Baumschule

Rekursive Stored Procedures mit InterBase: Baumstrukturen effizient bearbeiten und auswerten

von Karsten Strobel

Den Wald vor lauter Bäumen nicht sehen? Dieses Sprichwort wird in der Welt der relationalen Datenbanken bittere Realität, verfügt man nicht über geeignete Hilfsmittel zur Visualisierung solcher Datenstrukturen. Denn in Datenbanktabellen führen Bäume ein schlichtes Dasein als Stapel ordentlich übereinander geschichteter Äste, mit Markierungen für den Zusammenbau. Nicht viel anders als bei der Möbelmontage kommt man ohne eine bebilderte Anleitung nicht sehr weit.

Ob Organigramme, Vertriebsstrukturen, Matrjoschka-Steckpuppen oder Schachteln in Kartons auf Paletten im LKW: Abhängigkeiten dieser Art lassen sich am besten als Baum darstellen. Diese hierarchische Organisation und Visualisierung von Daten aller Art ist sehr beliebt. Vermutlich hat der Windows Explorer mit einem hohen Anteil zur Popularität dieser Darstellungsform beigetragen. Schließlich befindet sich eine Datei immer in genau einem ganz bestimmten Order, und nie hauptsächlich hier, aber ein bisschen auch dort.

Auch die sonst wenig ergiebige Beispieldatenbank Employee.gdb, die zu jeder Standardinstallation des InterBase-Servers gehört, wartet mit einer solchen Baumstrukturierung auf. Abbildung 1 zeigt einen Ausschnitt aus einem mit dem Datenbankdesigner von IBExpert [1] per Reverse Engineering erzeugten Diagramm. Rechts oben sehen Sie die Tabelle DEPARTMENT, in der beispielhaft die Abteilungen eines Unternehmens gespeichert werden. Die Pfeile symbolisieren die als Foreign Keys deklarierten Beziehungen zwischen den Tabellen. Der rot eingefärbte Fremdschlüssel-Pfeil, der im Dreiviertelkreis von DEPARTMENT auf DEPARTMENT zurückverweist, fällt dabei etwas aus dem Rahmen. Hinter dieser sogenannten Selbstreferenzierung verbirgt sich eigentlich schon das ganze Geheimnis einer Baumstruktur.

Quellcode

Den Quellcode finden Sie auf der aktuellen Profi-CD sowie auf unserer Homepage unter
www.derentwickler.de

Schwierige Vaterbeziehung

In Listing 1 sehen Sie die beiden Definitionsanweisungen, mit denen die DEPARTMENT-Tabelle angelegt wurde.

Das CREATE TABLE-Statement erzeugt die Tabelle und legt eine drei Buchstaben lange Abteilungsnummer im Feld DEPT_NO als Hauptschlüssel, also als eindeutiges Kennzeichen jedes einzelnen Datensatzes, fest. Für das Feld MNGR_NO, das die Personalnummer des Abteilungsleiters speichert, wird eine Fermdschlüsselbeziehung zur Tabelle EMPLOYEE deklariert. Kurz gesagt, sorgt der Fremdschlüssel dafür, dass zu einer in MNGR_NO gespeicherten Personalnummer in der Mitarbeitertabelle EMPLOYEE auch garantiert ein passender Personaldatensatz mit identischer Personalnummer im Hauptschlüssel existiert. In einem zweiten Statement wird die gerade erstellte Tabelle mittels ALTER TABLE nochmals modifiziert und es wird ein weiterer Fremdschlüssel hinzugefügt. Dieser erstellt eine feste Beziehung zwischen einem in HEAD_DEPT zu speichernden Abteilungskürzel der übergeordneten Abteilung und dem damit gemeinten Datensatz in der selben Tabelle (Selbstreferenz). Jede Abteilung in unserem virtuellen Unternehmen kann also einer anderen Abteilung untergeordnet sein, die wieder einer weiteren Abteilung untergeordnet sein kann usw. An der Spitze stehen die obersten Abteilungen, bei denen das Feld HEAD_DEPT leer (NULL) bleibt. In den Beispieldaten ist dies nur bei der Abteilung „Corporate Headquarters“ der Fall. Dies ist die Wurzel des Abteilungsbaums. Abbildung 2 zeigt die vorerfassten Beispieldaten dieser Tabelle.

