PEARSON Education

ADDISON-WESLEY

Michael Ebner

C++Builder Datenbankprogrammierung

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Bonn • Reading, Massachusetts • Menlo Park, California • New York • Harlow, England
Don Mills, Ontario • Sydney • Mexico City • Madrid Amsterdam

3 Abfragen mit TQuery

Die Verbindung zu einer Datenbank kann nicht nur über die Komponente *TTable* erfolgen, sondern auch über die Komponente *TQuery*. Dabei handelt es sich um eine Komponente, die sich hauptsächlich für Abfragen eignet und dabei den Industrie-Standard SQL als Abfragesprache verwendet.

Wir wollen zunächst wieder einer kleines Beispiel programmieren, um uns danach ausführlich mit den Eigenschaften, Methoden und Ereignissen von *TQuery* zu beschäftigen.

3.1 Suche nach Telefonnummern

Ein Thema, das immer wieder für viel »Freude« bei Entwicklern und Anwendern sorgt, ist das Thema »Datenbanken und Telefonnummern«. Ganz Unbedarfte werden sich nichts dabei denken und dafür einfach ein numerisches Feld verwenden. Der Anwender gibt dann dort beispielsweise *0305134505* ein, und die Datenbank macht *305134505* daraus. Wenn der Anwender geduldig und tolerant ist, dann wird er sich daran gewöhnen, spätestens aber dann, wenn ausländische und einheimische Nummern durcheinander gehen, dürfte der Geduldsfaden reißen.

Man wird also nicht umhin kommen, für Telefonnummern ein alphanumerisches Feld zu verwenden und dies so groß zu dimensionieren, daß man zur Not auch noch eine Auslandstelefonnummer darin unterbringen kann.

Diesen Platz wird der Anwender sofort nutzen, um die Telefonnummer nach seiner »Hausnorm« zu formatieren. So wird aus der gerade erwähnten Nummer *030/5134505, 030 / 5134505, 030 / 513 45 05, 030 / 51 34 505, 030 513 45 05, 030 - 5135 45 05* oder was es da noch an mehr oder weniger sinnvollen Kombinationen gibt. So weit ist das alles kein Problem, die Datenbank akzeptiert das alles problemlos.

Wenn aber nach dieser Telefonnummer gesucht werden soll, wird es ziemlich schwierig, weil bei einer Stringsuche die exakte Schreibweise bekannt sein sollte. Nun könnte man einwenden, daß der Anwender, wenn er schon seine Telefonnummern formatiert, dies wenigstens so einheitlich tun sollte, daß er seine Einträge wiederfindet. Ich bin allerdings der Meinung, daß Programme, die sich professionell nennen, auch dann möglichst fehlerfrei arbeiten, wenn der Anwender ziemlich schlampig damit umgeht.

Zudem – und das ist wohl das gewichtigere Argument – muß man bei einer Datenbank stets davon ausgehen, daß mehrere Anwender damit arbeiten, und es ist reichlich naiv zu glauben, daß sich auch nur zwei Personen auf eine einheitliche Schreibweise von Telefonnummern einigen könnten. Zur Lösung dieses Problems gibt es nun mehrere Möglichkeiten:

- Durch Vorgabe einer Maske wird die Formatierung durch den Anwender verhindert, das Programm akzeptiert nur Ziffern. Dies ist die »Holzhammer-Methode«; sie funktioniert, aber eine Ziffernfolge wie *0305134505* ist alles andere als leicht (= fehlerfrei) zu erfassen.
- Die Eingabemaske zwingt den Anwender zur Einhaltung einer Norm. Dies ist gerade bei Telefonnummern mit deren Trennung in Vor- und Durchwahl, gegebenenfalls zuzüglich Anschlußnummer oder Auslandsvorwahl, und alles in variabler Länge, ein reichlich schwieriges Unterfangen. Ganz abgesehen davon, daß ich mir als Anwender nicht vom Rechner vorschreiben lassen möchte, wie ich meine Nummern zu formatieren habe.
- Die Telefonnummer wird zweimal gespeichert: Einmal als vom Anwender formatierter String und zum anderen als reine Ziffernfolge, welche das Programm aus der Anwendereingabe generiert. Angezeigt wird dabei nur die Anwendereingabe, gesucht nur in der Ziffernfolge. Das ist nun eine uneingeschränkt saubere Lösung, die lediglich den Nachteil hat, daß mehr Speicherplatz dafür benötigt wird.
- Man kann auch die Abfrage so formulieren, daß die Schreibweise keine Rolle spielt; wie das funktioniert, werden wir uns gleich ansehen. Auch diese Version hat einen Nachteil: Die Suche dauert etwas länger. Da die Abfrage selbst jedoch sehr schnell geht (zum Beispiel 911 msec bei direkter Suche, 982 msec bei der schreibweisentoleranten Suche), halte ich das für ein untergeordnetes Problem. Bei wirklich großen Datenmengen (ab 100 000 Adressen aufwärts) kann man dann zwischen dieser und der vorhergehenden Lösung abwägen.

3.1.1 Ein Programm zur Telefonnummernsuche

Das Beispielprogramm soll Datensätze über die Telefonnummer suchen. Dabei soll die Testdatenbank verwendet werden, welche in Kapitel 2 erstellt wurde. Die Zeit, welche für die Abfrage benötigt wird, soll vom Programm gemessen werden. Um das Programm zu verstehen, ist eine kurze Einweisung in den SQL-Befehl SELECT nötig. Im weiteren Verlauf des Buches werden wir die Abfragesprache SQL noch ausführlich behandeln.

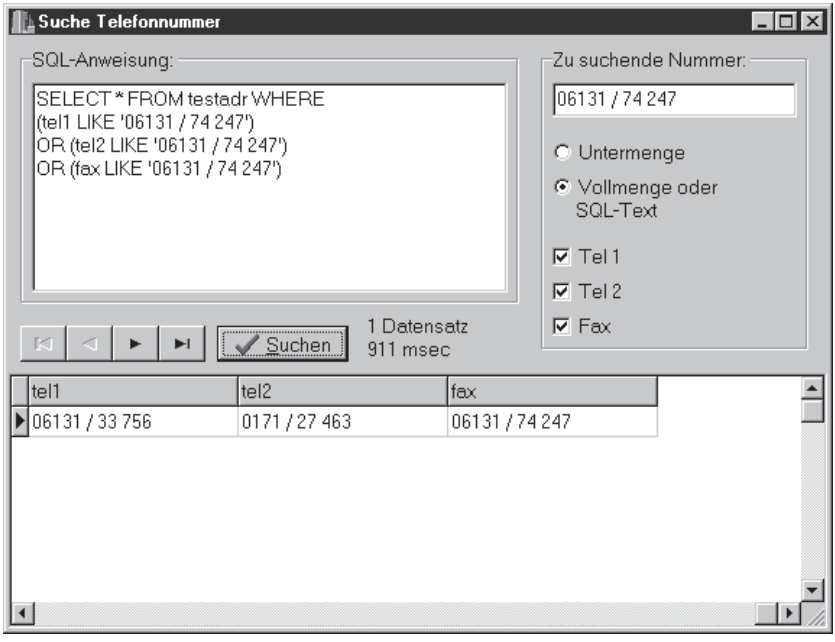

Bild 3.1: Suche nach Telefonnummern mit SQL

Kurz-Einführung in den SQL-Befehl SELECT

Der Befehl SELECT dient der Abfrage von Daten aus einer Tabelle. Seine einfachste Form lautet:

```
SELECT * FROM adressen
```

Dies würde heißen, daß als Datenmenge alle Spalten übergeben werden, welche in der Tabelle *adressen* vorhanden sind. Laut Konvention werden alle SQL-Schlüsselwörter mit Großbuchstaben und alle Variablen mit Kleinbuchstaben geschrieben.

Solange keine Beschränkung genannt wird, werden alle Datensätze der Tabelle übergeben, genau wie dies im Regelfall die Komponene *TTable* tun würde. Nun soll hier allerdings eine Beschränkung erfolgen, da ja nicht alle Adressen angezeigt werden sollen, sondern nur diejenige, nach der über die Telefonnummer gesucht wird. Dazu dient die WHERE-Klausel:

```
SELECT * FROM adressen
WHERE tel1 = "030 / 513 45 05"
```

Hier werdem dann nur noch die Datensätze übergeben, deren Feld *tel1* den Wert *030 / 513 45 05* aufweist. Hier ist eine exakte Schreibweise zwingend erforderlich.

Übergebene Strings müssen übrigens auch bei SQL in Anführungszeichen gesetzt werden, wobei hier sowohl einfache (') als auch doppelte (") Anführungszeichen verwendet werden können.

