2 Der SELECT-Befehl

Mit dem SELECT-Befehl werden die Daten aus der Datenbank abgefragt. Weil das Einfügen, Ändern und Löschen von Daten meist über eine Datenbank-Applikation erfolgt, dürfte der SELECT-Befehl derjenige Befehl sein, der am meisten verwendet wird. Grund genug, diesem Befehl ein ganzes Kapitel zu widmen.

Der vollständig abstrakte SELECT-Befehl lautet wie folgt:

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

Erschrecken Sie nicht über dieses „Ungetüm“, Sie müssen nicht alle diese Optionen auch tatsächlich verwenden. Gewöhnen Sie sich aber schon mal an den Gedanken, daß SQL-Anweisungen manchmal etwas komplizierter werden können.

2.1 Spalten

Wir wollen nun gleich die erste Beispielanweisung erstellen. Dabei gehe ich davon aus, daß Sie ISQL so weit beherrschen, daß Sie zumindest einfache SQL-Anweisungen durchführen können. Die Schritte dazu sind in Angang A aufgeführt.

Die erste Anweisung

Unsere erste SELECT-Anweisung soll folgendermaßen lauten:

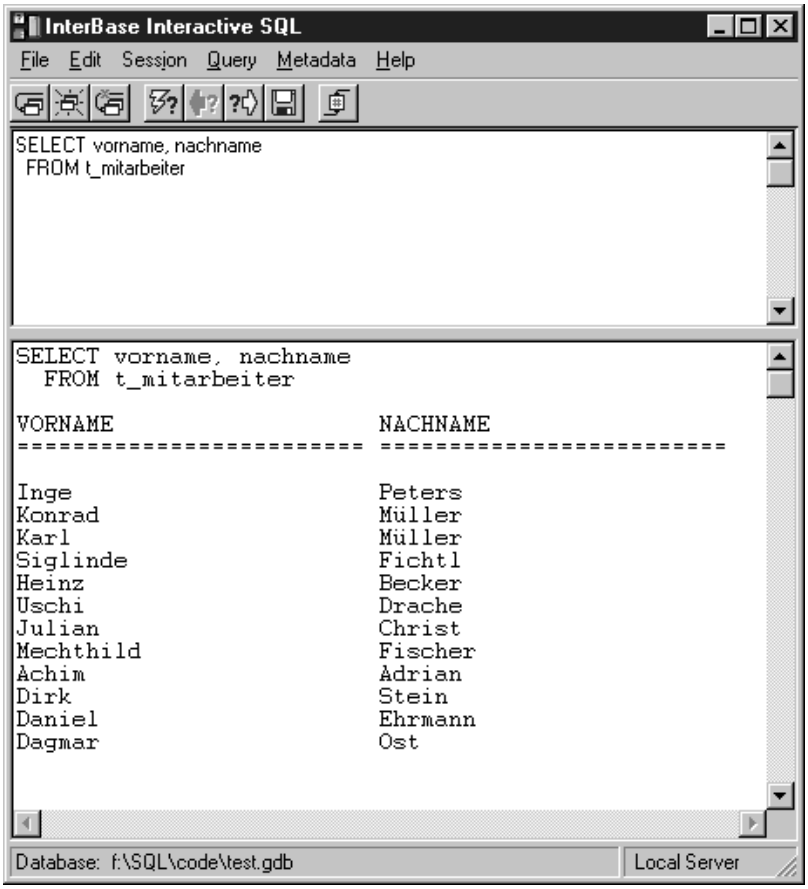

Abbildung2.1: Die erste SELECT-Anweisung

```
SELECT vorname, nachname
   FROM t_mitarbeiter
```

Wenn Sie diese Anweisung eingeben und mit CTRL+ENTER (oder dem betreffenden Speed-Button) starten, dann erhalten Sie ein Ergebnis ähnlich Abbildung 2.1. 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_mitarbeiter
```

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

  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.

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

  Wie Ihnen sicher schon aufgefallen ist, lautet der Tabellenname *t_mitarbeiter*. Das vorangestellt *t_* zeigt an, daß es sich um eine Tabelle handelt. Auch diese Maßnahme ist freiwillig und soll lediglich die Übersicht erhöhen.

- Vielleicht haben Sie sich auch schon gefragt, welche Datensätze denn aus der Tabelle *t_mitarbeiter* angezeigt werden. Nun, solange Sie das nicht irgendwie beschränken, werden alle Datensätze angezeigt. Wenn es sich dabei um einige wenige Datensätze handelt – hier sind es zwölf – dann ist das nicht weiter problematisch. Bei großen Tabellen können es aber leicht einige tausend Datensätze sein, dann würden Sie eben eine Weile warten, bis das System fertig ist. (Sie können das mit *SELECT vorname, nachname FROM t_kunde* gerne einmal ausprobieren.)

Übung 2.1 Apropos ausprobieren: Um den Lerneffekt zu erhöhen, werden Sie nun selbst eine SQL-Anweisung erstellen. Dazu sei verraten, daß es in *t_mitarbeiter* (unter anderem) noch die Spalte *plz* gibt. Erstellen Sie nun eine Anweisung, welche die Nachnamen und die Postleitzahlen anzeigt. Die Lösung finden Sie in Anhang D.

Alle Spalten anzeigen

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

```
SELECT vorname, nachname, strasse,
     plz, ort, vorgesetzter
  FROM t_mitarbeiter
```

Jokerzeichen

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

```
SELECT *
  FROM t_mitarbeiter
```

Allerdings haben Sie dann keinen Einfluß mehr auf die Reihenfolge, in welcher die Spalten angezeigt werden.

Keine doppelten Datensätze

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

```
SELECT plz
  FROM t_mitarbeiter
```

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 hilf uns dann das Schlüsselwort DISTINCT weiter.

```
SELECT DISTINCT plz
  FROM t_mitarbeiter
```

DISTINCT erkennt schon bei einem Unterschied in nur einer Spalte, daß unterschiedliche Datensätze vorliegen. Die folgenden beiden Anweisungen würden also eine identische Ergebnismenge liefern:

```
SELECT nachname, plz
   FROM t_mitarbeiter

SELECT DISTINCT nachname, plz
   FROM t_mitarbeiter
```

Datensätze zählen

Um Datensätze zu zählen, um die Summe, das Maximum, das Minimum oder den Durchschnitt von Spalten zu ermitteln, werden die Aggregatfunktionen verwendet. Diese sollen später noch ausführlich besprochen werden. Vorläufig interessiert uns nur die Funktion COUNT, mit deren Hilfe man Datensätze zählt.

