und das Handle freigegeben:

```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

15.1.4 Stored Procedures

Zum Zugriff auf STORED PROCUEDURES hat man in der Regel vier Möglichkeiten:

- Mit *TSQLDataSet* über eine SQL-Anweisung.
- Man kann die *TSQLDataSet*-Eigenschaft *CommandType* auf *ctStoredProc* setzen und dann eine Prozedur auswählen.
- Man kann *TSQLQuery* verwenden.
- Und schließlich gibt es noch *TSQLStoredProc*.

Wir wollen diese vier Möglichkeiten beim Zugriff auf die folgende STORED PROCEDURE vergleichen, die dem Beispielprojekt To-Do-Liste entnommen wurde:

```
ALTER PROCEDURE P_BENUTZER_ID
RETURNS
  (GEN INTEGER)
AS
BEGIN
  gen=GEN_ID(g_benutzer, 1);
  SUSPEND;
END^
```

Bei allen vier Komponenten wurde die Eigenschaft *NoMetadata* auf *true* gesetzt.

TSQLDataSet und SQL-Anweisung

Zunächst verwenden wir *TSQLDataSet* und die folgende SQL-Anweisung:

```
SELECT *
  FROM p_benutzer_id
```

Es werden hier die folgenden Anweisungen verwendet:

```
SQLDataSet2.Open;
s := SQLDataSet2.FieldByName('GEN').AsString;
SQLDataSet2.Close;
```

Dabei werden folgende Client-Aufrufe registriert:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT *
  FROM p_benutzer_id
```

```
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

Also ganz das, was wir von anderen Abfragen her kennen.

TSQLDataSet und ctStoredProc

Bei der nächsten *TSQLDataSet*-Instanz setzen wir *CommandType* auf *ctStoredProc* und *CommandText* auf *p_benutzer_id*. Statt *Open* müssen wir hier *ExecSQL* verwenden, den Wert erhalten wir über die Eigenschaft *Params*.

```
SQLDataSet3.ExecSQL;
s := SQLDataSet3.Params[0].AsString;
```

Obwohl wir die Eigenschaft *NoMetadata* auf *true* gesetzt haben, erhalten wir das folgende Protokoll:

```
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
SELECT 0, '', '', '', A.RDB$PARAMETER_NAME, A.RDB$PARAMETER_NUMBER,
    A.RDB$PARAMETER_TYPE, B.RDB$FIELD_TYPE, B.RDB$FIELD_SUB_TYPE,
    B.RDB$FIELD_SUB_TYPE, B.RDB$FIELD_LENGTH, 0, B.RDB$FIELD_SCALE,
    B.RDB$NULL_FLAG, A.RDB$SYSTEM_FLAG
  FROM RDB$PROCEDURE_PARAMETERS A, RDB$FIELDS B
  WHERE (A.RDB$FIELD_SOURCE = B.RDB$FIELD_NAME)
    AND (A.RDB$PROCEDURE_NAME = 'P_BENUTZER_ID')
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
EXECUTE PROCEDURE P_BENUTZER_ID
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_sql_info
```

```
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_execute2
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

TSQLQuery und TSQLStoredProc

Bei *TSQLQuery* erhält man dasselbe Protokoll wie bei einer SQL-Anweisung von *TSQLDataSet*. *TSQLStoredProc* gleicht *TSQLDataSet*, wenn *ctStoredProc* verwendet wird. Zur Abwechslung wollen wir die Möglichkeit nutzen, den Parameter über den Namen zu erhalten:

```
s := SQLStoredProc1.ParamByName('GEN').AsString;
```

Fazit

Wenn möglich, sollten Sie auf STORED PORCEDURES per SQL-Statement zugreifen, sei es mit *TSQLDataSet*, sei es mit *TQuery*.

15.2 Referenz dbExpress

Im Folgenden sollen die Eigenschaften, Methoden und Ereignisse der *dbExpress*-Komponenten beschrieben werden. Die Komponenten *TSQLDataSet*, *TSQLQuery*, *TSQLTable* und *TSQLStoredProc* sind zwar alle von *TDataSet* abgeleitet, implementieren aber nur einen geringen Teil der *TDataSet*-Elemente. Manche von dort geerbten Elemente bleiben wirkungslos, andere lösen bei ihrer Verwendung eine Exception aus.