Um Abfragen zu generieren, welche nach Ähnlichkeiten suchen, gibt es den LIKE-Operator, der zwei Joker-Zeichen erlaubt: Der Unterstrich (_) ersetzt dabei ein einzelnes Zeichen, während das Prozentzeichen (%) kein, ein oder mehrere Zeichen ersetzt.

```
SELECT * FROM adressen
WHERE tel1 LIKE "030%"
```

Diese Abfrage würde nun Datensätze zurückgeben, die mit *030* beginnen, also alle Berliner Telefonnummern. Die WHERE-Klauseln erlauben auch logische Verknüpfungen:

```
SELECT * FROM adressen
WHERE (tel1 LIKE "030%")
  OR (tel1 LIKE "040%")
```

Diese Anwendung würde nun alle Datensätze zurückgeben, die eine Berliner oder Hamburger Vorwahl haben.

```
SELECT * FROM adressen
WHERE (tel1 LIKE "030%")
  AND (tel2 LIKE "0177%")
```

Diese Anweisung sucht nun alle Berliner mit E-Netz-Handy. Zurück zu unserem eigentlichen Problem: Gesucht werden soll die Nummer 030 / 513 45 05, unabhängig von der Schreibweise. Die Lösung ist folgende SQL-Anweisung:

```
SELECT * FROM adressen
WHERE (tel1 LIKE "0%3%0%5%1%3%4%5%0%5%")
```

Erstellt wird nun ein Programm, das aus einer übergebenen Telefonnummer eine solche SQL-Anweisung erstellt.

Das Beispielprogramm

Zunächst wird ein Formular gemäß Bild 3.1 mit Komponenten bestückt. Bis auf das *DBGrid* und den *DBNavigator* handelt es sich um normale, also um nicht datensensitive Komponenten. Der *DBNavigator* wird übrigens auf die Schaltflächen reduziert, die auch tatsächlich benötigt werden, Näheres siehe in Kapitel 4.4.4. Des weiteren werden noch eine *TQuery*- und eine *TTable*-Komponente benötigt.

Der Button wird nun mit folgender *OnClick*-Ereignisbehandlungsprozedur verknüpft:

```
void __fastcall TForm1::BitBtn1Click(TObject *Sender)
{
  int i, t1, t2;
  AnsiString s, t = "";
  char u;
  char v[10];
  bool erster = true;
// Aufbereiten der Zeichenfolge
  s = Edit1->Text;
  for(i = 1; i<= strlen(&s[1]); i++)
  {
    u = s[i];
    if(('0' <= u) && (u <= '9')) t = t + u + "%";
  }
```

Zunächst wird die eingegebene Nummer entsprechend aufgearbeitet. Die Telefonnummer, welche der Anwender in *Edit1* eingegeben hat, wird in einzelne Zeichen zerlegt, die dann daraufhin untersucht werden, ob es sich um eine Ziffer handelt. Ist dies der Fall, dann wird diese gefolgt von einem %-Zeichen dem String *t* hinzugefügt.

```
// SQL-Anweisung generieren
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM testadr WHERE");
  if(CheckBox1->Checked == true)
  {
    if(RadioButton1->Checked == true)
      Memo1->Lines->Add("(tel1 LIKE '" + t + "')");
    else
      Memo1->Lines->Add("(tel1 LIKE '" + Edit1->Text + "')");
    erster = false;
  }
```

Bei der Generierung der SQL-Anweisung wird zunächst die alte Anweisung gelöscht und der Beginn der Anweisung eingefügt. Ist *CheckBox1.Checked* gleich *true*, dann soll im ersten Feld nach der Telefonnummer gesucht werden. Vom Zustand des *RadioButton1* hängt es ab, ob es sich um eine streng kongruente Suche handelt oder ob die Suche verschiedene Formatierungen toleriert. Wird eine strenge Suche gewählt, dann lassen sich auch direkt SQL-Sequenzen eingeben; so könnte man mit der Eingabe *030%* direkt nach Berliner Vorwahlen suchen, während bei der toleranten Suche nach *030* (*0%3%0*) auch Telefonnummern wie ***0****40 / 2****3****7 ****0****9 68* gefunden werden könnten. (Im Beispielprojekt *Tourplaner* werden wir ein Suchfenster entwickeln, welches auch explizit nach Vorwahlen sucht.) Zur Variablen *erster* kommen wir gleich.

```
  if(CheckBox2->Checked == true)
  {
    if(erster == true)s = ""; else s = "OR ";
    if(RadioButton1->Checked == true)
      s = s + "(tel2 LIKE '" + t + "')";
    else
      s = s + "(tel2 LIKE '" + Edit1->Text + "')";
    Memo1->Lines->Add(s);
    erster = false;
  }
```

Im Prinzip handelt es sich bei dieser Sequenz für das Feld *tel2* um die gleichen Anweisungen. Zusätzlich muß jedoch geprüft werden, ob die generierten SQL-Anweisungen die ersten sind (dazu dient die Variable *erster*) oder nicht. Ist letzteres der Fall, dann muß ein OR-Operator eingefügt werden.

Die Sequenz bezüglich der *CheckBox3* und des Fax-Feldes ist ähnlich aufgebaut und braucht hier nicht aufgeführt zu werden.

```
  if(erster == true)
    Memo1->Text = "SELECT * FROM testadr";
// SQL-Abfrage durchführen
  Screen->Cursor = crHourGlass;
  Query1->SQL = Memo1->Lines;
  t1 = GetCurrentTime();
  Query1->Open();
  t2 = GetCurrentTime();
  s = ("%n", i = Query1->RecordCount);
  if(Query1->RecordCount == 1)
    Label2->Caption = "1 Datensatz";
  else
    Label2->Caption = s + " Datensätze";
  s = ("%n", t2 - t1);
  Label3->Caption = s + " msec";
  Screen->Cursor = crDefault;
} // TForm1::BitBtn1Click
```

Zunächst wird nun daraufhin geprüft, ob die Suche für wenigstens ein Telefonnummernfeld aktiviert wurde, ansonsten würde nämlich die SQL-Anweisung *SELECT * FROM testadr WHERE* lauten, was keinen Sinn ergibt. In diesem Fall würde die SQL-Anweisung so formuliert, daß alle Datensätze angezeigt werden.

Während der Abfrage wird der Cursor in eine Sanduhr verwandelt. Die Anweisunungen werden nun von *Memo1* ist die Eigenschaft *SQL* von *Query1*

kopiert, danach wird die Ausführung mit *Query1->Open* gestartet. Da wir die Zeit der Ausführung messen wollen, wird direkt vor dem Start und nach Beendigung der Anweisung die Systemzeit in den Variablen *t1* und *t2* festgehalten. Anschließend werden die Datensätze gezählt, die benötigte Zeit wird berechnet, und beide Ergebnisse werden angezeigt.

3.2 Die Datenbanksprache SQL

Die Datenbanksprache SQL hat sich insbesondere im Bereich der Client-Server-Datenbanken als Standard durchgesetzt. Sie verbindet die Vorteile der weiten Verbreitung mit der leichten Erlernbarkeit und den mächtigen Funktionen.

Die Komponente *TQuery* setzt SQL als Abfragesprache ein, weshalb hier zunächst eine Einführung in SQL erfolgen soll, bevor wir uns mit den Eigenschaften, Methoden und Ereignissen von *TQuery* befassen. Abschließend sollen die Vor- und Nachteile von *TQuery* und *TTable* besprochen werden.

Wir wollen hier jedoch nicht den gesamten Sprachumfang von SQL besprechen; viele SQL-Befehle werden bei der Verwendung von *TQuery* wohl kaum eingesetzt, manche sind beim Zugriff auf die BDE auch nicht verwendbar, sondern können nur an einen SQL-Server weitergeleitet werden. Eine Besprechung dieser Befehle erfolgt dann bei Behandlung des *Local InterBase Servers*.

Den Befehlssatz von SQL kann man in zwei Gruppen einteilen:

- Die Daten-Definitionsanweisungen (data definition language, DDL), mit denen Tabellen, Ansichten, usw. definiert, geändert oder gelöscht werden.
- Die Daten-Manipulationsanweisungen (data manipulation language, DML), zu denen die Befehle zum Einfügen, Ändern und Löschen von Daten sowie der Abfragebefehl SELECT gehören.

Hier sollen zunächst nur der Befehl SELECT (DML) und der Befehl CREATE TABLE (DDL) besprochen werden.

3.2.1 Der Befehl SELECT

Die vollständige abstrakte SELECT-Befehl lautet

```
SELECT DISTINCT [columns]
  FROM [tables]
  WHERE [search_conditions]
  GROUP BY [columns] HAVING [search_conditions]
  ORDER BY [sort_orders]
```

Nicht alle Befehlsoptionen müssen dabei benutzt oder aufgeführt werden. Im einfachsten Fall lautet der SELECT-Befehl:

```
SELECT * FROM testadr
```

Dies würde bedeuten, daß alle Spalten (und alle Datensätze) der Tabelle *adressen* zurückgegeben werden.

Zum Test der folgenden Beispiele gibt es auf der beiliegenden CD (unter Kapitel 3) einen SQL-Editor, bei dem die Beispiel-Anweisungen sich über das Menü einfügen lassen; selbstverständlich ist eine anschließende Modifikation möglich. Im Text werden Hinweise auf diese Menüpunkte geklammert angeführt, beispielsweise: (Text1.1).

Spezifizieren von Spalten und Tabellen, JOINs

Nicht immer ist es wünschenswert, sich alle Spalten einer Tabelle anzeigen zu lassen. Nach dem Befehl SELECT können die Spalten angegeben werden, welche zurückgegeben werden sollen, oder es kann – wie wir das bislang getan haben – mit dem Joker-Zeichen * die Rückgabe aller Spalten veranlaßt werden:

```
SELECT vorname, nachname FROM testadr
```

Diese Anweisung (Text 1.1) würde nun lediglich die Spalten *vorname* und *nachname* zurückgeben.