```
SELECT COUNT(*)
   FROM t_mitarbeiter
```

Bei einer Aggregatfunktion muß als Parameter in Klammern angegeben werden, von welcher Spalte die Funktion gebildet werden soll. Beim Zählen der Datensätze in *t_mitarbeiter* ist das reichlich irrelevant, weil alle Spalten vollständig mit Werten belegt sind. Mit *COUNT(*)* wird die Zahl der Datensätze überhaupt ermittelt. Bei dieser Tabelle hätte man beispielsweise auch *COUNT(vorname)* verwenden können – man wäre zu demselben Ergebnis gelangt.

Bei anderen Tabellen macht es durchaus einen Unterschied, welche Spalte man verwendet. Schauen Sie sich zunächst mit *SELECT * FROM t_artikel* die Tabelle *t_artikel* an. In dieser Tabelle sind die Produkte unserer fiktiven Firma aufgeführt. In der Spalte Hersteller gibt es einige Datensätze mit dem Inhalt <*null*>, es ist also dort kein Hersteller angegeben.

Mit der folgenden Anweisung würde man nun die Zahl der Artikel ermitteln:

```
SELECT COUNT(*)
   FROM t_artikel
```

Um die Zahl der Artikel zu ermitteln, bei denen der Hersteller angegeben ist, geht man wie folgt vor:

```
SELECT COUNT(hersteller)
   FROM t_artikel
```

Übung 2.2 Nun etwas zum Knobeln: Ermitteln Sie (mit Hilfe einer SQL-Anweisung) die Zahl der unterschiedlichen Postleitzahlen in *t_mitarbeiter*.

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
```

Sie werden sich nun sicher fragen, was das bezwecken soll. Warten Sie noch einen Abschnitt.

Spalten zusammenfügen

Angenommen, wir wollten ein Rundschreiben an alle Mitarbeiter 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.

Nun, den Ausdruck auf Etiketten können wir von ISQL nicht erwarten, aber das Zusammenfassen von zwei oder mehr Spalten gehört zum Funktionsumfang von SQL:

```
SELECT vorname || nachname,
     strasse,
     plz || ort
  FROM t_mitarbeiter
```

Einen senkrechten Strich fügen Sie mit ALT GR + < ein.

Übung 2.3 Wenn Sie die Ergebnismenge betrachten, dann werden Sie feststellen, daß zwischen den zusammengefügten Werten der beiden Spalten kein Leerzeichen eingefügt wurde. Ändern Sie die Anweisung so ab, daß ein solches Leerzeichen eingefügt wird.

Übung 2.4 Erstellen Sie noch eine SELECT-Anweisung, welche die Spalten *hersteller*, *bezeichnung* und *preis* der Tabelle *t_artikel* anzeigt.

Der Preis soll mit dem Währungszeichen DM versehen werden. (Leser in anderen Staaten verwenden entsprechend andere Währungszeichen, und wer schon ein Euro-Zeichen auf dem Rechner hat, verwendet natürlich dieses.)

Spalten benennen

Vielleicht ist Ihnen schon aufgefallen, daß in ISQL die Spalten nicht mehr benannt werden, sobald Spalten zusammengefügt oder Konstanten verwendet werden. Bei anderen Programmen, beispielsweise bei Report-Generatoren, werden dann sehr „unschöne" Spaltennamen verwendet, beispielsweise *vorname* || " " || *nachname*. 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_mitarbeiter
```

Übung 2.5 Es ist auch möglich, „ganz normale" Spalten umzubenennen. Verwenden Sie die Anweisung aus Übung 2.4 und benennen Sie die Spalte *bezeichnung* in *produkt* um.

Berechnungen ausführen

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

```
SELECT hersteller, bezeichnung,
     preis || " DM" AS brutto,
     (preis / 1.16) || " DM" AS netto
  FROM t_artikel
```

Eine Klammersetzung ist bei Berechnungen nicht generell erforderlich. Würde die Klammern hier jedoch entfernt, würde die betreffende Zeile wie folgt interpretiert:

```
preis / (1.16 || " DM") AS netto
```

Einen String als Divisor akzeptiert das DMS nun einmal nicht und gibt eine entsprechende Fehlermeldung aus. In anderen Fällen würde die Eingabe akzeptiert, aber ein „falsches“ Ergebnis berechnet. Deshalb sollte man im Zweifelsfall lieber ein paar Klammern zu viel als zu wenig setzen.

Unter SQL stehen Rechenoperatoren für die vier Grundrechenarten zur Verfügung:

Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division

Tabelle2.1: Die Rechenoperatoren

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

Übung 2.6 Anläßlich des dreijährigen Bestehens unserer fiktiven Firma sollen alle Produkte um drei DM billiger verkauft werden. Erstellen Sie die entsprechende Preisliste und benennen Sie die Preisspalte mit *Sonderpreis*.

Anzeigen mit zwei Nachkommastellen

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

Eine dafür geeignete Funktion kennt *InterBase* leider nicht. Selbstverständlich wäre es möglich, eine entsprechende UDF (*User Defined Function*) zu programmieren, aber dazu wären eine entsprechende Entwicklungsumgebung und einige Programmierkenntnisse erforderlich.

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,
     CAST(preis AS NUMERIC(15,2)) AS brutto,
     CAST((preis / 1.16) AS NUMERIC (15,2))  AS netto
  FROM t_artikel
```

Leider erfolgt hier eine kleine Einschränkung: Wenn Sie jetzt noch versuchen würden, die Konstante *DM* anzuhängen, dann werden die Werte wieder mit allen Nachkommastellen angezeigt. Schade aber auch.

Übrigens: Wenn ich die Spalte nicht als FLOAT, sondern als NUMERIC (15,2) definiert hätte, würde das Problem gar nicht erst auftreten. Als „gelernter Delphi-Programmierer" weiß ich allerdings, daß grundsätzlich alle Nachkommastellen abgeschnitten werden, wenn ich Felder des Typs NUMERIC über die BDE importiere.

2.2 Joins

Wie Sie spätestens seit Kapitel 1 wissen, wird bei relationalen Datenbanken 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 kennenlernen werden.

2.2.1 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 kennenlernen werden.

Wie Ihnen vielleicht schon aufgefallen ist, enthält die Tabelle *t_mitarbeiter* keine Telefonnummern. Diese sind in der Tabelle *t_tele* enthalten. Verschaffen Sie sich mit *SELECT * FROM t_tele* einen Überblick über diese Tabelle.

Um diese Tabelle mit *t_mitarbeiter* zu verknüpfen, erstellen wir zunächst einen (hier völlig ungeeigneten) FULL JOIN:

```
SELECT
     t_mitarbeiter.vorname, t_mitarbeiter.nachname,
        t_tele.bezeichnung
   FROM t_mitarbeiter, t_tele
```

Schauen wir uns den Anfang der Ergebnismenge an:

```
VORNAME   NACHNAME        BEZEICHNUNG
========= ============== ====================

Inge      Peters         40
Inge      Peters         030 / 222 27 88
Inge      Peters         0177 / 412 76 41
Inge      Peters         2
Inge      Peters         030 / 182 22 86
Inge      Peters         0177 / 771 33 43
Inge      Peters         5
Inge      Peters         030 / 724 25 73
Inge      Peters         0177 / 422 35 35
Inge      Peters         12
Inge      Peters         030 / 124 56 45
Inge      Peters         0177 / 778 53 17
Inge      Peters         60
Inge      Peters         030 / 732 15 45
Inge      Peters         0177 / 315 61 32
Inge      Peters         52
Inge      Peters         030 / 181 63 55
Inge      Peters         0177 / 863 64 42
Konrad    Müller         40
Konrad    Müller         030 / 222 27 88
Konrad    Müller         0177 / 412 76 41
Konrad    Müller         2
```

Wie Sie sehen, wird jeder Datensatz der einen Tabelle mit jedem Datensatz der anderen Tabelle verknüpft – jeder Mitarbeiter erhält somit alle Telefonnummern der ganzen Belegschaft. Ein FULL JOIN ist also nicht ganz das, was hier benötigt wird.