Bei selbstreferenzierenden Tabellen wird der jeweils übergeordnete Daten-

Listing 1

```
CREATE TABLE DEPARTMENT (
 DEPT_NO   CHAR(3) NOT NULL,
 DEPARTMENT VARCHAR(25) NOT NULL,
 HEAD_DEPT  CHAR(3),
 MNGR_NO   SMALLINT,
 BUDGET    NUMERIC(15,2),
 LOCATION  VARCHAR(15),
 PHONE_NO  CHAR(8) DEFAULT '555-1234',

 PRIMARY KEY (DEPT_NO),
 FOREIGN KEY (MNGR_NO) REFERENCES EMPLOYEE
                                  (EMP_NO)
);

ALTER TABLE DEPARTMENT
ADD FOREIGN KEY (HEAD_DEPT) REFERENCES DEPARTMENT
                                  (DEPT_NO);
```

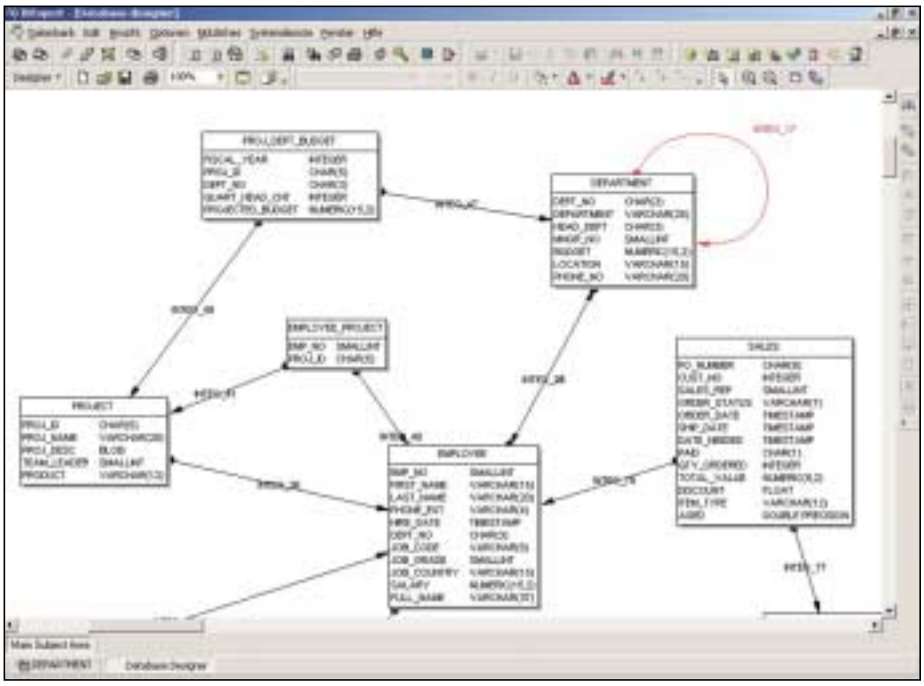

Abb. 1: Struktur der Employee-Datenbank

satz auch „Vater-Datensatz“ genannt, wobei der englische Ausdruck „parent record” gebräuchlicher ist. Diese Bezeichnung weist auf ein anderes Bild hin, das man sich von den baumstrukturierten Daten machen kann, nämlich einen Stammbaum. Das Wurzelelement heißt dann Urvater, dessen Söhne wiederum Söhne haben usw. Jeder einzelne Datensatz merkt sich dabei nur den Namen seines eigenen Vaters. Um den „Großvater“ zu finden, muss man also den Vater nach dessen Vater befragen. Der Urvater ist elternlos. Wie im richtigen Leben gibt es auch hier einige Verwandschaftsbeziehungen, die nicht möglich sind. So darf ein Datensatz nicht auf sich selbst als Vater verweisen. Ferner müssen Ringbeziehungen vermieden werden, d.h. ein Datensatz darf keinen seiner eigenen Nachkommen als Vater ausweisen. Die Einhaltung dieser Regeln muss man organisatorisch sicherstellen, indem man vermeidet, derlei Unsinn abzuspeichern. In der Employee-Datenbank wird man allerdings nicht am Abspeichern unzulässiger Beziehungen gehindert. Datenbankseitig ließe sich die Einhaltung der Beziehungsregeln zusätzlich durch die Programmierung geeigneter Trigger sicherstellen, die bei Zuwiderhandlung eine Exception auslösen. Unsinnige Daten können bei den noch zu zeigenden Prozeduren zu Endlosrekursionen – also zum Programmabsturz – führen.