Des weiteren besteht die Möglichkeit, mehrere Spalten zusammenzufassen und den Feldern vorgegebene Zeichen (-folgen) hinzuzufügen. Die nächste Beispielanweisung (Text 1.2) entnehmen Sie bitte Bild 3.2.

Die verschiedenen Spalten werden mit Hilfe des ||-Zeichens (zweimal Alt Gr + <) zusammengefügt. In der Regel wird es dabei notwendig sein, ein Leerzeichen einzufügen. Die Rückgabe von vorgegebenen Strings ist nicht auf die Verbindung mit Spaltenwerten beschränkt, Sie können auch alle Felder mit demselben String füllen. Sinvoll könnte dies beispielsweise dann sein, wenn mit *TBatchMove* Daten in eine andere Datenbank eingefügt werden sollen. Beachten Sie auch die Spaltenüberschrift im *DBGrid*.

Noch ein Hinweis: Wenn hier dauernd von *Spalten* gesprochen wird, dann ist damit das gemeint, was man im C++Builder *Felder* nennt.

Bisweilen möchte man Daten aus mehreren Tabellen kombinieren. Nehmen wir an, wir hätten in einer Tabelle die offenen Posten (Forderungen) einer Firma. Neben Datum und Betrag sind dort auch die Kunden gespeichert, die diesen Betrag schulden. Ganz im Sinne der Theorie relationaler Datenbanken wird dort aber nicht die volle Kundenadresse gespeichert, sondern nur die Kundennummer.

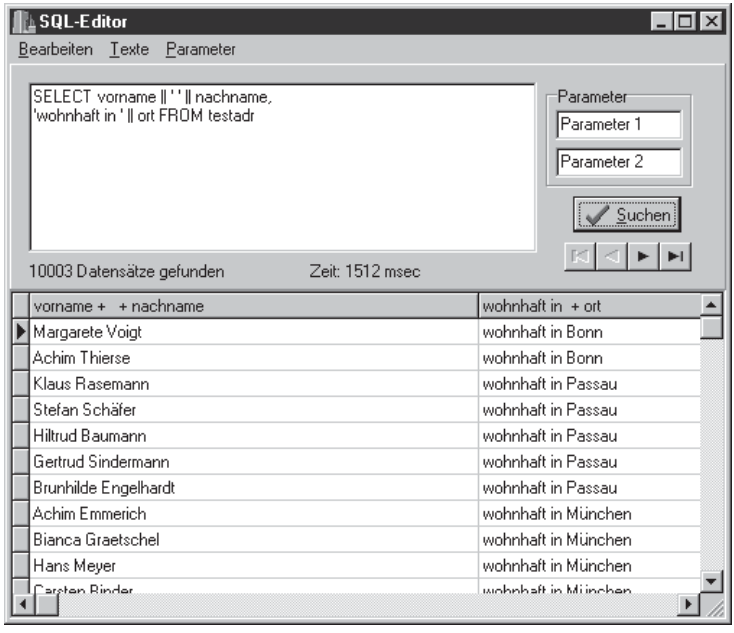

Bild 3.2: Zusammenfassung von Feldern

Wird nun eine Abfrage nach den offenen Posten gestartet, dann bekäme man aber gerne nicht nur die Kundennummern (wer kennt die schon alle auswendig), sondern auch die vollständigen Adressen angezeigt. Dazu ist eine Verbindung von mehreren Tabellen, ein sogenannter »JOIN« nötig. Die entsprechende Anweisung (Text 1.3) lautet:

```
SELECT offenpo.nummer, testadr.vorname || " " || testadr.nachname,
    testadr.straße, testadr.ort, offenpo.betrag, offenpo.datum
  FROM offenpo, testadr
  WHERE offenpo.kunde = testadr.nummer
```

Nach dem FROM-Befehl werden die Tabellen aufgelistet, aus denen bei dieser Abfrage Daten bezogen werden. Nach dem SELECT-Befehl folgen wie immer die einzelnen Spalten; neu ist hier, daß vor jedem Spaltennamen der Tabellenname steht. Die Notwendigkeit dieser Angabe ist leicht einzusehen: Sowohl die Tabelle *offenpo* als auch die Tabelle *testadr* haben ein Feld *nummer*; würde die Tabelle nicht spezifiziert, dann wüßte die Abfrage nicht, welche Spalte nun gemeint ist.

Tabellenname und Spaltenname werden durch einen Punkt getrennt. Dies ist der Grund, warum Sie bei Spaltennamen (Feldnamen) keine Punkte verwenden sollten, auch wenn die Datenbankoberfläche dies akzeptiert: Bei der ersten SQL-Anweisung würden sie Probleme bekommen.

Neben den Spalten und den Tabellen müssen Sie auch noch eine Verknüpfungsanweisung eingeben, wozu die Anweisung nach dem Schlüsselwort WHERE dient. In diesem Fall werden alle Datensätze angezeigt, bei denen die Felder *offenpo.kunde* und *testadr.nummer* übereinstimmen. Dies bezeichnet man als INNER JOIN oder als EQUI JOIN (engl. *equal* gleich). Andere JOINS werden wir uns gleich ansehen.

Zunächst aber zu einem anderen Problem: Das dauernde Wiederholen des Tabellennamens macht das Schreiben der SQL-Anweisung kompliziert und das Ergebnis unübersichtlich. Deshalb erlaubt es SQL, *Synonyme* (manchmal auch Tabellen-Alias genannt) zu verwenden (Text 1.4):

```
SELECT o.nummer, t.vorname || " " || t.nachname,
    t.straße, t.ort, o.betrag, o.datum
  FROM offenpo o, testadr t
  WHERE o.kunde = t.nummer
```

Dazu wird hinter den Tabellennamen (hinter dem Schlüsselwort FROM), getrennt durch ein Leerzeichen, jeweils das Synonym angeführt (hier *o* und *p*). An allen anderen Stellen kann dann statt des Tabellennamens das Synonym angegeben werden. Das Synonym besteht hier nur aus einem einzelnen Buchstaben, es sind aber auch längere Synonyme erlaubt.

Nun zurück zu den JOINs und zu unserer *Offene-Posten*-Tabelle. Für gewöhnlich wird man über eine Referenz sicherstellen, daß für alle *o.kunde*-Werte ein gültiger Datensatz in der Tabelle *testadr* vorliegt. Nun wollen wir einmal annehmen, daß dies nicht geschehen sei, und nun kommt es wie es kommen muß: Für eine oder mehrere offene Posten sind Kundennummern angegeben, für die kein Adressen-Datensatz existiert.

Nun benötigt die Buchhaltung eine Übersicht über die offenen Posten, damit die Bilanz erstellt werden kann. Bei einer Abfrage mit oben angegebener Anweisung würde eine Liste der offenen Posten erstellt, bei denen die Adresse des Kunden in der Adressentabelle vorhanden ist. Wird auf dieser Grundlage nun die Bilanz erstellt, dann kann das Ergebnis nicht stimmen, weil die offenen Posten mit »unbekannter« Adresse unterschlagen werden. (Im Gegensatz zu den Naturwissenschaften sind bei der Buchhaltung Näherungen grundsätzlich unzulässig.)

Wir benötigen also eine Anweisung, die anordnet: *Suche alle offenen Posten. Wo die Adresse des Kunden bekannt ist, gebe diese an*. Diese Anweisung wird mit einem OUTER JOIN realisiert (Text 1.5):

```
SELECT o.nummer,
    t.vorname || " " || t.nachname,
    t.straße, t.ort, o.betrag, o.datum
  FROM offenpo o LEFT OUTER JOIN testadr t
  ON o.kunde = t.nummer
```

Bei einem LEFT OUTER JOIN werden alle Datensätze der linken Seite (also der Tabelle, die vor dem LEFT OUTER JOIN steht) angezeigt. Wo sich Werte der rechten Seite zuordnen lassen, wird dies getan, wo dies nicht möglich ist, werden leere Zellen zurückgegeben.

Des weiteren gibt es auch den RIGHT OUTER JOIN (Text 1.6), der in diesem Fall alle Kundenadressen zurückgeben würde und, sofern bei diesem Kunden offene Posten bestehen, Datum und Betrag.

Zu klären ist noch, was bei Personen geschieht, bei denen mehrere Rechnungen offen stehen. Bei einem LEFT OUTER JOIN werden – wie nicht anders zu erwarten – alle Datensätze der Tabelle *offenpo* angezeigt, bisweilen tritt dieselbe Adresse mehrmals auf. Interessanter wird es beim RIGHT OUTER JOIN: Dort wird der Adressendatensatz zweimal zurückgegeben, einmal kombiniert mit dem ersten offenen Posten, das zweite mal mit dem zweiten; Bild 3.3 zeigt diese Verhaltensweise.

