Datenbankprogrammierung mit InterBase

PEARSON Education

ADDISON-WESLEY

Michael Ebner

Datenbank-Programmierung mit InterBase

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie;
detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

5 4 3 2 1

05 04 03

ISBN 3-8273-1684-7

ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Einbandgestaltung: Hommer Design, Haar bei München
Lektorat: Martin Asbach, masbach@pearson.de
Korrektorat: Alexandra Müller, Oer-Erkenschwick
Herstellung: Anna Plenk, aplenk@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker, Kevelaer
Printed in Germany

Inhaltsverzeichnis

Vorwort

Bei Datenbanksystemen denkt man für gewöhnlich an Oracle, DB2 und MS SQL. InterBase ist nach wie vor eine Art »Geheimtipp«: Geringer Ressourcen-Verbrauch, einfache Wartung, hohe Leistungsfähigkeit, geringe Kosten (mit der Version 6.0 und Firebird gibt es sogar zwei Open-Source-Ableger) und vor allem eine gute Unterstützung der Borland-Entwicklungswerkzeuge lassen InterBase zu einer Alternative werden, über die man nachdenken sollte.

Natürlich kann man auch bei InterBase viel falsch machen. Um Sie davor zu bewahren, wurde dieses Buch geschrieben.

Im ersten Teil streifen wir etwas die Datenbank-Theorie und besprechen ausführlich den von InterBase verwendeten SQL-Dialekt. Dieselbe Aufgabenstellung lässt sich meist mit verschiedenen SQL-Anweisungen lösen – wir wollen uns insbesondere in Kapitel 3 ansehen, welche Varianten die schnelleren sind.

Der zweite Teil behandelt die Besonderheiten von InterBase: Wie wird der Server eingerichtet, wie arbeitet man mit dem Administrationstool *IBConsole*, was wird in welchen Systemtabellen gespeichert.

InterBase wird meist dort verwendet, wo die Borland-Entwicklungswerkzeuge Delphi, Kylix und C++-Builder verwendet werden. Im dritten Teil dieses Buches wollen wir hier fünf Komponentensammlungen (IBX, IBO, FIBPlus, dbExpress und BDE) ansehen und deren Verhalten im Zugriff auf InterBase-Datenbanken vergleichen.

Zuletzt wollen wir noch USER DEFINED FUNCTIONS besprechen sowie das Third-Party-Tool *IBExpert*.

Berlin, den 18. Februar 2003

Michael Ebner

info@tabu-datentechnik.de

1 Einführung

InterBase ist ein Multiplattform-Client-Server-Datenbanksystem. Wir wollen uns zunächst mit der Frage beschäftigen, was überhaupt Datenbanken sind und was man unter Normalisierung versteht. Anschließend möchte ich eine Übersicht über die Besonderheiten von InterBase geben. Zuletzt werfen wir noch einen kleinen Blick auf die Installation.

1.1 Was sind Datenbanken?

Eine Datenbank ist eine Sammlung von nicht-redundanten Daten, die von mehreren Applikationen benutzt werden. So definiert es D.R. Howe *in Data Analysis for Data Base Design.*

Legen wir diesen Satz mal nicht auf die Goldwaage: Beim Thema nicht-redundant ist viel Wunschdenken dabei und oft genug greift auch nur eine einzelne Applikation auf eine Datenbank zu. Aber als Zielvorstellung ist diese Definition ganz brauchbar:

- die Datenbank als Sammlung von Daten
- mit der Zielsetzung, eine bestimmte Information nur ein einziges Mal zu speichern
- und der Möglichkeit, immer dann auf diese Datenbank zurückgreifen zu können, wenn diese Information benötigt wird

Nach dieser Definition ist eine Datenbank ein Datenbestand. Da die Informatik jedoch mehr industriell als wissenschaftlich geprägt ist, nimmt man es mit Definitionen nicht ganz so genau. So wird der Begriff *Datenbank* auch gerne für ein Datenbanksystem wie beispielsweise InterBase verwendet.

Versuchen wir hier eine Unterscheidung:

Datenbanksystem

Ein Datenbanksystem ist der Mittler zwischen Datenbestand und Anwendung.

Möchte eine Anwendung bestimmte Daten haben, dann schickt sie eine Anfrage an das DBS, beispielsweise mit dem SQL-Befehl SELECT. Das DBS sucht sich die Daten auf der Festplatte zusammen und schickt die Ergebnismenge zur Anwendung.

Ein solches Datenbanksystem ist beispielsweise InterBase, andere Produkte sind Oracle, DB2, Informix und noch viele andere.

Datenbank-Management-System

Der Begriff Datenbank-Management-System wird meist synonym für Datenbanksystem verwendet. Es kann aber auch das Programm sein, welches Zusatzaufgaben wie beispielsweise das Backup der Daten erledigt.

Datenbestand

Der Datenbestand umfasst die Daten, welche in die Datenbank geschrieben wurden. In der Regel werden diejenigen Daten, die das DBS für die Selbstverwaltung auf die Platten schreibt, nicht dazugezählt.

Wie gesagt, eine Datenbank ist nach Mehrheitsmeinung ein Datenbestand, der Begriff wird aber auch für Datenbanksysteme verwendet. Was nun genau gemeint ist, wird meist aus dem Kontext verständlich. Wenn man Missverständnisse ausschließen möchte, dann vermeidet man den Begriff gänzlich und spricht nur von Datenbeständen und Datenbanksystemen.

1.1.1 Historisches

Die ersten beiden Generationen von Datenbanken (wenn man diese schon so nennen möchte) waren so genannte File-Systeme (die erste Generation auf Band, die zweite auf Platte). In diesen File-Systemen wurden die Datensätze nacheinander abgespeichert.

Damit konnte man beispielsweise Adressen speichern und auch wieder zurückerhalten, aber bei allem, was darüber hinausging, fingen die Probleme an. Wenn man einen bestimmten Datensatz suchen wollte, dann konnte man nur alle Datensätze auslesen und vergleichen, ob der jeweilige Datensatz den gestellten Bedingungen entsprach.

Sequenzielle Suche

Bei den Systemen der ersten Generation war dabei noch nicht einmal ein so genannter wahlfreier Zugriff möglich: Wollte man den 365. Datensatz auslesen, dann wurde das Band bis zur Dateianfangsmarke (BOF, *begin of file*) zurückgespult und dann Datensatz für Datensatz ausgelesen, bis man den 365. erreicht hatte. Bei den Systemen der zweiten Generation hatte man dann wenigstens Festplatten; auf den gewünschten Datensatz konnte man hier (mehr oder minder) direkt zugreifen. Bei der Suche nach bestimmten Kriterien war man dann aber immer noch auf die sequenzielle Suche angewiesen (dies ist man häufig auch heute noch).

Redundanz, Inkonsistenz und Integrität

Bei diesen Systemen machten unter anderem Redundanz und Inkonsistenz sowie Integritätsprobleme Sorgen. Nehmen wir als Beispiel die Auftragsverwaltung eines Versandhauses, welche wir zu diesem Zweck sehr grob vereinfachen wollen, und zwar zu einer Kunden- und einer Auftragsdatei.

Zur Auftragsdatei gehören lauter Datensätze über laufende oder abgeschlossene Aufträge; ein Datensatz enthält häufig Angaben über Bestelldatum, Anzahl, Bestellnummer, Bezeichnung, Einzel- und Gesamtpreis der gelieferten Waren und natürlich über den Kunden. Hier gibt es nun prinzipiell zwei Möglichkeiten:

- Die eine ist, dass sämtliche Kundendaten aus der Kundendatei in die Auftragsdatei kopiert werden. Ein und dieselbe Adresse ist also doppelt vorhanden, man spricht hier von *Redundanz*. So etwas vermehrt nicht nur den Bedarf an Speicherplatz, es führt auch zur *Inkonsistenz*, wenn an nur einem Datensatz Änderungen durchgeführt werden. Nehmen wir einmal an, der Kunde zieht um, meldet dies der Firma und diese ändert entsprechend die Kundendatei.

  Nun hat der Kunde aber in der Umzugshektik vergessen, die Rechnung aus der letzten Lieferung zu begleichen. Die Buchhaltung untersucht, ob alle Rechnungen beglichen sind, findet den Vorgang und schickt an die alte Adresse eine Mahnung (welche natürlich zurückkommt, weil der Kunde an den Nachsendeauftrag auch nicht gedacht hat). Die Buchhaltung ist nun auch nicht »blöd« und schaut in der Kundendatei beispielsweise unter *Stefan Meier* nach, den es vielleicht siebenmal gibt. Ohne Kundennummer hat man nun ein Problem.

  Eine Variation der Geschichte: Zusammen mit der neuen Adresse wurde eine neue Kundennummer vergeben, weil man daraus beispielsweise die Filiale erkennen soll, welche den Kunden zu betreuen hat.

- Die andere Möglichkeit ist, dass man in der Auftragsdatei nur die Kundennummer speichert und sich der Rechner bei Bedarf einfach die nötigen Adressdaten aus der Kundendatei herausholt. Redundanz wird somit (in diesem Punkt) vermieden, bei den heutigen relationalen Datenbanken macht man das im Prinzip auch nicht anders.

  Nun bittet beispielsweise *Stefan Meier* darum, in Zukunft keinen Katalog mehr zu erhalten, die Adresse wird aus der Kundendatei gelöscht. Wenn die Buchhaltung nun eine Mahnung adressieren möchte, dann hat sie nur die Kundennummer – und somit auch ein Problem.

Prinzipiell wäre es möglich, die Anwendungsprogramme so zu erstellen, dass diese Probleme erkannt und vermieden werden. Nun ist es allerdings häufiger der Fall, dass für ein und denselben Datenbestand immer wieder neue Anwendungsprogramme verwendet werden. In diese jedes Mal von neuem die erforderlichen Sicherungen einzufügen ist unökonomisch (und dazu fehleranfällig). Es hat sich deshalb durchgesetzt, dass die Anwendungsprogramme nicht direkt auf den

Datenbestand zugreifen, sondern über ein spezielles, für den Anwender »unsichtbares« Programm, das (unter anderem) diese Sicherheitsmaßnahmen durchführt. Dieses Programm nennt man *Datenbanksystem (DBS)*.

Hierarchische und Netzwerk-Datenbanken

Durch diese Trennung von Anwendungs- und Datenverwaltungsprogramm entstanden die Datenbanken der dritten Generation (von manchen werden sie auch die »ersten echten Datenbanken« genannt). Vertreter dieser Spezies sind beispielsweise die hierarchischen Datenbanken oder die Netzwerk-Datenbanken. Solche Datenbanken sind stellenweise noch auf Großrechnern im Einsatz, werden aber heutzutage bei Neuprogrammierungen nicht mehr verwendet. Da sich SQL erst mit den relationalen Datenbanken durchgesetzt hat, werden Sie mit diesem Standard dort nicht viel anfangen können.

1.1.2 Desktop- oder Client-Server-Datenbank

Der gerade verwendete Begriff *Netzwerk-Datenbank* bezieht sich nicht darauf, dass die Datenbank über ein Netzwerk von mehreren Anwendern gleichzeitig genutzt werden kann – Großrechnersysteme haben immer eine Client-Server-Architektur. Lassen Sie uns auch diese Begriffe klären.

Stand-Alone-Datenbank

Am wenigsten Kopfzerbrechen macht eine *Stand-Alone*-Datenbank, welche zu den *Desktop-Datenbanken* gezählt wird. Die Daten befinden sich auf einem Arbeitsplatzrechner, auf die Daten kann immer nur ein Anwender mit immer nur einer Anwendung zugreifen. Es ist zwar möglich, dass über ein Netzwerk auch Anwender B auf die Daten zugreift, aber nur dann, wenn Anwender A seine Applikation geschlossen hat. Probleme, die dadurch entstehen, dass zwei Anwender zur selben Zeit am selben Datensatz etwas ändern wollen, können schon prinzipiell nicht auftreten; bei jeder größeren Datenbank wird aber der eine Arbeitsplatz zum Nadelöhr.

File-Share-Datenbank

Moderne Netzwerke bieten die Möglichkeit, dass mehrere Anwender auf ein und dieselbe Datei zugreifen. Auf diese Weise ist es auch möglich, dass mit zwei Datenbankanwendungen auf dieselbe Datenbankdatei zugegriffen wird. Diese Version der *Desktop*-Datenbank nennt man *File-Share*-Datenbank und damit ist schon ein echter Multi-User-Betrieb möglich.

Das Ganze hat jedoch (unter anderem) einen entscheidenden Nachteil: Die Datenverarbeitung erfolgt auf den Arbeitsplatzrechnern; für Abfragen muss jeweils der ganze Datenbestand (der jeweiligen Tabellen) zum Arbeitsplatzrechner transferiert werden, dementsprechend hoch ist die Belastung (und entsprechend niedrig die Performance) des Netzwerks.

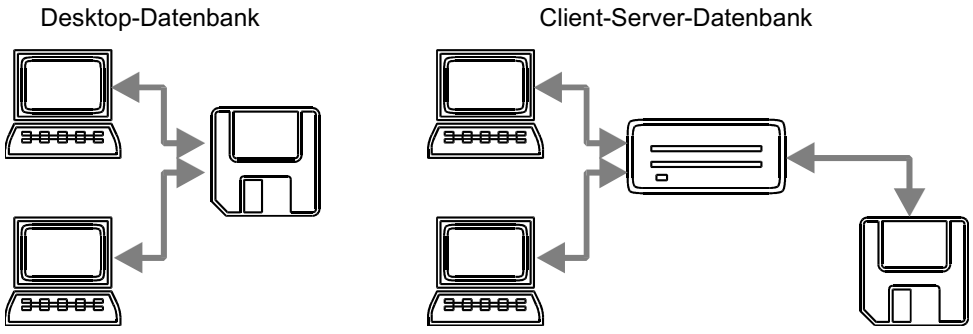


Abbildung 1.1: Unterschied zwischen Desktop- und Client-Server-Datenbank

Client-Server-Datenbank

Einen anderen Ansatz verfolgen *Client-Server*-Datenbanken: Zugriff auf die Dateien des Datenbestandes hat dort nur der Datenbank-Server (nicht zu verwechseln mit dem File-Server eines Netzwerks!), der die Arbeitsplatzrechner bedient. Anfragen werden also nicht auf dem Arbeitsplatzrechner bearbeitet, sondern auf dem Datenbank-Server (der hardwaremäßig entsprechend ausgerüstet sein sollte), es werden dann nur die Ergebnisse an die Arbeitsplatzrechner geschickt.

Ein Beispiel soll den Unterschied zur *File-Share*-Datenbank erläutern: Nehmen wir an, in einem großen Versandhaus werden Mahnungen geschrieben. Um Redundanzen zu vermeiden, sind in der Tabelle *Rechnungen* nur die Kundennummern gespeichert, beim Erstellen der vielleicht hundert Mahnungen müssten ebenso viele Kundenadressen in die Standardtexte (»Sicher haben Sie übersehen ...«) eingefügt werden. Eine entsprechende SQL-Anweisung könnte lauten:

```
SELECT a.vornamen || " " || a.nachnamen AS namen,
        a.straße, a.plz || " " || a.ort AS wohnort,
        r.datum,
        r.betrag,
        r.betrag + 5 AS mahnsumme
    FROM adressen a, rechnungen r
    WHERE (r.kunde = a.nummer)
        AND (r.datum < :Mahngrenze)
        AND (r.bezahlt IS NULL)
```

(Es macht nichts, wenn Sie diese Anweisung noch nicht ganz verstehen, das lernen Sie in Kapitel 3.) Bei einer *File-Share*-Datenbank würden nun (um einmal Größenordnungen zu schätzen) 300000 Rechnungsdatensätze und 100000 Kundendatensätze zum Arbeitsplatzrechner transferiert; das können gut und gerne 20 Mbyte an Daten sein.

Bei einem *Client-Server*-System würde der Server die Anfrage selbst bearbeiten und dann rund 10 Kbyte zum Arbeitsplatzrechner übertragen. Dies würde einer Beschleunigung um den Faktor 2000 entsprechen und bei manchen Abfragen sind die Verhältnisse noch viel extremer.

Hinzu kommt, dass *Client-Server*-Systeme meist viel besser auf den Umgang mit großen Datenmengen ausgerichtet sind. Dazu gehören dezidierte Zugangskontrollen und Zugriffsrechte oder – so banal sich das auch anhören mag – die Fähigkeit, bei laufendem Betrieb ein Backup zu ziehen. (Stellen Sie sich vor, Sie gehen nachts um 2.07 Uhr an einen Bankautomaten und das Display meldet: *Zwischen 2.00 Uhr und 2.13 Uhr keine Auszahlung, von unserem Server wird ein Backup gezogen.* Und eine Datensicherung alle 24 Stunden wäre eigentlich auch schon viel zu selten.)

Fazit der ganzen Problematik: Wenn Sie mit wirklich großen Datenmengen zu tun haben (und in der glückliche Lage sind, die Entscheidung treffen zu dürfen), dann scheuen Sie nicht den Mehraufwand (und die Mehrkosten) für eine Client-Server-Datenbank, letztlich lohnt sich das immer.

Bei angenommen drei Jahren Systemlaufzeit (sehr vorsichtig geschätzt) und zehn daran beschäftigten Mitarbeitern fallen allein rund eine Millionen Euro an Lohn- und Lohnnebenkosten an. Daran gemessen sind die Mehrkosten für C-S-Systeme wirklich »Peanuts«.

Inzwischen gibt es mehrere Open-Source-Client-Server-Systeme, in der Version 6.0 ist InterBase ja auch OpenSource, so dass die Lizenzkosten kein Argument mehr gegen Client-Server sind.

1.1.3 Multi-Tier-Systeme

Inzwischen geht der Trend dazu, Multi-Tier-Systeme – also mehrschichtige Datenbankanwendungen – zu entwickeln. Bei diesen mehrschichtigen Systemen sind zwischen den Clients und den Datenbank-Servern die *Application-Server* installiert. Abbildung 1.2 zeigt den grundsätzlichen Aufbau eines solchen Systems.

Warum Multi-Tier?

Nach einer Untersuchung der *Gartner Group* sind im Durchschnitt pro Client-Server-Anwendung 700 Clients im Einsatz. Auch wenn nicht alle Clients gleichzeitig auf den Server zugreifen, sind dies Größenordnungen, die den Server zum Nadelöhr des gesamten Systems machen.

Der Server wird durch Multi-Tier-Systeme schon allein dadurch entlastet, dass er weniger Clients und somit auch weniger Transaktionen verwalten muss. Zudem können viele der *Business-Rules* auf dem Application-Server implementiert werden, so dass fehlerhafte Anweisungen erst gar nicht zum Server gelangen. In manchen Fällen führen auch mehrere Anwender der gleichen Arbeitsgruppe dieselbe Abfrage aus, so dass die Daten nur einmal vom Server bezogen werden müssen.

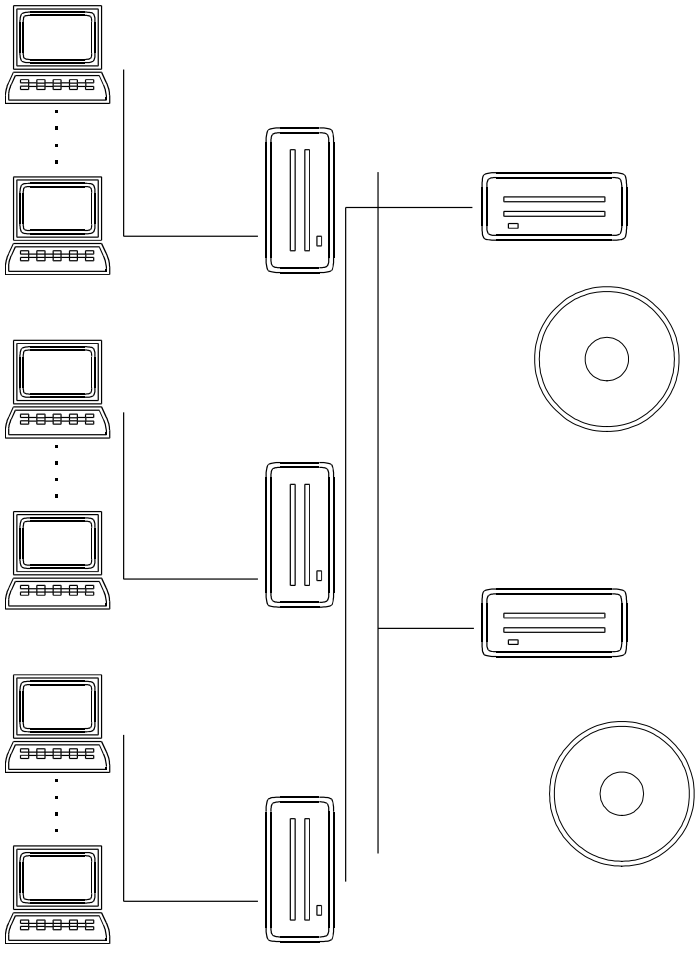


Abbildung 1.2: Prinzip eines Multi-Tier-Systems

Mit Hilfe von *Cached Updates* können auch die Daten vom Server geladen, in der Arbeitsgruppe bearbeitet und nach einiger Zeit wieder auf dem Server aktualisiert werden.

Darüber hinaus ist die Installation neuer Programme oder das Updaten bestehender Programme sehr aufwändig, wenn dies auf einer so großen Anzahl von Systemen erfolgen muss. Mit Hilfe der Multi-Tier-Technologie können *Thin Clients* erstellt werden. Solche Clients können mit vertretbarem Zeitaufwand bei jedem Systemstart vom Application-Server heruntergeladen werden und müssen somit gar nicht auf allen Clients installiert werden.

Selbst eine Verbreitung der Clients über Modem wäre denkbar. Bei einer Übertragungsgeschwindigkeit von langsamen 28800 Kbit/s wäre eine Anwendung von beispielsweise 200 Kbyte in knapp einer Minute heruntergeladen. Das würde man dann wohl nicht bei jedem Systemstart tun, sondern immer nur dann, wenn eine neue Version verfügbar ist.

Wann Multi-Tier?

Mehrschichtige Anwendungen sind anspruchsvoller und auch aufwändiger zu programmieren als die klassischen, zweischichtigen Client-Server-Anwendungen und somit sicher nicht kostengünstiger.

Für ein kleines Firmennetzwerk mit vielleicht fünf Arbeitsplätzen wäre ein solches Konzept um mehrere Größenordnungen überdimensioniert. In folgenden Fällen sollte man jedoch über die Erstellung einer mehrschichtigen Anwendung nachdenken:

- Die Zahl der Clients steigt über 100 oder mehrere Server arbeiten parallel, um einen ununterbrochenen Betrieb zu gewährleisten.
- Mehrere dezentrale, kleinere Netzwerke sind über ein WAN (*wide area network*) miteinander verbunden (Filialbetrieb). Bei solchen Systemen gilt es, Anzahl und Dauer der Serverzugriffe zu minimieren, weil dadurch Übertragungskosten entstehen. In diesem Fall würde jedes der kleineren Netzwerke von einem Application-Server (Anwendungsserver) versorgt.
- Viele Clients führen einen Fernzugriff (oder auch einen Offline-Zugriff) auf die Datenbank durch. So könnte beispielsweise eine Versicherungszentrale mit einigen hundert Vertretern verbunden sein. Diese benötigen meist nur einen bestimmten Satz an Informationen, die ihnen der Anwendungsserver zur Verfügung stellt.

Multi-Tier und Internet

Ein Webbrowser ist nun mal kein Datenbank-Client, wenn Sie eine Datenbank ins Internet bringen wollen, haben Sie schon mal zwingend drei Schichten: den Datenbank-Server, den Webserver (mit seinen externen Modulen) und den Browser.

Prinzipiell ist es keine Schwierigkeit, InterBase-Datenbanken im Internet verfügbar zu machen. Wenn es der ausschließliche Zweck der Datenbank ist, die Daten über das Internet bereitzustellen, dann ist InterBase häufig nicht erste Wahl. Sehr viele User, unklarer Transaktionstatus und häufiges Beenden der Datenbanksitzung sind nichts, wofür ein InterBase-Server optimiert wurde. Allenfalls dann, wenn sehr große Datenmengen gespeichert werden sollen (große Bilddatenbank mit hoch auflösenden Fotos beispielsweise), könnte sich die Verwendung dann wieder anbieten.

Anders sieht es aus, wenn eine ohnehin vorhandene InterBase-Datenbank webfähig gemacht werden soll, beispielsweise, damit der Außendienst auch über das Internet an benötigte Daten kommt. Hier spricht nichts dagegen, ein bestehendes System entsprechend zu erweitern.

1.2 Relationale Datenbanken

Der Begriff *relationale Datenbanken* geht zurück auf einen Artikel von E.F. Codd: *A Relational Model of Data for Large Shared Data Banks*, der 1970 veröffentlicht wurde. Inzwischen sind von Codd 333 Kriterien erstellt worden, die ein Datenbank-Management-System erfüllen muss, damit es sich relational nennen »darf«.

Nach Ansicht von Experten erfüllt derzeit kein einziges System alle 333 Kriterien. In der Praxis wird ein DBMS *relational* genannt, wenn es der »Philosophie« dieser Kriterien gerecht wird und die wesentlichsten Bedingungen erfüllt.

Nachdem fast alle heute gebräuchlichen Datenbanksysteme relationale Datenbanken sind und SQL direkt auf der »Philosophie« aufsetzt, wollen wir uns nun ein wenig damit beschäftigen.

1.2.1 Begriffe

Man kann nicht über relationale Datenbanken sprechen, ohne zuvor einige Begriffe zu klären. Manche dieser Begriffe werden auch in der Praxis verwendet, manche sind aber durch andere Begriffe ersetzt worden.

Relation

Eine *Relation* ist eine Tabelle. Relationale Datenbanken könnte man als »auf Tabellen basierende Datenbanken« bezeichnen. Sämtliche Daten werden in *Relationen*, also in Tabellen, gespeichert. Meist werden nicht nur die Daten, die der Anwender eingibt, sondern auch diejenigen, die das System zur Verwaltung benötigt, in Tabellen abgelegt.

Eine *Relation* (Tabelle) ist eine logische Verbindung von einer festen Anzahl von *Attributes* (Spalten) und einer variablen Anzahl von *Tuples* (Zeilen, Reihen). *Relationen* werden wir später noch ausführlicher behandeln.

Domain

Eine *Domain* ist ein Wertebereich, ähnlich dem, was in C oder Pascal ein Typ ist. Bei relationalen Datenbanken sind die Domains allerdings *atomar*, sie lassen sich also nicht weiter zerteilen (zumindest nicht sinnvoll).

Beispielsweise ist der *Name* eines Menschen nicht atomar, weil er sich in *Vorname* und *Nachname* (und ggf. akademische Grade) zerlegen lässt. *Vorname* und *Nachname* sind dann allerdings atomare Werte, also *Domains*.

Bei einer Datenbank sind stets einige *Domains* vordefiniert, meist hält man sich dabei an Bereiche, welche das binäre Zahlenmodell vorgibt (Integerzahlen). Es ist aber auch möglich, eigene *Domains* zu definieren. Hierzu zwei Beispiele (als SQL-Befehle, wir werden dies später behandeln):

```
CREATE DOMAIN dnachnamen AS VARCHAR(20);

CREATE DOMAIN dabteilungsnummer AS CHAR(3)
    CHECK    (VALUE = "000" OR
        (VALUE > "0" AND VALUE <= "999")
        OR VALUE IS NULL)
```

Beim ersten Beispiel fragt man sich, warum hierfür eigens eine eigene Domain definiert wird, schließlich könnte man die Angabe der Länge auch bei der Spaltendefinition machen. Im Prinzip ist das richtig. Nun gibt es allerdings Referenzen, also den Verweis auf andere Tabellenspalten. Dafür ist es zwingend erforderlich, dass beide *Attributes* (Spalten) derselben *Domain* unterliegen. Auch dies ist nun weiter kein Problem, es werden alle Nachnamen-*Attributes* als VAR-CHAR(20) definiert.

Nun wird später eine Änderung nötig; beispielsweise kommen Doppelnamen in Mode und die Nachnamen lauten nun beispielsweise *Leuthäuser-Schnarrenberger* oder *Freiherr Marschall von Biberstein* (diese Namen gibt's wirklich). Prinzipiell ist das auch noch kein Problem: Es müssen eben alle *Attributes* (Spalten) entsprechend umdefiniert werden. Sollte aber dabei auch nur eine Spalte vergessen werden, dann arbeitet die Datenbank nicht mehr korrekt. Deshalb ist es sicherer, auf eigens definierte *Domains* zu verweisen, denn dann muss die Änderung nur an einer einzigen Stelle vorgenommen werden.

Bei der Definition von *Domains* können auch etwas kompliziertere Wertebereiche vorgegeben werden, wie das zweite Beispiel zeigt. Die Definiton von *Domains* werden wir uns in Kapitel 4.1 noch näher ansehen.

NULL

Ein Ausdruck, der bei Einsteigern immer wieder für Verwirrung sorgt, ist der Wert NULL. NULL ist eben nicht gleich *null*. Vielleicht hätte man eine andere Bezeichnung dafür finden sollen (beispielsweise UNK für *unknown, unbekannt*), jedenfalls heißt der Wert nun NULL und steht für unbekannte oder nicht zutreffende Werte.

Während unbekannte Werte leicht vorstellbar sind, mag man sich zunächst fragen, was man sich unter nicht zutreffenden Werten eigentlich vorzustellen hat. Dazu ein Beispiel: Nehmen wir einmal an, in einer *Relation* (Tabelle) seien die Plätze von Konferenzräumen aufgelistet, dazu die Teilnehmernummer der Person, für die der

Platz reserviert ist. Nun hat aber der Konferenzraum mehr Plätze als Teilnehmer und so gibt es Plätze, zu deren *Attribut* (Spalte) *Teilnehmer* es keinen zutreffenden Wert gibt. Auf diesen Plätzen sitzt auch nicht der Teilnehmer *0* oder der Teilnehmer *x*, sondern für diese Spalte gibt es einfach keinen zutreffenden Wert.

Dieser Wert NULL macht übrigens aus einer zweiwertigen Logik eine dreiwertige, was einigermaßen verwirrend sein kann.

Degree und Kardinalität

Die Zahl der *Attributes* (Spalten) einer *Relation* (Tabelle) nennt man *Degree* (Ausdehnungsgrad). Relationen mit nur einer Spalte nennt man *unär*. So etwas kann durchaus sinnvoll sein, beispielsweise als Liste der gesetzlichen Feiertage – in der Regel wird (den Rechner) nur interessieren, an welchem Datum der Feiertag ist, und nicht, wie dieser genannt wird. Eine *Relation* mit einem *Degree* von *zwei* nennt man dementsprechend *binär*.

Die Zahl der *Tuples* (Datensätze) wird *Kardinalität* genannt. Während ein *Degree* von null keinen Sinn hat, kann es durchaus Relationen mit null Tuples, also leere Tabellen, geben. Als Beispiel sei die Tabelle der momentan offenen Posten genannt – wenn alle Kunden ihre Rechnungen bezahlt haben.

1.2.2 Keys (Schlüssel)

Im Zuge der so genannten »Normalisierung« (wir werden das später klären) werden viele Tabellen entstehen. Um diese Tabellen wieder richtig zusammensetzen zu können, aber auch, um die gewünschten Datensätze überhaupt finden zu können, werden sog. *Keys*, also Schlüssel, verwendet.

Candidate Key

Prinzipiell ist es ein sehr triviales Problem, Daten irgendwo zu speichern – interessant wird es erst, wenn man versucht, an diese Daten wieder heranzukommen, insbesondere dann, wenn man nicht an allen *Tuples* (Datensätzen) interessiert ist, sondern nur an einem ganz bestimmten.

Die Tabellen einer relationalen Datenbank sind nicht sequenziell organisiert (intern manchmal schon, aber nicht aus Sicht des Benutzers), es gibt also nicht den ersten, den letzten, den 385. Datensatz. Um einen Datensatz eindeutig zu bestimmen (und ihn damit suchen, besser gesagt, finden zu können), ist ein *Candidate Key,* ein »eindeutiger Schlüssel«, nötig.

Unter einem *Candidate Key* versteht man eine *Attribut*-Menge (also eine Menge von einer oder mehreren Spalten), die *unique* (eindeutig) ist. So ein *Candidate Key* ist im einfachsten Fall eine fortlaufende Nummer, beispielsweise eine Kundennummer oder eine Bestellnummer. Zwingend notwendig ist das (zumindest theoretisch)

nicht. Eine Person könnte man auch durch die Kombination mehrerer Personendaten eindeutig identifizieren – es wären allerdings viele Personendaten.

Die Kombination von Vor- und Nachnamen reicht da bei weitem nicht: Wir hatten schon in der Schule (bei einer Klassengröße von vielleicht 30 Schülern) zwei *Peter Glöckler*, bei größeren Datenmengen treten diese Probleme dann verstärkt auf. Die Adresse eignet sich kaum, weil sie sich schnell ändern kann. Besser ist schon eine Kombination von Vor- und Nachnamen, Geburtsdatum und Geburtsort – die Übereinstimmung zweier Personen in allen vier Spalten ist recht unwahrscheinlich. Andererseits: Wenn eine Übereinstimmung auftritt, dann wird sich die Datenbank weigern, den zweiten Datensatz anzunehmen, da können Sie sich »auf den Kopf stellen und mit den Füßen wackeln«.

Außerdem gibt es bei zusammengesetzten Schlüsseln ein zweites Problem: Solche Schlüssel werden zur Herstellung einer Referenz verwendet. Im Normalfall wird also in die Rechnungsdatei die Kundennummer aufgenommen, um die Kundenadresse mit den Rechnungsdaten zu verbinden. Die Alternative wäre, Vor- und Zunamen, Geburtsdatum und Geburtsort in die Rechnungsdatei aufzunehmen, der Speicherplatzbedarf für diesen Schlüssel wäre rund zehnmal so groß.

Außerdem dürfen solche *Candidate Keys* nicht geändert werden. Wenn die betreffende Person mit dem Familienstand auch ihren Nachnamen wechselt, dann kann dies in der Kundendatei nicht einfach geändert werden, weil dann die Rechnungen nicht mehr zugeordnet werden könnten – ein brauchbares DBS würde in einem solchen Fall von sich aus die Veränderung der Daten verhindern.

Candidate Keys sollten sich also aus *Attributes* (Spalten) zusammensetzen, deren Werte sich garantiert nie ändern. Also nicht *höchstwahrscheinlich nicht*, sondern *garantiert nicht*. Und hier gibt es eindeutig mehr unbrauchbare als brauchbare Eigenschaften. Geschlecht von Personen? Transsexuelle nach der Umwandlung. Namen von Nationalstaaten? Ein Blick in die jüngere Geschichte. Namen von Städten? Die Gemeindereform lässt grüßen. Einzig bei unären Relationen (Tabellen mit einer Spalte, im Prinzip Listen) mag man auf durchlaufende Nummern verzichten können, solange alle Werte wirklich nur einmal aufgenommen werden. (In einer Liste der gesetzlichen Feiertage wird der 3. Oktober nur einmal benötigt.)

An dieser Stelle möchte ich nochmals darauf hinweisen, dass Datenbanken – im Gegensatz zu Karteikarten – extrem unflexible Gebilde sind. Methoden wie »das notieren wir auf der Rückseite und machen vorne links unten ein dickes Kreuz, damit man auch nachsieht« lassen sich hier nicht anwenden. Deshalb gilt: Alle Tabellen haben eine durchlaufende oder von anderen Spalten unabhängige Nummer als *Candidate Key*, Ausnahmen bestätigen die Regel!

Primary Key

Jede Relation besitzt einen *Primary Key* (Primärschlüssel, manchmal auch Primärindex genannt), um einen Datensatz eindeutig zu identifizieren, der *Primary Key* muss also ein *Candidate Key* sein. Es ist aber durchaus möglich, dass es in der Relation einen zweiten (dritten, vierten ...) *Candidate Key* gibt, der nicht Primärschlüssel ist.

Denkbar wäre beispielsweise, dass nach einer Firmen-Fusion in einer Tabelle die Kundennummern von Firma A den Kundennummern der Firma B zugeordnet würden. Jeder Kunde lässt sich sowohl anhand der einen als auch der anderen Kundennummer zweifelsfrei identifizieren, aber *Primary Key* kann nur eine sein.

Es gilt auch zwischen Schlüsseln zu unterscheiden, die momentan (noch) *Candidate Key* sind (Vor- und Nachnamen, Geburtsdatum und Geburtsort), und solchen, die garantiert immer *Candidate Key* sein werden (also den durchlaufenden Nummern beispielsweise). Als *Primary Key* eignen sich nur Letztere.

Alternate Key

Zusätzlich zu den Primärschlüsseln lassen sich auch *Alternate Keys*, auch *Secondary Keys* (Sekundärschlüssel), definieren. Bei einer Tabelle, in der Projekte gespeichert werden, wird jedes Projekt mit einer eindeutigen, meist durchlaufenden Nummer gekennzeichnet. Darüber hinaus könnte man jedoch mit einem Sekundärschlüssel sicherstellen, dass der Projektname ebenfalls nur ein einziges Mal verwendet wird.

Foreign Key

Gerade bei relationalen Datenbanken werden sehr viele Master-Detail-Verknüpfungen erstellt. Ein Beispiel dafür wären die Rechnungstabelle und die Kundentabelle einer Firma:

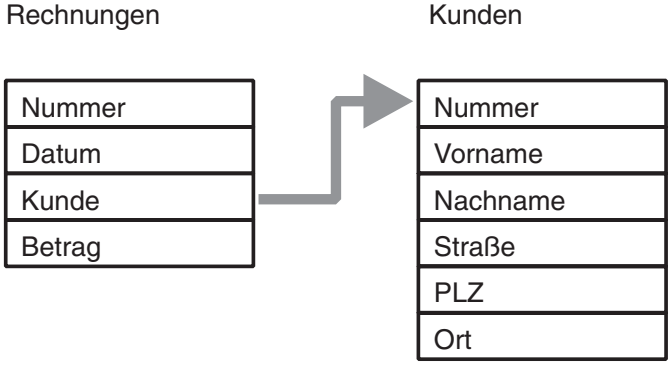


Abbildung 1.3: Einsatz eines Fremdschlüssels bei einer Master-Detail-Verknüpfung

In der Rechnungstabelle finden sich Rechnungsnummer, Datum und Betrag, außerdem interessiert die Adresse des Kunden. Es wird jedoch nicht die komplette Adresse des Kunden in der Rechnungstabelle gespeichert, sondern nur die Kundennummer – mit dieser kann dann in der Kundentabelle die Adresse ermittelt werden.

Die Kundennummer, also die Spalte *Kunde*, wäre hier ein *Foreign Key,* also ein Fremdschlüssel. Dieser verweist auf den Primärschlüssel einer anderen Tabelle und stellt über ihn die Verbindung zu dieser her. Der SQL-Befehl zur Erstellung der Tabelle *Rechnungen* würde folgendermaßen lauten:

```
CREATE TABLE rechnungen
        (nummer INTEGER NOT NULL,
        datum DATE NOT NULL,
        kunde INTEGER,
        betrag MONEY NOT NULL,
        PRIMARY KEY (nummer),
        FOREIGN KEY (kunde) REFERENCES kunden (nummer))
```

Beachten Sie bitte, dass der *Foreign Key* und der dazugehörende *Primary Key* der gleichen *Domain* unterliegen müssen.

Referenzielle Integrität

Eine wesentliche Aufgabe der DBS ist die Sicherstellung der *referenziellen Integrität*. Darunter versteht man, dass für jeden Wert einer Spalte, die als *Foreign Key* definiert ist, ein entsprechender Wert in der damit verbundenen Tabelle vorhanden sein muss. Es darf also nicht vorkommen, dass in der Tabelle *Rechnungen* ein Kunde *2345* angegeben ist, der in der Tabelle *Kunden* nicht existiert. Dazu sind zwei Sicherheitsmechanismen erforderlich:

- Wird in die Tabelle *Rechnungen* ein neuer Datensatz eingefügt (oder ein bestehender geändert), dann muss sichergestellt werden, dass für den Wert in *Kunde* ein Datensatz in *Kunden* vorhanden ist.
- Wird in der Tabelle *Kunden* ein Datensatz gelöscht (oder entsprechend geändert), dann muss das DBS prüfen, ob es in der Tabelle *Rechnungen*, Spalte *Kunde,* Einträge gibt, die auf diesen Datensatz verweisen. Ist dies der Fall, dann ist eine der drei folgenden Maßnahmen zu treffen:
  1. Die Aktion (Löschen oder Ändern) ist zu verweigern. Dies ist das Naheliegendste, aber nicht in allen Fällen zweckmäßig.
  2. Alle Werte der Spalte *Kunde* (in der Tabelle *Rechnungen*) werden auf NULL gesetzt. Dies geht allerdings nur, wenn bei der Erstellung der Tabelle nicht die Vorgabe NOT NULL getätigt worden ist. Es wäre auch möglich, den Wert der Spalte *Kunde* auf einen bestimmten Wert zu setzen, der für gelöschte Kunden steht.

In jedem Fall bleibt das Problem, dass der Kunde nicht mehr zu ermitteln ist. Das kann unter Umständen böse Folgen haben: Nehmen wir an, ein Kunde tätigt eine Bestellung und »bezahlt« per Einzugsermächtigung; kurz darauf weist er die Firma an, seine Adresse aus der EDV zu löschen, und diese kommt dem Wunsch nach. Nun geht dieser Kunde zu seiner Bank und widerspricht dem Einzug, die Firma erhält eine Rücklastschrift und möchte eine Mahnung schreiben, aber es fehlt die Adresse des Kunden ... (mögliche Rettung: Backup-Bänder).

3. Als dritte Möglichkeit bleibt, gleich auch noch alle damit verbundenen Rechnungen zu löschen. Die referenzielle Integrität wäre damit sichergestellt, aber im gerade erwähnten Beispiel könnte man dann noch nicht einmal nachvollziehen, welche Rechnung da nicht bezahlt wurde. (Vor Gericht könnte es dann so aussehen, als ob man ungerechtfertigt willkürliche Beträge einzöge.)

Fazit: Es gibt mehrere Möglichkeiten, im Falle des Löschens oder Änderns von Datensätzen die referenzielle Integrität zu erhalten. Welche dieser Möglichkeiten geeignet ist, hängt sehr von den Umständen ab.

Referenzielle Integrität und Datenschutz

Bestellt ein Kunde etwas bei einer Firma, dann willigt er in der Regel über die allgemeinen Geschäftsbedingungen darin ein, dass seine Daten gespeichert werden. Nun könnte der Kunde auf den Gedanken kommen, diese Einwilligung zu widerrufen, und um die Löschung der über ihn gespeicherten Daten ersuchen. (Das Problem stellt sich nicht nur bei Firmen, sondern beispielsweise auch bei Parteien, Vereinen und Gewerkschaften, nur dass man dann nicht von *Kunden* spricht.)

Nun werden für den Kundendatensatz in aller Regel Detaildatensätze vorhanden sein, die man so ohne weiteres nicht aus der Tabelle entfernen kann. Deshalb würde das Datenbanksystem die Löschung des Kunden verhindern. Was tun?

- In den meisten Fällen sind die Leute schon zufrieden, wenn sie keine Post mehr von der betreffenden Firma erhalten. Also ergänzt man die Tabelle um eine Spalte »gelöscht« und ändert alle Abfragen so, dass nur noch die »ungelöschten« Daten angezeigt werden. Die Sachbearbeiter sehen somit diesen Datensatz nicht mehr, an ihn werden keine Schreiben mehr verschickt, und dass der Datensatz tatsächlich noch gespeichert ist, wissen nur ein paar Leute aus der EDV.
- Man kann den Datensatz »leer« machen: Für Name, Straße und Ort werden leere Strings eingesetzt, lediglich die Kundennummer bleibt bestehen. Verständlicherweise hat man dann jedoch ein Problem, wenn man die Adresse doch mal wieder benötigt, beispielsweise weil die letzte Rechnung vom Konto des Kunden eingezogen wurde und dieser nun plötzlich die Sache zurückbuchen lässt. (Tipp: Backup-Bänder!)

- Was man nie und unter gar keinen Umständen jemals auch nur in Erwägung ziehen sollte, ist, eine Kundennummer irgendwann an einen anderen Kunden zu vergeben.

Indizes

Indizes und Schlüssel werden immer mal wieder durcheinander gebracht, schließlich wird für jeden Schlüssel ein Index aufgebaut – aber nicht alle Indizes sind Schlüssel. Ein Index ist ein Suchbaum, mit dessen Hilfe sich Datensätze schneller finden lassen. Nehmen wir an, das RDBMS bekommt folgende Anweisung:

```
SELECT * FROM testadr
    WHERE nummer = 12345
```

Es soll der Datensatz mit der Kundennummer (Primärschlüssel) *12345* ausgegeben werden. Würde hier eine sequenzielle Suche gestartet, dann würde das RDBMS den ersten Datensatz suchen, vergleichen, ob die Nummer passt, dann den nächsten wieder vergleichen und so weiter.

Man darf sich die Tabelle nicht als ein großes Array vorstellen, bei dem die 12345. Zeile auch dem Datensatz mit der Kundennummer 12345 entspricht. Es ist ohne weiteres möglich, dass die Hälfte der Datensätze schon wieder gelöscht worden ist. Außerdem muss die Kundennummer ja nicht eine fortlaufende Nummer sein; das kann von Karteikarten übertragen und daher völlig durcheinander geraten sein. Es bleibt zunächst nichts anderes als die sequenzielle Suche.

Nun wird aber anhand des Primärschlüssels eine Art Suchbaum aufgebaut (das ist eine extra Tabelle, bei Desktop-Datenbanken eine separate Datei). Hier müssen dann nicht 12345 Datensätze gelesen und verglichen werden, sondern die Suche geht in etwa folgendermaßen vor sich (bei der höchsten Kundennummer von 100000):

- Ist die Kundennummer größer oder kleiner als 50000? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 25000? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 12500? Sie ist kleiner.
- Ist die Kundennummer größer oder kleiner als 6250? Sie ist größer.
- Ist die Kundennummer größer oder kleiner als 9375? Sie ist größer.

Und so weiter und so fort. Nach noch ein paar solchen Vergleichen (die ich Ihnen jetzt ersparen möchte) hat man die physikalische Adresse des Datensatzes 12345, sucht anhand der Adresse diesen auf der Platte und gibt ihn aus. Dies alles erledigt das RDBMS selbstständig, für den Anwender spielt sich der Vorgang also »unsichtbar« ab. Ob eine Suche sequenziell oder mit einem Index erfolgt, lässt sich nur daraus erahnen, wie schnell das Ergebnis bereitsteht.

Um nach anderen Attributmengen (also Spalten oder Zusammenfassungen von Spalten) beschleunigt zu suchen, lassen sich auch für diese Attributmengen solche Indizes erstellen. Diese belegen zusätzlichen Platz auf der Festplatte und beschleunigen das Zustandekommen des Abfrageergebnisses.

Allerdings müssen solche Indizes auch gepflegt werden: Für jeden neuen Datensatz ist auch ein Eintrag in jeweils jedem Index erforderlich. Werden größere Mengen von Daten in eine Tabelle geschrieben, dann kann es sinnvoll sein, alle Indizes vorher zu löschen und anschließend neu aufzubauen.

1.2.3 Weitere Elemente relationaler Datenbanksysteme

Relationale Datenbanksysteme realisieren noch einige weitere Elemente, beispielsweise VIEWS, STORED PROCEDURES, Transaktionen oder detailliert einstellbare Zugriffsrechte.

VIEWS

Bei allen (vernünftigen) Client-Server-Datenbanken lassen sich VIEWS, also Ansichten, definieren. Dabei handelt es sich um eine eingeschränkte oder zusammenfassende Sicht auf Tabellen. Meist sind das umfangreiche JOINS, die man nicht immer wieder formulieren möchte.

Manchmal wird man auch VIEWS erstellen, damit von einer Tabelle nicht alle Daten sichtbar sind. Nehmen wir einmal an, in der Personaltabelle eines Unternehmens befänden sich die interne Durchwahl und das monatliche Gehalt (es findet sich noch viel mehr, aber das lassen wir aus Gründen der Übersichtlichkeit weg):

```
CREATE TABLE mitarbeiter
    (nummer AUTOINC,
    namen VARCHAR(20),
    durchwahl SMALLINT,
    gehalt MONEY,
    PRIMARY KEY (nummer));
```

Nun soll sich jeder Mitarbeiter zwar die Durchwahl seiner Kollegen beschaffen dürfen, aber das Gehalt geht ihn nun einmal nichts an. Deswegen erstellt man eine VIEW auf diese Tabelle:

```
CREATE VIEW durchwahlen AS
    SELECT nummer, namen, durchwahl FROM mitarbeiter.
```

Mit Hilfe von Zugriffsberechtigungen erhält jeder den (Lese-)Zugriff auf die VIEWS *durchwahlen*, aber nur die Buchhaltung und die Personalabteilung erhalten den Zugriff auf die Tabelle *Mitarbeiter*.

STORED PROCEDURES

STORED PROCEDURES bieten die Möglichkeit, komplexere Aktionen auf dem Datenbank-Server auszuführen. So sind innerhalb solcher Prozeduren Schleifen und Verzweigungen möglich. Manche STORED PROCEDURES liefern Werte und ähneln darin einer Abfrage oder einer VIEW, manche fügen Daten ein oder ändern oder löschen Daten. Wie Prozeduren in einer Programmiersprache können auch STORED PROCEDURES Parameter entgegennehmen.

Bei manchen Datenbanksystemen besteht die Möglichkeit von TRIGGERN: Wird in einer Tabelle ein Datensatz eingefügt, geändert oder gelöscht, dann wird davor und/oder danach eine Art STORED PROCEDURE aufgerufen, die hier zusätzliche Funktionalität bereitstellt – beispielsweise weitere Daten einfügt.

Transaktionen

Mit Hilfe von Transaktionen kann man eine Reihe von Aktionen zu einer Gruppe zusammenfassen, die entweder gemeinsam bestätigt oder gemeinsam verworfen werden. Beliebtes Beispiel dafür ist die doppelte Buchhaltung: Wird ein Konto belastet, muss derselbe Betrag einem anderen gutgeschrieben werden. Mit Hilfe von Transaktionen kann man nun gewährleisten, dass diese beiden Aktionen immer nur gemeinsam ausgeführt werden.

Transaktionen werden auch verwendet, um eine konsistente Datensicht zu garantieren: Müssen beispielsweise einige statistische Auswertungen durchgeführt werden, dann ist es sehr ungünstig, wenn sich dabei der Datenbestand ändert. Mit Hilfe von Transaktionen können auch solche Abfragen im laufenden Betrieb durchgeführt werden.

Zugriffsberechtigungen

Bei größeren Client-Server-Systemen kann für jede Tabelle detailliert eingestellt werden, welcher Benutzer Daten lesen, einfügen, ändern oder löschen darf. Ebenso kann festgelegt werden, wer zum Ausführen welcher STORED PROCEDURE berechtigt ist.

1.2.4 Normalisierung

Unter *Normalisierung* versteht man die schrittweise Optimierung der Datenbank durch Veränderung der Tabellendefinition. Dazu werden mehrere *Normalformen* mit jeweils zusätzlichen Bedingungen definiert.

Eine Datenbank befindet sich beispielsweise in der dritten Normalform, wenn sie den Bedingungen der ersten, zweiten und dritten, nicht aber der vierten Normalform genügt. Würde sie beispielsweise den Bedingungen der ersten, dritten, vierten und fünften, nicht aber der zweiten Normalform genügen, dann befände sie sich in der ersten Normalform.

- Bedingungen der ersten Normalform:
  1. Keine doppelten Datensätze.
  2. Die Tuplereihenfolge und Attributreihenfolge muss beliebig sein dürfen. Die Funktionalität der Datenbank darf also nicht davon abhängen, dass auf eine Zeile oder eine Spalte eine bestimmte nächste Zeile bzw. Spalte folgt. Die Zeilen werden ausschließlich über den Primärschlüssel, die Spalten ausschließlich über den Spaltennamen angesprochen.
  3. Alle Attributwerte unterliegen einer Domäne und sind somit atomar.
- (Zusätzliche) Bedingung der zweiten Normalform ist, dass jedes Schlüssel-Attribut funktional abhängig ist vom Gesamtschlüssel, nicht aber von Teilen desselben.
- Bedingung der dritten Normalform ist, dass es keine funktionalen Abhängigkeiten zwischen Attributen gibt, die nicht als Schlüssel definiert sind.

Es gibt noch weitere Normalformen, die wir allerdings hier ignorieren wollen (ab der dritten Normalform ist eine Datenbank schon sehr brauchbar). Wer sich näher für die Normalisierung interessiert, dem sei das Buch *Relationale Datenbanken* von Hermann Sauer empfohlen (Addison-Wesley, ISBN 3-8273-2060-7). Wenn Sie bislang nur Bahnhof verstanden haben, dann ist dies nicht weiter tragisch, denn wir werden die Problematik zum besseren Verständnis an einem Beispiel erläutern. Betrachten Sie zunächst Abbildung 1.4.

Nummer	Name	Durchwahl	A-Nummer	A-Name	A-Chef	P-Nummer	P-Name
123	Müller	56	7	Konstruktion	Maier	8, 9	Zündmaschinen, Multiplexer

Abbildung 1.4: Relation genügt nicht der ersten Normalform

Der Datensatzaufbau könnte der Mitarbeiter-Datei einer Firma entnommen worden sein: Diese Tabelle entspricht noch nicht einmal der ersten Normalform, da hier zwei Spalten Werte haben, die nicht atomar sind (*P-Nummer, P-Namen*). Auch wenn man mit einigen Tricks solche Daten in eine Tabelle zwängt (bei der Nummer eine *VarChar-Domain*), handelt man sich dabei nichts als Ärger ein; beispielsweise stehen Sie dann vor einem Problem, wenn Sie über solche Spalten eine Referenz bilden möchten.

Es gibt noch zwei weitere Möglichkeiten, die erste Normalform nicht zu erfüllen: Zum einen wären das Konstruktionen, die eine bestimmte Anordnung der Datensätze (oder Spalten) erfordern. So könnte man zum Beispiel auf die Idee kommen, dass der Datensatz, der als Erster in der Tabelle steht, die Abteilung darstellt, die für den Mitarbeiter »disziplinarisch« zuständig ist (also dort, wo er zuerst erfasst wurde). Es gibt allerdings bei relationalen Datenbanken keine Reihenfolge der Datensätze (und der Attribute), so dass solche Konstruktionen gewagt sind (auch wenn man es meistens hinbekommt, dass die Sache in der Regel funktioniert).

Zum anderen gibt es das Problem der doppelten Datensätze. Sie kennen es vielleicht auch, dass man von einer Firma stets zwei Anschreiben, zwei Kataloge, zwei Weihnachtsgrüße bekommt. Liegen dabei unterschiedliche Kundennummern vor, dann sind es streng genommen verschiedene Datensätze (und somit liegt kein Verstoß gegen die erste Normalform vor).

Erst recht problematisch wird es, wenn keine Kundennummer definiert ist und die Tabelle dadurch keinen *Candidate Key* besitzt. Nehmen wir einmal an, *Michael Mustermann* bekäme alles doppelt und würde der Firma schreiben, dass es zukünftig auch einmal reicht. Dort überprüft man das zunächst. Die entsprechende SQL-Anweisung lautet (auf Straße, PLZ und Wohnort verzichten wir hier der Kürze wegen):

```
SELECT * FROM kundenadressen
    WHERE (vornamen = "Michael")
        AND (nachnamen = "Mustermann")
```

Tatsächlich werden zwei Datensätze ausgegeben. Die Möglichkeit, einen von beiden zu löschen, gibt es allerdings nicht. Denn mit der SQL-Anweisung

```
DELETE FROM kundenadressen
    WHERE (vornamen = "Michael")
        AND (nachnamen = "Mustermann")
```

löscht man nun einmal beide Datensätze. Wenn der Anwender dieses Problem nicht kennt, dann gibt es für Herrn Mustermann zukünftig gar keinen Katalog mehr.

Wenn er es kennt, dann wird er auf einem Blatt Papier die Mustermannschen Daten notieren, beide Datensätze löschen und den Datensatz erneut eingeben. (Gemäß Murphys Gesetz hat das Programm ein Feature, das Neukunden automatisch den Katalog zukommen lässt. Auf seine Bitte, künftig nur noch einen Katalog zu bekommen, wird ihm somit kommentarlos ein dritter zugeschickt). Wieder bestätigt sich die Regel, dass alle Tabellen als Primärindex eine (durchlaufende) Nummer bekommen sollten.

Die erste Normalform

Die Tabelle in Abbildung 1.5 genügt der ersten Normalform. Die Reihenfolge der Datensätze ist also beliebig, doppelte Datensätze gibt es auch nicht und die Werte aller Spalten sind atomar.

Nummer	Name	Durchwahl	A-Nummer	A-Name	A-Chef	P-Nummer	P-Name
123	Müller	56	7	Konstruktion	Maier	8	Zündmaschinen
123	Müller	56	7	Konstruktion	Maier	9	Multiplexer

Abbildung 1.5: Relation genügt der ersten Normalform

Gegen die zweite Normalform kann nur verstoßen werden, wenn der Primärschlüssel aus mehreren Spalten zusammengesetzt ist, was hier nicht der Fall ist (zumindest nicht, solange (sinnvollerweise) nur *Nummer* Primärschlüssel ist).

Sinn der Sache ist es, keine unnötig zusammengesetzten Primärschlüssel zu erhalten. Wäre beispielsweise der Primärschlüssel aus *Nummer* und *Name* zusammengesetzt, dann wären nicht nur alle übrigen Spalten vom Gesamtschlüssel abhängig, sondern auch vom Feld *Nummer* (*Müller* könnte es mehrere geben).

Wenn Sie sich an die Regel halten, alle Datensätze mit einer (durchlaufenden) Nummer als Primärschlüssel zu versehen, dann werden Sie auch nie gegen die Regeln der zweiten Normalform verstoßen.

Die dritte Normalform

Nach den Regeln der dritten Normalform sind keine funktionalen Abhängigkeiten zwischen Spalten erlaubt, die nicht als Primärschlüssel definiert sind.

In Abbildung 1.5 finden sich solche Abhängigkeiten:

- Abteilungsnummer, Abteilungsname und Abteilungsleiter sind voneinander funktional abhängig. Abteilung *sieben* ist immer *Konstruktion* und hat immer den Chef *Maier* (umgekehrt ist das nicht zwingend; es könnte eine weitere Konstruktionsabteilung geben, die dann auch eine andere Nummer hätte).
- Projektnummer und Projektnamen sind auch voneinander abhängig.

Warum stören diese Abhängigkeiten? Normalerweise sind in einer solchen Tabelle mehrere, beispielsweise 80 Datensätze aus der Abteilung *sieben*, und jedes Mal wird wiederholt, dass die Abteilung *Konstruktion* und der Chef *Maier* heißt.

Eine solche Redundanz ist nicht nur ein großzügiger Umgang mit Speicherplatz, wenn der Abteilungsleiter wechselt, dann sind auch 80 Datensätze anstatt eines Datensatzes zu ändern. Dabei steigt dann auch die Gefahr, dass ein Teil der Datensätze eben nicht geändert wird und dass somit die Integrität der Datenbank aufgehoben wird, also widersprüchliche Angaben in der Datenbank zu finden sind.

Um diese funktionalen Abhängigkeiten zu reduzieren, wird die ursprüngliche Tabelle einfach in mehrere kleinere Tabellen aufgeteilt, wie dies Abbildung 1.6 zeigt.

Um die Daten aus den verschiedenen Tabellen wieder zusammenzuführen, verwendet man einen JOIN, siehe Kapitel 3.4. Soll es für den Anwender so aussehen, als ob die Daten in einer einzigen Tabelle gespeichert sind, dann kann man eine entsprechende VIEW oder STORED PROCEDURE definieren.

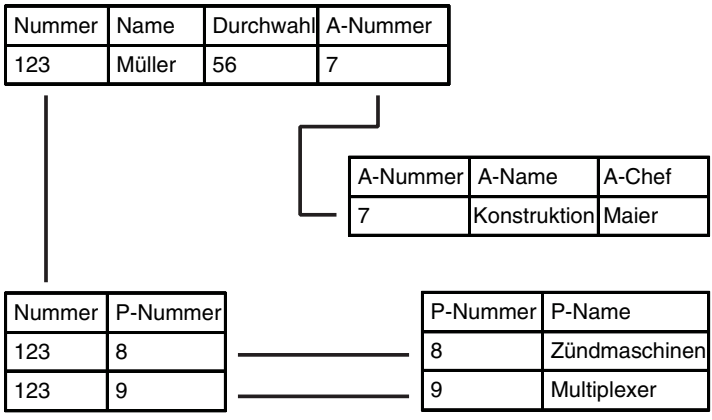


Abbildung 1.6: Aufteilung in mehrere Tabellen

1.2.5 Das erweiterte Entity-Relationship-Modell

Mit den Normalisierungsregeln lässt sich ein System vorhandener Tabellen in ein System optimierter Tabellen umwandeln. Bisweilen muss aber eine sehr komplexe und vielschichtige »Realität« in Tabellen gepresst werden, ohne dass dabei Eigenschaften und Zusammenhänge verloren gehen.

Um das Erstellen der Tabellen zu erleichtern und zu systematisieren, wurde das Entity-Relationship-Modell geschaffen. Dadurch, dass hier nur drei Grade der Entity-Beziehung und die obligatorische Mitgliedschaft als zusätzliches Kriterium definiert werden, ist dieses Modell komplizierter als eigentlich erforderlich. Wir werden deshalb die Zahl der Entity-Grade etwas erweitern und so die Notwendigkeit beseitigen, die obligatorische Mitgliedschaft als zusätzliches Kriterium zu sehen.

Das Entity-Relationship-Modell

Das Entity-Relationship-Modell gliedert die »Realität« in *Entities* (Objekte, Hauptgruppen, also beispielsweise Kunden, Aufträge, Rechnungen ...) und *Relationships* (Beziehungen zwischen den Objekten).

Das ER-Modell kennt dabei drei Grade von Beziehungen:

- 1:1 – die Eins-zu-eins-Beziehung (Beispiel: eine Haupstadt in einem Land)
- 1:N – die Eins-zu-viele-Beziehung (Beispiel: ein Land, in dem es viele Städte gibt)
- N:M – die Viele-zu-viele-Beziehung (Beispiel: Flüsse, die durch mehrere Städte fließen, und Städte, durch die mehrere Flüsse fließen)

Als zusätzliches Kriterium gibt es die *obligatorische Mitgliedschaft*. Darunter versteht man, dass der entsprechende Spaltenwert nicht NULL sein darf. Beim gerade

erwähnten Beispiel der 1:1-Beziehung besteht bei beiden Attributen obligatorische Mitgliedschaft. Es gibt weder ein Land ohne Hauptstadt, noch gibt es eine Hauptstadt ohne Land.

Es sind durchaus auch *Entities* vorstellbar, bei denen die Mitgliedschaft nur bei einem Attribut obligatorisch (NOT NULL) ist, genauso wie es *Entities* geben kann, bei denen die Mitgliedschaft bei keinem der Attribute obligatorisch ist. (Beispiel: Mitarbeiter arbeiten an Projekten, manche Mitarbeiter – z.B. die Reinigungskräfte – arbeiten an keinem Projekt, an manchen Projekten arbeitet kein Mitarbeiter, weil sie derzeit ruhen.)

Das erweiterte ER-Modell

Ob eine obligatorische Mitgliedschaft besteht, wurde beim herkömmlichen ER-Modell in Blockdiagrammen dargestellt, siehe Abbildung 1.7.

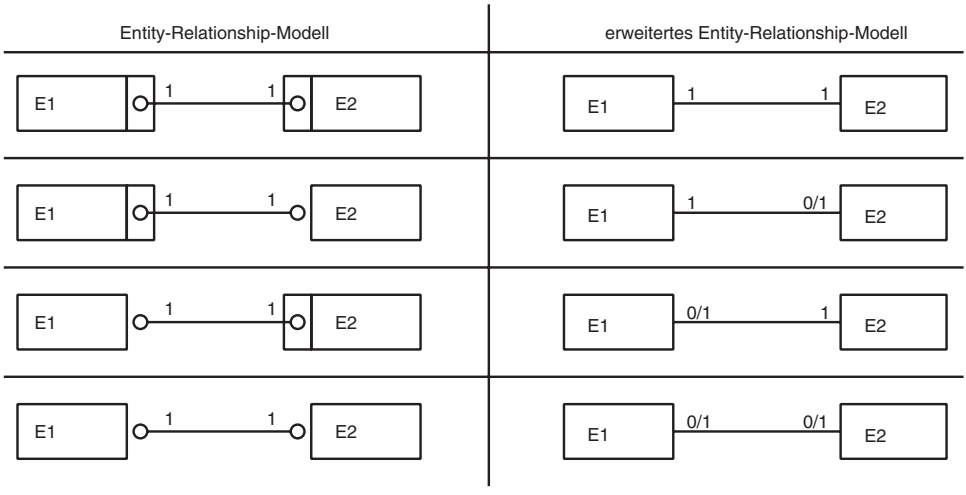


Abbildung 1.7: Konventionelles und erweitertes ER-Modell

Es ist nun aber anschaulicher, diese Information in die Bezeichnung der Beziehung aufzunehmen, so dass aus einer *1:1*-Beziehung je nach obligatorischer Mitgliedschaft *0/1:0/1-*, *1:0/1-*, *0/1:1-* oder *1:1*-Beziehungen entstehen. Hier besteht nun nicht mehr die Gefahr, dass man die Darstellungsformen von obligatorischer und nicht obligatorischer Mitgliedschaft durcheinander bringt. Zudem lässt sich diese Information auch außerhalb von Blockdiagrammen darstellen.

Aus *1:N-* und *N:M*-Beziehungen werden dann beispielsweise *0/1:0/1/N-* oder auch *1/N:0/1/M*-Beziehungen. Theoretisch denkbar wären hier beispielsweise auch 1:N-Beziehungen, wenn zu einem Entity mehrere andere gehören (Gemarkung : Grundstücke). Um hier das EER-Modell vom ER-Modell zu unterscheiden, könnte man die jeweiligen Werte unterstreichen (1:1, 1:N).

Transformation von Relationships in Attribute oder Relationen

Die Verknüpfungen (*Relationships*) der einzelnen Objekte (*Entities*) geschehen durch das Einfügen zusätzlicher Attribute (Spalten) oder durch Erstellung neuer Relationen (Tabellen).

- Bei der 1:1-Beziehung besteht grundsätzlich kein Grund, verschiedene Tabellen zu verwenden, Hauptstädte und Länder sind einfach verschiedene Spalten derselben Tabelle. Ausnahmen sind eigentlich nur bei Desktop-Datenbanken sinnvoll:
  1. Wenn die Tabelle sonst zu groß wird.
  2. Wenn zur Realisierung unterschiedlicher Zugriffsrechte nicht auf VIEWS zurückgegriffen werden kann.

  Die 1:1-Verknüpfung wird dann dadurch hergestellt, dass die »zweite« Tabelle auf den Primärschlüssel der ersten referenziert wird.
- Bei der 0/1:1- (oder 1:0/1-) Beziehung hängt die Vorgehensweise davon ab, wie das Verhältnis der Zahl der *null*-Werte zur Zahl der *eins*-Werte ist:
  1. Ist der *eins*-Wert die Regel (die meisten Lehrer sind Klassenlehrer irgendeiner Klasse), dann wird die Verknüpfung über eine weitere Spalte hergestellt, die ggf. NULL-Werte enthält.
  2. Ist der *null*-Wert die Regel (die wenigsten Mitarbeiter fahren einen Dienstwagen), dann wird eine zusätzliche Tabelle erstellt.

  Abbildung 1.8 zeigt die beiden Möglichkeiten.

L-Nummer	L-Name	L-Klasse
1	Schneider	NULL
2	Poncelet	9
3	Will	8
4	Kowalski	10
5	Beck	5a

Nummer	Name
1	Maier
2	Müller
3	Schulze
4	Lehmann
5	Schütz
6	Jansen
7	Goedecke
8	Anton

0/1

1

Nummer	Fahrer	Fahrzeug	Kennzeichen
1	7	VW-Bus	B-UM 123
2	8	Opel Corsa	B-AB 456

Abbildung 1.8: Die Herstellung der 0/1:1-Beziehung

- Die beiden vorhin genannten Möglichkeiten gibt es auch bei der 0/1:0/1-Beziehung. Nun kann es allerdings auch vorkommen, dass bei beiden *Entities* die Zahl der *null*-Werte überwiegt. Als Beispiel seien die Professoren einer Uni und die Preisträger eines Wissenschaftspreises genannt. Nur sehr wenige Dozenten haben den Preis erhalten und auch nur wenige Preisträger stammen von dieser Uni.

  Hier wird man dann eine zusätzliche Tabelle zur Herstellung der Verknüpfung erstellen, wie dies Abbildung 1.9 zeigt (die Namen sind hier frei erfunden). In diesem Fall ist es nicht unbedingt erforderlich, als Primärschlüssel für diese Tabelle eine durchlaufende Nummer zu verwenden – eine Kombination aus den beiden Spalten reicht hier völlig aus.

Nummer	Name	Fachbereich
1	Maier	Physik
2	Müller	Physik
3	Schulze	Physik
4	Lehmann	Physik
5	Schütz	Maschinenbau
6	Jansen	Maschinenbau
7	Goedecke	Chemie
8	Anton	Chemie

Nummer	Jahr	Name	Fachbereich
1	2011	Lehn	Physik
2	2011	Kolb	Mathematik
3	2011	Goedecke	Chemie
4	2011	Smith	Literatur
5	2012	Schulze	Physik

Dozent	Preis
7	3
3	5

Abbildung 1.9: Dritte Möglichkeit einer 0/1:0/1-Verknüpfung

An diesem Beispiel wird auch noch einmal deutlich, welchen Vorteil die Verwendung von (durchlaufenden) Nummern als Primärindex hat: Prinzipiell hätte man für die Preisträger-Tabelle auch Jahreszahl und Fachbereich verwenden können (solange der Preis immer nur an eine Person vergeben wird), damit wäre der Datensatz eindeutig zu identifizieren gewesen. Bei der Verknüpfungstabelle hätte man dann aber statt einer einzelnen Nummer Jahreszahl und Fachbereich als Fremdschlüssel benötigt.

- Bleiben wir bei der Uni: Die Dozenten schreiben auch Fachbücher, die (neben Werken anderer Autoren) in der Bibliothek stehen. Hier haben wir dann die *0/1:0/1/N*-Beziehung: Die einzelnen Dozenten haben keins, eins oder mehrere Werke geschrieben, aber längst nicht alle Bücher der Bibliothek sind von einem Dozenten dieser Uni geschrieben worden. Hier wird dann die Verknüpfung über eine eigene Relation hergestellt, wie dies in Abbildung 1.10 gezeigt wird.

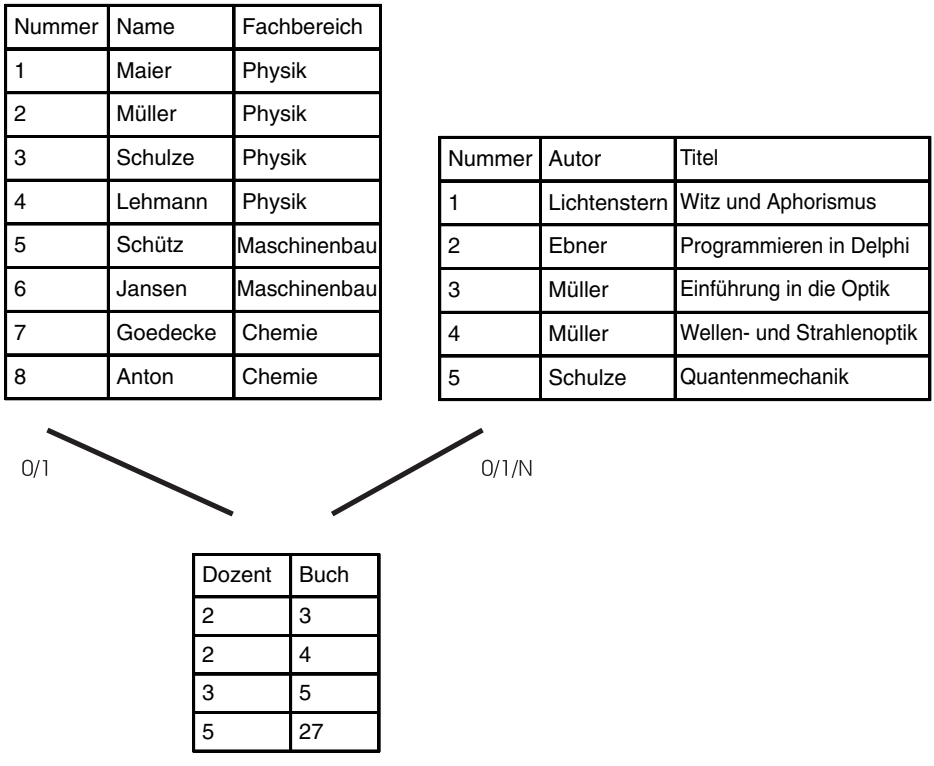


Abbildung 1.10: Die 0/1:0/1/N-Verknüpfung

Auf dieselbe Weise werden auch *0/1:1/N*- und *0/1:N*-Beziehungen aufgebaut. Eine Ausnahme machen die *Relationships*, bei denen kaum *null*-Werte auf der *0/1*-Seite vorhanden sind; diese können auch nach dem Muster der 1:0/1/N-Beziehung aufgebaut werden.

- Als nächstes Beispiel sei die Auftrags-Kunden-Beziehung einer Firma genannt, wobei es sich streng genommen um eine *1:1/N*-Beziehung handelt: Ein Auftrag kann nur von einem Kunden stammen (von wem sonst) und Kunde ist, wer mindestens einen Auftrag erteilt hat. Nun wird die Kundentabelle meist aber auch als Grundlage für den Katalogversand verwendet, so dass sich darin auch potenzielle Kunden befinden, also Personen, die um einen Katalog gebeten, aber noch keinen Auftrag erteilt haben. Hier würde es

sich dann um eine *1:0/1/N*-Beziehung handeln, was aber bei der Ausgestaltung keinen Unterschied macht: Da so oder so jeder Auftrag einen Kunden hat, wird in die Auftragstabelle eine Spalte eingefügt, die auf den Kunden verweist (siehe Abbildung 1.11).

Nummer	Kunde	Datum	Betrag
1	3	1. 1. 96	100,00
2	3	2. 3. 96	200,00
3	3	4. 5. 96	123,45
4	4	2. 2. 96	234,56
5	4	1. 3. 98	345,67

Nummer	Name	Ort
1	Lichtenstern	Bad Buchau
2	Ebner	Berlin
3	Müller	Hamburg
4	Maier	Hamburg
5	Schulze	München

Abbildung 1.11: Die 1:(0)/1/N-Beziehung

- Recht übersichtlich wird es dann wieder bei der *(0)/(1)/N:(0)/(1)/M*-Beziehung. Als Beispiel sollen hier – wie bereits erwähnt – die Städte und Flüsse dienen: Durch manche Städte fließen mehrere Flüsse und die meisten Flüsse fließen durch mehrere Städte. Hier wird man wohl stets auf eine extra Beziehungsrelation zurückgreifen (also auf eine Tabelle, deren eine Spalte auf die erste Tabelle und deren zweite Spalte auf die zweite Tabelle verweist). Auf ein Bild wurde hier verzichtet, inzwischen werden Sie das selbst hinbekommen.

Eine Ausnahme könnte dann sinnvoll sein, wenn auf einer Seite die *null*- oder *eins*-Werte sehr deutlich überwiegen, der *n*-Wert dagegen höchst selten ist; ein passendes Beispiel fällt mir aber dazu nicht ein.

1.3 Was ist InterBase?

InterBase ist ein Client-Server-Datenbanksystem.

Open Source

InterBase ist ein kommerzielles Produkt. Allerdings wurde die Version 6.0 als Open Source veröffentlicht und kann somit (unter Beachtung einiger Restriktionen) ohne Lizenzgebühren verwendet werden.

Aufbauend auf diese Version 6.0 hat sich auch eine »Open-Source-Schwester« namens Firebird entwickelt, die von einer kleinen Programmierergemeinde weiterentwickelt wird und auch in Zukunft Open Source bleiben wird.

Geeignet für große Datenmengen

InterBase kann bis zu 65536 Datenbankdateien verwalten, deren Größe nur vom Betriebssystem beschränkt wird. Nehmen wir an, die Grenze würde bei 2 Gbyte liegen, dann könnte (wenn kein Schadow verwendet wird) die Größe der Datenbank insgesamt bei 128 Tbyte liegen – vorausgesetzt, es lassen sich so viele Festplatten betreiben. In der Praxis laufen Bilddatenbanken von mehreren Tbyte Größe auf InterBase.

Mit Hilfe von BLOBs kann InterBase auch sehr große Datenmengen in einem einzelnen Datenbankfeld speichern. Liegt die Größe einer Datenbankseite bei 4 kbyte, dann können 4 Gbyte in einem BLOB gespeichert werden, bei einer Datenseitengröße von 8 kbyte sind es sogar 32 Gbyte.

Wartungsarm und betriebssicher

Im Gegensatz zum Marktführer im Bereich der Datenbanken kann man bei InterBase nicht viel einstellen, man muss es auch nicht. InterBase wird meist installiert und kann dann vergessen werden – einen eigenen Datenbankadministrator braucht es in den wenigsten Fällen.

Wenn ein InterBase-Server abstürzt – beispielsweise wegen eines Stromausfalls des Servers –, dann fährt man Rechner und Datenbank neu hoch und die Datenbank funktioniert wieder wie vorher. Wenn man auf das performancesteigernde Caching verzichtet und alle Änderungen direkt auf die Platte schreiben lässt, dann ist im Falle eines Server-Absturzes noch nicht einmal mit Datenverlust zu rechnen.

Geringer Ressourcenbedarf

InterBase benötigt nicht viele Ressourcen, weder auf der Festplatte noch im Arbeitsspeicher. Alle Dateien im */bin*-Verzeichnis haben zusammen etwa 6 Mbyte.

InterBase profitiert wie alle (mir bekannten) Datenbanken von möglichst viel Arbeitsspeicher. Ab Version 7.0 unterstützt der Server Multi-Prozessor-Maschinen.

Mehr-Generationen-Architektur

Die Strategie von InterBase zur Vermeidung von Lese- und Schreibkonflikten ist nicht das Sperren von Datensätzen (*Locking*), sondern das Anlegen von mehreren Generationen (*Versioning*). Dadurch kann die Datensicht während einer Transaktion konstant gehalten werden, ohne dass dies schreibende Transaktionen behindert. Backups im laufenden Betrieb sind überhaupt kein Problem.

Wenn es gewünscht wird, können Transaktionen auch so gestartet werden, dass sie die bearbeiteten Tabellen sperren.

Großer Sprachumfang

InterBase unterstützt Sprachelemente wie TRIGGER, STORED PROCEDURES, GENERATORS, EXCEPTIONS, EVENTS...

Mittels externer DLLs können beliebige Funktionen als USER DEFINED FUNCTIONS realisiert werden.

Nach der Theorie der relationalen Datenbanken ist es zwar eine »Todsünde«, aber wenn man meint, Arrays zu brauchen, können auch diese erstellt werden.

Mehrere Plattformen

InterBase unterstützt in der aktuellen Version Windows, Linux und Sun Solaris sowie die Schnittstellen JDBC und ODBC. Datenbanken können mittels Backup und Restore sehr einfach auf andere Betriebssysteme übertragen werden.

1.3.1 Neu in InterBase 7.0

In der Version 7.0 finden sich unter anderem die folgenden Neuerungen:

- Die Speicherverwaltung wurde multi-thread-fähig gemacht, somit profitiert jetzt InterBase massiv von Multi-Prozessor-Maschinen.
- Es wurde der Datentyp *boolean* eingeführt mit den Schlüsselworten BOOLEAN, TRUE, FALSE und UNKNOWN.
- Bei der Definition von TRIGGERN und STORED PROCEDURES kann auf SET TERM verzichtet werden.
- Die Standard-Dateierweiterung wurde von *gdb* nach *ib* geändert, um die Probleme mit der Windows-XP-Systemwiederherstellung zu umgehen.
- Viele Server-Daten werden über temporäre System-Tabellen zugänglich gemacht. Mit deren Hilfe können nun auch Verbindungen und Abfragen einzeln terminiert werden.

1.4 Installation von InterBase 7.0

Die Installation ist nicht weiter problematisch, wir wollen sie deshalb nur kurz anschneiden:

Vor der Installation wäre es angebracht, mal wieder ein Backup aller Datenbanken zu erstellen. Ein solches benötigen Sie ohnehin, wenn Sie eine Datenbank auf eine neue ODS (*on disc structure*) umstellen wollen.

Haben Sie bereits eine frühere Version von InterBase auf dem Rechner, dann sollte diese vor der Installation entfernt werden. Über START | EINSTELLUNGEN | SYSTEMSTEUERUNG | SOFTWARE kommen Sie in eine Liste der bereits installierten Software und können diese entfernen.

Dann legen Sie die CD ein und warten, bis AutoStart die Installation beginnt. Haben Sie AutoStart deaktiviert, dann starten Sie auf der CD die Datei *setup.exe*.

Abbildung 1.12: Installation von InterBase

An dieser Stelle müssen Sie nun entscheiden, ob der Server oder nur ein Client installiert werden soll. Pro Datenbank gibt es exakt einen Rechner, der als Server arbeitet. Dort installieren Sie *InterBase*. Auf allen anderen Rechnern installieren Sie *InterBase Client only.*

Prinzipiell wäre es möglich, bei verschiedenen Datenbanken auch mehrere Server zu verwenden. Damit kann man in der Regel die Performance verbessern. Wenn Sie jedoch keine dezidierten Server verwenden, sondern ein Peer-To-Peer-Netzwerk von Arbeitsplatzrechnern haben, ist es nicht ganz ungefährlich, einfach mal auf jeden Rechner den Server aufzuspielen: Beim Einrichten der Clients muss man nun peinlich genau darauf achten, dass nun der Client auch auf den tatsächlichen Datenbankserver zugreift und nicht auf seinen lokalen Server, der dann über das Netzwerk auf die Datenbankdatei zugreifen will – dies würde nämlich dazu führen, dass dann zwei Server auf die gleiche Datenbankdatei zugreifen, was mit großer Sicherheit zu einem Defekt der Datenbank führt. Solange Sie keinen Server auf Ihren Arbeitsplatzrechnern haben, besteht diese Gefahr schon prinzipiell nicht.

Nehmen wir an, wir klicken auf *InterBase* und installieren so den Server und automatisch auch den Client. Zunächst erhalten Sie das Willkommensfenster, dann das *Licence Agreement* und die *Install Information*. InterBase prüft auch, ob der Server noch läuft. Wenn Sie vorherige Versionen deinstalliert haben, ist dies ohnehin nicht mehr möglich.

Wenn Sie sich die Deinstallation gespart haben, dann könnten Sie darauf hingewiesen werden, den Guardian doch bitte zu beenden. Wenn der Guardian beendet ist, läuft auch der Server nicht mehr.

Abbildung 1.13: Während der Installation darf kein InterBase Guardian aktiv sein

Nun werden Sie nach *Certificate ID* und *Key* gefragt. In der InterBase-Schachtel liegt dafür ein Umschlag, in dem sich diese Informationen befinden. Liegen diese Informationen nicht vor, dann können Sie *Evaluate the Product* wählen und haben damit eine 90-Tage-Testversion.

Anschließend werden Sie gefragt, welche Komponenten Sie installieren wollen. Auf einem produktiv arbeitenden Server beispielsweise benötigen Sie keine Beispielprogramme.

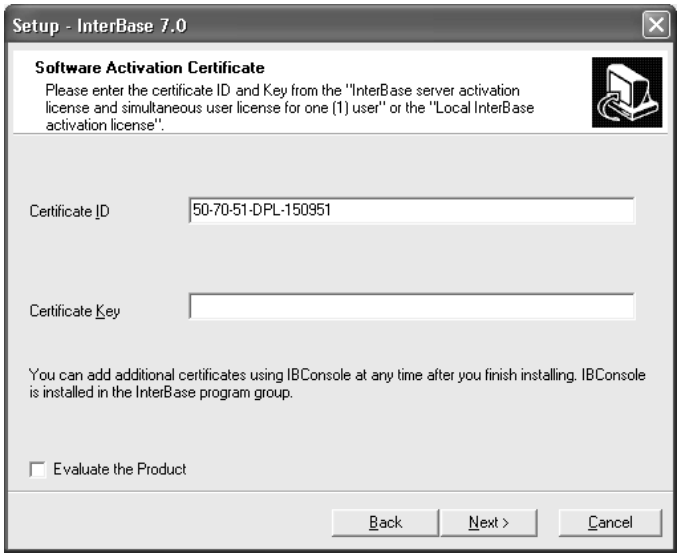


Abbildung 1.14: ID und Key

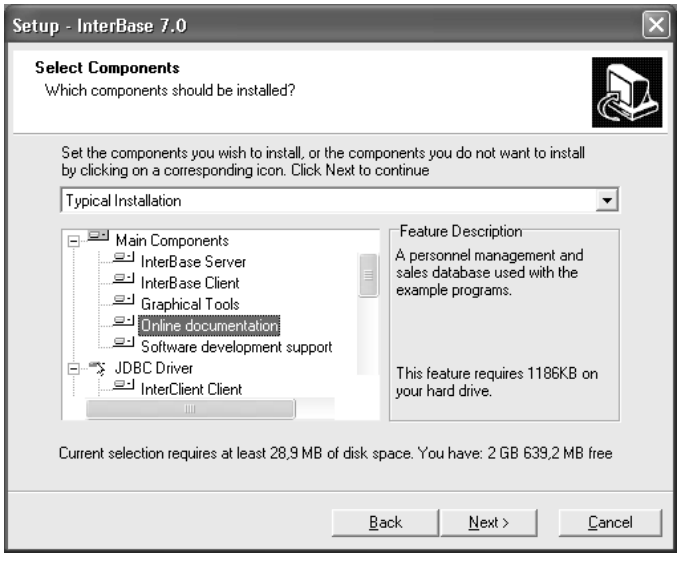


Abbildung 1.15: Auswahl der Komponenten

Dann will das Installationsprogramm natürlich wissen, in welchem Verzeichnis das Programm installiert werden soll und wie der Ordner im Startmenü genannt werden soll – Abbildungen dazu ersparen wir uns jetzt. Nach einer Zusammenfassung der Installationskonfiguration wird InterBase dann installiert. Es kann anschließend ohne Neustart des Rechners gestartet werden.

2 Testdaten generieren

Zur Programmierung von Datenbankanwendungen, insbesondere zum Testen der Ausführungsgeschwindigkeiten, sollte man eine Datenbank verwenden, deren Größe der Größe der später verwendeten Datenbank entspricht. Die Abfrage einer Tabelle mit zehn Datensätzen ist nie unperformant – bei 10000 Datensätzen kann dies schon ganz anders aussehen.

Im Idealfall besteht diese Datenbank schon und kann während der Programmierung benutzt werden. Häufig ist dies jedoch nicht der Fall, und man muss nun daran gehen, sich Testdaten zu generieren. Wie man dabei vorgeht, soll in diesem Kapitel beschrieben werden.

2.1 Das SQL-Script der Datenbank

Unsere Datenbank soll die Auftragsverwaltung einer fiktiven Firma darstellen. Eine solche Auftragsverwaltung wird normalerweise etwas anspruchsvoller gestaltet, für eine Testdatenbank eignet sich jedoch besser eine vereinfachte Ausführung.

In dieser Datenbank finden sich fünf Tabellen:

- Die Tabelle *t_adressen* beinhaltet die Kundenadressen.
- In der Tabelle *t_mitarbeiter* finden sich die Beschäftigten unserer fiktiven Firma.
- Was die Firma verkauft, nämlich Poster, ist in der Tabelle *t_produkt* aufgeführt.
- Die Bestellungen finden wir in der Tabelle *t_bestellung*, die dazugehörenden Posten in der Tabelle *t_posten*.

Nun ein Blick in das SQL-Skript:

```
SET SQL DIALECT 3;

/* CREATE DATABASE 'D:\InterBase\Progs\erzeug\testdat.gdb' PAGE_SIZE 4096
DEFAULT CHARACTER SET ISO8859_1 */

/* Domain definitions */
CREATE DOMAIN "D_ANREDE" AS VARCHAR(4) CHARACTER SET ISO8859_1;
CREATE DOMAIN "D_ID" AS INTEGER NOT NULL;
CREATE DOMAIN "D_NAME" AS VARCHAR(20) CHARACTER SET ISO8859_1;
CREATE DOMAIN "D_PLZ" AS VARCHAR(5) CHARACTER SET ISO8859_1;
CREATE DOMAIN "D_PREIS" AS FLOAT;
CREATE DOMAIN "D_TEL" AS VARCHAR(16) CHARACTER SET ISO8859_1;
```

Die erste Anweisung setzt den SQL-Dialekt, und zwar auf die aktuelle Version, nämlich drei. Die Anweisung CREATE DATABASE ist auskommentiert, das erledigt man einfacher mit *IBConsole*. Für die verwendeten Spalten werden durchgehend selbst definierte Domains verwendet, die zuvor definiert werden müssen. Bei allen Strings wird durchgehend der Zeichensatz ISO8859_1 verwendet.

```
/* Table: T_ADRESSEN, Owner: SYSDBA */
CREATE TABLE "T_ADRESSEN"
(
  "ID"          "D_ID",
  "ANREDE"      "D_ANREDE",
  "VORNAME"     "D_NAME",
  "NACHNAME"    "D_NAME",
  "STRASSE"     "D_NAME",
  "PLZ"         "D_PLZ",
  "ORT"         "D_NAME",
  "TEL"         "D_TEL",
 PRIMARY KEY ("ID")
);
```

In der Tabelle *t_adressen* findet man die für eine Adresse nötigen Spalten sowie die Kundennummer ID als Primärschlüssel.

```
/* Table: T_BESTELLUNG, Owner: SYSDBA */
CREATE TABLE "T_BESTELLUNG"
(
  "ID"          "D_ID",
  "KUNDE"       "D_ID",
  "MITARBEITER" "D_ID",
  "DATUM"       DATE,
 PRIMARY KEY ("ID")
);
```

Jede Bestellung hat eine Bestellnummer ID, ein Datum sowie Verweise auf einen Kunden- und einen Mitarbeiter-Datensatz.

```
/* Table: T_MITARBEITER, Owner: SYSDBA */
CREATE TABLE "T_MITARBEITER"
(
  "ID"          "D_ID",
  "VORNAME"     "D_NAME",
  "NACHNAME"    "D_NAME",
 PRIMARY KEY ("ID")
);
```

Die Mitarbeitertabelle ist beschränkt auf den Vor- und Nachnamen. Wo der Kollege wohnt, mag in der Lohnbuchhaltung interessieren, aber nicht bei der Auftragsverwaltung.

```
/* Table: T_POSTEN, Owner: SYSDBA */
CREATE TABLE "T_POSTEN"
(
  "BESTELLUNG"  "D_ID",
  "POS"         "D_ID",
  "PRODUKT"     "D_ID",
  "STUECKZAHL"  "D_ID",
 PRIMARY KEY ("BESTELLUNG", "POS")
);
```

Die Posten der einzelnen Bestellungen werden in der Tabelle *t_posten* gespeichert. Zunächst eine Referenz auf die Bestellung, dann eine Postennummer (die wir bei jeder Bestellung neu mit eins beginnen werden), eine Referenz auf das Produkt und die Stückzahl. Wir bilden hier einen zusammengesetzten Primärschlüssel aus Bestellung und Postennummer.

```
/* Table: T_PRODUKT, Owner: SYSDBA */
CREATE TABLE "T_PRODUKT"
(
  "ID"          "D_ID",
  "KAT"         "D_NAME",
  "PHOTOGRAPH"  "D_NAME",
  "TITEL"       "D_NAME",
  "PREIS"       "D_PREIS",
 PRIMARY KEY ("ID")
);
```

Unsere fiktive Firma soll Poster verkaufen. (Von meiner früheren Idee, Hardware zu verkaufen – Leser von *SQL lernen* erinnern sich vielleicht –, bin ich abgekommen; bis der Leser das Buch in den Händen hält, sind die Daten deutlich veraltet …). Ein solches Poster hat eine Kategorie, einen Fotografen, einen Titel und einen Preis.

```
/*  Index definitions for all user tables */
CREATE INDEX "IX_ADRESSEN_NACHNAME" ON "T_ADRESSEN"("NACHNAME");
CREATE INDEX "IX_ADRESSEN_ORT" ON "T_ADRESSEN"("ORT");
CREATE INDEX "IX_ADRESSEN_PLZ" ON "T_ADRESSEN"("PLZ");
CREATE INDEX "IX_BESTELLUNG_DATUM" ON "T_BESTELLUNG"("DATUM");
```

Nun erstellen wir ein paar Indizes, vor allem für die Kundenadressen, aber auch für das Datum der Bestellung.

```
ALTER TABLE "T_BESTELLUNG" ADD FOREIGN KEY ("KUNDE")
  REFERENCES T_ADRESSEN ("ID");
ALTER TABLE "T_BESTELLUNG" ADD FOREIGN KEY ("MITARBEITER")
  REFERENCES T_MITARBEITER ("ID");
```

```
ALTER TABLE "T_POSTEN" ADD FOREIGN KEY ("BESTELLUNG")
  REFERENCES T_BESTELLUNG ("ID");
ALTER TABLE "T_POSTEN" ADD FOREIGN KEY ("PRODUKT")
  REFERENCES T_PRODUKT ("ID");
```

Es folgen ein paar Fremdschlüssel: Die Tabelle *t_bestellung* referenziert die Kunden- und die Mitarbeiter-Tabelle, die Tabelle *t_posten* darf nur Datensätze erhalten, für die es eine Bestellung und ein Produkt gibt.

```
CREATE GENERATOR "G_ADRESSEN";
CREATE GENERATOR "G_BESTELLUNG";
CREATE GENERATOR "G_MITARBEITER";
CREATE GENERATOR "G_PRODUKT";
```

Die Spalte ID soll jeweils selbst inkrementierend sein. Wie bei Interbase üblich, basteln wir so etwas mittels Generator und Trigger. Die Tabelle *t_posten* enthält keine Spalte ID und benötigt somit auch keinen Generator.

```
SET TERM ^ ;

/* Triggers only will work for SQL triggers */

CREATE TRIGGER "TRIG_ADRESSEN" FOR "T_ADRESSEN"
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  NEW.ID = GEN_ID(g_adressen, 1);
END
 ^

...

COMMIT WORK ^
SET TERM ;^
```

Von den vier Triggern wollen wir uns hier nur einen ansehen, die anderen sind identisch aufgebaut. Vor dem Einfügen eines neuen Datensatzes holt dieser Trigger einen neuen Generatorwert und weist ihn der Spalte ID zu. Dies tut er unabhängig davon, ob die Spalte ID schon einen Wert enthält oder nicht. Wir werden später sehen, was dies für Konsequenzen hat.

Kleine Anmerkung: Die Anweisung SET TERM ist ab InterBase 7.0 nicht mehr erforderlich.

```
/* Grant Roles for this database */
/* Grant permissions for this database */
```

Spezielle Rollen werden nicht definiert, ebenso wenig weitere Zugriffsberechtigungen.

2.2 Das Programm zum Generieren der Daten

Das Programm zum Generieren der Daten ist relativ »geradeaus« gestrickt: In lokalen *TClientDataSet*-Tabellen werden Rohdaten gehalten, die mittels der Random-Funktion zusammengemischt werden.

Zunächst werden wir auch den Zugriff auf InterBase recht einfach halten. Erwartungsgemäß wird die Performance dann »verbesserungsfähig« sein und diese Verbesserungen werden wir dann vornehmen.

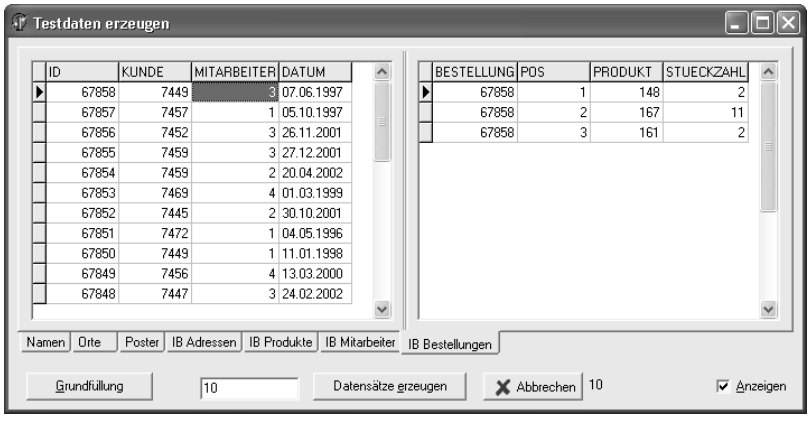


Abbildung 2.1: Programm zum Erstellen der Testdaten

Für die Rohdaten gibt es vier Tabellen:

- In der Tabelle *cds_namen* finden sich Vor-, Zu- und Straßennamen.
- Die Tabelle *cds_orte* beinhaltet einige Ortsnamen nebst dazugehörenden Postleitzahlen und Telefon-Vorwahlen.
- Die Kategorien der Poster-Titel findet man in *cds_kat*, die dazugehörenden Poster-Titel in *cds_poster*.

Alle Tabellen werden im Programmverzeichnis gespeichert. Zu diesem Zweck wird der Eigenschaft *FileName* ein Dateiname ohne Pfadangabe zugewiesen.

Die Tabelle *cds_poster* wird mittels einer Master-Detail-Verknüpfung mit *cds_kat* verbunden. Zu diesem Zweck wird die Eigenschaft *MasterSource* von *cds_poster* auf *ds_kat* gesetzt sowie die Spalte *kat* mit der Spalte *Nummer* von *cds_kat* verknüpft.

2.2.1 Zugriff auf die InterBase-Datenbank

Für den Zugriff auf InterBase gibt es mehrere Möglichkeiten. Wir wollen dafür *dbExpress* verwenden, weil dies die Chance bietet, dieses Programm auch mit anderen Datenbanken einzusetzen.

Für den Zugriff verwenden wir die Komponente *TSQLConnection*. Abbildung 2.2 zeigt, wie die Verbindungsparameter zu setzen sind, außerdem setzen wir die Eigenschaft *LoginPrompt* auf *false*.

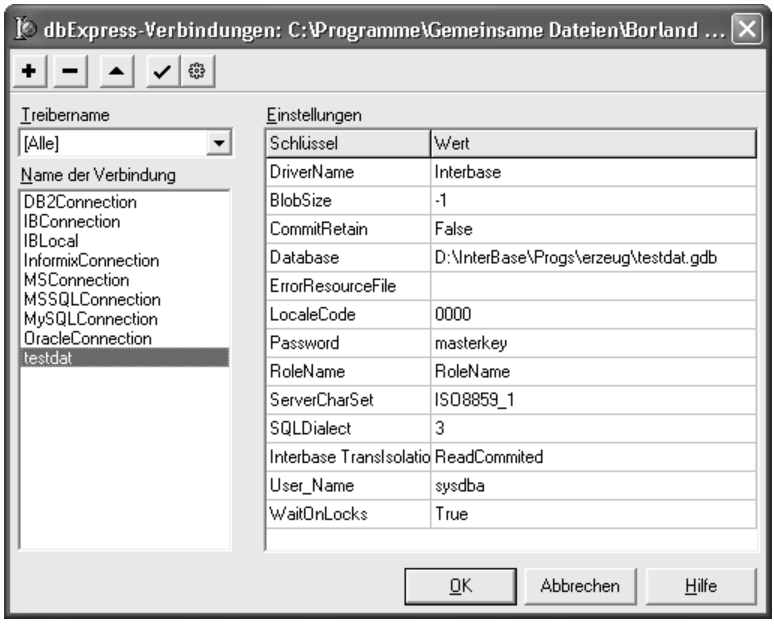


Abbildung 2.2: Einstellen der Verbindungsparameter

2.2.2 Erstellen einer Kundenadresse

Für den Zugriff auf die Tabelle *t_adressen* verwenden wir vier Komponenten:

- Eine *TSQLDataSet*-Instanz namens *sql_adresse*:
  - *SQLConnection* setzen wir auf *SQLConnection1*.
  - Die Eigenschaft *CommandType* erhält den Wert *ctQuery*, als *CommandText* verwenden wir *select * from T_ADRESSEN order by ID desc*.
  - Wir legen persistente *TField*-Instanzen an und setzen bei *sql_adresseID* die Eigenschaft *Required* auf *false*. (Andernfalls würde sich das Programm später über einen fehlenden Wert beschweren – den setzt jedoch erst der Trigger.)

- Eine *TDataSetProvider*-Komponente namens *prov_adresse*:
  - Diese Komponente muss lediglich über die Eigenschaft *DataSet* mit *sql_adresse* verbunden werden.
- Eine *TClientDataSet*-Instanz namens *cds_adresse*:
  - Die Eigenschaft *ProviderName* wird auf *prov_adresse* gesetzt.
  - Auch hier legen wir wieder persistente *TField*-Instanzen an und setzen bei *cds_adresseID* die Eigenschaft *Required* auf *false*.
  - *PacketRecords* erhält den Wert 30.
- Zuletzt wird noch eine *TDataSource* namens *ds_adresse* mit *cds_adresse* verbunden. An diese DataSource wird dann das Grid gehängt.

Zum Erstellen einer einzelnen Adresse verwenden wir die Prozedur *CreateAdresse*:

```
procedure Tdm.CreateAdresse;
  ...
begin
  with cds_adresse do
  begin
    Append;
    DoNamen;
    DoOrte;
    DoUpdate(cds_adresse);
  end; {with cds_adresse do}
end; {procedure Tdm.CreateAdresse}
```

In der Tabelle *cds_adresse* wird ein neuer Datensatz eingefügt. Mit den lokalen Prozeduren *DoNamen* und *DoOrte* – wir werden sie gleich besprechen – werden die Daten zugewiesen. *DoUpdate* trägt die Daten dann in die Datenbank ein.

```
function Tdm.DoUpdate(DataSet: TClientDataSet): boolean;
begin
  result := false;
  with DataSet do
  begin
    try
      Post;
      ApplyUpdates(-1);
      Close;
      Open;
      result := true;
    except
      Cancel;
    end;
  end; {with cds_mitarbeiter do}
end; {procedure Tdm.DoUpdate}
```

Die Funktion *DoUpdate* versucht, die Methode *Post* aufzurufen, anschließend die Änderungen zum Server zu übertragen und mit *Open* und *Close* die Datenmenge zu aktualisieren. Gelingt das alles, gibt die Funktion *true* zurück. Im Falle, dass eine Exception auftritt, wird der Vorgang mit *Cancel* abgebrochen.

```
  procedure DoNamen;
  begin
    DoRandom(cds_namen);
    if cds_namen['Geschlecht'] = 'w'
      then cds_adresse['Anrede'] := 'Frau'
      else cds_adresse['Anrede'] := 'Herr';
    cds_adresse['vorname'] := cds_namen['vorname'];
    DoRandom(cds_namen);
    cds_adresse['nachname'] := cds_namen['nachname'];
    DoRandom(cds_namen);
    cds_adresse['strasse'] := cds_namen['straße']
      + ' ' + IntToStr(random(300) + 1);
  end; {procedure DoNamen}
```

Die lokale Prozedur *DoNamen* wählt mit *DoRandom* jeweils einen Datensatz aus, die entsprechende Spalte wird dann von *cds_namen* nach *cds_adresse* kopiert. Aus dem Geschlecht wird eine Anrede generiert und der Straßenname wird mit einer Hausnummer ergänzt.

```
procedure Tdm.DoRandom(DataSet: TClientDataSet);
begin
  DataSet.RecNo := random(DataSet.RecordCount) + 1;
end;
```

Die Prozedur *DoRandom* positioniert den Satzzeiger auf einen zufällig gewählten Datensatz der übergebenen Datenmenge.

```
  procedure DoOrte;
  begin
    DoRandom(cds_orte);
    cds_adresse['plz'] := cds_orte['plz'];
    cds_adresse['ort'] := cds_orte['ort'];
    if Random(10) = 1
      then cds_adresse['tel'] := GetTelefon(cds_orte['vorwahl']);
  end; {procedure DoOrte}
```

Da bei den Orten Ortsname, Postleitzahl und Vorwahl zusammenpassen sollen, wird *DoRandom* nur einmal aufgerufen. Für das Erzeugen einer Telefonnummer wird die lokale Prozedur *GetTelefon* verwendet.

```
function GetTelefon(Vorwahl: string): string;
var
  i, l: integer;
  s: string;
begin
  l := Length(Vorwahl);
  case l of
    3:  for i := 1 to 9 do
          if (i = 4) or (i = 7)
            then s := s + ' '
            else s := s + IntToStr(Random(8) + 1);
    4:  for i := 1 to 7 do
          if i = 4
            then s := s + ' '
            else s := s + IntToStr(Random(8) + 1);
    5:  for i := 1 to 6 do
          if i = 3
            then s := s + ' '
            else s := s + IntToStr(Random(8) + 1);
  end; {case l of}
  result := Vorwahl + ' / ' + s;
end; {function GetTelefon}
```

Je nach Länge der Vorwahl wird die Rufnummer unterschiedlich lang. Darüber hinaus wird sie ein- oder zweimal mit einem Leerzeichen unterbrochen. Dies erhöht die Lesbarkeit, vereinfacht aber nicht die Suche. Was in der Praxis eher ein Gegenargument wäre, ist für uns eine Begründung – schließlich können wir dann sehen, wie sich eine solche Suche realisieren lässt.

2.2.3 Die Mitarbeiter-Tabelle

Die Tabelle *t_mitarbeiter* wird nach demselben Muster, jedoch deutlich einfacher aufgebaut.

```
procedure Tdm.CreateMitarbeiter;
begin
  with cds_mitarbeiter do
  begin
    Append;
    cds_mitarbeiter['id'] := GetID(sql_mitarbeiter_id);
    DoRandom(cds_namen);
    cds_mitarbeiter['vorname'] := cds_namen['vorname'];
    DoRandom(cds_namen);
    cds_mitarbeiter['nachname'] := cds_namen['nachname'];
```

```
    DoUpdate(cds_mitarbeiter);
  end; {with cds_mitarbeiter do}
end; {procedure Tdm.CreateMitarbeiter}
```

2.2.4 Die Produkte

Für die Tabelle *t_produkt* wird zunächst eine Kategorie und – dank Master-Detail-Verknüpfung – ein passender Titel generiert. Für den Namen des Fotografen bemühen wir wieder unsere Namenstabelle, wobei wir hier Vor- und Nachname in einer Spalte vereinen. Zuletzt wird noch ein Preis zwischen 10 und 30 Euro festgelegt.

```
procedure Tdm.CreateProdukt;
var
  s: string;
begin
  with cds_produkt do
  begin
    Append;
    cds_produkt['id'] := GetID(sql_produkt_id);
    DoRandom(cds_kat);
    cds_produkt['kat'] := cds_kat['name'];
    DoRandom(cds_poster);
    cds_produkt['titel'] := cds_poster['name'];
    DoRandom(cds_namen);
    s := cds_namen['vorname'];
    DoRandom(cds_namen);
    cds_produkt['photograph'] := s + ' ' + cds_namen['nachname'];
    cds_produkt['preis'] := 10 + random(2000) / 100;
    DoUpdate(cds_produkt);
  end; { with cds_produkt do }
end; {procedure Tdm.CreateProdukt}
```

2.2.5 Die Bestellungen

Zuletzt wollen wir noch die Bestellungen und deren Posten generieren:

```
procedure Tdm.CreateBestellung;
var
  i, j: integer;
begin
  with cds_bestellung do
  begin
    Append;
    DoRandom(cds_adresse);
```

```
    cds_bestellung['kunde'] := cds_adresse['id'];
    DoRandom(cds_mitarbeiter);
    cds_bestellung['mitarbeiter'] := cds_mitarbeiter['id'];
    cds_bestellung['datum']
      := StrToDate('01.01.1996') + random(2500);
    if not DoUpdate(cds_bestellung)
      then exit;
  end; {with cds_bestellung do}

  with cds_posten do
  begin
    j := random(10) + 2;
    for i := 1 to j do
    begin
      Append;
      cds_posten['bestellung'] := cds_bestellung['id'];
      cds_posten['pos'] := i;
      DoRandom(cds_produkt);
      cds_posten['produkt'] := cds_produkt['id'];
      cds_posten['stueckzahl'] := random(12) + 1;
      if i = j
        then DoUpdate(cds_posten)
        else Post;
    end; {for i := 1 to j do}
  end; {with cds_posten do}
end; {procedure Tdm.CreateBestellung}
```

Für die Tabelle *t_bestellung* werden zunächst ein Kunde und ein Bearbeiter gewählt, außerdem wird ein Datum erzeugt. Gibt *DoUpdate* den Wert *true* zurück, dann werden bis zu elf Posten dieser Bestellung generiert. Die Schleifenvariable *i* wird dabei in die Spalte *pos* geschrieben, für Produkt und Stückzahl werden wieder zufällige Werte erzeugt.

Die einzelnen Posten werden dann (auf »gut Glück«) mit Post in die Client-Datenmenge übernommen, beim letzten wird dann mit *DoUpdate* alles zur Datenbank übertragen.

2.2.6 Die Grundfüllung

Man könnte nun die Tabellen nacheinander füllen, hätte dann aber eine Testdatenbank, in der die Datensätze »brav« hintereinander stehen. Dies will man normalerweise so haben (und das lässt sich mit Backup und Restore auch bewerkstelligen), für Testzwecke bevorzugen wir jedoch die realitätsnähere Anordnung »wild durcheinander«. Zu diesem Zweck werden wir eine Prozedur erstellen, welche die

einzelnen Tabellen in zufälliger Reihenfolge füllt. Dabei werden wir auch berücksichtigen, dass – zumindest bei einer wirtschaftlich arbeitenden Firma – öfters ein Auftrag eingeht, als ein Mitarbeiter eingestellt wird.

Da nun der Geschäftsbetrieb nicht ohne ein Minimum von Mitarbeitern und Produkten beginnen kann, wollen wir uns auch um die Grundfüllung der Tabellen kümmern:

```
procedure TForm1.btnGrundfuellungClick(Sender: TObject);
var
  i: integer;
begin
  dm.CreateMitarbeiter;
  dm.CreateMitarbeiter;
  dm.CreateMitarbeiter;
  for i := 0 to 100
    do dm.CreateProdukt;
  for i := 0 to 100
    do dm.CreateAdresse;
end; {procedure TForm1.btnGrundfuellungClick}
```

Zunächst stellen wir drei Mitarbeiter ein, anschließend füllen wir unser Sortiment und ein paar neue Kunden wollen wir auch schon erzeugen.

2.2.7 Datensätze erzeugen

Nun wollen wir uns um die »realitäts-angenäherte« Füllung unserer Datenbank kümmern:

```
procedure TForm1.btnErzeugenClick(Sender: TObject);
var
  i, j: integer;
begin
  FAbbruch := false;
  CheckBox1.Checked := false;
  dm.SQLConnection1.StartTransaction(FDesc);
  try
    for i := 1 to StrToIntDef(Edit1.Text, 1) do
    begin
      j := random(100000);
      case j of
        1: dm.CreateMitarbeiter;
        2..100: dm.CreateProdukt;
        101..10000: dm.CreateAdresse;
      else
        dm.CreateBestellung;
      end;
```

```
      if FAbbruch
        then Break;
      Label1.Caption := IntToStr(i);
      Application.ProcessMessages;
    end; {for i := 0 to StrToIntDef(Edit1.Text, 1) do}
    dm.SQLConnection1.Commit(FDesc);
  except
    dm.SQLConnection1.Rollback(FDesc);
  end;
  CheckBox1.Checked := true;
end; {procedure TForm1.btnErzeugenClick}
```

FAbbruch und *CheckBox1* sollen unsere Aufmerksamkeit noch ein wenig entbehren. Damit nicht für jeden einzufügenden Datensatz eine eigene Transaktion gestartet wird, beginnen wir zunächst explizit eine eigene Transaktion.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  randomize;
  FDesc.TransactionID := 12345;
  FDesc.IsolationLevel := xilREADCOMMITTED;
end;
```

Die Transaktionsbeschreibung *FDesc* füllen wir in der Prozedur *FormCreate*, dort findet sich auch die von alten »Pascal-Hasen« sicher schon vermisste Routine *randomize*. Die *TransactionID* ist ein willkürlich gewählter Wert und unser *IsolationLevel* ist *ReadCommitted*.

Weiter in der Routine *btnErzeugenClick*: Wie oft wir Datensätze einfügen, geben wir mit *Edit1* an, entsprechend oft wird die *for..to..do*-Schleife durchlaufen. Für die Entscheidung, welche Tabelle wir füllen, weisen wir *j* einen zufälligen Wert zu und verzweigen dann mit einer *case*-Anweisung. Auf diese Weise können wir problemlos verschiedene Häufigkeiten verwenden.

Wird *FAbbruch* auf *true* gesetzt, dann soll der Vorgang abgebrochen werden. *FAbbruch* wird mittels eines Buttons auf *true* gesetzt, und damit das nicht erst nach dem Abarbeiten der Schleife passiert, brauchen wir einen Aufruf von *Application.ProcessMessages*. Bei dieser Gelegenheit zeigen wir auch an, wie weit wir sind.

Zum Schluss wird die Transaktion mit *Commit* bestätigt oder – wenn bis hierhin tatsächlich eine Exception nicht abgefangen wurde – mit *Rollback* zurückgenommen.

Um die Rechenzeit für die Aktualisierung der Anzeige zu sparen, werden die Grids von den DataSourcen abgeklemmt. Man kann jedoch »zwischendrin« mal neugierig sein:

```
procedure TForm1.CheckBox1Click(Sender: TObject);
begin
  if CheckBox1.Checked then
  begin
    DBGrid1.DataSource := dm.ds_namen;
      ...
    DBGrid9.DataSource := dm.ds_posten;
  end  {if CheckBox1.Checked then}
  else
  begin
    DBGrid1.DataSource := nil;
      ...
    DBGrid9.DataSource := nil;
  end; {else CheckBox1.Checked then}
end; {procedure TForm1.CheckBox1Click}
```

2.3 Verbesserung der Performance

Solange die Tabellen noch vergleichsweise leer sind, arbeitet das Programm mit akzeptabler Geschwindigkeit – zumal die Erstellung eines großen Datenbestandes ohnehin nach »Nachtlauf« ruft. Sobald jedoch mal einige MByte erreicht sind, wird die Sache richtig zäh. Wie zäh, hängt natürlich von der vorhandenen Rechenleistung ab, bei mir dauerte das Eintragen eines einzelnen Datensatzes dann knapp vier Sekunden.

Der Schluss, dass InterBase für die Verwaltung großer Datenmengen nicht geeignet ist, wäre allerdings etwas voreilig – das Problem liegt einzig daran, wie wir unseren Client programmiert haben: Hier wurde nämlich wie bei einer Desktop-Datenbank gedacht: Ich habe einen neuen Datensatz, den trage ich ein, und dann aktualisiere ich meine Datenmenge.

Das Eintragen des Datensatzes wäre dabei noch gar nicht einmal das Problem, aber wenn nach jedem neuen Datensatz ein SELECT auf eine Tabelle mit zigtausend Datensätzen ausgeführt wird, dann erhöht das verständlicherweise nicht die Performance.

Wir werden nun einen anderen Weg gehen: Die neuen Datensätze werden nur in die Client-Datenmengen eingetragen. Und wenn dann tausend Datensätze beisammen sind, dann werden diese »in einem Rutsch« auf den Server geschrieben. Diese Vorgehensweise bedingt jedoch, dass wir die IDs nicht erst auf dem Server schreiben, sondern schon auf dem Client. Sonst würden – vor allem in der Datenmenge *cds_posten* – Referenzen auf Datensätze ohne ID gebildet und das führt geradewegs zu einer Exception.

2.3.1 Änderungen der Datenbank

Wenn wir jedoch die IDs bereits auf dem Client zuweisen, dann darf dies auf dem Server nicht wieder passieren, sonst stimmt keine Referenz mehr. Die Trigger werden also nach dem folgenden Muster umgebaut:

```
SET TERM^;

CREATE TRIGGER "TRIG_ADRESSEN" FOR "T_ADRESSEN"
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL)
    THEN NEW.ID = GEN_ID(g_ADRESSEN, 1);
END^

ALTER PROCEDURE "P_ADRESSEN_ID"
RETURNS("GEN" INTEGER)
AS
BEGIN
  GEN = GEN_ID(g_ADRESSEN, 1);
  SUSPEND;
END^

SET TERM;^
```

Bei dieser Gelegenheit erstellen wir gleich noch eine STORED PROCEDURE zum Zugriff auf den Generatorwert. Für die anderen drei Tabellen wird analog vorgegangen.

2.3.2 Änderung des Clients

Für den Zugriff auf die STORED PROCEDURE wird nun eine *TSQLDataSet*-Instanz namens *sql_adresse_id* verwendet:

- *SQLConnection* setzen wir auf *SQLConnection1*.
- Die Eigenschaft *CommandType* erhält den Wert *ctQuery,* als *CommandText* verwenden wir *select * from P_ADRESSEN_ID*.

Auch hier gehen wir für die anderen drei Tabellen entsprechend vor.