DEPT_NO	DEPARTMENT	HEAD_DEPT	MNGR_NO	BUDGET	LOCATION	PHONE_NO
000	Corporate Headquarters		105	1.000.000,00	Monterey	(408) 555-1234
100	Sales and Marketing	000	85	2.000.000,00	San Francisco	(415) 555-1234
110	Pacific Rim Headquarters	100	34	600.000,00	Kuaui	(808) 555-1234
115	Field Office: Japan	110	118	500.000,00	Tokyo	3 5350 0901
116	Field Office: Singapore	110		300.000,00	Singapore	3 55 1234
120	European Headquarters	100	36	700.000,00	London	71 235-4400
121	Field Office: Switzerland	120	141	500.000,00	Zurich	1 211 7767
123	Field Office: France	120	134	400.000,00	Cannes	58 68 11 12
125	Field Office: Italy	120	121	400.000,00	Milan	2 430 39 39
130	Field Office: East Coast	100	11	500.000,00	Boston	(617) 555-1234
140	Field Office: Canada	100	72	500.000,00	Toronto	(416) 677-1000
100	Marketing	100		1.500.000,00	San Francisco	(415) 555-1234
600	Engineering	000	2	1.100.000,00	Monterey	(408) 555-1234
620	Software Products Div.	600		1.200.000,00	Monterey	(408) 555-1234
621	Software Development	620		400.000,00	Monterey	(408) 555-1234
622	Quality Assurance	620	9	300.000,00	Monterey	(408) 555-1234
623	Customer Support	620	15	650.000,00	Monterey	(408) 555-1234
670	Consumer Electronics Div.	600	107	1.150.000,00	Burlington, VT	(802) 555-1234
671	Research and Development	670	20	460.000,00	Burlington, VT	(802) 555-1234
672	Customer Services	670	94	850.000,00	Burlington, VT	(802) 555-1234
900	Finance	000	46	400.000,00	Monterey	(408) 555-1234

Abb. 2: Beispieldaten aus DEPARTMENT

Nützlicher Zentralismus

Stored Procedures sind Anweisungssequenzen in Form benannter Prozeduren, die im Datenbankserver gespeichert werden und vom Server ausgeführt werden können. Sie gehören zu den Metadaten, die im Gegensatz zu den Anwendungsdaten einer Datenbank als strukturelle Inhalte anzusehen sind. Bei InterBase gibt es zwei Kategorien solcher Prozeduren, nämlich ausführbare Prozeduren und selektierbare Prozeduren. Erstere werden mit dem SQL-Kommando EXECUTE PROCEDURE gestartet, Letztere können mit SELECT angesprochen werden. Selektierbare Prozeduren werden natürlich auch ausgeführt; die Bezeichnung „ausführbare Prozeduren“ hat nur etwas mit dem Namen des Ausführungskommandos zu tun. Mit Hilfe von Prozeduren lassen sich viele Aufgaben an den Server delegieren, die sonst das Anwendungsprogramm auf der Client-Seite erfüllen müsste. Dieser Ansatz hat viele Vorteile, die zwei wichtigsten davon sind: Das Volumen der Datenübertragung zwischen Client und Server lässt sich erheblich reduzieren und verschiedene Anwendungen

Listing 2

```
CREATE PROCEDURE LIST_DEPT_TREE (AROOT CHAR(3),
                                 AROOT_LEVEL INTEGER)
RETURNS (RDEPT_NO CHAR(3), RLEVEL INTEGER)
AS
BEGIN
 SELECT DEPT_NO FROM DEPARTMENT WHERE DEPT_
                              NO = :AROOT INTO :RDEPT_NO;

 IF (RDEPT_NO IS NOT NULL) THEN
 BEGIN
  RLEVEL = AROOT_LEVEL;
  SUSPEND;
 END
END
```

können sich die Funktionalität solcher zentral gespeicherten Prozeduren teilen.