Des weiteren gibt es den FULL OUTER JOIN (Text 1.7): Dieser führt die Datensätze beider Tabellen auf und verbindet diese, wo dies möglich ist. Es würden also Kunden mit unbezahlten Rechnungen, Kunden ohne unbezahlte Rechnungen und unbezahlte Rechnungen ohne Kunden zurückgegeben. In unserer Beispielanwendung startet die Rückgabe immer mit dem ersten Datensatz aus

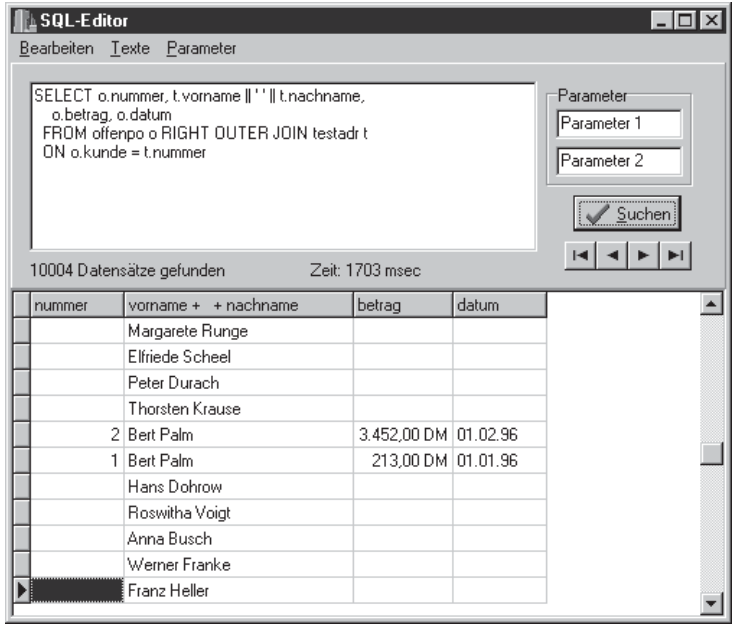

Bild 3.3: Doppelte Einträge beim OUTER JOIN

der Tabelle *testadr* (zumindest auf meinem Rechner), auch wenn man die Angabe der beiden Tabellen vertauscht hat.

Ein Kuriosum ist der *Self-Join*, bei dem derselben Tabelle zwei unterschiedliche Synonyme zugewiesen werden (Text 1.8):

```
SELECT * FROM testadr t, testadr a
  WHERE (t.tel1 = a.tel1)
    AND (t.nummer <> a.nummer)
```

Hier werden alle Datensätze zurückgegeben, welche dieselbe Telefonnummer *tel1* haben, aber eine andere Datensatznummer; gesucht werden also verschiedene Personen mit der gleichen Telefonnummer. Hätten zwei Datensätze dieselbe Telefonnummer, die aber jeweils anders geschrieben wäre, würden diese allerdings nicht ausgegeben. In meiner Testadressen-Tabelle befanden sich immerhin 42 Adressen mit 21 verschiedenen Übereinstimmungen.

Bild 3.4 zeigt, wie man mit einem solchen *Self-Join* die Datenbank auf doppelte Einträge – in diesem Fall nur Vor- und Nachnamen – hin untersucht (Text 1.9). Die Abfrage kombiniert hier in allen möglichen Variationen, so daß aus drei gleichen Namen sechs Datensatzeinträge werden. Die Anweisung DISTINCT sorgt dafür, daß nur Datensätze mit verschiedenen Daten zurückgegeben werden. Würde man auf die Anzeige der Nummern verzichten, dann würden erheblich

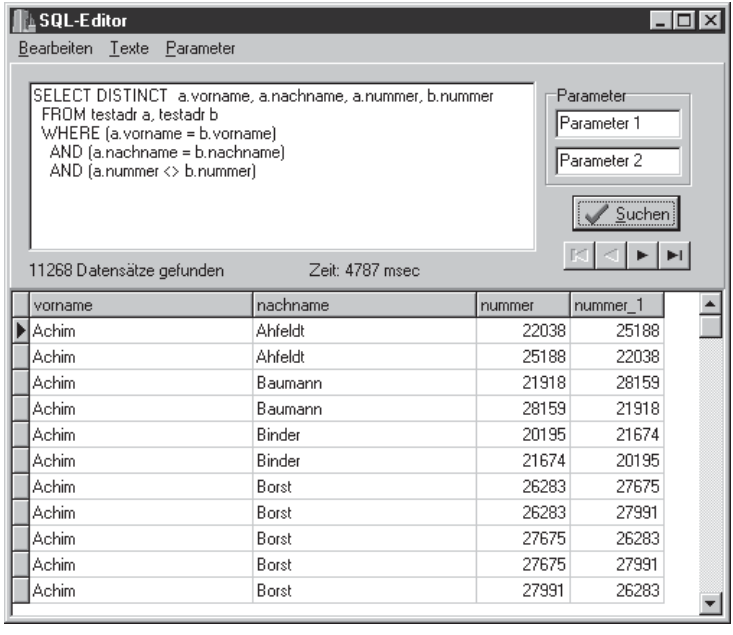

Bild 3.4: Anzeige doppelter Vor- und Nachnamen

weniger Datensätze zurückgegeben (bei meiner Testadressen-Datenbank 2866 statt 11 698).

Selektieren mit der WHERE-Anweisung

Die WHERE-Anweisung, die wir schon ein wenig verwendet haben, beschränkt die Rückgabe auf diejenigen Werte, welche den angegebenen Anforderungen genügen. Beispielsweise lassen sich alle Berliner Adressen aussortieren (Text 2.1):

```
SELECT * FROM testadr
  WHERE ort = "Berlin"
```

Die SELECT-Anweisung erlaubt sowohl AND(*und*)- als auch OR(*oder*)-Verknüpfungen. Soll nach einem Teilstring gesucht werden, verwendet man die LIKE-Anweisung (Text 2.2):

```
SELECT * FROM testadr
  WHERE (ort = "Berlin")
    AND (tel2 LIKE "0161%")
```

Diese Anweisung würde alle Berliner Adressen heraussuchen, die ein C-Netz-Telefon haben, deren *tel2* also mit *0161* beginnt. Das *%*-Zeichen ist ein sogenanntes »Joker-Zeichen« und ersetzt entweder kein, ein oder mehrere andere Zeichen. Es gibt auch das _-Zeichen, welches exakt ein anderes beliebiges Zeichen ersetzt (Text 2.3):

```
SELECT * FROM testadr
  WHERE ((ort = "Berlin") OR (ort = "Hamburg"))
    AND (tel2 LIKE "0161 / _5%")
```

Diese Anweisung würde alle Berliner und Hamburger zurückgeben, deren C-Netz-Durchwahl als zweites Zeichen eine *5* hat (suchen Sie mal lieber keinen Sinn in dieser Anweisung). Die zusätzliche Einklammerung der Sequenz *(ort = "Berlin") OR (ort = "Hamburg")* ist zwingend erforderlich, weil die Abfrage sonst folgendermaßen interpretiert würde:

```
SELECT * FROM testadr
  WHERE (ort = "Berlin") OR ((ort = "Hamburg")
    AND (tel2 LIKE "0161 / _5%"))
```

Eine Abfrage nach leeren Datensätzen läßt sich über das Schlüsselwort NULL erzielen. Hier wird allerdings nicht mit dem Gleichheitszeichen gearbeitet, sondern mit dem Schlüsselwort IS, bei der Suche nach Feldern mit Inhalt lautet die Sequenz dann IS NOT NULL. Das folgende Beispiel sucht nach Berliner Adressen mit Funktelefon (Text 2.4):

```
SELECT * FROM testadr
  WHERE (ort = "Berlin")
    AND (tel2 IS NOT NULL )
```

SQL erlaubt auch die Operatoren größer (>), kleiner (<), größer / gleich (>=) und kleiner / gleich (<=). Die folgende Abfrage sucht alle Adressen, die dem Postleitzahlenbreich *2* zugehören (Text 2.5):

```
SELECT * FROM testadr
  WHERE ((plz >= "20 000") AND (plz < "30 000"))
```

Statt zwei Vergleiche mit einer UND-Bedingung zu verknüpfen, kann man auch den BETWEEN-Operator verwenden (Text 2.6).

```
SELECT * FROM testadr
  WHERE plz BETWEEN "20 000" AND "29 999"
```

Die nächste Anweisung sucht nach den Adressen in den Postleitzahlgebieten 2 und 8. Hier wird statt mit größer / kleiner-Operatoren mit LIKE und einem String-Vergleich gearbeitet (Text 2.7):

```
SELECT * FROM testadr
  WHERE (plz LIKE "2%")
    OR (plz LIKE "8%")
```

Im Vergleich dazu die Abfrage, welche die gleiche Ergebnismenge zurückgibt, aber den BETWEEN-Operator verwendet (Text 2.8).

```
SELECT * FROM testadr
  WHERE (plz BETWEEN "20 000" AND "29 999")
    OR (plz BETWEEN "80 000" AND "89 999")
```

Werden viele Gleichheitsbedingungen mit einer ODER-Verknüpfung verbunden, so kann man dies auch übersichtlicher mit dem IN-Operator gestalten. In diesem Fall wird eine Person namens *Claudia* gesucht, die in einer deutschen Millionenstadt lebt (Text 2.9):

```
SELECT * FROM testadr
  WHERE (vorname = "Claudia")
    AND (ort IN ("Berlin", "Hamburg", "München"))
```

Die Aggregat-Funktionen

Zurück zu der Tabelle der offenen Posten. Nehmen wir einmal an, die Buchhaltung möchte wissen, wie hoch die Summe aller Außenstände ist. Nun ist es aufwendig und fehleranfällig, sich die Liste aller Außenstände anzeigen zu lassen und diese dann mit dem Taschenrechner zu addieren. Einfacher ist es, sich die Summe gleich von der Abfrage anzeigen zu lassen (Text 3.1):

```
SELECT SUM(betrag) FROM offenpo
```

Hier wird eine Tabelle mit nur einem Wert (und der Spaltenüberschrift) zurückgegeben, nämlich die Summe der Außenstände. Neben der Summe gibt es noch weitere Aggregatfunktionen (Text 3.2):

```
SELECT SUM(betrag), COUNT(betrag),
  MIN(betrag), MAX(betrag),
  AVG(betrag) FROM offenpo
```

Die Funktion COUNT zählt die Datensätze, MIN und MAX ermitteln den jeweils kleinsten und größten Wert, die Funktion AVG bildet den Durchschnitt. Mit einer WHERE-Klausel läßt sich die Datenmenge, über welche die Funktion gebildet wird, weiter einschränken. Im folgenden Beispiel (Bild 3.5) werden die Summe und die anderen Funktionen aller Außenstände mit Berliner Adressen ermittelt (Text 3.3).