```
procedure Tdm.CreateAdresse;
begin
  with cds_adresse do
  begin
    Append;
    cds_adresse['id'] := GetID(sql_adresse_id);
```

```
    DoNamen;
    DoOrte;
    Post;
  end; {with cds_adresse do}
end; {procedure Tdm.CreateAdresse}
```

Nach dem Hinzufügen eines neuen Datensatzes holen wir erst einmal über *GetID* einen neuen Wert für die Spalte ID. Statt *DoUpdate* wird dann nur noch *Post* aufgerufen.

```
function Tdm.GetID(DataSet: TSQLDataSet): integer;
begin
  with DataSet do
  begin
    Close;
    Open;
    result := Fields[0].AsInteger;
  end;
end; {function Tdm.GetID}
```

Mit *GetID* holen wir mit der jeweils übergebenen Datenmenge einen neuen Generatorwert.

```
procedure TForm1.btnErzeugenClick(Sender: TObject);
var
  i, j: integer;
begin
  FAbbruch := false;
  CheckBox1.Checked := false;
  for i := 0 to StrToIntDef(Edit1.Text, 1) do
  begin
    j := random(100000);
    case j of
      1: dm.CreateMitarbeiter;
      2..100: dm.CreateProdukt;
      101..10000: dm.CreateAdresse;
    else
      dm.CreateBestellung;
    end;
    if i mod 1000 = 0
      then DoEintragen;
    if FAbbruch
      then Break;
    Label1.Caption := IntToStr(i);
    Application.ProcessMessages;
  end; {for i := 0 to StrToIntDef(Edit1.Text, 1) do}
```

```
  DoEintragen;
  CheckBox1.Checked := true;
end; {procedure TForm1.btnErzeugenClick}
```

Die Routine *btnErzeugenClick* beginnt nun keine Transaktion mehr, außerdem wird nach jeweils 1000 Datensätzen und am Ende *DoEintragen* aufgerufen.

```
  procedure DoEintragen;
  begin
    try
      dm.SQLConnection1.StartTransaction(FDesc);
      dm.cds_adresse.ApplyUpdates(-1);
      dm.cds_mitarbeiter.ApplyUpdates(-1);
      dm.cds_produkt.ApplyUpdates(-1);
      dm.cds_bestellung.ApplyUpdates(-1);
      dm.cds_posten.ApplyUpdates(-1);
      dm.SQLConnection1.Commit(FDesc);
    except
      dm.SQLConnection1.Rollback(FDesc);
    end;
  end; {procedure DoEintragen}
```

DoEintragen startet nun eine neue Transaktion, ruft für die beteiligten Datenmengen *ApplyUpdates* auf und bestätigt dann (hoffentlich) die Transaktion mit *Commit*. Beachten Sie bitte, dass diejenigen Datensätze, auf die Referenzen gebildet werden, vor denjenigen eingetragen werden müssen, die solche Referenzen bilden.

3 InterBase SQL

SQL steht für *Structured Query Language*, auf Deutsch etwa *strukturierte Abfragesprache* – eine gewaltige Untertreibung: Der SQL-Sprachumfang umfasst nicht nur die Anweisung für Datenbankabfragen (SELECT), sondern auch zum Ändern dieser Daten (INSERT, UPDATE, DELETE), ja gar zur Definition der ganzen Datenbank (CREATE TABLE und viele andere).

Dabei ist SQL der einzig relevante Standard für Datenbanken. Bei so gut wie allen Client-Server-Systemen wird die Kommunikation zwischen Client und Server über SQL abgewickelt und auch Desktop-Datenbanksysteme wie beispielsweise dBase oder Access lassen sich per SQL-Anweisung steuern. Wenn Sie mit Datenbanken zu tun haben, kommen Sie an SQL kaum vorbei.

Obwohl es bei SQL verschiedene Standards gibt und viele Hersteller bei den Details auch eigene Wege gehen, implementieren eigentlich alle Systeme die SQL-1-DDL-Anweisungen SELECT, INSERT, UPDATE und DELETE. Wenn Sie SQL beherrschen, können Sie an jeder SQL-fähigen Datenbank mit den grundlegenden Arbeiten beginnen, ohne auch nur einen Blick ins Handbuch zu werfen – so, wie Sie bei einem Auto auch sofort losfahren können, ohne sich in der Anleitung davon zu überzeugen, dass das Gaspedal wirklich ganz rechts ist.

3.1 Kleine SQL-Geschichte

Bevor wir uns in die Praxis »stürzen«, soll für diejenigen, die daran interessiert sind, noch ein kleiner geschichtlicher Abriss von SQL erfolgen. Für das Verständnis der kommenden Kapitel ist die Kenntnis des hier Dargestellten jedoch nicht erforderlich.

SEQUEL

Anfang der 70er Jahre, also etwa zehn Jahre vor dem Beginn des PC-Zeitalters, hat man bei IBM die Bedeutung des relationalen Datenbankmodells erkannt und im Rahmen des *System/R-Projekts* mit der Umsetzung begonnen. In diesem Zusammenhang wurde die Sprache SEQUEL (»Structured English Query Language«) entwickelt. Später wurde die Sprache dann in SQL (»Structured Query Language«, in der Literatur manchmal auch »Standard Query Language«) umbenannt.

SQL1

Zu Beginn der 80er Jahre begann das *American National Standards Institute* (ANSI) mit der Standardisierung dieser Sprache. 1986 wurde der Standard *SQL1* (auch *SQL86*) verabschiedet.

SQL1 implementierte bereits alle DML (Data Manipulation Language)-Anweisungen (SELECT, INSERT, UPDATE und DELETE). Auch waren bereits die SELECT-Klauseln WHERE, GROUP BY und HAVING implementiert, ebenso die Aggregatfunktionen (COUNT, SUM, MIN, MAX, AVG). Außerdem kannte man Unterabfragen und in diesem Zusammenhang die Operatoren ALL, ANY und EXISTS. Auch den INNER JOIN kannte man schon, wenn man ihn damals auch noch nicht so genannt hat.

Die Gruppe der DDL (Data Definition Language)-Anweisungen war allerdings noch recht übersichtlich (CREATE TABLE, CREATE VIEW und GRANT sowie die dazugehörenden DROP-Befehle).

SQL89

Mit der am 3. Oktober 1989 veröffentlichten Norm ANSI X3135-1989 wurde der SQL-Standard um die Möglichkeit von Integritätsprüfungen erweitert, der neue Standard wird allgemein SQL89 genannt.

SQL89 implementiert unter anderem die CHECK-Klausel, also die Möglichkeit, Gültigkeitsprüfungen für einzufügende Werte vorzunehmen, darüber hinaus Primär-, Sekundär- und Fremdschlüssel.

SQL2

Mit dem Standard SQL2 wurde der Sprachumfang deutlich erweitert (das »Normblatt« besteht aus etwa 500 Seiten ...). Unter den Neuerungen befinden sich:

- Domänen, also vordefinierte Typen. Außerdem wurden einige neue Datentypen eingeführt.
- Die Möglichkeit, mit ALTER TABLE Tabellendefinitionen zu ändern.
- Der OUTER JOIN sowie die Möglichkeit, den INNER JOIN auch so zu nennen.
- Transaktionen.

Die meisten Datenbanksysteme unterstützen noch nicht vollständig den SQL2-Standard, gehen andererseits mit ihren Möglichkeiten meist aber auch über diesen Standard hinaus.

SQL3

Mit dem Standard SQL3 (verabschiedet im Dezember 1999, deshalb auch oft SQL 99 genannt) wird der Sprachumfang nochmals erweitert. Neu hinzugekommen ist unter anderem die Möglichkeit, mit SQL auch objektorientiert zu programmieren.

3.2 InterBase SQL

Wir wollen uns nun in diesem und in den folgenden Kapiteln mit *InterBase SQL* beschäftigen, einem SQL-Dialekt, der den Standard recht genau umsetzt, darüber hinaus jedoch noch einige weitere Möglichkeiten bietet.

Die SQL-Anweisungen werden normalerweise in zwei Gruppen eingeteilt:

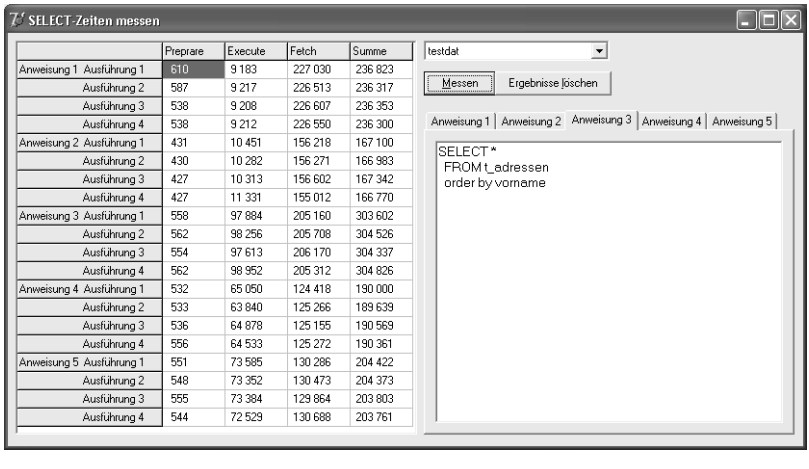
- Die *Data Manipulation Language* (DML) umfasst alles, was mit den eigentlichen Daten zu tun hat, also die Befehle SELECT, INSERT, UPDATE und DELETE.
- Mit der *Data Definition Language* (DDL) werden die Metadaten einer Datenbank bearbeitet. Darunter fallen beispielsweise Anweisungen zur Erstellung von Tabellen (CREATE TABLE).

In Fachbüchern wird die Einteilung bisweilen genauer vorgenommen, so dass beispielsweise die DML nur die Anweisungen INSERT, UPDATE und DELETE umfasst und der SELECT-Befehl der *Data Selection Language* (DSL) zugeordnet wird, was systematisch sicher korrekter ist (SELECT ändert nun mal keine Daten). In der Praxis beschränkt man sich jedoch auf die Einteilung in DML und DDL.

Wir wollen in diesem Kapitel die DML besprechen, im nächsten Kapitel soll dann der allgemeine Teil der DDL das Thema sein, die Besonderheiten bei STORED PROCEDURES und TRIGGER seien dann Kapitel 5 vorbehalten.

3.2.1 Ein kleines Testprogramm

Möchte man eine bestimmte Ergebnismenge erhalten, dann gibt es normalerweise mehrere Möglichkeiten, eine entsprechende SQL-Anweisung zu formulieren. Diese unterscheiden sich in den Ausführungsgeschwindigkeiten, und das mitunter heftig. Somit sollte uns nicht nur die SQL-Syntax interessieren, sondern auch die Ausführungsgeschwindigkeiten der einzelnen Anweisungen. Um diese zu messen, erstellen wir ein kleines Testprogramm:



		Preprare	Execute	Fetch	Summe
Anweisung 1	Ausführung 1	610	9 183	227 030	236 823
	Ausführung 2	587	9 217	226 513	236 317
	Ausführung 3	538	9 208	226 607	236 353
	Ausführung 4	538	9 212	226 550	236 300
Anweisung 2	Ausführung 1	431	10 451	156 218	167 100
	Ausführung 2	430	10 282	156 271	166 983
	Ausführung 3	427	10 313	156 602	167 342
	Ausführung 4	427	11 331	155 012	166 770
Anweisung 3	Ausführung 1	558	97 884	205 160	303 602
	Ausführung 2	562	98 256	205 708	304 526
	Ausführung 3	554	97 613	206 170	304 337
	Ausführung 4	562	98 952	205 312	304 826
Anweisung 4	Ausführung 1	532	65 050	124 418	190 000
	Ausführung 2	533	63 840	125 266	189 639
	Ausführung 3	536	64 878	125 155	190 569
	Ausführung 4	556	64 533	125 272	190 361
Anweisung 5	Ausführung 1	551	73 585	130 286	204 422
	Ausführung 2	548	73 352	130 473	204 373
	Ausführung 3	555	73 384	129 864	203 803
	Ausführung 4	544	72 529	130 688	203 761

Abbildung 3.1: Messen der Ausführungszeiten

Das Programm erlaubt die Eingabe von bis zu fünf SQL-Anweisungen. Diese werden in jeweils vier Durchläufen gemessen, um die Streuung der Messwerte zu berücksichtigen – beispielsweise ist die allererste *Prepare*-Zeit fast immer leicht erhöht (und direkt nach dem Programmstart deutlich erhöht). Um hier nicht falsche Schlussfolgerungen zu ziehen, müssen die einzelnen Anweisungen mehrmals ausgeführt werden.

Dabei messen wir die *Prepare*-, die *Execute*- und die *Fetch*-Zeit getrennt. Die *Prepare*-Zeit ist die Zeit, welche der Server braucht, um die SQL-Anweisung zu interpretieren. Um sie dann auszuführen, wird die *Execute*-Zeit benötigt. Anschließend rufen wir alle Datensätze ab und messen dabei die *Fetch*-Zeit.

Um den Einfluss des Clients zu minimieren, wurde das Programm mit dbExpress erstellt. Zwischen den einzelnen Verbindungen kann mittels einer ComboBox umgeschaltet werden, dabei wäre auch die Verwendung von anderen Datenbanken als InterBase möglich. Die Namen der einzelnen Verbindungen werden beim Erstellen der Formulars in die ComboBox geladen – bei dieser Gelegenheit wird auch das StringGrid beschriftet:

```
procedure TForm1.FormCreate(Sender: TObject);

  procedure DoStringGrid;
  var
    i, j: integer;
  begin
    with StringGrid1 do
    begin
      ColWidths[0] := 150;
      Cells[1,0] := 'Preprare';
      Cells[2,0] := 'Execute';
      Cells[3,0] := 'Fetch';
      Cells[4,0] := 'Summe';
      for i := 0 to 4
        do for j := 0 to 3
          do if j = 0
            then cells[0, i * 4 + j + 1]
              := 'Anweisung ' + IntToStr(i + 1)
                + '  Ausführung ' + IntToStr(j + 1)
            else cells[0, i * 4 + j + 1]
              := '                      Ausführung ' + IntToStr(j + 1);
    end; {with StringGrid1 do}
  end; {procedure DoStringGrid}

  procedure DoConnections;
  var
    Ini: TIniFile;
```

```
  begin
    Ini := TIniFile.Create('C:\Programme\Gemeinsame Dateien\Borland' +
    'Shared\DBExpress\dbxconnections.ini');
    try
       Ini.ReadSections(ComboBox1.Items);
    finally
      Ini.Free;
    end;
  end; {procedure DoConnections}

begin
  DoStringGrid;
  DoConnections;
end; {procedure TForm1.FormCreate}
```

Wird ein anderer Eintrag der ComboBox gewählt, dann wird die Verbindung geschlossen, die gewählte Verbindung eingestellt und die Verbindung wieder geöffnet:

```
procedure TForm1.ComboBox1Change(Sender: TObject);
begin
  with SQLConnection1 do
  begin
    Connected := false;
    ConnectionName := ComboBox1.Text;
    LoadParamsFromIniFile();
    Connected := true;
  end; {with SQLConnection1 do}
end; {procedure TForm1.ComboBox1Change}
```

Für die einzelnen SQL-Anweisungen wird ein String-Array mit den Feldern null bis vier erstellt. Vor dem Wechsel auf eine andere TabControl-Seite wird der Inhalt von *Memo1* dort gesichert, nach dem Wechsel wird die neue Anweisung dorthin zurückgeschrieben:

```
procedure TForm1.TabControl1Changing(Sender: TObject;
  var AllowChange: Boolean);
begin
  FAnweisungen[TabControl1.TabIndex] := Memo1.Lines.Text;
end;

procedure TForm1.TabControl1Change(Sender: TObject);
begin
  Memo1.Lines.Text := FAnweisungen[TabControl1.TabIndex];
end;
```

Damit beim Messen alle SQL-Anweisungen im Array auf dem aktuellen Stand sind, wird zunächst noch einmal *TabControl1Changing* aufgerufen. Anschließend wird zunächst durch die Ausführungen und in der inneren Schleife dann durch die Anweisungen iteriert – damit soll erreicht werden, dass die verschiedenen Ausführungen derselben Anweisung nicht hintereinander gemessen werden und alles schon fertig im Cache liegt. Natürlich berücksichtigt es InterBase, wenn die Daten schon mal im Speicher liegen, das ist in der Praxis ja auch der Fall und deswegen ist der erste Messwert meist auch weniger aussagekräftig.

```
procedure TForm1.btnMeasureClick(Sender: TObject);
var
  ausf, anw: integer;
  b: boolean;
begin
  TabControl1Changing(nil, b);
  for ausf := 0 to 3
    do for anw := 0 to 4
      do Messen(anw, 4 * anw + ausf + 1);
end; {procedure TForm1.btnMeasureClick}
```

Ist keine Anweisung vorhanden, dann wird gleich am Anfang abgebrochen, andernfalls wird die Anweisung der *TSQLDataSet*-Eigenschaft *CommandText* zugewiesen. Die eigentliche Messung wird an die Anweisung *Zeitmessen* delegiert, der als Parameter auch eine parameterlose Prozedur übergeben wird. Die Gesamtzeit wird in *Summe* addiert, diese Variable wird zu Beginn mit null initialisiert.

```
procedure TForm1.Messen(Anweisung, Reihe: integer);
var
  Summe: integer;
begin
  if FAnweisungen[Anweisung] = ''
    then exit;
  Summe := 0;
  SQLDataSet1.Active := false;
  SQLDataSet1.CommandText := FAnweisungen[Anweisung];
  Zeitmessen(1, Reihe, DoPrepare, Summe);
  Zeitmessen(2, Reihe, DoExecute, Summe);
  Zeitmessen(3, Reihe, DoFetch, Summe);
  StringGrid1.Cells[4, Reihe]
    := FormatFloat('### ### ###', Summe);
  SQLDataSet1.Prepared := false;
end; {procedure TForm1.Messen}
```

Die Anweisung *DoPrepare* setzt die Eigenschaft *Prepared* auf *true*, mit *DoExecute* wird die Datenmenge geöffnet. *DoFetch* ruft alle (!) Datensätze ab:

```
procedure TForm1.DoFetch;
begin
  with SQLDataSet1
    do while not EOF
      do Next;
end;
```

Die Zeitmessung erfolgt mit QueryPerformanceCounter, so dass wir durchaus Zeiten im µs-Bereich messen können. Das Ergebnis wird gleich in das StringGrid geschrieben, und damit dessen Anzeige auch gleich aktualisiert wird, rufen wir *Application.ProcessMessages* auf.

```
procedure TForm1.Zeitmessen(ACol, ARow: integer; Routine: TTestProc;
  var ASumme: integer);
var
  c, t1, t2: int64;
  i: integer;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  Routine;
  QueryPerformanceCounter(t2);
  i := 1000000 * (t2 - t1) div c;
  StringGrid1.Cells[ACol, ARow]
    := FormatFloat('### ### ###', i);
  ASumme := ASumme + i;
  Application.ProcessMessages;
end;
```

Die Falle »No Metadata«

Lassen Sie die Eigenschaft *NoMetadata* von *SQLDataSet1* vorerst auf *false* und setzen Sie dafür *WantTabs* von *Memo1* auf *true*. Verbinden Sie über *ComboBox1* mit IBLocal und geben Sie die folgenden beiden SQL-Anweisungen ein:

```
SELECT *
FROM employee

SELECT *
  FROM employee
```

Es ist essenziell, dass Sie FROM mit einem Tab- und nicht mit Leerzeichen einrücken! Schauen wir uns die Ausführungszeiten an:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	690	20 558	270	21 518
Ausführung 2	510	20 850	269	21 629
Ausführung 3	464	20 752	281	21 497
Ausführung 4	467	20 637	271	21 375
Anweisung 2 Ausführung 1	545	2 011	270	2 826
Ausführung 2	550	2 013	269	2 832
Ausführung 3	573	2 031	270	2 874
Ausführung 4	546	2 065	270	2 881

Abbildung 3.2: Die Auswirkungen eines Tabulator-Zeichens

Ich gebe ja gerne zu, dass ich anfangs meinen Augen nicht getraut habe – das Einfügen eines Tabulator-Zeichens beschleunigt die *Execute*-Zeit um etwa den Faktor zehn. Die Ursache ist dann relativ trivial, wenn man sich das Monitor-Protokoll ansieht: Sofern ein Tab-Zeichen eingefügt wird, verzichtet *TSQLDataSet* darauf, die Metadaten abzurufen.

Denselben Effekt erhält man auch, wenn man die Eigenschaft *NoMetadata* von *SQLDataSet1* auf *true* setzt, was Sie nun schleunigst tun sollten.

3.3 Die SELECT-Anweisung

Mit dem SELECT-Befehl werden die Daten aus der Datenbank abgefragt. Der vollständig abstrakte SELECT-Befehl lautet wie folgt:

```
SELECT [columns]
  FROM [tables]
  WHERE [search_conditions]
  GROUP BY [columns] HAVING [search_conditions]
  UNION [select-statement]
  PLAN [query-plan]
  ORDER BY [sort_orders]
```

Es sind jedoch nicht alle Elemente zwingend erforderlich, sondern nur SELECT und FROM.

3.3.1 Spalten

Die Messung von Ausführungsgeschwindigkeiten ist erst dann richtig interessant, wenn größere Datenmengen verwendet werden. Wir wollen deshalb als Testdatenbank die in Kapitel 2 erstellte Testdatenbank *testdat.gdb* und nicht die *employee.gdb* verwenden. Als Client verwenden Sie *IBConsole* oder ein ähnliches Tool.

Beginnen wir mit einer minimalistischen SELECT-Anweisung:

```
SELECT vorname, nachname
  FROM t_adressen
```

Wenn Sie *IBConsole* verwenden, dann rufen Sie *InteractiveSQL* auf, geben die Anweisung dort ein und führen sie mit QUERY | EXECUTE aus. Die SQL-Anweisung wird dann gelöscht und die Ergebnismenge angezeigt.

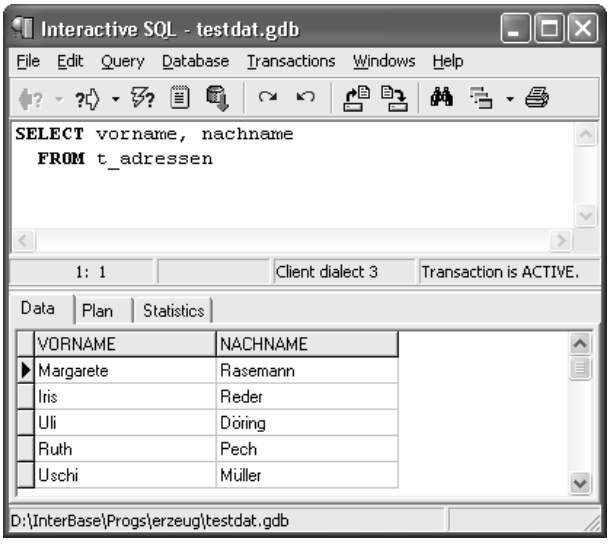


Abbildung 3.3: Die Anweisung in IBConsole

Was sind nun die Elemente dieser SELECT-Anweisung?

- Zunächst einmal das Schlüsselwort SELECT. Damit tun wir kund, dass wir Daten abfragen möchten.
- Dann nennen wir die Spaltennamen der beiden Spalten, deren Daten wir haben möchten.
- Anschließend folgt das Schlüsselwort FROM. Der Rechner weiß nun, dass wir die Tabelle(n) nennen werden, aus denen wir die Daten haben möchten. Das tun wir dann auch.

Ins Deutsche übersetzt heißt diese Anweisung: Zeige mir die Daten der Spalten *vorname* und *nachname* an, beide in der Tabelle *t_mitarbeiter* zu finden.

Schauen Sie sich noch einmal genau die Anweisung an:

```
SELECT vorname, nachname
  FROM t_adressen
```

- Vielleicht ist Ihnen aufgefallen, dass zwei Wörter durchgehend mit Großbuchstaben geschrieben worden sind, nämlich SELECT und FROM. Dabei handelt es sich um so genannte SQL-Schlüsselwörter. Alle Tätigkeiten, die ein Datenbank-Management-System beherrscht, werden über solche Schlüsselwörter aufgerufen. Derer werden wir noch viele kennen lernen.

  Diese Schlüsselwörter sollten Sie tunlichst vermeiden, wenn Sie selbst Spalten, Tabellen oder was auch immer benennen, das DBS kommt sonst ganz fürchterlich durcheinander.

  Die Großschreibung der SQL-Schlüsselwörter ist freiwillig. Man erkennt aber viel einfacher die Syntax einer Anweisung und findet so auch viel schneller eventuelle Fehler. Bei komplizierten Anweisungen werden Sie dafür dankbar sein. Ebenso können Zeilenumbrüche und Einrückungen nach Belieben vorgenommen werden.

- Kleingeschrieben werden dagegen alle Bezeichner, also alles, was irgendein Benutzer selbst benannt hat: Spalten und Tabellen, VIEWS, STORED PROCEDURES und noch vieles andere mehr. Bei der Vergabe der Bezeichner sind Sie relativ frei. Lediglich SQL-Schlüsselwörter sowie Leer- und Sonderzeichen sollten Sie vermeiden.

  Leider setzt es sich bei einigen SQL-Generatoren durch, die Bezeichner groß und die SQL-Schlüsselwörter klein zu schreiben. Natürlich ist das auch eine Frage des persönlichen Geschmacks, aber die Struktur einer SQL-Anweisung lässt sich viel leichter überblicken, wenn die Bezeichner kleingeschrieben werden.

- Wie Ihnen sicher schon aufgefallen ist, lautet der Tabellenname *t_adressen*. Das vorangestellte *t_* zeigt an, dass es sich um eine Tabelle handelt. Auch diese Maßnahme ist freiwillig und soll lediglich die Übersichtlichkeit erhöhen.

- Vielleicht haben Sie sich auch schon gefragt, welche Datensätze denn aus der Tabelle *t_adressen* angezeigt werden. Nun, solange Sie das nicht irgendwie beschränken, werden alle Datensätze angezeigt. Wenn es sich dabei um einige wenige Datensätze handelt, dann ist das nicht weiter problematisch. Auch bei großen Tabellen werden für gewöhnlich nur so viele Datensätze abgerufen, wie auch tatsächlich angezeigt werden. Nur in Ausnahmefällen werden wirklich alle Datensätze zum Client übertragen – das wird dann aber dauern.

Alle Spalten anzeigen

Als Nächstes sollen alle Spalten der Tabelle *t_mitarbeiter* angezeigt werden. Dazu könnte man wie folgt vorgehen:

```
SELECT id, anrede, vorname, nachname, strasse, plz, ort, tel
  FROM t_adressen
```

Nun ist das etwas viel Schreibarbeit. Das haben auch die Entwickler von SQL eingesehen und ein Joker-Zeichen definiert:

```
SELECT *
  FROM t_adressen
```

Allerdings haben Sie dann keinen Einfluss mehr auf die Reihenfolge, in der die Spalten angezeigt werden.

Nun wollen wir neugierig werden und testen, ob die Verwendung des Jokerzeichens einen Einfluss auf die Performance hat. Die erste Anweisung ist mit Jokerzeichen, die zweite listet die Spalten auf.

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	641	2 032	226 594	229 267
Ausführung 2	537	1 971	226 781	229 289
Ausführung 3	512	1 954	226 840	229 306
Ausführung 4	518	1 957	227 601	230 076
Anweisung 2 Ausführung 1	485	1 889	228 530	230 904
Ausführung 2	479	1 871	227 182	229 532
Ausführung 3	484	1 870	227 774	230 128
Ausführung 4	482	1 870	227 035	229 387

Abbildung 3.4: Einfluss des Jokerzeichens

Wie Abbildung 3.4 zeigt, erhöht die Spaltenliste ganz leicht die *Prepare*-Zeit – schließlich muss hier mehr interpretiert werden – und die *Execute*-Zeit. Die *Fetch*-Zeiten sind in etwa dieselben, schließlich werden dieselben Daten übertragen. Wenn Sie die Reihenfolge der Spalten in der Spaltenliste wild vertauschen, so hat dies keinen signifikanten Effekt.

Nun stellen wir uns vor, dass wir für ein Rundschreiben Etiketten aller Kunden drucken wollen. Dafür benötigen wie die Spalten bis auf ID und Tel. Man könnte nun eine entsprechende Spaltenliste formulieren, man könnte aber auch aus Bequemlichkeit das Jokerzeichen verwenden und dann nicht alle Spalten verwenden. Die erste Anweisung listet die Spalten auf, die zweite ist mit Joker-Zeichen:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	611	2 047	202 361	205 019
Ausführung 2	488	1 992	203 432	205 912
Ausführung 3	483	1 987	202 584	205 054
Ausführung 4	486	1 990	203 250	205 726
Anweisung 2 Ausführung 1	490	1 872	228 248	230 610
Ausführung 2	491	1 868	227 038	229 397
Ausführung 3	487	1 867	227 192	229 546
Ausführung 4	489	1 991	227 251	229 731

Abbildung 3.5: Kürze der Spaltenliste

Hier haben wir nun nicht nur die bereits besprochenen Auswirkungen bezüglich der *Execute*-Zeit, da mit einer kürzeren Spaltenliste auch weniger Daten übertragen werden müssen, sinkt auch die *Fetch*-Zeit.

Fazit: Wenn Sie eine Abfrage in Hinsicht der Performance optimieren wollen, dann kann es helfen, Joker-Zeichen zu vermeiden. Das Joker-Zeichen hat jedoch den Vorteil, dass später hinzugekommene Spalten mit in die Ergebnismenge aufgenommen werden.

Keine doppelten Datensätze

Als Nächstes tun wir so, als ob es uns interessiert, welche Postleitzahlen denn die Kunden haben:

```
SELECT plz
  FROM t_adressen
```

Bei genauer Betrachtung des Ergebnisses stellen wir fest, dass einige Postleitzahlen mehrmals auftreten. (Später werden Sie lernen, die Häufigkeit des jeweiligen Auftretens zu zählen.)

Nun interessiert uns hier eher, in welchem Bereich die Postleitzahlen überhaupt liegen. Hier hilft uns dann das Schlüsselwort DISTINCT weiter.

```
SELECT DISTINCT plz
  FROM t_adressen
```

DISTINCT erkennt schon bei einem Unterschied in nur einer Spalte, dass unterschiedliche Datensätze vorliegen. Somit bleibt DISTINCT wirkungslos, wenn eine Schlüsselspalte (oder eine Spaltenkombination, die einen Schlüssel bildet) in die Spaltenliste aufgenommen wird.

Beobachten wir noch, welche Auswirkungen DISTINCT auf die Ausführungszeiten hat (die erste Anweisung ohne DISTINCT, die zweite mit):

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	1 838	12 960	132 966	147 764
Ausführung 2	441	12 793	133 475	146 709
Ausführung 3	452	13 907	134 466	148 825
Ausführung 4	446	12 853	135 086	148 385
Anweisung 2 Ausführung 1	441	82 380	74	82 895
Ausführung 2	426	68 620	76	69 122
Ausführung 3	429	67 459	75	67 963
Ausführung 4	438	67 821	77	68 336

Abbildung 3.6: Auswirkung von DISTINCT

Kurioserweise ist die Anweisung mit DISTINCT einen Hauch schneller vorbereitet. Sie braucht länger in der Ausführung, da aber deutlich weniger Daten übertragen werden müssen – eine Auswirkung der Methode, mit der wir die Testdaten erstellt haben –, ist die Fetch-Zeit um Größenordnungen geringer. In der Summe ergibt sich ein deutlicher Vorteil für die Variante mit DISTINCT, von daher sollten Sie es gar nicht in Erwägung ziehen, solche Auswertungen auf dem Client zu machen.

Konstanten

Wir können der Ergebnismenge einer SELECT-Anweisung auch Konstanten hinzufügen. Diese sind in Anführungszeichen zu setzen, damit sie von den Spaltennamen unterschieden werden können.

```
SELECT vorname, 'Mitarbeiter'
  FROM t_mitarbeiter
```

Beachten Sie bitte, dass Sie nur im SQL-Dialekt 1 einfache und doppelte Anführungszeichen synonym verwenden dürfen – beim Verwenden von SQL-Dialekt 3 müssen für Konstanten einfache Anführungszeichen verwendet werden.

Spalten zusammenfügen

Angenommen, wir wollten ein Rundschreiben an alle Kunden versenden und zu diesem Zweck Adressetiketten drucken. In der ersten Zeile wären Vorname und Nachname, in der zweiten die Straße und in der dritten die Postleitzahl und der Ort zu drucken.

Hier würde es sich nun anbieten, die betreffenden Spalten zusammenzufügen, jeweils unter Verwendung eines Leerzeichens:

```
SELECT vorname || ' ' || nachname,
    strasse,
    plz || ' ' || ort
  FROM t_adressen
```

Einen senkrechten Strich – eine so genannte Pipe – fügen Sie mit Alt Gr + < ein.

Spalten benennen

Vielleicht ist Ihnen schon aufgefallen, dass in ISQL die Spalten mit *F_1* und *F_2* benannt werden, sobald Spalten zusammengefügt oder Konstanten verwendet werden. Bei anderen Programmen, beispielsweise bei Reportgeneratoren, kommen in solchen Fällen noch unschönere Spaltenbezeichner heraus, unser Zeitmess-Programm würde gar die Ausführung verweigern. Es gibt jedoch die Möglichkeit, (nicht nur) in solchen Fällen die Spalte explizit zu benennen.

```
SELECT vorname || ' ' || nachname AS Name,
    strasse,
    plz || ' ' || ort AS Ort
  FROM t_adressen
```

Selbstverständlich können Sie auch ganz gewöhnliche Spalten umbenennen.

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	609	2 116	208 490	211 215
Ausführung 2	477	2 247	209 469	212 193
Ausführung 3	480	2 101	209 408	211 989
Ausführung 4	472	2 079	208 734	211 285
Anweisung 2 Ausführung 1	464	1 855	191 392	193 711
Ausführung 2	464	1 856	192 018	194 338
Ausführung 3	466	1 854	191 115	193 435
Ausführung 4	494	1 885	192 100	194 479

Abbildung 3.7: Zusammenfügen von Spalten

Das Zusammenfügen der Spalten sorgt nicht nur für längere *Execute*-Zeiten, sondern auch für etwa längere *Fetch*-Zeiten, was sich nicht mit zwei zusätzlichen Leerzeichen allein erklären lässt. Von daher liegt die Vermutung nahe, dass Daten erst beim Fetchen zusammengeführt werden. Wir wollen aber auch nicht aus den Augen verlieren, dass der zeitliche Mehraufwand lediglich bei etwa 9 % liegt.

Berechnungen ausführen

Innerhalb von SQL-Statements können Sie auch Berechnungen ausführen. Wir wollen nun wieder die Tabelle *t_produkt* anzeigen, neben dem gespeicherten Preis, der zuzüglich Mehrwertsteuer zu verstehen ist, soll auch der Brutto-Preis angegeben werden. Dabei wird ein Mehrwertsteuersatz von 16 % zugrunde gelegt.

```
SELECT
    kat, photograph, titel,
    preis,
    preis * 1.16 AS brutto
  FROM t_produkt
```

Unter SQL stehen die folgenden Rechenoperationen zur Verfügung:

Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division

Tabelle 3.1: Die Rechenoperatoren

Beachten Sie bitte auch, dass ein Dezimalpunkt und kein Dezimalkomma verwendet wird.

Anzeigen mit zwei Nachkommastellen

Bei der Berechnung des Nettopreises im letzten Abschnitt ist Ihnen sicher aufgefallen, dass das Ergebnis auf etwa 13 Stellen genau angezeigt wurde. Angenehmer wäre hier eine Darstellung, die auf den Cent genau – sprich auf zwei Nachkommastellen – gerundet ist.

Eine dafür geeignete Funktion kennt *InterBase* leider nicht, somit bleibt uns nur der »Griff in die Trick-Kiste«: Mit der Anweisung CAST wandelt man einen Datentyp in einen anderen um. Wir wandeln nun unsere Gleitkommazahl in einen numerischen Wert mit zwei Nachkommastellen um und schon erfolgt die Anzeige in der gewünschten Weise.

```
SELECT hersteller, bezeichnung,
    preis AS brutto,
    CAST((preis / 1.16) AS NUMERIC (15,2)) AS netto
  FROM t_produkt
```

Was mit diesem Trick leider nicht geht, ist die Darstellung des Butto-Preises auf zwei Nachkommastellen genau. Feinheiten der Formatierung sind eben nichts, womit sich ein Datenbankserver beschäftigen möchte – es bleibt, dafür eine *User Defined Function* (UDF) zu definieren.

Und nun wollen wir noch sehen, wie schnell solche Berechnungen und Umwandlungen vorgenommen werden. Aus Gründen der Vergleichbarkeit verwenden wir dazu die folgenden Anweisungen:

```
SELECT
    preis
  FROM t_produkt

SELECT
    preis * 1.16 AS brutto
  FROM t_produkt

SELECT
    CAST((preis) AS NUMERIC (15,2)) AS brutto
  FROM t_produkt

SELECT
    CAST((preis * 1.16) AS NUMERIC (15,2)) AS brutto
  FROM t_produkt
```

		Preprare	Execute	Fetch	Summe
Anweisung 1	Ausführung 1	538	2 871	347	3 756
	Ausführung 2	419	2 800	346	3 565
	Ausführung 3	410	2 805	346	3 561
	Ausführung 4	304	2 877	346	3 527
Anweisung 2	Ausführung 1	351	3 022	346	3 719
	Ausführung 2	323	3 013	346	3 682
	Ausführung 3	332	3 009	346	3 687
	Ausführung 4	378	3 015	346	3 739
Anweisung 3	Ausführung 1	365	2 970	388	3 723
	Ausführung 2	364	2 967	388	3 719
	Ausführung 3	330	2 985	388	3 703
	Ausführung 4	347	2 973	388	3 708
Anweisung 4	Ausführung 1	345	3 171	388	3 904
	Ausführung 2	348	3 170	388	3 906
	Ausführung 3	355	3 171	388	3 914
	Ausführung 4	345	3 170	418	3 933

Abbildung 3.8: Berechnungen und Typumwandlungen

Das Durchführen einer Berechnung erhöht die *Execute*-, nicht aber die *Fetch*-Zeit. Von daher ist klar, dass erst alle Datensätze berechnet und erst dann gegebenenfalls abgerufen werden. Die Typenumwandlung dagegen verlängert auch die *Fetch*-Zeiten – das liegt nun allerdings nicht an der Umwandlung selbst, sondern daran, dass NUMERIC-Daten einfach längere Fetch-Zeiten (und dafür kürzere *Execute*-Zeiten) haben als FLOAT-Daten.

Kleiner Exkurs – die Execute- und Fetch-Zeiten von Datentypen

Die eben gemachte Aussage wollen wir dann doch unter »fairen« Bedingungen verifizieren. Dazu erstellen wir eine Tabelle mit fünf Spalten, die später die Zahlen von 1 bis 9999 aufnehmen sollen:

```
CREATE TABLE t_test9
  (ID INTEGER NOT NULL PRIMARY KEY,
  f_int INTEGER,
  f_flo FLOAT,
  f_num NUMERIC(4,0),
  f_cha VARCHAR(4));
```

Die Feinheiten beim Erstellen einer Tabelle werden im nächsten Kapitel besprochen. Zum Füllen dieser Tabelle verwenden wir eine STORED PROCEDURE – hier muss ich Sie auf das übernächste Kapitel vertrösten:

```
SET TERM^;
CREATE PROCEDURE p_fuellen
AS
  DECLARE VARIABLE i INTEGER;
BEGIN
  i = 1;
  WHILE (i < 10000) DO
  BEGIN
```

```
    INSERT INTO t_test9
      VALUES(:i, :i, :i, :i, :i);
    i = i + 1;
  END
END^
SET TERM;^
EXECUTE PROCEDURE p_fuellen;
```

Nun füllen wir unseren Zeitmesser mit den folgenden Anweisungen:

```
SELECT id
  FROM t_test9

SELECT f_int
  FROM t_test9

SELECT f_flo
  FROM t_test9

SELECT f_num
  FROM t_test9

SELECT f_cha
  FROM t_test9
```

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	519	41 770	146 116	188 405
Ausführung 2	408	42 948	144 891	188 247
Ausführung 3	407	41 946	146 125	188 478
Ausführung 4	406	42 055	146 241	188 702
Anweisung 2 Ausführung 1	408	43 064	145 993	189 465
Ausführung 2	404	43 239	145 962	189 605
Ausführung 3	406	42 224	146 209	188 839
Ausführung 4	404	41 851	145 985	188 240
Anweisung 3 Ausführung 1	408	42 036	144 063	186 507
Ausführung 2	403	43 027	144 267	187 697
Ausführung 3	408	43 121	145 178	188 707
Ausführung 4	406	42 193	145 947	188 546
Anweisung 4 Ausführung 1	406	35 318	156 959	192 683
Ausführung 2	408	34 177	157 008	191 593
Ausführung 3	408	34 317	156 620	191 345
Ausführung 4	408	34 357	156 545	191 310
Anweisung 5 Ausführung 1	410	38 872	157 128	196 410
Ausführung 2	407	38 018	158 938	197 363
Ausführung 3	406	38 117	158 682	197 205
Ausführung 4	405	38 082	157 406	195 893

Abbildung 3.9: Die Zeiten bei verschiedenen Spalten-Typen

Wir sehen, dass die Typen INTEGER und FLOAT längere *Execute*-, dafür aber kürzere *Fetch*-Zeiten haben als NUMERIC und CHAR.

3.4 JOINS

Bei relationalen Datenbanken wird der Datenbestand auf viele Tabellen aufgeteilt. Um die Daten in einer brauchbaren Form zu erhalten, müssen diese Tabellen bei den Abfragen dann wieder zusammengefügt werden. Zu diesem Zweck verwendet man ein JOIN.

Es gibt mehrere Arten von JOINS, die wir nun nacheinander kennen lernen werden. Dafür verwenden wir jetzt nicht unsere Testdatenbank, sondern die Beispieldatenbank *employee.gdb*.

3.4.1 FULL JOIN

Der FULL JOIN ist in der Praxis weitgehend bedeutungslos – wir sehen gleich, warum – und wird hier nur der Vollständigkeit halber erwähnt. Der FULL JOIN ist vom Namen her recht leicht zu verwechseln mit dem FULL OUTER JOIN, den wir später kennen lernen werden.

In der Tabelle *employee_project* finden wir eine Liste der Mitarbeiter und der Projekte, an denen diese Mitarbeiter gerade arbeiten. Diese Tabelle wollen wir mit der Mitarbeiterliste verknüpfen, so dass wir nicht die Mitarbeiternummer, sondern Vor- und Nachnamen angezeigt bekommen.

```
SELECT
    employee.first_name, employee.last_name,
    employee_project.proj_id
  FROM employee, employee_project
```

In *Abbildung 3.10* sehen wir den Anfang der Ergebnismenge:

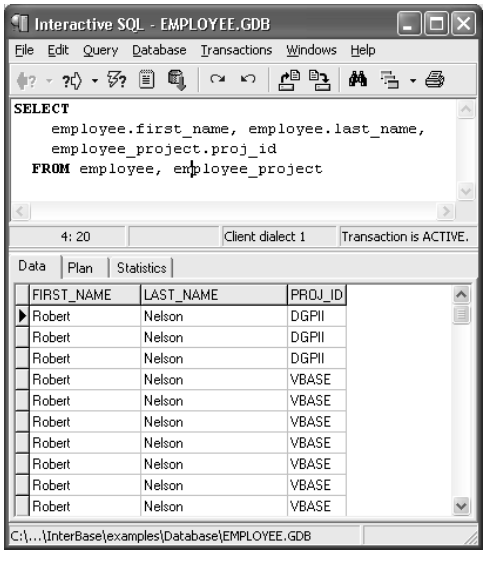


Abbildung 3.10: FULL JOIN

Wie Sie sehen, wird jeder Datensatz der einen Tabelle mit jedem Datensatz der anderen Tabelle verknüpft – jeder Mitarbeiter erhält somit alle Projekte, und dies dazu noch so oft, wie überhaupt Mitarbeiter an diesem Projekt beschäftigt sind. Ein FULL JOIN ist also nicht ganz das, was hier benötigt wird.

3.4.2 INNER JOIN

Der INNER JOIN – auch EQUI-JOIN genannt – ordnet jedem Datensatz der einen Tabellen nur die »dazugehörenden« Datensätze der anderen Tabelle zu. Nun beschäftigt das DMS keinen Hellseher: Die Information darüber, was denn unter »dazugehörend« zu verstehen ist, muss somit in der SQL-Anweisung erfolgen.

```
SELECT
    employee.first_name, employee.last_name,
    employee_project.proj_id
  FROM employee, employee_project
  WHERE employee.emp_no = employee_project.emp_no
```

Die Information darüber, wie denn die Tabellen zu verknüpfen sind, wird in der WHERE-Klausel gegeben. In unserem Beispiel geht das wie folgt vor sich: In beiden Tabellen gibt es eine Spalte *emp_no*, die Mitarbeiternummer. Zu jedem Datensatz in *employee* sucht nun der Server alle Datensätze in *employee_project* mit der gleichen Mitarbeiternummer. Wir sehen, dass Mitarbeiter durchaus auch an mehreren Projekten beschäftigt sein können:

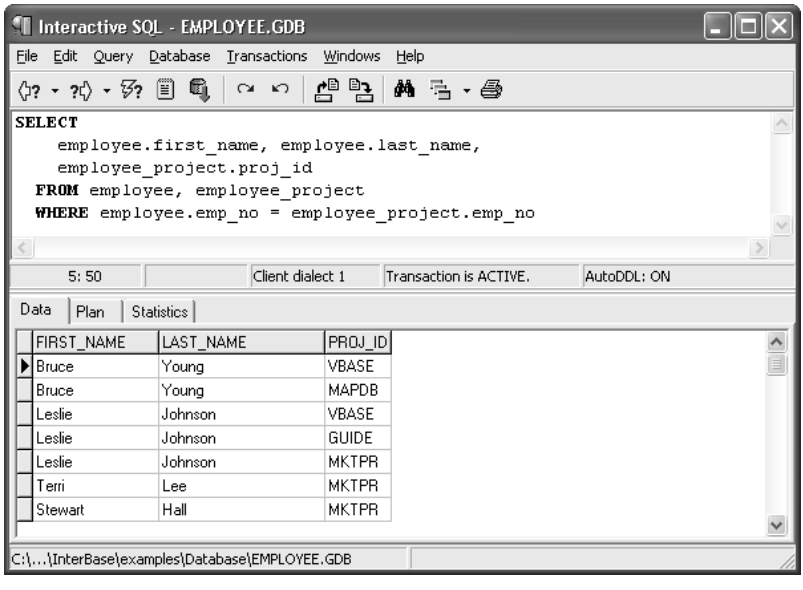


Abbildung 3.11: INNER JOIN

Übrigens: Mit Hilfe der WHERE-Klausel kann man auch Datensätze filtern. (Beispielsweise: Zeige mir alle Kunden an, die aus Berlin kommen.) Bei sehr umfangreichen SQL-Anweisungen kann man dann leicht ein wenig durcheinander kommen, was denn nun Verknüpfung für einen INNER JOIN und was denn nun Filtern der Datenmenge ist.

Deshalb ist auch eine Schreibweise möglich, die auf die WHERE-Klausel verzichtet und stattdessen eine ON-Klausel verwendet. Diese Schreibweise ist an den OUTER JOIN angelehnt, den wir gleich besprechen.

```
SELECT
    employee.first_name, employee.last_name,
    employee_project.proj_id
  FROM employee
    INNER JOIN employee_project
      ON employee.emp_no = employee_project.emp_no
```

Nun noch mal zurück zu unserem INNER JOIN: Ihnen ist sicher aufgefallen, dass vor jeden Spaltenbezeichner der dazugehörende Tabellenbezeichner gesetzt worden ist, getrennt durch einen Punkt.

Damit weisen wir das DMS darauf hin, in welcher Tabelle die betreffende Spalte zu suchen ist. Bei der Aufzählung der anzuzeigenden Spalten wäre das noch nicht erforderlich gewesen, die Anweisung hätte auch wie folgt funktioniert:

```
SELECT
    first_name, last_name,
    proj_id
  FROM employee
    INNER JOIN employee_project
      ON employee.emp_no = employee_project.emp_no
```

Das funktioniert aber auch nur so lange, wie die verwendeten Spaltenbezeichner nur in einer der beiden Tabellen vorkommen. In der ON-Klausel ist dies jedoch nicht der Fall, somit sind die Tabellenbezeichner hier zwingend, möchte man nicht einen FULL JOIN erhalten.

Tabellen-Alias

Nun ist es allerdings müßig, jedes Mal den kompletten Tabellennamen zu nennen, und die Übersichtlichkeit fördert das auch nicht besonders. Deshalb besteht die Möglichkeit, Tabellen-Aliase zu verwenden.

```
SELECT
    e.first_name, e.last_name,
    p.proj_id
```

```
  FROM employee e
    INNER JOIN employee_project p
      ON e.emp_no = p.emp_no
```

Bei der Nennung der verwendeten Tabellen wird dem Tabellennamen jeweils der Tabellen-Alias nachgestellt. Beide werden durch ein Leerzeichen getrennt. Der Alias ist nicht auf einen Buchstaben beschränkt, in der Regel ist es aber zu empfehlen, möglichst kurze Aliase zu verwenden. Selbstverständlich ist es auch möglich, nur einen Teil der beteiligten Tabellennamen durch Aliase zu ersetzen. Im Rest der Anweisung ersetzt der Alias dann den Tabellennamen.

Die Ausführungszeit ändert sich nicht signifikant, wenn Sie einen Alias verwenden.

JOINS über mehrere Tabellen

Mit Hilfe eines JOINS können nicht nur zwei, sondern auch sehr viel mehr Tabellen zusammengefügt werden. In der Praxis sind JOINS über 20 oder mehr Tabellen gar keine Seltenheit.

```
SELECT
    e.first_name, e.last_name,
    q.proj_name
  FROM employee e
    INNER JOIN employee_project p
      ON e.emp_no = p.emp_no
    INNER JOIN project q
      ON p.proj_id = q.proj_id
```

Hier ersetzen wir die Projekt-ID durch den Projekt-Namen, den wir in der Tabelle *project* finden.

3.4.3 OUTER JOIN

Vielleicht ist Ihnen schon aufgefallen, dass wir bei unserer Liste ein paar Mitarbeiter »verloren« haben, nämlich alle, die derzeit keinem Projekt zugeordnet sind.

Kennzeichen des INNER JOIN/EQUI JOIN ist es, dass nur diejenigen Datensätze aufgenommen werden, bei denen zu allen beteiligten Tabellen Datensätze vorhanden sind.

Wollen wir nun alle Mitarbeiter in der Ergebnismenge haben, dann müssen wir einen LEFT OUTER JOIN verwenden:

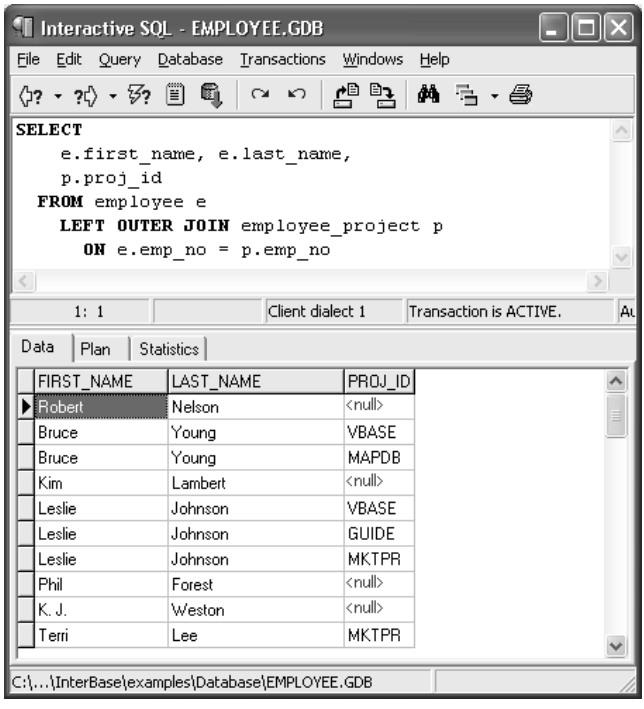


Abbildung 3.12: LEFT OUTER JOIN

Bei einem LEFT OUTER JOIN werden alle Datensätze der Tabelle »auf der linken Seite« verwendet, hier im Beispiel also *employee*. Diese werden dann – soweit möglich – mit den Datensätzen der rechten Seite, hier im Beispiel also *employee_project*, verknüpft. Beachten Sie bitte auch, dass bei einem OUTER JOIN die Verknüpfung nur über die ON-, nicht aber über die WHERE-Klausel erfolgen kann.

Die Existenz eines LEFT OUTER JOIN legt die Vermutung nahe, dass es auch einen RIGHT OUTER JOIN gibt. Da jedoch ein Fremdschlüssel dafür sorgt, dass in der Tabelle *employee_project* nur Mitarbeiternummern sind, für die es auch einen Mitarbeiter gibt, würde sich hier ein RIGHT OUTER JOIN nicht von einem INNER JOIN unterscheiden – mit dem Unterschied minimal kürzerer Ausführungszeiten beim RIGHT OUTER JOIN.

Es gibt jedoch ein Projekt, an dem gerade niemand arbeitet. Wenn wir das in der Liste haben wollen, so verknüpfen wir drei Tabellen und setzen einen RIGHT OUTER JOIN ein:

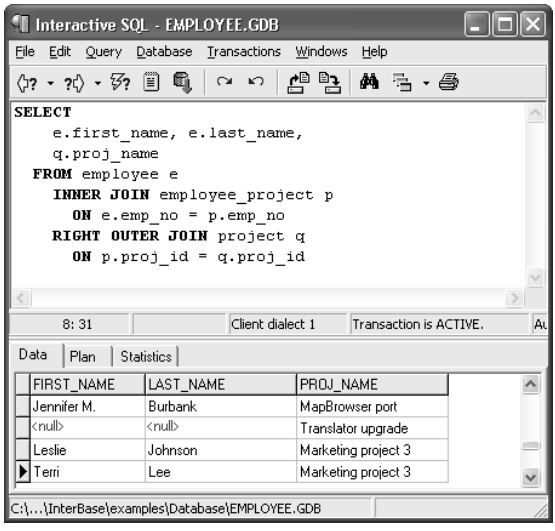


Abbildung 3.13: RIGHT OUTER JOIN

FULL OUTER JOIN

Bei einem FULL OUTER JOIN werden alle Datensätze der beteiligten Tabellen angezeigt. Wo es möglich ist, werden Verknüpfungen vorgenommen. Wenn wir also in der Datenmenge neben den entsprechenden Verknüpfungen auch noch alle Mitarbeiter haben wollen, die gerade an keinem Projekt arbeiten, und dazu noch alle Projekte, die derzeit ruhen, dann brauchen wir einen FULL OUTER JOIN:

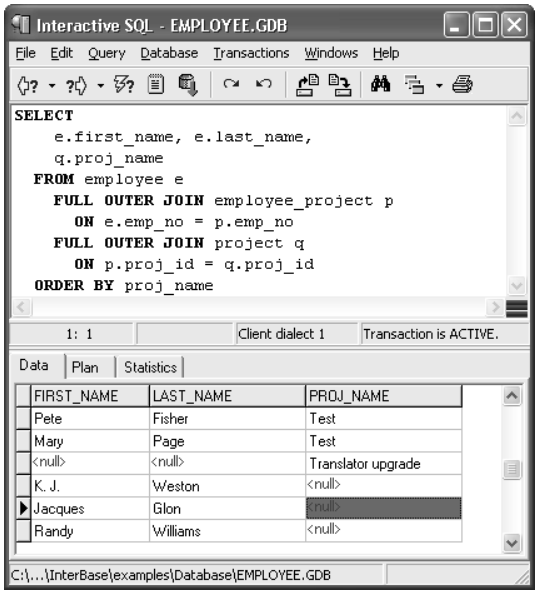


Abbildung 3.14: FULL OUTER JOIN

Die ORDER-Klausel wurde hier nur dazu eingesetzt, beide »Randmengen« gleichzeitig auf dem Screenshot zu haben – das Sortieren von Datenmengen werden wir noch ausführlich besprechen.

3.4.4 SELF JOIN

Ein SELF JOIN ist ein JOIN, bei dem die Tabelle mit sich selbst verknüpft wird. Das setzt dann auch voraus, dass für ein und dieselbe Tabelle zwei verschiedene Aliase verwendet werden.

Wozu das gut sein soll? Nehmen wir an, wir wollen zu jedem Mitarbeiter seinen Abteilungsleiter anzeigen: Dazu verknüpfen wir die Mitarbeiter-Tabelle mit der Tabelle der Abteilungen. Dort haben wir die Spalte *mngr_no*, die dann wieder die Mitarbeitertabelle referenziert:

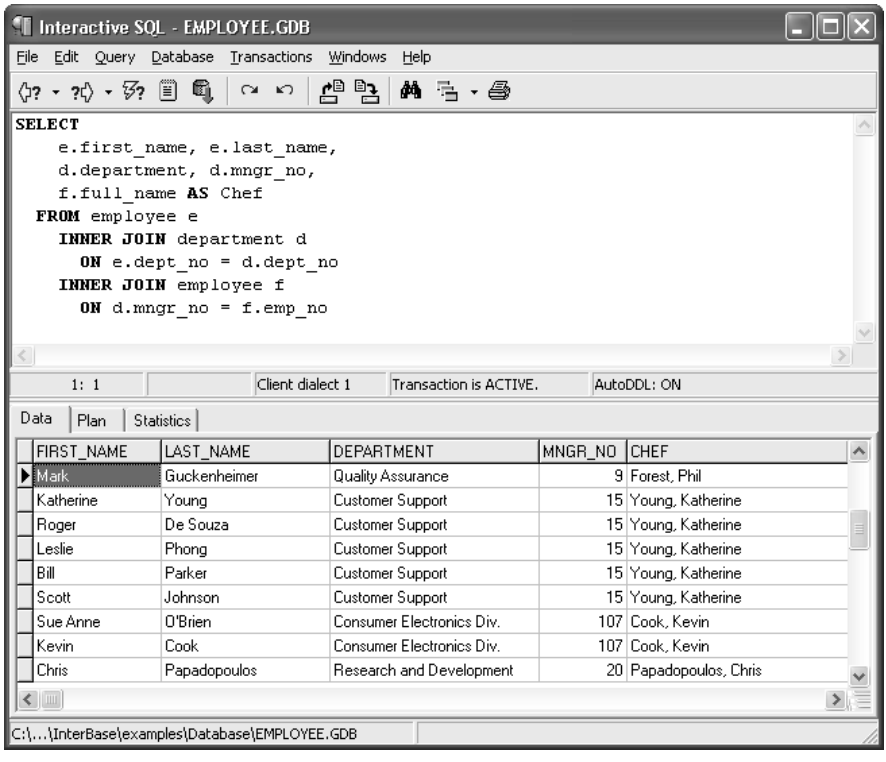


Abbildung 3.15: Self-JOIN

Ausführungszeiten

Natürlich ist es nicht egal, in welcher Reihenfolge Sie die einzelnen Tabellen auflisten – zumindest dann nicht, wenn Sie den letzten Rest an Geschwindigkeit herausholen wollen. Die Feinheiten besprechen wir dann beim Query-Plan.

3.5 WHERE

Die WHERE-Klausel haben wir schon beim INNER JOIN verwendet. Viel häufiger wird sie jedoch zum Filtern des Datenbestands verwendet.

Das Filtern von Datenbeständen wird erst bei sehr großen Datenbeständen so richtig interessant. Deshalb werden wir dafür vor allem die Tabelle *t_adressen* verwenden – die Kundenkartei unserer fiktiven Firma. Mit der folgenden Anweisung ermitteln wir den Datensatz mit der Nummer 100:

```
SELECT *
    FROM t_adressen
    WHERE id  = 257
```

Mit der Anweisung *WHERE id = 257* beschränken wir die Anzeige auf diejenigen Datensätze, bei denen das Feld *id* den Wert 257 hat. Da in *id* eine durchlaufende Nummer gespeichert ist, wird maximal ein Datensatz angezeigt.

Mit der eben genannten SELECT-Anweisung wird übrigens nicht unbedingt der 257. Datensatz angezeigt. Dies wäre nur dann der Fall, wenn in der Tabelle Datensätze mit den Nummern 1 bis 257 gespeichert wären. Genauso gut kann es sein, dass in einer Tabelle von mehr als tausend Datensätzen keiner die Bedingung *nummer = 257* erfüllt, sei es, dass die Zählung nicht mit eins begonnen wurde, sei es, dass ein betreffender Datensatz bereits gelöscht wurde.

Suche nach Strings

Selbstverständlich kann in SQL nicht nur nach Nummern, sondern auch nach Strings gefiltert werden. Beachten Sie bitte, dass String-Konstanten in einfache Anführungszeichen zu setzen sind, damit sie das DMS von Bezeichnern unterscheiden kann. (Im SQL-Dialekt 1 wären auch doppelte Anführungszeichen erlaubt.)

```
SELECT *
  FROM t_adressen
  WHERE tel = '06074 / 77 365'
```

Wenn Sie die Testdatenbank von der CD kopiert haben, sollte nun exakt derselbe Datensatz angezeigt werden. Vergleichen wir kurz die Ausführungszeiten:

Wir stellen fest, dass die Suche nach der Spalte *ID* deutlich schneller geht als die Suche nach der Spalte *Tel*. Zunächst haben wir bei der Suche nach der *ID* minimal höhere *Prepare*-Zeiten – das haben Primärschlüssel-Spalten nun mal so an sich. Dafür ist die *Execute*-Zeit um Größenordnungen geringer, da für jede Schlüssel-Spalte automatisch ein Index angelegt wird. Außerdem geht das Suchen in einer Integer-Spalte schneller als in einer String-Spalte.

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	677	566	9	1 252
Ausführung 2	482	500	9	991
Ausführung 3	481	500	9	990
Ausführung 4	484	503	8	995
Anweisung 2 Ausführung 1	472	33 392	17	33 881
Ausführung 2	462	32 526	17	33 005
Ausführung 3	460	32 813	17	33 290
Ausführung 4	453	32 465	16	32 934

Abbildung 3.16: Auswirkung von Inidzes

Bei der Suche nach dem Primärschlüssel kann der Server gleich nach dem Abrufen des Datensatzes EOF melden, bei der Suche nach der Spalte *Tel* wird dafür ein weiterer *Fetch*-Versuch benötigt, deshalb ist diese Zeit doppelt so hoch – der angezeigte Datensatz ist exakt derselbe.

Suchgeschwindigkeit und Datentypen

Erinnern wir uns an die Tabelle *t_test9* mit den Zahlen von 1 bis 9999. Wir wollen nun hier die Ausführungsgeschwindigkeiten vergleichen für die folgenden Abfragen:

```
SELECT *
  FROM t_test9
  WHERE ID = 12345

...  WHERE f_int = 12345

...  WHERE f_flo = 12345

...  WHERE f_num = 12345

...  WHERE f_cha = '12345'
```

Wie *Abbildung 3.1* zeigt, haben wir wieder die längsten Prepare-Zeiten bei der Spalte *ID*, obwohl es sich hier um eine INTEGER-Spalte handelt, die ohne Index besonders schnell vorbereitet wird. Bei den *Execute*-Zeiten haben wir geringfügige Unterschiede zwischen den einzelnen Typen und einen sehr großen Vorteil bei indizierten Spalten. Die *Fetch*-Zeit ist dann bei allen Anweisungen gleich – ein Blick in das Monitorfenster zeigt dann auch, dass hier selbst bei der VARCHAR-Spalte kein zweiter *Fetch*-Versuch erfolgt.

Nun wollen wir für die Spalte *f_int* einen Index erstellen:

```
CREATE INDEX ix_test9_int
  ON t_test9 (f_int)
```

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	648	395	1	1 044
Ausführung 2	510	368	1	879
Ausführung 3	518	370	1	889
Ausführung 4	521	372	1	894
Anweisung 2 Ausführung 1	427	35 456	1	35 884
Ausführung 2	415	35 492	1	35 908
Ausführung 3	415	35 074	1	35 490
Ausführung 4	411	35 495	1	35 907
Anweisung 3 Ausführung 1	500	38 908	1	39 409
Ausführung 2	494	39 100	1	39 595
Ausführung 3	497	40 237	1	40 735
Ausführung 4	495	39 927	1	40 423
Anweisung 4 Ausführung 1	500	41 107	1	41 608
Ausführung 2	494	39 915	1	40 410
Ausführung 3	526	39 770	1	40 297
Ausführung 4	527	41 116	1	41 644
Anweisung 5 Ausführung 1	540	50 572	1	51 113
Ausführung 2	494	51 061	1	51 556
Ausführung 3	496	51 080	1	51 577
Ausführung 4	523	49 910	1	50 434

Abbildung 3.17: WHERE-Klausel auf verschiedene Datentypen

Anweisung 2 Ausführung 1	452	334	1	787
Ausführung 2	439	331	1	771
Ausführung 3	438	331	1	770
Ausführung 4	435	329	1	765

Abbildung 3.18: Beschleunigung durch einen Index

Wie wir sehen, erhöht der Index leicht die *Prepare*-Zeit und vermindert deutlich die *Execute*-Zeit. Warum jedoch ist hier die Ausführungszeit kürzer als beim Primärschlüssel? Werfen wir einen Blick in die Datenbank-Statistik:

```
T_TEST9 (150)

  Index IX_TEST9_INT (1)
  Depth: 2, leaf buckets: 18, nodes: 9999
  Average data length: 1.00, total dup: 0, max dup: 0
  Fill distribution:
       0 - 19% = 0
      20 - 39% = 1
      40 - 59% = 0
      60 - 79% = 0
      80 - 99% = 17
```

```
  Index RDB$PRIMARY29 (0)
  Depth: 2, leaf buckets: 35, nodes: 9999
  Average data length: 1.00, total dup: 0, max dup: 0
  Fill distribution:
       0 - 19% = 0
      20 - 39% = 1
      40 - 59% = 34
      60 - 79% = 0
      80 - 99% = 0
```

Wenn wir die Werte für *leaf buckets* und die Füllung der Seiten vergleichen, dann ist der Index *ix_test9_int* ein wenig symmetrischer und somit effizienter – was auch kein Kunststück ist: Beim Erstellen von *ix_test9_int* war die Datenbank schon völlig gefüllt, da fällt es dem Server nicht schwer, einen effektiven Index zu erstellen. Den Primärschlüssel gibt es schon von Anfang an und mit jedem neu eingefügten Datensatz kommt er ein Stück mehr aus dem Gleichgewicht.

Index, Index über alles?

Man könnte nun zu dem Schluss kommen, dass man möglichst für alle Spalten Indizes erstellt. Jeder Index muss aber auch gepflegt werden, wenn Datensätze eingefügt, geändert oder gelöscht werden, deshalb sollte man es auch nicht übertreiben.

Zudem ist nicht jeder Index so effektiv wie hier. Nehmen wir einmal an, Sie indizierten die Spalte *Anrede* in der Tabelle *t_adressen*. Dort gibt es nur zwei Einträge und dementsprechend hoch wäre die Zahl der Dubletten:

```
T_ADRESSEN (128)
   Index IX_ADRESSEN_ANREDE (4)
  Depth: 2, leaf buckets: 13, nodes: 8289
  Average data length: 0.00, total dup: 8287, max dup: 4249
  Fill distribution:
       0 - 19% = 0
      20 - 39% = 1
      40 - 59% = 0
      60 - 79% = 0
      80 - 99% = 12
```

Doch selbst hier würde ein (gut ausgewogener!) Index Vorteile bringen, die längere *Execute*-Zeit würde durch die geringere *Fetch*-Zeit kompensiert:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	717	2 952	123 067	126 736
Ausführung 2	596	2 911	122 880	126 387
Ausführung 3	570	3 019	122 094	125 683
Ausführung 4	572	2 910	122 902	126 384
Anweisung 2 Ausführung 1	645	2 334	147 642	150 621
Ausführung 2	568	2 822	147 879	151 269
Ausführung 3	562	2 317	147 072	149 951
Ausführung 4	569	2 314	147 667	150 550

Abbildung 3.19: Index mit hoher Dupletten-Zahl

3.5.1 Logische Verknüpfungen

So richtig interessant wird die WHERE-Klausel erst dann, wenn man mehrere Filter-Bedingungen miteinander logisch verknüpft. Wir haben das beim INNER JOIN schon einmal kurz kennen gelernt.

Die AND-Verknüpfung

Bei einer AND-Verknüpfung müssen alle Bedingungen erfüllt sein, damit ein Datensatz in die Ergebnismenge aufgenommen wird.

```
SELECT *
  FROM t_adressen
  WHERE (vorname = 'Michael')
    AND (nachname = 'Weber')
```

In dieser Anweisung wird nach dem Kunden namens *Michael Weber* gesucht. Es wäre hier nicht zwingend, die einzelnen Filterbedingungen in Klammern zu setzen.

Die OR-Verknüpfung

Bei einer OR-Verknüpfung muss eine der Bedingungen erfüllt sein, damit ein Datensatz in die Ergebnismenge aufgenommen wird.

```
SELECT *
  FROM t_adressen
  WHERE (vorname = 'Leonore')
    OR (vorname = 'Ernst')
```

Mit dieser Anweisung werden alle Datensätze ermittelt, deren Feld vorname die Werte Leonore oder Ernst enthalten.

Die NOT-Verknüpfung

Mit Hilfe des Schlüsselwortes NOT wird eine Suchbedingung negiert.

```
SELECT *
  FROM t_adressen
  WHERE NOT (anrede = 'Herr')
```

Mit dieser Anweisung werden alle Frauen in der Kundentabelle erfasst, was aber einen Tick länger dauert als *WHERE (anrede = 'Frau')*, zudem würde ein vorhandener Index nicht genutzt werden.

Kombination logischer Verknüpfungen

Sie können AND-, OR- und NOT-Operatoren beliebig miteinander kombinieren. Achten Sie auf korrekte Klammersetzung, damit das DMS die Anweisung auch so interpretiert, wie Sie es beabsichtigen.

```
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE (((vorname = 'Daniel')
        AND (nachname = 'Unglert'))
    OR ((vorname = 'Stefanie')
        AND (nachname = 'Busch')))
    AND NOT (ort = 'Berlin')
```

Diese Anweisung ermittelt alle Kunden namens *Daniel Unglert* und *Stefanie Busch*, deren Wohnsitz nicht in Berlin liegt.

Übrigens: Die folgende Anweisung geht nicht, weil das DMS den Spalten-Alias *name* in der WHERE-Klausel noch nicht kennt:

```
SELECT vorname || ' ' || nachname AS name, ort
  FROM t_adressen
  WHERE ((name = 'Daniel Ost')
      OR (name = 'Stefanie Busch'))
    AND NOT (ort = 'Berlin')

/* geht nicht ! */
```

Hier sehen Sie dann auch, wie man in SQL-Statements Kommentare einfügt. Kommentare sind Texte, die vom DMS ignoriert werden. Sie sind dazu gedacht, SQL-Anweisungen zu erläutern. Man kann sie auch dazu verwenden, einen Teil einer SQL-Anweisung außer Kraft zu setzen, ohne den entsprechenden Text gleich löschen zu müssen:

```
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE (((vorname = 'Daniel')
        AND (nachname = 'Unglert'))
```

```
    OR ((vorname = 'Stefanie')
        AND (nachname = 'Busch')))
    /* AND NOT (ort = 'Berlin') */
```

Übrigens: Wenn Sie einmal in die Verlegenheit gelangen sollten, nach kompletten Namen zu suchen, dann können Sie folgendermaßen vorgehen:

```
SELECT vorname || ' ' || nachname AS name, ort
  FROM t_adressen
  WHERE ((vorname || ' ' || nachname = 'Daniel Unglert')
      OR (vorname || ' ' || nachname = 'Stefanie Busch'))
    AND NOT (ort = 'Berlin')
```

Wir wollen nun wieder messen, was denn schneller geht: Vorname und Nachname mittels OR-Verknüpfung zusammenfügen oder eine gemeinsame Zeichenkette? Und damit wir auch gleich die Ursache sehen, wollen wir als dritte Anweisung bei der OR-Verknüpfung auf vorhandene Indizes verzichten, weswegen wir eine entsprechende PLAN-Anweisung formulieren:

```
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE (((vorname = 'Daniel')
        AND (nachname = 'Unglert'))
    OR ((vorname = 'Stefanie')
        AND (nachname = 'Busch')))
    AND NOT (ort = 'Berlin')
  PLAN (t_adressen NATURAL)
```

Das Ergebnis ist recht eindeutig:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	759	2 756	12	3 527
Ausführung 2	608	2 687	12	3 307
Ausführung 3	627	2 697	11	3 335
Ausführung 4	611	2 791	12	3 414
Anweisung 2 Ausführung 1	552	102 972	17	103 541
Ausführung 2	537	102 889	16	103 442
Ausführung 3	703	103 004	17	103 724
Ausführung 4	565	102 889	17	103 471
Anweisung 3 Ausführung 1	590	86 689	17	87 296
Ausführung 2	584	87 539	18	88 141
Ausführung 3	591	86 521	17	87 129
Ausführung 4	587	87 349	17	87 953

Abbildung 3.20: Zusammengefügte Suchspalten

Mit der OR-Verknüpfung geht es um Größenordnungen schneller als mit zusammengesetzten Spalten, weil dann der Index auf die Spalte *Nachname* nicht mehr genutzt werden kann. Schalten wir diesen Index aus (PLAN NATURAL), dann geht es mit der OR-Verknüpfung immer noch schneller, aber der Unterschied schrumpft auf etwa 18 %.

3.5.2 Die Vergleichs-Operatoren

Wenn wir bislang Datenmengen mit WHERE gefiltert haben, dann haben wir immer den Gleichheitsoperator eingesetzt. SQL kennt jedoch eine ganze Reihe von anderen Operatoren, die wir nun besprechen wollen.

Die <- und >-Operatoren

Mit den <- und >-Operatoren beschränkt man die Anzeige auf diejenigen Datensätze, die größer oder kleiner als ein bestimmtes Kriterium sind.

```
SELECT *
    FROM t_adresse
    WHERE (id < 100)
```

Diese Anweisung ermittelt alle Kunden, deren Nummer unter 100 liegt. Der Datensatz mit der Nummer 100 wird dabei nicht angezeigt. Wenn wir auch den Datensatz mit der Nummer 100 angezeigt haben möchten, dann müssen wir eine der drei folgenden Anweisungen nehmen:

```
SELECT *
  FROM t_adressen
  WHERE (id <= 100)

SELECT *
  FROM t_adressen
  WHERE (id < 101)

SELECT *
    FROM t_adressen
    WHERE (id < 100)
        OR (id = 100)
```

Wie an der letzten der drei Anweisungen zu sehen ist, gibt es in SQL immer die Möglichkeit, es ein wenig komplizierter als nötig zu machen. Die zweite Anweisung hätte man übrigens auch folgendermaßen umschreiben können:

```
SELECT *
    FROM t_adressen
    WHERE (101 > id)
```

Und die erste so:

```
SELECT *
    FROM t_adressen
    WHERE (100 >= id)
```

Beachten Sie jedoch, dass bei *kleiner/gleich*- und *größer/gleich*-Operatoren das Gleichheitszeichen immer an zweiter Stelle steht. Die Unterschiede in den Ausführungszeiten verlieren sich in der Messgenauigkeit, lediglich die Variante mit der OR-Verknüpfung braucht einen Hauch länger.

Übrigens: Die <- und >-Operatoren sind nicht auf Zahlen beschränkt, sondern funktionieren eigentlich mit allem, was sich (vernünftig) sortieren lässt, beispielsweise auch mit Strings:

```
SELECT *
  FROM t_adressen
  WHERE (nachname > 'M')
    AND (nachname < 'N')
    AND (ort = 'Berlin')
```

Diese Anweisung liefert alle Berliner Kunden, deren Nachnamen mit *M* beginnen. (Hinweis: *Ma* ist für den Computer größer als *M*.) Für das Filtern nach String-Feldern gibt es aber geeignetere Funktionen, die wir auch gleich kennen lernen werden.

Der BETWEEN-Operator

Nehmen wir an, wir möchten alle Kunden mit einer ID zwischen 500 und 600. Das könnte man wie folgt formulieren:

```
SELECT *
  FROM t_adressen
  WHERE (id >= 500)
    AND (id <= 600)
```

Etwas übersichtlicher geht es mit dem Operator BETWEEN:

```
SELECT *
  FROM t_adressen
  WHERE (id BETWEEN 500 AND 600)
```

Beim Operator BETWEEN zählen die beiden Grenzen immer noch mit zum Bereich.

Der LIKE-Operator

Zu den flexibelsten Operatoren zählt der LIKE-Operator. Er erlaubt die Verwendung von so genannten Joker-Zeichen, also Zeichen, die für beliebige andere Zeichen stehen können. Das verwenden wir hier wieder für die Suche nach Kunden, deren Nachnamen mit *M* beginnen.

```
SELECT *
  FROM t_adressen
  WHERE (nachname LIKE 'M%')
```

Das Joker-Zeichen % steht für keins, eins oder mehrere andere Zeichen. Es werden also alle Berliner Datensätze ermittelt, deren Nachnamen mit einem *M* beginnen, dem beliebig viele Zeichen beliebigen Inhalts folgen.

Neben dem Joker-Zeichen % gibt es auch noch das Joker-Zeichen _, das für exakt ein beliebiges anderes Zeichen steht. Mit der folgenden Anweisung suchen wir nach Kunden, deren Nachnamen *Müller*, *Miller*, *Möller* oder so ähnlich lauten.

```
SELECT *
    FROM t_adressen
    WHERE (nachname LIKE 'M_ller')
```

Die Funktion UPPER

Die Funktion UPPER wandelt alle Zeichen des ihr übergebenen Parameters in Großbuchstaben um. Das hat nur indirekt mit der WHERE-Klausel zu tun: Wie Sie inzwischen sicher festgestellt haben, ist bei den Vergleichen die Groß- und Kleinschreibung zu beachten.

Bisweilen möchte man die Groß- und Kleinschreibung allerdings ignorieren, beispielsweise auch diejenigen Datensätze erhalten, bei denen der erste Buchstabe versehentlich klein- oder der zweite versehentlich großgeschrieben wurde. Dies erreicht man dadurch, dass man die Suchspalte und das Suchkriterium einheitlich in Großbuchstaben umwandelt.

```
SELECT *
    FROM t_kunde
    WHERE (UPPER(nachname) = UPPER("fUCHS"))
        AND (ort = "Berlin")
```

Wir ahnen ja schon, dass dies die Ausführungsgeschwindigkeit nicht verbessert, aber wir wollen es natürlich genau wissen und vergleichen es mit der folgenden Anweisung:

```
SELECT *
  FROM t_adressen
  WHERE nachname = 'Fuchs'
```

Als dritte Anweisung schalten wir dann mit der Zeile *PLAN (t_adressen NATURAL)* noch den Index aus.

Als vierte Anweisung messen wir dann mit nur einer UPPER-Anweisung:

```
SELECT *
  FROM t_adressen
  WHERE (UPPER(nachname) = 'FUCHS')
```

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	679	39 332	25 953	65 964
Ausführung 2	604	39 796	26 144	66 544
Ausführung 3	634	39 191	25 885	65 710
Ausführung 4	571	39 634	26 566	66 771
Anweisung 2 Ausführung 1	574	2 296	1 604	4 474
Ausführung 2	581	2 279	1 605	4 465
Ausführung 3	567	2 214	1 564	4 345
Ausführung 4	582	2 203	1 675	4 460
Anweisung 3 Ausführung 1	567	29 130	19 308	49 005
Ausführung 2	566	30 316	19 149	50 031
Ausführung 3	519	29 183	19 469	49 171
Ausführung 4	596	29 292	19 091	48 979
Anweisung 4 Ausführung 1	614	35 248	23 313	59 175
Ausführung 2	575	35 168	23 427	59 170
Ausführung 3	566	35 766	23 586	59 918
Ausführung 4	557	35 168	23 340	59 065

Abbildung 3.21: Auswirkungen der Funktion UPPER

Die Umwandlung mit UPPER braucht ihre Zeit, und wann immer es möglich ist, sollten Sie zumindest den Suchstring nicht vom Server umwandeln lassen. Die größte »Bremse« ist jedoch die Tatsache, dass nun der Index nicht mehr verwendet wird.

In der Praxis ist es jedoch meist wichtiger, einen Datensatz überhaupt zu finden, als das letzte Quentchen an Performance herauszukitzeln, und erfreulicherweise funktioniert die Sache inzwischen auch mit Umlauten.

Suche nach Zahlen

Der Operator LIKE lässt sich übrigens auch bei der Suche nach Zahlen verwenden. Dabei werden Felder, die nicht im String-Format vorliegen, automatisch umgewandelt.

```
SELECT *
  FROM t_adressen
  WHERE id LIKE '233%'
```

Der STARTING WITH-Operator

Bei vielen WHERE-Klauseln wird nach dem Anfang eines Strings gesucht. Wie Sie inzwischen wissen, ist das mit Hilfe des LIKE-Operators problemlos möglich.

Allerdings fordert die Flexibilität des LIKE-Operators ihren Preis in Form längerer Ausführungszeiten. Schneller arbeitet der Operator STARTING WITH.

```
SELECT *
  FROM t_adressen
  WHERE nachname STARTING WITH 'Ma'
```

Diese Anweisung ermittelt alle Kunden, deren Nachnamen mit *Ma* beginnen. Beachten Sie, dass auch STARTING WITH zwischen Groß- und Kleinschreibung unterscheidet.

Die Sache mit den Ausführungszeiten wollen wir natürlich etwas genauer wissen und vergleichen diese Anweisung mit den folgenden beiden:

```
SELECT *
  FROM t_adressen
  WHERE nachname LIKE 'Ma%'

SELECT *
  FROM t_adressen
  WHERE (nachname > 'Ma')
    AND (nachname < 'Mb')
```

Die Unterschiede: Vielleicht 5 % Vorteil für STARTING WITH, bei früheren InterBase-Versionen war das deutlicher.

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	709	2 190	1 208	4 107
Ausführung 2	478	2 095	1 200	3 773
Ausführung 3	490	2 102	1 202	3 794
Ausführung 4	472	2 091	1 232	3 795
Anweisung 2 Ausführung 1	518	2 217	1 268	4 003
Ausführung 2	506	2 213	1 266	3 985
Ausführung 3	484	2 206	1 307	3 997
Ausführung 4	488	2 204	1 265	3 957
Anweisung 3 Ausführung 1	590	2 214	1 265	4 069
Ausführung 2	498	2 203	1 368	4 069
Ausführung 3	496	2 205	1 264	3 965
Ausführung 4	488	2 206	1 273	3 967

Abbildung 3.22: Ausführungszeiten bei STARTING WITH

Wir haben hier allerdings nicht besonders praxisnah gemessen. In der Regel würde man so vorgehen, dass man die SQL-Anweisung mit einem Parameter formuliert:

```
SELECT *
  FROM t_adressen
  WHERE nachname STARTING WITH :starting
```

Diesen Parameter würde man dann nach dem Vorbereiten setzen, und zwar mit einem Wert, den der Benutzer eingibt. Bei der LIKE-Anweisnung sähe das wie folgt aus:

```
SELECT *
  FROM t_adressen
  WHERE nachname LIKE :like
```

Ändern wir unser Testprogramm dementsprechend ab:

```
procedure TForm1.Messen(Anweisung, Reihe: integer);
    ...
  Zeitmessen(1, Reihe, DoPrepare, Summe);
  ParamSetzen;
  Zeitmessen(2, Reihe, DoExecute, Summe);
    ...
end; {procedure TForm1.Messen}
procedure TForm1.ParamSetzen;
begin
  with SQLDataSet1.Params do
  begin
    if Items[0].Name = 'starting'
      then Items[0].AsString := Edit1.Text;
    if Items[0].Name = 'like'
      then Items[0].AsString := Edit1.Text + '%';
  end; {with SQLDataSet1.Params do}
end; {procedure TForm1.ParamSetzen}
```

Nun vergleichen wir wieder die Ausführungszeiten:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	705	2 230	1 233	4 168
Ausführung 2	569	2 487	1 449	4 505
Ausführung 3	569	2 193	1 232	3 994
Ausführung 4	653	2 206	1 232	4 091
Anweisung 2 Ausführung 1	506	24 851	23 480	48 837
Ausführung 2	716	25 215	22 916	48 847
Ausführung 3	487	25 085	24 632	50 204
Ausführung 4	864	25 584	23 420	49 868

Abbildung 3.23: Vorteil für STARTING WITH bei Parametern

Nun hat STARTING WITH deutlich die Nase vorne. Woher kommt das? Bei der ersten Messung lag die SQL-Anweisung vollständig vor, beim Vorbereiten konnte also erkannt werden, dass der Anfang einer Spalte gesucht wird, dass also der Index verwendet werden konnte.

Wird dieselbe Anweisung nun mit einem Parameter statt einer Konstanten vorbereitet, weiß der Server nicht, ob später am Anfang, in der Mitte oder am Ende des Strings gesucht werden soll, der Parameterwert hätte ja auch *%er* lauten können. Damit konnte der Index nicht mehr verwendet werden, so dass die Ausführungszeiten deutlich nach oben gehen.

Der CONTAINING-Operator

Mit Hilfe des CONTAINING-Operators kann ein String auf das beliebige Vorkommen eines anderen Strings untersucht werden.

```
SELECT *
  FROM t_adressen
  WHERE (nachname CONTAINING 'ä')
    AND (ort = 'Berlin')
```

Diese Anweisung ermittelt alle Berliner Kunden, deren Nachnamen den Buchstaben *ä* enthalten. Die Länge des zu suchenden Strings ist nicht auf ein Zeichen beschränkt.

Der IS NULL-Operator

Der Wert NULL steht in SQL für nicht zugewiesene Werte. Nehmen wir an, wir wollen eine Liste aller Kunden, die kein Telefon haben. »Geradeaus« gedacht könnte man es mit folgender Anweisung versuchen:

```
SELECT *
  FROM t_adressen
  WHERE tel = ''
```

Damit erhalten Sie jedoch keinen einzigen Datensatz, weil NULL halt etwas anderes ist als ein leerer String. Sie müssen stattdessen formulieren:

```
SELECT *
  FROM t_adressen
  WHERE tel IS NULL
```

Für die Negation von IS NULL gibt es zwei Möglichkeiten:

```
SELECT *
  FROM t_adressen
  WHERE NOT tel IS NULL

SELECT *
  FROM t_adressen
  WHERE tel IS NOT NULL
```

Der IN-Operator

Nehmen wir an, Sie suchen alle Kunden mit den Namen *Borst*, *Thiel* und *Fichtl*. Dies könnten Sie folgendermaßen formulieren:

```
SELECT *
  FROM t_adressen
  WHERE (nachname = 'Borst')
    OR (nachname = 'Thiel')
    OR (nachname = 'Fichtl')
```

Einfacher und übersichtlicher geht es mit dem IN-Operator:

```
SELECT *
  FROM t_adressen
  WHERE (nachname IN ('Borst', 'Thiel', 'Fichtl'))
```

Nach dem IN-Operator folgt in Klammern die Menge der Elemente, die das Suchkriterium definieren. Auswirkungen auf die Ausführungszeit habe ich nicht festgestellt.

3.6 GROUP BY

Die GROUP-Klausel dient zum Zusammenfassen von Datensätzen für die Auswertung durch statistische Funktionen.

Die statistischen Funktionen

In SQL sind die in *Tabelle 3.2* aufgeführten statistischen Funktionen definiert.

Name	Funktion
COUNT	Anzahl
SUM	Summe
MIN	Minimum
MAX	Maximum
AVG	Durchschnitt

Tabelle 3.2: Die statistischen Funktionen in SQL

Mit der folgenden Anweisung

```
SELECT
    COUNT(preis) AS Anzahl,
    SUM(preis) AS Summe,
    MIN(preis) AS minimaler_Preis,
    Max(preis) AS maximaler_Preis,
    AVG(preis) AS durchschnittlicher_Preis
  FROM t_produkt
```

Wir erfahren also, wie viele Artikel wir führen, wie viel es kosten würde, jeden Artikel exakt einmal zu kaufen, wie viel der billigste und wie viel der teuerste Artikel kosten würde und was der durchschnittliche Preis unserer Waren ist.

Natürlich würden die Ergebnisse wieder mit allen verfügbaren Nachkommastellen angezeigt. Wollte man das vermeiden, würde man sich mit einer Typenumwandlung weiterhelfen:

```
SELECT
    COUNT(preis) AS Anzahl,
    CAST(SUM(preis) AS NUMERIC(10,2)) AS Summe,
    CAST(MIN(preis) AS NUMERIC(10,2)) AS minimaler_Preis,
    CAST(MAX(preis) AS NUMERIC(10,2)) AS maximaler_Preis,
    CAST(AVG(preis) AS NUMERIC(10,2))
AS durchschnittlicher_Preis
  FROM t_produkt
```

Die Aggregatfunktionen kann man auch mit der WHERE-Klausel kombinieren, um beispielsweise festzustellen, wie viele Poster Richard Wagner abbilden:

```
SELECT COUNT(*)
  FROM t_produkt
  WHERE titel = 'Wagner'
```

Mit *COUNT(*)* zählen Sie alle Datensätze, die den Kriterien der WHERE-Klausel entsprechen, während bei beispielsweise *COUNT(Preis)* nur diejenigen Datensätze berücksichtigt werden, die in der Spalte *Preis* keinen NULL-Wert stehen haben. Mit dem Stern sind Sie zudem auch noch einen Tick schneller.

Wenn wir schon beim Thema Geschwindigkeit sind: Die Ausführungszeiten der einzelnen Aggregatfunktionen weichen nur minimal voneinander ab (Summe und Durchschnitt dauern etwas länger).

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	602	956	8	1 566
Ausführung 2	345	871	6	1 222
Ausführung 3	346	871	6	1 223
Ausführung 4	341	870	6	1 217
Anweisung 2 Ausführung 1	376	920	6	1 302
Ausführung 2	345	909	6	1 260
Ausführung 3	345	936	6	1 287
Ausführung 4	462	933	6	1 401
Anweisung 3 Ausführung 1	394	395	6	795
Ausführung 2	395	259	6	660
Ausführung 3	380	259	6	645
Ausführung 4	388	259	6	653
Anweisung 4 Ausführung 1	414	268	6	688
Ausführung 2	381	253	6	640
Ausführung 3	372	253	6	631
Ausführung 4	376	257	6	639
Anweisung 5 Ausführung 1	358	923	6	1 287
Ausführung 2	348	924	6	1 278
Ausführung 3	347	925	6	1 278
Ausführung 4	349	924	6	1 279

Abbildung 3.24: Ausführungszeiten von Aggregatfunktionen

Wenn in *Abbildung 3.24* ein deutlicher Vorteil für MIN und MAX zu erkennen ist, dann liegt das daran, dass dafür ein Index eingerichtet wurde – für MIN benötigt man einen aufsteigenden Index, für MAX einen absteigenden. Wären die Daten nicht bereits im Speicher gewesen, wäre der Unterschied deutlicher ausgefallen. Die anderen Aggregatfunktionen profitieren nicht von einem Index.

3.6.1 Daten gruppieren

Nehmen wir einmal an, wir wollten wissen, wie viele Poster wir pro Kategorie verkaufen. Dazu gruppieren wir die Daten nach der Spalte *kat*.

```
SELECT kat, COUNT(*)
  FROM t_produkt
  GROUP BY kat
```

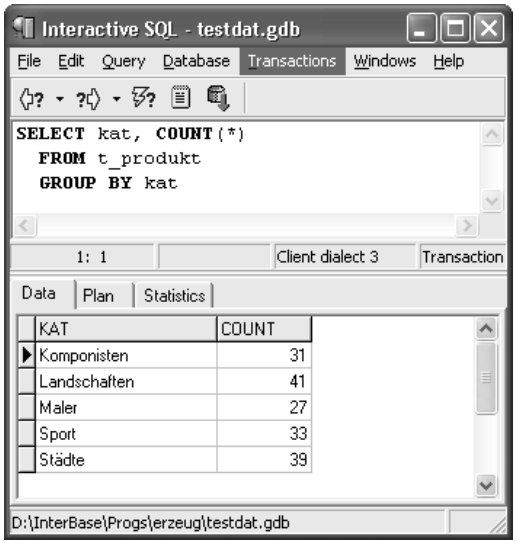


Abbildung 3.25: Gruppieren der Daten

Bei der GROUP BY-Klausel gibt es Folgendes zu beachten: Alle Spalten, die angezeigt werden, aber keine statistischen Funktionen sind, müssen in die GROUP BY-Klausel aufgenommen werden! Dies ist bei genauer Betrachtung auch leicht einzusehen: Was würde es beispielsweise für einen Sinn machen, auch noch die Artikelbezeichnungen anzuzeigen, wenn nicht nach diesen gruppiert werden soll? Wie soll das angezeigt werden?

Gruppieren nach mehreren Kriterien

Man kann natürlich auch nach mehreren Kriterien gruppieren, beispielsweise nach Kategorie und Titel:

```
SELECT kat, titel, COUNT(*)
  FROM t_produkt
  GROUP BY kat, titel
```

Das geht dann auch über mehrere Tabellen: Aus welchem Ort stammen wie viele Bestellungen und welchen Umsatz haben diese gemacht?

```
SELECT a.ort, COUNT(b.id), SUM(p.stueckzahl * o.preis)
  FROM t_adressen a
    INNER JOIN t_bestellung b
      ON a.id = b.kunde
    INNER JOIN t_posten p
      ON b.id = p.bestellung
    INNER JOIN t_produkt o
      ON p.produkt = o.id
  GROUP BY a.ort
```

(Nein, damit bringen Sie *IBConsole* nicht zum Absturz, aber ein wenig Zeit braucht InterBase dafür dann schon ...)

3.6.2 Die HAVING-Klausel

Weil wir den Datenbestand selbst generiert haben, ist die Anzahl der Orte überschaubar – normalerweise hätte man bei solchen Statistiken stets mehrseitige Ergebnisse. Wollte man nun die Anzeige auf diejenigen Orte beschränken, aus denen mindestens 5000 Bestellungen kommen, käme man mit der WHERE-Klausel nicht mehr weiter, weil hier nicht nach Tabellenspalten gefiltert werden kann.

Hier muss dann mit der HAVING-Klausel gefiltert werden:

```
SELECT a.ort, COUNT(b.id)
  FROM t_adressen a
    INNER JOIN t_bestellung b
      ON a.id = b.kunde
  GROUP BY a.ort
  HAVING COUNT(b.id) >= 5000
```

Um die Sache zu beschleunigen, soll hier auf den Umsatz verzichtet werden ...

HAVING ohne Aggregatfunktionen

Die HAVING-Klausel ist nicht auf Aggregatfunktionen beschränkt. Mit ihr kann auch – ähnlich der WHERE-Klausel – der Datenbestand gefiltert werden. Im Gegensatz zur WHERE-Klausel erfolgt die Filterung jedoch nach der Gruppierung, somit können nur diejenigen Spalten berücksichtigt werden, die auch in der Ergebnismenge vorhanden sind.

```
SELECT kat, COUNT(*)
  FROM t_produkt
  GROUP BY kat
  HAVING kat <> 'Sport'
```

Wir wollen nun vergleichen, wie lange diese Anweisung mit einer WHERE-Klausel gedauert hätte:

```
SELECT kat, COUNT(*)
  FROM t_produkt
  WHERE kat <> 'Sport'
  GROUP BY kat
```

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	642	2 278	18	2 938
Ausführung 2	462	2 122	16	2 600
Ausführung 3	477	2 124	17	2 618
Ausführung 4	460	2 117	16	2 593
Anweisung 2 Ausführung 1	504	2 352	17	2 873
Ausführung 2	466	2 451	17	2 934
Ausführung 3	462	2 325	16	2 803
Ausführung 4	463	2 329	16	2 808

Abbildung 3.26: HAVING- und WHERE-Klausel

Wir haben hier einen minimalen Vorteil für die HAVING-Klausel. Es ist für den Server aufwändiger, bei jedem Datensatz zu entscheiden, ob er in die Ergebnismenge kommt, als einfach alles durchzurechnen und dann die betreffende Zeile aus der Ergebnismenge zu löschen.

3.7 ORDER BY

Es ist Ihnen sicher schon aufgefallen, dass die Zeilen einer Abfrage mal nach der einen, mal nach der anderen und manchmal nach gar keiner der Spalten sortiert gewesen sind. Solange keine explizite Anweisung erfolgt, ist es dem DMS auch freigestellt, in welcher Reihenfolge die Zeilen übermittelt werden.

Wir wollen nun die Tabelle *t_adressen* nach der Spalte *nachname* sortieren:

```
SELECT id, vorname, nachname, ort
  FROM t_adressen
  ORDER BY nachname
```

Alternativ hätte man formulieren können:

```
SELECT id, vorname, nachname, ort
  FROM t_adressen
  ORDER BY 3
```

Die folgende Anweisung würde auch gehen:

```
SELECT *
  FROM t_adressen
  ORDER BY nachname
```

Nicht aber:

```
SELECT *
  FROM t_adressen
  ORDER BY 4
```

Die Verwendung einer Spaltenzahl als Sortierkriterium erfordert zwingend die Aufstellung einer Spaltenliste, ermöglicht dann aber auch das Sortieren nach Aggregatfunktionsspalten.

Absteigend sortieren

Solange nichts anderes angegeben wird, werden die Reihen aufsteigend, also von *a* nach *z*, sortiert. Mit Hilfe des Schlüsselwortes DESC können wir absteigend sortieren.

```
SELECT *
  FROM t_adressen
  ORDER BY nachname DESC
```

Beachten Sie, dass *DESC* dem Spaltenbezeichner nachgestellt wird.

Nach mehreren Spalten sortieren

Bei einem umfangreicheren Datenbestand kann es gewünscht sein, dass nach mehreren Spalten sortiert wird.

```
SELECT id, vorname, nachname, ort
  FROM t_adressen
  ORDER BY nachname, vorname
```

3.7.1 Ausführungszeiten beim Sortieren

Neugierig, wie wir nun mal sind, wollen wir nun auch wissen, wie viel Zeit das Sortieren in Anspruch nimmt. Dazu vergleichen wir die folgenden Anweisungen:

```
SELECT *
  FROM t_adressen

SELECT *
  FROM t_adressen
  ORDER BY vorname
```

```
SELECT *
  FROM t_adressen
  ORDER BY vorname DESC

SELECT *
  FROM t_adressen
  ORDER BY nachname

SELECT *
  FROM t_adressen
  ORDER BY nachname DESC
```

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	624	1 892	226 859	229 375
Ausführung 2	530	1 873	227 901	230 304
Ausführung 3	665	1 930	227 927	230 522
Ausführung 4	530	2 000	227 956	230 486
Anweisung 2 Ausführung 1	535	110 909	203 733	315 177
Ausführung 2	551	110 433	204 416	315 400
Ausführung 3	547	110 181	204 189	314 917
Ausführung 4	560	110 289	203 802	314 651
Anweisung 3 Ausführung 1	600	115 005	207 243	322 848
Ausführung 2	604	113 810	207 553	321 967
Ausführung 3	605	115 143	207 159	322 907
Ausführung 4	601	114 770	207 092	322 463
Anweisung 4 Ausführung 1	604	2 232	269 300	272 136
Ausführung 2	608	2 231	269 524	272 363
Ausführung 3	607	2 239	269 525	272 371
Ausführung 4	745	2 244	268 478	271 467
Anweisung 5 Ausführung 1	558	113 314	207 748	321 620
Ausführung 2	553	113 427	207 096	321 076
Ausführung 3	554	113 238	207 614	321 406
Ausführung 4	545	114 604	207 081	322 230

Abbildung 3.27: Sortieren

Wenig verwunderlich: Ohne Sortierung geht's am schnellsten. Sortieren wir nach dem Vornamen, dann steigt die *Execute*-Zeit deutlich an, während die *Fetch*-Zeit ein wenig sinkt. Dass eine Sortierung die *Execute*-Zeit nicht senkt, haben wir ja durchaus so erwartet, aber warum sinkt die *Fetch*-Zeit? Werfen wir einen Blick in die Datenbank-Statistik:

```
T_ADRESSEN (128)
  Primary pointer page: 146, Index root page: 147
  Data pages: 247, data page slots: 247, average fill: 80%
  Fill distribution:
    0 - 19% = 0
```

```
20 - 39% = 1
40 - 59% = 0
60 - 79% = 143
80 - 99% = 103
```

So, wie wir messen, liegen alle Daten bereits im Speicher. Werden die Daten unsortiert abgeholt, dann greift der Server auf die Seiten zu, die er so von der Festplatte gelesen hat, und dort liegen die Daten auf Seiten mit einer Auslastung von etwa 80 %. Wird beim Sortieren jedoch eine Ergebnismenge aufgebaut, dann liegen die Daten so, dass sie direkt hintereinander weg gelesen werden können.

Aufsteigende Sortierung geht einen Tick schneller als absteigende Sortierung, kurioserweise auch beim Abholen der Datensätze.

Das Sortieren nach dem Nachnamen geht dann wieder deutlich schneller – Sie ahnen es bestimmt: Dafür wurde ein Index definiert. Dass dabei die *Fetch*-Zeit ansteigt, liegt daran, dass hier beim Ausführen keine vollständige Ergebnismenge aufgebaut wird, sondern nur eine Liste mit den Adressen der Datensätze. Beim Abholen müssen anhand dieser Liste erst die Datensätze geholt werden, was zwar vergleichsweise schnell geht, solange die Daten alle im Speicher liegen, aber den Vorgang halt nicht beschleunigt.

Wir sehen hier auch deutlich, dass ein aufsteigender Index nicht hilft, wenn absteigend sortiert werden soll.

3.8 UNION

Mit UNION haben Sie die Möglichkeit, zwei Abfragen zusammenzufügen. Normalerweise nimmt man das dazu, Daten aus zwei Tabellen zusammenzuführen, dafür haben wir jetzt aber kein passendes Beispiel, so dass wir die Ergebnismengen von zwei Abfragen auf die gleiche Tabelle zusammenführen:

```
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE Ort = 'München'
UNION
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE Ort = 'Berlin'
```

Sie werden sich nun sicher fragen, ob man das nicht besser wie folgt formulieren würde:

```
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE Ort = 'München'
    OR Ort = 'Berlin'
```

Und damit hätten Sie ganz Recht, die zweite Anweisung würde messbar schneller ausgeführt:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	798	34 708	32 006	67 512
Ausführung 2	660	34 904	31 907	67 471
Ausführung 3	658	34 567	32 028	67 253
Ausführung 4	657	35 376	32 512	68 545
Anweisung 2 Ausführung 1	588	3 683	47 211	51 482
Ausführung 2	593	3 721	46 960	51 274
Ausführung 3	595	3 707	46 969	51 271
Ausführung 4	586	3 680	47 396	51 662

Abbildung 3.28: UNION gegen OR-Klausel

Die längere *Execute*-Zeit in Verbindung mit der geringeren *Fetch*-Zeit ist ein deutlicher Hinweis darauf, dass bei UNION eigens eine Ergebnismenge im Speicher aufgebaut wird. Diese wird – ohne dass die verlangt wurde – nach dem Vornamen sortiert. Wollten Sie nach dem Ort sortieren, dann müsste es nicht *ORDER BY ort*, sondern *ORDER BY 3* heißen.

Beachten Sie, dass dafür die Spalten beider Abfragen exakt vom selben Typ sein müssen. Dies führt dazu, dass die folgende Abfrage nicht funktioniert:

```
SELECT vorname, nachname, ort, strasse
  FROM t_adressen
  WHERE Ort = 'München'
UNION
SELECT vorname, nachname, ort, plz
  FROM t_adressen
  WHERE Ort = 'Berlin'

/* geht nicht */
```

Auch Spaltenlisten mit unterschiedlich vielen Spalten führen zu einer Fehlermeldung. Frühere InterBase-Versionen waren hier strenger, aber inzwischen können Sie zumindest die Spalten mischen:

```
SELECT vorname, nachname
  FROM t_adressen
  WHERE Ort = 'München'
UNION
SELECT nachname, ort
  FROM t_adressen
  WHERE Ort = 'Berlin'
```

Auch Konstanten unterschiedlicher Länge sind inzwischen erlaubt:

```
SELECT vorname, nachname, ort, 'Münchner'
  FROM t_adressen
  WHERE Ort = 'München'
UNION
SELECT vorname, nachname, ort, 'Berliner'
  FROM t_adressen
  WHERE Ort = 'Berlin'
```

(Am Rande: Das bekommen Sie mit einer OR-Verknüpfung dann nicht mehr hin.)

Und schon immer ein funktionierender »Griff in die Trick-Kiste« war die Anpassung der Spaltentypen mittels CAST:

```
SELECT vorname, nachname, CAST(plz AS VARCHAR(20))
  FROM t_adressen
  WHERE Ort = 'München'
UNION
SELECT vorname, nachname, ort
  FROM t_adressen
  WHERE Ort = 'Berlin'
```

3.9 Unterabfragen

Nehmen wir einmal an, Sie benötigen eine Liste aller Produkte, die überdurchschnittlich viel kosten. Hier könnte man nun erst den durchschnittlichen Preis aller Produkte ermitteln:

```
SELECT AVG(preis)
  FROM t_produkt
```

Das Ergebnis wäre 21,0529. Nun würde man eine zweite Anweisung formulieren, die nach diesem Durchschnitt filtert:

```
SELECT *
  FROM t_produkt
  WHERE preis > 21.0529
```

Ohne Notizzettel geht es mit einer Unterabfrage:

```
SELECT *
  FROM t_produkt
  WHERE preis > (SELECT AVG(preis)
    FROM t_produkt)
```

Die Ausführungszeit ist ziemlich exakt die Summe aus den beiden Einzel-Statements, so dass es von dieser Seite keine Argumente gegen eine Unterabfrage gibt.

Unterabfragen in der Spaltenliste

Unterabfragen sind nicht auf die WHERE-Klausel beschränkt, man kann sie beispielsweise auch in die Liste der anzuzeigenden Spalten aufnehmen. Davon wird in der Fachliteratur regelmäßig abgeraten, weil dies die Performance verschlechtern würde, man solle lieber einen JOIN nehmen.

Nehmen wir einmal an, wir würden dies glauben (Sie sollten nie etwas glauben, sondern jeweils im Einzelfall nachmessen) und erstellen einen JOIN für die Liste aller Produkte und die verkaufte Stückzahl:

```
SELECT o.kat, o.photograph, o.titel,
    SUM(p.stueckzahl)
  FROM t_produkt o
    INNER JOIN t_posten p
      ON p.produkt = o.id
  GROUP BY o.kat, o.photograph, o.titel
```

Nun formulieren wir die Sache noch mit einer Unterabfrage:

```
SELECT o.kat, o.photograph, o.titel,
    (SELECT SUM(p.stueckzahl)
       FROM t_posten p
       WHERE p.produkt = o.id) AS Gesamtmenge
  FROM t_produkt o
```

Die Umbenennung der Spalte wird durch dbExpress erzwungen. Um so viel vorwegzunehmen: Wirklich performant ist keine der beiden Anweisungen, stellen Sie sich also auf eine etwas längere Messdauer ein.

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	806	20 419 504	1 828 315	22 248 625
Ausführung 2	776	19 972 959	1 828 710	21 802 445
Ausführung 3	769	19 963 914	1 827 679	21 792 362
Ausführung 4	780	19 965 611	1 841 167	21 807 558
Anweisung 2 Ausführung 1	721	6 103 708	2 075 509	8 179 938
Ausführung 2	727	6 103 056	2 075 376	8 179 159
Ausführung 3	944	6 103 251	2 076 280	8 180 475
Ausführung 4	738	6 100 264	2 074 612	8 175 614

Abbildung 3.29: Unterabfrage schneller als JOIN

Bei dieser Anweisung ist die Unterabfrage deutlich schneller als der JOIN. Natürlich gibt es auch Aufgabenstellungen, bei denen das exakt andersherum ist. Von daher sollten Sie beide Varianten durchmessen und sich ein gesundes Misstrauen gegenüber Fachliteratur bewahren.

3.9.1 Funktionen für Unterabfragen

Für die Arbeit mit Unterabfragen gibt es die Funktionen ALL, ANY, SOME, EXISTS und SINGULAR, die ich alle für nicht unbedingt erforderlich halte, weil der gewünschte Zweck meist auch anders erreicht werden kann. In einigen Situationen werden sie jedoch schneller ausgeführt als anders formulierte Anweisungen.

ALL

Mit der Funktion ALL wird bestimmt, dass der gemachte Vergleich für alle Datensätze zutreffen muss, welche von der Unterabfrage ermittelt werden.

```
SELECT *
  FROM t_produkt
  WHERE preis > ALL (SELECT preis
    FROM t_produkt
    WHERE Titel = 'Wagner')
```

Diese Abfrage ermittelt alle diejenigen Datensätze der Tabelle *t_produkt*, deren Preis höher liegt als jeder Preis, der von der Unterabfrage ermittelt wird, welche die Preise aller Wagner-Poster ermittelt. Anders formuliert: Welche Poster sind teurer als mein teuerstes Wagner-Poster.

Man hätte das auch mit der Funktion MAX formulieren können:

```
SELECT *
  FROM t_produkt
  WHERE preis > (SELECT MAX(preis)
    FROM t_produkt
    WHERE Titel = 'Wagner')
```

Wenn man die Ausführungszeiten vergleicht, dann kommt man zu dem Schluss, dass man das hätte sogar mit MAX formulieren *sollen*:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	735	43 063	51	43 849
Ausführung 2	509	42 754	52	43 315
Ausführung 3	505	43 113	53	43 671
Ausführung 4	508	43 021	51	43 580
Anweisung 2 Ausführung 1	597	2 216	51	2 864
Ausführung 2	580	2 214	51	2 845
Ausführung 3	569	2 211	51	2 831
Ausführung 4	581	2 217	51	2 849

Abbildung 3.30: ALL gegen MAX

ANY und SOME

Die Funktionen ANY und SOME bewirken exakt dasselbe und können synonym verwendet werden. Die Funktionen ANY und SOME bestimmen, dass der gemachte Vergleich für mindestens einen Datensatz zutreffen muss, der von der Unterabfrage ermittelt wird.

```
SELECT *
  FROM t_produkt
  WHERE preis > ANY (SELECT preis
    FROM t_produkt
    WHERE Titel = 'Wagner')
```

Diese Abfrage ermittelt alle diejenigen Datensätze der Tabelle *t_produkt*, deren Preis höher liegt als ein beliebiger Preis, der von der Unterabfrage ermittelt wird. Anders formuliert: Die Abfrage ermittelt alle Artikel, die teurer sind als das billigste Wagner-Poster.

Auch das hätte man wieder mit einer Aggregatfunktion lösen können:

```
SELECT *
  FROM t_produkt
  WHERE preis > (SELECT MIN(preis)
    FROM t_produkt
    WHERE Titel = 'Wagner')
```

Aus Sicht der Ausführungszeiten ist es egal, ob Sie die Unterabfrage mit ANY oder SOME formulieren. Mit der Aggregatfunktion geht es hier wieder deutlich schneller:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	727	21 928	8 733	31 388
Ausführung 2	528	22 140	8 800	31 468
Ausführung 3	505	21 960	8 722	31 187
Ausführung 4	515	22 054	8 804	31 373
Anweisung 2 Ausführung 1	529	21 796	8 723	31 048
Ausführung 2	504	22 181	8 722	31 407
Ausführung 3	505	21 998	8 722	31 225
Ausführung 4	509	21 994	8 793	31 296
Anweisung 3 Ausführung 1	568	3 798	1 991	6 357
Ausführung 2	560	3 364	1 986	5 910
Ausführung 3	575	3 372	2 088	6 035
Ausführung 4	553	3 383	1 988	5 924

Abbildung 3.31: ANY und SOME gegen MIN

Die deutlich längeren *Fetch*-Zeiten lassen sich nicht mit der größeren Ergebnismenge allein erklären. Doch trotz der größeren Ergebnismenge ist die *Execute*-Zeit der ANY- und SOME-Unterabfrage nur etwa halb so groß wie die der ALL-Unterabfrage.

EXISTS

Die Funktion EXISTS wird nicht in Zusammenarbeit mit einem Operator eingesetzt, sondern bildet selbst eine Suchbedingung. Diese Suchbedingung ist immer dann erfüllt, wenn es mindestens einen Datensatz in der Unterabfrage gibt.

```
SELECT *
  FROM project p
  WHERE EXISTS (SELECT *
    FROM employee_project e
    WHERE e.proj_id = p.proj_id)
```

Hier verwenden wir wieder die Datenbank *employee.gdb*: Diese Abfrage ermittelt alle Projekte, denen derzeit Mitarbeiter zugeordnet sind.

Auch dies könnte man anders lösen, nämlich mit DISTINCT und einem JOIN:

```
SELECT DISTINCT p.proj_id,
    p.proj_name, p.proj_desc,
    p.team_leader, p.product
  FROM project p
  INNER JOIN employee_project e
    ON p.proj_id = e.proj_id
```

Die Version mit EXISTS ist hier jedoch schneller:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	722	620	25	1 367
Ausführung 2	513	546	25	1 084
Ausführung 3	518	543	25	1 086
Ausführung 4	502	541	25	1 068
Anweisung 2 Ausführung 1	546	1 029	25	1 600
Ausführung 2	521	1 027	25	1 573
Ausführung 3	519	1 004	25	1 548
Ausführung 4	539	996	25	1 560

Abbildung 3.32: EXISTS arbeitet schneller als ein JOIN

SINGULAR

Auch die Funktion SINGULAR wird ohne zusätzlichen Operator verwendet. Sie ist dann erfüllt, wenn von der Unterabfrage exakt ein Datensatz ermittelt wird.

```
SELECT *
  FROM t_produkt p
  WHERE NOT SINGULAR (SELECT *
    FROM t_produkt o
    WHERE o.photograph = p.photograph)
```

Diese Anweisung ermittelt die Poster aller Fotografen, die Bilder für mehr als ein Poster gemacht haben. Auch hier wieder eine Alternative, hier mit der Aggregatfunktion COUNT.