2.2.2 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
     t_mitarbeiter.vorname, t_mitarbeiter.nachname,
        t_tele.bezeichnung
  FROM t_mitarbeiter, t_tele
  WHERE t_mitarbeiter.nummer = t_tele.mitarbeiter
```

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: Jeder Datensatz in der Tabelle *t_tele* enthält eine Information darüber, zu welchem Mitarbeiter die Telefonnummer gehört, nämlich die Datensatznummer des betreffenden Mitarbeiters in der Tabelle *t_mitarbeiter*. Über diese beiden Spalten wird mit Hilfe der WHERE-Klausel die Verknüpfung hergestellt.

Übrigens: Mit Hilfe der WHERE-Klausel werden wir ab Kapitel 2.3 Datensätze filtern. (Beispielsweise: Zeige mir alle Kunden an, die aus Berlin kommen.) Bei sehr umfangreichen SQL-Anweisungen kann man dann leicht ein wenig durcheinanderkommen, 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 statt dessen eine ON-Klausel verwendet. Diese Schreibweise ist an den OUTER JOIN angelehnt, den wir ab Kapitel 2.2.3 besprechen.

```
SELECT
     t_mitarbeiter.vorname, t_mitarbeiter.nachname,
        t_tele.bezeichnung
  FROM t_mitarbeiter
     INNER JOIN t_tele
        ON t_mitarbeiter.nummer = t_tele.mitarbeiter
```

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
     vorname, nachname, bezeichnung
  FROM t_mitarbeiter
     INNER JOIN t_tele
        ON t_mitarbeiter.nummer = mitarbeiter
```

Das funktioniert aber auch nur so lange, als die verwendeten Spaltenbezeichner nur in einer der beiden Tabellen vorkommen. Die Tabellen *t_mitarbeiter* und *t_tele* enthalten dagegen beide die Spalte *nummer*, und das zwingt uns, in der ON- oder der WHERE-Klausel bei dieser Spalte die Tabelle explizit anzugeben.

Tabellen-Alias

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

```
SELECT
     m.vorname, m.nachname, t.bezeichnung
  FROM t_mitarbeiter m
     INNER JOIN t_tele t
        ON m.nummer = t.mitarbeiter
```

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. Statt beispielweise *t_mitarbeiter.vorname* schreiben Sie dann nur noch *t.vorname*.

Übung 2.7 Schreiben Sie die eben genannte SQL-Anweisung in einen JOIN mit WHERE-Klausel um. Verwenden Sie dabei die Tabellen-Aliase *mit* und *te*.

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.

Zurück zu unserer Mitarbeiter-Telefonliste. Sie fragen sich sicher schon längst, warum denn manche Telefonnummern nur zwei Stellen umfassen. Nun, dabei handelt es sich um die interne Durchwahl in der Firma.

Wenn man das weiß, ist es ja sofort einsichtig. Wir wollen die Information darüber, was denn hinter den einzelnen Telefonnummern steckt, nun aber auch in die Telefonliste aufnehmen. Wenn Sie sich noch an die Tabelle *t_tele* erinnern, dann wissen Sie auch um die Spalte *art*. (Kleiner Tip am Rande: Mit *SELECT * FROM t_tele* können Sie Ihr Gedächtnis auffrischen.)

Da in der Spalte *art* nur Nummern zu finden sind, ahnen Sie sicher auch schon, daß hier wieder die Verknüpfung zu einer weiteren Tabelle erfolgt. Dabei handelt es sich um die Tabelle *t_art*.

```
NUMMER     BEZEICHNUNG
========   ========================

1          Durchwahl in Firma
2          Telefon privat
3          Fax privat
4          Handy
5          e-Mail in Firma
6          e-Mail privat
7          Telefon bei Mutti
```

Um nun die gewünschte Liste zu erstellen, verwenden Sie die folgende Anweisung:

```
SELECT
     m.vorname, m.nachname,
     a.bezeichnung, t.bezeichnung
  FROM t_mitarbeiter m, t_tele t, t_art a
  WHERE (m.nummer = t.mitarbeiter)
     AND (a.nummer = t.art)
```

Erwartungsgemäß sind bei drei Tabellen zwei Verknüpfungsbedingungen erforderlich. Da das DMS aber nur eine WHERE-Klausel erwartet, müssen die beiden Bedingungen mit AND verknüpft werden.

Übung 2.8 Nun wieder etwas zum Knobeln: Erstellen Sie den JOIN über die drei Tabellen mit Hilfe von ON-Klauseln. Beachten Sie, daß – im Gegensatz zur WHERE-Klausel – mehrere ON-Klauseln erlaubt sind.

2.2.3 OUTER JOIN

Vielleicht ist Ihnen schon aufgefallen, daß wir bei unserer Telefonliste ein paar Mitarbeiter „verloren“ haben, nämlich alle, zu denen keine Telefonnummern gespeichert sind; diese würden ja auf einer Telefonliste auch keinen Sinn machen.

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

Nun wollen wir jedoch eine Mitarbeiterliste erstellen, in der auch die Telefonnummer (soweit bekannt) der Mitarbeiter aufgenommen werden. Da eine Mitarbeiterliste für gewöhnlich aber alle Mitarbeiter umfaßt, können wir hier keinen INNER JOIN verwenden.

```
SELECT
     m.vorname, m.nachname, t.bezeichnung
  FROM t_mitarbeiter m
     LEFT OUTER JOIN t_tele t
        ON m.nummer = t.mitarbeiter
```

Bei einem LEFT OUTER JOIN werden alle Datensätze der Tabelle „auf der linken Seite“ verwendet, hier im Beispiel also *t_mitarbeiter*. Diese werden dann – soweit möglich – mit den Datensätzen der rechten Seite, hier im Beispiel also *t_tele*, 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.

```
SELECT
     m.vorname, m.nachname, t.bezeichnung
  FROM t_mitarbeiter m
     RIGHT OUTER JOIN t_tele t
        ON m.nummer = t.mitarbeiter
```

Hier würden nun Telefonnummen angezeigt und den Mitarbeitern zugeordnet. Es würden auch Telefonnummern angezeigt, zu denen kein passender Mitarbeiter-Datensatz existiert. Solche Datensätze sind allerdings in *t_tele* nicht vorhanden, so daß die Ergebnismenge der eines INNER JOINs gleichen würde.

Prinzipiell kann man auch INNER und OUTER JOINs mischen. Man muß allerdings damit rechnen, nicht das gewünschte Ergebnis zu erhalten, beispielsweise, wenn man in der Mitarbeiterliste anzeigen wollte, welcher Art die angegebenen Telefonnummern sind:

```
SELECT
     m.vorname, m.nachname,
     t.bezeichnung, a.bezeichnung
  FROM t_mitarbeiter m
     LEFT OUTER JOIN t_tele t
        ON m.nummer = t.mitarbeiter
     INNER JOIN t_art a
        ON t.art = a.nummer
```

Die Verwendung des INNER JOINs würde dazu führen, daß nur die Mitarbeiter mit Telefon angezeigt würden.