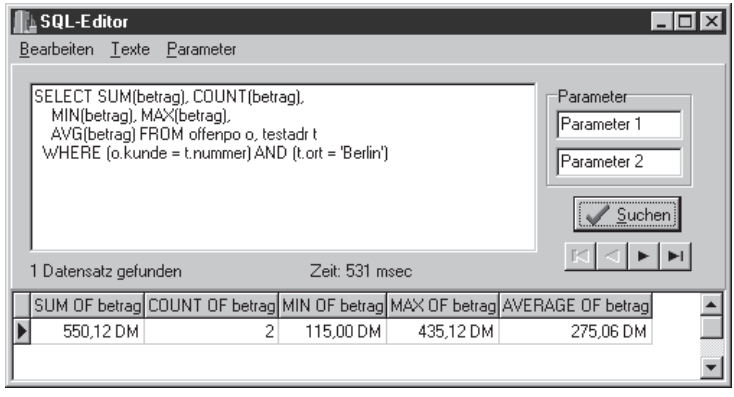

Bild 3.5: Aggregat-Funktion mit Equi-Join

Will man nun auf diese Weise die Außenstände nach Städten (oder wonach auch immer) sortieren, dann ist man ziemlich beschäftigt. Deshalb gibt es auch die Möglichkeit, die Datenmenge zu gruppieren und die Aggregatfunktionen über die jeweilige Gruppe zu bilden (Text 3.4):

```
SELECT t.ort, SUM(o.betrag), COUNT(o.betrag),
    MIN(o.betrag), MAX(o.betrag),
    AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort
```

Hier wird die Datenmenge nach Orten gruppiert und von den Außenständen der Kunden, die im jeweiligen Ort wohnen, werden die Summe und die übrigen Aggregatfunktionen gebildet. Da es sich um einen Equi-Join handelt, werden nur diejenigen Orte angezeigt, in denen säumige Kunden wohnen. Alle anderen Orte werden nicht angezeigt, genausowenig ein »leerer« Ort für die offenen Forderungen, deren Kundennummern sich nicht zuordnen lassen.

Um alle Orte anzuzeigen, auch wenn die Zahl der offenen Posten dort Null beträgt, wird ein RIGHT oder FULL OUTER JOIN benötigt. Bei letzterem wird dann auch ein Datensatz ohne Ort-Namen angezeigt, welcher die Außenstände aufsummiert, deren Kundennummern sich nicht zuordnen lassen.

```
SELECT t.ort, SUM(o.betrag), COUNT(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o FULL OUTER JOIN testadr t
  ON (o.kunde = t.nummer)
  GROUP BY t.ort
```

Für gewöhnlich werden alle Datensätze nach dem Wert in der ersten Spalte sortiert. Es wäre aber denkbar, daß eine andere Sortierreihenfolge gewünscht wird. Beispielsweise soll nach der Höhe der Außenstände sortiert werden (Text 3.5).

```
SELECT t.ort, SUM(o.betrag), COUNT(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort
  ORDER BY SUM(o.betrag)

/* funktioniert nicht */
```

Das Sortieren von Datensätzen geschieht mit der Anweisung ORDER BY, welche wir später noch ausführlicher behandeln werden. Das Problem in diesem Fall ist, daß weder die 16- noch die 32-Bit-BDE die Anweisung *ORDER BY SUM(o.betrag)* akzeptieren. (Die 16-Bit-BDE, die beispielsweise bei Delphi 1.0 zum Einsatz kommt, akzeptiert die Anweisung *ORDER BY o.betrag*.) Mit dem »Trick« einer Spaltenumbenennung kommt man hier jedoch problemlos ans Ziel (Text 3.6):

```
SELECT t.ort, COUNT(o.betrag) AS gesamtsumme, SUM(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort
  ORDER BY gesamtsumme
```

In vielen Firmen werden Forderungen, welche unter einer bestimmten Grenze liegen, nicht weiter verfolgt – der Aufwand wäre zu groß. Nehmen wir einmal an, diese Grenze läge bei 100,- DM, und nun soll eine Statistik der nach Orten gruppierten Außenstände erstellt werden, bei denen die Bagatellbeträge ignoriert werden. Auch diese Möglichkeit besteht, und zwar mit der HAVING-Klausel (Text 3.7):

```
SELECT t.ort, COUNT(o.betrag), SUM(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.ort HAVING SUM(o.betrag) >= 100
```

Bei der HAVING-Klausel wird die Gruppierung eingeschränkt wie mit der WHERE-Klausel die Abfrage; auch die Syntax ist dieselbe. Auf ähnliche Weise kann man feststellen, welche Kunden mehrere Rechnungen nicht bezahlt haben (Text 3.8):

```
SELECT t.nachname, t.vorname, COUNT(o.betrag), SUM(o.betrag),
     MIN(o.betrag), MAX(o.betrag),
     AVG(o.betrag) FROM offenpo o, testadr t
  WHERE (o.kunde = t.nummer)
  GROUP BY t.nachname, t.vorname HAVING COUNT(o.betrag) >= 2
```

Beachten Sie dabei, daß in die GROUP-Anweisung sowohl *t.nachname* als auch *t.vorname* aufgeführt werden müssen, sonst generiert die BDE eine Fehlermeldung. In der nächsten Anweisung wird eine kleine Statistik aus der Adressendatenbank erstellt. Zum einen wird nach Städten aufgelistet, wieviel Adressen es gibt und wie viele dort eine Funktelefonnummer haben. Als weitere Spalte wird dann der Prozentsatz der Funktelefone pro Adressen berechnet (Text 3.9):

```
SELECT t.ort, COUNT(t.nummer), COUNT(t.tel2),
     (COUNT(t.tel2) * 100) / COUNT(t.nummer)
  FROM testadr t
  GROUP BY t.ort
```

Sortieren der Einträge

Solange nichts anderes angegeben wird, werden die Datensätze in alphabetischer bzw. numerischer Reihenfolge aufsteigend bezüglich der ersten Spalte sortiert. Diese Reihenfolge läßt sich mit der ORDER BY-Anweisung verändern (Text 4.1):

```
SELECT * FROM testadr t
  ORDER BY t.nachname, t.vorname, t.ort
```

Hier werden die Einträge in aufsteigender alphabetischer Reihenfolge zunächst nach *t.nachname* sortiert. Bei gleichen Einträgen in *t.nachname* werden diese nach *t.vorname* sortiert, zuletzt nach *t.ort*. Eine absteigende Sortierung wird mit dem Schlüsselwort DESC erreicht (Text 4.2):

```
SELECT * FROM testadr t
  ORDER BY t.nachname, t.vorname DESC, t.ort
```

Hier wird bei gleichen *t.nachname* (welche aufsteigend sortiert werden) nach *t.vorname* absteigend sortiert; die Liste beginnt mit *Xaver Ackermann* statt mit *Achim Ackermann*. Für jede Spalte, nach der sortiert wird, kann bestimmt werden, ob sie aufsteigend (ASC, kann entfallen) oder absteigend (DESC) sortiert wird.

```
SELECT t.nummer, t.nachname || ", " || t.vorname AS Name
    FROM testadr t
  ORDER BY name
```

Hier werden die Spalten *t.nachname* und *t.vorname* zu einem Feld zusammengefaßt (Text 4.3). Diesem wird die Bezeichnung *name* zugewiesen, der gesamte String ist Grundlage für die Sortierung. Werden Felder, Rechenergebnisse oder zusammengefügte Strings mit AS umbenannt, dann erscheint diese Bezeichnung auch als Spaltenüberschrift im *DBGrid* (aus diesem Grund beginnt *Name* auch mit einem Großbuchstaben).