```
SELECT *
  FROM t_produkt p
  WHERE 1 < (SELECT COUNT(*)
    FROM t_produkt o
    WHERE o.photograph = p.photograph)
```

Die Unterschiede in der Ausführungszeit sind hier minimal.

3.10 PLAN

Im Inneren eines SQL-Servers gibt es einen Optimizer, der entscheidet, wie der Server beim Ausführen einer Abfrage vorgeht, beispielsweise, ob er einen Index verwendet. Für gewöhnlich tut dieser Optimizer das, was er soll, nämlich den schnellsten Weg finden. Es gibt jedoch die Möglichkeit, hier manuell einzugreifen.

Hier gleich eine Warnung vorweg: Für die Erforschung der Arbeitsweise von InterBase ist die PLAN-Klausel sehr hilfreich, für die praktische Arbeit ist sie weitgehend nutzlos, weil der Optimizer fast immer besser beurteilen kann, wie der schnellste Weg aussieht. Für die »wirkliche Arbeit« sollten Sie die PLAN-Klausel nur dann einsetzen, wenn Sie wirklich wissen, was Sie tun.

In diesem Zusammenhang ist auch wichtig zu wissen, dass es einen großen Unterschied macht, ob die Daten im Speicher oder auf der Festplatte liegen. Bei unserem Testprogramm haben wir die Daten grundsätzlich im Speicher liegen, weil anders keine reproduzierbaren Aussagen möglich wären – je nachdem, was für Anweisungen davor ausgeführt wurden, käme man zu ganz unterschiedlichen Messwerten.

Nun soll zwar in der Praxis auch angestrebt werden, dem Server so viel Speicher zur Verfügung zu stellen, dass er die Datenbank – zumindest alle häufig verwendeten Tabellen – komplett im Speicher halten kann, aber das ist nicht immer möglich. Von daher können bei einer wirklich praxisnahen Messung dann wieder ganz andere Ergebnisse herauskommen.

3.10.1 Sortieren einer Datenmenge

Beginnen wir mit dem Sortieren unserer Kundendaten nach dem Nachnamen:

```
SELECT *
  FROM t_adressen
  ORDER BY nachname
```

Da es für die Spalte *nachname* auch einen Index gibt, wird dieser für die Ausführung der Anweisung verwendet. Wollten wir, dass InterBase darauf verzichtet, dann könnten wir mittels einer PLAN-Klausel dies verlangen:

```
SELECT *
  FROM t_adressen
  PLAN (t_adressen NATURAL)
  ORDER BY nachname
```

Die PLAN-Klausel könnte auch dazu benutzt werden, explizit die Verwendung von *ix_adressen_nachname* vorzuschreiben:

```
SELECT *
  FROM t_adressen
  PLAN (t_adressen ORDER ix_adressen_nachname)
  ORDER BY nachname
```

Wenn ein Index für die Sortierung verwendet werden soll, dann muss davor in der PLAN-Klausel ORDER gesetzt werden. Schauen wir uns kurz die Ausführungszeiten an:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	650	2 211	265 206	268 067
Ausführung 2	564	2 172	264 351	267 087
Ausführung 3	542	2 155	270 282	272 979
Ausführung 4	545	2 169	270 520	273 234
Anweisung 2 Ausführung 1	658	95 723	202 996	299 377
Ausführung 2	578	96 469	203 392	300 439
Ausführung 3	591	95 581	204 754	300 926
Ausführung 4	592	95 576	203 131	299 299
Anweisung 3 Ausführung 1	761	2 189	270 160	273 110
Ausführung 2	758	2 205	263 393	266 356
Ausführung 3	755	2 184	263 624	266 563
Ausführung 4	763	2 176	265 437	268 376

Abbildung 3.33: Sortieren mit und ohne Index

Die Execute-Zeit liegt deutlich höher, wenn Sie keinen Index verwenden, dafür ist die Fetch-Zeit etwas besser. Warum das so ist, wurde in Kapitel 3.7.2 bereits erklärt.

Ob Sie den Server entscheiden lassen, dass er einen Index verwenden soll, oder ob sie ihm das per PLAN-Klausel selbst vorschreiben, spielt weder bei der *Execute*- noch bei der *Fetch*-Zeit eine Rolle. Allerdings liegt die *Prepare*-Zeit ein gutes Stück höher – die Idee, hier dem Server Arbeit abnehmen zu wollen, sollten Sie ganz schnell wieder verwerfen.

3.10.2 PLAN und JOIN

Interessant wird der Query-PLAN bei einem JOIN. Nehmen wir einmal an, wir wollten eine Liste aller Bestellungen, und zwar mit Vor- und Nachname des Kunden und dem Nachnamen des Mitarbeiters, der diese Bestellung bearbeitet hat. Einen solchen JOIN würde man wie folgt formulieren:

```
SELECT b.id,
    a.vorname, a.nachname,
    m.nachname AS Mitarbeiter
  FROM t_bestellung b
    INNER JOIN t_adressen a
      ON a.id = b.kunde
    INNER JOIN t_mitarbeiter m
      ON m.id = b.mitarbeiter
  WHERE b.id BETWEEN 1 AND 100
  ORDER BY b.id
```

Aus Rücksicht auf die Ausführungszeiten wollen wir nur die ersten 100 Bestellungen haben und nicht alle 75660, darüber hinaus soll nach der *id* sortiert werden, außerdem ist die Spaltenliste ein wenig gekürzt.

Geradeaus gedacht, haben wir einen INNER JOIN formuliert. Man könnte dieselbe Ergebnismenge aber auch mit einem LEFT OUTER JOIN, einem RIGHT OUTER JOIN sowie einem FULL OUTER JOIN erhalten. Unterschiede gibt es hier dennoch, und zwar bei den Ausführungszeiten:

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	897	20 829	404	22 130
Ausführung 2	795	20 746	402	21 943
Ausführung 3	782	20 845	404	22 031
Ausführung 4	775	20 647	401	21 823
Anweisung 2 Ausführung 1	778	5 588	398	6 764
Ausführung 2	774	5 608	393	6 775
Ausführung 3	779	5 580	393	6 752
Ausführung 4	889	5 607	395	6 891
Anweisung 3 Ausführung 1	720	6 281 734	409	6 282 863
Ausführung 2	719	4 010 886	405	4 012 010
Ausführung 3	745	4 007 592	406	4 008 743
Ausführung 4	728	4 007 604	406	4 008 738
Anweisung 4 Ausführung 1	783	22 027 691	403	22 028 877
Ausführung 2	780	22 028 392	405	22 029 577
Ausführung 3	782	22 030 774	405	22 031 961
Ausführung 4	779	22 027 983	407	22 029 169

Abbildung 3.34: Die verschiedenen JOINS

Dass der FULL OUTER JOIN gegenüber dem LEFT OUTER JOIN über dreitausendmal länger braucht, ist zwar beeindruckend, aber in der Praxis wäre man ohnehin nicht auf die Idee gekommen, hier einen FULL OUTER JOIN zu formulieren. Dass aber der LEFT OUTER JOIN etwa dreimal so schnell ist wie der normalerweise verwendete INNER JOIN, ist dann schon bemerkenswert.

Schauen wir uns einmal die Pläne der verschiedenen JOINS an, zunächst den des INNER JOIN:

```
PLAN SORT (JOIN (M NATURAL,
  B INDEX (RDB$PRIMARY2,RDB$FOREIGN7),
  A INDEX (RDB$PRIMARY1)))
```

Dazu im Vergleich den LEFT OUTER JOIN:

```
PLAN SORT (JOIN (JOIN (B INDEX (RDB$PRIMARY2),
  A INDEX (RDB$PRIMARY1)),
  M INDEX (RDB$PRIMARY3)))
```

Beim INNER JOIN werden erst alle Mitarbeiter gelesen, dann mittels *RDB$FOREIGN7* die Bestellungen hinzugefügt – solange deren *id* zwischen 1 und 100 liegt – und zuletzt die Kundenadressen ergänzt.

Beim LEFT OUTER JOIN wird mit den Bestellungen angefangen, denen werden erst die Kundenadressen und dann die Mitarbeiter hinzugefügt. Während beim INNER JOIN die Tabelle *t_bestellung* für jeden Mitarbeiter einmal gelesen wird, erfolgt dies beim LEFT OUTER JOIN nur ein einziges Mal.

Werfen wir einen Blick auf den RIGHT OUTER JOIN:

```
PLAN SORT (JOIN (M NATURAL,
  JOIN (A NATURAL,
  B INDEX (RDB$FOREIGN6))))
```

Hier wird zuerst ein JOIN zwischen allen (!) Adressen und den Bestellungen erstellt, immerhin unter Verwendung von *RDB$FOREIGN6*, dem Index für den Fremdschlüssel auf die Kundentabelle. Anschließend werden die Mitarbeiter hinzugefügt, danach auf die *id* gefiltert.

Noch trostloser sieht es beim FULL OUTER JOIN aus:

```
PLAN SORT (JOIN (M NATURAL,
  JOIN (A NATURAL,
  B INDEX (RDB$PRIMARY2))))
```

Hier wird eine Ergebnismenge aller Adressen aufgebaut, denen zwar nicht alle Bestellungen hinzugefügt werden – wegen *BETWEEN 1 AND 100* wird hier schon mittels *RDB$PRIMARY2* gefiltert –, die aber für den JOIN den vorhandenen Index nicht nutzt. Dass die Mitarbeitertabelle ohne Index gelesen wird, ist bei der Größe dieser Tabelle dann auch nicht mehr entscheidend.

Mit PLAN-Klausel

Nun könnte man auf die Idee kommen, dem INNER JOIN den Query-Plan des LEFT OUTER JOIN »unterzuschieben« – dieser weigert sich aber, die Mitarbeitertabelle mittels Index zu lesen. Wenn wir hier »nachgeben«, dann bleibt die Änderung der Ausführungsreihenfolge:

```
SELECT b.id,
    a.vorname, a.nachname,
    m.nachname AS Mitarbeiter
  FROM t_bestellung b
    INNER JOIN t_adressen a
      ON a.id = b.kunde
    INNER JOIN t_mitarbeiter m
      ON m.id = b.mitarbeiter
  WHERE b.id BETWEEN 1 AND 100
  PLAN SORT (JOIN (JOIN (B INDEX (RDB$PRIMARY2),
    A INDEX (RDB$PRIMARY1)),M NATURAL))
  ORDER BY b.id
```

Zum Vergleich der Query-Plan ohne Verwendung einer PLAN-Klausel:

```
PLAN SORT (JOIN (M NATURAL,
  B INDEX (RDB$PRIMARY2,RDB$FOREIGN7),
  A INDEX (RDB$PRIMARY1)))
```

Und siehe da, nun ist der INNER JOIN nicht nur schneller geworden, sondern auch noch schneller als der LEFT OUTER JOIN (wenn auch nicht viel, *Anweisung 3*).

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	907	20 819	404	22 130
Ausführung 2	844	20 597	400	21 841
Ausführung 3	715	20 767	403	21 885
Ausführung 4	710	20 746	402	21 858
Anweisung 2 Ausführung 1	778	5 600	400	6 778
Ausführung 2	760	5 574	398	6 732
Ausführung 3	765	5 568	393	6 726
Ausführung 4	767	5 571	393	6 731
Anweisung 3 Ausführung 1	916	5 114	410	6 440
Ausführung 2	919	5 115	408	6 442
Ausführung 3	904	5 116	409	6 429
Ausführung 4	921	5 109	407	6 437

Abbildung 3.35: INNER JOIN mit PLAN-Klausel

Auch hier sehen wir wieder, dass die Verwendung einer PLAN-Klausel die *Prepare*-Zeit erhöht. Der Versuch, mittels dieses Plans nun auch den LEFT OUTER JOIN zu beschleunigen, schlug fehl, er wurde sogar deutlich langsamer.

Zum Vergleich: Unterabfragen

Wollen wir für diese Aufgabenstellung nun auch noch Unterabfragen verwenden. Zunächst nur eine Unterabfrage für die Mitarbeitertabelle:

```
SELECT b.id,
    a.vorname, a.nachname,
    (SELECT m.nachname
      FROM t_mitarbeiter m
      WHERE m.id = b.mitarbeiter) AS Mitarbeiter
  FROM t_bestellung b
    INNER JOIN t_adressen a
      ON a.id = b.kunde
  WHERE b.id BETWEEN 1 AND 100
  ORDER BY b.id
```

Hier würde der Query-Plan wie folgt aussehen:

```
PLAN (M INDEX (RDB$PRIMARY3))
PLAN JOIN (B ORDER RDB$PRIMARY2,A INDEX (RDB$PRIMARY1))
```

Und nun noch als Version gänzlich ohne JOIN:

```
SELECT b.id,
    (SELECT a.vorname
      FROM t_adressen a
      WHERE a.id = b.kunde) AS Vorname,
    (SELECT a.nachname
      FROM t_adressen a
      WHERE a.id = b.kunde) AS Nachname,
    (SELECT m.nachname
      FROM t_mitarbeiter m
      WHERE m.id = b.mitarbeiter) AS Mitarbeiter
  FROM t_bestellung b
  WHERE b.id BETWEEN 1 AND 100
  ORDER BY b.id
```

Was zu folgendem Plan führt:

```
PLAN (M INDEX (RDB$PRIMARY3))
PLAN (A INDEX (RDB$PRIMARY1))
PLAN (A INDEX (RDB$PRIMARY1))
PLAN (B ORDER RDB$PRIMARY2)
```

Selbstverständlich kann ich es mir nicht verkneifen, der »Niemals Unterabfragen-Fraktion« den Vergleich der Ausführungszeiten zu präsentieren (als erste Anweisung wieder den INNER JOIN ohne expliziten Plan):

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	912	20 826	404	22 142
Ausführung 2	690	20 741	403	21 834
Ausführung 3	681	20 813	404	21 898
Ausführung 4	847	20 656	400	21 903
Anweisung 2 Ausführung 1	735	6 134	380	7 249
Ausführung 2	730	6 265	381	7 376
Ausführung 3	788	6 115	377	7 280
Ausführung 4	729	6 228	379	7 336
Anweisung 3 Ausführung 1	716	8 658	378	9 752
Ausführung 2	728	8 561	377	9 666
Ausführung 3	714	8 548	376	9 638
Ausführung 4	732	8 557	375	9 664

Abbildung 3.36: JOIN gegen Unterabfrage

Nun, wir haben gesehen, dass der LEFT OUTER JOIN schneller ist und dass man mittels einer PLAN-Klausel den INNER JOIN noch schneller bekommt, aber im Vergleich zu dem, was ein unvorbereiteter INNER JOIN hier(!) braucht, sind die Unterschiede belanglos.

Und auch an dieser Stelle will ich den Hinweis wiederholen: Je nachdem, welche Tabellen und Indizes Sie verwenden, kann das immer mal wieder ganz anders aussehen. Gehen Sie dort, wo es auf Performance ankommt, nicht nach »Schema F« vor, sondern messen Sie die verschiedenen Möglichkeiten durch.

3.10.3 PLAN bei der WHERE-Klausel

Gibt es für eine Spalte einen Index, dann wird dieser im Regelfall verwendet, wenn darauf gefiltert wird, und das beschleunigt auch das Filtern. Vergleichen wir folgende Anweisungen:

```
SELECT *
  FROM t_adressen
  WHERE nachname = 'Müller'

SELECT *
  FROM t_adressen
  WHERE nachname = 'Müller'
  PLAN (t_adressen NATURAL)

SELECT *
  FROM t_adressen
  WHERE nachname = 'Müller'
  PLAN (t_adressen INDEX (ix_adressen_nachname))
```

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	679	2 232	1 537	4 448
Ausführung 2	494	2 167	1 526	4 187
Ausführung 3	493	2 166	1 527	4 186
Ausführung 4	497	2 163	1 527	4 187
Anweisung 2 Ausführung 1	520	29 829	18 212	48 561
Ausführung 2	491	29 874	18 290	48 655
Ausführung 3	492	30 446	18 213	49 151
Ausführung 4	499	32 796	19 068	52 363
Anweisung 3 Ausführung 1	696	2 187	1 556	4 439
Ausführung 2	691	2 249	1 531	4 471
Ausführung 3	706	2 255	1 535	4 496
Ausführung 4	769	2 232	1 537	4 538

Abbildung 3.37: Index und WHERE-Klausel

Im Gegensatz zum Sortieren erhöhen wir hier nicht nur die *Execute*-, sondern auch die *Fetch*-Zeit deutlich, wenn wir auf den Index verzichten. Und auch hier wieder erhöht die PLAN-Klausel die *Prepare*-Zeit.

Bei der folgenden Anweisung wäre es unsinnig, über den Index zu gehen, deshalb wird dies auch vom Server vermieden:

```
SELECT *
  FROM t_adressen
  WHERE nachname  <> 'Müller'
```

Nach meiner Erfahrung muss man schon sehr außergewöhnliche SQL-Anweisungen schreiben, damit der Optimizer im Zusammenhang mit einer WHERE-Klausel suboptimale Query-Pläne erstellt. Von daher besteht auch wenig Anlass, hier mit der PLAN-Klausel zu agieren.

3.11 INSERT, UPDATE, DELETE

Die Befehle INSERT, UPDATE und DELETE dienen dem Einfügen, Ändern und Löschen von Daten. Solche Statements werden oft von den Datenzugriffskomponenten eigenständig generiert, in einigen Fällen müssen Sie das jedoch selbst formulieren.

3.11.1 INSERT

Die vollständig abstrakte INSERT-Anweisung lautet wie folgt:

```
INSERT INTO tabellenname
        (spalte1, spalte2...)
    VALUES
        (wert1, wert2...)
```

Das folgende Beispiel zeigt, wie die Tabelle *t_produkt* ergänzt wird:

```
INSERT INTO t_produkt (kat, photograph, titel, preis)
  VALUES ('Städte', 'Manfred Jäckle', 'Ulm', 14.30)
```

Sie müssen gar nicht alle Spalten belegen, es fehlt hier beispielsweise die Spalte *id*. Normalerweise würde das zu einer Exception führen, weil wir *id* als NOT NULL definiert haben. Es gibt hier jedoch auch einen Trigger, der automatisch einen Generatorwert einfügt.

Man kann auf die Aufzählung der Spalten verzichten, wenn man Werte für alle Spalten hat und die in der »richtigen« Reihenfolge – also so wie in der Tabellendefinition – angibt:

```
INSERT INTO t_produkt
VALUES (175, 'Städte', 'Manfred Jäckler', 'Bochum', 14.30)
```

Wenn Sie für eine Tabelle einen Generator erstellt haben, dann sollten Sie diesen auch tunlichst immer verwenden, sonst führt dies früher oder später zu Problemen.

Daten aus einer Unterabfrage

Bislang haben wir bei INSERT eine Werteliste mit Konstanten übergeben. Es ist aber auch möglich, die einzufügenden Werte von einer Unterabfrage zu übernehmen. Statt des Schlüsselwortes VALUES und der Werteliste wird dann die Unterabfrage eingefügt.

```
INSERT INTO t_mitarbeiter (vorname, nachname)
  SELECT vorname, nachname
    FROM t_adressen
    WHERE id = 100
```

Diese Anweisung kopiert einen Datensatz aus *t_adressen* in die Mitarbeiter-Tabelle.

Unterabfrage für einen Spaltenwert

Es gibt auch die Möglichkeit, nur den Wert für eine einzelne Spalte aus einer Unterabfrage zu beziehen. Wir hatten vorhin bei *t_produkt* das Problem eines Generatorwertes, wenn wir auf die Aufzählung der Spalten verzichten wollen. Mittels einer Unterabfrage könnte man zu solch einem Generatorwert kommen.

```
INSERT INTO t_produkt
  VALUES ((SELECT GEN_ID(g_produkt, 1)
      FROM t_produkt
      WHERE id = 1),
   'Städte', 'Manfred Jäckler', 'Brunsbüttel', 14.30)
```

Die Unterabfrage ist hier in Klammern zu setzen und sie ist so zu formulieren, dass sie nur einen einzelnen Wert zurückgibt – aus diesem Grund erfolgt die Beschränkung mit *WHERE id = 1*.

3.11.2 UPDATE

Die vollständig abstrakte UPDATE-Anweisung lautet wie folgt:

```
UPDATE tabellenname
    SET spalte1 = wert1, spalte2 = wert2, ...
    WHERE bedingung
```

Im folgenden Beispiel wird die Spalte *titel* auf Mannheim gesetzt, und zwar für alle Datensätze, für die gilt *id = 177* – das ist aber nur ein einziger.

```
UPDATE t_produkt
  SET titel = 'Mannheim
  WHERE id = 177
```

Würde die WHERE-Klausel entfallen, so würde die Änderung bei allen Datensätzen vorgenommen.

UPDATE mit Unterabfrage

In einer UPDATE-Anweisung sind auch Unterabfragen möglich, und zwar sowohl bei der Zuweisung der Werte als auch bei der WHERE-Klausel. In folgendem Beispiel wurde beides verwendet:

```
UPDATE t_produkt
  SET titel = (SELECT titel
    FROM t_produkt
    WHERE id = 5)
  WHERE id = (SELECT MIN(id)
    FROM t_produkt
    WHERE titel = 'Mannheim')
```

Mit der Aggregatfunktion MIN wird garantiert, dass diese Unterabfrage nur einen einzigen Datensatz zurückgibt.

3.11.3 DELETE

Die vollständig abstrakte DELETE-Anweisung lautet wie folgt:

```
DELETE FROM tabellenname
  WHERE bedingung
```

Wir haben die DELETE-Anweisung schon verwendet, um neu eingefügte Datensätze wieder zu löschen:

```
DELETE FROM t_produkt
  WHERE id = 177
```

Selbstverständlich können Sie die WHERE-Klausel so formulieren, dass mehrere Datensätze gelöscht werden:

```
DELETE FROM t_produkt
  WHERE id > 100
```

In diesem Fall würden alle Datensätze gelöscht, deren Nummer größer als hundert ist – vorausgesetzt, dass keine Fremdschlüsselverletzung dies verhindern würde.

DELETE ohne WHERE-Klausel

Sie können auch eine DELETE-Anweisung ohne WHERE-Klausel formulieren – dann werden eben alle Datensätze gelöscht.

```
DELETE FROM t_produkt
```

Die Anweisung ist syntaktisch korrekt, die Ausführung wird aber trotzdem verweigert, weil ein Fremdschlüssel die Tabelle referenziert.

DELETE mit Unterabfrage

Auch bei der DELETE-Anweisung sind Unterabfragen erlaubt. Nehmen wir einmal an, wir wollen alle Produkte aus dem Sortiment werfen, die bislang kein einziges Mal verkauft wurden:

```
DELETE FROM t_produkt
  WHERE id IN (SELECT o.id
    FROM t_produkt o
    LEFT OUTER JOIN t_posten p
    ON p.produkt = o.id
    GROUP BY o.id
    HAVING COUNT(p.stueckzahl) = 0)
```

Ausführen sollten Sie so etwas jedoch allenfalls im Nachtlauf, weil für jeden Datensatz in der Tabelle *t_produkt* die Unterabfrage ausgeführt wird. Wenn dann *t_produkt* nicht vollständig in den Speicher passt und jedes Mal die Sache von der Festplatte gelesen werden muss, dann dauert's halt mal wieder ein wenig länger.

4 Definition der Metadaten

Unter den Metadaten versteht man die Daten, welche den Aufbau einer Datenbank beschreiben: Welche Tabellen gibt es, welche Spalten enthalten diese, wer darf darauf zugreifen und so weiter.

4.1 Domänen

Domänen sind Wertebereiche, auf denen dann die einzelnen Tabellenspalten beruhen; sie entsprechen den Typen bei C oder Pascal. Die Tabelle *rdb$relation_fields* – in ihr werden die Spaltendefinitionen der einzelnen Tabellen gespeichert – referenziert dazu die Tabelle *rdb$fields*. Für jede Spalte, für die nicht explizit eine Domäne angegeben wird, wird automatisch eine erstellt. Domänennamen wie *rdb$22* sind Beispiele für solche automatisch erstellten Domänen.

Domänen sind aber nicht nur Wertebereiche. Es ist bei SQL auch möglich, Gültigkeitsprüfungen durchführen zu lassen oder Standardwerte vorzugeben.

Warum Domänen verwenden?

Für den Einsteiger ist es oft nicht ersichtlich, warum in diesem oder jenem Fall eine Domäne definiert wird, die lediglich auf einen Standard-Typ verweist. Für eine Liste mit Namen könnte man einfach formulieren:

```
CREATE TABLE t_namen
    (nummer INTEGER NOT NULL,
    vornamen VARCHAR(20),
    nachnamen VARCHAR(30)
    PRIMARY KEY (nummer))
```

Mit dem Einsatz von DOMÄNEN wird alles erst einmal komplizierter, da zunächst die DOMAINS zu definieren sind:

```
CREATE DOMAIN d_nummer AS INTEGER NOT NULL;
CREATE DOMAIN d_vornamen AS VARCHAR(20);
CREATE DOMAIN d_nachnamen AS VARCHAR(30);

CREATE TABLE t_namen
    (nummer d_nummer,
    vornamen d_vornamen,
    nachnamen d_nachnamen,
    PRIMARY KEY (nummer))
```

Was soll nun in diesem Fall der Vorteil von Domänen sein? Im hier gezeigten Beispiel gibt es keinen. Nun bestehen Datenbanken aber in der Regel aus vielen Tabellen mit einer meist schon unübersichtlichen Zahl von Spalten. Und da fangen dann die Probleme an:

Nehmen wir einmal an, wir haben mehrere Tabellen, in denen Namen gespeichert sind, ähnlich wie in *t_namen*. Um Probleme der später geschilderten Art zu vermeiden, wurden diese bei der Definition der Datenbank alle einheitlich als VARCHAR(30) gespeichert.

Nun muss in eine der Tabellen ein Name eingegeben werden, der länger als 30 Zeichen ist. Wenn so etwas im laufenden Betrieb vorkommt, ist das zwar ärgerlich, aber nicht weiter tragisch: Inzwischen kann man auch bei InterBase bei bereits gefüllten Tabellen den Typ der Spalte ändern, solange das Datenbanksystem dabei nicht die Gefahr eines Datenverlustes sieht – und das ist bei der Verbreiterung einer VARCHAR-Spalte bestimmt nicht der Fall.

Die Gefahr dabei ist jedoch, dass man nur diejenige Spalte verbreitert, die aktuell zu klein ist, andere aber nicht, obwohl die Datenbanksoftware davon ausgeht, dass bestimmte Spalten gleich breit sind. Der Versuch, den Inhalt von der einen Spalte in die andere zu kopieren, wird dann zu einer Exception führen, dann wird auch diese Spalte verbreitert, nach zwei Wochen die dritte … Und jedes Mal geht ein Programmierer auf die Fehlersuche und kann ein Anwender nicht weiterarbeiten.

Bei der Verwendung einer Domäne hätte man diese verbreitert und alle darauf beruhenden Spalten hätten auch danach eine einheitliche Länge.

Außerhalb des englischsprachigen Raums gibt es einen weiteren Grund für den Einsatz von Domänen: Die Angabe von Zeichensatz und Sortierreihenfolge bei jeder VARCHAR-Spalte würde andernfalls erhebliche Schreibarbeit nach sich ziehen. Bei einer Domäne muss das nur einmal eingegeben werden:

```
CREATE DOMAIN st AS VARCHAR(25)
    CHARACTER SET ISO8859_1 COLLATE DE_DE;
```

4.1.1 Datentypen

Bei der Definition einer Domäne muss zwingend ein Datentyp angegeben werden. InterBase bietet fünf Kategorien von Datentypen:

- Zeichenketten (»Strings«)
- Ganzzahlen
- Rationale Zahlen
- Datum und Zeit
- BLOBs

Zeichenketten

Als Zeichenketten gibt es CHAR(n) und VARCHAR(n), in Klammern muss jeweils die Anzahl der Zeichen angegeben werden. CHAR ist eine Zeichenkette fester Länge, VARCHAR eine Zeichenkette flexibler Länge. Was bedeutet das in der Praxis?

```
CREATE TABLE t_strings
(nummer INTEGER NOT NULL RPIMARY KEY,
text_1 CHAR(20),
text_2 VARCHAR(20));

INSERT INTO t_strings VALUES (1, 'Test', 'Test');

SELECT *
FROM t_strings;
```

	NUMMER	TEXT_1	TEXT_2
▶	1	Test	Test

Abbildung 4.1: Unterschied zwischen CHAR und VARCHAR

Wie *Abbildung 4.1* zeigt, werden die fehlenden Zeichen bei CHAR durch Leerzeichen ersetzt. So etwas ist nur sehr selten zweckmäßig, so dass VARCHAR deutlich häufiger verwendet wird.

Zeichenketten können bis zu 32767 Byte groß werden. Da die meisten Zeichensätze 8 Bit breit sind, können ebenso viele Zeichen gespeichert werden. Bei 16 Bit breiten Zeichensätzen (Unicode) können dann nur noch 16384 Zeichen gespeichert werden.

Eigentlich gäbe es da noch den Typen NCHAR(n), ausgeschrieben NATIONAL CHAR(n), welcher automatisch den Zeichensatz ISO8859_1 verwendet. Da aber auch dies eine Zeichenkette fester Länge ist, wird er nicht besonders oft verwendet.

Ganzzahlen

InterBase stellt für Ganzzahlen zwei Typen zur Verfügung:

- Der 16-Bit-Typ SMALLINT mit einem Wertebereich von –32768 bis 32767
- Der 32-Bit-Typ INTEGER mit einem Wertebereich von –2147.483648 bis 2147.483647

Rationale Zahlen

Rationale Zahlen sind entweder Gleit- oder Festkommazahlen:

- FLOAT ist ein 32-Bit-Gleitkommatyp mit einem Wertebereich von +/- 1,175 * 10^{-38} bis 3,402 * 10^{38}
- DOUBLE PRECISION ist ein 64-Bit-Gleitkommatyp mit einem Wertebereich von +/- 2,225 * 10^{-308} bis 1,797 * 10^{308}
- Die Typen NUMERIC(precision, scale) und DECIMAL(precision, scale) sind Festkommazahlen, die intern als Ganzzahlen gespeichert werden und bei der Ein- und Ausgabe entsprechend umgerechnet werden. Durch die Speicherung als Ganzzahlen werden Rundungsungenauigkeiten vermieden.
- Mit dem Parameter *precision* wird angegeben, wie viele Stellen die zu speichernden Zahlen maximal haben, *scale* spezifiziert die Anzahl der zu speichernden Nachkommastellen. Man braucht jedoch *precision* nicht übermäßig ernst zu nehmen: In eine Domäne vom Typ NUMERIC(5,2) kann problemlos die Zahl 123456,78 gespeichert werden – durch die *precison* von fünf muss der Typ INTEGER als Basistyp verwendet werden und 12345.678 passt da noch in den Speicher. Um den Wertebereich wirksam zu beschränken, sollte deshalb lieber eine CONSTRAINT-Klausel verwendet werden.
- Je nach *precision* werden die Festkommatypen als 16-, 32- oder 64-Bit-Werte gespeichert, Näheres kann bei Interesse im *data definition guide* nachgeschlagen werden.

Datum und Zeit

Zum Speichern von Datum und Zeit gibt es beim SQL-Dialekt 3 drei Typen:

- Der Typ DATE speichert ein Datum zwischen dem 1. Januar 100 und dem 29. Februar 32768. Es handelt sich dabei um einen 32-Bit-Typen.
- Der Typ TIME speichert eine Uhrzeit, und zwar in Einheiten einer zehntausendstel Sekunde seit Mitternacht. Dafür wird ein 32-Bit-Typ verwendet.
- TIMESTAMP kombiniert nun DATE und TIME und benötigt dafür 64 Bit.

Beim SQL-Dialekt 1 gibt es nur den Typen DATE, der TIMESTAMP entspricht. Die Umstellung von Dialekt 1 nach 3 »hakt« dann auch etwas an dieser Stelle.

BLOBs

Der Datentyp BLOB steht für *binary large object*. Mit BLOB können beispielsweise Bilder, Videos oder Audiodaten gespeichert werden. Wie groß ein solcher BLOB werden kann, hängt von der *Database Page Size* und vom Betriebssystem ab. Bei der inzwischen »üblichen« Seitengröße von 4 Kbyte könnte der BLOB 4 Gbyte groß werden. Da die Dateigröße unter Windows jedoch auf 2 Gbyte begrenzt ist, dürfte unter Windows der BLOB nicht größer als 2 Gbyte werden.

Wenn Sie mehrere große BLOBs speichern wollen, dann wird die Datenbankgröße über die maximale Dateigröße des Betriebssystems steigen. Dies zwingt Sie dann dazu, die Datenbank auf mehrere Dateien aufzuteilen – dann darf sie aber auch einige Tbyte groß werden.

Für BLOBs können Sie mehrere SUB_TYPES definieren. SUB_TYPE 1 ist von InterBase bereits vordefiniert und beinhaltet Text. Im folgenden sehen Sie eine solche Definition:

```
CREATE DOMAIN d_memo AS BLOB SUB_TYPE 1
```

Datenkonvertierung

InterBase verhält sich bei »Daten-Mischmasch« ziemlich gutmütig, die folgende Anweisung beispielsweise bereitet keine Probleme:

```
SELECT *
    FROM t_mitarbeiter
    WHERE nummer BETWEEN 1 AND '10'
```

Sollte die Umwandlung einmal nicht automatisch passieren, kann die Funktion CAST verwendet werden, welche die Typen DATE, CHARACTER und NUMERIC ineinander umwandelt:

```
... WHERE hire_date = CAST(interview_date AS DATE)
```

Arrays

Zu den Forderungen der ersten Normalform gehört, dass alle Spaltenwerte *atomar* sind, sich also nicht sinnvoll weiter unterteilen lassen – Spalten, die auf Arrays beruhen, erfüllen diese Bedingungen nicht.

Der Wunsch nach Arrays ist in der Regel ein Zeichen für eine nicht optimal entworfene Datenbank. Es gibt allerdings Spezialfälle, in der Regel aus der Technik oder der Wissenschaft, in denen sich eine Aufgabe mit einem Array deutlich einfacher lösen lässt. Für solche Aufgabenstellungen besteht die Möglichkeit, auch Arrays zu definieren.

Da Arrays recht selten verwendet werden, soll dieses Thema hier nicht vertieft werden. Im Bedarfsfall informieren Sie sich aus den InterBase-Handbüchern.

4.1.2 DEFAULT-Werte

Bei der Definition von Domänen haben Sie die Möglichkeit, Werte vorzugeben, die immer dann eingefügt werden, wenn der Anwender keine Eingabe macht. Hier gibt es grundsätzlich vier Möglichkeiten:

- einen explizit angegebenen Wert
- den Wert NULL

- die Variable USER
- das heutige Datum mit 'NOW'

Den Wert NULL als DEFAULT-Wert zu definieren macht nicht viel Sinn, schließlich fügt die Datenbank immer NULL-Werte ein, wenn der Anwender keine Eingabe macht. Die Variable USER beinhaltet den Wert des Benutzernamens, mit dem der Anwender sich beim Server eingeloggt hat.

Mit Hilfe von DEFAULT-Werten ist es sogar möglich, Spaltenwerte ganz automatisch einzufügen. Als Beispiel sei hier die Möglichkeit genannt, bei jedem Datensatz mitzuspeichern, wer ihn denn eingegeben hat:

```
CREATE DOMAIN d_username
    AS VARCHAR(15)
    DEFAULT USER

CREATE TABLE namen
     (vornamen VARCHAR(20),
     nachnamen VARCHAR(30),
     eingegeben d_username)

INSERT INTO namen (vornamen, nachnamen)
    VALUES ('Michael', 'Mustermann')

SELECT * FROM namen
```

	VORNAMEN	NACHNAMEN	EINGEGEBEN
▶	Michael	Mustermann	SYSDBA

Abbildung 4.2: USER als Vorgabewert

Dagegen macht die automatische Eingabe von Festwerten meist nicht viel Sinn – hier würde man dann die Möglichkeit benötigen, den DEFAULT-Wert zu überschreiben. Nehmen wir hier einmal die Adressendatenbank einer Schule. Für gewöhnlich werden die eingegebenen Adressen die von Schülern sein, nur manchmal muss ein Lehrer, ein Hausmeister, eine Reinigungskraft eingegeben werden:

```
CREATE DOMAIN dberuf AS VARCHAR(15) DEFAULT 'Schüler'

CREATE TABLE tadressen
  (vornamen VARCHAR(20),
          nachnamen VARCHAR(30),
           beruf dberuf)

INSERT INTO tadressen (vornamen, nachnamen)
    VALUES ('Adam', 'Amsel')
```

```
INSERT INTO tadressen (vornamen, nachnamen)
    VALUES ('Berta', 'Borst')

INSERT INTO tadressen (vornamen, nachnamen, beruf)
    VALUES ('Cäsar', 'Conradi', 'Rektor')
```

Ich möchte noch darauf hinweisen, dass ich diese Tabelle nur als Beispiel für DEFAULT-Werte entworfen habe. In der Praxis würde man selbstverständlich eine zweite Tabelle mit den verschiedenen Berufen anlegen und eine Referenz darauf bilden (und obendrein eine Spalte ID für den Primärschlüssel anlegen).

(Das Feld *beruf* benötigt jeweils 15 Bytes, bei angenommenen 500 Datensätzen sind dies rund 7,5 Kbyte. Ein Referenzfeld kommt mit zwei Byte aus, das wäre hier 1 Kbyte, und außerdem wäre die Änderung einfacher, wenn es statt *Schüler* nun »politisch korrekt« *SchülerIn* oder *Schüler(in)* heißen müsste.)

4.1.3 Eingabe erzwingen

In manchen Feldern dürfen keine NULL-Werte vorhanden sein, weil beispielsweise darauf eine Referenz gebildet wird oder eine Eingabe ohne diese Angabe nicht sinnvoll wäre. Hier kann dann der Befehl NOT NULL gegeben werden:

```
CREATE DOMAIN d_test AS VARCHAR(15) NOT NULL
```

Beachten Sie, dass Sie alle Felder, die Sie als Primär- oder Sekundärschlüssel verwenden wollen, als NOT NULL definieren müssen.

4.1.4 Gültigkeitsprüfungen

Es kommt immer mal wieder vor, dass Personen eine Datenbankanwendung bedienen, die nicht genau wissen, was sie tun müssen, oder die sich bei der Eingabe vertippen. Auf diese Weise wäre es möglich, dass Daten in die Datenbank gelangen, die nicht richtig sind. Vielfach lässt sich das nicht vermeiden, doch für einige Fälle kann man Gültigkeitsprüfungen implementieren, die dafür sorgen, dass die Aufnahme unsinniger Daten mit einer Fehlermeldung verweigert wird.

Das Gehalt eines Angestellten kann beispielsweise nie negativ sein:

```
CREATE DOMAIN d_gehalt
    AS FLOAT
    CHECK (VALUE > 0)
```

In der CHECK-Klausel wird der eingegebene Wert stets VALUE genannt. Beachten Sie auch, dass die CHECK-Klausel stets in Klammern zu setzen ist.

An dieser Stelle gleich eine Warnung: Bedenken Sie immer, dass die Datenbank sich weigern wird, einen Datensatz anzunehmen, der die CHECK-Bedingung(en) nicht erfüllt. Deshalb ist es nicht sinnvoll, hier beispielsweise den tariflichen Min-

destlohn einzutragen – sobald eine Aushilfskraft für ein paar Stunden eingestellt wird, müsste diese sehr großzügig bezahlt werden, damit das Gehalt eingegeben werden kann.

Bei der Gültigkeitsprüfung sind auch AND- und OR-Verknüpfungen erlaubt.

```
CREATE DOMAIN d_gehalt
    AS NUMERIC(7,2)
    CHECK ((VALUE > 0)
        AND (VALUE < 12000))
```

Darüber hinaus gibt es die Möglichkeit, über die CHECK-Bedingung Mengentypen zu definieren:

```
CREATE DOMAIN dberuf AS VARCHAR(15)
    DEFAULT "Schüler"
    CHECK (VALUE IN
        ('Schüler', 'Lehrer', 'Rektor', 'Hausmeister'))
```

Was ich davon halte, hier keine Extra-Tabelle zu erstellen und eine Referenz darauf zu bilden, habe ich vorhin bereits erwähnt. Hier kommt noch erschwerend hinzu, dass es extrem aufwändig wäre, während des Betriebs der Datenbank diese Menge beispielsweise um den Eintrag *Reinigungskraft* zu ergänzen.

Unterabfragen in der CHECK-Klausel

In der CHECK-Klausel sind auch Unterabfragen möglich. Bei der folgenden Domäne würde verhindert, dass auf *d_test* beruhende Spalten Werte annehmen, die nicht größer als die höchste Mitarbeiternummer sind.

```
CREATE DOMAIN d_test
    AS INTEGER
    CHECK (VALUE > (SELECT MAX(nummer)
        FROM t_mitarbeiter))
```

Konstruktionen wie die eben gezeigte sind mit Vorsicht zu genießen, da die Erfüllung der CHECK-Klausel nur zum Zeitpunkt der Eingabe oder des Änderns der Daten garantiert ist. Es wäre ohne weiteres möglich, anschließend Datensätze mit höherer Nummer in die Tabelle *t_mitarbeiter* einzugeben.

4.1.5 Zeichensatz und Sortierreihenfolge

Allen Domänen, die Texte enthalten, kann ein Zeichensatz und/oder eine Sortierreihenfolge zugewiesen werden.

```
CREATE DOMAIN st
    AS VARCHAR(25)
    CHARACTER SET ISO8859_1
    COLLATE DE_DE;
```

Der Zeichensatz *ISO8859_1* und die Sortierreihenfolge *DE_DE* eignen sich für die Anwendung im deutschsprachigen Raum, sie erlauben also die deutschen Umlaute und sortieren sie auch richtig ein.

4.1.6 Domänen ändern

Mit ALTER DOMAIN lässt sich eine Domäne ändern, inzwischen sogar deren Datentyp. Einzig eine NOT NULL-Anweisung lässt sich weder hinzufügen noch entfernen.

Erstellen wir zunächst mal eine Domäne:

```
CREATE DOMAIN d_test
    AS INTEGER
```

Nun wollen wir daraus einen VARCHAR machen:

```
ALTER DOMAIN d_test TYPE VARCHAR(20)
```

Die Zuweisung eines anderen Datentypen ist nur dann möglich, wenn InterBase nicht die Gefahr eines Datenverlustes wittert. So lassen sich beispielsweise Zeichenketten verbreitern, nicht jedoch schmaler machen. Die Umwandlung von INTEGER in VARCHAR(20) ist problemlos – umgekehrt geht das jedoch nicht mehr, selbst dann nicht, wenn noch keine einzige Tabellenspalte auf dieser Domäne beruht.

Zunächst wollen wir den DEFAULT-Wert *Schüler* hinzufügen. Zum Ändern einer Tabelle wird die Anweisung ALTER DOMAIN verwendet.

```
ALTER DOMAIN d_test
    SET DEFAULT 'Schüler'
```

Nun soll auch noch eine CHECK-Klausel eingefügt werden. In diesem Beispiel sollen nur solche Werte erlaubt sein, die den Buchstaben *ü* (in Kleinschreibung) enthalten.

```
ALTER DOMAIN d_test
    ADD CHECK (VALUE LIKE '%ü%')
```

So, wie sich Vorgabewert und CHECK-Klausel einzeln setzen lassen, so kann man sie auch einzeln wieder entfernen. Um die CHECK-Klausel zu entfernen, verwenden Sie die Anweisung DROP CONSTRAINT.

```
ALTER DOMAIN d_test
    DROP CONSTRAINT
```

Der Vorgabewert wird mit DROP DEFAULT entfernt.

```
ALTER DOMAIN d_test
    DROP DEFAULT
```

Beachten Sie auch, dass Sie keine CHECK-Klausel »ergänzen« können. Wird eine weitere CHECK-Klausel gewünscht, dann löschen Sie die bestehende und fügen Sie die alte und die neue CHECK-Klausel wieder hinzu, die Sie mit dem AND-Operator verknüpfen.

4.1.7 Domänen löschen

Sehr einfach ist dann die Anweisung, um eine Domäne zu löschen (beispielsweise um sie danach mit einem anderen Datentyp neu zu erstellen).

```
DROP DOMAIN d_test
```

Beachten Sie bitte, dass die Domäne nicht in irgendeiner Spaltendefinition verwendet werden darf, um gelöscht werden zu können. Dies hat weit reichende Folgen: Bevor eine Tabelle gelöscht werden kann, müssen zunächst darauf beruhende VIEWS, TRIGGER und STORED PROCEDURES entfernt werden.

Bei weit verbreiteten Domänen muss man tatsächlich die komplette Datenbank »abbauen«, um sie löschen zu können. Hier ist es in der Regel dann einfacher, die Datenbank per SQL-Script komplett neu zu erstellen und anschließend die Daten zu kopieren.

4.2 Tabellen

Gemäß der SQL-Systematik werden Tabellen mit CREATE TABLE erstellt, mit ALTER TABLE geändert und mit DROP TABLE gelöscht.

4.2.1 CREATE TABLE

Um neue Tabellen zu erstellen, wird die Anweisung CREATE TABLE verwendet. Bei dieser Anweisung sind recht viele Optionen möglich, die wir nun nach und nach behandeln werden.

Zuvor aber aus Gründen der Vollständigkeit eine Bemerkung: Sie können – wenn Sie dafür Gründe sehen – dateibasierte Tabellen erstellen. Dies ist in der Praxis selten und soll nicht weiter erläutert werden; Es wäre im *Data Definition Guide* im Kapitel *Using der EXTERNAL FILE option* ausführlich beschrieben.

Spalten erstellen

Mit der folgenden Anweisung wird eine Tabelle erstellt, die zwei Spalten (*nummer* und *bezeichnung*) besitzt:

```
CREATE TABLE t_test
    (nummer INTEGER,
    bezeichnung VARCHAR(20))
```

Nach der Anweisung CREATE TABLE wird der Tabellenname genannt, in diesem Fall *t_test*. Der Tabellenname muss ein gültiger Bezeichner sein, vermeiden Sie also deutsche Umlaute und SQL-Schlüsselwörter. Wenn Sie das Präfix *t_* voranstellen, dann können Sie Tabellenbezeichner leicht von anderen Bezeichnern unterscheiden. Außerdem besteht dann nicht die Gefahr, versehentlich ein SQL-Schlüsselwort zu verwenden.

In Klammern folgt nun die Tabellendefinition. In unserem Beispiel werden zwei Spalten erstellt: Die Spalte *nummer* speichert Ganzzahlen, die Spalte *bezeichnung* Zeichen mit maximal 20 Zeichen Länge.

Optionen der Spaltendefinition

Wir können bei der Spaltendefinition alle Optionen verwenden, die wir von den Domänen her kennen:

```
CREATE TABLE t_test
    (nummer INTEGER
        NOT NULL,
    datum DATE
        DEFAULT 'NOW,
    preis FLOAT
        NOT NULL
        CHECK (preis >0.01),
    bemerkung VARCHAR(20)
        CHARACTER SET ISO8859_1
        COLLATE DE_DE)
```

- Mit NOT NULL kann eine Eingabe erzwungen werden. Dies ist insbesondere dann erforderlich, wenn für diese Spalte ein Schlüssel erstellt werden soll.
- Sie können DEFAULT-Werte definieren.
- Mit einer CHECK-Klausel kann eine Gültigkeitsprüfung vorgenommen werden. Beachten Sie bitte, dass hier nicht mehr das Spalten-Synonym VALUE verwendet wird, sondern der Spaltenbezeichner.
- Um einen Zeichensatz und eine Sortierreihenfolge festzulegen, verwenden Sie CHARACTER SET und COLLATE.

Domänenbasierte Spaltendefinition

Statt eines Spaltentyps kann auch eine Domäne verwendet werden. Diese wird dann mit allen Optionen Grundlage der Spaltendefinition. Zur Anweisung PRIMARY KEY kommen wir später:

```
CREATE DOMAIN ln AS INTEGER NOT NULL;

CREATE DOMAIN pl AS VARCHAR(6)
    CHARACTER SET ISO8859_1 COLLATE DE_DE;
```

```
CREATE DOMAIN tl AS VARCHAR(15)
    CHARACTER SET ISO8859_1 COLLATE DE_DE;

CREATE DOMAIN st AS VARCHAR(25)
    CHARACTER SET ISO8859_1 COLLATE DE_DE;

CREATE TABLE t_kunde
    (nummer ln,
    vorname st,
    nachname st,
    strasse st,
    plz pl,
    ort st,
    tel tl,
    fax tl,
    PRIMARY KEY(nummer));
```

An diesem Beispiel sehen Sie sehr schön, dass Sie sich mit Domänen erheblich Schreibarbeit ersparen können, wenn Sie einen Zeichensatz und eine Sortierreihenfolge verwenden.

Sie können auch bei domänenbasierten Spalten die vorhin genannten Optionen verwenden.

```
CREATE TABLE t_test
    (nummer ln
        CHECK (nummer > 1000),
    bezeichnung st
        CHECK (UPPER(bezeichnung) NOT LIKE '%Ä%' ))
```

In diesem Fall werden die Optionen der Domäne und der Tabellendefinition miteinander kombiniert.

Berechnete Spalten

Sie können auch berechnete Spalten definieren. Deren Inhalt wird nicht in der Datenbank gespeichert, sondern bei der Abfrage jeweils berechnet. Für berechnete Spalten wird die COMPUTED BY-Klausel verwendet.

```
CREATE TABLE t_test
    (nummer INTEGER NOT NULL,
    stueckzahl INTEGER NOT NULL,
    preis FLOAT NOT NULL,
    gesamtpreis COMPUTED BY (stueckzahl * preis))
```

Hier im Beispiel wird der Gesamtpreis aus Preis mal Stückzahl berechnet.

Mit Hilfe von berechneten Spalten können auch Konstanten in die Tabellendefinition aufgenommen werden:

```
CREATE TABLE t_test
    (nummer INTEGER NOT NULL,
    wert COMPUTED BY ('Test'))
```

Auch die Verwendung von Unterabfragen ist möglich. Beachten Sie, dass diese Unterabfrage nicht zum Zeitpunkt des Einfügens eines Datensatzes ausgeführt wird, sondern dann, wenn Daten aus der Tabelle ausgelesen werden.

```
CREATE TABLE t_test
    (vorname st,
    nachname st,
    chef COMPUTED BY ((SELECT nachname
        FROM t_mitarbeiter
        WHERE vorgesetzter = nummer)))
```

Solche Konstanten und Unterabfragen kann man genauso gut in den SELECT-Anweisungen unterbringen. Sinnvoll ist so etwas nur, wenn man mit Hilfe von Tools (Reportgeneratoren beispielsweise) nur auf die komplette Tabelle zugreifen, jedoch keine SELECT-Statements formulieren kann.

Beachten Sie auch, dass berechnete Spalten das Einfügen von Daten verkomplizieren:

```
INSERT INTO t_test
    VALUES ('Uli', 'Busch')

/* geht nicht */
```

Diese Anweisung lässt sich nicht ausführen, weil bei der INSERT-Anweisung die Spaltenliste nur dann weggelassen werden kann, wenn in alle Spalten Werte eingefügt werden. Dies wäre bei der obigen Anweisung nicht der Fall, da ja in die Spalte *chef* gar kein Wert eingefügt werden kann. Die Anweisung wäre folgendermaßen abzuändern:

```
INSERT INTO t_test
        (vorname, nachname)
    VALUES
        ('Uli', 'Busch')
```

4.2.2 Schlüssel und Indizes

Viele Anwender, die von Desktop-Datenbanksystemen kommen, bringen *Index* und *Schlüssel* durcheinander: Ein Schlüssel ist eine Spalte oder eine Kombination von Spalten, welche jeden Wert beziehungsweise jede Wertekombination nur einmal erlaubt. Ein Index ist ein Suchbaum, welcher das Finden von Datensätzen beschleunigt.

Für jeden Schlüssel erstellt InterBase automatisch einen Index. Für Indizes werden aber keine Schlüssel erstellt.

Primärschlüssel

Der Primärschlüssel ist der Schlüssel, mit dessen Hilfe ein Datensatz für gewöhnlich identifiziert wird. In den meisten Fällen handelt es sich um eine durchlaufende Nummer, oft wird diese mit Hilfe eines Generators erzeugt.

Beachten Sie bitte, dass eine Tabelle jeweils nur einen Primärschlüssel haben darf. Alle anderen Schlüssel sind Sekundärschlüssel. Diese haben zwar exakt dieselbe Funktion, werden aber anders genannt.

Es gibt zwei Möglichkeiten, einen Primärschlüssel zu erstellen:

```
CREATE TABLE t_test
    (nummer INTEGER
        NOT NULL PRIMARY KEY,
    bezeichnung st)
```

Die Anweisung PRIMARY KEY kann der Spalte hinzugefügt werden, welche den Primärschlüssel bildet, in diesem Fall die Spalte *nummer*. Beachten Sie auch, dass Schlüsselspalten stets als NOT NULL zu definieren sind.

Die andere Möglichkeit ist die Aufnahme einer PRIMARY KEY-Klausel am Ende der Tabellendefinition:

```
CREATE TABLE t_gruppe
    (nummer ln,
    bezeichnung st,
    PRIMARY KEY(nummer));
```

Dieses Beispiel entstammt unserer Beispieldatenbank.

Die Aufnahme der PRIMARY KEY-Klausel am Ende der Tabellendefinition ist auch die einzige Möglichkeit, Primärschlüssel über mehrere Spalten zu erstellen:

```
CREATE TABLE t_test
    (abteilung INTEGER NOT NULL,
    mitarbeiter INTEGER NOT NULL,
    name st,
    PRIMARY KEY (abteilung, mitarbeiter))
```

Der Datensatz wird hier durch Abteilungs- und Mitarbeiternummer identifiziert. In diesem Fall wären gleiche Mitarbeiternummern erlaubt, wenn denn die Abteilung differiert. Bis auf die Ausnahme von Verknüpfungstabellen ist die Verwendung zusammengesetzter Primärschlüssel meist weniger zu empfehlen.

Sekundärschlüssel

Sekundärschlüssel dienen dazu, Redundanzen in der Datenbank zu vermeiden. Es können keine zwei Datensätze in einem Sekundärschlüssel denselben Wert haben.

Bei der Anwendung von Sekundärschlüsseln sollte man eine gewisse Vorsicht walten lassen: Nehmen wir einmal an, Sie erstellen auf Ihre Mitarbeitertabelle einen Sekundärschlüssel über die Spalten *vorname* und *nachname*. Nun stellt die Firma eine Person namens *Konrad Müller* ein. Einen Mitarbeiter solchen Namens gibt es aber bereits, also wird sich die Datenbank weigern, eine entsprechende Eingabe anzunehmen. (Eine Möglichkeit wäre hier, den Mitarbeiter in *Konrad Mueller* umzubenennen oder dem Vor- und/oder Nachnamen ein Leerzeichen anzuhängen. Dann wäre der Datensatz zumindest in der Datenbank. Erfahrungsgemäß ziehen solche Mogeleien aber irgendwann »Ärger« nach sich.)

Sekundärschlüssel werden mit dem Schlüsselwort UNIQUE erzeugt. Wird der Sekundärschlüssel über eine einzige Spalte gebildet, dann kann die UNIQUE-Anweisung der Spaltendefinition angehängt werden:

```
CREATE TABLE t_test
    (nummer ln PRIMARY KEY,
    test INTEGER NOT NULL UNIQUE)
```

Auch die an Sekundärschlüsseln beteiligten Spalten müssen als NOT NULL definiert werden.

Ein Sekundärschlüssel kann auch am Ende der Tabellendefinition erzeugt werden – dies ist sogar zwingend erforderlich, wenn der Sekundärschlüssel mehrere Spalten umfasst:

```
CREATE TABLE t_kunde2
    (nummer ln,
    vorname st NOT NULL,
    nachname st NOT NULL,
    strasse st NOT NULL,
    plz pl NOT NULL,
    ort st,
    tel tl,
    fax tl,
    PRIMARY KEY(nummer),
    UNIQUE (vorname, nachname, strasse, plz));
```

In diesem Beispiel wird die Annahme einer neuen Kundenadresse vermieden, wenn es schon einen Datensatz gibt, der in Vorname, Nachname, Straße und Postleitzahl gleicht.

Fremdschlüssel

Bei normalisierten Datenbanken wird der Datenbestand auf mehrere Tabellen aufgeteilt, welche bei der Abfrage mittels eines JOINS wieder zusammengefügt werden. Nehmen wir einmal an, in *t_tele* wären Telefonnummern und in *t_art* die Arten dieser Telefonnummern (privat, dienstlich, Fax …) gespeichert.

```
SELECT
        t.bezeichnung,
        a.bezeichnung
    FROM t_tele t
        INNER JOIN t_art a
            ON t.art = a.nummer
```

Nun soll sichergestellt werden, dass in die Tabelle *t_tele* nur solche Datensätze aufgenommen werden, welche in der Spalte *art* einen Wert enthalten, der in der Tabelle *t_art* eine Entsprechung findet. Prinzipiell könnte man dafür eine entsprechende CHECK-Klausel formulieren:

```
CREATE TABLE t_tele2
    (nummer ln,
    mitarbeiter ln,
    art ln
        CHECK (art IN (SELECT nummer
            FROM t_art)),
    bezeichnung st,
    PRIMARY KEY(nummer));
```

Diese Vorgehensweise hat allerdings einen entscheidenden Nachteil: Angenommen, wir fügen einen Datensatz ein, der in der Spalte *art* den Wert sieben (*Telefon bei Mutti*) enthält. Nun hindert uns aber das DBS nicht daran, anschließend den Datensatz sieben aus der Tabelle *t_art* zu löschen. Damit hätten wir allerdings eine Referenz auf einen nicht existierenden Datensatz, die – wie der Fachmann sagt – *referenzielle Integrität* wäre verletzt.

Nun könnte man einen TRIGGER erstellen, der vor dem Löschen eines Datensatzes prüft, ob Referenzen darauf existieren.

Es geht aber auch einfacher: Wenn wir einen Fremdschlüssel erstellen, dann passt das DMS von selbst darauf auf, dass die referenzielle Integrität erhalten bleibt. Solange nichts anderes vorgegeben wird, verweigert das DMS die Ausführung entsprechender INSERT-, DELETE- und UPDATE-Anweisungen.

Es gibt wieder die Möglichkeit, den Fremdschlüssel in der jeweiligen Spaltendefinition oder am Ende der Tabellendefinition zu erstellen.

```
CREATE TABLE t_tele2
    (nummer ln,
    mitarbeiter ln,
```

```
art ln REFERENCES t_art (nummer),
bezeichnung st,
PRIMARY KEY(nummer))
```

Nach dem Schlüsselwort REFERENCES wird die Tabelle genannt, auf welche die Referenz gebildet wird. In Klammern wird dann die Spalte genannt, die referenziert werden soll.

Beachten Sie, dass beide Spalten dieselben Typen haben müssen. Es ist beispielsweise nicht möglich, von einer INTEGER-Spalte aus eine VARCHAR-Spalte zu referenzieren.

Sie können den Fremdschlüssel auch am Ende der Tabellendefinition erzeugen. Dies ist zwingend erforderlich, wenn Sie Referenzen über mehrere Spalten erzeugen wollen.

Noch zwei Anmerkungen zu Fremdschlüsseln: Sie können nur Referenzen auf Tabellen erstellen, die bereits existieren. Wenn Sie sich keine Gedanken darüber machen wollen, in welcher Reihenfolge Sie die Tabellen erstellen müssen, dann erstellen Sie zunächst sämtliche Tabellen ohne Fremdschlüssel und fügen diese dann anschließend mit ALTER TABLE ein. Wir werden das noch detailliert besprechen.

Vermeiden Sie zirkuläre Referenzen: Wenn die Tabelle A die Tabelle B referenziert, sollte nicht auch die Tabelle B die Tabelle A referenzieren, sonst könnte es Schwierigkeiten geben, zumindest beim Einfügen des ersten Datensatzes.

Verhalten bei Fremdschlüsselverletzungen

Versuchen Sie, einen Wert in die Datenbank einzugeben, der in der referenzierten Tabelle keine Entsprechung findet, dann wird das DBS die Annahme verweigern.

Vielfältiger sind die Möglichkeiten, wenn Sie einen Datensatz aus einer Tabelle löschen wollen, obwohl noch Referenzen darauf bestehen, oder wenn Sie einem solchen Datensatz in der referenzierten Spalte einen anderen Wert zuweisen wollen. Im Normalfall wird auch hier die Ausführung der Anweisung verweigert. Es lassen sich aber auch andere Verhaltensweisen vorgeben:

- Wird NO ACTION vorgegeben, dann wird – wie gehabt – die Ausführung der Anweisung unterbunden.
- Wird bei CASCADE ein Datensatz gelöscht, dann werden alle Datensätze in anderen Tabellen, die den gelöschten Datensatz referenziert haben, ebenfalls gelöscht.

  Wird bei CASCADE ein Datensatz im referenzierten Feld geändert, dann werden alle referenzierenden Datensätze entsprechend angepasst.
- Bei SET DEFAULT werden alle referenzierenden Spalten auf den DEFAULT-Wert der Spaltendefinition gesetzt.

- Dementsprechend werden bei SET NULL alle referenzierenden Spalten auf den Wert NULL gesetzt.

Diese vier Möglichkeiten lassen sich getrennt für ON UPDATE und ON DELETE einstellen.

```
CREATE TABLE t_tele2
    (nummer ln,
    mitarbeiter ln,
    art ln
        DEFAULT 1,
    bezeichnung st,
    PRIMARY KEY(nummer),
    FOREIGN KEY (art)
        REFERENCES t_art (nummer)
        ON DELETE SET DEFAULT
        ON UPDATE CASCADE);

INSERT INTO t_tele2  SELECT * FROM t_tele
```

Sekundär- und Fremdschlüssel benennen

Nehmen wir einmal an, Sie erstellen einen Sekundär- oder Fremdschlüssel:

```
CREATE TABLE t_test4
(id INTEGER NOT NULL PRIMARY KEY,
zahl INTEGER NOT NULL UNIQUE)
```

Nun wollen Sie den Sekundärschlüssel wieder loswerden – und dafür benötigen Sie den Namen. Gut, das ist zwar auch nicht weiter tragisch, wenn man weiß, in welcher Systemtabelle man nachschauen muss:

```
SELECT *
       FROM rdb$relation_constraints
       WHERE rdb$relation_name = UPPER('t_test4')
```

	RDB$CONSTRAINT_NAME	RDB$CONSTRAINT_TYPE	RDB$RELATION_NAME
▶	INTEG_23	NOT NULL	T_TEST4
	INTEG_24	PRIMARY KEY	T_TEST4
	INTEG_25	NOT NULL	T_TEST4
	INTEG_26	UNIQUE	T_TEST4

Abbildung 4.3: Der Name des Sekundärschlüssels

In diesem Fall wäre der Name des Sekundärschlüssels *INTEG_26* und mit diesem Namen könnte man ihn wieder entfernen:

```
ALTER TABLE t_test4
      DROP CONSTRAINT INTEG_26
```

Nun kann man sich das Suchen in der Systemtabelle auch sparen, indem man dem Sekundärschlüssel gleich einen Namen gibt.

```
CREATE TABLE t_test4
(id INTEGER NOT NULL PRIMARY KEY,
zahl INTEGER NOT NULL,
CONSTRAINT u_zahl UNIQUE(zahl))
```

Bei Fremdschlüsseln würde man analog dazu vorgehen:

```
CONSTRAINT f_zahl FOREIGN KEY (zahl) REFERENCES t_test4(id))
```

Indizes

Bei nicht-indizierten Spalten arbeitet das DBS mit einer sequenziellen Suche: Werden beispielsweise die Datensätze gesucht, deren Feld *nachname* den Wert *Müller* hat, dann liest das DBS einen Datensatz nach dem anderen aus, prüft den Inhalt des Feldes *nachname* und entscheidet dann, ob der Datensatz in die Ergebnismenge aufgenommen wird oder nicht. Wie Sie sich leicht vorstellen können, ist der Rechner damit einige Zeit beschäftigt.

Ist die Spalte *nachname* dagegen indiziert, dann liest das DBS aus dem Index die Adressen der betreffenden Datensätze aus und sammelt dann nur noch die betreffenden Datensätze ein.

Wie der Index genau aufgebaut ist, soll uns an dieser Stelle nicht interessieren (über die Vor- und Nachteile der einzelnen Verfahren könnte man ganze Kapitel schreiben). Wir wollen uns aber merken, dass man die Ausführung von SQL-Anweisungen (in der Regel SELECT-Anweisungen) mit Indizes in manchen Fällen erheblich beschleunigen kann.

```
SELECT *
    FROM t_adressen
    WHERE (nachname STARTING WITH 'B')
    ORDER BY nachname
```

Die Ausführung dieser Anweisung dauerte (von einer Delphi-Anwendung aus gemessen, damit eine genaue Zeitmessung durchgeführt werden kann) ohne Index auf dem Feld *nachname* 2911 ms, mit Index dagegen nur 125 ms – was natürlich rechnerabhängig ist, aber Ihnen zumindest ein Gefühl für die Größenordnung der Beschleunigung gibt.

Den Index würde man mit CREATE INDEX erstellen:

```
CREATE INDEX ix_adressen_nachname ON t_adressen(nachname);
```

Nach der Anweisung CREATE INDEX müssen Sie dem Index zunächst einen Namen zuweisen. Dieser wird dazu benötigt, den Index später wieder löschen zu können oder ihn von der Datenzugriffskomponente aus auszuwählen. Es stört also nicht groß, wenn der Indexbezeichner ein wenig länger ausfällt, dafür sollte klar

erkennbar sein, was er indiziert. Nach dem Schlüsselwort ON geben Sie zunächst die betreffende Tabelle ein und in Klammern dann die Spalte(n), über die der Index erstellt wird.

Ein Index arbeitet jedoch nur in einer Richtung. Würden wir nun unsere Datenmenge in umgekehrter Reihenfolge sortieren, dann würde *ix_adressen_nachname* den Vorgang nicht beschleunigen:

```
SELECT *
    FROM t_adressen
    WHERE (nachname STARTING WITH "B")
    ORDER BY nachname DESC
```

Man könnte zu diesem Zweck jedoch einen »rückwärts« aufgebauten Index erstellen:

```
CREATE DESC INDEX ix_adressen_nachname ON t_adressen(nachname);
```

Beachten Sie, dass das Schlüsselwort DESC zwischen CREATE und INDEX steht.

```
CREATE INDEX i_kunde_test
    ON t_kunde (nachname, vorname)
```

Sie können auch Indizes erstellen, die über mehrere Spalten reichen. Das macht aber nur dann Sinn, wenn Sie auch stets über diese Spalten sortieren. Andernfalls indizieren Sie die Spalten lieber einzeln. Sie vermeiden dann auch, dass der Index zu groß wird und InterBase sich weigert, ihn zu erstellen.

```
DROP INDEX i_kunde_nachname
```

Um einen Index zu löschen, verwenden Sie die Anweisung DROP INDEX. Sie können einen Index auch deaktivieren:

```
ALTER INDEX i_kunde_nachname_desc INACTIVE
```

Das Löschen oder Deaktivieren von Indizes kann zwei Gründe haben: Zunächst einmal muss bei jedem eingefügten, geänderten oder gelöschten Datensatz der Index aktualisiert werden. Nach meiner Beobachtung nimmt dieser Vorgang etwa 20 ms pro 1000 Datensätze in Anspruch (auf meinem Rechner), kann also bei einem einzelnen Datensatz vollkommen vernachlässigt werden.

Sollen jedoch sehr viele Datensätze eingefügt werden, dann addieren sich diese Zeiten. Dann kann es sogar sinnvoll sein, den Datensatz vorher zu löschen oder zu deaktivieren und anschließend wieder neu aufzubauen.

Außerdem »verschleißt« ein Index im Laufe der Zeit: Mit jeder Änderung in der Tabelle wird der Suchbaum ein wenig unsymmetrischer, dementsprechend lässt dann auch die Wirksamkeit nach. Es ist deshalb sinnvoll, von Zeit zu Zeit den Index abzubauen und wieder neu zu erstellen.

```
ALTER INDEX i_kunde_nachname_desc ACTIVE
```

Ein Index wird auch dann wieder komplett neu aufgebaut, wenn er vom deaktivierten in den aktivierten Zustand gesetzt wird. Es macht übrigens keinen Unterschied, ob Sie DROP INDEX/CREATE INDEX oder ALTER INDEX verwenden. Bei ALTER INDEX merkt sich InterBase lediglich die Tabelle und die beteiligten Spalten.

Wenn Sie ein Backup und ein Restore ausführen, dann werden alle Indizes neu erstellt, zudem werden die Daten wieder gleichmäßig auf die Datenbankseiten verteilt.

Wann sollten Indizes erstellt werden?

Es ist nun nicht sinnvoll, für »alles und jedes« Indizes zu erstellen.

- Spalten, die an Primär-, Sekundär- und Fremdschlüsseln beteiligt sind, werden von InterBase automatisch indiziert.
- Bei kurzen Tabellen (unter 30 Datensätze) ist eine Indizierung völlig überflüssig.
- Bei Spalten, die nur wenige unterschiedliche Werte speichern (nur *Herr* und *Frau* beispielsweise), kann eine Indizierung sogar kontraproduktiv sein.
- Die Indizierung selten verwendeter Spalten bringt wenig bis nichts.
- Sinnvoll ist eine Indizierung vor allem da, wo große Datenmengen gefiltert, sortiert oder gruppiert werden.
- Wenn Sie eine Spalte mal aufsteigend, mal absteigend sortieren, dann kann es sinnvoll sein, diese Spalte sowohl aufsteigend als auch absteigend zu indizieren.

4.2.3 Gültigkeitsprüfungen

Im Gegensatz zu Gültigkeitsprüfungen auf Domänen-Ebene lassen sich bei der Tabellendefinition auch Gültigkeitsprüfungen implementieren, die Spaltenwerte untereinander vergleichen.

```
CREATE TABLE t_test
    (nummer ln,
    minpreis FLOAT NOT NULL,
    maxpreis FLOAT NOT NULL
        CHECK (minpreis < maxpreis),
    PRIMARY KEY (nummer))
```

In diesem Beispiel würde dafür gesorgt, dass der Wert von *maxpreis* stets über dem von *minpreis* liegt.

```
INSERT INTO t_test VALUES (1, 3, 2)
```

Mit dieser INSERT-Anweisung verstoßen Sie gegen die Check-Klausel und erhalten die Meldung *Operation violates CHECK constraint INTEG_83 on view or table*

T_TEST. Nun weiß vielleicht nicht jeder Anwender, was denn nun *CHECK constraint INTEG_83* ist. Deshalb gibt es auch die Möglichkeit, benannte Gültigkeitsprüfungen zu erstellen.

```
CREATE TABLE t_test
    (nummer ln,
    minpreis FLOAT NOT NULL,
    maxpreis FLOAT NOT NULL,
    PRIMARY KEY (nummer),
    CONSTRAINT minpreis_kleiner_maxpreis
        CHECK (minpreis < maxpreis))
```

In diesem Fall würde die Fehlermeldung *Operation violates CHECK constraint MINPREIS_KLEINER_MAXPREIS on view or table T_TEST* lauten. Solche benannten Gültigkeitsprüfungen bieten darüber hinaus den Vorteil, dass man nicht erst umständlich ihren Namen suchen muss, wenn man sich von ihnen trennen möchte.

4.2.4 Tabellen ändern

Mit der Anweisung ALTER TABLE können Sie Tabellen ändern. Sie können dabei:

- Spalten hinzufügen oder löschen
- Spalten umbenennen, ihren Typ oder ihre Position ändern
- Gültigkeitsprüfungen hinzufügen oder löschen
- Primär-, Sekundär- oder Fremdschlüssel hinzufügen oder löschen

Spalten hinzufügen oder löschen

Sie können mit SQL-Befehlen nicht die Definition einzelner Spalten ändern. Sie können aber Spalten hinzufügen und Spalten löschen.

```
ALTER TABLE t_art
    ADD kurz VARCHAR(3)

ALTER TABLE t_art
    DROP kurz
```

Mit ADD wird eine Spalte hinzugefügt, die Spaltendefinition erfolgt in der gewohnten Art und Weise. Mit DROP wird eine Spalte gelöscht.

Wenn Sie sich die Tabelle mit der hinzugefügten Spalte anzeigen lassen, dann sehen Sie, dass InterBase diese Spalte mit NULL-Werten gefüllt hat. Was passiert aber, wenn wir die Spalte als NOT NULL definieren?

```
ALTER TABLE t_art
    ADD kurz VARCHAR(3) NOT NULL
```

Hier reagiert InterBase »pfiffig«: Anstatt die Ausführung dieser Anweisung zu verweigern, wird die Spalte nun mit leeren Strings aufgefüllt. Wenn uns leere Strings nicht passen, dann muss eben ein DEFAULT-Wert angegeben werden:

```
ALTER TABLE t_art
    ADD kurz VARCHAR(3)
    DEFAULT 'x'
    NOT NULL
```

Spalten ändern

Sie können eine Spalte umbenennen, ihren Typ und ihre Position ändern. Nehmen wir als Beispiel die folgende Tabelle:

```
CREATE TABLE t_test6
(id INTEGER NOT NULL PRIMARY KEY,
eins VARCHAR(20),
zwei VARCHAR(20))
```

Um die Spalte *eins* zu verbreitern, verwenden Sie die folgende Anweisung:

```
ALTER TABLE t_test6
ALTER eins TYPE VARCHAR(30)
```

Wie schon bei den Domänen erläutert: Die Zuweisung eines anderen Typen funktioniert, solange InterBase keinen Datenverlust befürchtet. Das Verbreitern von Spalten geht somit immer, das Umwandeln nach CHAR und VARCHAR auch. (Der *Data Definition Guide* bestreitet zwar die Möglichkeit, die Zeit- und Datumswerte nach VARCHAR umzuwandeln, der Server selbst sieht das jedoch nicht so eng …).

Um eine Spalte an eine andere Position zu schieben, verwenden Sie POSITION.

```
ALTER TABLE t_test6
ALTER zwei POSITION 1
```

Beachten Sie dabei, dass die erste Spalte den Index null hat, die Spalte *zwei* würde somit an die zweite Position geschoben. Das Umbenennen von Spalten erfolgt dann mit TO:

```
ALTER TABLE t_test6
ALTER zwei TO bemerkung
```

Gültigkeitsprüfungen hinzufügen und löschen

Nehmen wir einmal an, unsere Firma möchte die Telefonkosten senken und in diesem Zuge vermeiden, dass Handy-Nummern in die Kundentabelle gelangen. Mit ADD CONSTRAINT kann man eine entsprechende Gültigkeitsprüfung einrichten.

```
ALTER TABLE t_kunde
    ADD CONSTRAINT kein_handy
        CHECK (tel NOT STARTING WITH '01')
```

Gültigkeitsprüfungen wirken nur für neu einzufügende oder zu ändernde Datensätze. Wenn in der Tabelle Datensätze enthalten sind, denen die Kriterien der Gültigkeitsprüfung nicht genügen, dann erhalten Sie noch nicht einmal eine entsprechende Warnung.

Es wäre nicht erforderlich, eine Gültigkeitsprüfung mit CONSTRAINT zu benennen, es würde auch folgendermaßen gehen:

```
ALTER TABLE t_kunde
    ADD CHECK (tel NOT STARTING WITH '01')
```

Wenn Sie die Gültigkeitsprüfung wieder loswerden wollen, dann tun Sie sich mit einem Namen natürlich leichter:

```
ALTER TABLE t_kunde
    DROP CONSTRAINT kein_handy
```

Schlüssel einfügen und löschen

Sie können auch Primär-, Sekundär- und Fremdschlüssel einfügen und löschen.

In der Praxis wird davon vor allem bei den Fremdschlüsseln Gebrauch gemacht. Ein Fremdschlüssel lässt sich nur erstellen, wenn die referenzierte Tabelle bereits existiert. Würde man die Fremdschlüssel bereits bei der Tabellendefinition erstellen, dann müsste man sich intensiv Gedanken über die Reihenfolge machen, in der die Tabellen erstellt werden.

Einfacher ist es deshalb, zunächst alle Tabellen zu erstellen und erst anschließend die Fremdschlüssel einzufügen:

```
ALTER TABLE t_bestellung
    ADD FOREIGN KEY (kunde)
        REFERENCES t_kunde (id);
```

Diese Anweisung entstammt dem SQL-Script zur Erstellung unserer Beispieldatenbank. In diesem Script sind mehrere solcher Anweisungen enthalten.

Im Gegensatz zu Gültigkeitsprüfungen wird das Hinzufügen von Schlüsseln verweigert, wenn die vorhandenen Datensätze die Schlüsselbedingung nicht erfüllen. Dies ist der Fall, wenn sich bei Primär- und Sekundärschlüssel bereits doppelte Werte in der Spalte befinden. Bei Fremdschlüsseln ist das der Fall, wenn Referenzen auf nicht vorhandene Datensätze gebildet würden.

Das Löschen von Schlüsseln haben wir bereits im Abschnitt *Sekundär- und Fremdschlüssel benennen* thematisiert.

4.2.5 Tabelle löschen

Zum Löschen einer Tabelle wird die Anweisung DROP TABLE verwendet.

```
DROP TABLE t_test
```

Beim Löschen einer Tabelle sind einige Besonderheiten zu beachten:

- Gelöscht werden kann eine Tabelle nur dann, wenn keine Transaktion mehr läuft, welche die Tabelle verwendet. Wenn Sie sich beispielsweise mit SELECT die Tabelle angesehen oder mit INSERT Daten eingefügt haben, dann müssen Sie zunächst die Transaktion mit TRANSACTIONS|COMMIT oder TRANSACTIONS|ROLLBACK beenden.
- Wenn die Tabelle Ziel einer Referenz ist, in einer VIEW, STORED PROCEDURE oder einem TRIGGER verwendet wird, kurz gesagt, wenn andere Metadaten die Tabelle verwenden, kann diese ebenfalls nicht gelöscht werden.

  Dies hat zur Folge, dass meist die halbe Datenbank »abgebaut« werden muss, bevor man sich von einer Tabelle trennen kann. Hier ist es dann meist einfacher, die Datenbank (nach entsprechender Änderung des SQL-Scripts) komplett neu zu erstellen und die Daten anschließend zu kopieren.
- Wenn der angemeldete Benutzer weder Systemadministrator noch Ersteller der Tabelle ist, kann er sie ebenfalls nicht löschen.

4.3 Ansichten

Eine Ansicht (VIEW) ist eine vordefinierte Abfrage, auf die wie auf eine Tabelle zugegriffen wird.

Nehmen wir einmal an, die Firma bräuchte eine Liste aller Kunden für Direktmarketing per Telefon. Dafür könnte man die folgende Ansicht erstellen:

```
CREATE VIEW v_telefonliste AS
  SELECT vorname, nachname, tel
  FROM t_adressen
  WHERE tel IS NOT NULL
```

Um eine Ansicht zu erstellen, wird die Anweisung CREATE VIEW verwendet. Dieser Anweisung folgt der Name der VIEW. Das Präfix *v_* soll der Unterscheidung von anderen Bezeichnern dienen. Nach dem Schlüsselwort AS wird die Ansicht mit einer SELECT-Anweisung definiert.

Um sich die Daten in einer VIEW anzusehen, wird eine SELECT-Anweisung ausgeführt:

```
SELECT * FROM v_ telefonliste
```

Beim Erstellen einer Ansicht kann in der SELECT-Anweisung die WHERE-Klausel verwendet werden, außerdem ist die Erstellung von JOINS möglich. Nicht gestat-

tet dagegen sind Funktionen und somit die GROUP BY- und die HAVING-Klausel, darüber hinaus sind die ORDER- und die UNION-Klausel nicht gestattet. Brauchen Sie diese Optionen, dann müssen Sie eine STORED PROCEDURE verwenden.

Beim Zugriff auf eine VIEW können Sie dann Optionen wie die ORDER-Klausel verwenden:

```
SELECT *
    FROM v_telefonliste
    ORDER BY nachname
```

Im Zusammenhang mit einer Ansicht sind zwei Begriffe gebräuchlich:

- Von einer vertikalen Teilmenge (*vertical subset*) spricht man, wenn Spalten von der Anzeige ausgeschlossen werden.
- Eine horizontale Teilmenge (*horizontal subset*) schließt dagegen einige Reihen von der Anzeige aus.

Selbstverständlich können Sie auch einige Spalten und einige Reihen gleichzeitig von der Anzeige ausschließen.

Benennen von Spalten

Nach dem Bezeichner der Ansicht und dem Schlüsselwort AS kann eine Liste der Spaltenbezeichner erstellt werden. Bisweilen zwingt InterBase zur Erstellung einer solchen Spaltenliste, weil Spalten ansonsten keine oder keine eindeutigen Namen hätten, was im folgenden Beispiel bei den zusammengefügten Spalten der Fall ist.

```
CREATE VIEW v_bestellung (bestellnummer, datum, kunde, bearbeiter) AS
  SELECT
    b.id,
    b.datum,
    k.vorname || ' ' || k.nachname,
    m.vorname || ' ' || m.nachname
  FROM t_bestellung b
    INNER JOIN t_adressen k
      ON k.id = b.kunde
    INNER JOIN t_mitarbeiter m
      ON m.id = b.mitarbeiter
```

4.3.1 Daten in einer Ansicht ändern

Man unterscheidet zwischen aktualisierbaren und nicht aktualisierbaren Ansichten. Prinzipiell können in alle Ansichten neue Datensätze eingefügt oder bestehende Datensätze geändert und gelöscht werden, wenn InterBase erkennen kann, was genau zu tun ist, und die dafür nötigen Datenbankoperationen prinzipiell erlaubt sind.

Umgekehrt ist eine solche Aktualisierung nicht möglich, wenn einer oder mehrere der folgenden Gründe vorliegen:

- Ansichten, die auf mehreren Tabellen basieren, sind nicht aktualisierbar, es sei denn, es werden entsprechende Trigger für die einzelnen Update-Fälle erstellt.
- Abfragen mit Aggregatfunktionen, User Defined Functions, Unterabfragen, einer DISTINCT- oder einer HAVING-Klausel können nicht aktualisiert werden.
- Alle Spalten, die nicht in die Ansicht aufgenommen werden, erlauben den Wert NULL oder haben einen DEFAULT-Wert.

CHECK OPTION

Mit Hilfe der CHECK OPTION kann man dafür sorgen, dass nur diejenigen Datensätze in einer aktualisierbaren Ansicht geändert werden können, die in derselben auch angezeigt werden.

```
CREATE TABLE t_test8
(id INTEGER  NOT NULL PRIMARY KEY);

INSERT INTO t_test8 VALUES (1);
INSERT INTO t_test8 VALUES (2);
INSERT INTO t_test8 VALUES (3);
INSERT INTO t_test8 VALUES (4);
INSERT INTO t_test8 VALUES (5);

CREATE VIEW v_test8 AS
  SELECT * FROM t_test8
    WHERE id < 4
  WITH CHECK OPTION;
```

Würde man nun versuchen, einen Datensatz zu löschen oder zu ändern, der nicht angezeigt wird, so würde dieses Statement kommentarlos ignoriert:

```
UPDATE v_test8
  SET id = 8
  WHERE id = 4;

DELETE FROM v_test8
  WHERE id = 4;
```

Versucht man dagegen, einen Datensatz einzufügen, der anschließend nicht angezeigt würde, führt dies zu der Fehlermeldung *Operation violates CHECK constraint on view or table*.

```
INSERT INTO v_test8 VALUES(6)
```

Dasselbe Ergebnis erhielte man, wenn man versuchte, einen sichtbaren Datensatz so zu ändern, dass er anschließend nicht mehr sichtbar wäre:

```
UPDATE v_test8
  SET id = 8
  WHERE id = 2;
```

Zugriffsberechtigung

Das Thema *Zugriffsberechtigung* werden wir erst in Kapitel 4.4 behandeln. Es sei aber im Moment schon so viel verraten, dass Sie nicht nur einzelnen Anwendern die Verwendung der VIEW gestatten können, Sie können auch bestimmen, ob diese lesen, einfügen, ändern und/oder löschen dürfen.

In vielen Fällen wird es zu empfehlen sein, nur die SELECT-Rechte zu vergeben, damit die Anwender nicht – beabsichtigt oder unbeabsichtigt – die darunter liegenden Daten ändern.

4.3.2 Eine Ansicht löschen

Um eine Ansicht zu löschen, verwendet man die Anweisung DROP VIEW.

```
DROP VIEW v_test
```

Um eine Ansicht löschen zu können, müssen erst alle tangierenden Transaktionen abgeschlossen werden. Außerdem dürfen keine weiteren Ansichten auf der betreffenden VIEW beruhen (was möglich ist, von mir aber nicht unbedingt empfohlen wird).

Die Änderung einer Ansicht ist nicht möglich, der Befehl ALTER VIEW existiert nicht. (Es wäre mir ohnehin unklar, was er sinnvollerweise bewirken könnte – vielleicht die Änderung der CHECK OPTION.)

Zusammenfassend möchte ich noch feststellen, dass die VIEW recht vielen Beschränkungen unterworfen ist. Wesentlich flexibler und leistungsfähiger ist die STORED PROCEDURE, die wir in Kapitel 5 besprechen werden.

4.4 Zugriffsrechte

In Datenbanken werden häufig sensible Daten gespeichert. Diese sind nicht nur vor Verlust zu schützen – beispielsweise durch ein regelmäßig durchgeführtes Backup –, es ist auch sicherzustellen, dass nur diejenigen Anwender die Daten lesen können, die dazu auch berechtigt sind.

Solche Zugriffsrechte könnten dezidiert per SQL-Statement an jeden einzelnen Benutzer vergeben werden. Dafür wird jeder einzelne Benutzer beim Datenbankserver angemeldet und erhält dann einen Benutzernamen und ein Passwort. Mit

diesen Zugangsdaten meldet er sich nun über seine Datenbankanwendung bei der Datenbank an. Versucht die Datenbankanwendung nun, auf ein Element (Tabelle, Ansicht, STORED PROCEDURE) zuzugreifen, für das der angemeldete Benutzer keine Rechte hat, so wird dieser Versuch mit einer Fehlermeldung quittiert.

Ich möchte nicht verhehlen, dass ich kein Freund dieser Art von Zugriffssteuerung bin: Ich implementiere diese Zugriffsverwaltung lieber selbst und habe dann die Möglichkeit, nicht zugelassene Operationen erst gar nicht anzubieten. Außerdem erfährt der Anwender dann nie das Passwort, mit dem er an der Datenbankanwendung vorbei auf die Datenbank kommen würde.

4.4.1 GRANT

Zugriffsrechte werden mit GRANT an Benutzer und Rollen vergeben.

Benutzerrechte einräumen

Die folgende Anweisung wird dem Benutzer *emil* das Leserecht auf die Tabelle *t_artikel* einräumen.

```
GRANT SELECT ON t_artikel TO emil
```

Benutzerrechte werden mit der Anweisung GRANT zugewiesen. Mit dem Schlüsselwort SELECT zeigen wir an, dass Leserechte vergeben werden sollen. Nach ON wird die Tabelle angegeben, für die das Leserecht eingeräumt wird, nach TO der Name des Benutzers, der das Recht erhalten soll.

```
GRANT INSERT, UPDATE, DELETE
    ON t_artikel
    TO emil, susi
```

Es ist auch möglich, mehrere Rechte gleichzeitig zu vergeben oder Rechte mit einer Anweisung mehreren Benutzern gleichzeitig zu gewähren. In diesem Beispiel dürfen die Benutzer *emil* und *susi* in der Tabelle *t_artikel* Datensätze einfügen, ändern und löschen. Es ist nicht möglich, mit einer Anweisung Rechte für mehrere Tabellen einzuräumen.

Das Referenzrecht

Mit dem Schlüsselwort REFERENCES wird die Referenzierung von Tabellen erlaubt. Nehmen wir an, Tabelle A erstellt einen Fremdschlüssel auf Tabelle B. Wird nun in Tabelle A ein Datensatz eingefügt, dann muss geprüft werden, ob in Tabelle B ein entsprechender Eintrag vorhanden ist, dazu muss Tabelle B referenziert werden und dafür ist das Referenz-Recht (oder das Lese-Recht) erforderlich.

```
GRANT REFERENCES ON T_GRUPPE TO EMIL
```

ALL

In vielen Fällen wird man an Benutzer alle Tabellenrechte vergeben. Um die Tipparbeit zu minimieren, kann man mit dem Schlüsselwort ALL arbeiten.

```
GRANT SELECT, INSERT, UPDATE, DELETE
    ON t_test TO emil

GRANT ALL ON t_test TO emil
```

Diese beiden Anweisungen führen zu demselben Ergebnis.

PUBLIC

Wenn viele Benutzer beim System registriert sind, wäre es recht aufwändig, ihnen die Zugriffsrechte einzeln zu gewähren. Mit dem Schlüsselwort PUBLIC kann deshalb das Zugriffsrecht allen Benutzern gewährt werden.

```
GRANT SELECT ON t_test TO PUBLIC
```

Auch diejenigen Benutzer, die erst nach der Ausführung dieser Anweisung beim System registriert werden, erhalten auf diese Weise die betreffenden Rechte.

GRANT OPTION

Im Regelfall kann nur der Besitzer einer Tabelle oder Ansicht sowie der SYSDBA Benutzerrechte vergeben. (In der Regel ist der SYSDBA auch der Besitzer aller Tabellen.)

```
GRANT SELECT ON t_test TO emil WITH GRANT OPTION
```

Mit dem Zusatz WITH GRANT OPTION erhalten die angegebenen Benutzer nicht nur die gewährten Rechte, sondern auch die Möglichkeit, diese an andere Benutzer weiterzugeben. Es können natürlich nur diejenigen Rechte weitergegeben werden, welche in der betreffenden GRANT-Anweisung erteilt wurden – bei diesem Beispiel also nur das SELECT-Recht.

Beschränkung auf bestimmte Spalten

Soll das Recht auf einen Tabellenzugriff auf bestimmte Spalten beschränkt werden, so muss dafür nicht in allen Fällen eine Ansicht definiert werden.

```
GRANT UPDATE (preis) ON t_artikel TO emil
```

Die UPDATE- und REFERENCES-Rechte können auf eine Teilmenge der vorhandenen Spalten beschränkt werden, die dann aufzuzählen sind. In diesem Beispiel wäre es *emil* möglich, die Preise des Sortiments anzupassen.

STORED PROCEDURES

STORED PROCEDURES wollen wir erst im nächsten Kapitel besprechen. Der Aspekt der Zugriffsrechte soll jedoch bereits hier erläutert werden:

```
GRANT INSERT, UPDATE ON t_adressen TO PROCEDURE p_beispiel
```

Wenn eine Prozedur (oder ein Trigger, für den alles hier Gesagte sinngemäß gilt) auf eine Tabelle zugreifen möchte, die einem anderen Benutzer gehört, dann braucht sie dafür die entsprechenden Rechte. Einer Prozedur werden die Rechte genauso wie einem Benutzer zugewiesen, dem Bezeichner wird jedoch das Wort PROCEDURE vorangestellt.

```
GRANT EXECUTE ON PROCEDURE p_beispiel TO emil, susi
```

Benutzer, die fremde Prozeduren ausführen möchten, benötigen dafür das EXECUTE-Recht.

4.4.2 REVOKE

Rechte können nicht nur vergeben, sondern mit REVOKE auch wieder zurückgenommen werden.

```
REVOKE SELECT ON t_art FROM emil
```

Die Syntax der Anweisung ähnelt dem GRANT-Befehl, es heißt allerdings statt GRANT..TO nun REVOKE..FROM.

```
REVOKE ALL ON t_gruppe FROM emil
```

Mit dem Schlüsselwort ALL können alle Rechte bezüglich einer Tabelle entzogen werden, auch wenn diese gar nicht alle erteilt wurden.

```
REVOKE DELETE ON t_test FROM PUBLIC
```

Mit dem Schlüsselwort PUBLIC können Rechte von allen Benutzern widerrufen werden, auch dann, wenn diese Rechte gar nicht allen Benutzern erteilt worden sind. Beachten Sie bitte auch, dass Rechte, die als PUBLIC vergeben wurden, auch nur FROM PUBLIC widerrufen werden können.

Bitte beachten Sie auch Folgendes:

- Wird einem Benutzer ein Recht entzogen, das er WITH GRANT OPTION erhalten hat, dann wird dieses Recht auch allen Benutzern entzogen, denen er es weitergewährt hat.
- Wird einem Benutzer ein Recht von mehreren verschiedenen Benutzern gewährt, dann müssen es alle diese Benutzer auch widerrufen, damit er es auch tatsächlich verliert.

4.4.3 Benutzergruppen

Seit Version 5.0 von InterBase lassen sich Benutzergruppen anlegen. Dadurch vermeidet man erstens viel Schreibarbeit, wenn viele Rechte an viele Benutzer zu vergeben sind, und erreicht darüber hinaus eine einheitliche Behandlung der Benutzergruppen.

```
CREATE ROLE benu
```

Eine Benutzergruppe wird mit CREATE ROLE erstellt. Außer dem Namen der Benutzergruppe kann nichts angegeben werden.

```
GRANT SELECT ON t_mitarbeiter TO benu
```

Mit GRANT kann ein Recht nicht nur einem einzelnen Benutzer, sondern auch einer Benutzergruppe gewährt werden.

```
GRANT benu TO emil, susi
```

Um einen oder mehrere Benutzer in die Benutzergruppe aufzunehmen, wird ebenfalls die GRANT-Anweisung verwendet.

```
DROP ROLE benu
```

Wenn Sie eine Benutzergruppe löschen, dann verlieren die darin aufgenommenen Benutzer automatisch die über die Benutzergruppe gewährten Rechte.

4.5 Generatoren

Mit einem Generator ist es möglich, solche selbst inkrementierenden Felder zu konstruieren. Dies ist insbesondere dann vorteilhaft, wenn man einen eindeutigen Primärschlüssel benötigt, beispielsweise eine Personalnummer, eine Kundennummer, eine Rechnungsnummer. Sie können in einer Datenbank mehrere Generatoren erstellen, welche allerdings unterschiedliche Namen haben müssen.

```
CREATE GENERATOR nummer_gen
SET GENERATOR nummer_gen TO 234
```

Mit der Anweisung CREATE GENERATOR wird ein neuer Generator erstellt. Dieser wird mit der Zahl 0 initialisiert. Wird eine andere Zahl benötigt, dann kann ein Generator jederzeit (also auch »im Betrieb«) mit SET GENERATOR auf einen anderen Wert gesetzt werden.

```
INSERT INTO test_1 (nummer, namen)
    VALUES (GEN_ID(nummer_gen, 1), "Eins")
```

Um einen Generatorwert einzufügen, wird in einer INSERT- oder UPDATE-Anweisung die Funktion GEN_ID verwendet, als Parameter werden der Name des Generators sowie der Wert, um den er erhöht werden soll, übergeben. Normalerweise wird der Generatorwert jeweils um eines erhöht, als Parameter wird dann 1 übergeben.

```
INSERT INTO test_1 (nummer, namen)
    VALUES (GEN_ID(nummer_gen, -3), "Zwei")
```

Wie dieses Beispiel zeigt, muss der Generator nicht zwangsweise immer um eins erhöht werden, es sind sogar negative Zahlen möglich. Wenn Sie einen Generator dafür verwenden, eindeutige Werte – beispielsweise für Primärschlüssel – zu ermitteln, dann sollten Sie darauf achten, dass der Generatorwert einheitlich positiv oder negativ verändert wird, sonst haben Sie irgendwann keine eindeutigen Werte mehr.

Sie können auch 0 als Parameter übergeben, wenn Sie den Generatorwert ermitteln wollen, ohne ihn zu verändern.

4.5.1 Mit TRIGGER und STORED PROCEDURE

Um selbst inkrementierende Felder zu erzeugen, wird in der Regel ein Trigger verwendet, vor jeder INSERT-Operation wird ein neuer Generatorwert erzeugt und in die betreffende Spalte geschrieben. Allerdings funktioniert diese Vorgehensweise nicht besonders gut mit den verschiedenen Datenmengenkomponenten von Delphi (beziehungsweise Kylix oder C++Builder), sei es, dass diese Komponenten selbst einen Generatorwert einfügen, sei es, dass man manuell einen Generatorwert einfügen muss, damit diese Komponenten vernünftig funktionieren.

Deshalb sollte man im TRIGGER auf IS NULL prüfen und nur dann einen Generatorwert einfügen, wenn die Spalte noch leer ist. Zur Abfrage des Generators erstellt man zweckmäßigerweise eine STORED PROCEUDRE. Viele Delphi-Tools, beispielsweise das in Kapitel 13 vorgestellte IBExperts, nehmen dem Programmierer die damit verbundene Schreibarbeit ab und erstellen sowohl den TRIGGER als auch die STORED PROCEDURE auf Wunsch automatisch.

```
SET TERM^;

CREATE TRIGGER "TRIG_ADRESSEN" FOR "T_ADRESSEN"
ACTIVE BEFORE INSERT POSITION 0
AS
BEGIN
  IF (NEW.ID IS NULL)
    THEN NEW.ID = GEN_ID(g_ADRESSEN, 1);
END^
```

```
CREATE PROCEDURE "P_ADRESSEN_ID"
RETURNS("GEN" INTEGER)
AS
BEGIN
  GEN = GEN_ID(g_ADRESSEN, 1);
  SUSPEND;
END^

SET TERM;^
```

5 TRIGGER und STORED PROCEDURES

SQL besteht aus recht wenigen sehr mächtigen Befehlen. Diese Befehle weisen den Computer nicht an, bestimmte Tätigkeiten durchzuführen, sondern ein bestimmtes Ergebnis hervorzubringen. Wie das intern geschieht, bleibt dem Datenbanksystem überlassen.

Nun gibt es Aufgaben, die mit diesem Befehlsatz nicht zu lösen sind. Man könnte die entsprechende Funktionalität nun in die Anwenderprogramme integrieren. Bei Client-Server-Systemen müssen jedoch die Daten in einem solchen Fall über das Netzwerk vom Server zu den Clients gespielt werden – bei großen Datenmengen ein recht zeitaufwändiges Verfahren.

Deshalb mussten Wege gefunden werden, zumindest einfachere Aufgaben auch auf dem Server ausführen zu können. Zu diesem Zweck stellt InterBase drei Features bereit:

- Über STORED PROCEDURES können Aufgaben mit einer Prozedurensprache gelöst werden. Man kann sich das wie eine Makro- oder eine Script-Sprache vorstellen.
- Es können TRIGGER definiert werden, die immer dann ausgelöst werden, wenn Datensätze eingefügt, gelöscht oder geändert werden. Diese TRIGGER sind im Prinzip auch Prozeduren und können somit auch komplexere Aufgaben bewältigen.
- Zusätzliche Funktionen können als USER DEFINED FUNCTIONS von DLLs in das Programm geladen werden, wir werden das im nächsten Kapitel besprechen.

TRIGGER und STORED PROCEDURES sind von der Programmierung her ziemlich ähnlich, auch wenn sie verschiedene Aufgaben erfüllen. Wir wollen sie deshalb hier gemeinsam besprechen.

5.1 STORED PROCEDURES

Eine STORED PROCEDURE ist ein Unterprogramm, das Parameter entgegennehmen und Ergebnisse zurückgeben kann. In anderen Programmiersprachen würde man so etwas eine Prozedur oder eine Funktion nennen.

5.1.1 Zwei Beispielprozeduren

Bevor wir uns mit den einzelnen Sprachelementen der Prozedursprache auseinander setzen, wollen wir zunächst zwei Beispielprozeduren erstellen.

Eine SELECT-Prozedur

Eine SELECT-Prozedur wird mit Hilfe einer SELECT-Anweisung aufgerufen. Für gewöhnlich liefert sie eine Datenmenge.

Nehmen wir an, wir wollten eine Liste der Orte erstellen, die wir in der Tabelle *t_adressen* haben. Dafür reicht eine einfache SELECT-Anweisung mit DISTINCT-Klausel:

```
SELECT DISTINCT ort
       FROM t_adressen
```

Nun wollen wir jedoch diese Liste durchnumeriert haben, so, wie dies in *Abbildung 5.1* zu sehen ist.

NUMMER	NAME
1	Altshausen
2	Bad Oldeslohe
3	Berlin
4	Bonn
5	Bremen
6	Colbitz
7	Dietzenbach
8	Freiberg

Abbildung 5.1: Liste der Orte mit durchlaufender Nummer

Dafür formulieren wir nun eine STORED PROCEDURE:

```
SET TERM ^;
CREATE PROCEDURE p_orte_test
RETURNS
  (nummer INTEGER,
  name VARCHAR(20))
AS
  DECLARE VARIABLE i INTEGER;
BEGIN
  i = 1;
  FOR SELECT
      DISTINCT ort
    FROM t_adressen
    INTO :name
    DO
```

```
    BEGIN
      nummer = i;
      i = i + 1;
      SUSPEND;
    END
END^
SET TERM ;^
```

Die Anweisung SET TERM wollen wir am Ende klären.

Eine STORED PROCEDURE wird mit der Anweisung CREATE PROCEDURE erstellt, auf die zunächst der Prozedurenname folgt. Die Präfix *p_* dient dazu, Prozeduren von anderen Elementen unterscheiden zu können, und ist freiwillig.

Im Anschluss an den Prozedurennamen folgt in Klammern die Parameterliste, also eine Reihe von Werten, die beim Aufruf der Prozedur an diese übergeben werden. Parameter werden zur Lösung der momentanen Problemstellung nicht benötigt, deshalb ist im Beispiel die Parameterliste entfallen.

Nach dem Schlüsselwort RETURNS folgt die Liste der Rückgabewerte. Wir haben hier zwei Rückgabewerte, nämlich die beiden Spalten unserer Abfrage. Wie Sie gleich sehen werden, kann über einen Rückgabewert auch eine ganze Spalte zurückgegeben werden.

Nach dem Schlüsselwort AS erfolgt die Definition der Prozedur. Zunächst werden alle benötigten Variablen definiert – das sind Speicherstellen zum Zwischenspeichern von Werten –, dann erfolgt in einem BEGIN..END-Block die Liste der Anweisungen, welche die Prozedur ausführen soll.

Zunächst setzen wir die Variable *i* auf eins, dann kommt eine FOR..SELECT..DO-Anweisung. Das ist zunächst einmal eine SELECT-Anweisung, die eine Reihe von Datensätzen liefert. Für jeden dieser Datensätze werden dann die Anweisungen ausgeführt, die im DO-Teil stehen.

Mit Hilfe der DISTINCT-Klausel wird wieder eine Liste aller Ortsnamen ermittelt. Diese werden mittels INTO im Rückgabewert *name* gespeichert. Beachten Sie den Doppelpunkt vor *name*. Dieser dient dazu, Parameter, Variablen und Rückgabewerte von Spaltenbezeichnern zu unterscheiden.

Für den DO-Teil reicht eine einzelne Anweisung nicht aus, deshalb benötigen wir einen BEGIN..END-Block. Dort weisen wir den Wert unserer Schleifenvariablen dem Rückgabewert *nummer* zu, anschließend wird *i* inkrementiert. Die Anweisung SUSPEND dient dazu, die zugewiesenen Werte an die Ergebnismenge zu übertragen. Sie ist für jeden zu übermittelnden Datensatz aufzurufen.

Nun zum Thema SET TERM: Innerhalb der STORED PROCEDURE müssen Sie die FOR SELECT...DO SUSPEND-Konstruktion mit einem Semikolon abschließen. Ein solches Semikolon schließt aber für gewöhnlich eine eingegebene Anweisung ab, wenn mehrere Anweisungen voneinander zu trennen sind. Also würde

Interactive SQL nach dem Semikolon die STORED PROCEDURE für beendet halten und wegen des Fehlens des abschließenden END ein unerwartetes Ende bemängeln.

Deswegen setzen wir das Terminationszeichen mit der Anweisung SET TERM auf das Zeichen ^ und schließen nach dem END die Definition der STORED PROCEDURE mit eben diesem Zeichen ab. In einer weiteren Anweisung setzen wir das Terminationszeichen auf das Semikolon zurück.

Ab Version 7 erkennt InterBase jedoch von selbst, wo eine STORED PROCEDURE ihr Ende hat, die Anweisung SET TERM ist dann überflüssig – sie stört aber auch nicht, wenn sie trotzdem gemacht wird. Da viele Leser noch bei früheren Versionen bleiben werden – insbesondere bei der Open-Source-Version 6.0 –, werden wir hier im Buch die SET TERM-Anweisung mit dazunehmen.

Zugriff auf die STORED PROCEDURE

Auf die Prozedur können Sie nun wie auf eine Tabelle zugreifen:

```
SELECT * FROM p_orte_test
```

Weil eine solche Prozedur mit einer SELECT-Anweisung aufgerufen wird, nennt man sie SELECT-Prozedur.

Die Ergebnismenge der Prozedur können Sie nun auch sortieren, vorwärts oder rückwärts. Nachdem dank der DISTINCT-Anweisung die Ergebnismenge ohnehin schon vorwärts sortiert ist, bleibt nur das Sortieren rückwärts.

```
SELECT *
       FROM p_orte_test
       ORDER BY name DESC
```

Da die Nummerierung an der Ergebnismenge der STORED PROCEDURE und nicht an der anschließend ausgeführten SELECT-Anweisung hängt, könnte man mit demselben Effekt auch nach der Nummer rückwärts sortieren – selbst die Ausführungszeiten würden nur minimal voneinander abweichen.

Apropos Ausführungszeiten: Vergleichen wir den Zugriff auf die STORED PROCEDURE mit dem Direktzugriff auf *t_adressen* unter Zuhilfenahme einer DISTINCT-Klausel.

Überraschenderweise ist die STORED PROCEDURE einen Hauch schneller, und zwar dank der etwas kürzeren *Execute*-Zeit. Die deutlich längeren *Fetch*-Zeiten erklären sich dadurch, dass ja auch noch die Nummer übertragen werden muss.

	Preprare	Execute	Fetch	Summe
Anweisung 1 Ausführung 1	591	83 254	65	83 910
Ausführung 2	544	83 176	65	83 785
Ausführung 3	548	82 904	64	83 516
Ausführung 4	544	82 993	64	83 601
Anweisung 2 Ausführung 1	493	84 219	51	84 763
Ausführung 2	491	83 652	52	84 195
Ausführung 3	494	84 763	51	85 308
Ausführung 4	497	83 926	52	84 475

Abbildung 5.2: STORED PROCEDURE gegen Direktzugriff

Eine Prozedur löschen

Wenn Sie eine Prozedur erstellt haben, die zwar syntaktisch korrekt ist (und deshalb vom DBS nicht mit einer Fehlermeldung abgewiesen wurde), aber nicht wunschgemäß arbeitet, muss diese wieder gelöscht werden.

```
DROP PROCEDURE p_gruppe_az_sort
```

Um eine Prozedur zu löschen, wird die Anweisung DROP PROCEDURE verwendet. Eine Prozedur kann nur dann gelöscht werden, wenn keine anderen Prozeduren oder TRIGGER darauf aufbauen.

Eine EXECUTE-Prozedur

Eine EXECUTE-Prozedur liefert keine Datenmenge zurück, sondern führt einige Anweisungen aus, beispielsweise INSERT-, DELETE- oder UPDATE-Befehle.

Nehmen wir einmal an, wir haben eine Tabelle *t_artikel*, in der – der Name legt die Vermutung ja schon nahe – Artikel gespeichert werden:

```
CREATE TABLE t_artikel
    (nummer INTEGER,
    gruppe INTEGER,
    hersteller VARCHAR(20),
    bezeichnung VARCHAR(20),
    preis FLOAT,
    PRIMARY KEY(nummer));
```

Um hier ein wenig Ordnung in die Sache zu bringen, werden alle Artikel einer Gruppe untergeordnet. Sind die Artikel beispielsweise die Produktpalette eines Hardware-Händlers, dann könnten die Gruppen *Prozessoren, Speicher, Festplatten ...* sein.

```
CREATE TABLE t_gruppe
    (nummer INTEGER,
    gruppe VARCHAR(20),
    PRIMARY KEY(nummer));
```

```
ALTER TABLE t_artikel
    ADD FOREIGN KEY (gruppe)
        REFERENCES t_gruppe (nummer);
```

Wir wollen nun eine Prozedur erstellen, die neue Datensätze in die Tabelle *t_artikel* einfügt. Dabei soll allerdings nicht die Gruppennummer, sondern die Gruppenbezeichnung als Parameter übergeben werden.

Die Gruppennummer hat die Prozedur selbst zu ermitteln, falls erforderlich, muss sie einen neuen Datensatz in die Tabelle *t_gruppe* einfügen.

```
SET TERM ^;
CREATE PROCEDURE p_artikel_ins
    (gruppe VARCHAR(25),
    bezeichnung VARCHAR(25),
    preis FLOAT,
    hersteller VARCHAR(25))
AS
    DECLARE VARIABLE zahl INTEGER;
BEGIN
    SELECT COUNT(nummer)
        FROM t_gruppe
        WHERE gruppe = :gruppe
        INTO :zahl;
    IF (:zahl = 0) THEN
        INSERT INTO t_gruppe (gruppe)
            VALUES (:gruppe);
    SELECT nummer
        FROM t_gruppe
        WHERE gruppe = :gruppe
        INTO :zahl;
    INSERT INTO t_artikel
            (gruppe, bezeichnung, preis, hersteller)
        VALUES
            (:zahl, :bezeichnung, :preis, :hersteller);
END^
SET TERM ;^
```

Im Gegensatz zur vorherigen Prozedur werden hier Parameter verwendet, schließlich müssen die Werte, die *p_artikel_ins* in die Tabelle einfügen soll, ja irgendwie an die Prozedur übergeben werden. Dafür sind diesmal keine Rückgabewerte erforderlich.

Nach dem Schlüsselwort AS wird diesmal eine Variable deklariert, auch hier muss der Typ der Variablen spezifiziert werden. Achten Sie auch darauf, die Variablendeklaration mit einem Semikolon abzuschließen. Wenn Sie mehrere Variablen deklarieren, müssen Sie jede Deklaration einzeln mit einem Semikolon abschließen.

Zunächst wird dann ermittelt, ob der Wert für die Gruppe bereits in der Tabelle *t_gruppe* enthalten ist. Zu diesem Zweck wird die Anzahl der entsprechenden Werte ermittelt, diese sollte null oder eins sein. Die Anzahl der Werte wird in die Variable *zahl* geschrieben.

Mit einer IF...THEN-Anweisung wird nun geprüft, ob die Anzahl der Werte gleich null ist. Wenn dies der Fall ist, wird der Wert in die Tabelle *t_gruppe* eingefügt.

Nun sollte der Wert einmal in der Tabelle *t_gruppe* enthalten sein. Mit der nächsten SELECT-Anweisung ermitteln wir die Nummer des betreffenden Datensatzes, auch diese Nummer wird in die Variable *zahl* geschrieben, wobei der vorherige Wert verloren geht.

Zuletzt erfolgt die eigentliche Aufgabe der Prozedur, die INSERT-Anweisung für die Tabelle *t_artikel*. Beachten Sie, dass in die Spalte *gruppe* der Wert der Variablen *zahl* geschrieben wird.

Um die Prozedur aufzurufen, wird die Anweisung EXECUTE PROCEUDRE verwendet, welcher der Prozedurname folgt. Anschließend werden der Reihe nach alle Werte für die Parameter aufgeführt. Diese Parameterliste ist in Klammern zu setzen:

```
EXECUTE PROCEDURE p_artikel_ins
    (Lichtsteuer-Software', 'LightControl Mini', 50,
        'TABU Datentechnik')
```

Beide Anweisungen sollten problemlos akzeptiert werden. Wenn es Probleme bei der Ausführung der Anweisungen gibt, könnte das daran liegen, dass die TRIGGER für die Tabellen *t_gruppe* oder *t_artikel* deaktiviert oder gelöscht sind.

Die Prozedur *p_artikel_ins* hat allerdings ein paar kleine »Schönheitsfehler«, einen davon wollen wir nun beheben.

Die verbesserte EXECUTE-Prozedur

Wir sind bislang davon ausgegangen, dass jeder Gruppenbezeichner nur einmal in der Tabelle *t_gruppe* vorhanden ist. Es ist aber durchaus möglich, dass ein und derselbe Wert mehrmals in der Spalte *gruppe* steht. Das könnte durch den Einsatz eines Sekundärschlüssels vermieden werden.

Man könnte aber auch die STORED PROCEDURE so umbauen, dass sie solche Dubletten gleich entfernt:

```
SET TERM ^;
CREATE PROCEDURE p_artikel_ins_del
    (gruppe VARCHAR(25),
    bezeichnung VARCHAR(25),
    preis FLOAT,
    hersteller VARCHAR(25))
```

```
AS
    DECLARE VARIABLE zahl SMALLINT;
BEGIN
    SELECT COUNT(nummer)
        FROM t_gruppe
        WHERE gruppe = :gruppe
        INTO :zahl;
    IF (:zahl = 0) THEN
        INSERT INTO t_gruppe (gruppe)
            VALUES (:gruppe);
    IF (:zahl > 1) THEN
    BEGIN
        SELECT MIN(nummer)
            FROM t_gruppe
            WHERE gruppe = :gruppe
            INTO :zahl;
        UPDATE t_artikel
            SET gruppe = :zahl
            WHERE gruppe IN (SELECT nummer
                FROM t_gruppe
                WHERE gruppe = :gruppe);
        DELETE FROM t_gruppe
            WHERE (gruppe = :gruppe)
                AND NOT (nummer = :zahl);
    END
    SELECT nummer
        FROM t_gruppe
        WHERE gruppe = :gruppe
        INTO :zahl;
    INSERT INTO t_artikel
            (gruppe, bezeichnung, preis, hersteller)
        VALUES
            (:zahl, :bezeichnung, :preis, :hersteller);
END^
SET TERM ;^
```

Bis auf den Prozedurenbezeichner gleicht der Kopf der Prozedur *p_artikel_ins_del* dem von *p_artikel_ins*, auch die ersten beiden Anweisungen sind identisch. In der Variablen *zahl* steht nun die Anzahl der Datensätze, die den als Parameter übergebenen Gruppenbezeichner enthalten. Mit einer IF-Anweisung wird nun das Programm in all den Fällen verzweigt, in denen *zahl* größer oder gleich zwei ist.

Für den Fall, dass der Gruppenbezeichner zu oft vorhanden ist, müssen nicht nur die überzähligen Datensätze gelöscht werden. Damit das Löschen möglich ist, müssen auch diejenigen Datensätze in der Tabelle *t_artikel*, die eine Referenz auf die zu löschenden Datensätze bilden, entsprechend abgeändert werden.

Langer Rede kurzer Sinn: Im Ausführungsteil der IF-Anweisung müssen mehrere Befehle ausgeführt werden. Dies bedingt die Verwendung eines BEGIN..END-Blocks: Zunächst wird die kleinste Nummer derjenigen Datensätze ermittelt, die den gleichen Gruppenbezeichner aufweisen, dieser Wert wird in die Variable *zahl* geschrieben. Der betreffende Datensatz soll derjenige sein, der in der Tabelle *t_gruppe* bleibt.

Nun sind alle Datensätze in *t_artikel*, die einen der betreffenden Datensätze in *t_gruppe* referenzieren, so abzuändern, dass sie mit der Spalte *gruppe* denjenigen Datensatz referenzieren, den wir in der Tabelle *t_gruppe* belassen.

Anschließend werden alle überzähligen Datensätze aus der Tabelle *t_gruppe* gelöscht.

Auf zwei Aspekte möchte ich in diesem Zusammenhang noch hinweisen: Wenn Sie die Erweiterungen, die hier bei der STORED PROCEDURE erforderlich werden, mit dem Aufwand für das Einrichten eines Sekundärschlüssels vergleichen, dann können Sie erkennen, welche Probleme eine »suboptimal« erstellte Datenbank aufwerfen kann.

Wenn Sie die Erweiterung der STORED PROCEDURE mit den Anweisungen vergleichen, die in einer 3. GL-Sprache wie C oder Pascal erforderlich gewesen wären, um dasselbe Problem zu lösen, dann erkennen Sie vielleicht die Vorzüge des bei SQL verwendeten mengenorientierten Ansatzes.

5.1.2 Übersicht über die Prozeduren-Sprache

In der Prozeduren-Sprache können Sie die Anweisungen SELECT, INSERT, UPDATE und DELETE verwenden. Darüber hinaus gibt es einige Anweisungen, die nur innerhalb von STORED PROCEDURES und TRIGGERN erlaubt sind.

FOR SELECT...DO

Mit FOR SELECT...DO lässt sich eine Schleife bilden, die alle Datensätze einer Tabelle durchläuft. Sie benötigen fast immer diese Anweisung, wenn Sie SELECT-Prozeduren erstellen, deren Ergebnismenge mehr als eine Zeile umfassen soll: Rufen Sie im DO-Teil der Anweisung dann den SUSPEND-Befehl auf. Eine SELECT-Prozedur mit mehrzeiliger Ergebnismenge ist im einfachsten Fall folgendermaßen aufgebaut:

```
    ...
BEGIN
    FOR SELECT
        ...
        INTO :spalte1, :spalte2, :spalte3
    DO SUSPEND;
END
```

Der DO-Teil einer solchen Anweisung kann nicht nur die Anweisung SUSPEND umfassen. Es ist durchaus auch möglich, dass Sie hier sehr umfangreiche Operationen vornehmen. In diesem Fall müssen Sie – wie bereits erwähnt – um die verwendeten Anweisungen einen BEGIN..END-Block bilden.