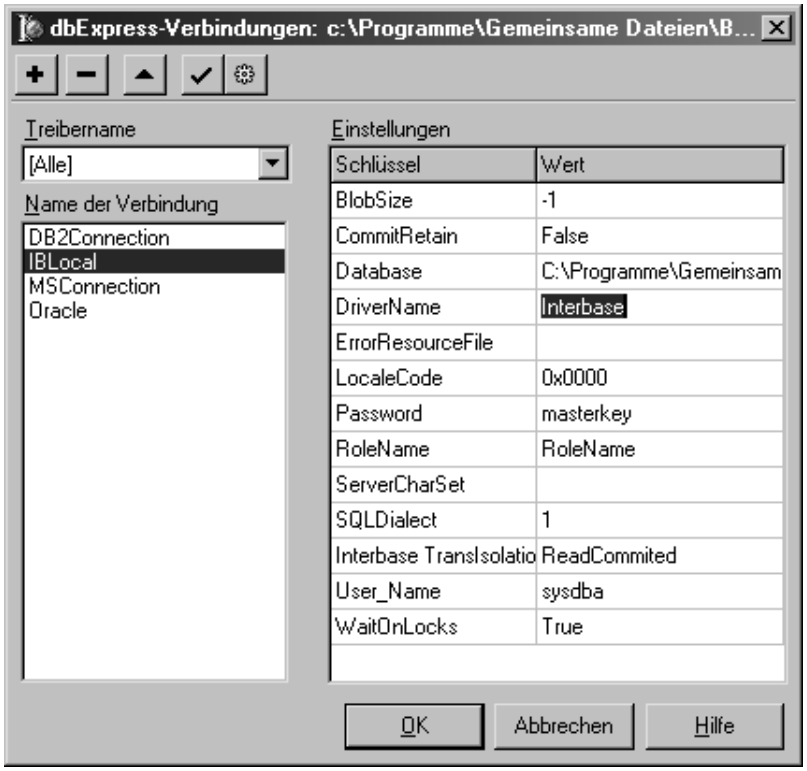
Da alle vier Komponenten von *TCustomSQLDataSet* abgeleitet sind, wollen wir bei der Beschreibung von *TCustomSQLDataSet* auch alle Elemente von *TDataSet* besprechen, die sich hier (sinnvoll) verwenden lassen.

15.2.1 TSQLConnection

Mit der Komponente *TSQLConnection* wird die Verbindung zur Client-DLL des Datenbanksystems hergestellt. Des Weiteren wird hier die Transaktionssteuerung vorgenommen.

Verbindung zur Datenbank

Um die Verbindung zur Datenbank herzustellen, verwendet man am einfachsten den Komponenteneditor von *TSQLConnection*:



Die Einstellungen, die Sie hier vornehmen müssen, hängen stark vom verwendeten Datenbanksystem ab. Bei InterBase beispielsweise benötigen wir unter *Database* den Namen der **.gdb*-Datei. Wenn auf den Login-Dialog verzichtet werden soll, dann müssen auch noch *User_Name* und *Password* gesetzt werden.

- ConnectionName (Eigenschaft, veröffentlicht)

  ```
  property ConnectionName: string;
  ```

  Mit *ConnectionName* wird der Name der Verbindung angegeben. Wenn Sie die Eigenschaften einer Verbindung ändern oder eine neue Verbindung anlegen, dann werden deren Parameter in der Datei *Dbxconnections.ini* gespeichert. Möchten Sie in anderen Anwendungen nun dieselben Parameter verwenden, dann müssen Sie lediglich *ConnectionName* entsprechend wählen.

- Connected (Eigenschaft, veröffentlicht)

  ```
  property Connected: Boolean;
  ```

  Wird *Connected* auf *true* gesetzt, dann wird eine Verbindung zur Datenbank hergestellt.

- Params (Eigenschaft, veröffentlicht)

  ```
  property Params: TStrings;
  ```

  Beim Setzen von *ConnectionName* werden die Parameter aus der Datei *Dbxconnections.ini* nach *Params* kopiert, die Verbindung arbeitet dann mit dem Inhalt von *Params*.

- LoadParamsOnConnect (Eigenschaft, veröffentlicht),
  LoadParamsFromIniFile (Methode)

  ```
  property LoadParamsOnConnect: Boolean default false;
  procedure LoadParamsFromIniFile(AFileName: String = '' );
  ```

  Per Voreinstellung ist der Inhalt von *Params* für die Datenbankverbindung relevant, nicht der von *Dbxconnections.ini*. Sollen aber beim Herstellen der Verbindung – gewöhnlich also beim Programmstart – die Parameter aus *Dbxconnections.ini* geladen werden, dann ist *LoadParamsOnConnect* auf *true* zu setzen. Alternativ kann man auch manuell *LoadParamsFromIniFile* aufrufen.

- DriverName, VendorLib, LibraryName (Eigenschaften, veröffentlicht)

  ```
  property DriverName: string;
  property LibraryName: string;
  property VendorLib: string;
  ```

  Beim Setzen von *ConnectionName* werden für diese drei Eigenschaften passende Werte gesetzt, die im Normalfall nicht abgeändert werden brauchen. *DriverName* ist der Name des Datenbanktreibers, also beispielsweise *InterBase*. *LibraryName* ist der Name der DLL, mit der *dbExpress* gerade arbeitet. Im Falle von InterBase wäre dies *dbexpint.dll*. *VendorLib* ist der Name des Datenbank-Clients, im Falle von InterBase also *GDS32.DLL*.