Listing 2 enthält eine sehr einfache, selektierbare Prozedur, die ich später noch weiter ausbauen möchte. Eine Prozedur wird mit CREATE PROCEDURE <name_der_prozedur> angelegt, und wie Sie sehen, verfügt sie über eine Liste von Ein- und Ausgabeparametern. Danach folgt das Schlüsselwort DO und ein BEGIN END-Block mit Anweisungen. Die Stored Procedure-Sprache erlaubt die DML-Anweisungen von SQL (SELECT, INSERT, UPDATE, DELETE) und eine ganze Reihe weiterer Elemente, die in normalem SQL nicht verfügbar sind, wie IF-Statements, WHILE- und FOR-Schleifen, lokale Variablen etc..

Um einen unschönen Sprachenmix zu vermeiden, wurden die Bezeichner von Prozedur und Parametern in Englisch gehalten. Sie können diese Prozedur in die Employee-Datenbank einfügen und mit einem SELECT-Befehl aufrufen:

```
SELECT * FROM LIST_DEPT_TREE('000', 1)

RDEPT_NO  RLEVEL
000    1
```

Zugegeben, das Ergebnis ist noch nicht sonderlich aufregend. Die Prozedur selektiert lediglich den Datensatz DEPARTMENT-Tabelle, bei dem die DEPT_NR mit dem Eingabeparameter ADEPT_NR übereinstimmt. Wie schon erwähnt, stehen Ihnen innerhalb von Stored Procedures (und auch Triggers) zu den normalen SQL-Befehlen einige Spracherweiterungen zur Verfügung. Von einer davon macht diese einfach Prozedur bereits Gebrauch: Sie verwendet eine INTO-Klausel, um das Selektionsergebnis einer Variablen zuzuweisen. Um innerhalb einer SQL-Anweisung Variablen anzusprechen, muss man deren Bezeichnern einen Doppelpunkt voranstellen, damit sie nicht mit evtl. gleichnamigen Tabellenfeldern verwechselt werden können. Das Ergebnis der Selektion ist nichts anderes als der gesuchte DEPT_NR-Wert, der mittels INTO an den Rückgabeparameter RDEPT_NR weitergegeben wird. Alle Rückgabeparameter und alle lokalen Variablen sind beim Eintritt in eine Prozedur automatisch mit NULL initialisiert. Wenn nach der SELECT..INTO-Anweisung mit „IF (RDEPT_NO IS NOT NULL)“ der NULL-Status von RDEPT_NR abgeprüft wird, dann wird diese Bedingung also nur erfüllt, falls ein Datensatz mit der gesuchten ADEPT_NR gefunden wurde. Ist dies der Fall, wird der Eingabeparameter AROOT_LEVEL an den Ausgabeparameter RLEVEL zugewiesen und anschließend SUSPEND aufgerufen. Dieser Befehl ist nur bei selektierbaren Prozeduren von Bedeutung, oder – besser gesagt – er macht eine Prozedur zu einer solchen.

Die Besonderheit der Ausführung einer Prozedur mit SELECT besteht darin, dass sie eine tabellarische Ergebnismenge beliebiger Länge zurückliefern kann. Je nachdem, wie oft innerhalb der Prozedur SUSPEND aufgerufen wird, variiert die Anzahl der vom Aufrufer empfangenen Datensätze, denn jedes SUSPEND schickt den zu diesem Zeitpunkt auf die Ausgabeparameter geladenen Werte als einen Ergebnisdatensatz an den Aufrufer zurück. Versuchen Sie mal, eine zweite SUSPEND-Zeile einzufügen. Ändern Sie dazu einfach den Quellcode der Prozedur ab und tragen Sie sie mit ALTER PROCEDURE (statt CREATE PROCEDURE) in die Datenbank ein. Beim erneuten Aufrufen mit der oben gezeigten SELECT-Syntax erhalten Sie dann zwei identische Ergebniszeilen anstatt einer. SUSPEND unterbricht die Ausführung der Prozedur mit der Rückgabe einer Ergebniszeile an den Aufrufer so lange, bis dieser bereit ist, den nächsten Ergebnisdatensatz entgegenzunehmen.