```
SET TERM ^;
CREATE PROCEDURE p_adressen_aender
AS
    DECLARE VARIABLE dat INTEGER;
    DECLARE VARIABLE lang VARCHAR(15);
BEGIN
    FOR
        SELECT nummer, tel
        FROM t_adressen
        WHERE tel NOT STARTING WITH "0"
        INTO :dat, :lang
    DO
    BEGIN
        lang = "030 / " || lang;
        UPDATE t_adr2
            SET tel = :lang
            WHERE nummer = :dat;
    END
END^
SET TERM ;^
```

Die STORED PROCEDURE *p_adressen_aender* ist für den Fall erstellt, dass die Berliner Filiale hin und wieder eine Diskette mit Adressen schickt, bei denen die Telefonnummern ohne Vorwahl aufgeführt sind. Die Prozedur durchläuft nun in einer FOR SELECT...DO-Schleife sämtliche Tabellenreihen und fügt vor jeder Nummer den String *030 /* ein.

Beachten Sie auch folgende Zeile:

```
    lang = "030 / " || lang;
```

Um einer Variablen einen anderen, »berechneten« Wert zuzuweisen, verwendet man das Gleichheitszeichen als Zuweisungsoperator. Im Ausdruck, welcher der Variablen zugewiesen wird, kann sich die Variable selbst befinden. Beachten Sie auch, dass hier vor den Variablennamen keine Doppelpunkte stehen. Diese werden nur dort verwendet, wo eine Verwechslung mit Spaltenbezeichnern prinzipiell möglich wäre.

Am Rande: Diese STORED PROCEDURE ist ein hübsches Beispiel dafür, wie man etwas Einfaches auch kompliziert erledigen kann. Das gleiche Ergebnis erhält man nämlich auch mit folgender Anweisung:

```
UPDATE t_adressen
    SET tel = "030 / " || tel
    WHERE tel NOT STARTING WITH "0"
```

Die SELECT-Anweisung

Eine SELECT-Anweisung muss natürlich nicht in einer Schleife verwendet werden – insbesondere dann, wenn Aggregatfunktionen ermittelt werden, macht dies meist auch nicht viel Sinn. Die Syntax bleibt in diesem Fall dieselbe:

```
SET TERM ^;
CREATE PROCEDURE p_artikel_minmax
RETURNS
    (minimum FLOAT,
    maximum FLOAT)
AS
BEGIN
    SELECT MIN(Preis), MAX(Preis)
        FROM t_artikel
        INTO :minimum, :maximum;
    SUSPEND;
END^
SET TERM ;^
```

Die Variablen hinter INTO müssen in derselben Reihe aufgeführt werden wie die Spalten oder Aggregatfunktionen hinter SELECT. Die Anweisung SUSPEND ist erforderlich, damit die Datensätze an die aufrufende SELECT-Anweisung übergeben werden.

Die WHILE...DO-Anweisung

Wenn in STORED PROCEDURES eine Schleife verwendet wird, dann im Regelfall dafür, um mehrere Zeilen aus einer Tabelle auszulesen. Dafür wird – wie bereits vorgestellt – die FOR SELECT..DO-Anweisung verwendet.

In manchen Fällen werden jedoch andere Schleifen benötigt. Hier kennt InterBase die WHILE..DO-Schleife. Solange die Bedingung im WHILE-Teil erfüllt ist, wird die Anweisung im DO-Teil ausgeführt.

Wir wollen eine WHILE..DO-Schleife verwenden, um eine mathematische Funktion, nämlich die Fakultät einer Zahl, zu berechnen. Die Fakultät einer Zahl berechnet sich wie folgt:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$$

Abbildung 5.3: Berechnung der Fakultät

Um die Fakultät zu berechnen, wird eine Schleife mit *x* Schleifendurchläufen verwendet, als Schleifenvariable verwenden wir *i*. Die Schleifenvariable wird in jeder Schleife mit dem Funktionsergebnis *f* multipliziert, anschließend wird sie inkrementiert (um eins erhöht).

```
SET TERM ^;
CREATE PROCEDURE p_fakult (x INTEGER)
RETURNS
    (f INTEGER)
AS
    DECLARE VARIABLE i INTEGER;
BEGIN
    f = 1;
    i = 1;
    WHILE (i <= x) DO
    BEGIN
        f = f * i;
        i = i + 1;
    END
    IF (x < 1) THEN
        f = NULL;
    SUSPEND;
END^
SET TERM ;^
```

Wie Ihnen vielleicht aufgefallen ist, werden zu Beginn des Ausführungsteils die Variablen *f* und *i* initialisiert, also mit Werten belegt. Würde man dies nicht tun, dann hätten Sie den Wert NULL, der auch dann beibehalten würde, wenn mit ihnen die folgenden Operationen durchgeführt werden.

Damit aus einer Schleife keine Endlosschleife wird, sollten Sie immer darauf achten, dass die WHILE-Bedingung irgendwann nicht mehr erfüllt wird. Für Werte kleiner 1 ist die Fakultät nicht definiert, wir geben hier dann NULL zurück.

Eine Fakultät lässt sich auch mit einem rekursiven Aufruf der berechnenden Funktion berechnen, wie wir im nächsten Abschnitt sehen werden.

Verzweigungen mit IF...THEN...ELSE

Die IF..THEN-Verzweigung haben wir bereits häufiger verwendet. Sie lässt sich um einen ELSE-Zweig erweitern. Die Anweisung im ELSE-Teil wird immer dann ausgeführt, wenn die IF-Bedingung nicht zutrifft. Es wird also entweder die Anweisung im THEN-Teil oder die Anweisung im ELSE-Zweig ausgeführt. Es ist nicht möglich, dass beide oder keine der Anweisungen ausgeführt werden.

Als Beispiel wollen wir nun die Fakultät mit Hilfe einer Rekursion berechnen. Statt die Prozedur mit DROP PROCEDURE zu löschen und mit CREATE PROCEDURE neu zu erstellen, werden wir sie mit ALTER PROCEDURE ändern. (Im Regelfall werden Sie umfangreichere Prozeduren nicht auf Anhieb vollständig und fehlerfrei erstellen. Mit ALTER PROCEDURE können Sie Ihre Prozeduren schrittweise entwickeln.)

```
SET TERM ^;
ALTER PROCEDURE p_fakult (x INTEGER)
    RETURNS (f INTEGER)
AS
    DECLARE VARIABLE num INTEGER;
BEGIN
    IF (x < 1) THEN
    BEGIN
        f = NULL;
        SUSPEND;
        EXIT;
    END
    IF (x=1) THEN   f = 1;
    ELSE
    BEGIN
        num = :x - 1;
        EXECUTE PROCEDURE p_fakult :num
            RETURNING_VALUES :f;
        f = :f * x;
    END
    SUSPEND;
END ^
SET TERM ;^
```

In dieser Prozedur verwenden wir einige Neuheiten: Zunächst muss gleich zu Beginn des Ausführungsteils festgestellt werden, ob der Parameter nicht kleiner eins ist. Ist dies der Fall, dann wird dem Rückgabewert *f* der Wert NULL zugewiesen und anschließend die Ausführung der Prozedur mit EXIT abgebrochen. Würde diese Prüfung erst am Ende der Prozedur erfolgen, dann würde sich die Prozedur so lange selbst aufrufen, bis irgendwo ein Überlauf erfolgte.

Hat der Parameter *x* den Wert eins, dann verzweigt die IF..THEN..ELSE-Anweisung in den THEN-Teil und gibt den Wert eins zurück.

Hat der Wert dagegen einen Wert größer eins (bei Werten kleiner eins gelangt die Prozedurausführung gar nicht bis zu diesem Punkt), dann verzweigt die IF..THEN..ELSE-Anweisung in den ELSE-Teil. Hier ruft sich die Prozedur nun selbst auf.

Um eine andere Prozedur aufzurufen, verwenden Sie die Anweisung EXECUTE PROCEDURE. Nach der Nennung des Prozedurennamens werden die Parameter übergeben, nach dem Schlüsselwort RETURNING VALUES werden die Ergebnisse zurückgenommen.

Für gewöhnlich ruft eine Prozedur eine andere Prozedur auf. Ruft eine Prozedur sich selbst auf, dann spricht man von einer Rekursion. Hier ist dann auf eine funktionierende Abbruchbedingung zu achten, damit sich dieser Vorgang nicht endlos wiederholt.

Nehmen wir an, es soll die Fakultät der Zahl 4 berechnet werden. Die erste Prozedur verzweigt in den ELSE-Teil und ruft die zweite Prozedur mit dem Parameter 3 auf. Die zweite Prozedur verzweigt ebenfalls in den ELSE-Teil und ruft die dritte Prozedur mit dem Parameter 2 auf. Die dritte Prozedur verzweigt abermals in den ELSE-Teil und ruft eine vierte Prozedur mit dem Parameter 1 auf.

Die vierte Prozedur verzweigt nun in den THEN-Teil und springt mit dem Ergebnis 1 zur dritten Prozedur zurück. Die dritte Prozedur multipliziert diese 1 mit ihrem Parameter 2 und springt mit dem Ergebnis 2 zur zweiten Prozedur zurück. Die zweite Prozedur multipliziert diese 2 mit ihrem Parameter 3 und springt mit dem Ergebnis 6 zurück zur ersten Prozedur. Die erste Prozedur multipliziert diese 6 mit ihrem Parameter 4 und gibt 24 als Ergebnis zur aufrufenden SELECT-Anweisung aus.

Statt mit EXECUTE PROCEDURE könnte man den rekursiven Aufruf auch mit SELECT erledigen:

```
...
IF (x=1) THEN   f = 1;
    ELSE
    BEGIN
        SELECT * FROM p_fakult (:x - 1)
            INTO :f;
        f = :f * x;
    END
    SUSPEND;
...
```

SELECT oder EXECUTE PROCEDURE

Wie Sie in der Übung gesehen haben, lässt sich eine Prozedur sowohl mit SELECT als auch mit EXECUTE PROCEDURE aufrufen. Den Unterschied wollen wir anhand des folgenden Beispiels ermitteln:

```
SET TERM ^;
CREATE PROCEDURE p_liste RETURNS (r INTEGER)
AS
BEGIN
```

```
    r = 0;
    WHILE (r < 5) DO
    BEGIN
        r = r + 1;
        SUSPEND;
    END
END ^
SET TERM ;^
```

Wir wollen die Prozedur zunächst mit einer SELECT-Anweisung aufrufen. Wie Sie sehen, erscheinen hier die Werte eins bis fünf in der Ergebnismenge.

```
SELECT * FROM p_liste

          R
===========

          1
          2
          3
          4
          5
```

Alternativ dazu soll die Prozedur nun mit EXECUTE PROCEDURE gestartet werden. Hier ist nur der Wert eins in der Ergebnismenge.

```
EXECUTE PROCEDURE p_liste

          R
===========
          1
```

Wie Sie sehen, kann die Ergebnismenge von EXECUTE PROCEDURE maximal eine Zeile umfassen.

In diesem Punkt unterscheiden sich die verschiedenen SQL-Server. Bei InterBase kann eine Prozedur, die eine Ergebnismenge zurückgibt, immer mit SELECT aufgerufen werden. EXECUTE PROCEDURE dagegen wird bei Prozeduren verwendet, die keine Ergebnismenge liefern.

Variablen und Parameter

In einer Prozedur gibt es folgende Variablen:

- Die lokalen Variablen, die nur innerhalb der Prozedur gültig sind. Sie werden zu Beginn der Prozedur zwischen AS und BEGIN deklariert.
- Die Eingangs-Parameter, die nach dem Prozedur-Namen aufgelistet sind.

- Die Rückgabe-Parameter, die nach dem Schlüsselwort RETURNS aufgeführt werden.

```
CREATE PROCEDURE p_test
    (eingangsparameter1 INTEGER,
    eingangsparameter2 VARCHAR(20))
RETURNS (ausgangsparameter1 INTEGER,
        ausgangsparameter2 VARCHAR(20))
AS
    DECLARE VARIABLE lokalevariable1 INTEGER;
    DECLARE VARIABLE lokalevariable2 VARCHAR(20);
BEGIN
    ...
END
```

Innerhalb des Anweisungsteils werden alle Variablen und Parameter gleich behandelt. Beachten Sie auch, dass immer dann, wenn die Gefahr besteht, dass Variablen mit Spaltenbezeichnern verwechselt werden, diese zur Unterscheidung mit einem vorangestellten Doppelpunkt gekennzeichnet werden müssen.

5.1.3 Zugriffsberechtigung für Prozeduren

Selbstverständlich kann nicht jeder Benutzer einfach auf jede Prozedur zugreifen. Auch dafür müssen die Rechte erst vergeben werden:

```
GRANT EXECUTE ON PROCEDURE p_gruppe_az_sort TO emil
```

Ohne explizite Vergabe der Rechte können der Besitzer und der SYSDBA eine Prozedur ausführen.

Greift die Prozedur ihrerseits auf andere Prozeduren oder auf Tabellen zu, dann braucht der Benutzer auch für diese Tabellen und Prozeduren die entsprechenden Rechte – es sei denn, die Prozedur hat selbst diese Rechte.

Rechte an Prozeduren vergeben

In vielen Fällen soll ein Benutzer eine Prozedur ausführen können, nicht aber die darunter liegenden Tabellen einsehen oder verändern können. Dies lässt sich jedoch nicht vermeiden, wenn man ihm sämtliche Rechte dafür gewähren muss.

Dies kann dadurch vermieden werden, dass die benötigten Rechte nicht an die Benutzer vergeben werden, sondern an die Prozedur selbst.

```
GRANT EXECUTE ON  PROCEDURE p_gruppe_az
    TO PROCEDURE p_gruppe_az_sort;
```

5.1.4 Fehlerbehandlung

Tritt in einer Prozedur ein Fehler auf, dann wird die Ausführung der Prozedur mit einer Fehlermeldung abgebrochen. Es gibt jedoch auch die Möglichkeit, eine Fehlerbehandlung zu programmieren. In diesem Fall wird dann auf eine Fehlermeldung verzichtet und es werden stattdessen die vorgesehenen Anweisungen ausgeführt.

Berechnung statistischer Funktionen

Man kennt in der Statistik nicht nur den Mittelwert M (SQL-Funktion AVG), sondern auch noch die Varianz V und die Standardabweichung s. Wie diese definiert sind, zeigt *Abbildung 5.4*:

$$M = \frac{\sum_{0}^{n} x_i}{n}$$

$$V = \frac{\sum_{0}^{n} (x_i - M)^2}{n}$$

$$s = \sqrt{V} = \sqrt{\frac{\sum_{0}^{n} (x_i - M)^2}{n}}$$

Abbildung 5.4: Mittelwert, Varianz und Standardabweichung

Die Standardabweichung können wir im Moment noch nicht berechnen, weil wir noch keine Funktion für die Wurzel haben, wohl aber die Varianz:

```
SET TERM ^;
CREATE PROCEDURE p_werte_stat
RETURNS
    (varianz FLOAT)
AS
    DECLARE VARIABLE qwt FLOAT;
    DECLARE VARIABLE wert FLOAT;
    DECLARE VARIABLE mittel FLOAT;
    DECLARE VARIABLE zahl FLOAT;
BEGIN
    SELECT AVG(wert)
        FROM t_werte
        INTO :mittel;
```

```
    SELECT COUNT(wert)
        FROM t_werte
        INTO :zahl;
    qwt = 0;
    FOR SELECT wert
        FROM t_werte
        INTO :wert
    DO qwt = qwt + ((wert - mittel) * (wert - mittel));
    varianz = qwt / zahl;
    SUSPEND;
END ^
SET TERM ;^
```

Die Prozedur berechnet die Varianz der Spalte *Wert* in der Tabelle *t_werte*. Wären nun in dieser Tabelle keine Datensätze, würde bei der Berechnung der Varianz eine Division durch null durchgeführt, die Prozedur wird mit der Fehlermeldung -802 (*arithmetic exception, numeric overflow, or string truncation*) abgebrochen.

WHEN...DO

Wir wollen die Prozedur nun so abändern, dass eine solche Fehlermeldung abgefangen wird und als Ergebnis der Wert NULL zurückgegeben wird.

```
SET TERM ^;
ALTER PROCEDURE p_werte_stat
RETURNS
    (varianz FLOAT)
AS
    DECLARE VARIABLE qwt FLOAT;
    DECLARE VARIABLE wert FLOAT;
    DECLARE VARIABLE mittel FLOAT;
    DECLARE VARIABLE zahl FLOAT;
BEGIN
    SELECT AVG(wert)
        FROM t_werte
        INTO :mittel;
    SELECT COUNT(wert)
        FROM t_werte
        INTO :zahl;
    qwt = 0;
    FOR SELECT wert
        FROM t_werte
        INTO :wert
    DO qwt = qwt + ((wert - mittel) * (wert - mittel));
    BEGIN
        varianz = qwt / zahl;
```

```
        WHEN SQLCODE -802
            DO varianz = NULL;
    END
    SUSPEND;
END^
SET TERM ;^
```

Die Änderungen beginnen nach der FOR SELECT..DO-Schleife. Zunächst einmal muss ein neuer BEGIN..END-Block eingefügt werden. Nur in diesem Block werden die Fehlermeldungen abgefangen. Es folgt die Anweisung, bei welcher der Fehler auftreten kann.

Die Fehlerbehandlung erfolgt mit der WHEN..DO-Anweisung. Wenn in diesem Fall der SQL-Fehlercode -802 auftritt, dann wird der Variablen *varianz* der Wert NULL zugewiesen. Implizit werden dabei auch der Abbruch der Prozedur und die Anzeige der Fehlermeldung vermieden.

Im WHEN-Teil können nicht nur SQL-Fehlercodes, sondern auch EXCEPTIONS und GDSCODE-Nummern angegeben werden.

EXCEPTIONS

Die Fehlermeldungen, die von ISQL angezeigt werden, sind bisweilen etwas »interpretationsbedürftig«, außerdem auf Englisch und für den Laien meist überhaupt nicht verständlich. Deshalb besteht die Möglichkeit, eigene EXCEPTIONS zu definieren, deren Fehlermeldungen dann selbst erstellt werden.

Abbildung 5.5: Fehlermeldung einer selbst definierten EXCEPTION

Um eine EXCEPTION zu erstellten, verwenden Sie die Anweisung CREATE EXCEPTION und geben Sie den Namen sowie die Fehlermeldung an.

```
CREATE EXCEPTION e_werte_stat_error
    "Keine Werte in der Tabelle t_werte"
```

Mit Hilfe der Fehlerbehandlung kann eine solche EXCEPTION ausgelöst werden.

```
BEGIN
    varianz = qwt / zahl;
    WHEN SQLCODE -802
        DO EXCEPTION e_werte_stat_error;
END
```

Events

Für gewöhnlich gehen bei einem Client-Server-Datenbank-System die Aktionen vom Client aus, und der Server reagiert darauf. Nun gibt es Aufgaben, die sich mit einem solchen Mechanismus nicht gut lösen lassen. Ein Beispiel:

Bei Firmen oder Vereinen ist es manchmal üblich, das tausendste Mitglied oder den hunderttausendesten Kunden entsprechend zu feiern. Es wäre dann Aufgabe des jeweiligen Datenbanksystems, diesen Vorgang zu melden. Würde man eine solche Logik auf der Seite des Clients implementieren, dann müsste nach jedem angelegten Datensatz geprüft werden, wie hoch die Zahl der Mitglieder beziehungsweise Kunden ist, was unnötig das Netzwerk belastet.

Viel besser wäre es, wenn der Server die Anzahl der Datensätze selbständig überwacht und sich gegebenenfalls bei den interessierten Clients meldet. Um so etwas zu realisieren, gibt es Events:

```
CREATE TRIGGER trig_adressen_event FOR t_adressen
AFTER INSERT
AS
DECLARE VARIABLE num INTEGER;
BEGIN
  SELECT COUNT(id)
    FROM t_adressen
    INTO :num;
  IF (num = 10000)
    THEN POST_EVENT "Jubilaeum";
END
```

Nach dem Einfügen eines jeden Datensatzes prüft nun der Trigger, ob die Zahl der Datensätze bei 10000 liegt. Ist dies der Fall, dann wird der Event *Jubilaeum* ausgelöst.

Bei der Verwendung von Events sind jedoch zwei Sachen zu beachten: Zum einen wird der Event erst dann ausgelöst, wenn die Transaktion mit COMMIT beendet wird. Das vermeidet zwar einen »Fehlalarm«, aber wenn nun in einer Transaktion mehrere Datensätze eingefügt werden, dann erfolgt die Nachricht erst beim Aufruf von COMMIT. Der Event würde bei diesem Beispiel zwar auch dann ausgelöst, wenn inzwischen die Zahl über 10000 liegen würde, aber man kann sich nicht darauf verlassen, dass der letzte Datensatz derjenige ist, der den Event ausgelöst hat.

Zum anderen benötigt die Client-Anwendung eine spezielle Komponente, die sich von solchen Events benachrichtigen lassen kann. IBX hat mit *IBEvents* eine solche Komponente, bei dbExpress können Sie dagegen von Events keinen Gebrauch machen.

5.1.5 Prozeduren löschen und ändern

Um eine Prozedur zu löschen, verwendet man die Anweisung DROP PROCEDURE.

```
DROP PROCEDURE p_werte_stat
```

Eine Prozedur kann nur dann gelöscht werden, wenn sie von keiner anderen Prozedur verwendet wird.

Nicht nur aus diesem Grund ist es unzweckmäßig, eine Prozedur zu löschen und anschließend neu zu erstellen. Effektiver ist es, die Prozedur mit ALTER PROCEDURE zu ändern. Dieser Befehl – wir haben ihn ja bereits verwendet – gleicht weitgehend CREATE PROCEDURE, löst aber keine Fehlermeldung aus, wenn eine Prozedur des angegebenen Namens schon besteht.

Eine Prozedur lässt sich auch dann mit ALTER PROCEDURE überschreiben, wenn sie von anderen Prozeduren verwendet wird. Leider achtet InterBase dabei noch nicht einmal darauf, dass benötigte Parameter- oder Rückgabewerte erhalten bleiben. Sie sollten beim Einsatz von ALTER PROCEDURE deshalb eine gewisse Vorsicht walten lassen.

5.2 TRIGGER

Ein TRIGGER ist eine Art Prozedur, die jedoch nicht vom Benutzer, sondern vom System selbst gestartet wird. Auf diese Weise lassen sich zusätzliche Maßnahmen ergreifen, wenn in Tabellen Datensätze eingefügt, geändert oder gelöscht werden.

5.2.1 CREATE TRIGGER

Der TRIGGER *trig_art* aus unserer Beispieldatenbank fügt immer dann, wenn der Spalte *nummer* kein Wert zugewiesen wird, einen Generatorwert ein. Auf diese Weise bastelt man sich selbst inkrementierende Felder.

```
SET TERM ^;
CREATE TRIGGER trig_adressen
    FOR t_adressen
    ACTIVE
    BEFORE INSERT
    POSITION 0
AS
```

```
BEGIN
    IF(NEW.id IS NULL)
        THEN NEW.id = GEN_ID(g_adressen, 1);
END ^
SET TERM ;^
```

Um einen TRIGGER zu erstellen, verwendet man die Anweisung CREATE TRIGGER, gefolgt vom Namen des TRIGGERS. Nach dem Schlüsselwort FOR wird die Tabelle genannt, für die der TRIGGER definiert wird.

Das Schlüsselwort ACTIVE zeigt an, dass der TRIGGER aktiviert sein soll, inaktive TRIGGER würden mit INACTIVE erstellt. Da ACTIVE der Vorgabewert ist, werden aktive TRIGGER erstellt, wenn nichts angegeben wird. Die Option INACTIVE wird vor allem im Zusammenhang mit ALTER TRIGGER verwendet, um Trigger vorübergehend zu deaktivieren.

Mit BEFORE zeigen wir an, dass der TRIGGER vor der Durchführung der folgenden Aktion – in diesem Fall INSERT – ausgelöst werden soll. Soll er danach ausgelöst werden, dann ist AFTER zu verwenden. TRIGGER können vor beziehungsweise nach den Anweisungen INSERT, UPDATE oder DELETE gesetzt werden.

An eine Aktion lassen sich mehrere TRIGGER koppeln, die dann in der Reihenfolge von POSITION ausgeführt werden. Der Wert von POSITION liegt zwischen 0 und 32767, wobei der TRIGGER mit der Zahl null als Erstes ausgeführt wird. Haben mehrere TRIGGER dieselbe POSITION-Nummer, dann ist die Reihenfolge ihrer Ausführung zufällig.

Nach dem Schlüsselwort AS werden Variablen deklariert, was in diesem Beispiel nicht erforderlich ist, zwischen BEGIN und END folgt dann der Ausführungsteil. Dort wird geprüft, ob bereits eine Nummer vorgegeben ist, andernfalls wird ein Generatorwert eingefügt.

Neu ist hier der Bezeichner *NEW.id*. Mit NEW in Kombination mit einem Spaltenbezeichner wird auf die neu einzufügenden oder zu ändernden Spaltenwerte zugegriffen, während man mit OLD auf die bisherigen Werte zugreifen kann.

Aktionen verhindern

Wenn man innerhalb eines BEFORE-TRIGGERS eine Exception auslöst, welche nicht behandelt wird, dann kann man die Ausführung der entsprechenden Aktion verhindern.

Nehmen wir an, wir wollten in der Tabelle *t_art* das Löschen der Datensätze eins bis sieben verhindern. Dazu wird zunächst eine EXCEPTION definiert.

```
CREATE EXCEPTION e_art_loschen     "Die Datensätze bis
    zur Nummer 7 dürfen nicht gelöscht werden"
```

Nun erstellen wir einen TRIGGER, der prüft, ob ein zu löschender Datensatz eine Nummer kleiner gleich sieben hat, und gegebenenfalls die EXCEPTION auslöst.

```
SET TERM ^;
CREATE TRIGGER trig_art_del
    FOR t_art
    BEFORE DELETE
AS
BEGIN
    IF (OLD.nummer <= 7) THEN
        EXCEPTION e_art_loschen;
END ^
SET TERM ;^
```

Neben ACTIVE fehlen hier auch die POSITION-Angaben. In einem solchen Fall wird die Position 0 verwendet.

Erfreulicherweise funktioniert diese Sicherung selbst dann, wenn der zu löschende Datensatz über die Spalte *bezeichnung* spezifiziert wird.

```
DELETE FROM t_art
    WHERE bezeichnung = "Handy"
```

Hier erkennt das DBS dann selbst, dass der spezifizierte Datensatz die Nummer vier hat, und verweigert das Löschen.

5.2.2 TRIGGER ändern und löschen

Um einen TRIGGER zu löschen, wird die Anweisung DROP TRIGGER verwendet. Vorher müssen alle Transaktionen abgeschlossen werden, welche den TRIGGER verwendet haben. (Außerdem müssen Sie Besitzer des TRIGGER und/oder SYSDBA sein.)

Um einen TRIGGER zu ändern, verwenden Sie ALTER TRIGGER. Sie können dabei, wie bei einer STORED PROCEDURE, den TRIGGER komplett neu definieren.

Sie können aber auch einzelne Angaben des TRIGGER-Kopfes ändern. Am häufigsten wird dabei der TRIGGER de- oder reaktiviert.

```
ALTER TRIGGER trig_art_del
INACTIVE

ALTER TRIGGER trig_art_del
ACTIVE
```

Sie können aber auch die auslösende Aktion und/oder die Position des TRIGGERS ändern.

```
ALTER TRIGGER trig_art_del
      AFTER INSERT
      POSITION 11
```

Was Sie nicht ändern können, auch nicht bei einer kompletten Redefinition des TRIGGERS, ist die Tabelle, an die der TRIGGER gekoppelt ist. Wenn Sie diese ändern wollen, können Sie nur den TRIGGER komplett löschen und mit CREATE TRIGGER neu erstellen.

5.2.3 Ansichten über mehrere Tabellen aktualisieren

Normalerweise sind Ansichten, die einen JOIN beinhalten, nicht aktualisierbar. Mit Hilfe von drei Triggern kann man dafür sorgen, dass auch bei solchen Ansichten INSERT-, UPDATE- und DELETE-Operationen möglich sind.

Am Anfang des Kapitels haben wir beim Besprechen der EXECUTE-Prozedur die Tabellen *t_artikel* und *t_gruppe* verwendet. Ein Ansicht beider Tabellen könnte wie folgt aussehen:

```
CREATE VIEW v_artikel AS
       SELECT
              a.nummer,
              g.gruppe,
              a.hersteller,
              a.bezeichnung,
              a.preis
       FROM t_artikel a
              LEFT OUTER JOIN t_gruppe g
                     ON g.nummer = a.gruppe
```

Wir wollen nun Trigger für die verschiedenen Datenänderungs-Operationen erstellen:

INSERT

Der Trigger *trig_artikel_ins* verteilt den einzufügenden Datensatz auf die Tabellen *t_artikel* und *t_gruppe*.

```
SET TERM ^;
CREATE TRIGGER trig_artikel_ins
    FOR v_artikel
    BEFORE INSERT
AS
    DECLARE VARIABLE zahl INTEGER;
BEGIN
    SELECT COUNT(*)
        FROM t_gruppe
        WHERE gruppe = NEW.gruppe
        INTO :zahl;
    IF (zahl = 0) THEN
        INSERT INTO t_gruppe (bezeichnung)
```

```
        VALUES (NEW.gruppe);
    SELECT MIN(nummer)
        FROM t_gruppe
        WHERE gruppe = NEW.gruppe
        INTO :zahl;
    INSERT INTO t_artikel (nummer, gruppe, hersteller,
            bezeichnung, preis)
        VALUES (NEW.nummer, :zahl, NEW.hersteller,
            NEW.bezeichnung, NEW.preis);
END^
SET TERM ;^
```

Zunächst wird geprüft, ob der Gruppenname schon in der Tabelle *t_gruppe* vorhanden ist. Ist dies nicht der Fall, dann wird er in der Tabelle *t_gruppe* (und nicht *t_artikel*!) eingefügt. Der TRIGGER *trig_gruppe_ins* sorgt dafür, dass der Datensatz auch in die Tabelle *t_gruppe* geschrieben wird.

Anschließend wird die Nummer des Datensatzes ermittelt und in die Variable *zahl* geschrieben. Zuletzt erfolgt der INSERT-Befehl zum Einfügen des neuen Datensatzes in die Tabelle *t_artikel*.

UPDATE

Den Trigger für das Einfügen eines Datensatzes erstellt man wie folgt:

```
SET TERM ^;
CREATE TRIGGER trig_artikel_upd
    FOR v_artikel
    BEFORE UPDATE
AS
    DECLARE VARIABLE zahl INTEGER;
BEGIN
    SELECT COUNT(*)
        FROM t_gruppe
        WHERE gruppe = NEW.gruppe
        INTO :zahl;
    IF (zahl = 0) THEN
        INSERT INTO t_gruppe (bezeichnung)
            VALUES (NEW.gruppe);
    SELECT MIN(nummer)
        FROM t_gruppe
        WHERE gruppe = NEW.gruppe
        INTO :zahl;
    UPDATE t_artikel
        SET nummer = NEW.nummer,
            gruppe = :zahl,
            hersteller = NEW.hersteller,
```

```
            bezeichnung = NEW.bezeichnung,
            preis = NEW.preis
        WHERE (nummer = OLD.nummer)
END^
SET TERM ;^
```

Sicherheitshalber muss damit gerechnet werden, dass der Wert der Spalte *gruppe* bei der UPDATE-Anweisung geändert wird. Deshalb wird auch hier wieder die Gruppennummer ermittelt und in die Spalte *gruppe* von *t_artikel* geschrieben.

In der WHERE-Klausel reicht auch hier wieder die Primärschlüsselspalte, selbst wenn die WHERE-Anweisung eine komplett andere Spalte spezifiziert.

```
UPDATE v_artikel SET hersteller = "Test"
    WHERE hersteller = "Intel"
```

DELETE

Sehr übersichtlich ist der TRIGGER für die DELETE-Anweisung:

```
SET TERM ^;
CREATE TRIGGER trig_artikel_del
    FOR t_artikel
    BEFORE DELETE
AS
BEGIN
    DELETE FROM t_artikel
        WHERE (nummer = OLD.nummer);
END^
SET TERM ;^
```

Beachten Sie, dass auch dieser TRIGGER als BEFORE definiert werden muss.

Beim Löschen eines Datensatzes kann es vorkommen, dass ein Eintrag in *t_gruppe* nicht mehr referenziert wird. Hier erstellen wir einen TRIGGER, der sicherstellt, dass dieser Datensatz in *t_gruppe* dann gelöscht wird:

```
SET TERM ^;
CREATE TRIGGER trig_artikel_del_gruppe
    FOR v_artikel
    AFTER DELETE
AS
    DECLARE VARIABLE nummer INTEGER;
    DECLARE VARIABLE anzahl INTEGER;
BEGIN
    SELECT MIN(nummer)
        FROM t_gruppe_neu
        WHERE gruppe = OLD.gruppe
        INTO :nummer;
```

```
    SELECT COUNT(*)
        FROM t_artikel
        WHERE gruppe = :nummer
        INTO :anzahl;
    IF (anzahl = 0) THEN
        DELETE FROM t_gruppe
            WHERE nummer = :nummer;
END ^
SET TERM ;^
```

6 InterBase einrichten

Bei kleineren Projekten braucht man sich um die Einrichtung von InterBase weiter keine Sorgen zu machen: Man installiert den Interbase-Server auf dem Rechner, der als Server laufen soll, und die Sache ist erledigt. Im Gegensatz zu manch anderem SQL-Server ist InterBase erfreulich genügsam hinsichtlich der benötigten Ressourcen und des Wartungsaufwands.

Es überrascht wohl nicht, dass ein minimalistisch eingerichtetes System auch kein Optimum an Performance bringt. In vielen Fällen ist das auch gar nicht erforderlich: Moderne Hardware ist so leistungsstark, dass auch suboptimal eingerichtete Systeme oft ausreichend performant laufen. Anders sieht das aus, wenn viele Clients auf den Server zugreifen, wenn es hier zu einem hohen Datendurchsatz kommt, wenn kurze Antwortzeiten erforderlich sind – dann sollte man die vorhandenen Optimierungs-Potenziale nutzen, anstatt viel Geld in teure Hardware zu investieren.

Nach meiner Erfahrung sollte Optimierung zuallererst bei der Datenbankanwendung ansetzen: Hier bestimmen Sie, ob der Server viel oder wenig zu tun hat, wie viele Datensätze über das Netzwerk müssen, ob JOINS zweckmäßig formuliert sind oder nicht. Prüfen Sie auch, ob die Datenbank sinnvoll erstellt ist, also vernünftig normalisiert, mit Indizes dort, wo es sich lohnt. Bei der Datenbankanwendung und bei der Datenbank können Sie Performance-Gewinne realisieren, die im Bereich von Zehnerpotenzen liegen.

Erst der zweite Schritt sollte sich dann mit dem Server und dessen Konfigurierung beschäftigen. Können weitere Festplatten sinnvoll eingesetzt werden, ist mehr Arbeitsspeicher sinnvoll, welche Parameter sollten verändert werden. Hier sind auch beachtliche Performance-Gewinne möglich, diese werden aber selten die Größenordnung dessen erreichen, was mit Optimierung der Datenbankanwendung möglich ist.

6.1 Die Hardware

Im einfachsten Fall wird InterBase auf einem bestehenden Rechner installiert und läuft dann. Bei einem Einplatz-System oder einem kleinen Peer-to-Peer-Netzwerk muss man sich meist keine Gedanken darüber machen, wie die Hardware-Situation aussieht. Achten Sie darauf, dass die Backup-Frage geklärt ist, und widmen Sie sich wichtigeren Aufgaben.

Steigt nun jedoch die Zahl der Benutzer – sagen wir mal, ab fünf fängt man an, sich Gedanken zu machen –, und handelt es sich um ein System, mit dem häufig gearbeitet wird, dann sollte der Server für diese Aufgabe entsprechend eingerichtet werden.

Einen dezidierten Server verwenden

Während in kleinen Peer-to-Peer-Netzwerken der Datenbankserver durchaus noch auf einer Arbeitsstation nebenher laufen kann, sollte man bei größeren Netzwerken einen eigenen Server verwenden, nach Möglichkeit nach dem Motto »eine Aufgabe – ein Server«. Vermeiden Sie es, einen Server sowohl als Print- und File-Server als auch als Datenbankserver zu verwenden.

Es ist jedoch nicht unbedingt nötig, gleich für jede Datenbank einen eigenen Server zu verwenden – vorausgesetzt, die Ressourcen erlauben eine gemeinsame Nutzung. Es gibt aber auch die Strategie, dass man ausgediente Arbeitsplatzrechner in den Serverraum stellt, statt sie bei eBay zu verhökern, und dass sie dort den eigentlichen Datenbankserver von irgendeiner kleineren Datenbank entlasten.

Ausreichend Speicher vorsehen

Ein Datenbankserver profitiert fast immer von mehr Speicher. Ziel sollte es sein, die komplette Datenbank im Speicher halten zu können. Das wird sich zwar nicht erreichen lassen, wenn Sie eine mehrere Tbyte große Bild-Datenbank betreiben, aber eine typische Auftragsdatenbank mit einer Größe zwischen 10 Mbyte und 300 Mbyte sollte schon im RAM betrieben werden.

Schnelles RAM ist dabei zwar grundsätzlich besser als langsames RAM, wichtiger ist dabei jedoch viel RAM: Die Unterschiede zwischen dem Zugriff auf den Arbeitsspeicher und dem Zugriff auf die Festplatte liegen bei einigen Zehnerpotenzen, die Unterschiede beim Arbeitsspeicher sind hier deutlich geringer.

Mehrere Festplatten verwenden

In aller Regel sind die Festplatten das Nadelöhr eines jeden Datenbankservers. Das Problem ist hier nicht nur der beschränkte Datendurchsatz, sondern auch die Position des Schreib-/Lesekopfes, der meist erst dort positioniert werden muss, wo er Daten lesen oder schreiben soll.

Dieses Problem lässt sich dadurch mildern, dass man mehrere Festplatten verwendet. Mehrere Festplatten heißt hier auch wirklich, mehrere Platten in den Server zu schrauben, mit logischen Laufwerken auf derselben Platte verbessern Sie überhaupt nichts.

Ein erster Schritt in die richtige Richtung ist, die Datenbank auf einer anderen Platte zu halten als die Windows-Auslagerungsdatei. Wird nämlich Arbeitsspeicher knapp, wird Windows gezwungen, dessen Inhalt auf die Festplatte auszulagern. Das geschieht meist dann, wenn der Datenbankserver größere Datenmengen

einliest – und finden beide Vorgänge auf derselben Platte statt, dann muss der Schreib-/Lesekopf ständig zwischen Datenbank und Auslagerungsdatei hin- und herspringen, was die Performance erheblich senkt. Bei zwei Platten dagegen wird auf der einen gelesen und auf der anderen geschrieben und alle modernen Festplatten-Controller erlauben einen deutlich höheren Datendurchsatz als die daran angeschlossenen Festplatten.

Bei wichtigen und hoch belasteten Servern wird man dann dazu übergehen, für jede Aufgabe eine eigene Festplatte (oder ein eigenes Festplatten-Array) zu verwenden:

- Für das Betriebssystem und die Exe des Servers wird eine eigene Platte oder sogar ein Spiegelsatz verwendet. Da diese Platte nur beim Starten des Servers in Anspruch genommen wird, kann man sie beispielsweise noch dafür nutzen, die Backups dort abzulegen, um sie von dort mit einem Streamer zu sichern.
- Die Auslagerungsdatei des Betriebssystems bekommt eine eigene Platte. Diese Platte braucht – im Vergleich zu heute üblichen Größen – nicht besonders groß zu werden (wenn das Betriebssystem anfängt, mehrere Gbyte auszulagern, ist ohnehin »alles zu spät«), aber sie sollte sehr schnell sein.
- Die temporären Dateien von InterBase werden auf einer eigenen Platte abgelegt. Temporäre Dateien werden vor allem angelegt, wenn wirklich große Datenmengen sortiert werden sollen und der Arbeitsspeicher dafür nicht ausreicht. Hier sollte man überschlagen, wie groß solche Datenmengen werden können, und dafür jeweils die doppelte Menge an Speicher zur Verfügung stellen.
- Zuletzt noch die Platte für die Datenbank und die Shadows. Bei großen Datenbanken wird man hier ein RAID 5-Array anlegen, um dessen Vorteile bezüglich Speicherplatz, Geschwindigkeit und Ausfallsicherheit nutzen zu können. Bei kleineren Datenbanken sollte ein Shadow reichen.

Derzeit sind IDE/ATAPI-Platten deutlich günstiger als SCSI und es sieht momentan nicht danach aus, als ob sich daran deutlich etwas ändern würde, die Preisstruktur bei den Controllern ist ähnlich. In der mittleren Leistungsklasse würde ich sagen: Lieber zwei IDE-Platten als eine SCSI. Setzt man mehrere Festplatten ein, dann kommt man hier schnell an die Grenze von vier Geräten, so dass ein weiterer Controller erforderlich wird – hier gibt es auch schon Controller, die RAID unterstützen.

Für wirklich leistungsfähige Server wird man jedoch nach wie vor auf SCSI zurückgreifen, vor allem auch wegen der Möglichkeit, mittels Hot-Swap-Platten ein RAID 5-Array aufzubauen.

Prozessor

Der Prozessor ist in den meisten Datenbankservern von untergeordneter Wichtigkeit. Natürlich wird ein System mit schnellerem Prozessor nicht langsamer, aber erfahrungsgemäß investiert man lieber in mehr Speicher als in eine schnellere CPU.

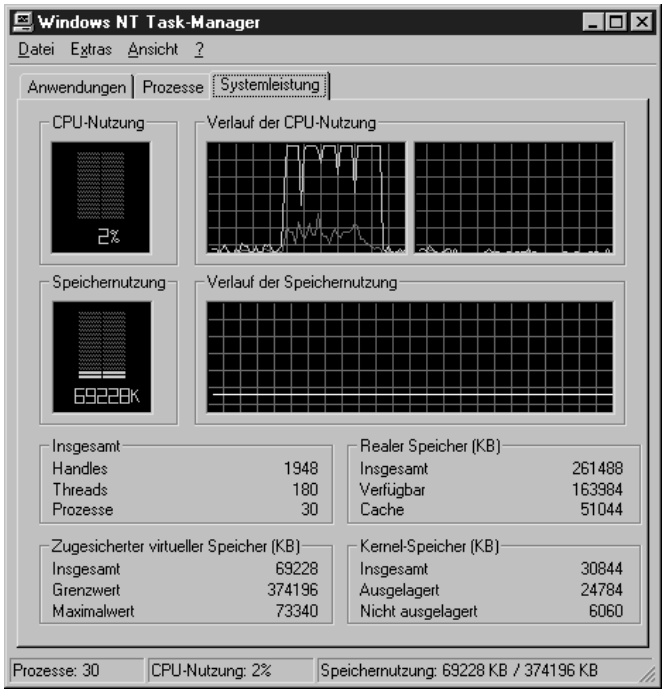


Abbildung 6.1: Ausgelasteter Prozessor

Um zu beurteilen, ob sich eine schnellere CPU lohnen könnte, schauen Sie sich die CPU-Auslastung im Task-Manager an. *Abbildung 6.1* zeigt hier, dass die linke CPU längere Zeit voll ausgelastet ist. Haben Sie längere Zeit solche Situationen, dann sollten Sie über eine schnellere CPU nachdenken. Bewegt sich die Auslastung durchschnittlich eher im unteren Bereich, wird eine schnellere CPU nicht viel nutzen.

Ab InterBase 7 wird ein Mehr-Prozessor-System unterstützt. Bei InterBase-Versionen bis 6.5 kann unter Windows ein Mehr-Prozessor-System sogar nachteilig sein, wenn man es nicht richtig konfiguriert: Hat auf dem Server nur der InterBase-Prozess Arbeit – bei einem dezidierten Datenbankserver nicht ungewöhnlich –, dann wird nur eine CPU belastet. Das Betriebssystem stellt nun unbeschäftigte Prozessoren fest und verlegt nun den Prozess in der Absicht, alle Prozessoren gleichmäßig zu belasten, auf einen anderen Prozessor – was nun an anderer Stelle für eine unbeschäftigte CPU sorgt. Also wird der Prozess ein weiteres Mal verlegt und so weiter ...

Es ist also bis InterBase 6.5 nötig, den Server-Prozess bei Mehr-Prozessor-Systemen auf einen Prozessor »festzunageln«. Dazu muss allerdings InterBase als Programm und nicht als Prozess laufen. Rufen Sie dazu zunächst den InterBase-Manager aus der Systemsteuerung auf:

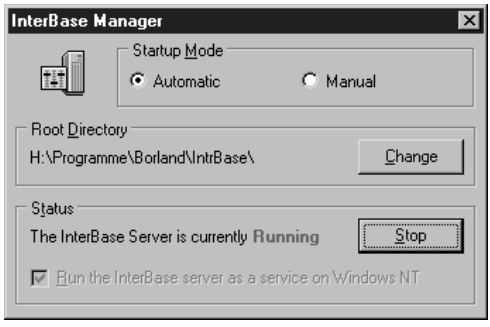


Abbildung 6.2: InterBase als Programm ausführen

Stoppen Sie vorübergehend den Server und deaktivieren Sie dann die Option *Run the InterBase server as a service on Windows NT*. Anschließend können Sie den Server wieder starten.

Nun öffnen Sie den Task-Manager (Alt+Strg+Entf), wählen die Registerseite *Prozesse* und wählen dort *ibserver.exe*.

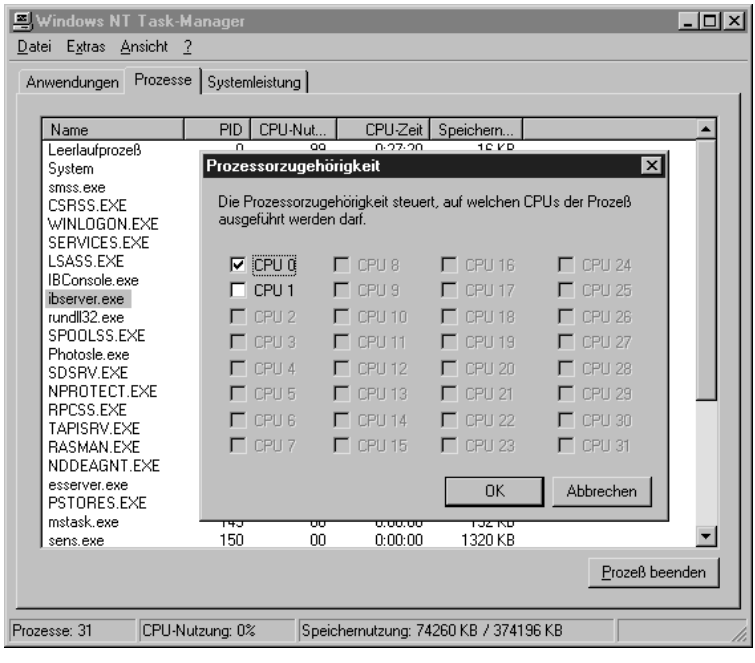


Abbildung 6.3: Prozessorzugehörigkeit festlegen

Über das Kontextmenü (rechte Maustaste) öffnen Sie nun den Dialog zur Einstellung der Prozessorzugehörigkeit. Diese Option steht Ihnen nur auf Mehr-Prozessor-Systemen zur Verfügung – auf Ein-Prozessor-Systemen würde dies auch keinen Sinn machen. Wählen Sie nun einen der möglichen Prozessoren. Wenn Sie bereits einen anderen Prozess auf einen Prozessor festgelegt haben, sollten Sie nun einen anderen verwenden.

Ab InterBase 7 kann dann die Arbeit auf mehrere Prozessoren verteilt werden, InterBase 7 zieht dann erheblichen Nutzen aus einem Mehr-Prozessor-System (wenn auch nicht ganz so viel wie aus ordentlich Speicher ...).

Netzwerk

Werden große Datenmengen übertragen, dann stellt das Netzwerk neben den Festplatten das zweite Nadelöhr dar. Hier kann man mit dem Design der Datenbankanwendung Entlastung schaffen, indem statistische Auswertungen auf dem Server durchgeführt werden statt auf dem Client, so dass nur einige Ergebnisse statt des gesamten Datenbestands übertragen werden müssen.

Was die Netzwerkkarten betrifft, gilt: Je schneller, desto besser, 100 Mbit-Karten sollten es inzwischen schon sein. Verwenden Sie lieber ein Switch statt ein Hub, wenn Sie größere Netzwerke betreiben: Ein Hub verteilt die Daten an alle anderen Rechner, ein Switch stellt sie direkt an den Zielrechner durch.

Stromversorgung

Eine Stromversorgung hat zwar keinen Einfluss auf die Performance, sehr wohl aber auf die Ausfallsicherheit. InterBase ist eigentlich sehr robust, wenn der Server durch einen Stromausfall abstürzt, dann kann man ihn anschließend ohne Wartungsarbeiten wieder in Betrieb nehmen. Allerdings werden die Datenänderungen aus nicht bestätigten Transaktionen verloren sein, ebenso von Datenbankseiten, die noch nicht wieder zurückgeschrieben wurden – mit *forced writes* minimieren Sie diese Gefahr.

Der erste Gedanke sollte hier einer unterbrechungsfreien Spannungs-Versorgung (USV) gelten. Ein solches Gerät enthält einen Akku, mit dessen Hilfe der Betrieb ungefähr eine Viertel Stunde weiter aufrechterhalten werden kann. Das sollte reichen, um kurzfristige Störungen zu überbrücken und bei längeren Unterbrechungen das System geordnet herunterzufahren.

Versuchen Sie, das System an einer möglichst unzugänglichen Stelle anzuschließen, das minimiert die Gefahr, dass jemand mal eben den Server lahm legt, weil er eine freie Steckdose benötigt. Manche Sicherungsautomaten lösen durch den Einschaltstrom eines Staubsaugers aus, deshalb sollte der Stromkreis für den Server eigens abgesichert sein.

Bei größeren Netzwerken steht der Server ohnehin im Serverraum, zu dem niemand außer den Administratoren Zugang hat, vor allem keine Reinigungskräfte. Hier setzt man dann oft auch redundante Netzteile ein. Dies geschieht weniger in Sorge um einen Netzteil-Ausfall als vielmehr in der Absicht, die beiden Stromleitungen auf verschiedene unterbrechungsfreie Spannungsversorgungen und diese auf zwei völlig getrennte Stromkreise legen zu können. Für ganz wichtige Server gibt es Hot-Swap-Netzteile, die sich also im laufenden Betrieb auswechseln lassen.

Datensicherung

Auch ein stabiles Datenbanksystem schützt einen nicht vor einen Festplatten-Crash, vor Feuer oder vor Diebstahl. Von daher sollten Sie in angemessenen Zeiträumen Ihre Daten sichern. Prinzipiell könnten Sie dieses Backup auf einer anderen Festplatte speichern, das hilft Ihnen jedoch nicht gegen Feuer und Diebstahl, nicht, wenn der Festplatten-Controller »Amok läuft« oder ein defektes Netzteil alle Platten zerstört. Ein Backup auf einer anderen Platte hilft also fast gar nicht. (Ist aber immer noch ein wenig besser als ein Backup auf der gleichen Platte oder gar kein Backup.) Um den InterBase-Server möglichst wenig zu belasten, werden Sie das Backup zwar zunächst auf einer anderen Platte speichern, von dort aus aber möglichst umgehend auf ein anderes Medium spielen.

Anderes Medium heißt in fast allen Fällen Streamer, weil nur hier wirklich große Datenmengen zu einem akzeptablen Preis gespeichert werden können.

Dort, wo die Größe des Backup-Files unter 500 Mbyte liegt, kann man auch darüber nachdenken, dies auf eine CD-RW zu speichern. Ein CD-Brenner ist inzwischen Standard und die Rohlinge sind günstiger als Streamer-Bänder – so kommt man in kleinen Firmen an der Anschaffung eines Streamers vorbei.

(Natürlich ist auch dem Autor bekannt, dass auf eine CD mehr als 500 Mbyte passen. Wenn man jedoch bei 500 Mbyte beginnt, auf Streamer umzustellen, ist dieser rechtzeitig beschafft, wenn die CD mal nicht mehr reicht – man kann das Backup zwar auf mehrere Dateien und somit auf mehrere CDs aufteilen, aber 5 Gbyte möchte ich damit nicht mehr sichern müssen.)

Mehr für eine dauerhafte Archivierung als für eine regelmäßige Ausfallsicherung wäre ein MO-Medium geeignet. Hier geht man von einer Haltbarkeit von über 30 Jahren aus und im Gegensatz zu Streamer-Bändern ist die Sache auch sehr unempfindlich. Allerdings reicht die Größe der Medien nicht in den Bereich von Streamer-Bändern.

Backup-Medien liegen fast immer in unmittelbarer Nähe des Servers und auch nur manchmal dort in einem Datenschutzschrank. Wenn das Büro abbrennt, fehlt also nicht nur der Server, sondern auch das Backup und bei Hochwasser oder Diebstahl sieht das nicht grundsätzlich anders aus. Mit entsprechender Organisation und Disziplin könnte man hier Abhilfe schaffen, aber Disziplin ist halt manchmal ein Problem. Von daher könnte man kleinere Backups – vielleicht bis etwa 50 Mbyte – auch im Internet speichern.

Hat die Firma ohnehin einen Web-Hoster, bei dem mehr Speicherplatz im Paket ist, als man jemals brauchen wird, und hat man ohnehin DSL und Flat-Rate, was spricht dann dagegen, im Nachtlauf das Backup auf die Platten des Hosters zu kopieren? Zwischen 1:00 und 5:00 Uhr ist hier in Deutschland die Netzbelastung ohnehin gering, der Datenbankserver ist minimal genutzt, warum nicht die Datei automatisch in eine andere Stadt kopieren?

6.1.1 Hardware-Vorschläge

Wir wollen uns hier drei Vorschläge für sinnvolle Hardware-Kombinationen ansehen, und zwar ein kleines, ein mittleres und ein großes System:

Kleiner Server

Kleine Server werden in der Regel mit IDE-Platten betrieben, die Controller haben meist die Möglichkeit, vier Geräte anzuschließen, was wir nutzen wollen.

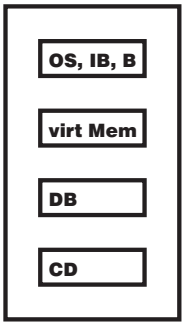


Abbildung 6.4: Kleiner Server

Auf der ersten Platte haben wir das Betriebssystem und die InterBase-Exen. Dorthin wird zunächst auch das Backup gespeichert und das InterBase-Swap-File legen wir dort auch an.

Die zweite Platte ist für die Auslagerungsdatei. Diese Platte muss nicht groß, aber sie sollte sehr schnell sein. Auf der dritten Platte haben wir dann die Datenbank, das vierte Gerät ist das CD-Laufwerk – irgendwie muss man ja die Programme auf den Rechner bringen.

Wie werden die Backups gespeichert? Hier bieten sich drei Möglichkeiten an:

- Das Backup passt auf eine CD und das CD-Laufwerk ist ein Brenner.
- Ein Streamer, der über USB oder die parallele Schnittstelle angeschlossen wird. Steht nur ein IDE-Streamer zur Verfügung, dann müssen wir auf eine der drei Platten verzichten, in der Regel wird die Auslagerungsdatei dann auf der OS-Platte gelassen.
- Backup über ein Netzwerk, entweder ein LAN oder das Internet.

Zum Thema Prozessor und Speicher: 500 MHz und 256 Mbyte scheinen mir derzeit angemessen, wobei der Prozessor-Takt meist von untergeordneter Bedeutung ist.

Mittlerer Server

Der Engpass bei unserem kleinen Server war die Zahl der zulässigen Festplatten. Eine brauchbare Arbeitsteilung war leider ebenso wenig möglich wie ein Shadow. Das wollen wir nun ändern:

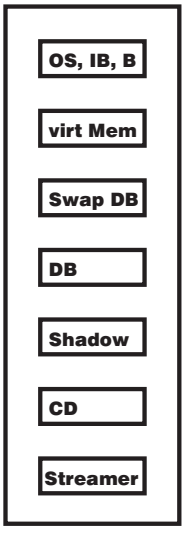


Abbildung 6.5: Mittlerer Server

Wir haben nun sieben Geräte, das bedingt einen weiteren IDE-Controller oder SCSI. Für die Swap-Files von InterBase haben wir nun eine eigene Platte und einen Streamer bringen wir auch noch unter. Darüber hinaus verwenden wir einen Shadow: Damit wird die Datenbank parallel auf zwei Platten gehalten – crasht eine, springt die andere ein. Können wir mehr Platten verwenden, dann könnte man die Datenbank und den Shadow auf jeweils zwei Platten halten.

Den Prozessor würde man dann in der Größenordnung von 1 GHz wählen, wenn man alte Arbeitsplatzrechner einer neuen Verwendung zuführt – bei Neugeräten nimmt man irgendeinen günstigen Prozessor, die sind heute schon alle schneller als 1 GHz. Wichtiger wäre ausreichend Speicher: 512 Mbyte wäre da ein Anfang, wenn das Mainboard so viel verkraftet, wären auch 1 Gbyte nicht übertrieben.

Großer Server

Wenn wir in die Größenordnung von 50 gleichzeitig zugreifenden Benutzern kommen, dann muss auch der Server ordentlich Reserven haben.

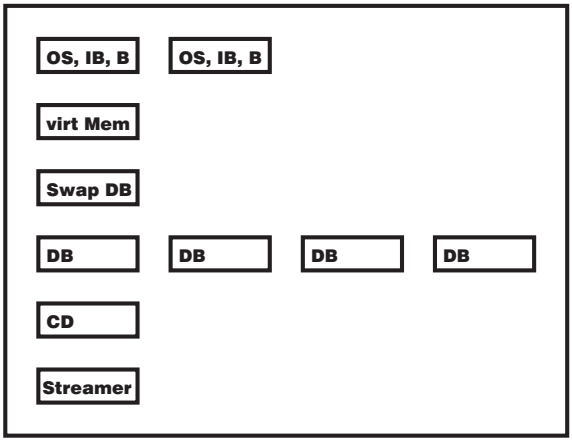


Abbildung 6.6: Großer Server

Hier ist dann nicht nur Geschwindigkeit ein Thema, sondern auch Ausfallsicherheit. Von daher packen wir das Betriebssystem und die Programme auf einen Spiegelsatz und für die Datenbank legen wir ein Hot-Swap-RAID 5-Array an. Hier denkt man dann auch nicht mehr über Software-RAID-Lösungen nach, sondern verwendet einen Hardware-RAID-Controller mit möglichst viel Cache.

Mit einem Mehr-Prozessor-System sorgt man dafür, dass das System auch dann »geschmeidig« läuft, wenn ein User umfangreiche statistische Daten ermittelt. Und über den Vorteil von viel Speicher habe ich mich jetzt schon oft genug ausgelassen.

6.2 Das Betriebssystem

Der InterBase-Server ist für mehrere Betriebssysteme erhältlich und es ist auch kein Problem, Client und Server unter verschiedenen Systemen zu betreiben. Noch ist Windows das wichtigste Betriebssystem, und da sich der Autor mit anderen Systemen auch nur wenig auskennt, wollen wir uns darauf beschränken – die meisten Linux- und Unix-Administratoren wissen ohnehin, was sie zu tun haben.

Unter Windows 9x läuft InterBase immer als Programm, unter Windows NT läuft es normalerweise als Systemdienst. Nur dann, wenn Sie InterBase bis 6.5 auf einem Mehr-Prozessor-System betreiben, sollten Sie es als Programm ausführen, damit Sie die Prozessorzugehörigkeit festlegen können.

Arbeiten Sie unter Windows NT, brauchen Sie dafür keine NT-Server-Lizenz, Workstation reicht völlig. Unter NT Server haben Sie jedoch die Möglichkeit, den Server als Datei- und Drucker-Server oder als Anwendungsserver zu optimieren (SYTEMSTEUERUNG | NETZWERK | DIENSTE | SERVER) – wählen Sie *Anwendungsserver*.

Wenn Sie InterBase als Service betreiben, sollten Sie sich nur als Anwender anmelden, wenn Sie auf dem System arbeiten wollen, weil dies zusätzliche Ressourcen belegt.

Zur Schonung der System-Ressourcen sollten Sie auch keinen Bildschirmschoner verwenden, vor allem keinen OpenGL-Bildschirmschoner. Wenn der Rechner ohnehin im Server-Raum steht, dann kann man den Bildschirm einfach ausschalten – das spart auch Strom. Wenn es unbedingt ein Bildschirmschoner sein muss, dann empfiehlt Borland *Marquee*.

Systemwiederherstellung und XP

InterBase läuft auch unter Windows XP, dort besteht jedoch die Gefahr, dass der erste Zugriff auf eine Datei ungewohnt lange dauert und mit heftigen Aktivitäten der Festplatte verbunden ist. Schuld daran ist eine Technik namens Systemwiederherstellung, welche von Dateien, die geöffnet werden, erst einmal eine vollständige Kopie anlegt – bei großen Datenbankdateien dauert das halt ein wenig.

Die Endungen der Dateien, von denen eine Kopie erstellt wird, finden Sie in der Datei *C:\WINDOWS\system32\Restore\filelist.xml*. Es handelt sich dabei um eine versteckte Datei, gegebenenfalls müssen Sie die Explorer-Optionen so abändern, dass solche angezeigt werden. Hier finden Sie die »alte« Endung *gdb*, nicht jedoch die seit InterBase 7 übliche Endung *ib* – dort stellt sich das Problem nicht.

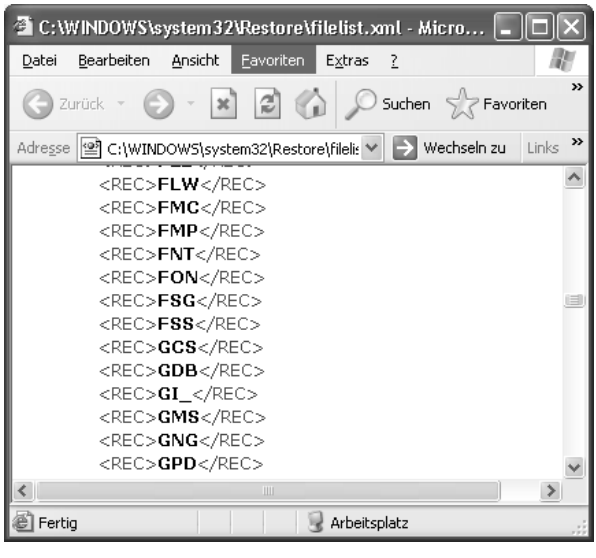


Abbildung 6.7: Endungen für die Systemwiederherstellung

Sie können in dieser Datei die Zeile *<REC>GDB</REC>* löschen, um Datenbankdateien von der Systemwiederherstellung auszunehmen. Alternativ können Sie die Systemwiederherstellung unter SYSTEMSTEUERUNG|SYSTEM|SYSTEMWIEDERHERSTELLUNG gänzlich deaktivieren:

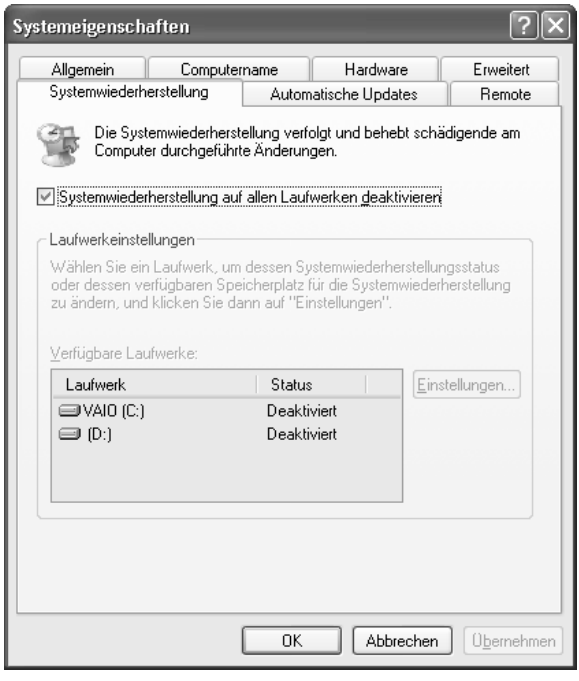


Abbildung 6.8: Systemwiederherstellung vollständig deaktivieren

6.3 Der InterBase-Server

Bei der Installation des InterBase-Servers können Sie nicht viel falsch machen. Installieren Sie den Server im vorgesehenen Verzeichnis, ein Client wird dabei gleich automatisch aufgespielt.

Anwendung oder Service

Per Voreinstellung werden unter NT der InterBase-Server und der Guardian als Service betrieben. Der Guardian ist der Wächter, der auf den Server aufpasst. Läuft aus irgendwelchen Gründen der Server nicht mehr, dann wird er vom Guardian neu gestartet.

(Das mag nun den Verdacht erregen, dass es mit der Stabilität des Servers nicht zum Besten bestellt ist. Dieser Verdacht ist zumindest so lange unbegründet, wie der InterBase-Server alleine läuft. Nun hat man aber die Möglichkeit, für so genannte *user defined functions* eigene DLLs einzubinden. Zumindest während der Entwicklungsphase solcher UDFs häufen sich nach meiner Erfahrung die Server-Abstürze.)

Mit dem *InterBase-Manager* können Sie einstellen, ob der Server automatisch gestartet werden soll oder manuell und ob er als Service läuft oder als Anwendung. Den *InterBase-Manager* finden Sie in der Systemsteuerung.

Abbildung 6.9: Der InterBase Manager

Normalerweise läuft InterBase als Service und erledigt brav im Hintergrund seine Arbeit. Sie können InterBase jedoch auch als Anwendung betreiben – das ist beispielsweise nötig, wenn Sie die Prozessorzugehörigkeit festlegen wollen. Vorher sollten Sie jedoch den Service beenden.

Der InterBase-Server wird nicht direkt gestartet, sondern man startet den *Guardian*. Dieser stellt fest, dass der Server nicht läuft, und startet diesen. Soll InterBase als Anwendung betrieben werden, dann ist beim Start des Guardians ein *-a* anzuhängen.

```
bin\ibguard.exe -a
```

In der Taskbar-Notification-Area (also rechts unten) wird der Guardian nun durch ein kleines Icon repräsentiert:

Abbildung 6.10: InterBase als Programm

Über das Kontextmenü können Sie nun nicht nur den Server herunterfahren, sondern sich auch die aktuellen Eigenschaftswerte ansehen. Diese sind jedoch nur mäßig interessant.

Die Eigenschaften

Vom Server-Manager aus können Sie die Eigenschaften vom Server und vom Guardian einsehen. Beim Guardian interessiert allenfalls, wie oft dieser den Server neu gestartet hat.

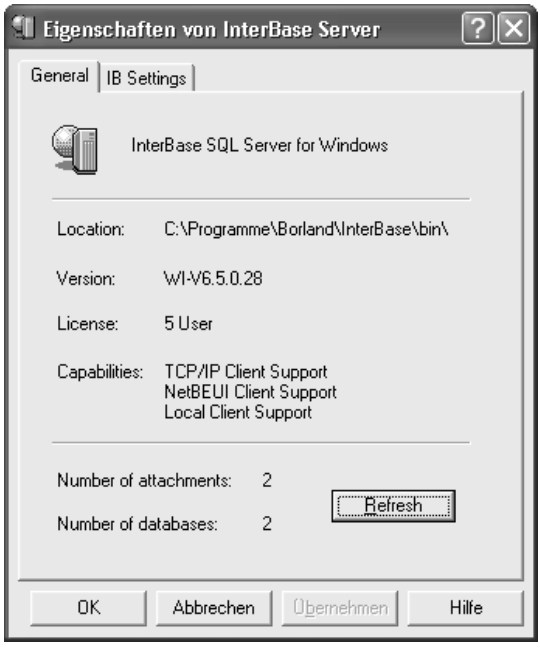


Abbildung 6.11: Eigenschaften des Servers

Auf der Registerseite *General* der Server-Eigenschaften sehen Sie vor allem, wie viele Datenbanken gerade belegt sind und für wie viele User Lizenzen vorliegen.

Auf der Registerseite *IBSettings* können Sie einstellen, wie viele Seiten pro Datenbank als Cache verwendet werden sollen. Der vorgegebene Wert ist 2048, bei einer Seitengröße von 4 kbyte wäre das eine Gesamtgröße von 8 Mbyte. Bei wichtigen Datenbanken sollte dieser Wert vergrößert werden, aber nicht an dieser Stelle, weil sich das auf alle Datenbanken auswirken würde. Wir werden noch sehen, wie man dies für jede Datenbank einzeln spezifizieren kann.

Die Log-Datei

Im InterBase-Verzeichnis finden Sie eine Datei namens *interbase.log*, in der alles Ungewöhnliche protokolliert wird. Wenn Client oder Server Probleme machen, dann sollten Sie dort mal einen Blick hineinwerfen:

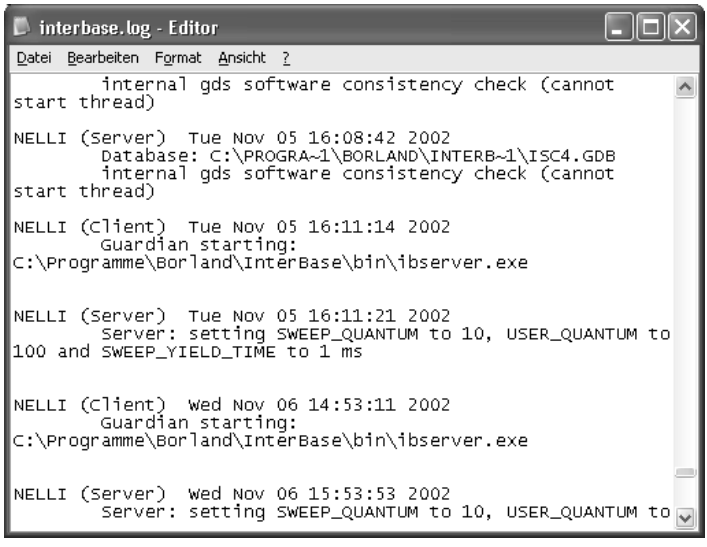


Abbildung 6.12: Die Log-Datei

Die Konfigurationsdatei

Ebenfalls im InterBase-Verzeichnis finden Sie die Datei *ibconfig* (ohne Endung), die Sie mit einem Texteditor öffnen können.

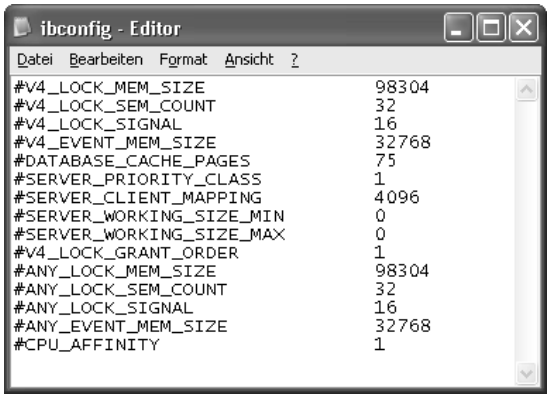


Abbildung 6.13: Die Konfiguationsdatei

Was die einzelnen Einträge bedeuten, können Sie im *Operation Guide* nachschlagen, wir wollen uns nur die wichtigsten Einträge ansehen:

Mit SERVER_WORKING_SIZE_MIN und SERVER_WORKING_SIZE_MAX können Sie festlegen, wie viel Speicher der Server minimal und wie viel er maximal beansprucht. Normalerweise haben diese Einträge den Wert Null, so dass darüber das Betriebssystem entscheidet – und diese Entscheidung ist meist optimal.

Nehmen wir einmal an, Sie wollen möglichst viel Speicher dem InterBase-Server zur Verfügung stellen und weisen dem Minimalwert einen recht hohen Wert zu, dann steht InterBase stets eine große Menge an Arbeitsspeicher zur Verfügung, was sich zunächst einmal gut anhört. Braucht nun aber ein anderer Prozess – und sei es das Betriebssystem – Speicher, dann muss dieser gegebenenfalls über die Auslagerungsdatei beschafft werden, obwohl InterBase gar nicht allen reservierten Speicher aktuell benötigt. Das Auslagern kann dann mehr Zeit in Anspruch nehmen, als wenn von InterBase reservierte Seiten ausgelagert worden wären.

Der langen Rede kurzer Sinn: Führen Sie hier nur Änderungen durch, wenn Sie auch die Folgen abschätzen können.

Platz für Sortier-Dateien

Eine Ergänzung kann jedoch in der Konfigurationsdatei sinnvoll sein, nämlich die von TMP_DIRECTORY. Wenn InterBase große Datenmengen sortieren muss, dann ist dafür der Arbeitsspeicher mitunter zu klein, zumal das Ergebnis ja noch eine Zeit lang gehalten werden muss, bis die Datenmenge geschlossen wird. Für solche Zwecke legt InterBase Sortier-Dateien an, und zwar normalerweise in dem Verzeichnis, das mit der Umgebungsvariablen INTERBASE_TMP spezifiziert ist. (Umgebungsvariablen finden Sie in der Systemsteuerung unter *System*.)

Gibt es diese Umgebungsvariable nicht – bei der Installation wird sie nicht angelegt –, dann greift InterBase auf die Umgebungsvariable TMP zurück, im Falle deren Fehlens wird das Verzeichnis *c:\TEMP* verwendet.

Solange Sie keine eigene Platte für diese Dateien vorsehen, können Sie alles so lassen, wie es ist. Wenn Sie jedoch eine eigene Platte dafür verwenden, dann sollten Sie entweder die Umgebungsvariable INTERBASE_TMP setzen oder in der Konfigurationsdatei den folgenden Eintrag ergänzen:

```
INTERBASE_TMP 1000000000 "E:\"
```

Die Größe spezifizieren Sie in Byte, der Pfad muss in doppelten Anführungszeichen stehen. Mit mehreren Einträgen können Sie auch verschiedene Pfade angeben und somit auch mehrere Platten für Sortier-Dateien vorsehen.

Borland spricht in der Dokumentation davon, dass Sortier-Dateien in Extremfällen größer als die Datenbank werden können. Sie sollten deshalb nicht zu sparsam mit dem Platz sein.

6.4 Die Datenbank

Wenn Sie bislang den Eindruck gewonnen haben, dass Sie nicht viel einstellen können und noch viel weniger einstellen müssen, dann liegen Sie durchaus richtig. Zum einen liegt das daran, dass InterBase sehr pflegeleicht ist, zum anderen werden die wesentlichen Einstellungen auf Datenbank-Ebene vorgenommen.

Diese Einstellungen werden für gewöhnlich mit *IBConsole* oder einem anderen Tool vorgenommen, wir werden sie im Rahmen von *IBConsole* im nächsten Kapitel besprechen.

Natürlich können Sie auch alles per Kommandozeilen-Aktion einstellen, Näheres dazu finden Sie in den Handbüchern. Einige Sachen lassen sich jedoch ausschließlich per Kommandozeile einstellen.

7 IBConsole

Zur Verwaltung von InterBase-Servern gibt es das Programm IBConsole. Von dort aus können Sie Benutzer anlegen, Backups ziehen (und natürlich auch wieder einspielen) oder *Interactive SQL* aufrufen.

IBConsole wird mit InterBase zusammen ausgeliefert. Es gibt einige andere, teils kostenpflichtige, Verwaltungstools für InterBase, beispielsweise das in Kapitel 13 besprochene IBExpert. Mit solchen Tools kann man viele Verwaltungsaufgaben komfortabler erledigen. Für viele Benutzer wird jedoch IBConsole völlig ausreichend sein.

Für jede registrierte Datenbank können Sie die Daten und die Metadaten ansehen – Letztere sowohl als SQL-Script als auch in Tabellenform.

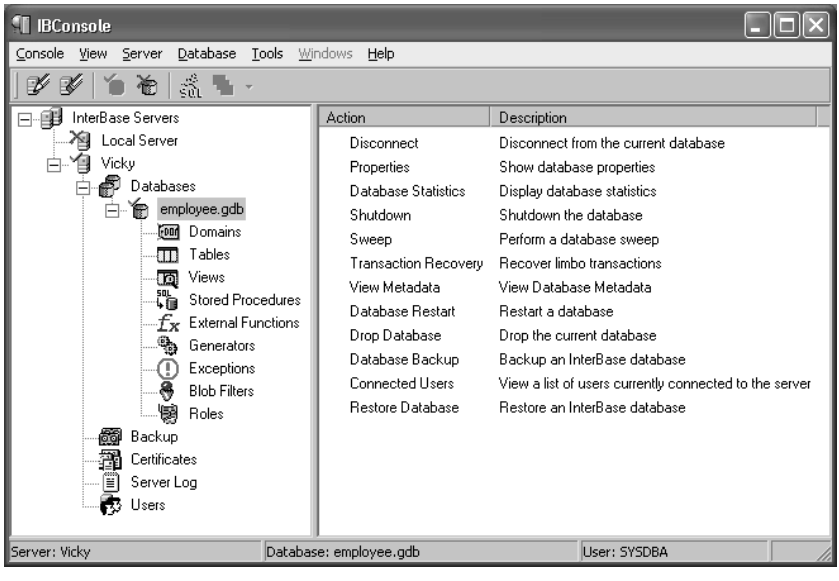


Abbildung 7.1: IBConsole

Per Voreinstellung ist lediglich der lokale Server registriert, also der Server, der auf demselben Rechner läuft. Sie können sich jedoch mit beliebigen anderen Servern verbinden.

7.1 Registrieren einer Server-Verbindung

Wir wollen nun auf eine Datenbank zugreifen, die auf einem anderen Rechner liegt. Dieser muss natürlich über ein funktionierendes Netzwerk verbunden sein. Hier im Beispiel soll die Verbindung über eine TCP/IP-Verbindung eingerichtet werden.

Ermitteln der IP-Adresse

Zunächst benötigen wir die IP-Adresse des Servers. Dazu rufen wir in der *Systemsteuerung* die *Netzwerkverbindungen* auf. Hier interessiert uns das TCP/IP-Protokoll. Ist dies nicht vorhanden, dann muss es nachinstalliert werden.

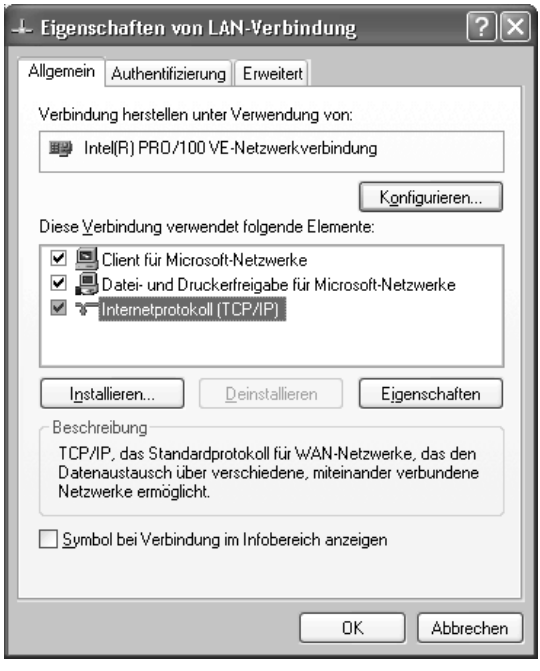


Abbildung 7.2: Die installierten Netzwerk-Protokolle

Nun klicken wir auf *Eigenschaften*:

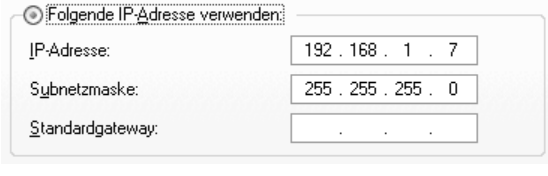


Abbildung 7.3: Die IP-Adresse

Je nach Windows-Version sieht das immer ein wenig anders aus, aber irgendwo ist die IP-Adresse zu finden, die wir uns notieren.

Einrichten eines Host-Namens

Die folgenden Schritte erfolgen nun auf dem Client. Dort öffnen wir die Datei *Hosts*, die irgendwo im Windows-Verzeichnis liegt und mit dem (Text-)Editor geöffnet werden kann. Dort werden für die IP-Adressen Host-Namen vergeben.

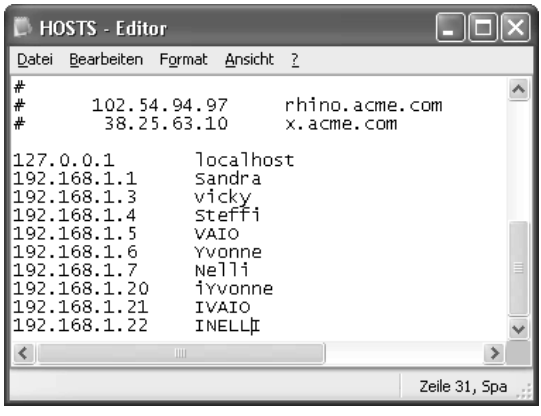


Abbildung 7.4: Vergeben eines Host-Namens

Hier gibt es zwei Möglichkeiten: Entweder ist für die IP-Adresse des Servers schon ein Host-Name vergeben, dann muss dieser lediglich notiert werden. Anderenfalls muss hier ein neuer Eintrag vorgenommen werden.

Um sicherzugehen, dass die Verbindung steht, wechseln wir in die Eingabeaufforderung und setzen einen Ping ab (das Kommando *ping*, ein Leerzeichen und dann der Host-Name).

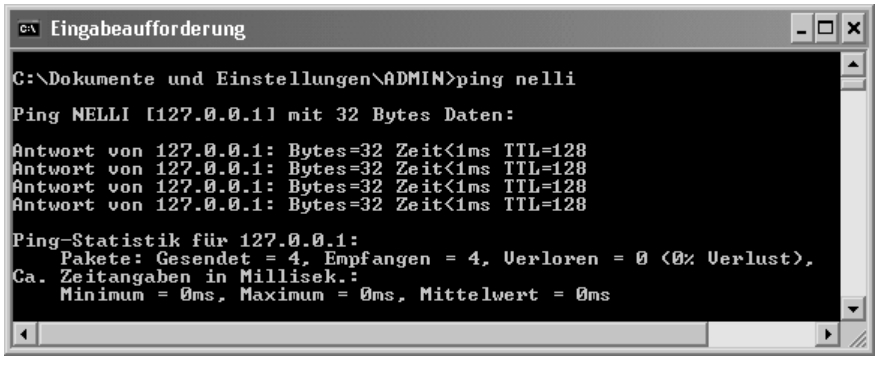


Abbildung 7.5: Überprüfung der Verbindung mit einem Ping

Wenn die Verbindung steht, dann sollte der Server antworten. Wenn er das nicht tut, dann ist irgendwo ein Fehler (Netzwerkkabel steckt nicht, Netzwerk ist nicht oder falsch eingerichtet, oder – wird zunehmend häufiger – eine Firewall verhindert die Verbindungsaufnahme).

Server registrieren

Starten Sie nun *IBConsole* und rufen Sie dort aus dem Kontextmenü des Root-Knotens den Menüpunkt *Register* auf.

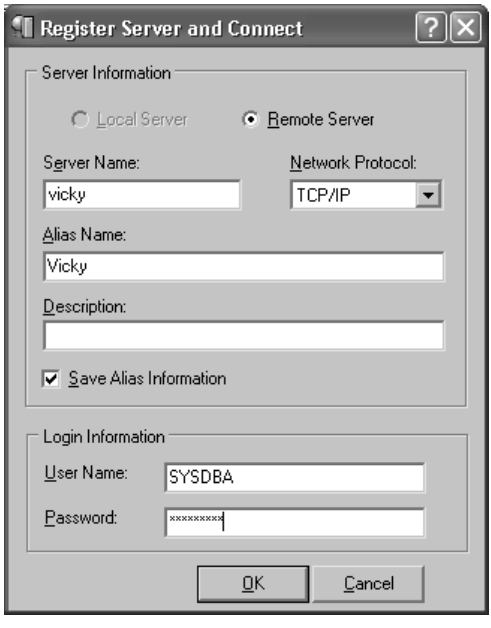


Abbildung 7.6: Den Server registrieren

Als *ServerName* müssen Sie den Host-Namen eingeben, auf dem der Server läuft, als *Network Protocol* wählen wir TCP/IP. Der Alias-Name kann frei vergeben werden. *User Name* ist – solange nichts anderes eingestellt wurde – *SYSDBA*, *Passwort* ist *masterkey*.

Verbindung testen

Nun rufen wir aus dem Kontextmenü *Communication Diagnostics* auf und wechseln zunächst auf die Registerseite *TCP/IP*. Hier wählen wir den Host-Namen und als *Service* den Wert *gds_db*. Schlägt die Verbindung fehl, dann versuchen Sie statt *gds_db* mal die Nummer 3050.

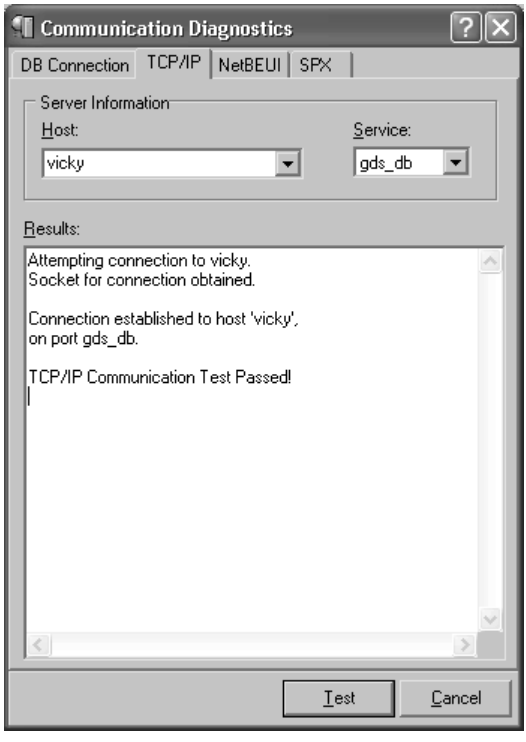


Abbildung 7.7: Testen der Verbindung

Funktioniert 3050, nicht jedoch *gds_db*, dann fehlt der Eintrag in der Datei *Services*, die auch irgendwo im Windows-Verzeichnis liegt. Hier wird der Servicename (*gds_db*) der Port-Nummer (3050) zugeordnet.

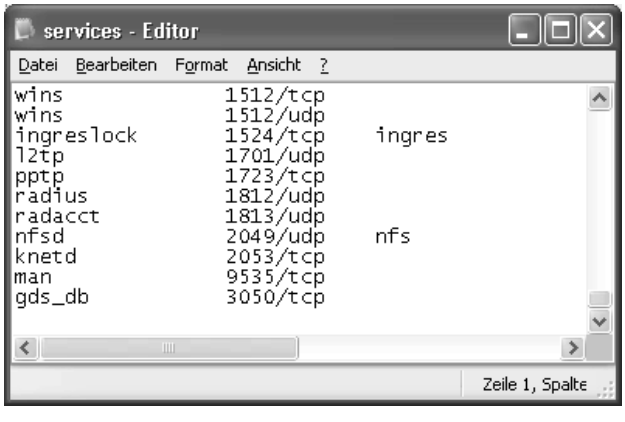


Abbildung 7.8: Die Liste der Services

Geht auch 3050 nicht, aber der Ping, dann dürfte irgendwo eine Firewall sitzen, die den Port 3050 sperrt – dies wäre dann zu ändern.

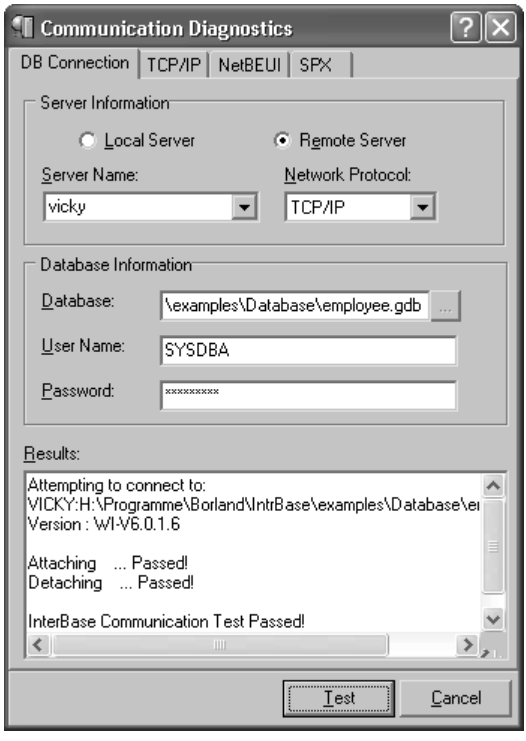


Abbildung 7.9: Verbindung zur Datenbank

Nun gehen wir auf die Registerseite *DB Connection*. Hier testen wir nicht die Verbindung zu einem Server, sondern gleich zu einer Datenbank – also brauchen wir zusätzlich den Dateinamen der Datenbank sowie die Login-Informationen.

Registrieren der Datenbank

Ist die Verbindung erfolgreich getestet, dann ist es an der Zeit, die Datenbank zu registrieren. Dazu wird aus dem Kontextmenü des betreffenden Servers der Menüpunkt *Register* aufgerufen. Hier geben Sie zunächst den Dateinamen ein. Da ein solcher Dateiname ziemlich lang werden kann, können Sie einen Alias-Namen vergeben, der stattdessen in der Baumansicht angezeigt wird.

Die Login-Informationen werden gespeichert, so dass sie später nicht noch mal eingegeben werden müssen. Darüber hinaus sollte der Zeichensatz angegeben werden, der in der Datenbank verwendet wird.

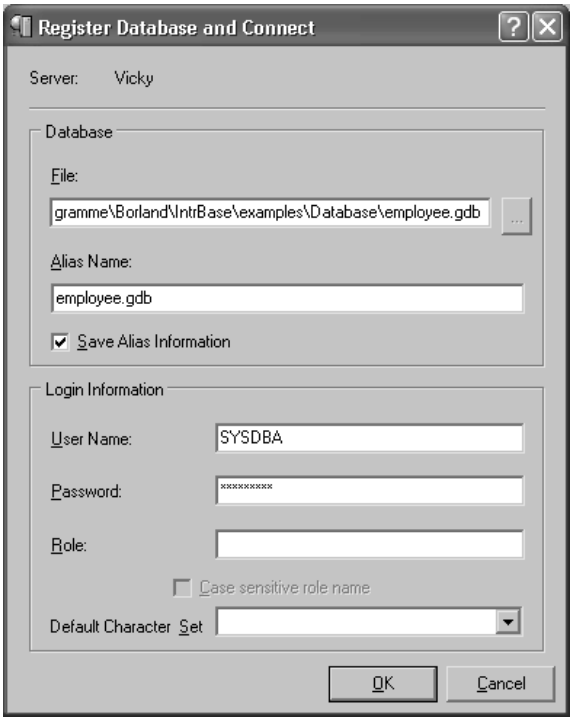


Abbildung 7.10: Registrieren der Datenbank

7.2 Verwaltung des Servers

Jeder registrierte Server kann von IBConsole aus verwaltet werden. Dabei interessieren vor allem die Datenbanken, damit werden wir uns in Kapitel 7.3 beschäftigen. Darüber hinaus können Benutzer angelegt, Zertifikate registriert, Backups gezogen und wieder eingespielt und kann das Logfile des Servers eingesehen werden.

Benutzer anlegen

Per Voreinstellung gibt es einen Benutzer *SYSDBA* mit dem Passwort *masterkey*. Für produktiv arbeitende Datenbanken sollten Sie zumindest das Passwort ändern, damit ein Minimum an Sicherheit gewährleistet ist.

Wie viele weitere Benutzer Sie anlegen, ist Ermessenssache. Vorgesehen wäre, dass Sie für jede Person, die mit der Datenbank oder einer Datenbankanwendung arbeitet, einen Benutzernamen und ein Passwort vergeben. In größeren Firmen ist dies sicher ganz lustig, wenn man für jeden neuen Mitarbeiter zunächst bei mehreren Servern Benutzer anlegen muss.

Die Alternative wäre, dass sich die Datenbankanwendung mit einem eigenen Passwort anmeldet, das der Mitarbeiter auch nie zu Gesicht bekommt. Er könnte dann nicht mit einem entsprechenden Tool direkt auf die Datenbank zugreifen und so die Geschäftslogik vor STORED PROCEDURES und Datenbankanwendung umgehen. Die Verwaltung der Rechte muss man dann jedoch selbst implementieren, wodurch man doch etwas flexibler auf die aktuellen Bedürfnisse Rücksicht nehmen kann, als wenn man die Benutzerverwaltung von InterBase dafür verwendet.

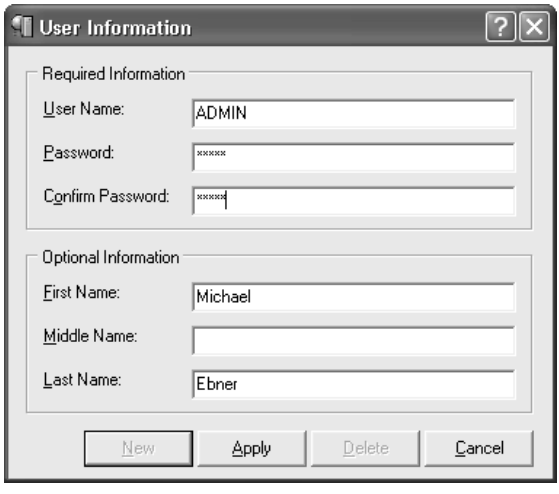


Abbildung 7.11: Einen neuen Benutzer anlegen

Um einen neuen Benutzer anzulegen, wählen Sie aus dem Kontextmenü ADD USER. Zusätzlich zu Benutzernamen und Passwort können Sie auch noch den Klartextnamen des Benutzers eingeben, *Middle Name* könnte man für die Abteilung missbrauchen.

Die Benutzer-Informationen können Sie auch ändern, allerdings nicht den Benutzernamen. Sie können auch Benutzer löschen.

Die Benutzerinformationen liegen bis Version 6.5 in der Datenbank *isc4.gdb* und ab Version 7 in *admin.ib*, jeweils in der Tabelle *users*, das Passwort wenigstens verschlüsselt. Beide Datenbanken liegen im InterBase-Verzeichnis, bei einer Standard-Installation also auf *C:\Programme\Borland\InterBase*.

Zertifikate

Die Verwaltung der Lizenzen erfolgt mit Zertifikaten, also mit einer Liste von IDs und den dazugehörenden Schlüsseln. Diese Verfahrensweise kommt insbesondere Systemhäusern entgegen, die sich Benutzerlizenzen in großer Stückzahl kaufen und flexibel auf die Server ihrer Kunden aufteilen können.

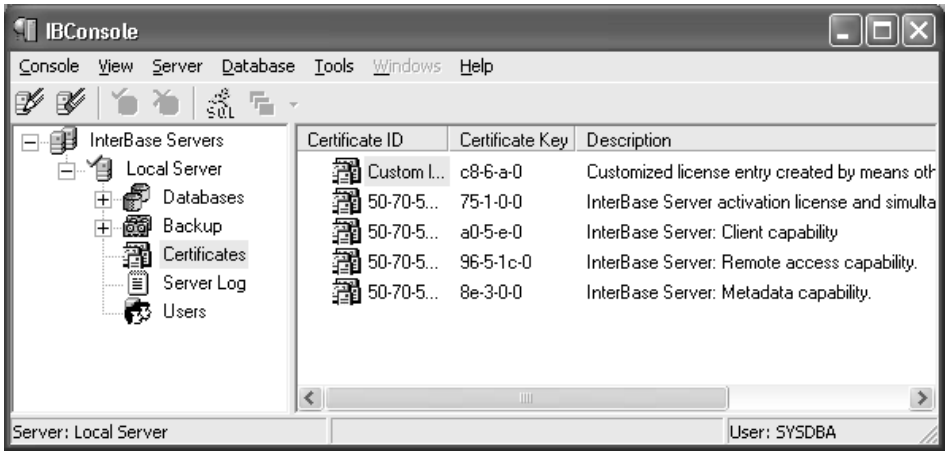


Abbildung 7.12: Die Liste der Zertifikate

Über die Zertifikate lässt sich auch ein wenig die Sicherheit eines Servers steuern. Entfernt man beispielsweise die *Metadata capibility*, so können keine Metadaten mehr geändert werden. Dann allerdings kann noch nicht einmal ein Index neu aufgebaut werden, indem man ihn zunächst inaktiv und dann wieder aktiv setzt. Allein die Metdadaten-Änderungen im Zusammenhang mit dem Einspielen eines Backups sind noch möglich.

Backups

Wenn Datenbanken im industriellen Umfeld produktiv arbeiten, dann hat der Datenbestand oft einen Wert, der den für Hard- und Software um ein Vielfaches übersteigt. Ein Totalverlust ihrer Daten dürfte die meisten Firmen ruinieren. Deshalb muss von der Datenbank regelmäßig eine Sicherungskopie gezogen werden.

Wählen Sie zu diesem Zweck aus dem Kontextmenü BACKUP/RESTORE | BACKUP.

Zunächst einmal wählen Sie die Datenbank, die gesichert werden soll. Das Backup kann auf denselben Rechner oder auf einen anderen Rechner gespielt werden.

Darüber hinaus haben Sie die Möglichkeit, das Backup auf mehrere Dateien aufzuteilen. Das brauchen Sie nicht nur, wenn die Speicherung auf vergleichsweise kleine Medien gespielt werden soll, beispielsweise auf CDR oder MOD. Bei großen Datenbanken kann auch die Backup-Datei größer als 2 beziehungsweise 4 Gbyte werden und übersteigt damit die Grenze der 32-Bit-Dateisysteme.

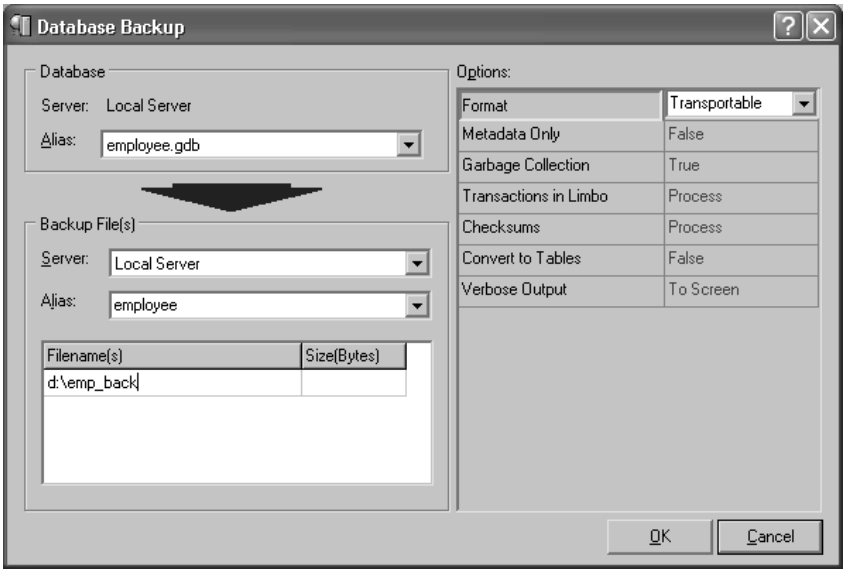


Abbildung 7.13: Ein Backup erstellen

Zusätzlich können Sie nun einige Optionen einstellen.

- Wenn Sie das transportable Format verwenden, dann kann das Backup auch auf einem Server eingespielt werden, der unter einem anderen Betriebssystem läuft – dies ist auch die Methode, um die Daten von einem Server zu einem anderen zu übertragen.
- Wenn Sie die Metadaten extrahieren wollen, dann setzen Sie *Metadata Only* auf *true*. Zum Extrahieren der Metadaten ließe sich auch ein SQL-Script generieren.
- Wird ein Datensatz mit DELETE gelöscht oder mit UPDATE geändert, dann kann die alte Version nicht gleich von der Festplatte entfernt werden, weil ja noch laufende Transaktionen diese Daten benötigen könnten. Damit diese veralteten Datensätze nicht ewig auf der Platte verbleiben, führt der Server hin und wieder eine *garbage collection*, eine »Müllabfuhr«, durch und schaut, was nun entfernt werden kann.

  Für gewöhnlich ist es sinnvoll, dies auch im Rahmen eines Backups durchzuführen, damit veraltete Datensätze nicht mitgesichert werden. Belassen Sie deshalb *garbage collection* auf *true*.
- *Transactions in Limbo*, also Transaktionen in der Schwebe, können durch Systemfehler oder Fehler beim 2-Phasen-Commit auftreten. Um solche Transaktionen und die daraus resultierenden Datenänderungen zu ignorieren, setzen Sie diese Option auf *Ignore*.

  Mit *Transactions in Limbo* sind keine gewöhnlichen Transaktionen gemeint, die während des Backups einfach noch nicht beendet wurden. Von daher treten sie nicht besonders häufig auf.

- Während des Backups werden die Quersummen der Datenseiten überprüft. Tritt hier ein Fehler auf, dann kann auch kein Backup gezogen werden. Soll dies trotzdem geschehen, dann ist *Checksums* auf *Ignore* zu setzen. Sie sollten dann entsprechend misstrauisch bezüglich des Inhalts der Datenbank sein, wenn Sie ein solches Backup mittels Restore wieder einspielen.
- Sie können Tabellen auch in externen Dateien halten. Damit diese bei einem Backup mitgespeichert werden, müssen Sie die Option *Convert To Tables* aktivieren.
- Wenn *VerboseOutput* den Wert *To File* oder *To Screen* hat, dann erhalten Sie ein detailliertes Protokoll des Backups.

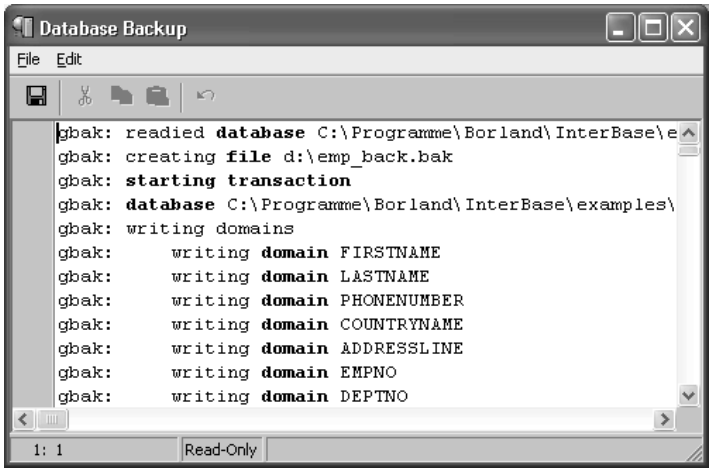


Abbildung 7.14: Das Protokoll des Backups

Die Liste der Backups

IBConsole führt eine Liste aller Backups, die über IBConsole erstellt wurden:

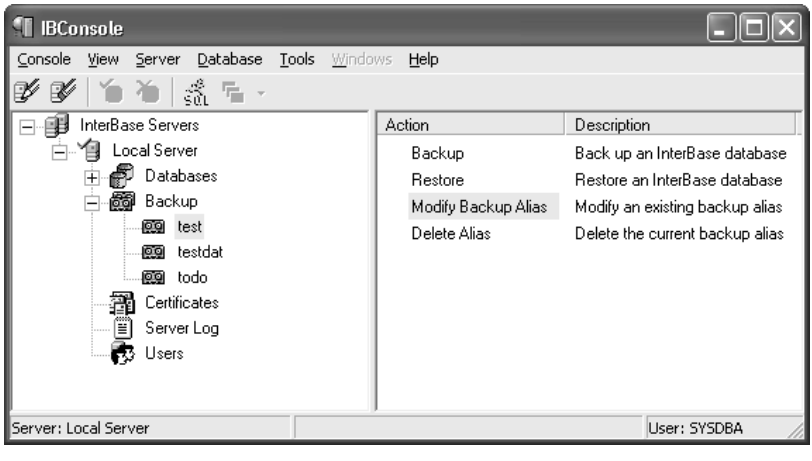


Abbildung 7.15: Liste der Backups

Dies hat den Vorteil, dass Sie nicht lange suchen müssen, wenn Sie ein Backup wieder einspielen wollen. Zudem ersparen Sie sich viel Tipparbeit, wenn Sie Backups auf viele Dateien aufteilen.

Restore

Zum Wiederherstellen verwenden Sie BACKUP/RESTORE | RESTORE aus dem Kontextmenü der Datenbank oder RESTORE aus dem Kontextmenü des Backups.

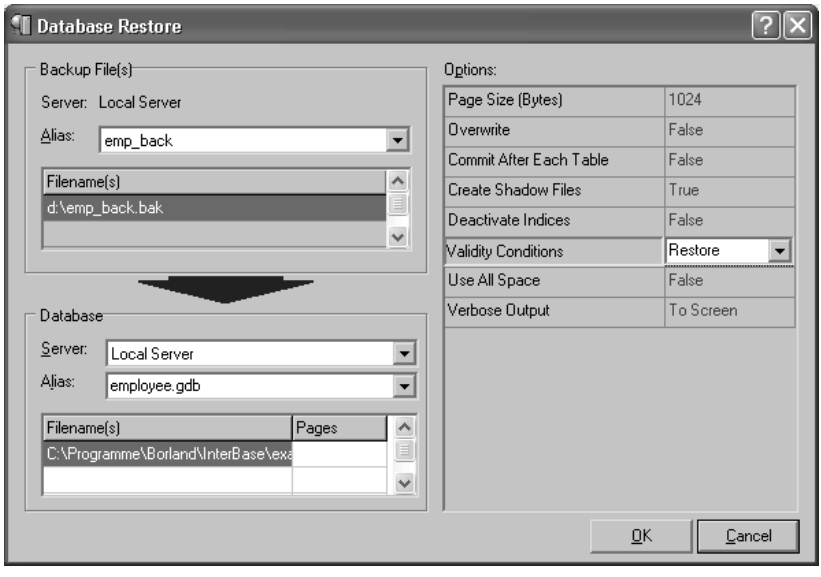


Abbildung 7.16: Restore

Hat das Backup einen Alias, dann müssen Sie sich nicht mit der Eingabe der Backup-Dateien und Datenbankdateien aufhalten. Gerade bei sehr großen Datenbanken können das doch mehrere hundert Dateien sein und die Ruhe und Gelassenheit sind gerade beim Einspielen eines Backups oft limitiert ...

Auch beim Restore können Sie wieder eine ganze Reihe von Optionen einstellen:

- Die *PageSize* ist die Größe der einzelnen Datenbankseiten, 4096 ist hier fast immer der ideale Wert. Wollen Sie die Seitengröße während des Betriebs einer Datenbank verändern, so müssen Sie den Weg über Backup und Restore gehen.
- Soll eine bestehende Datenbank überschrieben werden, dann setzen Sie *Overwrite* auf *true*. Während des Restores dürfen keine Benutzer mit der Datenbank verbunden sein.
- Ein Restore läuft für gewöhnlich in einer großen Transaktion. Geht die Sache schief, wird der Vorgang komplett zurückgenommen. Macht das Einspielen eines Backups Probleme, kann es hilfreich sein, wenigstens einzelne Tabellen wiederherzustellen. Dafür wählt man *Commit After Each Table*.

- Shadows werden bei der Definition der Datenbank mit angelegt. Sollen Sie mit wiederhergestellt werden, dann ist *Create Shadow Files* auf *True* zu belassen. Soll jedoch eine Datenbank mit einem Backup auf mehrere Dateien aufgeteilt werden, dann möchte man dies für gewöhnlich auch mit den Shadows tun. Sie sollten dann nicht während des Restores wiederhergestellt werden, sondern im Anschluss daran manuell.
- Über vorübergehend deaktivierte eindeutige Indizes (Schlüssel) können Dubletten in die Datenbank kommen. Versucht man nun, eine solche Datenbank wiederherzustellen, dann scheitert das Einfügen der Dubletten am eindeutigen Index. Dies kann man mit *Deactive Indices* verhindern.

  Aus Performance-Gründen wäre es keine schlechte Idee, alle Indizes im Anschluss an das Restore neu aufzubauen. Wenn das ohnehin erfolgen soll, dann besteht auch kein Anlass, den Server während des Restores mit der Wartung der Indizes zu belasten.
- Wenn Gültigkeitsprüfungen während des Betriebs einer Datenbank hinzugefügt werden, dann müssen davor eingefügte Datensätze nicht zwingend diesen Bedingungen entsprechen. Weil aber bei einem Backup zunächst die Metadaten erstellt und dann die Daten eingefügt werden, kommt es hier zu Problemen. Um diese zu umgehen, setzt man *Validity Conditions* auf *Ignore*.
- Normalerweise füllt InterBase bei einem Backup die Seiten zu etwa 80 %. Damit hat die Datenbank etwas »Luft«, wenn neue Datensätze eingefügt werden, und muss nicht gleich eine neue Seite beginnen. Die Datenbank wird dadurch jedoch etwas größer, weil mehr Seiten belegt werden müssen.

  Ist zu erwarten, dass sich die Datenbank nicht oder allenfalls minimal ändert, beispielsweise, weil sie als Read-Only-Datenbank auf CD-ROM gebrannt wird, dann können die Seiten mittels *Use All Space* zu 100 % gefüllt werden.
- Mit *Verbose Output* erstellen Sie ein Protokoll des Restore-Vorgangs, wahlweise als Fenster oder Datei oder halt überhaupt nicht.

7.3 Verwaltung der Datenbanken

In der Baumansicht werden auch alle registrierten Datenbanken sowie deren Bestandteile angezeigt. Die Ansicht unterscheidet dabei verbundene und nicht verbundene Datenbanken. Erstere werden mit einem grünen Häkchen gekennzeichnet, während nicht verbundene Datenbanken mit einem roten Kreuz signalisiert werden.

Abhängig davon, ob eine Verbindung zur Datenbank besteht oder nicht, werden dann auch andere Menüpunkte auf der rechten Seite und im Kontextmenü angezeigt.

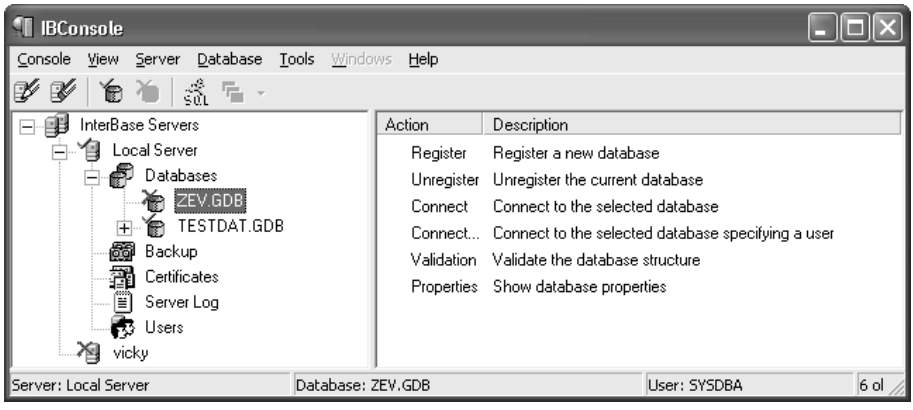


Abbildung 7.17: Eine nicht verbundene Datenbank

Nicht verbundene Datenbanken

Bei nicht verbundenen Datenbanken haben Sie die Möglichkeit, neue Datenbanken zu registrieren oder die aktuelle Datenbank aus der Liste zu entfernen. Sie können sich mit den bei der Registrierung angegebenen Zugangsdaten verbinden oder Sie geben mit CONNECT AS explizit Benutzername und Passwort an.

Mit VALIDATION können Sie eine Datenbank auf Fehler prüfen, Näheres dazu im nächsten Abschnitt. Unter PROPERTIES finden Sie einige Eigenschaften der Datenbank, beispielsweise den Dateinamen der Datenbankdateien. Bei unverbundenen Datenbanken sind die angebotenen Informationen jedoch recht spärlich. Im Kontextmenü finden Sie noch CREATE DATABASE, das werden wir uns in einem späteren Abschnitt noch genauer ansehen.

Database Validation

Für gewöhnlich ist eine InterBase-Datenbank sehr stabil. Es gibt jedoch einige Dinge, die kleinere oder größere Schäden nach sich ziehen können:

- Eine Datenbankanwendung und damit ein InterBase-Client können unerwartet terminieren. Prinzipiell ist dies nicht weiter ein Problem, weil die Daten bereits abgeschlossener Transaktionen ohnehin schon mit COMMIT bestätigt oder mit ROLLBACK verworfen wurden. Alle Transaktionen, die noch nicht abgeschlossen sind, werden mit ROLLBACK verworfen.

  So weit, so gut, allerdings können in diesem Zusammenhang Datenbankseiten angefordert worden sein, die nicht wieder freigegeben werden. Das stört nicht weiter als dass halt ein paar Seiten unbenutzt sind und somit die Datenbank größer als erforderlich ist.
- Auf einer Datenbankseite könnte die Prüfsumme nicht stimmen. Eigentlich dürfte das überhaupt nicht vorkommen, aber wenn es einen Hardwarefehler gibt, wenn zwei Server auf dieselbe Datenbankdatei zugreifen oder wenn

irgendwelche »Spezialisten« mit dem Hex-Editor an der Datenbankdatei herumkurieren, dann stimmen die Prüfsummen nicht mehr. Von einer solchen Datenbank lässt sich nur dann ein Backup ziehen, wenn man die Prüfsummen ignoriert.

Der langen Rede kurzer Sinn: Es kann ja mal vorkommen, dass eine Datenbank defekt ist. Um sie zu prüfen und dabei zu reparieren, gibt es *Database Validation*.

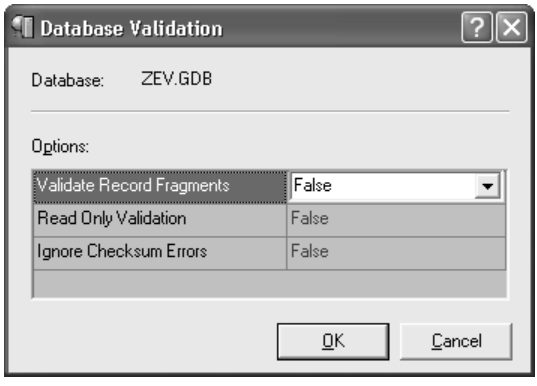


Abbildung 7.18: Überprüfung einer Datenbank

Normalerweise werden nur die Datenbankseiten geprüft. Sollen auch die einzelnen Datensätze geprüft werden, dann ist *Validate Record Fragments* auf *True* zu setzen. Bei der Validation werden die Fehler gleich – so weit es geht – behoben. Soll dies nicht passieren, sollen also die Fehler nur angezeigt werden, dann setzt man *Read Only Validation* auf *True*.

Bei Prüfsummenfehlern kann eine Validation nur erfolgen, wenn *Ignore Checksum Errors* auf *True* gesetzt wird. Wenn die Datenbank dabei gleich repariert wird, ist auf den betreffenden Seiten mit Datenverlust zu rechnen. Nach der Validation sollte dann ein Backup-Restore-Durchgang durchgeführt werden.

Für die Validation wird exklusiver Zugang zur Datenbank benötigt, andere Benutzer müssen gegebenenfalls getrennt werden. Nach der Validation wird ein Bericht angezeigt.

Erstellen einer neuen Datenbank

Prinzipiell kann eine neue Datenbank auch per SQL-Script erstellt werden, Sie können jedoch auch den folgenden Dialog verwenden:

Die wichtigste Angabe ist der Name (einschließlich Laufwerksbuchstabe und Pfad) der Datenbankdatei beziehungsweise der Datenbankdateien. Die Endung dafür kann frei vergeben werden, es kann auch auf eine Endung verzichtet werden. Die übliche Endung bis Version 6.5 ist *.gdb*, ab Version 7 *.ib*.

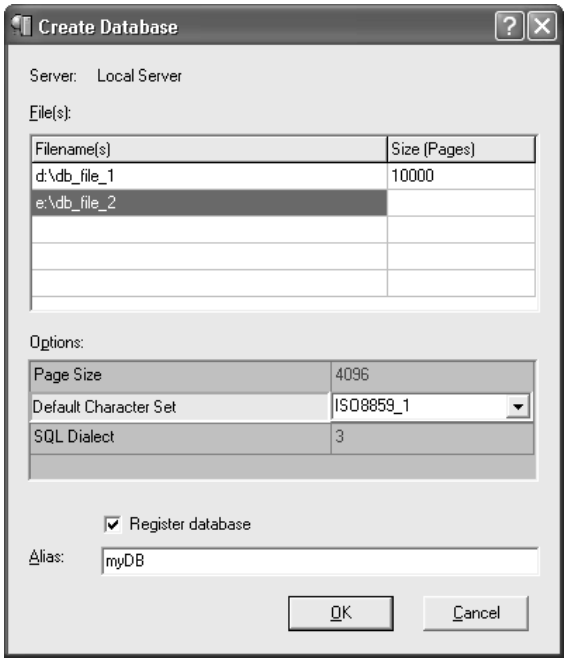


Abbildung 7.19: Eine Datenbank erstellen

Bei allen Dateien bis auf die letzte muss eine Größe angegeben werden. Bei beispielsweise 10000 Seiten und einer Seitengröße von 4096 Byte würde die Datenbankdatei bis etwa 40 Mbyte wachsen. InterBase verteilt dabei die Daten nach eigenem Ermessen auf die beteiligten Datenbankdateien. Da der Festplattenzugriff normalerweise der Flaschenhals eines jeden Datenbankservers ist, kann man die Performance verbessern, indem man die Datenbank auf mehre Dateien verteilt, vorausgesetzt, diese liegen auf verschiedenen physikalischen Festplatten (die Verteilung auf mehrere Partitionen derselben Platte ist dagegen kontraproduktiv).

Die Verteilung auf mehrere Datenbankdateien ist auch dann erforderlich, wenn die Größe der Datenbank die maximale Dateigröße (beispielsweise 2 Gbyte) übersteigt. InterBase kann bis zu 65536 Datenbankdateien verwalten, wobei Shadows mitgezählt werden. Wird auf einen Shadow verzichtet und liegt die Beschränkung durch das Betriebssystem bei 2 Gbyte, dann wäre die maximale Datenbankgröße 128 Tbyte.

Mit der *Page Size* wird festgelegt, wie groß die einzelne Datenbankseite ist. Da die meisten Betriebssysteme ihren Festplattenzugriff in 4 Kbyte großen Blöcken durchführen, liegt das Performance-Maximum fast immer bei 4096.

Mit *Default Character Set* kann ein Sprachtreiber für die gesamte Datenbank spezifiziert werden. Die Sprache lässt sich jedoch auch für jede einzelne Tabellenspalte

angeben. InterBase unterstützt zwei SQL-Dialekte, wobei 1 der alte und 3 der neue Dialekt ist. Neue Datenbanken sollten in der Regel mit SQL-Dialekt 3 begonnen werden, der Dialekt lässt sich jedoch auch im laufenden Betrieb ändern.

Wird die Option *Register Database* gewählt, dann wird die neu erstellte Datenbank gleich bei IBConsole registriert. Zu diesem Zweck muss ein Alias angegeben werden.

Verbundene Datenbanken

Etwas anders sieht das Menü bei verbundenen Datenbanken aus:

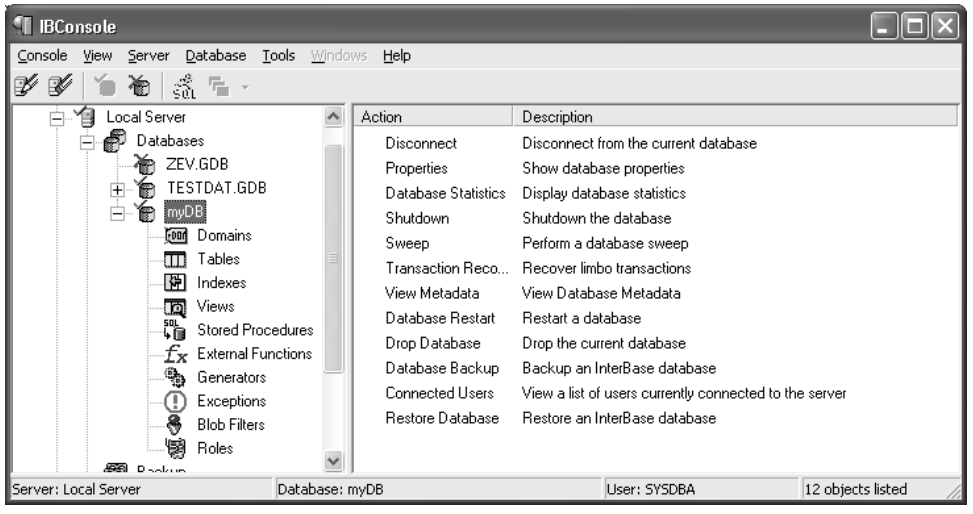


Abbildung 7.20: Eine verbundene Datenbank

Mit DISCONNECT wird die Verbindung zur Datenbank getrennt. Mit PROPERTIES kann man sich die Eigenschaften der Datenbank ansehen und sie auch verändern:

Auf der Registerseite *Alias* sehen Sie den Aliasnamen in IBConsole und den Dateinamen der ersten Datenbankdatei. Die weiteren Datenbankdateien werden auf der Registerseite *General* aufgelistet. Eine *On Disk Structure* von *11.0* kennzeichnet eine InterBase 7.0-Datenbank. Um eine Datenbank von einer älteren ODS aus umzuwandeln, führen Sie ein Backup und ein Restore durch.

Wenn hier etwas von *Allocated DB Pages = 10002* und einer *Page Size = 4096* geschrieben wird, dann müssten eigentlich über 40 Mbyte auf der Platte sein – in Wahrheit ist es noch nicht einmal eines. Diese Angaben sind somit von beschränktem Informationsgehalt, da InterBase erforderlichenfalls auch viel mehr Speicher reservieren würde.

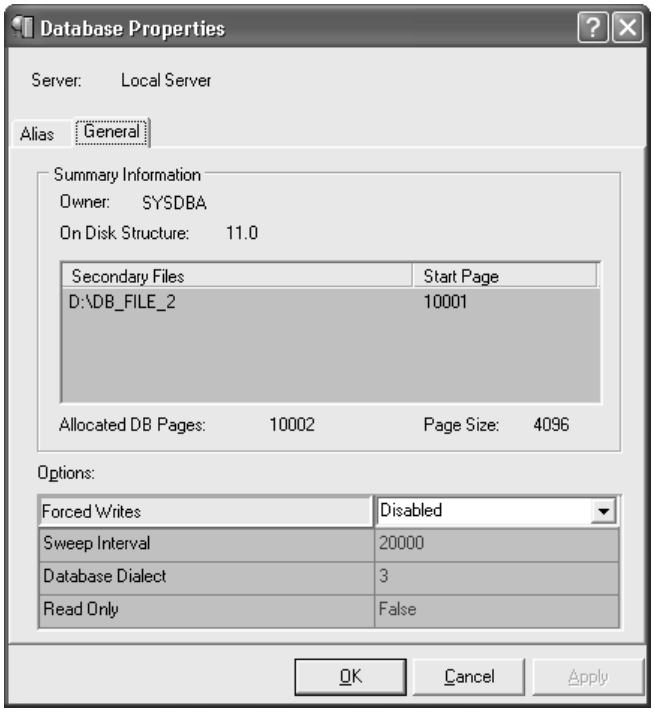


Abbildung 7.21: Die Eigenschaften einer Datenbank

Kommen wir zu den Optionen: Indem man *Forced Writes* auf *Enabled* setzt, wird jede Datenänderung sofort auf die Festplatte geschrieben – per Voreinstellung überlässt InterBase dem Betriebssystem das Schreiben und dieses benutzt erst einmal den Festplatten-Cache.

Der Weg über den Cache ist natürlich der schnellere, kann aber zu Datenverlust führen, wenn der Rechner unerwartet terminiert. In Situationen, in denen mit häufigen Serverabstürzen zu rechnen ist, wenn es auf Performance nicht besonders ankommt oder wenn die Daten besonders wichtig sind, sollte *Forced Writes* auf *Enabled* gesetzt werden. In einem eben durchgeführten Test waren die Unterschiede auch alles andere als gewichtig (etwa 550 msec statt 500 msec).

Das *Sweep Interval* besprechen wir im nächsten Abschnitt. InterBase unterstützt den alten SQL-Dialekt 1 und den neueren Dialekt 3, die sich durch einige Feinheiten unterscheiden. So dürfen String-Konstanten beim SQL-Dialekt 3 nur in einfachen Anführungszeichen stehen, die doppelten Anführungszeichen sind den Bezeichnern vorbehalten – SQL-Dialekt 1 akzeptiert für String-Konstanten sowohl einfache als auch doppelte Anführungszeichen. Der Dialekt kann im laufenden Betrieb geändert werden.

Wird *Read Only* auf *True* gesetzt, dann kann die Datenbank nicht mehr verändert werden. Wenn Sie eine Datenbank auf CD-ROM brennen wollen, dann müssen Sie daraus erst eine Read-Only-Datenbank machen, damit beispielsweise die Transaktionsverwaltung keine Zugriffe auf die Datenbankdatei versucht. Sie sollten dabei dann auch ein Backup und Restore mit der Option *Use All Space* durchführen, damit die Datenbankseiten vollständig gefüllt werden.

Datenbank-Sweep

InterBase ist ein Mehr-Generationen-Datenbanksystem, von einem Datensatz kann es also mehrere Generationen (man könnte auch sagen: mehrere Versionen) geben. Nehmen wir an, es laufen schon ein paar Transaktionen, wenn eine neue Transaktion gestartet wird und einen Datensatz ändert. Dann behält die Datenbank zusätzlich auch noch die alte Version dieses Datensatzes, weil ja einerseits die Transaktion mit ROLLBACK rückgängig gemacht werden könnte und außerdem bei Lesezugriffen der alten SNAPSHOT-Transaktionen ja auch noch nach dem COMMIT der ändernden Transaktion der alte Datensatz ermittelt werden muss.

Im Laufe der Zeit würden sich immer mehr veraltete Versionen ansammeln, die Datenbank würde dadurch immer weiter anwachsen. Um dies zu vermeiden, gibt es die automatische »Müllbeseitigung«, die *garbage collection*: Immer dann, wenn eine Transaktion die für sie aktuelle Version eines Datensatzes ermittelt, werden alle nicht mehr benötigten Versionen entfernt. Das funktioniert jedoch nur, solange die Daten sequenziell gelesen werden. Beim Zugriff über einen Index wird diese Müllbeseitigung nicht durchgeführt.

Bei Tabellen, in denen immer nur über einen Index gelesen wird, kann sich somit eine Menge Müll ansammeln. Abhilfe würde hier ein Backup schaffen, weil dafür alle Datensätze sequenziell ausgelesen werden – und ein Backup kann man eigentlich gar nicht oft genug ziehen.

Da ein Backup den Server stärker belastet als ein einfacher Datenbank-Sweep, könnte man Letzteren auch manuell ausführen, nämlich mit dem Menüpunkt SWEEP. Darüber hinaus gäbe es auch die Möglichkeit, ein *Sweep Interval* einzustellen, der Server startet dann nach beispielsweise 20000 Transaktionen (Alternativ bietet das Handbuch an: Wenn die Differenz zwischen OAT und OIT diese Schwelle erreicht hat. Das erscheint mir aber wenig plausibel ...) eigenständig einen Thread, der einen solchen Sweep durchführt.

Diese Automatik kann man abschalten, indem man das *Sweep Interval* auf null setzt. Davon möchte ich jedoch abraten: Wenn dann entsprechende Aktionen nicht manuell ausgeführt werden, dann erfolgen sie überhaupt nicht mehr. Wenn Sie Sorge haben, dass zu Zeitpunkten maximaler Serverbelastung dann zufälligerweise auch noch ein Sweep durchgeführt wird, dann setzen Sie das *Sweep Interval* auf einen so hohen Wert, dass er gerade nicht erreicht wird, wenn täglich ein

Backup gezogen wird. Wird dann – aus welchen Gründen auch immer – längere Zeit kein Backup gezogen, dann sammelt sich wenigstens nicht so viel Müll in der Datenbank an.

Weitere Aktionen mit verbundenen Datenbanken

Für manche Aktionen, beispielsweise für die Validation oder das Einspielen eines Backups, benötigen Sie exklusiven Zugriff auf die Datenbank. Mit *Connected Users* erhalten Sie eine Liste der verbundenen Benutzer. Bei Datenbanken mit vielen Benutzers hilft dies allerdings nicht weiter, bis der Administrator allen Benutzern mitgeteilt hat, dass sie sich abmelden sollen, haben sich schon wieder ein Dutzend neue angemeldet. Mit *Shutdown* lässt sich deshalb eine Datenbank herunterfahren.

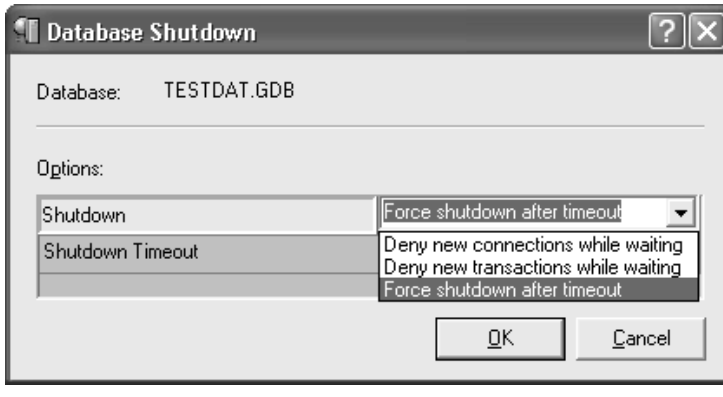


Abbildung 7.22: Eine Datenbank herunterfahren

Hier können Sie einstellen, ob der Server neue Verbindungen beziehungsweise neue Transaktionen verweigern oder gleich alle Verbindungen nach der eingestellten Zeit beenden soll. Um die Datenbank dann wieder für alle Benutzer zugänglich zu machen, verwenden Sie *Database Restart*.

Mit *Transaction Recovery* können Sie Transaktionen, die sich aufgrund eines Zwei-Phasen-Commits in der Schwebe befinden, manuell zurücksetzen. Wenn Sie keine Transaktionen über mehrere Datenbanken erstellen, werden Sie solche *Limbo Transactions* nie erhalten.

Mit *View Metadata* können Sie das SQL-Script der gesamten Datenbank einsehen. Diese Option nutzt man häufig dazu, das SQL-Script zu modifizieren und zu speichern, um dann die Datenbank zu löschen und mittels SQL-Script neu zu erstellen.

Mit *Drop Database* kann die Datenbank gelöscht werden.

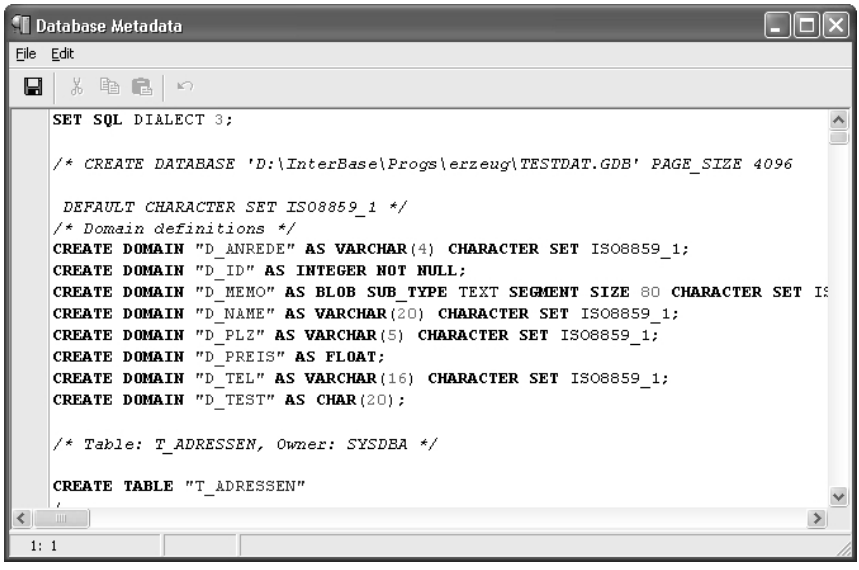


Abbildung 7.23: Das SQL-Script

7.3.1 Statistik

Über den Menüpunkt DATABASE STATISTICS können Sie die statistischen Daten einer Datenbank erhalten. Bei umfangreichen Datenbanken sollten Sie der Versuchung widerstehen, aus Bequemlichkeit einfach *All Options* zu wählen, denn zum einen kann die Ausführung dieser Anweisung ein wenig länger dauern, zum anderen bekommen Sie damit nicht alle verfügbaren Informationen.

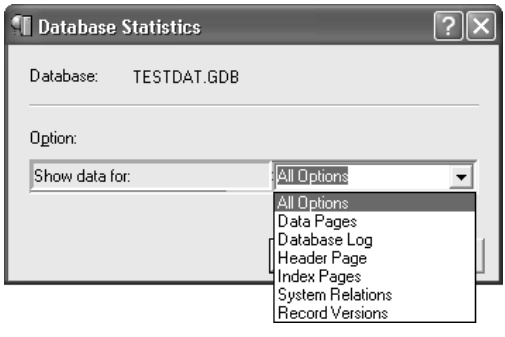


Abbildung 7.24: Die Datenbankstatistik

Header

Schauen wir uns den Header einer solchen Datenbankstatistik an. Da der Header sehr schnell ausgelesen ist, wird er bei allen der verfügbaren Optionen mit angezeigt.