- KeepConnection (Eigenschaft, veröffentlicht)

  ```
  property KeepConnection: Boolean default true;
  ```

  Hat *KeepConnection* den Wert *false*, dann wird die Verbindung zur Datenbank abgebaut, wenn die letzte offene Datenmenge geschlossen wird.

- AfterConnect, AfterDisconnect, BeforeConnect, BeforeDisconnect

  ```
  property AfterConnect(Sender: TObject);
  property AfterDisconnect(Sender: TObject);
  property BeforeConnect(Sender: TObject);
  property BeforeDisconnect(Sender: TObject);
  ```

  Diese Ereignisse treten vor beziehungsweise nach dem Erstellen oder Trennen der Datenbankverbindung auf.

Login

- LoginPrompt (Eigenschaft, veröffentlicht)

```
property LoginPrompt: Boolean default true;
```

Um zu vermeiden, dass sich der Anwender beim Server anmelden muss, setzt man *LoginPrompt* auf *false*. Der Benutzername und das Passwort müssen dann bei den Parametern angegeben sein.

- OnLogin (Ereignis)

```
property OnLogin(Database: TSQLConnection; LoginParams: TStrings)
```

Hat *LoginPrompt* den Wert *true*, dann wird das Ereignis *OnLogin* aufgerufen, bevor der Anmelde-Dialog angezeigt wird. Hier können Sie in *LoginParams* Benutzernamen und Passwort setzen, um die Anzeige des Login-Dialogs zu vermeiden.

Transaktionssteuerung

- StartTransaction, Commit, Rollback (Methoden)

```
procedure StartTransaction(TransDesc: TTransactionDesc);
procedure Commit(TransDesc: TTransactionDesc);
procedure Rollback(TransDesc: TTransactionDesc);
```

Mit *StartTransaction* wird eine Transaktion gestartet, mit *Commit* bestätigt, mit *Rollback* zurückgenommen. Allen drei Methoden muss ein Parameter vom Typ *TTransactionDesc* übergeben werden:

```
TTransactionDesc = packed record
  TransactionID: LongWord;
  GlobalID: LongWord;
  IsolationLevel: (xilDIRTYREAD, xilREADCOMMITTED,
    xilREPEATABLEREAD, xilCUSTOM);
  CustomIsolation: LongWord;
end;
```

Dabei muss *TransactionID* auf einen beliebigen, aber eindeutigen und innerhalb der Transaktion einheitlichen Wert sowie *IsolationLevel* auf einen der ersten drei Werte gesetzt werden.

Wird nicht explizit eine Transaktion gestartet, so geschieht dies automatisch. Diese wird dann jedoch nach jeder »Anweisung« beendet.

- InTransaction, TransactionsSupported (Eigenschaften, öffentlich, nur Lesen)

```
property InTransaction: Boolean;
property TransactionsSupported: LongBool;
```

Mit *InTransaction* kann man prüfen, ob derzeit eine Transaktion gestartet ist. Mit *TransactionsSupported* wird geprüft, ob das Datenbanksystem Transaktionen unterstützt – bei MySQL ist dies derzeit nicht der Fall.

Informationen über die Datenbank

- GetTableNames, GetFieldNames, GetIndexNames (Methoden)

```
procedure GetTableNames(List: TStrings;
  SystemTables: Boolean = False);
procedure GetFieldNames(const TableName: string; List: TStrings);
procedure GetIndexNames(const TableName: string; List: TStrings);
```

Mit *GetTableNames* kann man die Liste aller Tabellen in die String-Liste *List* schreiben. Normalerweise werden diejenigen Tabellen nach *List* geschrieben, die den Kriterien von *TableScope* genügen. Setzt man *SystemTables* auf *true*, dann liefert *GetTableNames* ausschließlich Systemtabellen.

Mit *GetFieldNames* erhält man die Liste aller Spalten in der angegebenen Tabelle, mit *GetIndexNames* die Liste aller Indizes.

- TableScope (Eigenschaft, veröffentlicht)

```
property TableScope: set of (tsSynonym, tsSysTable, tsTable, tsView)
  default [tsTable,tsView];
```

Mit *TableScope* spezifiziert man, welche Tabellen von *GetTableNames* ermittelt werden sollen.

- GetProcedureNames, GetProcedureParams (Methoden)

```
procedure GetProcedureNames(List: TStrings);
procedure GetProcedureParams(ProcedureName: String; List: TList);
```

Mit *GetProcedureNames* erhält man eine Liste aller STORED PROCEDURES, mit *GetProcedureParams* die Liste aller Parameter in der angegebenen Prozedur.