Selbstverliebte Prozeduren

Listing 3 stellt die erweiterte Prozedur dar, die nicht nur den mit AROOT angegebenen Datensatz liest und zurückgibt, sondern sich auch um die Auflistung der untergeordneten Datensätze kümmert – also jene, die den zuerst angesprochenen Datensatz als HEAD_DEPT referenzieren. Da die Prozedur bereits existiert, muss sie nun geändert werden. Das Kommando dafür lautet ALTER PROCEDURE. Nach dem SUSPEND-Befehl der ersten Version folgt jetzt ein SELECT-Statement, das in eine FOR-Schleife eingefasst ist. FOR-Schleifen gehören auch zu

den speziellen Sprachelementen der Stored Procedures (und Triggers). Anders als bei Schleifen anderer Programmiersprachen wird hier keine Laufvariable mit Start- und Endwert definiert. Stattdessen bestimmt das an das FOR-Schlüsselwort anfügte SELECT-Statement die Häufigkeit der Wiederholung jenes Anweisungsblocks, der auf das DO-Schlüsselwort nach dem SELECT-Ausdruck folgt. Für jeden Datensatz (FOR), den die SELECT-Anweisung ermittelt und deren Spalten an die hinter INTO aufgelisteten Variablen übergeben werden, wird der DO-Block einmal ausgeführt.

In unserem Beispiel haben wir es mit einer geschachtelten FOR-Schleife zu tun. Die äußere Schleife ermittelt alle Abteilungen, die die AROOT-Abteilung als HEAD_DEPT kennen, ihr also direkt untergeordnet sind. Die jeweilige Abteilungsnummer DEPT_NO dieser Unterabteilungen wird – eine nach der anderen – an die lokale Variable LSUB_DEPT übergeben. Der dann folgende Anweisungsblock enthält die innere Schleife. Wieder wird mit FOR SELECT eine Anzahl von Datensätzen gelesen und an Variablen übergeben. Diesmal sind die Zielvariablen die Rückgabeparameter der Prozedur. Im unteren DO-Block wird für jede einzelne Kombination von Rückgabeparametern einmal SUSPEND aufgerufen, wodurch Zeile für Zeile an den Aufrufer der Prozedur übergeben wird.

Die SELECT-Anweisung der inneren FOR-Schleife ist von besonderem Interesse. Sie liest keine normale Tabelle aus, sondern bezieht ihre Ergebnismenge von derselben Stored Procedure. Hier wird also ein rekursiver Aufruf – ein Selbstaufruf – ausgeführt. Was bewirkt das? Die Prozedur wird für jede der Root-Ebene untergeordnete Abteilungsebene einmal aufgerufen, und verrichtet die selbe Arbeit wieder, aber mit anderen Parametern. Beim erneuten Eintritt versteht die Prozedur also unter AROOT etwas anderes und liefert die Daten zu dieser, sowie – durch erneuten rekursiven Aufruf – zu ihren direkten Unterabteilungen, an den Aufrufer zurück. Der Aufrufer ist im Falle einer rekursiven Ausführung aber nicht mehr der Anwender, sondern die aufrufende Prozedur. Deshalb wird das Ergebnis des SELECT .. FROM LIST_DEPT_TREE im DO-Block auf SUSPEND an den nächsten Aufrufer weitergeleitet. Mit jeder neuen Rekursions-Schachtelung erhöht sich außerdem AROOT_LEVEL. Die Rekursionstiefe entspricht der Schachtelungstiefe des Baums, und RLEVEL gibt genau diese Information zurück. Der Baum wird durch diesen Trick quasi aufgefächert.

Listing 4 zeigt das Ergebnis dieser rekursiven Klettertour durch den Datenbaum. Der erste SELECT gibt als Ursprungsabteilung ‚000' an. Um die Arbeitsweise zu verdeutlichen, wird im zweiten SELECT einfach ‚670' als Ursprungsabteilung vorgegeben. Dieser zweite Versuch liefert nur den Teilbaum ab diesem Startpunkt zurück.