```
Service started at 22.01.2003 22:16:50

Database "D:\InterBase\Progs\erzeug\TESTDAT.GDB"

Database header page information:
  Flags      0
  Checksum    12345
  Generation    17
  Page size    4096
  ODS version    11.0
  Oldest transaction  27
  Oldest active    28
  Oldest snapshot    28
  Next transaction  32
  Bumped transaction  1
  Sequence number    0
  Next attachment ID  0
  Implementation ID  16
  Shadow count    0
  Page buffers    0
  Next header page  0
  Database dialect  3
  Creation date    Jan 22, 2003 21:52:27
  Attributes

    Variable header data:
  Sweep interval:    20000
  *END*
```

Hier erhalten Sie eine Menge Informationen, die für gewöhnlich überhaupt nicht interessieren, wir wollen uns damit auch nicht beschäftigen – gegebenenfalls ziehen Sie den *Operation Guide* zu Rate. Was im Header wirklich interessant ist, sind die Informationen über die Transaktionen:

Die *Oldest Transaction* ist die älteste Transaktion, die für den Server noch interessant ist. Dazu gehören alle aktiven Transaktionen, aber auch alle Transaktionen, die mit ROLLBACK zurückgenommen wurden, und alle Transaktionen, die im Zuge eines 2-Phasen-Commits noch nicht endgültig beendet wurden. Die *Oldest active* ist dagegen die älteste Transaktion, die weder mit COMMIT noch mit ROLLBACK beendet wurde, die *Oldest snapshot* die älteste noch nicht abgeschlossene Transaktion mit dem Isolationsgrad SNAPSHOT. Beim Vorhalten alter Versionen eines Datensatzes muss nur auf diejenigen Transaktionen besondere Rücksicht genommen werden, die den Isolationsgrad SNAPSHOT haben. *Next transaction* ist die Nummer, welche die Transaktion erhält, die als Nächstes gestartet wird.

Je größer die Differenz zwischen *Next transaction* und *Oldest snapshot*, desto mehr muss der Server veraltete Datensatz-Versionen vorhalten. Eine große Differenz ist ein Zeichen dafür, dass die Laufzeit der Transaktionen zu lang ist.

Die Datenbankseiten

Der zweite interessante Block sind die *Database Pages*, die besonders dann informativ werden, wenn Sie die Option *Record Versions* wählen. Es werden alle Tabellen und zu den einzelnen Tabellen Indizes aufgeführt:

```
T_ADRESSEN (128)
    Primary pointer page: 170, Index root page: 171
    Average record length: 80.28, total records: 8289
    Average version length: 9.00, total versions: 1,
    max versions: 1
    Data pages: 249, data page slots: 249, average fill: 80%
    Fill distribution:
     0 - 19% = 1
    20 - 39% = 0
    40 - 59% = 0
    60 - 79% = 147
    80 - 99% = 101
```

Auf welchen Seiten die Pointer und die Pages liegen, sollte nicht weiter interessieren. Die *Average record length* ist die durchschnittliche Länge eines Datensatzes – InterBase komprimiert die Daten beim Speichern, entfernt also überflüssige Leerzeichen. *Total records* liefert die Zahl der Datensätze.

Muss der Server eine zusätzliche Version eines Datensatzes anlegen, dann tut er dies nicht vom gesamten Datensatz, sondern nur von den abweichenden Spalten. Die durchschnittliche Länge der zusätzlichen Datensätze, die *Average version length*, ist somit kürzer als die durchschnittliche Datensatzlänge. Die Anzahl dieser Dubletten erfahren Sie in *total versions*, das Maximum der unterschiedlichen Versionen pro Datensatz in *max versions*.

Die Anzahl der Datenbankseiten wird in *Data pages* mitgeteilt, die Anzahl der Zeiger auf Datenbankseiten in *data page slots*, diese Zahl ist unabhängig davon, ob diese Seiten noch in Verwendung sind oder nicht.

Der durchschnittliche Füllgrad ist *average fill*. In der *Fill distribution* erfahren Sie, wie viele Seiten wie stark gefüllt sein. Bei einem Restore versucht InterBase, die Seiten mit etwa 80 % zu füllen (es sei denn, Sie wählen *Use All Space*), damit die Datenbank noch »atmen« kann, also die verschiedenen Versionen eines Datensatzes noch auf dieselbe Datenbankseite passen, damit sie mit einem Festplattenzugriff eingelesen werden.

Bei einer frisch wiederhergestellten Datenbank sollten alle Datenbankseiten mit etwa 80 % gefüllt sein, das »etwa« ist dabei abhängig von der Größe der Datensätze, InterBase beginnt nicht »mitten im Datensatz« eine neue Seite, nur um exakt auf 80 % zu kommen. Die letzte Datenbankseite wird dann so weit gefüllt, wie es sich halt gerade ergibt. Im laufenden Betrieb kommen immer mehr Seiten hinzu, welche eher mäßig gefüllt sind – hier kann dann mit einem Restore eine Beschleunigung erzielt werden.

```
Index IX_ADRESSEN_ANREDE (4)
Depth: 2, leaf buckets: 13, nodes: 8289
Average data length: 0.00, total dup: 8287, max dup: 4249
Fill distribution:
     0 - 19% = 0
    20 - 39% = 1
    40 - 59% = 0
    60 - 79% = 0
    80 - 99% = 12
```

Auch für jeden Index werden statistische Daten angezeigt. Die Tiefe sollte nicht größer sein als 3 und je größer die Zahl der Dubletten ist, desto wirkungsloser ist der Index. Hier wurde – zu Demonstrationszecken – ein Index auf die Anrede erstellt, also auf eine Spalte, in der nur die Werte *Herr* und *Frau* vorkommen. Mit *total dub* erfragen Sie die Zahl der Datensätze, für die es Dubletten gibt, mit *max dub* die Anzahl der Dubletten für den Datensatz, der die meisten Dubletten aufweist. Ein solcher Index hat natürlich eine sehr niedere Selektivität, aber selbst ein solcher Index beschleunigt noch den Zugriff, wie die Messung in Kapitel 3 gezeigt hat.

7.4 Die Metadaten einer Datenbank

Für alle verbundenen Datenbanken werden in einer Baumansicht die Metadaten angezeigt, sortiert nach den einzelnen Elementen, beispielsweise Domains, Tables, Indizes ...

Für alle diese Elemente können entsprechende Dialogfenster aufgerufen werden, die weitere Informationen bieten und teilweise auch nützliche Aktionen zulassen. Da sich diese Dialogfenster in vielen Teilen gleichen, wollen wir nicht alle besprechen, sondern uns auf Tabellen und Indizes beschränken.

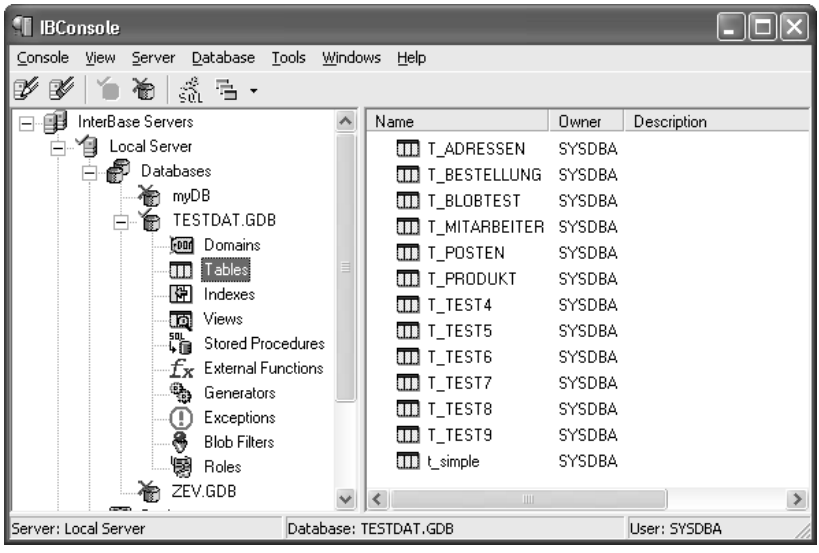


Abbildung 7.25: Die Baumansicht der Metadaten

7.4.1 Tabellen

Auf der Registerseite *Properties* sehen Sie in tabellarischer Form Informationen über die Tabelle:

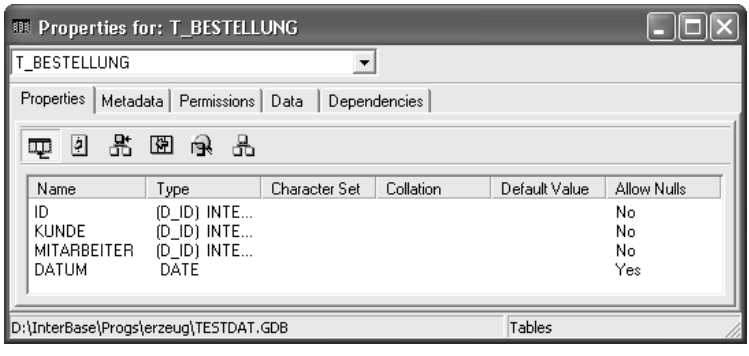


Abbildung 7.26: Die Spalten einer Tabelle

Links oben auf der Registerseite sehen Sie sechs Buttons, mit deren Hilfe Sie wählen können, welche Informationen angezeigt werden sollen:

- Spalten und deren Domänen
- Trigger
- Gültigkeitsprüfungen auf Tabellenebene
- Indizes

- Primär- und Sekundärschlüssel
- Fremdschlüssel

Diese Informationen werden nur angezeigt, können jedoch nicht verändert werden.

Metadata

Auf der Registerseite *Metadata* finden Sie den Ausschnitt aus dem SQL-Script, der die gewählte Tabelle betrifft. Dabei werden auch diejenigen Elemente angezeigt, von denen die Tabelle abhängt (Domänen) oder die fest zu der betreffenden Tabelle gehören (Trigger, Fremdschlüssel).

Die Informationen können Sie über die Zwischenablage kopieren, verwenden Sie dazu das Kontextmenü. Das SQL-Script können Sie nicht editieren, wenn Sie Teile in die Zwischenablage ausschneiden, hat dies keine Rückwirkungen auf die Datenbank.

Permissions

Welche Rechte für welche Benutzer eingerichtet sind, ersehen Sie der Registerseite *Permissions*.

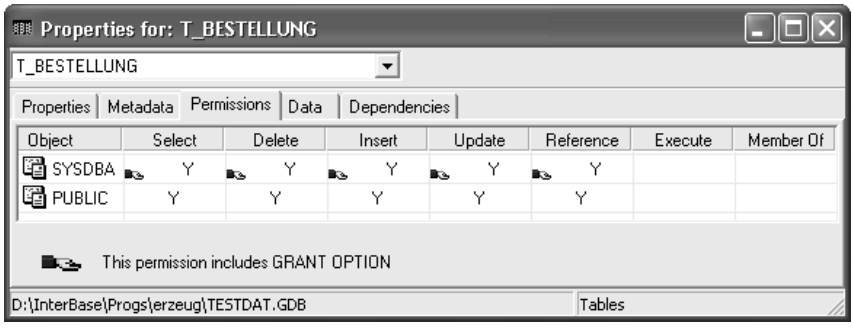


Abbildung 7.27: Die Rechte der einzelnen Benutzer

Von dieser Stelle aus können Sie auch recht komfortabel Rechte vergeben. Führen Sie dazu einen Doppelklick auf die Ansicht aus und wählen Sie den Benutzer, dessen Rechte Sie verändern wollen:

Mit OK gelangen Sie in den *Grant Editor*. Wollen Sie die Rechte eines bereits vorhandenen Benutzers modifizieren, so springen Sie aus der Ansicht der Rechte mit einem Doppelklick auf seinen Namen direkt in den *Grant Editor*.

Hier wählen Sie nun, welche Rechte der Benutzer haben soll. Soll der Benutzer diese Rechte auch weitergeben dürfen, dann ist jeweils die Option *with grant option* zu wählen.

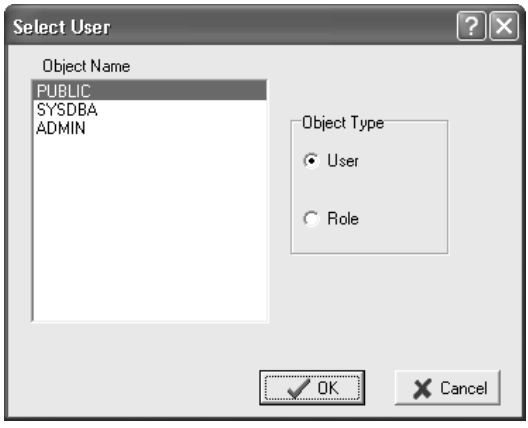


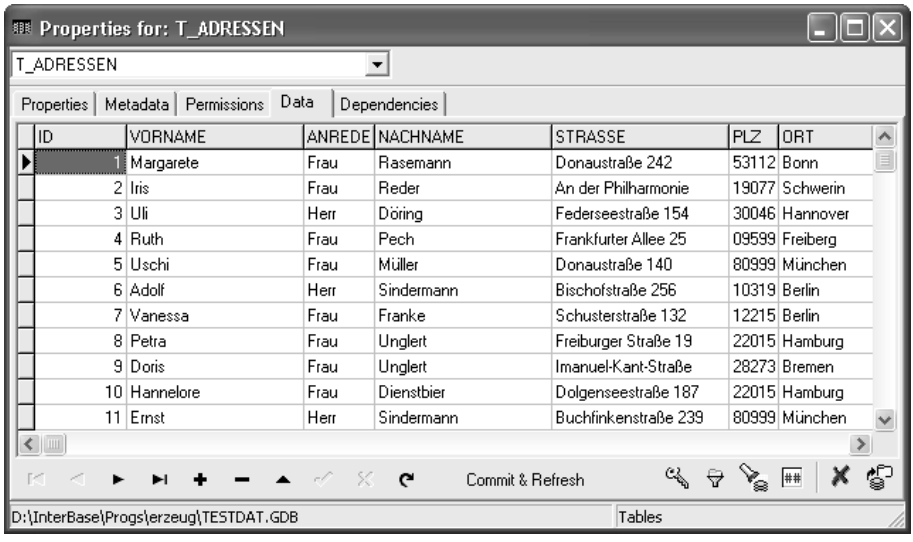
Abbildung 7.28: Den Benutzer wählen

Abbildung 7.29: Die Rechte zuweisen

Data

Auf der Registerseite *Data* können Sie die Daten der gewählten Tabelle einsehen und auch ändern:

Über die Handhabung des Navigators auf der linken Seite des Fensters brauche ich Sie wohl nicht mehr aufzuklären. In InterBase 7.0 sind auf der rechten Seite einige Buttons hinzugekommen:



ID	VORNAME	ANREDE	NACHNAME	STRASSE	PLZ	ORT
1	Margarete	Frau	Rasemann	Donaustraße 242	53112	Bonn
2	Iris	Frau	Reder	An der Philharmonie	19077	Schwerin
3	Uli	Herr	Döring	Federseestraße 154	30046	Hannover
4	Ruth	Frau	Pech	Frankfurter Allee 25	09599	Freiberg
5	Uschi	Frau	Müller	Donaustraße 140	80999	München
6	Adolf	Herr	Sindermann	Bischofstraße 256	10319	Berlin
7	Vanessa	Frau	Franke	Schusterstraße 132	12215	Berlin
8	Petra	Frau	Unglert	Freiburger Straße 19	22015	Hamburg
9	Doris	Frau	Unglert	Imanuel-Kant-Straße	28273	Bremen
10	Hannelore	Frau	Dienstbier	Dolgenseestraße 187	22015	Hamburg
11	Ernst	Herr	Sindermann	Buchfinkenstraße 239	80999	München

Abbildung 7.30: Die Daten der Tabelle

- Mit dem Schlüssel können Sie einen Index wählen und die Tabelle entsprechend sortieren.
- Mit etwas gutem Willen kann man das nächste Symbol als einen Kaffeefilter ansehen. Damit können Sie eine Filter-Bedingung erstellen.

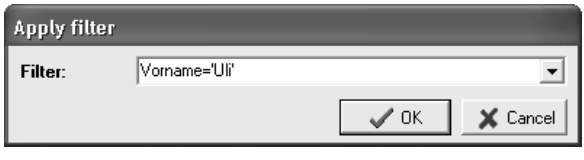


Abbildung 7.31: Filtern der Tabelle

- Auf dem nächsten Button scheint eine Taschenlampe auf die Festplatten: Wenn Sie einen Index gewählt haben, können Sie den Cursor damit auf dem ersten Vorkommen des eingegebenen Wertes positionieren. Im folgenden Beispiel wurde der Index *Nachname* gewählt:

Abbildung 7.32: Positionieren des Cursors

- Mit dem nächsten Button können Sie die Ausgabe der indizierten Spalte formatieren – ich habe keinen praktischen Nutzen dafür gefunden.
- Mit einem Mausklick auf das rote Kreuz leeren Sie – nach einer Sicherheitsabfrage – die Tabelle.
- Zuletzt können Sie noch die Daten exportieren. Wählen Sie dazu die Spalten, und zwar in der Reihenfolge, wie sie später exportiert werden sollen. Zur Not müssen Sie einige Spalten aus der Liste nehmen und erneut einfügen.

  Mit dem *Filename* wählen Sie auch das Format, in dem exportiert werden soll. Bei *Delimited File* wird für jeden Datensatz eine Zeile angelegt, die einzelnen Spalten werden durch das mit *Deliminter* (soll wohl heißen *Delimiter* ...) spezifizierte Zeichen getrennt. Damit kann man die klassische CSV (*comma seperated value-list*) erstellen.

  Alternativ stehen die Formate XML und *Raw Data* zur Verfügung, Letzteres exportiert Binär-Daten, deren Weiterverarbeitung etwas problematisch sein könnte.

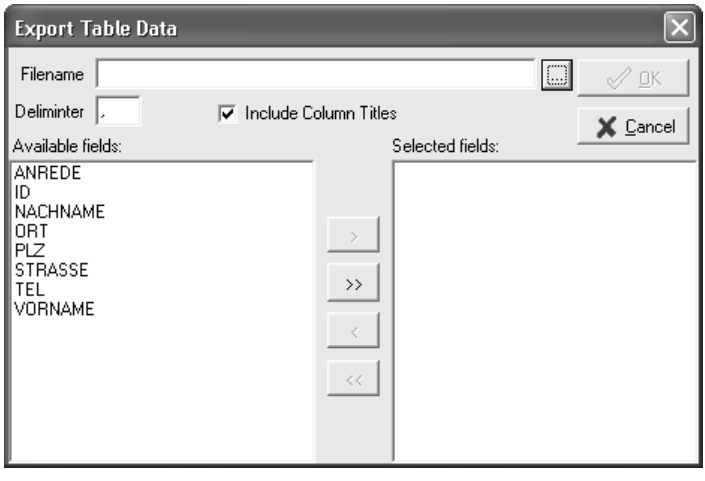


Abbildung 7.33: Daten exportieren

Dependencies

Zuletzt kann noch angezeigt werden, von welchen Elementen die Tabelle abhängt (*Show Depended On Objects*) und welche Elemente die Tabelle benutzen (*Show Dependent Objects*). Es werden dabei jedoch nicht alle entsprechenden Elemente angezeigt: So ist *t_adressen* beispielsweise abhängig von verschiedenen Domänen, die aber bei *Show Depended On Objects* nicht aufgeführt werden.

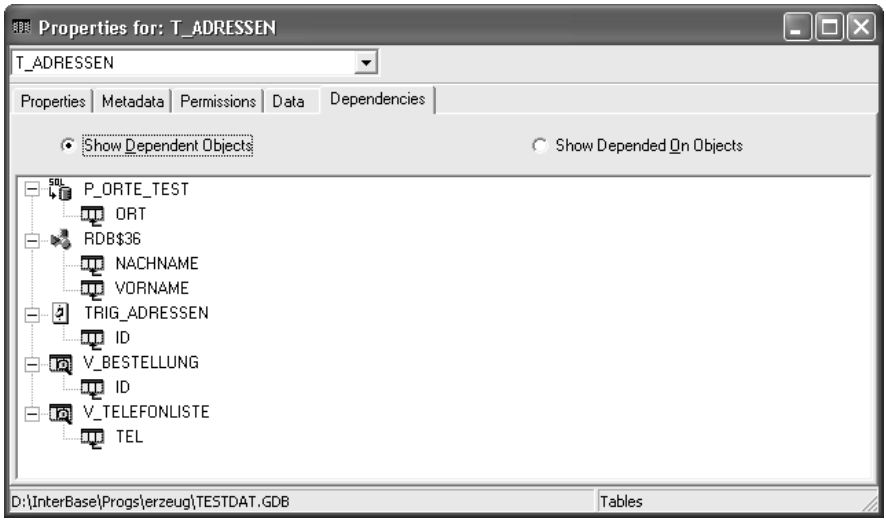


Abbildung 7.34: Die Abhängigkeiten

7.4.2 Indizes

Für jeden Index wird ein Dialog nach *Abbildung 7.35* angezeigt. Neben der Tabelle und den verwendeten Spalten interessieren hier vor allem die Attributes: Ist der Index aktiv, ist er auf- oder absteigend und ist er eindeutig, also ein Schlüssel.

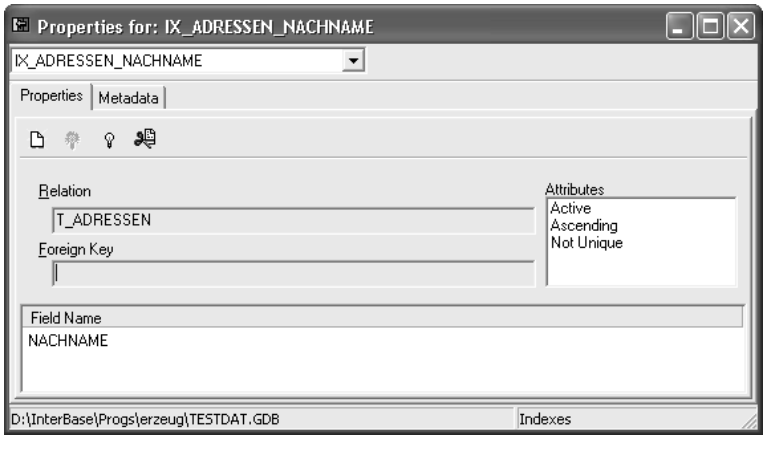


Abbildung 7.35: Die Index-Eigenschaften

Bei den Tool-Buttons finden Sie zwei »Glühbirnen«, mit denen Sie den Index an- und abschalten können. Alle Indizes auf Tabellen, in die häufiger Daten eingefügt werden, sollten hin und wieder ab- und anschließend wieder angeschaltet werden, damit diese Indizes neu aufgebaut werden und anschließend performanter laufen.

Mit dem Tool-Button ganz links können Sie einen neuen Index erstellen:

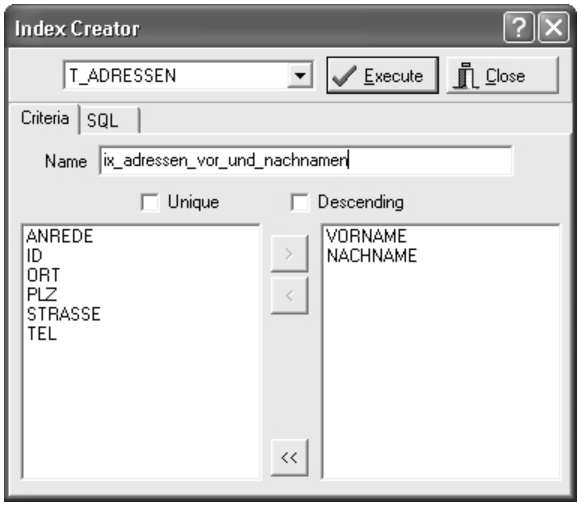


Abbildung 7.36: Einen Index erstellen

7.5 Interactive SQL

Mit dem Programm *Interactive SQL* können Sie SQL-Anweisungen zum Server schicken. Handelt es sich um SELECT-Statements, dann wird die Ergebnismenge in einem DBGrid angezeigt:

Interactive SQL wird von *IBConsole* aus für die aktuelle Tabelle aufgerufen, entweder mit TOOLS | INTERACTIVE SQL oder dem entsprechenden Tool-Button.

Transaktionen

Beim Aufruf von *Interactive SQL* ist noch keine Transaktion aktiv. Wird nun eine Anweisung ausgeführt, so wird automatisch eine Transaktion gestartet. Diese läuft so lange, bis sie mit TRANSACTIONS | COMMIT bestätigt oder mit TRANSACTIONS | ROLLBACK verworfen wird. Bei der darauf folgenden Anweisung wird dann wieder automatisch eine Transaktion gestartet.

Die Tabellenansichten bei den Metadaten und *Interactive SQL* verwenden dieselbe Transaktion. Wenn Sie auf der Registerseite *Data* den Button COMMIT & REFRESH anklicken, dann wird die Ergebnisanzeige in *Interactive SQL* gelöscht, weil der Transaktionskontext verloren geht. Innerhalb von *IBConsole* das Transaktionsverhalten erforschen zu wollen geht leider nicht, zumal auch keine Transaktionsparameter eingestellt werden können.

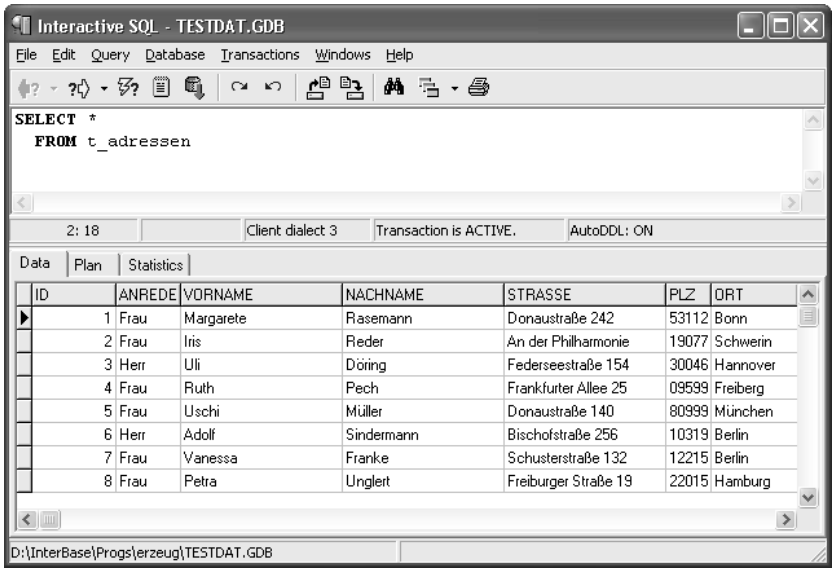


Abbildung 7.37: Interactive SQL

Die SQL-Anweisungen

Interactive SQL speichert alle eingegebenen SQL-Anweisungen. Mit QUERY|PREVIOUS und QUERY|NEXT kann in dieser Liste navigiert werden, um frühere SQL-Anweisungen erneut auszuführen. Diese Liste kann ausgedruckt, gespeichert und auch wieder geladen werden, die entsprechenden Menüpunkte finden Sie unter FILE.

Mit QUERY|LOAD SCRIPT können Sie ein SQL-Script laden, darunter finden Sie auch einen Menüpunkt, um den Editor-Inhalt als ein solches zu speichern.

Das Ergebnis einer Abfrage können Sie auch als Textdatei speichern:

```
ID      VORNAME                NACHNAME
====    ====================   ====================
1       Hiltrud                Durach
2       Bianca                 Hoffmann
3       Margarete              Faust
4       Klaus                  Runge
5       Eleonore               Ehrmann
```

Eigentlich sollte das auch als CSV gehen, doch eine solche Anweisung wird ignoriert.

Query-Plan und Statistik

Dort, wo die Daten angezeigt werden, sehen Sie drei Registerreiter mit den Aufschriften *Data*, *Plan* und *Statistics*. Hier finden Sie eine gewisse Hilfe zum Optimieren Ihrer SQL-Anweisung.

Auf der Registerseite *Plan* finden Sie die SQL-Anweisung wiederholt und ergänzt um den Query-Plan:

```
Statement: SELECT b.id,
    a.vorname, a.nachname,
    m.nachname AS Mitarbeiter
  FROM t_bestellung b
    INNER JOIN t_adressen a
      ON a.id = b.kunde
    INNER JOIN t_mitarbeiter m
      ON m.id = b.mitarbeiter
  WHERE b.id BETWEEN 1 AND 100
  ORDER BY b.id

PLAN JOIN (B ORDER RDB$PRIMARY2,M INDEX (RDB$PRIMARY3),
  A INDEX (RDB$PRIMARY1))
```

Das Thema Query-Plan ist in Kapitel 3 ausführlicher behandelt. Hier im Beispiel würde die Bestellung gemäß ihres Primärschlüssels sortiert, die Kundenadresse und die Mitarbeiternamen würden anhand ihrer Primärschlüssel gefunden.

Etwas vereinfacht kann gelten, dass eine Anweisung umso schneller ausgeführt werden kann, je mehr der Server auf Indizes zurückgreifen kann. Das hier gezeigte Beispiel wäre unter diesem Gesichtspunkt optimal formuliert.

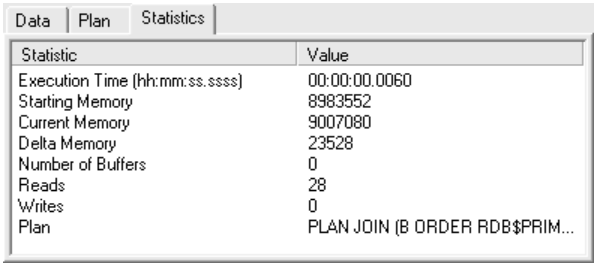


Abbildung 7.38: Die Statistik der Abfrage

Auf der Registerseite *Statistics* sehen Sie, wie schnell der Server die Abfrage ausgeführt hat, wie sich der verfügbare Speicher entwickelt hat, ob – und wenn ja – wie viele Buffers verwendet wurden, sowie die Zahl der Lese- und Schreibvorgänge.

Optionen

Unter EDIT|OPTIONS öffnen Sie den folgenden Dialog, mit dem Sie einige Optionen einstellen können:

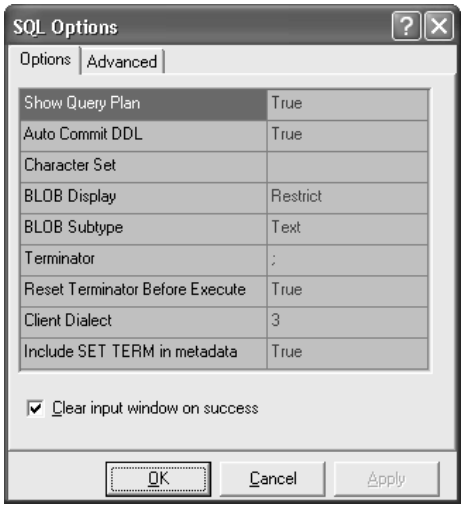


Abbildung 7.39: IBConsole Optionen

Ist die Option *Auto Commit DDL* aktiviert, dann werden nach Änderungen der Metadaten sofort die Transaktionen bestätigt. Mit *Clear input window on success* löschen Sie nach dem Ausführen einer SQL-Anweisung das Editor-Fenster. Im Gegensatz zu früheren Versionen von *Interactive SQL* ist es dabei unerheblich, ob die Anweisung erfolgreich ausgeführt werden konnte oder nicht. (Mal sehen, wie lange das so bleibt ...)

7.6 Kommandozeilentools

Es mag Sie verwundern, einen Abschnitt über die Kommandozeilentools ausgerechnet in einem Kapitel über IBConsole zu finden, ersetzt doch IBConsole eben diese Kommandozeilentools. Leider jedoch nicht vollständig. Wie Sie vorgehen können, wenn Sie mit IBConsole nicht weiterkommen erfahren Sie in diesem Abschnitt.

Anzahl der Buffers einstellen

Per Voreinstellung hat jede Datenbank 2048 Buffers, es wird also Speicher reserviert, um 2048 Datenbankseiten im RAM zu halten. Wenn eine Datenbankseite die empfohlene Größe von 4096 byte hat, dann werden pro Datenbank 8 Mbyte Speicher für diese Zwecke reserviert.

Grundsätzlich arbeitet eine Datenbank dann am schnellsten, wenn sämtliche Daten im Speicher sind und Zugriffe auf die Festplatte unterbleiben können. Von daher wäre es nicht verkehrt, den Wert für Buffers so einzustellen, dass alle Seiten für Daten und Indizes vollständig im Speicher gehalten werden können.

Nun ist es nicht unüblich, dass auf einem Datenbankserver mehrere Datenbanken betrieben werden, von denen zumindest einige gleichzeitig konnektiert sind. Für jede dieser Datenbanken wird nun entsprechend Speicher bereitgestellt. Das geht so lange gut, als der Rechner ausreichend über Arbeitsspeicher verfügt. Ist dies nicht der Fall, dann beginnt das Betriebssystem, Speicherseiten in die Auslagerungsdatei zu verschieben, also wieder auf die Festplatte. Der Effekt wäre dann keine Beschleunigung, sondern eine deutliche Verlangsamung. Sie sollten an der Anzahl der Buffers nur dann etwas ändern, wenn Sie wirklich wissen, was Sie tun.

Prinzipiell könnte man die Anzahl der Buffer auf Server-Ebene einstellen – diese würde dann für alle Datenbanken gelten. Nehmen wir an, Sie haben auf dem Server eine große Datenbank für die Auftragsverwaltungssoftware, die beispielsweise 50 Mbyte groß ist, dementsprechend stellen Sie die Anzahl der Buffers auf 12800. Würden Sie die Anzahl der Buffers nun auf Server-Ebene einstellen, dann würde für alle anderen Datenbanken entsprechend viel Speicher reserviert, obwohl die beispielsweise nur 1 Mbyte groß sind. Da InterBase die Datenbankseiten maximal einmal im Speicher hält – was sollten hier auch Dubletten nutzen – würde hier viel Speicher verschenkt, der an anderer Stelle dringend gebraucht werden könnte.

Der langen Rede kurzer Sinn: Wir stellen die Anzahl der Buffers auf Datenbankebene ein.

Die derzeitige Anzahl der Buffer können Sie über IBConsole mit dem Befehl DATABASE STATISTICS ermitteln. Im Header finden Sie dazu die Angabe *Page Buffers*.

gfix

Um die Zahl der Buffer zu Verändern, verwenden wir das Kommandozeilentool gfix. Rufen Sie dazu die Eingabeaufforderung auf und wechseln Sie zunächst in das \Interbase\Bin-Verzeichnis.

Damit Sie mit den Kommandozeilentools arbeiten dürfen, müssen Sie Benutzernamen und Passwort in entsprechenden Systemvariablen speichern. Führen Sie also nacheinander die folgenden Anweisungen aus:

```
SET ISC_USER=sysdba
SET ISC_PASSWORD=masterkey
```

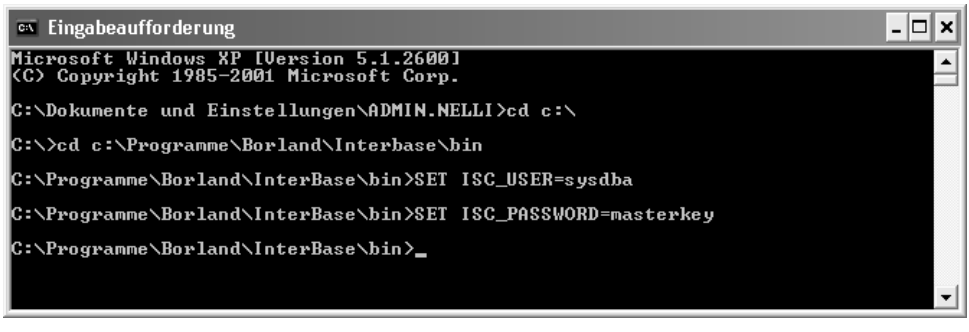


Abbildung 7.40: Benutzernamen und Password

Nun setzen Sie die Anzahl der Buffers auf den gewünschten Wert, beispielsweise 4096. Den Datenbanknamen setzen Sie natürlich entsprechend der Datenbank, deren Bufferzahl Sie verändern wollen.

```
gfix -buffers 4096 d:\interbase\progs\erzeug\testdat.gdb
```

Sie können nun erneut sich den Header von DATABASE STATISTICS anzeigen lassen und werden feststellen, dass sich der betreffende Wert nun geändert hat.

isql

Bei den anderen Kommandozeilentools ist die Vorgehensweise ähnlich wie bei gfix beschrieben: In der \InterBase\Bin-Verzeichnis wechseln, Benutzername und Passwort in die Systemvariablen – das braucht nur einmal zu erfolgen – und dann können die Befehle abgesetzt werden; Näheres dazu finden Sie im *Operation Guide*.

Eine Ausnahme von dieser Vorgehensweise bildet isql, so dass dies hier auch noch beschrieben werden soll. Das Programm isql rufen Sie per Doppelklick im Explorer auf.

Zunächst müssen Sie eine Datenbank erstellen oder sich mit einer solchen verbinden. Zum Verbinden wählen Sie CONNECT und dann den Dateinamen der Datenbank. Im Gegensatz zu den anderen Kommandozeilentools müssen Sie nun auch noch Benutzernamen und Passwort angeben, die Systemvariablen helfen hier leider nicht weiter.

Jede Anweisung muss mit einem Semikolon abgeschlossen werden und wird dann mit der Enter-Taste ausgeführt. Betätigen Sie Enter ohne abschließendes Semikolon, dann wird einfach nur eine neue Zeile begonnen – gerade bei sehr umfangreichen SQL-Anweisungen ist es sehr hilfreich, hier ein wenig Struktur hineinzubringen. Wenn Sie eine neue Anweisung beginnen, dann lautet das Prompt *SQL>*, bei weiteren Zeilen *CON>*.

Sie können hier beliebige SQL-Anweisungen eingeben und auch Transaktionen starten und beenden. Wenn Sie SELECT-Statements ausführen, dann wird das Ergebnis wie in *Abbildung 7.41* ausgegeben.

Abbildung 7.41: isql

8 Die Systemtabellen

In einer Datenbank gibt es nicht nur die Daten, sondern auch die so genannten Metadaten. Darunter versteht man die Definition der Tabellen, der TRIGGER, der STORED PROCEDURES und was man sonst noch so alles definieren kann. Auch diese Metadaten werden in Tabellen gespeichert, in den so genannten Systemtabellen. Dabei handelt es sich um Tabellen, die beim Erstellen einer Datenbank automatisch angelegt werden und mit dem Präfix *RDB$* (im Falle von temporären Systemtabellen mit *TMP$*) beginnen.

Die Systemtabellen sind nicht nur dann interessant, wenn man Programme wie *IBConsole* selbst schreiben möchte – manche Informationen erhält man am schnellsten im Direktzugriff auf die Systemtabellen, manche Probleme lassen sich auch dadurch lösen, dass man diese Daten verändert; tun Sie das aber nur dann, wenn Sie ganz genau wissen, was Sie tun.

Eine vollständige Erläuterung aller Systemtabellen wollen wir uns ersparen – bei Interesse schauen Sie in Kapitel 7 der *Language Referenz*.

8.1 Die Datenbank

Auch wenn es sicher nicht die interessantesten Systemtabellen sind, wollen wir aus systematischen Gründen mit den Systemtabellen zur Datenbank beginnen:

RDB$DATABASE

```
CREATE TABLE "RDB$DATABASE"
  ("RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$RELATION_ID"    SMALLINT,
  "RDB$SECURITY_CLASS" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$CHARACTER_SET_NAME"    CHAR(67) CHARACTER SET UNICODE_FSS);
```

Die Systemtabelle zur Datenbank selbst ist sehr übersichtlich, weil alle relevanten Informationen in anderen Tabellen gespeichert sind.

Die Spalte RDB$DESCRIPTION werden Sie in vielen Systemtabellen finden. Sie erlaubt es dem Benutzer, einen Kommentar einzugeben. Programme wie *IBExpert* stellen Editoren für diese Spalten zur Verfügung.

In der Spalte RDB$CHARACTER_SET_NAME wird der Name des voreingestellten Zeichensatzes gespeichert.

RDB$FILES

```
CREATE TABLE "RDB$FILES"
  ("RDB$FILE_NAME" VARCHAR(253),
  "RDB$FILE_SEQUENCE" SMALLINT,
  "RDB$FILE_START" INTEGER,
  "RDB$FILE_LENGTH" INTEGER,
  "RDB$FILE_FLAGS" SMALLINT,
  "RDB$SHADOW_NUMBER" SMALLINT);
```

Die Tabelle RDB$FILES verwaltet alle sekundären Datenbankdateien und alle Shadows. Die Dateinamen dieser Dateien finden sich in der Spalte RDB$FILE_ NAME, mit RDB$FILE_SEQUENCE erhalten Sie eine Nummer, in deren Reihenfolge sie verwendet werden. Da RDB$FILE_SEQUENCE vom Typ SMALLINT ist, können maximal 65535 sekundäre Datenbankdateien und Shadow-Files verwendet werden.

In RDB$FILE_START wird gespeichert, mit welcher Datenbankseite die Datenbankdatei oder der Shadow beginnt, die Länge in Seiten enthält RDB$FILE_ LENGTH. Für einen Shadow beinhaltet RDB$SHADOW_NUMBER, auf welche Datenbankdatei sich die Shadow-Datei bezieht. Handelt es sich um eine sekundäre Datenbankdatei, so ist der Wert null.

Am Rande: Die Tabelle RDB$LOG_FILES ist zwar noch vorhanden, wird aber nicht mehr verwendet.

RDB$PAGES

```
CREATE TABLE "RDB$PAGES"
  ("RDB$PAGE_NUMBER" INTEGER,
  "RDB$RELATION_ID" SMALLINT,
  "RDB$PAGE_SEQUENCE" INTEGER,
  "RDB$PAGE_TYPE" SMALLINT);
```

In der Tabelle RDB$PAGES werden die Datenbankseiten verwaltet. RDB$-RELATION_ ID verweist auf die dazugehörende Tabelle, RDB$PAGE_TYPE spezifiziert, ob es sich um Daten- oder Indexseiten handelt.

Widerstehen Sie der Versuchung, an dieser Tabelle etwas ändern zu wollen, mit ziemlicher Sicherheit würden Sie die Datenbank dadurch unbrauchbar machen!

RDB$DEPENDENCIES

Wenn ein Element, beispielsweise eine VIEW, von einem anderen Element, beispielsweise einer Tabelle, abhängt, dann kann Letzteres nicht gelöscht werden, solange Ersteres existiert. Um diese Abhängigkeiten schnell ermitteln zu können, werden sie in der Tabelle RDB$DEPENDENCIES gespeichert.

```
CREATE TABLE "RDB$DEPENDENCIES"
  ("RDB$DEPENDENT_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DEPENDED_ON_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DEPENDENT_TYPE" SMALLINT,
  "RDB$DEPENDED_ON_TYPE" SMALLINT);
```

Das abhängige Element wird in RDB$DEPENDENT_NAME gespeichert, RDB$-DEPENDENT_TYPE spezifiziert, um was es sich dabei handelt – null wäre beispielsweise eine Tabelle.

Das Element, von dem das andere Element abhängig ist, wird in RDB$DEPENDED_ON_NAME genannt, sein Typ in RDB$DEPENDED_ON_TYPE. Ist das Ziel einer Abhängigkeit eine Spalte, dann wird diese in RDB$FIELD_NAME genannt.

8.2 Tabellen

Eine Tabelle RDB$TABLES werden Sie vergeblich suchen: Entsprechend dem Sprachgebrauch relationaler Datenbanken handelt es sich nicht um Tabellen, sondern um Relationen, dementsprechend nennt sich die Tabelle RDB$RELATIONS.

RDB$RELATIONS

```
CREATE TABLE "RDB$RELATIONS"
  ("RDB$VIEW_BLR" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$VIEW_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$RELATION_ID" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT,
  "RDB$DBKEY_LENGTH" SMALLINT,
  "RDB$FORMAT" SMALLINT,
  "RDB$FIELD_ID" SMALLINT,
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$SECURITY_CLASS" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$EXTERNAL_FILE" VARCHAR(253),
  "RDB$RUNTIME" BLOB SUB_TYPE SUMMARY SEGMENT SIZE 80,
  "RDB$EXTERNAL_DESCRIPTION" BLOB
    SUB_TYPE EXTERNAL_FILE_DESCRIPTION SEGMENT SIZE 80,
  "RDB$OWNER_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DEFAULT_CLASS" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FLAGS" SMALLINT);
```

Alle Tabellen und alle VIEWS werden in dieser Tabelle gespeichert. Die Anordnung der Spalten folgt leider nicht ganz ihrer Priorität.

In der Spalte RDB$RELATION_ID finden wir die ID der Tabelle (oder der VIEW). Da es sich hier um eine Spalte vom Typ SMALLINT handelt, können maximal 65535 Tabellen und VIEWS in einer InterBase-Datenbank gespeichert werden. Den Namen der Tabelle oder VIEW finden wir in der Spalte RDB$RELATION_NAME.

Handelt es sich um eine VIEW, dann wird deren Definition als SQL-Anweisung in der Spalte RDB$VIEW_SOURCE gespeichert, übersetzt in eine binäre Anweisung findet man sie in der Spalte RDB$VIEW_BLR. Wenn Sie als Entwickler Ihre Arbeitsleistung schützen wollen, können Sie die SQL-Anweisung manuell löschen – die VIEW lässt sich dann zwar noch ausführen, ihr Quelltext ist dann für andere jedoch nicht mehr einsehbar.

Wenn es sich um eine Systemtabelle handelt, dann hat RDB$SYSTEM_FLAG den Wert eins, bei benutzererstellten Tabellen den Wert null. Sie sollten diese Information nicht manuell abändern!

Wenn eine Tabelle auf einer externen Datei beruht, dann finden Sie deren Dateinamen in der Spalte RDB$EXTERNAL_FILE, eine optional vergebene Beschreibung in der Spalte RDB$EXTERNAL_DESCRIPTION.

Der Besitzer eine Tabelle ist der Benutzer, der sie erstellt hat, und steht in der Spalte RDB$OWNER_NAME.

RDB$RELATION_FIELDS

In der Tabelle RDB$RELATION_FIELDS finden wir die Spaltendefinitionen der einzelnen Tabellen.

```
CREATE TABLE "RDB$RELATION_FIELDS"
  ("RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FIELD_SOURCE" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$QUERY_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$BASE_FIELD" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$EDIT_STRING" VARCHAR(125),
  "RDB$FIELD_POSITION" SMALLINT,
  "RDB$QUERY_HEADER" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$UPDATE_FLAG" SMALLINT,
  "RDB$FIELD_ID" SMALLINT,
  "RDB$VIEW_CONTEXT" SMALLINT,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$DEFAULT_VALUE" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$SYSTEM_FLAG" SMALLINT,
```

```
  "RDB$SECURITY_CLASS" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$COMPLEX_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$NULL_FLAG" SMALLINT,
  "RDB$DEFAULT_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$COLLATION_ID" SMALLINT);
```

Den Namen der Spalte finden wir in RDB$FIELD_NAME, den Namen den Tabelle in RDB$RELATION_NAME. Es gibt auch eine Nummer für die Spalte, RDB$FIELD_ID, in der Reihenfolge dieser Nummern werden die Spalten physikalisch gespeichert. Wenn InterBase dafür Gründe hat, kann diese Reihenfolge durch ein Restore verändert werden, Sie sollten deshalb davon Abstand nehmen, diesen Wert zu verwenden – vor allem sollten Sie ihn nicht verändern.

Die Reihenfolge, wie die Spalten bei einer *SELECT* *-Anweisung erscheinen, wird durch den Wert der Spalte RDB$FIELD_POSITION spezifiziert, wobei die Spalte mit dem Wert null als Erste erscheint. Sie können diese Spalte bearbeiten, wenn mehrere Spalten derselben Tabelle den gleichen Wert haben, dann ist deren Position zufällig.

Jede Spalte basiert auf einer Domäne, diese ist in RDB$FIELD_SOURCE gespeichert und referenziert die Tabelle RDB$FIELDS. Jede VIEW basiert auch auf einer Spalte in einer Tabelle oder einer anderen VIEW, der Name dieser Spalte wird in RDB$BASE_FIELD gespeichert.

Haben Sie für eine Spalte einen Default-Wert definiert, dann finden Sie die SQL-Anweisung in RDB$DEFAULT_SOURCE und den binären Wert in RDB$-DEFAULT_VALUE. Haben Sie eine Spalte als NOT NULL definiert, dann erhält RDB$NULL_FLAG den Wert eins, ansonsten bleibt er NULL. Eine abweichende Sortierreihenfolge findet man in der Spalte RDB$COLLATION_ID.

RDB$RELATION_CONSTRAINTS

Gültigkeitsprüfungen im weitesten Sinne finden sich in der Tabelle RDB$-RELATION_ CONSTRAINTS.

```
CREATE TABLE "RDB$RELATION_CONSTRAINTS"
  ("RDB$CONSTRAINT_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$CONSTRAINT_TYPE" CHAR(11),
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DEFERRABLE" CHAR(3),
  "RDB$INITIALLY_DEFERRED" CHAR(3),
  "RDB$INDEX_NAME" CHAR(67) CHARACTER SET UNICODE_FSS);
```

Den Namen der Gültigkeitsprüfung finden Sie in RDB$CONSTRAINT_NAME, den der dazugehörenden Tabelle in RDB$RELATION_NAME.

Für RDB$CONSTRAINT_TYPE sind die folgenden Werte vorgesehen:

- PRIMARY für Primärschlüssel
- UNIQUE für Sekundärschlüssel
- FOREIGN KEY für Fremdschlüssel
- PCHECK für selbst definierte Gültigkeitsprüfungen und
- NOT NULL für zwingende Eingabe.

Primär-, Sekundär- und Fremdschlüssel bedienen sich eines Indexes, der in RDB$INDEX_NAME referenziert wird.

RDB$INDICES

In der Tabelle RDB$INDICES finden wir Informationen über alle Indizes:

```
CREATE TABLE "RDB$INDICES"
  ("RDB$INDEX_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$INDEX_ID" SMALLINT,
  "RDB$UNIQUE_FLAG" SMALLINT,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$SEGMENT_COUNT" SMALLINT,
  "RDB$INDEX_INACTIVE" SMALLINT,
  "RDB$INDEX_TYPE" SMALLINT,
  "RDB$FOREIGN_KEY" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$SYSTEM_FLAG" SMALLINT,
  "RDB$EXPRESSION_BLR" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$EXPRESSION_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$STATISTICS" DOUBLE PRECISION);
```

Den Namen des Indexes finden wir in Spalte RDB$INDEX_NAME, den der Tabelle in Spalte RDB$RELATION_NAME. Die zu einer Tabelle gehörenden Indizes werden in der Spalte RDB$INDEX_ID jeweils mit eins beginnend durchnummeriert. Handelt es sich um einen eindeutigen Index, dann hat RDB$UNIQUE_FLAG den Wert eins, andernfalls den Wert null.

RDB$INDEX_TYPE spezifiziert, ob es sich um einen aufsteigenden (null) oder absteigenden (eins) Index handelt, aktive Indizes haben in der Spalte RDB$INDEX_INACTIVE den Wert null, inaktive den Wert eins.

Ein Fremdschlüssel referenziert immer eine Spalte, die eindeutig ist, die also einen Primär- oder Sekundärschlüssel verwendet. Der Name dieses Schlüssels findet sich bei Fremdschlüsseln in der Spalte RDB$FOREIGN_KEY.

Der Query-Optimizer entscheidet anhand der Selektivität eines Indexes, ob er den Index verwendet oder ob er lieber alle Datensätze liest. Diese Selektivität wird in der Spalte RDB$STATISTICS gespeichert und immer dann aktualisiert, wenn der Index erzeugt oder aktiv geschaltet wird. Man kann die Selektivität auch mit SET STATISTICS ermitteln, allerdings ist es meist effizienter, gleich den Index neu aufzubauen.

RDB$INDEX_SEGMENTS

Die Spalten, welche den Index bilden, werden in RDB$INDEX_SEGMENTS gespeichert.

```
CREATE TABLE "RDB$INDEX_SEGMENTS"
  ("RDB$INDEX_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FIELD_POSITION" SMALLINT);
```

RDB$INDEX_NAME ist der Name des Indexes, RDB$FIELD_NAME der Name der Spalte und die Reihenfolge der einzelnen Spalten ergibt sich aus RDB$FIELD_POSITION.

Wie man leicht erkennt, kann nur ein gesamter Index auf- oder absteigend sein, »gemischte« Indizes sind in der SQL-Syntax nicht vorgesehen und könnten in den Systemtabellen auch gar nicht gespeichert werden.

RDB$REF_CONSTRAINTS

Wie Schlüsselverletzungen in Fremdschlüsseln zu behandeln sind, speichert die Tabelle RDB$REF_CONSTRAINTS.

```
CREATE TABLE "RDB$REF_CONSTRAINTS"
  ("RDB$CONSTRAINT_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$CONST_NAME_UQ" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$MATCH_OPTION" CHAR(7),
  "RDB$UPDATE_RULE" CHAR(11),
  "RDB$DELETE_RULE" CHAR(11));
```

Den Namen des Fremdschlüssels findet man in RDB$CONSTRAINT_NAME, den des dazugehörenden Primär- oder Sekundärschlüssels in RDB$CONST_NAME_UQ.

Wie sich InterBase im Falle einer Fremdschlüsselverletzung verhalten soll, lässt sich für den Delete- und Update-Fall getrennt einstellen und wird in den Spalten RDB$UPDATE_RULE und RDB$DELETE_RULE gespeichert. Vorgesehen dafür sind der Vorgabewert RESTRICT sowie die Alternativen NO ACTION, CASCADE, SET NULL und SET DEFAULT.

RDB$CHECK_CONSTRAINTS

In RDB$CHECK_CONSTRAINTS werden die Gültigkeitsprüfungen gespeichert.

```
CREATE TABLE "RDB$CHECK_CONSTRAINTS"
  ("RDB$CONSTRAINT_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$TRIGGER_NAME" CHAR(67) CHARACTER SET UNICODE_FSS);
```

Gültigkeitsprüfungen werden über Trigger realisiert, der Name des Triggers steht in RDB$TRIGGER_NAME, der Name der Gültigkeitsprüfung in RDB$CONSTRAINT_NAME.

Wenn in RDB$TRIGGER_NAME der Name einer Spalte statt eines Triggers steht, dann handelt es sich um eine NOT NULL-Prüfung, diese werden auch in dieser Tabelle gespeichert.

RDB$VIEW_RELATIONS

Die Tabellen, die zu einer VIEW gehören, werden in RDB$VIEW_RELATIONS gespeichert.

```
CREATE TABLE "RDB$VIEW_RELATIONS"
  ("RDB$VIEW_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$VIEW_CONTEXT" SMALLINT,
  "RDB$CONTEXT_NAME" CHAR(67) CHARACTER SET UNICODE_FSS);
```

RDB$VIEW_NAME bezeichnet die VIEW, RDB$RELATION_NAME die jeweilige Tabelle, die einzelnen Tabellen werden mit RDB$VIEW_CONTEXT durchnummeriert. Wird ein Tabellen-Alias verwendet, dann steht dieser in RDB$CONTEXT_NAME.

8.3 Domänen

RDB$FIELDS

Die Definition aller Domänen finden Sie in der Systentabelle RDB$FIELDS.

```
CREATE TABLE "RDB$FIELDS"
  ("RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$QUERY_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$VALIDATION_BLR" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$VALIDATION_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$COMPUTED_BLR" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$COMPUTED_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
```

```
  "RDB$DEFAULT_VALUE" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$DEFAULT_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$FIELD_LENGTH" SMALLINT,
  "RDB$FIELD_SCALE" SMALLINT,
  "RDB$FIELD_TYPE" SMALLINT,
  "RDB$FIELD_SUB_TYPE" SMALLINT,
  "RDB$MISSING_VALUE" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$MISSING_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$SYSTEM_FLAG" SMALLINT,
  "RDB$QUERY_HEADER" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$SEGMENT_LENGTH" SMALLINT,
  "RDB$EDIT_STRING" VARCHAR(125),
  "RDB$EXTERNAL_LENGTH" SMALLINT,
  "RDB$EXTERNAL_SCALE" SMALLINT,
  "RDB$EXTERNAL_TYPE" SMALLINT,
  "RDB$DIMENSIONS" SMALLINT,
  "RDB$NULL_FLAG" SMALLINT,
  "RDB$CHARACTER_LENGTH" SMALLINT,
  "RDB$COLLATION_ID" SMALLINT,
  "RDB$CHARACTER_SET_ID" SMALLINT,
  "RDB$FIELD_PRECISION" SMALLINT);
```

Den Namen der Domäne finden Sie in RDB$FIELD_NAME. Wenn Sie bei einer Tabellen-Definition für die Spalten keine Domänen, sondern einfache Typen verwenden, dann legt InterBase automatisch Domänen an, und zwar für jede Spalte eine eigene. Diese beginnen mit RDB$ und bestehen dann aus einer durchlaufenden Nummer.

Handelt es sich um eine berechnete Spalte, dann finden Sie die dazugehörende SQL-Anweisung in der Spalte RDB$COMPUTED_SOURCE und den Binärcode in RDB$COMPUTED_BLR. Dasselbe finden Sie für Default-Werte in den Spalten RDB$DEFAULT_SOURCE und RDB$DEFAULT_VALUE.

Den zugrunde liegenden Typ finden Sie als Integer-Wert in der Spalte RDB$FIELD_TYPE, die Klartextnamen dafür sind in der Tabelle RDB$TYPES hinterlegt. Der Länge der Spalten – relevant vor allem für CHAR und VARCHAR – finden Sie in RDB$FIELD_LENGTH. Handelt es sich um die Datentypen DECIMAL oder NUMERIC, so finden Sie *Precision* und *Scale* in den Spalten RDB$FIELD_ PRECISION und RDB$FIELD_SCALE.

Bei BLOBS finden Sie in RDB$FIELD_SUB_TYPE den BLOB-Typ und in RDB$SEGMENT_LENGTH die Länge für den Buffer des BLOBS – keine Sorge, InterBase stellt automatisch mehr Speicher bereit, wenn das nicht ausreichen sollte.

Hat RDB$NULL_FLAG den Wert 1, dann ist die Domäne als NOT NULL definiert. Werden im Zuge einer Tabellendefinition automatisch Domänen angelegt, dann werden diese grundsätzlich nicht als NOT NULL gespeichert, gegebenenfalls vorhandene NOT NULL-Klauseln werden in RDB$RELATION_FIELDS gespeichert.

Der Zeichensatz wird in RDB$CHARACTER_SET_ID gespeichert, die Sortierreihenfolge in RDB$COLLATION_ID. In RDB$CHARACTER_LENGTH finden Sie die Anzahl der zulässigen Zeichen, bei Mehr-Byte-Zeichensätzen kann das von RDB$FIELD_LENGTH abweichen.

RDB$TYPES

Alles, was es in InterBase-Datenbanken so an Typen gibt, wird in RDB$TYPES aufgelistet.

```
CREATE TABLE "RDB$TYPES"
  ("RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$TYPE" SMALLINT,
  "RDB$TYPE_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

Die Nummer des Typs finden Sie in RDB$TYPE, seinen Namen in RDB$TYPE_NAME.

In der Tabelle finden sich nicht nur Datentypen, sondern auch die Typen von Elementen (VIEW, TRIGGER, PROCEDURE), von Zeichensätzen und einigem mehr. Die folgende Abbildung gibt hier einen Überblick:

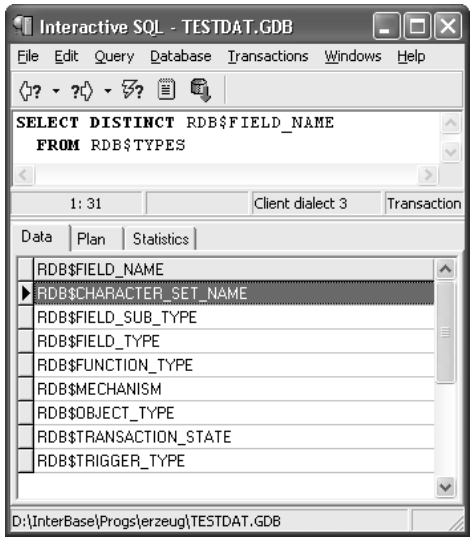


Abbildung 8.1: Die einzelnen Typ-Kategorien

RDB$FIELD_DIMENSIONS

Array-Definitionen werden in RDB$FIELD_DIMENSIONS gespeichert.

```
CREATE TABLE "RDB$FIELD_DIMENSIONS"
  ("RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DIMENSION" SMALLINT,
  "RDB$LOWER_BOUND" INTEGER,
  "RDB$UPPER_BOUND" INTEGER);
```

RDB$FIELD_NAME referenziert die Spalte in der Tabelle RDB$FIELDS. Die einzelnen Dimensionen des Arrays werden in RDB$DIMENSION hochgezählt, das jeweils unterste und oberste Feld wird in RDB$LOWER_BOUND und RDB$UPPER_BOUND gespeichert.

RDB$CHARACTER_SETS

Die Liste der Zeichensätze findet man in RDB$CHARACTER_SETS.

```
CREATE TABLE "RDB$CHARACTER_SETS"
  ("RDB$CHARACTER_SET_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FORM_OF_USE" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$NUMBER_OF_CHARACTERS" INTEGER,
  "RDB$DEFAULT_COLLATE_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$CHARACTER_SET_ID" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$FUNCTION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$BYTES_PER_CHARACTER" SMALLINT);
```

Den Namen des Zeichensatzes finden Sie in RDB$CHARACTER_SET_NAME, die Anzahl der Bytes pro Zeichen in RDB$BYTES_PER_CHARACTER.

RDB$COLLATIONS

Eine Liste der Sortierreihenfolgen finden Sie in RDB$COLLATIONS.

```
CREATE TABLE "RDB$COLLATIONS"
  ("RDB$COLLATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$COLLATION_ID" SMALLINT,
  "RDB$CHARACTER_SET_ID" SMALLINT,
  "RDB$COLLATION_ATTRIBUTES" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$FUNCTION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS);
```

Den Namen der Sortierreihenfolge finden Sie in RDB$COLLATION_NAME. Zu jeder Sortierreihenfolge gehört ein Zeichensatz, der mit RDB$CHARACTER_SET_ID referenziert wird. Die einzelnen Sortierreihenfolgen pro Zeichensatz werden in RDB$COLLATION_ID mit null beginnend durchnummeriert.

8.4 Prozeduren und Trigger

RDB$PROCEDURES

Alle STORED PROCEDURES werden in RDB$PROCEDURES gespeichert.

```
CREATE TABLE "RDB$PROCEDURES"
  ("RDB$PROCEDURE_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$PROCEDURE_ID" SMALLINT,
  "RDB$PROCEDURE_INPUTS" SMALLINT,
  "RDB$PROCEDURE_OUTPUTS" SMALLINT,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$PROCEDURE_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$PROCEDURE_BLR" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$SECURITY_CLASS" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$OWNER_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$RUNTIME" BLOB SUB_TYPE SUMMARY SEGMENT SIZE 80,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

Den Namen der Prozedur finden wir in RDB$PROCEDURE_NAME, eine durchlaufende Nummer in RDB$PROCEDURE_ID. Die Zahl der an die Prozedur übergebenen Parameter steht in RDB$PROCEDURE_INPUTS, die von den Prozedur zurückgegebenen Werte stehen in RDB$PROCEDURE_OUTPUTS.

Der Quelltext der Prozedur wird in RDB$PROCEDURE_SOURCE gespeichert, seine binäre Übersetzung in RDB$PROCEDURE_BLR. Sie haben die Möglichkeit, den Quelltext manuell zu entfernen, wenn Sie verhindern wollen, dass Dritte ihn einsehen können.

Den Besitzer einer Prozedur beinhaltet RDB$OWNER_NAME, dieser Benutzer darf (neben SYSDBA) auch die Rechte an der Prozedur vergeben.

RDB$PROCEDURE_PARAMETERS

In RDB$PROCEDURES wird nur die Anzahl der Ein- und Ausgabeparameter gespeichert. Informationen über die einzelnen Parameter finden wir in RDB$PROCEDURE_PARAMETERS.

```
CREATE TABLE "RDB$PROCEDURE_PARAMETERS"
  ("RDB$PARAMETER_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$PROCEDURE_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$PARAMETER_NUMBER" SMALLINT,
  "RDB$PARAMETER_TYPE" SMALLINT,
  "RDB$FIELD_SOURCE" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

In RDB$PARAMETER_NAME steht der Name des Parameters, in RDB$PROCEDURE_NAME der Name der Prozedur, in der er verwendet wird. Die einzelnen Parameter werden in RDB$PARAMETER_NUMBER mit null beginnend durchnummeriert.

Hat RDB$PARAMETER_TYPE den Wert null, handelt es sich um einen Eingabeparameter, ein Ausgabeparameter ist durch den Wert eins gekennzeichnet. Der Typ des Parameters wird durch RDB$PARAMETER_TYPE spezifiziert, diese Spalte referenziert die Tabelle RDB$FIELDS.

RDB$TRIGGERS

Die Liste aller Trigger beinhaltet RDB$TRIGGERS.

```
CREATE TABLE "RDB$TRIGGERS"
  ("RDB$TRIGGER_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$TRIGGER_SEQUENCE" SMALLINT,
  "RDB$TRIGGER_TYPE" SMALLINT,
  "RDB$TRIGGER_SOURCE" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$TRIGGER_BLR" BLOB SUB_TYPE BLR SEGMENT SIZE 80,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$TRIGGER_INACTIVE" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT,
  "RDB$FLAGS" SMALLINT);
```

Der Name des Triggers steht in RDB$TRIGGER_NAME, der Name der dazugehörenden Tabelle in RDB$RELATION_NAME. Mit RDB$TRIGGER_TYPE wird spezifiziert, wann der Trigger ausgelöst wird:

1) BEFORE INSERT
2) AFTER INSERT
3) BEFORE UPDATE
4) AFTER UPDATE

5) BEFORE DELETE
6) AFTER DELETE

Wenn mehrere Trigger für die gleiche Tabelle denselben Wert für RDB$TRIGGER_TYPE haben, dann entscheidet RDB$TRIGGER_SEQUENCE über die Reihenfolge der Ausführung – Trigger mit niedrigeren Werten werden zuerst ausgeführt. Bei Gleichheit auch in dieser Spalte entscheidet die alphabetische Reihenfolge des Triggernamens.

In RDB$TRIGGER_SOURCE steht der Quelltext des Triggers, in RDB$TRIGGER_BLR seine binäre Übersetzung. Auch hier kann der Quelltext manuell gelöscht werden. Wird RDB$TRIGGER_INACTIVE auf eins gesetzt, dann ist der Trigger deaktiviert.

8.5 Rechteverwaltung

Die User werden auf Server-Ebene angemeldet und in der Datenbank *admin.db* gespeichert.

RDB$ROLES

```
CREATE TABLE "RDB$ROLES"
  ("RDB$ROLE_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$OWNER_NAME" CHAR(67) CHARACTER SET UNICODE_FSS);
```

In der Tabelle RDB$ROLES gibt es nur den Namen der Rolle und den Benutzer, der diese Rolle definiert hat.

RDB$USER_PRIVILEGES

Die vergebenen Rechte stehen in der Systemtabelle RDB$USER_PRIVILEGES.

```
CREATE TABLE "RDB$USER_PRIVILEGES"
  ("RDB$USER" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$GRANTOR" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$PRIVILEGE" CHAR(6),
  "RDB$GRANT_OPTION" SMALLINT,
  "RDB$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FIELD_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$USER_TYPE" SMALLINT,
  "RDB$OBJECT_TYPE" SMALLINT);
```

RDB$USER ist der Benutzer, der das Recht erhält, RDB$GRANTOR der Benutzer, der das Recht vergibt. Um welches Recht es sich handelt, wird mit einem Zeichen in RDB$PRIVILEGE abgekürzt: I steht beispielsweise für INSERT, U für UPDATE und X für EXECUTE. Steht in RDB$GRANT_OPTION der Wert eins, dann darf dieses Recht weitergegeben werden.

In RDB$RELATION_NAME ist angegeben, auf welche Tabelle oder welche Prozedur sich das Recht bezieht, ist es auf eine bestimmte Spalte beschränkt, dann wird diese in RDB$FIELD_NAME spezifiziert.

8.6 Sonstiges

RDB$EXCEPTIONS

Eine Liste der Exceptions findet sich in RDB$EXCEPTIONS.

```
CREATE TABLE "RDB$EXCEPTIONS"
  ("RDB$EXCEPTION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$EXCEPTION_NUMBER" INTEGER,
  "RDB$MESSAGE" VARCHAR(78),
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

RDB$EXCEPTION_NAME ist der Name der Exception, RDB$MESSAGE die dazugehörende Fehlermeldung.

RDB$FILTERS

Haben Sie eigene BLOB-Filter definiert, dann werden diese in RDB$FILTERS aufgelistet.

```
CREATE TABLE "RDB$FILTERS"
  ("RDB$FUNCTION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$MODULE_NAME" VARCHAR(253),
  "RDB$ENTRYPOINT" CHAR(67),
  "RDB$INPUT_SUB_TYPE" SMALLINT,
  "RDB$OUTPUT_SUB_TYPE" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

Ein Blob-Filter ist eine Routine in einer externen DLL. Die Routine wird mit RDB$ENTRYPOINT spezifiziert, der Dateiname der DLL mit RDB$MODULE_NAME.

RDB$FUNCTIONS

Wenn Sie USER DEFINED FUNCTIONS (UDF) einbinden, dann werden diese in RDB$FUNCTIONS gespeichert.

```
CREATE TABLE "RDB$FUNCTIONS"
  ("RDB$FUNCTION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$FUNCTION_TYPE" SMALLINT,
  "RDB$QUERY_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$DESCRIPTION" BLOB SUB_TYPE TEXT SEGMENT SIZE 80
    CHARACTER SET UNICODE_FSS,
  "RDB$MODULE_NAME" VARCHAR(253),
  "RDB$ENTRYPOINT" CHAR(67),
  "RDB$RETURN_ARGUMENT" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

Eine UDF ist eine Routine in einer externen DLL. Die Routine wird mit RDB$ENTRYPOINT spezifiziert, der Dateiname der DLL mit RDB$MODULE_NAME. RDB$RETURN_ARGUMENT spezifiziert, welcher der Parameter der Rückgabewert ist.

RDB$FUNCTION_ARGUMENTS

Die einzelnen Parameter einer UDF werden in RDB$FUNCTION_ARGUMENTS aufgeführt.

```
CREATE TABLE "RDB$FUNCTION_ARGUMENTS"
  ("RDB$FUNCTION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$ARGUMENT_POSITION" SMALLINT,
  "RDB$MECHANISM" SMALLINT,
  "RDB$FIELD_TYPE" SMALLINT,
  "RDB$FIELD_SCALE" SMALLINT,
  "RDB$FIELD_LENGTH" SMALLINT,
  "RDB$FIELD_SUB_TYPE" SMALLINT,
  "RDB$CHARACTER_SET_ID" SMALLINT,
  "RDB$FIELD_PRECISION" SMALLINT,
  "RDB$CHARACTER_LENGTH" SMALLINT);
```

Der Name der Funktion steht in RDB$FUNCTION_NAME, die einzelnen Parameter werden mit RDB$ARGUMENT_POSITION durchnumeriert.

Die Typen der einzelnen Parameter stehen in RDB$FIELD_TYPE, diese Spalte referenziert die Tabelle RDB$TYPES. Steht in RDB$MECHANISM der Wert null, dann wird der Parameter *by value* übergeben, bei einer eins *by reference*. Die Länge steht in RDB$FIELD_LENGTH und ist insbesondere bei Strings interessant, in RDB$CHARACTER_SET_ID wird dabei der Zeichensatz angegeben.

RDB$GENERATORS

```
CREATE TABLE "RDB$GENERATORS"
  ("RDB$GENERATOR_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "RDB$GENERATOR_ID" SMALLINT,
  "RDB$SYSTEM_FLAG" SMALLINT);
```

Ein Generator besteht im Wesentlichen aus seinem Namen und einer eindeutigen Nummer. Der Generatorwert wird nicht in der Systemtabelle gespeichert.

RDB$TRANSACTIONS

Werden Transaktionen über mehrere Datenbanken erstellt, dann werden diese in RDB$TRANSACTIONS gespeichert.

```
CREATE TABLE "RDB$TRANSACTIONS"
  ("RDB$TRANSACTION_ID" INTEGER,
  "RDB$TRANSACTION_STATE" SMALLINT,
  "RDB$TIMESTAMP" TIMESTAMP,
  "RDB$TRANSACTION_DESCRIPTION" BLOB
    SUB_TYPE TRANSACTION_DESCRIPTION SEGMENT SIZE 80);
```

In RDB$TRANSACTIONS wird der Status der Multi-Database-Transaktion gespeichert: Bei null ist die Transaktion *limbo*, also noch nicht abgeschlossen, bei eins mit COMMIT und bei zwei mit ROLLBACK beendet.

8.7 Die temporären Tabellen

Die temporären Tabellen werden nur im Speicher gehalten. In ihnen werden Informationen wie die Liste aller aktuellen Verbindungen gehalten. Auch hier wollen wir uns auf einige Tabellen beschränken.

TMP$DATABASE

Die Tabelle TMP$DATABASE enthält eine Liste aller Datenbanken, zu denen vom Client aus eine Verbindung besteht.

```
CREATE TABLE "TMP$DATABASE"
  ("TMP$DATABASE_ID" INTEGER,
  "TMP$DATABASE_PATH" VARCHAR(253),
  "TMP$ATTACHMENTS" SMALLINT,
  "TMP$STATEMENTS" SMALLINT,
  "TMP$ALLOCATED_PAGES" INTEGER,
  "TMP$POOLS" INTEGER,
  "TMP$PROCEDURES" SMALLINT,
  "TMP$RELATIONS" SMALLINT,
  "TMP$TRIGGERS" SMALLINT,
  "TMP$ACTIVE_THREADS" SMALLINT,
  "TMP$SORT_MEMORY" INTEGER,
  "TMP$CURRENT_MEMORY" INTEGER,
  "TMP$MAXIMUM_MEMORY" INTEGER,
  "TMP$PERMANENT_POOL_MEMORY" INTEGER,
  "TMP$CACHE_POOL_MEMORY" INTEGER,
```

```
  "TMP$TRANSACTIONS" SMALLINT,
  "TMP$TRANSACTION_COMMITS" INTEGER,
  "TMP$TRANSACTION_ROLLBACKS" INTEGER,
  "TMP$TRANSACTION_PREPARES" INTEGER,
  "TMP$TRANSACTION_DEADLOCKS" INTEGER,
  "TMP$TRANSACTION_CONFLICTS" INTEGER,
  "TMP$TRANSACTION_WAITS" INTEGER,
  "TMP$NEXT_TRANSACTION" INTEGER,
  "TMP$OLDEST_INTERESTING" INTEGER,
  "TMP$OLDEST_ACTIVE" INTEGER,
  "TMP$OLDEST_SNAPSHOT" INTEGER,
  "TMP$CACHE_BUFFERS" INTEGER,
  "TMP$CACHE_PRECEDENCE" INTEGER,
  "TMP$CACHE_LATCH_WAITS" INTEGER,
  "TMP$CACHE_FREE_WAITS" INTEGER,
  "TMP$CACHE_FREE_WRITES" INTEGER,
  "TMP$SWEEP_INTERVAL" INTEGER,
  "TMP$SWEEP_ACTIVE" CHAR(1),
  "TMP$SWEEP_RELATION" CHAR(67) CHARACTER SET UNICODE_FSS,
  "TMP$SWEEP_RECORDS" INTEGER,
  "TMP$PAGE_READS" INTEGER,
  "TMP$PAGE_WRITES" INTEGER,
  "TMP$PAGE_FETCHES" INTEGER,
  "TMP$PAGE_MARKS" INTEGER,
  "TMP$RECORD_SELECTS" INTEGER,
  "TMP$RECORD_INSERTS" INTEGER,
  "TMP$RECORD_UPDATES" INTEGER,
  "TMP$RECORD_DELETES" INTEGER,
  "TMP$RECORD_PURGES" INTEGER,
  "TMP$RECORD_EXPUNGES" INTEGER,
  "TMP$RECORD_BACKOUTS" INTEGER);
```

TMP$DATABASE_ID ist die ID des Datensatzes, TMP$DATABASE_PATH der Pfad der primären Datenbankdatei. TMP$ATTACHMENTS ist die Zahl der aktuellen Datenbankverbindungen, diese werden in TMP$ATTACHMENTS aufgelistet, TMP$STATEMENTS ist die Zahl der momentan ausgeführten SQL-Anweisungen, diese sind in TMP$STATEMENTS gelistet.

TMP$ALLOCATED_PAGES ist die Zahl der Datenbankseiten in allen Datenbankdateien, TMP$POOLS die Zahl der Speicher-Pools, TMP$PROCEDURES zählt die geladenen Prozeduren, TMP$RELATIONS die geladenen Tabellen einschließlich der Systemtabellen, TMP$TRIGGERS ist die Anzahl der geladenen Trigger und TMP$ACTIVE_THREADS die Anzahl der aktiven Threads.

In TMP$SORT_MEMORY steht, wie viel Speicher für den Sortier-Puffer alloziert wurde, TMP$CURRENT_MEMORY ist der aktuelle Speicherbedarf der Daten-

bank insgesamt, TMP$MAXIMUM_MEMORY der maximale Speicherbedarf während dieser Datenbanksitzung.

Die Zahl der aktiven – also noch nicht mit COMMIT oder ROLLBACK beendeten – Transaktionen steht in TMP$TRANSACTIONS, TMP$TRANSACTION_COMMITS ist die Zahl der bestätigten, TMP$TRANSACTION_ROLLBACKS die Zahl der verworfenen Transaktionen.

Die nächste zu vergebende Transaktionsnummer ist TMP$NEXT_TRANSACTION, die Nummer der ältesten noch aktiven Transaktion ist TMP$OLDEST_ACTIVE, die der ältesten aktiven SNAPSHOT-Transaktion TMP$OLDEST_SNAPSHOT.

Es gibt noch eine Reihe weiterer statistischer Daten, beispielsweise die Zahl der Lese- und Schreibvorgänge.

TMP$ATTACHMENTS

Alle Datenbankverbindungen werden in TMP$ATTACHMENTS gespeichert.

```
CREATE TABLE "TMP$ATTACHMENTS"
  ("TMP$ATTACHMENT_ID" INTEGER,
  "TMP$DATABASE_ID" INTEGER,
  "TMP$POOL_ID" INTEGER,
  "TMP$POOL_MEMORY" INTEGER,
  "TMP$STATEMENTS" SMALLINT,
  "TMP$TRANSACTIONS" SMALLINT,
  "TMP$TIMESTAMP" TIMESTAMP,
  "TMP$QUANTUM" INTEGER,
  "TMP$USER" CHAR(67) CHARACTER SET UNICODE_FSS,
  "TMP$USER_IP_ADDR" CHAR(31),
  "TMP$USER_HOST" CHAR(31),
  "TMP$USER_PROCESS" INTEGER,
  "TMP$STATE" CHAR(31),
  "TMP$PRIORITY" CHAR(31),
  "TMP$DBKEY_ID" INTEGER,
  "TMP$ACTIVE_SORTS" SMALLINT,
  "TMP$PAGE_READS" INTEGER,
  "TMP$PAGE_WRITES" INTEGER,
  "TMP$PAGE_FETCHES" INTEGER,
  "TMP$PAGE_MARKS" INTEGER,
  "TMP$RECORD_SELECTS" INTEGER,
  "TMP$RECORD_INSERTS" INTEGER,
  "TMP$RECORD_UPDATES" INTEGER,
  "TMP$RECORD_DELETES" INTEGER,
  "TMP$RECORD_PURGES" INTEGER,
  "TMP$RECORD_EXPUNGES" INTEGER,
  "TMP$RECORD_BACKOUTS" INTEGER);
```

TMP$ATTACHMENT_ID ist die Nummer der Datenbankverbindung, TMP$-DATABASE_ID spezifiziert, auf welche Datenbank sie sich bezieht. Die Zahl der Transaktionen und Anweisungen findet man in TMP$TRANSACTIONS und TMP$STATEMENTS. In TMP$TIMESTAMP wird notiert, wann die Verbindung aufgebaut wurde.

Der Name des Benutzers wird in TMP$USER abgelegt, seine IP-Adresse in TMP$USER_ IP_ADDR und der Host-Name seines Rechners in TMP$USER_HOST. Bei einer lokalen Verbindung ist TMP$USER_IP_ADDR gleich NULL.

Die Zahl der Datenmengen im Sortier-Puffer findet man in TMP$ACTIVE_SORTS. Es finden sich dann noch einige weitere statistische Daten, die Zahl der Lese- und Schreibvorgänge, die Zahl der INSERT-Statements und vieles andere mehr.

TMP$RELATIONS

Alle Tabellen, die seit dem Beginn der Datenbankverbindung verwendet wurden, werden in TMP$RELATIONS gelistet.

```
CREATE TABLE "TMP$RELATIONS"
  ("TMP$RELATION_ID" SMALLINT,
  "TMP$DATABASE_ID" INTEGER,
  "TMP$RELATION_NAME" CHAR(67) CHARACTER SET UNICODE_FSS,
  "TMP$USE_COUNT" SMALLINT,
  "TMP$SWEEP_COUNT" SMALLINT,
  "TMP$SCAN_COUNT" INTEGER,
  "TMP$FORMATS" SMALLINT,
  "TMP$POINTER_PAGES" INTEGER,
  "TMP$DATA_PAGES" INTEGER,
  "TMP$GARBAGE_COLLECT_PAGES" INTEGER,
  "TMP$PAGE_READS" INTEGER,
  "TMP$PAGE_WRITES" INTEGER,
  "TMP$PAGE_FETCHES" INTEGER,
  "TMP$PAGE_MARKS" INTEGER,
  "TMP$RECORD_IDX_SELECTS" INTEGER,
  "TMP$RECORD_SEQ_SELECTS" INTEGER,
  "TMP$RECORD_INSERTS" INTEGER,
  "TMP$RECORD_UPDATES" INTEGER,
  "TMP$RECORD_DELETES" INTEGER,
  "TMP$RECORD_PURGES" INTEGER,
  "TMP$RECORD_EXPUNGES" INTEGER,
  "TMP$RECORD_BACKOUTS" INTEGER);
```

Alle verwendeten Tabellen erhalten eine eindeutige Nummer, es wird vermerkt, zu welcher Datenbank sie gehören, und erwartungsgemäß wird auch der Tabellenname vermerkt.

TMP$USE_COUNT ist die Anzahl der Statements, welche mit der betreffenden Tabelle gearbeitet haben und TMP$SWEEP_COUNT die Zahl der Aufräumvorgänge. Die Zahl der Datenseiten pro Tabelle finden Sie in TMP$DATA_PAGES.

Von den statistischen Daten sind vor allem TMP$RECORD_IDX_SELECTS und TMP$RECORD_SEQ_SELECTS interessant: Das erste ist die Zahl der Datensätze, die über einen Index gelesen wurden, das zweite die Zahl der Datensätze, die sequenziell gelesen wurden. Anhand dieses Verhältnisses kann man abschätzen, ob ein Index sinnvoll ist.

TMP$STATEMENTS

In TMP$STATEMENTS werden alle aktuellen Anweisungen gelistet.

```
CREATE TABLE "TMP$STATEMENTS"
  ("TMP$STATEMENT_ID" INTEGER,
  "TMP$ATTACHMENT_ID" INTEGER,
  "TMP$TRANSACTION_ID" INTEGER,
  "TMP$SQL" VARCHAR(4094),
  "TMP$POOL_ID" INTEGER,
  "TMP$POOL_MEMORY" INTEGER,
  "TMP$CLONE" SMALLINT,
  "TMP$TIMESTAMP" TIMESTAMP,
  "TMP$QUANTUM" INTEGER,
  "TMP$INVOCATIONS" INTEGER,
  "TMP$STATE" CHAR(31),
  "TMP$PRIORITY" CHAR(31),
  "TMP$PAGE_READS" INTEGER,
  "TMP$PAGE_WRITES" INTEGER,
  "TMP$PAGE_FETCHES" INTEGER,
  "TMP$PAGE_MARKS" INTEGER,
  "TMP$RECORD_SELECTS" INTEGER,
  "TMP$RECORD_INSERTS" INTEGER,
  "TMP$RECORD_UPDATES" INTEGER,
  "TMP$RECORD_DELETES" INTEGER,
  "TMP$RECORD_PURGES" INTEGER,
  "TMP$RECORD_EXPUNGES" INTEGER,
  "TMP$RECORD_BACKOUTS" INTEGER);
```

Das Statement erhält eine eindeutige Nummer, es wird vermerkt, zu welcher Verbindung es gehört (daraus ergibt sich dann auch die Datenbank), auch die Transaktion wird gespeichert, ist diese Spalte NULL, dann handelt es sich um ein Systemtabellen-Statement, das intern abgesetzt wurde.

In TMP$STATE wird der Status der Anweisung vermerkt. Es gibt ACTIVE, INACTIVE, STALLED (»abgewürgt«) und CANCELLED (aufgehoben). Es folgt dann wieder der Satz statistischer Daten.

TMP$TRANSACTIONS

Auch für die Liste aller offenen oder schwebenden Transaktionen gibt es eine Tabelle:

```
CREATE TABLE "TMP$TRANSACTIONS"
  ("TMP$TRANSACTION_ID" INTEGER,
  "TMP$ATTACHMENT_ID" INTEGER,
  "TMP$POOL_ID" INTEGER,
  "TMP$POOL_MEMORY" INTEGER,
  "TMP$TIMESTAMP" TIMESTAMP,
  "TMP$SNAPSHOT" INTEGER,
  "TMP$QUANTUM" INTEGER,
  "TMP$SAVEPOINTS" INTEGER,
  "TMP$READONLY" CHAR(1),
  "TMP$WRITE" CHAR(1),
  "TMP$NOWAIT" CHAR(1),
  "TMP$COMMIT_RETAINING" CHAR(1),
  "TMP$STATE" CHAR(31),
  "TMP$PRIORITY" CHAR(31),
  "TMP$TYPE" CHAR(31),
  "TMP$PAGE_READS" INTEGER,
  "TMP$PAGE_WRITES" INTEGER,
  "TMP$PAGE_FETCHES" INTEGER,
  "TMP$PAGE_MARKS" INTEGER,
  "TMP$RECORD_SELECTS" INTEGER,
  "TMP$RECORD_INSERTS" INTEGER,
  "TMP$RECORD_UPDATES" INTEGER,
  "TMP$RECORD_DELETES" INTEGER,
  "TMP$RECORD_PURGES" INTEGER,
  "TMP$RECORD_EXPUNGES" INTEGER,
  "TMP$RECORD_BACKOUTS" INTEGER);
```

Erste Aufmerksamkeit ist auf die Spalte TMP$TIMESTAMP zu richten: Transaktionen sollten in Mehrbenutzerumgebungen möglichst kurz dauern, damit der Server für alte Transaktionen keine abweichenden Datensatzversionen vorhalten muss. Idealerweise dauern Transaktionen einige Sekunden, sie sollten nicht (!) über die Laufzeit der Datenbankanwendung gehen.

Einige CHAR[1]-Spalten erhalten die Werte Y (*Yes*) und N (*No*), wenn es sich um eine Read-Only-Transaktion handelt (TMP$READONLY), wenn die Transaktion bereits Daten geschrieben hat (TMP$WRITE), wenn die Transaktion nicht auf andere wartet (TMP$NOWAIT) und wenn bereits *Commit Retaining* aufgerufen wurde (TMP$COMMIT_RETAINING).

Der Status der Transaktion steht in TMP$STATE, es gibt ACTIVE, LIMBO (»schwebend«), COMMITING (»bestätigend«) und PRECOMMITED. TMP$TYPE hat den

Wert SNAPSHOT oder READ_COMMITTED. Und dann gibt es wieder den Block der statistischen Daten.

Beispiel aus der Praxis

Wozu kann man die temporären Tabellen in der Praxis gebrauchen? Nehmen wir mal an, der Server arbeitet mal wieder im »Kriechgang«, vermutlich hat irgendein Anwender ein Statement losgelassen, das besonders ineffizient formuliert ist und deshalb die Maschine ordentlich auslastet.

Mit der folgenden Abfrage erfragen wir alle offenen Statements und sortieren sie nach der Zeit, so dass wir das älteste Statement ganz oben angezeigt bekommen:

```
SELECT
    s.tmp$timestamp,
    s.tmp$statement_id,
    s.tmp$sql,
    a.tmp$attachment_id,
    a.tmp$user,
    a.tmp$user_ip_addr,
    a.tmp$user_host
  FROM tmp$statements s
    INNER JOIN tmp$attachments a
      ON (a.tmp$attachment_id = s.tmp$attachment_id)
  WHERE s.tmp$state = 'ACTIVE'
  ORDER BY s.tmp$timestamp
```

Damit wir auch gleich wissen, wo die Sache herkommt, verwenden wir einen JOIN auf die Liste der Datenbankverbindung und lassen uns User, IP und Hostnamen anzeigen. Das Statement selbst ist auch in der Ergebnismenge, man kann ja mal schauen, ob es sich nicht ein wenig effizienter formulieren lässt ...

Und wenn es erst einmal wichtiger ist, den Server wieder flott zu bekommen, dann kann man die betreffende Anweisung auch abbrechen – statt 12345 ist dann natürlich die entsprechende Statement-ID zu verwenden:

```
UPDATE tmp$statements
  SET tmp$state = 'CANCEL'
  WHERE tmp$statement_id = 12345
```

9 Delphi und InterBase

InterBase ist ein Datenbanksystem, das vor allem mit Delphi, Kylix und C++Builder eingesetzt wird. Alle drei Entwicklungsumgebungen stammen aus dem Hause Borland, verhalten sich ähnlich und verwenden eine teilweise identische Klassenbibliothek.

Da Delphi die am häufigsten eingesetzte Entwicklungsumgebung von diesen dreien ist, soll der Zugriff auf InterBase anhand von Delphi besprochen werden. Das Geschriebene ist fast vollständig auch für Kylix gültig und mit entsprechender Anpassung des Codes auch für C++Builder.

Komponenten für den Zugriff auf InterBase

Prinzipiell wäre es möglich, die Client-DLL von InterBase in das eigene Projekt einzubinden und mittels entsprechender DLL-Aufrufe auf InterBase zuzugreifen. Wenn Sie genau wissen, was Sie tun müssen, erhalten Sie auf diese Weise durchaus performante Anwendungen – die Programmierung selbst dürfte jedoch ein Mehrfaches an Zeit in Anspruch nehmen, als wenn Sie entsprechende Komponenten einsetzen.

Für den Zugriff auf Delphi gibt es viele Möglichkeiten, fünf von ihnen wollen wir in diesem Kapitel vergleichen:

- Die *Borland Database Engine* (BDE) und die Komponenten *TTable* und *TQuery* sind der Klassiker unter den Möglichkeiten und haben uns seit Delphi 1 begleitet. Allerdings hat Borland inzwischen angekündigt, dass die BDE nicht weiterentwickelt wird. Bei neuen Projekten sollte man sich somit lieber nach anderen Möglichkeiten umsehen.
- Die *dbExpress*-Komponenten setzen mittels einer Zwischen-DLL auf die Client-DLLs aller relevanten SQL-Server auf und bieten einen performanten, wenn auch nur uni-direktionalen und ungepufferten Zugriff. Diese Komponenten sind seit Delphi 6 dabei.
- Seit Delphi 5 enthalten sind die IBX-Komponenten, die ausschließlich mit InterBase arbeiten, dafür aber dessen Features gut unterstützen.
- Im Vertrieb von *Better Office* gibt es die *FIBPlus*-Komponenten. Auch sie arbeiten nur mit InterBase zusammen. Wir wollen uns ansehen, ob sich diese Investition lohnen könnte.
- Und dann gibt es noch die *IBObjects* von *Jason Wharton*. Diese beschränken sich nicht nur auf den Zugriff, sondern bieten auch eine ganze Reihe von visuellen Komponenten. Möchte man diese Komponenten zu mehr als zu Testzwecken einsetzen, dann wird hier eine Registrierungsgebühr fällig.

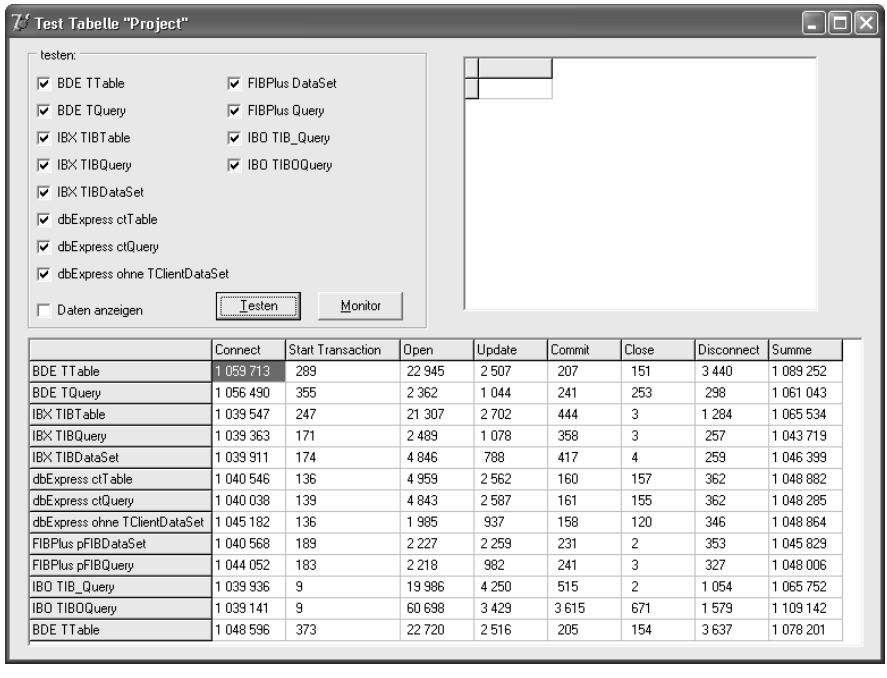
9.1 Messen der Zugriffszeiten

Wenn mehrere Möglichkeiten zur Verfügung stehen, dann interessieren vor allem zwei Aspekte:

- Wie schnell – also vor allem: wie einfach – kann man damit programmieren?
- Wie performant laufen die damit erstellten Anwendungen?

Den dritten Aspekt – was kosten die Komponenten – kann man getrost vernachlässigen. Im Vergleich zu den Personalkosten bei einem professionellen Projekt sind die Kosten für die Tools fast immer »peanuts«.

Wir wollen nun ein kleines Programm schreiben, das die Ausführungszeiten für einen einfachen Tabellenzugriff für alle diese Komponentengruppen vergleicht.



	Connect	Start Transaction	Open	Update	Commit	Close	Disconnect	Summe
BDE TTable	1 059 713	289	22 945	2 507	207	151	3 440	1 089 252
BDE TQuery	1 056 490	355	2 362	1 044	241	253	298	1 061 043
IBX TIBTable	1 039 547	247	21 307	2 702	444	3	1 284	1 065 534
IBX TIBQuery	1 039 363	171	2 489	1 078	358	3	257	1 043 719
IBX TIBDataSet	1 039 911	174	4 846	788	417	4	259	1 046 399
dbExpress ctTable	1 040 546	136	4 959	2 562	160	157	362	1 048 882
dbExpress ctQuery	1 040 038	139	4 843	2 587	161	155	362	1 048 285
dbExpress ohne TClientDataSet	1 045 182	136	1 985	937	158	120	346	1 048 864
FIBPlus pFIBDataSet	1 040 568	189	2 227	2 259	231	2	353	1 045 829
FIBPlus pFIBQuery	1 044 052	183	2 218	982	241	3	327	1 048 006
IBO TIB_Query	1 039 936	9	19 986	4 250	515	2	1 054	1 065 752
IBO TIBOQuery	1 039 141	9	60 698	3 429	3 615	671	1 579	1 109 142
BDE TTable	1 048 596	373	22 720	2 516	205	154	3 637	1 078 201

Abbildung 9.1: Messung der Ausführungszeiten

Dafür verwenden wir die Beispiel-Datenbank *employee.gdb* und dort die Tabelle *project*. Wir wollen uns zunächst einmal das Programm ansehen und dann die Messwerte interpretieren.

9.1.1 Das Hauptfenster

Wir beginnen mit einem Hauptfenster gemäß Abbildung 9.1: Ein paar Checkboxen, ein StringGrid, ein DBGrid und zwei Buttons.

```
procedure Tfrm_main.FormCreate(Sender: TObject);
begin
  randomize;
  with StringGrid1 do
  begin
    ColWidths[0] := 160;
    ColWidths[2] := 100;
    Cells[1,0] := 'Connect';
    Cells[2,0] := 'Start Transaction';
      ...
    Cells[0,13] := 'BDE TTable';
  end; {with StringGrid1 do}
end; {procedure Tfrm_main.FormCreate}
```

Beim Erstellen des Formulars wird das StringGrid beschriftet, auch die Spaltenbreiten müssen an zwei Stellen modifiziert werden.