Abhilfe schafft hier ein Verändern der Tabellenreihenfolge:

```
SELECT
     m.vorname, m.nachname,
     t.bezeichnung, a.bezeichnung
  FROM t_tele t
     INNER JOIN t_art a
        ON t.art = a.nummer
     RIGHT OUTER JOIN t_mitarbeiter m
        ON m.nummer = t.mitarbeiter
```

Sie können sich folgende Regel merken: Ein JOIN ist immer eine Verbindung der vorherigen Ergebnismenge (Tabelle oder JOIN) und der dazugefügten Tabelle.

Um die Reihenfolge zu steuern, in welcher die JOINs erstellt werden, kann man Klammern setzen:

```
SELECT
     m.vorname, m.nachname,
     t.bezeichnung, a.bezeichnung
   FROM t_mitarbeiter m
     LEFT OUTER JOIN (t_tele t
          INNER JOIN t_art a
             ON t.art = a.nummer)
        ON m.nummer = t.mitarbeiter
```

Man kann auch bei der zweiten Verknüpfung ein LEFT OUTER JOIN verwenden:

```
SELECT
     m.vorname, m.nachname,
     t.bezeichnung, a.bezeichnung
   FROM t_mitarbeiter m
     LEFT OUTER JOIN t_tele t
        ON m.nummer = t.mitarbeiter
     LEFT OUTER JOIN t_art a
        ON t.art = a.nummer
```

Übung 2.9 Stellen Sie diese SQL-Anweisung so um, daß lauter RIGHT OUTER JOINS verwendet werden.

FULL OUTER JOIN

Bei einem FULL OUTER JOIN werden alle Datensätze der beteiligten Tabellen angezeigt. Wo es möglich ist, werden Verknüpfungen vorgenommen.

Ein sinnvolles Beispiel gibt es in unserer Datenbank nicht. Die folgende Anweisung würde nicht nur Mitarbeiter ohne Telefon, sondern auch Telefonnummern ohne dazugehörenden Mitarbeiter auflisten:

```
SELECT
     m.vorname, m.nachname,
     t.bezeichnung
  FROM t_mitarbeiter m
     FULL OUTER JOIN t_tele t
        ON m.nummer = t.mitarbeiter
```

Nun wird jedoch in der Tabelle *t_tele* per Referenz dafür gesorgt, daß es eben keine Einträge gibt, die nicht auf einen vorhandenen Mitarbeiter-Datensatz verweisen.

Wenn Sie irgendwo einen FULL OUTER JOIN sinnvoll einsetzen können, dann könnte das ein Hinweis darauf sein, daß die Datenbank etwas schlampig konzipiert wurde.

2.2.4 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? Ein Beispiel: In unserer Tabelle *t_mitarbeiter* gibt es die Spalte *vorgesetzter*, in welchem die Personennummer des jeweils direkten Vorgesetzten gespeichert ist. Wer sich hinter dieser Nummer verbirgt, wird wieder anhand der Tabelle *t_mitarbeiter* ermittelt.

```
SELECT m.vorname, m.nachname,
     v.vorname || " " || v.nachname  AS vorgesetzter
  FROM t_mitarbeiter m, t_mitarbeiter v
  WHERE m.vorgesetzter = v.nummer
```

Übung 2.10 Handelt es sich bei diesem SELF JOIN um einen INNER JOIN, einen LEFT OUTER JOIN, einen RIGHT OUTER JOIN oder um einen FULL OUTER JOIN? Schreiben Sie den SELF JOIN so um, daß keine WHERE-, sondern eine ON-Klausel verwendet wird.

2.3 WHERE

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

Das Filtern von Datenbeständen wird erst bei sehr großen Datenbeständen so richtig interessant. Deshalb werden wir dafür vor allem die Tabelle *t_kunde* verwenden – die Kundenkartei unserer fiktiven Firma.

Bei dieser Tabelle sollten Sie eine Anweisung wie *SELECT * FROM t_kunde* vermeiden, weil das System dann eine ganze Weile lang beschäftigt wäre.

Übung 2.11 Ermitteln Sie die Zahl der Datensätze in *t_kunde*.

Mit der folgenden Anweisung ermitteln wir den Datensatz mit der Nummer 100:

```
SELECT *
   FROM t_kunde
   WHERE nummer = 100
```

Mit der Anweisung *WHERE nummer = 100* beschränken wir die Anzeige auf diejenigen Datensätze, bei denen das Feld *nummer* den Wert 100 hat. Da in *nummer* eine durchlaufende Nummer gespeichert ist, wird maximal ein Datensatz angezeigt.

Mit der eben genannten SELECT-Anweisung wird übrigens nicht unbedingt der hundertste Datensatz angezeigt. Dies wäre nur dann der Fall, wenn in der Tabelle Datensätze mit den Nummern eins bis hundert gespeichert wären. Genausogut kann es sein, daß in einer Tabelle von mehr als hundert Datensätzen keiner die Bedingung *nummer = 100* erfüllt, sei es, daß die Zählung nicht mit eins begonnen wurde, sei es, daß ein betreffender Datensatz bereits gelöscht wurde.

Weil SQL mengen- und nicht satzorientiert arbeitet, gibt es übrigens keine Möglichkeit, den hundertsten (ersten, letzten) Datensatz zu ermitteln.

Suche nach Strings

Selbstverständlich kann in SQL nicht nur nach Nummern, sondern auch nach Strings gefiltert werden. Beachten Sie bitte, daß String-Konstanten in Anführungszeichen zu setzen sind, damit sie das DMS von Bezeichnern unterscheiden kann.

```
SELECT *
   FROM t_kunde
   WHERE nachname = "Weber"
```

Übung 2.12 Ermitteln Sie die Zahl der Datensätze in der Tabelle *t_kunde* mit dem Vornamen *Doris*.

2.3.1 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 kennengelernt.

Die AND-Verknüpfung

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

```
SELECT *
   FROM t_kunde
   WHERE (vorname = "Michael")
      AND (nachname = "Weber")
```

In dieser Anweisung wird nach den Kunden namens *Michael Weber* gesucht. Beachten Sie, daß die einzelnen Filterbedingungen in Klammern zu setzen sind.

Übung 2.13 Suchen Sie die Telefonnummern aller Damen namens *Stefanie*, die in *Hamburg* wohnen.

Die OR-Verknüpfung

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

```
SELECT *
   FROM t_kunde
   WHERE (vorname = "Leonore")
      OR (vorname = "Ernst")
```

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

Übung 2.14 Suchen Sie aller Kunden mit den Namen *Maier*, *Meier*, *Meyer* und *Meyr*.

Die NOT-Verknüpfung

Mit Hilfe des Schlüsselwortes NOT wird eine Suchbedingung negiert.

```
SELECT vorname, nachname
   FROM t_mitarbeiter
   WHERE NOT (nummer = vorgesetzter)
```

Mit dieser Anweisung werden alle Mitarbeiter ermittelt, die einem anderen untergeben sind, bei denen also nicht die Felder *nummer* und *vorgesetzter* denselben Wert enthalten.