Die Möglichkeit der Umbenennung der Spaltenüberschriften ist im nächsten Beispiel besonders sinnvoll (Versuchen Sie es einmal ohne...) (Text 4.4):

```
SELECT o.datum, EXTRACT (DAY FROM o.datum) AS tag,
    EXTRACT (MONTH FROM o.datum) AS monat,
    EXTRACT (YEAR FROM o.datum) AS jahr
  FROM offenpo o
```

Hier werden der Tag, der Monat und das Jahr mit EXTRACT als Einzelwerte aus dem Datum ermittelt. Zuletzt wollen wir noch ein wenig rechnen, und zwar wieder mit der Tabelle der offenen Posten. Zunächst soll eine Mahngebühr von 5,- DM eingeführt werden, dann soll nicht nur das Datum der Rechnungsstellung angezeigt werden, sondern auch das Datum drei Wochen später, denn dann soll die Mahnung geschrieben werden (Text 4.5):

```
SELECT t.vorname, t.nachname, o.betrag AS rechnungssumme,
    o.betrag + 5 AS mahnsumme, o.datum, o.datum + 21 AS mahndatum
  FROM offenpo o, testadr t
  WHERE o.kunde = t.nummer
```

Nun wollen wir aus der Rechnungssumme auch noch den Nettobetrag berechnen (MWSt 15 %, wobei ich nicht weiß, ob das während der »Laufzeit« des Buches aktuell bleibt ...) (Text 4.6):

```
SELECT t.vorname, t.nachname, o.betrag AS rechnungssumme,
    o.betrag/1.15 AS nettosumme, o.datum, o.datum + 21 AS mahndatum
  FROM offenpo o LEFT OUTER JOIN testadr t
  ON o.kunde = t.nummer
```

Hier werden wiederum alle offenen Posten angezeigt (OUTER JOIN).

Unterabfragen

Bisweilen kommt es vor, daß man für eine Abfrage einen Wert benötigt, den man erst aus einer anderen Abfrage erhält. Hier ist es dann sinnvoll, eine Unterabfrage zu verwenden. Als Beispiel sollen hier die Kunden ermittelt werden, deren (einzelne) Außenstände über dem Durchschnitt liegen (Text 4.7):

```
SELECT t.vorname, t.nachname, o.betrag AS rechnungssumme
  FROM offenpo o LEFT OUTER JOIN testadr t
    ON o.kunde = t.nummer
  WHERE o.betrag >
    (SELECT AVG(o.betrag) FROM offenpo o)
```

Der Mittelwert der einzelnen Außenstände wird hier durch die Unterabfrage ermittelt; bei der Unterabfrage handelt es sich wieder um einen SELECT-Befehl mit der bekannten Syntax.

3.2.2 Erstellen von Tabellen

Für gewöhnlich wird man Tabellen, Indizes und ähnliches mit der Datenbankoberfläche erstellen. Bisweilen kann es aber sinnvoll sein, eine Tabelle aus den Komponenten *TTable* oder *TQuery* heraus anzulegen. Leider sind dabei einige wichtige Merkmale bei Desktop-Datenbanken nicht verfügbar, so daß diese Vorgehensweise schnell an ihre Grenzen stößt. Wir werden deshalb auf die *DDL* (*data definition language*) erst bei der Behandlung des *Local InterBase Servers* näher eingehen. Die Erstellung einer Tabelle erfolgt mit der SQL-Anweisung CREATE TABLE (Text 5.1):

```
CREATE TABLE "test_1.db"
  (nummer AUTOINC,
  vorname CHAR(20),
  nachname CHAR(20),
  straße CHAR(20),
  plz CHAR(6),
  ort CHAR(20),
  tel1 CHAR(20),
  tel2 CHAR(20),
```

```
  fax CHAR(20),
  PRIMARY KEY(nummer))

  /* Bearbeiten | SQL-Anweisung ausführen
```

Zunächst beachten Sie bitte, daß die SQL-Anweisung nicht mit dem Button *Suchen* gestartet werden darf, sondern es muß dafür der Menü-Punkt BEARBEITUNG | SQL-ANWEISUNG AUSFÜHREN verwendet werden. Im Listing sehen Sie an dieser Stelle auch, wie man Kommentare in SQL-Anweisungen einfügt.

Durch die Dateiendung *.db* wird die BDE angewiesen, eine Paradox-Tabelle zu erstellen. Die Nummer soll selbstinkrementierend automatisch vergeben werden, dafür wird der Spaltentyp AUTOINC verwendet. Alphanumerische Felder haben hier den Spaltentyp CHAR, die Länge muß dabei vorgegeben werden. Weitere häufig benötigte Feldtypen sind:

- *SMALLINT* (16 Bit) und *INTEGER* (32 Bit)
- *NUMERIC* (Gleitkommazahl)
- *DATE* (Datum)
- *MONEY* (Währung)

Näheres zu den Typen in der Online-Hilfe oder in den Kapiteln über den *Local InterBase Server*. Die Tabelle soll nun mit einem Datensatz versehen werden (Text 5.2):

```
INSERT INTO "test_1.db" (vorname, nachname, straße,
  plz, ort, tel1, tel2, fax) VALUES ("Ludwig", "Meier",
  "Mozartstraße 32", "10 001", "Berlin",
  "030 / 123 45 67", "", "030 / 123 45 68");

  /* Bearbeiten | SQL-Anweisung ausführen
```

Die Werte werden in der Reihenfolge in die Tabelle eingefügt, in der die Spalten vor dem Schlüsselwort VALUES aufgeführt sind. Vergessen Sie nicht, die Ausführung mit BEARBEITEN | SQL-ANWEISUNG AUSFÜHREN zu starten. Mit der bekannten Anweisung schaut man sich nun die Tabelle an (Text 5.3):

```
SELECT * FROM test_1
```

Zum Schluß soll die Tabelle wieder entfernt werden. Dazu verwendet man den Befehl DROP TABLE (Text 5.4):

```
DROP TABLE test_1

  /* Bearbeiten | SQL-Anweisung ausführen
```

3.3 Parameter

Die Verwendung von SQL-Anweisungen kann mit Hilfe von Parametern weiter vereinfacht werden. Nehmen wir einmal an, daß alle Datensätze gesucht werden sollten, deren Spalte *vorname* gleich dem ist, was der Anwender in *Edit1* eingibt, und deren Spalte *nachname* gleich dem ist, was in *Edit2* angegeben wurde. Die entsprechende Anweisung könnte man folgendermaßen konstruieren:

```
void __fastcall TForm1::ohneParameter1Click(TObject *Sender)
{
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM testadr");
  Memo1->Lines->Add("  WHERE (vorname = '" + Edit1->Text + "')");
  Memo1->Lines->Add("    AND(nachname = '" + Edit2->Text + "')");
}
```

Man kann das Problem aber auch mittels der Verwendung von Parametern lösen:

```
void __fastcall TForm1::mitParameter1Click(TObject *Sender)
{
  Memo1->Clear();
  Memo1->Lines->Add("SELECT * FROM testadr");
  Memo1->Lines->Add("  WHERE (vorname = :para1)");
  Memo1->Lines->Add("    AND(nachname = :para2)");
}

void __fastcall TForm1::Suchen1Click(TObject *Sender)
{
  AnsiString s;
  int t1, t2;
  try
  {
    Screen->Cursor = crHourGlass;
    Query1->SQL->Clear();
    Query1->SQL = Memo1->Lines;
    Query1->ParamByName("para1")->AsString = Edit1->Text;
    Query1->ParamByName("para2")->AsString = Edit2->Text;
    t1 = GetCurrentTime();
    Query1->Open();
    t2 = GetCurrentTime();
    s = ("%n", Query1->RowsAffected);
```

```
    Label1->Caption = s + " Zeilen bearbeitet";
    s = ("%n", t2 - t1);
    Label2->Caption = "Zeit: " + s + " msec";
    Screen->Cursor = crDefault;
  } //try
  catch(...)
  {
    Screen->Cursor = crDefault;
    MessageDlg("SQL-Anweisung fehlerhaft", mtWarning,
      TMsgDlgButtons() << mbOK,0);
  } //catch
} // TForm1::Suchen1Click
```

In der SQL-Anweisung werden die Parameter *para1* und *para2* durch Voranstellung eines Doppelpunktes definiert, diesen werden dann über die Eigenschaft *Params* oder die Methode *ParamByName* die Texte von *Edit1* bzw. *Edit2* zugewiesen. In der Eigenschaft *Params* müssen die Parameter in derselben Reihenfolge aufgeführt sein wie sie in der SQL-Anweisung definiert sind, etwas weniger fehleranfällig ist hier die Methode *ParamByName*, die mit den Parameternamen arbeitet. Bevor die Parameter zugewiesen werden, muß der Komponente *TQuery* die SQL-Anweisung zugewiesen werden (woran sonst sollte sie erkennen, welche Art von Parametern sie entgegennehmen soll?).

Über Parameter läßt sich auch eine Master-Detail-Beziehung zu einer anderen Tabelle herstellen. Nehmen wir einmal an, daß die *Table1/DataSource1*-Kombination uns die Personen der *testadr*-Tabelle anzeigt. Nun wüßten wir bei der Anzeige der Personen (etwa bei einem Neuauftrag) auch gerne, wie viele Rechnungen von diesen Personen noch nicht bezahlt worden sind. Es soll also in einem *DBGrid* angezeigt werden, welche Außenstände der betreffende Kunde bei der Firma hat. Bild 3.6 zeigt ein solches Fenster. (Es würde sich besser machen, wenn man die Kundenadresse über einzelne *DBEdit*-Felder und nicht über ein *DBGrid* anzeigen würde, aber hier kommt es momentan nicht auf Schönheit an.)