```
procedure Tfrm_main.btnTestenClick(Sender: TObject);
var
  summe: integer;

  ...

begin
  Screen.Cursor := crHourGlass;
  try
    do_bde_ttable;
    do_bde_tquery;
    do_ibx_tibtable;
    do_ibx_tibquery;
    do_ibx_tibdataset;
    do_dbe_table;
    do_dbe_query;
    do_dbe_oc;
    do_fib_dataset;
    do_fib_query;
    do_ibo_query;
    do_ibo_query2;
    do_bde_ttable2;
  finally
    Screen.Cursor := crDefault;
  end;
end; {procedure Tfrm_main.btnTestenClick}
```

Für die einzelnen Komponenten wurde jeweils eine lokale Prozedur erstellt. Diese Prozeduren sind nahezu identisch aufgebaut, wir wollen uns die Sache am Beispiel von *do_bde_ttable* ansehen:

```
  procedure do_bde_ttable;
  begin
    if not chk_bde_ttable.Checked
      then exit;
    summe := 0;
    Zeitmessen(1, 1, frm_bde.DoConnect, Summe);
    Zeitmessen(2, 1, frm_bde.DoStartTransaction, Summe);
    try
      Zeitmessen(3, 1, frm_bde.DoTTableOpen, Summe);
      Sleep(1000);
      Zeitmessen(4, 1, frm_bde.DoTTableUpdate, Summe);
    finally
      Zeitmessen(5, 1, frm_bde.DoCommit, Summe);
      Zeitmessen(6, 1, frm_bde.DoTTableClose, Summe);
      Zeitmessen(7, 1, frm_bde.DoDisconnect, Summe);
      StringGrid1.Cells[8, 1]
        := FormatFloat('### ### ###', Summe);
    end;
  end; {procedure do_bde_ttable}
```

Wenn die entsprechende Checkbox nicht gewählt ist, dann wird die Prozedur abgebrochen. Zum Messen der Ausführungszeiten der einzelnen Aktionen wird die Routine *Zeitmessen* aufgerufen, die wir uns gleich ansehen werden. Diese Prozedur misst die Ausführungszeit der übergebenen Routine und schreibt sie in diejenige Zelle von *StringGrid1*, deren Spalte und Zeile als Parameter übergeben werden. Die einzelnen Zeiten werden dann in der Variablen *Summe* aufsummiert, deren Inhalt zuletzt in das StringGrid geschrieben wird.

```
procedure Tfrm_main.Zeitmessen(ACol, ARow: integer; Routine: TTestProc; var
ASumme: integer);
var
  c, t1, t2: int64;
  i: integer;
begin
  QueryPerformanceFrequency(c);
  QueryPerformanceCounter(t1);
  Routine;
  QueryPerformanceCounter(t2);
  i := 1000000 * (t2 - t1) div c;
  StringGrid1.Cells[ACol, ARow]
    := FormatFloat('### ### ###', i);
  ASumme := ASumme + i;
  Application.ProcessMessages;
end; {procedure TForm1.Zeitmessen}
```

Die Zeitmessung erfolgt mittels *QueryPerformanceCounter*, dies ermöglicht uns, auf die µs genau zu messen. Genauigkeit ist hier jedoch relativ, die einzelnen Ausführungszeiten sind doch gewissen Schwankungen unterworfen. Zur leichteren Lesbarkeit wird die Anzeige mit Leerzeichen formatiert. Damit die Anzeige gleich aktualisiert wird, rufen wir *Application.ProcessMessages* auf.

Zugriff mit der BDE

Zunächst wollen wir die Zeiten bei *TTable* und *TQuery* messen:

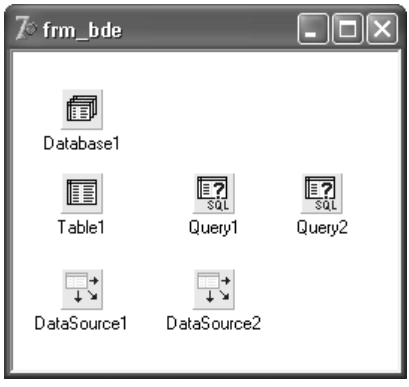


Abbildung 9.2: Die Komponenten der BDE

Über *Database1* greifen wir auf die Datenbank zu, der Alias *IBLocal* dürfte von der Installation her noch eingerichtet sein, als internen Alias verwenden wir *_INTERN* (der Unterstrich sorgt dafür, dass dieser Eintrag bei alphabetischer Sortierung ganz oben steht).

Setzen Sie die Parameter wie folgt:

```
USER NAME=SYSDBA
PASSWORD=masterkey
```

und sorgen Sie dafür, dass *LoginPrompt* auf *false* gesetzt wird.

Bei *Table1* wird *TableName* auf *project* gesetzt, *Query1* verwendet die folgende Abfrage:

```
SELECT *
  FROM project
  ORDER BY proj_id
```

Die Sortierung erfolgt aus Gründen der Vergleichbarkeit mit *TTable*. Da wir *RequestLive* auf *false* belassen, benötigen wir *Query2* zum Einfügen eines neuen Datensatzes:

```
UPDATE project
  SET proj_name = :name
  WHERE proj_id = :id
```

Darüber hinaus gibt es eine Menge von Routinen, welche die einzelnen Aktionen ausführen:

```
procedure Tfrm_bde.DoCommit;
begin
  Database1.Commit;
end;

procedure Tfrm_bde.DoConnect;
begin
  Database1.Connected := true;
end;

procedure Tfrm_bde.DoDisconnect;
begin
  Database1.Connected := false;
end;

procedure Tfrm_bde.DoStartTransaction;
begin
  Database1.StartTransaction;
end;
```

Für *TTable* und *TQuery* gemeinsam verwendet werden die Routinen, welche die Verbindung zur Datenbank herstellen beziehungsweise wieder abbauen, sowie das Transaktions-Management.

```
procedure Tfrm_bde.DoTTableOpen;
begin
  Table1.Open;
  if frm_main.chk_anzeigen.Checked then
  begin
    frm_main.DBGrid1.DataSource := DataSource1;
    Application.ProcessMessages;
  end;
end;

procedure Tfrm_bde.DoTTableClose;
begin
  Table1.Close;
end;
```

Ist die Checkbox *chk_anzeigen* aktiviert, dann wird nach dem Öffnen der Datenmenge das Grid an *DataSource1* gehängt.

```
procedure Tfrm_bde.DoTTableUpdate;
begin
  with Table1 do
  begin
    if not FindKey(['VBASE'])
      then ShowMessage('Fehler');
    Edit;
    FieldByName('PROJ_NAME').AsString := IntToStr(random(100000));
    Post;
  end;
end;
```

In der Routine *DoTTableUpdate* wird zunächst der Datensatz mit dem Schlüsselwert *VBASE* gesucht und dann ein zufälliger Wert in das Feld *PROJ_NAME* geschrieben.

```
procedure Tfrm_bde.DoTQueryUpdate;
begin
  with Query2 do
  begin
    ParamByName('ID').AsString := 'VBASE';
    ParamByName('Name').AsString := IntToStr(random(100000));
    ExecSQL;
  end;
end;
```

Die Routinen *DoTQueryOpen* und *DoTQueryClose* ähneln so sehr ihren *TTable*-Kollegen, dass wir sie uns erst gar nicht ansehen wollen. Beim Einfügen eines neuen Datensatzes werden jedoch ein INSERT-Statement und *ExecSQL* verwendet.

Verwendung von IBX

Bei der Verwendung der anderen Komponentengruppen bleibt vieles sehr ähnlich, wir wollen deshalb nur noch die Unterschiede zur BDE-Variante besprechen.

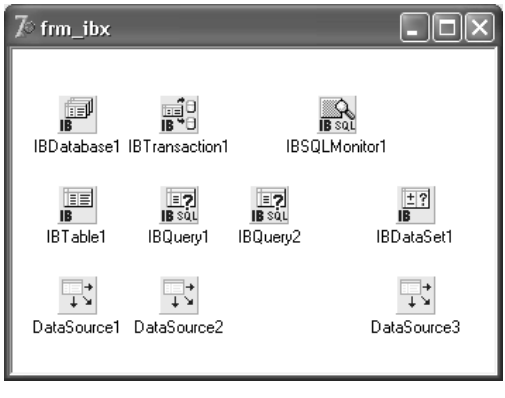


Abbildung 9.3: Die IBX-Komponenten

Bei Verwendung der IBX-Komponenten benötigt man neben *TIBDatabase* auch noch *TIBTransaction* – der Connect läuft dann über *TIBDatabase*, die Transaktionssteuerung über *TIBTransaction*.

Die Sache mit *TIBTable* und *TIBQuery* läuft fast exakt so wie bei der BDE. Allerdings kennt *TIBTable* kein *FindKey*, so dass wir hier *Locate* verwenden.

```
procedure Tfrm_ibx.DoTIBTableUpdate;
begin
  with IBTable1 do
  begin
    if not Locate('PROJ_ID', Variant('VBASE'), [])
      then ShowMessage('Fehler');
    Edit;
    FieldByName('PROJ_NAME').AsString := IntToStr(random(100000));
    Post;
  end;
end;
```

Ähnlich funktioniert die Sache bei *TIBDataSet*, hier muss jedoch die Eigenschaft *ModifySQL* gesetzt werden. Denken wir darüber jedoch nicht näher nach, sondern rufen den DATASET-EDITOR aus dem Kontextmenü auf.

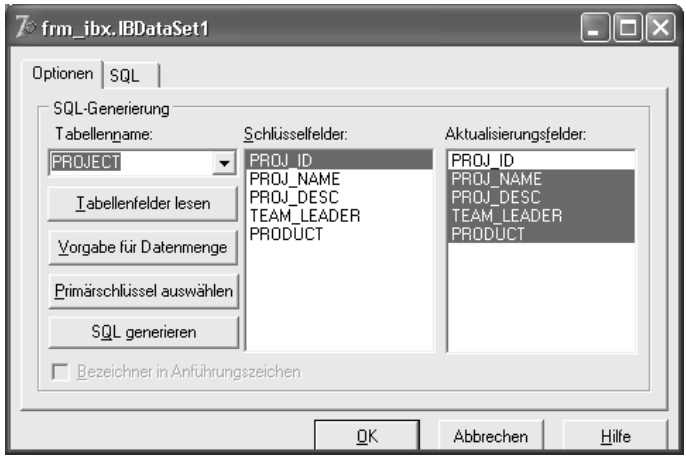


Abbildung 9.4: Der DataSet-Editor

Wählen Sie hier, welche der Felder Schlüssel-Felder sind und welche aktualisiert werden sollen. Anschließend klicken Sie auf den Button *SQL generieren*.

Der DataSet-Editor hat nun eine Reihe von SQL-Anweisungen erstellt, von denen wir hier aber nur die UPDATE-Anweisung benötigen.

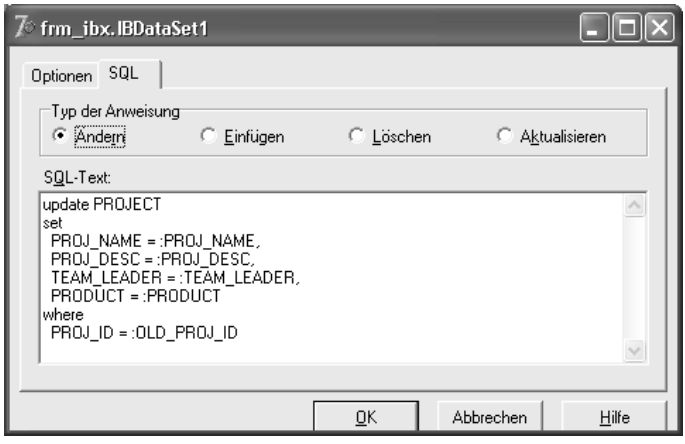


Abbildung 9.5: Die erstellen SQL-Anweisungen

Um die Datenbankverbindung genauer untersuchen zu können, verwenden wir die Komponente *TIBSQLMonitor*, deren Eigenschaft *Enabled* wir jedoch auf *false* lassen. Wenn ein Zugriff auf die Client-DLL protokolliert wird, schreiben wir das in das Monitorfenster, das wir später noch besprechen wollen:

```
procedure Tfrm_ibx.IBSQLMonitor1SQL(EventText: String;
  EventTime: TDateTime);
begin
  frm_monitor.mem_ibx.Lines.Add(EventText);
end;
```

Zugriff mit dbExpress

Die *dbExpress*-Komponenten liefern nur eine unidirektionale Datenmenge, für die Anzeige in einem DBGrid müssen die Daten mit einer *TClientDataSet*-Instanz zwischengespeichert werden. Hier in diesem Fall wäre dabei sogar der Einsatz von *TSimpleDataSet* denkbar. Wir wollen aus Gründen der Vergleichbarkeit – in der Praxis kommt man an dieser Vorgehensweise selten vorbei – jedoch die Datenmengenkette aus *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* diskret aufbauen.

Bei *SQLDataSet1* setzen wir *CommandType* auf *ctTable* und wählen für *CommandText* die Tabelle *Project*. Die Eigenschaft *IndexFieldNames* von *ClientDataSet1* setzen wir auf *proj_id*.

Bei *SQLDataSet2* setzen wir *CommandType* auf *ctQuery* und geben für *CommandText* das passende SELECT-Statement ein. Bei der Eigenschaft *IndexFieldNames* von *ClientDataSet1* verfahren wir analog zu *ClientDataSet1.*

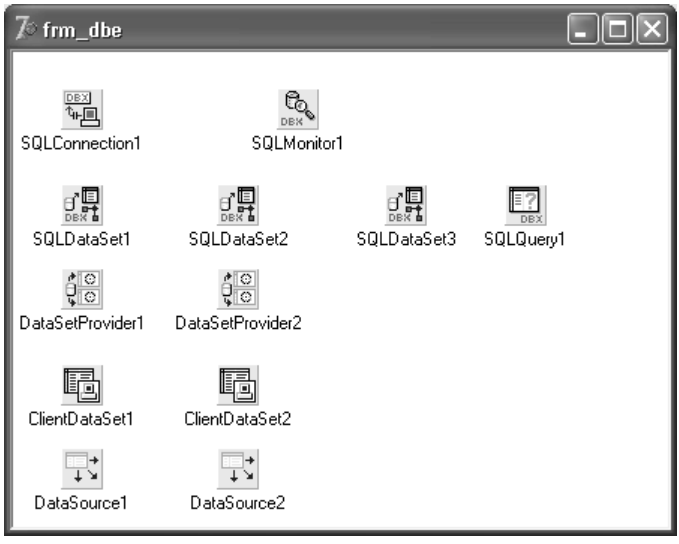


Abbildung 9.6: dbExpress

Von *SQLDataSet2* ziehen wir eine Kopie (*SQLDataSet3*), die jedoch nicht mit einem Provider verbunden wird und somit auch kein DBGrid speisen kann. Zur Aktualisierung nutzen wir *SQLQuery1*, die passende SQL-Anweisung haben wir ja schon einige Male verwendet.

Vor dem Aufruf von *StartTransaction* müssen die Transaktionsparameter auf passende Werte gesetzt werden:

```
procedure Tfrm_dbe.DoStartTransaction;
begin
  FDesc.TransactionID := 1;
  FDesc.IsolationLevel := xilREADCOMMITTED;
  SQLConnection1.StartTransaction(FDesc);
end;
```

Beim Ändern des Datensatzes wird zunächst die Client-Datenmenge geändert, anschließend werden die Änderungen mit *ApplyUpdates* zum Server übertragen.

```
procedure Tfrm_dbe.DoTQueryUpdate;
begin
  with ClientDataSet2 do
  begin
    if not FindKey(['VBASE'])
      then ShowMessage('Fehler');
    Edit;
    FieldByName('PROJ_NAME').AsString := IntToStr(random(100000));
    Post;
```

```
    ApplyUpdates(0);
  end;
end;
```

Auch hier sehen wir wieder die Möglichkeit vor, die Datenbankverbindung zu belauschen:

```
procedure Tfrm_dbe.SQLMonitor1LogTrace(Sender: TObject;
  CBInfo: pSQLTRACEDesc);
begin
  frm_monitor.mem_dbe.Lines.Add(CBInfo.pszTrace);
end;
```

Zugriff mit FIBPlus

Der Zugriff auf die Daten erfolgt hier stets mit *TpFIBDataSet*, jedoch wird einmal auch über diese Komponente aktualisiert, das andere Mal mit *TpFIBQuery*.

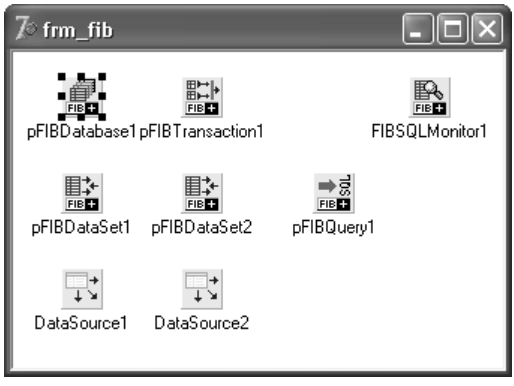


Abbildung 9.7: FIBPlus

Die Vorgehensweise gleicht der von IBX, von leichten Unterschieden bei den Bezeichnern einmal abgesehen.

Die Komponenten von FIBPlus bieten deutlich mehr Eigenschaften als die von IBX, so dass auch mehr Feinheiten konfiguriert werden können – im Gegensatz zu IBO sind diese jedoch mittels Objekteigenschaften zusammengefasst, so dass die Übersichtlichkeit nicht gänzlich verloren geht.

Zugriff mit IBO

Zuletzt wollen wir auch noch die Ausführungszeiten von IBO messen.

Zum einen verwenden wir *TIB_Query*. Diese Komponente ist nicht von *TDataSet* abgeleitet, *TDataSource* hat darauf keinen Zugriff – *IBObjects* hat eine eigene DataSource-Komponente und auch eigene datensensitive Dialogkomponenten. Die mit *IB_Query1* gemessenen Zeiten sind somit nur dann vergleichbar, wenn die Anzeige nicht aktiviert ist.

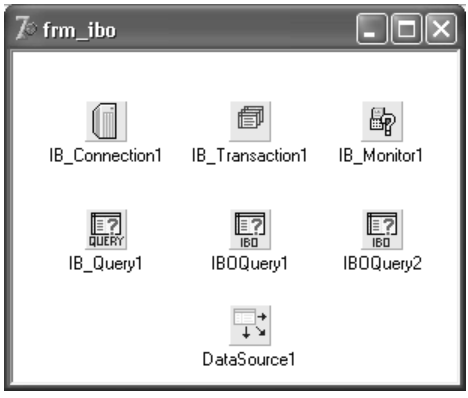


Abbildung 9.8: Die IBO-Komponenten

Von *TDataSet* abgeleitet ist dagegen *TIBOQuery* und hier führen wir die Änderung der Datenmenge mit *IBOQuery2* durch.

Die Komponenten von *IBObjects* sind durch eine sehr große Menge von Eigenschaften gekennzeichnet, die zudem nicht durch Objekteigenschaften zusammengefasst wurden. Von allen hier verwendeten Komponentengruppen sind sie sicher die unübersichtlichsten, was entsprechende Auswirkungen auf die Einarbeitungszeit hat.

Das Monitorfenster

Um die Datenbankverbindung belauschen zu können, richten wir uns noch ein kleines Monitorfenster ein.

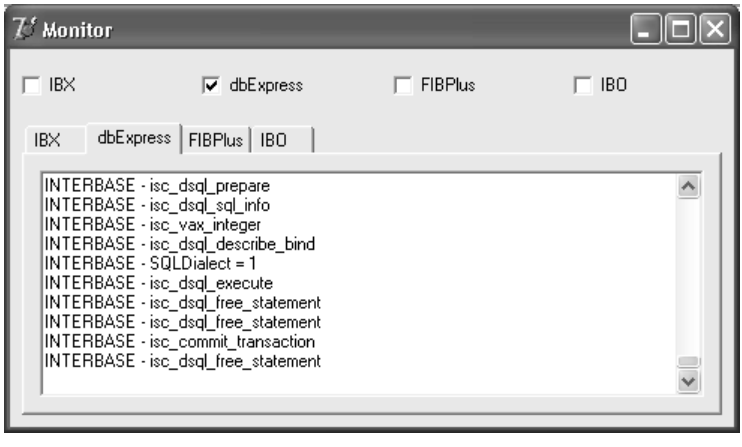


Abbildung 9.9: Das Monitorfenster

Die BDE müssen wir hier leider draußen lassen, dafür gibt es den SQL-Monitor in der Delphi-IDE. Mit den jeweiligen Checkboxen lässt sich die Protokollierung ein- und ausschalten:

```
procedure Tfrm_monitor.chk_ibxClick(Sender: TObject);
begin
  frm_ibx.IBSQLMonitor1.Enabled := chk_ibx.Checked;
end;
```

Die OnClick-Ereignisbehandlungsroutinen der anderen Checkboxen sind ähnlich aufgebaut.

9.2 Vergleich der Zugriffszeiten

Bei einem ersten Blick auf die Zugriffszeiten fällt auf, dass die Gesamtzeiten recht nah beieinander liegen und dass sie von den Connect-Zeiten bestimmt werden.

	Connect	Start Transaction	Open	Update	Commit	Close	Disconnect	Summe
BDE TTable	1 059 713	289	22 945	2 507	207	151	3 440	1 089 252
BDE TQuery	1 056 490	355	2 362	1 044	241	253	298	1 061 043
IBX TIBTable	1 039 547	247	21 307	2 702	444	3	1 284	1 065 534
IBX TIBQuery	1 039 363	171	2 489	1 078	358	3	257	1 043 719
IBX TIBDataSet	1 039 911	174	4 846	788	417	4	259	1 046 399
dbExpress ctTable	1 040 546	136	4 959	2 562	160	157	362	1 048 882
dbExpress ctQuery	1 040 038	139	4 843	2 587	161	155	362	1 048 285
dbExpress ohne TClientDataSet	1 045 182	136	1 985	937	158	120	346	1 048 864
FIBPlus pFIBDataSet	1 040 568	189	2 227	2 259	231	2	353	1 045 829
FIBPlus pFIBQuery	1 044 052	183	2 218	982	241	3	327	1 048 006
IBO TIB_Query	1 039 936	9	19 986	4 250	515	2	1 054	1 065 752
IBO TIBOQuery	1 039 141	9	60 698	3 429	3 615	671	1 579	1 109 142
BDE TTable	1 048 596	373	22 720	2 516	205	154	3 637	1 078 201

Abbildung 9.10: Erster Vergleich der Ausführungszeiten

Die langen Connect-Zeiten entstehen dadurch, dass auf dem Rechner bereits ein anderer Client auf diese Datenbank zugreift – sei es, dass in der Delphi-IDE irgendeine Datenbankverbindung offen geblieben ist, sei es, dass Programme wie *IBConsole* auf diese Datenbank zugreifen.

(Beim allerersten Zugriff treten oft längere Zeiten auf. Damit diese nicht *TTable* zugeordnet werden, wurde *TTable* am Schluss noch ein weiteres Mal gemessen.)

Wenn eine Datenbankverbindung in der IDE offen geblieben ist, dann sieht man das manchmal recht deutlich an der praktisch nicht vorhandenen Connect-Zeit, wie *Abbildung 9.2* zeigt. Bei IBX verändert eine offene Datenbankverbindung diese Zeiten jedoch nicht.

	Connect	S
BDE TTable	1 155 230	
BDE TQuery	1 027 165	
IBX TIBTable	1 020 857	
IBX TIBQuery	1 019 723	
IBX TIBDataSet	1 021 694	
dbExpress ctTable	1 032 167	
dbExpress ctQuery	1 020 093	
dbExpress ohne TClientDataSet	1 020 249	
FIBPlus pFIBDataSet	3	
FIBPlus pFIBQuery	1 038 110	
IBO TIB_Query	1 043 145	
IBO TIBOQuery	1 043 996	
BDE TTable	1 053 359	

Abbildung 9.11: In der IDE offen gebliebene Datenbankverbindung

Sind nun alle anderen Datenbankverbindungen geschlossen, geht der Connect deutlich schneller. Allerdings treten dann sporadisch absurd lange Disconnect-Zeiten auf.

	Connect	Sta	Op	Up	Cor	Clo	Disconnect	Su
BDE TTable	199 220	18	3	9	26	22	7 064	24
BDE TQuery	46 210	18	9	7	28	21	4 196	68
IBX TIBTable	36 331	17	3	2	44	1	9 012 299	9 0
IBX TIBQuery	31 686	18	9	8	38	1	4 017	53
IBX TIBDataSet	42 592	18	1	1	42	1	3 730	68
dbExpress ctTable	29 481	18	1	1	17	16	9 015 671	9 0
dbExpress ctQuery	31 368	18	1	1	17	15	3 869	57
dbExpress ohne TClientDataSet	50 105	17	1	8	16	90	9 017 363	9 0
FIBPlus pFIBDataSet	40 253	20	2	1	24	1	9 013 950	9 0
FIBPlus pFIBQuery	32 741	21	1	8	33	3	3 815	55
IBO TIB_Query	43 601	2	2	1	26	1	4 840	92
IBO TIBOQuery	48 367	6	6	9	3 8	12	9 250	13
BDE TTable	52 191	18	3	9	21	15	7 090	99

Abbildung 9.12: Lange Disconnect-Zeit bei fehlender zweiter Verbindung

Diese langen Disconnect-Zeiten treten bei allen Komponenten manchmal auf, eine Regelmäßigkeit konnte ich dabei nicht feststellen.

9.2.1 Start Transaction und Open

Nun sind die Connect- und Disconnect-Zeiten nicht diejenigen, die bei normalen Datenbankanwendungen die Performance bestimmen. Schauen wir uns die Zeiten für das Starten der Transaktion und das Öffnen der Datenbankverbindung etwas näher an:

	C	Start Transaction	Open
BDE TTable	1	295	25 653
BDE TQuery	1	294	3 799
IBX TIBTable	1	166	20 339
IBX TIBQuery	1	162	2 788
IBX TIBDataSet	1	157	5 162
dbExpress ctTable	1	144	5 464
dbExpress ctQuery	1	142	5 377
dbExpress ohne TClientDataSet	1	144	2 285
FIBPlus pFIBDataSet	1	193	2 499
FIBPlus pFIBQuery	1	197	2 503
IBO TIB_Query	1	9	20 020
IBO TIBOQuery	1	10	59 371
BDE TTable	1	298	23 464

Abbildung 9.13: Start Transaction und Open

IBO

Zunächst fällt auf, dass die IBO-Komponenten extrem kurze Zeit für das Starten einer Transaktion benötigen. Hier liegt der Verdacht nahe, dass gar keine Transaktion gestartet, sondern lediglich ein Flag gesetzt wird, das dafür sorgt, dass bei Bedarf das Starten einer Transaktion nachgeholt wird.

Um dies zu verifizieren, wurden die Anweisungen zum Arbeiten mit der Datenmenge auskommentiert:

```
  procedure do_ibo_query2;
  begin
    if not chk_ibo_query2.Checked
      then exit;
    summe := 0;
    Zeitmessen(1, 12, frm_ibo.DoConnect, Summe);
    Zeitmessen(2, 12, frm_ibo.DoStartTransaction, Summe);
    try
//      Zeitmessen(3, 12, frm_ibo.DoIBOQueryOpen, Summe);
//      Sleep(1000);
//      Zeitmessen(4, 12, frm_ibo.DoIBOQueryUpdate, Summe);
    finally
      Zeitmessen(5, 12, frm_ibo.DoCommit, Summe);
//      Zeitmessen(6, 12, frm_ibo.DoIBOQueryClose, Summe);
      Zeitmessen(7, 12, frm_ibo.DoDisconnect, Summe);
      StringGrid1.Cells[8, 12]
        := FormatFloat('### ### ###', Summe);
    end;
  end; {procedure do_ibo_query2}
```

Schaut man sich nun das Protokoll an, so findet man tatsächlich keinen Aufruf für das Starten einer Transaktion – sobald man jedoch den Aufruf von *Open* wieder dazunimmt, würde eine Transaktion gestartet.

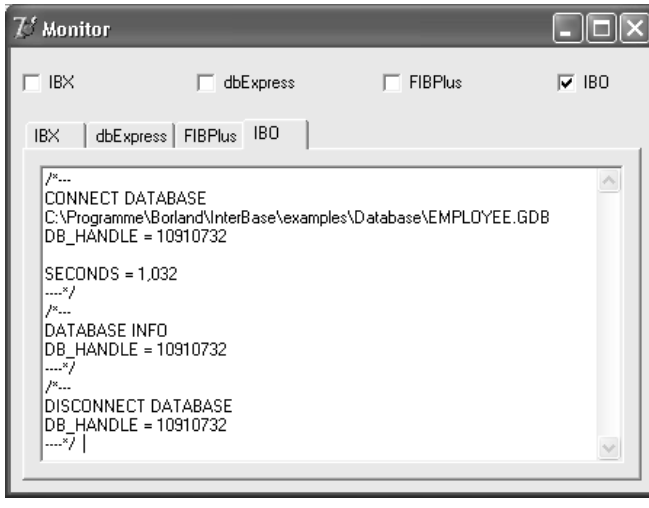


Abbildung 9.14: IBO startet keine Transaktion

Es bleibt die Frage, warum die Transaktion nicht sofort gestartet wird. Nimmt man die Zeiten von *StartTransaction* und *Open* zusammen, dann scheint diese Vorgehensweise noch nicht einmal ansatzweise einen Performance-Vorteil zu bringen – im Gegenteil: Die IBO-Komponenten fallen durch ausgesprochen lange Ausführungszeiten auf.

Versuchen wir, der Sache mit Hilfe des Monitorprotokolls auf den Grund zu gehen. Dieses musste leider stark gekürzt werden, weil es sonst 32 Seiten belegt hätte, die ich lieber mit Sinnvollerem fülle. Der Umfang des Protokolls ist jedoch nur zum Teil dem häufigen Zugriff auf die Client-DLL zu verdanken – IBObjects protokolliert auch äußerst ausführlich. (Selbstverständlich wurden alle Zeitmessungen – bei IBObjects und bei den anderen Komponentengruppen – bei abgeschaltetem Monitor durchgeführt.)

Um Ihnen einen Eindruck von der Ausführlichkeit des Protokolls zu geben, sind die ersten zehn Statements noch gänzlich ungekürzt.

```
/*===
PROFILE MARKED START(loading RelationsByID) -- Before start transaction
====*/
/*---
START TRANSACTION
DB HANDLE COUNT 1
TR_HANDLE = 10910452
```

```
----*/
/*---
DATABASE INFO
DB_HANDLE = 10910756
----*/
/*===
PROFILE DIFF REPORT(loading RelationsByID) -- Before prepare
*** OVERALL SERVER PROCESS ***
  Current Memory = 3.621.888 (+1.024)
  Num Buffer Reads = 218 (+4)
  Num Buffer Writes = 2 (+2)
  Num Page Writes = 2 (+2)
PROFILE DATA MARKED AS START
====*/
/*---
PREPARE STATEMENT
TR_HANDLE = 10910452
STMT_HANDLE = 10910524

SELECT R.RDB$RELATION_ID || '=', R.RDB$RELATION_NAME
FROM RDB$RELATIONS R
ORDER BY 1 ASC

PLAN SORT ((R NATURAL))

FIELDS = [  Version 1 SQLd 2 SQLn 30
  < SQLType: 449 SQLLen: 7 > = <NIL>
  RDB$RELATIONS.RDB$RELATION_NAME = <NIL> ]

SECONDS = 0,060
----*/
/*---
DATABASE INFO
DB_HANDLE = 10910756
----*/
/*===
PROFILE DIFF REPORT(loading RelationsByID) -- After prepare
*** OVERALL SERVER PROCESS ***
  Current Memory = 3.821.568 (+199.680)
  Max Memory = 3.929.184 (+305.248)
  Num Buffer Reads = 612 (+394)
  Num Page Reads = 54 (+31)
*** INDEXED READS ***  65
  16  RELATION_ID=2
  16  RELATION_ID=5
```

```
  15  RELATION_ID=12
  9  RELATION_ID=6
  7  RELATION_ID=8
  2  RELATION_ID=9
PROFILE DATA MARKED AS START
====*/
/*===
//>>> STATEMENT PREPARED <<<//
TIB_Statement.API_Prepare()
TIB_SchemaCursor: "DefaultSession." stHandle=10910524
====*/
/*---
DATABASE INFO
DB_HANDLE = 10910756
----*/
/*===
PROFILE DIFF REPORT(loading RelationsByID) -- Before execute
PROFILE DATA MARKED AS START
====*/
```

Ab hier wurde nun gekürzt:

```
EXECUTE STATEMENT
OPEN CURSOR
FETCH
  RDB$RELATIONS.RDB$RELATION_NAME = 'RDB$PAGES
FETCH
  RDB$RELATIONS.RDB$RELATION_NAME = 'RDB$FILES
FETCH
  RDB$RELATIONS.RDB$RELATION_NAME = 'RDB$TYPES
FETCH
  RDB$RELATIONS.RDB$RELATION_NAME = 'COUNTRY
FETCH
  RDB$RELATIONS.RDB$RELATION_NAME = 'JOB
...
```

Hier werden nun erst einmal die Namen aller Daten- und Systemtabellen gefetcht, das würde nun munter noch einige Seiten weitergehen ...

```
CLOSE CURSOR
START TRANSACTION
ALLOCATE STATEMENT
PREPARE STATEMENT
  SELECT RDB$PROCEDURE_NAME
    FROM RDB$PROCEDURES
    ORDER BY RDB$PROCEDURE_NAME ASC
    PLAN (RDB$PROCEDURES ORDER RDB$INDEX_21)
```

```
EXECUTE STATEMENT
OPEN CURSOR
FETCH
  RDB$PROCEDURES.RDB$PROCEDURE_NAME = 'ADD_EMP_PROJ
FETCH
  RDB$PROCEDURES.RDB$PROCEDURE_NAME = 'ALL_LANGS
FETCH
...
```

Nun geht es weiter mit allen STORED PROCEDURES.

```
CLOSE CURSOR
PREPARE STATEMENT
  SELECT S.RDB$FIELD_NAME
      , I.RDB$RELATION_NAME
    FROM RDB$RELATION_CONSTRAINTS C
        , RDB$INDICES I
        , RDB$INDEX_SEGMENTS S
      WHERE C.RDB$CONSTRAINT_TYPE = 'PRIMARY KEY'
        AND C.RDB$RELATION_NAME = I.RDB$RELATION_NAME
        AND C.RDB$INDEX_NAME = I.RDB$INDEX_NAME
        AND I.RDB$INDEX_NAME = S.RDB$INDEX_NAME
      ORDER BY I.RDB$RELATION_NAME ASC
         , S.RDB$FIELD_POSITION ASC
      PLAN SORT (JOIN (C NATURAL,
I INDEX (RDB$INDEX_5,RDB$INDEX_31),S
INDEX (RDB$INDEX_6)))
EXECUTE STATEMENT
OPEN CURSOR
FETCH
  RDB$INDEX_SEGMENTS.RDB$FIELD_NAME = 'COUNTRY
  RDB$INDICES.RDB$RELATION_NAME = 'COUNTRY
FETCH
  RDB$INDEX_SEGMENTS.RDB$FIELD_NAME = 'CUST_NO
  RDB$INDICES.RDB$RELATION_NAME = 'CUSTOMER
  ...
```

Auch die Abfrage der indizierten Spalten wollen wir uns nun nicht in aller Ausführlichkeit antun.

```
PREPARE STATEMENT
  SELECT *
    FROM project
    PLAN (PROJECT NATURAL)
OPEN CURSOR
FETCH
FIELDS = [  Version 1 SQLd 5 SQLn 5
```

```
  PROJECT.PROJ_ID = 'VBASE'
  PROJECT.PROJ_NAME = '52423'
  PROJECT.PROJ_DESC = BLOB ID ( 132, 12 )
  PROJECT.TEAM_LEADER = 45
  PROJECT.PRODUCT = 'other' ]
CLOSE CURSOR
```

Die Komponente *TIB_Query* ist neugierig – und das nicht nur beim erstmaligen Öffnen. Warum bei jedem *Open* eine Liste aller Tabellen- und STORED PROCEDURE-Bezeichner abgerufen wird, bleibt wohl das Geheimnis des Entwicklers ...

Die BDE

Seit Jahr und Tag haben erfahrene Entwickler immer wieder »gepredigt«, bei SQL-Datenbanken *TQuery* statt *TTable* zu verwenden, und das war wohl ziemlich richtig. Wir wollen nun jedoch genau wissen, warum die Sache mit *TTable* etwa siebenmal länger dauert als mit *TQuery.*

Dazu starten wir den *SQL Monitor* und schauen uns erst mal *TQuery* an (hier nur für die Anweisung *Open*):

```
5       16:40:57  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
6       16:40:57  SQL Vendor: INTRBASE - isc_dsql_prepare
7       16:40:57  SQL Vendor: INTRBASE - isc_dsql_sql_info
8       16:40:57  SQL Vendor: INTRBASE - isc_vax_integer
9       16:40:57  SQL Execute: INTRBASE - SELECT *
  FROM project
  order by proj_id

10      16:40:57  SQL Vendor: INTRBASE - isc_dsql_execute
11      16:40:57  SQL Stmt: INTRBASE - Fetch
12      16:40:57  SQL Vendor: INTRBASE - isc_dsql_fetch
13      16:40:57  SQL Data Out: INTRBASE - Column = 1, Name = PROJ_ID,
        Type = fldZSTRING, Precision = 5, Scale = 0, Data = DGPII
14      16:40:57  SQL Data Out: INTRBASE - Column = 2, Name = PROJ_NAME,
        Type = fldZSTRING, Precision = 20, Scale = 0, Data = DigiPizza
15      16:40:57  SQL Data Out: INTRBASE - Column = 4, Name = TEAM_LEADER,
        Type = fldINT16, Precision = 1, Scale = 0, Data = 24
16      16:40:57  SQL Data Out: INTRBASE - Column = 5, Name = PRODUCT,
        Type = fldZSTRING, Precision = 12, Scale = 0, Data = other
17      16:40:58  SQL Stmt: INTRBASE - Fetch
  ...
23      16:40:58  SQL Stmt: INTRBASE - Fetch
  ...
29      16:40:58  SQL Stmt: INTRBASE - Fetch
  ...
35      16:40:58  SQL Stmt: INTRBASE - Fetch
  ...
```

```
47      16:40:58  SQL Stmt: INTRBASE - Fetch
48      16:40:58  SQL Vendor: INTRBASE - isc_dsql_fetch
49      16:40:58  SQL Stmt: INTRBASE - EOF
50      16:40:58  SQL Stmt: INTRBASE - Reset
51      16:40:58  SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Das SELECT-Statement wird vorbereitet und an den Server geschickt. Anschließend werden die Datensätze abgerufen – an dieser Stelle habe ich ein wenig gekürzt, Sie sehen es anhand der Zeilennummern –, danach wird die Anweisung wieder freigegeben.

Kleine Bemerkung am Rande: Diese Anweisungen müssten Sie selbst aufrufen, wenn Sie auf die Komponenten verzichten und direkt auf die Client-DLL zugreifen wollten.

Die Sache ist bei *TQuery* ziemlich geradeaus programmiert. Vergleichen wir dies mit *TTable*:

```
16:54:50  SQL Execute: INTRBASE -
  select R.RDB$FIELD_NAME, F.RDB$FIELD_TYPE,
      F.RDB$FIELD_SUB_TYPE, F.RDB$DIMENSIONS,
      F.RDB$FIELD_LENGTH, F.RDB$FIELD_SCALE,
      F.RDB$VALIDATION_BLR, F.RDB$COMPUTED_BLR,
      R.RDB$DEFAULT_VALUE, F.RDB$DEFAULT_VALUE,
      R.RDB$NULL_FLAG
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
      and R.RDB$RELATION_NAME = 'PROJECT'
    order by R.RDB$FIELD_POSITION ASC
  ...
16:54:50  SQL Execute: INTRBASE -
  select I.RDB$INDEX_NAME, I.RDB$UNIQUE_FLAG,
      I.RDB$INDEX_TYPE, F.RDB$FIELD_NAME
    from RDB$INDICES I, RDB$INDEX_SEGMENTS F
    where I.RDB$RELATION_NAME = 'PROJECT'
      and I.RDB$INDEX_NAME = F.RDB$INDEX_NAME
    order by I.RDB$INDEX_ID, F.RDB$FIELD_POSITION ASC
  ...
16:54:50  SQL Execute: INTRBASE -
  select R.RDB$FIELD_NAME, F.RDB$VALIDATION_BLR,
      F.RDB$COMPUTED_BLR, R.RDB$DEFAULT_VALUE,
      F.RDB$DEFAULT_VALUE, R.RDB$NULL_FLAG
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
      and R.RDB$RELATION_NAME = 'PROJECT'
    order by R.RDB$FIELD_POSITION ASC
  ...
```

```
16:54:50  SQL Prepare: INTRBASE -
  SELECT PROJ_ID ,PROJ_NAME ,PROJ_DESC ,TEAM_LEADER ,PRODUCT
  FROM PROJECT
  ORDER BY  PROJ_ID ASC
72      16:54:50  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
73      16:54:50  SQL Vendor: INTRBASE - isc_dsql_prepare
16:54:50  SQL Execute: INTRBASE -
SELECT PROJ_ID ,PROJ_NAME ,PROJ_DESC ,TEAM_LEADER ,PRODUCT
FROM PROJECT
ORDER BY  PROJ_ID ASC
75      16:54:50  SQL Vendor: INTRBASE - isc_dsql_execute
76      16:54:50  SQL Stmt: INTRBASE - Fetch
77      16:54:50  SQL Vendor: INTRBASE - isc_dsql_fetch
78      16:54:50  SQL Data Out: INTRBASE - Column = 1, Name = PROJ_ID,
        Type = fldZSTRING, Precision = 5, Scale = 0, Data = DGPII
79      16:54:50  SQL Data Out: INTRBASE - Column = 2, Name = PROJ_NAME,
        Type = fldZSTRING, Precision = 20, Scale = 0, Data = DigiPizza
80      16:54:50  SQL Data Out: INTRBASE - Column = 4, Name = TEAM_LEADER,
        Type = fldINT16, Precision = 1, Scale = 0, Data = 24
81      16:54:50  SQL Data Out: INTRBASE - Column = 5, Name = PRODUCT,
        Type = fldZSTRING, Precision = 12, Scale = 0, Data = other
82      16:54:51  SQL Stmt: INTRBASE - Close
83      16:54:51  SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Bevor *TTable* auf die eigentliche Tabelle zugreift, wird sie erst mal neugierig und greift dreimal auf Systemtabellen zu, um Feld- und Index-Informationen zu erfragen. Diese drei Tabellen sind zudem JOINs, was den Vorgang naturgemäß nicht beschleunigt. (Die ganzen Fetch-Aufrufe habe ich rigoros rausgekürzt, Sie sehen es an den Zeilennummern; auch die Formatierung der SQL-Anweisungen wurde von mir vorgenommen, um die Lesbarkeit zu erhöhen.)

Der langen Rede kurzer Sinn: Dort, wo es auf Performance ankommt, sollten Sie bei SQL-Datenbanksystemen *TTable* lieber vergessen.

IBX

Schauen wir uns IBX an. Die Zeiten für das Starten einer Transaktion weichen erwartungsgemäß nicht weit voneinander ab, sie sind durch die Streuung der Messwerte begründet.

IBX TIBTable	1	166	20 339
IBX TIBQuery	1	162	2 788
IBX TIBDataSet	1	157	5 162

Abbildung 9.15: StartTransaction und Open bei IBX

Deutliche Unterschiede findet man jedoch beim Aufruf von *Open*. Auch hier bemühen wir wieder den Monitor, zunächst für *TIBQuery*:

```
IBQuery1: [Prepare] select * from PROJECT
  order by PROJ_ID
  Plan: PLAN (PROJECT ORDER RDB$PRIMARY12)
: [Start transaction]
: [Prepare]
  Select F.RDB$COMPUTED_BLR, F.RDB$DEFAULT_VALUE,
      R.RDB$FIELD_NAME
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$RELATION_NAME = :RELATION
      and R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
      and ((not F.RDB$COMPUTED_BLR is NULL)
        or (not F.RDB$DEFAULT_VALUE is NULL))
    Plan: PLAN JOIN (R INDEX (RDB$INDEX_4),F INDEX (RDB$INDEX_2))
: [Fetch] ...
: [Execute]...
: [Fetch] ...
  SEOFReached
: [Commit (Hard commit)]
IBQuery1: [Execute] select * from PROJECT
  order by PROJ_ID
IBQuery1: [Fetch] select * from PROJECT
  order by PROJ_ID
```

Beachten Sie bitte, dass die Monitorprotokolle von verschiedenen Komponentengruppen nur sehr bedingt miteinander vergleichbar sind.

Es fallen hier zwei Dinge auf: Zum einen wird hier – im Gegensatz zu *TQuery* – auch eine Systemtabelle abgefragt. Es mag nun verwundern, dass IBX hier trotzdem schneller sein soll. Des Rätsels Lösung ist hier, dass diese Systemtabelle nur beim allerersten Aufruf abgefragt wird – anschließend merken sich die IBX-Komponenten diese Information. Die Zahlen in *Abbildung 9.1* stammen jedoch von einem wiederholten Aufruf, sonst wäre die Zeit etwa 5000 µs länger.

Darüber hinaus wird hier von der Tabelle *project* nur ein einzelner Datensatz abgerufen. Dies erklärt auch den leichten Geschwindigkeitsvorteil gegenüber *TQuery*.

Vergleichen wir das mit *TIBDataSet*:

```
IBDataSet1: [Prepare] select * from PROJECT
Plan: PLAN (PROJECT NATURAL)
IBDataSet1: [Prepare] delete from PROJECT
  where
    PROJ_ID = :OLD_PROJ_ID
    Plan: PLAN (PROJECT INDEX (RDB$PRIMARY12))
IBDataSet1: [Prepare] insert into PROJECT
```

```
    (PROJ_NAME, PROJ_DESC, TEAM_LEADER, PRODUCT)
  values
    (:PROJ_NAME, :PROJ_DESC, :TEAM_LEADER, :PRODUCT)
Plan:
IBDataSet1: [Prepare] update PROJECT
  set
    PROJ_NAME = :PROJ_NAME,
    PROJ_DESC = :PROJ_DESC,
    TEAM_LEADER = :TEAM_LEADER,
    PRODUCT = :PRODUCT
  where
    PROJ_ID = :OLD_PROJ_ID
  Plan: PLAN (PROJECT INDEX (RDB$PRIMARY12))
IBDataSet1: [Prepare] Select
    PROJ_ID,
    PROJ_NAME,
    PROJ_DESC,
    TEAM_LEADER,
    PRODUCT
  from PROJECT
  where
    PROJ_ID = :PROJ_ID
  Plan: PLAN (PROJECT INDEX (RDB$PRIMARY12))
: [Start transaction]
: [Prepare] Select F.RDB$COMPUTED_BLR, F.RDB$DEFAULT_VALUE,
    R.RDB$FIELD_NAME
  from RDB$RELATION_FIELDS R, RDB$FIELDS F
  where R.RDB$RELATION_NAME = :RELATION
    and R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
    and ((not F.RDB$COMPUTED_BLR is NULL)
    or (not F.RDB$DEFAULT_VALUE is NULL))
Plan: PLAN JOIN (R INDEX (RDB$INDEX_4),F INDEX (RDB$INDEX_2))
: [Fetch]...
: [Execute] ...
: [Fetch] ...
SEOFReached
: [Commit (Hard commit)]
IBDataSet1: [Execute] select * from PROJECT
IBDataSet1: [Fetch] select * from PROJECT
```

Im Prinzip ist die Vorgehensweise hier wie bei *TIBQuery*, allerdings werden die vier SQL-Statements vorbereitet, die für das Ändern und Aktualisieren der Daten benötigt werden. Man könnte die Sache dadurch beschleunigen, dass man die SQL-Anweisungen für INSERT und DELETE – diese werden hier nicht verwendet – löscht, so dass sie auch nicht vorbereitet werden. Auch hier ist der zweite Aufruf wieder deutlich schneller, da der Zugriff auf die Systemtabelle entfällt.

Was sich nur unwesentlich beschleunigt, ist der zweite Zugriff von *TIBTable*. Hier liegt der Verdacht nahe, dass die Systemtabelle in jedem Fall abgefragt und die Sache nur deswegen schneller wird, weil InterBase die Daten noch im Cache hat.

```
IBTable1: [Prepare]
  Select R.RDB$FIELD_NAME, R.RDB$FIELD_POSITION,
      F.RDB$COMPUTED_BLR, F.RDB$DEFAULT_VALUE,
      F.RDB$NULL_FLAG, F.RDB$FIELD_LENGTH,
      F.RDB$FIELD_SCALE, F.RDB$FIELD_TYPE,
      F.RDB$FIELD_SUB_TYPE, F.RDB$EXTERNAL_LENGTH,
      F.RDB$EXTERNAL_SCALE, F.RDB$EXTERNAL_TYPE
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$RELATION_NAME = 'PROJECT'
      and R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
    order by R.RDB$FIELD_POSITION
    Plan: PLAN SORT (JOIN (R INDEX (RDB$INDEX_4),
      F INDEX (RDB$INDEX_2)))
IBTable1: [Execute] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
  SEOFReached
: [Commit (Hard commit)]
: [Start transaction]
IBTable1: [Prepare]
  Select I.RDB$INDEX_NAME, I.RDB$UNIQUE_FLAG,
      I.RDB$INDEX_TYPE, I.RDB$SEGMENT_COUNT,
      S.RDB$FIELD_NAME
    from RDB$INDICES I, RDB$INDEX_SEGMENTS S
    where I.RDB$INDEX_NAME = S.RDB$INDEX_NAME
      and I.RDB$RELATION_NAME = 'PROJECT'
    Plan: PLAN JOIN (I INDEX (RDB$INDEX_31),S INDEX (RDB$INDEX_6))
IBTable1: [Execute] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Prepare]
  Select RDB$FIELD_NAME from RDB$INDEX_SEGMENTS
    where RDB$INDEX_NAME = 'PRODTYPEX'
    ORDER BY RDB$FIELD_POSITION
    Plan: PLAN SORT ((RDB$INDEX_SEGMENTS INDEX (RDB$INDEX_6)))
```

```
IBTable1: [Execute]...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ...
IBTable1: [Fetch] ((andere Tabelle))
IBTable1: [Fetch] ...
SEOFReached
IBTable1: [Fetch] ((andere Tabelle))
SEOFReached
: [Commit (Hard commit)]
IBTable1: [Prepare]
  select PROJECT.*, RDB$DB_KEY as IBX_INTERNAL_DBKEY
    from PROJECT
    order by PROJ_ID
    Plan: PLAN (PROJECT ORDER RDB$PRIMARY12)
IBTable1: [Prepare]
  select PROJECT.*, RDB$DB_KEY as IBX_INTERNAL_DBKEY
    from PROJECT
    where RDB$DB_KEY = :IBX_INTERNAL_DBKEY
    Plan: PLAN (PROJECT INDEX ())
: [Start transaction]
: [Prepare]
  Select F.RDB$COMPUTED_BLR, F.RDB$DEFAULT_VALUE,
       R.RDB$FIELD_NAME
    from RDB$RELATION_FIELDS R, RDB$FIELDS F
    where R.RDB$RELATION_NAME = :RELATION
      and R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME
      and ((not F.RDB$COMPUTED_BLR is NULL)
        or (not F.RDB$DEFAULT_VALUE is NULL))
    Plan: PLAN JOIN (R INDEX (RDB$INDEX_4),F INDEX (RDB$INDEX_2))
: [Fetch] ...
: [Execute] ...
: [Fetch] ...
SEOFReached
: [Commit (Hard commit)]
: [Start transaction]
IBTable1: [Prepare]
  Select RDB$SYSTEM_FLAG, RDB$DBKEY_LENGTH
    from RDB$RELATIONS
    where RDB$RELATION_NAME = 'PROJECT'
    Plan: PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))
IBTable1: [Fetch] ...
IBTable1: [Execute] ...
: [Commit (Hard commit)]
IBTable1: [Prepare]
  delete from PROJECT
```

```
    where RDB$DB_KEY = :IBX_INTERNAL_DBKEY
    Plan: PLAN (PROJECT INDEX ())
IBTable1: [Prepare]
  insert into PROJECT (PROJ_ID, PROJ_NAME, PROJ_DESC,
      TEAM_LEADER, PRODUCT)
    values (:PROJ_ID, :PROJ_NAME, :PROJ_DESC, :TEAM_LEADER,
      :PRODUCT)
    Plan:
IBTable1: [Prepare]
  update PROJECT
    set PROJ_ID = :PROJ_ID, PROJ_NAME = :PROJ_NAME,
      PROJ_DESC = :PROJ_DESC, TEAM_LEADER = :TEAM_LEADER,
      PRODUCT = :PRODUCT
    where RDB$DB_KEY = :IBX_INTERNAL_DBKEY
    Plan: PLAN (PROJECT INDEX ())
: [Start transaction]
IBTable1: [Prepare]
  Select USER from RDB$RELATIONS
    where RDB$RELATION_NAME = 'PROJECT'
    Plan: PLAN (RDB$RELATIONS INDEX (RDB$INDEX_0))
IBTable1: [Fetch] ...
IBTable1: [Execute] ...
: [Commit (Hard commit)]
IBTable1: [Execute]
  select PROJECT.*, RDB$DB_KEY as IBX_INTERNAL_DBKEY
    from PROJECT
    order by PROJ_ID
IBTable1: [Fetch] ...
```

Nun ja, sieben Tabellen-Abfragen und noch ein paar DML-Statements vorbereitet, das kann ja nicht performant werden. (Und bedenken wir bitte: Im »Ernstfall« müssten alle Daten noch über ein Netzwerk und die Server-Belastung würden sie auch nicht gerade reduzieren ...).

Fazit: *TIBTable* sollte man ganz schnell wieder vergessen. Der Unterschied zwischen *TIBQuery* und *TIBDataSet* besteht vor allem darin, dass *TIBDataSet* die DML-Statements vorbereitet. Wenn man bedenkt, wie viel Ärger (und Rechenzeit) *TIBDataSet* dadurch erspart, dass es veränderte Datensätze einzeln nachlädt, sollte das kein Argument gegen *TIBDataSet* sein, zumal die Vorbereitungszeit bei *TIBQuery*-Lösungen dann halt an anderer Stelle anfallen würden.

dbExpress

Der Aufruf von *StartTransaction* erfolgt bei dbExpress am schnellsten (wenn man mal vom »unechten« Aufruf bei IBO absieht). Dies ist ein Zeichen dafür, dass außer dem entsprechenden Aufruf der Client-DLL nichts weiter ausgeführt wird.

dbExpress ctTable	1	144	5 464
dbExpress ctQuery	1	142	5 377
dbExpress ohne TClientDataSet	1	144	2 285

Abbildung 9.16: StartTransaction und Open bei dbExpress

Der Aufruf von *Open* erfolgt dann sehr schnell, wenn keine *TClientDataSet*-Instanz mit *TSQLDataSet* verbunden wird. Ein Blick in das Monitorprotokoll zeigt auch, dass hier nicht viel passiert:

```
INTERBASE - isc_dsql_allocate_statement
select * from PROJECT
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

Die Anweisung wird vorbereitet, ausgeführt und dann wird ein Datensatz abgerufen. Bei der Datenmengenkette sieht dies ganz anders aus:

```
INTERBASE - isc_dsql_allocate_statement
select * from PROJECT
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
INTERBASE - isc_dsql_execute

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_get_segment
INTERBASE - isc_close_blob
INTERBASE - isc_dsql_free_statement
  ... (es folgen 5 solcher Blöcke)
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_free_statement
```

Es wird hier schnell klar, aus welchen Gründen die Sache hier vergleichsweise langsam ist: Es werden wirklich alle Datensätze abgerufen und dabei werden auch noch alle BLOBs zum Client transferiert. Das Abrufen aller Daten ist nun mal ein Kennzeichen von *TClientDataSet* – bei einigen wenigen Datensätzen macht es nun wirklich keinen Sinn, dies mit *PacketRecords* zu beschränken. Was man jedoch tun sollte, ist, das Laden der BLOBs zu vermeiden. Dazu setzen wir die Option *poFetchBlobsOnDemand* von *DataSetProvider1*; nun werden die BLOBs nicht mehr automatisch heruntergeladen, sondern nur dann, wenn sie explizit angefordert werden. Nun sieht das Monitorprotokoll schon sehr viel übersichtlicher aus:

```
INTERBASE - isc_dsql_allocate_statement
 select * from PROJECT
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
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_dsql_fetch
INTERBASE - isc_dsql_free_statement
```

An Ausführungszeit werden hier ungefähr 2000µs eingespart – bei einer längeren Tabelle wäre es sicherlich deutlich mehr. Auch damit ist man noch nicht ganz auf dem Niveau von *TQuery* – das Füllen der Datenmengenkette braucht halt eben seine Zeit. Wenn man jedoch bedenkt, dass man hier schon eine ohne großen Aufwand aktualisierbare Datenmenge hat, dann muss das als durchaus performant gelten. (Im Vergleich zu *TIBQuery* sollte man nicht vergessen, dass auch diese Komponente nur einen einzelnen Datensatz abruft, wenn nichts anderes verlangt wird.)

Fazit: Wenn man auf das Übertragen der BLOBs verzichtet, dann ist *TSQLDataSet* sehr performant – insbesondere dann, wenn man nicht mehrere Datensätze gleichzeitig anzeigen möchte und deshalb auf *TClientDataSet* verzichten kann.

FIBPlus

Bei *FIBPlus* ist es für das Öffnen der Datenmenge egal, ob Sie für die Aktualisierung *TpFIBDataSet* oder *TpFIBQuery* verwenden, die Unterschiede liegen im Bereich der Streuung der Messwerte. Zu den Geschwindigkeiten von IBX gibt es keine größeren Unterschiede: Bei *StartTransaction* hat IBX leicht die Nase vorne, bei *Open* FIBPlus.

Man könnte nun einwenden, dass *TpFIBDataSet* gleich eine aktualisierbare Datenmenge liefert und somit mit *TIBDataSet* verglichen werden müsste, was zum Öffnen der Datenmenge deutlich länger braucht. Dies liegt aber daran, dass alle DML-Statements dort vorbereitet werden, was bei *TpFIBDataSet* nicht der Fall ist – die Konsequenzen werden wir nachher noch sehen.

Schauen wir kurz in das Monitorprotokoll:

```
pFIBDataSet1: [Prepare] SELECT *
  FROM project
  Plan:
: [Start transaction](1835)
: [Prepare] select COUNT(RDB$RELATION_NAME)
  from RDB$RELATIONS
  where RDB$FLAGS = 1
    and RDB$RELATION_NAME=?RT
  Plan:
: [Fetch] ...
COUNT = 0
: [Execute] ...
  RT = 'FIB$FIELDS_INFO'
  Execute tick count 0
: [Fetch] ...
  COUNT = 0
: [Execute] ...
  RT = 'FIB$DATASETS_INFO'
  Execute tick count 0
: [Rollback](1835)
: [Start transaction](1836)
: [Prepare]  SELECT A1.RDB$RELATION_ID VER
  FROM RDB$RELATIONS A1
  WHERE A1.RDB$SYSTEM_FLAG = 0
    AND A1.RDB$RELATION_NAME = ?TN
    AND NOT A1.RDB$VIEW_BLR IS NULL
  UNION SELECT RDB$FORMAT
  FROM RDB$RELATIONS R
  WHERE R.RDB$SYSTEM_FLAG = 0
    AND R.RDB$RELATION_NAME = ?TN
    AND R.RDB$VIEW_BLR IS NULL
  Plan:
: [Fetch] ...
VER = 3
: [Execute]  ...
  TN = 'PROJECT'
  TN = 'PROJECT'
```

```
: [Prepare]  SELECT R.RDB$FIELD_NAME , R.RDB$FIELD_SOURCE ,
    F.RDB$COMPUTED_BLR , R.RDB$DEFAULT_SOURCE DS ,
    F.RDB$DEFAULT_SOURCE DS1 , F.RDB$FIELD_TYPE ,
    F.RDB$DIMENSIONS
  FROM RDB$RELATION_FIELDS R
    JOIN RDB$FIELDS F
      ON (R.RDB$FIELD_SOURCE = F.RDB$FIELD_NAME )
  WHERE R.RDB$RELATION_NAME = :TN
  ORDER BY R.RDB$FIELD_POSITION
  Plan:
: [Fetch]  ...
  RDB$FIELD_NAME = PROJ_ID
  RDB$FIELD_SOURCE = PROJNO
  RDB$COMPUTED_BLR = NULL
  DS = NULL
  DS1 = NULL
  RDB$FIELD_TYPE = 14
  RDB$DIMENSIONS = NULL

: [Execute] ...
  TN = 'PROJECT'
  Execute tick count 0
: [Fetch] ...
  RDB$FIELD_NAME = PROJ_NAME
  RDB$FIELD_SOURCE = RDB$10
  RDB$COMPUTED_BLR = NULL
  DS = NULL
  DS1 = NULL
  RDB$FIELD_TYPE = 37
  RDB$DIMENSIONS = NULL
: [Fetch]  ...
  RDB$FIELD_NAME = PROJ_DESC
  RDB$FIELD_SOURCE = RDB$11
  RDB$COMPUTED_BLR = NULL
  DS = NULL
  DS1 = NULL
  RDB$FIELD_TYPE = 261
  RDB$DIMENSIONS = NULL

: [Fetch]  ...
  RDB$FIELD_NAME = TEAM_LEADER
  RDB$FIELD_SOURCE = EMPNO
  RDB$COMPUTED_BLR = NULL
  DS = NULL
  DS1 = NULL
```

```
  RDB$FIELD_TYPE = 7
  RDB$DIMENSIONS = NULL
: [Fetch] ...
  RDB$FIELD_NAME = PRODUCT
  RDB$FIELD_SOURCE = PRODTYPE
  RDB$COMPUTED_BLR = NULL
  DS = NULL
  DS1 = DEFAULT 'software'
  RDB$FIELD_TYPE = 37
  RDB$DIMENSIONS = NULL
: [Fetch] ...
  RDB$FIELD_NAME = PRODUCT
  RDB$FIELD_SOURCE = PRODTYPE
  RDB$COMPUTED_BLR = NULL
  DS = NULL
  DS1 = DEFAULT 'software'
  RDB$FIELD_TYPE = 37
  RDB$DIMENSIONS = NULL
End of file reached
: [Commit (Hard commit)](1836)
pFIBDataSet1: [Execute] SELECT *
  FROM project
Execute tick count 0
pFIBDataSet1: [Fetch] ...
  PROJ_ID = VBASE
  PROJ_NAME = 68809
  PROJ_DESC = Test
  TEAM_LEADER = 45
  PRODUCT = other
```

Auch *TpFIBDataSet* schaut erst einmal neugierig in die Systemtabellen und merkt sich das Ergebnis und auch hier wird »ohne Not« nicht mehr als ein Datensatz abgerufen.

Fazit: *TpFIBDataSet* spielt in Sachen Performance durchaus »in der ersten Liga«.

9.2.2 Update, Commit, Close und Disconnect

Die Abfrage von Daten ist für gewöhnlich die am meisten zeitkritische Aktion, sie ist aber nicht die einzige. Wir wollen uns nun ansehen, wie schnell die übrigen Aktionen ausgeführt werden.

Auch hier liegen die einzelnen Komponenten wieder deutlich auseinander. Schauen wir uns an, woran das liegen könnte.

	Q	S	Op	Update	Commit	Close	Disconnect	S
BDE TTable	1		2	3 098	213	143	3 335	1
BDE TQuery	1		2	1 223	266	200	339	1
IBX TIBTable	1		2	2 915	450	2	187	1
IBX TIBQuery	1		2	1 288	372	1	332	1
IBX TIBDataSet	1		4	992	371	1	180	1
dbExpress ctTable	1		3	3 043	161	124	453	1
dbExpress ctQuery	1		5	3 379	166	144	293	1
dbExpress ohne TClientDataSet	1		2	1 146	158	75	326	1
FIBPlus pFIBDataSet	1		2	2 568	239	1	428	1
FIBPlus pFIBQuery	1		2	1 254	294	3	371	1
IBO TIB_Query	1		2	4 418	530	1	996	1
IBO TIBOQuery	1		5	3 932	3 826	373	1 514	1
BDE TTable	1		2	2 748	207	147	3 321	1

Abbildung 9.17: Die übrigen Ausführungszeiten

BDE

Fangen wir nun diesmal mit *TTable* an:

```
22:49:09  SQL Prepare: INTRBASE -
UPDATE PROJECT
  SET PROJ_NAME=?
  WHERE PROJ_ID=?
    AND PROJ_NAME=?
    AND TEAM_LEADER=?
    AND PRODUCT=?
108     22:49:09  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
109     22:49:09  SQL Vendor: INTRBASE - isc_dsql_prepare
110     22:49:09  SQL Vendor: INTRBASE - isc_dsql_sql_info
111     22:49:09  SQL Vendor: INTRBASE - isc_vax_integer
112     22:49:09  SQL Data In: INTRBASE - Param = 1, Name = PROJ_NAME,
        Type = fldZSTRING, Precision = 20, Scale = 0, Data = 81355
113     22:49:09  SQL Data In: INTRBASE - Param = 2, Name = PROJ_ID,
        Type = fldZSTRING, Precision = 5, Scale = 0, Data = VBASE
114     22:49:09  SQL Data In: INTRBASE - Param = 3, Name = PROJ_NAME,
        Type = fldZSTRING, Precision = 20, Scale = 0, Data = 31683
115     22:49:09  SQL Data In: INTRBASE - Param = 4, Name = TEAM_LEADER,
        Type = fldINT16, Precision = 1, Scale = 0, Data = 45
116     22:49:09  SQL Data In: INTRBASE - Param = 5, Name = PRODUCT,
        Type = fldZSTRING, Precision = 12, Scale = 0, Data = other
117     22:49:09  SQL Execute: INTRBASE -
UPDATE PROJECT
  SET PROJ_NAME=?
  WHERE PROJ_ID=?
    AND PROJ_NAME=?
```

```
    AND TEAM_LEADER=?
    AND PRODUCT=?
118     22:49:09  SQL Vendor: INTRBASE - isc_dsql_execute
119     22:49:09  SQL Data In: INTRBASE - Rows affected = 1
120     22:49:09  SQL Stmt: INTRBASE - Close
121     22:49:09  SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Zunächst kommt ein ganz gewöhnlicher UPDATE-Befehl. *TTable* erkennt, dass hier nur eine Spalte verändert wird, und generiert die SQL-Anweisung entsprechend. Anhand der umfangreichen WHERE-Bedingung erkennt man, dass *UpdateMode* von *Table1* den Wert *upWhereAll* hat – mit *upWhereKeyOnly* könnte hier ein wenig optimiert werden. Anschließend werden die Parameter übertragen, die Anweisung wird ausgeführt und die Sache geschlossen.

```
122     22:49:09  SQL Prepare: INTRBASE -
  SELECT PROJ_ID ,PROJ_NAME ,PROJ_DESC ,TEAM_LEADER ,PRODUCT
    FROM PROJECT
    WHERE PROJ_NAME=?
123     22:49:09  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
124     22:49:09  SQL Vendor: INTRBASE - isc_dsql_prepare
125     22:49:09  SQL Data In: INTRBASE - Param = 1, Name = PROJ_NAME,
        Type = fldZSTRING, Precision = 20, Scale = 0, Data = 81355
126     22:49:09  SQL Execute: INTRBASE - SELECT PROJ_ID ,PROJ_NAME
        ,PROJ_DESC ,TEAM_LEADER ,PRODUCT  FROM PROJECT WHERE PROJ_NAME=?
127     22:49:09  SQL Vendor: INTRBASE - isc_dsql_execute
128     22:49:09  SQL Stmt: INTRBASE - Fetch
129     22:49:09  SQL Vendor: INTRBASE - isc_dsql_fetch
130     22:49:09  SQL Stmt: INTRBASE - Fetch
131     22:49:09  SQL Vendor: INTRBASE - isc_dsql_fetch
132     22:49:09  SQL Stmt: INTRBASE - EOF
133     22:49:09  SQL Stmt: INTRBASE - Close
134     22:49:09  SQL Vendor: INTRBASE - isc_dsql_free_statement
135     22:49:09  SQL Stmt: INTRBASE - Close
136     22:49:09  SQL Vendor: INTRBASE - isc_dsql_free_statement
```

Nun generiert *TTable* eine Abfrage, um nur den geänderten Datensatz neu vom Server zu laden – damit könnten beispielsweise über Trigger geänderte Daten mit erfasst werden.

```
137     22:49:09  SQL Transact: INTRBASE - XACT Commit
138     22:49:09  SQL Vendor: INTRBASE - isc_commit_transaction
139     22:49:09  SQL Transact: INTRBASE - XACT ReadCommitted
140     22:49:09  SQL Connect: INTRBASE - Disconnect GENERAL
141     22:49:09  SQL Vendor: INTRBASE - isc_detach_database
```

Schließlich wird ein Commit ausgeführt und die Verbindung zur Datenbank getrennt.

Vergleichen wir dies mit *TQuery*, nehmen dort aber noch einmal den *Open*-Befehl mit auf:

```
10      23:16:55  SQL Execute: INTRBASE -
  SELECT *
    FROM project
    order by proj_id

11      23:16:55  SQL Vendor: INTRBASE - isc_dsql_execute
12      23:16:55  SQL Stmt: INTRBASE - Fetch
13      23:16:55  SQL Vendor: INTRBASE - isc_dsql_fetch
...
18      23:16:56  SQL Prepare: INTRBASE -
  UPDATE project
    SET proj_name = ?
    WHERE proj_id = ?
19      23:16:56  SQL Vendor: INTRBASE - isc_dsql_allocate_statement
20      23:16:56  SQL Vendor: INTRBASE - isc_dsql_prepare
21      23:16:56  SQL Vendor: INTRBASE - isc_dsql_sql_info
22      23:16:56  SQL Vendor: INTRBASE - isc_vax_integer
23      23:16:56  SQL Data In: INTRBASE - Param = 1, Name = ,
        Type = fldZSTRING, Precision = 5, Scale = 0, Data = 12796
24      23:16:56  SQL Data In: INTRBASE - Param = 2, Name = ,
        Type = fldZSTRING, Precision = 5, Scale = 0, Data = VBASE
25      23:16:56  SQL Execute: INTRBASE -
  UPDATE project
    SET proj_name = ?
    WHERE proj_id = ?
26      23:16:56  SQL Vendor: INTRBASE - isc_dsql_execute
27      23:16:56  SQL Data In: INTRBASE - Rows affected = 1
28      23:16:56  SQL Stmt: INTRBASE - Close
29      23:16:56  SQL Vendor: INTRBASE - isc_dsql_free_statement
        23:16:56  SQL Stmt: INTRBASE - Fetch
        ...
        23:16:56  SQL Stmt: INTRBASE - Fetch
        ...
        23:16:56  SQL Stmt: INTRBASE - Fetch
        ...
        23:16:56  SQL Stmt: INTRBASE - Fetch
        ...
54      23:16:56  SQL Stmt: INTRBASE - Fetch
  ...
60      23:16:56  SQL Stmt: INTRBASE - Fetch
61      23:16:56  SQL Vendor: INTRBASE - isc_dsql_fetch
62      23:16:56  SQL Stmt: INTRBASE - EOF
```

```
63      23:16:56  SQL Stmt: INTRBASE - Reset
64      23:16:56  SQL Vendor: INTRBASE - isc_dsql_free_statement
65      23:16:56  SQL Transact: INTRBASE - XACT Commit
66      23:16:56  SQL Vendor: INTRBASE - isc_commit_transaction
67      23:16:56  SQL Transact: INTRBASE - XACT ReadCommitted
68      23:16:56  SQL Stmt: INTRBASE - Close
69      23:16:56  SQL Vendor: INTRBASE - isc_dsql_free_statement
70      23:16:56  SQL Connect: INTRBASE - Disconnect GENERAL
71      23:16:56  SQL Vendor: INTRBASE - isc_detach_database
```

Wir hatten vorhin stillschweigend akzeptiert, dass *TQuery* alle Datensätze abruft, obwohl diese nicht angezeigt werden. Schauen wir uns die Sache noch einmal an: Zunächst wird das SELECT-Statement ausgeführt, dann ein einzelner Datensatz abgerufen. Danach erfolgt die UPDATE-Anweisung, anschließend werden die restlichen Datensätze abgerufen. Warum?

Sobald ein *Commit* ausgeführt wird, verliert die Transaktion ihren Kontext, der Client kann dann keine Daten mehr vom Server abrufen. Bei der BDE ist es jedoch üblich, dass die Daten aber auch nach einem *Commit* zur Verfügung stehen. Was tut *TQuery* in ihrer Not? Sie ruft alle Datensätze ab und speichert sie lokal.

Nun sind das hier sechs Datensätze, zudem wurden bei der Messung Client und Server auf demselben Rechner betrieben. Wären es jetzt aber 10000 Datensätze, die zudem noch über ein Netzwerk übertragen werden müssten, sähe die Sache schon ein wenig anders aus. Würde man jedoch die Datenmenge erst schließen und erst dann *Commit* aufrufen, könnte die Ausführung dieser Anweisung in etwa 150 µs erfolgen.

Bleibt noch die Frage, warum *TQuery* besser abgeschnitten hat: Die Datenmenge wurde nicht aktualisiert. Während *TTable* den geänderten Datensatz nachgeladen hat, blieb die Anzeige bei *TQuery* unverändert. Von daher war der Vergleich nicht ganz fair. Auf der anderen Seite kommt es nicht selten vor, dass Datenmengen erst angezeigt und dann geändert werden, ohne dass dann noch eine Kontrolle der Änderung erfolgen muss.

Fazit: Bleiben Sie bei *TQuery*, wenn Sie noch mit der BDE arbeiten. Aber rufen Sie erst *Close* und dann *Commit* auf!

IBX

Zunächst fällt auf, dass die *Close*-Zeiten allesamt minimal sind. Wenn wir an das denken, was uns gerade bei *TQuery* aufgefallen ist, dann wundert uns das auch nicht weiter – die Datenmenge ist nach dem Aufruf von *Commit* nämlich schon geschlossen.

Beginnen wir mit *TIBTable*:

```
IBTable1: [Execute]
  update PROJECT
    set PROJ_ID = :PROJ_ID,
      PROJ_NAME = :PROJ_NAME,
      PROJ_DESC = :PROJ_DESC,
      TEAM_LEADER = :TEAM_LEADER,
      PRODUCT = :PRODUCT
    where RDB$DB_KEY = :IBX_INTERNAL_DBKEY
  PROJ_ID = VBASE
  PROJ_NAME = 60147
  PROJ_DESC = <BLOB>
  TEAM_LEADER = 45
  PRODUCT = other
  IBX_INTERNAL_DBKEY = „
IBTable1: [Prepare]
  select PROJECT.*,
      RDB$DB_KEY as IBX_INTERNAL_DBKEY
    from PROJECT
    where RDB$DB_KEY = :IBX_INTERNAL_DBKEY
    Plan: PLAN (PROJECT INDEX ())
IBTable1: [Execute] ...
IBTable1: [Fetch] ...
IBTransaction1: [Commit (Hard commit)]
IBDatabase1: [Disconnect]
```

Die UPDATE-Anweisung ist bereits vorbereitet, sie wird nun ausgeführt, anschließend wird der geänderte Datensatz nachgeladen. Vergleichen wir das mit *TIBDataSet*:

```
IBDataSet1: [Execute] update PROJECT
set
  PROJ_NAME = :PROJ_NAME,
  PROJ_DESC = :PROJ_DESC,
  TEAM_LEADER = :TEAM_LEADER,
  PRODUCT = :PRODUCT
where
  PROJ_ID = :OLD_PROJ_ID

  PROJ_NAME = 66905
  PROJ_DESC = <BLOB>
  TEAM_LEADER = 45
  PRODUCT = other
  OLD_PROJ_ID = VBASE
IBDataSet1: [Execute] Select
  PROJ_ID,
  PROJ_NAME,
```

```
  PROJ_DESC,
  TEAM_LEADER,
  PRODUCT
from PROJECT
where
  PROJ_ID = :PROJ_ID

  PROJ_ID = VBASE
IBDataSet1: [Fetch] ...
IBTransaction1: [Commit (Hard commit)]
IBDatabase1: [Disconnect]
```

Die Vorgehensweise ist im Prinzip dieselbe, nur die Anweisung zur Aktualisierung des geänderten Datensatzes ist eine andere. Wenn man bedenkt, dass die UPDATE-Aktion bei *TIBTable* etwa dreimal so lange braucht und dass hier deutlich mehr Anweisungen ausgeführt werden als nur das Nachladen eines Datensatzes, dann kann man nur zu dem Schluss kommen, dass die bei *TIBTable* gewählte Vorgehensweise um Größenordnungen unperformanter sein muss. Aber wir waren ja ohnehin der Ansicht, *TIBTable* lieber nicht einsetzen zu wollen ....

Warum braucht *TIBQuery* länger als *TIBDataSet*?

```
IBQuery2: [Prepare] UPDATE project
  SET proj_name = :name
  WHERE proj_id = :id

  Plan: PLAN (PROJECT INDEX (RDB$PRIMARY12))
IBQuery2: [Execute] UPDATE project
  SET proj_name = :name
  WHERE proj_id = :id

  NAME = 53003
  ID = VBASE
IBTransaction1: [Commit (Hard commit)]
IBDatabase1: [Disconnect]
```

Weil die UPDATE-Anweisung noch nicht vorbereitet wurde. Und das dauert länger, als einen geänderten Datensatz nachzuladen – Letzteres erfolgt hier nämlich nicht.

Fazit: Benötigen Sie eine Aktualisieung der Anzeige, dann arbeiten Sie mit *TIBDataSet*. Können Sie darauf verzichten, dann ist *TIBQuery* die (etwas!) performantere Alternative, gegebenenfalls müssen Sie *Prepare* explizit aufrufen.

dbExpress

Ob Sie bei *TSQLDataSet ctTable* oder *ctQuery* einstellen, spielt keine Rolle. Wird für die Client-Datenmenge *ApplyUpdates* aufgerufen, dann erstellt der Provider ein UPDATE-Statement und trägt damit die Änderungen in die Datenmenge ein.

```
INTERBASE - isc_dsql_allocate_statement
  update PROJECT  set
      PROJ_NAME = ?
    where
      PROJ_ID = ? and
      PROJ_NAME = ? and
      TEAM_LEADER = ? and
      PRODUCT = ?
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_transaction
INTERBASE - isc_dsql_free_statement
```

Der Unterschied zur Lösung ohne *TClientDataSet* kann im Monitorprotokoll vernachlässigt werden:

```
INTERBASE - isc_dsql_allocate_statement
  UPDATE project
    SET proj_name = ?
    WHERE proj_id = ?
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_sql_info
INTERBASE - isc_vax_integer
INTERBASE - isc_dsql_describe_bind
INTERBASE - SQLDialect = 1
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_commit_transaction
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
```

Der wesentliche Unterschied besteht darin, dass die WHERE-Klausel nur den Primärschlüssel aufnimmt. Der Verdacht, dass damit die Zeitunterschiede erklärt werden können, hat sich jedoch nicht bestätigt – mit Hilfe von *UpdateMode* kann man ja den Provider dazu bringen, nur den Primärschlüssel zu verwenden. Man sollte jedoch nicht vergessen, dass im Update-Fall zunächst die Client-Datenmenge aktualisiert werden muss, dann generiert der Provider eine SQL-Anweisung und erst dann erfolgt die Aktualisierung über die Client-DLL.

Bei der Verwendung einer Datenmengenkette ist *TSQLDataSet* bereits geschlossen, wenn *Commit* aufgerufen wird – das wird erledigt, wenn der letzte Datensatz in die Client-Datenmenge geladen wurde. Von daher ist wenig verwunderlich, dass bei dbExpress die Transaktionen am schnellsten bestätigt werden. Beim Aufruf von *Close* muss dann nur noch die Client-Datenmenge verworfen werden – hier den Speicher freizuräumen scheint ein wenig Zeit in Anspruch zu nehmen. Die Disconnect-Zeiten liegen dann im unteren Mittelfeld.

FIBPlus

Bei FIBPlus ist die Aktualisierung über eine eigene Query schneller, als wenn dies über *TpFIBDataSet* erledigt wird. Aufmerksame Leser ahnen sicher schon, woran das liegt. Schauen wir uns erst die Version mit *TpFIBDataSet* an:

```
pFIBDataSet1: [Prepare]
  UPDATE PROJECT SET
      PROJ_NAME = ?PROJ_NAME,
      PROJ_DESC = ?PROJ_DESC,
      TEAM_LEADER = ?TEAM_LEADER,
      PRODUCT = ?PRODUCT
    WHERE
      PROJ_ID = ?OLD_PROJ_ID
    Plan:
pFIBDataSet1: [Execute] ...
pFIBDataSet1: [Prepare]
  SELECT *
    FROM project
    WHERE
      (
         project.PROJ_ID = ?OLD_PROJ_ID
      )
    Plan:
pFIBDataSet1: [Execute] ...
pFIBDataSet1: [Fetch] ...
pFIBTransaction1: [Commit (Hard commit)](2112)
pFIBDatabase1: [Disconnect]
```

Nach dem Ändern des Datensatzes wird der geänderte Datensatz nachgeladen, um Änderungen durch TRIGGER mit zu erfassen. Um das dafür erforderliche Statement vorzubereiten und auszuführen, braucht man natürlich ein wenig Zeit. Bei der Lösung mit *TpFIBQuery* werden die Änderungen nur auf den Server geschrieben:

```
pFIBQuery1: [Prepare]
  UPDATE project
    SET proj_name = :name
    WHERE proj_id = :id
pFIBQuery1: [Execute] ...
pFIBTransaction1: [Commit (Hard commit)](2116)
pFIBDatabase1: [Disconnect]
```

Wir haben vorhin gesehen, dass FIBPlus die Datenmengen schneller offen hat als IBX. Dies liegt vor allem daran, dass bei IBX alle Statements bereits beim Öffnen der Datenmenge vorbereitet werden. Dies erfolgt bei FIBPlus erst beim Ausführen der Anweisung, was dann im Vergleich zu IBX länger braucht. Was man nun bräuchte, wäre die Möglichkeit, diese Anweisungen manuell vorzubereiten – eine solche habe ich jedoch nicht gefunden.

IBO

Die längsten Update- und Commit-Zeiten finden sich bei IBO. Dabei finden sich im Monitorprotokoll (abgesehen von der hier wieder herausgekürzten Ausführlichkeit) keine Auffälligkeiten – nun gut, die Zeiten sind ja auch nicht um Größenordnungen schlechter.

```
PREPARE STATEMENT
  update PROJECT
    set
      PROJ_NAME = ? /* PROJ_NAME */ ,
      PROJ_DESC = ? /* PROJ_DESC */ ,
      TEAM_LEADER = ? /* TEAM_LEADER */ ,
      PRODUCT = ? /* PRODUCT */
    where
      PROJ_ID = ? /* OLD_PROJ_ID */
      PLAN (PROJECT INDEX (RDB$PRIMARY12))
DESCRIBE INPUT
STMT_HANDLE = 10911752
PARAMS = [  Version 1 SQLd 5 SQLn 5
  < SQLType: 448 SQLLen: 20 > = <NIL>
  < SQLType: 521 SQLLen: 8 > = <NIL>
  < SQLType: 501 SQLLen: 2 > = <NIL>
  < SQLType: 448 SQLLen: 12 > = <NIL>
  < SQLType: 452 SQLLen: 5 > = <NIL> ]
EXECUTE STATEMENT
```

```
PARAMS = [  Version 1 SQLd 5 SQLn 5
  [PROJ_NAME] = '54990'
  [PROJ_DESC] = BLOB ID ( 132, 25 )
  [TEAM_LEADER] = 45
  [PRODUCT] = 'other'
  [OLD_PROJ_ID] = 'VBASE' ]
CLOSE CURSOR
CLOSE CURSOR
COMMIT
COMMIT
DEALLOCATE STATEMENT
DEALLOCATE STATEMENT
DEALLOCATE STATEMENT
DISCONNECT DATABASE
```

Kurioserweise werden die Datensätze bei *TIBOQuery* nach dem Aufruf von *Update* noch einmal vollständig heruntergeladen; Wenn man sich dann doch die Unterschiede bei den *Close*-Zeiten vor Augen führt, erkennt man, dass hier – ähnlich wie bei *TQuery* – vor dem *Commit* die Datensätze zwischengespeichert werden. Das erklärt hier auch die deutlich längere *Commit*-Zeit.

Fazit: Auch hier fallen die Komponenten von IBObjects durch verbesserungsfähige Performance auf. Die Unterschiede von *TIB_Query* und *TIBOQuery* erklären sich dadurch, dass die Datensicht bei *TIBOQuery* gehalten wird.

9.3 Listendruck

Die Tabelle *project* mit ihren sechs Datensätzen stellt natürlich keine besondere Herausforderung dar – dieses Beispiel wurde nur deshalb gewählt, damit die Monitorprotokolle halbwegs überschaubar bleiben.

Um die Sache auch noch mal an einer größeren Datenmenge zu testen, wollen wir nun für die Tabelle *t_adressen* unserer Testdatenbank eine Liste aller Kunden mit Vor- und Nachnamen erstellen. Solche Aufgaben fallen beispielsweise im Bereich des Listendrucks an – wir wollen uns aber hier die Zeit sparen, diese Liste tatsächlich zu drucken, es wären hier auch keine Erkenntnisse bezüglich der untersuchten Komponenten zu erwarten.

9.3.1 Erstellen der Liste

Für alle Komponenten, die von *TDataSet* abgeleitet sind, verwenden wir zur Erstellung der Liste die folgende Routine:

```
procedure Tfrm_main.ListeMachen(ADataSet: TDataSet);
var
  sl: TStringList;
  s: string;
begin
  sl := TStringList.Create;
  try
    with ADataSet do
    begin
      while not EOF do
      begin
        s := FieldByName('ID').AsString;
        s := s + ';' + FieldByName('Vorname').AsString;
        s := s + ';' + FieldByName('Nachname').AsString;
        sl.Add(s);
        Next;
      end;
    end; {with ADataSet do}
  finally
    sl.Free;
  end;
end; {procedure Tfrm_main.ListeMachen}
```

Dabei wird eine Stringliste erstellt, durch die Datenmenge iteriert und für jeden Datensatz ein String mit ID, Vorname und Nachname gebildet, der dann der Liste hinzugefügt wird.

Die Routinen für *TpFIBQuery* und *TIB_Query* sehen fast identisch aus.

9.3.2 Die Zeiten

Werfen wir einen Blick auf die Ausführungszeiten:

	Connect	Start Transaction	Open	Liste	Commit	Close	Disconnect	Summe
BDE TTable	1 040 816	300	24 618	793 739	464	407	3 432	1 863 776
BDE TQuery	1 046 226	297	4 301	672 856	222	1 871	429	1 726 202
IBX TIBTable	1 042 843	184	21 221	793 291	923	2	205	1 858 669
IBX TIBQuery	1 044 351	181	4 726	757 101	760	2	213	1 807 334
IBX TIBDataSet	1 045 940	184	4 459	760 631	773	2	194	1 812 183
dbExpress ctTable	1 029 953	157	370 768	166 793	209	533	524	1 568 937
dbExpress ctQuery	1 028 979	154	410 143	164 401	221	6 047	493	1 610 438
dbExpress ohne TClientDataSet	1 042 369	147	3 543	394 751	227	109	487	1 441 633
FIBPlus pFIBDataSet	1 033 116	222	4 123	1 232 684	570	2	284	2 271 001
FIBPlus pFIBQuery	1 039 160	197	3 482	452 261	341	2	307	1 495 750
IBO TIB_Query	1 045 886	8	23 301	3 338 040	7 147	1	9 897	4 424 280
IBO TIBOQuery	1 044 667	8	51 082	2 651 681	605	6 941	10 313	3 765 297
BDE TTable	1 060 409	301	24 768	797 374	223	412	3 492	1 886 979

Abbildung 9.18: Die Ausführungszeiten beim Erstellen der Liste

Warum bei *IBObjects* der Aufruf von *StartTransaction* so schnell geht, haben wir ja bereits besprochen.

Wenn wir uns die Zeiten für *Open* ansehen, dann fällt zunächst auf, dass die Zeiten für die dbExpress-Datenmengenketten böse aus dem Rahmen fallen. Das liegt einfach daran, dass hier bereits alle Datensätze auf den Client übertragen werden, was natürlich ein wenig dauert – dafür geht das Erstellen der Liste dann umso schneller. Korrekterweise muss man beide Zeiten addieren und dann vergleichen und dann schlagen sich diese Komponenten hervorragend.

Noch einen Tick schneller geht es, wenn wir unidirektionale Datenmengen nehmen, also dbExpress ohne *TClientDataSet* oder *TpFIBQuery*. Die Hoffnung, *TQuery* nennenswert beschleunigen zu können, indem man die Eigenschaft *UniDirectional* auf *true* setzt, hat sich allerdings nicht bestätigt.

Eine Stufe weniger performant sind BDE und IBX, die Unterschiede sind nicht wirklich relevant. Die Komponente *TpFIBDataSet* enttäuscht etwas und die Komponenten von IBObjects bilden das Schlusslicht.

9.4 Fazit

Es bleibt die Frage, welche Komponentengruppe man letztendlich verwenden soll. Beginnen wir diese Fragestellung von der anderen Seite: Welche Komponentengruppe sollte man lieber nicht verwenden?

Zunächst einmal würde ich IBObjects ausschließen. Diese Komponenten sind wegen der Vielzahl ihrer Eigenschaften sehr unübersichtlich, man braucht also viel Zeit, um sich hier einzuarbeiten. Dass einige Datenmengenkomponenten nicht von *TDataSet* abgeleitet und somit zu einer Vielzahl anderer Komponenten inkompatibel sind, ist ein weiterer gewichtiger Grund. So etwas würde man vielleicht noch in Kauf nehmen, wenn diese Komponenten eine überragende Performance aufweisen würden – das Gegenteil ist jedoch der Fall. Gründe, dafür auch noch Geld auszugeben, kann ich nicht erkennen.

Die BDE schlägt sich performancemäßig ganz wacker, viele Programmierer haben darauf viel Erfahrung gesammelt, entsprechende Fachliteratur ist weit verbreitet, in den Internet-Foren findet man immer jemanden, der das aktuelle Problem schon einmal gelöst hat. Allerdings ist die BDE ein auslaufendes Modell. Das soll nicht heißen, dass sie irgendwann nicht mehr funktioniert, aber beim nächsten »Jahr-2000-Problem« haben sich darauf aufbauende Programme dann wohl erledigt. Man wird sich wohl davon trennen müssen.

Es bleiben drei Komponentengruppen, von denen ich zumindest nicht abraten kann.

FIBPlus würde ich als etwa gleichwertig mit IBX einstufen – mal hat die eine Komponentengruppe die Nase vorne, mal die andere. FIBPlus ist ein wenig unübersichtlicher, aber mehr Eigenschaften bedeuten ja meist auch mehr Möglichkeiten, wenn man erst einmal weiß, was man tun muss. Das bleibt wohl überwiegend eine Sache der persönlichen Vorliebe.

Den größten Nachteil sehe ich in der relativ überschaubaren Entwickler-Gemeinde, da kann es schon einmal etwas länger dauern, bis man jemand gefunden hat, der einem weiterhelfen kann. Die Frage der Kosten würde ich nicht überbewerten – im Vergleich zu den Personalkosten bei der Entwicklung kann das vernachlässigt werden.

Mit IBX entwickeln mehr Programmierer, Fachliteratur dazu ist auf dem Markt, die Kinderkrankheiten früherer Versionen scheinen weitgehend behoben zu sein und diese Komponenten sind ohnehin bei Delphi (und »Verwandten«) dabei.

Bleibt die Frage nach dbExpress: Wir haben hier gesehen, dass diese Komponenten im Hinblick auf die Performance erste Wahl sind. Da sie nicht viel Möglichkeiten bieten, hat man sich auch sehr schnell eingearbeitet. Man sollte jedoch nicht übersehen, dass die Vorgehensweise in vielen Fällen doch ein wenig anders ist, als man es von der BDE oder IBX her kennt. Und man muss sich damit abfinden, dass es einen solchen Komfort wie das automatische Setzen eines Generatorwertes einfach nicht gibt.

Der wirkliche Hauptvorteil liegt daran, dass man nicht an ein Datenbanksystem gebunden ist. Solange man sich auf den ausgetretenen Pfaden der Datenbankprogrammierung bewegt, sollte die Umstellung des Datenbanksystems mit vernachlässigbarem Aufwand durchzuführen sein.

Die Entscheidung, welche Komponentengruppe letztlich zu nehmen ist, möchte ich Ihnen nicht abnehmen (und Sie würden sich vermutlich auch energisch gegen entsprechende Versuche verwahren). Wir werden in diesem Buch noch mal etwas ausführlicher IBX und dbExpress besprechen, zwischen diesen beiden Alternativen dürfte normalerweise die letzte Entscheidung gefällt werden.

10 IBX

IBX ist eine Sammlung von Komponenten, die direkt mit der Client-DLL von InterBase arbeiten. Somit kann auf keine andere Datenbank als auf die verschiedenen InterBase-Derivate zugegriffen werden. Solche Komponenten-Sammlungen für InterBase gibt es inzwischen einige, im Gegensatz zu FIBPlus und IBObjects wird IBX jedoch mit Delphi ausgeliefert.

10.1 Arbeiten mit IBX

Wir wollen uns zunächst anhand einiger Beispiele die Arbeit mit IBX ansehen. Anschließend wollen wir die vorhandenen Komponenten im Detail besprechen.

10.1.1 Zugriff auf eine Tabelle

Beginnen wir mit dem einfachsten Fall, dem Zugriff auf eine Tabelle. Zu diesem Zweck bestücken wir ein Formular mit den folgenden Komponenten:

- TIBDatabase
- TIBTransaction
- TIBDataSet
- TDataSource
- TDBGrid
- TDBNavigator

Für eine so einfache Beispielanwendung ersparen wir uns das Umbenennen der Komponenten.

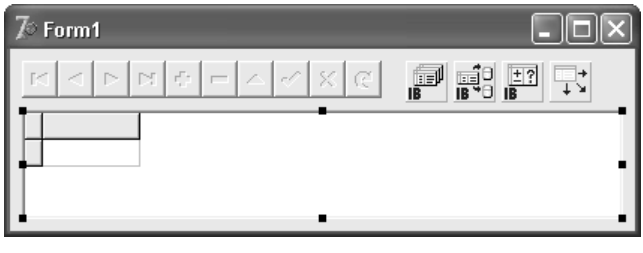


Abbildung 10.1: Die benötigten Komponenten

Wir wollen nun auf die Tabelle *employee* der InterBase-Beispiel-Datenbank *employee.gdb* zugreifen. Dazu führen wir zunächst einen Doppelklick auf *IBDatabase1* aus:

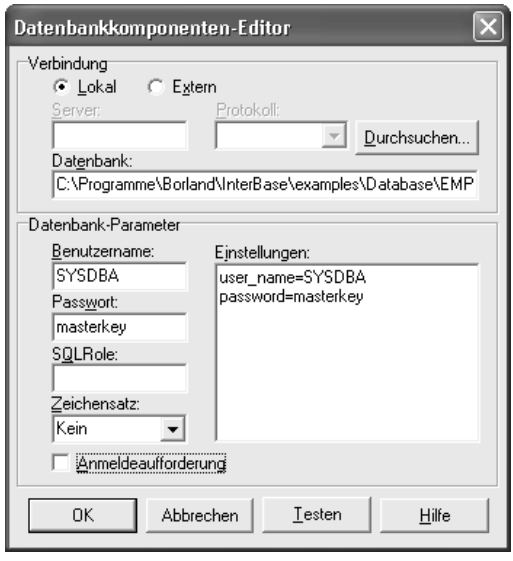


Abbildung 10.2: Der Komponenten-Editor von TIBDatabase

Wählen Sie zunächst die Datei der Datenbank aus und weisen Sie anschließend den Benutzernamen und das Passwort zu. Um nicht ständig von einem Login-Dialog belästigt zu werden, deaktivieren wir die Option *Anmeldeaufforderung*.

Direkt von diesem Dialog aus lässt sich die Datenbankverbindung testen, nach einem erfolgreichen Test schließen wir den Dialog mit *OK*. Die Eigenschaft *DefaultTransaction* setzen wir auf *IBTransaction1*, deren Eigenschaft *DefaultDatabase* setzen wir wiederum auf *IBDatabase1*. Nun setzen wir die Eigenschaft *Connected* von *IBDatabase1* auf *true*.

TIBDataSet

Wenden wir uns der Komponente *IBDataSet1* zu: Zunächst setzen wir *Database* auf *IBDatabase1*. Die Eigenschaft *Transaction* sollte dabei automatisch auf *IBTransaction1* gesetzt werden – ist dies nicht der Fall, holen wir es nach.

Nun rufen wir den Eigenschaftseditor von *SelectSQL* auf:

Mit einem Doppelklick auf den Tabellennamen *employee* und einem weiteren Doppelklick auf das Jokerzeichen * erzeugen wir das benötigte SELECT-Statement. Leider vereinfacht dieser Eigenschaftseditor lediglich das Erstellen von einfachen Anweisungen – für kompliziertere Anweisungen verwenden Sie am besten ein externes Tool und kopieren die Anweisung über die Zwischenablage.

Abbildung 10.3: Der SQL-Editor

Nun setzen Sie *Active* auf *true* und hängen über *DataSource1* das Grid und den Navigator an diese Datenmenge.

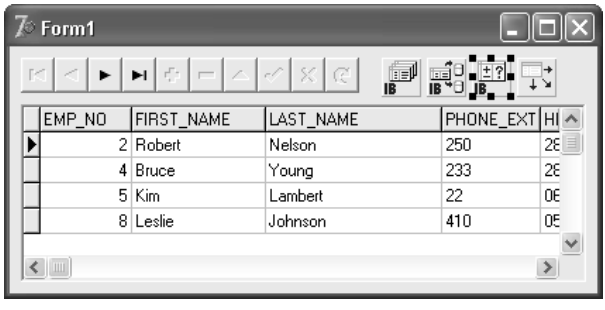


Abbildung 10.4: Eine Read-Only-Datenmenge

Wir erhalten zunächst nur eine Read-Only-Datenmenge, wie Sie unschwer an den verfügbaren Buttons im Navigator sehen können. Daran würde sich auch nichts ändern, wenn Sie das Projekt starten.

Eine aktualisierbare Datenmenge

Damit die Datenmenge aktualisierbar wird, benötigt *TIBDataSet* SQL-Anweisungen für das Einfügen, Ändern und Löschen von Datensätzen – keine Sorge, man kann sich diese automatisch erstellen lassen. Rufen Sie dazu aus dem Kontextmenü (rechte Maustaste) dieser Komponente den Eintrag DATASET-EDITOR auf:

Hier müssen Sie lediglich festlegen, welches die Schlüsselfelder sind und welche Spalten aktualisiert werden sollen. Per Voreinstellung sind jeweils alle Spalten gewählt. Bei den Aktualisierungsfeldern lassen wir das so, und wenn Sie auf den Button *Primärschlüssel auswählen* klicken, dann wird die Spalte *emp_no* als einzige Schlüsselspalte ausgewählt.

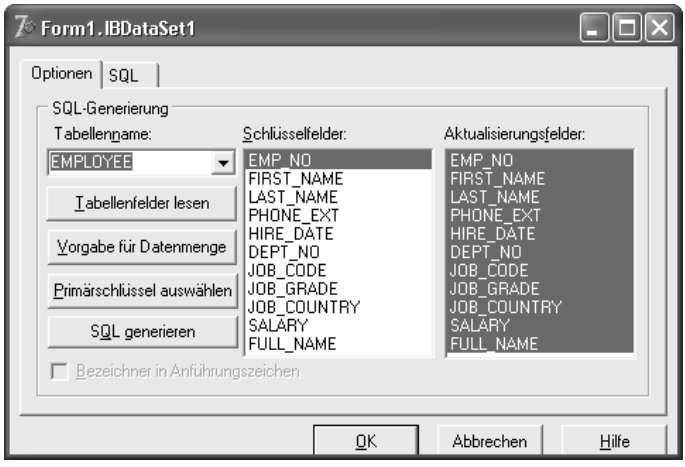


Abbildung 10.5: Erstellen von SQL-Anweisungen

Darüber, dass die Buttons am unteren Rand grafisch etwas »verunglückt« sind, sehen wir gnädig hinweg – immerhin sind die anderen Buttons inzwischen breit genug für ihre Beschriftung und so habe ich die Hoffnung, dass in ein, zwei Versionen auch das andere Problemchen gelöst ist ... – und lassen mit *SQL generieren* den Dialog arbeiten.

Werfen wir einen kurzen Blick auf die erzeugten SQL-Anweisungen – die Spaltenlisten habe ich dabei ein wenig gekürzt, damit das nicht ganz so ausufert:

```
update EMPLOYEE
  set
    EMP_NO = :EMP_NO,
    FIRST_NAME = :FIRST_NAME,
    ...
FULL_NAME = :FULL_NAME
  where
    EMP_NO = :OLD_EMP_NO

insert into EMPLOYEE
    (EMP_NO, FIRST_NAME, ... FULL_NAME)
  values
    (:EMP_NO, :FIRST_NAME, ... :FULL_NAME)

delete from EMPLOYEE
  where
    EMP_NO = :OLD_EMP_NO

Select
    EMP_NO,
```

```
  FIRST_NAME,
    ...
  FULL_NAME
from EMPLOYEE
where
  EMP_NO = :EMP_NO
```

Intime Kenner der Tabelle *employee* werden bei diesen Statements »zusammenzucken«, ist doch die Spalte *full_name* eine berechnete Spalte und kann somit nicht aktualisiert werden. Der Versuch, die Datenmenge zu öffnen, wird somit auch gleich mit einer Fehlermeldung quittiert:

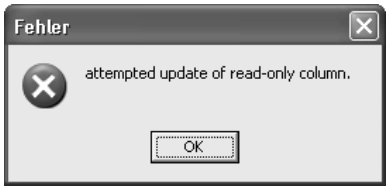


Abbildung 10.6: Versuch, berechnete Spalten zu ändern

Da *TIBDataSet* die einzelnen SQL-Statements schon beim Öffnen der Datenmenge vorbereitet, bedarf es überhaupt keiner Ausführung der betreffenden Anweisungen, die Datenmenge lässt sich gleich gar nicht öffnen.

Rufen wir nochmals den Dataset-Editor auf, entfernen *full_name* aus der Liste der zu aktualisierenden Spalten und generieren die SQL-Anweisung neu. Erfreulicherweise bleibt diese Spalte trotzdem im Select-Statement. Mit diesem Select-Statement kann *TIBDataSet* einen geänderten Datensatz einzeln vom Server holen. Auf diese Weise werden vom Server durchgeführte Änderungen sofort angezeigt, ohne gleich die komplette Datenmenge neu abrufen zu müssen.

Nun lässt sich die Datenmenge wieder öffnen und anhand des Navigators sehen wir, dass die Sache inzwischen auch aktualisierbar ist.

10.1.2 Transaktionen

InterBase erlaubt eine sehr dezidierte Steuerung des Transaktionsverhaltens. Diese Möglichkeit kann jedoch nur mit auf InterBase zugeschnittenen Komponenten (oder im API-Direktzugriff) genutzt werden, so dass wir dieses Thema erst hier besprechen.

Was sind Transaktionen?

Transaktionen sind eine Vorgehensweise, den Mehrbenutzerbetrieb einer Datenbank zu organisieren.

Es ist sicher nicht schwer vorstellbar, dass es zu gewissen Konflikten kommen kann, wenn mehrere Benutzer an den gleichen Daten Änderungen vornehmen:

- Wenn statistische Auswertungen durchgeführt werden oder wenn ein Backup einer Datenbank gefahren wird, dann benötigt man eine konsistente Datensicht, die Daten dürfen sich während eines solchen Vorgangs nicht ändern.
- Wenn zwei Benutzer »gleichzeitig« denselben Datensatz ändern, dann muss irgendwie sichergestellt werden, dass dabei kein großer Unsinn herauskommt.

Zudem gibt es Situationen, in denen eine Gruppe von Datenänderungen entweder alle gemeinsam oder überhaupt nicht durchgeführt werden sollen. Auch dafür werden Transaktionen verwendet.

Eine Transaktion ist die Zusammenfassung von Lese- und Schreibvorgängen. Sie wird mit START TRANSACTION begonnen und mit COMMIT oder ROLLBACK beendet. Wird sie mit COMMIT beendet, dann werden alle Schreibvorgänge (INSERT, UPDATE, DELETE) bestätigt, mit ROLLBACK werden alle Schreibvorgänge verworfen.

Versioning und Locking

Um die Datensicht konsistent zu halten, gibt es zwei Strategien, das Versioning und das Locking. Beim Locking werden alle Datensätze (manchmal auch Datenseiten oder Tabellen) gesperrt, welche gelesen wurden, sie können während der Laufzeit der betreffenden Transaktion nicht mehr geändert werden. Das Locking hat einen erheblichen Nachteil: Wenn der gesamte Datenbestand gelesen wird, dann ist während der Laufzeit dieser Transaktion keine Datenänderung mehr möglich. Ein Backup im laufenden Betrieb ist damit so gut wie unmöglich.

Im Gegensatz dazu legt das Versioning bei jeder Datenänderung einen so genannten Delta-Record an, in dem die alten Daten gespeichert werden. Dieser Delta-Record bleibt so lange bestehen, bis er von keiner Transaktion mehr benötigt wird. (Teilweise bleibt er noch ein wenig länger bestehen, insbesondere dann, wenn sehr viel über Indizes gelesen wird, weil das Wegräumen veralteter Versionen im Zuge sequenzieller Lesezugriffe erfolgt.)

InterBase unterstützt sowohl das Versioning als auch das Locking. Solange keine triftigen Gründe dagegen sprechen, sollten Sie beim Versioning bleiben.

Transaktionslaufzeit

Sowohl beim Versioning als auch beim Locking ist es essenziell, dass Transaktionen möglichst kurz laufen. Nehmen wir einmal an, der Geschäftsführer ruft ein Programm auf, mit dem die aktuellen Geschäftszahlen ermittelt werden, und zu diesem Zweck wird die halbe Firmen-Datenbank eingelesen. Nun wird er zu einer Sitzung gerufen, das Programm wird nicht beendet und die dazugehörende Transaktion läuft ein paar Stunden. Wird hier Locking eingesetzt, dann können solange keine Änderungen an den entsprechenden Tabellen vorgenommen werden, somit

wäre die Firmen-EDV so gut wie lahm gelegt. Bei Versioning muss der Server »nur« immer mehr veraltete Datenänderungen vorhalten, was die Sache zwar nicht gerade beschleunigt, aber immerhin kann man überhaupt noch arbeiten.

Beim Versioning gibt es jedoch zwei Varianten: SNAPSHOT und READ COMMITED – den genauen Unterschied werden wir uns noch ansehen. Im Moment nur so viel: Für SNAPSHOT müssen alte Versionen vorgehalten werden, für READ COMMITED nicht. Deshalb sollte man SNAPSHOT nur dort einsetzen, wo es erforderlich ist, nämlich bei allen statistischen Abfragen, und hier sollten die Transaktionen auch sehr kurz gehalten werden, zur Not dadurch, dass man alle Daten zum Client überträgt, dort lokal speichert und dann gleich die Transaktion beendet.

Braucht man bei schreibenden Transaktionen SNAPSHOT, dann ist es nicht verkehrt, erst beim Mausklick auf den Ok-Button die Transaktion zu starten, die Änderungen wegzuschreiben und dann sofort die Transaktion zu beenden.

Bei den Transaktionslaufzeiten gilt jedoch auch »Du sollst es nicht übertreiben«. Nehmen wir an, wie haben einen Batch-Import von tausend Adressen aus einer Datei. Dann beginnt man nicht für jeden Datensatz eine neue Transaktion, sondern fasst alle Änderungen in einer einzigen Transaktion zusammen.

Die Komponente TIBTransaction

Für die Transaktionssteuerung gibt es bei IBX die Komponente *TIBTransaction*. Diese implementiert nicht nur die Methoden *StartTransaction*, *Commit* und *Rollback*, sondern auch *CommitRetaining* und *RollbackRetaining*. Diese bestätigen oder verwerfen die Änderungen, ohne die Transaktion zu beenden. Wozu soll das gut sein?

Alle Lese- und Schreibvorgänge finden im Kontext einer Transaktion statt. Wird die Transaktion beendet, dann können nicht nur keine Datenänderungen mehr durchgeführt werden, sondern eine Abfrage verliert auch ihre Gültigkeit und somit wird die dazugehörende Datenmengenkomponente geschlossen. Braucht man die Daten weiterhin, dann muss man eine neue Transaktion starten, die Abfrage wiederholen und die Daten erneut zum Client übertragen – all das beschäftigt den Server.

Wenn man jetzt zwar Datenänderungen bestätigen möchte, aber nicht die Transaktion beenden, dann kann man *CommitRetaining* aufrufen. Dabei bleibt die Transaktion und somit die Datenmengenkomponente geöffnet, aber die Datenänderungen werden bestätigt. *RollbackRetaining* verhält sich analog dazu. Der Forderung nach möglichst kurzen Transaktionslaufzeiten kann man mit den *Retaining*-Methoden jedoch nicht nachkommen.

Um einzustellen, ob man nun *Versioning* oder *Locking* verwendet, ob (bei *Versioning*) READ COMMITED oder SNAPSHOT gewählt ist, ob die Transaktion nur lesen oder auch schreiben darf, kann man mit der *TIBTransaction*-Eigenschaft *Params* sehr dezidiert einstellen. Die dabei möglichen Optionen finden Sie im Refe-

renzteil dieses Kapitels. Für Einsteiger und Bequeme gibt es jedoch einen Komponenten-Editor, der die vier gebräuchlichsten Varianten zur Auswahl bereitstellt.

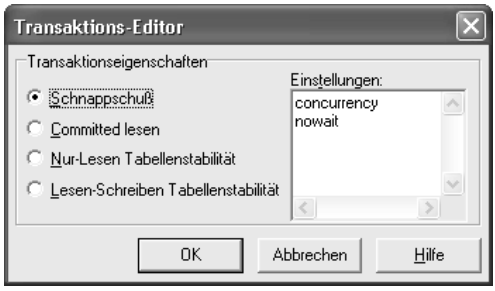


Abbildung 10.7: Der Transaktionseditor

10.1.3 Die Transaktionsspielwiese

Die »Transaktionsspielwiese« ist ein kleines Programm, mit dessen Hilfe wir das Verhalten bei den verschiedenen Transaktionsparametern austesten können.

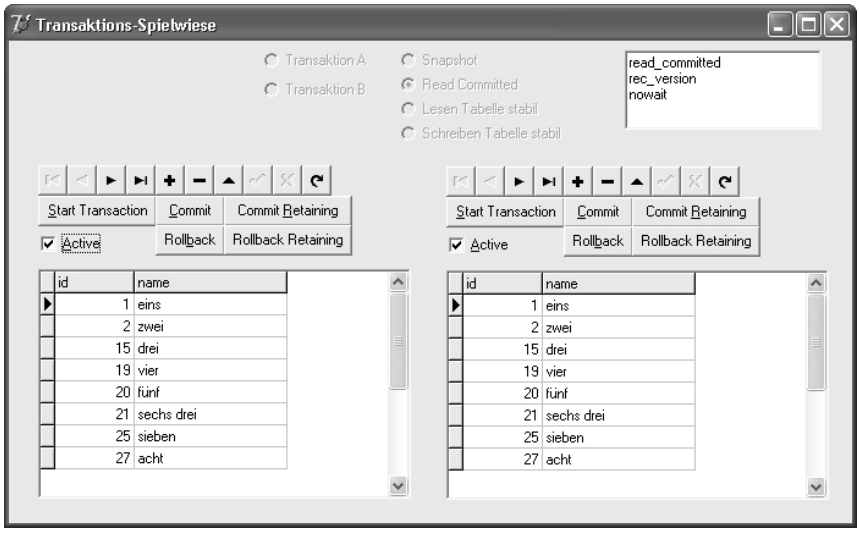


Abbildung 10.8: Die Transaktionsspielwiese

Die beiden Grids laufen über verschiedene Datenmengenkomponenten. Das linke Grid läuft immer über Transaktion A, das rechte – je nach Radiobutton – über Transaktion A oder Transaktion B. Von Transaktion B können die Parameter eingestellt werden, während Transaktion A gemäß der Voreinstellung der Komponente immer Snapshot (*concurrency notwait*) ist.

Lassen Sie uns ein wenig spielen ...

Abschottung der Transaktionen

Wenn wir die CheckBox *Active* markieren, dann wird die entsprechende Datenmenge geöffnet, bei Bedarf wird dabei automatisch eine Transaktion gestartet.

Wir wählen den Radiobutton *Transaktion A*, damit beide Datenmengenkomponenten dieselbe Transaktion nutzen, und öffnen beide Datenmengen. Dann führen wir in einem der beiden Grids eine beliebige Änderung durch (INSERT, UPDATE oder DELETE) und bestätigen Sie gegebenenfalls mit einem *Post*, also mit dem Haken-Button im Navigator. Die Änderung wird im anderen Grid noch nicht angezeigt.

Nun schließen wir die nicht aktualisierte Datenmenge und öffnen sie erneut, die Änderungen sind nun sichtbar. (Ein Mausklick auf den Refresh-Button ersetzt nicht das erneute Ausführen der Abfrage durch Schließen und erneutes Öffnen der Datenmenge).

Beenden wir nun beide Transaktionen – es ist egal, ob mit Commit oder Rollback, nur nicht mit den *Retaining*-Methoden. Dann wählen wir den Radiobutton *Transaktion B*, damit beide Datenmengenkomponenten verschiedene Transaktionen nutzen, öffnen wieder beide Datenmengen und führen erneut eine Datenänderung durch. Diesmal ist es nicht möglich, diese Datenänderung durch erneutes Ausführen der Abfrage sichtbar zu machen.

Datenmengen, die verschiedenen Transaktionen zugehören, verhalten sich isoliert: Von Änderungen im Kontext anderer Transaktionen erfahren sie zunächst einmal nichts.

Snapshot

Wird im Transaktionseditor die Option *Snapshot* gewählt, dann werden die Parameter *concurrency nowait* gesetzt. *Concurrency* steht für Versioning und somit im Gegensatz zu *Consistency*, was für *Locking* steht. Zu *nowait* kommen wir gleich noch.

Öffnen wir beide Datenmengen, führen links eine Änderung durch und beenden diese Transaktion mit *Commit*. Nun führen wir die Abfrage der rechten Datenmenge neu aus und stellen fest, dass hier die Änderungen noch nicht sichtbar sind. Erst dann, wenn wir die Transaktion beenden und eine neue Transaktion starten, sehen wir rechts die Änderungen.

Bei Snapshot-Transaktionen bleibt die Datensicht während der gesamten Laufzeit der Transaktion konstant.

Was hat es nun mit dem *nowait* auf sich? Starten wir auf beiden Seiten eine neue Transaktion und ändern links einen Datensatz, ohne die Transaktion zu beenden. Nun ändern wir auf der rechten Seite denselben Datensatz. Beim Versuch, *Post* aufzurufen, erhalten wir eine Exception: Der Server erkennt, dass dieser Datensatz schon von einer anderen Transaktion geändert wurde, und vermeidet über diese Exception einen Schreibkonflikt.

Nun ist aber gar nicht sicher, dass es überhaupt zu einem Schreibkonflikt gekommen wäre: Die Datenänderungen der linken Transaktion hätten ja auch mit Rollback zurückgenommen werden können. Hätte man statt *nowait* den Parameter *wait* gewählt, dann hätte der Server mit dem Ausführen der Methode *Post* so lange gewartet, bis die linke Transaktion beendet gewesen wäre, und hätte dann die Datenänderung erlaubt oder erst dann eine Exception ausgelöst.

Die Spekulation, welche Variante die bessere ist, ist hier überflüssig: Das Programm hängt in der Abarbeitung der Methode *Post*, somit kann weder *Commit* noch *Rollback* aufgerufen werden und nach ein paar Sekunden sieht das Programm ein, dass das so nichts wird, und löst eine Deadlock-Exception aus.

Da die meisten schreibenden Transaktionen mit *Commit* abgeschlossen werden, dürfte *nowait* ohnehin die sinnvollere Alternative sein.

Read Commited

Im Transaktionseditor ist *Read Commited* die Zusammenfassung der Parameter *read_committed rec_version nowait*.

Öffnen wir beide Datenmengen, führen links eine Änderung durch und beenden diese Transaktion mit *Commit*. Nun führen wir die Abfrage der rechten Datenmenge neu aus und stellen fest, dass die Änderungen auch hier schon sichtbar sind.

Bei *Read Commited*-Transaktionen kann sich die Datensicht während der Transaktion ändern.

So genannte *Dirty Read*-Transaktionen, also das Lesen von Änderungen, deren dazugehörende Transaktionen noch nicht mit Commit bestätigt sind, wird von InterBase nicht unterstützt. Damit wird vermieden, dass Änderungen sichtbar werden, die bereits mit Rollback verworfen wurden oder demnächst verworfen werden.

Der Parameter *read_committed* kann mit *rec_version* und *no_ rec_version* kombiniert werden. Bei *rec_version* werden noch nicht bestätigte Änderungen von anderen Transaktionen einfach ignoriert, bei *no_rec_version* wird bei nicht bestätigten Änderungen entweder gewartet (*wait*) oder eine Exception ausgelöst (*nowait*).

Tabellen-Stabilität

Die *Tabellen-Stabilität* gibt es in zwei Varianten: *read consistency* und *write consistency. Read* und *write* unterscheiden sich dadurch, dass *read*-Transaktionen keine Schreibvorgänge durchführen dürfen – ein entsprechender Versuch würde mit Exception quittiert. Der Parameter *read* kann auch mit anderen Transaktionsarten kombiniert werden. Solange nichts Entsprechendes spezifiziert wird, liegt automatisch eine *write*-Transaktion vor, es darf dann sowohl gelesen als auch geschrieben werden.

Mit *consistency* stellt man sicher, dass während der Transaktionslaufzeit alle Tabellen, die von der betreffenden Transaktion gelesen wurden, im Kontext anderer Transaktionen nicht geändert werden können.

Zusammenfassung

Merken wir uns:

- Datenmengen, die verschiedenen Transaktionen zugehören, verhalten sich isoliert: Von Änderungen im Kontext anderer Transaktionen erfahren sie zunächst einmal nichts.
- Bei Snapshot-Transaktionen bleibt die Datensicht während der gesamten Laufzeit der Transaktion konstant.
- Bei *Read Commited*-Transaktionen kann sich die Datensicht während der Transaktion ändern.
- So genannte *Dirty Read*-Transaktionen, also das Lesen von Änderungen, deren dazugehörende Transaktionen noch nicht mit Commit bestätigt sind, werden von InterBase nicht unterstützt.
- Mit *consistency* stellt man sicher, dass während der Transaktionslaufzeit alle Tabellen, die von der betreffenden Transaktion gelesen wurden, im Kontext anderer Transaktionen nicht geändert werden können.
- Da die meisten schreibenden Transaktionen mit *Commit* abgeschlossen werden, dürfte *nowait* die sinnvollere Alternative sein.

10.1.4 Bilder und Daten speichern

InterBase erlaubt es, sehr große Dateneinheiten als BLOB zu speichern. Die Erfahrung aus verschiedensten Internet-Foren zeigt, dass gerade viele Einsteiger hier immer wieder Schwierigkeiten haben. Wir wollen uns deshalb ein kleines Beispiel basteln, indem wir ein Bitmap sowie den Inhalt eines Records als Stream in die Datenbank speichern. Dabei verwenden wir alternativ Felder und Parameter.

Um ein Bild zu laden, schauen Sie sich das Kontextmenü von *DBImage1* an.

Neben dem Bild speichern wir den Inhalt des folgenden Records in der Datenbank:

```
type
  TTest = record
    eins: string[30];
    zwei: integer;
    drei: boolean;
  end;
```

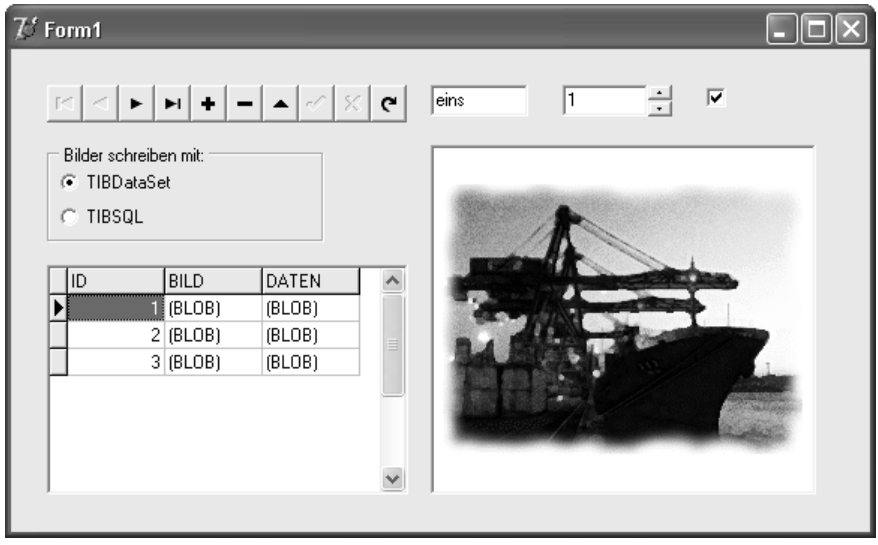


Abbildung 10.9: BLOBs speichern

Die Daten werden über ein Edit-Feld, eine UpDown-Komponente und eine CheckBox gesetzt. Wird der Zustand einer dieser Komponenten verändert, dann wird die Routine *ControlsChange* aufgerufen.