Es ist aber zu beachten: Wenn innerhalb einer Prozedur wiederum eine Prozedur aufgerufen wird, dann muss die aufgerufene Prozedur bereits deklariert sein. Für die Programmierung einer Rekursion bedeutet dies, dass Sie die Prozedur zunächst ohne rekursiven Aufruf erzeugen, und dann mit ALTER PROCEDURE den Selbstaufruf hinzufügen müssen. Sie können z.B. den Prozedurkopf mit Ein- und Ausgabeparametern, aber mit leerem Prozedurkörper anlegen, wobei Sie zwischen BEGIN und END nur EXIT; kodieren, und dann den gesamten Programmtext einschließlich Rekursion per Alterierungskommando nachschieben.

Die von LIST_DEPT_TREE ausgegebene Baumstruktur ist zwar schon etwas leichter lesbar als die simple Datenliste aus Abbildung 2, aber man wünscht sich natürlich noch mehr Komfort als eine bloße Textliste dieser Daten. Bei GUI-Anwendungen sind grafische Baumansichten (Treeviews) gängig. Mit Unterstützung durch die Stored Procedure, die bereits eine Voranalyse des Baums liefert, kann man in einer Delphi-Anwendung eine sol-

Listing 3

```
ALTER PROCEDURE LIST_DEPT_TREE (AROOT CHAR(3),
                                AROOT_LEVEL INTEGER)
RETURNS (RDEPT_NO CHAR(3), RLEVEL INTEGER)
AS
DECLARE VARIABLE LSUB_DEPT CHAR(3);
BEGIN
 SELECT DEPT_NO FROM DEPARTMENT WHERE DEPT_
                              NO = :AROOT INTO :RDEPT_NO;

 IF (RDEPT_NO IS NOT NULL) THEN
 BEGIN
  RLEVEL = AROOT_LEVEL;
  SUSPEND;
  FOR
   SELECT DEPT_NO FROM DEPARTMENT
   WHERE HEAD_DEPT = :AROOT
   ORDER BY DEPT_NO
   INTO :LSUB_DEPT
  DO
  BEGIN
   FOR
    SELECT RDEPT_NO, RLEVEL FROM
LIST_DEPT_TREE(:LSUB_DEPT, :AROOT_LEVEL+1)
    INTO :RDEPT_NO, :RLEVEL
   DO
    SUSPEND;
  END
 END
END
```

Listing 4

```
select * from LIST_DEPT_TREE('000', 1);

RDEPT_NO  RLEVEL
  000     1
  100     2
  110     3
  115     4
  116     4
  120     3
  121     4
  123     4
  125     4
  130     3
  140     3
  180     3
  600     2
  620     3
  621     4
  622     4
  623     4
  670     3
  671     4
  672     4
  900     2

select * from LIST_DEPT_TREE('670', 1);

RDEPT_NO  RLEVEL
  670     1
  671     2
  672     2
```

che Visualisierung ziemlich einfach programmieren. Als Vorarbeit müssen wir aber noch zwei kleine Probleme lösen: Erstens ist es nicht ganz praxisgerecht, dass beim Aufruf der Prozedur die Nummer der obersten Abteilung bekannt sein muss, und es könnten ja auch mehrere „Kopfabteilungen“ existieren. Zweitens brauchen wir zu jeder Abteilung außer der von der Prozedur abgelieferten Abteilungsnummer und der Schachtelungstiefe, noch ein paar Extra-Infos. Mindestens den Abteilungsnamen möchte man noch sehen.