Das zweite *DBGrid* mit den offenen Posten wird über *Query1/DataSource2* mit Daten versorgt. Die Eigenschaft *SQL* von *Query1* lautet nun folgendermaßen:

```
SELECT * FROM offenpo
  WHERE kunde = :nummer
```

Nun wird der Parameter *nummer* nicht über die Eigenschaft *Params* zugewiesen (die bleibt in diesem Fall leer), sondern die Eigenschaft *DataSource* wird auf *DataSource1* gesetzt. Jedesmal, wenn in der Kundentabelle ein neuer Datensatz aktuell wird, wird *Query1* die Kundennummer neu zugewiesen, und *Query1* sucht sich anhand der Kundennummer dann die Außenstände dieser Person heraus. Der Parameter muß dabei den Namen der Spalte haben, aus der er den Verknüpfungswert – hier die Kundennummer – beziehen soll.

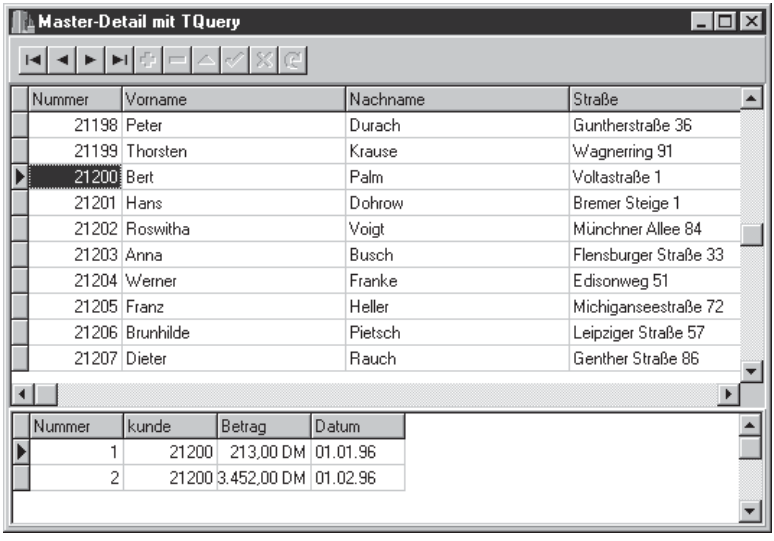

Bild 3.6: Master-Slave-Beziehung mit TQuery

Wenn zusätzlich noch die Summe der Außenstände des betreffenden Kunden angezeigt werden soll, dann wird eine zweite Abfrage benötigt, welche auf dieselbe Weise mit *DataSource1* verknüpft wird. Die SQL-Anweisung muß hier dann folgendermaßen lauten:

```
SELECT SUM(betrag) FROM offenpo
  WHERE kunde = :nummer
```

3.4 Referenz TQuery

Hier sollen nun die Eigenschaften, Methoden und Ereignisse von *TQuery* besprochen werden. Die Komponente *TQuery* ist – ebenso wie *TTable* – von *TDBDataSet* abgeleitet. Die Mehrzahl der Eigenschaften, Methoden und Ereignisse übernimmt sie daher von *TDataSet* und *TDBDataSet*. Die Referenz dieser beiden Komponenten erfolgte bereits in Kapitel 2 und soll hier nicht widerholt werden. Bei den geerbten Eigenschaften, Methoden und Ereignissen wird deshalb lediglich auf die jeweilige Klasse verwiesen.

Alle Eigenschaften und Methoden, die in *TComponent*, *TPersistent* oder *TObject* implementiert wurden, werden im folgenden nicht erwähnt.

3.4.1 Die veröffentlichten Eigenschaften von TQuery

- Active (*TDataSet*)
- AutoCalcFields (*TDataSet*)
- CachedUpdates (*TDataSet*)
- Constrained (*TQuery*)

  ```
  __property bool Constrained;
  ```

  Ist *Constrained* gleich *true*, dann dürfen bei Paradox- und dBase-Tabellen keine Datensätze eingefügt werden, welche nicht die Bedingungen der aktuellen SQL-Anweisung erfüllen. Angenommen, die SQL-Anweisung einer *TQuery*-Komponente würde folgenermaßen lauten:

  ```
  SELECT * FROM testadr
    WHERE ort = "Berlin"
  ```

  Die folgenden Anweisungen werden dann nicht ausgeführt, wenn *Constrained* gleich *true* ist:

  ```
  INSERT INTO testadr (nachname, ort)
    VALUES ('Meyer', 'Hamburg')

  UPDATE testadr
    SET ort = 'Frankfurt'
    WHERE nummer = 12345
  ```

- DatabaseName (*TDBDataSet*)
- DataSource (*TQuery*)

  ```
  __property Db::TDataSource DataSource;
  ```

  Gibt an, aus welcher Datenmenge die Parameter entnommen werden, die anderweitig nicht definiert sind. Die Parameternamen und die Spaltennamen müssen dabei übereinstimmen. Mit Hilfe der Eigenschaft *DataSource* kann eine aus *TQuery*-Komponenten bestehende Master-Detail-Verknüpfung erstellt werden.

- Filter (*TDataSet*)
- Filtered (*TDataSet*)
- FilterOptions (*TDataSet*)

- ParamCheck (*TQuery*)

  ```
  __property bool ParamCheck;
  ```

  Ist *ParamCheck* gleich *true*, dann werden alle Parameter gelöscht, wenn eine neue SQL-Anweisung zugewiesen wird. Per Voreinstellung ist *ParamCheck* gleich *true* und sollte in der Regel auf diesem Wert bleiben, weil ansonsten nicht sichergestellt ist, daß die Parameter zur SQL-Anweisung passen.

- Params (*TQuery*)

  ```
  __property TParams* Params
  ```

  Die Array-Eigenschaft *Params* enthält alle Parameter einer SQL-Anweisung. Die Indexzählung beginnt bei Null für den ersten in der SQL-Anweisung erwähnten Parameter. Gegeben sei folgende SQL-Anweisung:

  ```
  SELECT * FROM testadr
    WHERE (vorname = :vor)
      AND (nachname = :nach)
  ```

  Mit den folgenden Anweisungen kann man nun die Parameter setzen:

  ```
  Query1->Params[0]->AsString = "Yvonne";
  Query1->Params[1]->AsString = "Bach";
  ```

  Die Verwendung der Eigenschaft *Params* führt leicht zu Fehlern, wenn die SQL-Anweisung modifiziert und deshalb die Reihenfolge der Parameter verändert wird. Sicherer ist hier die Verwendung der Methode *ParamByName*.

- RequestLive (*TQuery*)

  ```
  __property bool RequestLive;
  ```

  Ist *RequestLive* – gemäß der Voreinstellung – gleich *false*, dann liefert die Komponente eine Ereignismenge, welche nicht editiert werden kann. Sollen jedoch Datensätze eingefügt, geändert oder gelöscht werden, muß die Eigenschaft *RequestLive* auf *true* gesetzt werden.

- SessionName (*TDBDataSet*)

- SQL (*TQuery*)

  ```
  __property Classes::TStrings* SQL;
  ```

  Die Eigenschaft *SQL* beinhaltet die SQL-Anweisung von *TQuery*. Es sind dabei nicht nur Abfragen erlaubt, sondern auch Datenänderungsbefehle (INSERT, UPDATE, DELETE) und Datendefinitionsbefehle (CREATE TABLE, DELETE TABLE, usw.).

  Wird die Eigenschaft *SQL* modifiziert, dann wird die Datenmenge automatisch geschlossen. Bei Abfragen muß sie anschließend mit *Open* geöffnet werden, alle anderen SQL-Anweisungen werden mit *ExecSQL* ausgeführt.

- UniDirectional (*TQuery*)

  ```
  __property bool UniDirectional;
  ```

  Legt fest, ob der BDE-Datencursor in beide Richtungen bewegt werden kann oder nicht; per Voreinstellung *false* und in der Regel ohne Bedeutung.
- UpdateMode (*TDBDataSet*)
- UpdateObject (*TDataSet*)

3.4.2 Die öffentlichen Eigenschaften von TQuery

- BOF (*TDataSet*, nur Lesen)
- Bookmark (*TDataSet*)
- CanModify (*TDataSet*, nur Lesen)
- Database (*TDataSet*, nur Lesen)
- DBHandle (*TDBDataSet*, nur Lesen)
- DBLocale (*TDBDataSet*, nur Lesen)
- DBSession (*TDBDataSet*, nur Lesen)
- DefaultFields (*TDataSet*, nur Lesen)
- Designer (*TDataSet*, nur Lesen)
- EOF (*TDataSet*, nur Lesen)
- ExpIndex (*TDataSet*, nur Lesen)
- FieldCount (*TDataSet*, nur Lesen)
- FieldDefs (*TDataSet*)
- Fields (*TDataSet*)
- FieldValues (*TDataSet*)
- Found (*TDataSet*, nur Lesen)
- Handle (*TDataSet*, nur Lesen)
- KeySize (*TDataSet*, nur Lesen)
- Local (*TQuery*, nur Lesen)

  ```
  __property bool Local;
  ```

  Ist *true*, wenn sich die Abfrage nur auf *Paradox*- oder *dBase*-Tabellen bezieht. Richtet sich die Abfrage an einen SQL-Server, ist *Local* gleich *false*.
- Locale (*TDataSet*, nur Lesen)