```
procedure TForm1.ControlsChange(Sender: TObject);
begin
  if FUpdate or not FUpdate2
    then exit;
  if IBDataSet1.State = dsBrowse
    then IBDataSet1.Edit;
  FTest.eins := Edit1.Text;
  FTest.zwei := UpDown1.Position;
  FTest.drei := CheckBox1.Checked;
  FStream.Position := 0;
  FStream.Write(FTest, SizeOf(FTest));
  IBDataSet1DATEN.LoadFromStream(FStream);
end;
```

Um Endlosschleifen zu vermeiden, gibt es die beiden boolschen Felder *FUpdate* und *FUpdate2*. Ist die Datenmenge noch nicht im Edit- oder Insert-Modus, dann wird *Edit* aufgerufen. Nun wird der Inhalt des Records *FTest* gesetzt. *FStream* ist ein Feld vom Typ *TMemoryStream*, der Inhalt wird mit der *TBlobField*-Methode *LoadFromStream* in das Datenfeld geschrieben.

```
procedure TForm1.IBDataSet1AfterScroll(DataSet: TDataSet);
var
  Stream: TMemoryStream;
```

```
begin
  FUpdate := true;
  try
    Stream := TMemoryStream.Create;
    IBDataSet1DATEN.SaveToStream(Stream);
    Stream.Position := 0;
    Stream.Read(FTest, SizeOf(FTest));
    Edit1.Text := FTest.eins;
    UpDown1.Position := FTest.zwei;
    CheckBox1.Checked := FTest.drei;
  finally
    FUpdate := false;
    Stream.Free;
  end;
end; {procedure TForm1.IBDataSet1AfterScroll}
```

Während das Bild von *TDBImage* automatisch angezeigt wird, muss der Record eigens ausgelesen werden. Dazu wird der Inhalt des Datenfeldes über einen Stream in den Record geschrieben, anschließend werden die Controls aktualisiert.

```
procedure TForm1.Bilddateiffnen1Click(Sender: TObject);
begin
  if OpenPicDlg1.Execute then
  begin
    if RadioGroup1.ItemIndex = 0 then
    begin
      if IBDataSet1.State = dsBrowse
        then IBDataSet1.Edit;
      IBDataSet1BILD.LoadFromFile(OpenPicDlg1.FileName);
    end
    else
    begin
      with IBSQL1 do
      begin
        ParamByName('old_id').AsInteger := IBDataSet1ID.AsInteger;
        ParamByName('Bild').LoadFromFile(OpenPicDlg1.FileName);
        ParamByName('Daten').LoadFromStream(FStream);
        ExecQuery;
      end; {with IBSQL1 do}
      IBDataSet1.Close;
      IBDataSet1.Open;
    end; {else RadioGroup1.ItemIndex = 0 then}
  end; {if OpenPictureDialog1.Execute then}
end; {procedure TForm1.Bilddateiffnen1Click}
```

Um ein Bild aus einer Datei in die Datenbank zu schreiben, verwenden wir einmal *TIBDataSet*, zum anderen *TIBSQL*. Die Anweisung von *IBSQL1* entnehmen wir der Eigenschaft *ModifySQL* von *IBDataSet1*.

```
update T_BLOBTEST2
set
  BILD = :BILD,
  DATEN = :DATEN
where
  ID = :OLD_ID
```

Zunächst das Einfügen über *IBDataSet1*: Wenn die Datenmenge noch im Browse-Modus ist, wird *Edit* aufgerufen. Anschließend wird das Bild mit der *TBlobField*-Methode *LoadFromFile* geladen. *Post* müsste dann manuell aufgerufen werden.

Soll *TIBSQL* verwendet werden, dann werden die Felder – und zwar alle – über Parameter gesetzt, anschließend wird *ExecQuery* aufgerufen. Damit die Änderungen auch sichtbar werden, ist die Abfrage von *IBDataSet1* neu auszuführen.

```
procedure TForm1.Einfgen1Click(Sender: TObject);
var
  MyBitmap: TBitmap;

  procedure DoIBSQL;
  var
    Stream: TMemoryStream;
  begin
    Stream := TMemoryStream.Create;
    try
      with IBSQL1 do
      begin
        ParamByName('old_id').AsInteger := IBDataSet1ID.AsInteger;
        MyBitmap.SaveToStream(Stream);
        ParamByName('Bild').LoadFromStream(Stream);
        ParamByName('Daten').LoadFromStream(FStream);
        ExecQuery;
      end; {with IBSQL1 do}
      IBDataSet1.Close;
      IBDataSet1.Open;
    finally
      Stream.Free;
    end;
  end;

begin
  if Clipboard.HasFormat(CF_BITMAP) then
  begin
```

```
    MyBitmap := TBitmap.Create;
    try
      MyBitmap.Assign(Clipboard);
      if RadioGroup1.ItemIndex = 0 then
      begin
        if IBDataSet1.State = dsBrowse
          then IBDataSet1.Edit;
        IBDataSet1BILD.Assign(MyBitmap);
      end
      else DoIBSQL;
    finally
      MyBitmap.Free;
    end;
  end; {if Clipboard.HasFormat(CF_BITMAP) then}
end; {procedure TForm1.Einfgen1Click}
```

Alternativ dazu lässt sich das Bild auch über die Zwischenablage einfügen. Dazu ist es erforderlich, den Inhalt der Zwischenablage zunächst in ein Bitmap zu laden, das zu diesem Zweck erstellt wird. Dieses Bitmap lässt sich der *TBlobField*-Instanz *IBDataSet1BILD* direkt zuweisen.

Wird dasselbe über Parameter versucht, dann muss das Bitmap zunächst in einen Stream geladen werden, der dann dem Parameter zugewiesen wird.

10.2 Referenz der IBX-Komponenten

In Kapitel 10.2 sollen die Komponenten der Palettenseite *InterBase* besprochen werden, in Kapitel 10.3 die der Palettenseite *InterBase Admin*.

10.2.1 TIBDatabase

TIBDatabase stellt die Verbindung zum InterBase-Client her. *TIBDatabase* ähnelt in vielem *TDatabase*.

Verbindung zur Datenbank

Die Verbindung zur Datenbank stellt man für gewöhnlich über den Komponenten-Editor (Doppelklick auf die Komponente) her, wir haben ihn bereits in Kapitel 10.1.1 beschrieben. Möchte man die Verbindung zur Laufzeit spezifizieren, muss man auf die folgenden Eigenschaften zurückgreifen:

- DatabaseName (Eigenschaft, veröffentlicht)

  ```
  property DatabaseName: String;
  ```

  Mittels *DatabaseName* wird die Datenbank angegeben, mit der die Komponente verbunden werden soll.

- Connected (Eigenschaft, veröffentlicht), Open, Close (Methoden)

```
property Connected: Boolean;
procedure Open;
procedure Close;
```

Um die Verbindung zur Datenbank herzustellen, setzen Sie *Connected* auf *true* oder rufen *Open* auf. Um die Verbindung zu beenden, wird *Close* aufgerufen oder *Connected* auf *false* gesetzt.

- AllowStreamed (Eigenschaft, veröffentlicht)

```
property AllowStreamedConnected: Boolean;
```

Hat man zur Entwurfszeit die Eigenschaft *Connected* auf *true* gesetzt, dann wird die Verbindung beim Starten des Programms automatisch geöffnet. Dies mag nicht immer erwünscht sein, beispielsweise dann nicht, wenn man den Dateinamen der Datenbankdatei erst aus einer Ini-Datei lesen möchte oder wenn man die Zeitdauer des Herstellens der Datenbankverbindung messen möchte.

Hier kann man nun *AllowStreamedConnected* auf *false* setzen, so dass die Verbindung auf jeden Fall explizit aufgebaut werden muss.

- Params, LoginPrompt (Eigenschaften, veröffentlicht)

```
property Params: TStrings;
property LoginPrompt: Boolean;
```

Über die Eigenschaft *Params* können Parameter wie der Benutzername oder das Passwort eingestellt werden. Dies ist besonders dann wichtig, wenn kein Anmelde-Dialog angezeigt werden soll und somit die Eigenschaft *LoginPrompt* auf *false* gesetzt wird.

Beachten Sie bitte die im Vergleich zu *TDatabase* leicht abweichende Schreibweise des Benutzernamens:

```
user_name=SYSDBA
password=masterkey
lc_ctype=ISO8859_1
```

- Diverse Ereignisse

```
property OnAfterConnect(Sender: TObject);
property OnAfterDisconnect(Sender: TObject);
property OnBeforeConnect(Sender: TObject);
property OnBeforeDisconnect(Sender: TObject);
```

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

- IdleTimer (Eigenschaft, veröffentlicht), OnIdleTimer (Ereignis)

```
property IdleTimer: Integer;
property OnIdleTimer(Sender: TObject);
```

Mit *IdleTimer* wird die Zeit in Sekunden eingestellt, die auf einen Verbindungsaufbau gewartet wird. Verstreicht diese Zeit, ohne dass eine Verbindung zustande kommt, dann tritt das Ereignis *OnIdleTimer* auf.

- IsReadOnly (Eigenschaft, öffentlich, nur Lesen)

```
property IsReadOnly: Boolean;
```

Ist die Datenbank schreibgeschützt, dann hat *IsReadOnly* den Wert *true*.

- SQLDialect (Eigenschaft, veröffentlicht), OnDialectDowngradeWarning (Ereignis)

```
property SQLDialect: Integer;
property OnDialectDowngradeWarning: TNotifyEvent;
```

InterBase unterstützt zwei SQL-Dialekte, nämlich 1 und 3. Diese unterscheiden sich vor allem darin, dass bei Dialekt 1 einfache und doppelte Anführungszeichen synonym verwendet werden, während bei Dialekt 3 einfache Anführungszeichen für String-Konstanten und doppelte Anführungszeichen für Bezeichner verwendet werden.

Mit der Eigenschaft *SQLDialect* kann spezifiziert werden, welcher Dialekt verwendet werden soll. Diese Eigenschaft kann bei geöffneter Verbindung geändert werden.

Unterstützt die Datenbank nur den Dialekt 1, dann wird *SQLDialect* automatisch auf diesen Wert gesetzt, wenn die Eigenschaft vorher den Wert drei hatte – dabei wird dann das Ereignis *OnDialectDowngradeWarning* ausgelöst.

- CreateDatabase, DropDatabase (Methoden)

```
procedure CreateDatabase;
procedure DropDatabase;
```

Um eine Datenbank zu erstellen, setzen Sie die Eigenschaft *DatabaseName*, als Parameter mindestens den Benutzernamen, das Passwort und die Seitengröße und rufen dann *CreateDatabase* auf.

Mit *DropDatabase* kann die Datenbank gelöscht werden.

Transaktionen

Im Gegensatz zu *TDatabase* erfolgt mit *TIBDatabase* keine Transaktionssteuerung – dafür ist die Komponente *TIBTransaction* zuständig. Mit einer *TIBDatabase* können mehrere *TIBTransaction*-Komponenten verbunden werden.

- DefaultTransaction (Eigenschaft, veröffentlicht)

```
property DefaultTransaction: TIBTransaction;
```

Wird bei einer IB-Datenmengenkomponente die Eigenschaft *Database* gesetzt, wird die Eigenschaft *Transaction* automatisch auf die mit *DefaultTransaction* eingestellte *TIBTransaction*-Instanz gesetzt.

- Transactions, TransactionCount (Eigenschaften, öffentlich, nur Lesen)

```
property Transactions [Index: Integer]: TIBTransaction;
property TransactionCount: Integer;
```

Die Array-Eigenschaft *Transactions* referenziert alle mit der *TIBDatabase*-Komponente verbundenen Transaktionskomponenten. Mit *TransactionCount* kann deren Zahl festgestellt werden.

- AddTransaction, RemoveTransaction, RemoveTransactions, FindTransaction (Methoden)

```
function AddTransaction(TR: TIBTransaction): Integer;
procedure RemoveTransaction(Idx: Integer);
procedure RemoveTransactions;
function FindTransaction (TR: TIBTransaction): Integer;
```

Mit der Methode *AddTransaction* wird der Datenbankkomponente eine neue Transaktionskomponente hinzugefügt. Als Rückgabewert liefert diese Funktion den Index der neuen Transaktionskomponente in *Transactions*.

Um eine Transaktionskomponente aus der Liste zu entfernen, wird *RemoveTransaction* verwendet – um alle Transaktionskomponenten zu entfernen, *RemoveTransactions*. Um den Index einer Transaktionskomponente zu ermitteln, wird *FindTransaction* verwendet.

Datenmengen

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

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

Die Array-Eigenschaft *DataSets* referenziert alle Datenmengen-Komponenten, welche diese Datenbankverbindung nutzen. Ihre Gesamtzahl kann mit *DataSetCount* ermittelt werden.

Der folgende Quelltext zeigt, wie die Methode *CloseDataSets* diese beiden Eigenschaften in einer Schleife nutzt.

```
procedure TIBDatabase.CloseDataSets;
var
  i: Integer;
begin
  for i := 0 to DataSetCount - 1 do
    if (DataSets[i] <> nil) then
      DataSets[i].close;
end;
```

- CloseDataSets (Methode)

```
procedure CloseDataSets;
```

Schließt alle verbundenen Datenmengen.

Informationen über die Datenbank

- GetTableNames, GetFieldNames (Methoden)

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

GetTableNames ermittelt die Namen aller Tabellen in der Datenbank. Im Gegensatz zu *TDatabase* fehlen hier die Namen der VIEWS. Wird *SystemTables* auf *true* gesetzt, dann werden auch die Systemtabellen mit angegeben.

```
IBDatabase1.GetTableNames(Form1.ListBox1.Items);
```

Mit *GetFieldNames* werden die Spaltennamen der angegebenen Tabelle ermittelt.

- TraceFlags (Eigenschaft)

```
property TraceFlags: TTraceFlags;
```

Mittels *TIBSQLMonitor* können die SQL-Anweisungen protokolliert werden, die zum Server geschickt werden. Über die Eigenschaft *TraceFlags* wird dabei spezifiziert, welche Anweisungen dabei beachtet werden sollen.

Die Bedeutung der einzelnen Flags entnehmen Sie bitte der Online-Hilfe.

- Has_COMPUTED_BLR, Has_DEFAULT_VALUE (Methoden)

```
function Has_COMPUTED_BLR(Relation, Field : String) : Boolean;
function Has_DEFAULT_VALUE(Relation, Field : String) : Boolean;
```

Mit diesen beiden Funktionen kann ermittelt werden, ob eine bestimmte Spalte eine berechnete Spalte ist und ob sie einen Default-Wert hat.

10.2.2 TIBTransaction

Die Komponente *TIBTransaction* implementiert die Transaktionsverwaltung. Für den Zugriff auf eine InterBase-Datenbank benötigen Sie mindestens eine *TIBDatabase*- und eine *TIBTransaction*-Komponente.

Transaktionsverwaltung

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

```
procedure StartTransaction;
property Params: TStrings;
```

Mit *StartTransaction* wird eine Transaktion neu gestartet. Dabei spezifizieren die mit *Params* definierten Parameter die Transaktionsart.

Es sind folgende Parameter definiert, von denen wir die wichtigsten besprechen wollen:

```
TPBConstantNames: array[1..isc_tpb_last_tpb_constant] of string =
    ('consistency',
    'concurrency',
    'shared',
    'protected',
    'exclusive',
    'wait',
    'nowait',
    'read',
    'write',
    'lock_read',
    'lock_write',
    'verb_time',
    'commit_time',
    'ignore_limbo',
    'read_committed',
    'autocommit',
    'rec_version',
    'no_rec_version',
    'restart_requests',
    'no_auto_undo');
```

Per Voreinstellung sind Transaktionen *concurrency*, es können also mehrere Transaktionen gleichzeitig mit denselben Daten arbeiten. Wenn Sie *consistency* verwenden, dann werden verwendete Tabellen für andere Transaktionen gesperrt.

Normalerweise bleibt die Datensicht während der Laufzeit einer Transaktion konstant, es sei denn, Sie verwenden *read_committed* – dann können Sie auch Änderungen sehen, die von anderen Transaktionen bestätigt wurden. Der Parameter *read_committed* kann mit *rec_version* und *no_ rec_version* kombiniert werden. Bei *rec_version* werden noch nicht bestätigte Änderungen von anderen Transaktionen einfach ignoriert, bei *no_rec_version* wird bei nicht bestätigten Änderungen entweder gewartet (*wait*) oder eine Exception ausgelöst (*nowait*).

Neu gestartete Transaktionen können sowohl Daten lesen als auch solche schreiben (dies würde *write* entsprechen), es sei denn, sie werden mit *read* gestartet – dann können sie nur Daten lesen.

Per Voreinstellung können Tabellen von mehreren Transaktionen gemeinsam genutzt werden (*shared*), Sie können aber Tabellen auch vom gemeinsamen Zugriff sperren (*protected*) und müssen in diesem Zusammenhang dann Lese- und Schreibschutz definieren (*lock_read* und *lock_write*, die Details finden Sie im API-Guide).

- Commit, CommitRetaining, Rollback, RollbackRetaining (Methoden)

```
procedure Commit;
procedure CommitRetaining;
procedure Rollback;
procedure RollbackRetaining;
```

Mit *Commit* werden alle Änderungen bestätigt und die Transaktion wird abgeschlossen, während *Rollback* alle Änderungen verwirft, bevor die Transaktion abgeschlossen wird.

Mit *CommitRetaining* werden Änderungen bestätigt, die Transaktion wird aber nicht abgeschlossen. Ebenso werden mit *RollbackRetaining* Änderungen verworfen, ohne die Transaktion zu beenden.

Da der Datenbankserver eine SQL-Anweisung nur im Rahmen einer Transaktion ausführt, werden auch alle zu einer Transaktion gehörenden Datenmengen geschlossen, wenn die Transaktion beendet wird. Soll das Schließen der Datenmengen vermieden werden, dann sind *CommitRetaining* beziehungsweise *RollbackRetaining* zu verwenden.

Rufen Sie die Methoden zum Beenden einer Transaktion nur auf, solange eine Transaktion aktiv ist, sonst lösen Sie eine Exception aus. Sie können mit der Eigenschaft *InTransaction* prüfen, ob gerade eine Transaktion aktiv ist.

```
if IBTransaction1.InTransaction
  then IBTransaction1.CommitRetaining;
```

- InTransaction (Eigenschaft, öffentlich, nur Lesen)

```
property InTransaction: Boolean;
```

Mit *InTransaction* kann geprüft werden, ob momentan eine Transaktion gestartet ist.

- Active (Eigenschaft, veröffentlicht)

```
property Active: Boolean;
```

Mit *Active* kann zur Entwurfszeit eine Transaktion begonnen und somit mit Live-Daten gearbeitet werden. Wird *Active* auf *false* gesetzt, dann wird *Rollback* aufgerufen.

- DefaultAction (Eigenschaft, veröffentlicht)

```
property DefaultAction: (taRollback, taCommit, taRollbackRetaining,
taCommitRetaining);
```

Mit *DefaultAction* wird spezifiziert, wie die Transaktion beendet werden soll, wenn sie von der Komponente automatisch beendet werden muss, beispielsweise, weil sie zu lange inaktiv ist oder weil die Datenbankverbindung getrennt wird. Per Voreinstellung wird die Transaktion mit *Commit* beendet.

Wird die Transaktion aufgrund eines Sachverhalts geschlossen, der es nicht erlaubt, den Transaktionskontext fortzuführen – beispielsweise wenn die Datenbankverbindung getrennt wird –, dann wird auch bei *taCommitRetaining* die Transaktion mit *Commit* abgeschlossen und auch bei *taRollbackRetaining* mit *Rollback*.

- IdleTimer (Eigenschaft, veröffentlicht), OnIdleTimer (Ereignis)

```
property IdleTimer: Integer;
property OnIdleTimer(Sender: TObject);
```

Mit *IdleTimer* wird die Zeit in Sekunden eingestellt, die eine Transaktion inaktiv sein darf, bevor sie automatisch zurückgesetzt wird. Wird diese Zeit überschritten, dann wird *DefaultAction* ausgeführt, anschließend tritt das Ereignis *OnIdleTimer* auf.

Solange *IdleTimer* den Vorgabewert null hat, wird die Transaktion nie zurückgesetzt.

Datenbanken

- DefaultDatabase (Eigenschaft, veröffentlicht)

```
property DefaultDatabase: TIBDatabase;
```

Mit *DefaultDatabase* wird die »hauptsächlich eingesetzte« (in den meisten Fällen ausschließlich verwendete) Datenbank angegeben.

- Databases, DatabaseCount (Eigenschaften, nur Lesen)

```
property Databases[Index: Integer]: TIBDatabase;
property DatabaseCount: Integer;
```

Die mit der Transaktionskomponente verbundenen Datenbankkomponenten können mit der Array-Eigenschaft *Databases* referenziert werden. Die Anzahl dieser Datenbankkomponenten kann mit *DatabaseCount* ermittelt werden.

Eine Transaktion kann durchaus mehrere Datenbanken umfassen – InterBase verwendet dann automatisch ein *two-phase-Commit*.

- AddDatabase, RemoveDatabase, RemoveDatabases, FindDatabase (Methoden)

```
function AddDatabase(db: TIBDatabase): Integer;
procedure RemoveDatabase(Idx: Integer);
procedure RemoveDatabases;
function FindDatabase (db: TIBDatabase): Integer;
```

Um der Liste der Datenbanken eine neue hinzuzufügen, wird *AddDatabase* verwendet, um eine aus der Liste zu entfernen, *RemoveDatabase*. *RemoveDatabases* entfernt alle Datenbanken aus der Liste und *FindDatabase* ermittelt den Index einer Datenbankkomponente.

10.2.3 TIBCustomDataSet

Die Komponente *TIBCustomDataSet* ist der gemeinsame Vorfahre von *TIBTable*, *TIBQuery*, *TIBStoredProc* und *TIBDataSet*.

In *TIBCustomDataSet* werden einige Eigenschaften implementiert, die ausschließlich bei *TIBDataSet* veröffentlicht werden – sie sollen auch erst dort besprochen werden.

Status der Datenmenge

- Database, Transaction (Eigenschaften, veröffentlicht)

```
property Database: TIBDatabase;
property Transaction: TIBTransaction;
```

Mit *Database* wird die *TIBDatabase*-Komponente ausgewählt, über welche die Verbindung zur Datenbank hergestellt wird.

Jede Datenmengenkomponente muss zwingend mit einer Transaktionskomponente verbunden werden, dafür gibt es die Eigenschaft *Transaction*. Wird die Eigenschaft *Database* gesetzt und hat die betreffende *TIBDatabase*-Instanz eine *DefaultTransaction*, dann wird diese automatisch nach *Transaction* übernommen.

- Open, Close (Methoden, TDataSet), Active (Ereignis, veröffentlicht)

```
procedure Open;
procedure Close;
property Active: boolean;
```

Mit *Open* wird die Datenmenge geöffnet, mit *Close* wird sie geschlossen. Soll die Datenmenge zur Laufzeit geöffnet und geschlossen werden, dann wird die Eigenschaft *Active* verwendet.

- 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.

- BeforeDatabaseDisconnect, AfterDatabaseDisconnect, DatabaseFree, BeforeTransactionEnd, AfterTransactionEnd, TransactionFree (Ereignisse)

```
property BeforeDatabaseDisconnect(Sender: TObject);
property AfterDatabaseDisconnect(Sender: TObject);
property DatabaseFree(Sender: TObject);
property BeforeTransactionEnd(Sender: TObject);
property AfterTransactionEnd(Sender: TObject);
property TransactionFree(Sender: TObject);
```

Alle IBX-Datenmengenkomponenten arbeiten nur in Verbindung zu einer *TIBDatabase*-Komponente und im Kontext einer Transaktion, also zusammen mit einer *TIBTRansaction*-Instanz. Wenn die Datenbankverbindung geschlossen oder die Transaktion beendet wird, dann werden die entsprechenden Ereignisse aufgerufen, ebenso wenn die entsprechenden Komponenten freigegeben werden.

Wenn die Eigenschaften *Database* beziehungsweise *Transaction* auf *nil* gesetzt werden, löst dies nicht die Ereignisse *DatabaseFree* beziehungsweise *TransactionFree* aus.

- 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.
- *dsEdit*: Der aktuelle Datensatz kann geändert werden. Diesen Status erhält man mit dem direkten oder indirekten Aufruf von *Edit*.
- *dsInsert*: Nach dem direkten oder indirekten Aufruf von *Insert* oder *Append* kann ein neuer Datensatz ein- oder angefügt werden.
- *dsSetKey*: Nur bei *TTable*. Die Datensatzsuche ist aktiviert oder eine *SetRange*-Operation wird durchgeführt. Es können keine Datensätze geändert oder eingefügt werden.
- *dsCalcFields*: Ein *OnCalcFields*-Ereignis wurde ausgelöst, es gibt somit Felder, deren Inhalt nicht verändert werden kann.
- *dsFilter*: Es wird eine gefilterte Datenmenge angezeigt. Im Gegensatz zu den Angaben in der Online-Hilfe können Änderungen vorgenommen und neue Datensätze eingefügt werden, es kann jedoch sein, dass diese nach der *Post*-Anweisung nicht mehr angezeigt werden.
- Im Zusammenhang mit *CachedUpdates* sind folgende Stati möglich:
- *dsNewValue*: Temporäre, interne Statuszuweisung, die anzeigt, dass gerade ein Zugriff auf die Eigenschaft *TField.NewValue* erfolgt.
- *dsOldValue*: Temporäre, interne Statuszuweisung, die anzeigt, dass gerade ein Zugriff auf die Eigenschaft *TField.OldValue* erfolgt.
- *dsCurValue*: Temporäre, interne Statuszuweisung, die anzeigt, dass gerade ein Zugriff auf die Eigenschaft *TField.CurValue* erfolgt.

- 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.

- Plan (Eigenschaft, öffentlich, nur Lesen)

```
property Plan: string;
```

Mit *Plan* kann der Query-Plan einer Abfrage ermittelt werden. Da die einzelnen SQL-Anweisungen direkt beim Öffnen der Datenmenge vorbereitet werden, steht ab dann auch der Query-Plan zur Verfügung.

Navigieren in der Datenmenge

- First, Prior, Next, Last (Methoden)

```
procedure First;
procedure Prior;
procedure Next;
procedure Last;
```

Mit Hilfe dieser Methoden kann der Datenzeiger zum ersten (*First*), vorhergehenden (*Prior*), nächsten (*Next*) oder letzten (*Last*) Datensatz verschoben werden.

Für gewöhnlich rufen die InterBase-Komponenten nur so viele Datensätze vom Server ab, wie angezeigt werden sollen. Deshalb scheint auch die Anzeige großer Datenmengen schnell zu gehen. Wenn Sie jedoch nun die Methode *Last* aufrufen, werden alle Datensätze vom Server zum Client übertragen, was besonders bei langsamen Netzwerken viel Zeit in Anspruch nehmen kann.

Den Aufruf von *Last* kann man oft vermeiden, wenn man stattdessen die Datenmenge schließt, die Sortierreihenfolge umdreht und dann die Datenmenge wieder öffnet – in den meisten Fällen geschieht das deutlich schneller als ein Aufruf von *Last*, insbesondere, wenn man für die nötige Sortierreihenfolge einen passenden (also absteigenden) Index definiert.

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

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

Die Eigenschaft *BOF* (*EOF*) ist dann gleich *true*, wenn

- die Methode *First* (*Last*) aufgerufen wird,
- das Verschieben des Datenzeigers fehlschlägt, weil er schon auf dem ersten (letzten) Datensatz steht.

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

```
while not Table1.EOF {BOF} do
begin
    Table1.Next {Prior};
end;
```

Vergessen Sie nicht, die Methode *Next* oder *Prior* aufzurufen (oder eine andere Methode, welche den Datenzeiger verschiebt), anderenfalls hätten Sie eine Endlosschleife. Das Gleiche würde passieren, wenn die Methode den Datenzeiger zum entgegengesetzten Ende der Datenmenge verschiebt.

- MoveBy (Methode)

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

Mit der Methode *MoveBy* kann man um die im Parameter angegebene Zahl von Datensätzen nach hinten navigieren.

Soll zum Dateianfang hin navigiert werden, dann muss ein negativer Parameter übergeben werden. Trifft diese Funktion auf Dateiende oder Dateianfang, dann wird an dieser Stelle gestoppt und die Eigenschaft *EOF* beziehungsweise *BOF* auf *true* gesetzt. Die Zahl der Datensätze, um die der Datenzeiger wirklich bewegt worden ist, wird als Funktionsergebnis zurückgegeben.

- FindFirst, FindPrior, FindNext, FindLast (Methoden)

```
function FindFirst: Boolean;
function FindPrior: Boolean;
function FindNext: Boolean;
function FindLast: Boolean;
```

Mit Hilfe dieser Methoden kann man in mit *OnFilterRecord* gefilterten Datenmengen navigieren, wenn bei jedem Aufruf einer dieser Methoden der gesamte Datenbestand neu durchgefiltert werden soll.

Greifen wir mit *TTable* auf die Tabelle *t_art* unserer Beispieldatenbank zu, die in der Spalte *nummer* die Werte von 1 bis 14 enthält. Diese Datenmenge wird mit der folgenden Ereignisbehandlungsroutine auf die Datensätze 11 und 12 beschränkt:

```
procedure TForm1.Table1FilterRecord(DataSet: TDataSet;
    var Accept: Boolean);
begin
```

```
    Accept := ((Table1NUMMER.AsInteger > 10)
        and (Table1NUMMER.AsInteger < 13));
end;
```

Mit Hilfe des Debuggers schauen wir uns den Inhalt der Spalte *nummer* an. Bei Verwendung der Methode *First* wird *OnFilterRecord* exakt 14-mal aufgerufen, für jeden Datensatz ein einziges Mal. Danach werden die Datensätze 11 und 12 angezeigt. Beim darauf folgenden Aufruf von *Next* wird *OnFilterRecord* nicht aufgerufen, bei noch einem Aufruf von *Next* wird *OnFilterRecord* für die Datensätze 13 oder 14 aufgerufen – es wird sicherheitshalber nochmals geschaut, ob inzwischen weitere Datensätze der Filterbedingung – die sich ja geändert haben kann – entsprechen.

Nun die ganze Geschichte mit *FindFirst*: Das Ereignis *OnFilterRecord* wird nacheinander in der folgenden Reihenfolge aufgerufen:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1,
12, 13, 14,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Um 14 Datensätze zu filtern, wird *OnFilterRecord* 35-mal aufgerufen. Wir stehen nun auf Datensatz 11 und rufen *FindNext* auf. Nun wird *OnFilterRecord* für die folgenden Datensätze aufgerufen:

```
12, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
13, 14,
10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Ein weiterer Aufruf von *FindNext* beschäftigt *OnFilterRecord* dann nur mit den Datensätzen 13 und 14.

Aus welchen Gründen mit diesen Methoden der Datenbestand manchmal mehrfach durch die *OnFilterRecord*-Ereignisbehandlungsroutine geschickt wird, ist mir zugegebenermaßen völlig rätselhaft.

- Locate (Methode)

```
function Locate(const KeyFields: string; const KeyValues: Variant;
  Options: TLocateOptions): Boolean;
```

Die Methode *Locate* wird verwendet, um einen Datensatz anhand seiner Feldwerte zu finden. Das folgende Beispiel setzt den Datenzeiger auf den ersten Datensatz, dessen Feld *Vorname* den Wert *Bert* und dessen *Nachname* den Wert *Heller* hat.

```
Table1.Locate('Vorname; Nachname', VarArrayOf(['Bert', 'Heller']), []);
```

Mit Hilfe des Parameters *Options* kann dafür gesorgt werden, dass die Groß- und Kleinschreibung bei der Suche nicht beachtet wird und dass ein Datensatz auch dann gefunden wird, wenn er nur teilweise mit den angegebenen Werten übereinstimmt.

```
TLocateOptions = set of (loCaseInsensitive, loPartialKey);
```

Hätten wir *loPartialKey* gesetzt, dann hätte *Locate* auch einen *Bert Hellermann* gefunden. Mit dem Funktionsergebnis erfahren wir, ob die Suche erfolgreich war.

Soll nur in einer einzelnen Spalte gesucht werden, dann muss keine Typenumwandlung erfolgen:

```
Table1.Locate('bezeichnung', 'Monitor', []);
```

- GetBookmark, GotoBookmark, FreeBookmark, CompareBookmarks (Methoden)

```
TBookmark = Pointer;
function GetBookmark: TBookmark;
procedure GotoBookmark(Bookmark: TBookmark);
procedure FreeBookmark(Bookmark: TBookmark);
function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
```

Mit Hilfe der erwähnten Methoden lassen sich in der Datenmenge Lesezeichen setzen, die später wieder aufgerufen werden können.

```
private
   BM1: Pointer;
   BM2: Pointer;
```

Deklarieren Sie vor der Verwendung entsprechend viele *Pointer*-Variablen.

```
procedure TForm1.Setzen2Click(Sender: TObject);
begin
   BM1 := Table1.GetBookmark;
end;
```

Mit *GetBookmark* wird die aktuelle Position des Satzzeigers in der Variablen gespeichert.

```
procedure TForm1.Aufrufen2Click(Sender: TObject);
begin
    Table1.GotoBookmark(BM1);
end;
```

Mit *GotoBookmark* kann man an die gespeicherte Stelle zurückspringen. *FreeBookmark* löscht ein Lesezeichen, mit *CompareBookmark* kann festgestellt werden, ob zwei Lesezeichen auf denselben Datensatz zeigen.

- Bookmark (Eigenschaft)

```
property Bookmark: string;
```

Mittels der Eigenschaft *Bookmark* kann man die aktuelle Position des Satzzeigers in einem String speichern. Wenn man diesen String wieder der Eigenschaft *Bookmark* zuweist, bewegt man den Satzzeiger wieder an die betreffende Stelle.

Der String beginnt gewöhnlich mit einem (ASCII-)Null-Zeichen, Sie können ihn somit nicht mit beispielsweise *TEdit* anzeigen.

- AfterScroll, BeforeScroll (Ereignisse)

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

Diese Ereignisse treten vor beziehungsweise nach dem Verschieben des Datenzeigers auf. Beachten Sie in diesem Zusammenhang, dass bei einigen *TDataSet*-Ereignissen das Präfix *On* entfällt.

Ändern der Daten

- Append, AppendRecord (Methoden)

```
procedure Append;
procedure AppendRecord(const Values: array of const);
```

Mit *Append* wird hinter den letzten Eintrag der Tabelle gesprungen und diese in den Eingabe-Modus versetzt; die Eingabe muss mit *Post* (oder Methoden, welche *Post* aufrufen) bestätigt oder mit *Cancel* abgebrochen werden. Wird für die Tabelle ein Index verwendet, dann wird der Datensatz nach dem Aufruf von *Post* an der richtigen Position eingefügt.

Um neue Werte einzugeben, werden der Eigenschaft *Fields* die entsprechenden Werte zugewiesen; werden keine Werte zugewiesen, dann behalten diese den Wert NULL.

```
with Table1 do
begin
    Append;
    Fields[0].AsString := 'Karo';
    Fields[1].AsString := 'Ass';
    Post;
end;
```

Der Methode *AppendRecord* werden die einzugebenden Werte als Parameter übergeben, die Bestätigung mittels *Post* kann unterbleiben. Der neue Datensatz wird an die Tabelle angehängt.

```
Table1.AppendRecord('Karo', 'Ass');
```

- Insert, InsertRecord (Methoden)

```
procedure Insert;
procedure InsertRecord(const Values: array of const);
```

Die Methode *Insert* ähnelt *Append*, springt aber im Gegensatz dazu nicht an das Ende der Tabelle, sondern fügt den neuen Datensatz an der aktuellen Datenzeigerposition ein.

Wird für die Tabelle ein Index verwendet, dann wird der neue Datensatz an der richtigen Position eingefügt. Mit *InsertRecord* werden die übergebenen Parameter als neuer Datensatz an aktueller Datenzeigerposition eingefügt.

- Edit (Methode)

```
procedure Edit;
```

Die Methode *Edit* versetzt die Datenmenge in den Status *dsEdit*. Anschließend können die Daten des aktuellen Datensatzes geändert werden, abschließend muss die Änderung mit *Post* bestätigt oder mit *Cancel* verworfen werden.

```
with Table1 do
begin
    Edit;
    FieldByName('Feld1').AsString := 'Wert 1';
    FieldByName('Feld2').AsString := 'Wert 2';
    Post;
end;
```

- Delete (Methode)

```
procedure Delete;
```

Die Methode *Delete* löscht den aktuellen Datensatz. Dies muss nicht mit *Post* bestätigt werden.

- Post, Cancel (Methoden)

```
procedure Post;
procedure Cancel;
```

Mit der Methode *Post* werden die vorgenommenen Änderungen bestätigt, mit *Cancel* werden sie verworfen. Die Datenmenge hat danach den Status *dsBrowse*.

- ClearFields (Methode)

```
procedure ClearFields;
```

Mit *ClearFields* werden alle Einträge des aktuellen Datensatzes auf NULL gesetzt, ohne jedoch den Datensatz selbst zu löschen.

- Diverse Ereignisse (Ereignisse)

```
property AfterCancel(DataSet: TDataSet)
property AfterDelete(DataSet: TDataSet)
property AfterEdit(DataSet: TDataSet)
property AfterInsert(DataSet: TDataSet)
property AfterPost(DataSet: TDataSet)

property BeforeCancel(DataSet: TDataSet)
property BeforeDelete(DataSet: TDataSet)
property BeforeEdit(DataSet: TDataSet)
```

```
property BeforeInsert(DataSet: TDataSet)
property BeforePost(DataSet: TDataSet)

property OnNewRecord(DataSet: TDataSet)
```

Die erwähnten Ereignisse treten vor beziehungsweise nach den entsprechenden Methoden auf. Beachten Sie in diesem Zusammenhang, dass bei einigen *TDataSet*-Ereignissen das Präfix *On* entfällt.

In den entsprechenden Ereignisbehandlungsmethoden kann man nun beispielsweise verhindern, dass die Datenmenge in den Zustand *dsEdit* versetzt wird, indem man eine Exception auslöst.

```
procedure TForm1.Table1BeforeEdit(DataSet: TDataset);
begin
        Raise EDatabaseError.Create
                ('Datenmenge darf nicht geändert werden');
end;
```

Ein für den Anwender ähnliches Verhalten erreicht man, indem man dem Ereignis *OnAfterEdit* folgende Anweisungen zuweist:

```
procedure TForm1.Table1AfterEdit(DataSet: TDataset);
begin
    ShowMessage('Datenmenge kann nicht editiert werden');
    Table1.Cancel;
end;
```

Das Ereignis *OnNewRecord* tritt sowohl bei *Append*- als auch bei *Insert*-Aktionen auf, und zwar nach *BeforeInsert* und vor *AfterInsert*.

- OnDeleteError, OnEditError, OnPostError (Ereignisse)

```
property OnDeleteError(DataSet: TDataSet; E: EDatabaseError;
             var Action: TDataAction);
property OnEditError(DataSet: TDataSet; E: EDatabaseError;
             var Action: TDataAction);
property OnPostError(DataSet: TDataSet; E: EDatabaseError;
             var Action: TDataAction);
```

Tritt bei den dazugehörenden Aktionen ein Fehler auf, dann werden die erwähnten Ereignisse aufgerufen. Mittels des Variablenparameters *Action* kann bestimmt werden, wie auf den Fehler reagiert werden soll.

- *daFail:* Die Aktion wird abgebrochen, eine Fehlermeldung wird angezeigt.
- *daAbort:* Die Aktion wird ohne Fehlermeldung abgebrochen.
- *daRetry:* Die Aktion wird wiederholt – vorher sollte jedoch versucht werden, den Fehler zu beheben.

- Constraints (Eigenschaft, veröffentlicht)

```
property Constraints: TCheckConstraints;
```

SQL-Server erlauben umfangreiche Gültigkeitsprüfungen der zu speichernden Daten. Werden Dateneingaben jedoch nicht gleich zum Server geschickt, weil *Cached Updates* oder *TClientDataSet* verwendet wird, dann sollten die Gültigkeitsprüfungen schon vom Client durchgeführt werden, um hier spätere Probleme zu vermeiden.

Gültigkeitsprüfungen, die nur eine Spalte betreffen – beispielsweise ein beschränkter Wertebereich bei Zahlen –, können mit den Eigenschaften des jeweiligen *TField*-Nachfolgers realisiert werden. Mit Hilfe der Eigenschaft *Constraints* können Gültigkeitsprüfungen vorgenommen werden, die mehrere Spalten umfassen.

TCheckConstrains ist eine Kollektion von *TCheckConstraint*-Elementen. *TCheckConstraint* implementiert unter anderem die folgenden Eigenschaften:

- In *CustomConstraint* wird die Beschränkung in SQL-Syntax formuliert, beispielsweise:
- (nummer < 10) OR (nummer2 > 10)
- Beschränkungen, die von einem Datenbank-Server oder einem Daten-Dictionary importiert werden, findet man in der Eigenschaft *ImportedConstraint*.
- Wird gegen eine Beschränkung verstoßen, dann wird eine *EDBEngineError*-Exception ausgelöst, welcher der in *ErrorMessage* gespeicherte Text zugewiesen wird.

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* wollen wir in diesem Buch nicht besprechen. Informieren Sie sich in der Online-Hilfe oder einem Delphi-Datenbankbuch.

- 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 nicht zu 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';
```

- 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 dem Feld *Vorname* einen Wert zuzuweisen, formuliert man einfach:

```
Table1['Vorname'] := 'Anja';
```

- Lookup (Methode)

```
function Lookup(const KeyFields: string; const KeyValues: Variant;
        const ResultFields: string): Variant;
```

Mit *Lookup* kann der Inhalt eines bestimmten Datensatzes ermittelt werden, ohne den Satzzeiger zu bewegen. Der gesuchte Datensatz wird mit *KeyFields* und *KeyValues* spezifiziert (siehe *Locate*). Mit *ResultFields* gibt man an, welche Felder in die Ergebnismenge geschrieben werden sollen – mehrere Felder sind dabei durch Semikola zu trennen.

Das folgende Beispiel zeigt, wie ein Varianten-Record zerlegt und in einen String geschrieben wird (würde nur eine Spalte zurückgegeben, dann wäre das Ergebnis direkt mit *string* zuweisungskompatibel).

```
var
   v: Variant;
begin
   v := Query1.Lookup('Nummer', '30', 'Preis;Hersteller');
   Caption := string(VarArrayGet(v, [0])) + ';'
           + string(VarArrayGet(v, [1]));
```

- FieldDefs (öffentlich)

```
property FieldDefs: TFieldDefs;
```

Mit Hilfe von *FieldDefs* können die Metadaten einer Tabelle ermittelt und verändert werden. Diese Eigenschaft wird vor allem dazu benötigt, um zur Laufzeit Tabellen zu erstellen. *TFieldDefs* wird in einem eigenen Abschnitt besprochen.

- 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, Modified (Eigenschaften, öffentlich, nur Lesen)

```
property CanModify: Boolean;
property Modified: Boolean;
```

Mit Hilfe der Eigenschaft *CanModify* kann festgestellt werden, ob die Datenmenge geändert werden kann. Sie wird beispielsweise auch dann den Wert *false* haben, wenn bei SQL-JOINS die *TQuery*-Eigenschaft *RequestLive* auf *true* gesetzt wird. Vorsicht: Diese Eigenschaft ermittelt nicht, ob Sie die Schreibberechtigung auf dem betreffenden Datenbank-Server haben.

Die Eigenschaft *Modified* ist gleich *true*, wenn der aktuelle Datensatz geändert, aber *Post* oder *Cancel* noch nicht aufgerufen wurden.

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

```
property FieldCount: Integer;
property RecordCount: Integer;
property RecNo: Integer;
function IsSequenced: Boolean;
```

Mit Hilfe dieser Eigenschaften können die Zahl der Spalten (*FieldCount*), die Zahl der Datensätze (*RecordCount*) und die Nummer des aktuellen Datensatzes (*RecNo*) ermittelt werden.

```
Label1.Caption := 'Datensatz ' + IntToStr(Table1.RecNo)
       + ' von ' + IntToStr(Table1.RecordCount);
```

Durch die Zuweisung von *RecNo* kann der Satzzeiger auf den betreffenden Datensatz gesetzt werden. *RecNo* wird nur dann unterstützt, wenn *IsSequenced* den Wert *true* hat.

Beachten Sie, dass bei *RecordCount* nur die Zahl der Datensätze ermittelt wird, die bereits zum Client übertragen wurden. Angenommen, wir hätten eine Abfrage mit 10000 Datensätzen und zeigt diese in einem Grid an, dann wäre *RecordCount* direkt nach dem Öffnen der Abfrage bei vielleicht 20. Erst nach dem Aufruf von *Last* oder *FetchAll* können Sie eine Aussage über die tatsächliche Anzahl der Datensätze treffen.

- 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);
```

- AutoCalcFields (Eigenschaft, veröffentlicht), OnCalcFields (Ereignis)

```
property AutoCalcFields: Boolean;
property OnCalcFields(DataSet: TDataSet)
```

Um einem berechneten Feld einen Wert zuzuweisen, wird das Ereignis *OnCalcFields* verwendet. Mit *AutoCalcFields* kann die Aufrufhäufigkeit des Ereignisses verändert werden, siehe Online-Hilfe.

Filtern der Datenmenge

- Filtered (Eigenschaft, veröffentlicht), OnFilterRecord (Ereignis)

```
property Filtered: Boolean;
property OnFilterRecord(DataSet: TDataSet; var Accept: Boolean);
```

Wenn man eine Datenmenge filtern möchte, dann verwendet man eine entsprechende WHERE-Klausel, zur Not unter Verwendung einer USER DEFINED FUNCTION (UDF). Nun sind immer Situationen möglich, in denen selbst eine UDF nicht zum Ziel führt und die Daten auf dem Client gefiltert werden müssen. Dies hat den Nachteil, dass alle Datensätze zum Client übertragen werden müssen.

Das Ereignis *OnFilterRecord* wird für jeden Datensatz aufgerufen, wenn die Eigenschaft *Filtered* auf *true* gesetzt wird. Wird der Variablenparameter *Accept* auf *true* gesetzt, dann wird der Datensatz aufgenommen, wird er auf *false* gesetzt, unterbleibt dies.

```
procedure TForm1.IBTable1FilterRecord(DataSet: TDataSet;
   var Accept:     Boolean);
begin
   if DataSet.FieldByName('Nachnamen').AsString = 'Rasemann'
           then Accept := true
           else Accept := false;
end;
```

Bei *IBTable* kann man mit der Eigenschaft *Filtered* auch die Filterung über die Eigenschaft *Filter* aktivieren.

Es gibt Situationen, in denen eine Datenmenge nacheinander nach sehr vielen Kriterien gefiltert werden muss. Wenn die Summe der Datensätze, die bei den einzelnen Filtervorgängen anfällt, die Zahl der Datensätze insgesamt deutlich übersteigt, dann kann es zweckmäßig sein, die gesamte Datenmenge zunächst einmal in eine Client-Datenmenge zu übertragen und dann lokal zu filtern.

Aktualisieren der Datenmenge

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

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

Mit *Refesh* aktualisieren Sie die Datenmenge. Davor wird das Ereignis *BeforeRefresh* aufgerufen, anschließend das Ereignis *AfterRefresh*.

- FetchAll (Methode)

```
procedure FetchAll;
```

Normalerweise fordert die Datenmengenkomponente nur diejenigen Daten aus der Datenbank an, welche momentan zur Anzeige benötigt werden. Scrollt der Anwender beispielsweise durch eine Datenmenge, dann werden laufend neue Datensätze angefordert, was die Netzwerkbelastung erhöht.

Mit der Methode *FetchAll* kann man alle Datensätze von der Cursor-Position ausgehend bis zum Dateiende der Datenbank lesen und lokal zwischenspeichern.

Cached Updates

- CachedUpdates, UpdateObject (Eigenschaften, veröffentlicht)

```
property CachedUpdates: Boolean;
property UpdateObject: TIBDataSetUpdateObject;
```

Wird *CachedUpdates* auf *true* gestellt, dann werden zwischengespeicherte Aktualisierungen verwendet. Soll eine *TIBUpdateObject*-Komponente verwendet werden, so ist diese mit der Eigenschaft *UpdateObject* einzustellen.

- ApplyUpdates, CancelUpdates, RevertRecord (Methoden)

```
procedure ApplyUpdates;
procedure CancelUpdates;
procedure RevertRecord;
```

Mit *ApplyUpdates* werden die Änderungen in die Datenbank geschrieben, mit *CancelUpdates* verworfen. Beachten Sie, dass die Daten nur dann wirklich in die Datenbank übernommen werden, wenn die Transaktion mit COMMIT abgeschlossen wird.

Mit *RevertRecord* werden die Änderungen am aktuellen Datensatz rückgängig gemacht. Das ist möglich, solange zwischengespeicherte Aktualisierungen verwendet werden und der Datensatz noch nicht zum Server geschickt wurde.

- UpdatesPending (Eigenschaft, öffentlich, nur Lesen)

```
property UpdatesPending: Boolean;
```

Gibt es zwischengespeicherte Aktualisierungen, die noch nicht zum Server übertragen wurden, dann hat *UpdatesPending* den Wert *true*.

- UpdateRecordTypes (Eigenschaft, öffentlich)

```
property UpdateRecordTypes: TIBUpdateRecordTypes;
```

Mit *UpdateRecordTypes* wird spezifiziert, welche Datensätze momentan sichtbar sind. Funktioniert nur bei der Verwendung von zwischengespeicherten Aktualisierungen.

- OnUpdateRecord, OnUpdateError (Ereignisse)

```
property OnUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction);
property OnUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TIBUpdateAction);
```

Das Ereignis *OnUpdateRecord* tritt auf, wenn ein Datensatz von den zwischengespeicherten Aktualisierungen auf den Server übertragen wird, *OnUpdateError*, wenn dabei ein Fehler auftritt.

Der Parameter *UpdateKind* informiert darüber, ob das Ereignis beim Ändern (*ukModify*), Einfügen (*ukInsert*) oder Löschen (*ukDelete*) aufgetreten ist.

Mit Hilfe des Variablenparameters *UpdateAction* kann bestimmt werden, welche Aktion ausgeführt werden soll – beispielsweise, auf welche Art und Weise der Fehler behandelt werden soll. Der Parameter *UpdateAction* kann folgende Werte annehmen:

- *uaApplied:* Die Änderung wird eingetragen und aus dem Zwischenspeicher gelöscht.
- *uaAbort:* Die Aktualisierung wird ohne Fehlermeldung abgebrochen.
- *uaFail:* Die Aktualisierung wird mit Fehlermeldung abgebrochen.
- *uaRetry:* Es wird erneut versucht, die Änderung einzutragen. Zuvor sollte der Fehler möglichst behoben werden. (Tritt nur bei *OnUpdateError* auf.)
- *uaSkip:* Die Aktualisierung wird übergangen, die entsprechenden Daten bleiben im Zwischenspeicher.

10.2.4 TIBTable

Die Komponente *TIBTable* dient zum Zugriff auf eine einzelne Tabelle. Die Komponente ist von *TIBCustomDataSet* abgeleitet und ähnelt *TTable*.

Die Verwendung von *TIBTable* mag dann sinnvoll sein, wenn bestehende Programme mit minimalem Aufwand von der BDE nach IBX umgestellt werden sollen. Man erkauft sich die leichtere Umstellung jedoch mit deutlichen Einbußen bei der Performance, die Messungen aus Kapitel 9 sind da recht eindeutig. Auch bei simplen Beispielprogrammen mag wenig gegen den Einsatz von *TIBTable* sprechen.

Wenn Sie Anwendungen neu erstellen oder wenn im Zuge der Umstellung ohnehin umfangreichere Änderungen nötig sind, dann sollten Sie gleich zu *TIBDataSet* und vielleicht noch *TIBQuery* greifen.

Verbindung zur Tabelle

- TableName, TableTypes (Eigenschaften, veröffentlicht)

```
property TableName: string;
property TableTypes: set of (ttSystem, ttView);
```

  Mit der Eigenschaft *TableName* wird der Name derjenigen Tabelle oder VIEW angegeben, die *TTable* öffnen soll.

  Im Objektinspektor existiert für die Eigenschaft *TableName* eine ComboBox, mit der aus den vorhandenen Tabellen ausgewählt werden kann. Sollen dabei auch die VIEWS angezeigt werden, dann ist *ttView* in die Menge *TableTypes* aufzunehmen. Ebenso ist mit *ttSystem* zu verfahren, wenn Systemtabellen angezeigt werden sollen.

- CreateTable, DeleteTable (Methoden)

```
procedure CreateTable;
procedure DeleteTable;
```

  Mit diesen Methoden können zur Laufzeit Tabellen erstellt und gelöscht werden. Üblicherweise werden Änderungen der Metadaten jedoch per SQL-Anweisung vorgenommen, zumindest das Erstellen von Tabellen geht damit auch etwas einfacher. Wir wollen daher auf ein Beispiel verzichten.

- Exists (Eigenschaft, öffentlich, nur Lesen)

```
property Exists: Boolean;
```

  Wenn die angegebene Tabelle nicht (mehr) existiert, hat *Exists* den Wert *false*.

- ReadOnly (Eigenschaft, veröffentlicht)

```
property ReadOnly: Boolean;
```

  Wird *ReadOnly* auf *true* gesetzt, kann die *TTable*-Instanz keine Änderungen am Datenbestand vornehmen.

- EmptyTable (Methode)

```
procedure EmptyTable;
```

  Mit dieser Methode werden alle Datensätze aus der Tabelle gelöscht, die Tabellenstruktur wird jedoch beibehalten.

- GotoCurrent (Methode)

```
procedure GotoCurrent(Table: TIBTable);
```

  Die Methode *GotoCurrent* setzt den Datenzeiger auf denselben Datensatz wie den Datenzeiger der als Parameter übergebenen Tabelle. Auf diese Weise lassen sich sehr einfach zwei *TIBTable*-Instanzen synchronisieren.

Indizes

Viele Operationen bei *TIBTable* (Bereiche, Suche nach Schlüsseln) sind nur dann möglich, wenn die verwendeten Felder indiziert sind.

- DefaultIndex (Eigenschaft, veröffentlicht)

  ```
  property DefaultIndex: Boolean;
  ```

  Hat *DefaultIndex* den Wert *true*, dann werden die Daten entsprechend dem Primärschlüssel oder – falls ein solcher nicht existiert – nach dem Index sortiert, den die Komponente für den ersten hält.
- IndexName (Eigenschaft, veröffentlicht)

  ```
  property IndexName: string;
  ```

  Die Eigenschaft *IndexName* gibt den Namen des Sekundärindex an, nach dem die Datenmenge sortiert wird. Bleibt die Eigenschaft leer, dann wird nach dem Primärindex sortiert.
- IndexFieldNames (Eigenschaft, veröffentlicht)

  ```
  property IndexFieldNames: string;
  ```

  Alternativ zu *IndexName* können mit *IndexFieldNames* die einzelnen Spalten angegeben werden, aus denen ein Index gebildet werden soll.
- IndexFields (Eigenschaften, öffentlich)

  ```
  property IndexFields: [Index: Integer]: TField;
  property IndexFieldCount: Integer;
  ```

  Die an einem Index beteiligten Spalten werden mit *IndexFields* ermittelt, die Anzahl dieser Spalten mit *IndexFieldCount*.
- GetIndexNames (Methoden)

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

  Mit *GetIndexNames* werden die Namen aller vorhandenen Indizes abgefragt und in eine Liste geschrieben.
- IndexDefs (Eigenschaft, öffentlich)

  ```
  property IndexDefs: TIndexDefs;
  ```

  Um Informationen über die Indizes zu erhalten, wird die Eigenschaft *IndexDefs* verwendet.

Master-Detail-Verknüpfungen

- MasterSource (Eigenschaft, veröffentlicht)

  ```
  property MasterSource: TDataSource;
  ```

  Mit der Eigenschaft *MasterSource* wird angegeben, welche Datenquelle diese Datenmenge als Master-Tabelle steuert.

- MasterFields (Eigenschaft, veröffentlicht)

```
property MasterFields: string;
```

Der Eigenschaft *MasterFields* werden diejenigen Feldnamen zugewiesen, über die bei der Master-Tabelle die Verknüpfung gesteuert wird. Mehrere Feldnamen werden dabei durch Semikola getrennt. Im Gegensatz zu Desktop-Datenbanken müssen diese Felder in Anzahl und Feldart dem momentan verwendeten Schlüssel der Detailtabelle nicht (!) entsprechen.

Zur Entwurfszeit steht Ihnen zum Setzen dieser Eigenschaft der Feldverbindungsdesigner zur Verfügung.

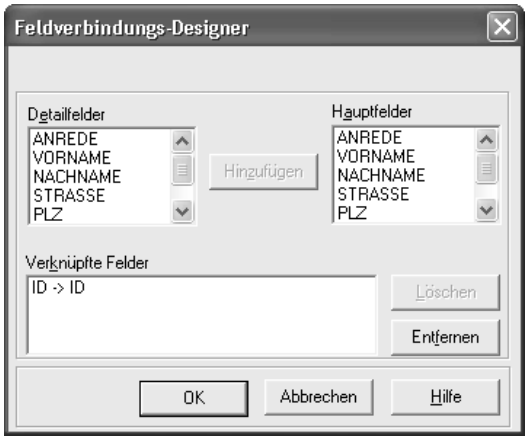


Abbildung 10.10: Der Feldverbindungsdesigner

Filtern der Datenmenge

- Filter, Filtered (veröffentlicht)

```
property Filter: string;
property Filtered: Boolean;
```

Eine Datenmenge kann mit der Eigenschaft *Filter* und/oder mit dem Ereignis *OnFilterRecord* gefiltert werden. Damit diese Filter aktiv sind, muss die Eigenschaft *Filtered* auf *true* gesetzt werden.

Der Inhalt der Eigenschaft *Filter* ist als WHERE-Klausel (ohne das Schlüsselwort WHERE) zu formulieren und wird auch in eine solche umgesetzt. Die Daten werden also nicht (!) lokal gefiltert.

TIBQuery

Die Komponente *TIBQuery* dient dem Zugriff auf die Datenbank mittels einer SQL-Anweisung. Dabei sind sowohl Abfragen als auch Datenänderungen oder die Definition von Metadaten möglich. Die Komponente ähnelt *TQuery*.

Wird eine aktualisierbare Datenmenge gewünscht, dann ist es effektiver, mit *TIBDataSet* zu arbeiten, statt *TIBQuery* mit *TIBUpdateSQL* zu kombinieren.

Die SQL-Anweisung

- SQL (Eigenschaft, veröffentlicht)

  ```
  property SQL: TStrings;
  ```

  Die Eigenschaft *SQL* erhält die SQL-Anweisung für die Abfrage oder die Datenmanipulation. Eine Abfrage wird mit *Open* geöffnet, eine Datenmanipulation mit *ExecSQL* ausgeführt.

  Die Datenbanksprache *SQL* wird in den Kapiteln 3 bis 5 beschrieben.

- ExecSQL (Methode)

  ```
  procedure ExecSQL;
  ```

  Wird der Eigenschaft *SQL* eine Anweisung zur Datenmanipulation zugewiesen (Erstellen von Tabellen, Ändern von Daten), dann kann *TQuery* keine Datenmenge öffnen und deshalb darf weder die Methode *Open* verwendet noch die Eigenschaft *Active* auf *true* gesetzt werden. Um eine solche SQL-Anweisung auszuführen, muss die Methode *ExecSQL* eingesetzt werden.

- RowsAffected (Eigenschaft, öffentlich, nur Lesen)

  ```
  property RowsAffected: Integer;
  ```

  Mit *RowsAffected* kann ermittelt werden, wie viele Datensätze bei der letzten Anweisung geändert oder gelöscht wurden. Bei Anweisungen, die keine Daten ändern können (SELECT beispielsweise), wird -1 zurückgegeben.

- Prepared (Eigenschaft, öffentlich), Prepare, Unprepare (Methoden)

  ```
  property Prepared: Boolean;
  procedure Prepare;
  procedure UnPrepare;
  ```

  Bevor eine Abfrage ausgeführt werden kann, muss sie vorbereitet werden. Der Server interpretiert dabei das SQL-Statement, ermittelt, welche Tabellen er in welcher Reihenfolge lesen soll und ob dabei Indizes verwendet werden.

  Wenn die Abfrage nicht explizit vorbereitet wird, dann erfolgt das automatisch, allerdings bei jeder einzelnen Abfrage. Oft werden jedoch mehrere identische Abfragen hintereinander ausgeführt, die sich allenfalls durch die Parameter unterscheiden. Es ist dann ineffektiv, die Abfrage jedes Mal zu interpretieren.

  Deshalb besteht die Möglichkeit, eine Abfrage explizit vorzubereiten, indem *Prepared* auf *true* gesetzt wird oder die Methode *Prepare* aufgerufen wird. Vergessen Sie nicht, *Unprepare* aufzurufen (oder *Prepared* auf *false* zu setzen), wenn Sie die Abfrage ändern.

Parameter

- Params (Eigenschaft, veröffentlicht)

  ```
  property Params[Index: Word]: TParams;
  ```

  Mit der Eigenschaft *Params* kann auf die Parameter einer Abfrage zugegriffen werden. Um Verwechslungen zu vermeiden, sollten Sie dafür jedoch die Methode *ParamByName* bevorzugen.

- ParamByName (Methode)

  ```
  function ParamByName(const Value: string): TParam;
  ```

  Statt über den Index – wie bei der Eigenschaft Params – können Parameter auch über den Parameternamen gesetzt werden. Dazu dient die Methode *ParamByName*.

- DataSource (Eigenschaft, veröffentlicht)

  ```
  property DataSource: TDataSource;
  ```

  Die Eigenschaft *DataSource* wird benötigt, um Master-Detail-Beziehungen mit Hilfe von *TQuery*-Komponenten herzustellen.

- ParamCount (Eigenschaft, öffentlich, nur Lesen)

  ```
  property ParamCount: Word;
  ```

  Mit *ParamCount* kann die Anzahl der vorhandenen Parameter abgefragt werden.

10.2.5 TIBStoredProc

Die Komponente *TIBStoredProc* dient zum Zugriff auf STORED PROCEDURES und ähnelt *TStoredProc*. Sie sollten nur auf auszuführende Prozeduren mit dieser Komponente zugreifen. Für Prozeduren, die eine Ergebnismenge liefern, verwenden Sie *TIBQuery* oder *TIBDataSet*.

- StoredProcName (Eigenschaft, veröffentlicht)

  ```
  property StoredProcName: String;
  ```

  Mit dieser Eigenschaft wird die auszuführende Prozedur ausgewählt.

- ExecProc (Methode)

  ```
  procedure ExecProc;
  ```

  Mit *ExecProc* wird die Prozedur ausgeführt.

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

  ```
  property Params[Index: Word]: TParams;
  function ParamByName(const Value: string): TParam;
  ```

Mit der Eigenschaft *Params* kann auf die Parameter einer Abfrage zugegriffen werden. Um Verwechslungen zu vermeiden, sollten Sie dafür jedoch die Methode *ParamByName* bevorzugen.

- ParamCount (Eigenschaft, öffentlich, nur Lesen)

```
property ParamCount: Word;
```

Mit *ParamCount* kann die Anzahl der vorhandenen Parameter abgefragt werden.

- Prepared (Eigenschaft, öffentlich), Prepare, Unprepare (Methoden)

```
property Prepared: Boolean;
procedure Prepare;
procedure UnPrepare;
```

Bevor eine Prozedur ausgeführt werden kann, muss sie vorbereitet werden. Im Vergleich zu einer Abfrage ist der Zeitbedarf dabei gering, weil die Prozedur ja ohnehin schon binär gespeichert wurde.

Trotzdem besteht die Möglichkeit, eine Prozedur explizit vorzubereiten, indem *Prepared* auf *true* gesetzt wird oder die Methode *Prepare* aufgerufen wird – bei einer wiederholten Ausführung können Sie so den Vorgang noch ein wenig beschleunigen. Vergessen Sie nicht, *Unprepare* aufzurufen (oder *Prepared* auf *false* zu setzen), wenn Sie die Prozedur ändern.

- StoredProcedureNames (Eigenschaft, öffentlich, nur Lesen)

```
property StoredProcedureNames: TStrings;
```

Mit *StoredProcedureNames* kann eine Liste aller STORED PROCEDURES in der Datenbank erstellt werden.

10.2.6 TIBUpdateSQL

Die Komponente *TIBUpdateSQL* dient dazu, Abfragen aktualisierbar zu machen, die für gewöhnlich keine aktualisierbare Ergebnismenge liefern – beispielsweise JOINS. Diese Komponente ähnelt der Komponente *TUpdateSQL* und ist nur dazu gedacht, Anwendungen mit minimalem Aufwand von der BDE nach IBX umzustellen.

Wenn Sie Anwendungen neu beginnen, dann sollten Sie gleich *TIBDataSet* verwenden.

- DeleteSQL, InsertSQL, ModifySQL, RefreshSQL (Eigenschaften, veröffentlicht)

```
property DeleteSQL: TStrings;
property InsertSQL: TStrings;
property ModifySQL: TStrings;
property RefreshSQL: TStrings;
```

Mit diesen vier Eigenschaften werden die SQL-Anweisungen vorgegeben, die beim Löschen, Einfügen oder Ändern von Datensätzen beziehungsweise beim Aktualisieren der Ergebnismenge ausgeführt werden.

- Query (Eigenschaft, öffentlich, nur Lesen)

```
property Query[UpdateKind: TUpdateKind]: TQuery;
```

Die Komponente *TUpdateObject* verwaltet drei *TQuery*-Komponenten, welche die Aufgaben des Löschens, des Einfügens und des Änderns von Datensätzen zwischen sich aufteilen. Mit Hilfe der Eigenschaften *DeleteSQL*, *InsertSQL* und *ModifySQL* kann auf die Eigenschaft *SQL* dieser drei *TQuery*-Instanzen zugegriffen werden. Sollen andere Eigenschaften, Methoden oder Ereignisse verwendet werden, dann ist die Array-Eigenschaft *Query* zu verwenden.

- DataSet (Eigenschaft, öffentlich, nur Lesen)

```
property DataSet: TDataSet;
```

Die Eigenschaft *DataSet* gibt die Datenmenge an, mit der die Komponente verbunden ist. Normalerweise wird die Eigenschaft zur Entwurfszeit automatisch gesetzt.

- SetParams, ExecSQL, Apply (Methoden)

```
procedure SetParams(UpdateKind: TUpdateKind);
procedure ExecSQL(UpdateKind: TUpdateKind);
procedure Apply(UpdateKind: TUpdateKind);
```

Für gewöhnlich löst die über *DataSet* verbundene Datenmengenkomponente automatisch die Aktualisierung aus. Muss dies – aus welchen Gründen auch immer – manuell geschehen, dann werden mit *SetParams* die Parameter gesetzt, mit *ExecSQL* wird anschließend die Aktualisierung vorgenommen. Mit *Apply* werden diese beiden Methoden hintereinander abgearbeitet.

10.2.7 TIBDataSet

Die Komponente *TIBDataSet* erlaubt es, für alle möglichen Situationen eigene SQL-Anweisungen zu verfassen. Diese Komponente verhält sich in etwa wie eine Kombination aus *TIBQuery* und *TIBUpdateSQL*, ohne jedoch die Änderungen zwischenzuspeichern.

- SelectSQL, RefreshSQL, InsertSQL, UpdateSQL, DeleteSQL (Eigenschaften, veröffentlicht)

```
property SelectSQL: TStrings;
property RefreshSQL: TStrings;
property InsertSQL: TStrings;
property UpdateSQL: TStrings;
property DeleteSQL: TStrings;
```

Mittels dieser Eigenschaften werden die einzelnen SQL-Statements zugewiesen. Mehr dazu im Beispiel.

Beim Öffnen der Datenmenge werden alle diese SQL-Statements vorbereitet, was natürlich etwas Zeit braucht (die dann beim Ausführen dieser Anweisungen wieder eingespart wird). Wenn beispielsweise sichergestellt ist, dass Datensätze ohnehin nicht gelöscht werden, dann können Sie das Öffnen der Datenmenge dadurch etwas beschleunigen, dass Sie *DeleteSQL* leer lassen.

- GeneratorField (Eigenschaft, veröffentlicht)

```
property GeneratorField: TIBGeneratorField;
```

Für Primärschlüsselspalten bietet sich die Verwendung eines Generators an, der für durchlaufende, eindeutige Nummern sorgt. Mittels der Eigenschaft *GeneratorField* kann man sehr einfach Generatorwerte abrufen und in die Datenmenge einfügen.

Für die Eigenschaft *GeneratorField* gibt es einen Eigenschaftseditor, mit dessen Hilfe Sie den *Generator* und das *Feld* wählen können, in das der neue Generatorwert eingefügt wird. *Erhöhen um* legt fest, wie der Generator bei jedem neuen Aufruf verändert wird, und kann normalerweise auf eins gelassen werden.

Zuletzt muss noch spezifiziert werden, wann der neue Generatorwert angefordert wird:

- Bei *On New Record* wird der Generatorwert nach dem Aufruf von *Append* oder *Insert* vom Server geholt. Dies hat den Vorteil, dass der Generatorwert während der Bearbeitung des Datensatzes bereits angezeigt wird, allerdings »verfällt« unnötigerweise ein Generatorwert, wenn die Bearbeitung dann mit *Cancel* abgebrochen wird.
- Soll dies vermieden werden, dann wird mit *On Post* der Generatorwert erst dann angefordert, wenn die Methode *Post* aufgerufen wird – während der Bearbeitung wird dann die Nummer nicht angezeigt, weil sie noch nicht generiert wurde.
- Wird *On Server* gewählt, dann muss der Wert – beispielsweise mittels eine Triggers – auf dem Server eingefügt werden. Die betreffende Spalte wird auf dem Client jedoch für Eingaben gesperrt.

Abbildung 10.11: Eigenschaftseditor für GeneratorField

- Prepared (Eigenschaft, öffentlich), Prepare, Unprepare (Methoden)

```
property Prepared: Boolean;
procedure Prepare;
procedure UnPrepare;
```

  Bevor eine Abfrage ausgeführt werden kann, muss sie vorbereitet werden. Der Server interpretiert dabei das SQL-Statement, ermittelt, welche Tabellen er in welcher Reihenfolge lesen soll und ob dabei Indizes verwendet werden.

  Wenn die Abfrage nicht explizit vorbereitet wird, dann erfolgt das automatisch, allerdings bei jeder einzelnen Abfrage. Oft werden jedoch mehrere identische Abfragen hintereinander ausgeführt, die sich allenfalls durch die Parameter unterscheiden. Es ist dann ineffektiv, die Abfrage jedes Mal zu interpretieren.

  Deshalb besteht die Möglichkeit, eine Abfrage explizit vorzubereiten, indem *Prepared* auf *true* gesetzt wird oder die Methode *Prepare* aufgerufen wird. Vergessen Sie nicht, *Unprepare* aufzurufen (oder *Prepared* auf *false* zu setzen), wenn Sie die Abfrage ändern.

- ForcedRefresh (Eigenschaft, veröffentlicht)

```
property ForcedRefresh: Boolean;
```

  Normalerweise wird die Datenmenge nur dann aktualisiert, wenn sie das für nötig hält. Soll mittels *RefreshSQL* bei jedem Aufruf von *Post* die Datenmenge aktualisiert werden, dann ist *ForcedRefresh* auf *true* zu setzen.

- QSelect, QRefresh, QInsert, QUpdate, QDelete (Eigenschaften, öffentlich, nur Lesen)

```
property QSelect: TIBSQL;
property QRefresh: TIBSQL;
property QInsert: TIBSQL;
property QUpdate: TIBSQL;
property QDelete: TIBSQL;
```

  Die Statements werden mittels einzelner *TIBSQL*-Komponenten abgesetzt (siehe nächster Abschnitt). Mittels dieser Eigenschaften erhält man einen Zeiger auf diese Komponenten.

10.2.8 TIBSQL

Die Komponente TIBSQL ist die spartanischte Möglichkeit, auf InterBase zuzugreifen. In der Regel dürfte man diese Komponente für DML- und DDL-Anweisungen nutzen, es können darüber aber auch SELECT-Statements gefahren werden.

TIBSQL ist nicht von *TDataSet* und somit auch nicht von *TIBCustomDataSet*, sondern direkt von *TComponent* abgeleitet. Interessant ist diese Komponente deshalb vor allem dann, wenn kleine Tools für Spezialaufgaben in der Dateigröße minimiert werden müssen.

- SQL (Eigenschaft, veröffentlicht), ExecQuery, Close (Methoden)

```
property SQL: TStrings;
procedure ExecQuery;
procedure Close;
```

Mit der Eigenschaft SQL wird das auszuführende Statement zugewiesen, mit *ExecSQL* wird es ausgeführt. Liefert das Statement eine Ergebnismenge, dann kann diese mit *Close* geschlossen werden.

- Params (Eigenschaft, öffentlich, nur Lesen), ParamByName (Methode)

```
property Params: TIBXSQLDA;
function ParamByName(Idx: String): TIBXSQLVAR;
```

Einen Zeiger auf die *TIBXSQLDA*-Instanz liefert die Eigenschaft *Params*. Mehr dazu in der Online-Hilfe. Auf die einzelnen Parameter kann auch mittels der Methode *ParamByName* zugegriffen werden.

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

```
property BOF: Boolean;
property EOF: Boolean;
procedure Next;
```

TIBSQL stellt nur eindirektionale Cursor zur Verfügung, es gibt somit auch nur eine Richtung, in der Datenmenge zu navigieren. Mit *EOF* kann man ermitteln, ob man das Ende der Datenmenge erreicht hat.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with IBSQL1 do
  begin
    ExecQuery;
    while not EOF do
    begin
      Memo1.Lines.Add(FieldByName('Nachname').AsString);
      Next;
    end;
    Close;
  end; {with IBSQL1 do}
end; {procedure TForm1.Button1Click}
```

- Fields (Eigenschaft, öffentlich, nur Lesen), FieldByName (Methode)

```
property Fields[const Idx: Integer]: TIBXSQLVAR;
function FieldByName[FieldName: String]: TIBXSQLVAR;
```

Mit *Fields* beziehungsweise *FieldByName* kann man auf die einzelnen Spalten einer Ergebnismenge zugreifen. Der Typ *TIBXSQLVAR* ähnelt weitgehend *TField*, so finden Sie beispielsweise auch hier Eigenschaften wie *AsString* oder *AsInteger*.

10.2.9 TIBDatabaseInfo

Mittels *TIBDatabaseInfo* erhalten Sie Informationen über die Datenbank. Diese Informationen werden Ihnen von öffentlichen Nur-Lesen-Eigenschaften zur Verfügung gestellt. Wir wollen nur die wichtigeren dieser Eigenschaften hier behandeln.

- Database (Eigenschaft, veröffentlicht)

  ```
  property Database: TIBDatabase;
  ```

  Über diese Eigenschaft verbinden Sie die Komponente mit der Datenbankkomponente. Eine Transaktionskomponente ist hier nicht erforderlich.

Datenbank

- DBFileName

  ```
  property DBFileName: String;
  ```

  Dateiname der primären Datenbankdatei.
- ReadOnly

  ```
  property ReadOnly: Long;
  ```

  Wenn es sich um eine Read-Only-Datenbank handelt, hat *ReadOnly* den Wert eins, ansonsten den Wert null.
- DBSQLDialect

  ```
  property DBSQLDialect: Long;
  ```

  DBSQLDialect kann den Wert eins oder drei haben.
- SweepInterval

  ```
  property SweepInterval: Long;
  ```

  Das SweepInterval ermitteln Sie mit der gleichnamigen Eigenschaft.
- UserNames

  ```
  property UserNames: TStringList;
  ```

  Die Liste aller mit der Datenbank verbundenen Benutzer erhalten Sie mit *UserNames*. Wird für mehrere Verbindungen derselbe Benutzername verwendet, dann erscheint dieser auch mehrmals in der Liste.

Lese- und Schreibvorgänge

- Reads, ReadIdxCount, ReadSeqCount

  ```
  property Reads: Long;
  property ReadIdxCount: TStringList;
  property ReadSeqCount: TStringList;
  ```

Mit der Eigenschaft *Reads* ermitteln Sie die Gesamtzahl der Lesevorgänge seit der Verbindung zur Datenbank, davon mit *ReadIdxCount* die Zahl derjenigen, die über einen Index erfolgen, und mit *ReadSeqCount* diejenigen, die sequenziell erfolgen.

Es mag Sie verwundern, dass eine Eigenschaft, die eine Anzahl liefern soll, vom Typ *TStringList* ist. Diese beiden Eigenschaften schlüsseln jedoch die Anzahl der betreffenden Lesevorgänge gleich nach den einzelnen Tabellen auf. Das sieht beispielsweise wie folgt aus:

```
2=103
3=5
4=40
5=103
6=9
9=12
128=104
```

Die unteren Tabellennummern sind Systemtabellen, *128* ist in unserer Beispieldatenbank die Tabelle *t_adressen*, in diesem Fall erfolgten hier 104 Lesevorgänge über einen Index.

- Writes, InsertCount, UpdateCount, DeleteCount

```
property Writes: Long;
property InsertCount: TStringList;
property UpdateCount: TStringList;
property DeleteCount: TStringList;
```

Mit *Writes* ermittelt man die Schreibvorgänge insgesamt, mit den anderen Eigenschaften werden sie nach Insert-, Update- und Delete-Vorgängen sowie nach Tabellen aufgeschlüsselt.

- ForcedWrites

```
property ForcedWrites: Long;
```

Werden Schreibvorgänge sofort auf die Platte geschrieben, dann hat *ForcedWrites* den Wert eins, andernfalls den Wert null.

Speicherverwaltung

- Allocation

```
property Allocation: Long;
```

Allocation ermittelt die Anzahl der reservierten Datenbankseiten.

- CurrentMemory, MaxMemory

```
property CurrentMemory: Long;
property MaxMemory: Long;
```

Mit *CurrentMemory* ermittelt man die Menge des aktuell belegten Speichers (in Byte, auf dem Server), mit *MaxMemory* die Menge, die seit Beginn der Datenbankverbindung maximal belegt wurde.

- PageSize

```
property PageSize: Long;
```

Die Größe der einzelnen Datenbankseiten ermitteln Sie mit *PageSize*.

10.2.10 TIBMonitor

Mit der Komponente *TIBMonitor* können Sie mitverfolgen, welche Informationen mit dem InterBase-Client ausgetauscht werden. Vergessen Sie nicht, dazu die Eigenschaft *TraceFlags* sowohl in der Komponente selbst als auch bei *TIBDatabase* entsprechend zu setzen.

- OnSQL (Ereignis)

```
property OnSQL(EventText: String);
```

Das Ereignis *OnSQL* tritt dann auf, wenn eine SQL-Anweisung zum InterBase-Client geschickt wird.

```
procedure TForm1.IBSQLMonitor1SQL(EventText: String);
begin
   Memo1.Lines.Add(EventText);
end;
```

- Enabled (Eigenschaft, veröffentlicht)

```
property Enabled: Boolean;
```

Deaktivieren Sie den Monitor, indem Sie *Enabled* auf *false* setzen.

10.2.11 TIBEvents

Die Komponente *TIBEvents* setzt *InterBase*-Events in Delphi-Ereignisse um. Beachten Sie bitte, dass der Event erst dann ausgelöst wird, wenn die dazugehörende Transaktion mit COMMIT beendet wird.

- Database (Eigenschaft, veröffentlicht)

```
property Database: TIBDatabase;
```

Über diese Eigenschaft verbinden Sie die Komponente mit der Datenbankkomponente. Eine Transaktionskomponente ist hier nicht erforderlich.

- Events (Eigenschaft, veröffentlicht)

```
property Events: TStrings;
```

In der Eigenschaft *Events* zählen Sie die Events auf, auf welche die Komponente reagieren soll. Die Beschränkung auf maximal 15 Events pro Komponente existiert nicht mehr, *TIBEvents* spaltet automatisch neue Threads ab, wenn es mehr als 15 Events werden.

Abbildung 10.12: Eigenschaftseditor für Events

Für die Eigenschaft *Events* wird ein Eigenschaftseditor zur Verfügung gestellt. Bitte beachten Sie die Groß- und Kleinschreibung der Events.

- OnEventAlert, OnError (Ereignisse)

```
property OnEventAlert(Sender: TObject; EventName: String;
        EventCount: Integer; var CancelAlerts: Boolean);
property OnError(Sender: TObject; ErrorCode: Integer);
```

Das Ereignis *OnEventAlert* tritt ein, wenn InterBase ein Event auslöst.

Fehler, die beim Bearbeiten von Events auftreten, werden über das Ereignis *OnError* gemeldet – dieses Ereignis dient nicht (!) zum Abfangen aller Exceptions. Um Exceptions abzufangen – auch solche, die Sie in InterBase selbst erstellt haben –, verwenden Sie eine *try..except..end*-Konstruktion.

- Registered (Eigenschaft, veröffentlicht), RegisterEvents, UnRegisterEvents (Methoden)

```
property Registered: Boolean;
procedure RegisterEvents;
procedure UnRegisterEvents;
```

Die Komponente reagiert nur dann auf Events, wenn *Registered* den Wert *true* hat. Diese Eigenschaft können Sie auch mittels der beiden Methoden auf *true* beziehungsweise *false* setzen.

- AutoRegister (Eigenschaft, veröffentlicht)

```
property AutoRegister: Boolean;
```

Hat *AutoRegister* den Wert *true*, dann werden die Events registriert, sobald eine Verbindung zur Datenbank erstellt wird.

10.2.12 TIBExtract

Die Komponente *TIBExtract* dient dazu, gezielt Teile der Metadaten zu ermitteln.

- Database, Transaction (Eigenschaften, veröffentlicht)

```
property Database: TIBDatabase;
property Transaction: TIBTransaction;
```

Mit *Database* wird die *TIBDatabase*-Komponente ausgewählt, über welche die Verbindung zur Datenbank hergestellt wird, mit *Transaction* die *TIBTransaction*-Instanz.

Jede Datenmengenkomponente muss zwingend mit einer Transaktionskomponente verbunden werden, dafür gibt es die Eigenschaft *Transaction*. Wird die Eigenschaft *Database* gesetzt und hat die betreffende *TIBDatabase*-Instanz eine *DefaultTransaction*, dann wird diese automatisch nach *Transaction* übernommen.

- ExtractObject (Methode), Items (Eigenschaft, öffentlich, nur Lesen)

```
procedure ExtractObject(ObjectType : TExtractObjectTypes; ObjectName :
String = ''; ExtractTypes : TExtractTypes = []);;
property Items: TStrings;
```

Um sich den gewünschten Teil der Metadaten anzeigen zu lassen, wird zunächst mit den entsprechenden Parametern *ExtractObject* aufgerufen, das SQL-Script steht danach in der Eigenschaft *Items*.

Die Typen für die Parameter sind wie folgt definiert:

```
TExtractObjectTypes = (eoDatabase, eoDomain, eoTable, eoView,
eoProcedure, eoFunction, eoGenerator, eoException, eoBLOBFilter, eoRole,
eoTrigger, eoForeign, eoIndexes, eoChecks, eoData);
TExtractType = (etDomain, etTable, etRole, etTrigger, etForeign, etIndex,
etData, etGrant, etCheck, etAlterProc);
TExtractTypes = set of TExtractType;
```

Um beispielsweise alles über die Tabellendefinition von *t_mitarbeiter* sowie die dazugehörenden Domänen-Definitionen zu erfahren, würde man folgenden Aufruf formulieren:

```
IBExtract2.ExtractObject(eoTable, 'T_MITARBEITER', [etDomain, etTable]);
```

Das Ergebnis wäre das folgende:

```
/* Domain definitions */
CREATE DOMAIN "D_ID" AS INTEGER NOT NULL;
CREATE DOMAIN "D_NAME" AS VARCHAR(20) CHARACTER SET ISO8859_1;

/* Table: T_MITARBEITER, Owner: SYSDBA */

CREATE TABLE "T_MITARBEITER"
(
```

```
  "ID"          "D_ID",
  "VORNAME"     "D_NAME",
  "NACHNAME"    "D_NAME",
 PRIMARY KEY ("ID")
);
```

10.2.13 TIBClientDataSet

Von der Beschreibung von *TIBClientDataSet* wollen wir Abstand nehmen. Diese Komponente ist eine Kombination aus *TIBDataSet*, *TDataSetProvider* und *TClientDataSet* und die Erfahrung zeigt, dass man solche Komponentenketten lieber diskret aufbaut. Zudem müssten wir uns dann intensiv mit *TClientDataSet* beschäftigen, was vom eigentlichen Thema dieses Buches ziemlich weit wegführen würde.

10.3 InterBase Admin

Auf der Palettenseite InterBase finden sich eine Reihe von Komponenten zur Durchführung von Verwaltungsaufgaben.

10.3.1 TIBCustomService

Bis auf die Installationskomponenten sind alle Komponenten dieser Palettenseite von *TIBCustomService* abgeleitet. Wir wollen uns daher zunächst diese Klasse etwas näher ansehen.

- ServerName, Protocol, Params, LoginPrompt (Eigenschaften, veröffentlicht)

  ```
  property ServerName: String;
  property Protocol: TProtocol;
  property Params: TStrings;
  property LoginPrompt: Boolean;
  ```

  Um die Verbindung zum Server herzustellen, müssen der Name des Servers und das Protokoll angegeben werden. Mittels *Params* werden Benutzernamen und Passwort spezifiziert. Soll ein Login-Dialog unterbleiben, ist *LoginPrompt* auf *false* zu setzen.

  Einige Komponenten arbeiten mit einer bestimmten Datenbank, die dann auch anzugeben ist. Es wird auch ein Komponenten-Editor zur Verfügung gestellt, mit dessen Hilfe diese Eigenschaften gewählt werden können.

- Active, Attach, Detach (Methoden)

  ```
  property Active: Boolean;
  procedure Attach;
  procedure Detach;
  ```

Um die Verbindung zum Server herzustellen, wird *Active* auf *true* gesetzt oder die Methode *Attach* aufgerufen. Zum Trennen der Verbindung setzt man *Active* auf *false* oder ruft *Detach* auf.

- OnAttach, OnLogin (Ereignisse)

```
property OnAttach(Sender: TObject);
property OnLogin(Database: TIBCustomService; LoginParams: TStrings);
```

Wird eine Verbindung zum Server hergestellt, dann wird das Ereignis *OnAttach* aufgerufen. Während des Anmeldevorgangs wird *OnLogin* aufgerufen, über *LoginParams* können Benutzernamen und Passwort angegeben werden.

TIBControlService

Eine häufig genutzte Vorfahren-Komponente ist auch *TIBControlService*, so dass wir die damit implementierte Eigenschaft auch nur hier behandeln wollen.

- IsServiceRunning (Eigenschaft, öffentlich, nur Lesen)

```
property IsServiceRunning: Boolean;
```

Während der Server die Aufgabe bearbeitet, hat *IsServiceRunning* den Wert *true*. Um verschiedene Schritte streng zu serialisieren, verwenden Sie folgende Konstruktion:

```
while IsServiceRunning do Sleep(5);
```

Damit wird so lange gewartet, bis der Server mit seiner jeweiligen Aufgabe fertig ist.

10.3.2 TIBConfigService

Mit der Komponente *TIBConfigService* können Sie einige Datenbankeinstellungen vornehmen:

- BringDatabaseOnline, ShutdownDatabase (Methoden)

```
procedure BringDatabaseOnline;
procedure ShutdownDatabase(Options: TShutdownMode; Wait: Integer);
```

Mit *BringDatabaseOnline* wird eine Datenbank (wieder) gestartet, mit *ShutdownDatabase* kann sie heruntergefahren werden. Mittels des Parameters *Options* kann bestimmt werden, wie rigoros dabei vorgegangen werden soll:

- *Forced*: Die Datenbank wird nach der mit *Wait* bestimmten Zeit heruntergefahren. Soll die Datenbank sofort heruntergefahren werden, setzen Sie *Wait* auf null (Sekunden).
- *DenyTransaction*: Neue Transaktionen werden für die mit *Wait* festgelegte Zeit abgelehnt, danach wird die Datenbank heruntergefahren. Wenn noch Transaktionen aktiv sind, schlägt dieser Versuch fehl.

- *DenyAttachment*: Neue Verbindungen werden für die mit *Wait* festgelegte Zeit abgelehnt, danach wird die Datenbank heruntergefahren. Wenn dann noch Verbindungen mit der Datenbank aktiv sind, schlägt dieser Versuch fehl.

- SetAsyncMode (Methode)

```
procedure SetAsyncMode(Value: Boolean);
```

  Mit dieser Methode können Sie den Wert für FORCED WRITES setzen, allerdings in Negation: Rufen Sie *SetAsyncMode* mit *True* auf, dann werden die Schreibzugriffe gepuffert, was sie beschleunigt, aber im Falle eines Serverabsturzes auch zu Datenverlusten führen kann. Setzen Sie *Value* auf *false*, dann werden alle Änderungen direkt auf die Festplatte geschrieben.

- SetDBSqlDialect, SetReadonly (Methoden)

```
procedure SetDBSqlDialect(Value: Integer);
procedure SetReadonly(Value: Boolean);
```

  Mit diesen Methoden können Sie den SQL-Dialekt setzen und eine Read-Only-Datenbank erstellen.

- SetPageBuffers, SetSweepInterval (Methoden)

```
procedure SetPageBuffers (Value: Integer);
procedure SetSweepInterval(Value: Integer);
```

  Mit diesen Methoden spezifizieren Sie die Zahl der Puffer-Seiten sowie das Sweep-Intervall.

10.3.3 TIBBackupService

Mit der Komponente *TIBBackupService* kann eine Datenbank gesichert werden.

- DatabaseName, BackupFile (Eigenschaft, veröffentlicht), ServiceStart (Methode)

```
property DatabaseName: String;
property BackupFile: TStrings;
procedure ServiceStart;
```

  Mit *DatabaseName* wird die Datenbank spezifiziert, von der ein Backup gezogen werden soll, die Dateinamen der Sicherungsdateien werden mit *BackupFile* angegeben, ausgeführt wird der Dienst mit *ServiceStart*.

  Die Eigenschaft *BackupFile* ist als Werteliste aufgebaut und beinhaltet sowohl die Dateinamen als auch die Dateigröße (in Bytes) der einzelnen Dateien. Mit den folgenden Werten könnte man die Sicherung auf drei Disketten ziehen:

```
c:\backup\file_1=1440000
c:\backup\file_2=1440000
c:\backup\file_3=
```

- Verbose (Eigenschaft, veröffentlicht), Eof (Eigenschaft, öffentlich, nur Lesen), GetNextLine (Methode)

  ```
  property Verbose: Boolean;
  property Eof: Boolean;
  function GetNextLine: String;
  ```

  Soll ein Protokoll des Vorgangs erstellt werden, dann ist *Verbose* auf *true* zu setzen. *GetNextLine* wird dann wiederholt so oft aufgerufen, bis *Eof* den Wert *true* hat.

```
srvBackup.ServiceStart;
if srvBackup.Verbose then
begin
  while not srvBackup.Eof do
  begin
    Application.ProcessMessages;
    frmVerbose.edOutput.Lines.Add(srvBackup.GetNextLine)
  end;
end {if srvBackup.Verbose then}
else
begin
  while srvBackup.IsServiceRunning
    do Application.ProcessMessages;
end; {else srvBackup.Verbose then}
```

- Options (Eigenschaft, veröffentlicht)

  ```
  property Options: TBackupOptions;
  ```

  Mit Hilfe der Mengeneigenschaft *Options* können noch einige Einstellungen vorgenommen werden. Die Menge enthält unter anderem die folgenden Elemente:

  - *IgnoreChecksums*: Während des Backups werden die Quersummen der Datenseiten überprüft. Tritt hier ein Fehler auf, dann kann auch kein Backup gezogen werden. Soll dies trotzdem geschehen, dann ist das Element *IgnoreChecksums* zu setzen. Sie sollten dann entsprechend misstrauisch bezüglich des Inhalts der Datenbank sein, wenn Sie ein solches Backup mittels Restore wieder einspielen.
  - *IgnoreLimbo*: Transaktionen in der Schwebe können durch Systemfehler oder Fehler beim 2-Phasen-Commit auftreten. Um solche Transaktionen und die daraus resultierenden Datenänderungen zu ignorieren, setzen Sie das Element *IgnoreLimbo*.
  - *MetadataOnly*: Es werden nur die Metadaten gesichert.
  - *NoGarbageCollection*: Wird ein Datensatz mit DELETE gelöscht oder mit UPDATE geändert, dann kann die alte Version nicht gleich von der Festplatte entfernt werden, weil ja noch laufende Transaktionen diese Daten benötigen könnten. Damit diese veralteten Datensätze nicht ewig auf der Platte ver-

bleiben, führt der Server hin und wieder eine *garbage collection*, eine »Müllabfuhr«, durch und schaut, was nun entfernt werden kann. Für gewöhnlich ist es sinnvoll, dies auch im Rahmen eines Backups durchzuführen, damit veraltete Datensätze nicht mitgesichert werden. Sie sollten dieses Element normalerweise nicht setzen.

- *NonTransportable*: Wenn Sie das transportable Format verwenden, dann kann das Backup auch auf einem Server eingespielt werden, der unter einem anderen Betriebssystem läuft – dies ist auch die Methode, um die Daten von einem Server zu einem anderen zu übertragen. Wenn darauf verzichtet werden kann, dann können Sie das Element *NonTransportable* setzen.

10.3.4 TIBRestoreService

Um ein Backup wieder einzuspielen, wird die Komponente *TIBRestoreService* verwendet.

- BackupFile, DatabaseName (Eigenschaften, veröffentlicht), ServiceStart (Methode)

```
property BackupFile: [Index: Integer] String;
property DatabaseName: TStrings;
procedure ServiceStart;
```

Die Sicherungsdatei(en) werden mit BackupFile spezifiziert, die Namen der Datenbankdateien mit DatabaseName, *ServiceStart* spielt dann die Sicherungsdatei ein.

Die Eigenschaft *DatabaseName* ist als Werteliste aufgebaut und verbindet die Dateinamen der Datenbankdateien mit der Anzahl der Datenbankseiten:

```
d:\data\file_1.ib=10000
e:\data\file_2.ib=10000
f:\data\file_3.ib=
```

- PageSize (Eigenschaft, veröffentlicht)

```
property PageSize: Integer;
```

Die Seitengröße in Byte wird mit der Eigenschaft *PageSize* eingestellt und sollte im Regelfall den Wert 4096 haben.

- Verbose (Eigenschaft, veröffentlicht), Eof (Eigenschaft, öffentlich, nur Lesen), GetNextLine (Methode)

```
property Verbose: Boolean;
property Eof: Boolean;
function GetNextLine: String;
```

Soll ein Protokoll des Vorgangs erstellt werden, dann ist *Verbose* auf *true* zu setzen. *GetNextLine* wird dann wiederholt so oft aufgerufen, bis *Eof* den Wert *true* hat. Ein Beispiel ist bei *TIBBackupService* zu finden.

- Options (Eigenschaft, veröffentlicht)

```
property Options: TRestoreOptions;
```

Auch bei *TIBRestoreService* lassen sich einige Optionen einstellen:

- *DeactiveIndices*: Über vorübergehend deaktivierte eindeutige Indizes (Schlüssel) können Dubletten in die Datenbank kommen. Versucht man nun, eine solche Datenbank wiederherzustellen, dann scheitert das Einfügen der Dubletten am eindeutigen Index. Dies kann man mit *DeactiveIndices* verhindern. Aus Performance-Gründen wäre es keine schlechte Idee, alle Indizes im Anschluss an das Restore neu aufzubauen. Wenn das ohnehin erfolgen soll, dann besteht auch kein Anlass, den Server während des Restores mit der Wartung der Indizes zu belasten.
- *NoShadow*: Shadows werden bei der Definition der Datenbank mit angelegt. Sollen Sie nicht mit wiederhergestellt werden, dann ist das Element *NoShadow* zu setzen.
- *NoValidityCheck*: Wenn Gültigkeitsprüfungen während des Betriebs einer Datenbank hinzugefügt werden, dann müssen davor eingefügte Datensätze nicht zwingend diesen Bedingungen entsprechen. Weil aber bei einem Backup zunächst die Metadaten erstellt und dann die Daten eingefügt werden, kommt es hier zu Problemen. Um diese zu umgehen, setzt man das Element *NoValidityCheck*.
- *OneRelationAtATime*: Normalerweise werden erst die Metadaten und dann die Daten wiederhergestellt. Möchte man eine Tabelle nach der anderen vollständig wiederherstellen, dann ist zu *OneRelationAtATime* setzen.
- *Replace*: Eine bestehende Datenbankdatei wird erst dann ersetzt, wenn das *Replace* den Wert *true* hat, und auch nur dann, wenn exklusiver Zugriff auf sie besteht.
- *CreateNewDB*: Soll eine neue Datenbankdatei erstellt werden, dann ist das Element *CreateNewDB* zu setzen.
- *UseAllSpace*: Wird *UseAllSpace* gesetzt, dann werden die Datenbankseiten zu 100 % und nicht zu 80 % gefüllt. Dies beschleunigt die Lese-Vorgänge, verlangsamt aber Datenänderungen, so dass diese Vorgehensweise vor allem bei Read-Only-Datenbanken zu empfehlen ist.

10.3.5 TIBValidationService

Zur Überprüfung der Datenbank wird *TIBValidationService* verwendet.

- DatabaseName (Eigenschaft, veröffentlicht), ServiceStart (Methode)

```
property DatabaseName: String;
procedure ServiceStart;
```

Mit *DatabaseName* wird die Datenbank spezifiziert, ausgeführt wird der Dienst mit *ServiceStart*.

- Options (Eigenschaft, veröffentlicht)

```
property Options: TValidateOptions;
```

Die folgenden Optionen könenn gesetzt werden:

- *LimboTransactions*: Es wird eine Liste aller schwebenden Transaktionen zurückgegeben.
- *CheckDB*: Die Datenbank wird geprüft, aber nicht repariert. Setzen Sie diese Option zusammen mit *ValidateDB*.
- *IgnoreChecksum*: Ignoriert alle Quersummenfehler.
- *KillShadows*: Alle fehlerhaften Shadows werden gelöscht.
- *MendDB*: Eine defekte Datenbank wird so aufbereitet, dass ein Backup gezogen werden kann.
- *SweepDB*: Die Datenbank wird sequenziell gelesen, dabei werden alle veralteten Datensatzversionen gelöscht.
- *ValidateDB*: Die Datenbankstruktur wird überprüft.
- *ValidateFull*: Es werden auch alle Datensatzfragmente überprüft.

- GlobalAction (Eigenschaft, veröffentlicht)

```
property GlobalAction: TTransactionGlobalAction;
```

Mit GlobalAction wird spezifiziert, wie mit schwebenden Mehr-Datenbank-Transaktionen zu verfahren ist:

- *CommitGlobal*: Die Transaktionen werden für die aktuelle Datenbank abgeschlossen.
- *RollbackGlobal*: Die Transaktionen werden für die aktuelle Datenbank zurückgesetzt.
- *RecoverTwoPhaseGlobal*: Für die Transaktionen wird ein zweiphasiges Commit durchgeführt: Wenn sich die Transaktion auch in den anderen Datenbanken bestätigen lässt, dann wird sie mit *Commit* beendet, andernfalls mit *Rollback* zurückgenommen.
- *NoGlobalAction*: Es wird weiter keine Aktion durchgeführt.

- Eof (Eigenschaft, öffentlich, nur Lesen), GetNextLine (Methode)

```
property Eof: Boolean;
function GetNextLine: String;
```

Um das Ergebnis der Prüfung anzuzeigen, wird *GetNextLine* so oft aufgerufen, bis *Eof* den Wert *true* hat. Ein Beispiel ist bei *TIBBackupService* zu finden.

10.3.6 TIBStatisticalService

Die Komponente *TIBStatisticalService* wird zum Ermitteln statistischer Daten verwendet – Details finden Sie in Kapitel 7.3.2.

- DatabaseName (Eigenschaft, veröffentlicht), ServiceStart (Methode)

```
property DatabaseName: String;
procedure ServiceStart;
```

  Mit *DatabaseName* wird die Datenbank spezifiziert, ausgeführt wird der Dienst mit *ServiceStart*.

- Options (Eigenschaft, veröffentlicht)

```
property Options: TStatOptions;
```

- Mit *Options* wird spezifiziert, welche statistischen Daten angezeigt werden sollen:
  - *DataPages*: Header, Protokoll, Datenseiten
  - *DbLog*: Header, Protokoll
  - *HeaderPages*: nur Header
  - *IndexPages*: Header, Protokoll, Datenseiten, Indizes
  - *SystemRelations*: Systemtabellen
- Eof (Eigenschaft, öffentlich, nur Lesen), GetNextLine (Methode)

```
property Eof: Boolean;
function GetNextLine: String;
```

  Um die Statistik anzuzeigen, wird *GetNextLine* so oft aufgerufen, bis *Eof* den Wert *true* hat. Ein Beispiel ist bei *TIBBackupService* zu finden.

10.3.7 TIBLogService

Über die Komponente TIBLogService kann auf das *interbase.log* zugegriffen werden.

- Eof (Eigenschaft, öffentlich, nur Lesen), GetNextLine (Methode)

```
property Eof: Boolean;
function GetNextLine: String;
```

Um den Inhalt des Logfiles anzuzeigen, wird *GetNextLine* so oft aufgerufen, bis *Eof* den Wert *true* hat. Ein Beispiel ist bei *TIBBackupService* zu finden.

10.3.8 TIBSecurityService

Die Benutzerverwaltung können Sie mit der Komponente *TIBSecurityService* realisieren. Wenn Sie das Handling mit dieser Komponente ein wenig seltsam finden, dann würde der Autor durchaus Ihre Meinung teilen ...

- UserInfo, UserInfoCount (Eigenschaften, öffentlich, nur Lesen), DisplayUser (Methode)

```
property UserInfo [Index: Integer]: TUserInfo;
property UserInfoCount: Integer;
procedure DisplayUser (UserName: String);
```

  Auf die einzelnen Datensätze der Benutzerdatenbank erhalten Sie mit der Array-Eigenschaft *UserInfo* einen Lesezugriff. Die Anzahl der Benutzer und somit der Array-Felder erhalten Sie mit *UserInfoCount*.

  Wenn Sie *DisplayUser* mit einem leeren String als Parameter aufrufen, dann erhalten Sie alle Benutzer angezeigt, übergeben Sie einen Benutzernamen, dann erhalten Sie die betreffenden Daten als erstes und einziges Array-Feld.

  Über den Record *TUserInfo* können Sie auf die relevanten Daten eines Benutzers zugreifen.

```
TUserInfo = record
  UserName: string;
  FirstName: string;
  MiddleName: string;
  LastName: string;
  GroupID: Integer;
  UserID: Integer;
end;
```

- AddUser, DeleteUser, ModifyUser (Methoden)

```
procedure AddUser;
procedure DeleteUser;
procedure ModifyUser;
```

  Mit diesen Methoden können User angelegt und gelöscht sowie die Benutzerdaten geändert werden. Sollen Benutzer gelöscht oder geändert werden, so ist zunächst mit *UserName* der Benutzer zu setzen.

- UserName, Password, FirstName, MiddleName, LastName (Eigenschaften, veröffentlicht)

```
property UserName: String;
property Password: String;
property FirstName: String;
property MiddleName: String;
property LastName: String;
```

Um einen neuen Benutzer anzulegen, setzen Sie diese Eigenschaften – zumindest jedoch *UserName* und *Password* – und rufen dann *AddUser* auf.

10.3.9 TIBServerProperties

Um Server-Eigenschaften zu ermitteln, verwenden Sie *TIBServerProperties*. Näheres in der Online-Hilfe, dort gibt es auch ausführliche Beispiele.

10.3.10 TIBLicensingService

Mit TIBLicensingService können Sie Server-Lizenzen hinzufügen und entfernen.

- ID, Key, Action (Eigenschaften, veröffentlicht)

  ```
  property ID: String;
  property Key: String;
  property Action: (LicenseAdd, LicenseRemove);
  ```

  Mit *ID* und *Key* werden die Lizenz-Informationen angegeben, mit *Action*, ob die Lizenz hinzugefügt oder entfernt werden soll.

- ServiceStart, AddLicense, RemoveLicense (Methoden)

  ```
  procedure ServiceStart;
  procedure AddLicense;
  procedure RemoveLicense;
  ```

  Um die Lizenz hinzuzufügen, ruft man *ServiceStart* oder *AddLicense* auf – analog ist beim Entfernen von Lizenzen zu verfahren.

10.3.11 TIBInstall und TIBUnInstall

Mit diesen Komponenten kann man InterBase installieren und wieder deinstallieren. Auch hier möchte ich Sie auf die Online-Hilfe verweisen.

11 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*, *DB2* und seit Delphi 7 auch auf *Informix* und *MS SQL* zugreifen.

Damit unterstützt dbExpress alle relevanten SQL-Server. Im Gegensatz zur BDE ist *dbExpress* jedoch für die 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.

Warum dbExpress?

Wenn es mit IBX, IBO und FIBPlus gleich drei Komponentenfamilien gibt, die für InterBase geradezu »maßgeschneidert« wurden, bei denen sich viel mehr Optionen konfigurieren lassen als mit dbExpress und die in der Programmierung auch noch einfacher sind, welche Motivation sollte es geben, sich mit dbExpress zu beschäftigen?

Der Grund dafür lautet zunächst »alle relevanten SQL-Server«. Wenn Sie ein Projekt mit speziellen InterBase-Komponenten erstellt haben, dann sind Sie auf InterBase festgelegt. Soll nun ein anderer Datenbankserver verwendet werden, dann kann der Aufwand der Umstellung stark in Richtung Neuentwicklung tendieren. (Das hängt natürlich davon ab, wie stark Sie spezielle Features nutzen.)

Nun sollte es relativ selten sein, dass während eines Projekts das Datenbanksystem gewechselt wird. Es wird aber auch der Versuch unterbunden, zumindest aber deutlich erschwert, Module aus einem Projekt in einem anderen zu verwenden. Sie haben beispielsweise eine Adressenverwaltung mit IBX erstellt, das nächste Projekt läuft auf Oracle, dann haben Sie erst mal ein Problem.

Der andere Grund ist, dass in nicht wenigen Fällen dbExpress bei der Geschwindigkeit die Nase vorn hat – der Verzicht auf viele Features vermeidet halt auch viel unnötigen Ballast. Ob diese Gründe einen gewissen Mehraufwand bei der Programmierung rechtfertigen, dürfen Sie gerne selbst entscheiden – ich selbst tendiere zunehmend zu dbExpress.

11.1 Mit dbExpress arbeiten

dbExpress stellt unidirektionale, nicht editierbare Datenmengen zur Verfügung – für Entwickler, die bislang mit anderen Komponenten gearbeitet haben, ist dies eine ziemliche Umstellung. Wer jedoch bislang direkt mit der Client-DLL eines Datenbanksystems gearbeitet hat, der wird sich gleich »zu Hause« fühlen. Lassen Sie uns nun ansehen, was man mit dbExpress tun darf und was nicht.

11.1.1 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. Unter dem Verbindungsnamen *IBLocal* wäre bereits eine Verbindung zu *employee.gdb* eingerichtet, aber um das Erstellen einer neuen Verbindung auch gleich »mitzunehmen«, wollen wir auf den Button mit dem Plus links oben klicken:

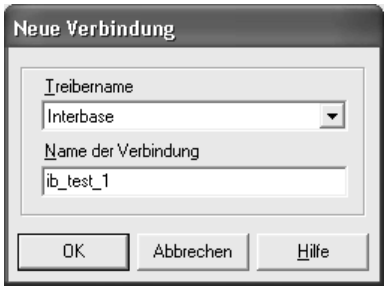


Abbildung 11.1: Neue Verbindung erstellen

Hier wählen wir nun *Interbase* als Treiber und geben einen bislang nicht genutzten Namen ein.

Nun stellen wir unter *Database* den Dateinamen der Datenbank inklusive Pfad ein. Der Benutzername und das Passwort werden erfreulicherweise bereits richtig gesetzt. Wenn wir nicht explizit eine Transaktion starten, dann wird der Isolationsgrad READ COMMITED verwendet und *CommitRetain* bleibt auf *false*, auch damit liegen wir in den meisten Fällen nicht verkehrt. Bei der Datenbank *employee.gdb* ist es auch angebracht, auf einen speziellen Sprachtreiber zu verzichten, *ServerCharSet* lassen wir somit leer und auch mit *SQLDialect* liegen wir in diesem Fall richtig.

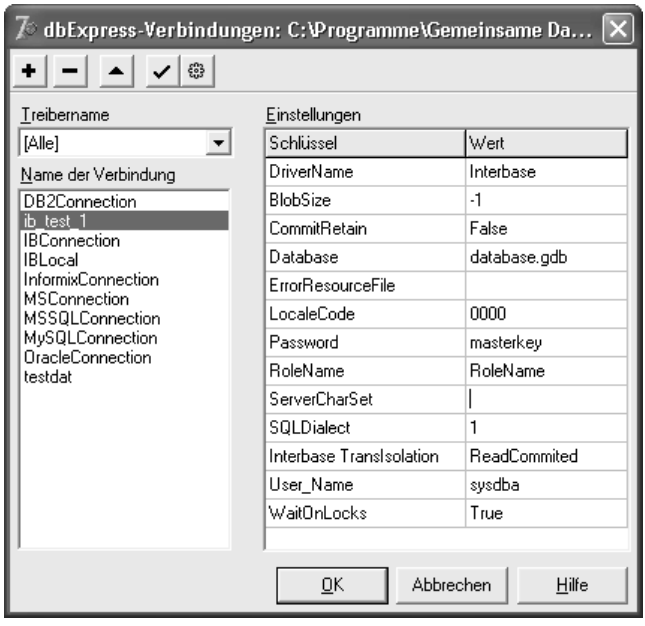


Abbildung 11.2: Die Verbindungsparameter

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, darüber hinaus die BDE-Komponente *TQuery*.

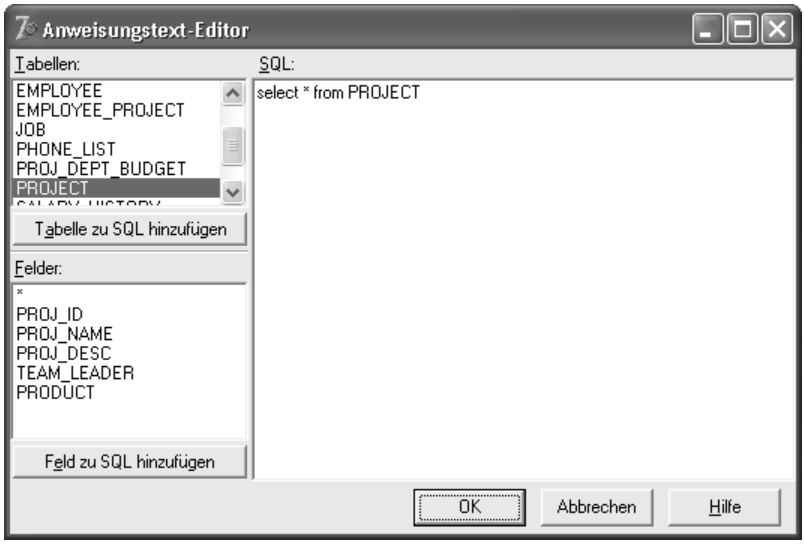


Abbildung 11.3: Eine SQL-Anweisung erstellen

```
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 wurden 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-Vor-

gang, 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 27168 µs, das Scrollen 115 µs. Würde *TSQLQuery* anstelle von *TSQLDataSet* verwendet, dann würde das Öffnen bei 27397 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 4323 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 (26862 statt 27168 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 im SELECT-Statement das Jokerzeichen verwenden – der Beschleunigungseffekt ist in etwa derselbe: Statt 26862 µs dauert das Öffnen der Abfrage nun nur noch 2884 µs.