Kombinationskünste

Um diese zwei Probleme zu lösen, kann man sich die Tatsache zu Nutze machen, dass eine selektierbare Stored Procedure in einer SELECT Anweisung mit anderen Tabellen verknüpft werden kann. Man kann also zum Beispiel die JOIN-Syntax von SELECT auch auf Prozeduren - und sogar im Mix mit normalen Tabellen – anwenden. Die Mächtigkeit dieses Merkmals wird deutlich, wenn Sie die folgende SELECT Anweisung ausprobieren:

```
SELECT p.*, d2.DEPARTMENT FROM DEPARTMENT d
LEFT JOIN LIST_DEPT_TREE(d.DEPT_NO, 1) p ON (1=1)
LEFT JOIN DEPARTMENT d2 ON (d2.DEPT_NO=p.RDEPT_NO)
WHERE d.HEAD_DEPT IS NULL
```

Hier wird die Tabelle DEPARTMENT (Alias “d”), aus der alle Kopfabteilungen selektiert werden (WHERE d.HEAD_DEPT IS NULL), mit unserer gespeicherten Prozedur per LEFT JOIN verknüpft, wobei der Prozedur als Parameter die Abteilungsnummer jeder Kopfabteilung (d.DEPT_NO) und der Startwert für die Anzeige der Verschachtelungsebene (1) übergeben wird. Da die Prozedur anhand ihres Eingangsparameters AROOT immer nur die zutreffenden Datensätze liefert, kann man auf ein JOIN-Kriterium praktisch verzichten. Mit „ON (1=1)“ wird nur die Syntaxvorschrift der JOIN-Klausel erfüllt. Soweit Teil Eins der Aufgabe. Das zweite Problem, also das Hinzuladen der jeweiligen Abteilungsbezeichnung, wird durch die zweite JOIN-Klausel, mit der noch einmal DEPARTMENT (Alias „d2“) angesprochen wird, gelöst. Hier wird zu jeder von der Prozedur aufge-

Listing 5

```
SELECT p.*, d2.DEPARTMENT FROM DEPARTMENT d
LEFT JOIN LIST_DEPT_TREE(d.DEPT_NO, 1) p ON (1=1)
LEFT JOIN DEPARTMENT d2 ON (d2.DEPT_NO=p.RDEPT_NO)
WHERE d.HEAD_DEPT IS NULL;

RDEPT_NO  RLEVEL  DEPARTMENT
000       1       Corporate Headquarters
100       2       Sales and Marketing
110       3       Pacific Rim Headquarters
115       4       Field Office: Japan
116       4       Field Office: Singapore
120       3       European Headquarters
121       4       Field Office: Switzerland
123       4       Field Office: France
125       4       Field Office: Italy
130       3       Field Office: East Coast
140       3       Field Office: Canada
180       3       Marketing
600       2       Engineering
620       3       Software Products Div.
621       4       Software Development
622       4       Quality Assurance
623       4       Customer Support
670       3       Consumer Electronics Div.
671       4       Research and Development
672       4       Customer Services
900       2       Finance
```

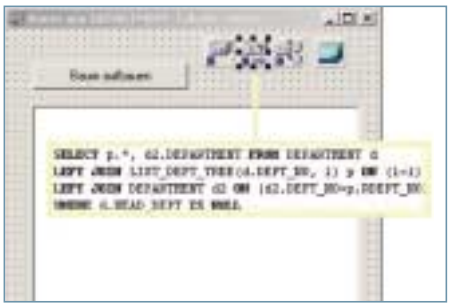

Abb. 3: Einfacher TreeView Dialog in der Delphi-IDE

zählten Abteilung der zugehörige Datensatz (d2.DEPT_NO=p.RDEPT_NO) verknüpft. Als Ergebnisspalten werden die Rückgabewerte der Prozedur (p.*) und der Abteilungsname (d2.DEPARTMENT) ausgegeben. Das Ergebnis dieses SELECT sehen Sie in Listing 5.

Als Bordmittel liefert Delphi die *TTreeView*-Komponente, mit der sich Baumstrukturen sehr einfach darstellen lassen. Leider hat *TTreeView* kein datensensitives Pendant. Sie können an so eine Komponente also nicht einfach eine Datenquelle anschließen und die Aufbereitung automatisch geschehen lassen, sondern müssen die Baumstruktur programmatisch erzeugen. Der schwierigste Teil dieser Aufgabe besteht darin, den Baum anhand der Daten korrekt zuzuschneiden, also die einzelnen Knotenpunkte (die Nodes, die der TreeView zeigen soll), hierarchisch zu ordnen. Um dies zu lösen, könnten Sie eine rekursive Funktion in Delphi programmieren, die alle Einträge der Tabelle DEPARTMENT durchforstet und den TreeView mit geordneten Nodes versorgt. Oder wir überlassen diese Aufgabe dem Datenbankserver. Genau hier kommt die schon fertige Prozedur LIST_DEPT_TREE, die für diesen Zweck bereits quasi mundgerecht zugeschnittene Daten liefert, wieder ins Spiel.