- SetTraceCallbackEvent (Methode),
  TraceCallbackEvent (Eigenschaft, öffentlich, nur Lesen)

```
procedure SetTraceCallbackEvent(Event: TSQLCallbackEvent;
  IClientInfo: Integer);
property TraceCallbackEvent: TSQLCallbackEvent;
```

Mit *TraceCallbackEvent* kann man die Kommunikation zum Datenbank-Server »belauschen«. Gehen Sie hier wie folgt vor:

```
function TraceCallback(CallType: TRACECat;
  CBInfo: Pointer): CBRType; stdcall;
var
  s: string;
  rec: pSQLTRACEDesc;
begin
  rec := CBInfo;
  s := rec.pszTrace;
  Form1.Memo1.Lines.Add(s);
  result := cbrUSEDEF;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  SQLConnection1.SetTraceCallbackEvent(TraceCallback, 1);
  SQLConnection1.Connected := true;
end;
```

Verbundene Datenmengen

- DataSets, DataSetCount (Eigenschaften, öffentlich, nur Lesen)

```
property DataSets[Index: Integer]: TCustomSQLDataSet;
property DataSetCount: Integer;
```

Über die Array-Eigenschaft *DataSets* erhält man Referenzen auf alle verbundenen Datenmengen. Deren Anzahl ermittelt man mit *DataSetCount*.

- CloseDataSets (Methode)

```
procedure CloseDataSets;
```

Mit *CloseDataSets* schließt man alle verbundenen Datenmengen.

Sonstiges

- SQLHourGlass (Eigenschaft, öffentlich)

```
property SQLHourGlass: Boolean default true;
```

Den Sanduhr-Cursor beim Ausführen von SQL-Anweisungen kann man abschalten, indem man *SQLHourGlass* auf *false* setzt.

- Execute, ExecuteDirect (Methoden)

```
function Execute(const SQL: string; Params: TParams;
  ResultSet: Pointer = nil): Integer;
function ExecuteDirect(const SQL: string): LongWord;
```

Mit *Execute* kann die übergebene SQL-Anweisung ausgeführt werden. Parameter werden mit *Params* übergeben. Liefert die Anweisung eine Ergebnismenge, dann wird dafür eine *TCustomSQLDataSet*-Instanz erzeugt und der Zeiger darauf *ResultSet* zugewiesen.

Mit *ExecuteDirect* kann man eine Anweisung ohne Parameter ausführen. Sollte diese eine Ergebnismenge liefern, wird diese verworfen.

Für die Ausführung von SQL-Anweisungen kann man auch *TSQLDataSet* und *TSQLQuery* verwenden.

15.2.2 TCustomSQLDataSet

Die Komponente TCustomSQLDataSet ist der gemeinsame Vorfahre von *TSQLDataSet*, *TSQLQuery*, *TSQLTable* und *TSQLStoredProc*. Da diese vier Komponenten die Elemente von *TDataSet* nur sehr unvollständig implementieren, wollen wir hier diejenigen Elemente von *TDataSet* besprechen, die verwendet werden können.

Verbindung zur Datenbank

- SQLConnection (Eigenschaft, veröffentlicht)

```
property SQLConnection: TSQLConnection;
```

Mit der Komponenteneigenschaft *SQLConnection* wird die Verbindungskomponente gewählt, über die eine Verbindung zur Datenbank hergestellt wird.

- TransactionLevel (Eigenschaft, öffentlich)

```
property TransactionLevel: SmallInt default 0;
```

Mit *TransactionLevel* wird spezifiziert, im Kontext welcher Transaktion das Öffnen der Datenmenge erfolgt. Dieser Wert muss mit demjenigen übereinstimmen, der bei *StartTransaction* verwendet wurde. Hat *TransactionLevel* den Vorgabewert Null, dann wird die zuletzt gestartete Transaktion verwendet.

Vorsicht: Nicht alle Datenbanksysteme unterstützen das Öffnen mehrerer Transaktionen.

- NoMetadata (Eigenschaft, veröffentlicht)

```
property NoMetadata: Boolean default false;
```

Wird *NoMetadata* auf *true* gesetzt, dann werden keine Indexinformationen vom Server abgerufen, was das Öffnen der Datenmenge beschleunigt. In manchen Fällen gibt es aber dann Probleme, wenn über einen Provider Datensätze eingetragen werden sollen.

Status der Datenmenge

- Active (Ereignis, veröffentlicht)

```
property Active: boolean;
```

Mit der Eigenschaft *Active* kann die Datenmenge geöffnet und geschlossen werden.

- Open, Close (Methoden, TDataSet)

```
procedure Open;
procedure Close;
```

Mit *Open* wird die Datenmenge geöffnet, mit *Close* wird sie geschlossen.