- Modified (*TDataSet*, nur Lesen)
- ParamCount (*TQuery*)

  ```
  __property unsigned short ParamCount;
  ```

  Die Eigenschaft *ParamCount* gibt die Anzahl der Parameter in der aktuellen Abfrage an.
- Prepared (*TQuery*)

  ```
  __property bool Prepared;
  ```

  Mit der Eigenschaft *Prepared* läßt sich feststellen, ob eine Abfrage vorbereitet ist. Um eine Abfrage vorzubereiten, kann die Eigenschaft *Prepared* auf *true* gesetzt werden, in der Regel wird man jedoch zu diesem Zweck die Methode *Prepare* aufrufen.
- RecordNo (*TDataSet*, nur Lesen)
- RecordCount (*TDataSet*, nur Lesen)
- RecordSize (*TDataSet*, nur Lesen)
- RowsAffected (*TQuery*, nur Lesen)

  ```
  __property int RowsAffected;
  ```

  Mit der Eigenschaft *RowsAffected* kann festgestellt werden, wie viele Datensätze bei der letzen SQL-Anweisung verarbeitet (eingefügt, aktualisiert oder gelöscht) wurden. Durch die Anzeige dieser Zahl kann der Anwender sehen, ob die von ihm formulierte SQL-Anweisung den gewünschten Erfolg hatte.

  ```
  AnsiString s;
  s = ("%n", Query1->RowsAffected);
  Label1->Caption = s + " Zeilen bearbeitet";
  ```

- SQLBinary (*TQuery*)

  ```
  __property char* SQLBinary;
  ```

  Die Eigenschaft *SQLBinary* zeigt auf einen binären Daten-Stream, mit dessen Hilfe *TQuery* mit der BDE kommuniziert. Diese Eigenschaft sollte nicht verwendet werden.
- State (*TDataSet*, nur Lesen)
- StmtHandle (*TQuery*, nur Lesen)

  ```
  __property Bde::hDBIStmt StmtHandle;
  ```

  Die Eigenschaft *StmtHandle* bezeichnet das Anweisungshandle der BDE. Diese Eigenschaft wird nur benötigt, wenn Funktionen der BDE-API direkt aufgerufen werden.

- Text (*TQuery*, nur Lesen)

  ```
  __property System::AnsiString Text;
  ```

  Die Eigenschaft *Text* enthält die vollständige SQL-Anweisung, so wie sie der BDE übergeben wird.

- UpdateRecordTypes (*TDataSet*)
- UpdatesPending (*TDataSet*, nur Lesen)

3.4.3 Die Methoden von TQuery

- ~TQuery (*TQuery*)

  ```
  __fastcall virtual ~TQuery(void);
  ```

  Der Destruktor *~TQuery* gibt den Speicher frei, der für die *TQuery*-Instanz reserviert gewesen ist. *~TQuery* sollte nicht direkt aufgerufen werden, statt dessen ist die Methode *Free* zu verwenden.

- ActiveBuffer (*TDataSet*)
- Append (*TDataSet*)
- AppendRecord (*TDataSet*)
- ApplyUpdates (*TDataSet*)
- Cancel (*TDataSet*)
- CancelUpdates (*TDataSet*)
- CheckBrowseMode (*TDataSet*)
- CheckOpen (*TDBDataSet*)
- ClearFields (*TDataSet*)
- Close (*TDataSet*)
- CommitUpdates (*TDataSet*)
- ControlsDisabled (*TDataSet*)
- CursorPosChanged (*TDataSet*)
- Delete (*TDataSet*)
- DisableControls (*TDataSet*)
- Edit (*TDataSet*)
- EnbleControls (*TDataSet*)

- ExecSQL (*TQuery*)

  ```
  void __fastcall ExecSQL(void);
  ```

  Mit der Methode *ExecSQL* werden SQL-Anweisungen zur Datenänderung (INSERT, UPDATE, DELETE) und zur Datendefinition (CREATE TABLE, DELETE TABLE, usw.) ausgeführt. Zum Öffnen von Datenmengen mit SELECT-Anweisungen muß die Methode *Open* verwendet werden (alternativ kann die Eigenschaft *Active* auf *true* gesetzt werden).

- FetchAll (*TDataSet*)
- FieldByName (*TDataSet*)
- FindField (*TDataSet*)
- FindFirst (*TDataSet*)
- FindLast (*TDataSet*)
- FindNext (*TDataSet*)
- FindPrior (*TDataSet*)
- First (*TDataSet*)
- FreeBookmark (*TDataSet*)
- GetBookmark (*TDataSet*)
- GetCurrentRecord (*TDataSet*)
- GetFieldList (*TDataSet*)
- GetFieldNames (*TDataSet*)
- GotoBookmark (*TDataSet*)
- Insert (*TDataSet*)
- InsertRecord (*TDataSet*)
- IsLinkedTo (*TDataSet*)
- Last (*TDataSet*)
- Locate (*TDataSet*)
- Lookup (*TDataSet*)
- MoveBy (*TDataSet*)
- Next (*TDataSet*)
- Open (*TDataSet*)
- ParamByName (*TQuery*)

```
TParam* __fastcall ParamByName(const System::AnsiString Value);
```

Auf die Parameterliste kann nicht nur über die Array-Eigenschaft *Params*, sondern auch mit der Methode *ParamByName* zugegriffen werden. Letztere verwendet nicht den Index, sondern den Namen des Parameters, und ist somit unanfällig für Umstellungen der SQL-Anweisung.

```
SELECT * FROM testadr
  WHERE (vorname = :vor)
    AND (nachname = :nach)
```

Für die oben aufgeführte SQL-Anweisung würde man die Parameter folgendermaßen setzen:

```
Query1->ParamByName("vor")->AsString = "Yvonne";
Query1->ParamByName("nach")->AsString = "Bach";
```

- Post (*TDataSet*)
- Prepare (*TQuery*)

  ```
  void __fastcall Prepare(void);
  ```

  Mit der Methode *Prepare* kann eine Abfrage an die BDE oder an den Server übermittelt werden, damit diese vor der Ausführung optimiert wird.
- Prior (*TDataSet*)
- Refresh (*TDataSet*)

  Anmerkung: Als Methode von *TQuery* aktualisiert *Refresh* die Abfrage nicht. Damit die aktuellsten Daten angezeigt werden, muß die Abfrage geschlossen und wieder geöffnet werden.

  ```
  Query1->Close();
  Query1->Open();
  ```
- Resync (*TDataSet*)
- RevertRecord (*TDataSet*)
- SetFields (*TDataSet*)
- TQuery (*TQuery*)

  ```
  __fastcall virtual TQuery(Classes::TComponent* AOwner);
  ```

  Der Konstruktor *TQuery* erzeugt eine neue *TQuery*-Instanz.
- UnPrepare (*TQuery*)

  ```
  void __fastcall UnPrepare(void);
  ```

  Hebt die Vorbereitung einer Abfrage auf.

- UpdateCursorPos (*TDataSet*)
- UpdateRecord (*TDataSet*)
- UpdateStatus (*TDataSet*)

3.4.4 Die Ereignisse von TQuery

- AfterCancel (*TDataSet*)
- AfterClose (*TDataSet*)
- AfterDelete (*TDataSet*)
- AfterEdit (*TDataSet*)
- AfterInsert (*TDataSet*)
- AfterOpen (*TDataSet*)
- AfterPost (*TDataSet*)
- AfterScroll (*TDataSet*)
- BeforeCancel (*TDataSet*)
- BeforeClose (*TDataSet*)
- BeforeDelete (*TDataSet*)
- BeforeEdit (*TDataSet*)
- BeforeInsert (*TDataSet*)
- BeforeOpen (*TDataSet*)
- BeforePost (*TDataSet*)
- BeforeScroll (*TDataSet*)
- OnCalcFields (*TDataSet*)
- OnDeleteError (*TDataSet*)
- OnEditError (*TDataSet*)
- OnFilterRecord (*TDataSet*)
- OnNewRecord (*TDataSet*)
- OnPostError (*TDataSet*)
- OnServerYield (*TDataSet*)
- OnUpdateError (*TDataSet*)
- OnUpdateRecord (*TDataSet*)

3.5 TTable oder TQuery?

Ob nun *TTable* oder *TQuery* eingesetzt werden soll, hängt vor allem vom beabsichtigten Einsatzzweck ab:

- Soll eine Verbindung mit allen Datensätzen einer einzelnen Tabelle hergestellt werden, dann kann sowohl *TTable* als auch *TQuery* (*SELECT * FROM testadr, RequestLive := true*) verwendet werden; bei der Ausführungszeit habe ich auch keine Unterschiede gemessen, solange die Eigenschaft *RequestLive* gleich *true* ist.
- Sollen die verbundenen Datensätze beschränkt werden, dann ist *TQuery* flexibler.
- Abfragen über mehrere Tabellen sind ohnehin nur mit *TQuery* möglich.
- Bei der Verwendung von Aggregatfunktionen ist *TQuery* vorzuziehen, die meisten Funktionen sind mit *TTable* gar nicht möglich.
- Soll es dem Anwender ermöglicht werden, eigene Abfragen zu erstellen, dann kann mit *TQuery* auf eine standardisierte und verbreitete Abfragesprache zurückgegriffen werden.