Listing 6

```
procedure TForm1.btnBaumAufbauenClick
                                    (Sender: TObject);
var
 i, level, lastlevel: Integer;
 node: TTreeNode;
 nodename: string;
begin
 IBDatabase1.Open;

 TreeView1.Items.Clear;
 node := nil;
 lastlevel := 0;

 with IBQuery1 do
 begin
  Open;
  while not EOF do
  begin
   level := FieldByName('RLEVEL').AsInteger;
   nodename := FieldByName('RDEPT_NO').AsString + '-' +
        FieldByName('DEPARTMENT').AsString;

   for i := level to lastlevel do
    node := node.Parent;

   node := TreeView1.Items.AddChild(node, nodename);
   lastlevel := level;
   Next;
  end;
  Close;
 end;

 TreeView1.FullExpand;
end;
```

Das zuvor eingearbeitete SELECT-Statement gibt den Abteilungsbaum (ohne Kenntnis der Kopfabteilungs-Namen und einschließlich Bezeichnung) aus. Dieser Befehl wird in einer Delphi-Anwendung einer TIBQuery Komponente in der SQL-Eigenschaft zugewiesen (Abbildung 3). Beim Drücken des Buttons soll diese Datenmenge geöffnet und der Baum grafisch aufgebaut werden. Listing 6 zeigt die fertige Lösung.

Nach dem Öffnen der Datenmenge wird in einer Schleife für jeden Datensatz mit *AddNode* ein Knoten eingefügt. *AddNode* erwartet zwei Parameter, nämlich einen Zeiger auf den übergeordneten (Vater-) Knoten und einen String mit dem Titel des neuen Knotens – in unserem Fall ist das die Abteilungsbezeichnung aus der Spalte DEPARTMENT. Für die erste Abteilung ist *Node* per Initialisierung *nil*. Dadurch trägt *AddNode* den ersten Knoten als Ursprungsknoten, also als Abteilung ohne Überabteilung, ein. *AddNode* gibt einen Zeiger auf das neu eingefügte *TTreeNode*-Objekt zurück. Ab dem zweiten Schleifendurchlauf erhalten *Node* und *LastLevel* immer den Knoten und die Schachtelungsebene des vorherigen Datensatzes. Immer wenn der Level kleiner wird, also der neue Knoten nicht unterhalb des letzten Knotens einzuordnen ist, muss aus der Parent-Eigenschaft von *Node* der richtige Vater ermittelt werden. Die Differenz zwischen altem und neuem Level gibt die Anzahl dieser Rückschritte im Generationenbaum an. Das Ergebnis ist in Abbildung 4 zu sehen.

Abb. 4: DEPARTMENT als grafischer Baum

Fazit

Die serverseitige Verarbeitung von baumartig strukturierten Daten mit rekursiven *Stored Procedures* bietet sich nicht nur für die Visualisierung von Daten an. Tatsächlich ist die reine Anzeige eine Aufgabe, die man unter Umständen auch clientseitig gut lösen kann. Hervorragende Unterstützung bietet hier übrigens die *ExpressQuantumDBTreeList*-Komponente von Developer Express [2].

Für die Bewertung und Bearbeitung solcher Daten sind *Stored Procedure* aber praktisch unschlagbar. Sie können rekursive Operationen, die vielleicht nur eine einzelne Ergebniszeile liefern (aber sehr viele Baumebenen betrachten müssen), vom Server, der den schnellsten und effektivsten Zugang zu diesen Daten hat, erledigen lassen. Ein weiteres Beispiel für diese Technik finden Sie übrigens in der Prozedur DEPT_BUDGET in der Employee-Demodatenbank.

Literatur & Links

[1] www.ibexpert.com/
[2] www.devexpress.com/