- AfterClose, AfterOpen, BeforeClose, BeforeOpen (Ereignisse)

```
property AfterClose(DataSet: TDataSet)
property AfterOpen(DataSet: TDataSet)
property BeforeClose(DataSet: TDataSet)
property BeforeOpen(DataSet: TDataSet)
```

Diese Ereignisse treten vor beziehungsweise nach dem Schließen beziehungsweise Öffnen der Datenmenge auf.

- State (Eigenschaft, öffentlich, nur Lesen)

```
property State: TDataSetState;
```

Die Eigenschaft *State* zeigt den Status der Datenmenge an und kann folgende Werte annehmen:

  - *dsInactive*: Die Datenmenge ist nicht aktiv, auf ihre Daten kann nicht zugegriffen werden.
  - *dsBrowse*: Die Daten können angezeigt, jedoch nicht geändert werden. Solange keine anderen Operationen durchgeführt werden, hat eine Datenmenge diesen Status.
  - *dsCalcFields*: Ein *OnCalcFields*-Ereignis wurde ausgelöst, es werden also gerade Felder berechnet.

- DisableControls, EnableControls, ControlsDisabled (Methoden)

```
procedure DisableControls;
procedure EnableControls;
function ControlsDisabled: Boolean;
```

Werden die Daten einer Datenmenge geändert, so wird im Regelfall laufend die Anzeige aktualisiert. Werden viele Daten in einer Schleife eingefügt, verlangsamt dies den Vorgang deutlich. Mit *DisableControls* kann deshalb diese Aktualisierung abgeschaltet und mit *EnableControls* anschließend wieder angeschaltet werden.

Mit *ControlsDisabled* kann ermittelt werden, ob die Aktualisierung abgeschaltet ist.

Navigieren in der Datenmenge

- First, Next (Methoden)

```
procedure First;
procedure Next;
```

Eigentlich unterstützen unidirektionale Datenmengen nur den Aufruf von *Next*. Ruft man *First* auf, dann wird die Abfrage neu ausgeführt.

- BOF, EOF (Eigenschaften, öffentlich, nur Lesen)

```
property BOF: Boolean;
property EOF: Boolean;
```

Die Eigenschaft *BOF* ist dann gleich *true*, wenn die Methode *First* aufgerufen wurde. Schlägt das Verschieben des Datenzeigers fehl, weil er schon auf dem letzten Datensatz steht, wird *EOF* auf *true* gesetzt.

Diese beiden Eigenschaften werden vor allem für die Abbruchbedingungen von Schleifen benötigt.

```
while not Table1.EOF do
begin
    ...
  Table1.Next;
end;
```

- MoveBy (Methode)

```
function MoveBy(Distance: Integer): Integer;
```

Mit der Methode *MoveBy* kann man um die im Parameter angegebene Zahl von Datensätzen nach hinten navigieren. Bei unidirektionalen Datenmengen funktioniert *MoveBy* nur mit positiven *Distance*-Werten.

- AfterScroll, BeforeScroll (Ereignisse)

```
property AfterScroll(DataSet: TDataSet);
property BeforeScroll(DataSet: TDataSet);
```

Diese Ereignisse treten vor beziehungsweise nach dem Verschieben des Datenzeigers auf.

Zugriff auf die Daten

- Fields (Eigenschaft, öffentlich)

```
property Fields[Index: Integer]: TField ;
```

Mit der Array-Eigenschaft *Fields* kann auf jedes Feld der Datenmenge zugegriffen werden.

Bei einem Verändern der Tabellenstruktur kann sich jedoch die Reihenfolge der Felder verändern. Verwenden Sie deshalb besser *FieldByName*.

Das Objekt *TField* wird in Kapitel 3.2 besprochen.

- FieldByName, FindField (Methoden)

```
function FieldByName(const FieldName: string): TField;
function FindField(const FieldName: string): TField;
```

Die Methode *FieldByName* greift über den Feldnamen auf das Feld zu – solange das betreffende Feld noch vorhanden ist, kommt es bei einer Änderung der Tabellenstruktur zu keinen Schwierigkeiten. Ist der als Parameter übergebene Spaltenname nicht vorhanden, dann löst *FieldByName* eine Exception aus, während *FindField* den Wert *nil* zurückgibt.

```
Table1.FieldByName('Vorname').AsString := 'Patty';
```

Das Objekt *TField* wird in Kapitel 3.2 besprochen.

- FieldValues (Eigenschaft, öffentlich)

```
property FieldValues[const FieldName: string]: Variant;
```

Mit besonders wenig Schreibaufwand kann mittels der Array-Eigenschaft *FieldValues* auf die Feldinhalte zugegriffen werden: Zum einen hat diese Eigenschaft den Datentyp *Variant*, so dass ohne weitere Konvertierung die Werte zugewiesen werden können. Zum anderen ist diese Eigenschaft die Default-Eigenschaft von *TDataSet*, sie muss also gar nicht erwähnt werden. Um den Inhalt des Feldes *Vorname* der Variablen *s* zuzuweisen, formuliert man einfach:

```
s := SQLDataSet1['Vorname'];
```

- DefaultFields (Eigenschaft, öffentlich, nur Lesen)

  ```
  property DefaultFields Boolean;
  ```

  Mit *DefaultFields* kann ermittelt werden, ob *TField*-Instanzen beim Öffnen der Datenmenge angelegt werden müssen (*true*), oder ob es persistente *TField*-Instanzen gibt (*false*).