Übung 2.15 Erstellen Sie die Telefonliste (Liste der Mitarbeiter mit Telefon) mit Vor- und Nachname, Art der Nummer und der Nummer selbst, ignorieren Sie dabei alle Handy-Nummern. Verwenden Sie einen JOIN mit ON-Klausel.

Kombination logischer Verknüpfung

Sie können AND-, OR- und NOT-Klauseln 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_kunde
   WHERE (((vorname = "Daniel")
           AND (nachname = "Ost"))
         OR ((vorname = "Stefanie")
           AND (nachname = "Busch")))
      AND NOT (ort = "Berlin")
```

Diese Anweisung ermittelt alle Kunden namens *Daniel Ost* 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_kunde
   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_kunde
   WHERE (((vorname = "Daniel")
           AND (nachname = "Ost"))
         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_kunde
   WHERE ((vorname || " " || nachname = "Daniel Ost")
        OR (vorname||" "||nachname = "Stefanie Busch"))
      AND NOT (ort = "Berlin")
```

Übung 2.16 Ermitteln Sie alle Kunden mit den Namen *Sigmund Maier* und *Tina Maier*, die weder in *Altshausen* noch in *Mainz* wohnen.

2.3.2 Die Vergleichs-Operatoren

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

Beachten Sie bitte, daß nicht alle der hier vorgestellten Operatoren in allen SQL-Dialekten verfügbar sind.

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_kunde
   WHERE (nummer < 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_kunde
  WHERE (nummer <= 100)

SELECT *
  FROM t_kunde
  WHERE (nummer < 101)

SELECT *
   FROM t_kunde
   WHERE (nummer < 100)
      OR (nummer = 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_kunde
   WHERE (101 > nummer)
```

Und die erste so:

```
SELECT *
   FROM t_kunde
   WHERE (100 >= nummer)
```

Beachten Sie jedoch, daß bei *kleiner/gleich*- und *größer/gleich*-Operatoren das Gleichheitszeichen immer an zweiter Stelle steht.

Übung 2.17 Nun gleich zwei Übungen: Zunächst alle Kunden-Datensätze über (einschließlich) 10000. Und dann alle Kunden-Datensätze zwischen (einschließlich) 500 und 600.

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_kunde
   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 kennenlernen werden.

Der BETWEEN-Operator

Aus Übung 2.17 kennen Sie das Problem, alle Datensätze zwischen 500 und 600 (einschließlich) zu ermitteln. Wir sind vermutlich derselben Ansicht, daß man das eigentlich etwas prägnanter formulieren können müßte. Dazu dient der Operator BETWEEN.

```
SELECT *
   FROM t_kunde
   WHERE (nummer BETWEEN 500 AND 600)
```

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

Übung 2.18 Stellen Sie auch die Suche aller Berliner Kunden, deren Nachname mit *M* beginnt, auf den Operator BETWEEN um.

Der LIKE-Operator

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

```
SELECT *
   FROM t_kunde
   WHERE (nachname LIKE "M%")
      AND (ort = "Berlin")
```

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

Übung 2.19 Ermitteln Sie alle in der Kundentabelle vorkommenden Nachnamen, die mit *er* enden.

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 in Berlin, deren Nachnamen *Müller*, *Miller*, *Möller* oder so ähnlich lauten.

```
SELECT *
   FROM t_kunde
   WHERE (nachname LIKE "M_ller")
      AND (ort = "Berlin")
```

Übrigens: Wenn Sie bei WHERE-Klauseln wie *WHERE (nachname = "Müller")* eine Fehlermeldung ob nicht konvertierbarer Zeichensätze erhalten, dann müssen Sie unter SESSION | ADVANCES SETTINGS den Zeichensatz einstellen, auf den die Datenbank aufbaut, bei unserer Beispieldatenbank *ISO8859_1*.

Das Thema „Ausführungsgeschwindigkeit von Abfragen“ wollen wir hier nicht ausführlicher behandeln. Ich möchte Sie aber darauf hinweisen, daß bei der Suche nach dem oder den Anfangsbuchstaben eine Konstruktion mit LIKE langsamer ist als mit anderen Operatoren. Besehen Sie sich in diesem Zusammenhang besonders den Operator STARTING WITH.

Übung 2.20 Ermitteln Sie alle Berliner Kunden, deren zweiter Buchstabe im Vornamen ein *u* ist.

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 groß geschrieben wurde. Dies erreich 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")
```

Der Haken an der Sache: Bei Umlauten funktioniert das Ganze nicht, auch nicht bei korrekt eingestelltem Zeichensatz:

```
SELECT nummer, nachname
   FROM t_kunde
   WHERE (UPPER(nachname) = UPPER("müLLER"))
      AND (ort = "Berlin")

/* geht nicht */
```

Und auch der Trick mit den Joker-Zeichen versagt hier:

```
SELECT nummer, UPPER(nachname)
   FROM t_kunde
   WHERE (UPPER(nachname) = UPPER("m_LLER"))
      AND (ort = "Berlin")

/* geht nicht */
```

Das Problem liegt daran, daß Konstanten nicht entsprechend dem eingestellten Zeichensatz umgewandelt werden:

```
SELECT nummer, UPPER(nachname), UPPER("müller")
   FROM t_kunde
   WHERE (UPPER(nachname) = "MÜLLER")
      AND (ort = "Berlin")


...
```

```
1020 MÜLLER     MüLLER
1415 MÜLLER     MüLLER
3229 MÜLLER     MüLLER
```

Übrigens: Seltsamerweise funktioniert folgende Konstruktion:

```
WHERE (UPPER(nachname)
   = UPPER("m") || "Ü" || UPPER("ller"))
```

nicht aber folgende:

```
WHERE (UPPER(nachname)
   = UPPER("m") || "_" || UPPER("ller"))
WHERE (UPPER(nachname)
   = UPPER("m") || "%" || UPPER("ller"))
```

Sie werden sich nun sicher fragen, warum ich so auf diesem Thema herumreite, wo es doch auch die Möglichkeit gibt, die Sache in Großbuchstaben einzugeben. Das ist im Moment richtig, aber es gibt durchaus Situationen, in denen Sie sich „dumm und dämlich“ suchen können, wenn Sie den Fehler nicht bei der Funktion UPPER suchen.

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_kunde
   WHERE nummer LIKE "233%"
```

Übung 2.21 Bei wie vielen Kunden endet die Kundennummer mit 99?

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. Wie groß die Unterschiede sind, hängt vom Datenbestand und vom DMS ab: Bei InterBase sind die Unterschiede deutlich, aber nicht weltbewegend (beispielsweise 1,7 s statt 2,6 s). Bei anderen DMS mag das anders aussehen.

```
SELECT *
   FROM t_kunde
   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.

Übung 2.22 Ein Kunde aus Berlin, der seinen Namen mit E.M. abkürzt, hat geschrieben. Welche Kunden kommen in Frage? Tragen Sie dem Umstand Rechnung, daß Sie nicht genau wissen, ob der erste Buchstabe für den Vor- oder für den Nachnamen steht.