Bei *TQuery* fällt die Beschleunigung deutlich geringer aus, die Abfrage ist in 4273 µs geöffnet.

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

11.1.2 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 *TClient-DataSet*-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 11091 µs (statt 3007 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:

Abbildung 11.4: Fehlermeldung beim Scrollen in die verkehrte Richtung

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.

11.1.3 TSQLClientDataSet – die »Krücke« aus Delphi 6

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 in Delphi 6 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 (49636 µs), das Scrollen auch (1861 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
```

11.1.4 TSimpleDataSet – keine wirkliche Besserung in Delphi 7

Dass Ihnen mit *TSQLClientDataSet* kein wirkliches Glanzlicht gelungen ist, haben dann wohl auch die Programmierer von Borland mitbekommen. In Delphi 7 ist *TSQLClientDataSet* sang- und klanglos verschwunden, dafür gibt es jetzt *TSimpleDataSet*.

TSimpleDataSet beinhaltet auch eine interne *TSQLConnection*-Komponente, so dass damit bis zu vier Komponenten ersetzt werden können. Können deshalb, weil man wohl bei jeder anspruchsvolleren Datenbankanwendung mehrere Datenzugriffskomponenten hat und diese dann tunlichst über eine eigene *TSQLConnection*-Instanz führt.

Es bleibt die Kette von *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet*. Für beide Datenmengenkomponenten lassen sich nun separat die persistenten TField-Instanzen anlegen – es hilft nur nicht mehr viel, weil die Eigenschaft *UpdateMode* nicht veröffentlicht ist.

Was man ebenfalls vermisst, ist die Eigenschaft *Options* des Providers und somit die Möglichkeit, auf BLOBs so lange zu verzichten, bis sie explizit geladen werden. Natürlich kann man bei der Formulierung des SQL-Statements auf die entsprechenden Spalten verzichten – aber mal eben nachladen kann man sie halt dann nicht. (Kommen Sie aber jetzt bitte nicht auf die Idee, mittels einer eigenen Datenzugriffskomponente explizit auf die BLOB-Spalte zuzugreifen, sondern bauen Sie lieber gleich die »Datenmengenkette« diskret aus *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* auf.)

Der einzige Lichtblick: Man hat dieser Komponente die Neugierde auf die Metadaten abgewöhnt. Und ein Trost: Man muss das ja nicht einsetzen, so viel länger dauert es auch nicht, drei Komponenten miteinander zu verknüpfen ...

11.1.5 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 einen Generatorwert zur Verfügung stellt:

```
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* und GetMetedata auf *false* 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 PROCEDURES per SQL-Statement zugreifen, sei es mit *TSQLDataSet*, sei es mit *TQuery*.

11.1.6 Erstellen einer Master-Detail-Verknüpfung

Beim Erstellen einer Master-Detail-Verknüpfung gibt es ein paar Details zu beachten, die wir uns nun etwas genauer ansehen wollen:

Master-Detail mit TSQLDataSet

Um rein mit *TSQLDataSet* eine Master-Detail-Verknüpfung zu erstellen, gehen Sie wie folgt vor:

- Die erste Datenmengenkomponente erhält eine SQL-Anweisung zum Zugriff auf die Master-Datenmenge.
- An diese erste Datenmengenkomponente wird eine *TDataSource*-Instanz gehängt.
- Die zweite Datenmengenkomponente erhält eine SQL-Anweisung mit WHERE-Klausel und Parameter. Über diese WHERE-Klausel wird die Master-Detail-Verknüpfung hergestellt.

```
select * from T_POSTEN
  where BESTELLUNG = :ID
```

- Über die Eigenschaft *DataSource* wird nun die zweite Datenmengenkomponente mit der ersten verknüpft. Sobald nun mit der ersten *TSQLDataSet*-Instanz zu einem anderen Datensatz navigiert wird, wird die zweite entsprechend aktualisiert.

Bei *TSQLQuery* geht man entsprechend vor, bei *TSQLTable* erfolgt die Verbindung über *MasterSource* und *MasterFields*.

TClientDataSet in der Detail-Datenmenge

Oft sind Formulare so aufgebaut, dass ein Master-Datensatz und in einem DBGrid alle Detail-Datensätze angezeigt werden. Der Einsatz von *TDBGrid* setzt jedoch die Verwendung von *TClientDataSet* voraus, damit die Datensätze zwischengespeichert werden.

Hängt man nun an die zweite *TSQLDataSet*-Instanz einen DataSetProvider und ein ClientDataSet, dann funktioniert die automatische Aktualisierung der Detail-Datenmenge nicht mehr: In dem Moment, in dem die *TClientDataSet*-Instanz geöffnet wird, ruft sie alle Datensätze ab und schließt dann die Detail-Datenmenge – die Datensätze sind ja nun lokal gespeichert.

Um das zu vermeiden, gibt es mehrere Möglichkeiten. Die einfachste geht folgendermaßen:

- Setzen Sie die Eigenschaft *MasterSource* der *TClientDataSet*-Instanz auf *DataSource1*.
- Rufen Sie über die Eigenschaft *MasterFields* den Feldverbindungs-Designer auf und erstellen auch hier eine Verknüpfung von *Bestellung* zu *ID*.

Wird nun in der Master-Datenmenge gescrollt, dann erkennt die Client-Datenmenge, dass sie neue Datensätze braucht, öffnet die Detail-Datenmenge und ruft dann die neuen Datensätze ab.

Abbildung 11.5: Erstellen einer Master-Detail-Verknüpfung

Kompletter Datenbestand in der Client-Datenmenge

Könnte man nun auf die WHERE-Klausel in der Detail-Datenmengenkomponente verzichten? Ja, man könnte. Die Folge wäre, dass alle Datensätze in die Client-Datenmenge geladen werden. Dadurch würde das Öffnen der Client-Datenmenge deutlich länger dauern, das Scrollen in der Master-Datenmenge würde jedoch beschleunigt, da ja dann keine SQL-Abfrage mehr auf dem Server ausgeführt werden muss, sondern alle Daten bereits auf dem Client liegen.

In den meisten Fällen ist das wenig sinnvoll. Wenn jedoch ohnehin alle Detail-Datensätze auf den Client geladen werden müssen (»Report-Druck«), dann kann es sinnvoll sein, dies »in einem Rutsch« zu erledigen, anstatt Server mit tausenden von SQL-Statements zu nerven.

11.2 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.

11.2.1 TSQLConnection

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

Verbindung zur Datenbank

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

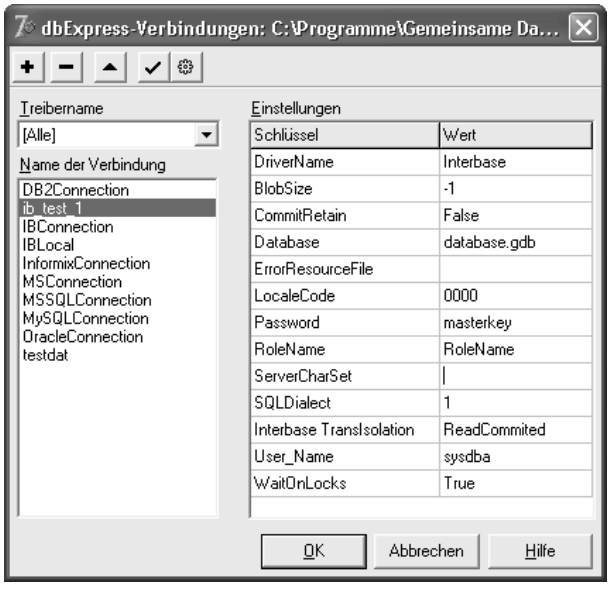


Abbildung 11.6: Der Verbindungseditor

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. Um einen Zeichensatz zu verwenden, muss *ServerCharSet* gesetzt werden. Darüber hinaus kann der *SQLDialect* eingestellt 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 zu 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, MultipleTransactionsSupported (Eigenschaften, öffentlich, nur lesen)

```
property InTransaction: Boolean;
property TransactionsSupported: LongBool;
property MultipleTransactionsSupported: 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. Um zu erfahren, ob das Datenbanksystem mehrere (veschachtelte oder überlappende) Transaktionen unterstützt, kann man *MultipleTransactionsSupported* verwenden.

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 sie verworfen.

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

11.2.2 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 der Online-Hilfe oder jedem brauchbaren Delphi-Buch beschrieben.

- 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 nicht zu 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 der Online-Hilfe oder jedem brauchbaren Delphi-Buch beschrieben.

- 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.

- MaxBlobSize

```
property MaxBlobSize: Integer;
```

Mit *MaxBlobSize* sollte man festlegen können, bis zu welcher Größe BLOBs maximal abgerufen wird. Zumindest im Zusammenspiel mit InterBase habe ich jedoch keine Wirkung feststellen können.

- 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);
```

- IndexDefs (Eigenschaft, öffentlich)

```
property IndexDefs: TIndexDefs;
```

Mit *IndexDefs* kann man ermitteln, welche Indizes für eine Tabelle definiert sind. Das folgende Beispiel listet die Indexnamen und in Klammern die daran beteiligten Spalten auf:

```
var
  i: integer;
begin
  with SQLDataSet1.IndexDefs do
  begin
    for i := 0 to Count - 1
      do ListBox1.Items.Add(Items[i].Name
        + ' (' + Items[i].Fields + ')');
  end; {with SQLDataSet1.IndexDefs do}
```

- 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);
  SetState(dsBrowse);
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. Darüber hinaus 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 zu 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.

11.2.3 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:

Abbildung 11.7: Eigenschaftseditor von TSQLDataSet

  - 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.

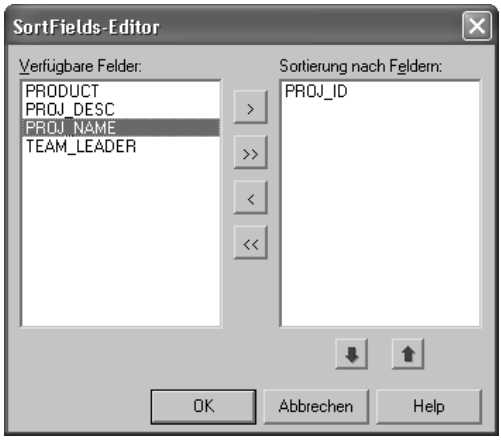


Abbildung 11.8: Sortieren der Spalten bei ctTable

- 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.

11.2.4 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.

11.2.5 TSQLTable

Die Komponente *TSQLTable* dient dem Zugriff auf eine einzelne Tabelle. Die Verwendung ist an die BDE-Komponente *TTable* 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*.
- GetIndexNames

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

  Mit *GetIndexNames* lassen sich die Namen aller Indizes, die in den entsprechenden Tabellen definiert sind, in einer Stringliste auflisten.
- DeleteRecords (Methode)

  ```
  procedure DeleteRecords;
  ```

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

11.2.6 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.

11.2.7 TSQLClientDataSet

Die Komponente *TSQLClientDataSet* ist der erste Versuch, *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* zu einer Komponente zusammenzufügen. Dieser Versuch kann nicht als besonders gelungen bewertet werden, deshalb ist diese Komponente auch nur in Delphi 6 vorhanden.

Zur Problematik dieser Komponente wurde in Kapitel µ.2.3 schon genug geschrieben, ich möchte das an dieser Stelle nicht wiederholen, sondern Ihnen lediglich den Rat geben, die Datenmengenkette aus *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* einzeln aufzubauen.

11.2.8 TSimpleDataSet

Die Komponente *TSimpleDataSet* ist der Versuch aus Delphi 7, *TSQLDataSet*, *TDataSetProvider* und *TClientDataSet* zu einer Komponente zusammenzufügen. Ich möchte ja gar nicht bestreiten, dass es sich um einen Schritt in die richtige Richtung handelt, aber nach wie vor würde ich die Datenmengenkette diskret aufbauen.

Eine Referenz dieser Komponente möchte ich mir an dieser Stellen verkneifen, zumal wir dabei detailliert auf *TClientDataSet* eingehen müssten – in einem Buch über InterBase kann man die Seiten sinnvoller füllen.

Zur Information nur so viel: *TSimpleDataSet* bringt eine interne Verbindungskomponente mit, die so lange verwendet wird, bis Sie eine andere *TSQLConnection*-Instanz auswählen. Im Kontextmenü finden Sie den Eintrag VERBINDUNG BEARBEITEN, mit dem Sie den aus *TSQLConnection* bekannten Verbindungseditor aufrufen. Auf die enthaltene *TSQLDataSet*-Instanz greifen Sie über die Eigenschaft *DataSet* zu.

11.2.9 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.)

11.2.10 Installation von dbExpress-Anwendungen

Wenn man berücksichtigt, dass man auf verschiedene Datenbanksysteme zugreifen kann, zeichnet sich *dbExpress* durch eine vergleichsweise 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 auf meinem Rechner zu einem *Internen Fehler* bei der Kompilierung.

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

12 USER DEFINED FUNCTIONS

InterBase implementiert lediglich die Standard-SQL-Funktionen wie beispielsweise SUM oder AVG. Andere Datenbanksysteme sind hier großzügiger ausgestattet. Dagegen bietet InterBase die Möglichkeit, Funktionen aus externen Bibliotheken zu importieren. Diese DLLs können – eine entsprechende Entwicklungsumgebung, wie beispielsweise Delphi, vorausgesetzt – vom Anwender selbst erstellt werden, so dass alle erwünschten Funktionen nachgerüstet werden können.

Wir wollen uns zunächst ansehen, wie solche Funktionen verwendet werden und welche Funktionen die Standard-Bibliothek *ib_udf.dll* bietet. Anschließend werden wir kurz besprechen, wie man selbst solche Bibliotheken erstellt.

12.1 DECLARE EXTERNAL FUNCTION

Im Verzeichnis *InterBase\UDF* gibt es die Datei *ib_udf.dll*. Im Gegensatz zu früheren InterBase-Versionen braucht diese Datei weder in das Windows-Verzeichnis noch in das Verzeichnis *InterBase\bin* kopiert zu werden.

Um eine Funktion aus einer externen Bibliothek verwenden zu können, muss sie erst einmal deklariert werden.

```
DECLARE EXTERNAL FUNCTION substr
    CSTRING(80), SMALLINT, SMALLINT
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT "IB_UDF_substr"
    MODULE_NAME "ib_udf.dll";
```

Die Deklaration beginnt mit der Anweisung DECLARE EXTERNAL FUNCTION, welcher der in InterBase verwendete Name der Funktion folgt. Anschließend folgt die Liste der an die Funktion zu übergebenden Parameter, im Beispiel ein String und zwei Ganzzahlen. (Der Typ CSTRING entspricht dem Typen VARCHAR.)

Nach dem Schlüsselwort RETURNS folgt der Typ des von der Funktion zurückzugebenden Wertes. Diesem Wert werden manchmal die Optionen BY VALUE oder FREE_IT angehängt. Was es damit auf sich hat, ist nur für den Programmierer interessant. Halten Sie sich exakt an das, was der Programmierer hier vorgibt.

Nach dem Schlüsselwort ENTRY_POINT folgt der Funktionsname in der Bibliothek. Dieser Name ist durch die Bibliothek vorgegeben, während der in InterBase verwendete Funktionsname durchaus vom Benutzer abgeändert werden kann. Nach dem Schlüsselwort MODULE_NAME wird der Dateiname der DLL genannt.

Die Funktion wird mit DECLARE EXTERNAL FUNCTION nur deklariert. InterBase prüft zu diesem Zeitpunkt nicht, ob eine passende Funktion überhaupt verfügbar ist. Wenn dies nicht der Fall ist, wird jedoch beim Versuch, diese Funktion zu verwenden, ein Fehler auftreten.

subStr

Die Funktion *substr(s, b, e)* liefert die Zeichen des Strings *s* von Position *b* bis Position *e*.

```
SELECT
    id,
    nachname,
    substr(nachname, 3, 5)
  FROM t_mitarbeiter
```

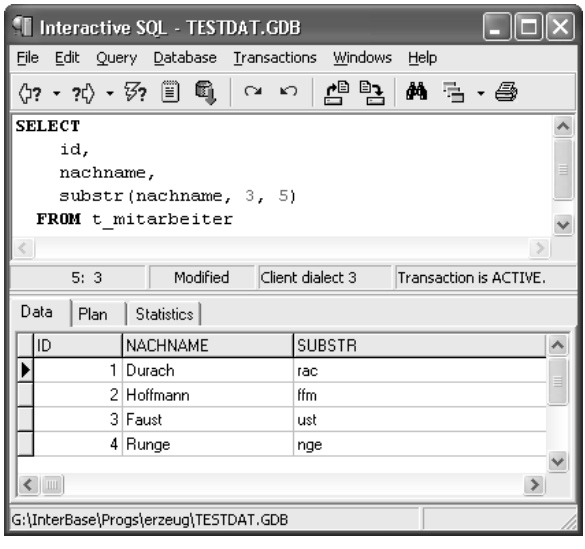


Abbildung 12.1: Verwendung einer USER DEFINED FUNCTION

Eine Funktion entfernen

Mit DROP EXTERNAL FUNCTION kann eine Funktion entfernt werden.

```
DROP EXTERNAL FUNCTION substr
```

Die Deklaration einer Funktion kann nicht geändert werden, eine Anweisung ALTER EXTERNAL FUNCTION existiert nicht.

12.2 Die Bibliothek ib_udf.dll

Ab Version 5 liegt InterBase die Bibliothek *ib_udf.dll* bei, die eine Reihe von Funktionen zur Verfügung stellt, die vom Anwender dann nur noch eingebunden werden müssen. Dafür gibt es im Verzeichnis *InterBase\Examples\UDF* die Datei *ib_udf.sql*.

Stringbearbeitung

Im ersten Abschnitt wollen wir die Funktionen besprechen, die zur Bearbeitung von Zeichenketten dienen. Diejenigen Funktionen, deren Parameter als CSTRING(80) definiert sind, können tatsächlich bis zu 32767 Zeichen übernehmen und bearbeiten.

ascii_char

Die Funktion *ascii_char* wandelt eine Zahl in das entsprechende ASCII-Zeichen um. Das Ergebnis von *ascii_char(70)* wäre *F*.

```
DECLARE EXTERNAL FUNCTION ascii_char
    INTEGER
    RETURNS CHAR(1)
    ENTRY_POINT "IB_UDF_ascii_char"
    MODULE_NAME "ib_udf.dll";
```

ascii_val

Die Funktion *ascii_val* ermittelt die ASCII-Nummer eines Zeichens. Das Ergebnis von *ascii_val(»F«)* wäre 70.

```
DECLARE EXTERNAL FUNCTION ascii_val
    CHAR(1)
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "IB_UDF_ascii_val"
    MODULE_NAME "ib_udf.dll";
```

lower

Mit der Funktion *lower* wandeln Sie alle Zeichen in Kleinbuchstaben um. Um alle Zeichen in Großbuchstaben umzuwandeln, verwenden Sie die SQL-Funktion UPPER.

```
DECLARE EXTERNAL FUNCTION lower
    CSTRING(80)
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT "IB_UDF_lower"
    MODULE_NAME "ib_udf.dll";
```

ltrim

Die Funktion *ltrim* entfernt alle führenden Leerzeichen aus einem String. Um abschließende Leerzeichen zu entfernen, verwenden Sie die Funktion *rtrim*.

```
DECLARE EXTERNAL FUNCTION ltrim
    CSTRING(80)
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT "IB_UDF_ltrim" MODULE_NAME "ib_udf.dll";
```

rtrim

Die Funktion *rtrim* entfernt alle abschließenden Leerzeichen aus einem String.

```
DECLARE EXTERNAL FUNCTION rtrim
    CSTRING(80)
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT "IB_UDF_rtrim"
    MODULE_NAME "ib_udf.dll";
```

strlen

Die Funktion *strlen* ermittelt die Länge eines Strings.

```
DECLARE EXTERNAL FUNCTION strlen
    CSTRING(32767)
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "IB_UDF_strlen"
    MODULE_NAME "ib_udf.dll";
```

substr

Die Funktion *substr(s, b, e)* liefert die Zeichen des Strings *s* von Position *b* bis Position *e*. (Beispiel: *substr(»abcdefgh«, 3, 6)* ergibt *cdef*.)

```
DECLARE EXTERNAL FUNCTION substr
    CSTRING(80), SMALLINT, SMALLINT
    RETURNS CSTRING(80) FREE_IT
    ENTRY_POINT "IB_UDF_substr"
    MODULE_NAME "ib_udf.dll";
```

12.2.1 Mathematische Funktionen

Im folgenden Abschnitt sollen alle mathematischen Funktionen mit Ausnahme der trigonometrischen Funktionen aufgeführt werden. Die trigonometrischen Funktionen werden in Abschnitt 7.2.3 behandelt.

abs

Die Funktion *abs* ermittelt den Betrag einer Zahl. (Für Nicht-Mathematiker: Der Betrag von 3 ist 3, der Betrag von -4 ist 4, der Betrag von -123,45 ist 123,45.)

```
DECLARE EXTERNAL FUNCTION abs
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_abs"
    MODULE_NAME "ib_udf.dll";
```

ceiling

Die Funktion rundet auf die nächste Ganzzahl auf. (Aus 3,45 wird 4, aus -3,45 wird -3.) Zum Abrunden verwenden Sie die Funktion *floor*.

```
DECLARE EXTERNAL FUNCTION ceiling
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_ceiling"
    MODULE_NAME "ib_udf.dll";
```

div

Die Funktion *div* berechnet das ganzzahlige Divisionsergebnis zweier Zahlen. (Beispiel *div(11, 4) ergibt 2*.) Den Divisionsrest berechnen Sie mit *mod*.

```
DECLARE EXTERNAL FUNCTION div
    INTEGER, INTEGER
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_div"
    MODULE_NAME "ib_udf";
```

floor

Die Funktion *floor* rundet eine Zahl auf die nächstkleinere Ganzzahl ab. (Aus 3.4 wird 3, aus -3.4 wird -4. Zur Erinnerung: Es sind Dezimalpunkte zu verwenden.) Zum Aufrunden verwenden Sie die Funktion *ceiling*.

```
DECLARE EXTERNAL FUNCTION floor
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_floor"
    MODULE_NAME "ib_udf.dll";
```

ln

Mit *ln* wird der natürliche Logarithmus einer Zahl berechnet.

```
DECLARE EXTERNAL FUNCTION ln
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_ln"
    MODULE_NAME "ib_udf.dll";
```

log

Mit *log* wird der Logarithmus des ersten Parameters zur Basis des zweiten Parameters berechnet.

```
DECLARE EXTERNAL FUNCTION log
    DOUBLE PRECISION, DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_log"
    MODULE_NAME "ib_udf.dll";
```

log10

Mit *log10* wird der Logarithmus zur Basis zehn berechnet.

```
DECLARE EXTERNAL FUNCTION log10
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_log10"
    MODULE_NAME "ib_udf.dll";
```

mod

Mit *mod* wird der Rest einer Ganzzahl-Divion ermittelt. (Beispiel *mod(11, 4)* ergibt 3.) Um das Ergebnis einer Ganzzahl-Division zu berechnen, verwenden Sie *div*.

```
DECLARE EXTERNAL FUNCTION mod
    INTEGER, INTEGER
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_mod"
    MODULE_NAME "ib_udf.dll";
```

pi

Die Funktion *pi* ermittelt die Kreiszahl *Pi* (3,141592654). Diese Funktion übernimmt keine Parameter.

```
DECLARE EXTERNAL FUNCTION pi
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_pi"
    MODULE_NAME "ib_udf.dll";
```

rand

Die Funktion *rand* generiert eine Zufallszahl zwischen null und eins.

```
DECLARE EXTERNAL FUNCTION rand
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_rand"
    MODULE_NAME "ib_udf.dll";
```

sign

Die Funktion *sign* ermittelt das Vorzeichen einer Zahl. Ist die Zahl positiv, lautet das Ergebnis 1, ist die Zahl negativ, lautet das Ergebnis -1. Das Ergebnis lautet 0, wenn die Zahl gleich null ist.

```
DECLARE EXTERNAL FUNCTION sign
    DOUBLE PRECISION
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "IB_UDF_sign"
    MODULE_NAME "ib_udf.dll";
```

sqrt

Die Funktion *sqrt* berechnet die Quadratwurzel einer Zahl.

```
DECLARE EXTERNAL FUNCTION sqrt
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_sqrt"
    MODULE_NAME "ib_udf.dll";
```

12.2.2 Trigonometrische Funktionen

Im folgenden Abschnitt werden die trigonometrischen Funktionen aufgeführt. Beachten Sie dabei, dass alle Gradangaben im Bogenmaß gemacht beziehungsweise erwartet werden.

acos

Die Funktion *acos* ermittelt den Arcus-Cosinus (Umkehrfunktion des Cosinus) einer Zahl.

```
DECLARE EXTERNAL FUNCTION acos
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_acos"
    MODULE_NAME "ib_udf.dll";
```

asin

Die Funktion *asin* ermittelt den Arcus-Sinus (Umkehrfunktion des Sinus) einer Zahl.

```
DECLARE EXTERNAL FUNCTION asin
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_asin"
    MODULE_NAME "ib_udf.dll";
```

atan

Die Funktion *atan* ermittelt den Arcus-Tangens (Umkehrfunktion des Tangens) einer Zahl.

```
DECLARE EXTERNAL FUNCTION atan
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_atan"
    MODULE_NAME "ib_udf.dll";
```

cos

Die Funktion *cos* berechnet den Cosinus einer Zahl.

```
DECLARE EXTERNAL FUNCTION cos
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_cos"
    MODULE_NAME "ib_udf.dll";
```

cosh

Die Funktion *cosh* berechnet den Cosinus hyperbolikus einer Zahl.

```
DECLARE EXTERNAL FUNCTION cosh
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_cosh"
    MODULE_NAME "ib_udf.dll";
```

cot

Die Funktion *cot* berechnet den Cotangens einer Zahl.

```
DECLARE EXTERNAL FUNCTION cot
DOUBLE PRECISION
RETURNS DOUBLE PRECISION BY VALUE
ENTRY_POINT "IB_UDF_cot" MODULE_NAME "ib_udf.dll";
```

sin

Die Funktion *sin* berechnet den Sinus einer Zahl.

```
DECLARE EXTERNAL FUNCTION sin
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_sin"
    MODULE_NAME "ib_udf.dll";
```

sinh

Die Funktion *sinh* berechnet den Sinus hyperbolikus einer Zahl.

```
DECLARE EXTERNAL FUNCTION sinh
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_sinh"
    MODULE_NAME "ib_udf.dll";
```

tan

Die Funktion *tan* ermittelt den Tangens einer Zahl.

```
DECLARE EXTERNAL FUNCTION tan
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_tan"
    MODULE_NAME "ib_udf.dll";
```

tanh

Die Funktion *tan* ermittelt den Tangens hyperbolikus einer Zahl.

```
DECLARE EXTERNAL FUNCTION tanh
    DOUBLE PRECISION
    RETURNS DOUBLE PRECISION BY VALUE
    ENTRY_POINT "IB_UDF_tanh"
    MODULE_NAME "ib_udf.dll";
```

12.2.3 Logische Verknüpfungen

Die folgenden Funktionen nehmen logische Verknüpfungen zweier Werte vor.

bin_and

Die Funktion *bin_and* nimmt eine logische UND-Verknüpfung von zwei Integer-Zahlen vor.

```
DECLARE EXTERNAL FUNCTION bin_and
    INTEGER, INTEGER
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "IB_UDF_bin_and"
    MODULE_NAME "ib_udf.dll";
```

bin_or

Die Funktion *bin_or* nimmt eine logische OR-Verknüpfung von zwei Integer-Zahlen vor.

```
DECLARE EXTERNAL FUNCTION bin_or
    INTEGER, INTEGER
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "IB_UDF_bin_or"
    MODULE_NAME "ib_udf.dll";
```

bin_xor

Die Funktion *bin_xor* nimmt eine logische XOR-Verknüpfung (Exclusive-Oder, Entweder-Oder) von zwei Integer-Zahlen vor.

```
DECLARE EXTERNAL FUNCTION bin_xor
    INTEGER, INTEGER
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "IB_UDF_bin_xor"
    MODULE_NAME "ib_udf.dll";
```

12.3 FreeUDFLib

Auf der Buch-CD finden Sie die Bibliothek *FreeUDFLib.dll* von *Gregory Deatz*. Interessant sind hier vor allem die in *ib_udf.dll* völlig vernachlässigten Datumsfunktionen. Im Folgenden soll ein Teil dieser Funktionen besprochen werden.

12.3.1 Datumsfunktionen

Die folgenden Funktionen führen Berechnungen mit dem Datum durch, extrahieren Teile davon oder wandeln es in eine andere Schreibweise um.

f_AddMonth

Die Funktion *f_addMonth(d, m)* addiert dem Datum *d* die Anzahl *m* Monate hinzu. Sollen Monate abgezogen werden, muss *m* negativ werden.

```
DECLARE EXTERNAL FUNCTION f_AddMonth
    DATE, INTEGER
    RETURNS DATE
    ENTRY_POINT "AddMonth"
    MODULE_NAME "FreeUDFLib.dll";
```

f_AddYear

Die Funktion *f_addYear(d, y)* addiert dem Datum *d* die Anzahl *y* Jahre hinzu. Sollen Jahre abgezogen werden, muss *y* negativ werden.

```
DECLARE EXTERNAL FUNCTION f_AddYear
    DATE, INTEGER
    RETURNS DATE
    ENTRY_POINT "AddYear"
    MODULE_NAME "FreeUDFLib.dll";
```

f_AgeInDays

Die Funktion *f_AgeInDays(b, e)* berechnet die Differenz von *e* minus *b* und gibt sie in Tagen an.

```
DECLARE EXTERNAL FUNCTION f_AgeInDays
    DATE, DATE
    RETURNS
    INTEGER BY VALUE
    ENTRY_POINT "AgeInDays"
    MODULE_NAME "FreeUDFLib.dll";
```

f_AgeInMonths

Die Funktion *f_AgeInMonths(b, e)* berechnet die Differenz von *e* minus *b* und gibt sie in Monaten an. Liegt *e* vor *b*, dann ist das Ergebnis negativ und wird abgerundet. (Beispiel: *e* liegt drei Tage vor *b*, dann ist das Ergebnis -1.)

```
DECLARE EXTERNAL FUNCTION f_AgeInMonths
    DATE, DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "AgeInMonths"
    MODULE_NAME "FreeUDFLib.dll";
```

f_AgeInWeeks

Die Funktion *f_AgeInWeeks(b, e)* berechnet die Differenz von *e* minus *b* und gibt sie in Wochen an. Liegt *e* vor *b*, dann ist das Ergebnis negativ und wird abgerundet. (Beispiel: *e* liegt drei Tage vor *b*, dann ist das Ergebnis -1.)

```
DECLARE EXTERNAL FUNCTION f_AgeInWeeks
    DATE, DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "AgeInWeeks"
    MODULE_NAME "FreeUDFLib.dll";
```

f_CDOWLong

Die Funktion *f_CDOWLong(d)* gibt den Wochentag des Datums in Langform an (beispielsweise *Sonntag*). Erfreulicherweise werden (bei entsprechender Windows-Einstellung) deutsche Bezeichnungen verwendet.

```
DECLARE EXTERNAL FUNCTION f_CDOWLong
    DATE
    RETURNS CSTRING(10)
    ENTRY_POINT "CDOWLong"
    MODULE_NAME "FreeUDFLib.dll";
```

f_CDOWShort

Die Funktion *f_CDOWShort(d)* gibt den Wochentag des Datums in Kurzform an (beispielsweise *So*).

```
DECLARE EXTERNAL FUNCTION f_CDOWShort
    DATE
    RETURNS CSTRING(4)
    ENTRY_POINT "CDOWShort"
    MODULE_NAME "FreeUDFLib.dll";
```

f_CMonthLong

Die Funktion *f_CDOWLong(d)* gibt den Monat des Datums in Langform an (beispielsweise *Februar*). Erfreulicherweise werden (bei entsprechender Windows-Einstellung) deutsche Bezeichnungen verwendet.

```
DECLARE EXTERNAL FUNCTION f_CMonthLong
    DATE
    RETURNS CSTRING(10)
    ENTRY_POINT "CMonthLong"
    MODULE_NAME "FreeUDFLib.dll";
```

f_CMonthShort

Die Funktion *f_CDOWLong(d)* gibt den Monat des Datums in Kurzform an (beispielsweise *Feb*).

```
DECLARE EXTERNAL FUNCTION f_CMonthShort
    DATE
    RETURNS CSTRING(4)
    ENTRY_POINT "CMonthShort"
    MODULE_NAME "FreeUDFLib.dll";
```

f_DayOfMonth

Die Funktion *f_DayOfMonth(d)* ermittelt den Tag des Datums (also 14 bei *14-Feb-1998*).

```
DECLARE EXTERNAL FUNCTION f_DayOfMonth
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "DayOfMonth"
    MODULE_NAME "FreeUDFLib.dll";
```

f_DayOfWeek

Die Funktion *f_DayOfWeek(d)* ermittelt den Wochentag des Datums (also 1 bei *Sonntag*).

```
DECLARE EXTERNAL FUNCTION f_DayOfWeek
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "DayOfWeek"
    MODULE_NAME "FreeUDFLib.dll";
```

f_DayOfYear

Die Funktion *f_DayOfYear(d)* berechnet, wie viele Tage vom Jahresanfang bis zum angegebenen Datum vergangen sind.

```
DECLARE EXTERNAL FUNCTION f_DayOfYear
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "DayOfYear"
    MODULE_NAME "FreeUDFLib.dll";
```

f_MaxDate

Die Funktion *f_MaxDate(d1, d2)* ermittelt das spätere der beiden Daten.

```
DECLARE EXTERNAL FUNCTION f_MaxDate
    DATE, DATE
    RETURNS DATE
    ENTRY_POINT "MaxDATE"
    MODULE_NAME "FreeUDFLib.dll";
```

f_MinDate

Die Funktion *f_MinDate(d1, d2)* ermittelt das frühere der beiden Daten.

```
DECLARE EXTERNAL FUNCTION f_MinDate
    DATE, DATE
    RETURNS DATE
    ENTRY_POINT "MinDATE"
    MODULE_NAME "FreeUDFLib.dll";
```

f_Month

Die Funktion *f_Month(d)* ermittelt den Monat des Datums.

```
DECLARE EXTERNAL FUNCTION f_Month
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "Month"
    MODULE_NAME "FreeUDFLib.dll";
```

f_Quarter

Die Funktion *f_Quarter(d)* ermittelt das Quartal des Datums.

```
DECLARE EXTERNAL FUNCTION f_Quarter
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "Quarter"
    MODULE_NAME "FreeUDFLib.dll";
```

f_WeekOfYear

Mit *f_WeekOfYear(d)* wird die Kalenderwoche des Datums bestimmt.

```
DECLARE EXTERNAL FUNCTION f_WeekOfYear
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "WeekOfYear"
    MODULE_NAME "FreeUDFLib.dll";
```

f_Year

Die Funktion *f_Year(d)* berechnet das Jahr des Datums.

```
DECLARE EXTERNAL FUNCTION f_Year
    DATE
    RETURNS INTEGER BY VALUE
    ENTRY_POINT "Year"
    MODULE_NAME "FreeUDFLib.dll";
```

12.3.2 Stringfunktionen

f_Left

Die Funktion *f_Left(s, i)* ermittelt die ersten *i* Zeichen des Strings *s*.

```
DECLARE EXTERNAL FUNCTION f_Left
    CSTRING(254), INTEGER
    RETURNS CSTRING(254)
    ENTRY_POINT "Left"
    MODULE_NAME "FreeUDFLib.dll";
```

f_Right

Die Funktion *f_Right(s, i)* ermittelt die letzten *i* Zeichen des Strings *s*.

```
DECLARE EXTERNAL FUNCTION f_Right
    CSTRING(254), INTEGER
    RETURNS CSTRING(254)
    ENTRY_POINT "Right"
    MODULE_NAME "FreeUDFLib.dll";
```

12.4 Funktionen in der GROUP BY-Klausel

Nehmen wir einmal an, Sie wollen feststellen, welcher Mitarbeiter in welchem Jahr wie viel Umsatz gemacht hat. Man könnte das mit der folgenden Anweisung versuchen:

```
SELECT
        m.nachname,
        f_year(b.datum),
        SUM(p.stueckzahl * p.preis) AS preis
    FROM t_bestellung b
        INNER JOIN t_posten p
            ON p.bestellung = b.nummer
        INNER JOIN t_mitarbeiter m
            ON b.bearbeiter = m.nummer
    WHERE (b.nummer  - 20) < (SELECT MIN(nummer)
        FROM t_bestellung)
    GROUP BY m.nachname, f_year(b.datum)
    ORDER BY nachname

/* geht nicht */
```

Die Ausführung dieser Anweisung scheitert allerdings an der Gruppierung nach dem Jahr. Sie könnten nach dem Datum gruppieren, würden dabei allerdings mehrere Datensätze pro Mitarbeiter und Jahr erhalten.

Noch einen Hinweis zur WHERE-Klausel: Die Dauer einer Bearbeitung des gesamten Datenbestandes liegt – je nach Geschwindigkeit des Rechners – in der Größenordnung von einigen Minuten. Deshalb ist es zu empfehlen, die Anweisung zunächst mit Hilfe eines eingeschränkten Datenbestandes zu entwickeln und die WHERE-Klausel zuletzt zu entfernen.

Wie kommen wir nun zu unserer Statistik? Probleme, die man mit den Standard-SQL-Anweisungen nicht in den Griff bekommt, lassen sich meist mit einer STORED PROCEDURE lösen.

```
SET TERM ^;
CREATE PROCEDURE p_bestellung_statistik(nummer INTEGER)
RETURNS
    (bestellung INTEGER,
    jahr INTEGER,
    quartal INTEGER,
    monat INTEGER,
    bearbeiter INTEGER,
    kunde INTEGER,
    preis FLOAT)
AS
BEGIN
    FOR SELECT
            b.nummer,
            f_year(b.datum),
            f_quarter(b.datum),
            f_month(b.datum),
            b.bearbeiter,
            b.kunde,
            SUM(p.stueckzahl * p.preis)
        FROM t_bestellung b
            INNER JOIN t_posten p
                ON b.nummer = p.bestellung
        WHERE b.nummer < :nummer
        GROUP BY b.nummer, b.datum, b.bearbeiter, b.kunde
        INTO :bestellung, :jahr, :quartal, :monat,
            :bearbeiter, :kunde, :preis
    DO SUSPEND;
END^
SET TERM ;^
```

Da wir hier vorausschauend handeln, liefert diese Prozedur nicht nur das Jahr, sondern auch das Quartal und den Monat der einzelnen Bestellungen, darüber hinaus Mitarbeiternummer, Bearbeiternummer sowie Gesamtpreis.

Um in der Testphase überschaubare Datenbestände und somit akzeptable Ausführungsgeschwindigkeiten zu erhalten, kann mit Hilfe des Parameters *nummer* und der WHERE-Klausel die Zahl der verwendeten Datensätze eingeschränkt werden.

Die komplette Statistik würde man nun mit der folgenden Anweisung erstellen:

```
SELECT
        m.nachname,
        b.jahr,
        SUM(b.preis)
    FROM p_bestellung_statistik(100000) b
        INNER JOIN t_mitarbeiter m
            ON m.nummer = b.bearbeiter
    GROUP BY nachname, jahr
```

12.5 Programmieren von UDF-DLLs

Um Sie gleich vorzuwarnen: Das Programmieren von USER DEFINED FUNCTIONS kann eine sehr nervtötende Sache werden. Solange Sie »ausgetretene Pfade« nicht verlassen und eng entlang vorhandener Beispiele programmieren, wird nicht viel passieren. Sobald Sie aber etwas »kreativer« werden, begeben Sie sich in die Gefahr, einen Serverabsturz nach dem anderen zu provozieren.

UDFs sind Routinen, welche direkt vom Server-Prozess aufgerufen werden. Geht dabei etwas schief, dann stürzt dabei der Server ab. Es versteht sich wohl von selbst, dass UDF-DLLs mehr als gründlich getestet werden sollten, bevor sie produktiv eingesetzt werden.

Telefonnummernsuche

Wir wollen hier die Funktion *telstr* erstellen, die bei der Telefonnummernsuche hilfreich ist. Telefonnummern werden wegen der führenden Null als String gespeichert und fast immer lassen sich dort wahlweise Leer- und Sonderzeichen einfügen, um die Nummer lesbarer zu gestalten. Solche »kreative Formatierung« ist jedoch alles andere als hilfreich, wenn man die Telefonnummer wiederfinden möchte.

Am effektivsten bekommt man die Nummer dann mit einer solchen Konstruktion:

```
SELECT *
  FROM t_adressen
  WHERE tel LIKE '0%3%0%1%2%3%4%5%6%7%'
```

Nach jeder Stelle der zu suchenden Nummer wird ein SQL-Jokerzeichen eingefügt, so dass hier beliebige Zeichen stehen können. Es ist zwar nicht gänzlich ausgeschlossen, durch die Länge des Suchstrings ist es jedoch hinreichend unwahrscheinlich, falsche Nummern zu finden.

Wir wollen nun eine UDF erstellen, die aus einer beliebigen Telefonnummer einen solchen Suchstring generiert.

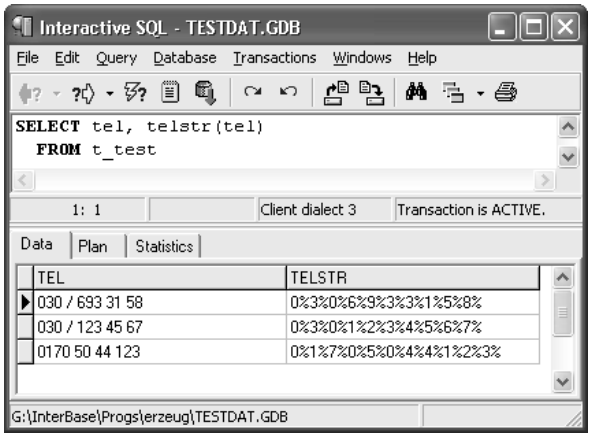


Abbildung 12.2: Generierung von Suchstrings

Die DLL

Da unsere DLL nur eine Routine umfassen soll, verzichten wir auf den Gebrauch weiterer Units.

```
library telstr_udf;

uses
  Windows,
  SysUtils,
  ib_util;

function TELSTR(pIn: PChar): PChar; stdcall;
var
  s, t: string;
  i: integer;
begin
  if pIn = nil then
  begin
    result := nil;
    exit;
  end;
```

```
  try
    t := '';
    s := StrPas(pIn);
    for i := 1 to Length(s) do
    begin
      if s[i] in ['0'..'9']
        then t := t + s[i] + '%';
    end;
    {Speicher muss wie folgt angefordert werden ! }
    i := Length(t);
    result := ib_util_malloc(i + 1);
//    FillChar(result, i + 1, 0); geht nicht !!!
    ZeroMemory(Result, i + 1);
    StrPCopy(result, t);
  except
    result := nil;
  end;
end;

exports
  TELSTR;

begin
  IsMultiThread := true;
end.
```

Das Projekt umfasst nur eine Funktion, nämlich TELSTR, und diese wird auch exportiert. Beachten Sie, dass die Groß- und Kleinschreibung zwar beliebig ist, aber beim Deklarieren der UDF dann genau beachtet werden muss.

Die Aufrufkonvention ist *stdcall*. In der Fachliteratur wird auch schon mal *cdecl* verwendet, das funktioniert nach meiner Beobachtung auch, das Handbuch schreibt jedoch *stdcall* vor. Da wir hier mit Strings arbeiten, sind Parameter und Funktionsergebnis nullterminierte Strings.

Die Funktion geht durch den übergebenen Parameter und schreibt alle Ziffern zusammen mit dem SQL-Jokerzeichen in das Ergebnis.

Für das Funktionsergebnis muss Speicher angefordert werden, den der Server dann wieder freigeben kann – das muss also zwingend (!) mit *ib_util_malloc* geschehen. Die Routine *ib_util_malloc* finden Sie in *ib_util.pas*, diese Unit wiederum im Verzeichnis *\InterBase\SDK\include*.

Der angeforderte Speicher wird mit *ZeroMemory* leer gemacht. Widerstehen Sie der Versuchung, dafür *FillChar* einzusetzen, *InterBase* würde »gnadenlos« abstürzen. Anschließend wird mit *StrPCopy* der Suchstring in das Ergebnis kopiert.

Sie können nun die DLL kompilieren und in das UDF-Verzeichnis kopieren. Wenn Sie das Ausgabeverzeichnis so ändern, dass die DLL gleich im UDF-Verzeichnis ausgegeben wird, dann achten Sie bitte darauf, dass Sie dazu den InterBase-Server beenden müssen, damit die Datei überschrieben werden kann. (Auch das Kopieren misslingt, wenn im UDF-Verzeichnis noch eine verwendete Vorgänger-Version liegt.)

Nun kann die UDF deklariert werden:

```
DECLARE EXTERNAL FUNCTION TELSTR
CSTRING(30)
RETURNS CSTRING(30) FREE_IT
ENTRY_POINT 'TELSTR' MODULE_NAME 'telstr_udf.dll';
```

Was ist zu beachten? Groß- und Kleinschreibung des Funktionsnamen ist egal, sowohl hier als auch bei der späteren Verwendung, nicht aber bei ENTRY_POINT und MODULE_NAME. Für Telefonnummern sollten 30 Zeichen ausreichen, auch beim Rückgabewert.

Mit der Direktive FREE_IT wird dem Server bedeutet, den von der UDF angeforderten Speicher freizugeben, sobald er nicht mehr benötigt wird. Vergessen Sie nicht diesen Aufruf, der Speicherbedarf würde sonst ins Gigantische wachsen.

13 Rundgang durch IBExpert

Das Programm IBConsole ist zwar ganz nett, aber eher ein Tool für die Verwaltung als eine große Hilfe beim Erstellen neuer Datenbanken. Natürlich kann man seine SQL-Scripts vollständig von Hand schreiben, es geht aber auch einfacher.

Dieses Problems haben sich einige Hersteller angenommen und so sind ein paar Tools auf dem Markt, die alle ihre Vorzüge, aber auch ihre Nachteile haben. Es würde den Rahmen dieses Buches sprengen, auch nur eines dieser Tools vollständig zu beschreiben, somit muss ein gründlicher Vergleich aller Produkte – so reizvoll er auch wäre – leider unterbleiben. Vermutlich würde ein solcher Vergleich auch nie fertig werden: Sobald man sich am Ende der Aufgabe wähnte, hätte schon wieder mindestens ein Hersteller eine neue Version auf den Markt gebracht.

Dennoch soll zumindest ein Tool oberflächlich beschrieben werden, damit Sie einen groben Eindruck bekommen, was diese Produkte leisten – letztlich liegen die Unterschiede ja vor allem im Detail.

Dass die Wahl hier auf IBExpert gefallen ist, liegt daran, dass es mir beim ersten Eindruck am meisten ausgereift erschienen ist. Natürlich gibt es auch hier Sachen, die noch nicht ganz rund laufen, noch Funktionen, die noch nicht implementiert sind. Insgesamt war der erste Eindruck jedoch schon ein wenig besser als bei anderen Tools.

13.1 Eine Datenbank registrieren

Nach dem Start des Programms sind erst einmal die meisten Buttons disabled, Sie müssen nun eine Datenbank registrieren oder eine neue erstellen. Wir wollen zunächst die Datenbank *employee.gdb* registrieren, rufen Sie dazu DATABASE | REGISTER DATABASE auf.

Hier müssen Sie die Angaben machen, die das Programm benötigt, um auf die Datenbank zugreifen zu können – im Falle von lokalen Datenbanken den Pfad sowie Benutzernamen und Passwort.

Darüber hinaus möchte IBExpert wissen, welchen Server Sie verwenden. Wie *Abbildung 13.2* zeigt, unterstützt das Programm die verschiedensten InterBase-Derivate.

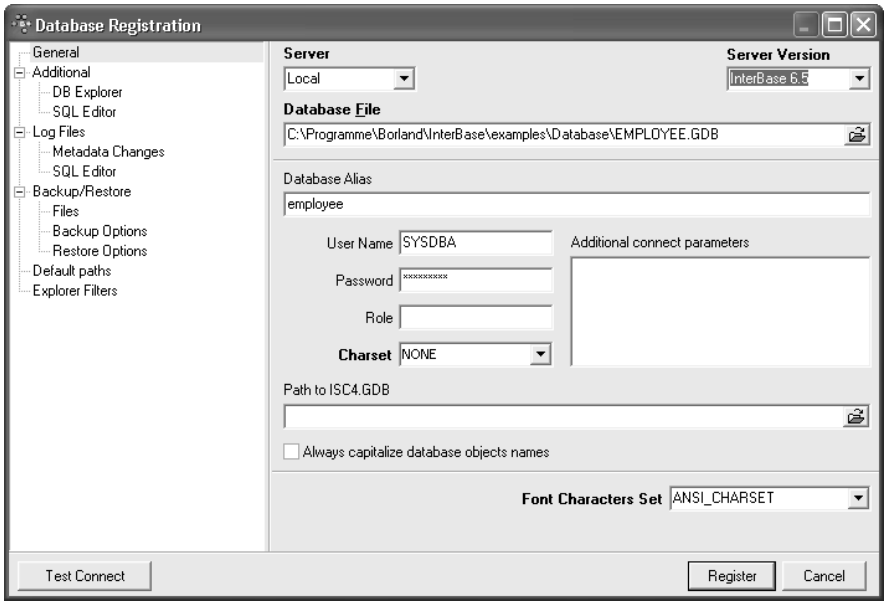


Abbildung 13.1: Eine Datenbank registrieren

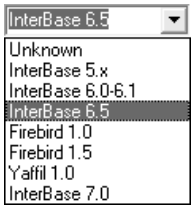


Abbildung 13.2: Die unterstützten Server

Nach dem Registrieren der Datenbank ist diese noch nicht verbunden. Sie sehen den Eintrag zwar im *DB Explorer*, aber er ist noch grau. Führen Sie einen Doppelklick darauf aus und das Programm stellt die Verbindung zu dieser Datenbank her.

13.1.1 Der DB Explorer

Der DB Explorer zeigt Ihnen an, was in der jeweiligen Datenbank alles zu finden ist.

Im oberen Teil erhalten Sie in einer Baumansicht einen Überblick über die einzelnen Elemente der Datenbank, unten sehen Sie dann detailliertere Informationen zur gewählten Rubrik oder dem gewählten Element.

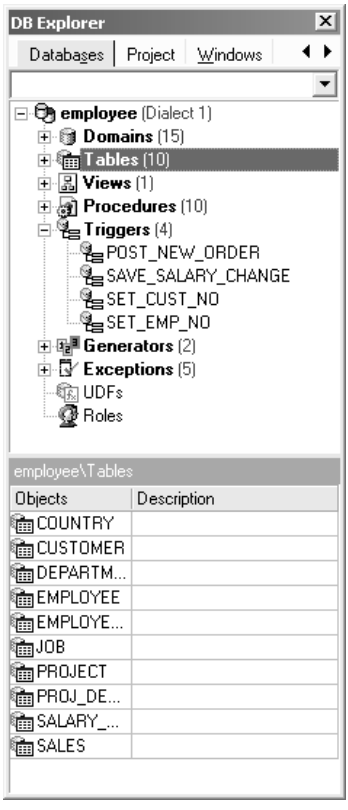


Abbildung 13.3: Der DB Explorer

Besonders produktiv ist der *DB Explorer* nicht, aber Sie gelangen mit einem Doppelklick auf das einzelne Element in entsprechende Experten-Fenster.

13.1.2 Die Tabellen-Ansicht

Beginnen wir mit der Tabellen-Ansicht, die Sie mit einem Doppelklick auf die gewünschte Tabelle öffnen. Solange Sie das MDI-Interface nicht deaktiviert haben, wird für jede Tabelle ein eigenes Experten-Fenster geöffnet.

Die Spalten

Auf der ersten Registerseite finden Sie eine Auflistung aller Spalten. Zu jeder einzelnen Spalte können Sie eine Beschreibung, eine *Field description*, eingeben. So etwas ist besonders bei der Entwicklung im Team hilfreich. Darüber hinaus können Sie sich anzeigen lassen, welche Elemente auf der betreffenden Spalte beruhen – solange es hier Abhängigkeiten (*Field dependencies*) gibt, brauchen Sie erst gar nicht den Versuch zu unternehmen, diese Spalte zu löschen.

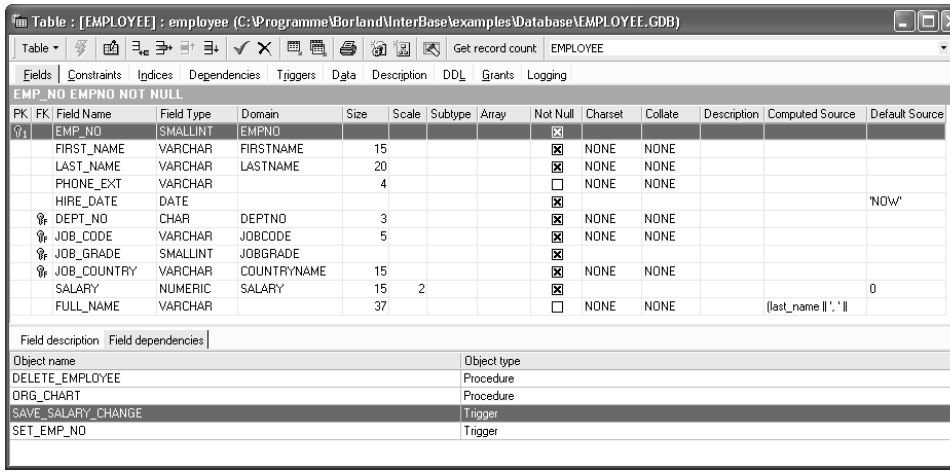


Abbildung 13.4: Die Tabellen-Ansicht

PK steht für *Primary Key*, *FK* für *Foreign Key*, die Symbole in diesen Spalten zeigen an, dass diese Spalte an einem Primär- oder Fremdschlüssel beteiligt ist.

Zum Ändern der Werte gibt es zwei Möglichkeiten: Wenn Sie einen Doppelklick auf eine Spalte ausführen, dann öffnet sich ein Spalten-Editor, mit dessen Hilfe Sie einige Werte ändern können.

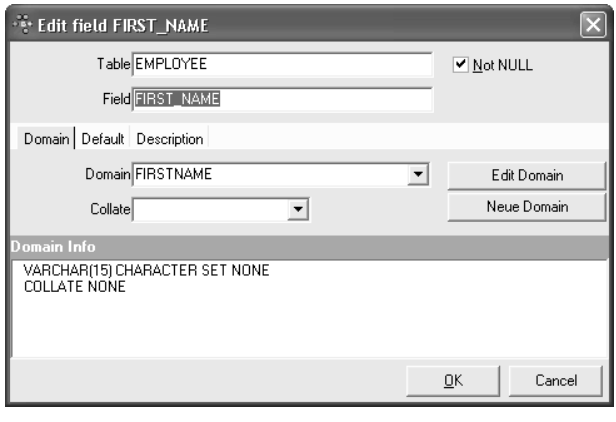


Abbildung 13.5: Spalten-Editor

Alternativ dazu können Sie den Button *Edit Table Structure* oder die Funktionstaste F2 drücken. Nun können Sie die Werte direkt in der Tabelle ändern. Sie werden dabei feststellen, dass manche Felder grau unterlegt sind und manche nicht. Damit wird dargestellt, welche Werte Sie ändern können. Den Spaltenbezeichner LAST_NAME können Sie beispielsweise nicht ändern, weil dieser in einer VIEW verwendet wird.

FIRST_NAME	VARCHAR	FIRSTNAME	15
LAST_NAME	VARCHAR	LASTNAME	20
PHONE_EXT	VARCHAR		4
HIRE_DATE	DATE		
DEPT_NO	CHAR	DEPTNO	3

Abbildung 13.6: Werte direkt in der Tabelle ändern

Wenn Sie Änderungen vornehmen, dann müssen Sie diese anschließend durchführen, indem Sie den Button *Compile* oder die Tastenkombination Ctrl+F9 drücken.

Constraints und Indizes

Auf der nächsten Registerseite haben wir die Constraints. Hier finden Sie

- den Primärschlüssel mit Namen und beteiligten Feldern
- alle Fremdschlüssel der betreffenden Tabelle mit den entsprechenden Informationen, also auch referenzierende Tabellen, die dort referenzierten Spalten sowie die UPDATE- und DELETE-Regeln
- die Gültigkeitsprüfungen
- die Sekundärschlüssel

Auf der dritten Registerseite finden wir die Indizes.

	Index	On field	Unique	Status	Sorting	Statistics
	NAMEX	LAST_NAME, FIRST_NAME	☐	☒	Ascending	0,023809524253010...
FK	RDB$FOREIGN8	DEPT_NO	☐	☒	Ascending	0,052631579339504...
FK	RDB$FOREIGN9	JOB_CODE, JOB_GRADE, JOB_COUNTRY	☐	☒	Ascending	0,037037037312984...
PK	RDB$PRIMARY7	EMP_NO	☒	☒	Ascending	0,023809524253010...

Abbildung 13.7: Die Indizes

Auch hier gilt: Alles, was grau unterlegt ist, können Sie nicht ändern. So können Sie den Index für den Primärschlüssel beispielsweise nicht deaktivieren.

Der Wert in der Spalte *Statistics* sollte der Kehrwert der eindeutigen Werte im Index sein. Der Konjunktiv wird hier verwendet, da dieser Wert nur dann neu berechnet wird, wenn der Index neu aufgebaut oder SET STATISTICS aufgerufen wird. Anhand dieses Wertes entscheidet der Query-Optimizer, ob es sinnvoll ist, diesen Index zu verwenden.

Abhängigkeiten und Trigger

Auf der Registerseite *Dependencies* haben Sie dann eine Liste aller Elemente, die von der jeweiligen Tabelle abhängen, sowie eine Liste aller Elemente, auf denen die Tabelle basiert. Wenn Sie ein Element wählen, dann wird unten im Fenster eine Bearbeitungsansicht angezeigt:

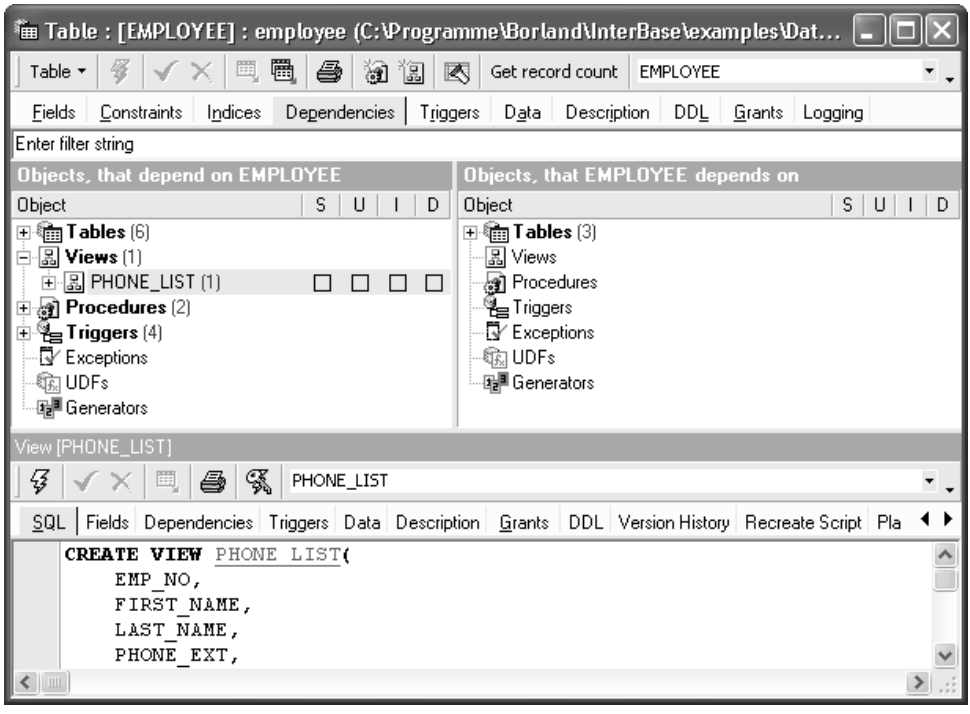


Abbildung 13.8: Abhängigkeiten

Bei der genauen Betrachtung von *Abbildung 13.1* wundern Sie sich vielleicht, warum hier vier Trigger erkannt werden, wo doch für die Tabelle *employee* nur zwei definiert sind. Das liegt daran, dass die CHECK-Constraint als Before-Insert- und Before-Update-Trigger in der Systemtabelle RDB$TRIGGERS zu finden ist.

Auf der Registerseite *Triggers* sind dann wieder nur die TRIGGER zu sehen, die auch als solche definiert wurden:

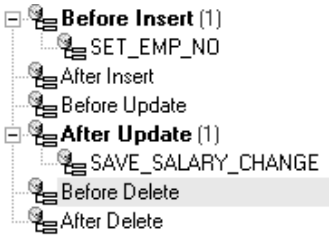


Abbildung 13.9: Trigger

Die Daten

Natürlich kann man sich auch die Daten anzeigen lassen. Wird *Grid View* gewählt, dann werden die Daten in tabellarischer Form angezeigt.

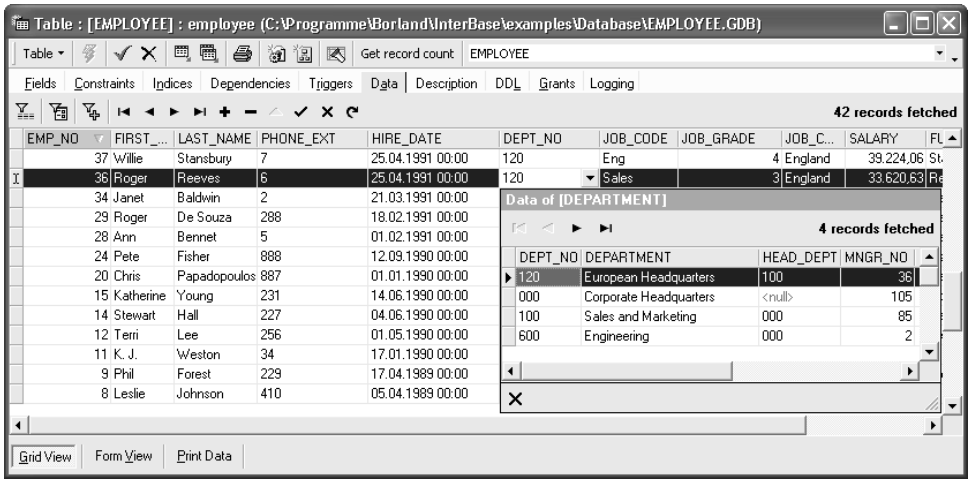


Abbildung 13.10: Ansicht im Grid

Um sich die Daten nach einer bestimmten Spalte sortiert anzeigen zu lassen, klickt man auf die Spaltenüberschrift. Mit Hilfe der Spaltenüberschriften lässt sich auch die Reihenfolge der Spalten ändern. Dort, wo Fremdschlüssel eingerichtet sind, wird automatisch eine Nachschlagetabelle hinterlegt, die alle Spalten der betreffenden Tabelle anzeigt.

Alternativ zur Grid-Ansicht gibt es *Form View*. Dort werden die Spalten mit einzelnen Komponenten dargestellt, so dass nur ein einzelner Datensatz angezeigt werden kann. Leider können die einzelnen Komponenten nicht frei platziert werden, so dass eine individuelle Eingabemaske – die man dann auch speichern müsste – zumindest derzeit noch nicht erstellt werden kann.

Möglichkeiten zur Steigerung der Flexibilität gibt es auch noch bei *Print Data*. Zwar kann der Report nicht nur gedruckt, sondern auch in verschiedenen Formaten gespeichert werden (darunter PDF, HTML und RTF), einzelne Spalten können jedoch nicht ausgeblendet werden.

Es ist jedoch möglich, die Daten lokal zu filtern. Dazu gibt es eine Maske, bei der die betreffenden Spalten gewählt und die gewünschten Filterwerte eingegeben werden. Auf eine Spalte können auch mehrere Filter gesetzt werden, die dann mit AND oder OR verknüpft werden können. Mit der CheckBox in der Spalte *A* lässt sich eine Filterbedingung vorübergehend deaktivieren, mit der CheckBox in der Spalte *CS* sorgt man für die Berücksichtigung von Groß- und Kleinschreibung.

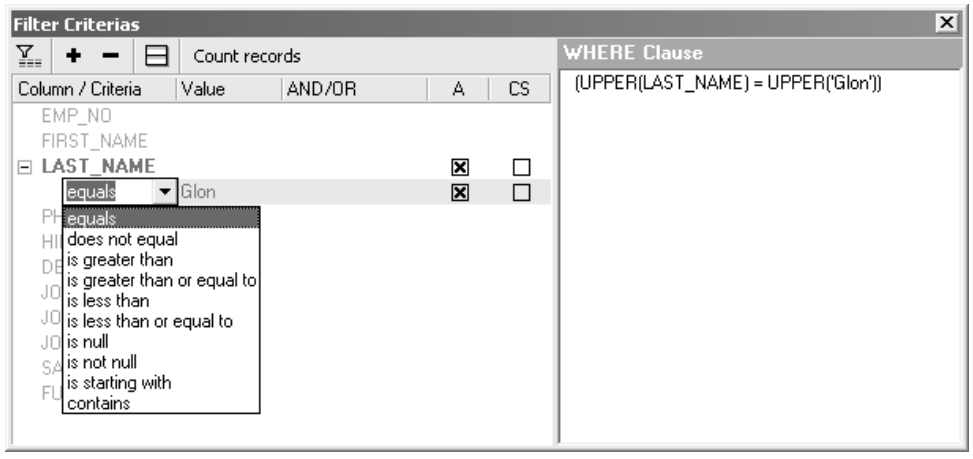


Abbildung 13.11: Filtern der Daten

Die generierte SQL-Anweisung wird auf der rechten Seite angezeigt und kann kopiert werden. Beachten Sie, dass die Daten lokal gefiltert werden, es werden dabei keine neuen SQL-Anweisungen zum Server geschickt.

Die weiteren Registerseiten

Auf der Seite *Description* können Sie eine Beschreibung der Tabelle eingeben. Für solche Beschreibungen gibt es entsprechende Spalten in den Systemtabellen, so dass auch Entwickler, die mit anderen Tools arbeiten, mit diesen Beschreibungen arbeiten können.

Auf der Registerseite *DDL* finden Sie das SQL-Script, das IBExpert aus der aktuellen Tabellen-Definition erstellt. Wenn Sie die Farbe stören sollte, mit der Strings angezeigt werden, dann können Sie dies unter OPTIONS | EDITOR OPTIONS | COLOR ändern.

Auf der Registerseite *Grants* können die Zugriffsrechte für die Tabelle spezifiziert werden.

Wenn Sie Datenänderungen protokollieren, können Sie die Änderungen auf der Registerseite *Logging* einsehen.

13.2 Datenänderungen protokollieren

Bei der Fehlersuche kann es sehr nützlich sein, wenn man nachvollziehen kann, wer wann welche Änderungen an einer Tabelle vorgenommen hat. Um Datenänderungen zu protokollieren, muss man eigentlich nur ein paar Tabellen erstellen, in denen das Protokoll abgelegt wird, und die interessanten Tabellen mit ein paar

TRIGGERN versehen, damit die entsprechenden Tabellen auch gefüllt werden. Eigentlich ist das nicht weiter schwierig, aber es ist doch deutlich komfortabler, diese Arbeit an IBExpert zu delegieren.

Objekte einrichten

Solange die dafür nötigen Objekte noch nicht vorhanden sind, fragt einen IBExpert, ob deren Erstellung genehmigt wird, sobald man auf die Registerseite *Logging* wechselt.

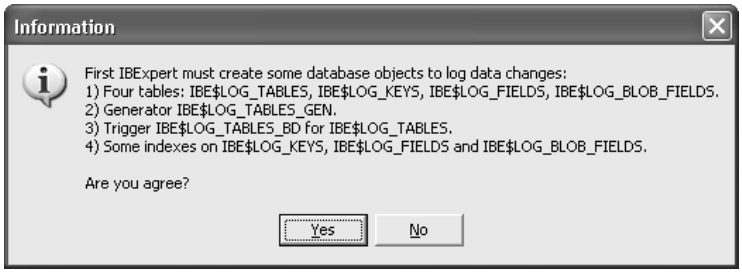


Abbildung 13.12: Zustimmungserklärung für die Log-Objekte

IBExpert erstellt dann die benötigten Objekte, als *Result* sollten Sie immer *Succesful* haben, sonst gibt es vielleicht schon ein Objekt dieses Namens. Geht alles glatt, dann schließen Sie die Transaktion mit *Commit* ab.

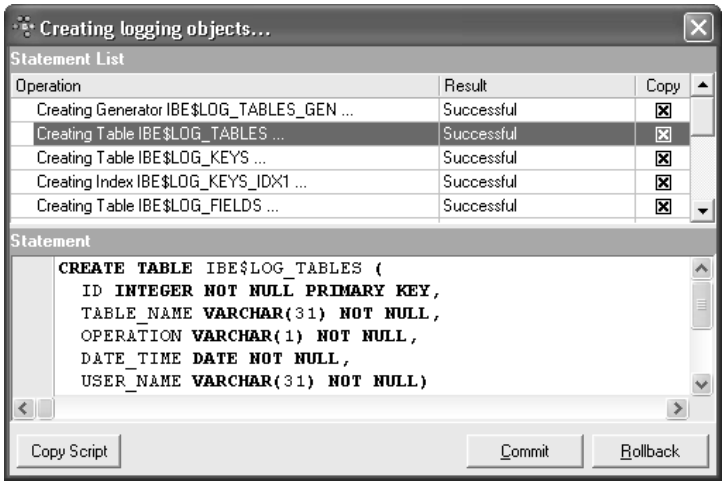


Abbildung 13.13: Commit der neu erstellten Objekte

Sie können jedoch auch das SQL-Script dafür kopieren.

Eine Tabelle zur Überwachung vorbereiten

Wir wollen nun die Tabelle *employee* zur Überwachung vorbereiten. Wählen Sie dazu TABLE | PREPARE TABLE FÜR LOGGING.

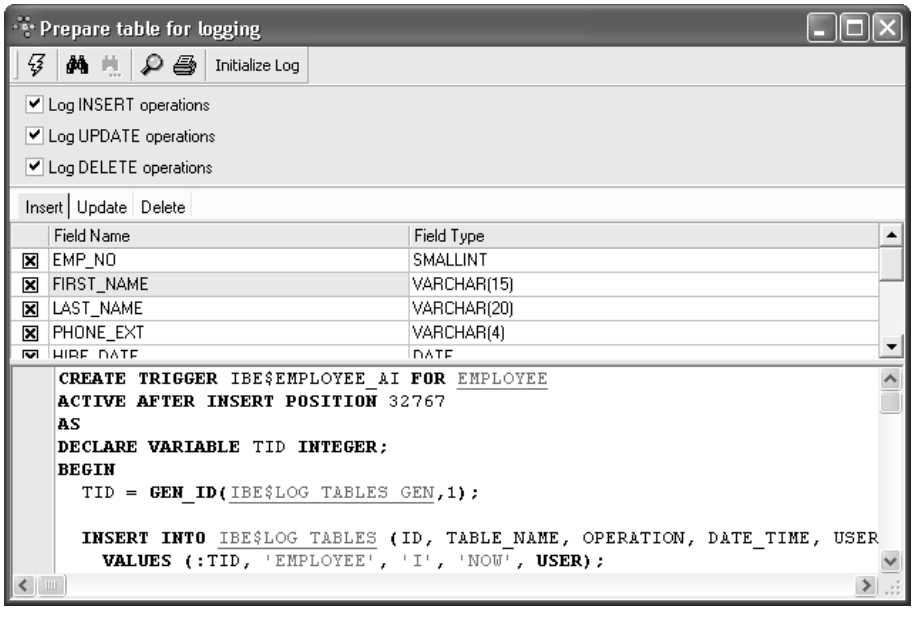


Abbildung 13.14: Tabelle zur Überwachung vorbereiten

Hier können Sie dezidiert einstellen, welche Datenänderung Sie protokollieren möchten, und für den INSERT-, den UPDATE- und den DELETE-Fall können Sie einzeln einstellen, welche Spaltenwerte dabei mitgespeichert werden sollen. Es mag zunächst verwundern, dass Sie diese Einstellung auch im DELETE-Fall machen können, aber auf diese Weise können Sie festhalten, welche Werte ein Datensatz hatte, bevor er gelöscht wurde.

In der Entwicklungsphase mag es sinnvoll sein, alle Änderungen zu protokollieren. Bei einer produktiven Datenbank würde man dann meist von der Vielzahl der Einträge »erschlagen« werden. Hier sollte man sich erst mal darüber Gedanken machen, welche Änderungen einen überhaupt interessieren.

Auch hier muss man mit dem Button *Compile* erst die nötigen Einträge durchführen, die dann auch mit *Commit* bestätigt werden müssen.

Die Änderungen einsehen

Nun kann man auf der Registerseite *Logging* verfolgen, welche Änderungen an der jeweiligen Tabelle vorgenommen wurden.

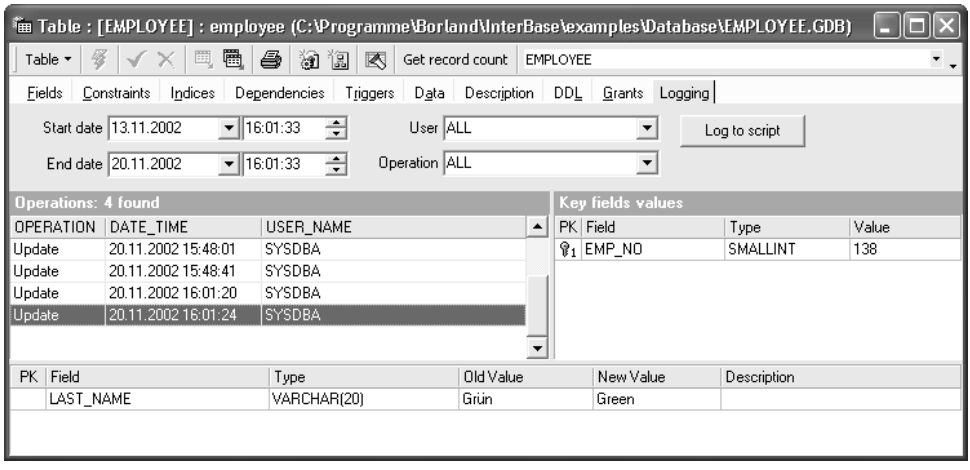


Abbildung 13.15: Die Liste der Änderungen

Selbst dann, wenn keine Spaltenwerte mitprotokolliert werden, werden die Primärschlüssel in *Key field values* angezeigt, so dass man zumindest den geänderten Datensatz ermitteln kann. Bei produktiven Datenbanken muss man gewöhnlich massiv von den Filtermöglichkeiten Gebrauch machen, um die interessanten Datensätze in erträglicher Zeit zu finden.

13.3 Datenbank visuell darstellen

Insbesondere dann, wenn man sich in fremde Datenbanken einarbeiten muss, kann es sehr hilfreich sein, sich die darin enthaltenen Verknüpfungen visuell darstellen zu lassen. Auch dies ist mit IBExpert möglich. Der umgekehrte Weg – neue Datenbanken visuell aufzubauen oder an bestehenden Datenbanken visuelle Änderungen vorzunehmen – ist bereits vorgesehen, aber derzeit noch nicht vollständig implementiert.

Die Funktion finden Sie unter TOOLS | DATABASE DESIGNER. Rufen Sie dort dann DESIGNER | REVERSE ENGINEER auf und wählen die gewünschte Datenbank aus. Sie werden noch nach einigen Details gefragt – beispielsweise, ob Sie ein neues Diagramm erstellen oder ein bestehendes ersetzen wollen –, dann macht sich das Tool an die Arbeit.

Das erste Ergebnis ist ein wenig ernüchternd: Zum einen sehen wir nur die Log-Tabellen, die IBExpert angelegt hat, zum anderen keine Verknüpfungen. Das mit den Verknüpfungen liegt daran, dass die Log-Tabellen einfach keine Fremdschlüssel haben, und die anderen Tabellen werden ein Stückchen tiefer angezeigt, man muss nur dorthin scrollen.

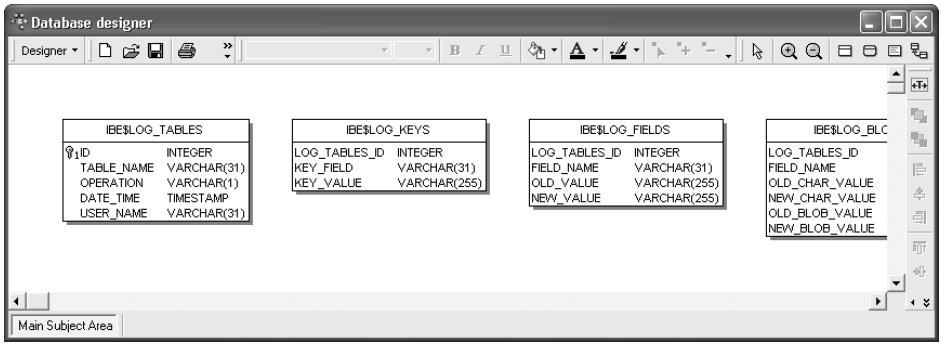


Abbildung 13.16: Die Log-Tabellen

Die Tabellen werden zunächst alle nebeneinander dargestellt, somit werden die Verknüpfungen auch meist von anderen Tabellen verdeckt. So ist das noch nicht verwendungsfähig, aber hier eine brauchbare Entflechtung zu erwarten wäre wohl ein wenig anspruchsvoll – also müssen wir selbst Hand anlegen.

Wählen Sie die Verkleinerungsfunktion (Lupe mit Minuszeichen) und verkleinern die Abbildung erst mal so lange, bis Sie alle Tabellen darstellen können. Anschließend ziehen Sie die Symbole erst einmal ein wenig auseinander, so dass Sie erkennen können, welche die zentralen Tabellen sind – bei der Datenbank *employee.gdb* wäre das die Tabelle *employee*. Diese ordnen wir nun zentral an, anschließend verteilen wir alle anderen Tabellen so, dass miteinander verbundene Tabellen möglichst nebeneinander liegen.

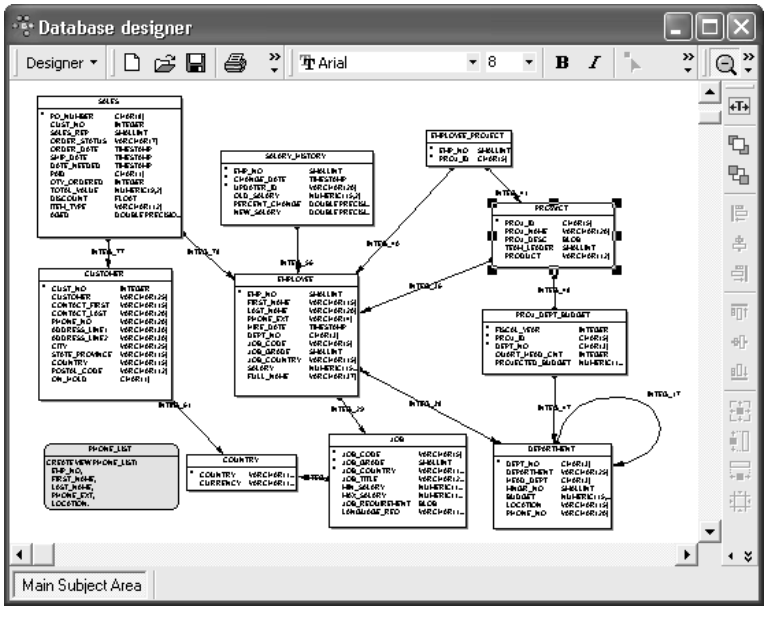


Abbildung 13.17: Die Datenbank employee.gdb

Eine neue Tabelle erstellen

Wir wollen nun eine kleine Datenbank visuell aufbauen und beginnen dafür ein neues Diagramm. Um eine neue Tabelle einzufügen, klicken wir auf das Tabellensymbol (rechts neben dem Verkleinerungssymbol) und dann auf die Position, an der die Tabelle eingefügt werden soll. Die neu eingefügt Tabelle können wir nun verschieben und in ihrer Größe verändern.

Mit einem Doppelklick auf die Tabelle öffnen wir einen Dialog, in dem wir zunächst den Tabellennamen auf *t_master* ändern. Damit die Tabelle später auch in das SQL-Script aufgenommen wird, aktivieren wir auch die Option *Generate*.

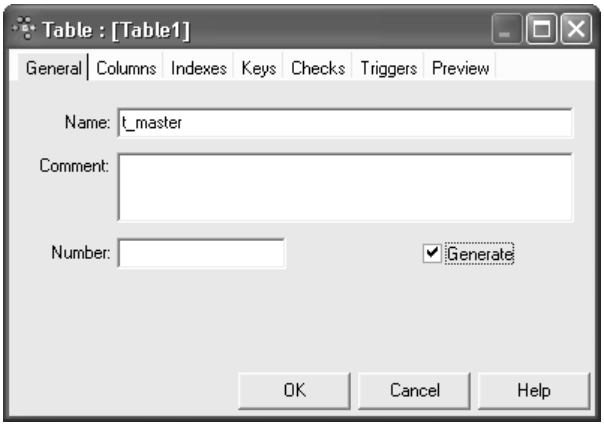


Abbildung 13.18: Eine Tabelle umbenennen

Nun wechseln wir auf die Registerseite *Columns*. Bevor wir Spalten anlegen, kümmern wir uns erst einmal um einige Domänen und klicken dafür auf den Button *New Domain/Edit Domain*:

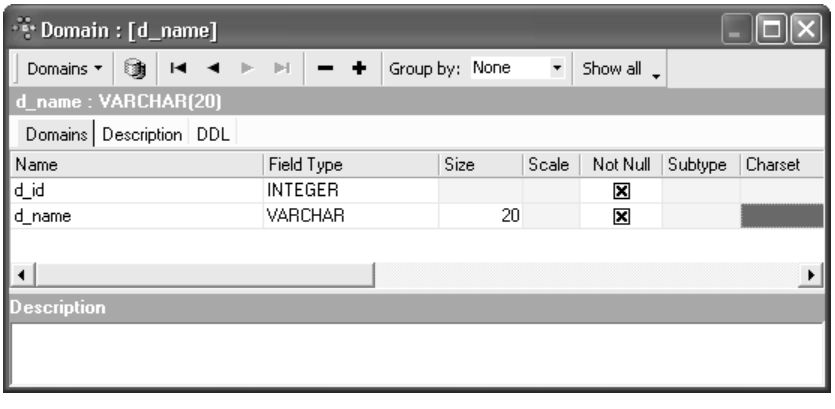


Abbildung 13.19: Anlegen von Domänen

Hier legen wir nun die Domänen *d_id* und *d_name* an – die Details entnehmen Sie *Abbildung 13.2*. Einen Button zum Bestätigen der Änderungen gibt es noch nicht – schließen Sie nach dem Anlegen einfach das Fenster.

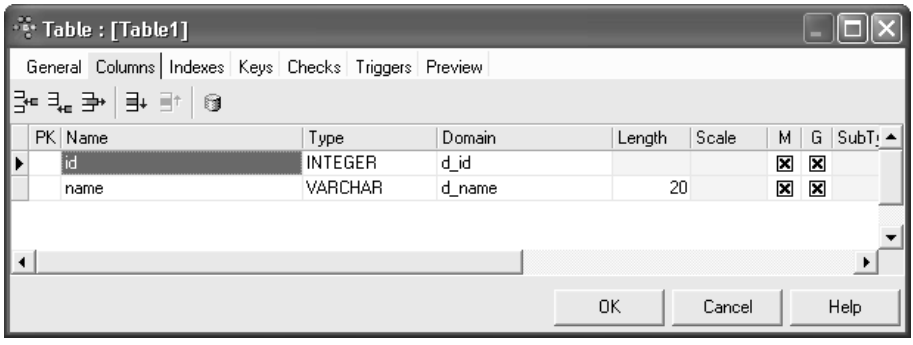


Abbildung 13.20: Anlegen der Spalten

Nun legen wir die Spalten *id* und *name* an. Da wir entsprechende Domänen haben, müssen wir lediglich diese auswählen, andere Informationen wie beispielsweise *Type* werden dann automatisch übernommen. Die Spalte *M* steht für *mandatory*, in der SQL-Syntax NOT NULL.

Lassen Sie die Option *G* aktiviert, sonst wird die Spalte später nicht in das Skript aufgenommen.

Beim Anlegen der Spalten vermisse ich die Möglichkeit, selbst inkrementierende Felder anzulegen. IBExpert beweist an anderer Stelle, dass es in der Lage ist, automatisch GENERATOR, TRIGGER und STORED PROCEDURE anzulegen.

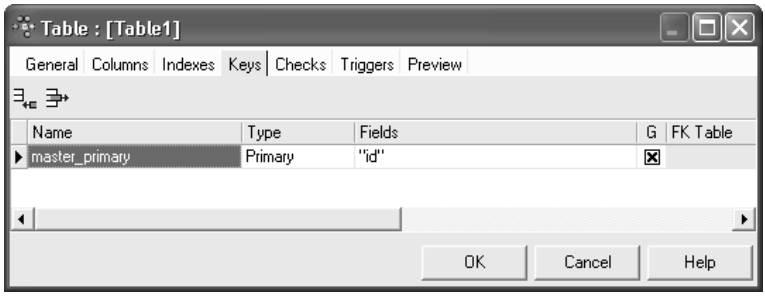


Abbildung 13.21: Einen Primärschlüssel anlegen

Nun legen wir auch noch einen Primärschlüssel an, Näheres entnehmen Sie *Abbildung 13.4*. Wir könnten einen Primärschlüssel auch dadurch anlegen, dass wir auf der Registerseite *Columns* einen Doppelklick in der Spalte *PK* durchführen. Erfreulicherweise erkennt IBExpert, dass hier an anderer Stelle ein Primärschlüssel angelegt wird, und ergänzt in dieser Spalte das Symbol.

Auf der Registerseite *Preview* können wir uns schon einmal ansehen, welches SQL-Script aus diesen Einstellungen generiert würde:

```
/* Table : "t_master" */

CREATE TABLE "t_master"(
  "id" INTEGER NOT NULL,
  "name" VARCHAR(20) NOT NULL);

ALTER TABLE "t_master"
  ADD CONSTRAINT "master_primary" PRIMARY KEY ("id");
```

Nun erstellen wir die Tabelle *t_detail*:

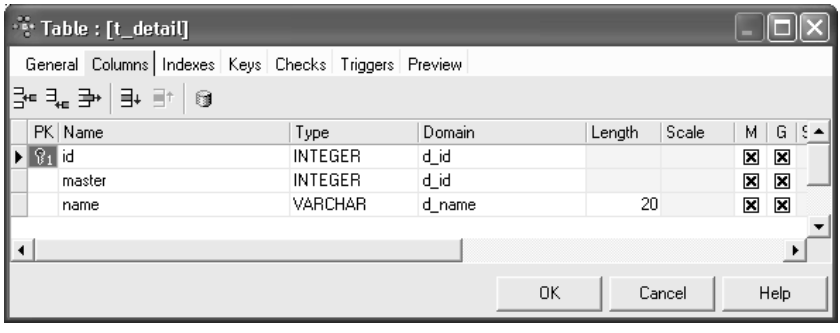


Abbildung 13.22: Die Tabelle t_detail

Hier wollen wir nun mit einem Doppelklick auf die Spalte *PK* den Primärschlüssel anlegen. Auf der Registerseite *Keys* sehen wir, dass der Primärschlüssel automatisch benannt wurde.

Ergänzen wir nun einen Fremdschlüssel, der die Tabelle *t_master* referenziert.

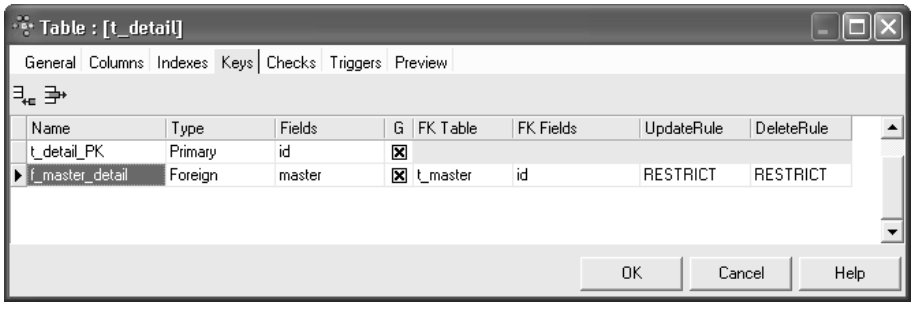


Abbildung 13.23: Einen Fremdschlüssel anlegen

Wenn wir den Dialog mit OK beenden, dann sehen wir im Diagramm nicht nur die Tabelle *t_detail*, sondern auch einen Verknüpfungspfeil entsprechend dem eingerichteten Fremdschlüssel.

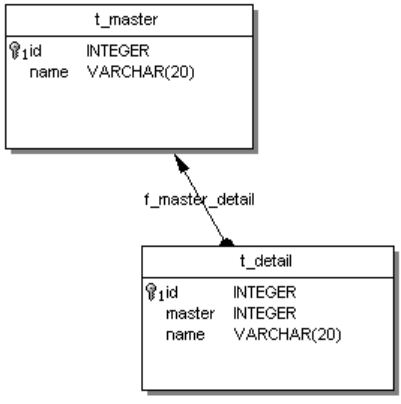


Abbildung 13.24: Eine Master-Detail-Verknüpfung

Mit einem Doppelklick auf den Pfeil könnten Sie einen Dialog öffnen, mit dessen Hilfe Sie die beteiligten Spalten sowohl auf der Master- als auch auf der Detail-Seite ändern könnten.

Klicken wir nun auf DESIGNER | GENERATE SCRIPT, dann erstellt uns dieses Tool das folgende SQL-Script (der kürzeren Darstellung wegen wurde ein wenig umformatiert):

```
/* Domains definition */
CREATE DOMAIN "d_id" AS INTEGER NOT NULL;
CREATE DOMAIN "d_name" AS VARCHAR(20)NOT NULL;

/* Tables definition */
CREATE TABLE "t_detail" (
    "id" "d_id" NOT NULL,
    "master" "d_id" NOT NULL,
    "name" "d_name" NOT NULL);

CREATE TABLE "t_master" (
    "id" "d_id" NOT NULL,
    "name" "d_name" NOT NULL);

/* Views definition */
/* Primary keys definition */
ALTER TABLE "t_detail"
  ADD CONSTRAINT "t_detail_PK" PRIMARY KEY ("id");
```

```
ALTER TABLE "t_master"
  ADD CONSTRAINT "master_primary" PRIMARY KEY ("id");

/* Foreign keys definition */
ALTER TABLE "t_detail" ADD CONSTRAINT "f_master_detail"
  FOREIGN KEY ("master") REFERENCES "t_master" ("id");

/* Unique keys definition */
/* Indices definition */
/* Triggers definition */
SET TERM ^ ;

SET TERM ; ^
```

Fazit

Die Standardaufgaben beim Erstellen von Tabellen beherrscht der *Database Designer* recht gut, wenn ich auch den Automatismus zum Erstellen von selbst inkrementierenden Spalten vermisse. VIEWS lassen sich auch visualisieren, man muss sie jedoch derzeit noch vollständig per SQL-Anweisung erstellen. Hier könnte ein entsprechender Experte helfen.

Das erstellte Diagramm lässt sich ein wenig grafisch gestalten, Schriften und Farben lassen sich beispielsweise individuell einstellen, auch kann man Textfenster mit Kommentaren ergänzen. Auf der rechten Seite finden sich einige Buttons, um Elemente einheitlich auszurichten, markieren Sie dafür mehrere Elemente.

Das Diagramm lässt sich auch drucken, jedoch noch nicht in anderen Formaten speichern. Möchte man das Diagramm ganz oder teilweise in einer technischen Dokumentation nutzen, dann muss man derzeit noch einen Screenshot ziehen – aber ich nehme an, es ist nur eine Frage der Zeit, bis es da andere Möglichkeiten gibt.

13.4 Testdaten erstellen

IBExpert bietet auch die Möglichkeit, sich Testdatensätze generieren zu lassen. Wir wollen uns auch dies etwas genauer ansehen. Damit wir damit keine andere Tabelle »verhunzen«, wollen wir dafür zunächst die Tabelle *t_test* erstellen.

13.4.1 Eine neue Tabelle erstellen

Die Möglichkeit, mit dem *Database Designer* Tabellen zu erstellen, kennen wir schon. Deshalb rufen wir nun den Menüpunkt DATABASE | NEW TABLE auf.

Der ersten Spalte geben wir den Bezeichner *id*, sie ist vom Typ INTEGER und wird als NOT NULL spezifiziert und zum Primärschlüssel gemacht. Wir wollen hier ein selbst inkrementierendes Feld haben und führen deshalb auf die Spalte *AutoInc* einen Doppelklick aus.

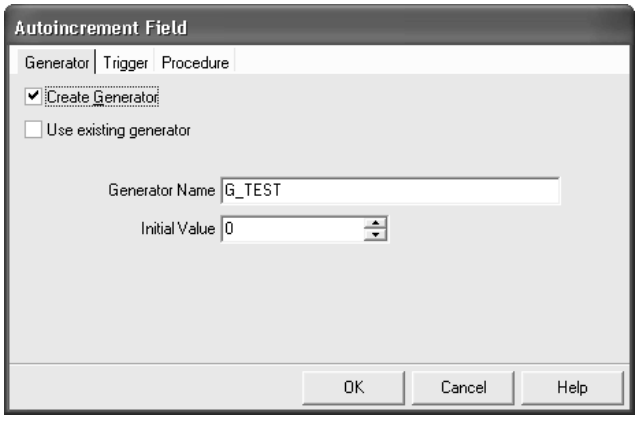


Abbildung 13.25: Generator erstellen

Zunächst wählen wir die Option *Create Generator* und benennen ihn mit G_TEST. Wenn wir *Use existing generator* gewählt hätten, hätten wir aus der Liste der bestehenden Generatoren wählen können. Wenn wir *Initial Value* auf null lassen, dann hat der erste Datensatz den Wert eins – das ist genau das, was wir haben wollen.

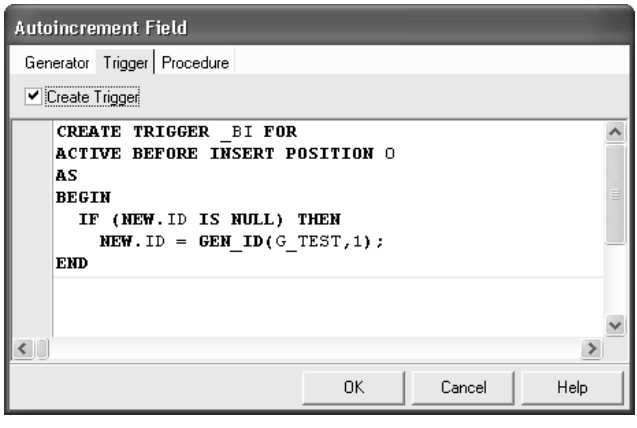


Abbildung 13.26: Einen Trigger erstellen

Wechseln wir auf die Registerseite *Trigger* und aktivieren dort die Option *Create Trigger*. IBExpert erstellt darauf automatisch einen Trigger, der erfreulicherweise auch die IS NULL-Prüfung vornimmt. Eine STORED PROCEDURE könnten wir ebenso einfach erstellen, wir brauchen aber hier keine.

Ergänzen wir die Spalten *vorname*, *nachname* und *gehalt*:

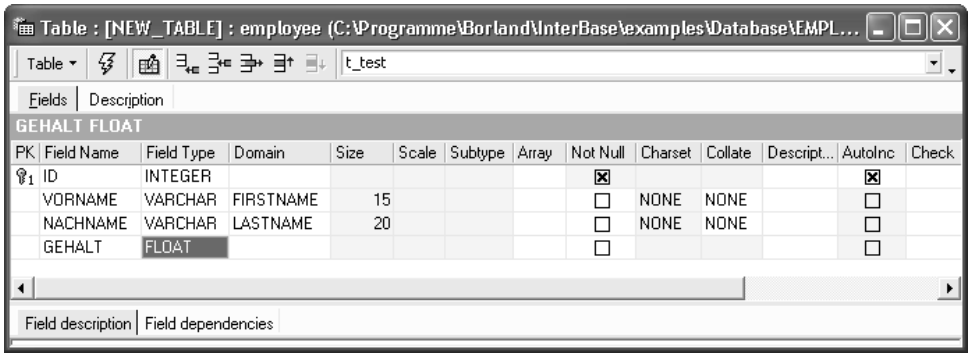


Abbildung 13.27: Die Tabelle t_test

Um die Tabelle zu erstellen, klicken wir auf den Button *Compile*. Dabei stellen wir fest, dass der Tabellenname leider nicht automatisch in den Trigger übernommen wird und somit diese Anweisung nicht ausgeführt werden kann. Bearbeiten wir diese Anweisung und kompilieren dann erneut.

13.4.2 Die Testdaten

Nun wählen wir TOOLS | TEST DATA GENERATOR und wählen die Tabelle *t_test*.

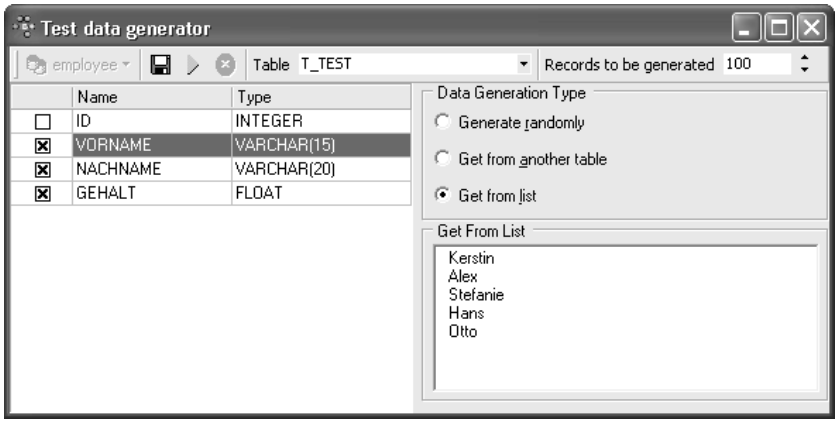


Abbildung 13.28: Der Testdaten-Generator

Die Spalte *ID* bekommt ihren Generatorwert, für die anderen drei Spalten wählen wir die CheckBox in der ersten Spalte und sorgen somit dafür, dass diese mit zufälligen Werten gefüllt werden. Dafür stehen uns drei Möglichkeiten zur Verfügung:

- Mit *Generate randomly* werden völlig zufällige Werte erstellt. Das bietet sich bei Zahlen an, beispielsweise bei der Spalte *Gehalt*, bei Namen würden einfach irgendwelche Buchstaben kombiniert, dort macht dies weniger Sinn.
- Wählen wir die Option *Get from another table*, dann könnten wir angeben, aus welcher Tabelle und welcher Spalte die Werte bezogen werden. Dabei kann spezifiziert werden, wie viele Datensätze dabei berücksichtigt werden sollen – bei sehr großen Tabellen kann es Sinn machen, das zu beschränken, weil die Tabelle dafür lokal gespeichert wird –, aus dieser Menge wird dann jeweils ein zufälliger Datensatz ausgewählt.
- Danach können wir eine Werteliste eingeben und die Option *Get from list* wählen – auch dabei wird jeweils ein zufälliger Wert gewählt.

Zuletzt wird dann noch die Anzahl der zu erzeugenden Datensätze gewählt, anschließend wird mit *Generate* IBExpert an die Arbeit geschickt.

Für einfache Testdaten bringt IBExpert sehr schnell brauchbare Ergebnisse, aber Telefonnummern, deren Vorwahl dann auch noch zur Postleitzahl passen, braucht man nicht zu erwarten.

13.5 Optimierung

IBExpert stellt auch einige Hilfsmittel zur Optimierung von SQL-Anweisungen zur Verfügung. Damit das auch »Spaß macht«, wollen wir dazu etwas größere Datenmengen verwenden und registrieren unsere Testdatenbank.

Die SQL-Anweisung

Aus Kapitel µ kennen wir den JOIN zwischen den Tabellen *t_bestellung*, *t_kunde* und *t_mitarbeiter*. Diesen wollen wir auch hier eingeben.

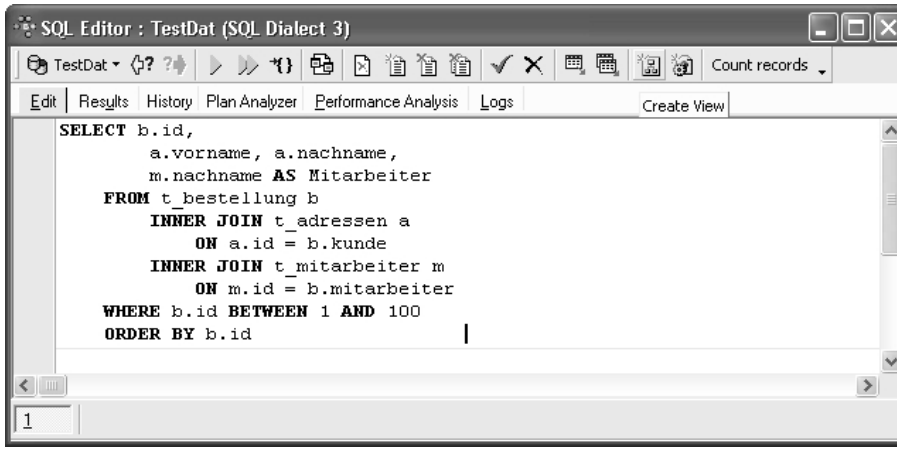


Abbildung 13.29: Ein JOIN über zwei Tabellen

Wir können diese Anweisung mit *Execute* ausführen, dann werden jedoch nur so viele Datensätze geholt, wie auch angezeigt werden. Sollen alle Datensätze geholt werden, dann ist *Execute and fetch all* zu verwenden. Mit *Prepare Query* kann man die Anweisung vorbereiten, ohne sie auszuführen – damit kann man sehr schnell den Query-Plan ermitteln.

Die Ergebnismenge

Führen wir die Anweisung aus:

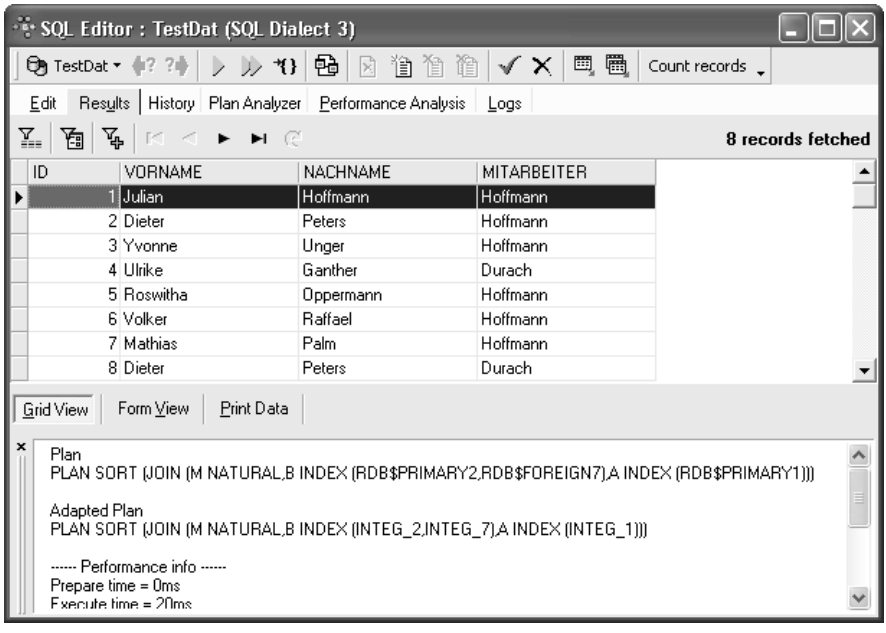


Abbildung 13.30: Die Ergebnismenge

Neben der Ergebnismenge bekommen wir hier auch gleich den Query-Plan angezeigt sowie einige statistische Informationen, beispielsweise die *Execute*-Zeit. Diese liegt mit 20 ms recht exakt bei der Zeit, die wir in Kapitel µ gemessen haben, was allerdings eher zufällig ist: Das Programm misst in 10-ms-Intervallen und damit vergleichsweise ungenau. Selbst die *Avg fetch time* wird aus so einer 10-ms-Messung ermittelt, die dann jedoch noch durch die Zahl der abgerufenen Datensätze geteilt wird.

Man mag nun einwenden, dass es für den Anwender ohnehin keinen Unterschied macht, ob seine Anweisung in 10 ms oder in 20 ms ausgeführt wird. Das ist richtig, aber je länger die Abfrage braucht, desto länger ist der Server beschäftigt. Und bei vielen Abfragen kann das den Unterschied machen zwischen einem Server, der mit der Arbeit zurechtkommt und einem Server, der Abfragen lange zurückstellen muss, weil er mit den vorherigen Anweisungen noch nicht durch ist.

Die Anzahl der Lese-Zugriffe

Auf der Registerseite *Performance Analysis* wird dargestellt, auf welche Tabelle wie oft zugegriffen wird.

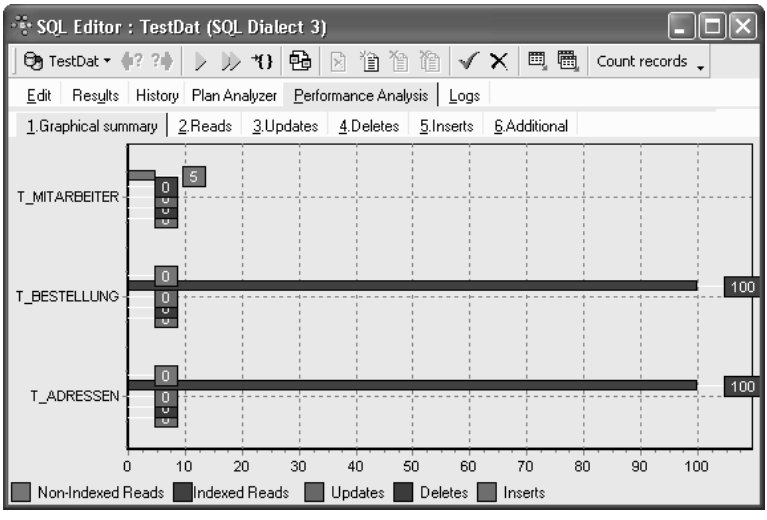


Abbildung 13.31: Die Zahl der Lese-Zugriffe

Dabei wird unterschieden zwischen Zugriffen, die über einen Index durchgeführt werden, und Zugriffen ohne Index. Hier im Beispiel werden nur fünf Lese-Zugriffe auf die Tabelle *t_mitarbeiter* ausgeführt – mehr Datensätze hat sie auch nicht.

Im Gegensatz dazu würden bei einem LEFT OUTER JOIN – der, wie wir uns erinnern, schneller ausgeführt wird – hundert Zugriffe auf *t_mitarbeiter* durchgeführt, die alle über den Index laufen. Bei RIGHT OUTER JOIN werden dagegen alle beteiligten Tabellen vollständig ausgelesen, also beispielsweise 378300 Datensätze bei *t_bestellung* – da braucht man gar nicht zu erwarten, dass dies performant läuft.

Weitere Features

In aller Kürze noch einige weitere Features dieses Fensters:

- Auf der Registerseite *History* werden die bislang ausgeführten Anweisungen aufgelistet. Mit einem Doppelklick auf die betreffende Zeile kopiert man diese Anweisung in das Anweisungsfeld.
- Die Daten lassen sich in zwölf verschiedenen Formaten exportieren, dabei lässt sich genau spezifizieren, wie die einzelnen Typen formatiert werden sollen. Es ist auch möglich, aus der Ergebnismenge INSERT-Anweisungen zu generieren, um die Daten beispielsweise in eine andere Datenbank zu transferieren.
- Mit CREATE VIEW und CREATE PROCEDURE erzeugt man eine Ansicht beziehungsweise eine Prozedur aus der aktuellen Abfrage.

13.6 Sonstiges

Zum Schluss unseres Rundgangs durch *IBExpert* wollen wir schnell noch einige weitere Features streifen. Leider werden wir noch nicht einmal in der hier gewählten Kürze alles besprechen können.

Das Script-Fenster

Mit TOOLS | SCRIPT EXECUTIVE öffnen Sie ein Script-Fenster. Auf der linken Seite finden Sie den *Script-Explorer*, der in längeren Scripts eine große Hilfe ist.

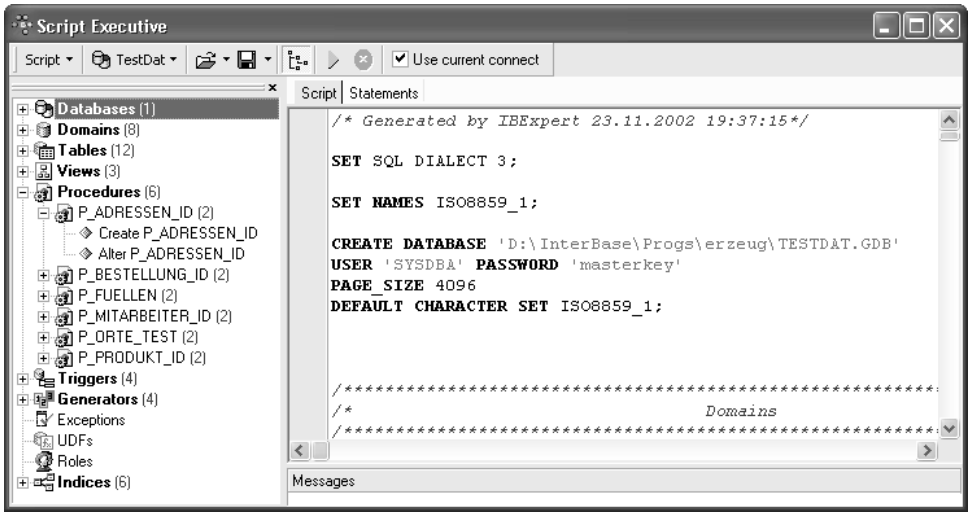


Abbildung 13.32: Das Script-Fenster

Aus einer vorhandene Tabelle können Sie leicht mit TOOLS | EXTRACT METADATA ein SQL-Script erzeugen, dabei kann auch im Einzelnen spezifiziert werden, welches Element in das Script aufgenommen werden soll und welches nicht.

Query Builder

Mit TOOLS | QUERY BUILDER haben Sie eine Möglichkeit, visuell Abfragen zu erstellen. Wir wollen den eben verwendeten JOIN visuell aufbauen.

Gehen Sie dazu wie folgt vor:

- Auf der rechten Seite haben Sie eine Liste aller Tabellen. Führen Sie einen Doppelklick auf *t_bestellung*, *t_adressen* und *t_mitarbeiter* durch, damit diese in der Arbeitsfläche angezeigt werden.
- Markieren Sie die Spalten, die in die Ergebnismenge aufgenommen werden sollen.

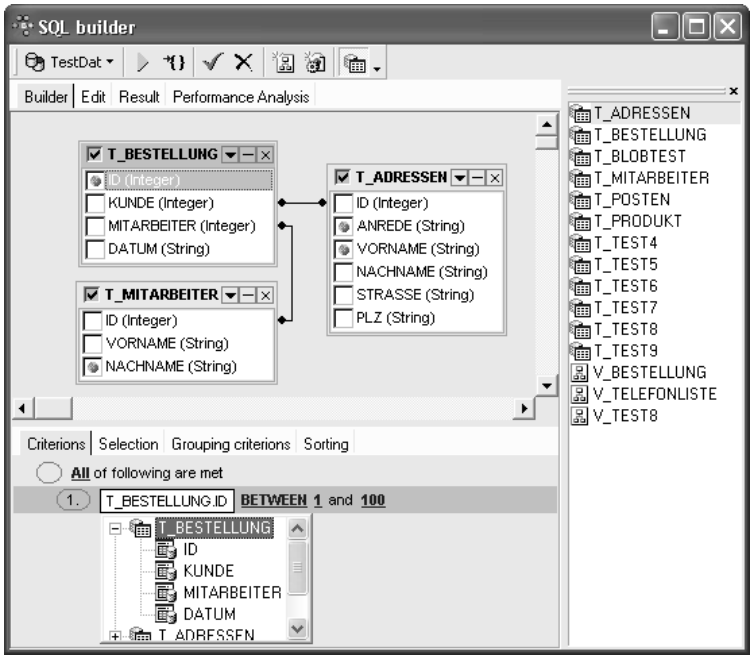


Abbildung 13.33: Einen JOIN visuell aufbauen

- Um einen JOIN herzustellen, ziehen Sie die Spalte der einen Tabelle per Drag&Drop auf die dazugehörende Spalte der anderen Tabelle. Per Voreinstellung wird ein INNER JOIN erstellt, Sie können mit einem Doppelklick auf die Linie einen Editor öffnen, um die Link-Eigenschaften einzustellen.

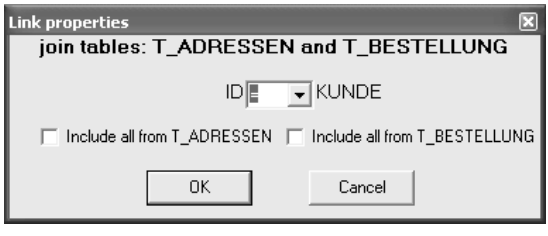


Abbildung 13.34: Bearbeitung einer Verknüpfung

Den Gleichheitsoperator sollten Sie nur dann ändern, wenn Sie genau wissen, was Sie tun.

- Auf der Registerseite *Criterions* finden Sie die Möglichkeit, Filter-Kriterien aufzustellen und zu verknüpfen. Die Spalten-Bezeichner können Sie aus einer Nachschlageliste auswählen, nur Konstanten müssen Sie von Hand eingeben.
- Auf der Registerseite *Sorting* können Sie wählen, nach welchen Spalten auf- oder absteigend sortiert werden soll.

Dokumentation

Mit *Tools|Generate HTML Documentation* können Sie eine Übersicht über die Datenbank erstellen, die auch alle Beschreibungen beinhaltet. Dies ist insbesondere dann recht nützlich, wenn ein Kunde eine ausführliche Dokumentation verlangt.

Abbildung 13.35: Die Dokumentation

Details

Und als Ergänzung, ohne den Anspruch der Vollständigkeit:

- Unter TOOLS|REPORT MANAGER können Sie die Reports verwalten, die Sie den einzelnen Datenbanken zugeordnet haben. Die benutzte Engine ist *FastReport*.
- Mit EDIT|CONVERT IDENTIFIERS/KEYWORDS können Sie in einer SQL-Anweisung die Schlüsselwörter und die Bezeichner getrennt wahlweise in Groß- und in Kleinbuchstaben umwandeln.
- Unter SERVICES|DATABASE STATISTICS erhalten Sie eine Statistik, die Sie auch tabellarisch anzeigen können.
- Unter OPTIONS|ENVIRONMENT OPTIONS können Sie eine andere Sprache wählen und auch sonst zahlreiche Einstellungen vornehmen.
- Es lassen sich STORED PROCEDURES debuggen, also Schritt für Schritt ausführen.

Fazit

IBExpert ist ein sehr mächtiges Werkzeug, mit dem man sehr produktiv arbeiten kann. An einigen wenigen Stellen muss noch ergänzt werden und vor allem wäre eine vollständige Online-Hilfe von Vorteil.

Stichwortverzeichnis

C