- CanModify (Eigenschaften, öffentlich, nur Lesen)

  ```
  property CanModify: Boolean;
  ```

  Mit Hilfe der Eigenschaft *CanModify* kann festgestellt werden, ob die Datenmenge geändert werden kann. Sie gibt bei unidirektionalen Datenmengen immer *false* zurück.

- FieldCount, RecordCount (Eigenschaften, öffentlich, nur Lesen)

  ```
  property FieldCount: Integer;
  property RecordCount: Integer;
  ```

  Mit *FieldCount* kann die Zahl der Spalten (*FieldCount*) ermittelt werden. *RecordCount* zählt die Datensätze und holt dabei alle noch nicht abgerufenen Datensätze vom Server.

- IsEmpty (Methode)

  ```
  function IsEmpty: Boolean;
  ```

  IsEmpty gibt *true* zurück, wenn die Datenmenge leer ist.

- GetFieldNames (Methode, TDataSet)

  ```
  procedure GetFieldNames(List: TStrings);
  ```

  Die Methode *GetFieldNames* ermittelt die Spaltennamen einer Tabelle und schreibt sie in die vorgegebene String-Liste.

  ```
  Table1.GetFieldNames(ListBox1.Items);
  ```

- OnCalcFields (Ereignis)

  ```
  property OnCalcFields(DataSet: TDataSet)
  ```

  Um einem berechneten Feld einen Wert zuzuweisen, wird das Ereignis *OnCalcFields* verwendet.

Aktualisieren der Datenmenge

- Refresh (Methode), BeforeRefresh, AfterRefresh (Ereignisse)

```
procedure Refresh;
property BeforeRefresh(DataSet: TDataSet);
property AfterRefresh(DataSet: TDataSet);
```

Mit *Refresh* aktualisieren Sie die Datenmenge. Davor wird das Ereignis *BeforeRefresh* aufgerufen, anschließend das Ereignis *AfterRefresh*. Bei den unidirektionalen Datenmengen sorgt der Aufruf von *Refresh* dafür, dass die Datenmenge geschlossen und neu geöffnet wird:

```
procedure TCustomSQLDataSet.InternalRefresh;
begin
  SetState(dsInactive);
  CloseCursor;
  OpenCursor(False);
end;
```

- IsUnidirectional (Eigenschaft, öffentlich, nur Lesen)

```
property IsUnidirectional: Boolean;
```

IsUnidirectional gibt *true* zurück, wenn es sich um eine unidirektionale Datenmenge handelt.

Parameter

Die folgenden Elemente sind in *TSQLTable* nicht veröffentlicht:

- Params (Eigenschaft, veröffentlicht), ParamByName (Methode)

```
property Params: TParams;
function ParamByName(const Value: string): TParam;
```

Mit Hilfe von *Params* und *ParamByName* kann man auf die Parameter zugreifen – entweder über deren Index oder über deren Namen.

- DataSource (Eigenschaft, veröffentlicht)

```
property DataSource: TDataSource;
```

Über die Eigenschaft *DataSource* können Master-Detail-Verbindungen hergestellt werden. Aus der angegebenen Datenquelle bezieht die Abfrage diejenigen Parameter, die ihr nicht explizit zugewiesen werden. Des Weiteren wird nach dem Scrollen der mit *DataSource* verbundenen Datenmenge die Client-Datenmenge mit aktualisierten Parametern neu ausgeführt.

- Prepared (Eigenschaft, öffentlich)

  ```
  property Prepared: Boolean;
  ```

  Vor der Ausführung einer SQL-Anweisung muss diese auf dem Server interpretiert werden. Wird dieselbe Anweisung mehrmals hintereinander ausgeführt, dann braucht sie nur ein einziges Mal interpretiert werden, selbst dann, wenn sich die Werte der Parameter geändert haben.

  Um zu verhindern, dass die Anweisung in einem solchen Fall nochmals interpretiert wird, setzt man *Prepared* auf true.

15.2.3 TSQLDataSet

Die Komponente *TSQLDataSet* ist der »vorgesehene« Weg, dbExpress einzusetzen. Die anderen Datenmengenkomponenten (*TSQLQuery*, *TSQLTable* und *TSQLStoredProc* dienen lediglich dazu, dem Programmierer die Umstellung zu erleichtern.)