Der CONTAINING-Operator

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

```
SELECT *
   FROM t_kunde
   WHERE (nachname CONTAINING "ä")
      AND (ort = "Berlin")
```

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

Übung 2.23 Jetzt mal wieder zwei Übungen: Schreiben Sie die eben erwähnte SQL-Anweisung so um, daß ein LIKE-Operator anstatt des CONTAINING-Operators verwendet wird.

Ermitteln Sie (mit Hilfe des CONTAINING-Operators) alle Kunden, deren Kundennumer drei aufeinanderfolgende 9-Zeichen enthält.

Der IS NULL-Operator

Der Wert NULL steht in SQL für nicht zugewiesene Werte. Um zu prüfen, ob ein Datensatz in einer Spalte leere Felder hat, muß der Operator IS NULL verwendet werden.

```
SELECT *
   FROM t_artikel
   WHERE hersteller IS NULL
```

Diese Anweisung zeigt alle Artikel an, bei denen keine Herstellerangabe vorhanden ist.

Wie die folgende Anweisung zeigt, sind ein leerer String und der Wert NULL etwas Verschiedenes:

```
SELECT *
   FROM t_artikel
   WHERE (hersteller IS NULL)
      OR (hersteller = "")
```

Für die Negation von IS NULL gibt es zwei Möglichkeiten:

```
SELECT *
   FROM t_artikel
   WHERE NOT (hersteller IS NULL)


SELECT *
   FROM t_artikel
   WHERE hersteller IS NOT NULL
```

Der Operator IS NOT NULL erspart das Setzen einer Klammer und hält – zumindest nach meinem Empfinden – die Anweisung übersichtlicher.

Übung 2.24 Nun wieder etwas zum Knobeln: Erstellen Sie die Liste aller Mitarbeiter mit den Spalten Vorname, Nachname, Telefonnummer, Art des Anschlusses und Vorgesetzter mit vollem Namen. Die Anzeige der Telefonnummer soll auf Durchwahl und Handy beschränkt werden. Selbstverständlich sollen auch diejenigen Mitarbeiter aufgenommen werden, zu denen keine Telefonnummern gespeichert sind. Viel Vergnügen ;-)

Der IN-Operator

Übung 2.25 Damit Sie die Erleichterung des IN-Operators so richtig schätzen werden, wollen wir das nächste Problem zunächst ohne ihn bearbeiten: Ermitteln Sie alle Berliner Kunden namens *Regina Müller*, *Hans Müller* und *Petra Müller*.

Sie werden mir sicher zustimmen, daß man dieses noch ein wenig einfacher formulieren können sollte – mit Hilfe des IN-Operators ist das möglich.

```
SELECT *
   FROM t_kunde
   WHERE (vorname IN ("Regina", "Hans", "Petra"))
      AND(nachname = "Müller")
      AND (ort = "Berlin")
```

Nach dem IN-Operator folgt in Klammern die Menge der Elemente, die das Suchkriterium definieren.

Übung 2.26 Zeigen Sie die Kunden mit den Nummern 15, 647 und 3456 an.

2.4 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 2.2 aufgeführten statistischen Funktionen definiert.

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

Tabelle 2.2: Die statistischen Funktionen in SQL

Die Funktion COUNT kennen Sie ja bereits, nun wollen wir auch die anderen Funktionen verwenden.

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

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

Übrigens: Die Anzahl der Nachkommastellen hängt offensichtlich von der Spaltenbreite und diese wiederum von der Spaltenüberschrift ab:

```
SELECT
     AVG(preis) AS durchschnittlicher_Preis
  FROM t_artikel

DURCHSCHNITTLICHER_PREIS
========================

      425.96429443359375


SELECT
     AVG(preis)
  FROM t_artikel

           AVG
===============

      425.96429
```

Übung 2.27 Lassen Sie die statistischen Werte von allen Artikeln berechnen, die vom Hersteller *ELSA* stammen.

2.4.1 Daten gruppieren

Wie Sie den durchschnittlichen Preis aller Artikel berechnen, ist Ihnen inzwischen bekannt. Wenn Sie den durchschnittlichen Preis aller Artikel eines Herstellers wissen wollen, dann setzen Sie eine WHERE-Klausel ein, das kennen Sie ja aus Übung 2.27.

Nun wollen wir aber die Hersteller in diesem Punkt vergleichen, und da wäre es ein wenig aufwendig, viele einzelne SQL-Anweisungen einzugeben und das Ergebnis auf einem Blatt Papier zu notieren. Deshalb gibt es die GROUP BY-Klausel.

```
SELECT
     hersteller, AVG(preis)
  FROM t_artikel
  GROUP BY hersteller
```

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?

Übung 2.28 Wie viele Artikel liefern uns die einzelnen Hersteller?

Gruppieren nach mehreren Kriterien

Die Tabelle *t_artikel* enthält in der Spalte *gruppe* eine Referenz auf die Tabelle *t_gruppe*. Auf diese Weise kann angezeigt werden, ob es sich bei einem Artikel um Speicher, um eine Festplatte oder was auch immer handelt.

Übung 2.29 Erstellen Sie eine Liste aller Artikel mit Angabe der Produktgruppe. Verwenden Sie dabei einen JOIN mit einer ON-Klausel. Stellen Sie den Preis mit zwei Nachkommastellen dar.

Nun wollen wir wieder wissen, wie viele Artikel wir von den einzelnen Herstellern beziehen, die Liste soll aber zusätzlich nach den Gruppenbezeichnungen gruppiert werden.

```
SELECT
     a.hersteller, g.bezeichnung, COUNT(a.preis)
  FROM t_artikel a
     INNER JOIN t_gruppe g
        ON a.gruppe = g.nummer
  GROUP BY a.hersteller, g.bezeichnung
```

Beachten Sie noch einmal, daß alle anzuzeigenden Spalten, die nicht auf statistischen Funktionen beruhen, explizit in der GROUP BY-Klau-

sel angegeben werden müssen.

Übung 2.30 Nun wieder zwei Übungen: Ermitteln Sie, wie viele Artikel in den einzelnen Produktgruppen vorhanden sind. Ermitteln Sie, wieviel das billigste und das teuerste Produkt der einzelnen Hersteller kostet.

2.4.2 Die HAVING-Klausel

Beim zweiten Teil der Übung 2.30 werden Sie feststellen, daß bei manchen Herstellern das billigste genauso viel wie das teuerste Produkt kostet – von diesen Herstellern vertreiben wir nur ein Produkt.

Nun wollen wir diese Hersteller von der Anzeige ausschließen. Die Verwendung der WHERE-Klausel scheidet dabei aus, weil in dieser keine Aggregat-Funktionen erlaubt sind. Zum Einsatz kommt hier die HAVING-Klausel.

```
SELECT
     a.hersteller,
     MIN(a.preis),
     MAX(a.preis)
  FROM t_artikel a
  GROUP BY a.hersteller
     HAVING COUNT(preis) > 1
```

Mit dieser HAVING-Klausel wird die Anzeige auf diejenigen Hersteller beschränkt, von denen wir mehr als einen Artikel vertreiben.

Übung 2.31 Mit der eben erwähnten HAVING-Klausel würden wir nicht diejenigen Fälle erfassen, bei denen wir von einem Hersteller mehrere Artikel, diese aber zum selben Preis vertreiben. Ändern Sie die Anweisung so ab, daß auch diese Hersteller von der Anzeige ausgeschlossen werden.

Im übrigen sei darauf hingewiesen, daß es egal ist, ob die Aggregat-Funktion der HAVING-Klausel in die Anzeige aufgenommen ist oder nicht.

HAVING ohne Aggregat-Funktionen

Die HAVING-Klausel ist nicht auf Aggregat-Funktionen beschränkt.

Mit ihr kann auch – ähnlich der WHERE-Klausel – der Datenbestand gefiltert werden.

Mit der folgenden Anweisung schließen wir den Hersteller *Elsa* aus der Anzeige aus.

```
SELECT
     a.hersteller,
     MIN(a.preis),
     MAX(a.preis)
  FROM t_artikel a
  GROUP BY a.hersteller
     HAVING a.hersteller <> "Elsa"
```

Übung 2.32 Ändern Sie die Anweisung so ab, daß Sie eine WHERE-Klausel statt einer HAVING-Klausel verwenden.

2.5 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 Namen der Berliner Kunden mit den Vornamen *Eugen* und *Stefanie* nach dem Nachnamen sortieren:

```
SELECT vorname, nachname
   FROM t_kunde
   WHERE ort = "Berlin"
      AND vorname IN ("Eugen", "Stefanie")
   ORDER BY nachname
```

Übung 2.33 Erstellen Sie eine Liste mit Hersteller, Produkt, Gruppe und Preis (mit zwei Nachkommastellen). Sortieren Sie die Liste nach dem Hersteller.

Absteigend sortieren

Bislang haben wir die Reihen aufsteigend, also von *a* nach *z* sortiert.

Mit Hilfe des Schlüsselwortes DESC können wir absteigend sortieren.

```
SELECT
     nummer, vorname, nachname
  FROM t_kunde k
  WHERE ort = "Bonn"
     AND nachname = "Maier"
  ORDER BY vorname DESC
```

Beachten Sie, daß *DESC* dem Spaltenbezeichner nachgestellt wird. Mit der WHERE-Klausel schränken wir die Anzeige auf die Bonner Kunden namens *Maier* ein.

Sie werden sich vielleicht fragen, warum ich den Datenbestand in den Beispielen immer einschränke: Zum einen wird dadurch die Anzeige schneller, weil nicht mehr so viel Zeilen übertragen werden müssen, zum anderen besteht dann noch die Chance, daß sich alle Zeilen anzeigen lassen. Und außerdem lernen Sie auf diese Weise, wie man die verschiedenen Klauseln der SELECT-Anweisung kombiniert.

Nach mehreren Spalten sortieren

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

```
SELECT
     nummer, nachname, vorname
  FROM t_kunde k
  WHERE ort = "Bonn"
     AND nachname < "D"
  ORDER BY nachname, vorname
```

Übung 2.34 Ermitteln Sie die Namen aller Berliner Kunden, deren Nachname mit den Buchstaben *N* und *M* beginnen. Fügen Sie dabei Vor- und Nachname zu einer Spalte zusammen. Sortieren Sie absteigend nach dem Nachnamen und dann aufsteigend nach dem Vornamen.

Nach Aggregatfunktionen sortieren

Nach den Ergebnisspalten von Aggregatfunktionen kann eine Abfra-

ge nicht sortiert werden. In einigen wenigen Fällen erreicht man das gewünschte Ergebnis, indem man die darunterliegende Spalte sortiert:

```
SELECT
     a.hersteller,
     MIN(a.preis) AS Minimum,
     MAX(a.preis) AS Maximum
   FROM t_artikel a
   GROUP BY a.hersteller
   ORDER BY preis
```

Konstruktionen wie

```
   ORDER BY minimum
```

oder

```
   ORDER BY MIN(preis)
```

werden leider nicht akzeptiert.

Ansonsten kann man eine STORED PROCEDURE erstellen. Das werden wir später besprechen, an dieser Stelle schon einmal unkommentiert die Vorgehensweise:

```
CREATE PROCEDURE p_art_anzahl
RETURNS
   (hersteller VARCHAR(25),
   anzahl INTEGER)
AS
BEGIN
   FOR SELECT
        hersteller,
        COUNT(preis)
      FROM t_artikel
      GROUP BY hersteller
      INTO :hersteller, :anzahl
   DO SUSPEND;
END
```

Diese STORED PROCEDURE können Sie dann wie eine Tabelle behandeln:

```
SELECT *
   FROM p_art_anzahl
   WHERE anzahl > 1
   ORDER BY anzahl DESC
```

2.6 UNION

Mit UNION haben Sie die Möglichkeit, zwei Abfragen zusammenzufügen. Nehmen wir einmal an, Sie möchten an alle Mitarbeiter und Kunden im Postleitzahlenbereich 10 319 ein Schreiben senden. Mit UNION können Sie dafür die Adressen aus zwei Tabellen zu einem gemeinsamen Bestand zusammenfügen:

```
SELECT
     nummer, vorname, nachname
   FROM t_mitarbeiter
   WHERE plz = "10 319"
UNION SELECT
     nummer, vorname, nachname
   FROM t_kunde
   WHERE plz = "10 319"
```

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

```
SELECT
     nummer, vorname, nachname, "Mitarbeiter"
   FROM t_mitarbeiter
   WHERE nachname = "Müller"
UNION SELECT
     nummer, vorname, nachname, "Kunde"
   FROM t_kunde
   WHERE nachname = "Müller"

/* geht nicht */
```

Die Konstante *Mitarbeiter* hat den Typ VARCHAR(11), während *Kunde* den Typ VARCHAR(5) hat. Mit einer Typenumwandlung bekommt man das Problem allerdings in den Griff:

```
SELECT
      nummer, vorname, nachname,
      CAST("Mitarbeiter" AS VARCHAR(12))
   FROM t_mitarbeiter
   WHERE nachname = "Müller"
UNION SELECT
      nummer, vorname, nachname,
         CAST("Kunde" AS VARCHAR(12))
   FROM t_kunde
   WHERE nachname = "Müller"
```

Übung 2.35 Erstellen Sie eine Liste aller Kunden mit dem Namen Ost, welche die Nummer, Vor- und Nachname sowie die Bezeichnung *Kunde* umfaßt. Fügen Sie dieser Liste den Datensatz der Mitarbeiterin *Frau Ost* hinzu. Der Vorname soll hier nicht genannt werden, in die Spalte ist der Wert *Frau* einzutragen, die Bezeichnung lautet *Mitarbeiterin*.

2.7 Unterabfragen

Wenden wir uns mal wieder der Tabelle *t_mitarbeiter* zu. Wie Sie sicher noch wissen, wird in der Spalte *vorgesetzter* die Mitarbeiter-Nummer des direkten Vorgesetzten gespeichert.

Übung 2.36 Ermitteln Sie (mit Hilfe einer SQL-Anweisung) die Mitarbeiter-Nummer des Chefs.