- CommandType, CommandText (Eigenschaften, veröffentlicht)

  ```
  property CommandType: (ctQuery, ctTable, ctStoredProc);
  property CommandText: string;
  ```

  Mit *CommandType* wird eingestellt, wie *CommandText* zu interpretieren ist:

  - Hat *CommandType* den Wert *ctQuery*, dann wird mit *CommandText* eine SQL-Anweisung eingegeben. Handelt es sich um ein SELECT-Statement, kann anschließend die Datenmenge mit *Open* geöffnet werden, anderenfalls kann die Anweisung mit *ExecSQL* ausgeführt werden.

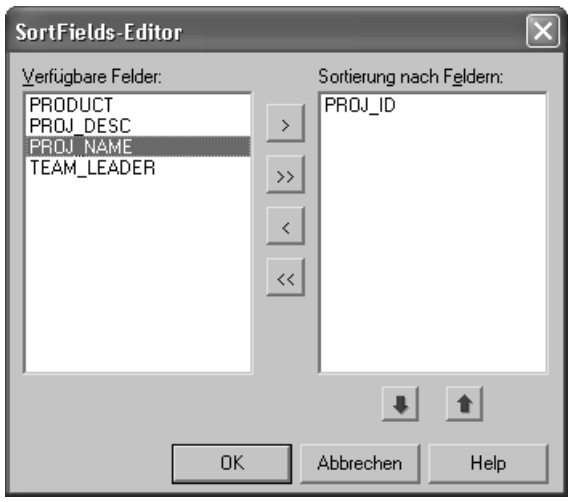
    Für *CommandText* steht dann ein Eigenschaftseditor zur Verfügung, der das Erstellen von Abfragen erleichtert:

- Hat *CommandType* den Wert *ctTable*, dann wird mit *CommandText* ein Tabellenname ausgewählt; der Objektinspektor stellt dafür eine Nachschlageliste zur Verfügung. Die Komponenten generiert automatisch eine SQL-Abfrage, welche alle Spalten der betreffenden Tabelle holt (SELECT * FROM tabelle).
- Bei *ctStoredProc* stellt der Objektinspektor eine Liste aller STORED PROCEDURES zur Verfügung.

- SortFieldNames (Eigenschaft, veröffentlicht)

```
property SortFieldNames: string;
```

Mit dem Eigenschaftseditor kann eine Spaltenliste erstellt werden, aus der – wenn *CommandType* den Wert *ctTable* hat – dann eine ORDER-Klausel gebildet wird.



- ExecSQL (Methode)

```
function ExecSQL(ExecDirect: Boolean = False): Integer;
```

Mit *ExecSQL* kann eine SQL-Anweisung oder eine STORED PROCEDURE ausgeführt werden. Da mit *ExecSQL* keine Datenmenge geöffnet wird, eignet sie sich nicht für SELECT-Statements und Prozeduren, die mehrere Datensätze zurückgeben.

Um die Anweisung ohne Vorbereitung direkt auszuführen, setzen Sie *ExecDirect* auf *true*. Der Rückgabewert der Funktion beinhaltet die Anzahl der bearbeiteten Datensätze.

15.2.4 TSQLQuery

Die Komponente *TSQLQuery* tut nichts anderes als *TSQLDataSet*, wenn *CommandType* auf *ctQuery* gesetzt ist. Die SQL-Anweisung wird jedoch nicht der Eigenschaft *CommandText*, sondern der Eigenschaft *SQL* zugewiesen.

15.2.5 TSQLTable

Die Komponente *TSQLTable* dient dem Zugriff auf eine einzelne Tabelle. Die Verwendung ist an *TTable* (Kapitel 5) angelehnt.

- TableNames (Eigenschaft, veröffentlicht)

  ```
  property TableName: string;
  ```

  Mit *TableName* wird der Name der Tabelle spezifiziert, für die eine Abfrage erstellt werden soll.

- MasterSource, MasterFields (Eigenschaften, veröffentlicht)

  ```
  property MasterSource: TDataSource;
  property MasterFields: string;
  ```

  Zum Aufbau einer Master-Detail-Verknüpfung setzt man von der Detailtabelle *MasterSource* auf die Datenquelle der Master-Datenmenge und erstellt mit dem Eigenschaftseditor von *MasterFields* eine Liste der Spalten, über welche die Verknüpfung hergestellt werden soll.

- IndexFieldNames (Eigenschaft, veröffentlicht)

  ```
  property IndexFieldNames: string;
  ```

  Aus den hier aufgeführten Spalten wird eine ORDER-Klausel generiert. Mehrere Spalten sind durch Semikola zu trennen.

- IndexName (Eigenschaft, veröffentlicht)

  ```
  property IndexName: string;
  ```

  IndexName sollten Sie lieber nicht verwenden: Die Komponente muss hier nämlich beim Datenbanksystem nachfragen, aus welchen Spalten sich der betreffende Index zusammensetzt, und generiert daraus dann die ORDER-Klausel. Setzen Sie lieber die Eigenschaft *IndexFieldNames*.

- DeleteRecords (Methode)

  ```
  procedure DeleteRecords;
  ```

  Mit *DeleteRecords* entfernt man alle Datensätze aus der aktuellen Tabelle.

15.2.6 TSQLStoredProc

Mit *TSQLStoredProc* können Sie STORED PROCEDURES ausführen. Die Verwendung ist an die BDE-Komponente *TStoredProc* angelehnt: Der Prozeduren-Name wird mit *StoredProcName* angegeben, die Prozedur mit *ExecProc* ausgeführt.

Sollte die Prozedur mehrere Ergebnismengen zurückgeben, dann schauen Sie sich in der Online-Hilfe die Methode *NextRecordSet* an – dort gibt es dann auch ein Beispiel.

15.2.7 TSQLClientDataSet

Die Komponente *TSQLClientDataSet* ist wie *TClientDataSet* von der Komponente *TCustomClientDataSet* abgeleitet und weist nahezu dieselbe Funktionalität auf. *TClientDataSet* ist in Kapitel 7 ausführlich beschrieben, eine Wiederholung soll hier nicht erfolgen.

- SQLConnection (Eigenschaft, veröffentlicht)

  ```
  property SQLConnection: TSQLConnection;
  ```

  Mit der Komponenteneigenschaft *SQLConnection* wird die Verbindungskomponente gewählt, über die eine Verbindung zur Datenbank hergestellt wird.

- CommandType, CommandText (Eigenschaften, veröffentlicht)

  ```
  property CommandType: (ctQuery, ctTable, ctStoredProc);
  property CommandText: string;
  ```

  Mit *CommandType* wird eingestellt, wie *CommandText* zu interpretieren ist:

  - Hat *CommandType* den Wert *ctQuery*, dann wird mit *CommandText* eine SQL-Anweisung eingegeben. Dabei solle es sich um ein SELECT-Statement handelt, da *TSQLClientDataSet* die Methode *ExecSQL* nicht kennt.

    Für *CommandText* steht ein Eigenschaftseditor zur Verfügung, der das Erstellen von Abfragen erleichtert.

  - Hat *CommandType* den Wert *ctTable*, dann wird mit *CommandText* ein Tabellenname ausgewählt; der Objektinspektor stellt dafür eine Nachschlageliste zur Verfügung. Die Komponente generiert automatisch eine SQL-Abfrage, welche alle Spalten der betreffenden Tabelle holt (SELECT * FROM tabelle).

  - Bei *ctStoredProc* stellt der Objektinspektor eine Liste aller STORED PROCEDURES zur Verfügung. Verwenden Sie hier nur Prozeduren, die eine Ergebnismenge liefern.

15.2.8 TSQLMonitor

Mit der Komponente *TSQLMonitor* können Sie den Verkehr zum Datenbank-Server »belauschen«.

TSQLMonitor liegt nur in der Enterprise-Version auf der Komponentenpalette. (Wenn Sie im Besitz der Professional-Version sind, dann schauen Sie sich doch mal genau die Datei *SqlExpr.pas* an ...)

Die Komponente *TSQLMonitor* setzt ihrerseits eine Call-Back-Routine ein – den gleichzeitigen Aufruf der *TSQLConnection*-Methode *SetTraceCallbackEvent* sollten Sie sich dann verkneifen.

- SQLConnection, Active (Eigenschaften, veröffentlicht)

  ```
  property SQLConnection: TSQLConnection;
  property Active: Boolean;
  ```

  Mit *SQLConnection* wird die Verbindungskomponente gewählt, deren DLL-Aufrufe protokolliert werden sollen. Mit *Active* kann die Protokollierung ein- und ausgeschaltet werden.

- OnTrace, OnLogTrace (Ereignisse)

  ```
  property OnTrace: TTraceEvent(Sender: TObject;
    CBInfo: pSQLTRACEDesc; var LogTrace: Boolean);
  property OnLogTrace: (Sender: TObject; CBInfo: pSQLTRACEDesc);
  ```

  Das Ereignis *OnTrace* tritt auf, bevor ein DLL-Aufruf in die Log-Liste geschrieben wird – man kann dies verhindern, indem man *LogTrace* auf *false* setzt.

  OnLogTrace tritt auf, nachdem der Aufruf in die Liste geschrieben wurde.

- TraceList (Eigenschaft, veröffentlicht)

  ```
  property TraceList: TStrings;
  ```

  Die DLL-Aufrufe werden in diese String-Liste geschrieben.

- AutoSave, FileName (Eigenschaften, veröffentlicht)

  ```
  property AutoSave: Boolean;
  property FileName: string;
  ```

  Mit diesen Eigenschaften kann man die Log-Liste automatisch in eine Datei speichern. (Alternativ kann man auch *LoadFromFile* und *SaveToFile* aufrufen.)