Nun wollen wir ermitteln, welche Mitarbeiter dem Chef direkt unterstellt sind. Man könnte dazu zunächst die SELECT-Anweisung aus Übung 2.36 starten, das Ergebnis auf einem Blatt Papier notieren, und dann die folgende Anweisung starten:

```
SELECT vorname, nachname
   FROM t_mitarbeiter
   WHERE (vorgesetzter <> nummer)
      AND vorgesetzter = 6
```

Eleganter ist es, die erste Anweisung gleich in die zweite einzubauen:

```
SELECT vorname, nachname
   FROM t_mitarbeiter
   WHERE (vorgesetzter <> nummer)
```

```
      AND (vorgesetzter = (SELECT nummer
         FROM t_mitarbeiter
         WHERE vorgesetzter = nummer))
```

Übung 2.37 Zeigen Sie Hersteller, Artikelbezeichnung und Preis (mit zwei Nachkommastellen) von allen Artikeln an, die überdurchschnittlich teuer sind.

Unterabfragen in der Spaltenliste

Unterabfragen sind nicht auf die WHERE-Klausel beschränkt, man kann sie beispielsweise auch in die Liste der anzuzeigenden Spalten aufnehmen. Die folgende Abfrage erstellt die Liste der Mitarbeiter mit den Durchwahlen in der Firma.

```
SELECT
      nummer,
      vorname,
      nachname,
      (SELECT bezeichnung FROM t_tele
         WHERE (mitarbeiter = t_mitarbeiter.nummer)
            AND art = 1)
   FROM t_mitarbeiter
```

Beachten Sie, daß eine solche Unterabfrage immer nur einen einzelnen Wert ermitteln darf. Deshalb ist die Beschränkung auf die Durchwahlen hier zwingend erforderlich.

Aufgabenstellungen wie die hier gelöste lassen sich in der Regel einfacher durch einen JOIN erledigen. Diesen bearbeitet das DMS meist auch schneller als eine Unterabfrage. Eine Ausnahme kann die Bildung von Aggregatfunktionen sein. Hier ist es manchmal effektiver, eine Unterabfrage zu verwenden, als nach vielen Spalten zu gruppieren.

Übung 2.38 Formulieren Sie die Anweisung so um, daß ein JOIN anstatt der Unterabfrage verwendet wird.

2.7.1 Funktionen für Unterabfragen

Für die Arbeit mit Unterabfragen gibt es die Funktionen ALL, ANY, SOME, EXISTS und SINGULAR, die ich alle für nicht besonders erforderlich halte, weil der gewünschte Zweck meist auch anders erreicht werden kann. Es ist jedoch nicht auszuschließen, daß Sie einmal mit SELECT-Statements konfrontiert werden, die eine dieser Funktionen enthalten. Deshalb sollen diese Funktionen kurz besprochen werden.

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_artikel
   WHERE preis > ALL (SELECT preis
      FROM t_artikel
      WHERE gruppe = 1)
```

Diese Abfrage ermittelt alle diejenigen Datensätze der Tabelle *t_artikel*, deren Preis höher liegt als jeder Preis, der von der Unterabfrage ermittelt wird. Die Datensätze der Gruppe eins sind die Prozessoren.

Anders formuliert: Die Abfrage ermittelt alle Artikel, die teurer sind als der teuerste Prozessor.

Übung 2.39 Formulieren Sie die Abfrage so um, daß auf die Funktion ALL verzichtet werden kann.

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_artikel
   WHERE preis > ANY (SELECT preis
      FROM t_artikel
      WHERE gruppe = 1)
```

Diese Abfrage ermittelt alle diejenigen Datensätze der Tabelle *t_artikel*, deren Preis höher liegt als ein beliebiger Preis, der von der Unterab-

frage ermittelt wird. Anders formuliert: Die Abfrage ermittelt alle Artikel, die teurer sind als der billigste Prozessor.

Übung 2.40 Formulieren Sie die Abfrage so um, daß auf die Funktion ANY verzichtet werden kann. (Ersetzen Sie ANY nicht durch SOME!)

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 t_art a
   WHERE EXISTS (SELECT *
      FROM t_tele  t
      WHERE t.art = a.nummer)
```

Diese Abfrage ermittelt nun all diejenigen Datensätze aus der Tabelle *t_art*, auf die in der Tabelle *t_tele* Referenzen gebildet werden.

Übung 2.41 Das geht auch ohne EXISTS.

Man könnte die Funktion EXISTS beispielsweise dazu verwenden, um alle Einträge aus *t_art* zu löschen, welche ohnehin nicht von *t_tele* referenziert werden.

```
DELETE FROM t_art a
   WHERE NOT EXISTS (SELECT *
      FROM t_tele  t
      WHERE art = a.nummer)
```

Wenn Sie diese Anweisung ausgeführt haben, dann nehmen Sie bitte mit FILE | ROLLBACK WORK die Transaktion zurück.

Übung 2.42 Auch das – Sie ahnen es sicher schon – läßt sich ohne die Verwendung von EXISTS formulieren.

SINGULAR

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

Um das experimentell nachzuvollziehen, fügen wir zunächst einen

Datensatz in die Tabelle *t_tele* ein, der den Datensatz sieben von *t_art* referenziert.

```
INSERT INTO t_tele
     (mitarbeiter, art, bezeichnung)
   VALUES (14, 7, "030 / 123 45 67")
```

Dieser Datensatz referenziert nun als einziger den Datensatz sieben von *t_art*. Nun wollen wir ermitteln, welche Datensätze von *t_art* exakt einmal referenziert werden.

```
SELECT * FROM t_art a
   WHERE SINGULAR (SELECT *
      FROM t_tele  t
      WHERE t.art = a.nummer)
```

Übung 2.43 Erstellen Sie eine SQL-Anweisung, die zur selben Ergebnismenge kommt, jedoch nicht SINGULAR verwendet.

2.8 Übungen

Wir wollen das Gelernte mit ein paar Übungen vertiefen. Dazu werden wir die Tabellen *t_bestellung* und *t_posten* verwenden, welche die Bestellungen bei unserer fiktiven Firma und die einzelnen Posten dieser Bestellungen enthalten.

Übung 2.44 Ermitteln Sie die Zahl der Datensätze in diesen beiden Tabellen. Was folgern Sie aus diesen Zahlen?

Übung 2.45 Damit Sie einen Überblick über den Aufbau der Datenbank bekommen, lassen Sie sich jeweils den ersten Datensatz anzeigen (den Datensatz mit dem geringsten Wert in der Spalte *nummer*). Was fällt Ihnen auf?

Übung 2.46 Zeigen Sie Vor- und Nachname des Kunden mit der Nummer 1234 an. Fügen Sie die Datumsangaben seiner Bestellungen hinzu. Sortieren Sie die Anzeige nach dem Datum.

Übung 2.47 Fügen Sie zur Abfrage in Übung 2.41 auch noch den Gesamtpreis der Bestellung hinzu. Beachten Sie dabei, daß der Gesamtpreis eines Postens als Stückzahl mal Einzelpreis gebildet werden muß. Verwenden Sie zur Ermittlung des Gesamtpreises eine Unterabfrage.

Übung 2.48 Stellen Sie die SELECT-Anweisung aus Übung 2.42 so

um, daß anstatt einer Unterabfrage ein JOIN verwendet wird.

Übung 2.49 Welcher Mitarbeiter hat 1998 wieviel Umsatz gemacht? Denken Sie zunächst über eine effektive Vorgehensweise nach.